Job Ready Master

Welcome to your new software project space

Use this space to track decisions, scope product requirements, share assets, and do any other work relating to your software project so it's easy for your team and stakeholders to find.

To start, you might want to:

- **Customise this overview** using the **edit icon** at the top right of this page.
- **Create a new page** by clicking the **+** in the space sidebar.

Udemy Software Labs

Status

SET A STATUS

Lead

Add a user profile for your team lead.

Team

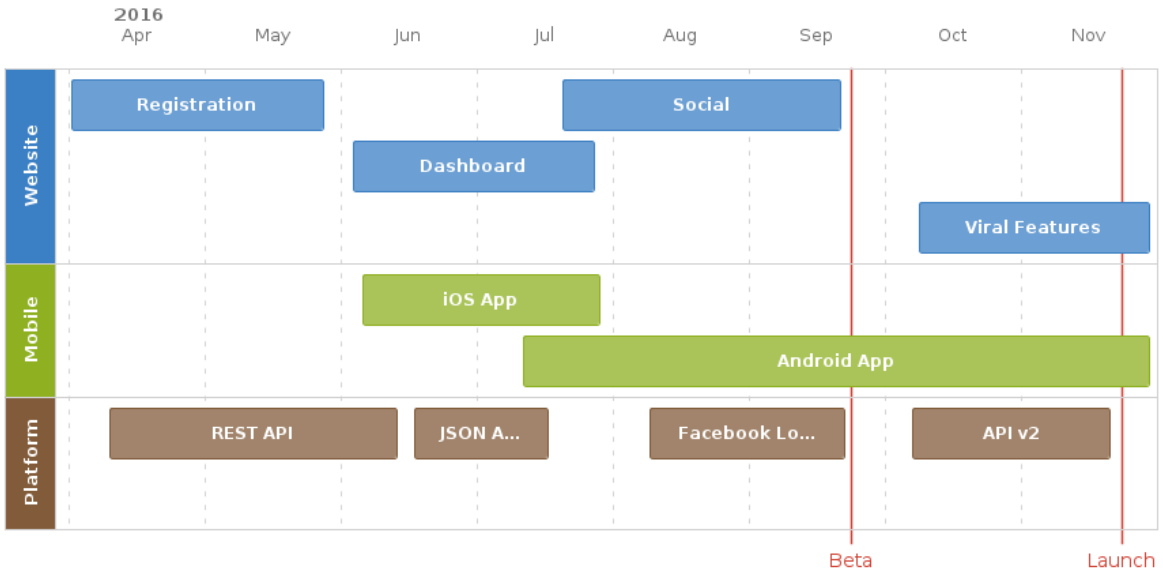
Add a user profile for each member of your team.

Recently updated

- IELTS Platform Architecture
  Jul 05, 2023 • contributed by Lock Huynh
- CCLMagic Solution
  Jun 21, 2023 • contributed by Ben Tran
- Amazon Cognito
  Jun 21, 2023 • contributed by Ben Tran
- AWS ACCOUNT SUSPENDED
  Jun 14, 2023 • contributed by Ben Tran
- AWS Route53
  Jun 14, 2023 • contributed by Ben Tran
- AWS App Runner & Amplify
  May 31, 2023 • contributed by Ben Tran
- AWS Lambda & AWS App Runner
  May 31, 2023 • contributed by Ben Tran

Roadmap

Edit this roadmap planner to link a Confluence page to each bar.



Jira Issues

Edit this Jira issues list to change what fields and issues are displayed.

Key	T	Created	Updated	Due	Assignee	Status	Resolution
JRM-174	☑	30/Sep/23 10:03 AM	02/Oct/23 1:26 PM	06/Oct/23	Unassigned	TO DO	Unresolved
JRM-173	☑	25/Sep/23 10:44 AM	25/Sep/23 10:44 AM	25/Sep/23	Unassigned	TO DO	Unresolved
JRM-171	☑	20/Sep/23 9:10 PM	20/Sep/23 9:10 PM	22/Oct/20	Unassigned	TO DO	Unresolved
JRM-168	☑	05/Sep/23 4:08 PM	05/Sep/23 4:08 PM		Unassigned	TO DO	Unresolved
JRM-167	☑	04/Sep/23 5:01 PM	04/Sep/23 5:01 PM	05/Sep/23	Unassigned	TO DO	Unresolved
JRM-166	☑	29/Aug/23 5:13 PM	29/Aug/23 5:14 PM	30/Aug/23	Unassigned	TO DO	Unresolved
JRM-165	☑	26/Aug/23 3:40 PM	26/Aug/23 3:40 PM		Unassigned	TO DO	Unresolved
JRM-164	☑	26/Aug/23 3:19 PM	26/Aug/23 3:19 PM	28/Aug/23	Unassigned	TO DO	Unresolved
JRM-163	☑	22/Aug/23 3:24 PM	22/Aug/23 3:24 PM	22/Aug/23	Unassigned	TO DO	Unresolved
JRM-148	☑	24/Jul/23 8:10 PM	24/Jul/23 8:10 PM	27/Jul/23	Unassigned	TO DO	Unresolved
JRM-146	☑	20/Jul/23 8:49 PM	20/Jul/23 8:49 PM	20/Jul/23	Unassigned	TO DO	Unresolved
JRM-143	☑	10/Jul/23 8:30 PM	10/Jul/23 8:30 PM	27/Jul/23	Unassigned	TO DO	Unresolved
JRM-142	☑	10/Jul/23 8:01 PM	10/Jul/23 8:03 PM	26/Jul/23	Unassigned	TO DO	Unresolved

13 issues

NEED INSPIRATION?

- Get a quick rundown on how to build Confluence spaces for any team or project
- See these tips on how to stay on top of your software project in Confluence

Documents

(Doc) Week 1: HTML & CSS

Doc: HTML

Intro to HTML

Welcome to Intro to HTML! First, we'll dive into HTML, which will structure the elements in our webpage. Next, we'll move onto CSS, which will help style the webpage. Lastly, we'll focus more on the layout to make sure everything is distributed how we want.

HTML stands for Hypertext Markup Language. This is the same "Hypertext" included in the "http" you see at the start of URLs in the browser, as it's part of how browsers will know how to render what you provide. By the end of this lesson, you'll be able to use HTML to add elements to display within a browser window, just as you would do with a webpage.

HTML is a common starting point for those looking to get into web development. Later in the course, you'll combine it with CSS, and even later in the Nanodegree, with Javascript, to create a full web experience. HTML itself is very much the basic building block of the web.

A Few Early HTML Tags

Let's quickly take a look at some early HTML tags. With HTML, tags help tell the browser how to render different elements on the page, especially once you get into the styling with CSS later, where the tags will point to specific styles to use throughout your webpage. We will just briefly cover these here, as you'll see more on them when we discuss the DOM.

`<!DOCTYPE html>`

Nearly every web document using HTML will start with this. This just tells the browser it is rendering an HTML document.

`<html>` and `</html>`

The first of these will likely be right after the doctype tag, while the second will "close" this tag, and won't be present until the end of the HTML document. These note that everything in between is HTML code.

`<body>` and `</body>`

The body is where most of the content you will actually see on your webpage goes. It is closed with a </body> tag. Note that in basic examples, it is not actually required, as the page assumes other content is within the body, but by the end of this lesson you should be using this regularly. There is actually a <head> section that can come before the body, but we will skip that for now.

Here is just a very brief example of how these might look in practice, without anything filled into the body yet.

Here is just a very brief example of how these might look in practice, without anything filled into the body yet.

```
<!DOCTYPE html>
<html>
  <body>
    <!-- Page content would go here -->
  </body>
</html>
```

We will start seeing all of these in practice soon!

Text Editors

The program used to open HTML documents is a browser.

The program used to create HTML documents is a text editor. Text editors are used to both create and modify many types of documents, among them HTML. There is also a set of more advanced text editors, called Integrated Development Environments (IDEs), with more extensive features.

The tool you choose as your text editor is pretty important, since it allows you to take advantage of modern technologies and code faster.

Hello World Example

```
<p>Hello World!</p>
```

This code is written with a text editor.

You can see the tags `<p>` and `</p>` that stand for paragraph.

When opened by a browser, these tags are not displayed but rather interpreted by the browser

The browser sees the `<p>` and `</p>` tags and understands that Hello World is a paragraph.

Remember that an HTML document can be opened in 2 ways:

1. by a text editor who sees the source code
2. by a browser who interprets the source code

Options

- Visual Studio Code
- Atom
- Sublime Text
- JetBrains
- NotePad++

Text Elements I

Text Element Structures

An HTML element is a unit of content in an HTML document that is formed using HTML tags.

The basic structure of an element is composed of 4 key items, one of which is optional.

```
<p class="dog-breed">Labrador Retriever</p>
```

1. The opening tag is the first HTML tag used to start an element.
2. The content is the info contained between the opening and closing tags. Only this content inside the opening and closing body tags is displayed to the screen.
3. The closing tag is the second tag used to define the end of a single element. Closings tags have a forward slash / inside of them, always after the left angle bracket.
4. (Optional) The attribute name and value.

Notice both tags are always surrounded by opening and closing angle brackets `<>`.

Note: developers use the terms "left bracket" and "opening bracket" interchangeably. SImilarly, you can use either "right bracket" or "closing bracket".

There are over 100 different types of HTML tags that each serve a specific use case.

In HTML, you'll mainly come across 2 types of HTML elements.

Block elements are meant to structure the main parts of your page, by dividing your content in coherent blocks.

Block elements are:

- paragraphs `<p>`
- lists:
  - unordered (with bullet points) `<ul>`, or
  - ordered - lists (with numbers) `<ol>`
- headings: from 1st level `<h1>` to 6th level headings - `<h6>`
- articles `<article>`
- sections `<section>`
- long quotes `<blockquote>`

Inline elements are meant to differentiate part of a text, to give it a particular function or meaning. Inline elements usually comprise a single or few words.

Inline elements are:

- links `<a>`
- emphasized words `<em>`
- important words `<strong>`

Text Elements II

Headings

Headings in HTML are comparable to headings in other media types. In journals, for instance, big headings are typically used to catch the attention of a reader. Other times, headings are used to define material, such as a film's title or an instructional article.

Headings are the primary way to outline the content of your webpage. They define the outline of your web page as both humans and search engines see it, which makes selecting relevant headings essential for a high-quality web page.

There are six distinct headings or heading components in HTML. Headings can be used for a multitude of reasons, such as titling segments, journals, or other types of content.

One way to think about headings on a web page is like headings in a book..

The `<h1>`, like the book title, introduces the topic that the web page is all about.

The `<h2>`, like book chapters, describe the main topics covered on the web page

Smaller headers like the `<h3>` to `<h6>` serve as other sub-headings that can be used within each section, just like a book chapter can be as a book chapter may be split up by multiple sub-topics.

Headings are ordered from the biggest to the smallest size. There are 6 levels of headings available, ranging from `<h1>` to `<h6>`, 1 being the most important one.

H1 is used for the primary headings. For subheadings, all other lower headings are used.

Here's an example of headers:

```
<h1>Observable Universe</h1>
<h2>Milky Way Galaxy</h2>
<h3>Earth</h3>
<h4>USA</h4>
<h5>Norfolk, VA</h5>
<h6>Main Street</h6
```

Which results in:

Observable Universe

Milky Way Galaxy

Earth

USA

Norfolk, VA

Main Street

Accessibility with Headers

For people who are blind or visually impaired, screen reading software is used to parse through text on a web page. A common technique these folks will use to navigate the page is to jump from heading to heading to determine the overall content of the page more easily. That's why it's best practice to not skip one or more heading levels. If you did skip headings and went from `<h1>` to `<h3>`, you may cause confusion since the user has to deal with a missing heading. Don't bum out any of your users - structure your headings properly!

Paragraphs

Paragraphs `<p>` are the most used HTML element, as they act as the default block-level element and are quick to write.

Below is the HTML code, as well as the “paragraphs” of text - note that there is nothing particularly special with the formatting of these.

```
<p>
  The sweet-faced and loving Labrador Retriever is actually one of the most
  popular dog breeds.
</p>
<p>
  Labs are extremely friendly with an easygoing and high-spirited personality
  which is great for bonding with the whole family.
</p>
```

The sweet-faced and loving Labrador Retriever is actually one of the most popular dog breeds.

Labs are extremely friendly with an easygoing and high-spirited personality which is great for bonding with the whole family.

Spans

The HTML `<span>` element is like a generic wrapper that is used to group text, mostly for styling purposes. Consider the following code:

```
<style>
p {
  color: black;
}
.red {
  color: red;
}
</style>
<p>
  This sentence needs some <span class=”red”>visual emphasis</span> to really bring home the point.
</p>
```

And its result:

This sentence needs some visual emphasis to really bring home the point.

In this code, the words `visual emphasis` have been put inside the `span` with the class `red`, so that those individual words can be styled separately from the rest of the `p` element. In this instance, the `span` words would be red, while the rest of the words would be black.

Blockquotes

Blockquotes are used to identify a citation.

```
<blockquote cite="https://www.wikiwand.com/en/Scooby-Doo_(character)">
  <p>Ruh-roh--RAGGY!!!</p>
  <footer>—Scooby Doo, <cite>Mystery Incorporated</cite></footer>
</blockquote>
```

Ruh-roh--RAGGY!!!

—Scooby Doo, Mystery Incorporated

Line Breaks

The spacing between code in an HTML file doesn’t affect the positioning of elements in the browser.

If you are interested in modifying the spacing in the browser, you can use HTML’s line break element.

```
<p>
  I jump in delight<br />
```

```
  I run off in frenzy<br />
  For now I have just realized<br />
  that the fun has arrived<br />
  the fun has begun<br />
  jumping all on one piece<br />
  almost feeling like I can't breathe<br />
  blood rushing through me<br />
  a second, a beat<br />
  I feel the air on my face<br />
  My fur rising up<br />
  Free as free as it can be<br />
  That's what you feel<br />
  When your owner has arrived<br />
</p>
```

I jump in delight
I run off in frenzy
For now I have just realized
that the fun has arrived
the fun has begun
jumping all on one piece
almost feeling like I can't breathe
blood rushing through me
a second, a beat
I feel the air on my face
My fur rising up
Free as free as it can be
That's what you feel
When your owner has arrived

HTML Lists

You can also organize content in list format.

Although I noted 2 lists in HTML, there are actually 3 total types of lists in HTML:

1. unordered - lists in no specific order
2. ordered - lists in a specific order
3. description - lists with name/value pairs

Depending on the use case, you may want to use one over the other. Just then, I used an ordered list because there was a specific number of lists I wanted to showcase.

Description lists are out of scope for this lesson, but you can learn more about them here.

Unordered Lists

If you want items in no particular order, like with a shopping list, you use the unordered list HTML tag - `<ul>`.

An unordered list outlines individual list items with a bullet point with each individual bullet added using the list item or `<li>` tag.

```
<p>New puppy shopping list</p>
<ul>
  <li>Treats</li>
  <li>Dog food</li>
  <li>Leash</li>
  <li>Collar</li>
  <li>Dishes</li>
  <li>ID tag</li>
</ul>
```

Which results in:

New puppy shopping list

- Treats
- Dog food
- Leash
- Collar
- Dishes
- ID tag

Ordered Lists

Ordered lists `<ol>` are like unordered lists, except that each list item is numbered.

They are useful when you need to list different steps in a process or rank items for first to last given the order of items is relevant.

Just like with unordered lists, you can add individual list items to the list using `<li>` tags.

```
<p>Steps after adopting a puppy</p>
<ol>
  <li>Spoil the puppy</li>
  <li>Be happy with your puppy</li>
  <li>Repeat</li>
</ol>

```

Which results in:

Steps after adopting a puppy

1. Spoil the puppy
2. Be happy with your puppy
3. Repeat

Ordered lists are automatically numbered by the browser, so the numbers don’t need to be included in your HTML.

Attributes

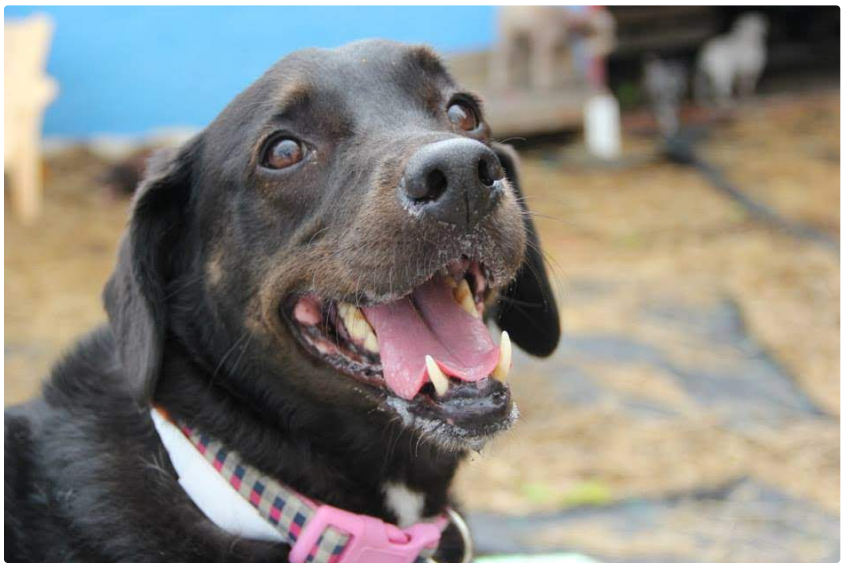
All HTML elements can have attributes. Attributes provide additional information about an element, and are always specified in the start tag. Attributes usually come in name/value pairs like `name="value"` Here are some popular ways attributes are used that we'll be covering in the sections to come!

Images

- The “source” (URL or file location) from where an image is taken through the `src` attribute
- The image’s alternative text (often a description for those with accessibility needs) is provided through the `alt` attribute
- The image size can be adjusted through the `width` and `height` attributes
- Images are self-closing - you add a slash at the end, instead of another `</img>` tag as we have seen before

In the example below, we are using a JPG image called “nefertiti”.

```
<img
  src="images/nefertiti.jpg"
  alt="A smiling Labrador Retriever"
  width="480"
  height="320"
/>

```



A smiling Labrador Retriever

Links

Links are essential in HTML, as the Web was initially designed to be an information network of documents “linked” between each other - you navigate from one document to another by clicking on links.

The “HyperText” part of HTML defines what kind of links we use: hypertext links, a.k.a hyperlinks.

In HTML, links are inline elements written with the `<a>` tag. The `href` attribute (hypertext reference) is used to define the destination of the link (where you navigate to when you click).

There are 3 types of destinations you can define:

- anchor targets, to navigate within the same page
- relative URLs, usually to navigate within the same website

- absolute URLs, usually to navigate to another website

You can also use additional attributes besides `a` and `href`:

- specify the relationship between the current and linked document with the `rel` attribute
- specify where to open the linked document with the `target` attribute

In the example below, we are setting the URL destination to The Labrador Club website, preventing this website from being able to access the `window.opener` property and ensuring it runs in a separate process with the `noopener` rel value. Finally we are requesting that the link open in a new window instead of the same one with the `_blank` target value.

```
<a href="https://thelabradorclub.com" rel="noopener" target="_blank"
  >Join The Labrador Retriever Club</a>

```

Comments

If you write something in your code without changing how your website will be displayed by the browser, you're writing comments.

Comments will be ignored by the browser and are only helpful to us people who write the code. They can help with readability both for yourself in the future, as well as others who are looking at your code, such as teammates and managers.

A comment starts with `<!--` and ends with `-->`.

Self-Enclosing Elements

Some HTML elements only have an opening tag:

```
<!-- line-break -->
<br />
<!-- image -->
<img src="https://dog.com/image.png" alt="Description" />
<!-- text input -->
<input type="text" />

```

Because they don’t have a closing tag and consequently can’t contain anything inside them, self-enclosing elements usually carry a few attributes, to provide them with additional information.

The DOM

Now that you've learned about some of the most common HTML elements, it's time to learn how to set up an HTML file. HTML files require certain elements to set up the document properly. You can let web browsers know that you are using HTML by starting your document with a document type declaration. The declaration looks like this:

`<!DOCTYPE html>`

This declaration is an instruction, and it must be the first line of code in your HTML document. It tells the browser what type of document to expect, along with what version of HTML is being used in the document. For now, the browser will correctly assume that the html in `<!DOCTYPE html>` is referring to HTML5, as it is the current standard.

In the future, however, a new standard will override HTML5. To make sure your document is forever interpreted correctly, always include `<!DOCTYPE html>` at the very beginning of your HTML documents.

The Head

So far you've done two things to set up the file properly:

1. Declared to the browser that your code is HTML with `<!DOCTYPE html>`
2. Added the HTML element (`<html>`) that will contain the rest of your code.

Remember the `<body>` tag? The `<head>` element is part of this HTML metaphor. It goes above our `<body>` element.

Metadata

The `<head>` element contains the metadata for a web page.

Metadata is information about the page that isn't displayed directly on the web page.

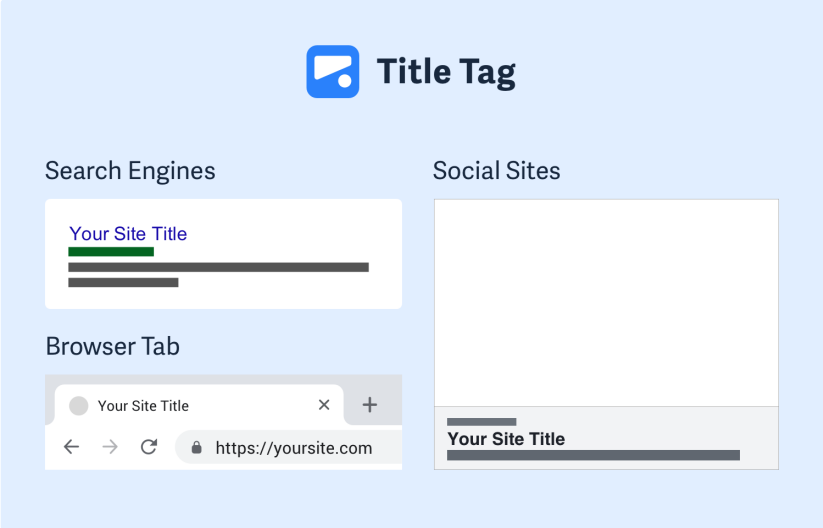
Unlike the information inside of the `<body>` tag, the metadata in the head is information about the page itself.

Title

A browser's tab displays the title specified in the `<title>` tag. The `<title>` tag is always inside of the `<head>`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Coding Journal</title>
  </head>
</html>

```

If we were to open a file containing the HTML code in the example above, the browser would display the words "My Coding Journal" in the title bar (or in the tab's title).



An example of titles

The Body

```
<body></body>

```

Once the file has a body, many different types of content – including text, images, and buttons – can be added to the body.

Hierarchy

HTML is organized as a collection of family tree relationships. As you saw in the last exercise, we placed `<p>` tags within `<body>` tags. When an element is contained inside another element, it is considered the child of that element. The child element is said to be nested inside of the parent element.

```
<body>
  <p>This paragraph is a child of the body</p>
</body>

```

In the example above, the `<p>` element is nested inside the `<body>` element. The `<p>` element is considered a child of the `<body>` element, and the `<body>` element is considered the parent. You can also see that we've added two spaces of indentation (using the space bar) for better readability.

Since there can be multiple levels of nesting, this analogy can be extended to grandchildren, great-grandchildren, and beyond. The relationship between elements and their ancestor and descendent elements is known as hierarchy.

Let's consider a more complicated example that uses some new tags:

```
<body>
  <div>
    <h1>Sibling to p, but also grandchild of body</h1>
    <p>Sibling to h1, but also grandchild of body</p>
  </div>
</body>

```

In this example, the `<body>` element is the parent of the `<div>` element. Both the `<h1>` and `<p>` elements are children of the `<div>` element. Because the `<h1>` and `<p>` elements are at the same level, they are considered siblings and are both grandchildren of the `<body>` element.

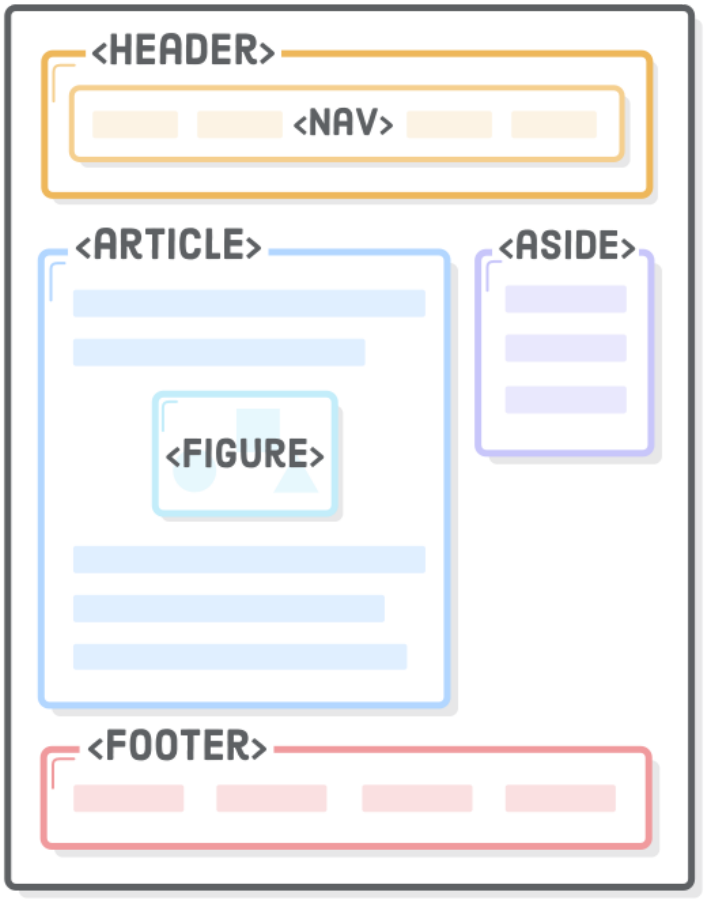
Again, understanding HTML hierarchy is important because child elements can inherit behavior and styling from their parent element. You'll learn more about webpage hierarchy when you start digging into CSS.

Semantic Elements

Structure elements allow you to organize the main parts of your page. They usually contain other HTML elements.

Here's what a typical webpage could include:

- `<header>` as the first element of the page, that can include the logo and the tagline.
- `<nav>` as a list of links that go to the different pages of the website.
- `<h1>` as the title of the page.
- `<article>` as the main content of the page, like a blog post.
- `<footer>` as the last element of the page, located at the bottom.



An example of the structure of semantic elements

Forms

Forms are a part of everyday life. When we use a physical form in real life, we write down information and give it to someone to process. Think of the times you've had to fill out information for various applications like a job, or a bank account, or dropped off a completed suggestion card — each instance is a form!

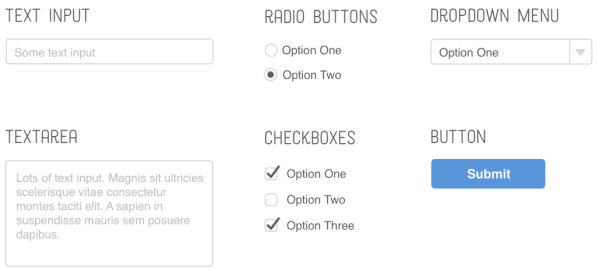
Just like a physical form, an HTML `<form>` element is responsible for collecting information to send somewhere else. Every time we browse the internet we come into contact with many forms and we might not even realize it. There's a good chance that if you're typing into a text field or providing an input, the field that you're typing into is part of a `<form>`!

In this lesson, we'll go over the structure and syntax of a `<form>` and the many elements that populate it.

HTML form elements let you collect input from your website's visitors. Mailing lists, contact forms, and blog post comments are common examples for small websites, but in organizations that rely on their website for revenue, forms are sacred and revered.

Forms are the "money pages." They're how e-commerce sites sell their products, how SaaS companies collect payment for their service, and how non-profit groups raise money online. Many companies measure the success of their website by the effectiveness of its forms because they answer questions like "how many leads did our website send to our sales team?" and "how many people signed up for our product last week?" This often means that forms are subjected to endless A/B tests and optimizations.

There are multiple types of HTML forms, such as text input, text areas, radio buttons, checkboxes, dropdown menus, and buttons.

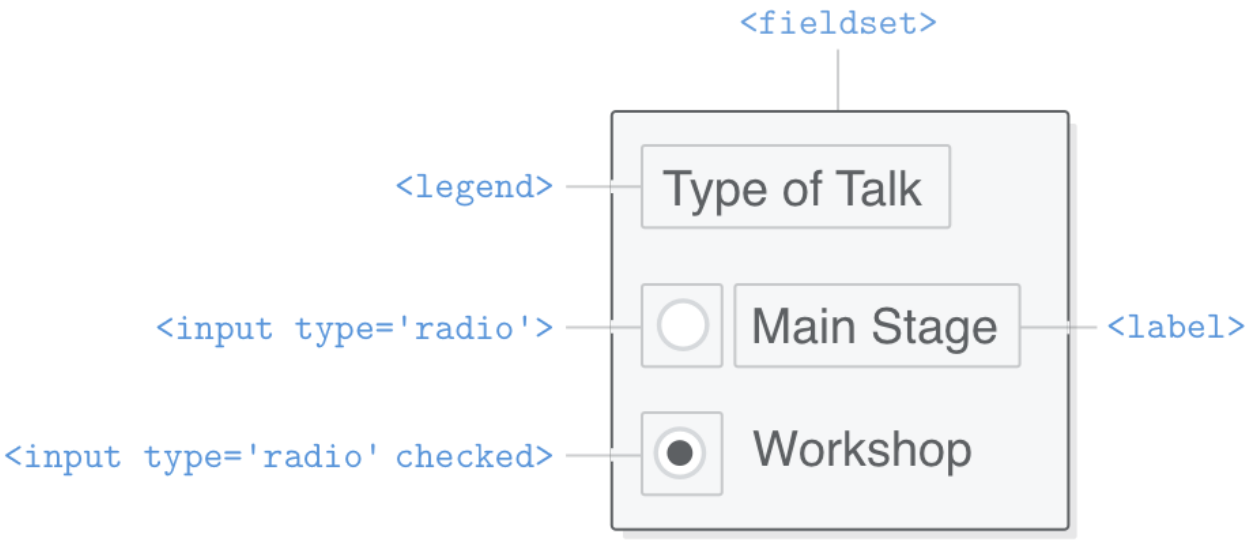


Examples of forms

Input Types, Select and Textarea

Text, checkbox and radio button forms are specified by an `input type`.

```
<!-- A text input -->
<input type="text" />
<!-- A checkbox -->
<input type="checkbox" />
<!-- A radio button -->
<input type="radio" />

```



An example of a ‘radio’ input type

Separately, a dropdown menu can be created using `select`.

```
<label for="color-select">Choose a color:</label>

<select id="color-select">
  <option value="">--Please choose an option--</option>
  <option value="blue">Blue</option>
  <option value="red">Red</option>
  <option value="green">Green</option>
  <option value="yellow">Yellow</option>
  <option value="orange">Orange</option>
  <option value="pink">Pink</option>
</select>

```

Choose a color: --Please choose an option-- Blue Red Green Yellow Orange Pink

---

Last is `textarea`, which creates a more free-form text field for the user to enter information.

```
<label for="learn">What do you hope to learn today?</label>

<textarea id="learn" name="learn" rows="5" cols="30">
I hope to learn about...
</textarea>

```

What do you hope to learn today?

More on forms

Follow these links to learn more on HTML forms, select, and textarea.

Doc: CSS

CSS Selectors

Tags

In this section, you'll learn how to use different visual CSS guidelines to style elements individually and by group.

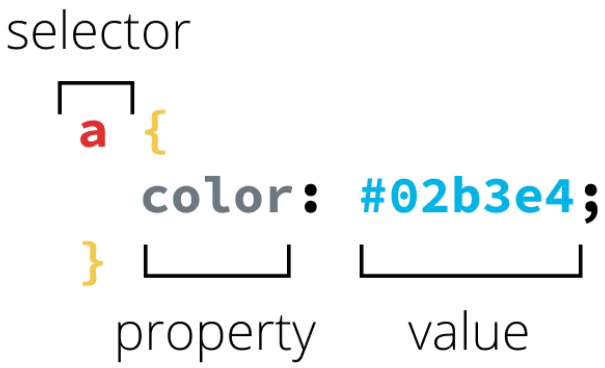
CSS can select HTML elements by using an element's tag name. A tag name is the word (or character) between HTML angle brackets.

For example, in HTML, the tag for a paragraph element is `<p>`. The CSS syntax for selecting `<p>` elements is:

```
p {
  color: red;
}

```

In the example above, all paragraph elements will be selected using a CSS selector. The selector in the example above is `p`. Note that the CSS selector matches the HTML tag for that element, but without the angle brackets.

In addition, two curly braces follow immediately after the selector (an opening and closing brace, respectively). Any CSS properties will go inside of the curly braces to style the selected elements.



An example of a CSS selector, property and value.

Classes

CSS is not limited to selecting elements by tag name. HTML elements can have more than just a tag name; they can also have attributes. One common attribute is the class attribute. It's also possible to select an element by its class attribute.

For example, consider the following HTML:

```
<p class="brand">Sole Shoe Company</p>

```

The paragraph element in the example above has a class attribute within the `<p>` tag. The class attribute is set to "brand". To select this element using CSS, we could use the following CSS selector:

```
.brand {

}

```

To select an HTML element by its class using CSS, a period (`.`) must be prepended to the class's name. In the above case, the class is "brand", so the CSS selector for it is `.brand`.

Ids

For situations where you need more specificity in styling, you may also select elements for CSS using an `id` attribute. You can have different ids associated with a class (although a class is not required). For example, consider the following HTML:

```
<p id=”solo” class="brand">Sole Shoe Company</p>

```

The `id` attribute can be added to an element, along with a class attribute. On the CSS side, the delineation is made by using `#` to represent an `id`, the same way `.` is used for `class`. The CSS to select and style the HTML element above could look like this:

```
#solo {
  color: purple;
}

```

Pseudo-classes

A CSS pseudo-class is a keyword added to a selector that specifies a special state of the selected element(s). For example, `:hover` can be used to change a button's color when the user's pointer hovers over it.

```
selector:pseudo-class {
  property: value;
}

```

For more information on pseudo-classes, see the Mozilla Dev Docs here.

Attributes

Attribute selectors are a special kind of selector that will match elements based on their attributes and attribute values.

Their generic syntax consists of square brackets (`[]`) containing an attribute name followed by an optional condition to match against the value of the attribute.

Attribute selectors can be divided into two categories depending on the way they match attribute values:

1. Presence and value attribute selectors and
2. Substring value attribute selectors.

These attribute selectors try to match an exact attribute value:

- `[attr]` This selector will select all elements with the attribute attr, whatever its value.
- `[attr=val]` This selector will select all elements with the attribute attr, but only if its value is val.
- `[attr~=val]` This selector will select all elements with the attribute attr, but only if val is one of a space-separated list of words contained in attr's value. (This one is a bit more complex, so checking some documentation might be helpful.)

Linking CSS

Inline

Although CSS is a different language than HTML, it's possible to write CSS code directly within HTML code using inline styles.

To style an HTML element, you can add the style attribute directly to the opening tag. After you add the attribute, you can set it equal to the CSS style(s) you'd like applied to that element.

```
<p style="color: red;">I'm learning to code!</p>

```

The code in the example above demonstrates how to use inline styling. The paragraph element has a style attribute within its opening tag. Next, the style attribute is set equal to `color: red;`, which will set the color of the paragraph text to red within the browser.

You might be wondering about the syntax of the following snippet of code: `color: red;`. At the moment, the details of the syntax are not important; you'll learn more about CSS syntax in other exercises. For now, it's important to know that inline styles are a quick way of directly styling an HTML element.

If you'd like to add more than one style with inline styles, simply keep adding to the style attribute. Make sure to end the styles with a semicolon (`;`).

```
<p style="color: red; font-size: 20px;">I'm learning to code!</p>

```

You can also stick CSS rules in the style attribute of an HTML element. In dummy.html, we have a link that doesn't actually go anywhere. Let's make it red via an inline style so we remember it's a dead link:

```
<p>
  Want to try crossing out an
  <a href="nowhere.html" style="color: #990000; text-decoration: line-through;">obsolete link</a>
  ? This is your chance!
</p>

```

Like page-specific styles, this is the same CSS syntax we've been working with. However, since it's in an attribute, it needs to be condensed to a single line. Inline styles are the most specific way to define CSS. The color and text-decoration properties we defined here trump everything. Even if we went back and added a text-decoration: none to our `<style>` element, it wouldn't have any effect.

Inline styles should be avoided at all costs because they make it impossible to alter styles from an external stylesheet. If you ever wanted to re-style your website down the road, you can't just change a few rules in your global styles.css file—you'd have to go through every single page and update every single HTML element that has a style attribute. It's horrifying.

That said, there will be many times when you need to apply styles to only a specific HTML element. For this, you should always use CSS classes instead of inline styles.

Style Tag

Inline styles are a fast way of styling HTML, but they also have limitations. If you wanted to style, for example, multiple `<h1>` elements, you would have to add inline styling to each element manually. In addition, you would also have to maintain the HTML code when additional `<h1>` elements are added.

Fortunately, HTML allows you to write CSS code in its own dedicated section with the `<style>` element. CSS can be written between opening and closing `<style>` tags. To use the `<style>` element, it must be placed inside of the `<head>` element.

```
<head>
  <style></style>
</head>

```

```
```

After adding a `<style>` tag in the head section, you can begin writing CSS code.

```
<head>
  <style>
    p {
      color: red;
      font-size: 20px;
    }
  </style>
</head>

```

The CSS code in the example above changes the color of all paragraph text to red and also changes the size of the text to 20 pixels. Note how the syntax of the CSS code matches (for the most part) the syntax you used for inline styling. The main difference is that you can specify which elements to apply the styling to.

External Stylesheets

When HTML and CSS code are in separate files, the files must be linked. Otherwise, the HTML file won't be able to locate the CSS code, and the styling will not be applied.

You can use the `<link>` element to link HTML and CSS files together. The `<link>` element must be placed within the head of the HTML file. It is a self-closing tag and requires the following three attributes:

- `href` — like the anchor element, the value of this attribute must be the address, or path, to the CSS file.
- `type` — this attribute describes the type of document that you are linking to (in this case, a CSS file). The value of this attribute should be set to text/css.
- `rel` — this attribute describes the relationship between the HTML file and the CSS file.

Because you are linking to a stylesheet, the value of `rel` should be set to stylesheet.

When linking an HTML file and a CSS file together, the `<link>` element will look like the following:

```
<link href="https://udacity.com/style.css" type="text/css" rel="stylesheet" />

```

Specifying the path to the stylesheet using a URL is one way of linking a stylesheet.

If the CSS file is stored in the same directory as your HTML file, then you can specify a relative path instead of a URL, like so:

```
<link href="./style.css" type="text/css" rel="stylesheet" />

```

Using a relative path is very common way of linking a stylesheet.

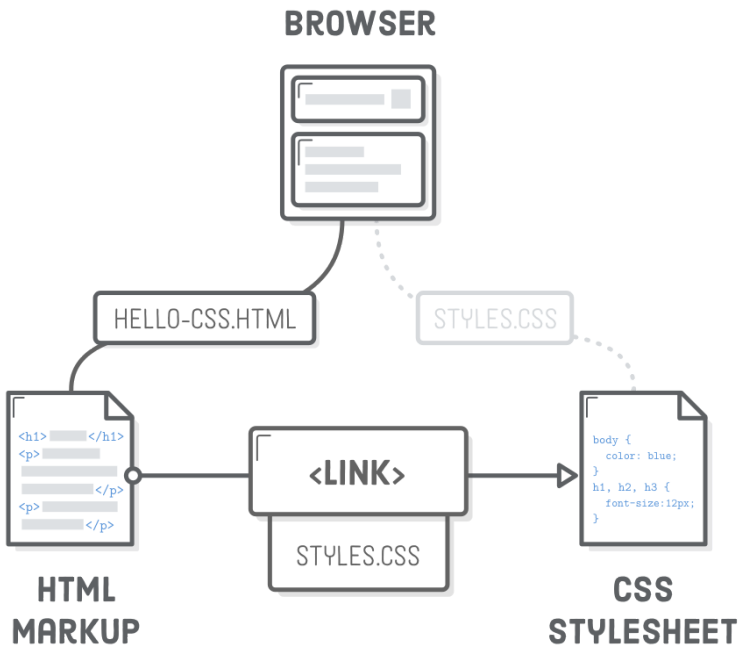
```
<link rel="stylesheet" href="styles.css" />

```

Note that in HTML5, you actually don't have to specify `type="text/css"` here.

So you've just learned that using the `<link>` element allows you to link HTML and CSS files together. What about linking a CSS file to another CSS file? You can have all your styles living inside one main CSS file, or you can use `@import` to break your styles (one for layout, one for images, one for blog cards, etc.) into a number of smaller, focused files. This makes it a lot easier to manage the styles they contain and your code is more scalable and modular!

```
// at the top of your main CSS file

@import “./layout”;
```

```
@import “./images”;
@import “./blog-cards”;

```



Relationship between HTML and CSS files

Specificity

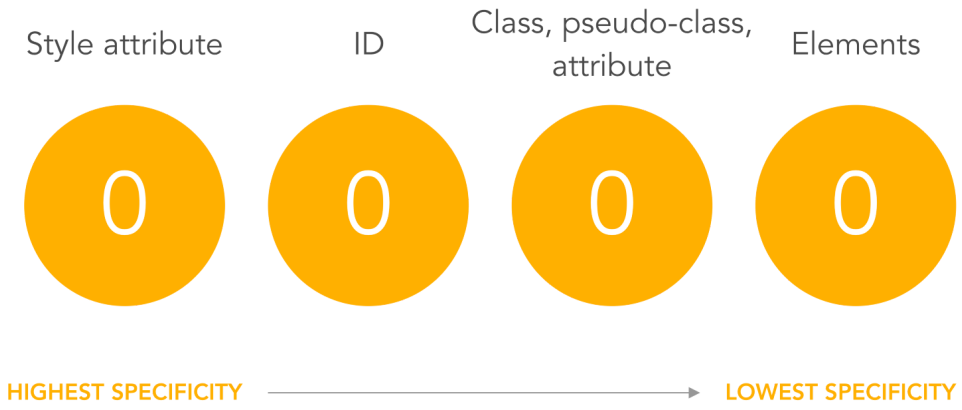
Because elements can have multiple CSS selectors, there is a hierarchy for the weight given to each type of selector. Here is the logical order of selectors from least to most weight assigned:

- Type selectors (e.g., h1) and pseudo-elements (e.g., `::before`).
- Class selectors (e.g., `.example)`, attributes selectors (e.g., `[type="radio"]`) and pseudo-classes (e.g., `:hover`).
- ID selectors (e.g., `#example`).

This concept can help you understand why your styles aren't being applied in the way you expect.

There is a way to escape or override the specificity evaluation of elements using the `!important` keyword after an individual CSS property rule, but a couple important reminders:

Always look for a way to use specificity before even considering `!important`. Never use `!important` on site-wide CSS.



Specificity levels

Style Attribute

If the element has an inline style, it automatically overrides all other styles.

(1,0,0,0)

```
<button style="color: red">Click me</button>
```

ID

Add a 1 to the ID section for each ID selector denoted.

(0,1,0,0)

```
#cat {
 color: red;
}
```

Class, Pseudo-Class, Attribute

Add a 1 to the Class, Pseudo-Class, Attribute section for each class, pseudo-class, or attribute selector denoted.

(0,0,1,0)

```
.cat {
  color: red;
}
```

Elements

Add a 1 to the Elements section for each element selector denoted.

(0,0,0,1)

```
button {
 background: orange;
}
```

Specificity explanation - example

CSS SELECTOR 1

```
ul > li {
  color: blue;
}
```

0	0	0	2
Style attribute	ID	Class, pseudo-class, attribute	Elements

CSS SELECTOR 2

```
.list > .list-item {
  color: red;
}
```

0	0	2	0
Style attribute	ID	Class, pseudo-class, attribute	Elements

OUTCOME

- List item 1
- List item 2
- List item 3

CSS selector 2 has a higher specificity than CSS selector 1, because it uses two class names in comparison to two selector names. Thus, the list item elements will have a color of red.

Specificity explanation - example

The Box Model

Just like CSS, HTML and JS are the three basic building blocks of the web, the box model is one of the basic building blocks for CSS.

Every beginner should first start with the basics. In case of CSS, the basics are learning the box model. Before proceeding with learning any other CSS concepts, this is the one you should master first!

The box model is the basic building block of CSS.

When a browser renders (draws) a webpage each element, for example a piece of text or an image, is drawn as a rectangular box following the rules of the CSS Box Model.

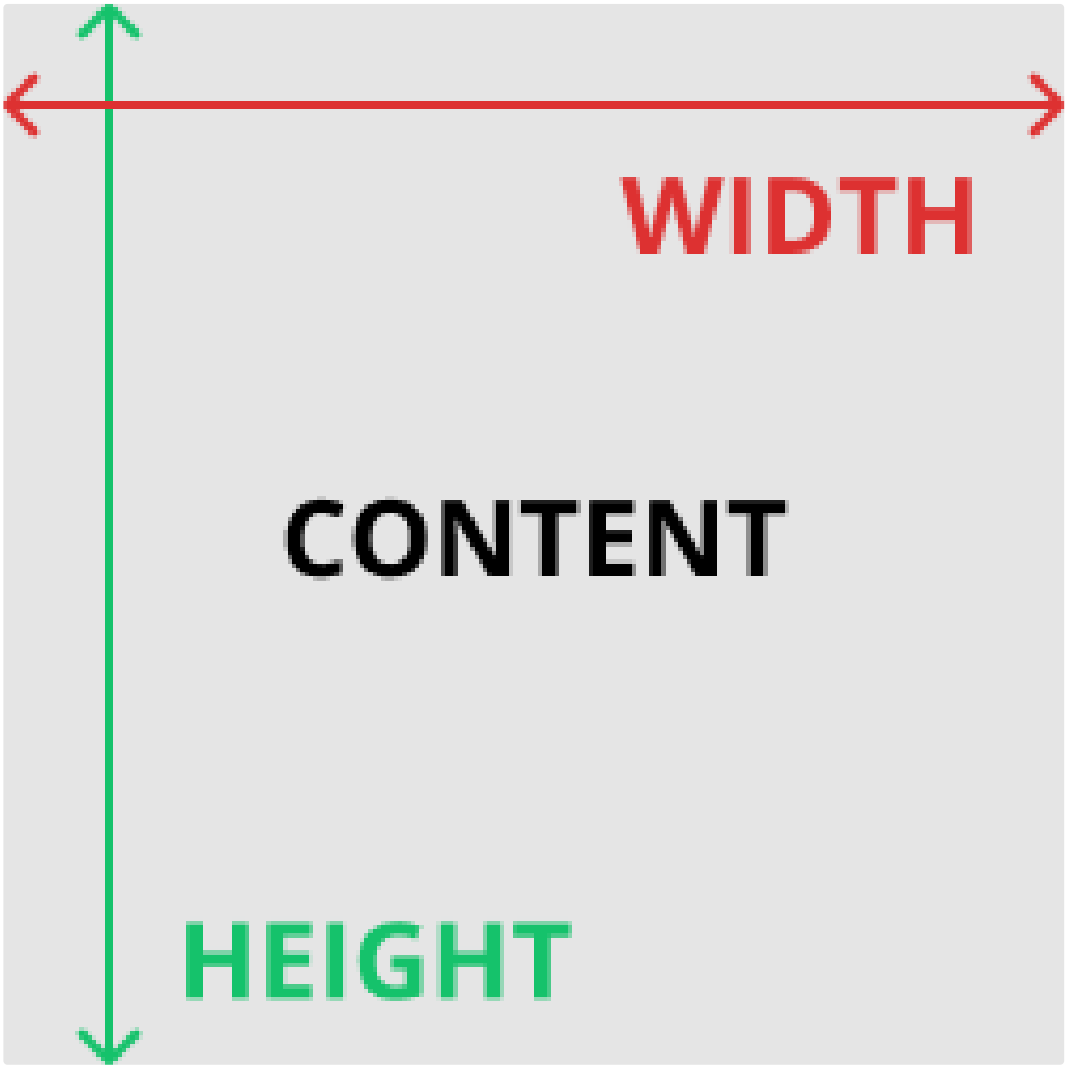
Before you dive deeper, you should understand that every element in web design is a rectangular box. You have probably heard this multiple times before, but this is an important concept that every developer should be aware of.

According to the box model concept, every element on a page is a rectangular box and may have width, height, padding, borders, and margins.

Now, let's see what the mysterious box model is all about.

Content

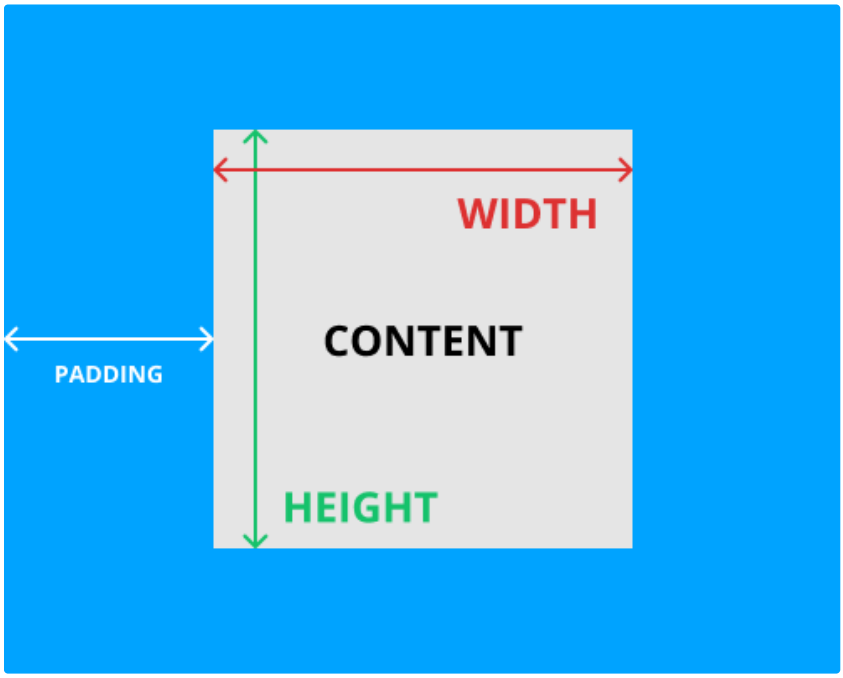
First, we have the content of the box itself, which has a height and width.



The content box has a height and width.

Padding

Next is padding - the space between the box's content and its border. Note that padding is *in addition* to the content's height and width, and is considered to be inside the element itself.



Padding extends outside the content box.

Border

Continuing our journey outward from the center of the CSS box model, we have the border: a line drawn around the content and padding of an element. The border property requires a new syntax that we've never seen before. First, we define the stroke width of the border, then its style, followed by its color.

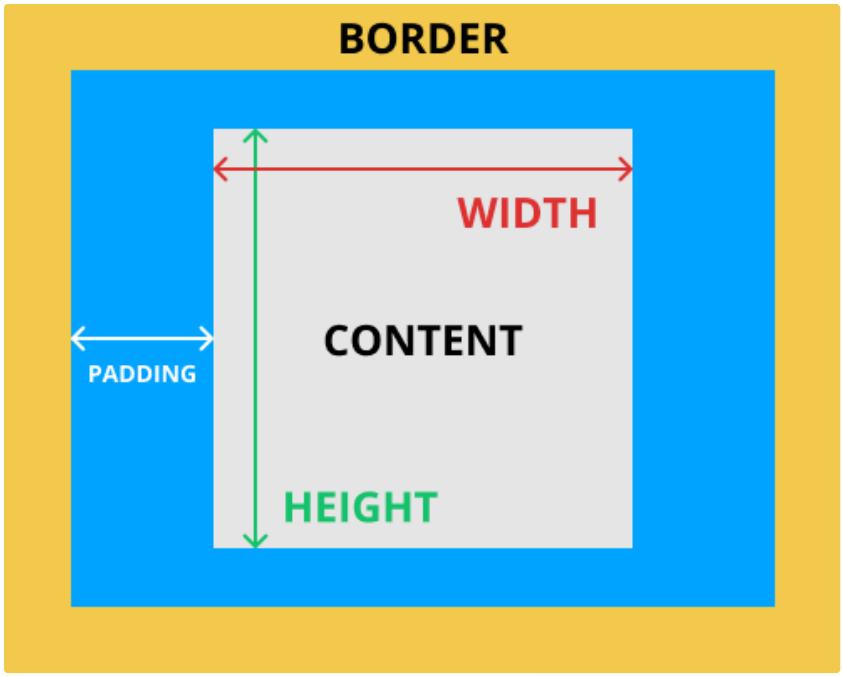
This tells the browser to draw a thin gray line around our heading. Notice how the border bumps right up next to the padding with no space in between. And, if you shrink your browser enough for the heading to be split over two lines, both the padding and the border will still be there.

Drawing a border around our entire heading makes it look a little 1990s, so how about we limit it to the bottom of the heading? Like padding, there are `-top`, `-bottom`, `-left`, and `-right` variants for the border property:

```
border-bottom: 1px solid #5D6063;
```

Borders are common design elements, but they're also invaluable for debugging. When you're not sure how a box is being rendered, add a border: 1px solid red; declaration to it. This will clearly show the box's padding, margin, and overall dimensions with just a single line of CSS. After you figured out why your stuff is broken, simply delete the rule.

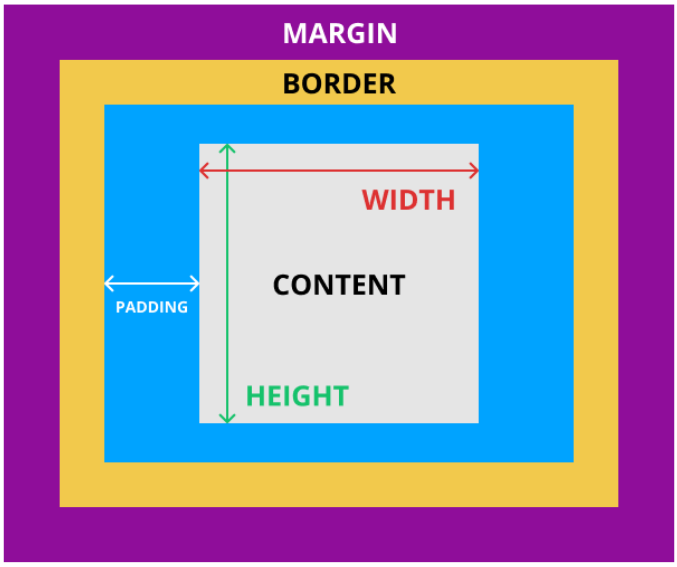
If you've ever used a table in a word processor or spreadsheet, then you should be familiar with borders. With CSS, you can add a border to just about anything.



The border is the line between the box's padding and margin.

Margin

Last is the margin, which surrounds the rest of the box. It is the space between the box and surrounding boxes.



The margin surrounds the rest of the box, and separates it from other boxes.

Display and Positioning: Inline & Block

The biggest distinction between outside display options is whether elements occupy the entire horizontal line they are on (remember that elements are organized hierarchically on the DOM) or if they only occupy the width they need and then the following element can be placed next to them on the same horizontal line.

Imagine there were two elements with the class box created with HTML like this:

```
<div class=”box”>Box 1</div>
<div class=”box”>Box 2</div>

```

In the CSS code if we set the `box` class display property to `block`, each rectangle would take up its own line and would be positioned one on top of the other. However, if we set the `display` property to `inline-block`, the rectangles would be displayed side by side on the same line. Here is what the CSS could look like:

```
.box{
  display: inline-block;
}

```

The value `inline` is most often used to highlight specific text within a larger text element, `span` elements are a common example. Elements set to `inline` display have no width or height and only occupy the space that their text property (or `.innerHTML` property) takes up. So in our `box` class example above, the rectangles would display on the same line but only occupy as much space as their text needs to display.

Display and Positioning: Z-index

When elements overlap, they are ordered on the z-axis (i.e., which element covers another). The z-index CSS property can be used to specify the z-order of these overlapping elements. Overlapping elements with a larger z-index cover those with a smaller one.

Elements may overlap for a number of reasons; for instance, elements positioned absolutely, or elements with negative values set for their `margin` property. In these instances we need a way to determine which element should be displayed on top. Without explicitly using `z-index` the last element written to the DOM (the last element you wrote in your code) will appear on top of all the others, and so on up the chain of your elements.

As an example, let's say we had two rectangles, positioned absolutely and overlapping each other-- the code for that could look like this:

```
<style>
.box {
  width: 200px;
   height: 200px;
  position: absolute;
}

#one {
  background: red;
  top: 100px;
  left: 150px;
}

#two {
  background: yellow;
  top: 80px;
  left: 100px;
}
</style>

<html>
  <div id=”one” class =”box”> Box One </div>
  <div id=”two” class =”box”> Box Two </div>
</html>

```

By default, the element with the id `two` would be displayed on top because it comes after element `one` in the code. However, if we wanted to display the first element on top we could use `z-index`, by giving each of the elements a numeric value for `z-index` and making sure that the value for element `one` is higher. The updated CSS could look like this:

```
#one {
  background: red;
  top: 100px;
  left: 150px;
  z-index:1;
}

#two {
  background: yellow;
  top: 80px;
  left: 100px;
  z-index: -1;
}

```

Notice that element `one` has been given a `z-index` value of `1`, and element `two` has been given a `z-index` value of `-1`. If more elements were involved we could use a wider range of values and the same rules would apply-- so that an element with `z-index` 100 would be displayed above an element with a `z-index` value of 99, and below.

Absolute vs Relative Units

Many CSS properties require a unit of measurement. There's a lot of units available, but the most common ones you'll encounter are `px` (pixel) and `em` (pronounced like the letter m). The former is what you would intuitively call a pixel, regardless of whether the user has a retina display or not, and the latter is the current font size of the element in question.

There are broadly two types of units of measurement for CSS properties, absolute and relative.

Absolute

- `px`
- `in`
- `mm`
- `cm`

Absolute measurements are set values regardless of anything having to do with your program or the browser. `px` is the most common absolute unit of measurement, and many font sizes on the web for example, are set to somewhere between 12px-30px, A font size set to `16px` will appear the same size no matter how big the screen. If however, you wanted to set a CSS property value based on some dynamic value, the width of a browser window for example, there are also relative units of measurement that can be used to define CSS properties.

Relative

- `%` - percentage of something, such as screen width
- `em` - A unit equivalent to the current font size - if 12px font, 2em would be 24px
- `vw` - units of viewport width (essentially the browser's rendering space). Each unit is 1/100th of width
- `vh` - the same as above but for viewport height

The em unit is very useful for defining sizes relative to some base font. For example, if you set the `font-size` of `body` to `16px`, you could then set other element's `font-size` value relative to that `16px`. Here's what that could look like:

```
body {
 font-size: 16px;
}

#one {
  font-size: 1.5em
}

#two {
  font-size: 0.5em
}

```

In this example, `one` would have font bigger than 16px, and `two` would have font smaller than 16px.

Typography

Text alignment

The aptly named text-align property defines the alignment of the text in an HTML element.

```
p {
  text-align: left;
}

```

Other accepted values are right, center, or justify.

Underlined Text

The text-decoration property determines whether text is underlined or not. By setting it to none, we can remove the default underline from all of our links. We'll discuss link styles in-depth later on.

```
a {
  text-decoration: none;
}

```

Deleted Text

The other common value for text-decoration is line-through to strike out "deleted" text. But, remember that meaning should always be conveyed through HTML—not CSS. It's better to use the `<ins>` and `<del>` elements instead of adding a line-through style to, say, an ordinary `<p>` element.

Line Height

Just as alignment isn't an arbitrary decision, neither is the space between text. In this section, we're concerned with the responsible use of three CSS properties:

- `margin-top` (or `padding-top`)
- `margin-bottom` (or `padding-bottom`)
- `line-height`

The first two should be pretty familiar by now, and they define the vertical space between separate paragraphs. The new `line-height` property determines the amount of space between lines in the same paragraph. In traditional typography, `line-height` is called "leading" because printers used little strips of lead to increase the space between lines of text.

Fonts

Font Family

`font-family` is another built-in CSS property that defines the typeface for whatever element you selected. It accepts multiple values because not all users will have the same fonts installed.

When using fonts on the web, you must first consider what fonts are available to your users. Every operating system, be it Windows, OS X, or Linux, comes with a set of pre-installed fonts that you can use for customizing your website. For a complete list of "web-safe" fonts, follow this link.

The way it works is fairly simple. When using the font-family property, you specify the font(s) you want to use in your HTML.

Then, the browser, starting from left to right, looks at the font(s) you've specified and checks to see if it can render the text using the font(s) you've provided. If it can't use the first font, then the browser moves to the next font, and so-on.

The purpose for specifying multiple fonts is because not all fonts are available on every operating system. So, specifying multiple, similar fonts ensures users have a consistent experience regardless of the operating system they are using.

Font Weight & Style

In CSS, font weights are expressed as numeric values between 100 and 900. Fortunately, there are relatively standardized, human-friendly terms for each of these numeric values. “Black” usually means 900, “bold” is 700, “regular” is 400, etc. Most families don’t supply a face for every single weight. For example, Roboto is missing “extra light” (200), “semi bold” (600), and “extra bold” (800).

It’s worth noting that each style and weight combination is designed as an entirely distinct face. In a high-quality font family, the condensed styles aren’t simply squashed versions of the roman faces, nor is the bold face merely a thicker version. Each letter in every face is hand-crafted to ensure it provides a uniform flow to its text.

This is particularly apparent in the italic and roman faces of many serif fonts. For instance, the lowercase “a” in Century Schoolbook FS (the font you’re reading right now) takes on a completely different shape when it’s italicized.

Emphasis & Importance

For emphasized (usually italics) words, use the `<em>` tag.

```
<p>
  We <em>have</em> to buy the latest version of the pet hair remover vacuum, the
  floor is covered with fur!
</p>

```

Which results in:

We *have* to buy the latest version of the pet hair remover vacuum, the floor is covered with fur!

For important words, use the `<strong>` tag. By default, `<strong>` elements are displayed in bold, but keep in mind that it is only the browser’s default behavior. Don’t use `<strong>` only to put some text in bold, but rather to give it more importance.

```
<p>
  My dog is the most <strong>important</strong> creature in my life right now.
</p>

```

External Fonts

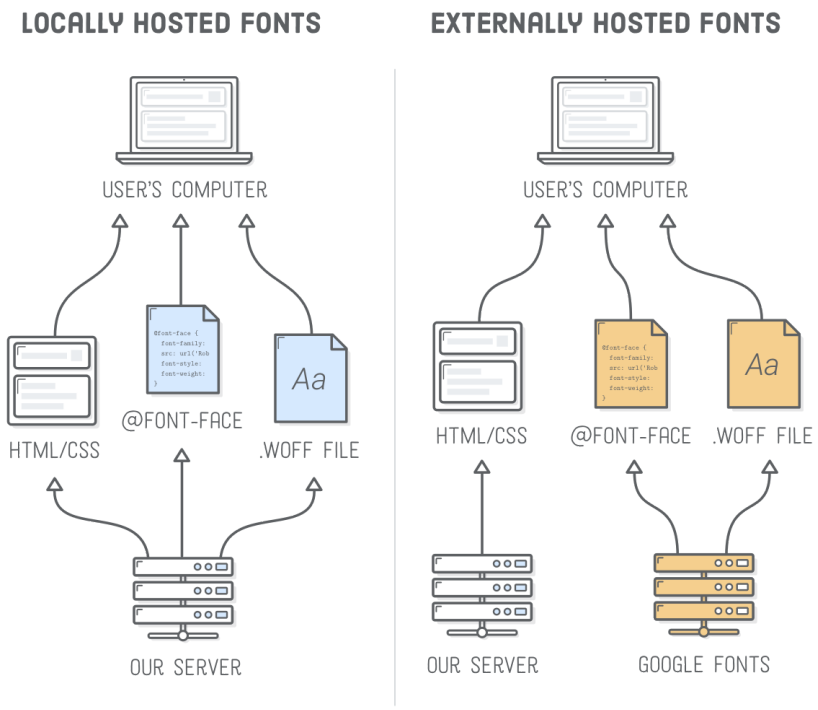
There are a number of ways to host fonts from external sources. One commonly used example is Google Fonts, which provides a great number of fonts free for use in web projects. If you go to the Google Fonts website linked below, you can select a font and then you will be provided with the line of code to link your font of choice. That link goes in the `head` section of your code like this:

```
<head>
  <link href="https://fonts.googleapis.com/css?family=Montserrat&display=swap" rel="stylesheet">
</head>

```

In the CSS portion of your code, you could then set an elements property like this:

```
.box{
  font-family: 'Montserrat', sans-serif;
}

```

In this example the Google font I selected was ‘Montserrat’.



An example of using local fonts vs externally hosted fonts.

Colors

Colors in CSS can be specified by the following methods:

1. Hexadecimal colors
2. RGB colors
3. Predefined/Cross-browser color names
4. RGBA colors
5. HSL colors
6. HSLA colors

Let's talk about the first 3 since those are the most common.

Hexadecimal Colors

A hexadecimal color is specified with: `#RRGGBB`, where the RR (red), GG (green) and BB (blue) hexadecimal integers specify the components of the color. All values must be between 00 and FF.

For example, the `#0000ff` value is rendered as blue, because the blue component is set to its highest value (ff) and the others are set to 00.

Example

Define different HEX colors:

```
#p1 {
  background-color: #ff0000;
}

#p2 {
  background-color: #00ff00;
}

#p3 {
  background-color: #0000ff;
}

```

RGB Colors

An RGB color value is specified with the `rgb()` function, which has the following syntax:

`rgb(red, green, blue)`

Each parameter (red, green, and blue) defines the intensity of the color and can be an integer between 0 and 255 or a percentage value (from 0% to 100%).

For example, the `rgb(0,0,255)` value is rendered as blue, because the blue parameter is set to its highest value (255) and the others are set to 0.

Also, the following values define equal color: `rgb(0,0,255)` and `rgb(0%,0%,100%)`.

Example

Define different RGB colors:

```
#p1 {
  background-color: rgb(255, 0, 0);
}
```

```
4
5  #p2 {
6    background-color: rgb(0, 255, 0);
7  }
8
9  #p3 {
10   background-color: rgb(0, 0, 255);
11 }
12
```

Predefined/Cross-browser Color Names

140 color names are predefined in the HTML and CSS color specification.

There's quite a few of these - check out this list to see more.

Doc: CSS Flexbox

Introducing Flexbox

The **flexbox** or flexible box model in CSS is a one-dimensional layout model that has flexible and efficient layouts with distributed spaces among items to control their alignment structure ie., it is a layout model that provides an easy and clean way to arrange items within a container. Flexbox can be useful for creating small-scales layouts & is responsive and mobile-friendly.

To use flexbox set the `display` property of a div to `flex`. The items inside that element will automatically become flex items, and you can then use the flexbox syntax in your CSS code.

Axes and Direction with Flexbox

The Flexbox model relies on two axes: the main axis and the cross axis. The main axis is defined by flex-direction, which has four possible values:

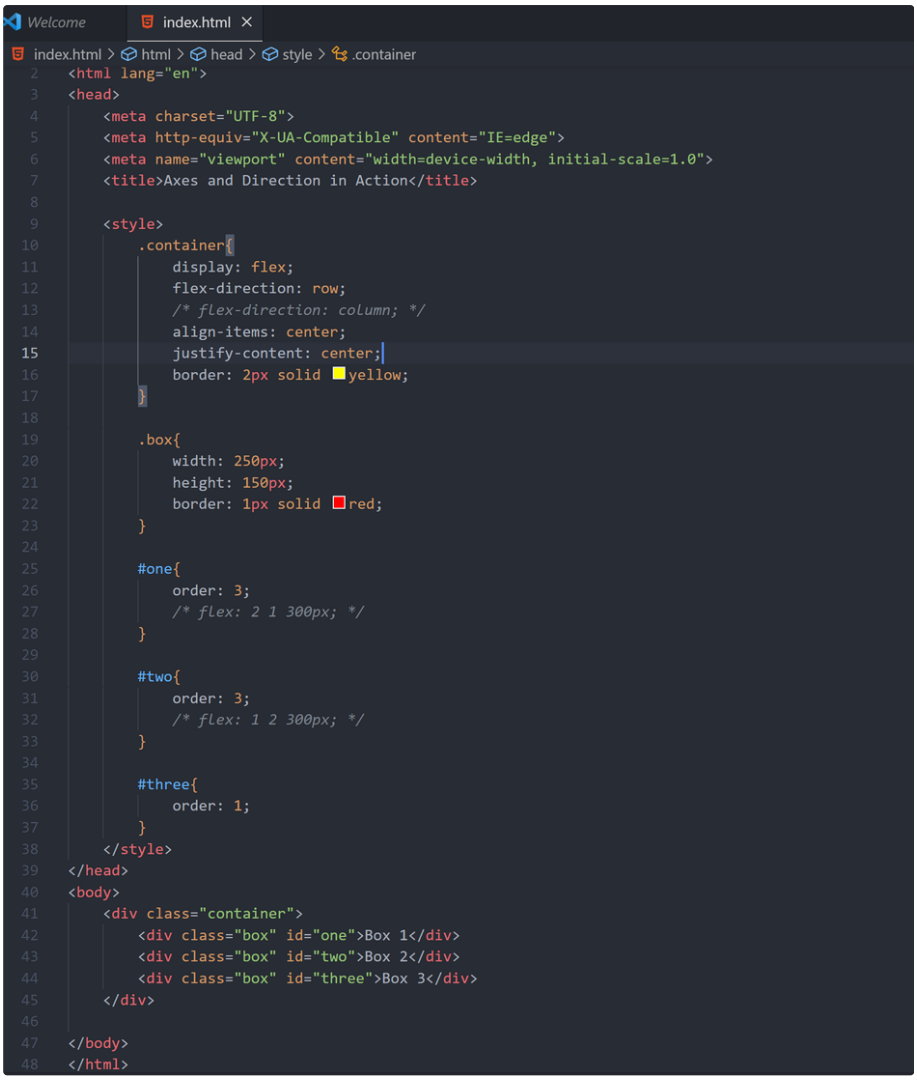
- `row`
- `row-reverse`
- `column`
- `column-reverse`

The two row settings will create the main axis horizontally - or `inline` direction. The two column settings will create the main axis vertically - or `block` direction. `block` or `inline` here refer to the CSS display settings which we have covered previously.

The axis determines the flow of your content - you can think of this as being either rows or columns - and they will be determined when you start aligning and justifying content within a flex container.

Axes and Direction in Action

After setting an element's display to flex, the next thing you will usually want to state is whether the elements inside the container should be laid out in rows or columns. You can do this using the `flex-direction` property, and setting its value to either column or row.



```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Axes and Direction in Action</title>

    <style>
        .container{
            display: flex;
            flex-direction: row;
            /* flex-direction: column; */
            align-items: center;
            justify-content: center;
            border: 2px solid yellow;
        }

        .box{
            width: 250px;
            height: 150px;
            border: 1px solid red;
        }

        #one{
            order: 3;
            /* flex: 2 1 300px; */
        }

        #two{
            order: 3;
            /* flex: 1 2 300px; */
        }

        #three{
            order: 1;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="box" id="one">Box 1</div>
        <div class="box" id="two">Box 2</div>
        <div class="box" id="three">Box 3</div>
    </div>

</body>
</html>
```

And this is the result which is displayed in browser:

To set the layout of the items in a flex container to either a row or column use the `flex-direction` property like this:

```
.container{
  display:flex;
  flex-direction: row
}
```

Axes and direction are important concepts for understanding flexbox. They are both conceptual and technical which can be tricky. One suggestion is to try and draw your flex containers out first in a notebook. This can be helpful for mapping out axes and direction.

Further Research

For more on axis and direction with flexbox see the documentation here.

Ordering Elements with Flexbox

There are three ways to explicitly set the order in which items will appear in a grid.

1. Moving the HTML code for the elements themselves to reorder
2. Appending `-reverse` to `row` or `column` will reverse the order in the specified row or column
3. Using the `order` property of the individual items inside the grid

Ordering Elements Demo

`flex-direction:row;` will lay elements out from left to right. But `flex-direction:row-reverse` will flip that order and display elements from right to left.

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Axes and Direction in Action</title>

    <style>
        .container{
            display: flex;
            flex-direction: row-reverse;
            /* flex-direction: column; */
            align-items: center;
            justify-content: center;
            border: 2px solid yellow;
        }

        .box{
            width: 250px;
            height: 150px;
            border: 1px solid red;
        }

        #one{
            order: 3;
            /* flex: 2 1 300px; */
        }

        #two{
            order: 2;
            /* flex: 1 2 300px; */
        }

        #three{
            order: 1;
        }
    </style>
</head>
<body>
    <div class="container">
        <div class="box" id="one">Box 1</div>
        <div class="box" id="two">Box 2</div>
        <div class="box" id="three">Box 3</div>
    </div>

</body>
</html>
```

And this is the result which is displayed in browser:

The row and columns settings for flex elements can be reversed by appending `-reverse` to either property value.

Summary

The order of flex items is by default determined by the order they appear in your code. You can explicitly change this order using either `-reverse` or with the CSS property `order`.

Further Research

For more information on ordering flex items, you can see this section of the flexbox MDN article.

Aligning Items & Justifying Content with Flexbox

To align items on the cross axis use `align-items` with the possible values:

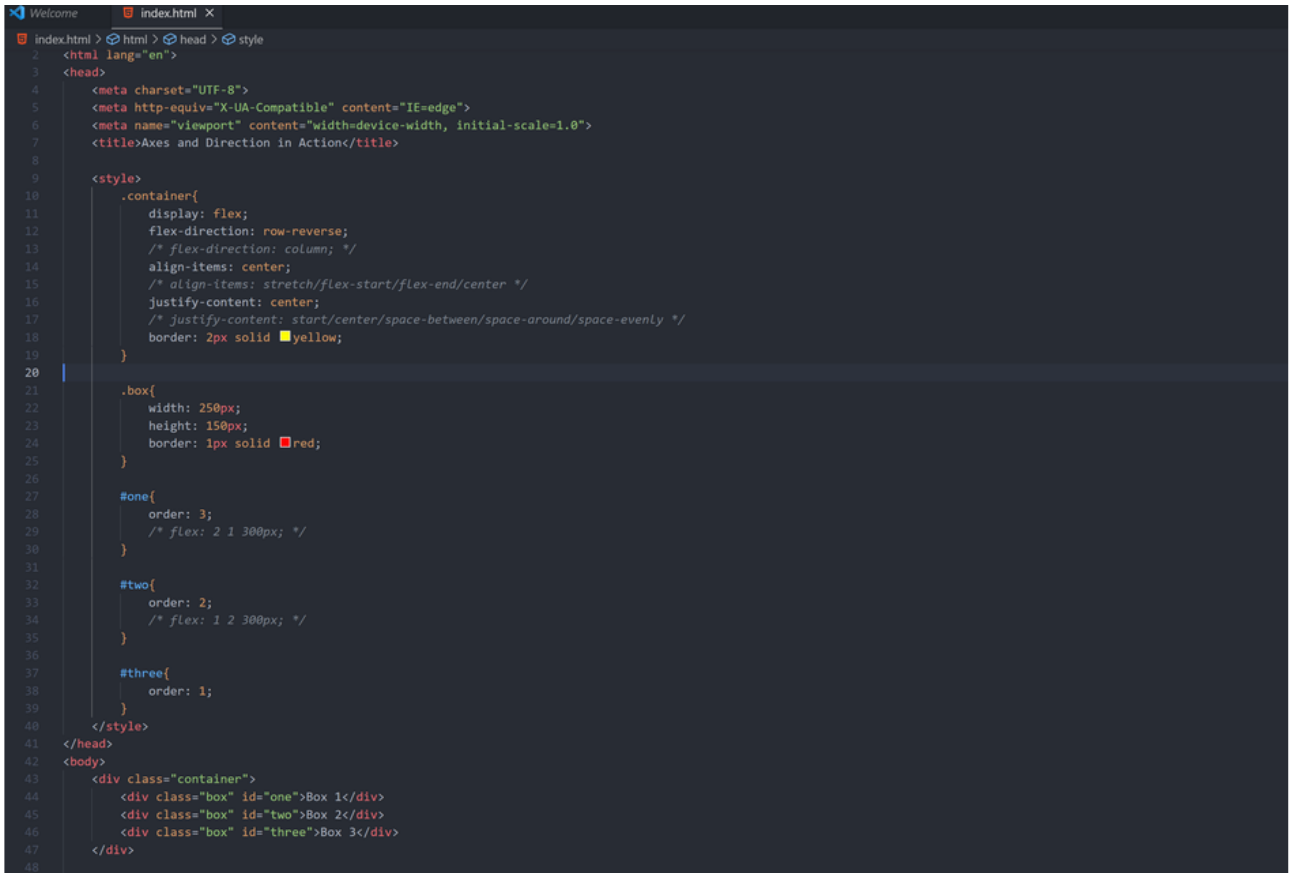
- stretch
- flex-start
- flex-end
- center

To justify content on the main axis use `justify-content`, which has the possible values:

- flex-start
- flex-end
- center
- space-around
- space-between
- space-evenly

Aligning & Justifying in Action

Aligning, justifying, and distributing is what flexbox is all about. Up next, we will focus in on two of the most powerful properties that flex introduces to achieve such easy and elegant layouts.

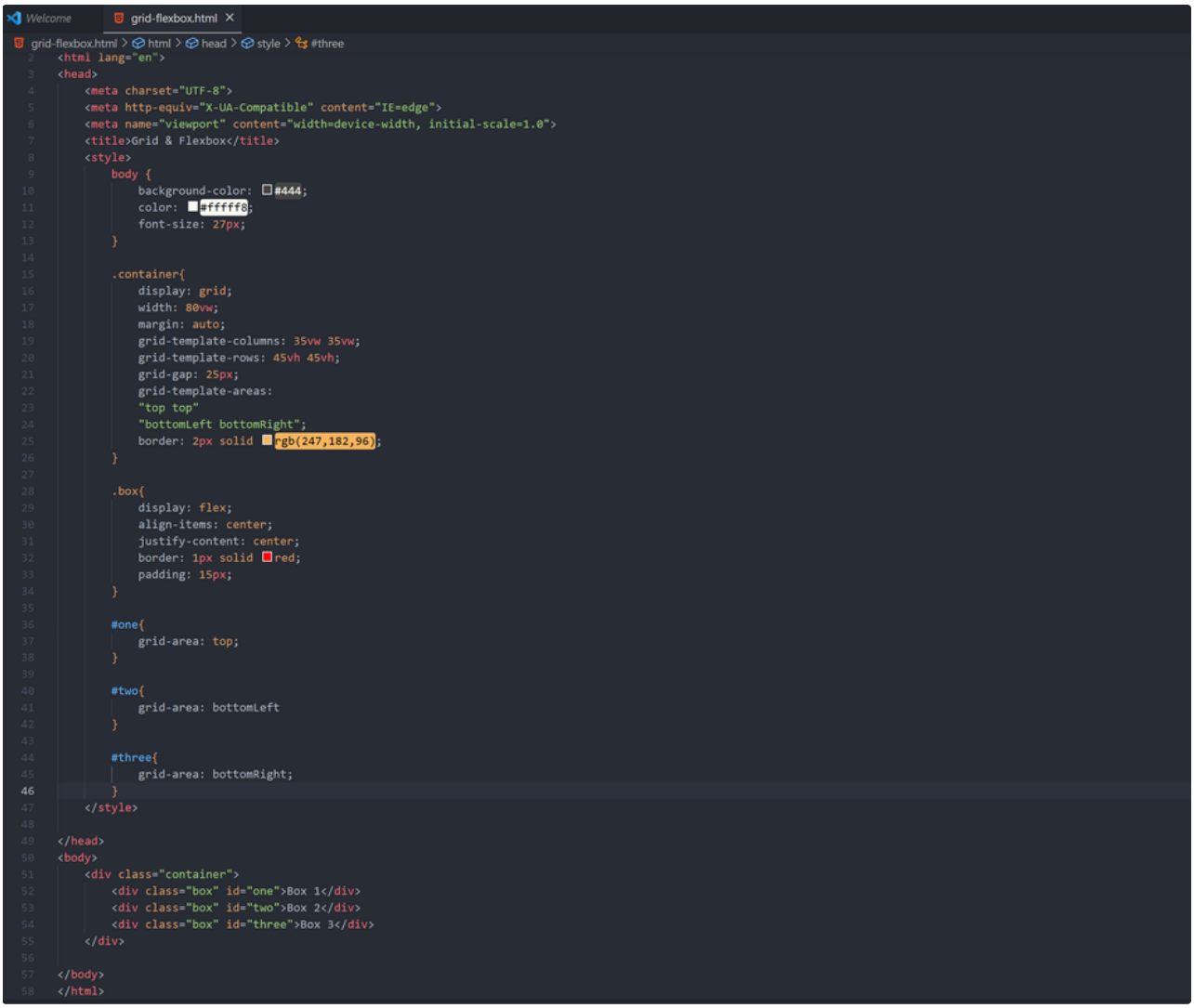


By setting different values for the properties `align-items` and `justify-content` you can easily create elegant distribution of elements across the available space.

Doc: CSS Grid

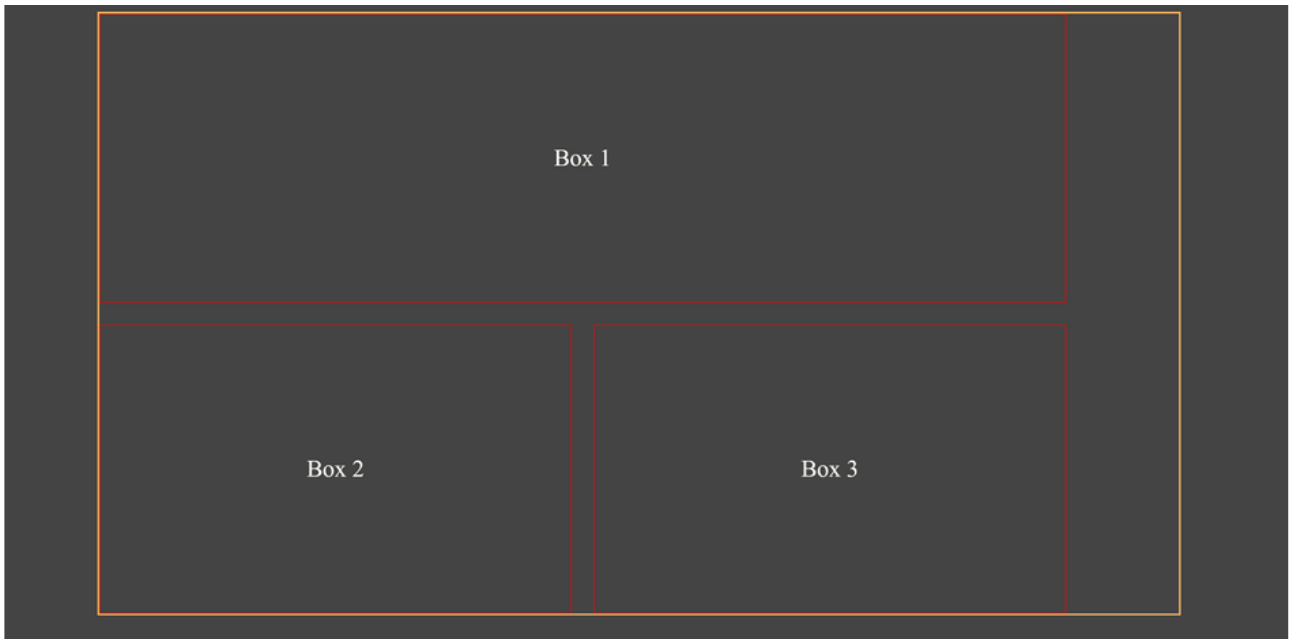
Grid vs Flexbox

An excellent way to use flexbox and grid together is by creating the layout of a webpage with grid and then organizing the flow of the content with flex. Let's see an example of this!

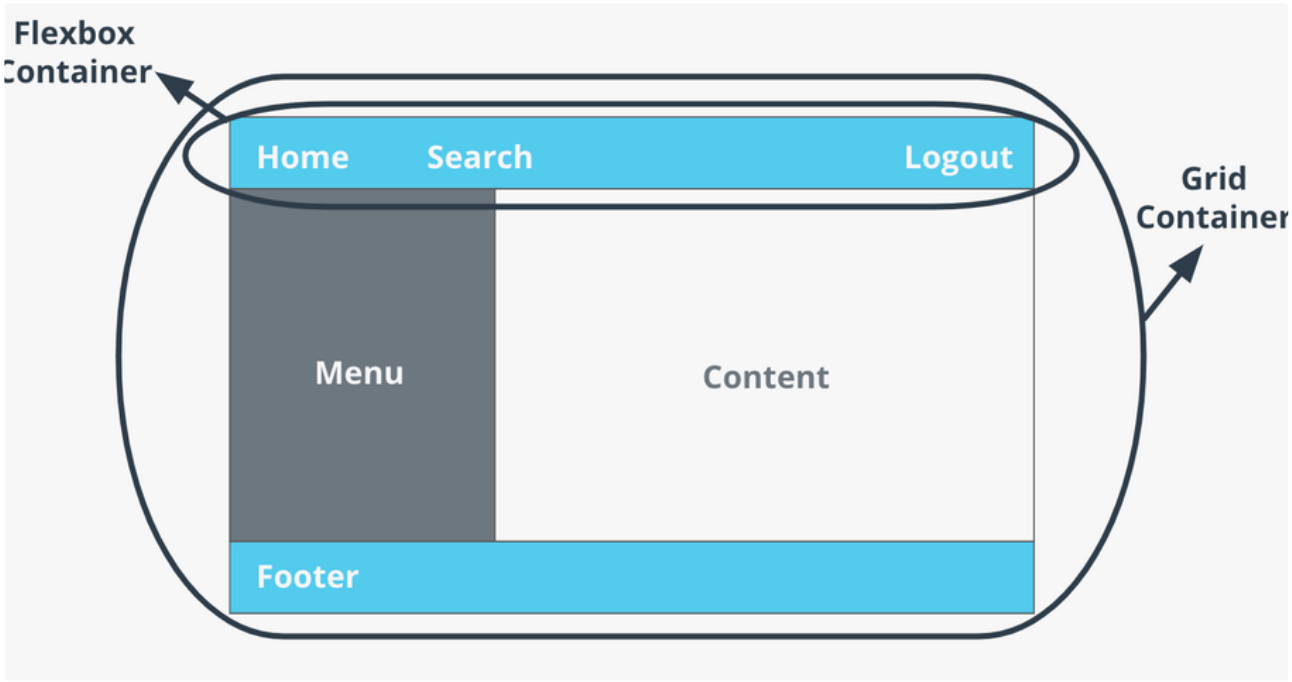


```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Grid & Flexbox</title>
    <style>
        body {
            background-color: #444;
            color: #ffffff8;
            font-size: 27px;
        }

        .container{
            display: grid;
            width: 80vw;
            margin: auto;
            grid-template-columns: 35vw 35vw;
            grid-template-rows: 45vh 45vh;
            grid-gap: 25px;
            grid-template-areas:
            "top top"
            "bottomLeft bottomRight";
            border: 2px solid rgb(247,182,96);
        }

        .box{
            display: flex;
            align-items: center;
            justify-content: center;
            border: 1px solid red;
            padding: 15px;
        }

        #one{
            grid-area: top;
        }

        #two{
            grid-area: bottomLeft
        }

        #three{
            grid-area: bottomRight;
        }
    </style>

</head>
<body>
    <div class="container">
        <div class="box" id="one">Box 1</div>
        <div class="box" id="two">Box 2</div>
        <div class="box" id="three">Box 3</div>
    </div>

</body>
</html>
```

And this is the result which is displayed in browser:



To use CSS Grid set the `display` property of the container element to `grid`.



Flexbox can work within Grid

Recap

CSS Grid v. Flexbox

- Grid is two dimensional, while Flex is one
- Grid is layout first, while Flexbox is content first
- Flex is for components of an app, Grid is for the app layout itself

CSS Grid does not replace Flexbox. Each can achieve things that the other is not capable of, and in fact, a true master can use Flexbox and CSS Grid together in harmony to create the ultimate webpage layout.

At the highest level CSS Grid excels at creating layouts for a webpage, while Flexbox is a master of content flow for each element that makes up the page layout.

Further Research

For more information on CSS Grid and Flexbox together see the articles below:

- Beginner's Guide to choosing between CSS Grid and Flexbox
- The Ultimate CSS Battle: Grid vs. Flexbox
- Does CSS Grid Replace Flexbox?

Rows & Columns

After setting the display property of your container div to grid, the next step is to set the rows and columns in your grid which can be done with the CSS properties:

- `grid-template-columns`
- `grid-template-rows`

And to define gutters between rows and columns you can use the property `grid-gap` on the parent container that has the `display` property set to `grid`.

Rows & Columns in Action

`grid-template-columns` is the property that defines the column layout of your grid - that is how many sections the page should be divided into vertically. The values for this property are the explicit value for each column and the number of columns is defined implicitly by the number of values entered. For example:

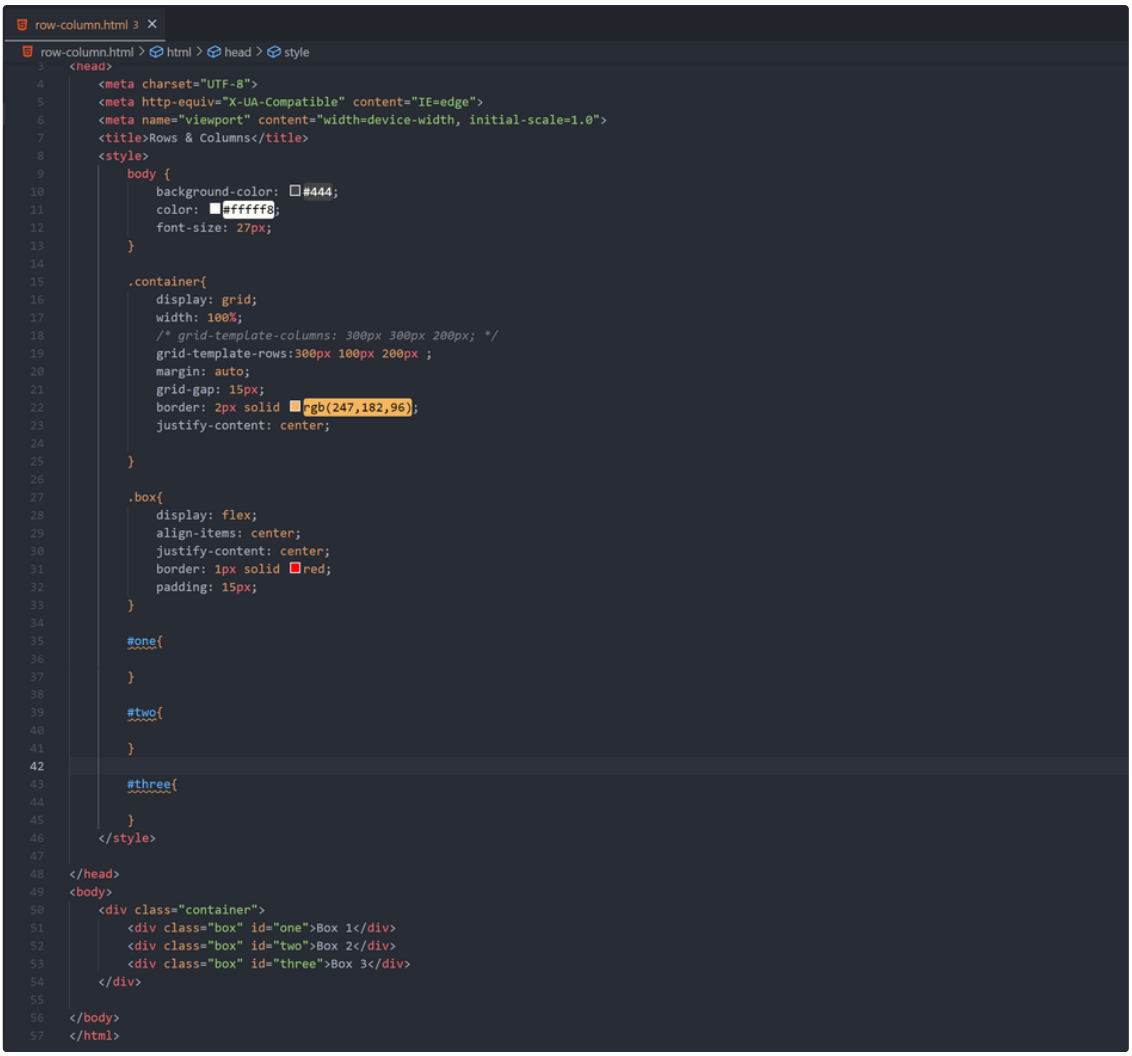
```
grid-template-columns: 60px 60px;

```

would create two columns of 60px each. If you had two items inside your grid and didn't set their position explicitly, the first item would be placed in the first column and the second item in the second. `grid-template-rows` follows the same logic, so that:

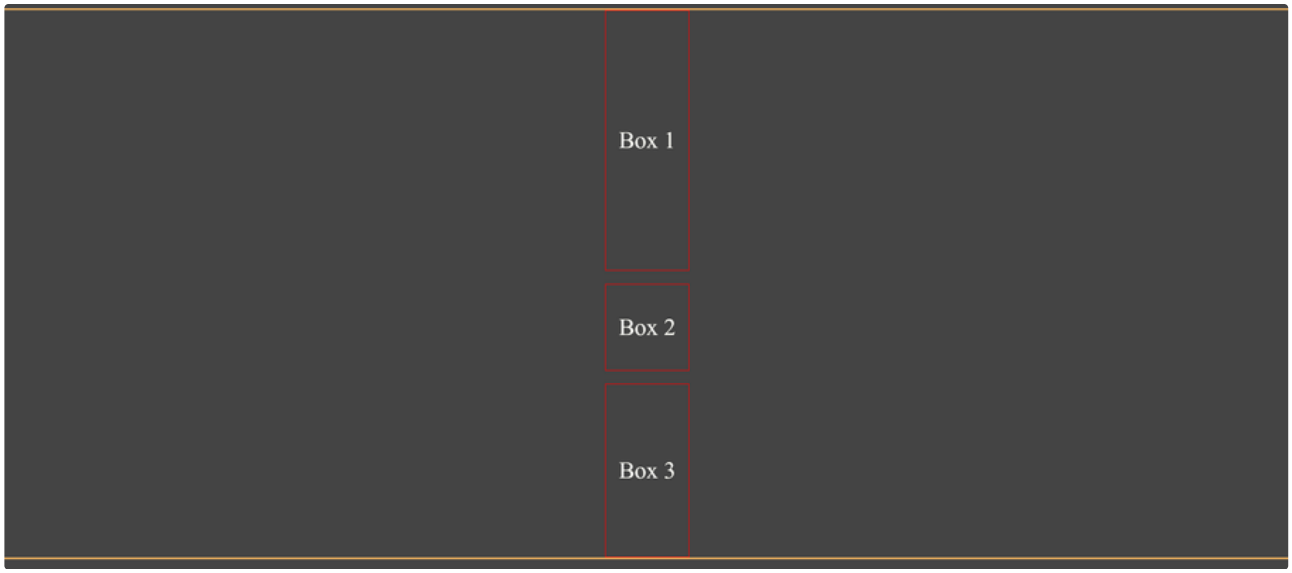
```
grid-template-columns: 60px 60px;
grid-template-rows: 160px 60px;

```

would create a grid with two columns and two rows.

Let's take a look at this example.



And this is the result which is displayed in browser:



The layout of a grid is defined using `grid-template-columns` and `grid-template-rows`.

A CSS Grid is made up of rows and columns which are defined using the CSS properties `grid-template-columns` and `grid-template-rows`, which take as values the size of each track.

Further Research

To explore grid setup deeper, you can check out this cool grid visualizer and generator.

Grid Areas

The `grid-area` property specifies a particular area or set of rows and columns that a grid item occupies. It is applied to the grid item itself with CSS. Here is an example:

```
.item{
  grid-area: 1/2/3/3
}

```

Because `grid-area` is shorthand for the properties: `grid-row-start`, `grid-column-start`, `grid-row-end` and `grid-column-end`, the code above places the item from rows 1-3, and columns 2-3.

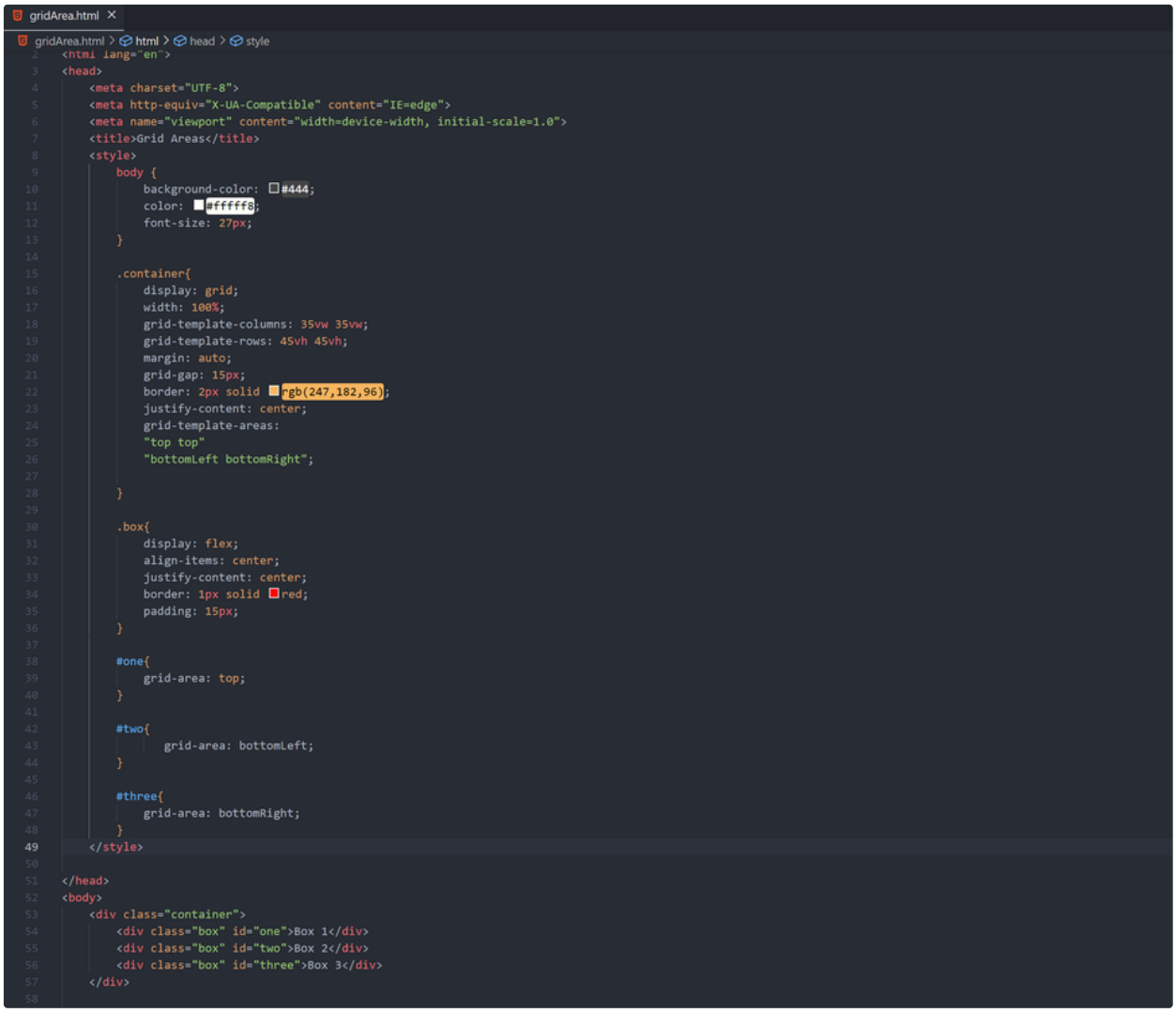
Working with Grid Areas

The `grid-area` property defines the space an element takes up in the grid by setting values for the row it starts and ends in, and the column it starts and ends in. In practice it could look like this:

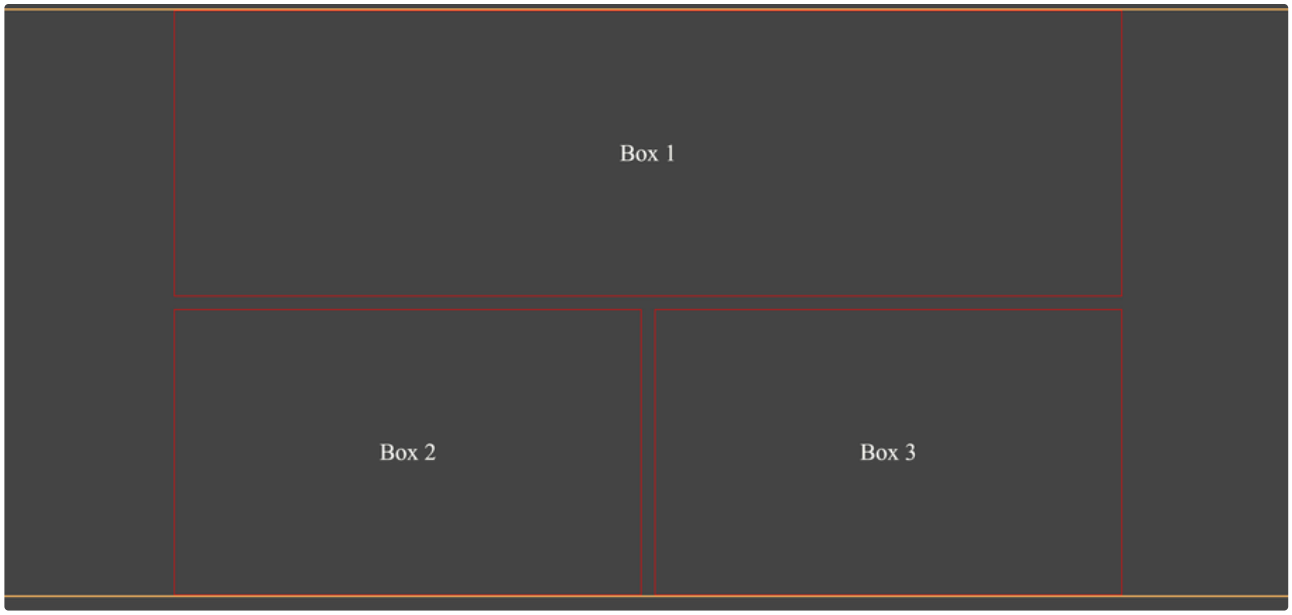
```
#one {
  /* row start/column start/ row end/ column end */
  grid-area: 1/2/3/3;
}

```

In this example the element with the `id` , `one` would start at the first row and the first column, and end at the third row (which is the end of the second row if there is no third row) and the third column.

Let's take a look at this example.



And this is the result which is displayed in browser:



The `grid-area` CSS property is a shorthand property for `grid-row-start`, `grid-column-start`, `grid-row-end` and `grid-column-end`, and it defines the area that an element occupies in a grid.

Grid Areas Summary

`grid-template-areas` is the property used to name the rows and columns of a grid and to set its layout. It could look like this:

```
  .container {
      display:grid;
      grid-template-columns: 300px 300px 300px;
      grid-template-rows: 250px 600px;
      grid-template-areas:
      "hd hd hd hd hd hd hd hd"
      "sd sd sd main main main main main"
      "ft ft ft ft ft ft ft ft";
  }

```

The named areas in the grid are then assigned to each element according to where you want them to be displayed in the grid:

```
.header {
  grid-area: hd;
}

```

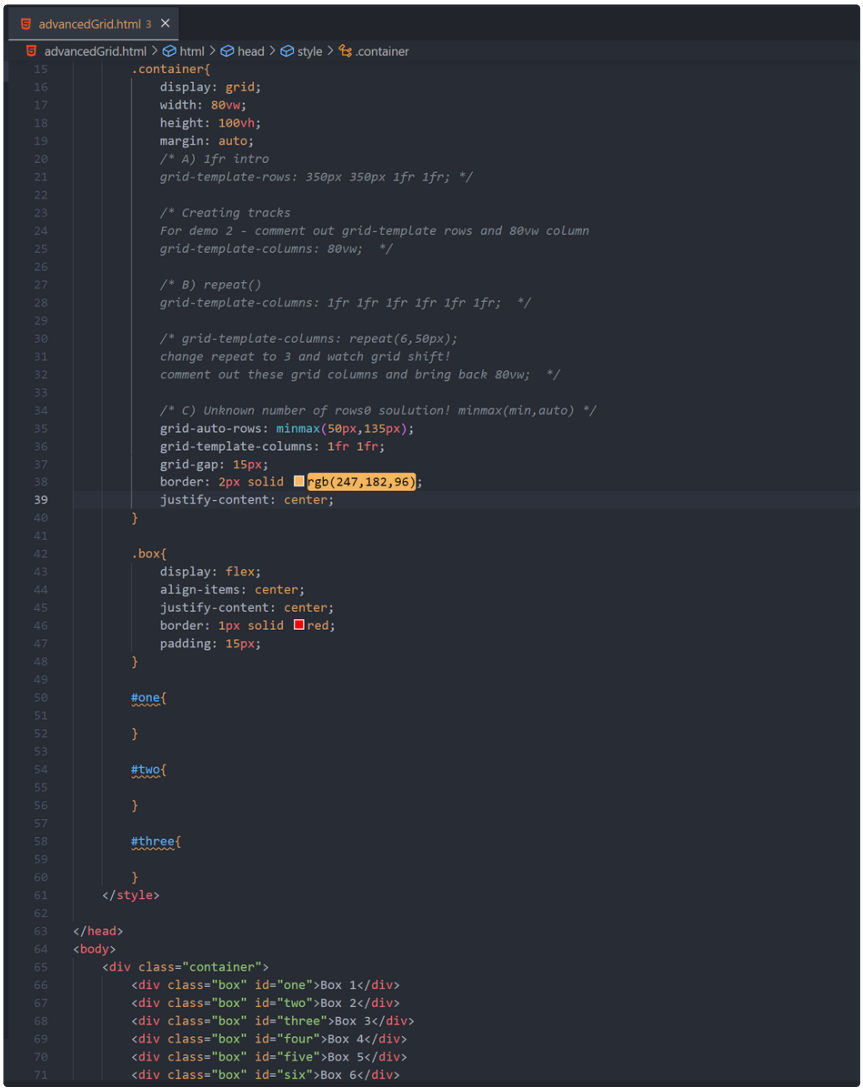
In the example above the element with the class `header` will stretch across the entire first row of columns because we have assigned it the `grid-area` `hd`, and we have defined the area `hd` with the value for `grid-template-areas` in the parent element.

Advanced Grid

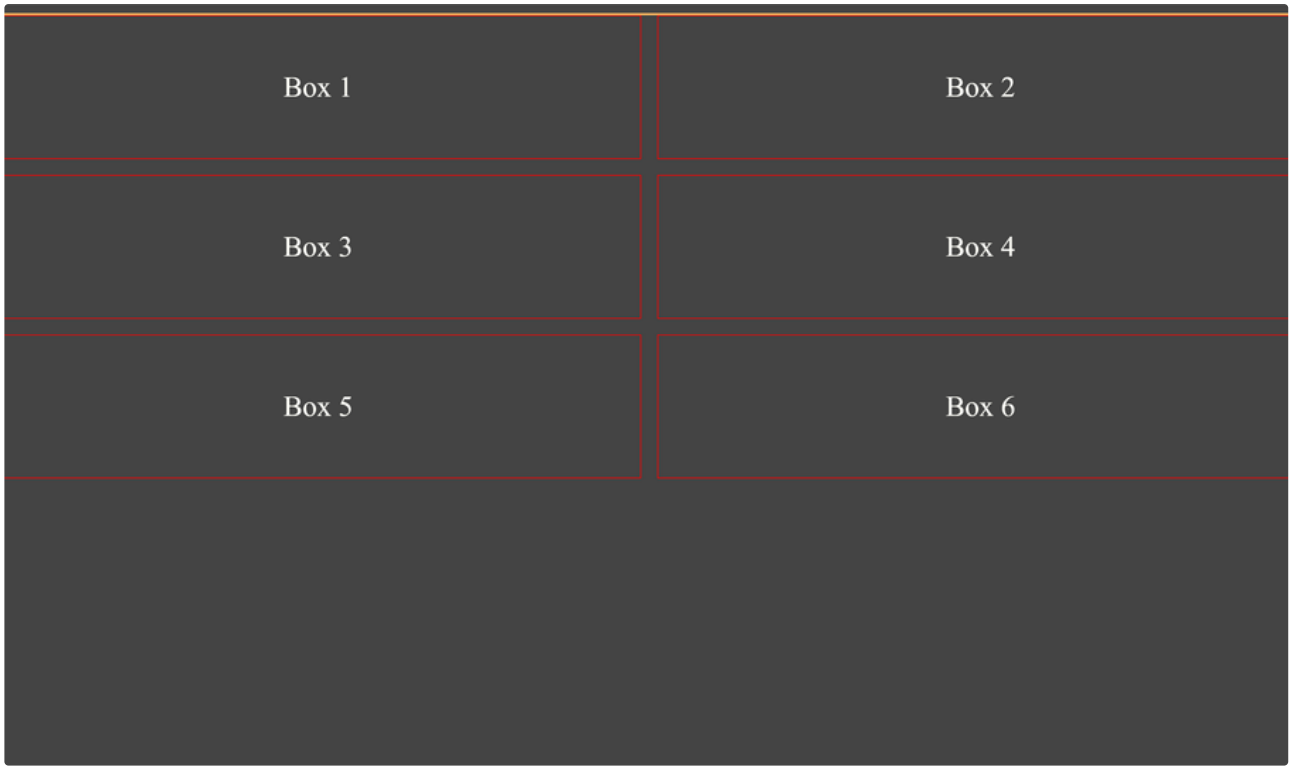
CSS Grid includes advanced capabilities for creating large and complex grids. Some of these are:

- the `fr` Unit
- Track listings with `repeat()` notation
- Track sizing and `minmax()`

Let's explore these advanced features in this example.



And this is the result which is displayed in browser:


Box 1
Box 2
Box 3
Box 4
Box 5
Box 6

Advanced Grid Playground

Advanced Grid Recap

The `fr` unit represents a fraction of the available space in the grid container.

The `repeat()` notation can be used to quickly layout many tracks for large grids. For example:

```
grid-template-columns: 1fr 1fr 1fr 1fr 1fr 1fr 1fr;

```

Could be written with repeat notation like this:

```
grid-template-columns: repeat(7, 1fr);

```

The `grid-auto-rows` property can be used to generate the number of rows based on the content and available space. The following code:

```
grid-auto-rows: minmax(100px, auto);

```

Would create rows that are at least 100px tall and can be as tall as the content inside them demands.

Further Research

For even more control over how your content is laid out, check out this article titled How Items Flow Into a CSS Grid.

Doc: Creating Responsive Layouts

Media Queries

While media queries can be used for a variety of things and in a number of ways, we are going to focus on what are called **breakpoints**, which are the `viewport` width at which we want our design to change. We then write the code inside that media query, with a set breakpoint, that we want to go into effect only when the viewport width that the app is being viewed on is at least the breakpoint width. Only the CSS that we want to change needs to go here - the original CSS rules will all still apply, and only the new CSS rules written inside the media query will override any pre-existing rules.

Key Term

viewport - the area of the window in which web content can be seen. We use the dimensions of the viewport (usually the width, but sometimes the height) as the basis of our media queries.

For more information about `viewport` see

- What is a viewport?
- Using the viewport meta tag to control layout on mobile browsers

Adding Media Queries in Code

Media queries are used to set different style rules for different devices or sized screens. We use breakpoints to set the condition of a media query. The logic is:

```
@media(feature:value)

```

Here media features are aspects of the device that our media (website) is being viewed on. The media feature we are most interested in for this lesson is `width`, which allows us to evaluate the viewport width of the browser and set conditions based on that evaluation. We actually write this feature `min-width` (or `max-width`) because `width` is one of many media features that are range features, which means they can be prefixed with `min-` or `max-` to express constraints, which is what we're looking for with our breakpoints! If the constraint of the breakpoint (viewport width being in the range below our breakpoint) is broken (the width is larger than the breakpoint) the new CSS rule takes effect. Here is an example of how that could look in action:

```
@media(min-width:900px) {
  body{
   background:red;
 }
}
```

Media queries are used to create responsive layouts using breakpoints. Below is an example of the syntax that is used for creating media queries:

```
@media(min-width:1100px) {
  body{
    font-size: 27px;
  }
}

```

In the example above, if the browser width of the webpage being viewed is above 1100px wide, then the font-size would become 27px.

Media queries can help change the layout for different screen sizes

Multiple Breakpoints

We have seen how to set a breakpoint and use Media Queries to create different layouts for smaller screens and larger screens, but there are some development moments that will call for 3 possible layouts.

A simple example would be creating 2 different breakpoints so that up to x width one set of CSS rules apply, then between x and y width a second set would apply, and then for anything beyond a width of y a third set of CSS rules would apply.

Here is an example of what that code could look like:

```
/* Anything smaller than first breakpoint 600px */
.container {
  // rules for small screen
}

/* Medium Screens */
@media (min-width: 600px) and (max-width:900px) {
  .container {
    // rules for medium-sized screen
  }
}

/* Large Screens */
@media (min-width:901px) {
  .container {
    // rules for large screen
  }
}
```

Complex media queries can be built using the keyword `and` to bound CSS rules between a range using `min-width` and `max-width`.

Further Research

Media Queries are actually a vast landscape of possibility, most of which you will probably never use - but, having a strong grasp of media queries and responsive breakpoints is essential for a web developer. For more information see the MDN docs entry on using Media Queries.

(Doc) Week 2: JavaScript & The DOM

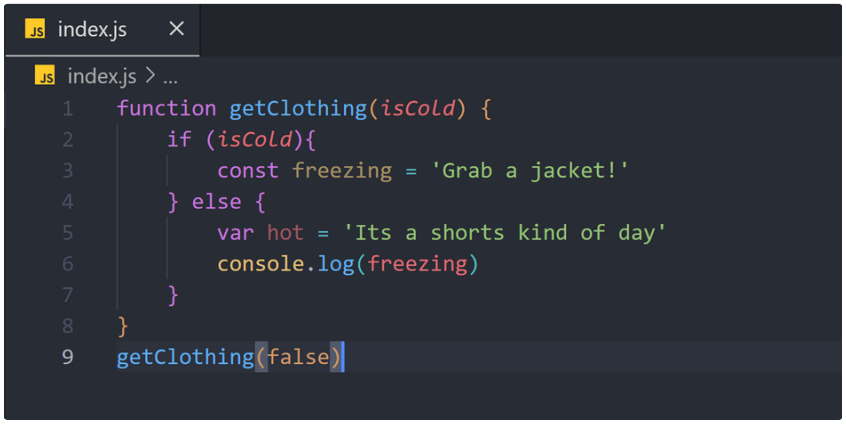
Doc: JavaScript Syntax

Let & Const

There are now two new ways to declare variables in JavaScript: **let** and **const**.

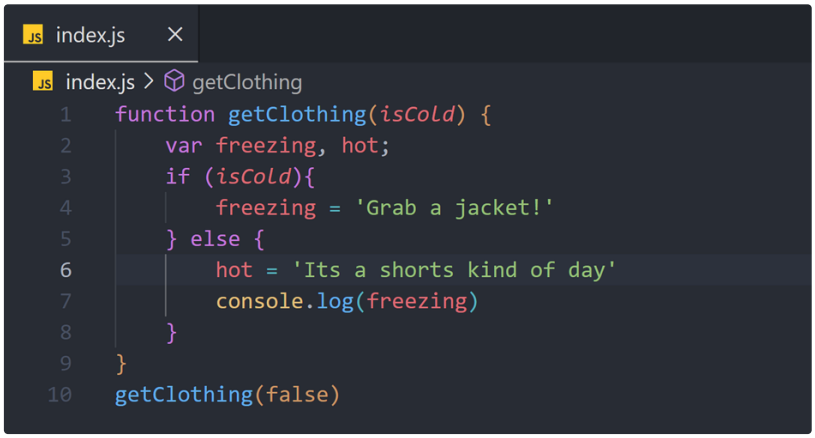
Up until now, the only way to declare a variable in JavaScript was to use the keyword `var`. To understand why `let` and `const` were added, it's probably best to look at an example of when using `var` can get us into trouble.

Take a look at the following code.



Hoisting

Hoisting is a result of how JavaScript is interpreted by your browser. Essentially, before any JavaScript code is executed, all variables declared with `var` are "hoisted", which means they're raised to the top of the function scope. So at run-time, the `getClothing()` function actually looks more like this…



let and const

Variables declared with `let` and `const` eliminate this specific issue of hoisting because they're scoped **to the block**, not to the function. Previously, when you used `var`, variables were either scoped *globally* or *locally* to an entire function scope.

If a variable is declared using `let` or `const` inside a block of code (denoted by curly braces `{ }`), then the variable is stuck in what is known as the **temporal dead zone** until the variable's declaration is processed. This behavior prevents variables from being accessed only until after they've been declared.

```
index.js
index.js > ...
1  function getClothing(isCold) {
2      if (isCold){
3          let freezing = 'Grab a jacket!'
4      } else {
5          let hot = 'Its a shorts kind of day'
6          console.log(freezing)
7      }
8  }
9  getClothing(false)
```

Rules for using let and const

`let` and `const` also have some other interesting properties.

- Variables declared with `let` can be reassigned, but can't be redeclared in the same scope.
- Variables declared with `const` must be assigned an initial value, but can't be redeclared in the same scope, and can't be reassigned.

```
index.js
index.js > ...
1  let instructor = 'James';
2  instructor = 'Richard';
3  console.log(instructor)
```

Use cases

The big question is when should you use `let` and `const`? The general rule of thumb is as follows:

- use `let` when you plan to reassign new values to a variable, and
- use `const` when you don't plan on reassigning new values to a variable.

Since `const` is the strictest way to declare a variable, we suggest that you always declare variables with `const` because it'll make your code easier to reason about since you know the identifiers won't change throughout the lifetime of your program. If you find that you need to update a variable or change it, then go back and switch it from `const` to `let`.

That's pretty straightforward, right? But what about `var`?

What about var?

Is there any reason to use `var` anymore? *Not really*.

There are some arguments that can be made for using `var` in situations where you want to globally define variables, but this is often considered bad practice and should be avoided. From now on, we suggest ditching `var` in place of using `let` and `const`.

Template Literals

Prior to ES6, the old way to concatenate strings together was by using the string concatenation operator (`+`).

```
const student = {
  name: 'Richard Kalehoff',
  guardian: 'Mr. Kalehoff'
};

const teacher = {
  name: 'Mrs. Wilson',
  room: 'N231'
}

let message = student.name + ' please see ' + teacher.name + ' in ' + teacher.room + ' to pick up your report ca

```

> ***Returns:*** *Richard Kalehoff please see Mrs. Wilson in N231 to pick up your report card.*

This works alright, but it gets more complicated when you need to build multi-line strings.

```
let note = teacher.name + ',\n\n' +
  'Please excuse ' + student.name + '.\n' +
  'He is recovering from the flu.\n\n' +
  'Thank you,\n' +
  student.guardian;

```

> ***Returns:***
> *Mrs. Wilson,*

> *Please excuse Richard Kalehoff.*
> *He is recovering from the flu.*

> *Thank you,*
> *Mr. Kalehoff*

However, that's changed with the introduction of *template literals* (previously referred to as "template strings" in development releases of ES6).

> ***NOTE:*** *As an alternative to using the string concatenation operator (* `+`), you can use the String's `concat()` method, but both options are rather clunky for simulating true string interpolation.

Template Literals

Template literals are essentially string literals that include embedded expressions.

Denoted with backticks (` `) instead of single quotes (`''`) or double quotes (`""`), template literals can contain placeholders which are represented using `${expression}`. This makes it *much easier* to build strings.

Here's the previous examples using template literals.

```
let message = `${student.name} please see ${teacher.name} in ${teacher.room} to pick up your report card.`;

```

> ***Returns:*** *Richard Kalehoff please see Mrs. Wilson in N231 to pick up your report card.*

By using template literals, you can drop the quotes along with the string concatenation operator. Also, you can reference the object's properties inside expressions.

Here's the examples using template literals:

```
let teacher = {
    name: "Hieu Mai",
    job: "Software Engineer"
}

let literals = `Hello my name is ${teacher.name}. I am ${teacher.job}.`
// output: Hello my name is Hieu Mai. I am Software Engineer.
```

```
> literals = `Hello my name is ${teacher.name}.
  I am ${teacher.job}.`
<- 'Hello my name is Hieu Mai.\nI am Software Engineer.'
> literals =
  `Hello!
  My name is ${teacher.name}.
  I am ${teacher.job}.`
<- 'Hello! \nMy name is Hieu Mai.\nI am Software Engineer.'
> literals
<- 'Hello! \nMy name is Hieu Mai.\nI am Software Engineer.'
>
```

TIP: *Embedded expressions inside template literals can do more than just reference variables. You can perform operations, call functions and use loops inside embedded expressions!*

Destructuring

In ES6, you can extract data from arrays and objects into distinct variables using *destructuring*.

This probably sounds like something you've done before, for example, look at the two code snippets below that extract data using pre-ES6 techniques:

```
const point = [10, 25, -34];

const x = point[0];
const y = point[1];
const z = point[2];

console.log(x, y, z);

```

> ***Prints:*** *10 25 -34*

The example above shows extracting values from an array.

```
const gemstone = {
  type: 'quartz',
  color: 'rose',
  carat: 21.29
};

const type = gemstone.type;
const color = gemstone.color;
const carat = gemstone.carat;

console.log(type, color, carat);

```

> ***Prints:*** *quartz rose 21.29*

And this example shows extracting values from an object.

Both are pretty straightforward, however, neither of these examples are actually using destructuring.

So what exactly is *destructuring*?

Destructuring

Destructuring borrows inspiration from languages like Perl and Python by allowing you to specify the elements you want to extract from an array or object *on the left side of an assignment*. It sounds a little weird, but you can actually achieve the same result as before, but with much less code; and it's still easy to understand.

Let's take a look at both examples rewritten using destructuring.

Destructuring values from an array

```
const point = [10, 25, -34];

const [x, y, z] = point;

console.log(x, y, z);

```

> ***Prints:*** *10 25 -34*

In this example, the brackets `[ ]` represent the array being destructured and `x`, `y`, and `z` represent the variables where you want to store the values from the array. Notice how you don't have to specify the indexes for where to extract the values from because the indexes are implied.

> ***TIP:*** *You can also ignore values when destructuring arrays. For example,* `const [x, , z] = point;` ignores the `y` coordinate and discards it.

What do you expect to be the value of `second` after running the following code?

Destructuring values from an object

```
const gemstone = {
  type: 'quartz',
  color: 'rose',
  carat: 21.29
};

const {type, color, carat} = gemstone;

console.log(type, color, carat);

```

> ***Prints:*** *quartz rose 21.29*

In this example, the curly braces `{ }` represent the object being destructured and `type`, `color`, and `carat` represent the variables where you want to store the properties from the object. Notice how you don't have to specify the property from where to extract the values. Because `gemstone` has a property named `type`, the value is automatically stored in the `type` variable. Similarly, `gemstone` has a `color` property, so the value of `color` automatically gets stored in the `color` variable. And it's the same with `carat`.

> ***TIP:*** *You can also specify the values you want to select when destructuring an object. For example,* `let {color} = gemstone;` will only select the `color` property from the `gemstone` object.

Object Literal Shorthand

A recurring trend in ES6 is to remove unnecessary repetition in your code. By removing unnecessary repetition, your code becomes easier to read and more concise. This trend continues with the introduction of new *shorthand* ways for initializing objects and adding methods to objects.

Let's see what those look like.

Object literal shorthand

You've probably written code where an object is being initialized using the same property names as the variable names being assigned to them.

But just in case you haven't, here's an example.

```
let type = 'quartz';
let color = 'rose';
let carat = 21.29;

const gemstone = {
  type: type,
  color: color,
  carat: carat
};

console.log(gemstone);

```

> ***Prints:*** *Object {type: "quartz", color: "rose", carat: 21.29}*

Do you see the repetition? Doesn't `type: type`, `color: color`, and `carat:carat` seem redundant?

The good news is that you can remove those duplicate variables names from object properties _if_ the properties have the same name as the variables being assigned to them.

Check it out!

Speaking of shorthand, there's also a shorthand way to add methods to objects.

To see how that looks, let's start by adding a `calculateWorth()` method to our `gemstone` object. The `calculateWorth()` method will tell us how much our gemstone costs based on its `type`, `color`, and `carat`.

```
let type = 'quartz';
let color = 'rose';
let carat = 21.29;

const gemstone = {
  type,
  color,
  carat,
  calculateWorth: function() {
    // will calculate worth of gemstone based on type, color, and carat
  }
};

```

In this example, an anonymous function is being assigned to the property `calculateWorth`, but is the **function** keyword *really* needed? In ES6, it's not!

Shorthand method names

Since you only need to reference the gemstone's `calculateWorth` property in order to call the function, having the function keyword is redundant, so it can be dropped.

```
let gemstone = {
  type,
  color,
  carat,
  calculateWorth() { ... }
};
```

Family of for Loops

The **for...of loop** is the most recent addition to the family of for loops in JavaScript.

It combines the strengths of its siblings, the **for loop** and the **for...in loop**, to loop over any type of data that is **iterable** (meaning it follows the iterable protocol). By default, this includes the data types String, Array, Map, and Set—notably absent from this list is the `Object` data type (i.e. `{}`). Objects are not iterable, by default.

Before we look at the for...of loop, let's first take a quick look at the other for loops to see where they have weaknesses.

The for loop

The for loop is obviously the most common type of loop there is, so this should be a quick refresher.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (let i = 0; i < digits.length; i++) {
  console.log(digits[i]);
}

```

> ***Prints:***
> *0*
> *1*
> *2*
> *3*
> *4*
> *5*
> *6*
> *7*
> *8*
> *9*

Really the biggest downside of a for loop is having to keep track of **the counter** and **exit condition**.

In this example, we're using the variable `i` as a counter to keep track of the loop and to access values in the array. We're also using `digits.length` to determine the exit condition for the loop. If you just glance at this code, it can sometimes be confusing exactly what's happening; especially for beginners.

While for loops certainly have an advantage when looping through arrays, some data is not structured like an array, so a for loop isn't always an option.

The for...in loop

The for...in loop improves upon the weaknesses of the for loop by eliminating the counting logic and exit condition.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const index in digits) {
  console.log(digits[index]);
}

```

> ***Prints:***
> *0*

```
1
2
3
4
5
6
7
8
9
```

But, you still have to deal with the issue of using an **index** to access the values of the array, and that stinks; it almost makes it more confusing than before.

Also, the for...in loop can get you into big trouble when you need to add an extra method to an array (or another object). Because for...in loops loop over all enumerable properties, this means if you add any additional properties to the array's prototype, then those properties will also appear in the loop.

```
Array.prototype.decimalfy = function() {
  for (let i = 0; i < this.length; i++) {
    this[i] = this[i].toFixed(2);
  }
};

const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const index in digits) {
  console.log(digits[index]);
}

```

Prints:

```
0
1
2
3
4
5
6
7
8
9
function() {
  for (let i = 0; i < this.length; i++) {
    this[i] = this[i].toFixed(2);
  }
}
```

Gross! This is why for...in loops are discouraged when looping over arrays.

> ***NOTE:*** *The* ***forEach loop*** *is another type of for loop in JavaScript. However,* `forEach()` is actually an array method, so it can only be used exclusively with arrays. There is also no way to stop or break a forEach loop. If you need that type of behavior in your loop, you'll have to use a basic for loop.

For...of loop

The **for...of loop** is used to loop over any type of data that is *iterable*.

You write a **for...of** loop almost exactly like you would write a **for...in** loop, except you swap out `in` with `of` and you can drop the **index**.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const digit of digits) {
  console.log(digit);
}

```

> ***Prints:***
> *0*
> *1*
> *2*
> *3*
> *4*
> *5*
> *6*
> *7*
> *8*
> *9*

This makes the for...of loop the most concise version of all the for loops.

> ***TIP:*** *It's good practice to use plural names for objects that are collections of values. That way, when you loop over the collection, you can use the singular version of the name when referencing individual values in the collection. For example,* `for (const button of buttons) {...}`.

But wait, there's more! The for...of loop also has some additional benefits that fix the weaknesses of the for and for...in loops.

You can stop or break a for...of loop at anytime.

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const digit of digits) {
  if (digit % 2 === 0) {
    continue;
  }
  console.log(digit);
}

```

> ***Prints:***
> *1*
> *3*
> *5*
> *7*
> *9*

And you don't have to worry about adding new properties to objects. The for...of loop will only loop over the values in the object.

```
Array.prototype.decimalfy = function() {
  for (i = 0; i < this.length; i++) {
    this[i] = this[i].toFixed(2);
  }
};

const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

for (const digit of digits) {
  console.log(digit);
}

```

> ***Prints:***
> *0*
> *1*
> *2*
> *3*
> *4*
> *5*
> *6*
> *7*
> *8*
> *9*

This time, the properties were *not* printed out to the console, like we saw on the prior page.

Spread operator

Spread operator

The **spread operator**, written with three consecutive dots (`...`), is new in ES6 and gives you the ability to expand, or *spread*, iterable objects into multiple elements.

Let's take a look at a few examples to see how it works.

```
const books = ["Don Quixote", "The Hobbit", "Alice in Wonderland", "Tale of Two Cities"];
console.log(...books);

```

> ***Prints:*** *Don Quixote The Hobbit Alice in Wonderland Tale of Two Cities*

```
const primes = new Set([2, 3, 5, 7, 11, 13, 17, 19, 23, 29]);
console.log(...primes);

```

> ***Prints:*** *2 3 5 7 11 13 17 19 23 29*

If you look at the output from the examples, notice that both the array and set have been expanded into their individual elements. So how is this useful?

> ***NOTE:*** *Sets are a new built-in object in ES6 that we'll cover in more detail in a future lesson.*

Combining arrays with concat

One example of when the spread operator can be useful is when combining arrays.

If you've ever needed to combine multiple arrays, prior to the spread operator, you were forced to use the Array's `concat()` method.

```
const fruits = ["apples", "bananas", "pears"];
const vegetables = ["corn", "potatoes", "carrots"];
const produce = fruits.concat(vegetables);
console.log(produce);

```

> ***Prints:*** *["apples", "bananas", "pears", "corn", "potatoes", "carrots"]*

...Rest Parameter

If you can use the spread operator to *spread* an array into multiple elements, then certainly there should be a way to bundle multiple elements back into an array, right?

In fact, there is! It's called the *rest parameter*, and it's another new addition in ES6.

Rest parameter

The **rest parameter**, also written with three consecutive dots (`...`), allows you to represent an indefinite number of elements as an array. This can be helpful in a couple of different situations.

One situation is when assigning the values of an array to variables. For example,

```
const order = [20.17, 18.67, 1.50, "cheese", "eggs", "milk", "bread"];
const [total, subtotal, tax, ...items] = order;
console.log(total, subtotal, tax, items);

```

> ***Prints:*** *20.17 18.67 1.5 ["cheese", "eggs", "milk", "bread"]*

This code takes the values of the `order` array and assigns them to individual variables using destructuring (as you saw in the Destructuring section earlier in this lesson). `total`, `subtotal`, and `tax` are assigned the first three values in the array, however, `items` is where you want to pay the most attention.

By using the rest parameter, `items` is assigned the *rest* of the values in the array (as an array).

Variadic functions

Another use case for the rest parameter is when you're working with variadic functions. **Variadic functions** are functions that take an indefinite number of arguments.

For example, let's say we have a function called `sum()` which calculates the sum of an indefinite amount of numbers. How might the `sum()` function be called during execution?

```
sum(1, 2);
sum(10, 36, 7, 84, 90, 110);
sum(-23, 3000, 575000);

```

There's literally an endless number of ways the `sum()` function could be called. Regardless of the amount of numbers passed to the function, it should always return the total sum of the numbers.

Using the arguments object

In previous versions of JavaScript, this type of function would be handled using the arguments object. The **arguments object** is an array-like object that is available as a local variable inside all functions. It contains a value for each argument being passed to the function starting at 0 for the first argument, 1 for the second argument, and so on.

If we look at the implementation of our `sum()` function, then you'll see how the arguments object could be used to handle the variable amount of numbers being passed to it.

```
function sum() {
  let total = 0;
  for(const argument of arguments) {
    total += argument;
  }
```

```
  return total;
}

```

Now this works fine, but it does have its issues:

1. If you look at the definition for the `sum()` function, it doesn't have any parameters.
   - This is misleading because we know the `sum()` function can handle an indefinite amount of arguments.
2. It can be hard to understand.
   - If you've never used the arguments object before, then you would most likely look at this code and wonder where the arguments object is even coming from. Did it appear out of thin air? It certainly looks that way.

Using the rest parameter

Fortunately, with the addition of the rest parameter, you can rewrite the `sum()` function to read more clearly.

```
function sum(...nums) {
  let total = 0;
  for(const num of nums) {
    total += num;
  }
  return total;
}

```

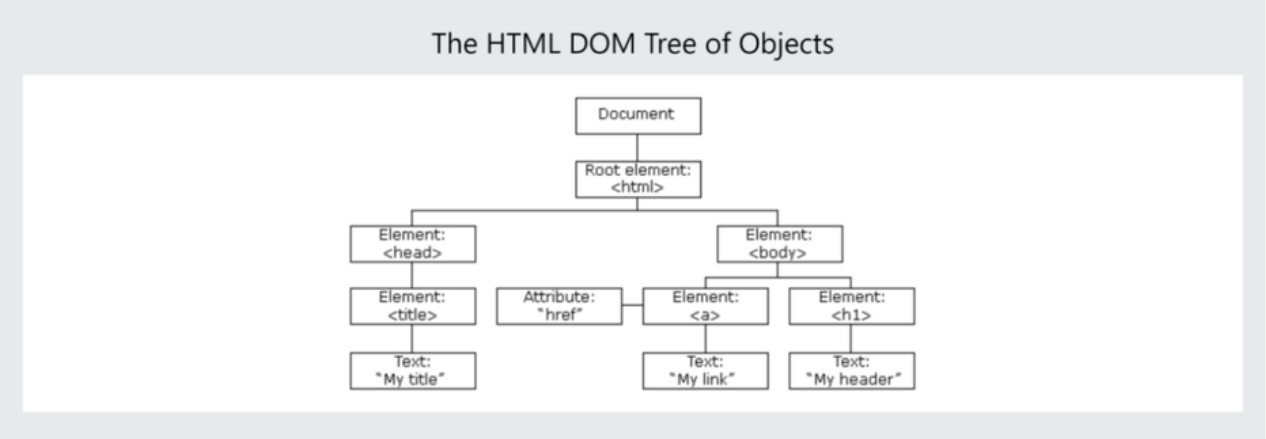
This version of the `sum()` function is both **more concise** and is **easier to read**. Remember, we use the `for...of loop` to loop over any type of data that is iterable. So we'll use `for...of` here rather than `for...in`.

THE-DOM

The HTML DOM (Document Object Model)

When a web page is loaded, the browser creates a Document Object Model of the page.

The HTML DOM model is constructed as a tree of Objects.



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

· JavaScript can change all the HTML elements in the page.

· JavaScript can change all the HTML attributes in the page.

· JavaScript can change all the CSS styles in the page.

· JavaScript can remove existing HTML elements and attributes.

· JavaScript can add new HTML elements and attributes.

· JavaScript can react to all existing HTML events in the page.

· JavaScript can create new HTML events in the page.

A JavaScript DOM object, keep in mind, is a tree-like structure with attributes and values. Consequently, a unique object provided by the browser, document, can be used to access the DOM.

Try this:

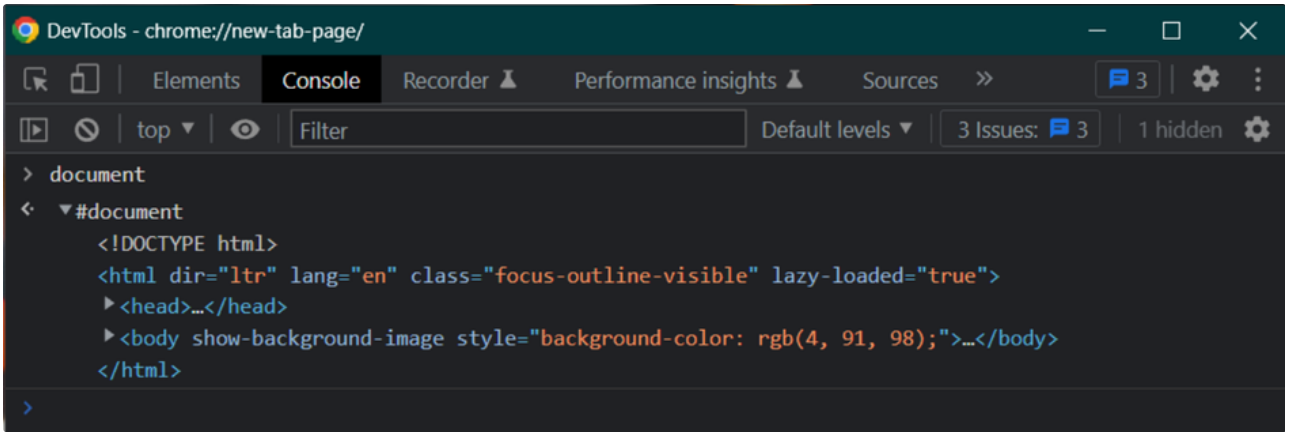
1.Open the console on this page

2. Type out the word documenta.

   a) Careful not to declare it (const document)

   b) Careful not to wrap it in quotes ("document")

3. Press enter



The DOM Recap

The DOM stands for "Document Object Model" and is a tree-like structure that is a representation of the HTML document, the relationship between elements, and contains the content and properties of the elements.

The DOM is not:

- Part of the JavaScript language

The DOM is:

- Constructed from the browser
- Is globally accessible by JavaScript code using the document object

Select Page Element By ID

Select An Element By ID

Let's take a look at how we can use JavaScript and the DOM to gain access to specific elements using their ID attribute.

Remember the `document` object from the previous section? Well, we're going to start using it! Remember the `document` object is an object, just like a JavaScript object. This means it has key/value pairs. Some of the values are just pieces of data, while others are functions (also known as **methods**!) that provide some type of functionality. The first DOM method that we'll be looking at is the `.getElementById()` method:

```
1 document.getElementById();
2
```

If we ran the code above in the console, we wouldn't get anything, because we did not tell it the ID of any element to get! We need to pass a string to `.getElementById()` of the ID of the element that we want it to find and subsequently return to us:

```
1 document.getElementById('footer');
2
```

One thing to notice right off the bat, is that we're passing `'footer'`, not `'#footer'`. It already knows that it's searching for an ID (its name is "getElementById", for a reason!).

If you'd like to read more about this method, check out its documentation page on MDN: Document: getElementById() method - Web APIs | MDN

Let's use this MDN documentation page to try out using this method.

Select Page Elements By Class Or Tag

Selecting Multiple Elements At Once

As I'm sure you remember from learning both HTML structure and CSS styling, an ID should be *unique* - meaning two or more elements should never have the same ID. Since IDs are unique, and since there will be only one element in the HTML with that ID, `document.getElementById()` will only ever return at most one element. So how would we select multiple DOM elements?

The next two DOM methods that we'll be looking at that both return multiple elements are:

- `.getElementsByClassName()`
- `.getElementsByTagName()`

Accessing Elements By Their Class

The first method we'll look at is `.getElementsByClassName()`:

```
1 document.getElementsByClassName();
2
```

Similarly to `.getElementById()`, if we ran the code above in the console, we wouldn't get anything, because we did not tell it the class to search for! Also just like `.getElementById()`, `.getElementsByClassName()` is expecting that we call it with a string of the class we want it to search for/return:

```
1 document.getElementsByClassName('brand-color');
2
```

If you'd like to read more about this method, check out its documentation page on MDN: Document: getElementsByClassName() method - Web APIs | MDN

Let's use this MDN documentation page to try out using this method.

More Ways To Access Elements

We've been looking at the:

- `.getElementById()`
- `.getElementsByClassName()`
- and `.getElementsByTagName()`

Now these DOM methods are standardized. However, not all browsers support every standard. They do *now*, for these three methods, but there are hundreds of other methods with varying levels of support.

That's why almost every method on MDN has a Browser compatibility table that lists when each browser started supporting that specific method.

The Browser compatibility table for the `.getElementsByClassName()` method.

Thankfully, all browsers have pretty much aligned to support the official standard.

However, back in the day, that wasn't the case. You had to write different code to perform the same action in different browsers. Then you had to write code to check which browser you were in to run the correct code for that browser. Let me tell you, it was a bit of a nightmare.

Several JavaScript libraries came along to help mitigate these issues. Let's take a brief look at the jQuery library.

The querySelector Method

We can use the `.querySelector()` method to select elements just like we do with CSS. We use the `.querySelector()` method and pass it a string that's just like a CSS selector:

```
// find and return the element with an ID of "header"
document.querySelector('#header');

// find and return the first element with the class "header"
document.querySelector('.header');

// find and return the first <header> element
document.querySelector('header');

```

Check out the `.querySelector()` method on MDN: Document: querySelector() method - Web APIs | MDN

The querySelectorAll Method

The `.querySelector()` method returns only *one* element from the DOM (if it exists). However, there are definitely times when you will want to get a list of all elements with a certain class or all of one type of element (e.g. all `<tr>` tags). We can use the `.querySelectorAll()` method to do this!

```
// find and return a list of elements with the class "header"
document.querySelectorAll('.header');

// find and return a list of <header> elements
document.querySelectorAll('header');

```

Here's the `.querySelectorAll()` method on MDN: Document: querySelectorAll() method - Web APIs | MDN

Recap

In this section, we took a brief look the history of browser support for standard DOM methods, the rise of the jQuery library, and how jQuery's success brought about new DOM methods. The new DOM methods we looked at are

- `.querySelector()` - returns a single element
- `.querySelectorAll()` - returns a list of elements

```
// find and return the element with an ID of "header"
document.querySelector('#header');

// find and return a list of elements with the class "header"
document.querySelectorAll('.header');

```

We also took a brief look that the list returned by `.querySelectorAll()` is a NodeList. We saw that it is possible to loop over a NodeList with either its `.forEach()` method, or the humble `for` loop:

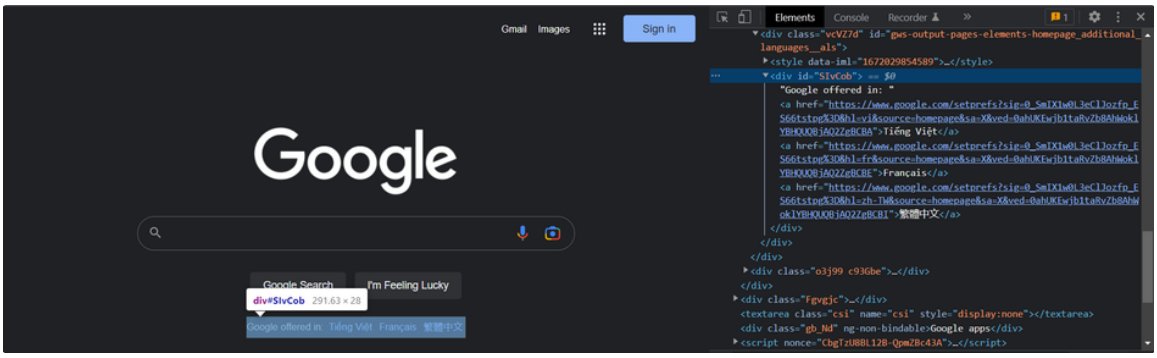
```
const allHeaders = document.querySelectorAll('header');

for(let i = 0; i < allHeaders.length; i++){
    console.dir(allHeaders[i]);
}

```

Further Research

- jQuery website
- .querySelector() method on MDN
- .querySelectorAll() method on MDN
- NodeList on MDN

Doc: Creating Content with JS

As I've mention in M2L2 that Js can change the content of HTML with DOM. So today we will dive into creating content with JS. First of all, we have HTML Google search like this to working with:



Update Existing Page Content

1. InnerHTML in JS

InnerHTML: Using innerHTML allows you to see exactly what's in the HTML markup contained within a string, including elements like spacing, line breaks and formatting.

```
> const ele = document.getElementById('SIvCob');
  ele.innerHTML
< 'Google offered in:  <a href="https://www.google.com/setprefs?sig=0_SmIX1w
  0L3eClJozfp_ES66tstpg%3D&amp;hl…;source=homepage&amp;sa=X&amp;ved=0ahUKEwj
  b1taRvZb8AhWoklYBHQUQBjAQ2ZgBCBA">Tiếng Việt</a>    <a href="https://www.g
  oogle.com/setprefs?sig=0_SmIX1w0L3eClJozfp_ES66tstpg%3D&amp;hl…;source=hom
  epage&amp;sa=X&amp;ved=0ahUKEwjb1taRvZb8AhWoklYBHQUQBjAQ2ZgBCBE">Français
  </a>    <a href="https://www.google.com/setprefs?sig=0_SmIX1w0L3eClJozfp_E
  S66tstpg%3D&amp;hl…;source=homepage&amp;sa=X&amp;ved=0ahUKEwjb1taRvZb8AhWo
  klYBHQUQBjAQ2ZgBCBI">繁體中文</a>  '
>
```

InnerHTML returns the string inside our div and the HTML (or XML) markup contained within our string, including any spacing, line breaks and formatting irregularities.

You should use innerHTML when you want to see the HTML markup and what exactly is in your element — including any spacing, line breaks and formatting irregularities.

If the text inside the element includes the characters &, <, or >, innerHTML will return these characters as HTML entities "&", "<" and ">".

2. InnerText in JS

InnerText: This approximates the "rendered" text content of a node and is aware of styling and CSS. It's most effective when you only need to see what's in the element without the formatting.

```
> const ele = document.getElementById('SIvCob');
  ele.innerText
< 'Google offered in: Tiếng Việt Français 繁體中文'
```

InnerText returns the string inside our div. It approximates the “rendered” text content of a node and is aware of styling and CSS.

Think of it this way: If a user highlighted the contents of an element on their screen and copied it to their clipboard, what you get with innerText is exactly what it would return.

You should use innerText when you only need to see what’s in the element without the formatting.

When using innerText it retrieves and sets the content of the tag as plain text. Whereas when you use innerHTML, it retrieves and sets the same content in HTML format.

3. TextContent in JS

TextContent: This retrieves and sets the content of the tag as plain text. It’s most effective when you want to see what’s in an element, plus styling.

```
> const ele = document.getElementById('SIvCob');
  ele.textContent
< 'Google offered in:  Tiếng Việt    Français    繁體中文  '
```

TextContent returns the content of all elements in the node, including script and style elements. It’s aware of formatting like spacing and line breaks and will return those, as well.

You should use textContent you want to see what’s in the element, plus any styling.

While innerText is very similar to textContent, there are important differences between them. Put simply, innerText is aware of the rendered appearance of text while textContent is not.

Add new Content

1. Create Element HTML

In an HTML document, the document.createElement() method creates the HTML element specified by tagName , or an HTMLUnknownElement if tagName isn't recognized.

Research Further: Create Element

As you've already discovered, the .createElement() method is a method on the document object:

createElement.js

```
1  // creates and returns a <p> element
2  document.createElement('p');
3
4  // creates and returns an <h1> element
5  document.createElement('h1');
```

2. Adding Content To The Page

You may have noticed that while using document.createElement() to create an element, the element wasn't actually included to the page. An element can only be created. It is not included in the DOM. The element doesn't appear on the page since it isn't added to the DOM (if you remember, the DOM is the parsed representation of the page). We must thus be able to add new elements to the DOM in order for them to appear on the page now that we can build entirely new elements.

The .appendChild() method can be used to add an element to the page!

Now, to use the .appendChild() method, it needs to be called on another element, not the document object!

```
7
8   // create a brand new <p> element
9   const newPara = document.createElement('p');
10
11  // select the body of the page
12  const body = document.querySelector('body');
13
14  // add the <p> element as the last child element of the body
15  body.appendChild(newPara);
```

The .appendChild() method is called on one element, and is passed the element to append. The element that is about to be appended is added as the last child. So, in the example above, the <p> element will appear in the DOM as a child of the <body>...but it will appear at the end, after all text and any other elements that might be in the <body>.

Inserting HTML In Other Locations

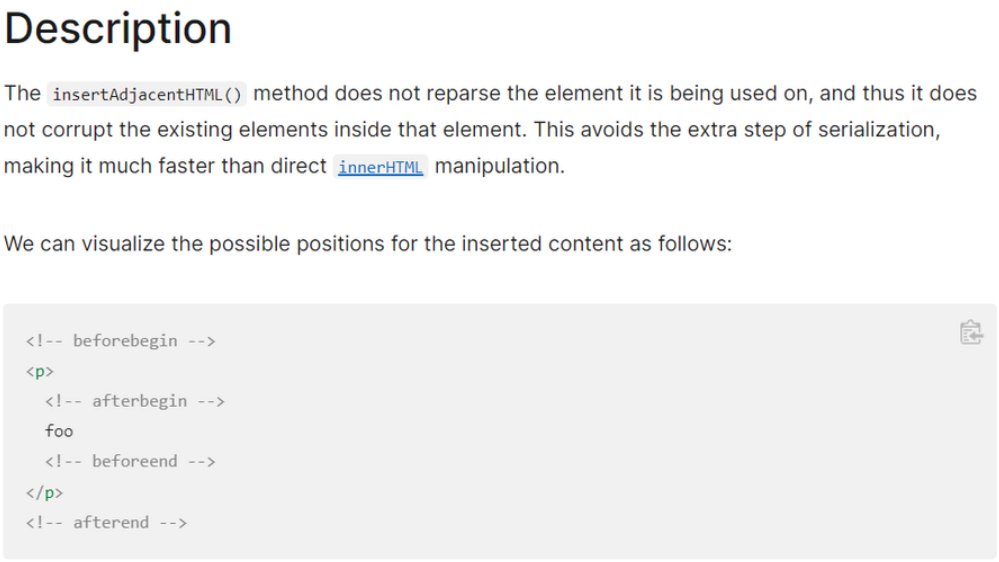
The .appendChild() method, by definition, adds an element as the parent element's last child. It cannot be the first child or any other position; it must be the last child. Wouldn't it be wonderful if we had some discretion about where to add the child element?

Enter the .insertAdjacentHTML() method! The .insertAdjacentHTML() method has to be called with two arguments:

- The location of the HTML.
- The HTML text that is going to be inserted.

The first argument to this method will let us insert the new HTML in one of four different locations

- beforebegin – inserts the HTML text as a previous sibling.
- afterbegin – inserts the HTML text as the first child.
- beforeend – inserts the HTML text as the last child.
- afterend – inserts the HTML text as a following sibling.



Description

The `insertAdjacentHTML()` method does not reparse the element it is being used on, and thus it does not corrupt the existing elements inside that element. This avoids the extra step of serialization, making it much faster than direct `innerHTML` manipulation.

We can visualize the possible positions for the inserted content as follows:

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  foo
  <!-- beforeend -->
</p>
<!-- afterend -->
```

Source: insertAdjacentHTML docs

Remove Page Content

In this short section, we learned two ways to remove an element from the page. You learn about:

- .removeChild()
- .remove()

The difference is that with .removeChild() must be called on the parent of the element being removed and must be passed the child to be removed, while .remove() can be called directly on the element to delete.

Imagine that we have a text in a p tag like this:

You want to remove the Hi and keep it for using later you need to use something like this:

```
1
2   let p = document.querySelector( 'p' )
3   let removed = p.removeChild( p.firstChild )
4   console.log( removed ) //<i>Hi</i>
5
```

But with remove method you really remove the child check this out:

```
8
9   let p = document.querySelector( 'p' )
10  let removed = p.childNodes[0].remove()
11  console.log( removed ) // undefined
12
```

In this section we will learn about the following helpful properties:

- .firstChild
- .firstElementChild
- .parentElement

The difference between .firstChild and .firstElementChild, is that .firstElementChild will always return the first element, while .firstChild might return whitespace (if there is any) to preserve the formatting of the underlying HTML source code.

For example:

```
> html
  ▼<div>
      " "
      <p>Hello World</p>
    </div>
> html.firstChild
    " "
> html.firstElementChild
    <p>Hello World</p>
>
```

Style Page Content

1. Modifying an Element's Style Attribute

Let's jump back to your knowledge of CSS. When trying to style an element, the most-specific rules that you can write for an element are written in that element's style attribute. Lucky for us, we can access an element's style attribute using the .style property!

```
JS stylePage.js > ...
1   const mainHeading = document.querySelector('h1');
2
3   mainHeading.style.color = 'red';
4
```

2. Adding Multiple Styles At Once

We've seen how the .style.<property> syntax will let us change just one piece of styling for an element. So if we wanted to set an element's color, background color, and font size, we'd have to use three separate lines of code:

```
JS stylePage.js > ...
1   const mainHeading = document.querySelector('h1');
2
3   mainHeading.style.color = 'blue';
4   mainHeading.style.backgroundColor = 'orange';
5   mainHeading.style.fontSize = '3.5em';
```

Fortunately, we can use the .style.cssText property to set multiple CSS styles at once!

```
6
7  const mainHeading = document.querySelector('h1');
8
9  mainHeading.style.cssText = 'color: blue; background-color: orange; font-size: 3.5em';
10
```

Notice that when using the .style.cssText property, you write the CSS styles just as you would in a stylesheet; so you write font-size rather than fontSize. This is different than using the individual .style.<property> way.

3. Accessing an Element's Classes

The first element property we'll look at is the .className property. This property returns a string of all of the element's classes. If an element has the following HTML:

```
12  <html>
13      <body >
14          <div id='hello' class="container image-card">Hello World!</div>
15          <div class="footer content">......</div>
16      </body>
17  </html>
```

We could use .className to access the list of classes:

```
19  const helloDiv = document.querySelector('#hello');
20
21  // store the list of classes in a variable
22  const listOfClasses = helloDiv.className;
23
24  // logs out the string "container image-card"
25  console.log(listOfClasses);
```

But the listOfClasses now just a String so how can I get a List for classes:

```
27  // store the list of classes in a variable
28  const listOfClasses = helloDiv.classList;
29
30  // logs out the list ["ank-student", "jpk-modal"]
31  console.log(listOfClasses);
```

We learned a ton of content in this section! We looked at:

- modifying individual styles with .style.<prop>
- updating multiple styles at once with .style.cssText
- getting/setting a list of classes with .className
- getting/setting/toggling CSS classes with .classList

My recommendation to you is that, out of the list of techniques you learned in this section, to use the .classList property more than any other. .classList is by far the most helpful property of the bunch, and it helps to keep your CSS styling out of your JavaScript code.

Update Existing Page Content

Add New Page Content

As you've already discovered, the `.createElement()` method is a method on the `document` object:

```
// creates and returns a <span> element
document.createElement('span');

// creates and returns an <h3> element
document.createElement('h3');

```

Here's the `.createElement()` documentation page on MDN: createElement docs

Adding Content To The Page

You may have noticed that using `document.createElement()` to create an element didn't actually add that newly created element anywhere on the page! Creating an element...just creates it. It doesn't add it to the DOM. Since the element isn't added to the DOM, it doesn't appear in the page (if you remember, the DOM is the parsed representation of the page). So, now that we can create brand new elements, we need to be able to add them to the DOM so that they'll show up on the page.

We can use the `.appendChild()` method to add an element to the page! Before we see how this element works, let's quickly define the word "append". There are several different definitions of the word, but I like the wording of the Cambridge Dictionary's the best:

> to add something to the end of a piece of writing

> definition of the word "append" (source)

Now, to use the `.appendChild()` method, it needs to be **called on another element**, not the `document` object!

```
// create a brand new <span> element
const newSpan = document.createElement('span');

// select the first (main) heading of the page
const mainHeading = document.querySelector('h1');

// add the <span> element as the last child element of the main heading
mainHeading.appendChild(newSpan);

```

I like the Cambridge Dictionary's version because it clearly states how the content is added *at the end*. The `.appendChild()` method is called on one element, and is passed the element to append. The element that is about to be appended is added as the last child. So, in the example above, the `<span>` element will appear in the DOM as a child of the `<h1>`...but it will appear *at the end*, after all text and any other elements that might be in the `<h1>`.

Here's the `.appendChild()` documentation page on MDN: [appendChild docs](Node: appendChild() method - Web APIs | MDN)

Creating Text Nodes

Just like you created new elements with the `.createElement()` method, you can also create new *text* nodes using the `.createTextNode()` method. Take a look at the following code that:

- creates a paragraph element
- creates a text node
- appends the text node to the paragraph
- appends the paragraph to the tag

```
const myPara = document.createElement('p');
const textOfParagraph = document.createTextNode('I am the text for the paragraph!');

myPara.appendChild(textOfParagraph);
document.body.appendChild(myPara);

```

However, since you already know about the `.textContent` property, the code below will provide the exact same result:

```
const myPara = document.createElement('p');

myPara.textContent = 'I am the text for the paragraph!';
document.body.appendChild(myPara);

```

Therefore, instead of creating a new text node and appending it to an element, it's faster and easier to just update the element's text with the `.textContent` property.

For more info, check out the documentation: createTextNode() docs

Inserting HTML In Other Locations

By definition, the `.appendChild()` method will add an element as the last child of the parent element. It's impossible to put it as the first child or anywhere else...it has to be the last child. Wouldn't it be nice if there were a little flexibility in where we could add the child element?

Enter the `.insertAdjacentHTML()` method! The `.insertAdjacentHTML()` method has to be called with two arguments:

- the location of the HTML
- the HTML text that is going to be inserted

The first argument to this method will let us insert the new HTML in one of four different locations

- `beforebegin` – inserts the HTML text as a previous sibling
- `afterbegin` – inserts the HTML text as the first child
- `beforeend` – inserts the HTML text as the last child
- `afterend` – inserts the HTML text as a following sibling

A visual example works best, and MDN's documentation has a fantastic example that I'll modify slightly:

```
<!-- beforebegin -->
<p>
    <!-- afterbegin -->
    Existing text/HTML content
    <!-- beforeend -->
</p>
<!-- afterend -->

```

Here's how we'd call `.insertAdjacentHTML()`:

```
const mainHeading = document.querySelector('#main-heading');
const htmlTextToAdd = '<h2>Skydiving is fun!</h2>';

mainHeading.insertAdjacentHTML('afterend', htmlTextToAdd);

```

Check out the documentation page for more information: insertAdjacentHTML docs

Add New Page Content Recap

In this section, we learned how to create new DOM elements and add them to the page. We looked at the following methods:

- `.createElement()` to create new elements
- `.appendChild()` to add a child element to a parent element as its last child
- `.createTextNode()` to create a text node
- `.insertAdjacentHTML()` to put HTML text anywhere around an element

Some important things to note are:

- if an element *already exists in the DOM* and this element is passed to `.appendChild()`, the `.appendChild()` method will *move it* rather than duplicating it
- an element's `.textContent` property is used more often than creating a text node with the `.createTextNode()` method
- the `.insertAdjacentHTML()` method's second argument has to be text, you can't pass an element

Further Research

- createElement on MDN
- createTextNode on MDN
- appendChild on MDN
- insertAdjacentHTML on MDN

Remove Page Content

What's in store!

In this quick section, you're going to learn how to remove content from the page. Specifically, we'll look at these new methods:

- `.removeChild()`
- `.remove()`

In the process, you'll also learn about these two properties:

- `.firstElementChild`
- `.parentElement`

Removing a Child Element

We can use the `.removeChild()` method to...wait for it...remove a child element. Basically, this is exactly the opposite of the `.appendChild()` method. So just like the `.appendChild()` method, the `.removeChild()` method requires:

- a parent element
- the child element that will be removed

```
<parent-element>.removeChild(<child-to-remove>);

```

Here's the `.removeChild()` documentation page on MDN: removeChild docs

A drawback (and workaround!) with the `.removeChild()` Method

Just like the `.appendChild()` method, there's a (somewhat minor) drawback to the `.removeChild()` method. Both methods:

- require access to parent to function

However, we don't actually need to have the parent element because there is a workaround! Just like the `.firstElementChild` property can be called on a parent element to access its first element, every element also has a `parentElement` property that refers to its parent! So if we have access to the child element that we're about to add or remove, you can use the `parentElement` property to "move focus" to the element's parent. Then we can call `.removeChild()` (or `.appendChild()`) on that referenced parent element.

Let's look at an example:

```
const mainHeading = document.querySelector('h1');

mainHeading.parentElement.removeChild(mainHeading);

```

Let's walk through this code.

```
const mainHeading = document.querySelector('h1');

```

The preceding code will select the first `<h1>` on the page and stores it in the `mainHeading` variable. Now to the next line:

```
mainHeading.parentElement.removeChild(mainHeading);

```

This starts with the `mainHeading` variable. It calls `.parentElement`, so the focus of the next code is directed at the parent element. Then `.removeChild()` is called on the parent element. Finally, `mainHeading` itself is passed as the element that needs to be removed from its parent.

So an element uses itself to remove itself from its parent. Pretty cool, huh?

Removing a Child Element (Part 2!)

We went through all of those steps selecting an element, using DOM traversal techniques like `.parentElement` and `.firstElementChild`, so that we can remove a child element. I showed you this way so that you can get some exposure and practice to moving around in the DOM.

Now, you might be glad (or frustrated! haha) to find out there's an easier way to do all this! We can remove the child element directly with the `.remove()` method:

```
const mainHeading = document.querySelector('h1');

mainHeading.remove();

```

Here's the `.remove()` documentation page on MDN: .remove() docs

Remove Page Content Recap

In this short section, we learned two ways to remove an element from the page. You learned about:

- `.removeChild()`
- `.remove()`

The difference is that with `.removeChild()` must be called on the parent of the element being removed and must be passed the child to be removed, while `.remove()` can be called directly on the element to delete.

We also learned about the following helpful properties:

- `.firstChild`
- `.firstElementChild`
- `.parentElement`

The difference between `.firstChild` and `.firstElementChild`, is that `.firstElementChild` will always return the first element, while `.firstChild` *might* return whitespace (if there is any) to preserve the formatting of the underlying HTML source code.

Further Research

- removeChild on MDN
- remove on MDN
- firstChild on MDN
- firstElementChild on MDN
- parentElement on MDN

Style Page Content

In this section, we'll be looking at controlling page and element styling using the following properties and methods:

- `.style.<prop>`
- `.cssText`
- `.setAttribute()`
- `.className`
- `.classList`

> *CSS Specificity*
>
> *To be successful in this section, it will help to have an understanding of how CSS Specificity works. According to the MDN, "specificity" is:*
>
> *the means by which browsers decide which CSS property values are the most relevant to an element and, therefore, will be applied.*
>
> *Basically, the closer the style rule is to an element, the more specific it is. For example, a rule in a style attribute on an element will override a style rule for that element in a CSS stylesheet. There is also the specificity of the type of selector being used. An _ID_ is more specific than a class.*
>
> *If you'd like to learn more about CSS Specificity, check out the following links:*
>
> *Specificity on MDN*

Modifying an Element's Style Attribute

Let's jump back to your knowledge of CSS. When trying to style an element, the most-specific rules that you can write for an element are written in that element's `style` attribute. Lucky for us, we can access an element's `style` attribute using the `.style` property!

```
const mainHeading = document.querySelector('h1');

mainHeading.style.color = 'red';

```

Now, I want to point out that when using the `.style` property, you can only modify *one* CSS style at a time. That's why the previous code has `.style.color = 'red'` and not just `.style = 'red'`.

Check out the documentation page for more information: style docs

Adding Multiple Styles At Once

We've seen how the `.style.<property>` syntax will let us change *just one* piece of styling for an element. So if we wanted to set an element's color, background color, and font size, we'd have to use three separate lines of code:

```
const mainHeading = document.querySelector('h1');

mainHeading.style.color = 'blue';
mainHeading.style.backgroundColor = 'orange';
mainHeading.style.fontSize = '3.5em';

```

...and that's just for setting *three* styles. Imagine if we needed 15 or 20 different styles! So the `.style.property` syntax is perfect for setting one style at a time, but it's not great for controlling multiple styles.

Fortunately, we can use the `.style.cssText` property to set multiple CSS styles at once!

```
const mainHeading = document.querySelector('h1');

mainHeading.style.cssText = 'color: blue; background-color: orange; font-size: 3.5em';

```

Notice that when using the `.style.cssText` property, you write the CSS styles just as you would in a stylesheet; so you write `font-size` rather than `fontSize`. This is different than using the individual `.style.<property>` way.

Setting An Element's Attributes

Another way to set styles for an element is to bypass the `.style.<property>` and `.style.cssText` properties altogether and use the `.setAttribute()` method:

```
const mainHeading = document.querySelector('h1');

mainHeading.setAttribute('style', 'color: blue; background-color: orange; font-size: 3.5em;');

```

Check out the documentation page for more information: style docs

`.setAttribute()` Is Not Just For Styling

The `setAttribute()` method is not *just* for styling page elements. You can use this method to set *any* attribute for an element. If you want to give an element an ID, you can do that!:

```
const mainHeading = document.querySelector('h1');

// add an ID to the heading's sibling element
mainHeading.nextElementSibling.setAttribute('id', 'heading-sibling');

// use the newly added ID to access that element
document.querySelector('#heading-sibling').style.backgroundColor = 'red';

```

The last two lines could've been combined into one to bypass setting an ID and just styling the element directly:

```
mainHeading.nextElementSibling.style.backgroundColor = 'red';

```

...but this was just to demonstrate that it's possible to set an attribute with JavaScript that affects the DOM which then can be used immediately

Accessing an Element's Classes

The first element property we'll look at is the `.className` property. This property returns a string of all of the element's classes. If an element has the following HTML:

```
<h1 id="main-heading" class="ank-student jpk-modal">Learn Web Development at Udacity</h1>
```

```
```

We could use `.className` to access the list of classes:

```
const mainHeading = document.querySelector('#main-heading');

// store the list of classes in a variable
const listOfClasses = mainHeading.className;

// logs out the string "ank-student jpk-modal"
console.log(listOfClasses);

```

The `.className` property returns a space-separated string of the classes. This isn't the most ideal format, unfortunately. We can, however, convert this space-separated string into an array using the JavaScript string method, `.split()`:

```
const arrayOfClasses = listOfClasses.split(' ');

// logs out the array of strings ["ank-student", "jpk-modal"]
console.log(arrayOfClasses);

```

Now that we have an *array* of classes, we can do any data-intensive calculations:

- use a `for` loop to loop through the list of class names
- use `.push()` to add an item to the list
- use `.pop()` to remove an item from the list

`.className` is a property, so we can set its value just by assigning a string to the property:

```
mainHeading.className = "im-the-new-class";

```

The above code *erases* any classes that were originally in the element's `class` attribute and replaces it with the single class `im-the-new-class`.

Since `.className` returns a string, it makes it hard to add or remove individual classes. As I mentioned earlier, we can convert the string to an array and then use different Array Methods to search for a class remove it from the list, and then update the `.className` with the remaining classes. However, we don't want to do all of that work! Let's use the newer `.classList` property.

The `.classList` Property

The `.classList` property is newer than the `.className` property, but is much nicer, check it out:

```
<h1 id="main-heading" class="ank-student jpk-modal">Learn Web Development at Udacity</h1>

```

```
const mainHeading = document.querySelector('#main-heading');

// store the list of classes in a variable
const listOfClasses = mainHeading.classList;

// logs out ["ank-student", "jpk-modal"]
console.log(listOfClasses);

```

Check out the documentation page on MDN: classList docs

The `.classList` property has a number of properties of its own. Some of the most popularly used ones are:

- `.add()` - to add a class to the list
- `.remove()` - to remove a class from the list
- `.toggle()` - to add the class if it doesn't exists or remove it from the list if it does already exist
- `.contains()` - returns a boolean based on if the class exists in the list or not

Style Page Content Recap

We learned a ton of content in this section! We looked at:

- modifying individual styles with `.style.<prop>`
- updating multiple styles at once with `.style.cssText`
- getting/setting a list of classes with `.className`
- getting/setting/toggling CSS classes with `.classList`

My recommendation to you is that, out of the list of techniques you learned in this section, to use the `.classList` property more than any other. `.classList` is by far the most helpful property of the bunch, and it helps to keep your CSS styling out of your JavaScript code.

Further Research

- style on MDN
- Article: Using dynamic styling information
- DOM methods to control styling
- nextElementSibling on MDN
- className on MDN
- classList on MDN
- Specificity on MDN

Doc: Working with Browser Events

Intro to Browser Events

Lesson Overview

To recap, we'll be looking at :

- **Events** - what they are
- **Responding to an event** - how to listen for an event and respond when one happens
- **Event Data** - harness the data that is included with an event
- **Stopping an event** - preventing an event from triggering multiple responses
- **Event Lifecycle** - the lifecycle stages of an event
- **DOM Readiness** - events to know when the DOM itself is ready to be interacted with

This lesson is chock full of incredibly useful information, so don't skim over anything!

Seeing An Event

There is a hidden world of events going on right now on this very page! It's really hard to actually *see* into this hidden world, though. So how can we know that events really *are* being announced? If they are being announced, how come they're not easy for us to see?

Fortunately, the Chrome browser has a special `monitorEvents()` function that will let us see different events as they are occurring.

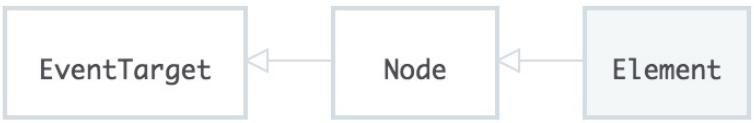
Check out the documentation on the Chrome DevTools site: monitorEvents documentation

Respond to Events

An Event Target

Do you remember the Node Interface and the Element interface from the first lesson? Do you remember how the Element Interface is a descendant of the Node Interface, and therefore inherits all of Node's properties and methods?

Well there was one piece that I totally skipped over then but am addressing now. The Node Interface inherits from the `EventTarget` Interface.



The EventTarget Interface is inherited by all nodes and elements.

The EventTarget page says that EventTarget:

> *is an interface implemented by objects that can receive events and may have listeners for them.*

and

> *Element, document, and window are the most common event targets, but other objects can be event targets too…*

As you can see from the image above, the EventTarget is at the top of the chain. This means that it does not inherit any properties or methods from any other interfaces. However, every other interface inherits from it and therefore contain its properties and methods. This means that each of the following is an "event target";

- the `document` object
- a paragraph element
- a video element
- etc.

Remember that both the ***document*** object and ***any DOM element*** can be an event target. And again, why is this?...because both the Element Interface and the Document Interface inherit from the EventTarget Interface. So any individual element inherits from the Element Interface, which in turn inherits from the EventTarget Interface. Likewise, the document object comes from the Document Interface, which in turn inherits from the EventTarget Interface.

If you take a look at the EventTarget Interface, you'll notice that it doesn't have *any* properties and only three methods! These methods are:

- `.addEventListener()`
- `.removeEventListener()`
- `.dispatchEvent()`

The one that we'll be looking at for the rest of this course will be the `.addEventListener()` method.

Adding An Event Listener

We've taken a brief look at this hidden world of events. Using the `.addEventListener()` method will let us *listen for* events and respond to them! I just said "*listen for* events". There are several ways to "phrase" this, so I want to give some examples:

- listen for an event
- listen to an event
- hook into an event

- respond to an event

...all of these mean the same thing and are interchangeable with one another.

Let's use some pseudo-code to explain how to set an event listener:

```
<event-target>.addEventListener(<event-to-listen-for>, <function-to-run-when-an-event-happens>);

```

So an event listener needs three things:

1. an event target - this is called the **target**
2. the type of event to listen for - this is called the **type**
3. a function to run when the event occurs - this is called the **listener**

The `<event-target>` (i.e. the *target*) goes right back to what we just looked at: everything on the web is an event target (e.g. the `document` object, a `<p>` element, etc.).

The `<event-to-listen-for>` (i.e. the *type*) is the event we want to respond to. It could be a click, a double click, the pressing of a key on the keyboard, the scrolling of the mouse wheel, the submitting of a form...the list goes on!

The `<function-to-run-when-an-event-happens>` (i.e. the *listener*) is a function to run when the event actually occurs.

Let's transform the pseudo-code to a *real* example of an event listener:

```
const mainHeading = document.querySelector('h1');

mainHeading.addEventListener('click', function () {
  console.log('The heading was clicked!');
});

```

Let's break down the snippet above:

- the target is the first `<h1>` element on the page
- the event type to listen for is a `"click"` event
- the listener is a function that logs `"The heading was clicked!"` to the console

Check out the documentation for more info: addEventListener docs

Add Event Listener to the Project

Running code in a browser's developer tools is fantastic for testing. But that event listener will only last until the page is refreshed. As with all *real* JavaScript code that we want to send to our users, our event listener code needs to be in a JavaScript file.

Let's try adding an Event Listener to our project's files!

index.html app.js

Working-with-Browser-Events > index.html > html > body

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Respond to Events</title>
</head>
<body>

    <h1>Respond to Events</h1>
    <p>Lorem ipsum dolor sit amet consectetur adipisicing elit. Quasi, harum

    <script src="./app.js"></script>
</body>
</html>
```

index.html app.js

Working-with-Browser-Events > app.js > document.addEventListener('click') callback

```
document.addEventListener('click', function(){
    const mainHeading = document.querySelector('h1');

    mainHeading.style.backgroundColor = 'red';
    mainHeading.style.color = 'white';
})
```

So far, we've only looked at the `"click"` event and a couple of other ones. When we used the `monitorEvents()` function in the previous section, we saw a number of different event types (e.g. `dblclick`, `scroll`, `resize`).

How do you know what events are even out there to listen for? The answer is easy - documentation! To see a full list of all of the possible events you can listen for, check out the Events documentation: list of events

Recap

In this section, you learned all about events, the EventTarget Interface, and how to add event listeners. We used the `.addEventListener()` method to attach listeners to:

- the `document`
- a Node
- an Element

...basically anything that inherits from the EventTarget Interface. We also saw that there are three main parts to an event listener:

1. an event target - the **target**
2. the type of event to listen for - the **type**
3. a function to run when the event occurs - the **listener**

Further Research

- addEventListener on MDN
- EventTarget Interface
- Introduction to events

Remove An Event Listener

We say that we can use an event target's `.addEventListener()` method to start listening for specific events and respond to them. Let's say you only want to listen for just the first click event, respond to it, and ignore all other click events. The `.addEventListener()` event will listen for and respond to *all* click events.

(The newest version of the `.addEventListener()` specification *does* allow for an object to be passed as a third parameter. This object can be used to configure how the `.addEventListener()` method behaves. Of note, there is an option to listen for only a single event. However, this configuration object is not widely supported just yet).

To remove an event listener, we use the `.removeEventListener()` method. It sounds straightforward enough, right? However, before we look at `.removeEventListener()`, we need to take a brief review of object equality. It seems like an odd jump, but it'll make sense in just a moment.

Are Objects Equal in JavaScript

Equality is a common task in most programming languages, but in JavaScript, it can be a little bit tricky because JavaScript does this thing called type coercion where it will try to convert the items being compared into the same type. (e.g. string, number,). JavaScript has the double equality (`==`) operator that *will allow type coercion*. It also has the triple equality (`===`) symbol that will prevent type coercion when comparing.

Hopefully, this is all review. But let's talk about *just* object equality, which includes objects, arrays, and functions. Try giving this quiz a shot:

Quiz question: Will the following equality test result in `true` or `false` ?

```
{ name: 'Richard' } === { name: 'Richard' }
```

Output: `false`

Quiz question:

Given this code:

```
var a = {
    myFunction: function quiz() { console.log('hi'); }
};
var b = {
    myFunction: function quiz() { console.log('hi'); }
};
```

Does the following code evaluate to `true` or `false` ?

```
a.myFunction === b.myFunction
```

Output: `false`

Quiz question:

Given this code:

```
function quiz() { ... }
var a = {
    myFunction: quiz
};
var b = {
    myFunction: quiz
};
```

Does the following code evaluate to `true` or `false` ?

```
a.myFunction === b.myFunction
```

Output: `true`

Ok, so why do we care about any of this object/function equality? The reason is that the `.removeEventListener()` method requires you to pass *the same exact listener function* to it as the one you passed to `.addEventListener()`.

Let's see some pseudo-code for the `.removeEventListener()`:

```
<event-target>.removeEventListener(<event-to-listen-for>, <function-to-remove>);

```

So an event listener needs three things:

1. an event target - this is called the **target**
2. the type of event to listen for - this is called the **type**
3. the function to remove - this is called the **listener**

Remember, the *listener* function must be the *exact* same function as the one used in the `.addEventListener()` call...not just an identical looking function. Let's look at a couple of examples.

This code will successfully add and then remove an event listener:

```
function myEventListeningFunction() {
    console.log('howdy');
}

// adds a listener for clicks, to run the `myEventListeningFunction` function
document.addEventListener('click', myEventListeningFunction);

// immediately removes the click listener that should run the `myEventListeningFunction` function
document.removeEventListener('click', myEventListeningFunction);

```

Now, why does this work? It works because both `.addEventListener()` and `.removeEventListener`:

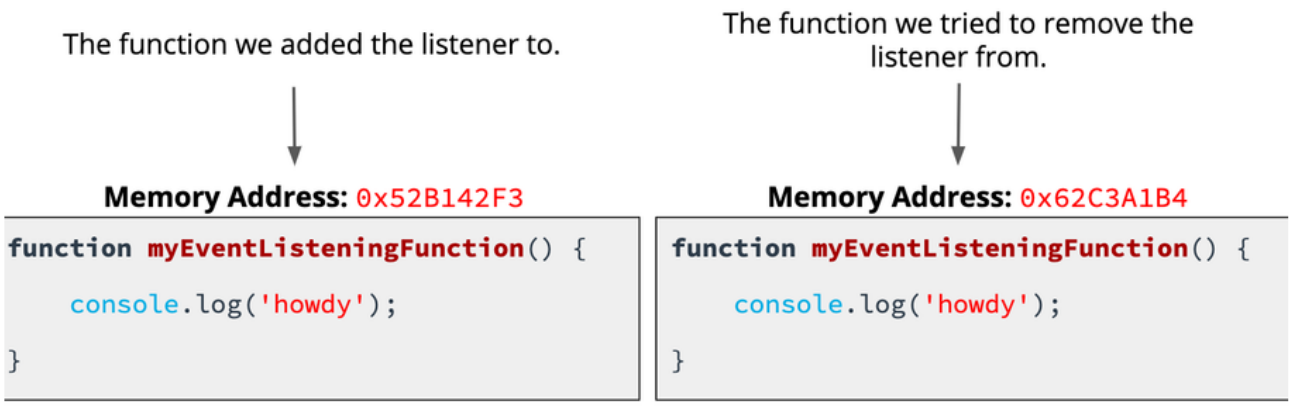
- have the same *target*
- have the same *type*
- and pass *the exact same listener*

Now let's look at an example that would *not* work (it does *not* remove the event listener):

```
// adds a listener for clicks, to run the `myEventListeningFunction` function
document.addEventListener('click', function myEventListeningFunction() {
    console.log('howdy');
});

// immediately removes the click listener that should run the `myEventListeningFunction` function
document.removeEventListener('click', function myEventListeningFunction() {
    console.log('howdy');
});

```

This code does *not* successfully remove the event listener. Again, why does this *not* work?

- both `.addEventListener()` and `.removeEventListener` have the same *target*
- both `.addEventListener()` and `.removeEventListener` have the same *type*
- `.addEventListener()` and `.removeEventListener` have their own distinct *listener* functions...they do not refer to the exact same function (**this is the reason the event listener removal fails!**)



Two functions can look the same, but live in two different places in memory. Looks can be deceiving!

When we wrote

```
function myEventListeningFunction() {
    console.log('howdy');
}

```

a second time, we actually created a completely new function that was stored in a completely new location in memory! They may look the same and do the same thing, but that doesn't make them the same. Imagine a scenario where you see two houses that look exactly the same. While the houses might look the same, their addresses are different! That's exactly what just happened in our previous example.

Recap

In this section, you learned about how to remove event listeners. You took a dive into object equality and how that plays a huge part in removing an event. Lastly, we also looked at how you can find out what event listener a DOM element has by using the DevTools.

Further Research

- removeEventListener on MDN
- Easily jump to event listeners
- Equality comparisons and sameness
- Article: Object Equality in JavaScript
- EventTarget Interface

(Doc) Week 3:

Node.js Overview

Node.js (Node) is an open source, cross-platform runtime environment for executing JavaScript code. Node is used extensively for server-side programming, making it possible for developers to use JavaScript for client-side and server-side code without needing to learn an additional language.

The first step to using Node.js is to install it on your computer. You can do so by following the link below and installing the Windows or Mac version, depending on what type of operating system you are using.

Node.js Download Page

If you've installed Node previously and need to update to a newer version, check out this article.

Using Node

One of the most useful features of Node is that it comes pre-installed with a standard package manager called **NPM**. NPM started as an easier way to download and manage dependencies of Node.js packages, but now it is also used as a tool in front-end JavaScript as well.

A package in Node.js contains all the files you need for a module. Modules are JavaScript libraries you can include in your project. There are hundreds of thousands of Node.js packages and NPM gives you easy access to all of them! For the purposes of this course we will make heavy use of the Node package called **Express**, as well as packages called **CORS** which allows for communication across the web, and Body-Parser (which is considered in the category of Middleware) which will allow us to parse the data we eventually will be passing through routes on our server.

Here is the code to install packages with NPM from the command line: `npm install package-name`

So to install the package called 'body-parser': `npm install body-parser`

And to install Cors: `npm install cors`

And then in a file named `Server.js` the installed package is included and made available in the code with: `const bodyParser = require('body-parser')`

> *You need to create the file* `Server.js` in Visual Studio Code.

Node invokes that `require()` function with a local file path as the function's only argument.

More on Node

- For more information about Node.js, you can read their website and documentation, or follow @node.js on Twitter.
- You can learn more about NPM from their website.

Express Overview

To install express, we use the `npm install express` command in the terminal. Using Express we set up an instance of our web app like this:

```
// Express to run server and routes
const express = require('express');

// Start up an instance of app
const app = express();
```

First, we include `express` in our project, and then we instantiate an instance of the app we are going to build in a file called `server.js`. Once we have created an instance of our app using Express, we can connect the other packages we have installed on the command line to our app in our code with the `.use()` method. Express version 4 and above require an extra middle-ware layer to handle a POST request (You will learn about POST requests in later lessons). This middle-ware is called as `bodyParser`. This used to be an internal part of the Express framework, but now you have to install it separately. Below is an example of how the body-parser and cors packages discussed earlier could be connected to the app instance.

```
/* Dependencies */
const bodyParser = require('body-parser')
/* Middleware*/
//Here we are configuring express to use body-parser as middle-ware.
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
// Cors for cross origin allowance
const cors = require('cors');
app.use(cors());
```

So far we have just seen how Express can be used to create an instance of a web app, and to include other Node packages in that web app, but the real fun of Express is unleashed with Routes. Before we can work with routes however, there are a couple other topics we need to cover. Next we'll learn what a server is, and how to create one locally to develop web projects on our own machines.

More on Express

You can learn more about Node and Express by reading the Express/Node introduction on the MDN web docs page.

What about `cors`, `urlencoded` and `json`?

While we won't cover `cors`, `urlencoded` and `json` in-depth here, if you want to read about more on each, see the following links:

- Cross-origin resource sharing (CORS) and related Express documentation for CORS
- URL Encoding and related Express documentation for `bodyParser`'s `urlencoded` functionality
- JavaScript Object Notation (JSON) and related Express documentation for `bodyParser`'s `json` functionality

Creating a Local Server

Another way you might see this same server code written is with an arrow function. Here is an example of the same code using an arrow function:

```
const server = app.listen(port, ()=>{console.log(`running on localhost: ${port}`)})

```

What's an arrow function?

Arrow functions are a shorter, more efficient way to write functions.

Here's another example of a regular function and then we'll write it as an arrow function:

Regular Function

`function addition(number){ return 4 + number }`

`addition(4);` `

Arrow Function

`const addition = number => 4 + number`

`addition(4);`

As you can see, there is less need for parenthesis and return statements, allowing the syntax to be much more compact. Both are acceptable ways of writing functions, and best practice would be for you to be consistent in how you write your code. Even if you choose not to write your code with arrow functions, you should still be aware of how they are written.

Curly brackets aren't required if only one expression is present, and parentheses are optional if there is only one parameter. So, our previous example could also be written as:

```
const addition = number => 4 + number

```

Steps to creating a local server

Set your variables

In this example, we set our variable to port 8000.

`const port = 8000;`

Utilize the `.listen()` method

Set your variable named server, and pass the listen method with two arguments port and listening.

`const server = app.listen(port, listening);`

The **port** argument refers to the port variable we set above. The *listening* argument refers to a callback function we create.

The listening function

This function will run when we execute the listen method to let you know that the server is running and on which port by logging messages to the console.

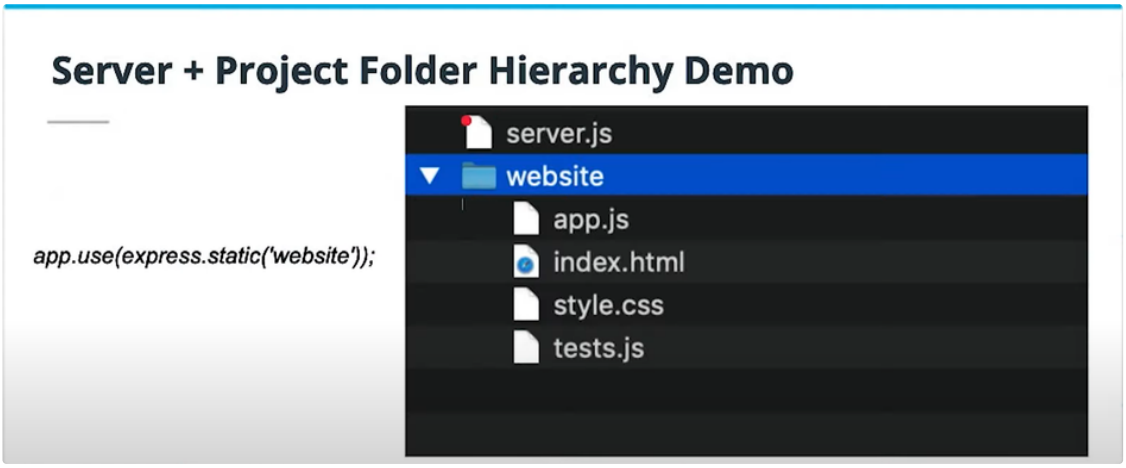
```
function listening(){
     console.log("server running");
     console.log(`running on localhost: ${port}`);
}
```

```
5
```

Supporting Material:

Arrow functions

Servers-File Structure Hierarchy



All of the files pictured above are included in the projects main folder, which is often referred to as the project 'root'. After making sure your project folders are setup in this way, the only thing you need to do is write one line of code in the `server.js` file that points the server code to the folder that has the `index.html` and any additional pages. In the example above the name of that folder is `website`, so in the `server.js` file the following line of code needs to be added:

```
app.use(express.static('website'));
```

This line of code connects our server-side code (the code in the `server.js` file) to our client-side code (the browser code written in the files housed in the `website` folder).

In this lesson, we covered how to create a server to run a web app locally in your browser, and to setup the ability to pass code between different parts of a web app through routes. We also covered the folder structure for web apps with an Express server. We will cover routes in the next lesson.

More on Express servers

For more information on Express you can visit their website, or see this basic example of an Express server.

Routes & GET Requests

In this example below, `app.get()` is used to make a GET request, the first parameter is the particular URL -- in this case our project home page, and a callback function to execute. Inside the callback function a response is sent using `.send()`, and in this case the response is a string that says 'hello world'. The real life execution of this code would mean that whenever the project home URL is visited in the browser, there will be a GET request made to the server, and the response will be shown in the browser, so the words 'hello world' would appear on the screen.

```
var express = require('express');
var app = express();

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function (req, res) {
  res.send('hello world');
})

```

Request and Response Parameters

The `req` parameter signifies the "request" from the client to the server. The `res` parameter signifies the "response" from the server to the client.

If you would like to read more about GET and the other HTTP request methods, you can check out the documentation here.

More Powerful GET Requests

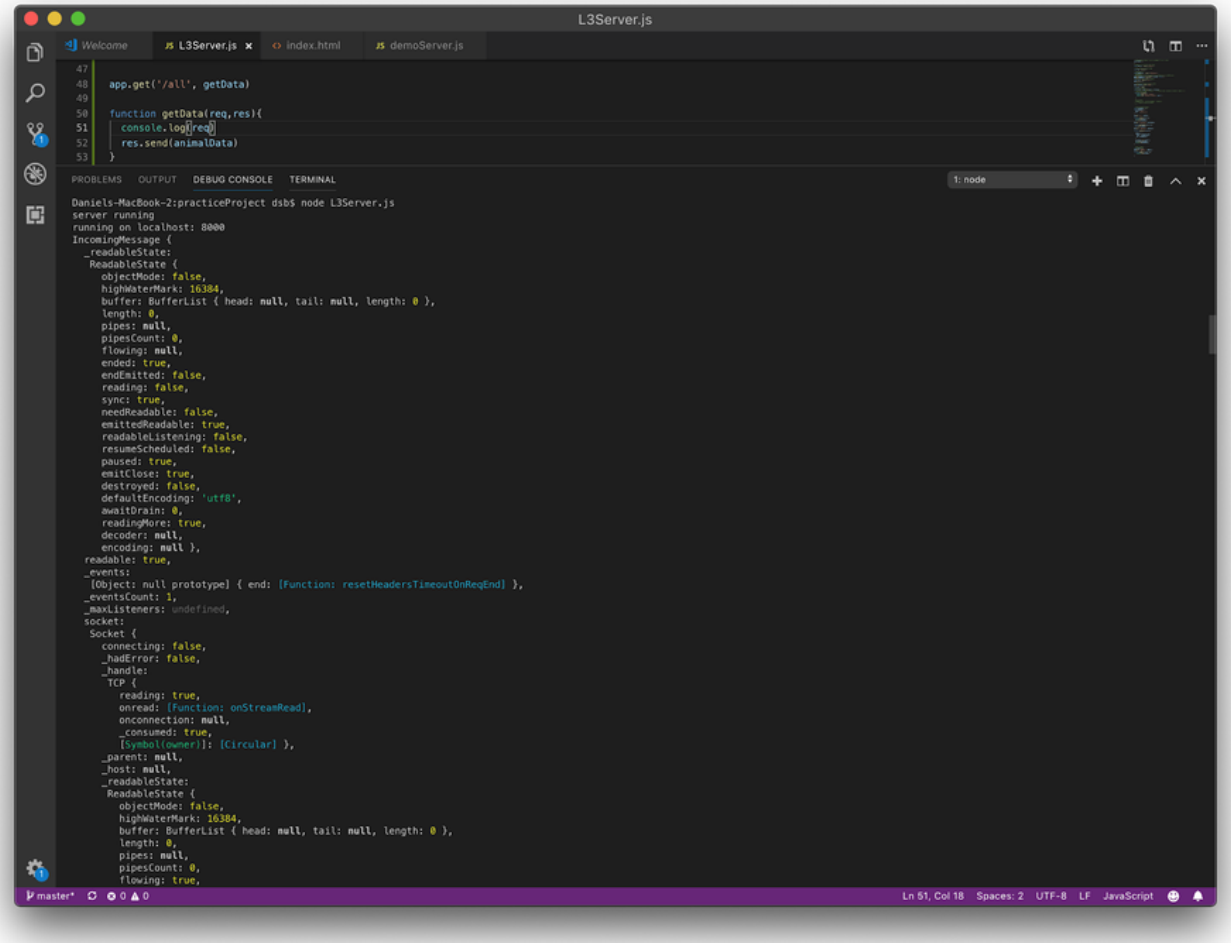
Hello world is all well and good, but suppose we wanted to make a GET request for some more useful data. GET requests can return all kinds of data, for example, imagine we wanted a JavaScript object to hold user data for us.

- At the top of the demo code we just looked at, we could create an empty JavaScript object with the code `const appData = {}`. The variable `appData` now acts as the endpoint for all our app data. Later we will learn how to POST data to the app endpoint, but first let's add the line of code that will return our JavaScript object when the GET request is made.

```
var express = require('express')
var app = express()
// Create JS object
const appData = {}
// Respond with JS object when a GET request is made to the homepage
app.get('/all', function (req, res) {
  res.send(appData)
})

```

In this example, we created a new route named '/all', so that the route 'localhost:3000/all' will now trigger the GET request, which will return the JavaScript object as laid out in the server code above.

- Notice, the callback function of the GET request takes two parameters, arbitrarily named `req` and `res` in this example. Every GET request produces a request, which is the data provided by the GET request, and a response, which is the data returned to the GET request. Below, you can see the long list of information that comes with each GET request:



A portion of the data returned by the `req` (request) parameter of a GET route.

More on Express routing and GET requests

In this lesson we learned how Express methods can be used to define routes and handle GET requests made to a server created with Node and Express. For more on Express routing methods and GET requests you can visit the 'Routing' guide in the Express documentation.

Routes & POST Requests

One way to collect and store user data so that you can access it later is through making an HTTP POST request. Analogous to the `.get()` Express method, there is also a `.post()` method to handle HTTP POST requests. An HTTP POST request sends data to the project's endpoint, where it is stored and can be accessed through a GET request, which we covered in the last lesson. Here is what a simple POST request could look like using the Express method `.post()`:

```
// POST method route
app.post('/', function (req, res) {
  res.send('POST received')
})
```

Here is how you could setup a basic POST route in the server side code.

First, Create an array to hold data:

```
const data = []

```

Then, create `post()` with a url path and a callback function:

```
app.post('/addMovie', addMovie )

```

In the callback function, add the data received from `request.body`. This is the key piece of information we are interested in from that long stretch of data we saw previously that the request (req) argument returns.

```
function addMovie (req, res){
   console.log(req.body)
   data.push(req.body)
}

```

In the next section we will cover how to execute this POST route with a request from the client side code.

Client Side & Server Side

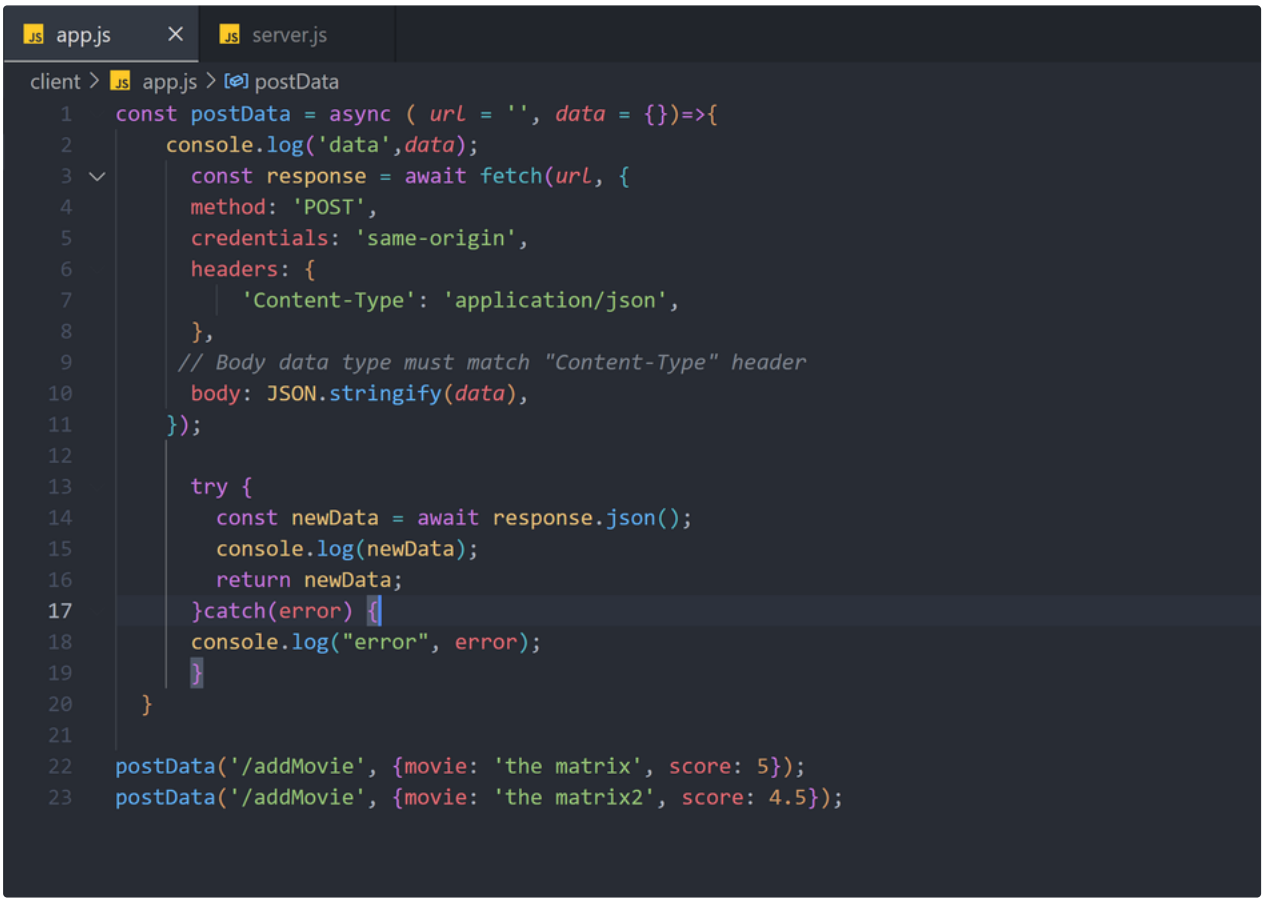
Server

- Set up in the beginning to handle anything outside what browsers do

Client

- Code that the browser executes
- The “finished product” that users see

Assuming we have set up a POST route in the file `server.js` file, we will move into the `website` folder and start writing client side code in a file named `app.js`. Here is the code we could use to make a POST request to our route:



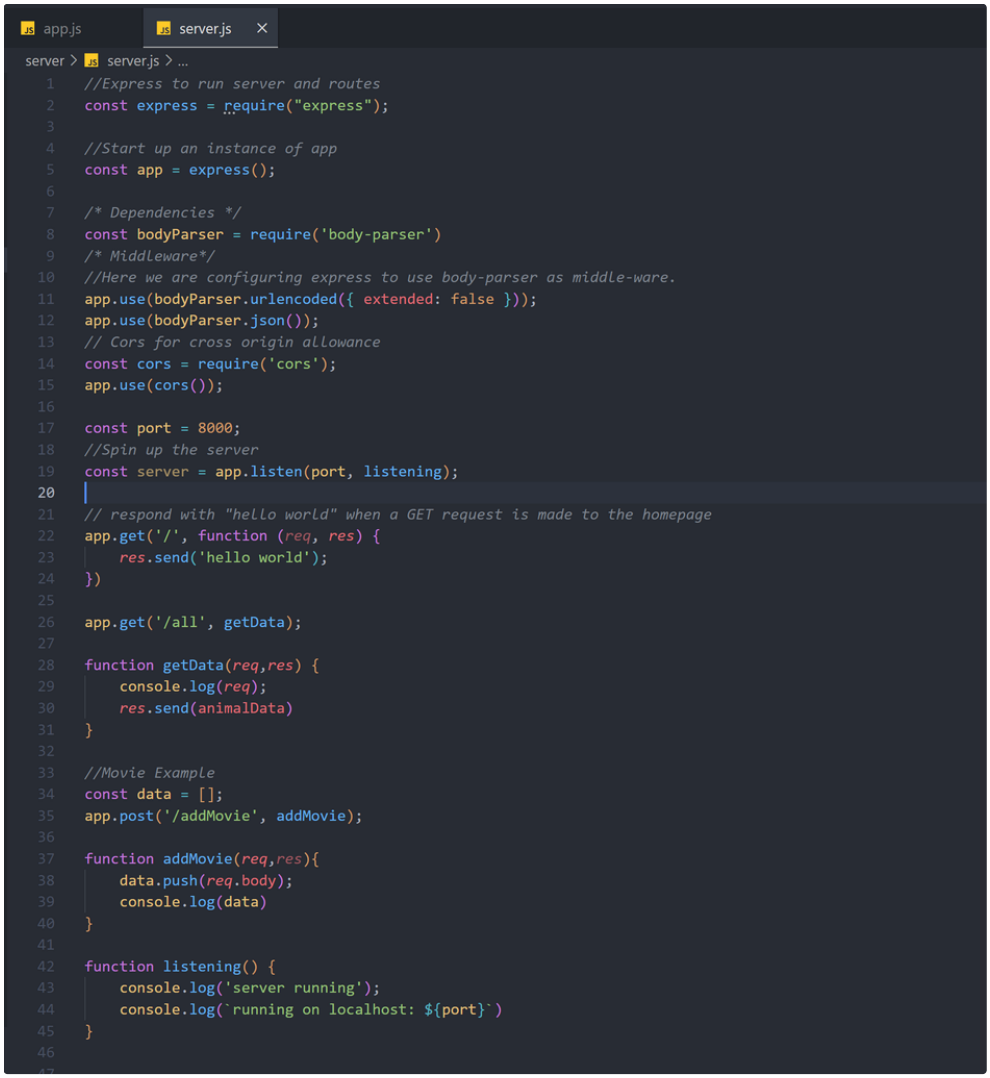
Let's focus in on the actual POST request, which is an object passed as the second parameter to `fetch()`. The First parameter is the URL we want to make the POST request to.

```
{
     method: 'POST',
     credentials: 'same-origin',
     headers: {
         'Content-Type': 'application/json',
     },
     body: JSON.stringify(data),
}

```

The credentials and headers are pretty boilerplate, but necessary for a successful POST request. The most important thing to notice is that `Content-Type` is set to json because we will be handling our data with JSON, for the most part.

Now we get to the juicy parts: the `method` is set to `POST` because we are accessing the POST route we setup in `server.js`. If we wanted to make a `GET` request from the client side, the `method` would be `GET`. The `body` of the request is the part we are most interested in because this is how we will access the data on the server side. When sending data to a web server, the data has to be a string. We can convert a JavaScript object into a string using the JavaScript method `JSON.stringify()`, which turns JavaScript objects and JSON data into a string for our server to receive the information. In this example, we are turning the JavaScript object passed in the `data` parameter into a string.

Here is the code of server.js file:



Async JS

JavaScript is a single threaded programming language, which means for the most part it will be run as a single process in your computer (essentially writing and running it top to bottom).

To be an effective web developer you have to be comfortable writing async code when the situation calls for it. For those times JavaScript does have a few async tricks up its sleeve. One of the most common is `setTimeout()` which allows you to break out of the inherent JS behavior of executing code line by line starting at the top.

Here is the async function code:

```
/*SYNC REVIEW*/

setTimeout(function(){ console.log('third') }, 3000);

function sync(){
console.log('first')
}

sync()
console.log('second')
```

Async Promises

While there have always been some async work arounds in JS, including setTimeout(), and AJAX, more recently a tool called Promises has been introduced natively to JavaScript, and Promises are now the accepted best practice for writing asynchronous functions in JavaScript.

You can think of Promises as a special function that either satisfy (resolve) or fail (reject) to execute a task, and then executes the corresponding actions, usually another task with the returned data in the case of 'resolved' and usually throw an error in the case of 'reject'.

Here is the basic anatomy of a Promise:

Syntax

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});

```

There are many methods to handle asynchronous work already, however Promises are the recommended option because they give you flexibility, intuitive syntax, and easy error handling. Promises are an amazing development in JavaScript, but until ES2017 (ES8) they still required extra boilerplate code, called generators, to run asynchronously. Now however, with the addition of native `async` functions to JavaScript, we can easily apply the async keywords to a Promise to execute asynchronous JavaScript code.

To make a `fetch()` call, or any other methods inside of a function, asynchronous we must use the keywords provided by JavaScript. Here is an example of an asynchronous fetch function using JavaScript keywords:

```
const postData = async ( url = '', data = {})=>{

      const response = await fetch(url, {
      method: 'POST', // *GET, POST, PUT, DELETE, etc.
      credentials: 'same-origin',
      headers: {
          'Content-Type': 'application/json',
      },
      body: JSON.stringify(data), // body data type must match "Content-Type" header
    });

      try {
        const newData = await response.json();
               return newData
      }catch(error) {
      console.log("error", error);
      // appropriately handle the error
      }
  }

  postData('/addMovie', {movie:' the matrix', score: 5})

```

`postData` is an async arrow function that is called with parameters on the last line of code. It is asynchronous because of the keyword `async` placed before its parameters.

```
const postData = async ( url = '', data = {})=>{
      const response = await fetch(url, {
      method: 'POST',
      credentials: 'same-origin',
      headers: {
          'Content-Type': 'application/json',
      },
      body: JSON.stringify(data),
    });

...
}

```

Once you mark a function as 'async' you have access to the keywords `await`, `try`, and `catch`.

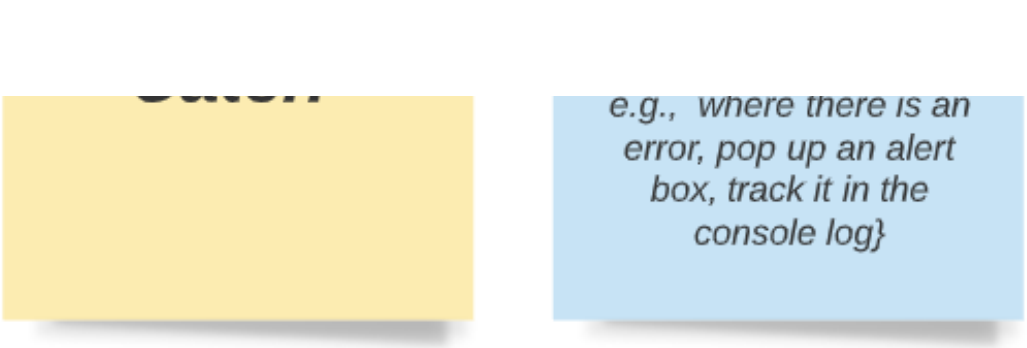
```
  try {
       const newData = await response.json();
       return newData
     } catch(error) {
     console.log("error", error);
     // appropriately handle the error
     }


```

The keywords `try` and `catch` mirror the Promise functionality of resolving or rejecting to execute a task. In this case, `if` and `else` are replaced with the keywords `try` and `catch`. The `await` keyword is used in places where the next action requires data from the current action, so we want to tell our program to wait until the data has been received before continuing with the next steps-- this is the magic of ASYNC JavaScript.

More on Async JS

For a more detailed overview on Promises and why they matter, read the article here.

Async	*Async is waiting for something to happen before it can execute the next line of code*
Await	*Wait until the API responds*
Try	*Once API responds, TRY to do {the thing you want to do with the information received from the API}*
Catch	*If there is an ERROR in the TRY block then CATCH it and process it {how to deal with error:*


e.g., where there is an error, pop up an alert box, track it in the console log}

Async Fetch

The **fetch()** method in JavaScript is used to request data from a server. The request can be of any type of API that returns the data in JSON or XML. The **fetch()** method requires one parameter, the URL to request, and returns a promise.

Syntax:

```
fetch('url')          //api for the get request
  .then(response => response.json())
  .then(data => console.log(data));

```

Parameters: This method requires one parameter and accepts two parameters:

- **URL:** It is the URL to which the request is to be made.
- **Options:** It is an array of properties. It is an **optional** parameter.

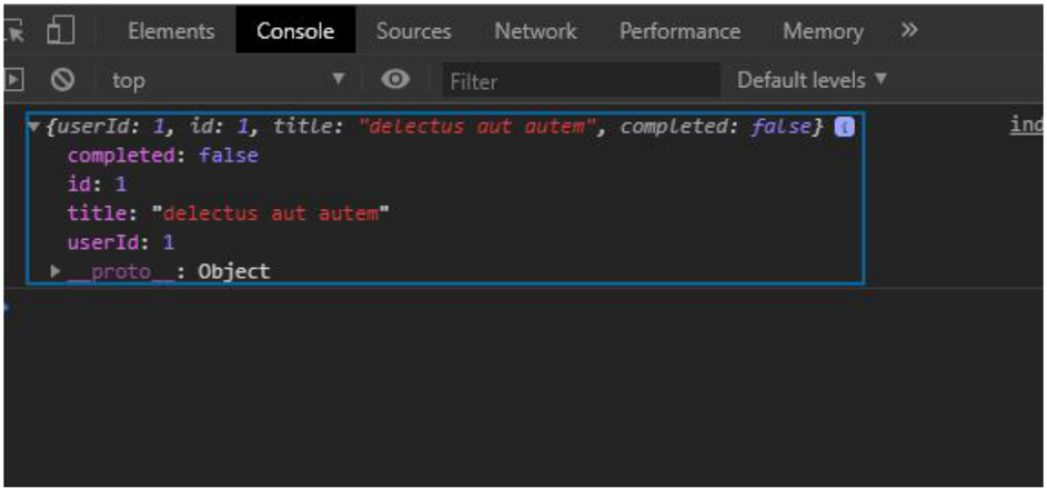
Return Value: It returns a promise whether it is resolved or not. The return data can be of the format JSON or XML. It can be an array of objects or simply a single object.

This example shows the use of the Javascript fetch() method.

NOTE: Without options, Fetch will always act as a get request.

```
    // API for get requests
    let fetchRes = fetch(
    "https://jsonplaceholder.typicode.com/todos/1");
    // fetchRes is the promise to resolve
    // it by using.then() method
    fetchRes.then(res =>
        res.json()).then(d => {
            console.log(d)
        })

```

Output:



Real-World Examples of Asynchronous JavaScript

Before asynchronous JavaScript with promises, if you wanted to query a database to retrieve information, such as a user's password and login name, you would need to write a long series of callback functions and if anything in your code failed, the entire process would stop. For example, if a program tried to access the database to get the user password, but it wasn't able to, instead of continuing to attempt to retrieve the login, the program would just stop without notice. With Asynchronous promises, the program could report an error for the password, and continue on retrieving the login.

In this example, moving the code from regular synchronous JavaScript to asynchronous JavaScript has 3 positive effects:

1. The asynchronous code will be much cleaner and shorter.
2. If there is an error in one part of the code, it will not block other portions of the code.
3. Debugging the code will be much easier because you will get much more specific and generally more helpful error messages from asynchronous code.

Here's another quick example to illustrate asynchronous JavaScript in the real world. Imagine you were making an application that converted international currency. The application would let a user enter a dollar amount in U.S. currency and select another country's currency to convert to. Without asynchronous JavaScript, programming this application could be a nightmare because there are so many dependent parts, such as the APIs you would need to get the conversion rates for every currency, the equations needed to transform the original dollar amount into the new currency, and the code to hold everything together. Asynchronous JavaScript can organize this code into discrete parts that can fail or succeed on their own without breaking the rest of the program. Additionally, asynchronous JavaScript lets you wait until one command is completed before executing the next.

In this example, that means your code can wait until it has received the information from the API about a specific currency, before trying to make the calculations of conversion. Without asynchronous JavaScript, a program would fail because it would take too long to retrieve the information from the API, and there would be no way to tell the program to wait until the previous action was finished before taking on the next command.

From this example, we can add a fourth feature of asynchronous Javascript which is that it allows you to complete one line of code, regardless of how long it takes, before moving on to the next line of code.

In Summary

Based on these examples, four features of asynchronous JavaScript in the real world are:

1. Clean and Concise Syntax
2. Better error handling
3. Easier Debugging
4. Ability to add timing to code

External Resources

1. Mozilla - Introducing Asynchronous Javascript
2. Introduction to Asynchronous JavaScript
3. Writing Asynchronous Tasks In Modern JavaScript

Introduction to Build Tools

You have learned to build a webpage with HTML and CSS, and to make it interactive with Javascript. You have all the tools you need to make a fully functional website - and that’s awesome! But you might be asking - where do I go from here? In this course we are going to cover some tools that will take your front end web development skills to a professional level. We will focus on a build tool called Webpack and let it take us on a tour through all the most common technologies and strategies you’ll see in a professional front end project.

This course is really about a mental shift. Up to this point we have built toy apps for the purpose of learning that typically only have one user - us. But how does our development need to change in order to build, say, a large company website with a whole team of devs? We’re going to explore that and a whole lot more in the coming modules.

Lesson Objectives

The prime objectives of this first lesson are:

1. To set up the context of why we need build-tools for front end development.
2. To understand how any build tool uses a configuration file where all the app development rules are defined.

Setting the Stage

In this section we are going to talk about why we need build tools for front end projects.

Go to the Job Ready Master website and inspect element to look at all of the assets on the Job Ready Master home page.

Pro tip: Don't just open inspector and go to the head tag element to see what's there. One of my favorite browser dev tools is the Network tab. Go there and you can look at all the requests made by the page, separate them by type, see the http request responsible, see the response, etc.. I could do an entire course just on this tab probably. You can learn a ton by taking a look at sites this way and it is a priceless debugging tool.

Build Tools

In this section we are going to talk about some general principles that all build tools have in common, and then introduce how Webpack fits into that role.

Build tools will manage all our assets so that we don't have to by combining them all into a single file (or sometimes a few files). We create a set of rules for the build tool to follow, telling it specifically how we want each type of asset handled, and then it follows our rules, takes all the assets and bundles them into a single large file, which has everything loading in the correct order and is much easier for us to deal with. Typically, files with names like bundle or main are the result of a build tool combining many assets into one.

What does it look like to write these rules for a build tool? Rules are written into config files. Just to give you a glimpse of where we're headed, here is an example webpack config file. Don't worry, this should look like gibberish right now, but we're just going to take a look at a few things.

```
1
2  module.exports = {
3    entry: './src/index.js',
4    output: {
5      path: path.join(__dirname, 'dist'),
6      filename: 'main.js',
7    },
8    module: {
9         rules: [
10           {
11                 test: /\.js$/,
12                 exclude: /node_modules/,
13                 loader: "babel-loader",
14           },
15           {
16                 test: /\.html$/,
17                 use: [{ loader: "html-loader"}],
18                 },
19                 {
20                     test: /\.scss$/,
21                     use: [ 'style-loader', 'css-loader', 'sass-loader' ]
22                 }
23        ]
24   },
25    plugins: [
26      new HtmlWebPackPlugin({
27             template: "./src/html/index.html",
28             filename: "./index.html",
29      })
30    ]
31  }
32
33
```

One thing to notice is that we're in javascript land! You can see that this config file is 100% normal javascript. Webpack is entirely built with js.

You can also see a whole section here in the middle titled "rules". Not surprisingly, this is where we declare the rules that will govern our different assets. You might also have noticed that each rule targets a certain type of file with regex.

Conclusion

To get a wider view of all the options in the build tool world, this is a good compare/contrast article to give you the lay of the land, though it was written a few years ago and not all of the details are correct for current versions, the technologies listed there representative of what is out there, though webpack would probably be at the top of that list if it had been written today.

Basics of Webpack

Introduction to Webpack

Lesson Objectives

The objectives of this lesson are to:

1. Understand the role and necessity of webpack as a build tool
2. Install webpack, and get started with a Node/Express web app
3. Understand to use the major components of webpack - entry point, output, loaders, and plugins
4. Explain the use to two different modes - production and development. We will also learn to use webpack-dev-server in development mode.

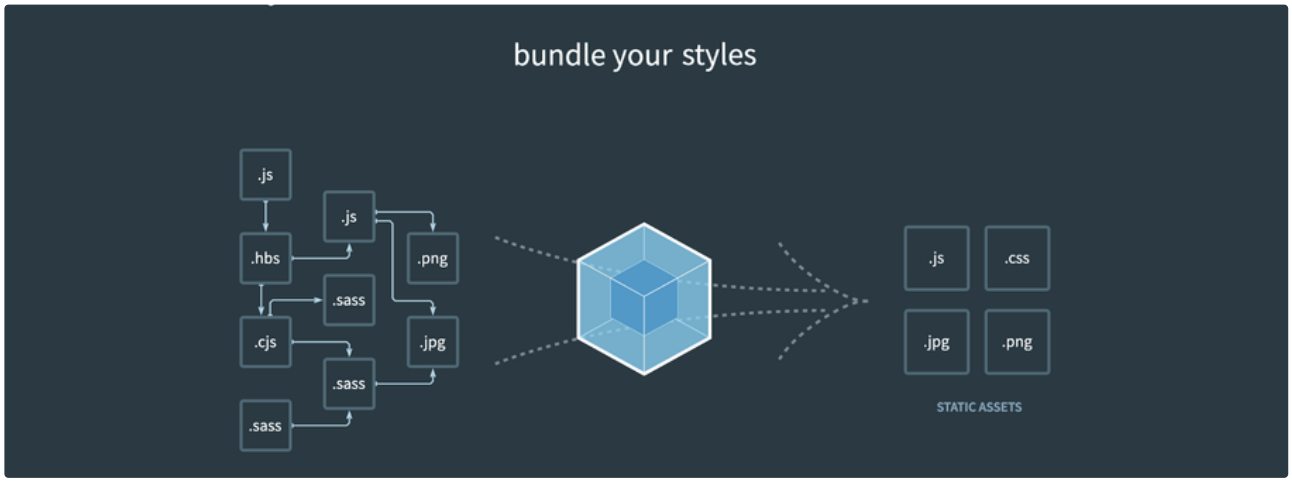
Getting Started with Webpack

The webpack documentation describes itself this way:

> At its core, **webpack** is a **static module bundler** for modern JavaScript applications.

But…what does that really mean?

This image from the Webpack website is a good visual.



Webpack bundles your assets, styles, images, and scripts. From webpack

The idea here is that on the left you have all the various asset file formats you will probably come across in a project. You might not recognize all the extensions, but just imagine that these are all your images, stylesheets, javascripts and more.

Webpack takes all the assets on the left and “bundles” or combines them into fewer files that are much easier to manage. Notice that multiple .js files on the left became one .js file on the right - that’s because the two files were combined into one large .js file.

Getting Started the Webpack

In this section, we're going to walk through the steps required to set up Webpack in your local machine.

This course is going to include a lot of work on your Git repository. We chose this style of practice because:

- We prefer learning by doing. By the end of the course, we want you to own a solid example to look back at for new projects.
- When working on your own repository, you have more freedom to try things and change things - which is an excellent way to learn! Just remember to stop and make a branch or at least commit before going off onto a rabbit trail.

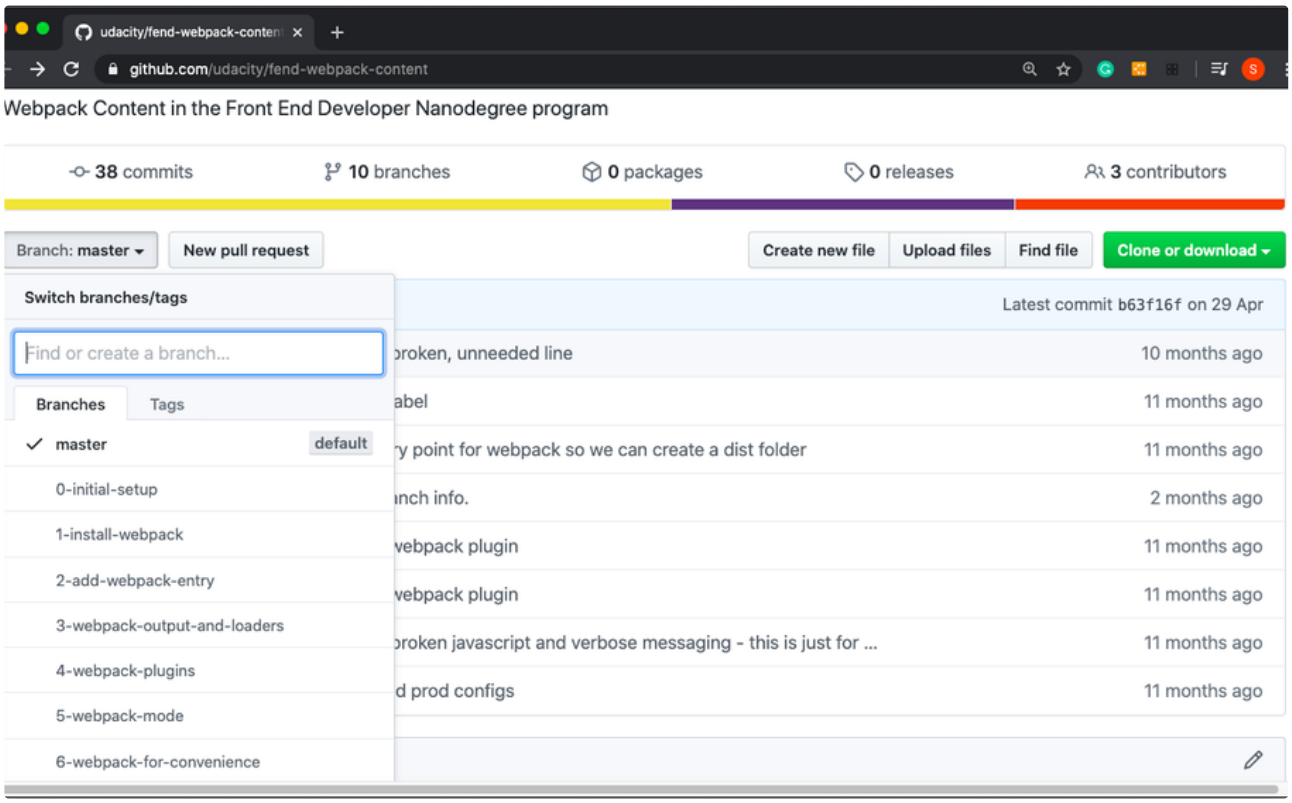
Choose a Mode of Practice

In order to get the starter code up and running on your machine, before we start with Webpack, you have to pick *either* of the two options here:

1. Fork the Webpack Express Example App - Github repository, and then clone it onto your local machine to avoid accidentally trying to push your updates to the Udacity master repository.
2. You can use the in-classroom workspaces, starting from the next page, spread throughout this lesson. In this option, that you don't need to deal with any environment setup.

About Github Repository - Webpack Express Example App

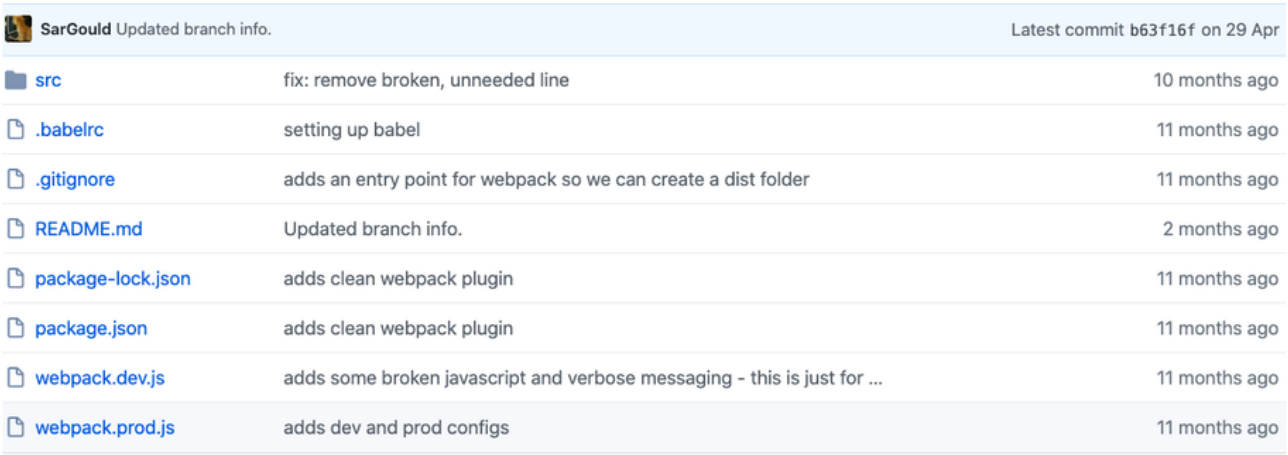
Each stage of the Webpack setup is reflected as a branch in the Github repository, and each branch has a Today I Learned for programmers ***file documenting the steps taken in the branch.*** Note the following ***six*** branches that correspond to different stages, as shown in the snapshot below:



This repository makes use of git branches for each stage of this lesson. You will want to use the stage before the exercise you are on, as a starting point. The first exercise is branch `0-initial-setup`, but you'll also want to use that as the basis for the second exercise. You'll use the next branch `1-install-webpack` as the basis for moving onto the third exercise.

If you need to check which branches are available, use `git branch -a`. Then, once you know which branch you want, use `git checkout <branch-name>`, e.g., `git checkout 0-initial-setup`.

Give a moment to have a glance at the different files available in the repository:

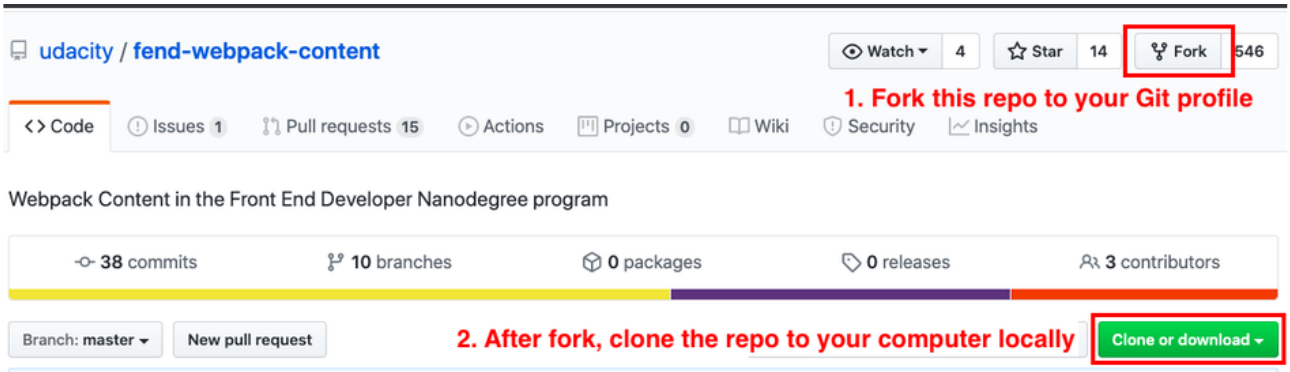


Once, we will start configuring the initial setup, you will notice the change in `package.json`, node_modules, and `package-lock.json`.

Steps for Initial Setup of the Starter Code - Locally in Your Machine

Follow the steps as explained below:

1. Fork this repo, then clone the branch of your choice from your forked repo down to your computer locally. When you fork the repo, you can add your own personal notes to the Today I Learned for programmers markdown file.



2. Open your terminal, go to a specific directory for this Nanodegree, and clone the Git repo using the command: `git clone https://github.com/<Your Github Username>/fend-webpack-content.git`

3. Get inside the newly downloaded directory, using `cd fend-webpack-content`

4. As we are just starting this process the first time, we will switch to the branch `0-initial-setup`, using:

```
git checkout 0-initial-setup
git branch

```

5. Install NPM in your project, giving the ability to use Node. NPM is by-default installed with Node.js. To see if you already have Node.js and NPM installed and check the installed version, run the following commands:

```
node -v
npm -v

```

You can upgrade to the latest version of npm using:

```
npm install -g npm@latest
npm install

```

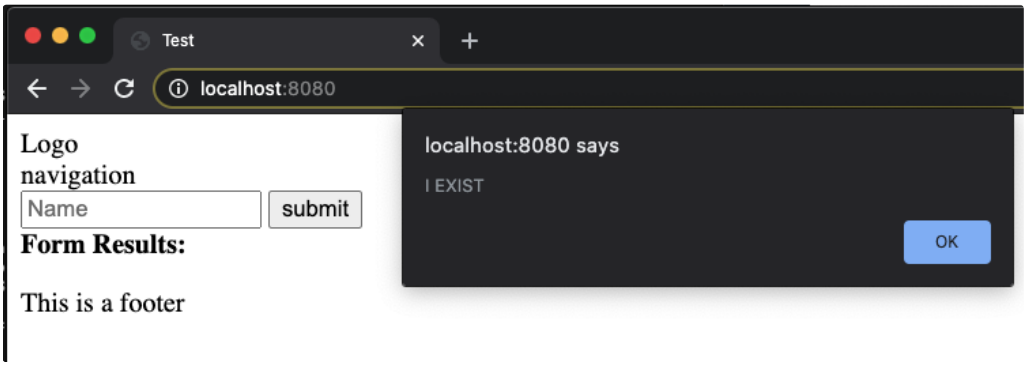
6. If the `npm install` throws `NPM error`, try clearing the cache and a fresh re-install, using:

```
npm cache clean
[sudo] npm install -g npm
npm install

```

If it asks to fix the vulnerabilities, run `npm audit fix`

7. Start your project, using `npm run start`

8. Check the website running at http://localhost:8080/. By default, this app runs on port 8080, but you can, of course, edit that in `server.js`. If everything, goes well, you'll see the app running as shown in the snapshot below:



Install Webpack

In this exercise we will install Webpack locally.

Current Version of Webpack

At the time this course is created, the most recent stable version of Webpack is version 4. Webpack 4 comes with some significant changes from previous versions, so you will find different information out there if you read an older tutorial.

One of the new things about version 4, is that webpack declared a few default settings that are automatically in place when you install webpack. So, you might not even need a webpack config file. But, our app won't be following all the default settings, so we will need a custom config file, `webpack.config.js`.

Install Webpack in Local Machine

Prerequisite - We suggest you have any code editor, such as Visual Studio Code in your local machine. It will help to edit the files, keep track of the changes, and easy navigation. In this exercise, we will install Webpack locally. The steps are as follows:

1. In the terminal (Mac/Linux) or command prompt (Windows), go to the project directory. If you wish to practice the steps mentioned in the video demonstration below, stay on the `0-initial-setup` branch. Else, switch to the branch `1-install-webpack` corresponding the current exercise where all the steps have already been carried out. **Note**: If you are switching branches don't forget to make a commit on the current branch before switching.

```
git checkout 1-install-webpack
git branch

```

2. Install Webpack and the command line instructions (CLI) tool using npm, as:

```
npm install

```

   Remember that `npm i` is just shorthand for `npm install`. A CLI is a terminal program that allows developers to run commands from the command line to communicate with the Webpack.

3. In `package.json`, verify if `webpack` and `webpack-cli` are added to the "dependencies" as:

```
"dependencies": {
"express": "^4.17.1",
"webpack": "^4.43.0",
"webpack-cli": "^3.3.11"
},

```

4. In `package.json`, add a build npm script as:

```
"scripts": {
"build": "webpack"
},

```

   Lastly, verify the dependency for the development in "devDependencies" block as:

```
"devDependencies":{
"webpack-dev-server": "^3.11.0",
},

```

5. Create a `webpack.config.js` file in the root location of your project, and add the necessary require statements, and a blank module.exports code as:

```
const path = require("path")
const webpack = require("webpack")
module.exports = {
}

```

   We will populate the module.exports with the entry code in the next exercise.

6. Try running Webpack using the command `npm run build`. **You can expect to get an error in the terminal because our config file is currently still empty.** We will start building the config file in the following lessons.
7. Let's commit the work done until now, using `git commit`.

Erratum - The `webpack.config.js` file has to be created in the root (/) location of your project.

Note: Let's see why we have mentioned in the video above that "the default entry point is not going to work for us because we have express installed."

The default location for the webpack entry point is `./src/index.js` - but because we are already set up with express and have a slightly different file structure, that file doesn't exist! Instead, we need to tell webpack to use a custom entry point. You will learn more about this in the next pages!

Webpack Entry

Let's create the Webpack entry point. Webpack broke at the last step because it didn't have an entry point.

Switch the Git Branch

If you want to practice the steps mentioned in the video demonstration below, then stay on the `1-install-webpack` branch.

Else, switch to the branch `2-add-webpack-entry` corresponding to the current exercise where all the steps have already been carried out, using

```
git checkout 2-add-webpack-entry
git branch

```

Note: If you are switching branches don't forget to make a commit on the current branch before switching.

> You can view the files specific to the current branch, in the Github repository online, as well. For your convenience, the todo steps for the current stage/exercise are mentioned in `STEP-2.md` file available in the root directory of the project.

Webpack Entry

Webpack is going to make a map of our app assets and all of their dependencies, but it needs somewhere to start. The default location for the webpack entry point is `./src/index.js` - but because we are already set up with express and have a slightly different file structure, that file doesn't exist! Instead, we need to tell webpack to use a custom entry point. In `webpack.config.js`, mention the entry point as:

```
module.exports = {
    entry: './src/client/index.js'
}

```

Quick Check -

1. For your easy reference, the `index.js` file at path `./src/client/` is already available in the `2-add-webpack-entry` branch that you are working on. If you are unable to find index.js in `./src/client/`, feel free to create the file here.
2. For building the app, you can use either `npm run build`.
3. After running the build command successfully, verify that a `dist` directory is created in the root, containing the bundled file `main.js`.

> To check what files are included in the webpack build, you can run webpack-cli command from the terminal: `npx webpack --display-modules` or `npx webpack --json > info.json` (if you want to generate a json file in the project folder).

Output and Loaders

We have setup webpack just enough to be performing the most basic function of webpack - creating a dist folder with a main.js file from our entry point. And all of that is great - but none of it is useful yet.

What's wrong? Let's take a look:

1. The distribution folder has no connection whatsoever to our app. If you start the express server, our app is still functioning exactly the same way it did in part 0.
2. The main.js file of our distribution folder contains none of the javascript or other assets we wrote for our webpage.

In short - there are some things wrong with our distribution folder. So it's time to take a look at customizing the webpack output. The "output" of webpack is - no surprise here - the distribution folder. It is where webpack drops or "outputs" the neat bundles of assets it creates from the individual files we point it to.

So we are going to solve the issues above by setting up our webpack output, along with a few other tasks required to make it all work.

Switch the Git Branch

If you want to practice the steps mentioned in the video demonstration below, then stay on the `2-add-webpack-entry` branch.

Else, you can switch to the branch `3-webpack-output-and-loaders` corresponding to the current exercise where all the steps have already been carried out.

```
git checkout 3-webpack-output-and-loaders
git branch

```

Note: If you are switching branches don't forget to make a commit on the current branch before switching.

> You will need to run `npm install` to install babel dependencies in `package.json` file.

About the `dist` Directory, Installing the Babel Tool and Loading the JS Dependencies

Steps Demonstrated

It might seem strange that we have to add ANOTHER library before we're even finished setting up the first library - but believe me, that's how it goes. Babel itself is also not an easy tool to use necessarily, it requires a bit of setup but it is widely used throughout the javascript world to translate new ES syntax into vanilla js that can run on browsers etc. They describe their tool like this:

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.

Once you have the hang of this setup, I have personally found it helpful to have babel even on projects that don't have Webpack. Sometimes I'll install it just for the convenience … like the convenience of no semicolons or being able to use import/export syntax.

First off, we need to install Babel via npm. Babel occasionally changes its install requirements, but at the moment these are the configurations that work and seem to be pretty stable:

`npm i -D @babel/core@^7.5.4 @babel/preset-env@^7.5.4 babel-loader@^8.0.6`

Remember that the -D will install these as **development** dependencies.

And now, just like webpack, babel also requires a config file. In this case, it is a fairly simple one.

Create a new file `.babelrc` in the root of the project. Fill it with this code:

```
{ ‘presets’: ['@babel/preset-env'] }

```

Now, we're about to go through a whole rigamarole of settings, these aren't the kind of settings I would commit to memory necessarily, but I will try to explain the steps as we go along.

First, we have both webpack and babel installed, now we have to get webpack to use babel. Doing that forces us to use a part of webpack we haven't explained yet - we will explain them I promise - but that would jump us just a little ahead of where we are, so for now, you can copy and paste this part. We will observe what it does and then circle back around to explain it.

So, back we go to the webpack config, because we have some things to add.

First, we could specify the "output" of our webpack config, it would look something like this:

`output: { ...output options }`

But at the moment, we don't need to add any custom settings there. The default settings are good enough for us - creating a dist folder in the root of our project.

So, output is set, but there's still the matter of getting webpack to use babel. For that we'll use a webpack loader. Loaders are what we will circle back around to in a second, but for now, add this to your webpack config.

```
    module: {
            rules: [
                    {
                test: /\.js$/,
                exclude: /node_modules/,
                loader: "babel-loader"
                    }
            ]
    }

```

Now, with that loader in place, we should be able to get going.

Go to your index.js file and import our two javascript files (also make sure you export them in the original files!). Then console log one of the functions. Here is an example:

```
import { checkForName } from './js/nameChecker'
import { handleSubmit } from './js/formHandler'

console.log(checkForName);

alert("I EXIST")

```

Really we don't need the alert any more, but either way, delete the current distribution folder and rerun the build command.

You have been successful if you see your console logged function in the main.js that is output.

Resource - Webpack Introductory Documentation

Have a look at the Webpack Introductory Documentation, where you can understand its core concepts - Entry, Output, Loaders, and Plugins.

Loader

In the last section we got webpack's output configured - but to use babel we had to add a loader to our webpack config. We used it then without knowing what it was, but now we can revisit it.

Let's take another look at that loader.

```
module: {
    rules: [
        {
            test: /\.js$/,
            exclude: /node_modules/,
            loader: "babel-loader"
        }
    ]
}

```

Now take a look at how Webpack describes loaders:

> Out of the box, webpack only understands JavaScript and JSON files. Loaders allow webpack to process other types of files and convert them into valid modules that can be consumed by your application.

So loaders allow us to transform files of one type into another type so that webpack can work with them. It might not have looked like we were transforming file types in the step shown above, but we were actually taking es6 files and running the babel-loader over them to turn them into vanilla js files.

There are all sorts of loaders for webpack - take a look at this list. In webpack, most of the things you need to do will end up needing a loader.

We will visit a few more loaders later, but for now, just notice how they work. The `rules` array will contain all of our loaders, each loader specifies what types of files it will run on by running a regex matcher - in the case above we are looking for all .js files - the `$` at the end simply means that nothing comes after that.

But simply looking for all the .js files in our project would be problematic, as we don't want to run this on all the files we have in our node modules. For that kind of use case, we also have an exclude option available to us, and then we simply name the loader to be run on the selected files. Some loaders will have different options, you can always look it up in the loader documentation.

Plugins

Switch the Git Branch

If you want to practice the steps mentioned in the video demonstration below, then use on the `3-webpack-output-and-loaders` branch.

Else, you can switch to the branch `4-webpack-plugins` corresponding to the current exercise where all the steps have already been carried out.

```
git checkout 4-webpack-plugins
git branch

```

Webpack Plugins

Plugins are one of the last vital concepts for webpack. The Webpack documentation explains them like this:

> While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables.

So, pretty much anything that we need to do that falls outside the range of loaders will be accomplished with plugins.

Plugins can do all sorts of things, from automatically adding asset references to an html file (which we'll cover in a second) to allowing for hot module replacement - which is used in React's Create React App to create an auto updating development server.

Mode

Switch the Git Branch

If you wish to practice the steps mentioned in the video demonstration below, use the `4-webpack-plugins` branch.

Else, you can switch to the branch `5-webpack-mode` corresponding to the current exercise where all the steps have already been carried out.

```
git checkout 5-webpack-mode
git branch

```

Webpack Mode

The last core Webpack concept we're going to cover is Mode. In the last concept, when you build, you probably noticed the warning:

Mode Warning in Configuration

The issue is that we haven't told webpack which mode to run in. Modes won't make sense until we delve into environments. You have probably already come across the idea of environments in code projects, or at least heard mention of a "development" or "prod" environment. But in order to use webpack to its true potential, we have to fully understand these concepts.

Production vs. Development Environments

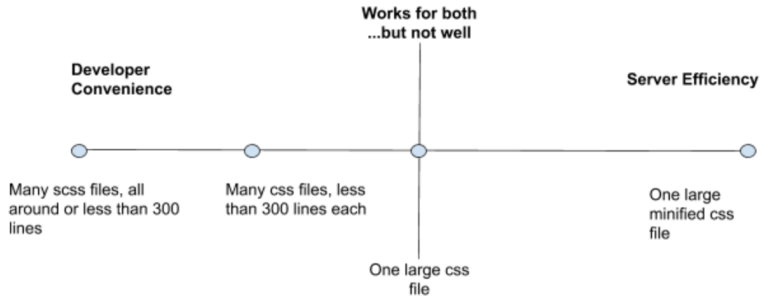
Developers refer to the various "states" of a website as environments. When we are developing a website, we call it the development environment - we run the server on localhost and use tools that are specifically convenient for us as developers. On the other hand, there is a production environment, which is our code on a server and where we can tune every tool and file for optimal efficiency, thereby giving our users the best experience when they use the webpage. There can be many environments for a project, like a testing environment or review environment, but we are going to focus on development and production for now.

There are lots of tools that aim to make writing code easier for developers. One of these tools, called "sass", we will cover in a future lesson, but for now just know that it is css with some developer friendly syntax and features. It's great that we have tools that make development easier, but if you take sass as an example - we can't run sass on a server. All our sass files have to be run through a transpiler in order to

become css that can go on a live webpage. No matter how awesome a development tool is, in the end our code will be judged by how well it runs on a server, and oftentimes what is best for the server is the opposite of what is convenient for developers. So how do we handle both of these environments? By utilizing build tools, we can make code that is convenient for our dev team, without sacrificing speed on the server.

One of the awesome features of webpack, is that it lets us apply configurations to our code based on the environment we are running. We can create a development environment (MODE in webpack) and run totally different loaders and plugins than we do for production mode.

Now we have learned the second part of why we use build tools. First, we learned that build tools allow devs to use the tools that are more convenient for them. The other side of that coin is that build tools simultaneously allow devs to optimize code for the server. Build tools like Webpack are one tool we can use to help us with organization in all environments. If that doesn't fully make sense now, don't worry too much, it will become more clear as we go along.



You can see that a lot of times, what is best for developers is the opposite of what is most efficient for the server. Webpack helps us have the best of both worlds.

Changes proposed for the Configuration Files

1. Create a copy of the `webpack.config.js`, and rename it as `webpack.prod.js`. This file should have `mode: 'production'` statement in `module.exports`.
2. Now, rename the `webpack.config.js` to `webpack.dev.js`. This file should have the following statements in `module.exports`

```
mode: 'development',
devtool: 'source-map',

```

Changes proposed for `package.json`

The statements added to the `package.json` for the configuration files of production and development modes separately in the "script" block are:

```
"scripts": {
    "build-prod": "webpack --config webpack.prod.js",
    "build-dev": "webpack --config webpack.dev.js --open"
},

```

Note that you should remove the `"build": "webpack"` script now from `package.json`, and only have the two related to `build-dev` and `build-prod`. This also means when you build your app with npm, you should use the correct script, e.g.

```
npm run build-dev

```

Note: Some students are facing dependency issues, if you are facing such issues, we would suggest you to modify the version of html plugin in package.json to `"html-webpack-plugin": "^3.2.0"` and then install the packages again `npm i`. Now you can run `npm run build` and the dist folder will be generated again.

Convenience in Webpack

We have made our tedious way through all of the webpack concepts. Now we get to actually make things awesome.

As you probably noted during this lesson so far, there are some things about the process we have now that aren't exactly smooth. Overall, I wouldn't call our set up a good dev experience and we don't have production set up at all. It is functional, and that's about it. But the whole point of using all these tools is to make our lives easier. So...what's going wrong? Let's use what we've learned to make some improvements.

Switch the Git Branch

If you wish to practice the steps mentioned in the video demonstration below, use the `5-webpack-mode` branch.

Else, you can switch to the branch `6-webpack-for-convenience` corresponding to the current exercise where all the steps have already been carried out.

```
git checkout 6-webpack-for-convenience

```

Install Webpack Dev Server

It helps in live reloading of the page, only for Development mode, and automatically re-builds the application.

Up next, have you noticed how we often have to remove the dist folder manually before re-running the build script? From what I have seen, when you rebuild, new code will be added to the bundled files, but if there was old code that you got rid of, webpack build does not remove the old stuff. So, we have been removing the dist folder via the terminal and before rebuilding.

Really though, that is an extra and unnecessary step. If we wanted to go really low tech, we could just edit our build script:

```
rm -rf dist && webpack-dev-server  --config webpack.dev.js --open

```

And there's really nothing wrong with that. Honestly, being comfortable customizing your npm scripts will make so many things easier. But, it is doing a little bit of extra work. That script blindly deletes everything and then rebuilds it, even if 99% of the code remained the same.

To be a little more efficient, there is a webpack plugin called Clean. From its documentation:

> By default, this plugin will remove all files inside webpack's output.path directory, as well as all unused webpack assets after every successful rebuild.

Now, some people will choose to go with the simpler blanket dist folder delete, but just to try it, lets install the clean plugin.

```
npm i -D clean-webpack-plugin

```

Then, as we learned before, to make webpack use a plugin, we have to do two things:

Add a require statement to the top of the webpack config file:

```
const { CleanWebpackPlugin } = require('clean-webpack-plugin');

```

Add the plugin to Plugins array in the module.exports. The clean plugin is a good example of a plugin that allows for various options. Take a look at this:

We could use CleanWebpackPlugin like this:

```
    new CleanWebpackPlugin()
```

```
```

And the above would function. When there are no options passed in, a plugin will run all the default settings, but we can also pass in our custom selections of the various plugin options, like this:

```
        new CleanWebpackPlugin({
                // Simulate the removal of files
                dry: true,
                // Write Logs to Console
                verbose: true,
                // Automatically remove all unused webpack assets on rebuild
                cleanStaleWebpackAssets: true,
                protectWebpackAssets: false
        })

```

You can’t know what options a plugin allows without reading the documentation for that plugin.

Add that code to your wepack.dev.js file and rerun the build script, now you’ll see a few lines added to the webpack output that tell you the clean plugin is functioning.

Note: If facing dependency issue, install: `npm i -D clean-webpack-plugin@^3.0.0`

Webpack Conclusion

Summary

Webpack Entry: Webpack is going to make a map of our app assets and all of their dependencies, but it needs somewhere to start. That entry point is Webpack entry.

Loaders: Loaders allow us to transform files of one type into another type so that Webpack can work with them

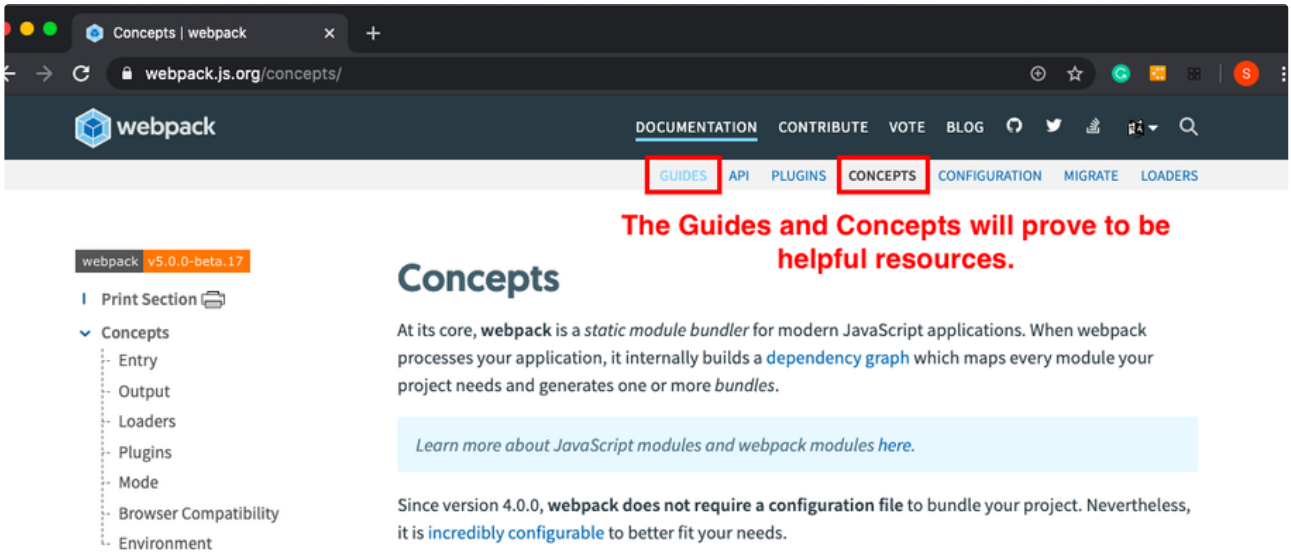
Plugins: While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management, and injection of environment variables.

Mode: One of the awesome features of Webpack, is that it lets us apply configurations to our code based on the environment we are running. We can create a development environment (MODE in Webpack) and run totally different loaders and plugins than we do for production mode

Additional Resources

If you want to go further with webpack, check out these topics next:

1. Multiple entry points with webpack
2. Using webpack to be more efficient with your styles and assets
3. Cache busting with webpack



Helpful official documentation

Sass and Webpack

Sass Basics

Lesson Objectives

In this lesson, we will learn about a CSS extension language - **Sass**, and how it can be used in conjunction with webpack. Sass provides an extra set of CSS language syntax that helps writing more efficient styles. The objectives are:

1. Explain the basics of Sass, and introduce the important features
2. Describe more about the features - nesting, variables, ampersand
3. Learn to use Sass with webpack

Introduction

The two of the famous CSS extension languages are:

1. Sass - Note that these files have an extension as `.scss`
2. Less - It has `.less` as file extension

We are going to learn about Sass in this lesson.

When a developer writes a `.scss` file, there is a problem. Browsers don't know what Sass is, they don't run Sass, they run CSS. Sass transpiles to CSS - or in more common english - Sass can be directly translated to CSS. Anything you write in Sass can be written in 100% pure CSS, they are equivalent to each other, but the Sass syntax is going to be much shorter and easier to write than the CSS.

Sass now comes with some of its own tools to run that translation process to create CSS, but since we're using webpack anyway, we're going to leverage webpack to do that job.

Sass Tutorial Resource

Learning sass is worth every minute of your time as you hone your front end developer skills. These days it is pretty much required knowledge and it could be a whole course of its own. Unfortunately, this course is jam-packed enough as it is so we aren't going to spend much time here. On the bright side, the sass website has a good tutorial that will take you through all the basics. In the next few sections, I'll highlight three of the concepts I think are most important to learn about sass.

Again, we strongly recommend you to give it some time to read through the preprocessing, variables, nesting, modules, inheritance, and operators, from the link above.

Sass Nesting

Nesting is one of the key features of Sass. HTML elements are nested, and so CSS is nested by nature, but that isn't reflected in CSS syntax. Sass allows you to write styles for nested elements in a much more intuitive way. Writing nested sass can mean that you don't have to create nearly as many individual classes, which can save a lot of time and markup. Not only that, but you are much more likely to be able to edit styles by only touching the CSS file, without having to go back and forth between the HTML and CSS. As a rule of thumb though, *if you find yourself nesting more than three levels deep*, it's probably time for a new class.

Nesting Example

Take a look at these examples of nesting:

```
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}

```

The code above, when translated to css, would become:

```
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}
nav li {
  display: inline-block;
}
nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}

```

I chose this feature as one of the core things to know about sass because it is probably the single feature that most impacts your ability to write compact and efficient styles.

Sass Variables

Another great sass feature is actually one that's available in vanilla css as well, but the intentional use of variables in stylesheets, especially when theming, can make for far more flexible and understandable styles.

```
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}

```

In CSS becomes:

```
body {
  font: 100% Helvetica, sans-serif;
  color: #333;
}

```

Perhaps that doesn't look impressive, but what it means most certainly is. Imagine, you've built a website with hard coded values for font all throughout. The client comes to two days before launch, after the last pass of QA, and tells you that everything looks good but they want to change the font (as happens from time to time). You might cringe, because it will take you an hour to go through every single reference to font in the whole app, replace it with the new one and change sizes proportionally. Or, you might sigh a sigh of relief because you used a sass variable, and now all of those 170 references to font are all using the same single variable, you change that one value in your code, and can go to bed early that night instead of staying up and working.

Sass Ampersand

The `&` is an extremely useful feature in Sass (and Less). It's used when *nesting*. It can be a nice time-saver when you know how to use it, or a bit of a time-waster when you're struggling and could have written the same code in regular CSS.

The `&` comes in handy when you're nesting and you want to create a more specific selector, like an element that has *both* of two classes, like this:

```
.some-class.another-class { }
```

You can do this while nesting by using the `&`.

```
.some-class {
  &.another-class {}
}
```

The `&` always refers to the parent selector when nesting. Think of the `&` as being removed and replaced with the parent selector. Like this

Webpack and Sass

Switch the Git Branch

You can continue to practice the exercise instructions below on `0-initial-setup` branch, or you can switch to `1-add-sass-loaders` branch corresponding to the current exercise where all the steps have already been carried out. Use the following checkout command in your project root directory:

```
git checkout 1-add-sass-loaders

```

Set up Sass with Webpack

So now for the webpack portion of all of this. Like we talked about earlier, we are going to use webpack loaders to turn our sass into css. First let's install all the tools we'll need:

```
npm i -D style-loader node-sass css-loader sass-loader

```

> Note: The latest `sass-loader` has some breaking changes. We recommend to use the versions of the dependencies from `package.json` in branch `1-add-sass-loaders` to avoid errors when running the webpack build script.
>
> For reference(node v14):
>
> In dev dependencies the versions that are be used are:
>
> `"@babel/core": "^7.10.2", "@babel/preset-env": "^7.10.2", "babel-loader": "^8.1.0", "css-loader": "^3.6.0", "html-webpack-plugin": "^3.2.0", "node-sass": "^4.14.1", "sass-loader": "^7.3.1", "style-loader": "^0.23.1"`

Then add this test case to the rules array in your dev webpack config.

```
{
        test: /\.scss$/,
        use: [ 'style-loader', 'css-loader', 'sass-loader' ]
}
```

Well, if you think back to how we got the javascript files into our `main.js`, we are missing one of those steps for our css. There are the following steps that you have to do to use webpack loaders:

1. Install the sass loader, using `npm i -D style-loader node-sass css-loader sass-loader`. Babel loader is optional to be install for this exercise, `npm i -D @babel/core @babel/preset-env babel-loader`

2. Call the loader to the "rules" array in the `webpack.dev.js` config while targeting the correct file extensions.

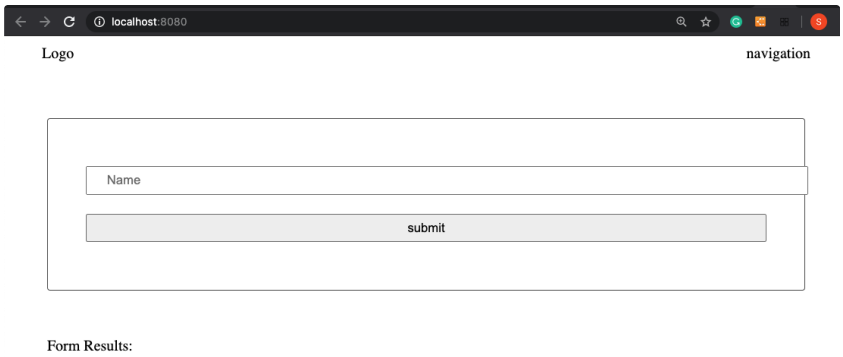
```
{
     test: /\.scss$/,
     use: [ 'style-loader', 'css-loader', 'sass-loader' ]
}

```

3. In the last exercise, we already had converted our `.css` to `.scss` in `client/style` directory. Now we need to import the files in `index.js`. And because of the dependency tree that webpack builds, if nothing ever is imported, it's as if it doesn't exist. So, to fix that, let's go to `client/index.js`. Because of css-loader, we can add lines like this:

```
import './styles/resets.scss'
import './styles/base.scss'
import './styles/footer.scss'
import './styles/form.scss'
```

```
import './styles/header.scss'

```

4. Run webpack and look for your styles in the bundled js file. The included `package.json` file in branch `1-add-sass-loaders` has been adjusted to not use webpack-dev-server. Now run `npm run build-dev` and `npm run start` so that you can verify in the browser, as shown in the snapshot below:



App running with style-loaders at port 8080

This time check the `main.js` for some of our sass styles, and they should just be there!

Troubleshooting: Most of the students are facing error while running `npm run start`, or are unable to view the `dist` folder. In both of these cases, we suggest you to run `npm run build-prod` first. This will create the `dist` folder and eventually remove the error while runnning `npm run start`.
To run `npm run build-prod`, you need to add the sass loaders in the `webpack.prod.js` file:

```
{
                test: /\.scss$/,
                use: [ 'style-loader', 'css-loader', 'sass-loader' ]
}

```

Now, one thing that might feel awkward, is that all of our css styles are being run through our main.js. Webpack natively only understands javascript, so it makes sense that until we tell it to do otherwise, it turns everything into javascript by default. Now, this isn't a problem for development mode necessarily, but it can cause styles to load a split second after content on the server, so we are going to have to clean that up for production mode later on.

Additional Resources

We know enough sass to be dangerous, and we can add it to a webpack project. We got a little more practice with loaders and learned that they can be a lot more helpful when chained together than just by themselves.

There are so many more things that you can do with loaders that we just don't have time to go into. Some of these things include:

- Loading images
- Working in typescript or other languages that compile to javascript
- Working with third party style and js libraries like Bootstrap

(Doc) Week 4:

React: Fundamentals 01

Why React?

Introduction to React Fundamentals

What is React?

Straight from Facebook's React documentation, React is:

> *A JavaScript library for building user interfaces.*

Put in other words, it's a popular front end library that developers use to create dynamic applications for the web.

The MERN Stack

React is part of the MERN stack, a collection of technologies based on JavaScript that's used for building full stack web apps:

- MongoDB, which is the application's database.
- Express, which handles the application's backend
- React, which is used to build the front end
- Node, which is the runtime environment for the application

Why React?

Now, why is React a popular choice for building front end user interfaces? Well for one, it's the **ease of use**. At its core, React is written with the same JavaScript that you already know, making key concepts more intuitive to pick up.

React is also based on **reusable components**. These components manage their own state, and by leveraging composition, we can use and reuse them to build more complex user interfaces.

Another key reason developers choose React is its **declarative API**. For the most part, you just describe the app's UI and how its data changes, and React efficiently manages all the renders and re-renders on its own. In short, React gives you all the tools you need to effectively build an interactive web app. We'll cover these tools and more throughout this course.

What is Composition?

From Wikipedia, Composition is:

> to combine simple functions to build more complicated ones

Let's take a look at how we can build up complex functions just by combining simple ones together.

Benefits of Composition

Because the concept of composition is such a large part of what makes React awesome and incredible to work with, let's dig into it a little bit. Remember that composition is just combining simple functions together to create complex functions. There are a couple of key ingredients here that we don't want to lose track of. These ingredients are:

- Simple functions
- Combined to create another function

Composition is built from *simple* functions. Let's look at an example:

```
function getProfileLink (username) {
  return 'https://github.com/' + username
}

```

This function is *ridiculously* simple, isn't it? It's just one line! Similarly, the `getProfilePic()` function is *also* just a single line:

```
function getProfilePic (username) {
  return 'https://github.com/' + username + '.png?size=200'
}

```

These are definitely *simple* functions, so to compose them, we'd just *combine* them together inside another function:

```
function getProfileData (username) {
  return {
    pic: getProfilePic(username),
    link: getProfileLink(username)
 }
}

```

Now we *could* have written `getProfileData()` *without* composition by providing the data directly:

```
function getProfileData (username) {
  return {
    pic: 'https://github.com/' + username + '.png?size=200',
    link: 'https://github.com/' + username
 }
}

```

There's nothing technically wrong with this at all; this is entirely accurate JavaScript code. But this *isn't* composition. There are also a couple of potential issues with this version that *isn't* using composition. If the user's link to GitHub is needed somewhere else, then duplicate code would be needed. A good function should follow the "DOT" rule:

> **D**o **O**ne **T**hing

This function is doing a couple of different (however minor) things; it's creating two different URLs, storing them as properties on an object, and then returning that object. In the composed version, each function just does one thing:

- `getProfileLink()` – just builds up a string of the user's GitHub profile link
- `getProfilePic()` – just builds up a string the user's GitHub profile picture
- `getProfileData()` – returns a new object

React & Composition

React makes use of the power of composition, heavily! React builds up pieces of a UI using **components**. Let's take a look at some pseudo code for an example. Here are three different components:

```
<Page />
<Article />
<Sidebar />

```

Now let's take these *simple* components, combine them together, and create a more complex component (that is, composition in action!):

```
<Page>
 <Article />
 <Sidebar />
</Page>

```

Now the `Page` component has the `Article` and `Sidebar` components *inside*. This is just like the earlier example where `getProfileData()` had `getProfileLink()` and `getProfilePic)_` inside it.

We'll dig into components soon, but just know that composition plays a huge part in building React components.

Recap

Composition occurs when *simple* functions are *combined* together to create *more complex* functions. Think of each function as a single building block that *does one thing* (DOT). When you combine these simple functions together to form a more complex function, this is **composition**.

Further Research

- Compose me That: Function Composition in JavaScript
- Functional JavaScript: Function Composition For Every Day Use

What is Declarative Code?

Difference Between Imperative and Declarative Code

Imperative Code

A lot of JavaScript is **imperative code**. If you don't know what "imperative" means here, then you might be scratching your head a bit. According to the dictionary, "imperative" means:

> expressing a command; commanding

When JavaScript code is written *imperatively*, we tell JavaScript exactly **what** to do and **how** to do it. Think of it as if we're giving JavaScript *commands* on exactly what steps it should take. For example, here is the humble `for` loop:

```
const people = ['Amanda', 'David', 'Andrew', 'Karen', 'Richard', 'Tyler'];
const excitedPeople = [];

for (let i = 0; i < people.length; i++) {
  excitedPeople[i] = people[i] + '!';
}

```

If you've worked with JavaScript any length of time, then this should be pretty straightforward. We're looping through each item in the `people` array, adding an exclamation mark to their name, and storing the new string in the `excitedPeople` array. Pretty simple, right?

This is *imperative* code, though. We're commanding JavaScript what to do at every single step. We have to give it commands to:

- Set an initial value for the iterator - (`let i = 0`)
- Tell the `for` loop when it needs to stop - (`i < people.length`)
- Get the person at the current position and add an exclamation mark - (`people[i] + '!'`)
- Store the data in the `i` th position in the other array - (`excitedPeople[i]`)
- Increment the `i` variable by one - (`i++`)

Remember the example of keeping the air temperature at 71º? In my old car, I would turn the knob to get the cold air flowing. But if it got too cold, then I'd turn the knob up higher. Eventually, it would get too warm, and I'd have to turn the knob down a bit, again. I'd have to manage the temperature myself with every little change. Doesn't this sound like an imperative situation to you? I have to manually do multiple steps. It's not ideal, so let's improve things!

Declarative Code

In contrast to imperative code, we've got **declarative code**. With declarative code, we don't code up all of the steps to get us to the end result. Instead, we *declare* what we want done, and JavaScript will take care of doing it. This explanation is a bit abstract, so let's look at an example. Let's take the imperative `for` loop code we were just looking at and refactor it to be more declarative.

With the imperative code we were performing all of the steps to get to the end result. What _is_ the end result that we actually want, though? Well, our starting point was just an array of names:

```
const people = ['Amanda', 'David', 'Andrew', 'Karen', 'Richard', 'Tyler'];

```

The end goal that we want is an array of the same names but where each name ends with an exclamation mark:

```
["Amanda!", "David!", "Andrew!", "Karen!", "Richard!", "Tyler!"]

```

To get us from the starting point to the end, we'll just use JavaScript's `.map()` function to declare what we want done.

```
const excitedPeople = people.map((name) => name + "!");

```

That's it! Notice that with this code we haven't:

- Created an iterator object
- Told the code when it should stop running
- Used the iterator to access a specific item in the `people` array
- Stored each new string in the `excitedPeople` array

...all of those steps are taken care of by JavaScript's `map()` Array method.

> 💡 What are `map()` and `filter()`?💡
>
> A bit rusty on JavaScript's `map()` and `filter()` Array methods? Or perhaps they're brand new to you. In either case, we'll be diving into them in the React is "just JavaScript" section. Hold tight!

React is Declarative

We'll get to writing React code very soon, but let's take another glimpse at it to show how it's declarative.

```
<button onClick={activateTeleporter}>Activate Teleporter</button>

```

It might seem odd, but this is valid React code and should be pretty easy to understand. Notice that there's just an `onClick` attribute on the button...we aren't using `addEventListener()` to set up event handling with all of the steps involved to set it up. Instead, we're just declaring that we want the `activateTeleporter()` function to run when the button is clicked.

Recap

Imperative code instructs JavaScript on *how* it should perform each step. With *declarative* code, we tell JavaScript *what* we want to be done, and let JavaScript take care of performing the steps.

React is declarative because we write the code that we *want*, and React is in charge of taking our declared code and performing all of the JavaScript/DOM steps to get us to our desired result.

Further Research

- Imperative vs Declarative Programming blog post
- Difference between declarative and imperative in React.js? from StackOverflow
- Array.prototype.map() on MDN
- Array.prototype.filter() on MDN

Unidirectional Data Flow

Data-Binding In Other Frameworks

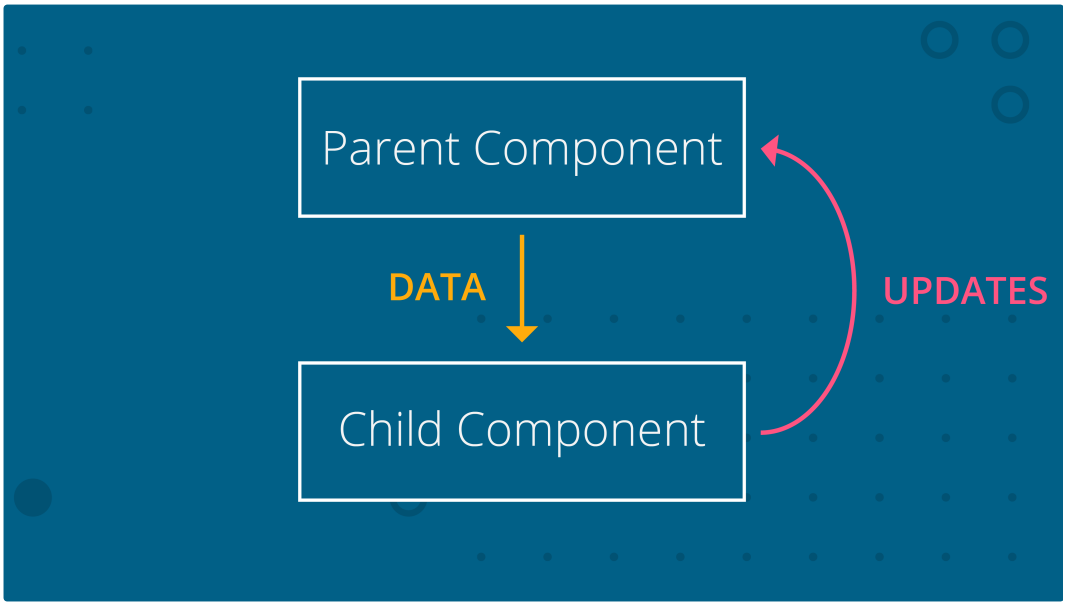
Front-end frameworks like Angular and Ember make use of two-way data bindings. In two-way data binding, the data is kept in sync throughout the app no matter where it is updated. If a model changes the data, then the data updates in the view. Alternatively, if the user changes the data in the view, then the data is updated in the model. Two-way data binding sounds really powerful, but it can make the application harder to reason about and know where the data is actually being updated.

If you'd like to learn more about how other frameworks bind data, feel free to check out their documentation below:

- Angular's two-way data binding
- Vue.js data binding
- Ember's two-way data binding

The Flow of Data in React

Data moves differently with React's *unidirectional* data flow. In React, the data flows from the parent component to a child component.



Data flows down from parent component to child component. Data updates are sent to the parent component where the parent performs the actual change.

In the image above, we have two components:

- A parent component
- A child component

The data lives in the parent component and is passed down to the child component. Even though the data lives in the parent component, both the parent and the child components can use the data. However, if the data must be updated, then only the parent component should perform the update. If the child component needs to make a change to the data, then it would send the updated data to the parent component where the change will actually be made. Once the change is made in the parent component, the child component will be passed the data (that has just been updated!).

Now, this might seem like extra work, but having the data flow in one direction and having one place where the data is modified makes it much easier to understand how the application works.

Now let's say that the `FlightPlanner` component has two child components: `LocationPicker` and `DatePicker`. `LocationPicker` itself is a parent component that has two child components: `OriginPicker` and `DestinationPicker`.

```
<FlightPlanner>

 <LocationPicker>
  <OriginPicker />
  <DestinationPicker />
 </LocationPicker>

 <DatePicker />

</FlightPlanner>
```

Recap

In React, data flows in only one direction, from parent to child. If data is shared between sibling child components, then the data should be stored in the parent component and passed to both of the child components that need it.

React is "Just JavaScript"

It's Just JavaScript

One of the great things about React is that a lot of what you'll be using is regular JavaScript. To make sure you're ready to move forward, please take a look at the following code:

```
const shelf1 = [
  { name: "name1", shelf: "a" },
  { name: "name2", shelf: "a" },
];
const shelf2 = [
  { name: "name3", shelf: "b" },
  { name: "name4", shelf: "b" },
];
const allBooks = [...shelf1, ...shelf2];

const filter = (books) => (shelf) => books.filter((b) => b.shelf === shelf);

const filterBy = filter(allBooks);
const booksOnShelf = filterBy("b");

```

Note that the above code leverages spread syntax and higher order functions. If *any* of the code looks confusing, or if you simply need a refresher on ES6, please complete our ES6 course before moving forward.

Functional Programming

Over the past couple of years, functional programming has had a large impact on the JavaScript ecosystem and community. Functional programming is an advanced topic in JavaScript and fills hundreds of books. It's too complex to delve into the benefits of functional programming (we've got to get to React content, right?). But React builds on a lot of the techniques of functional programming -- techniques that you'll learn as you go through this program.

However, there are a couple of important JavaScript functions that are vital to functional programming that we should look at. These are the Array's `map()` and `filter()` methods.

Recap

React builds on what you already know - JavaScript! You don't have to learn a special template library or a new way of doing things.

Two of the methods that you'll be using quite a lot are:

- `map()`
- `filter()`

It is critical that you are comfortable using these methods. In the next couple sections, you'll get some in-depth practice using these array methods.

Creating UI Elements

React uses JavaScript objects to create React elements. We'll use these React elements to describe what we want the page to look like, and React will be in charge of generating the DOM nodes to achieve the result.

Recall from the previous lesson the difference between imperative and declarative code. The React code that we write is *declarative* because we aren't telling React *what* to do; instead, we're writing React elements that describe what the page should look like, and React does all of the implementation work to get it done.

Enough theory, let's get to it and create some elements!

Getting Started

Here is the React's `createElement()` method:

```
1 React.createElement( /* type */, /* props */, /* content */ );
2
```

We'll take a deep dive into what all that entails in just a bit! We'll start things out with a project that's already set up. For now, *don't worry about creating a project or coding along*. There will be plenty of hands-on work for you to do soon enough! We'll start building our in-class project, the Contacts app, in the next section. You can use this React Sandbox.

> 💡 Trying Out React Code 💡

> React is an extension of JavaScript (i.e., a JavaScript *library*), but it isn't built into your browser. You wouldn't be able to test out React code samples in your browser console the way you would if you were learning JavaScript. In just a bit, we'll see how to install and use a React environment!

Rendering Elements onto the DOM

We used ReactDOM's `render()` method to render our element onto a particular area of a page. In particular, we rendered the `element` onto a DOM node called `root`. But where did this `root` come from?

Apps built with React typically have a single `root` DOM node. For example, an HTML file may contain a `<div>` with the following:

```
1 <div id="root"></div>
2
```

By passing this DOM node into `getElementById()`, React will end up controlling the entirety of its contents. Another way to think about this is that this particular `<div>` will serve as a "hook" for our React app; this is the area where React will take over and render our UI!

A Closer Look at `createElement()`

We just used React's `createElement()` method to construct a "React element". The `createElement()` method has the following signature:

```
1 React.createElement( /* type */, /* props */, /* content */ );
2
```

Let's break down what each item can be:

- `type` – either a string or a React Component This can be a string of any existing HTML element (e.g. `'p'`, `'span'`, or `'header'`) or you could pass a React *component* (we'll be creating components with JSX, in just a moment).
- `props` – either `null` or an object This is an object of HTML attributes and custom data about the element.

- `content` – `null`, a string, a React Element, or a React Component Anything that you pass here will be the content of the rendered element. This can include plain text, JavaScript code, other React elements, etc.

Recap

In the end, remember that React is only concerned with the "View" layer of our app. This is what the user sees and interacts with. As such, we can use `createElement()` to render HTML onto a document. More often than not, however, you'll use a syntax extension to describe what your UI should look like. This syntax extension is known as JSX, and just looks similar to plain HTML written right into a JavaScript file. The JSX gets transpiled to React's `createElement()` method that outputs HTML to be rendered in the browser.

A great mindset to have when building React apps is to think in components. Components represent the modularity and reusability of React. You can think of your component classes as factories that produce instances of components. These component classes should follow the single responsibility principle and just "do one thing." If it manages too many different tasks, it may be a good idea to decompose your component into smaller subcomponents.

Further Research

- Rendering Elements from the React docs

Building UI with JSX

What is JSX?

JSX stands for **JavaScript XML**. JSX allows us to write HTML in React. JSX makes it easier to write and add HTML in React.

`createElement()` Returns One Root Element

Recall that `createElement()` creates a single React element of a particular type. We'd normally pass in a tag such as a `<div>` or a `<span>` to represent that type, but note that it's possible that the content argument can be *another* React element!

Consider the following example:

```
const element = React.createElement('div', null,
 React.createElement('strong', null, 'Hello world!')
);

```

Here, "Hello world!" will be wrapped in a `<div>` when this React element renders as HTML. While we can indeed nest React elements, remember that the overall call still just returns a *single element*.

JSX Returns *One* Main Element, Too

When writing JSX, keep in mind that it must only return a *single element*. This element may have any number of descendants, but there *must* be a single root element wrapping your overall JSX (typically a `<div>` or a `<span>`). Check out the following example:

```
const message = (
 <div>
   <h1>All About JSX:</h1>
   <ul>
    <li>JSX</li>
    <li>is</li>
    <li>awesome!</li>
   </ul>
 </div>
);

```

See how there's only one `<div>` element in the code above and that all other JSX is nested inside it? This is how you have to write it if you want multiple elements. To be completely clear, the following is *incorrect* and will cause an error:

```
const message = (
 <h1>All About JSX:</h1>
 <ul>
   <li>JSX</li>
   <li>is</li>
   <li>awesome!</li>
 </ul>
);

```

In the above example, we have two sibling elements that are both at the root level (i.e. `<h1>` and `<ul>`) . This won't work and will give the error:

> Syntax error: Adjacent JSX elements must be wrapped in an enclosing tag

Since we know that JSX is really just a syntax extension for `createElement()`, this makes sense; `createElement()` takes in only one tag name (as a string) as its first argument.

Intro to Components

So far we've seen how `createElement()` and JSX can help us produce some HTML. Typically, though, we'll use one of React's key features, *components*, to construct our UI. Components refer to *reusable* pieces of code ultimately responsible for returning HTML to be rendered onto the page. More often than not, you'll see React components written with JSX.

Since React's main focus is to streamline building our app's UI, there is only one thing absolutely required in any React function-component: a `return` statement.

Let's go ahead and build our first function component!

Declaring Components in React

We can define the `ContactList` component as an arrow function like so:

```
const ContactList = () => {
  // ...
}

```

Sometimes, you'll also see components defined as more "traditional" functions like so:

```
function ContactList() {
  // ...
}

```

Both ways are functionally the same, and are mostly a matter of preference. Some minor differences include being able to omit the explicit `return` statement in the arrow function, but feel free to use whichever you feel most comfortable with.

Note that in legacy React code (i.e., prior to React 16.8), you'll see components written with ES6 class syntax as well:

```
import React, { Component } from 'react';

class ContactList extends Component {
  // ...
}

```

In this course, we'll prefer functional components to class components when building React applications.

Building UI with JSX with Recap

React components return a single root element, even if it contains multiple child elements. JSX is a syntax extension to JavaScript, and also returns a single root element. You'll typically see React components written with JSX, and in this course, we'll do the same.

Further Research

- Introducing JSX from the official React documentation

create-react-app

Using `create-react-app`

Under the hood, Create React App is powered by `react-scripts` to streamline the build process. Let's review what they are, then go through a demo of `create-react-app` together.

Before Using `create-react-app`

Note: If you already have Node.js on your machine, it's a good idea to reinstall it to make sure you have the latest version. Keep in mind that Node.js now comes with `npm` by default.

MacOS

The easiest way to install Node.js is to use the installer on the Node.js site.

Alternatively, you can use Homebrew. First, install Homebrew (if you don't already have it) by running following in your terminal:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

After installation, check that it was installed by running `brew --version`. You should see the version number that was installed.

1. Run `brew install node`
2. Run `node --version` to confirm that Node.js was installed
3. Check that `npm` was installed as well by running `npm --version`
4. Run `brew install yarn`
5. Run `yarn install && yarn --version`

Windows

1. Please download the Node.js installer, go through the installation process, and restart your computer once you're done.
2. If you'd like to use `yarn`, please follow the `yarn` installation instructions.
   - Run `yarn --version` to make sure `yarn` has been successfully installed.

Linux

1. Please follow these instructions to install Node.js.
2. Run `sudo apt-get install -y build-essential`.
3. Please follow the `yarn` installation instructions.
4. Run `yarn --version` to make sure `yarn` has been successfully installed.

Scaffolding Your React App

JSX is awesome, but it does need to be transpiled into regular JavaScript before reaching the browser. We typically use a transpiler like Babel to accomplish this for us. We can run Babel through a build tool, like Webpack which helps bundle all of our assets (JavaScript files, CSS, images, etc.) for web projects.

To streamline these initial configurations, we can use Facebook's `create-react-app` package to manage all the setup for us! This tool is incredibly helpful to get started in building a React app, as it sets up everything we need with *zero configuration*!

To use `create-react-app` (through the command-line with npm), go ahead and enter the following command into your terminal:

```
npx create-react-app <name of your app>

```

Be sure to replace `<name of your app>` above with the name your application (that is, without the `<` and `>` symbols). Note that with `npx` commands, you won't have to install `create-react-app` globally (as was done in the earlier years of React).

> 💡 *Issues with a Previous Installation?* 💡
>
> *If you've previously used* `create-react-app` on your local machine, you may see an error along the lines of "*You are running create-react-app x.x.x, which is behind the latest release (x.x.x)*" after running the above command. To continue using `create-react-app` without this error, uninstall `create-react-app`, clear your `npx` cache, then try creating a new app again:
>
> `npm uninstall -g create-react-app npx clear-npx-cache npx create-react-app <name of your app>`
>
> Feel free to review this issue in the official create-react-app repo for more information.

> 💡 *Dependency Issues?* 💡
>
> *When working with* `npm` commands and installing dependencies in this course, you may run into warnings about potential vulnerabilities or other issues. For the most part, these warnings generally shouldn't impact development. However, if you'd prefer to force a security audit, feel free to run `npm audit fix --force`. You can read more about npm-audit in the npm documentation.

> 💡 *The Yarn Package Manager* 💡
>
> *With earlier versions of Create React App, you may be asked to use* `yarn start` to start the development server. If you haven't used it before, Yarn is a package manager that's similar to npm. Yarn was created from the ground up by Facebook to improve on some key aspects that are slow or lacking in npm.

> *If you don't want to install Yarn, you don't have to! What's great about it is that almost every use of* `yarn` can be swapped with `npm` and everything will work just fine! So if the command is `yarn start`, you can use `npm start` to run the same command. Note that modern versions of `create-react-app` default to `npm` commands.

> ***React Version Updates*** 💡
>
> *React is a dynamic JavaScript library with features and functionality that is continuously updated. When new features are added to the library, React releases new versions of the library. You can see the version of React the* `create-react-app` command uses by looking in the `package.json` file created by running the command. React is declared as a library used in your application in this file, and the version number is specified next to it:
>
> `"react": "^17.0.2",`
>
> *The code examples and walkthroughs in this course use the React version 17 library. If you'd like to experiment with using React 18, you can install it and make minor changes to the code generated by* `create-react-app`.

Recap

Facebook's `create-react-app` is a command-line tool that scaffolds a React application. Using this, there is no need to install or configure module bundlers like Webpack, or transpilers like Babel. These come preconfigured (and hidden) with `create-react-app`, so you can jump right into building your app!

Further Research

- create-react-app on GitHub
- create-react-app Release Post from the React blog
- Updates to create-react-app from the React blog

Composing with Components

Component composition is a powerful pattern to make your components more reusable.

If you are already familiar with React, you're probably already using it (maybe without knowing its name).

What is component composition in React?

In React, we can make components more generic by accepting **props**, which are to React components what parameters are to functions.

Component composition is the name for **passing components as props to other components**, thus creating new components with other components.

An Example of Composition in React

Favor Composition Over Inheritance

You might have heard before that it's better to "favor composition over inheritance." This is a principle that I believe is difficult to learn today. Many of the most popular programming languages make extensive use of inheritance, and it has carried over into popular UI frameworks like the Android and iOS SDKs.

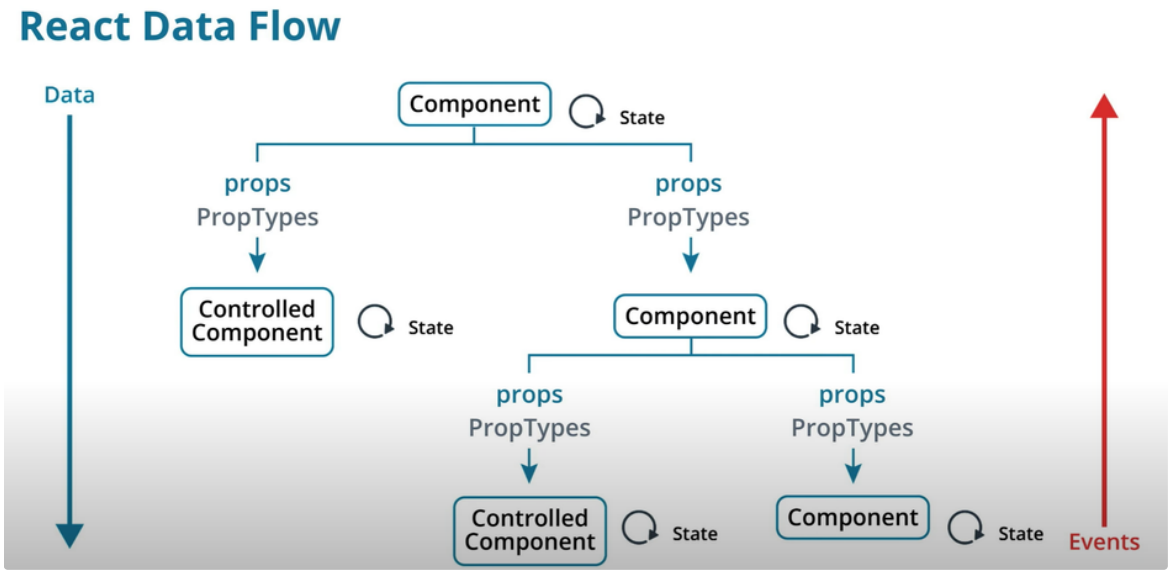
In contrast, React uses *composition* to build user interfaces. Instead of extending base components to add more UI or behavior, we compose elements in different ways using nesting and props. You ultimately want your UI components to be independent, focused, and *reusable*.

So if you've never understood what it means to "favor composition over inheritance," you'll definitely learn about it when using React!

State Management

Introduction to State Management

The Big Picture: React Data Flow

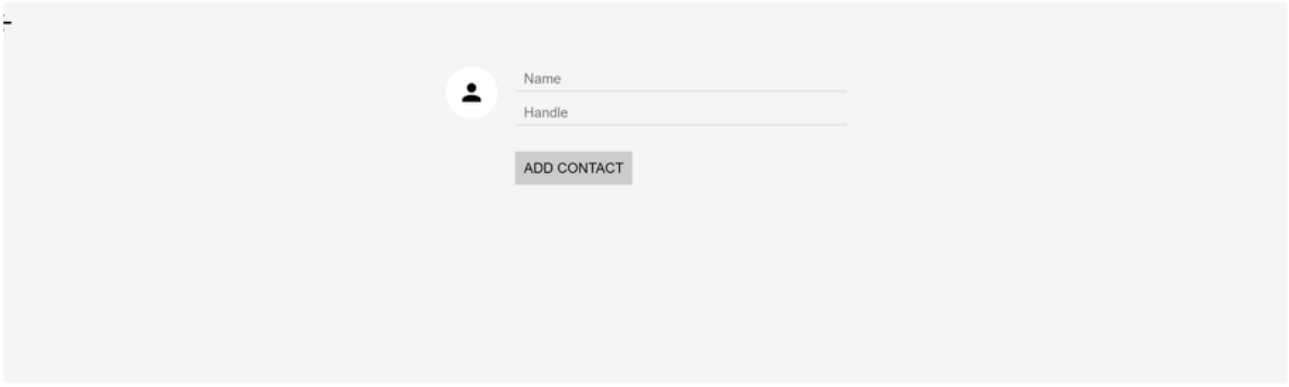


State Management

- **Passing Data with Props** between parent and child components
- **Add State to a Component**, which represents mutable data
- **Update State with useState(), one of the key Hooks of React**
- **Typechecking with PropTypes**, allowing us to ensure proper data flow to components
- **Building Forms with Controlled Components**, leveraging React state as the source of truth

Introducing the Contacts App

Through the rest of this lesson, we'll be building an in-class React project together: the Contacts App. We'll build out each feature and each part of the user interface step-by-step. Here is the Demo UI of Contacts App:

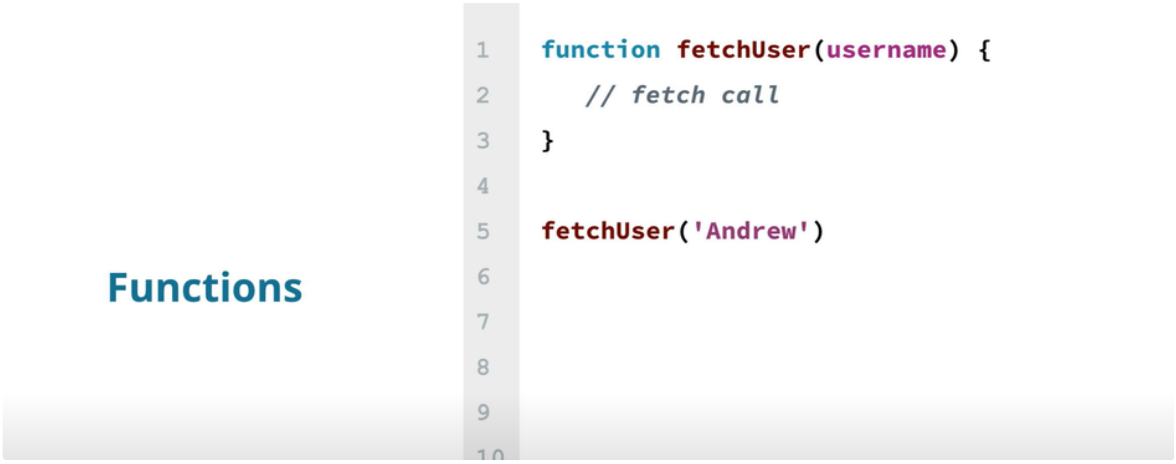



Search Contacts
Karen Isgrigg
@karen_isgrigg
Tyler McGinnis
@tylermcginnis
Trung
Student

Passing Data with Props

Passing Data with Props

With everyday JavaScript functions, we know how we can pass data into them: via the function's arguments. But how could we do the same thing with React Components? Let's check it out:



Passing Data with Props in the Contacts App

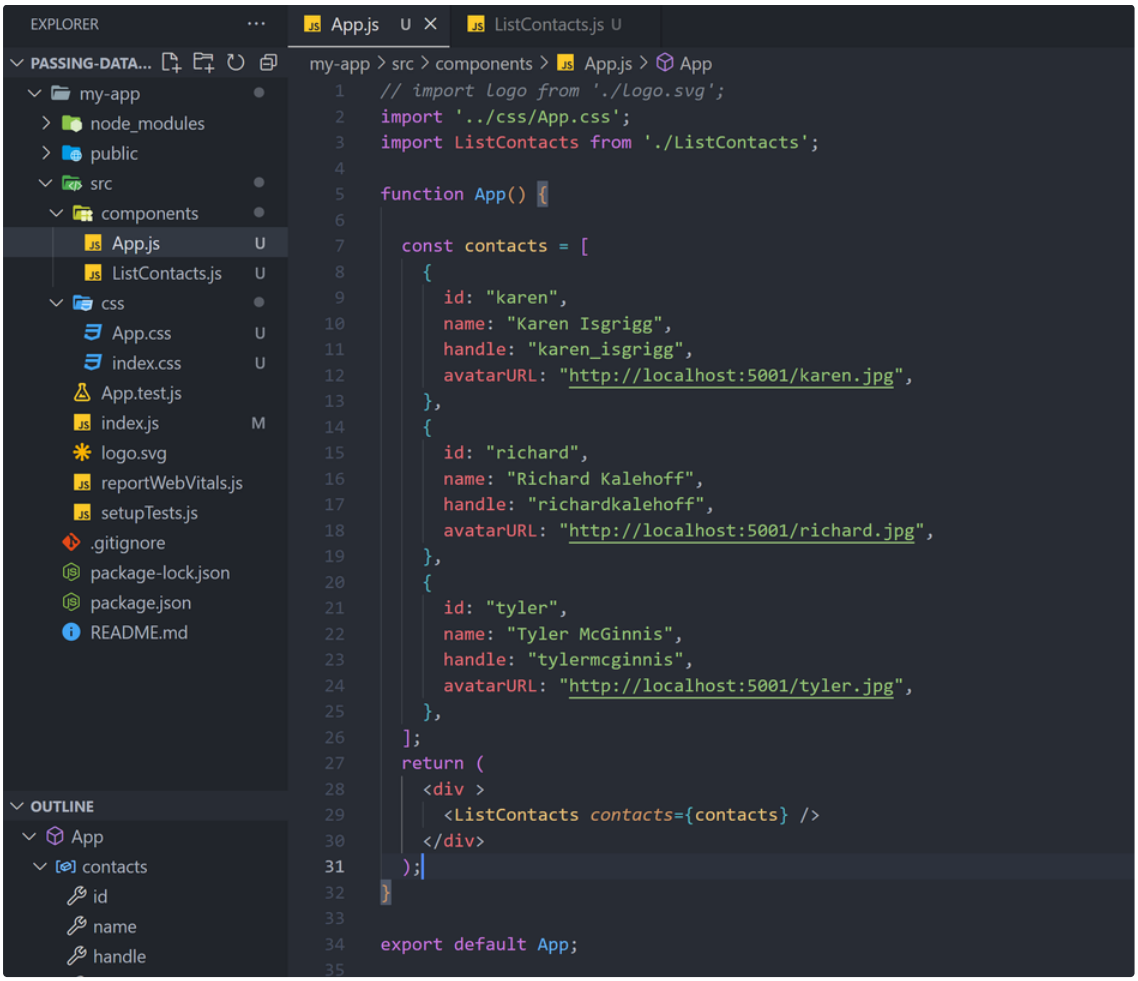
You'll be needing this `contacts` array to follow along with the example below:

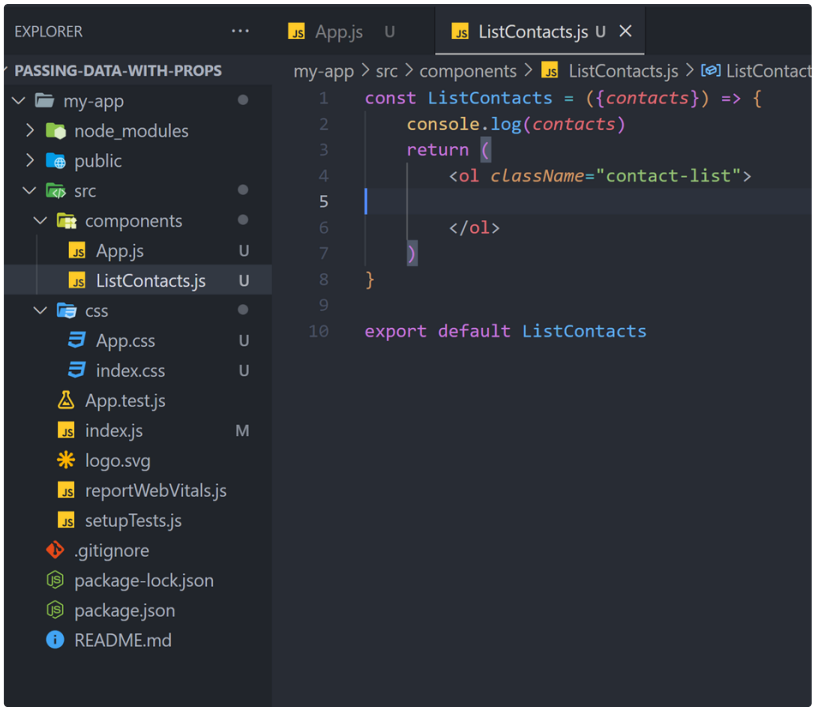
```
const contacts = [
  {
    id: "karen",
    name: "Karen Isgrigg",
    handle: "karen_isgrigg",
    avatarURL: "http://localhost:5001/karen.jpg",
  },
  {
    id: "richard",
    name: "Richard Kalehoff",
    handle: "richardkalehoff",
    avatarURL: "http://localhost:5001/richard.jpg",
  },
  {
    id: "tyler",
    name: "Tyler McGinnis",
    handle: "tylermcginnis",
```

```
    avatarURL: "http://localhost:5001/tyler.jpg",
  },
];

```

This `contacts` array is just temporary. Eventually, we'll be retrieving and storing contacts on our backend server. As of right now, though, we don't know how or where to make network requests. We'll get to this soon, so we'll just stick with this static list of contacts for now.

Now that we can the `contacts` array, let's add these contacts to our app! We'll do so by first adding it to our `App` component, then passing it down to a new component, `ListContacts`, as a prop.



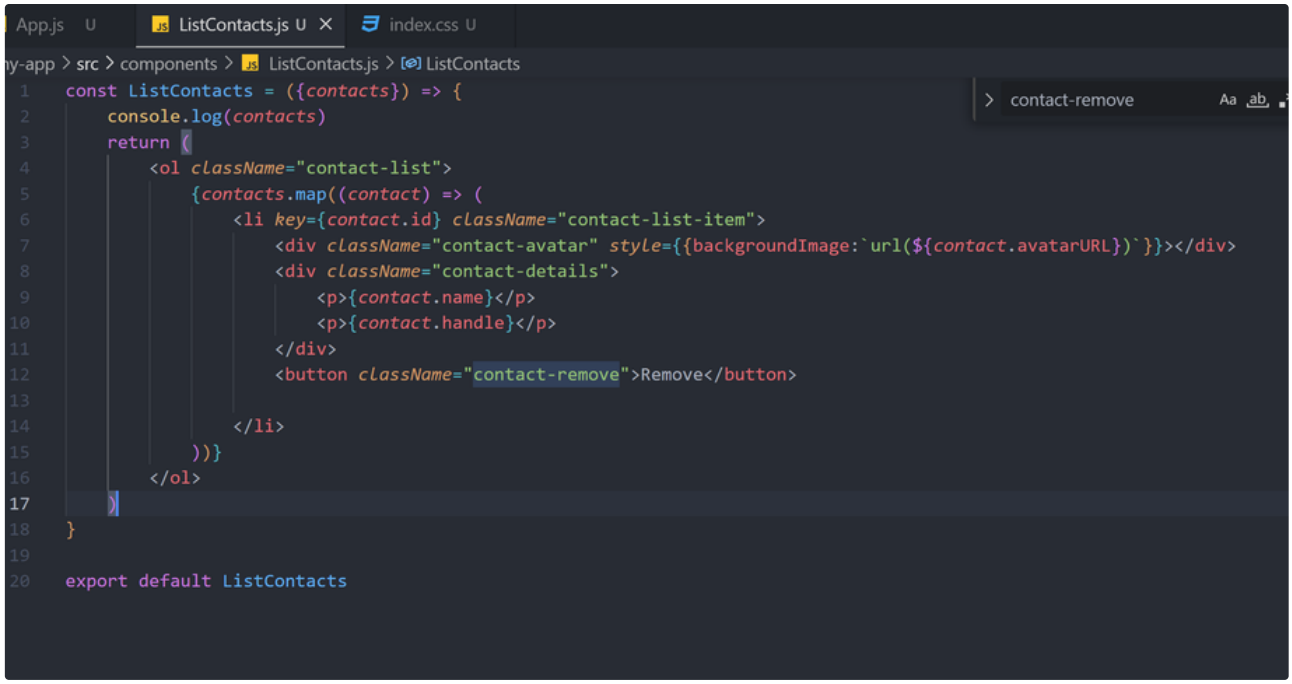


Rendering Contacts

At this point in our application, `ListContacts` has access to the `contacts` array passed down to it by its parent component, `App`. Next, we'll start building out the logic to render that data in the `ListContacts` component. Let's see it in action!

Rendering More Information

Beyond each contact's `name`, we know know that we have access to additional information, such as the contact's `handle`, as well as its `avatarURL`. Let's make use of this data in our application, as well as add some styling to the page.



Recap

A `prop` is any input that you pass to a React component. Just like an HTML attribute, a `prop` name and value are added to the component.

```
// passing a prop to a component

<LogoutButton text='Wanna log out?' />

```

In the code above, `text` is the `prop` and the string `'Wanna log out?'` is the value.

All props are stored on the `props` object. So to access this `text` `prop` from *inside* the component, we'd use `props.text`:

```
// access the prop inside the component

const App = (props) => {
  return <div>{props.text}</div>;
};

```

Alternatively, you can use object destructuring as a shorthand without writing `props` directly. The following code produces the same result as the code snippet above:

```
const App = ({ text }) => {
  return <div>{text}</div>;
};

```

Further Research

- Components and Props from the React Docs

Add State to a Component

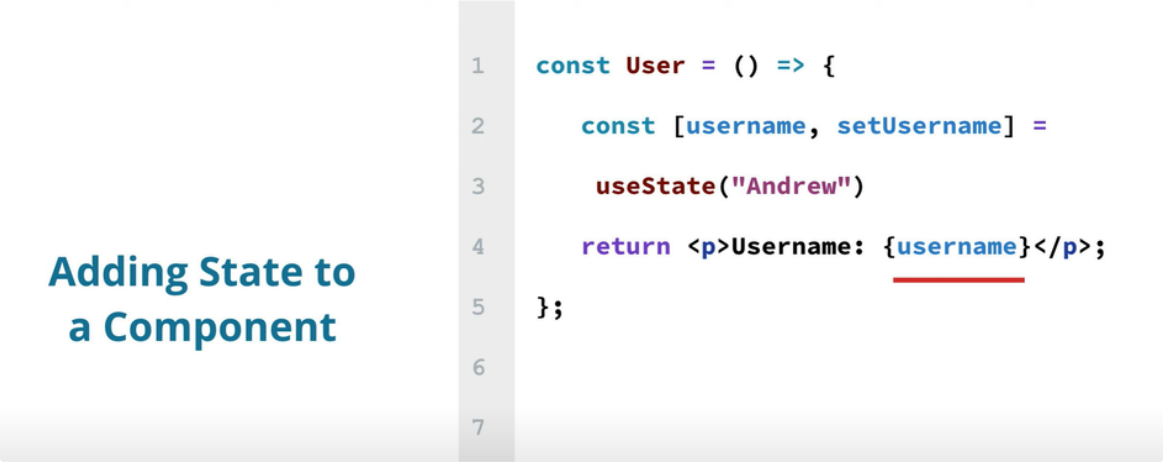
State

Earlier in this Lesson, we learned that `props` refer to attributes from parent components. In the end, props represent "read-only" data that are *immutable*.

A component's `state`, on the other hand, represents *mutable* data that ultimately affects what is rendered on the page. State is managed internally by the component itself and is meant to change over time, commonly due to user input (e.g., clicking on a button on the page).

In this section, we'll see how we can encapsulate the complexity of state management to individual components.

Add State to a Component



Moving Contacts into State

For the next portion of our app, we'll move our static `contacts` array into state. We'll first `import { useState } from "react";`, then:

- Create a variable to hold the current state (for a specific piece of state)
- Create a function to update that piece of state
- Note the *initial value* for that piece of state

> ⚠️ Props in Initial State ⚠️

> When defining a component's initial state, avoid initializing that state with `props`. This is an error-prone *anti-pattern*, since state will only be initialized with `props` when the component is first created.

> `const [user, setUser] = useState(props.user);`

> In the above example, if `props` are ever updated, the current state will not change unless the component is "refreshed." Using `props` to produce a component's initial state also leads to duplication of data, deviating from a dependable "source of truth."

Recap

By having a component manage its own state, any time there are changes made to that state, React will know and *automatically* make the necessary updates to the page.

This is one of the key benefits of using React to build UI components: when it comes to re-rendering the page, we just have to think about updating state. We don't have to keep track of exactly which parts of the page change each time there are updates. We don't need to decide how we will efficiently re-render the page. React compares the previous output and new output, determines what has changed, and makes these decisions for us. This process of determining what has changed in the previous and new outputs is called Reconciliation.

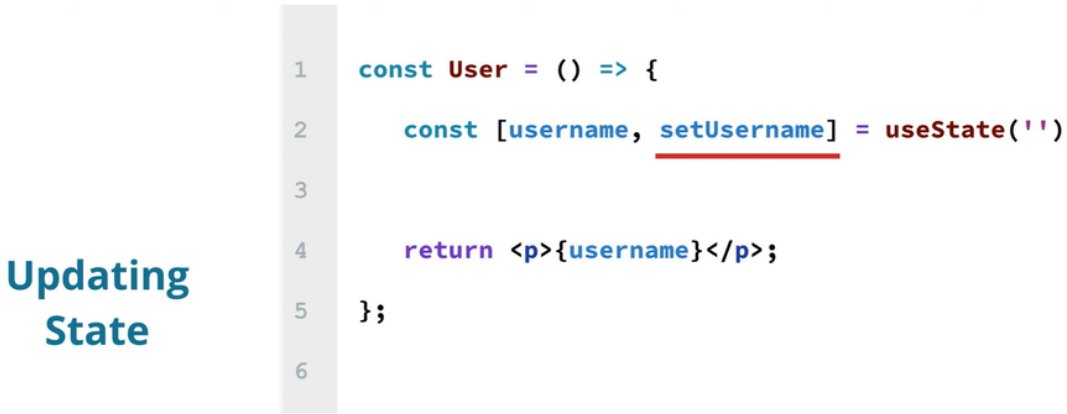
Further Research

- Identify Where Your State Should Live
- Reconciliation from the React documentation

Update State with useState()

Update State with `useState()`

Now that React is managing our list of contacts through component state, how do we go upon *updating* data in that state?



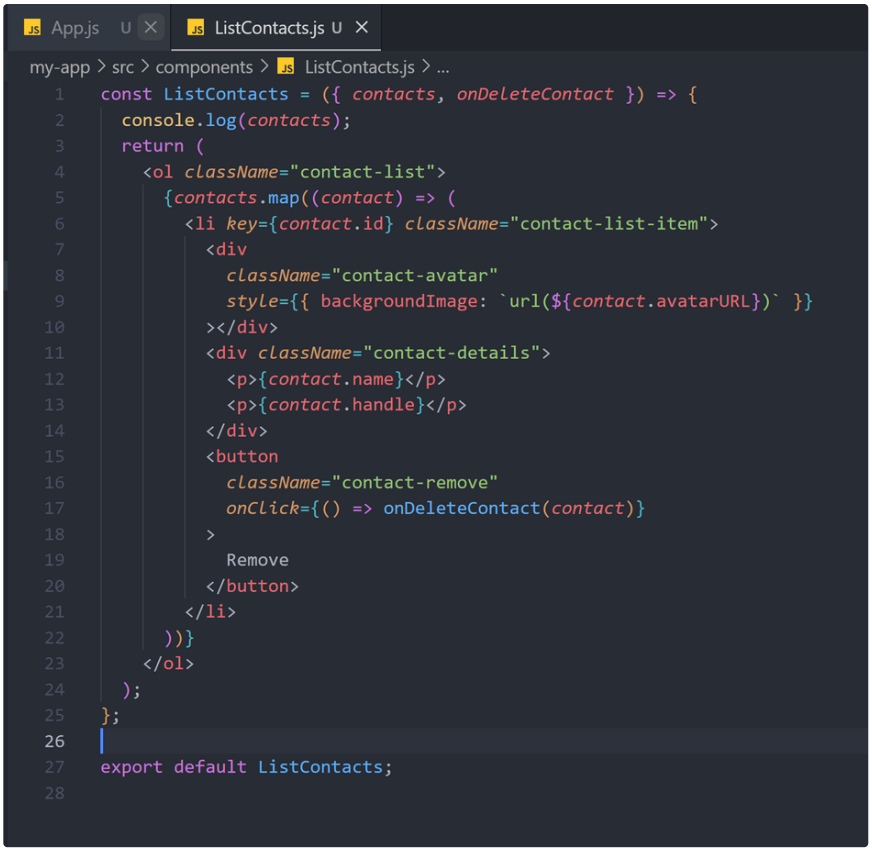
Deleting a Contact

Now that we know what function we can leverage to update state, let's go ahead and implement a new feature into our application: deleting a contact. We'll create a button that, when pressed, invokes a function that updates the `contacts` state inside the `App` component.

App.js U ListContacts.js U

my-app > src > components > App.js > App

```
// import logo from './logo.svg';
import { useState } from 'react';
import '../css/App.css';
import ListContacts from './ListContacts';

function App() {
  const removeContact = (contact) => {
    setContacts(contacts.filter(c => c.id !== contact.id))
  }

  const [contacts, setContacts] = useState([
    {
      id: "karen",
      name: "Karen Isgrigg",
      handle: "karen_isgrigg",
      avatarURL: "http://localhost:5001/karen.jpg",
    },
    {
      id: "richard",
      name: "Richard Kalehoff",
      handle: "richardkalehoff",
      avatarURL: "http://localhost:5001/richard.jpg",
    },
    {
      id: "tyler",
      name: "Tyler McGinnis",
      handle: "tylermcginnis",
      avatarURL: "http://localhost:5001/tyler.jpg",
    },
  ])
  return (
    <div >
      <ListContacts contacts={contacts} onDeleteContact = {removeContact}/>
    </div>
  );
}
export default App;
```



How State is Set

Earlier in this lesson, we saw how we can define a component's state at the time of initialization. Since state reflects *mutable* information that ultimately affects rendered output, a component may also update its state throughout its lifecycle using a function. As we've learned, when local state changes, React will trigger a re-render of the component.

Let's recap each step of the previous video to track how state is changed due to user input in our application. First, the user clicks on the "Remove" button in the `ListContacts` component:

```
<button className="contact-remove" onClick={() => onDeleteContact(contact)}>
  Remove
</button>;

```

The button listens for an `onClick` event, and since that had just occurred, it invokes a function which calls `onDeleteContact()`, passed down as a prop from the parent `App` component. Note that the contact to-be-removed is passed into this function as well.

Back up in the `App` component, note that the value of the `onDeleteContact` prop is the `removeContact()` method:

```
const removeContact = (contact) => {
  setContacts(contacts.filter((c) => c.id !== contact.id));
};

```

Recall that from Lesson 1, we know that `filter()` is called on an array, and returns a *new* array. As such, `contacts.filter((c) => c.id !== contact.id)` returns a new array where the contact to-be-removed is "filtered out." This is the new array that we now want in our `contacts` state, and we simply pass that new array into `setContacts()` so that React can do that for us.

This is all possible because within the `App` component, we've already defined a variable for the current state (i.e. `contacts`), a function to update that piece of state (i.e., `setContacts`), and an initial value for that state, which is the entire contacts array passed into `useState()`:

```
const [contacts, setContacts] = useState([
  {
    id: "tyler",
    name: "Tyler McGinnis",
    handle: "@tylermcginnis",
    avatarURL: "http://localhost:5001/tyler.jpg",
  },
  {
    id: "karen",
    name: "Karen Isgrigg",
    handle: "@karen_isgrigg",
    avatarURL: "http://localhost:5001/karen.jpg",
  },
  {
    id: "richard",
    name: "Richard Kalehoff",
    handle: "@richardkalehoff",
    avatarURL: "http://localhost:5001/richard.jpg",
  },
]);

```

And since state has changed, that's how a simple click of a button leads to React re-rendering the page for us -- effectively removing the contact from the page! You'll always want to use the appropriate function to update state. If you, say, updated the `contacts` variable directly, React would not know about those changes, and your application will be out of sync.

Recap

While a component can set its state when it initializes, we expect that state to change over time, usually due to user input. The component is able to change its own internal state using a function. Each time state is changed, React knows and will re-render the component. This allows for fast, efficient updates to your application's user interface.

Further Research

- Using the State Hook from the React documentation
- Using State Correctly from the React documentation

Type checking with PropTypes

Type checking a Component's Props with `PropTypes`

As we implement additional features into our app, we may soon find ourselves debugging our components more frequently. For example, what if the `props` that we pass to our components end up being an unintended data type (e.g., an object instead of an array)? `PropTypes` is a package that lets us define the data type we want to see right from the get-go, and warns us during development if the prop that's passed to the component doesn't match what is expected.

To use PropTypes in our app, we need to install prop-types:

```
npm install --save prop-types

```

Alternatively, if you have been using yarn to manage packages, feel free to use it as well to install:

```
yarn add prop-types

```

Recap

All in all, PropTypes is a great way to validate intended data types in our React app. Typechecking our data with PropTypes helps us identify these bugs during development to ensure a smooth experience for our app's users.

Further Research

- prop-types library from npm
- Typechecking With Proptypes from the React documentation

Building Forms with Controlled Components

Intro to Controlled Components

As we've seen, React is all about state management! We can extend that very pattern to *forms* for user input as well.

Adding a Search Field to `ListContacts`

Rather than having the users of our application scroll down an increasingly large list of contacts, we can improve user experience by implementing a search (or filter) functionality. Let's see just how we can do that!

ListContacts.js U App.js U

my-app > src > components > ListContacts.js > ...

```
import PropTypes from "prop-types";
import { useState } from 'react';

const ListContacts = ({ contacts, onDeleteContact }) => {
  const [query,setQuery] = useState("")
  const updateQuery = (query) => {
    setQuery(query.trim())
  }
  console.log("value", query);
  return (
    <div className="list-contacts">
      <div className="list-contacts-top">
        <input
          className="search-contacts"
          type="text"
          placeholder="Search Contacts"
          value={query}
          onChange={(event) =>updateQuery(event.target.value)}
        />
      </div>

      <ol className="contact-list">
        {contacts.map((contact) => (
          <li key={contact.id} className="contact-list-item">
            <div
              className="contact-avatar"
              style={{ backgroundImage: `url(${contact.avatarURL})` }}
            ></div>
            <div className="contact-details">
              <p>{contact.name}</p>
              <p>{contact.handle}</p>
            </div>
            <button
              className="contact-remove"
              onClick={() => onDeleteContact(contact)}
            >
              Remove
            </button>
```

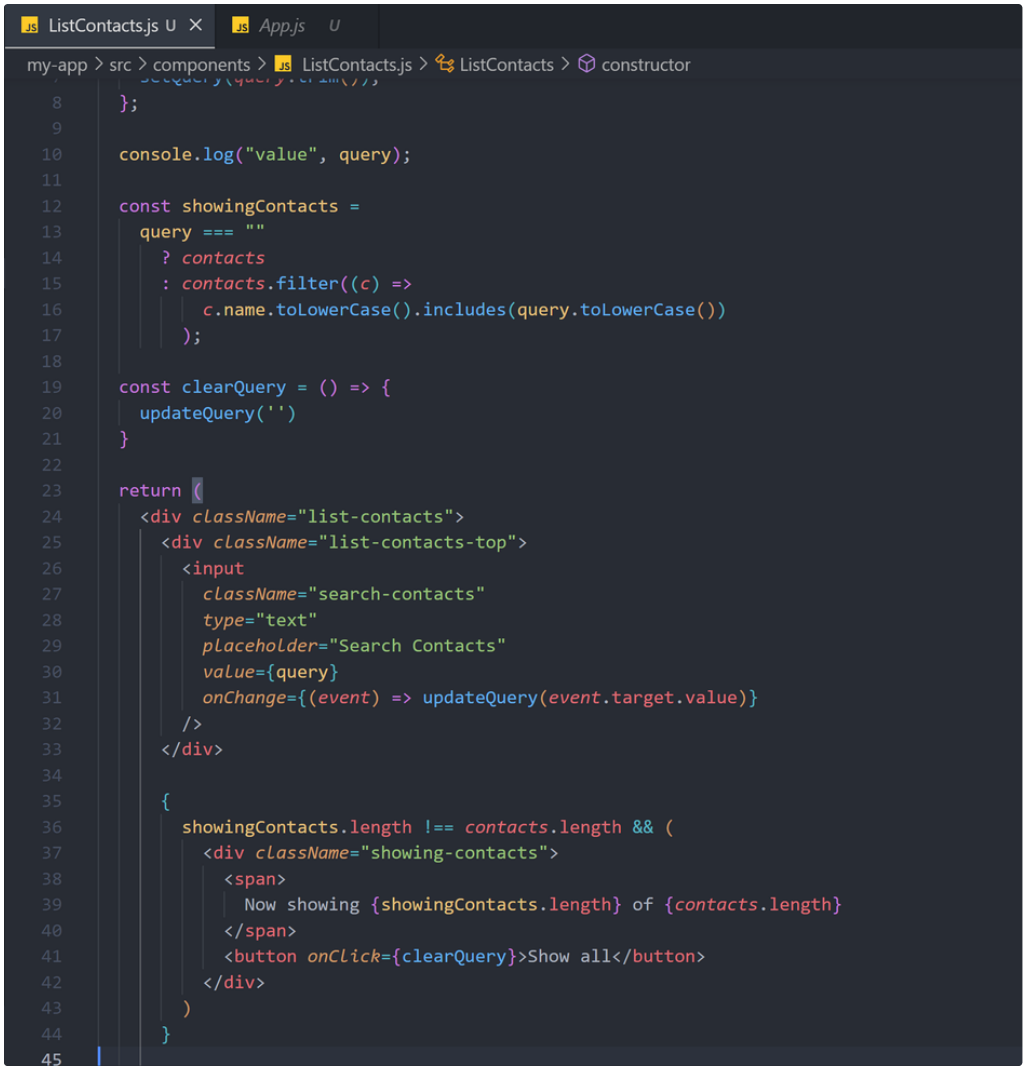
Note that the `value` attribute is set on the `<input>` element. Since the displayed value will always be the value in the component's state, we can treat state, then, as the "single source of truth" for the form's state.

To recap how user input affects the `ListContacts` component's own state:

1. The user enters text into the input field.
2. The `onChange` event listener invokes the `updateQuery()` function.
3. `updateQuery()` then calls `setQuery()` with the user's search query as the argument.
4. Because its state has changed, the `ListContacts` component re-renders.

Showing the Displayed Contacts Count

We're almost done working with the controlled component! Our last step is to make our app display the count of how many contacts are being displayed out of the overall total.



Do you feel comfortable with controlled components? If not, check out the documentation to see another example. We'll get some practice with controlled components shortly.

Recap

Controlled Components refer to components that render a form, but the "source of truth" for that form state lives inside of the component state rather than inside of the DOM. The benefits of Controlled Components are:

- Instant input validation
- Conditionally enable or disable buttons
- Enforce input formats

In our `ListContacts` component, not only does the component render a form, but it also controls what happens in that form based on user input. In this case, event handlers update the component's state with the user's search query. And as we've learned: any changes to React state will cause a re-render on the page, effectively displaying our live search results.

Further Research

- react-devtools on npm
- Forms in the React documentation

(Doc) Week 5:

React: Fundamentals 02

Hooks

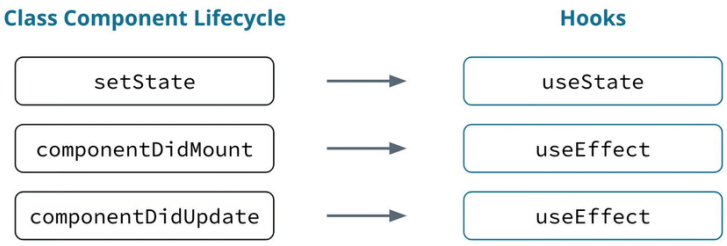
Overview of Hooks

Introducing Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Why Hooks?

Whether you're new to React, or had even written React in the past (prior to the release of version 16.8), the concept of **hooks** in React may sound like a popular, yet mysterious feature nowadays. What exactly are they, and why are they necessary when writing functional components in React? Let's take a closer look.



With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. **Hooks allow you to reuse stateful logic without changing your component hierarchy.** This makes it easy to share Hooks among many components or with the community.

Lifecycle and State in Class Components

*Note: This section is for-your-information only; it is not required for you to use this syntax (in fact, you should **not** use it) when building the applications in this course.*

The common way to write components prior to React v16.8 was largely via writing a *class*. While it may have been less intuitive than simply writing a *function*, it did provide some useful features.

First, classes had access to component state *directly*. This meant that there was no need to import a `useState` hook; we could just define our entire state as an object, like so:

```
class Contact extends React.Component {
  state = {
    clicked: true,
  };

  // ...
}

```

And to set state, we could call `this.setState()` and pass in a new state object, like:

```
this.setState({ click: false });

```

Along with having direct access to component state, there were also lifecycle methods we could leverage. These methods were automatically bound to the component instance, and React would call these methods naturally at certain times during the life of a component. There were a number of different lifecycle events, but here were some of the more commonly used methods and their associated events:

- `componentDidMount()`, invoked immediately *after* the component is *inserted* into the DOM
- `componentWillUnmount()`, invoked immediately *before* a component is *removed* from the DOM

- `getDerivedStateFromProps()`, invoked after a component is instantiated as well as when it receives brand new props

To use one of these, you would just add the method in your component, and React would call it automatically as needed. It was an easy way to run some code during different and specific parts of the lifecycle of React components.

Now, this is all fine for *class* components (which are actually backwards-compatible and can still be used today, despite the past tense in the previous paragraphs), but how would we be able to use these features in the modern *functional* components that we've written? This is where hooks come in.

Hooks Allow for Lifecycle and State in Functional Components

This is the main takeaway for this section of the course. By using hooks, we can access state and other React features (e.g., lifecycle events) in our functional components.

We encourage you to check out the resources below for more information about the motivation and introduction of hooks in React.

Further Research

- Motivation (for Hooks) in the React documentation
- Hooks at a Glance in the React documentation
- Hooks FAQ in the React documentation
- React v16.8: The One With Hooks in the React blog

Perform Side Effects with useEffect

Intro to Side Effects

What are side effects? Why would we even want to include them in the applications we build? Well, to begin, here's how React's documentation describes side effects:

> Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects. Whether or not you're used to calling these operations "side effects" (or just "effects"), you've likely performed them in your components before.

In other words, you can think of many side effects as something "outside" the scope of how a component normally runs. Whether it's data coming in from an asynchronous HTTP call, or some special function that's called during a specific stage of the component's life (e.g., when it's mounted to the DOM), these operations can greatly enrich your users' experience as they use your application.

The `useEffect` Hook

While we know that functional components don't have access to the lifecycle methods we read about earlier in this lesson, we can still use a special hook that allows us to implement side effects in components: `useEffect`.

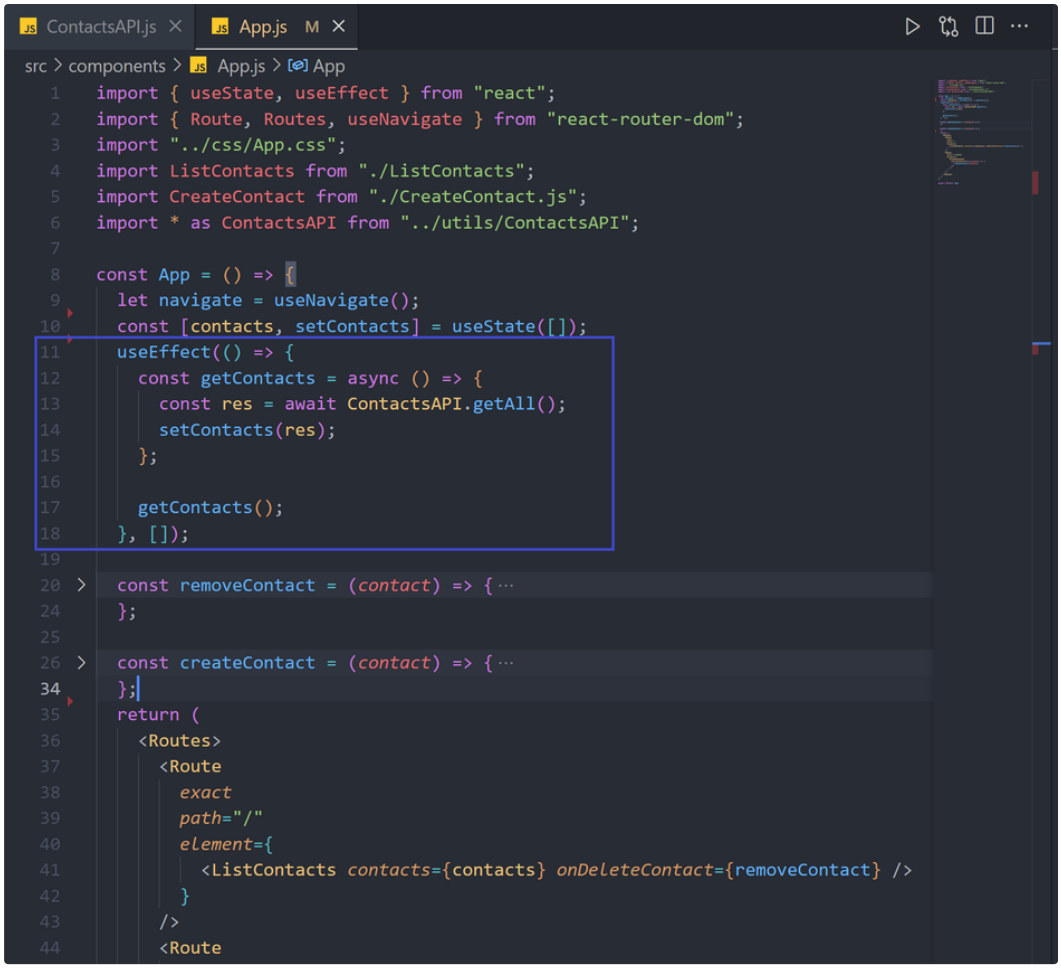
This hook allows us to run special code or custom logic at specific points of a component's lifecycle, including after the component is mounted to the DOM, after the component is updated, and even before the component is destroyed (i.e., unmounted from the DOM).

To use `useEffect` in a component, we'll first import it from React:

```
1 import { useEffect } from "react";
```

Let's see it all in the below code:

ContactsAPI.js file



App.js file

Recap of the Changes Made

After implementing the changes in the above example, our `App` component now looks something like this:

```
// ...

import { useState, useEffect } from "react";

const App = () => {
  const [contacts, setContacts] = useState([]);

  useEffect(() => {
    const getContacts = async () => {
      const res = await ContactsAPI.getAll();
      setContacts(res);
    };

    getContacts();
  }, []);

  // ...
};

```

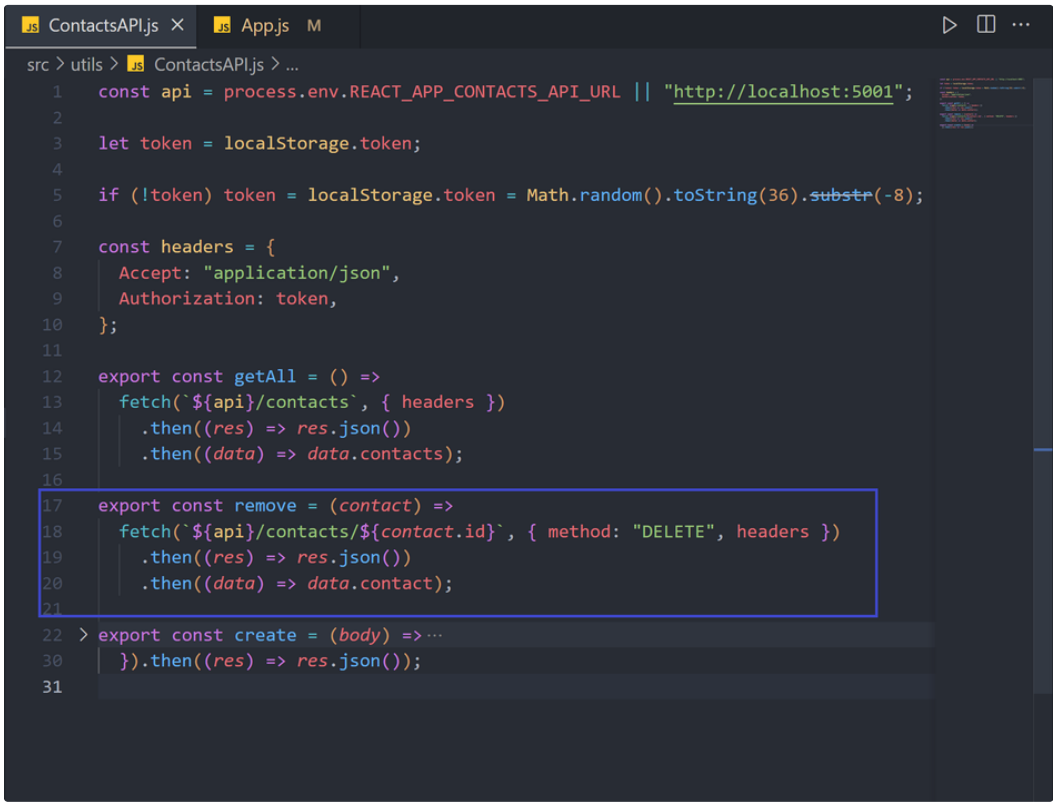
As a result, when the component is mounted to the DOM, it fetches contacts from a server to display on the screen. Let's break down how we got there:

1. To be able to use the hook in the first place, we import `useEffect`.
2. We then place `useEffect()` directly inside the component.
   a. The first argument is a function. Within it, we make an asynchronous request to our Contacts API. When it resolves, we pass the response into `setContacts()`, which updates our `contacts` state.
   b. The second argument is an *empty array*. We include this because we want the effect to run *only* during mount and unmount (i.e., not after every time props or state changes). The empty array also tells React that there are no dependencies needed.
3. Once our component is mounted to the DOM, the `contacts` array is populated. As such, React re-renders, and our contacts are shown on the screen.

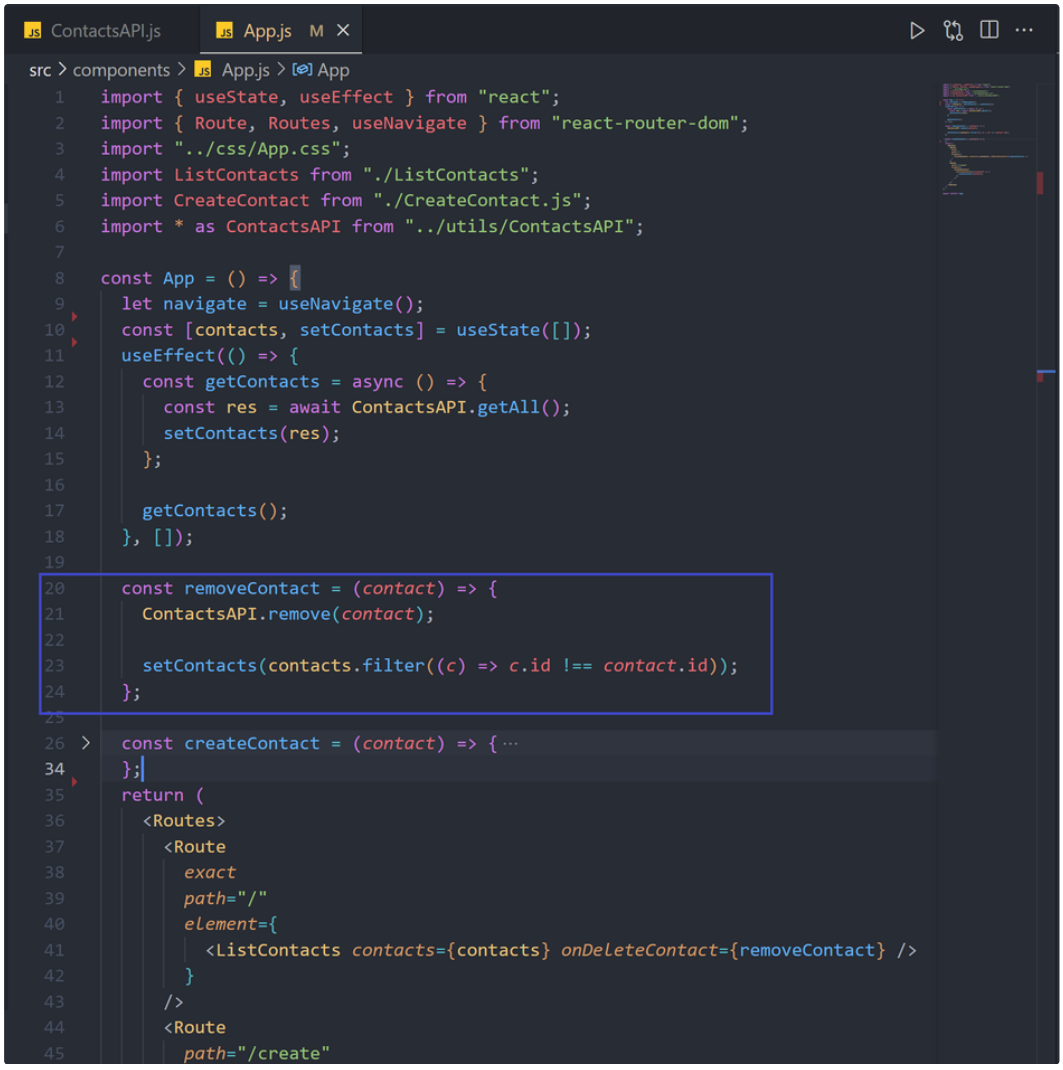
Removing Contacts from the Database

Currently, our application only removes the contact on the front end. If users refresh the page, the contact reappears. Let's make sure requests to remove contacts are also reflected in our backend server.

Let's see it all in the below code:



ContactsAPI.js file



App.js file

Recap

The `useEffect` hook is versatile, and mimics the lifecycle methods that React developers typically leverage in their applications. It's a great way to run custom functions or logic at specific points of a component's lifecycle, especially at the time it's mounted to the DOM.

Further Research

- Using the Effect Hook in the React documentation
- Understanding useEffect: the dependency array

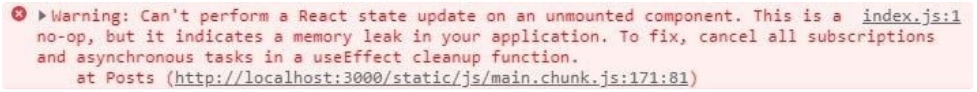
Side Effect Cleanup

In react, we use *useEffec*t when we need to do something after a component renders or when it needs to cause side effects. A side effect may be fetching data from a remote server, reading from or writing to local storage, setting event listeners, or setting up a subscription.

useEffect() allows us to manage component life-cycles within functional components. The *useEffect()* hook can be thought of as a combination of *componentDidMount*, *componentDidUpdate*, and *componentWillUnmount* combined.

However, sometimes we may encounter challenges at the junction of the component lifecycle and the side-effect lifecycle (start, in progress, complete).

When a side-effect completes, it attempts to update the state of a component that has already been unmounted. As a result, a React warning appears:



Memory leak warning

Cleanup Prevents Memory Leaks

As we just learned one of the main reasons to clean up side effects is to prevent memory leaks. In the above example, during the re-render, `useEffect` will try updating the state on that unmounted component. And as a result, we'll see a warning in the console:

```
Warning: Can't perform a React state update on an unmounted component.
This is a no-op, but it indicates a memory leak in your application.
To fix, cancel all subscriptions and asynchronous tasks in a useEffect cleanup function.

```

This is where we can leverage a cleanup function, which allows us to cancel asynchronous calls, unsubscribe from any subscriptions, or otherwise stop any unnecessary operations. In turn, this helps prevent memory leaks and performance issues in our application.

When is Cleanup Necessary?

One of the most common times you'll need cleanup is when your application is subscribing to data from an external source. In such cases, when the component is removed from the DOM, or perhaps even when the component is simply updated (e.g., when props or state change), you'll also want to make sure that subscription logic is removed.

For the most part, cleanup is *not* necessary when making simple network requests, such as fetching data from a server to display on the page. The React documentation even lists "network requests" under Effects Without Cleanup. However, while it's optional, it's a good practice to make sure your side effects are still properly cleaned up in cases when the component is updated (or unmounted from the DOM) but your asynchronous calls have not yet resolved. Along with preventing memory leaks, it's a great way to prevent race conditions as well.

Now, the question is: where in a component does this cleanup even take place? As it turns out, it happens right where the side effect is implemented in the first place: within `useEffect()`.

Returning a Cleanup Function

One of the best features of the `useEffect` hook is how versatile it is. That is, when you want to perform side effect cleanup, there's no additional hook or external dependency that you'll have to import. All you have to do is *return* a function from within your effect (i.e., the function passed into `useEffect()`). Consider the following example of a simple `Counter` component:

```
import { useEffect, useState } from "react";

const Counter = () => {
  const [count, setCount] = useState(0);
```

```
  useEffect(() => {
    console.log("This is the side effect.");
    return () => {
      console.log(
        "The component re-rendered. This is part of the cleanup before the next effect."
      );
    };
  });

  return (
    <div>
      <p>The current count is: {count}</p>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Increase the Count
      </button>
    </div>
  );
};

export default Counter;

```

In the example above, when the component mounts, `"This is the side effect."` will be printed to the console (as a result of our `useEffect` hook). Based on what we've learned in the previous **Perform Side Effects with useEffect** page, this is expected. Nothing surprising here.

However, what happens when the user clicks the button to increase the count? `setCount()` will update our `count` state. As we know, since state has updated, React will re-render our component. Since we're returning a function from `useEffect()`, React will perform any logic within it as "cleanup" of our side effect. As such, we'll see the following printed to the console:

```
"The component re-rendered. This is part of the cleanup before the next effect."

```

Since the side effect itself and its cleanup are still "part of the same effect," it makes sense that `useEffect()` can return a function with logic to clean up after that very effect.

At this point, we know *where* the cleanup takes place. However, *how do we actually perform that cleanup*?

Performing the Cleanup

Let's take a look at a real example where cleanup occurs in our Contacts app: within the ImageInput component. Consider the following snippet:

```
const ImageInput = ({ maxHeight, className, name }) => {
  // ...

  let fileInput;

  const [value, setValue] = useState("");

  useEffect(() => {
    setCanvas(document.createElement("canvas"));
    fileInput.form.addEventListener("reset", handleFormReset);
```

```
    return () => {
      if (fileInput) {
        fileInput.form.removeEventListener("reset", handleFormReset);
      }
    };
  }, [fileInput]);

  // ...
};

export default ImageInput;

```

When the component is mounted to the DOM, we create a `canvas` element and add an event listener to the `reset` element, with an associated `handleFormReset` function to handle that event:

```
setCanvas(document.createElement("canvas"));
fileInput.form.addEventListener("reset", handleFormReset);

```

This is great, but what happens if the `ImageInput` component is removed from the DOM? We don't want any lingering event listeners if they're not needed anymore. As such, we perform cleanup by returning a function that removes the event listener when the component is destroyed:

```
return () => {
  if (fileInput) {
    fileInput.form.removeEventListener("reset", handleFormReset);
  }
};

```

Overall, your cleanup logic will largely depend on whatever side effect or logic that you're looking to "undo." Think to yourself: is this operation (e.g., a subscription, an event listener, etc.) still necessary to carry out if the component no longer exists?

> 💡 `componentWillUnmount()` and `componentDidUpdate()` 💡
>
> If you've written React in the past, the cleanup within `useEffect` is similar to the logic you'd include in the componentWillUnmount and componentDidUpdate lifecycle methods when writing components as classes (rather than as functions).

Performing the Cleanup: Another Example

Let's revisit the warning from earlier in this section:

```
Warning: Can't perform a React state update on an unmounted component.
This is a no-op, but it indicates a memory leak in your application.
To fix, cancel all subscriptions and asynchronous tasks in a useEffect cleanup function.

```

You'll likely encounter this during your journey as a React developer, so let's consider how we can resolve this warning. That is, we want to prevent being able to set state on a component that doesn't exist on the DOM (i.e., the unmounted component).

Consider the following example `Login` component, and then we'll break it down together:

```
import { useEffect, useState } from "react";

const Login = () => {
```

```
  const [currentUser, setCurrentUser] = useState(null);

  // ...

  useEffect(() => {
    let mounted = true;

    if (user.exists) {
      if (mounted) {
        setCurrentUser(user);
      }

      // ...
    }

    return () => {
      mounted = false;
    };
  }, []);

  // ...
};

export default Login;

```

Now, in order to prevent state updates on an unmounted component, we can simply create a variable that *tracks* if a component is mounted in the first place. When the component is first mounted to the DOM, the value of the `mounted` we created is `true`. Under this condition (and only under this condition, we can run our side effects and custom logic, including calling `setCurrentUser()` to update state.

Then, when the component is unmounted, React will perform the cleanup that we built into the returned function. That is, we'll change the value of `mounted` to `false` -- effectively indicating that the `Login` component is no longer part of the DOM. As such, our function to update state (i.e., `setCurrentUser()`) will never be called on an unmounted component!

Recap

While side effect cleanup isn't always required, it's still a good practice to make sure any subscriptions, asynchronous calls, or DOM listeners (to name a few) are cleaned up appropriately. This is done in a returned function from within `useEffect()`. In turn, proper cleanup prevents memory leaks and race conditions, and provides a better overall experience for your users.

Further Research

- Avoid React state update warnings on unmounted components
- Effects Without Cleanup in the React documentation
- Effects with Cleanup in the React documentation

Using Additional Hooks

Additional Hooks

`useState` and `useEffect` are two powerful hooks that will help you build fully-featured React applications. You'll certainly want to get good practice with both before building the final project of this course.

Additionally, React also provides a collection of additional hooks that you can leverage as you build your applications. While they're not required in the scope of this course or any of the applications that we build here, some of these hooks still may interest you in your journey as a React developer:

- useContext
- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle
- useLayoutEffect
- useDebugValue

You will probably never need to use *all* hooks in any of the applications that you build, but feel free to check them out and see what may fit your needs as you develop more React applications. For more information, check out the Hooks API Reference in the React documentation.

Routing

In this lesson, we'll cover linking and routing in React projects including:

- **Single Page Applications**, which enable a more dynamic experience on the web
- **Dynamically Render Pages** by leveraging React state
- **Client-Side Routing with** `<BrowserRouter>`, allowing navigation without page reloads
- **Navigation with** `<Link>`**,** letting users to move across different pages in an app
- **Component Paths with** `<Route>`**,** which maps components to appropriate URL paths

Single Page Applications

Single Page Apps

Single page applications (or a *SPA*) can work in different ways. One way a single page app loads is by downloading *the entire* site's contents all at once. This way, when you're navigating around on the site, everything is already available to the browser, and it doesn't need to refresh the page. Another way single page apps work is by downloading everything that's needed to render the page the user requested. Then when the user navigates to a new page, asynchronous JavaScript requests are made for *just* the content that was requested.

Another key factor in a good single page app is that the URL controls the page content. Single page applications are highly interactive, and users want to be able to get back to a certain state using just the URL. Why is this important? Bookmarkability! (pretty sure that's not a word... yet) When you bookmark a site, that bookmark is *only* a URL, it doesn't record the state of that page.

Have you noticed that any of the actions you perform in the app do not update the page's URL? We need to create React applications that offer bookmarkable pages!

React Router

React Router turns React projects into single page applications. It does this by providing a number of specialized components that manage the creation of links, manage the app's URL, provide transitions when navigating between different URL locations, and so much more.

According to the React Router website:

> React Router is a collection of **navigational components** that compose declaratively with your application.

If you're interested, feel free to check out the official React Router website.

In the next section, we'll dynamically render content to the page based on a value in a component's state. We'll use this basic example as an idea of how React Router works by controlling what's being seen via state. Then we'll switch over to using React Router. We'll walk you through installing React Router, adding it to the project, and hooking everything together so it can manage your links and URLs.

Client-Side Routing with <BrowserRouter>

Install React Router

To use React Router in our app, we need to install react-router-dom.

```
npm install --save react-router-dom

```

In order for React Router to work properly, you need to wrap your entire app in a `BrowserRouter` component. Under the hood, `BrowserRouter` wraps the history library which makes it possible for your app to be made aware of changes in the URL.

Further Research

- history
- React Router on GitHub
- The React Router documentation
- `<BrowserRouter>` in the official documentation

Navigation with <Link>

React Router provides a `Link` component which allows you to add declarative, accessible navigation around your application. You'll use it in place of anchor tags (`<a>`) as you're typically used to. React Router's `<Link>` component is a great way to make navigation through your app accessible for users. Passing a `to` prop to your link, for example, helps guide your users to an absolute path (e.g., `/about`):

```
<Link to="/about">About</Link>

```

Since the `<Link>` component fully renders a proper anchor tag (`<a>`) with the appropriate `href`, you can expect it to behave how a normal link on the web behaves.

If you're experienced with routing on the web, you'll know that sometimes our links need to be a little more complex than just a string. For example, you can pass along query parameters or link to specific parts of a page. What if you wanted to pass state to the new route? To account for these scenarios, instead of passing a string to `Link`'s `to` prop, you can pass it an object like this:

```
<Link
  to={{
    pathname: "/courses",
    search: "?sort=name",
    hash: "#the-hash",
    state: { fromDashboard: true },
  }}
>
  Courses
</Link>;



```

You won't need to use this feature all of the time, but it's good to know it exists. You can read more information about `Link` in the official documentation.

Component Paths with <Route>

With a `Route` component, if you want to be able to pass props to a specific component that the router is going to render, you'll need to use `Route`'s `element` prop. As we just saw saw, `element` puts you in charge of rendering the component, which in turn allows you to pass any props to the rendered component as you'd like.

The `Route` component is a critical piece of building an application with React Router because it's the component which is going to decide which components are rendered based on the current URL path.

Here is the example:

```
export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="blogs" element={<Blogs />} />
          <Route path="contact" element={<Contact />} />
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}
```

Example Explained

We wrap our content first with `<BrowserRouter>`.

Then we define our `<Routes>`. An application can have multiple `<Routes>`. Our basic example only uses one.

`<Route>`s can be nested. The first `<Route>` has a path of `/` and renders the `Layout` component.

The nested `<Route>`s inherit and add to the parent route. So the `blogs` path is combined with the parent and becomes `/blogs`.

The `Home` component route does not have a path but has an `index` attribute. That specifies this route as the default route for the parent route, which is `/`.

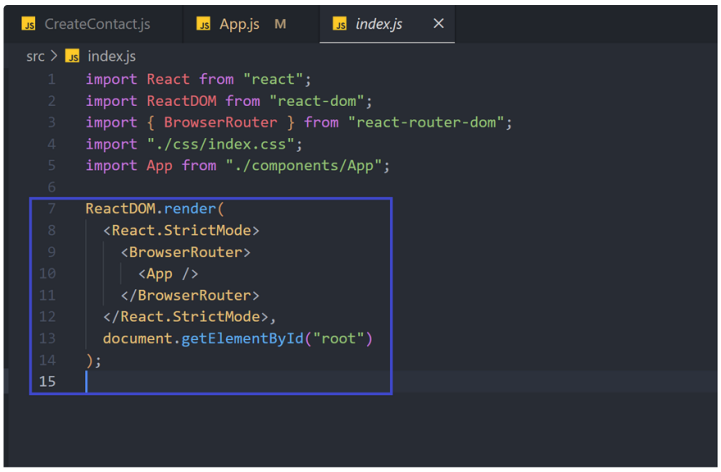
Setting the `path` to `*` will act as a catch-all for any undefined URLs. This is great for a 404 error page.

Further Research

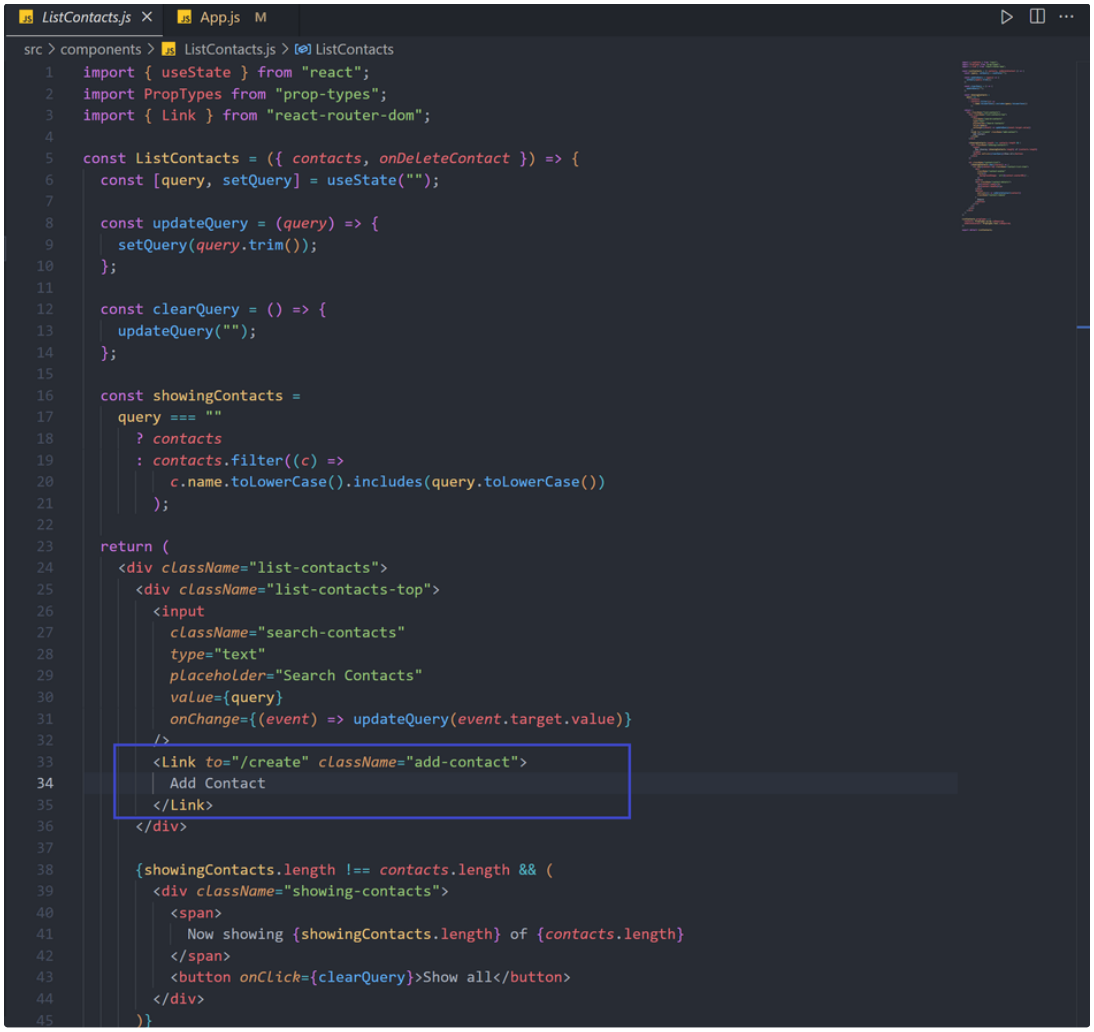
- `<Router>` in the official documentation

Finishing the Contact Form

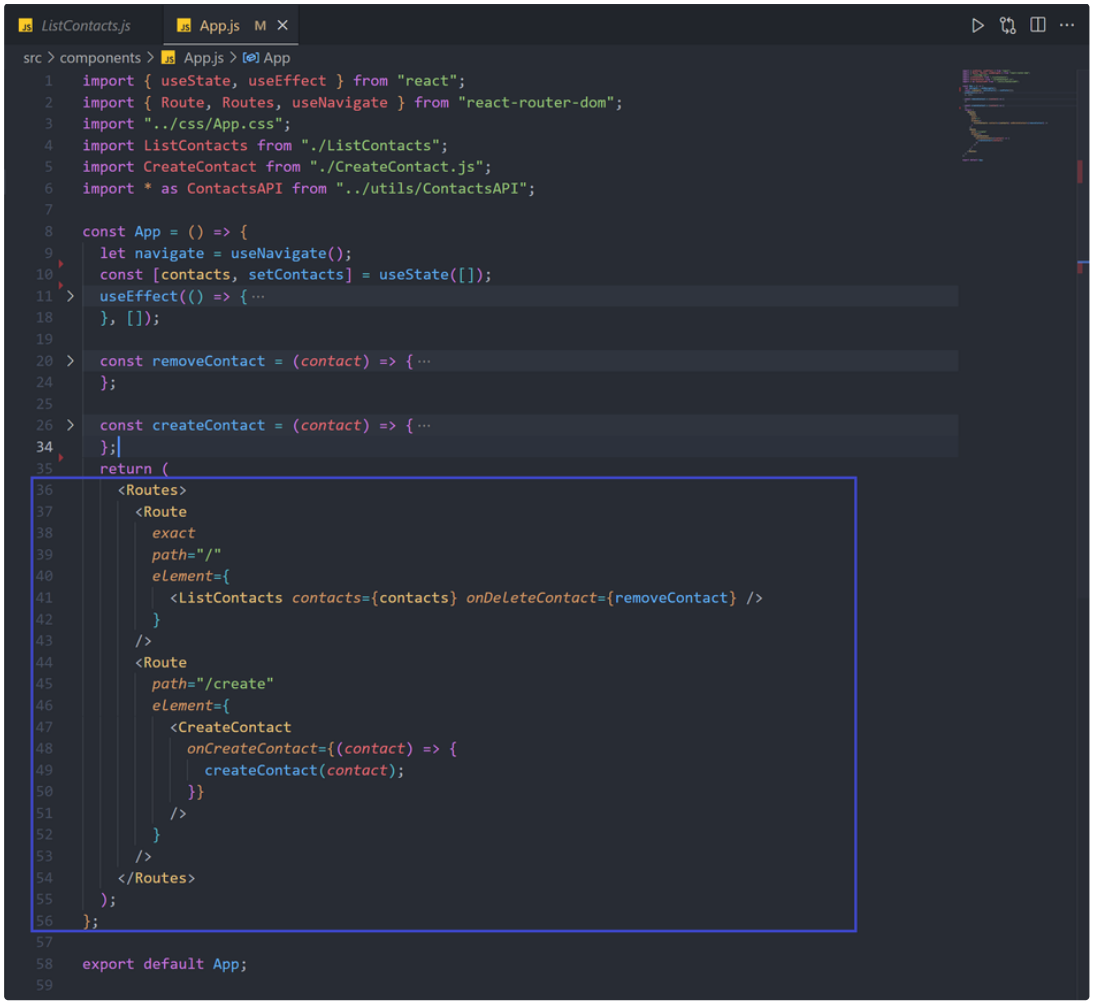
Client-Side Routing with <BrowserRouter>



Navigation with <Link>



Component Paths with <Route>



Create the Contact Form & Serialize the Form Data

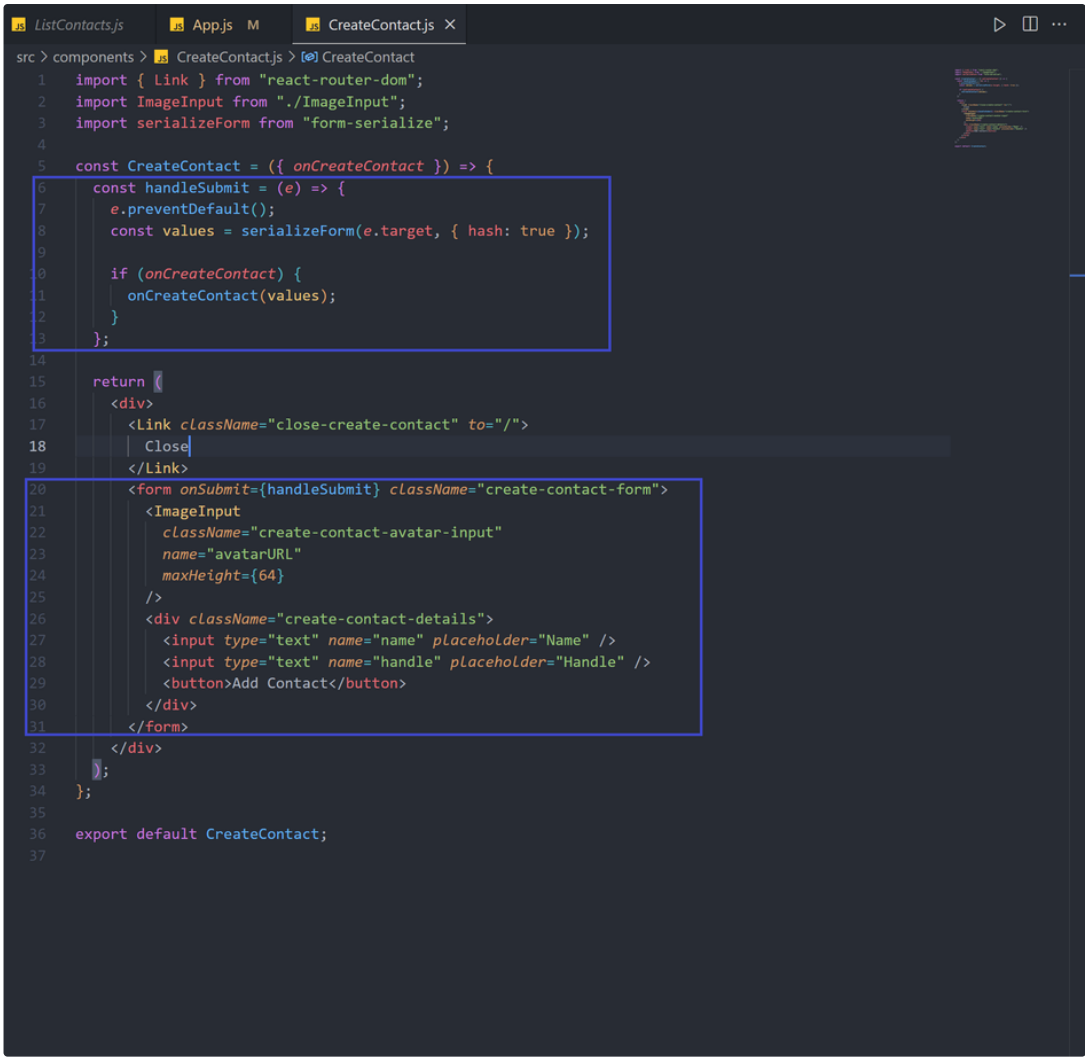
Right now, the page to create contacts is largely empty! Let's build out a form on that page so we start adding our own custom contacts.

The `ImageInput` component is a custom `<input>` that dynamically reads and resizes image files before submitting them to the server as data URLs. It also shows a preview of the image. We chose to give this component to you rather than build it ourselves because it contains features related to files and images on the web that aren't crucial to your education in this context. If you're curious, feel free to dive into the code, but know that it's not a requirement.

At this point, our form will serialize the values from user input (i.e., the `name` and `email`), adding them as a query string to the URL. We can add some additional functionality by having our app serialize these form fields on its own. After all, we want the app to ultimately handle creating the contact and saving it to the state.

To accomplish this, we'll use the form-serialize package to output this information as a regular JavaScript object for the app to use.

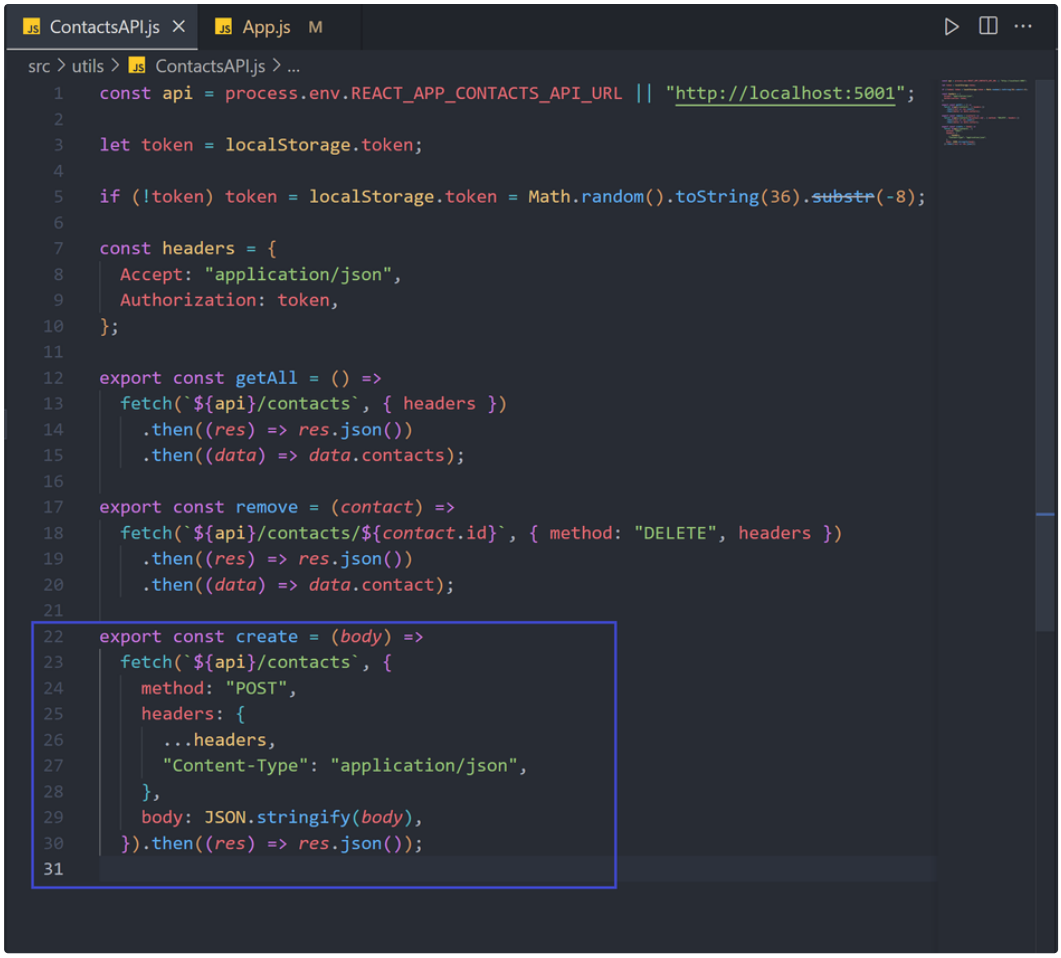
```
npm install --save form-serialize
```



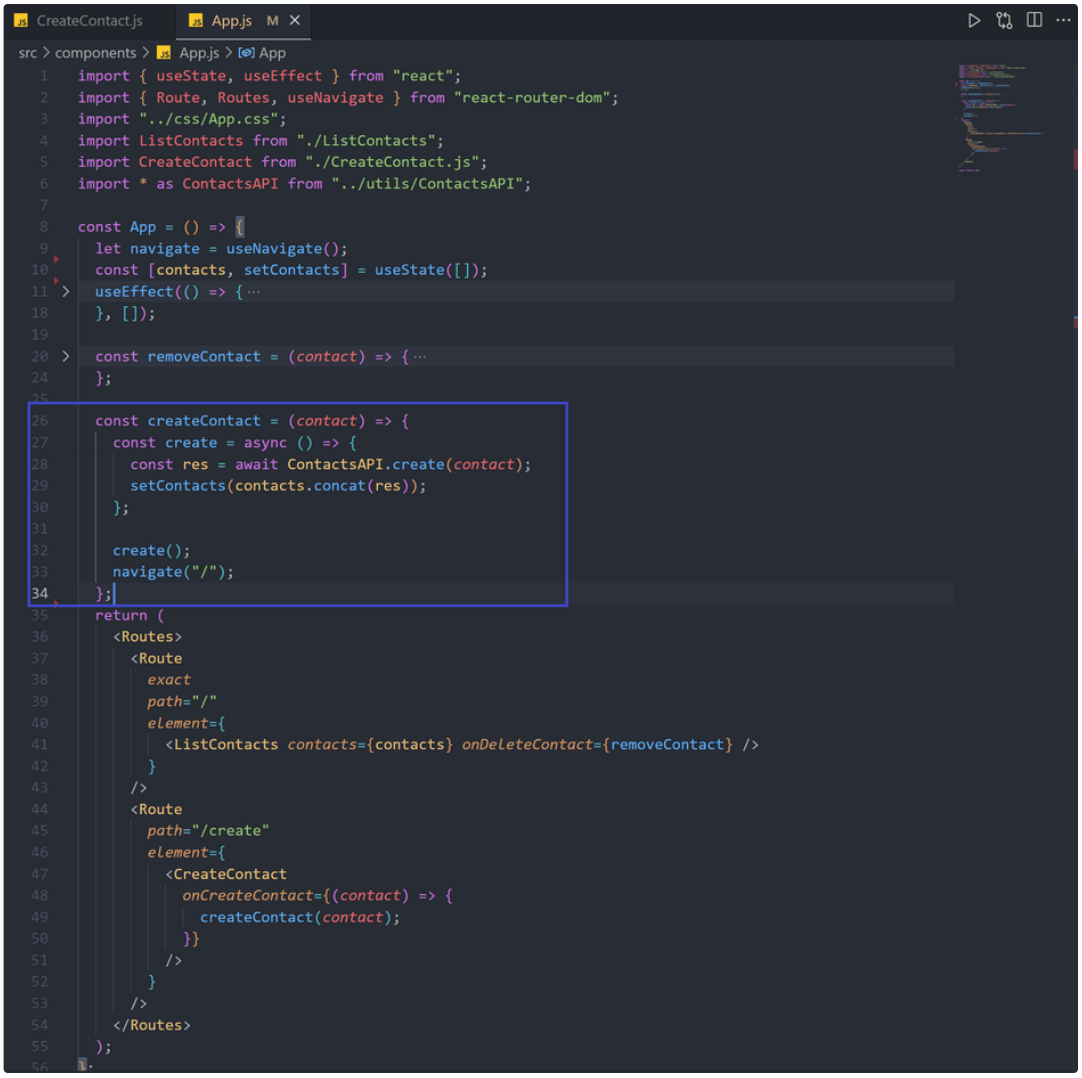
Update the Server with New Contacts

We have our contact form. We're serializing our data and passing it up to the parent component. We're almost there!

To have a fully functional app, all we need to do now is to save the contact to the server.



ContactsAPI.js file



App.js file

Further Research

- form-serialize on npm

Managing State

State management is essentially a way to facilitate communication and sharing of data across components. It creates a tangible data structure to represent the state of your app that you can read from and write to.

You'll learn techniques to make your state more predictable by moving your state to a central location and establishing strict rules for getting, listening, and updating that state.

In this lesson, we will learn about managing state through:

- Store
- Action
- Reducers

Predictable State Management

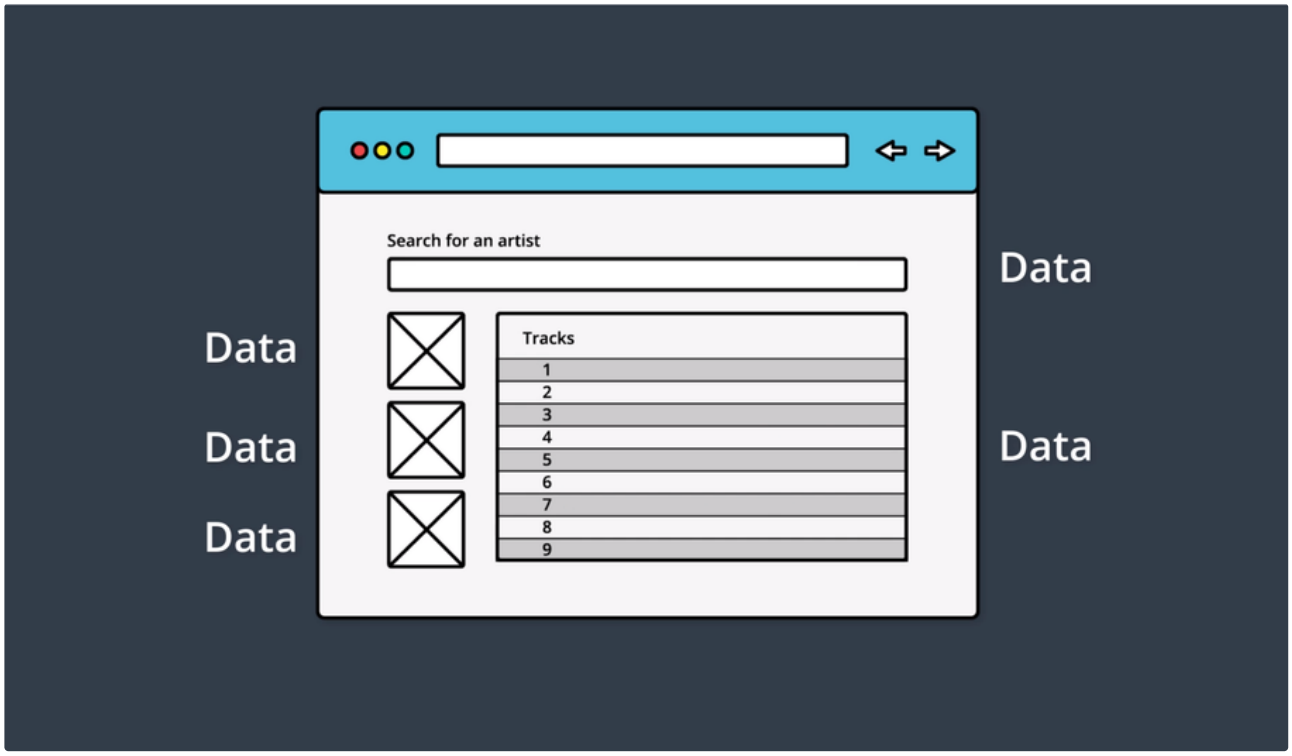
In managing State, We will focus on 3 keys:

- The Store
- State Trees
- State

Think about any application you've ever made?

→ Most likely app was composed of two things, UI and state.

A traditional app might look something like the image below:



The application's data is sprinkled throughout the app and this simple application has a lot of states:

- There are the images in the sidebar on the left
- There are rows of tracks in the main area
- Each Track will have its own information that it's maintaining
- There's the search field at the top that introduces new state to the app (the searched-for artist/track information)

And this is just one, simple page of this application. In most sites you use, there is information littered throughout every single page of the entire app.

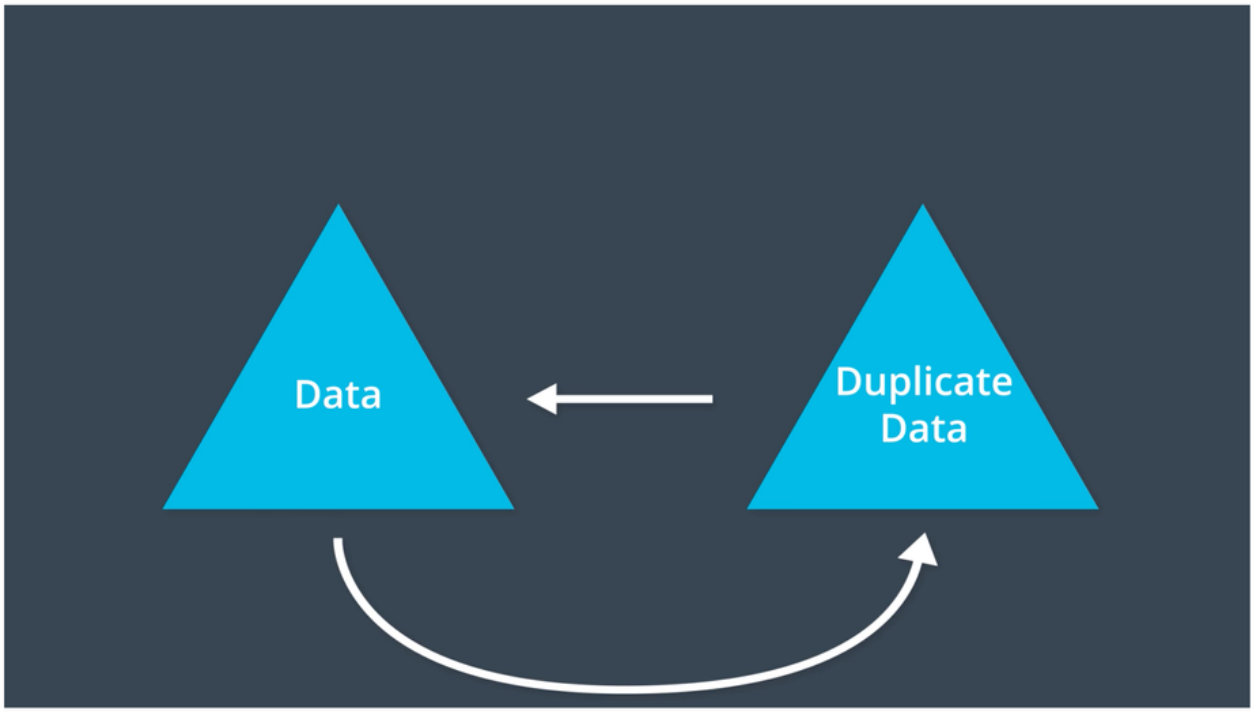
Remember that the main goal of Redux is to make the state management of an application more predictable.

Let's see what that might be look like:

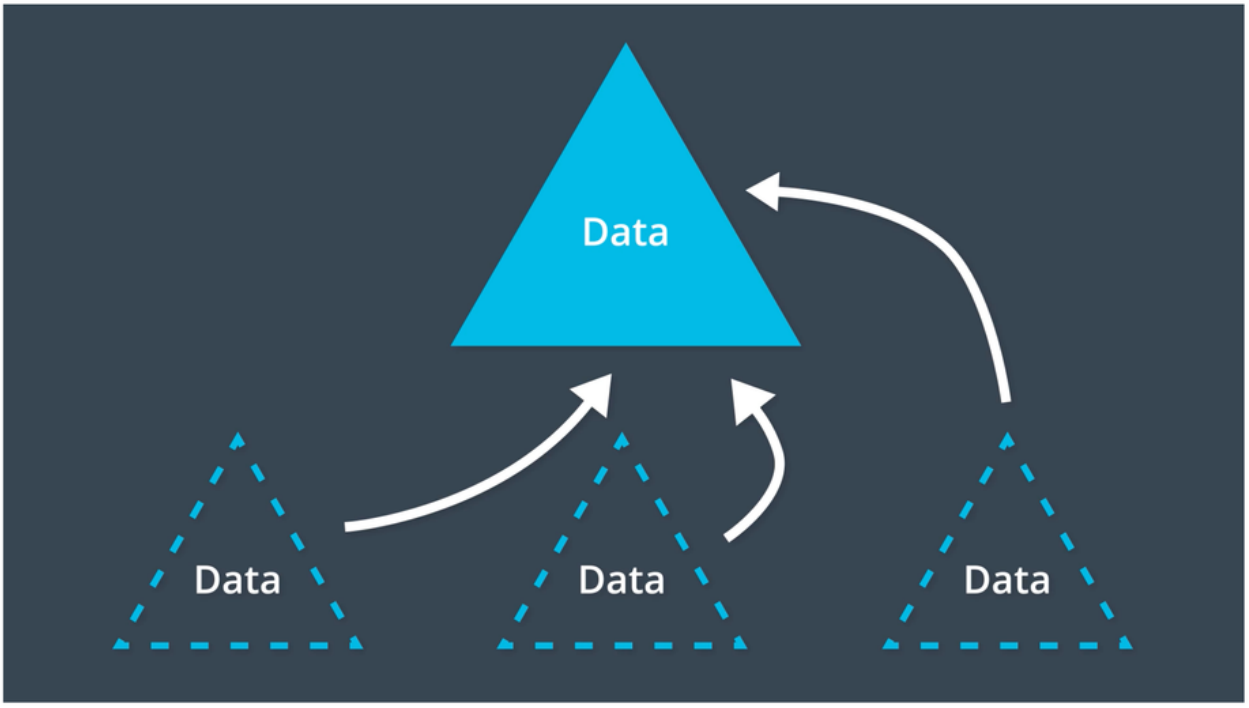
In the below example, the app appears exactly the same to the end user, however, it's functioning quite differently under the hood. All of the data is stored *outside of the UI code* and is just *referenced* from the UI code.

With a change like this, if the data needs to be modified at all, then all of the data is located in one place and needs to be only changed once. Then the areas of the app that are referencing pieces of data, will be updated since the source they're pulling from has changed.

Let's dive in this example, as discuss before, the application's data is sprinkled throughout our entire app, right? So what if it was all in one place? It mean all of our state for application in a single location. By using this way that would improve sharing state among different parts of our application.



Typically, when two parts of your app rely on the same piece of data, if they each have their own copy of that data, you have to do a lot of tricky work to make sure that they stay in sync. This is a pain point almost all of us have probably felt at one point. For example: You've changed your avatar under your profile then make a new post only to find out that your old Avatar was used for that new post



Now if all of our state is in one location, you avoid this problem entirely. Because each section of your app that needs it, that will just reference the data from the one location rather than duplicating it. Another benefit and really the most important benefit we care about is more predictable state changes.

If all of our state is in one location, we can set strict rules for how to update that state leading to a more predictable state.

When we putting all of our state in a single location. We will call it the state tree.

State Tree

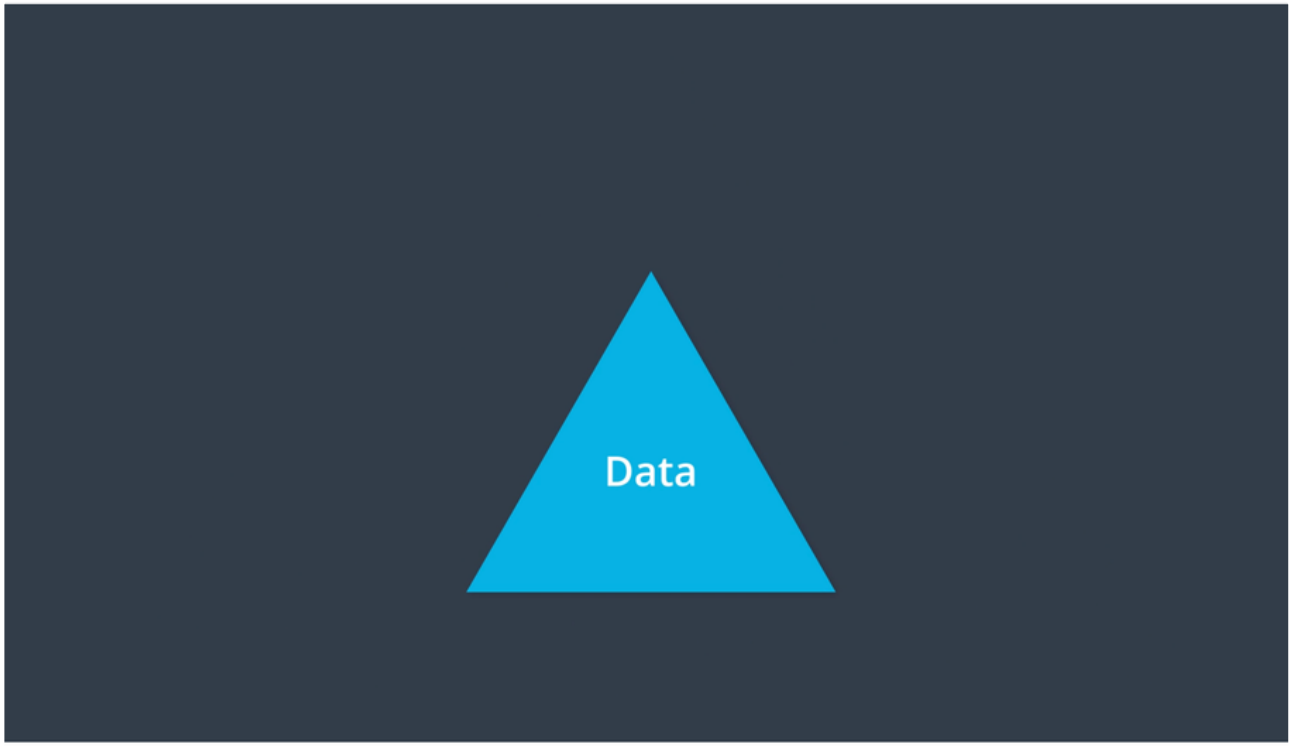
One of the key points of Redux is that all of the data is stored in a single object called the *state tree*. But what does a state tree actually look like? Here's an example:

```
{
  recipes: [
    { … },
    { … },
    { … }
  ],
  ingredients: [
    { … },
    { … },
    { … },
    { … },
    { … },
    { … }
  ],
  products: [
    { … },
    { … },
    { … },
    { … }
```

```
  ]
}
```

See how all of the data for this imaginary cooking site is stored in a single object? All of the state (or "application data") for this site is stored in one, single location. This is what we mean when we say "state tree": it's just all of the data stored in a single object.

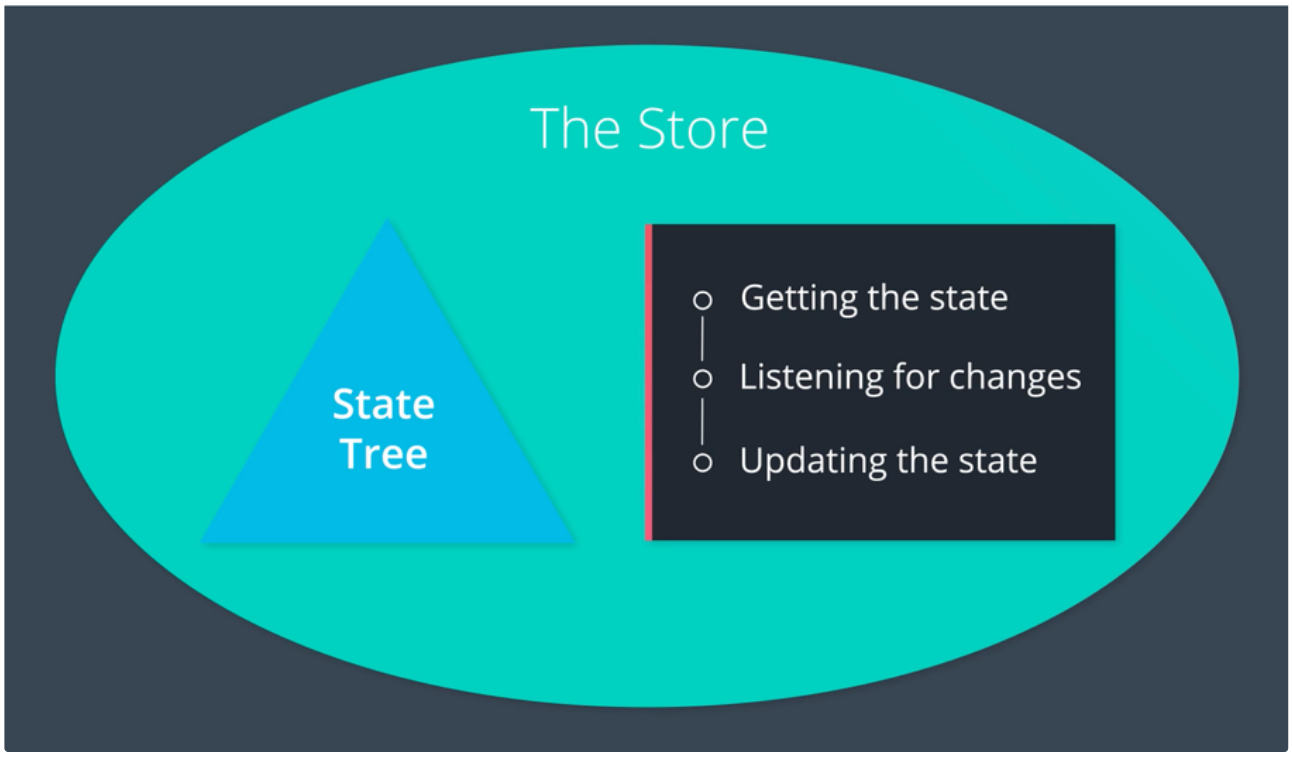
Whenever we refer to an application's "state tree", we'll use a triangle to convey this concept.



The next thing we need to figure out, that is how we'll actually interact with it. If we're actually going to build a real application with our state tree, there are three ways in which we'll need to interface with it

1. Firstly, we'll need a way of getting the state.
2. Secondly, we'll need a way to listen for when the state changes.
3. Thirdly, we'll need a way to update the state.

When we wrap all of these things together, that concept is called the store.



So when we talk about the store, we're talking about the state tree as well as three ways in which we'll interact with it: getting the state, listening for updates to the state and updating the state.

Summary

In this lesson, we looked at the data in an application. We saw that in traditional apps, the data is mixed in with the UI and markup. This can lead to hard-to-find bugs where updating the state in one location doesn't update it in every location.

We learned that the main goal that Redux is trying to offer is predictable state management. The way that Redux tries to accomplish this is through having a *single state tree*. This state tree is an object that stores the entire state for an application. Now that all state is stored in one location, we discovered three ways to interact with it:

1. Getting the state
2. Listening for changes to the state
3. Updating the state

Then we combine the three items above and the state tree object itself into one unit which we called *the store*.

Create Store: Getting and Listening

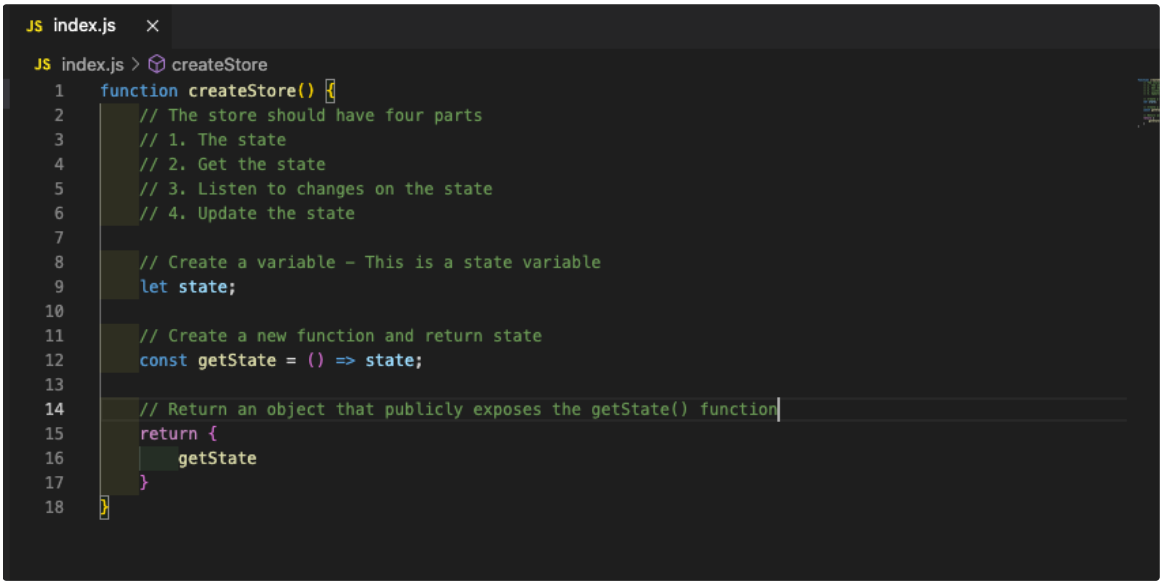
In this section, we'll be building the store by going to actually create the store code ourselves from scratch.

We'll start with a blank `index.js` file and create a factory function that creates *store* objects. Then we'll have the store keep track of the state, and we'll write the method to get the state from the store.

The Store should have four parts:

1. The state
2. Get the state
3. Listen to changes on the state
4. Update the state

Getting the State



In the above screenshot, we started building out the `createStore()` function. Currently, this factory function:

- Takes in no arguments
- Sets up a local (private) variable to hold the state
- Sets up a `getState()` function
- Returns an object that publicly exposes the `getState()` function

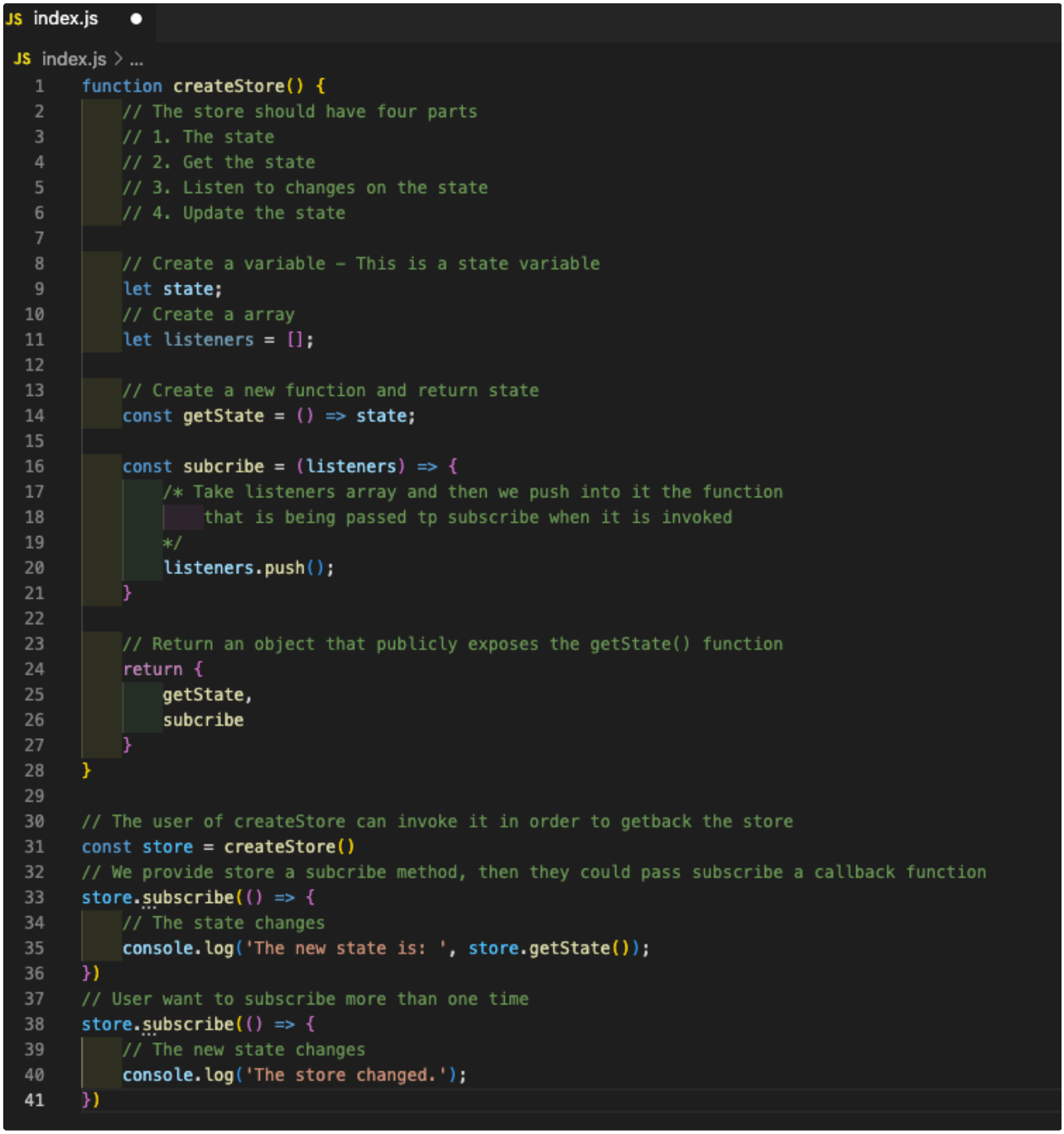
Our list of things we need to build for the store is shrinking:

- ~~The state tree~~
- ~~A way to get the state tree~~
- A way to listen and respond to the state changing
- A way to update the state

Listening to Changes

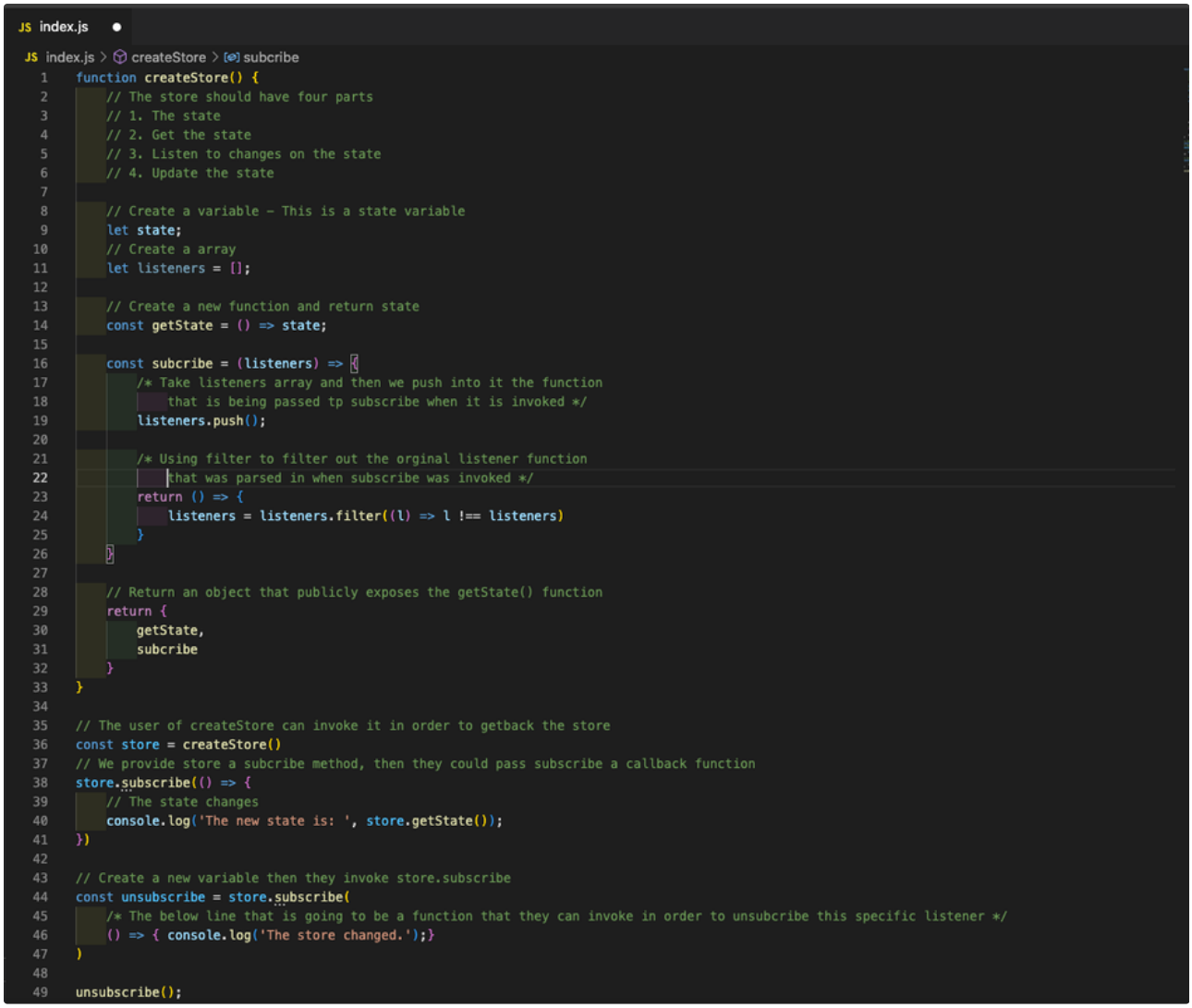
Now that we have some internal state inside of our store and we also have a way to get that state. The next feature we want to add is a ways to listen for changes on the state.

This is an example how to create a listening to changes on the state



```
JS index.js ●

JS index.js > ...
1   function createStore() {
2       // The store should have four parts
3       // 1. The state
4       // 2. Get the state
5       // 3. Listen to changes on the state
6       // 4. Update the state
7
8       // Create a variable - This is a state variable
9       let state;
10      // Create a array
11      let listeners = [];
12
13      // Create a new function and return state
14      const getState = () => state;
15
16      const subcribe = (listeners) => {
17          /* Take listeners array and then we push into it the function
18             that is being passed tp subscribe when it is invoked
19          */
20          listeners.push();
21      }
22
23      // Return an object that publicly exposes the getState() function
24      return {
25          getState,
26          subcribe
27      }
28  }
29
30  // The user of createStore can invoke it in order to getback the store
31  const store = createStore()
32  // We provide store a subcribe method, then they could pass subscribe a callback function
33  store.subscribe(() => {
34      // The state changes
35      console.log('The new state is: ', store.getState());
36  })
37  // User want to subscribe more than one time
38  store.subscribe(() => {
39      // The new state changes
40      console.log('The store changed.');
41  })
```

In the present, we have a way for the user to subscribe to changes, we also can provide them a way to unsubscribe for changes. The image below is a example how we can implement both subscribe and unsubscribe function.



Keep in mind that the following ES6 arrow functions:

```
const subscribe = (listener) => {
    listeners.push(listener)
    return () => {
      listeners = listeners.filter((l) => l !== listener)
    }
  }
```

are essentially equivalent to the following in ES5:

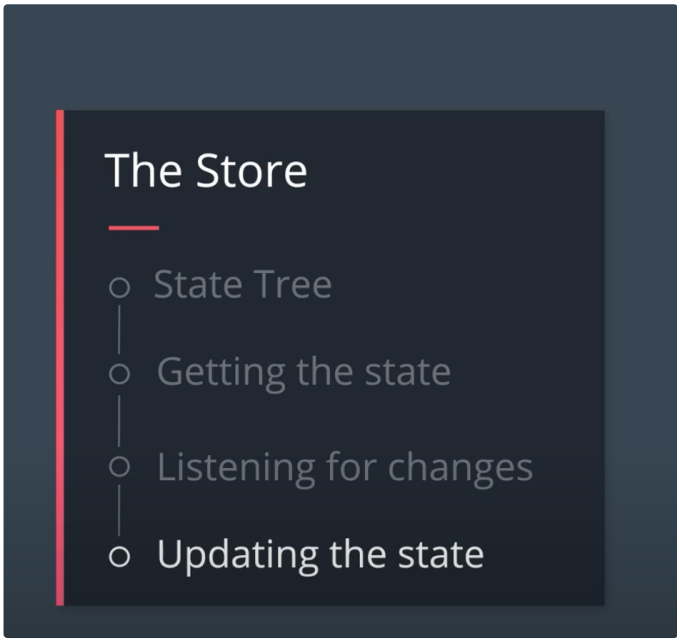
```
var subscribe = function subscribe(listener) {
  listeners.push(listener);
  return function () {
    listeners = listeners.filter(function (l) {
      return l !== listener;
    });
  };
};
```

Before we add our next piece of functionality to create store, we will discuss how the functionality has so far.

- First, it contains the state.
- Second, it has a getState function - which just return us the state.

- Finally, it has a subscribe function - which will let each of the listeners know whenever the state changes.

If you remember back to when we first talked about our store, there were four parts and we covered three of four part:



Now remember, the whole goal here is to increase the predictability of the state in our application. We can't just allow anything or anyone to update the state. If we did, that would drastically decrease predictability. In fact, the only way in which we can increase predictability in terms of updating the state, that is by establishing a strict set of rules for how updates can be made.

It is a hardly to understand, right? Let take an example to easy understand about it. In order for a NFL team to maximize their chances of winning, they always have to be on the same page. They need to operate as one cohesive unit. Every miscommunication can and will lead to negative consequences. So in a sense, NFL teams have the same goals. That is an increasing predictability. But how do they go about

accomplishing this? They create a Playbook and each player must know it. By this way, when the team runs a play from the playbook, players will know exactly what each member of the team will be doing. That is a maximizing predictability.

From this example, we have rule number one to increasing predictability. Just like NFL teams have a collection of place, we too can have a collection of events that can occur in our app which will change the state of our store.

> *Only an event can change the state of the store.*

When an event takes place in a Redux application, we use a plain JavaScript object to keep track of what the specific event was. This object is called an **Action**.

We would use an object with a type property that describes the event taking place. The object might look like this:

```
{
  type: "ADD_PRODUCT_TO_CART"
}
```

As you can see, an Action is clearly just a **plain JavaScript object**. What makes this plain JavaScript object special in Redux, is that *every Action must have a* `type` property. The purpose of the `type` property is to let our app (Redux) know *exactly* what event just took place. This Action tells us that a product was added to the cart.

Now, since an Action is just a regular object, we can include extra data about the event that took place:

```
{
  type: "ADD_PRODUCT_TO_CART",
  productId: 17
}
```

In this Action, we're including the `productId` field. Now we know exactly which product was added to the store!

One more note to keep in mind as you build your Action objects: it's better practice to pass as little data as possible in each action. For example, you should prefer passing the index or ID of a product rather than the *entire product object* itself.

Action Creators are functions that create/return action objects. For example:

```
const addItem = (item) => ({
  type: ADD_ITEM,
  item,
});

```

Summary

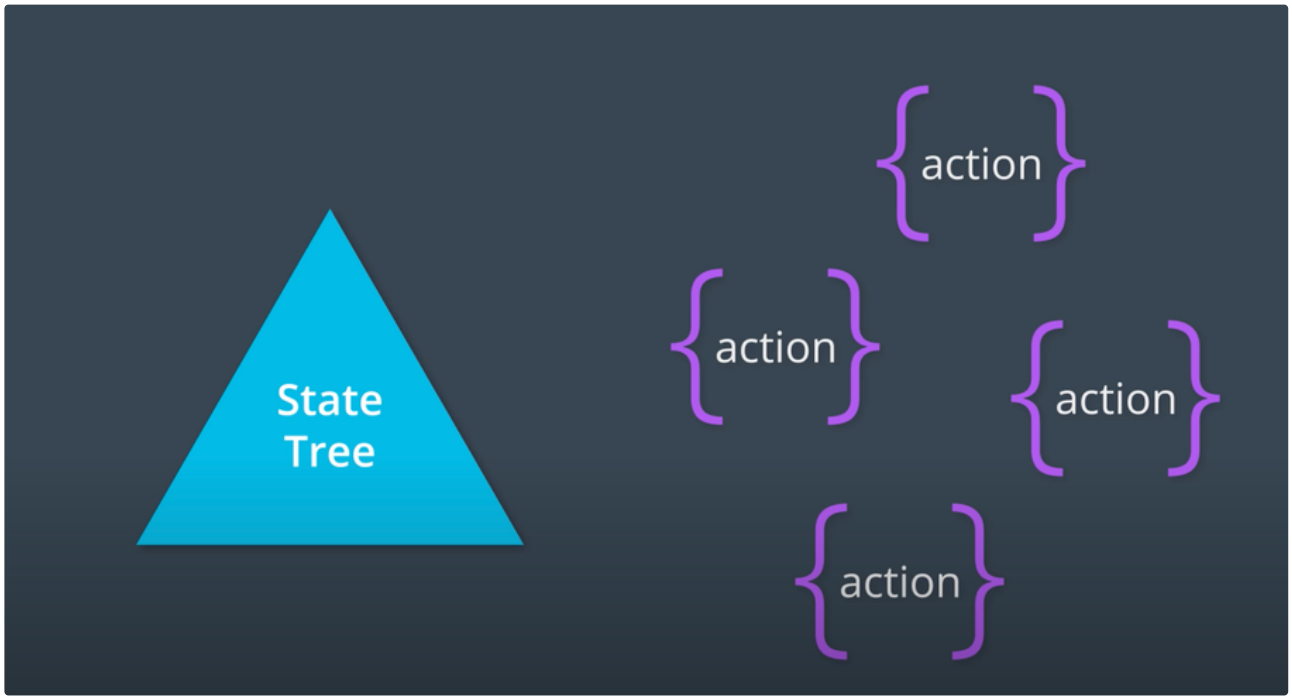
In this section, we started creating our store by building out a `createStore()` function. So far, this function keeps track of the state, and provides a method to get the state and one to keep track of listener functions that will be run whenever the state changes.

Updating State

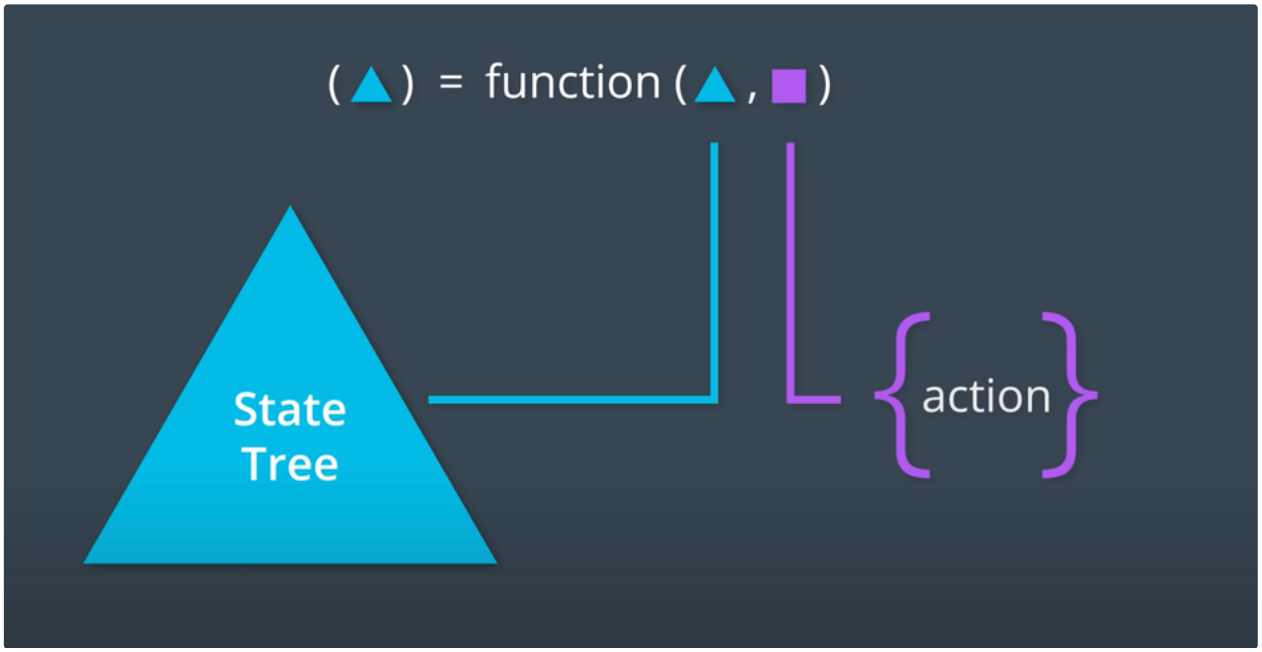
The whole goal of Redux is to increase predictability:

> *Redux is a predictable state container for JavaScript apps.*

Let's see dig into how we can use actions and our state tree to predictably manage an application's state.



We have the entire state of the application in the state tree. We also know about every action that can change the application state. So we have these two distinct pieces of data, but we need something to tie them together. Meaning, we need a way to update our state based on the current action which occurred. We will using function to do it, the function will take two arguments which are the current state and the action which occurred.



The function above has to be as predictable as possible. It mean we should be able to know what the return value will be of the function based on the input values. This sounds complicated but it turns out that the functional programming community has already solved this problem and they've given it a name, it is called pure function. This brings us to rule number two for increasing predictability.

> *The function that returns the new state needs to be a pure function.*

So far, our rules are:

1. Only an event can change the state of the store.
2. The function that returns the new state needs to be a pure function.

What are Pure Functions?

Pure functions are integral to how state in Redux applications is updated. By definition, pure functions:

1. Return the same result if the same arguments are passed in
2. Depend solely on the arguments passed into them
3. Do not produce side effects, such as API requests and I/O operations

If a function passes all three of these requirements, then it's a pure function. On the other hand, if it fails even one of these, then it's an impure function.

Let's check out an example of a pure function, `square()`:

```
// square() is a pure function

const square = (x) => x * x;
```

`square()` is a pure function because it outputs the same value every single time, given that the same argument is passed into it. There is no dependence on any other values to produce that result, and we can safely expect *just* that result to be returned -- no side effects

On the other hand, let's check out an example of an *impure* function, `calculateTip()`:

```
// calculateTip() is an impure function

const tipPercentage = 0.15;

const calculateTip = (cost) => cost * tipPercentage;
```

`calculateTip()` calculates and returns a number value. However, it relies on a variable (`tipPercentage`) that lives *outside* the function to produce that value. Since it fails one of the requirements of pure functions, `calculateTip()` is an impure function. However, we could convert this function to a pure function by passing in the outside variable, `tipPercentage`, as a second argument to this function!

```
const calculateTip = (cost, tipPercentage = 0.15) => cost * tipPercentage;
```

Why Pure Functions Are Great

For our purposes, the most important feature of a pure function is that it's predictable. If we have a function that takes in our state and an action that occurred, the function should (if it's pure!) return the exact same result *every single time*.

The Reducer Function

Reducers are a pure function in Redux. Pure functions are predictable. Reducers are the only way to change states in Redux. It is the only place where you can write logic and calculations. Reducer function will accept the previous state of app and action being dispatched, calculate the next state and returns the new object.

The following few things should never be performed inside the reducer −

- Mutation of functions arguments
- API calls & routing logic
- Calling non-pure function e.g. Math.random()

The following is the syntax of a reducer −

> `(state,action) => newState`

The reducer takes two parameters: `state` and `action`. You need to have an initial value so that when Redux calls the reducer for the first time with `undefined`, it will return the `initialState`. Then the function uses a `switch` statement to determine which type of action it's dealing with. If there is an unknown action, then it should return the `state`, so that the application doesn't lose its current state.

Let's look closer at the return statement in that reducer:

We're returning a brand new object rather than trying to change state. We then use the spread operator to create a copy of the state. Then we override the `username` property with the new value from `action.username`.

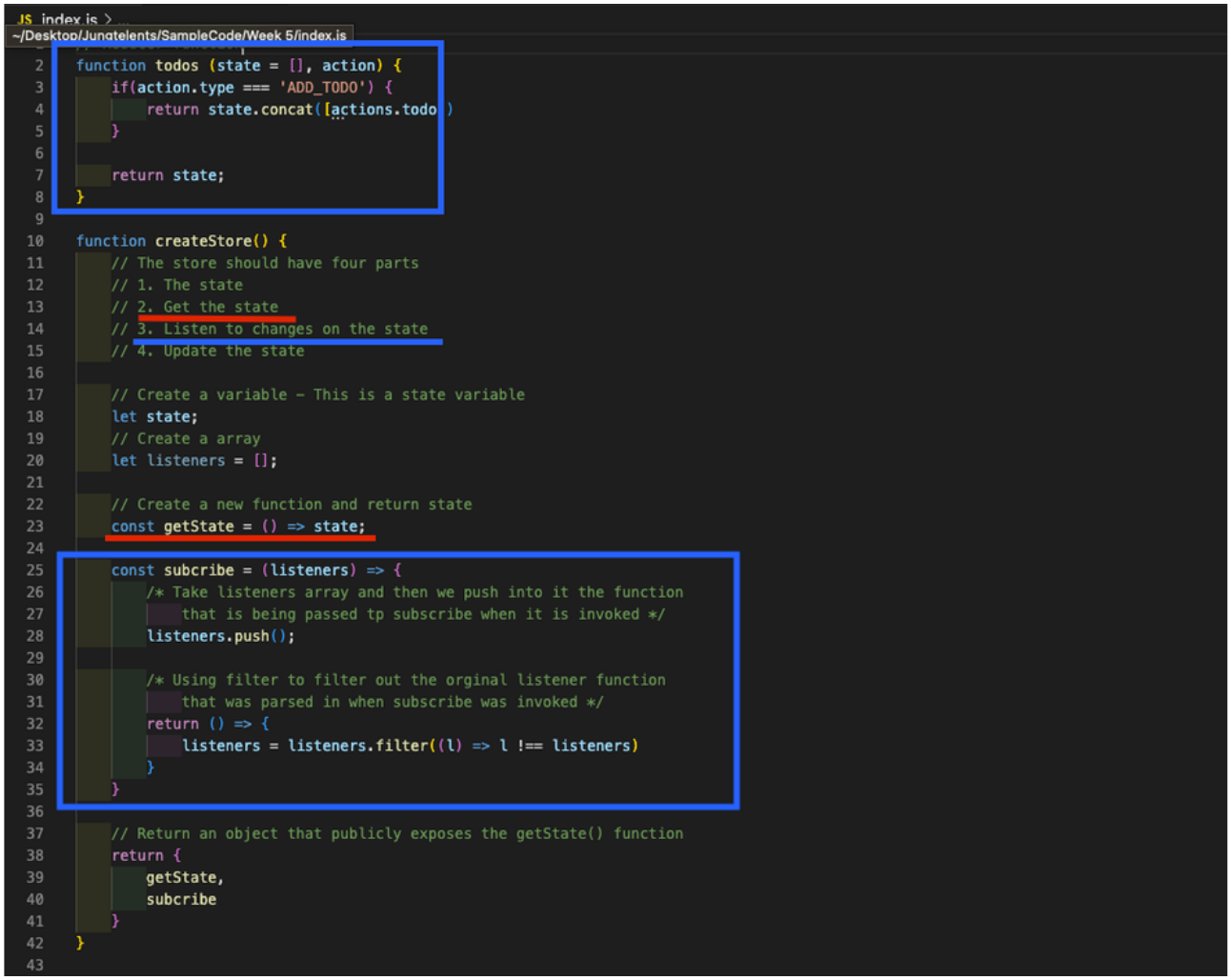
Let's review what have so far. There are three parts to our app:

- The actions represents the different events that will change the state of our store.
- The reducer is a function which takes in the current state and an action which occurred.
- The createStore is responsible for creating the actual store.

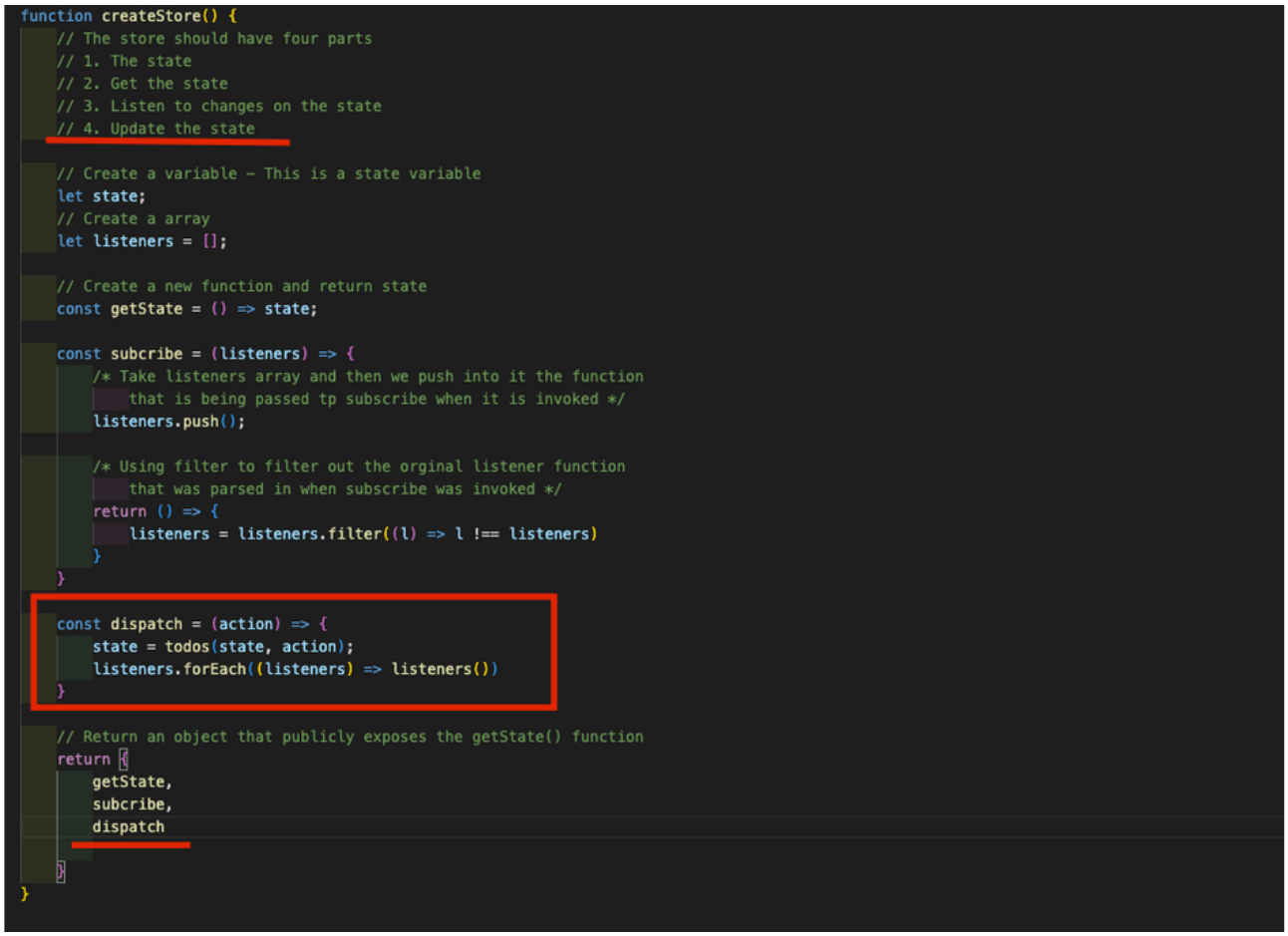
Create Store Dispatch

We have a way to get that state with our getState() method. (red-color)

We also have a way to listen to changes on it with subscribe() method. (blue-color)



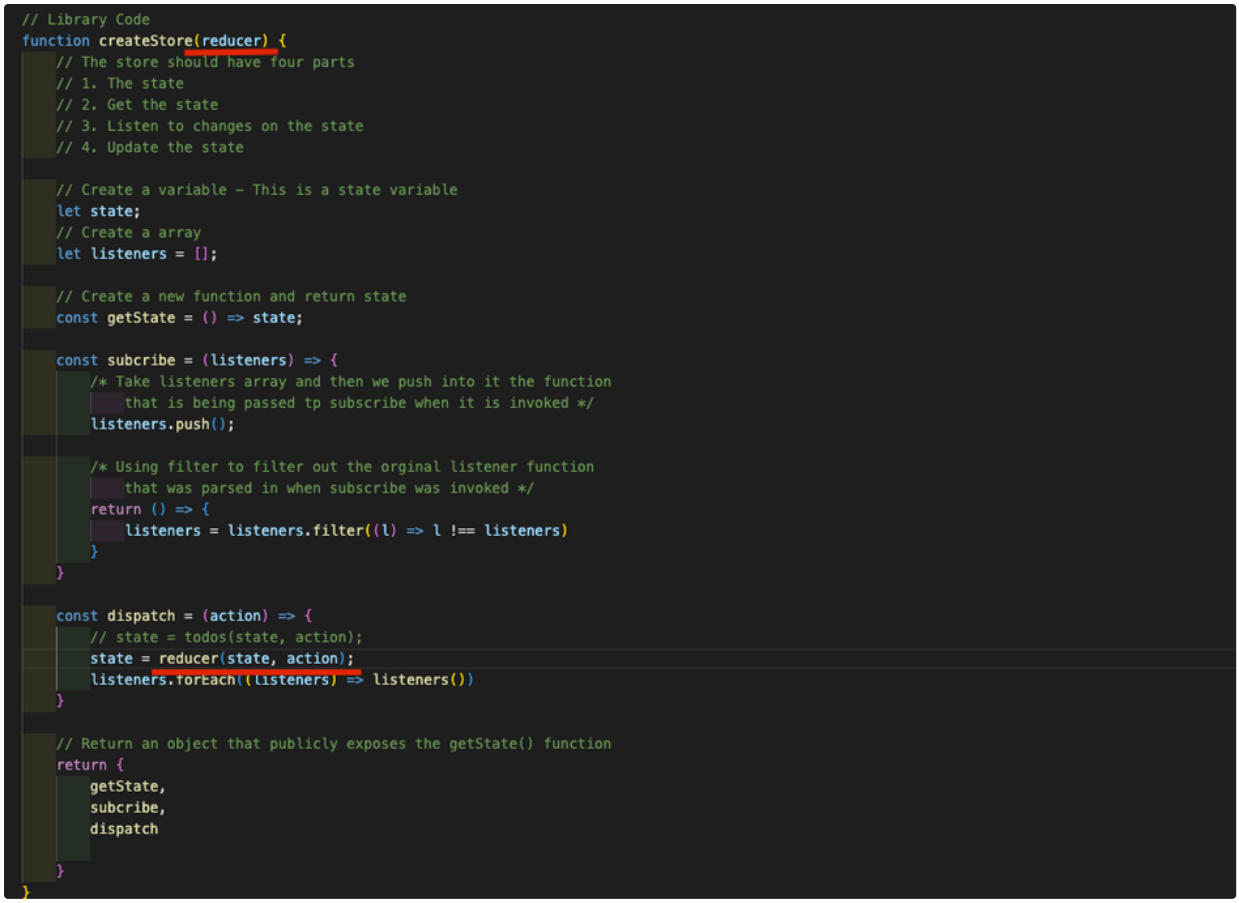
Now the only the other thing we need to figure out, that is how to actually update the state of our store. We will use a new function that we will call dispatch, dispatch function is responsible for updating the state inside of our actual store. It needs to receive the action which is going to tell dispatch the specific event that occurred inside of the application. We will assign the current state and passing it the action which occurred. Because we just modified the state, we need to loop through all of our listeners, which is just going to be an array of functions, and invoke each one of them, so that any listener to the user setup will be invoked. Finally, we will return the dispatch function.



```
function createStore() {
    // The store should have four parts
    // 1. The state
    // 2. Get the state
    // 3. Listen to changes on the state
    // 4. Update the state

    // Create a variable - This is a state variable
    let state;
    // Create a array
    let listeners = [];

    // Create a new function and return state
    const getState = () => state;

    const subcribe = (listeners) => {
        /* Take listeners array and then we push into it the function
            that is being passed tp subscribe when it is invoked */
        listeners.push();

        /* Using filter to filter out the orginal listener function
            that was parsed in when subscribe was invoked */
        return () => {
            listeners = listeners.filter((l) => l !== listeners)
        }
    }

    const dispatch = (action) => {
        state = todos(state, action);
        listeners.forEach((listeners) => listeners())
    }

    // Return an object that publicly exposes the getState() function
    return {
        getState,
        subcribe,
        dispatch
    }
}
```

The new `dispatch()` method is pretty small, but is vital to our functioning store code. To briefly recap how the method functions:

- `dispatch()` is called with an Action
- The reducer that was passed to `createStore()` is called with the current state tree and the action. This updates the state tree



```
// Library Code
function createStore(reducer) {
    // The store should have four parts
    // 1. The state
    // 2. Get the state
    // 3. Listen to changes on the state
    // 4. Update the state

    // Create a variable - This is a state variable
    let state;
    // Create a array
    let listeners = [];

    // Create a new function and return state
    const getState = () => state;

    const subcribe = (listeners) => {
        /* Take listeners array and then we push into it the function
            that is being passed tp subscribe when it is invoked */
        listeners.push();

        /* Using filter to filter out the orginal listener function
            that was parsed in when subscribe was invoked */
        return () => {
            listeners = listeners.filter((l) => l !== listeners)
        }
    }

    const dispatch = (action) => {
        // state = todos(state, action);
        state = reducer(state, action);
        listeners.forEach((listeners) => listeners())
    }

    // Return an object that publicly exposes the getState() function
    return {
        getState,
        subcribe,
        dispatch

    }
}
```

- Because the state has (potentially) changed, all listener functions that have been registered with the `subscribe()` method are called

Summary

In this section, we learned about a number of important points about Redux. We learned about pure functions, a Reducer function (which, itself, needs to be a *pure function*), dispatching changes in our store, and identifying which parts of our code are generic library code and which are specific to our app.

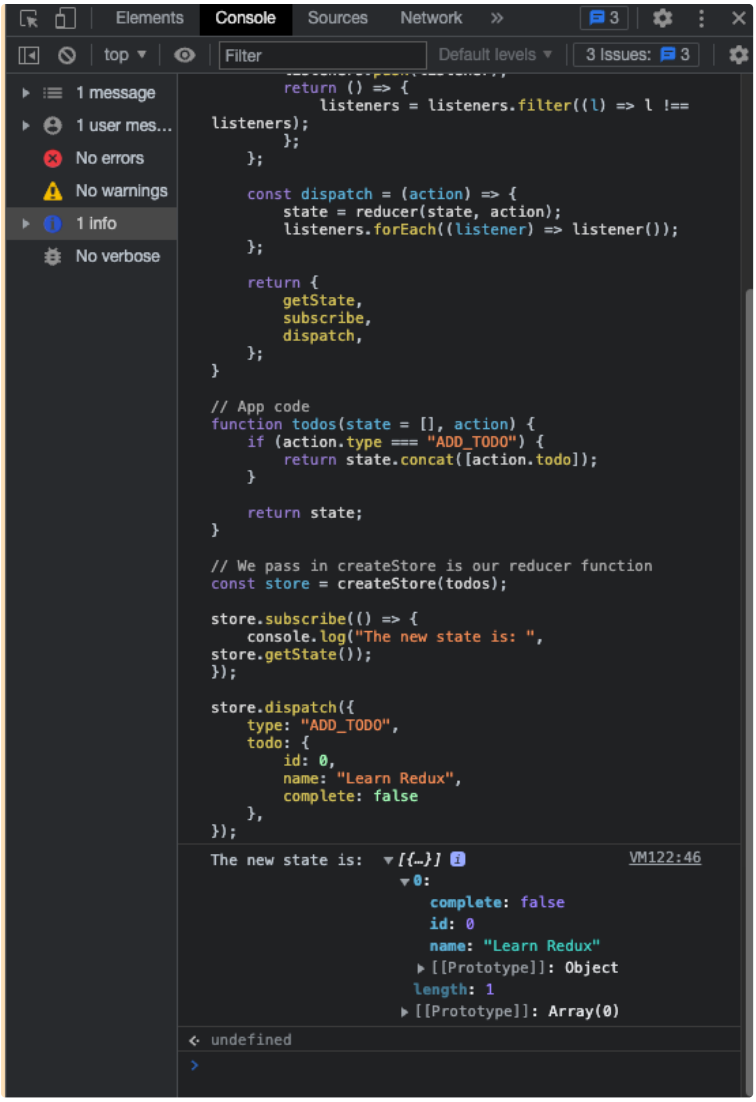
Putting it All Together

Let's create a new variable under the name store.

```
1 const store = createStore(todos);
```

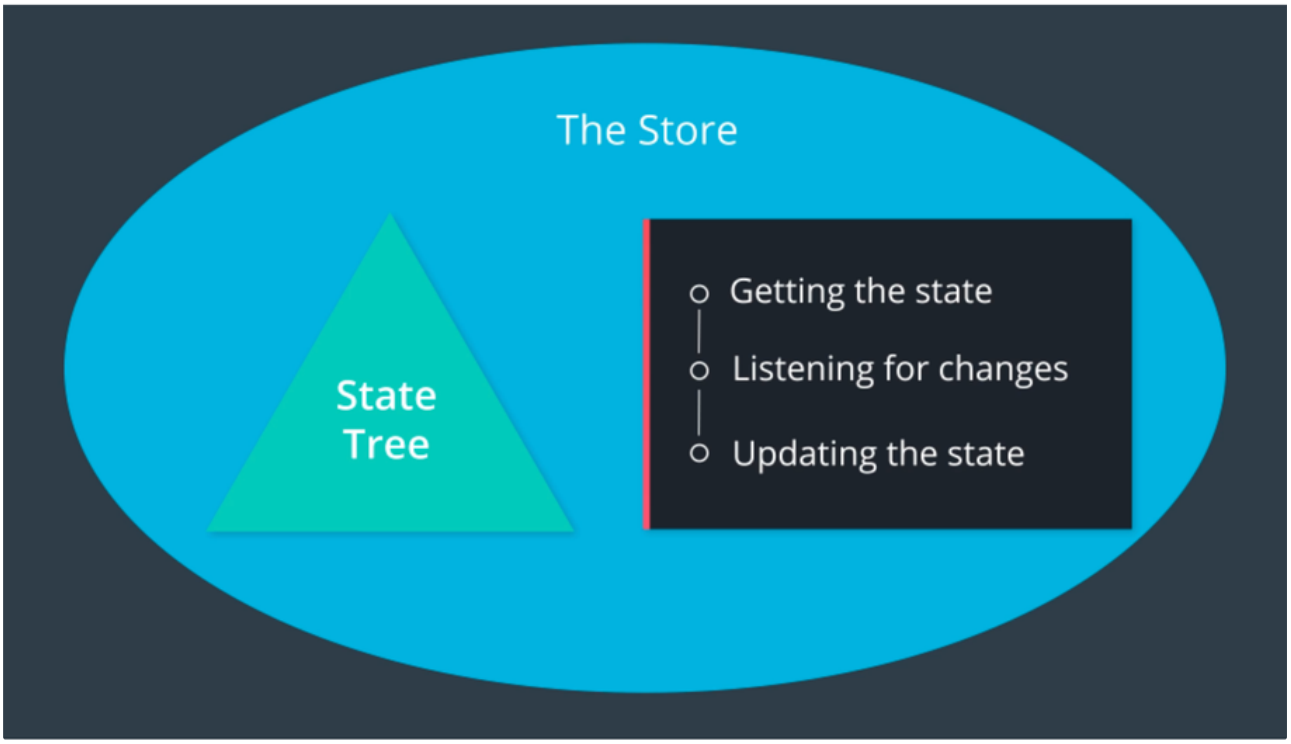
So now we can using three method in createStore() that are getState(), subscribe() and dispatch() like the image below.

After you edit the code above in the file, open the console of Chrome and paste all the code into the console and you will see the result.



Now we can call dispatch as many times as we want like image below.

Then we have two items in our state, learn Redux as well as Read a book. So whenever we want to update the state of our store, all we need to do now is called dispatch, passing it the action which occurred.



The Store contains the state tree and provides ways to interact with the state tree

We've finally finished creating the `createStore()` function! Using the image above as a guide, let's break down what we've accomplished:

- We created a function called `createStore()` that returns a *store* object
- `createStore()` must be passed a "reducer" function when invoked
- The store object has three methods on it:
  - `.getState()` - used to get the current state from the store
  - `.subscribe()` - used to provide a listener function the store will call when the state changes
  - `.dispatch()` - used to make changes to the store's state
- The store object's methods have access to the state of the store via closure

Managing More State

As of right now, our code is handling the `ADD_TODO` action. There are still a couple more actions that our app is supposed to be able to handle:

- The `REMOVE_TODO` action
- The `TOGGLE_TODO` action

New Actions

Our app can not only handle *adding* todo items -- it can now handle *removing* a todo item, as well as *toggling* a todo item (as complete or incomplete)! To make this all possible, we updated our `todos` reducer to be able to respond to actions of the `type` `REMOVE_TODO` and `TOGGLE_TODO`.

Our `todos` reducer originally looked like the following:

```
function todos(state = [], action) {
  if (action.type === "ADD_TODO") {
    return state.concat([action.todo]);
  }

  return state;
}
```

To resolve additional action types, we added a few more conditions to our reducer logic:

```
function todos(state = [], action) {
  if (action.type === "ADD_TODO") {
    return state.concat([action.todo]);
  } else if (action.type === "REMOVE_TODO") {
    // ...
  } else if (action.type === "TOGGLE_TODO") {
    // ...
  } else {
    return state;
  }
}
```

Note that just like the original `todos` reducer, we simply return the original state if the reducer receives an action type that it's not concerned with.

To remove a todo item, we called `filter()` on the state. This returns a new state (an array) with only todo items whose `id`'s *do not* match the `id` of the todo we want to remove:

```
function todos(state = [], action) {
  if (action.type === "ADD_TODO") {
    return state.concat([action.todo]);
  } else if (action.type === "REMOVE_TODO") {
    return state.filter((todo) => todo.id !== action.id);
  } else if (action.type === "TOGGLE_TODO") {
    // ...
  } else {
    return state;
  }
}
```

To handle toggling a todo item, we want to change the value of the `complete` property on whatever `id` is passed along on the action. We mapped over the entire state, and if `todo.id` matched `action.id`, we used `Object.assign()` to return a new object with merged

properties:

```
function todos(state = [], action) {
  if (action.type === "ADD_TODO") {
    return state.concat([action.todo]);
  } else if (action.type === "REMOVE_TODO") {
    return state.filter((todo) => todo.id !== action.id);
  } else if (action.type === "TOGGLE_TODO") {
    return state.map((todo) =>
      todo.id !== action.id
        ? todo
        : Object.assign({}, todo, { complete: !todo.complete })
    );
  } else {
    return state;
  }
}
```

We then refactored our entire `todos` reducer to use a `switch` statement rather than multiple `if` / `else` statements:

```
function todos(state = [], action) {
  switch (action.type) {
    case "ADD_TODO":
      return state.concat([action.todo]);
    case "REMOVE_TODO":
      return state.filter((todo) => todo.id !== action.id);
    case "TOGGLE_TODO":
      return state.map((todo) =>
        todo.id !== action.id
          ? todo
          : Object.assign({}, todo, { complete: !todo.complete })
      );
    default:
      return state;
  }
}
```

In the above snippet, we matched `cases` against an expression (i.e., `action.type`), and executed statements associated with that particular `case`.

Adding Goals to our App

Currently, the app keeps track of a single piece of state: a list of todo items.

Let's make the app a bit more complicated and add in a second piece of state for our app to track: goals.

Goals Reducer

```
function goals(state = [], action) {
  switch (action.type) {
    case "ADD_GOAL":
      return state.concat([action.goal]);
    case "REMOVE_GOAL":
      return state.filter((goal) => goal.id !== action.id);
    default:
      return state;
```

```
  }
}
```

Until now, we have two reducer functions:

- todos
- goals

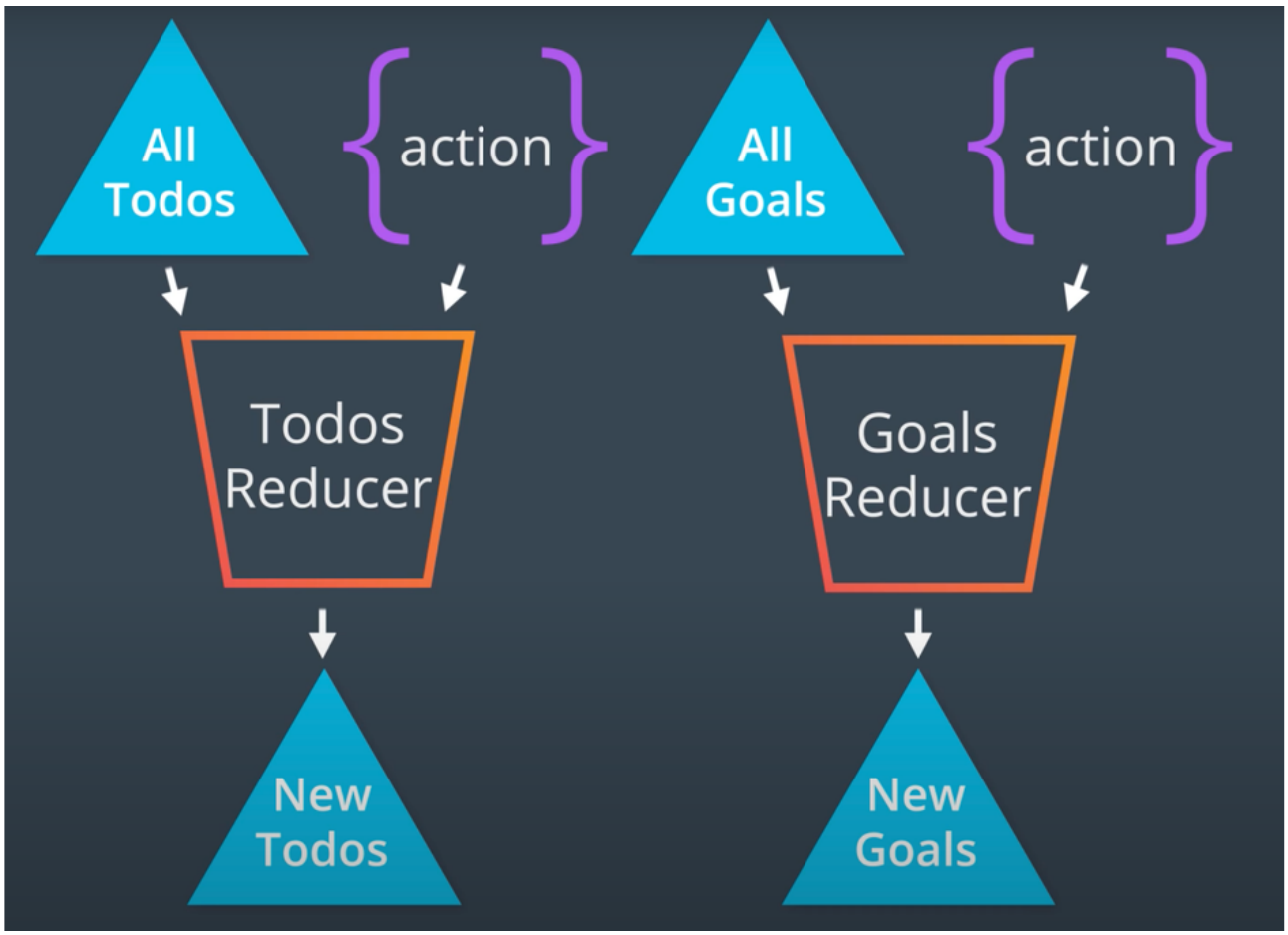
However, the `createStore()` function we built can only handle a *single* reducer function:

```
// createStore() takes one reducer function as an argument
const store = createStore(todos);
```

We can't call `createStore()` passing it two reducer functions:

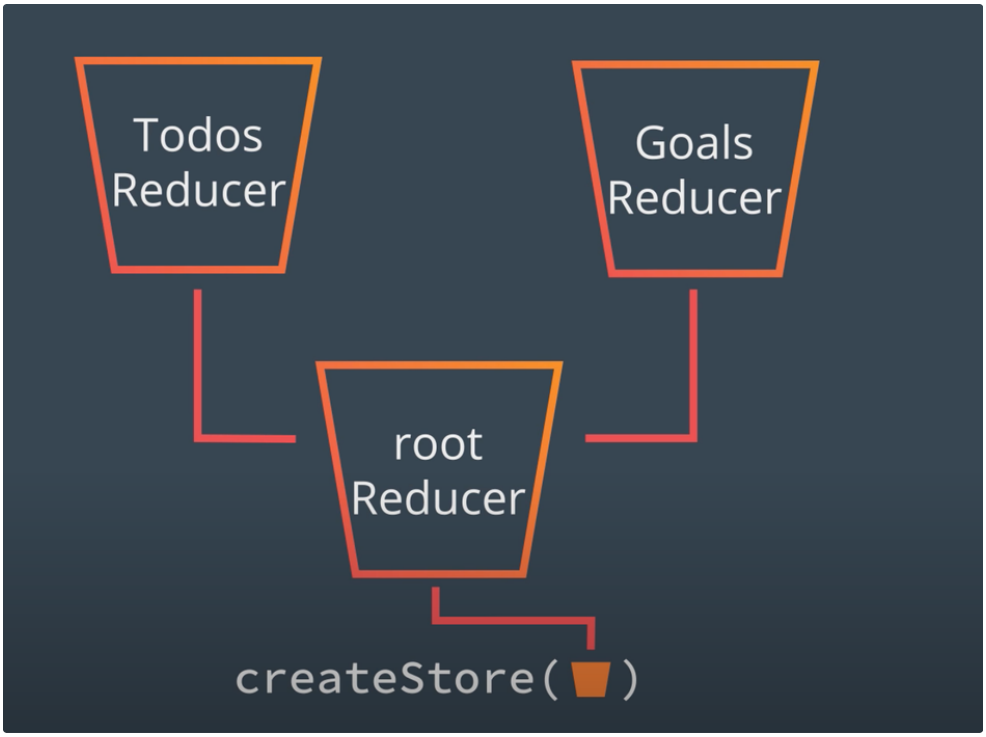
```
// this will not work
const store = createStore(todos, goals);
```

So we've got a problem, we got two reducer now and each is responsible for handling their specific slice of the state tree. This introduces a new problem, which is how we can use two or many reducer together?



Combine Reducers

Solving the problem above, we can create and using the root Reducer, then we can managing two or many reducer function. The main function of root Reducer will call the correct reducer whenever specific actions are dispatched.

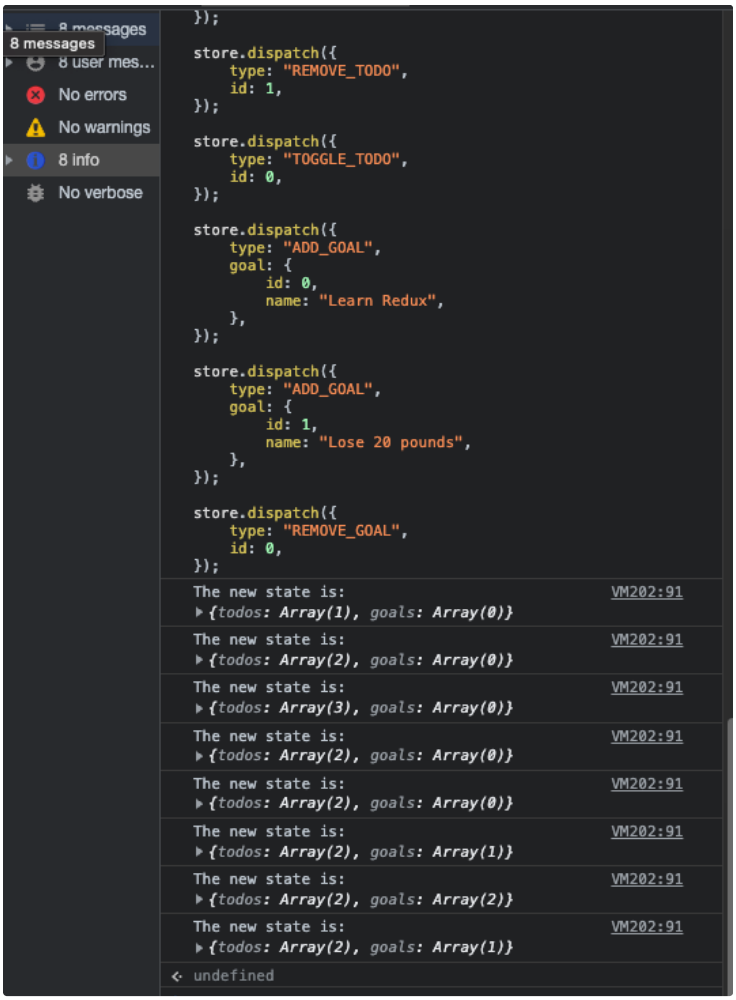


Whenever `dispatch` is called, we invoke our `app` function. The `app` function will then invoke the `todos` reducer as well as the `goals` reducer. Those will return their specific portions of the state. And then, the `app` function will return a state object with a `todos` property (the value of which is what the `todos` reducer returned) and a `goals` property (the value of which is what the `goals` reducer returned).

```
function todos(state = [], action) {
  switch (action.type) {
    case "ADD_TODO":
      return state.concat([action.todo]);
    case "REMOVE_TODO":
      return state.filter((todo) => todo.id !== action.id);
    case "TOGGLE_TODO":
      return state.map((todo) =>
        todo.id !== action.id
          ? todo
          : Object.assign({}, todo, { complete: !todo.complete })
      );
    default:
      return state;
  }
}

function goals(state = [], action) {
  switch (action.type) {
    case "ADD_GOAL":
      return state.concat([action.goal]);
    case "REMOVE_GOAL":
      return state.filter((goal) => goal.id !== action.id);
    default:
      return state;
  }
}

function app(state = {}, action) {
  return {
    todos: todos(state.todos, action),
```

```
    goals: goals(state.goals, action),
  };
}

// We pass the root reducer to our store because
// the createStore() function can only take in one reducer.

const store = createStore(app);
```

Our result will be like this



Summary

In this section, we bolstered our application to handle a number of different actions as well as an entirely new piece of state! In addition to our app handling the `ADD_TODO` action, it now handles:

- The `REMOVE_TODO` action
- The `TOGGLE_TODO` action

We also created the `goals` reducer which handles:

- An `ADD_GOAL` action
- A `REMOVE_GOAL` action

So our application can now manage the state of our todos and goals, and it can do all of this *predictably*!

Better Practices

Constants

We will convert all strings to variables. This action will improve our application and avoid the error thrown for misspelled action types.

```
const ADD_TODO = "ADD_TODO";
const REMOVE_TODO = "REMOVE_TODO";
const TOGGLE_TODO = "TOGGLE_TODO";
const ADD_GOAL = "ADD_GOAL";
const REMOVE_GOAL = "REMOVE_GOAL";
```

Then, our app code looks like this

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return state.concat([action.todo]);
    case REMOVE_TODO:
      return state.filter((todo) => todo.id !== action.id);
    case TOGGLE_TODO:
      return state.map((todo) =>
        todo.id !== action.id
          ? todo
          : Object.assign({}, todo, { complete: !todo.complete })
      );
    default:
      return state;
  }
}

function goals(state = [], action) {
  switch (action.type) {
    case ADD_GOAL:
      return state.concat([action.goal]);
    case REMOVE_GOAL:
      return state.filter((goal) => goal.id !== action.id);
    default:
      return state;
  }
}
```

Action Creators

```
store.dispatch({
  type: ADD_TODO,
  todo: {
    id: 0,
    name: "Walk the dog",
    complete: false,
  },
});

store.dispatch({
  type: ADD_TODO,
  todo: {
    id: 1,
```

```
    name: "Wash the car",
    complete: false,
  },
});

store.dispatch({
  type: ADD_TODO,
  todo: {
    id: 2,
    name: "Go to the gym",
    complete: true,
  },
});

store.dispatch({
  type: REMOVE_TODO,
  id: 1,
});

store.dispatch({
  type: TOGGLE_TODO,
  id: 0,
});

store.dispatch({
  type: ADD_GOAL,
  goal: {
    id: 0,
    name: "Learn Redux",
  },
});

store.dispatch({
  type: ADD_GOAL,
  goal: {
    id: 1,
    name: "Lose 20 pounds",
  },
});

store.dispatch({
  type: REMOVE_GOAL,
  id: 0,
});


```

In the code above, we are duplicating a lot of code here. Duplication isn't necessarily a bad thing. Now, We can think in this instance, we can clean it up quite a bit. So we'll do change all of hard coding object into the specific dispatch and location. We will make a function to do whole job and return us to the object. That thing we don't need to remember the type such as 'ADD_TODO' every time that we want to dispatch the add_todo type and we don't have to remember that we need to pass along a specific todo because we can just invoke the function and it will handle a lot of that for us. Since now our coding will be like this

```
function addTodoAction(todo) {
  return {
    type: ADD_TODO,
    todo,
  };
```

```
}

function removeTodoAction(id) {
  return {
    type: REMOVE_TODO,
    id,
  };
}

function toggleTodoAction(id) {
  return {
    type: TOGGLE_TODO,
    id,
  };
}

function addGoalAction(goal) {
  return {
    type: ADD_GOAL,
    goal,
  };
}

function removeGoalAction(id) {
  return {
    type: REMOVE_GOAL,
    id,
  };
}
```

After we define functions for our app, we will invoke the dispatch like

```
store.dispatch(
  addTodoAction({
    id: 0,
    name: "Walk the dog",
    complete: false,
  })
);

store.dispatch(
  addTodoAction({
    id: 1,
    name: "Wash the car",
    complete: false,
  })
);

store.dispatch(
  addTodoAction({
    id: 2,
    name: "Go to the gym",
    complete: true,
  })
);

store.dispatch(removeTodoAction(1));

store.dispatch(toggleTodoAction(0));
```

```
28
29 store.dispatch(
30   addGoalAction({
31     id: 0,
32     name: "Learn Redux",
33   })
34 );
35
36 store.dispatch(
37   addGoalAction({
38     id: 1,
39     name: "Lose 20 pounds",
40   })
41 );
42
43 store.dispatch(removeGoalAction(0));
```

Summary

We converted our actions to use JavaScript *constants* instead of strings. We also refactored our `dispatch()` calls from passing in unique objects directly to them, to calling special functions that create the action objects - these special functions that create action objects are called **Action Creators**.

UI

What We're Going to Build

Now we create an `index.html` file and all of the JavaScript code will be transferred over to script tags, let's start adding in a User Interface. Let's check the code link below to make sure we are on same page.

☐ UI - CodeSandbox

Since our project has two pieces of state, we'll need two areas:

1. Todo list area
2. Goals area

This is what our UI should look like when we're finished: a Todo List area with an input to add a new Todo item, and a Goals area with an input to add a new Goal.

So this is what we're going for. It's not the best looking website ever created, but this isn't focus on CSS ;-). If you want to make it stunningly beautiful, feel free to add some CSS to your project 👍🏼

We already have the Redux portion of our application working, but so far, we've just been manually running snippets of code to interact with the Redux Store. Let's create the UI above so that we can interact with the store using the browser.

We add all of code below into the body tags and above the script tags

```
<div>
    <h1>Todo List</h1>
    <input id="todo" type="text" placeholder="Add Todo" />
    <button id="todoBtn">Add Todo</button>
    <ul id="todos"></ul>
</div>
<div>
    <h1>Goals</h1>
    <input id="goal" type="text" placeholder="Add Goal" />
    <button id="goalBtn">Add Goal</button>
    <ul id="goals"></ul>
</div>
```

After you added all of the code, your UI website will look like this

Summary

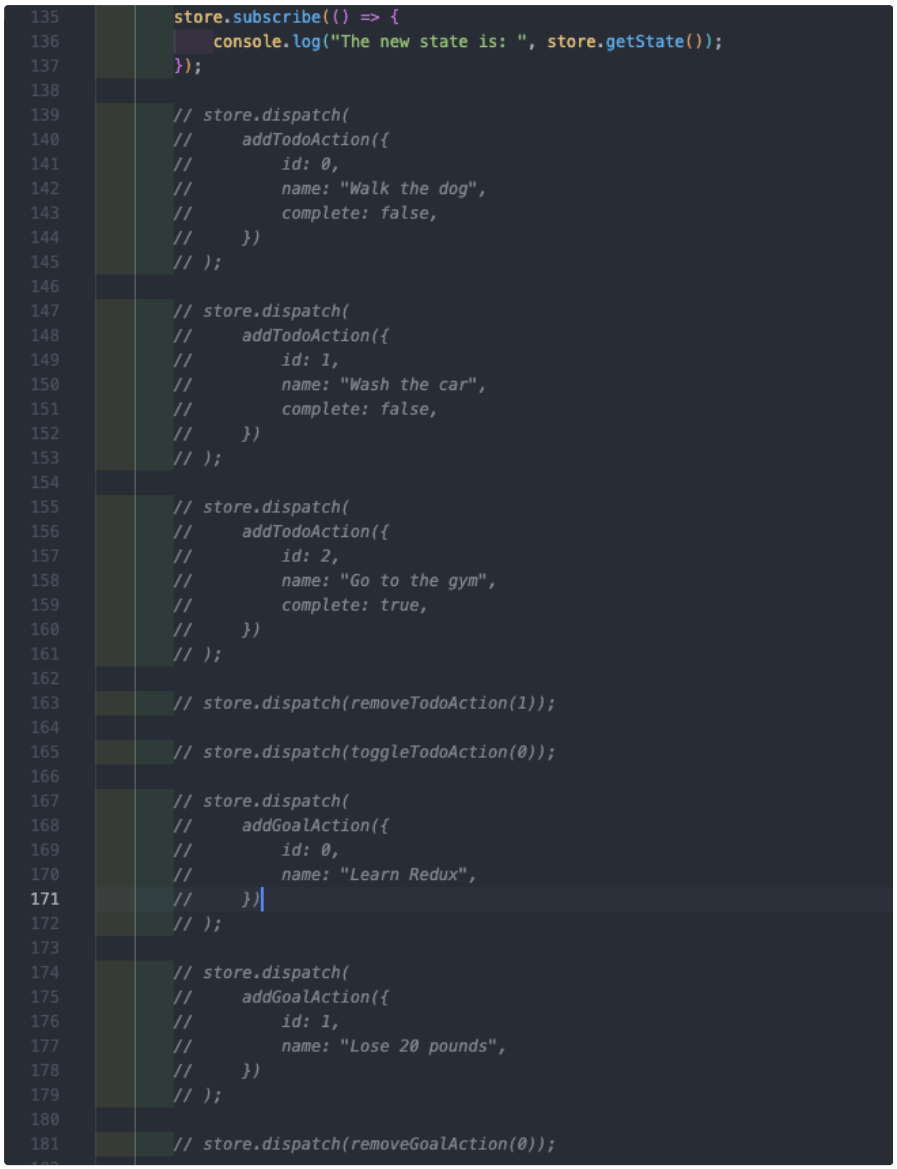
In this section, we added some minimal UI to our application. The actually state of our app hasn't changed at all, though.

In the next section, we'll hook up our shiny new UI to our state so that entering content via the UI will update the application's state.

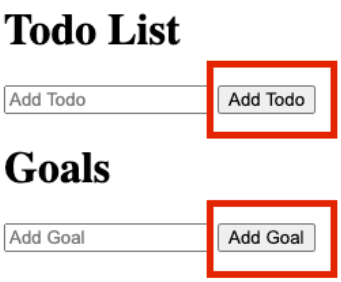
UI & State

Dispatching New Items

Now we have our simple UI, the next thing we want to do that make it so the user can add new Todo items as well as new goals from UI itself. The first thing, we will go through the code and comment out all of these dispatch invocations.



So that way, the only way that we are updating the state of the store is from the UI. But we still keep these around, just so we can reference them a little bit later when we do start dispatching things from the UI. We will make two different functions that are hooked up to both of our buttons.



So what these functions will be called, very fittingly, we have an addTodo function and then we will make an addGoal function.

```
function addTodo() {

}

function addGoal() {

}
```

After that, we want to grab the value of input field. First, we need to listen for when the buttons are clicked; we did this with the plain DOM addEventListener() method:

```
document.getElementById("todoBtn").addEventListener("click", addTodo);

document.getElementById("goalBtn").addEventListener("click", addGoal);
```

Pressing the `#todoBtn` will call `addTodo` which will add the new item to the state:

```
function addTodo() {
    const input = document.getElementById("todo");
    const name = input.value;
    input.value = "";

    store.dispatch(
        addTodoAction({
            name,
            complete: false,
            id: generateId(),
        })
    );
}
```

This method will extract the information from the input field, reset the input field, and then dispatch an `addTodoAction()` Action Creator with the text that the user typed into the input field.

Likewise, pressing the `#goalBtn` will call `addGoal` which will add the new item to the state:

```
function addGoal() {
    const input = document.getElementById("goal");
    const name = input.value;
    input.value = "";

    store.dispatch(
        addGoalAction({
            id: generateId(),
            name,
        })
    );
}
```

This method will extract the information from the input field, reset the input field, and then dispatch an `addTodoAction()` Action Creator with the text that the user typed into the input field.

Furthermore, we also need to create a generateId function to generate the unique Id for each action.

```
function generateId() {
    return (
        Math.random().toString(36).substring(2) +
        new Date().getTime().toString(36)
    );
}
```

The changes we just added made it so whenever the Todo input field is submitted, it will add a Todo item to the state. Likewise, whenever the Goal input field is submitted, it will add a new Goal item to the state.

Here's the commit with the changes made in the discuss above.

But we're using the UI to change the state of our store, and these changes are not reflecting the new state visually in the UI. Let's solve this problem now.

💡 *Need to Level Up Your DOM Skills?* 💡

> *Both the content in the previous we discuss, as well as the content in the following now depend on DOM-manipulation skills.*

- *Accessing elements with* `document.getElementById()`
- *Adding listeners with* `.addEventListener()`
- *Accessing the* `.value` property on an element
- *Creating a new element with* `.createElement()`
- *Adding new content with* `.appendChild()`
- *etc.*

Update UI

So now that we can add new todo items as well as new goals to the state of our store. The next thing we want to do is actually show each of those items on the UI themselves, rather than just in the console. Now what we want to do is we want to make two new functions, which are ***addTodoToDom*** function and the second one is ***addGoalToDom*** function.

```
function addTodoToDOM() {

}

function addGoalToDOM() {

}
```

These are responsible for going to take in a specific todo item or a specific goal, and then just as they imply, they are going to add that individual item to the DOM itself so that they will show up in the UI.

The addTodoToDOM function will take in a todo item. We will create a new const variable under the name node by using ***document.createElement()*** and this is going to be a list item. Next, we create the text variable by using ***document.createTextNode()***. We will also want to grab our node and append to it our text node. Then we can grab our node list by ***document.getElementById().***

```
function addTodoToDOM(todo) {
    const node = document.createElement('li');
    const text = document.createTextNode(todo.name);
    node.appendChild(text);

    document.getElementById('todos').appendChild(node);
}
```

Likewise, we do the same with ***addGoalToDOM()*** function

```
function addGoalToDOM(goal) {
    const node = document.createElement('li');
    const text = document.createTextNode(goal.name);
    node.appendChild(text);

    document.getElementById('goals').appendChild(node);
}
```

So now that we have these two functions which are going to help us in adding new items to the DOM, the next thing we want to do is actually invoke these functions for each todo item or for each goal that lives inside of our store.

```
store.subscribe(() => {
    // console.log("The new state is: ", store.getState());
    const {todos, goals} = store.getState();


    todos.forEach(addTodoToDOM);
    goals.forEach(addGoalToDOM)
});
```

Now the result will be like this when we add more todo or goal:

Todo List

Add Todo | Add Todo

- Cooking
- Cooking

Goals

Add Goal | Add Goal

- Meal

But you can see what is problem now? when we add the goal item and the todo item will add more one item at the same time. To solving this problem, we will do like this

```
const store = createStore(app);

store.subscribe(() => {
    // console.log("The new state is: ", store.getState());
    const {todos, goals} = store.getState();

    document.getElementById('todos').innerHTML = '';
    document.getElementById('goals').innerHTML = '';

    todos.forEach(addTodoToDOM);
    goals.forEach(addGoalToDOM)
});
```

Now our result render correctly when we update the new state:

Until now, we have done two action, that are add and remove. We still have one more action to consider that is ***toggleTodoAction().*** We will add some code like image below and see what happen

```
function addTodoToDOM(todo) {
    const node = document.createElement('li');
    const text = document.createTextNode(todo.name);

    node.appendChild(text);
    node.style.textDecoration = todo.complete ? 'line-through' : 'none';
    node.addEventListener('click', () => {
        store.dispatch(toggleTodoAction(todo.id))
    })
    document.getElementById('todos').appendChild(node);
}
```

Then we can mark whatever todo item that we have done

Here's the commit with the changes made in the discuss above.

Remove Items

The next functionality we need to add is being able to remove any one of these items. We will create a little helper function that is called ***createRemoveButton*** function.

```
function createRemoveButton (onClick) {
    const removeBtn = document.createElement('button');
    removeBtn.innerHTML = 'X';
    removeBtn.addEventListener('click', onClick);

    return removeBtn;
}
```

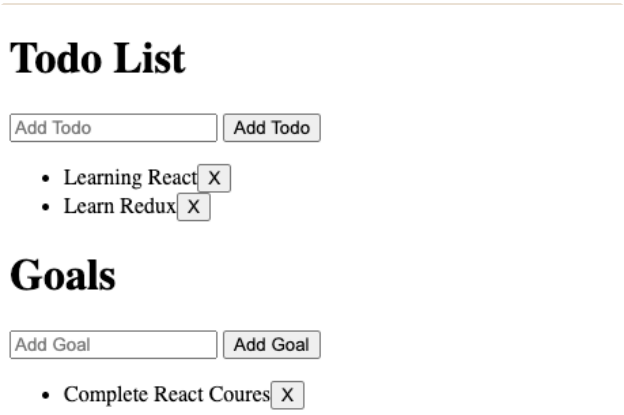
Then we go ahead to ***addTodoToDOM()*** and add the code like this

```
function addTodoToDOM(todo) {
    const node = document.createElement('li');
    const text = document.createTextNode(todo.name);

    const removeBtn = createRemoveButton(() => {
        store.dispatch(removeTodoAction(todo.id))
    })

    node.appendChild(text);
    node.appendChild(removeBtn);
    node.style.textDecoration = todo.complete ? 'line-through' : 'none';
    node.addEventListener('click', () => {
        store.dispatch(toggleTodoAction(todo.id))
    })
    document.getElementById('todos').appendChild(node);
}
```

Now we will do the same with ***addGoaltoDOM()***

```
function addGoalToDOM(goal) {
    const node = document.createElement('li');
    const text = document.createTextNode(goal.name);

    const removeBtn = createRemoveButton(() => {
        store.dispatch(removeGoalAction(goal.id));
    })

    node.appendChild(text);
    node.appendChild(removeBtn);
    document.getElementById('goals').appendChild(node);
}
```

Our UI will be look like now



Here's the commit with the changes made in the discuss above.

Summary

In this section, we connected our functioning state application with a front-end UI. We added some form fields and buttons to our UI that can be used to add new Todo items and Goal items to the state. Updating the state will *also* cause the entire application to re-render so that the visual representation of the application matches that of the info stored in the state object.

This is Redux

We're going to transition away from our custom code to using the actual Redux library.

> <script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.1.2/redux.min.js"></script>

Adding In Redux

The first step, we will add the script above into head tags in ***index.html*** file

```
<head>
    <title>UI Todos Goals</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.1.2/redux.min.js"></script>
</head>
```

Then we will delete all of our library codes specifically our ***createStrore()*** function

```
// Library Code
function createStore(reducer) {
    // The store should have four parts
    // 1. The state
    // 2. Get the state
    // 3. Listen to changes on the state
    // 4. Update the state

    let state;
    let listeners = [];

    const getState = () => state;

    const subscribe = (listener) => {
        listeners.push(listener);
        return () => {
            listeners = listeners.filter((l) => l !== listener);
        };
    };

    const dispatch = (action) => {
        state = reducer(state, action);
        listeners.forEach((listener) => listener());
    };

    return {
        getState,
        subscribe,
        dispatch,
    };
}
```

Delete all of this code

Next step, We will call Redux.createStore instead of calling our create store function. After you modify it, everything still works the same result.

```
const store = Redux.createStore(app);
```

Furthermore, there's one more thing that we can change now that is app(), which we have built before. We will delete or comment that function and using Redux.combineReducers() to do the same function of app().

```
// function app(state = {}, action) {
//     return {
//         todos: todos(state.todos, action),
//         goals: goals(state.goals, action),
//     };
// }
```

After that we will add Redux.combineReducers() like this

```
const store = Redux.createStore(Redux.combineReducers(
    {
        todos,
        goals,
    }
));
```

Here's the commit with the changes made in the discuss above.

Reducer composition sounds intimidating, but it's simpler than you might think. The idea is that you can create a reducer to manage not only each section of your Redux store, but also any nested data as well. Let's say we were dealing with a state tree like had this structure

```
{
  users: {},
  setting: {},
  tweets: {
    btyxlj: {
      id: 'btyxlj',
      text: 'What is a jQuery?',
      author: {
        name: 'Tyler McGinnis',
        id: 'tylermcginnis',
        avatar: 'twt.com/tm.png'
      }
    }
  }
}
```

We have three main properties on our state tree: users, settings, and tweets. Naturally, we'd create an individual reducer for both of those and then create a single root reducer using Redux's `combineReducers` method.

```
const reducer = combineReducers({
  users,
  settings,
  tweets,
});
```

`combineReducers`, under the hood, is our first look at reducer composition. `combineReducers` is responsible for invoking all the other reducers, passing them the portion of their state that they care about. We're making one root reducer, by composing a bunch of other reducers together. With that in mind, let's take a closer look at our tweets reducer and how we can leverage reducer composition again to make it more compartmentalised. Specifically, let's look how a user might change their avatar with the way our store is currently structured. Here's the skeleton with what we'll start out with:

```
function tweets(state = {}, action) {
  switch (action.type) {
    case ADD_TWEET:
    // ...
    case REMOVE_TWEET:
    //  ...
    case UPDATE_AVATAR:
    // ???
```

```
  }
}
```

What we're interested in is that last one, `UPDATE_AVATAR`. This one is interesting because we have some nested data - and remember, reducers have to be pure and can't mutate any state. Here's one approach:

```
function tweets(state = {}, action) {
  switch (action.type) {
    case ADD_TWEET:
    // ...
    case REMOVE_TWEET:
    // ...
    case UPDATE_AVATAR:
      return {
        ...state,
        [action.tweetId]: {
          ...state[action.tweetId],
          author: {
            ...state[action.tweetId].author,
            avatar: action.newAvatar,
          },
        },
      };
  }
}
```

That's a lot of spread operators. The reason for that is because, for every layer, we're wanting to spread all the properties of that layer on the new objects we're creating (because, immutability). What if, just like we separated our tweets, users, and settings reducers by passing them the slice of the state tree they care about, what if we do the same thing for our tweets reducer and its nested data. Doing that, the code above would be transformed to look like this

```
function author (state, action) {
  switch (action.type) {
      case : UPDATE_AVATAR
        return {
          ...state,
          avatar: action.newAvatar
        }
      default :
        state
  }
}

function tweet (state, action) {
  switch (action.type) {
      case ADD_TWEET :
        ...
      case REMOVE_TWEET :
        ...
      case : UPDATE_AVATAR
        return {
          ...state,
          author: author(state.author, action)
        }
      default :
        state
  }
}
```

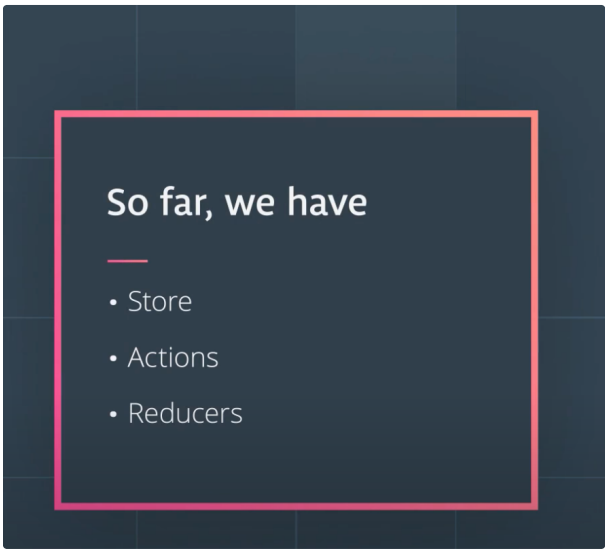
```
28
29 function tweets (state = {}, action) {
30   switch(action.type){
31       case ADD_TWEET :
32         ...
33       case REMOVE_TWEET :
34         ...
35       case UPDATE_AVATAR :
36         return {
37           ...state,
38           [action.tweetId]: tweet(state[action.tweetId], action)
39         }
40       default :
41         state
42   }
43 }
```

All we've done is separated out each layer of our nested tweets data into their own reducers. Then, just like we did with our root reducer, we're passing those reducers the slice of the state they care about.

Redux Middleware

Introduction to React Middleware

At this point, we have our store, actions, and reducers, which together make up the foundation of Redux.



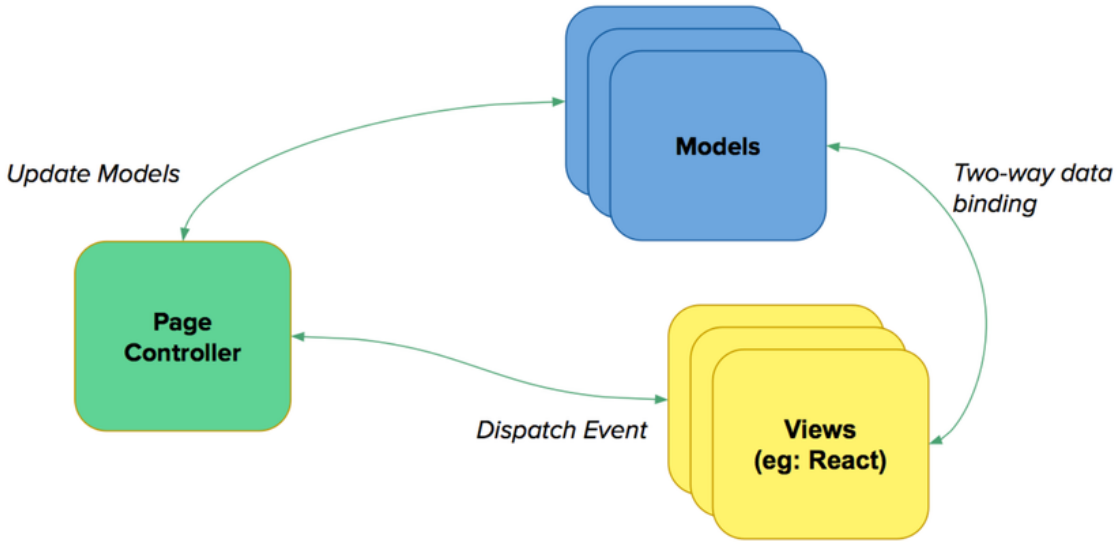
Throughout this lesson, we're going to take a look at Redux Middleware and how it allows us to hook into the Redux lifecycle and why that's beneficial.

By the end of the lesson, you will be able to:

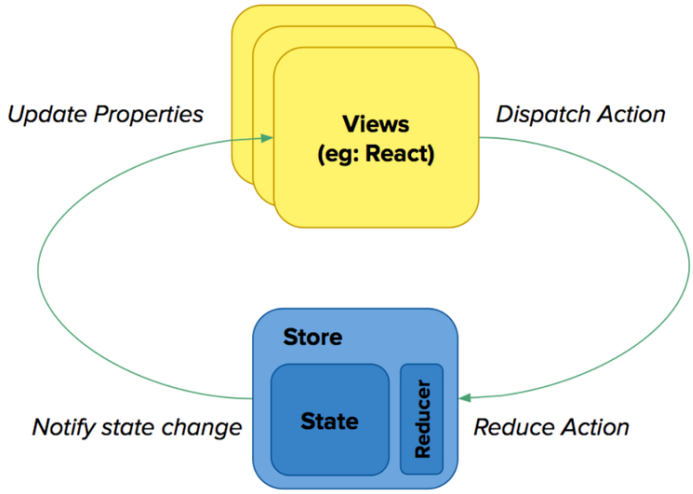
- Add middleware to the app
- Hook middleware to the Redux lifecycle using:
  - Reducers
  - Actions

Redux vs Traditional MVC

Redux's architecture was inspired by the Elm Architecture and differs from traditional MVC in the way it enforces a strict uni-direction flow (or cycle) of data from the View-tier to the Model and back again.



In traditional MVC based systems you will often find a stateful controller object which has both the Models and the Views injected into it. The controller will be responsible for fetching data from the models and passing that data through to the views for rendering; as well as binding to events from the view tier as the user interacts with the application. Depending on the exact MVC implementation the Controller may also be responsible for handling API calls and other custom business logic. As a result of all these responsibilities and dependencies, Controllers can typically be difficult to bring under test. Some MVC implementations can compound this problem further by introducing two-way data binding where the Controller is essentially side-stepped and the view and model are injected into each other. IMHO this is a classic case for simple over easy and should be avoided for long-term maintainability of software.



In contrast to MVC implementations, Redux does not require a centralised Controller, instead an application boils down to two tiers: The Store (which holds a reference to the single state object) and Views.

The view-tier has read-only access to the Store's state object; if it wishes to change the Store's state it must first dispatch an action object to the Store which will be processed by the reducer function. Redux provides a `dispatch` function which takes a single argument: an Action object whose interface enforces a `type` property.

The Store's state is modified by a Reducer function. The Reducer is a pure function which takes the current state object, and the action object, returning a the new state object. If the action does not result in a state-change, the reducer can return the current state object. These components are wrapped in Redux's `store` object which manages the single state object and has a subscription mechanism (`store.subscribe`) to notify other parts of the system when the state object has changed. Note that the state object is immutable, instead of being modified in place, the entire object is replaced with a new one when a change occurs.

This form of loose coupling and one-way data flow is what makes Redux applications both easy to predict and easy to test.

Redux doesn’t depend on React and can be used with other View abstractions or even just straight DOM manipulation

Keeping Count

The rest of this post will be dedicated to extending a simple application. The consists of a single React view component that offers a counter and two buttons. Users are able to manipulate the counter by pressing the increment and decrement buttons.

```
const Counter = (props) => {
    const { count, dispatch } = props;
    return (
        <div>
            <p>Counter value: {props.value}</p>
            <button onClick={() => dispatch({ type: "INC" })}>++</button>
            <button onClick={() => dispatch({ type: "DEC" })}>--</button>
        </div>
    );
};

export default connect((state) => state)(Counter);
```

The Counter component is connected to the Redux framework via the `connect` method which wraps the Counter component with a state-mapping function (`state => state`) - in this case we are simple accepting the `state` object from the Redux store with no changes. As a side-effect of being connected, the component also receives the `dispatch` function in its properties as well as the properties contained in the store's `state` object. (in this case, just `count`).

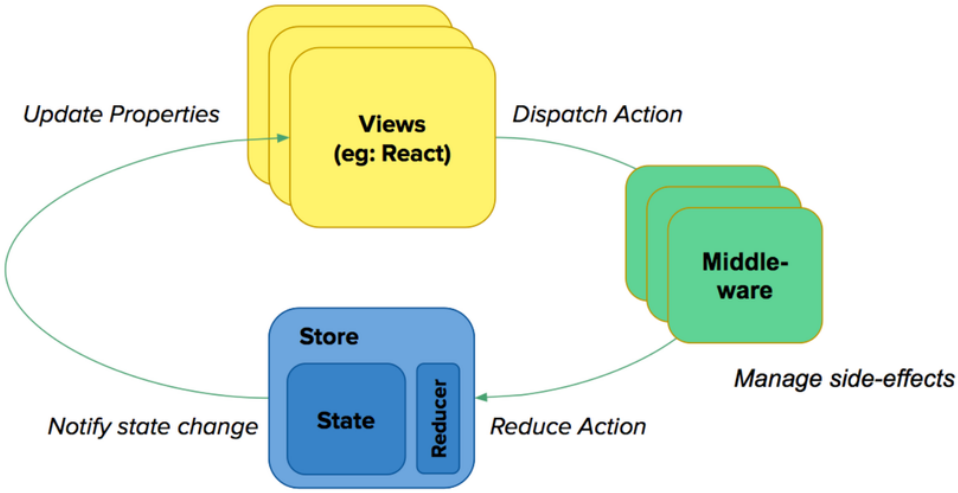
Each button invokes `dispatch` when clicked which creates a new action object with a type property of either 'INC' or 'DEC' based on how we want the counter's value to be mutated

```
const initialState = { value: 0 };
function reducer(state = initialState, action) {
    switch (action.type) {
        case "INC":
            return { ...state, value: state.value + 1 };
        case "DEC":
            return { ...state, value: state.value - 1 };
        default:
            return state;
    }
}
```

Next we have the reducer function for the application; here we simply switch on the incoming `action` object and return a new `state` value. Redux will automatically invoke your reducer function with the current `state` object whenever `dispatch` is called. Returning a new object (as is the case for both `INC` and `DEC` cases), will cause Redux to re-render the view tier thereby updating the counter's value on screen.

Bringing Back the Controller

So far our same application has been very simple and has not introduced side-effects. Side-effect is the term used when a function modifies some state elsewhere in the system, or interacts with the outside world (ie: makes an API call). Managing side-effects effectively is key to ensuring that your application stays both predictable and easy to bring under test as functions with side-effects require the reader to understand knowledge about the wider system's context and the system's state prior to the call. Redux provides a simple yet powerful mechanism for managing side-effects: Middleware.



Redux Middleware sits between the view-tier and the reducers giving you a hook to invoke asynchronous business logic after each action has been dispatched. Middleware is used to enhance a store when it is created through the `applyMiddleware` function provided by Redux:

```
const reduxStore = createStore(
    reducer,
    applyMiddleware(myMiddleware, someOtherMiddleware)
);
```

The Redux Middleware signature can look a little daunting at first with three nested functions required:

```
const myMiddleware = function (store) {
    return function (next) {
        return function (action) {
            /* middleware logic */
            return next(action);
        };
    };
};
```

The three functions are invoked right to left with the inner most function called first. These three functions each provide an essential role to your middleware:

1. The inner most function is invoked with the intercepted action - this is where your middleware's business logic will go.
2. The middle function receives a reference to the `next` piece of middleware in the chain - your middleware must invoke `next` otherwise the intercepted action will not be reduced and the app will essentially hang.
3. The outer most function receives a reference to the Redux Store API which can be used to get the current state object and dispatch new actions from your middleware.

ES6 arrow functions allow us to write this mass of functions slightly more succinctly:

```
(store) => (next) => (action) => {
    /* middleware logic */
    return next(action);
};
```

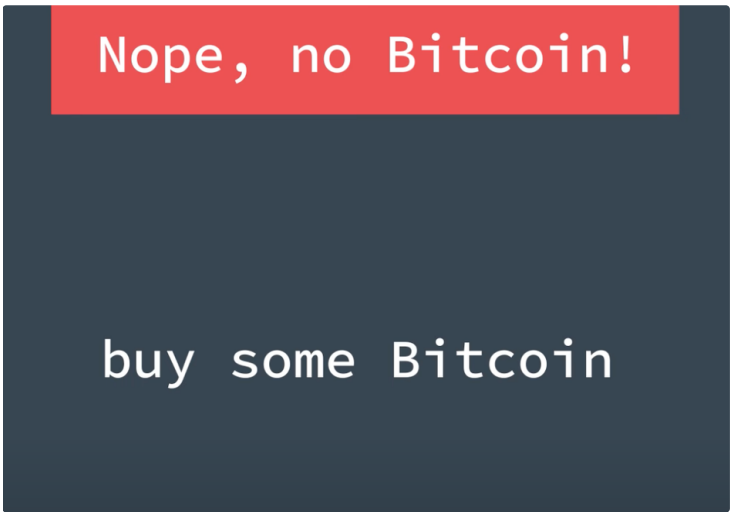
Okay, let's start getting a bit more concrete and implement the first piece of code for some custom middleware:

```
const reportMiddleware = store => next => action => {
    // Pass all actions to the next piece of middleware
    // (or the reducer if there is no other middleware).
    return next(action);
}
```

The `reportMiddleware` will intercept all actions after they've been dispatched. As per the user story, we only want to track 'INC' actions. Regardless of the type of action we must always pass the action through to `next` otherwise it will not be reduced and the app will no longer update when the user clicks on the buttons.

Customizing Dispatch

Let's go back our Todo and goals app. Assuming we have this issue, I keep wanting to invest in Bitcoin, but our financial adviser keeps insisting that is a bad idea. In fact, it's so bad that they wants us to add a new feature to our app that whenever we add a new todo item or new goal, if that goal or todo item contains the word Bitcoin, then instead of adding it to the individual list, they wants us to do alert that is a bad idea.



In order to do that, what we'll do?

We kind of want to hook into the moment after an action is dispatched, but before it ever hits our reducer and modifies the state. We will add some code like this

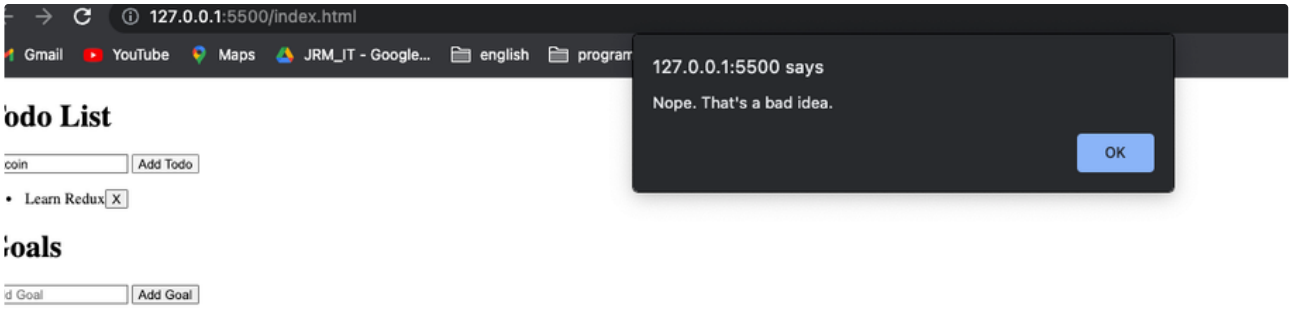
```
function checkAndDispatch(store, action) {
    if (
        action.type === ADD_TODO &&
        action.todo.name.toLowerCase().includes("bitcoin")
    ) {
        return alert("Nope. That's a bad idea.");
    }

    if (
        action.type === ADD_GOAL &&
        action.goal.name.toLowerCase().includes("bitcoin")
    ) {
        return alert("Nope. That's a bad idea.");
    }

    return store.dispatch(action);
}
```

checkAndDispatch() will take in the store as well as the action and then it's going to check the name property on the action contains the word Bitcoin. If it doesn't, then it's just going to call ***store.dispatch()*** as it normally would passing in the action and if it does, then we want to alert that warning and not do anything.

Now we have our ***checkAndDispatch(),*** so we can use this instead of the regular dispatch so that we can have these checks in place.

```
// DOM code
function addTodo() {
    const input = document.getElementById("todo");
    const name = input.value;
    input.value = "";

    checkAndDispatch(store,
        addTodoAction({
            name,
            complete: false,
            id: generateId(),
        })
    );
}

function addGoal() {
    const input = document.getElementById("goal");
    const name = input.value;
    input.value = "";

    checkAndDispatch(store,
        addGoalAction({
            id: generateId(),
            name,
        })
    );
}
```

Our result is:

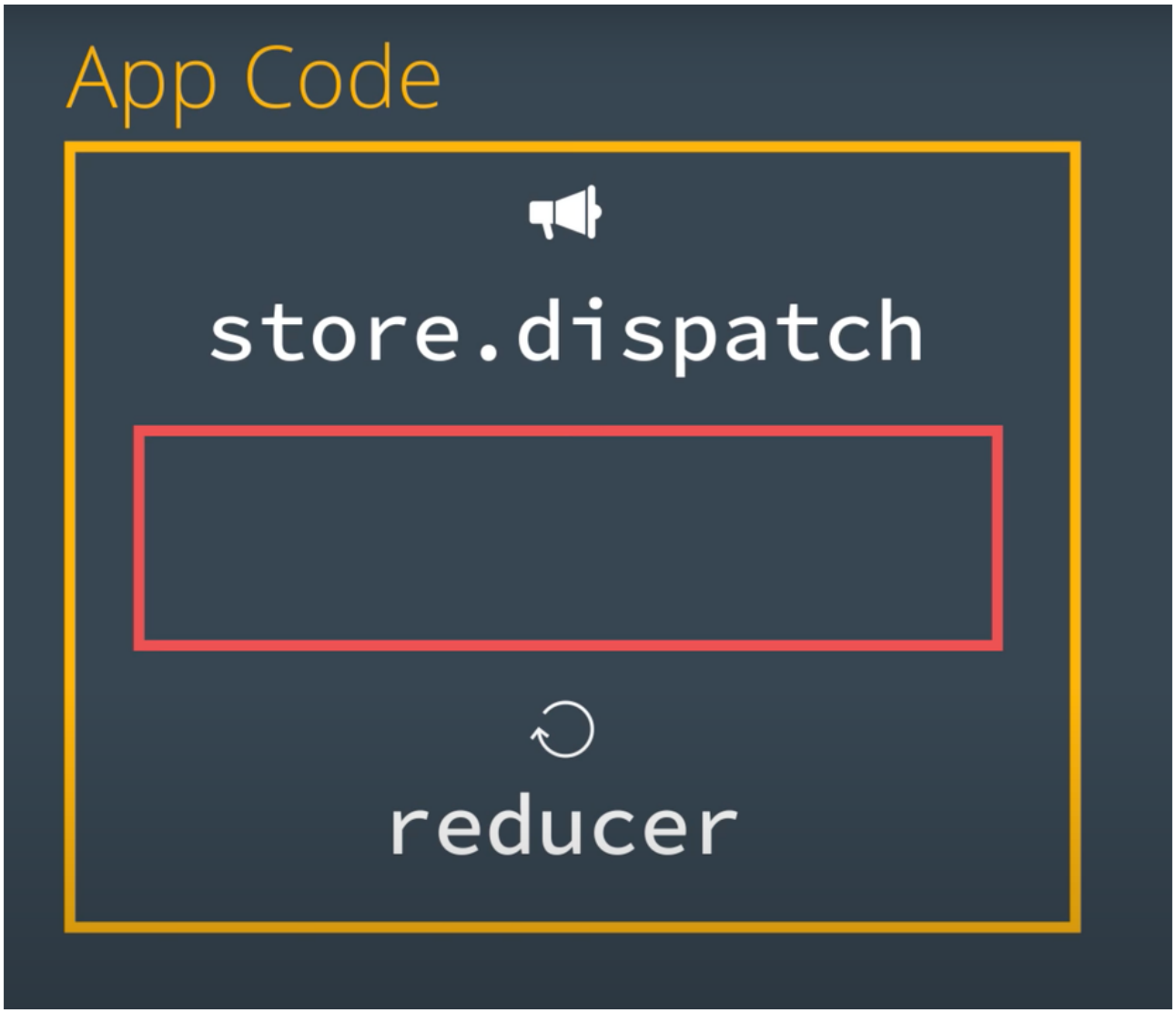


Here's the commit with the changes made in the discuss above.

React Middleware

In a standard Redux app, when ***store.dispatch()*** is called the reducer runs with our ***checkAndDispatch*** function, we need to run some verification code before the reducer run. So we run ***checkAndDispatch()*** and then if the verification passes, it runs **store.dispatch()** which calls the reducer.

For this change, we had to alter our app code works manually. Wouldn't it be nice if we could leave our ***store.dispatch()*** calls where they were and run some code between store.dispatch() and the reducer.



You've learned how Redux makes state management more predictable: in order to change the store's state, an action describing that change must be dispatched to the reducer. In turn, the reducer produces the *new* state. This new state replaces the previous state in the store. So the next time `store.getState()` is called, the new, most up-to-date state is returned.

Between the dispatching of an action and the reducer running, we can introduce code called **middleware** to *intercept* the action before the reducer is invoked. The Redux docs describe middleware as:

> …a third-party extension point between dispatching an action, and the moment it reaches the reducer.

What's great about middleware is that once it receives the action, it can carry out a number of operations, including:

- Producing a side effect (e.g., logging information about the store)
- Processing the action itself (e.g., making an asynchronous HTTP request)
- Redirecting the action (e.g., to another piece of middleware)
- Dispatching supplementary actions

…or even some combination of the above! Middleware can do any of these *before* passing the action along to the reducer.

Let's replace our `checkAndDispatch()` function with a real Redux middleware function.

```
// Middleware
function checker(store) {
    return function(next) {
        return function(action) {
            if (
                action.type === ADD_TODO &&
                action.todo.name.toLowerCase().includes("bitcoin")
            ) {
                return alert("Nope. That's a bad idea.");
            }

            if (
                action.type === ADD_GOAL &&
                action.goal.name.toLowerCase().includes("bitcoin")
            ) {
                return alert("Nope. That's a bad idea.");
            }

            return next(action);
        }
    }
}
```

Then we go to change all of checkAndDispatch parsing in the store to store.dispatch parsing in our action creator.

```
function createRemoveButton(onClick) {
    const removeBtn = document.createElement('button');
    removeBtn.innerHTML = 'X';
    removeBtn.addEventListener('click', onClick);

    return removeBtn;
}

function addTodoToDOM(todo) {
    const node = document.createElement('li');
    const text = document.createTextNode(todo.name);

    const removeBtn = createRemoveButton(() => {
        store.dispatch(removeTodoAction(todo.id));
    })

    node.appendChild(text);
    node.appendChild(removeBtn);
    node.style.textDecoration = todo.complete ? 'line-through' : 'none';
    node.addEventListener('click', () => {
        store.dispatch(toggleTodoAction(todo.id))
    })
    document.getElementById('todos').appendChild(node);
}

function addGoalToDOM(goal) {
    const node = document.createElement('li');
    const text = document.createTextNode(goal.name);

    const removeBtn = createRemoveButton(() => {
        store.dispatch(removeGoalAction(goal.id));
    })

    node.appendChild(text);
    node.appendChild(removeBtn);
    document.getElementById('goals').appendChild(node);
}
```

Next thing, we add second argument to createStore like this

```
const store = Redux.createStore(Redux.combineReducers({
    todos,
    goals,
}), Redux.applyMiddleware(checker));
```

We will refactor our `checkAndDispatch()` function by using ES6. This action will make the code look a lot cleaner.

```
// Middleware
const checker = (store) => (next) => (action) => {
    if (
        action.type === ADD_TODO &&
        action.todo.name.toLowerCase().includes("bitcoin")
    ) {
        return alert("Nope. That's a bad idea.");
    }

    if (
        action.type === ADD_GOAL &&
        action.goal.name.toLowerCase().includes("bitcoin")
    ) {
        return alert("Nope. That's a bad idea.");
    }

    return next(action);
}
```

Here's the commit with the changes made in the discuss above.

Where Middleware Fits

The way we had to structure our code originally, our `checkAndDispatch()` function *had* to run *before* `store.dispatch()`. Why is this? Because when `store.dispatch()` is invoked, it immediately calls the reducer that was passed in when `createStore()` was invoked. If you remember back to the first lesson, this is what our `dispatch()` function looked like (and is very similar to the real Redux `dispatch()` function):

```
const dispatch = (action) => {
  state = reducer(state, action);
  listeners.forEach((listener) => listener());
};
```

So you can see that calling `store.dispatch()` will *immediately* invoke the `reducer()` function. There's no way to run anything in between the two function calls. So that's why we had to make our `checkAndDispatch()` so that we can run verification code *before* calling `store.dispatch()`.

However, this isn't maintainable. If we wanted to add another check, then we'd need to write *another* preceding function, that then calls `checkAndDispatch()` that *then* calls `store.dispatch()`. Not maintainable at all.

With Redux's middleware feature, we can run code *between* the call to `store.dispatch()` and `reducer()`. The reason this works, is because Redux's version of `dispatch()` is a bit more sophisticated than ours was, and because we provide the middleware functions when we create the store.

> `const store = Redux.createStore( <reducer-function>, <middleware-functions> )`

Redux's `createStore()` method takes the reducer function as its first argument, but then it can take a second argument of the middleware functions to run. Because we set up the Redux store with knowledge of the middleware function, it runs the middleware function between `store.dispatch()` and the invocation of the reducer.

Applying Middleware

Just as the discussion above, we can implement middleware into a Redux app by passing it in when creating the store. More specifically, we can pass in the `applyMiddleware()` function as an optional argument into `createStore()`. Here's `applyMiddleware()`'s signature:

> `applyMiddleware(...middlewares)`

Note the spread operator on the `middlewares` parameter. This means that we can pass in as many different middleware as we want! Middleware is called in the order in which they were provided to `applyMiddleware()`.

We currently have the `checker` middleware applied to our app, but we'll soon add a new `logger` middleware as well. To create a Redux store that uses our `checker` middleware, we can do the following:

> `const store = Redux.createStore(rootReducer, Redux.applyMiddleware(checker));`

💡Functions Returning Functions 💡

Redux middleware leverages a concept called **higher-order functions**. A higher-order function is a function that either:

- *Accepts* a function as an argument
- *Returns* a function

Higher-order functions are a powerful programming technique that allow functions to be significantly more dynamic. You've actually already written a higher-order function in this course. The `createRemoveButton()` function is a higher-order function because the `onClick` parameter is expected to be a function (because `onClick` is set up as an event listener callback function.

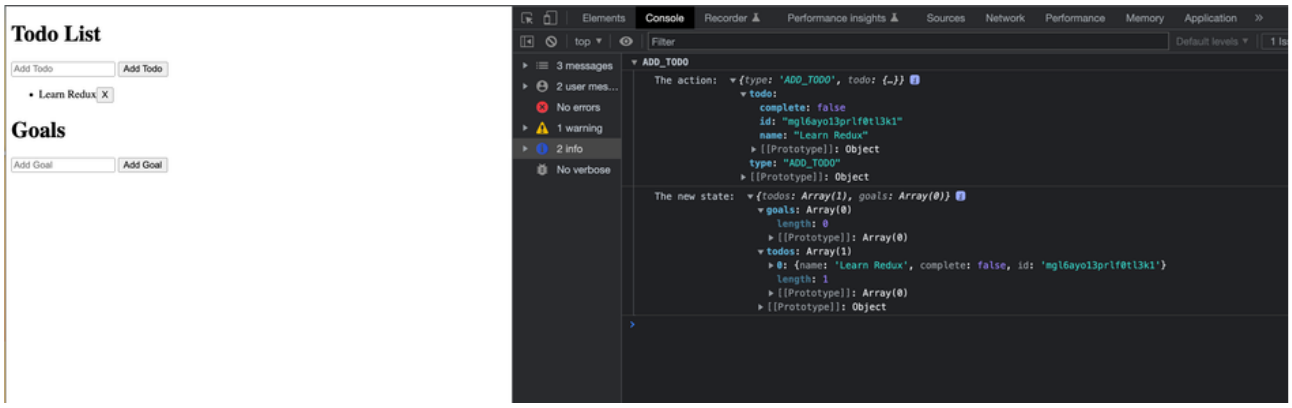
A New Middleware: Logging

Currently, our application is making use of a single middleware: `checker`. Because we can use multiple middleware functions in a single application, let's create a new middleware function called `logger` that will log out information about the state and action.

The benefits of this `logger()` middleware function are huge while developing the application. We'll use this middleware to intercept all dispatch calls and log out what the action is that's being dispatched and what the state changes to *after* the reducer has run. Being able to see this kind of information will be immensely helpful while we're developing our app. We can use this info to help us know what's going on in our app and to help us track down any pesky bugs that creep in.

```
const logger = (store) => (next) => (action) => {
    console.group(action.type);
    console.log("The action: ", action);
    const result = next(action);
    console.log("The new state: ", store.getState());
    console.groupEnd();
    return result;
}

const store = Redux.createStore(Redux.combineReducers({
    todos,
    goals,
}), Redux.applyMiddleware(checker, logger));
```

Now you can see and track what are happening in our app on the console log of chrome



Here's the commit with the changes made in the discuss above.

Further Research

The following might be a bit advanced at this point, but give them a quick read through right now and definitely bookmark them to come back and read later:

- Middleware Docs
- API for Redux's Middleware

(Doc) Week 6:

Redux with React

Introduction

One of the great things about Redux is that we can integrate it into just about any UI. This includes app built with React, Vue, HTML, or even vanilla Javascript

Lesson Outline

By the end of this lesson, you will be take the first steps to use Redux in apps that use React UI by:

- Converting our plain HTML application to one that uses React components
- Improved the code's organization by breaking out separate parts into reusable chunks

React as the UI

We're going to move away from our application being plain HTML and convert it to being powered by React. To do that, we'll need to add a number of libraries:

- react
- react-dom
- babel

Here are the packages that we'll be adding in the next discussion:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.1.2/redux.min.js"></script>
<script src="https://unpkg.com/react@17.0.2/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@17.0.2/umd/react-dom.development.js"></script>
<script src="https://unpkg.com/@babel/standalone@7.17.6/babel.min.js"></script>
```

Adding in React

We will import React, ReactDOM and babel into our plain HTML application.

```
<head>
    <title>UI Todos Goals</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.1.2/redux.min.js"></script>
    <script src="https://unpkg.com/react@17.0.2/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@17.0.2/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone@7.17.6/babel.min.js"></script>
</head>
```

Next, We are going to create a horizontal line to divide our application in half. One for the HTML application, and one for the React application.

```
<body>
    <!-- HTML application -->
    <div>
        <h1>Todo List</h1>
        <input id="todo" type="text" placeholder="Add Todo" />
        <button id="todoBtn">Add Todo</button>
        <ul id="todos"></ul>
    </div>
    <div>
        <h1>Goals</h1>
        <input id="goal" type="text" placeholder="Add Goal" />
        <button id="goalBtn">Add Goal</button>
        <ul id="goals"></ul>
    </div>

    <hr/>

    <!-- React application -->
```

Below <hr/> tag, on the React side, we will create a tag, so that React knows which element to hook into. In other words, this will be our root element. Then we will give this div an ID of app.

```
<!-- React application -->
<div id="app"></div>
```

Now we can start creating a few components for the React application. Let's create a <script> and give it the type text/babel.

```
<script type="text/babel">

</script>
```

For the components, we will make sure we use composition. First, we will create on of the child components, which will be called List and give it some props. This component will return an unordered list with a single list item.

```
<script type="text/babel">
    const List = (props) => {
        return (
            <ul>
                <li>LIST</li>
            </ul>
        )
    }
</script>
```

Now, we're also going to need the Todos component, which is going to be the parent component of list. We'll first just render TODOS and below that, we're going to render our List component.

```
const Todos = () => { return (
    <div>
        TODOS
        <List />
    </div>
    )
}
```

After that, we will do the same with Goals.

```
const Goals = () => { return (
    <div>
        GOALS
        <List />
    </div>
    )
}
```

Awesome. Along with using composition, we're going to have these two components: TODOS and GOALS. Each have components of our main app component. We will create const App and render the Todos component and Goals component. Last thing, we need to make sure that are all rendered on the page onto the DOM by ReactDOM.render().

```
const App = () => {
    return (
        <div>
            <Todos />
            <Goals />
        </div>
    )
}

ReactDOM.render(<App />, document.getElementById("app"));
```

Here's the commit with the changes made in the discuss above.

The changes we've just implemented should look pretty familiar - they were just converting parts of our app from HTML to being powered by React components.

Combining React and Redux

At this point, you've already learned React. You've built Redux and used it in a regular HTML application. But now, we've started converting that HTML to a React application.

In the following discussion, we're going to start connecting the React components to the Redux store. Please pay attention to a few things in the next image:

- Where the `store.dispatch()` code goes in a React component
- How a React component is passed the Redux store as a prop

Dispatching Todos

Next, we're going to build functionality in our Todos component in a way that it takes user input then updates to our store with data from the user input. First thing we want to do is make sure our new app component in React has access to the original store that we created. We're using this store and pass it as prop to our app component.

```
const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
}),
Redux.applyMiddleware(checker, logger)
);
```

Now we will call store into the ReactDOM.render() like this

```
ReactDOM.render(<App store={store}/>, document.getElementById("app"));
</script>
```

Now what the app is going to do with this store, that is pass that store to its Todos component as a prop. Next, we make sure passing props in the App component. For Todos it's going to grab that store from props.store

```
const App = (props) => {
    return (
        <div>
            <Todos store={props.store}/>
            <Goals />
        </div>
    )
}
```

Next thing we're going to do that is modify our UI for the Todos component a little bit and allow it to accept user input from the users. We'll take this Todos out and create a simple header. Below that, we will create a simple text input.

```
const Todos = () => { return (
    <div>
        <h1>Todo List</h1>
        <input type="text" placeholder="Add Todo" />
        <List />
    </div>
    )
}
```

For this one, we're actually not going to make it a controlled component. Recall that controlled components have the source of truth of a form controlled by a component state. In this case, we're not going to add it to a state, what we're going to do that is create a ref to be able to grab that user input.

```
const Todos = () => { return (
    <div>
        <h1>Todo List</h1>
        <input type="text" placeholder="Add Todo" ref={inputRef}/>
        <List />
    </div>
    )
}
```

The next thing, we will do is let the users submit their input, so we're going to create a quick button.

```
const Todos = () => { return (
    <div>
        <h1>Todo List</h1>
        <input type="text" placeholder="Add Todo" ref={inputRef}/>
        <button onClick={addItem}>Add Todo</button>
        <List />
    </div>
    )
}
```

Then we make sure our Todos component passing in props, this way we can access the store. Next, create a const variable under name inputRef, which will be assigned with React.useRef().

```
const Todos = (props) => {
    const inputRef = React.useRef();

    return (
        <div>
            <h1>Todo List</h1>
            <input type="text" placeholder="Add Todo" ref={inputRef}/>
            <button onClick={addItem}>Add Todo</button>
            <List />
        </div>
    )
}
```

Now we will create our event handler for adding an item, so we create const addItem method, which takes in an event and because it takes an event we're going to need to say event.preventDefault().

```
const addItem = (e) => {
    e.preventDefault();
}
```

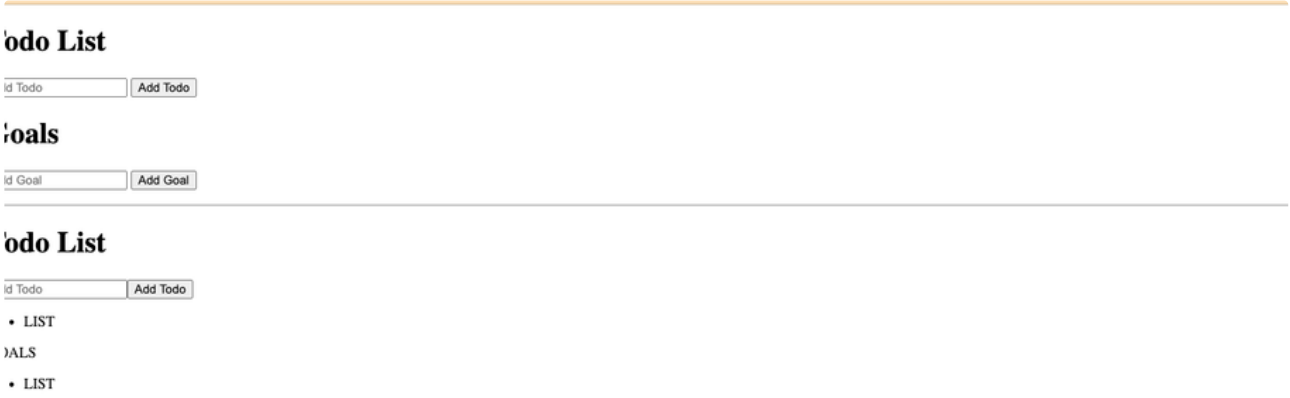
We also want to do is save the name of the to do from which we can get from inputRef.current.value. Once we grab it we also want to make sure we reset that back to an empty string, this way we can clear our input.

```
const addItem = (e) => {
    e.preventDefault();
    const name = inputRef.current.value;
    inputRef.current.value = "";
}
```

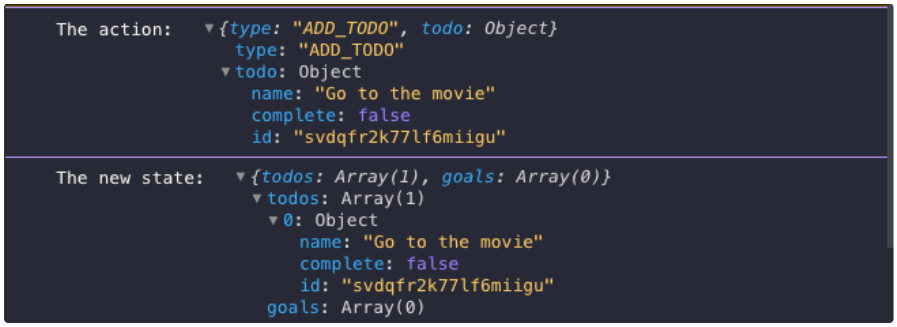
Last thing, we want to make sure our store is aware of these changes the user makes. In other words, we want the store to be able to know that the user added a single to-do item. So to do that we can say props.store.dispatch() and within dispatch we're going to invoke our action creator add to action. It will take in a single object and in this object we have three properties: name, complete, id. In this case, we're going to use a generate id function that we had created.

```
props.store.dispatch(
    addTodoAction({
        name,
        complete: false,
        id: generateId(),
    })
)
```

Now we go back browser, we can see UI of Todos task of React application look like HTML application. Keep in your mind, even though the front-end is built differently by HTML or React application, both of it will listening to the same store which defined above discussion.



You can check on the console log.



Here's the commit with the changes made in the discuss above.

In order to save time, we used the useRef hook and an uncontrolled component for our input field

When to Use Refs

The docs outline a few good use cases for `ref`s:

- Managing focus, text selection, or media playback
- Triggering imperative animations
- Integrating with third-party DOM libraries

Let's take a look at a similar example. Consider the following component, `MyComponent`:

```
1  import { useEffect, useRef } from "react";
2
3  function MyComponent() {
4    const paragraphRef = useRef();
5
6    useEffect(() => {
7      console.log(paragraphRef.current);
8    }, []);
9
10   return <p ref={paragraphRef}>Hello from the paragraph!</p>;
11 }
12
13 export default MyComponent;
```

Quite a bit is going on, so let's break things down step by step:

First, we invoke the `useRef()` hook and save the value to a variable: `paragraphRef`. This variable will be used to access the entire `<p>Hello from the paragraph!</p>` element later on.

Then, for the `ref` prop on the actual element, we set its value to `paragraphRef`. When the components mounts to the DOM, the `useEffect` hook will be called, which will log the current value of `paragraphRef` to the console: `<p>Hello from the paragraph!</p>`. As such, a `ref` has been successfully used to allow us access to DOM elements directly!

Dispatching Goals

Now we will do the same for the goals component.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();
        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addGoalAction({
                name,
                id: generateId(),
            })
        )
    }

    return (
    <div>
        <h1>Goals</h1>
            <input type="text" placeholder="Add Goal" ref={inputRef}/>
            <button onClick={addItem}>Add Goal</button>
        <List />
    </div>
    )
}

const App = (props) => {
    return (
        <div>
            <Todos store={props.store}/>
            <Goals store={props.store}/>
        </div>
    )
}
```

Here's the commit with the changes made in the discuss above.

Force Load App

If you remember, back when we built our JavaScript app, what we did was called store out subscribe. We did that to be notified whenever the store changed. Then whenever the store did change, what we did was we got our new goals and our new to-dos. Then for each those items, we added them to the DOM. We made it happen by invoking our add goal to DOM function and also invoking our add todo to DOM function.

```
store.subscribe(() => {
    const {
        todos,
        goals
    } = store.getState();

    document.getElementById('todos').innerHTML = '';
    document.getElementById('goals').innerHTML = '';

    todos.forEach(addTodoToDOM);
    goals.forEach(addGoalToDOM)
});

// DOM code
function addTodo() {
    const input = document.getElementById("todo");
    const name = input.value;
    input.value = "";

    store.dispatch(addTodoAction({
        name,
        complete: false,
        id: generateId(),
    }));
}

function addGoal() {
    const input = document.getElementById("goal");
    const name = input.value;
    input.value = "";

    store.dispatch(addGoalAction({
        id: generateId(),
        name,
    }));
}
```

However, the thing with React is that you don't actually need to do any of the DOM stuff because React is just handling all that by itself. And for us behind the scenes, that's one of the big benefits of using React in the first place. But at the same time, we also wanted to subscribe to the store inside of our React component. But this time, instead of adding items to the DOM, what we want todo is just re-render our components. Now we will use useEffect hook to implementing this one.

Then we call store.subscribe. Again stores coming from our props. We can create todos as well as the goals which are coming from store.getState().Then what we can do is pass down each of those down as a prop to the specific component.

```
const App = (props) => {

    React.useEffect(() => {
        props.store.subscribe(() => {

        })
    },[])

    const {todos, goals} = props.store.getState();

    return (
        <div>
            <Todos todos={todos} store={props.store}/>
            <Goals goals={goals} store={props.store}/>
        </div>
    )
}
```

Now we create state for the app. We keep it a simple and we have the initial value of this value just be a number. Now within subscribe, we can set the new value of that value to be incremented by 1. This will cause a re-render of the specific component. Because of the way React works, all of our child components will re-render as well, getting us the updated UI.

```
const App = (props) => {
    const [value, setValue] = React.useState(0);

    React.useEffect(() => {
        props.store.subscribe(() => {
            setValue((value) => value + 1);
        })
    },[])
```

Lists with React

In the step above, we can force a re-render whenever that state changes in that store. Now, what we want to do is build our list component. First thing, we have new information passed it as props, let't go ahead and make an accessible. We can add an items prop here and then point that over to props and goals.

```
<List items={props.goals}/>
```

Then we can do the same thing for our todos as well

```
<List items={props.todos}/>
```

Now we go back List component. Instead of rendering a list item that just says the word LIST. Let's actually make sure we render our goals or our todos. We can use map() over items and just create new list items form it. First, we're going to check that props and items exist, meaning that we do have items. If we do have those items, we can just map them out.

```
<script type="text/babel">
    const List = (props) => { return (
        <ul>
            {
                props.items && props.items.map()
            }
        </ul>
        )
    };
```

We'll call each item an item. For now, what we will return is a list item with just item's name.

```
<script type="text/babel">
    const List = (props) => { return (
        <ul>
            {
                props.items && props.items.map(item => (
                    <li key={item.id}>
                        <span>{item.name}</span>
                    </li>
                ))
            }
        </ul>
        )
    };
```

With the React, when you're mapping over something, you always want to make sure each individual list item has a unique key. In this case, we use item.id.

Now we go back browser and see our result.

Todo List

Add Todo | Add Todo

- Complete Redux X

Goals

Add Goal | Add Goal

- Learn Redux X

Todo List

Add Todo | Add Todo

- Complete Redux

Goals

Add Goal | Add Goal

- Learn Redux

As you can see, we can display list items now. The next thing we will do that is remove the item in React application.

We will add button, which is the same button in HTML application.

```
<script type="text/babel">
    const List = (props) => { return (
        <ul>
            {
                props.items && props.items.map(item => (
                    <li key={item.id}>
                        <span>{item.name}</span>
                        <button>X</button>
                    </li>
                ))
            }
        </ul>
        )
    };
```

Next thing, we'll make sure we have a handler for it. We create removeItem in todos components and call it like this.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();
        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addTodoAction({
                name,
                complete: false,
                id: generateId(),
            })
        )
    }

    const removeItem = todo => {
        props.store.dispatch(removeTodoAction(todo.id));
    }

    return (
        <div>
            <h1>Todo List</h1>
            <input type="text" placeholder="Add Todo" ref={inputRef}/>
            <button onClick={addItem}>Add Todo</button>
            <List remove={removeItem} items={props.todos}/>
        </div>
    )
}
```

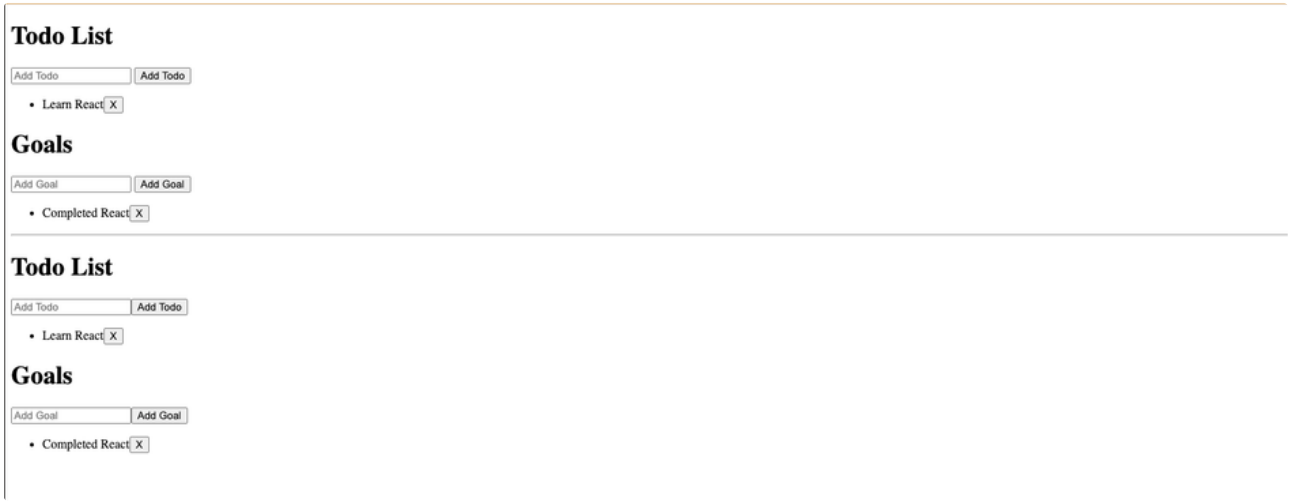
We also do the same with Goals component.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();
        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addGoalAction({
                name,
                id: generateId(),
            })
        )
    }

    const removeItem = goal => {
        props.store.dispatch(removeGoalAction(goal.id));
    }

    return (
    <div>
        <h1>Goals</h1>
            <input type="text" placeholder="Add Goal" ref={inputRef}/>
            <button onClick={addItem}>Add Goal</button>
        <List remove={removeItem} items={props.goals}/>
    </div>
    )
}
```

The last thing we need to do that is invoke that during a click event.

```
<script type="text/babel">
    const List = (props) => { return (
        <ul>
            {
                props.items && props.items.map(item => (
                    <li key={item.id}>
                        <span>{item.name}</span>
                        <button onClick={() => props.remove(item)}>X</button>
                    </li>
                ))
            }
        </ul>
        )
    };
```

The browser result look like the image below now and we can also remove the item.



Here's the commit with the changes made in the discuss above.

Toggling UI

At this point, both our plain HTML Javascript application and a React application are pretty much the same, except one thing. We are missing one functionality, which is toggle function. In this section, we will build this function.

The first thing we do that is creating a new function called toggle item in Todos component. Then we will use dispatch to call a new action, passing in the ID of the item to be toggled.

```
const toggleItem = (id) => {
    props.store.dispatch(toggleTodoAction(id));
}
```

Great. Now, we can pass that new function to our list component.

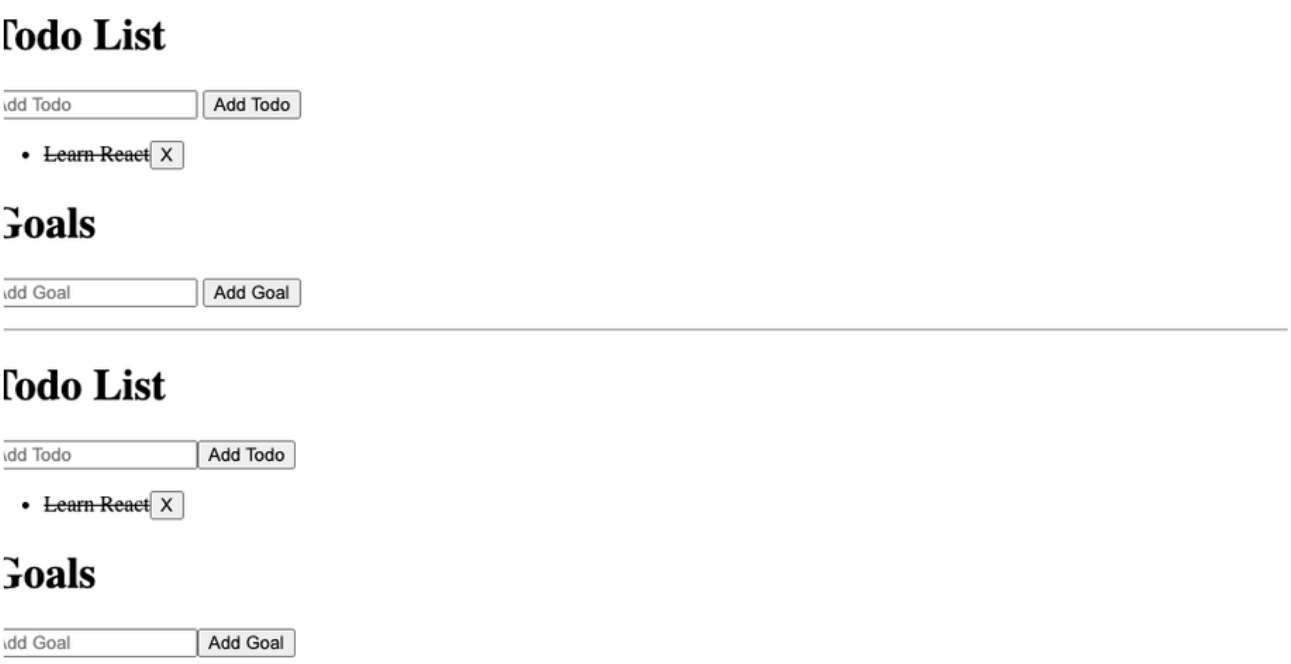
```
<List toggle={toggleItem} remove={removeItem} items={props.todos}/>
```

The last thing we need to do is create a new span that listens for a click event, then modifies that span with a strike-through. Let's go back to our list component.

Actually, instead of us creating a new span, we'll just modify the one that currently exists, and just give it some more functionality. So we can give it onClick() and style on it.

```
const List = (props) => { return (
    <ul>
        {
            props.items && props.items.map(item => (
                <li key={item.id}>
                    <span
                        style = {{
                            textDecoration: item.complete ? "line-through" : "none",
                        }}
                        onClick = {() => props.toggle && props.toggle(item.id)}
                    >{item.name}
                    </span>
                    <button onClick={() => props.remove(item)}>X</button>
                </li>
            ))
        }
    </ul>
    )
};
```

Let's go back browser and see what result we did above.



Now, we will comment the HTML and Javascript application. It mean we will only use application.

```
<body>
    <!-- HTML application -->
    <!-- <div>
        <h1>Todo List</h1>
        <input id="todo" type="text" placeholder="Add Todo" />
        <button id="todoBtn">Add Todo</button>
        <ul id="todos"></ul>
    </div>
    <div>
        <h1>Goals</h1>
        <input id="goal" type="text" placeholder="Add Goal" />
        <button id="goalBtn">Add Goal</button>
        <ul id="goals"></ul>
    </div>

    <hr/>  -->
```

```
    // store.subscribe(() => {
    //     const {
    //         todos,
    //         goals
    //     } = store.getState();

    //     document.getElementById('todos').innerHTML = '';
    //     document.getElementById('goals').innerHTML = '';

    //     todos.forEach(addTodoToDOM);
    //     goals.forEach(addGoalToDOM)
    // });

    // // DOM code
    // function addTodo() {
    //     const input = document.getElementById("todo");
    //     const name = input.value;
    //     input.value = "";

    //     store.dispatch(addTodoAction({
    //         name,
    //         complete: false,
    //         id: generateId(),
    //     }));
    // }

    // function addGoal() {
    //     const input = document.getElementById("goal");
    //     const name = input.value;
    //     input.value = "";

    //     store.dispatch(addGoalAction({
    //         id: generateId(),
    //         name,
    //     }));
    // }

    // document.getElementById("todoBtn").addEventListener("click", addTodo);

    // document.getElementById("goalBtn").addEventListener("click", addGoal);
```

```
// function createRemoveButton(onClick) {
//     const removeBtn = document.createElement('button');
//     removeBtn.innerHTML = 'X';
//     removeBtn.addEventListener('click', onClick);

//     return removeBtn;
// }

// function addTodoToDOM(todo) {
//     const node = document.createElement('li');
//     const text = document.createTextNode(todo.name);

//     const removeBtn = createRemoveButton(() => {
//         store.dispatch(removeTodoAction(todo.id));
//     })

//     node.appendChild(text);
//     node.appendChild(removeBtn);
//     node.style.textDecoration = todo.complete ? 'line-through' : 'none';
//     node.addEventListener('click', () => {
//         store.dispatch(toggleTodoAction(todo.id))
//     })
//     document.getElementById('todos').appendChild(node);
// }

// function addGoalToDOM(goal) {
//     const node = document.createElement('li');
//     const text = document.createTextNode(goal.name);

//     const removeBtn = createRemoveButton(() => {
//         store.dispatch(removeGoalAction(goal.id));
//     })

//     node.appendChild(text);
//     node.appendChild(removeBtn);
//     document.getElementById('goals').appendChild(node);
// }
```

Now our browser look like image below, and we can add, remove as well as toggle the item.

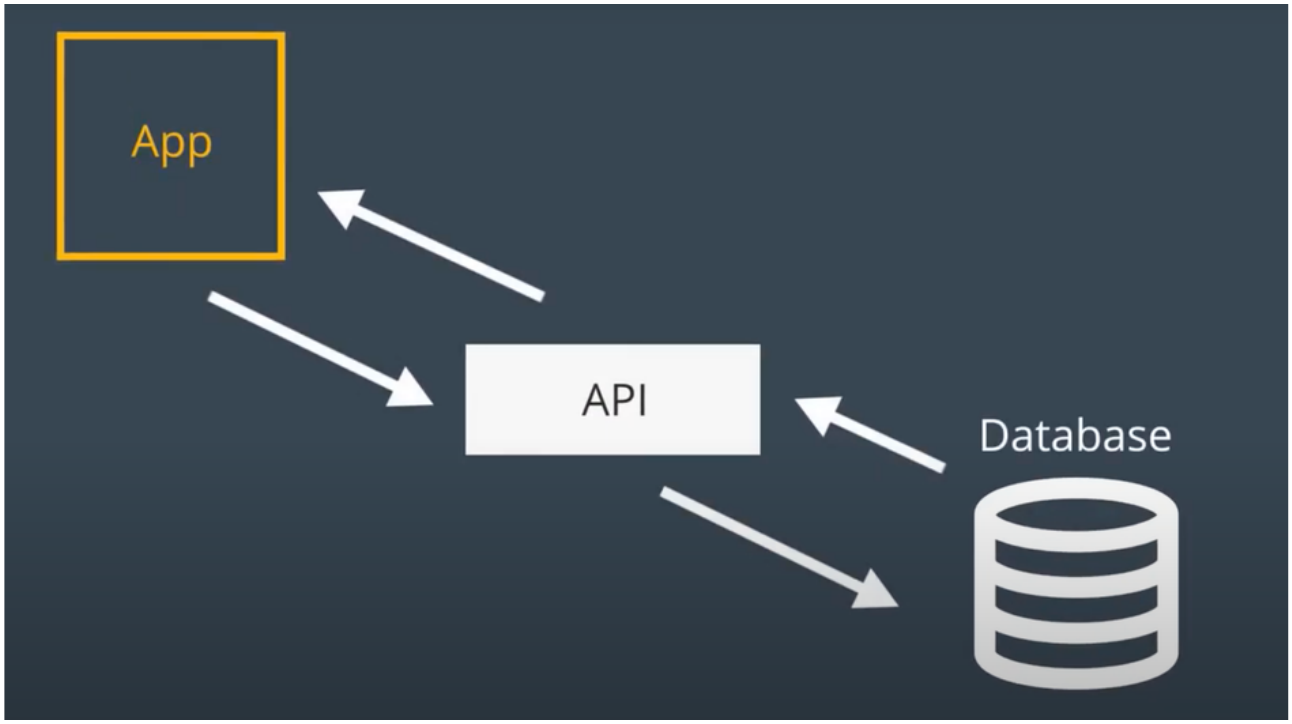
Here's the commit with the changes made in the discuss above.

Lesson Summary

In this section, we converted our plain HTML application to one using React components. We didn't implement any new features. Instead, we just improved the code's organization by breaking out parts of the application into separate, reusable chunks.

Introduction to Asynchronous Redux

At this point, although basic, our app is coming together nicely. We can add and remove different todos and goals, and that data is living inside of Redux. However, at this point, all of our data lives locally within the app itself. That isn't really realistic. In the real world, that data would likely exist in a database, and you'd interact with it through an API.



So that's what we're going to do in this lesson. We'll move all of our data to an external API; then, we'll see how Redux changes once our data becomes asynchronous.

Lesson Outline

In this lesson, we're going to be working with a (simulated) remote database. We'll use a provided API to interact with this database.

The important skill that you'll be learning in this lesson is how to make *asynchronous requests in Redux*. If you recall, the way Redux works right now is:

1. `store.dispatch()` calls are made
2. If the Redux store was set up with any middleware, those functions are run
3. The reducer is invoked

But how do we handle the case where we need to interact with an external API to fetch data. For example, what if our Todos app had a button that would load existing Todos from a database? If we dispatch that action, we currently do not have a way to wait for the list of remote Todo items to be returned.

After going through this lesson, you'll be able to make asynchronous requests and work with remote data in a Redux application.

External Data

We're going to use a database to interact with our Todos application. Note that we're *simulating* the database to keep that aspect of the project less complex.

This is the HTML script tag you need in order to add the "database" to your application.

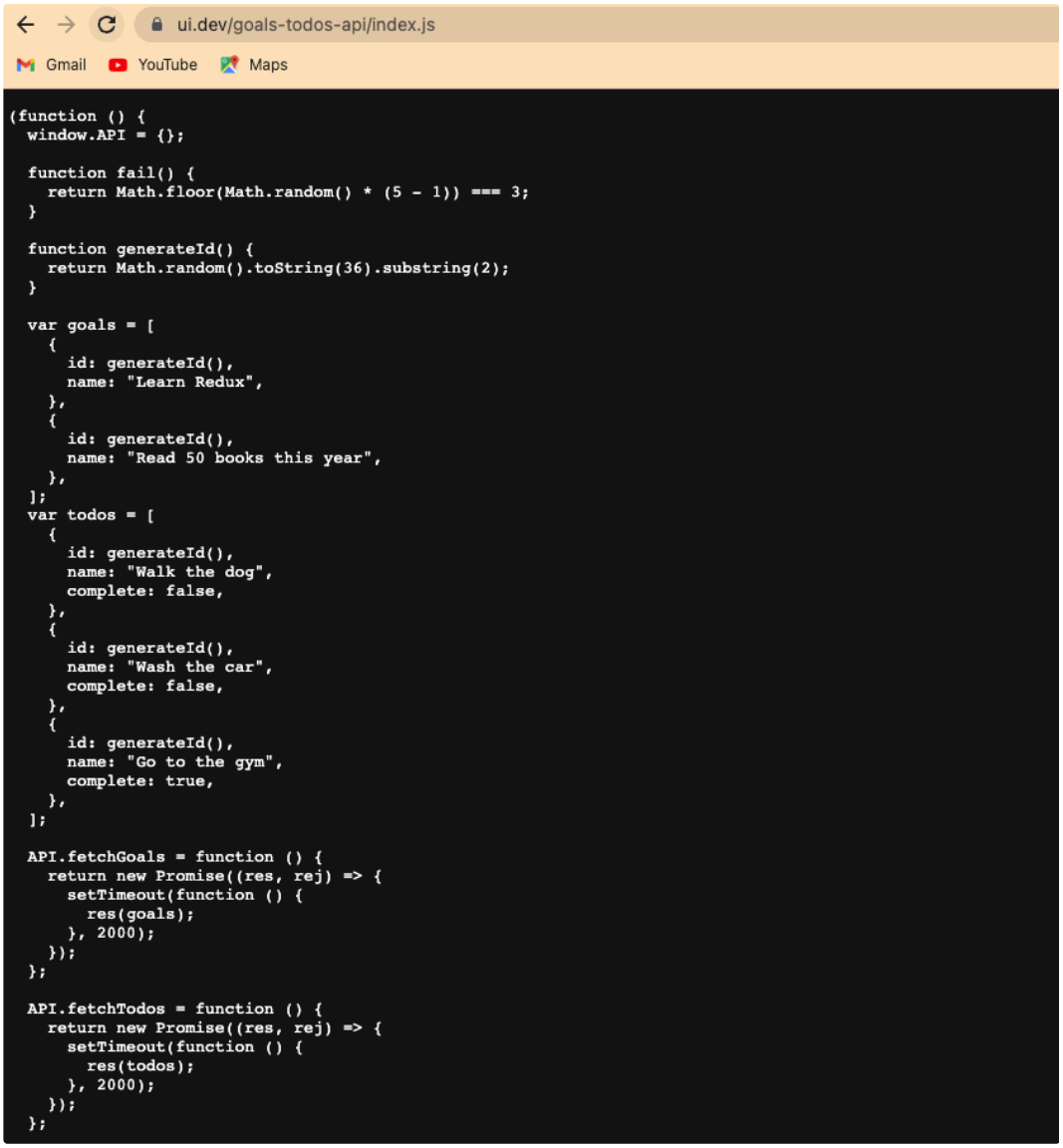
```
<script src="https://tylermcginnis.com/goals-todos-api/index.js"></script>
```

Using a Remote API

As of right now, all of the data inside our app is actually client-side only. Typically, in the real world, you'd actually be interacting with a server, and that server would go and interact with the database. Whenever you save a new to-do item or a new goal, you need to tell the server about that, and the server will tell a database about that. But right now, because we don't have a server, everything that we do is just client-side only. Now we will add the HTML script above to our code like this.

```
<head>
    <title>UI Todos Goals</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.1.2/redux.min.js"></script>
    <script src="https://unpkg.com/react@17.0.2/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@17.0.2/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone@7.17.6/babel.min.js"></script>
    <script src="https://tylermcginnis.com/goals-todos-api/index.js"></script>
</head>
```

We will pass this URL to a browser, and you can see something like this.



Let's analyse what did we have in here. First, it's going to add a new property to the window called API, and anytime that we want to interact with this data (that again is actually just living in our fake database) we would need a call, one of these methods. For example, if you wanted to fetch our goals, we'll just call API.fetchGoals.

```
API.fetchGoals = function () {
  return new Promise((res, rej) => {
    setTimeout(function () {
      res(goals);
    }, 2000);
  });
};
```

What it's going to do is return a promise that after two seconds, we'll resolve with the goals. Another example, if we want save a new to-do item, the use case that we need to handle is if that request fails (again, we're not actually going to go through the whole hassle of setting up a database and setting up a server. All we're going to do is that by including the script in our app).

```
API.saveTodo = function (name) {
  return new Promise((res, rej) => {
    setTimeout(() => {
      const todo = {
        id: generateId(),
        name: name,
        complete: false,
      };
      todos = todos.concat([todo]);
      fail() ? rej(todo) : res(todo);
    }, 300);
  });
};
```

At the present, we have access to actually all of these different methods. Ultimately, this will actually let us see how react works when a data is coming in asynchronously.

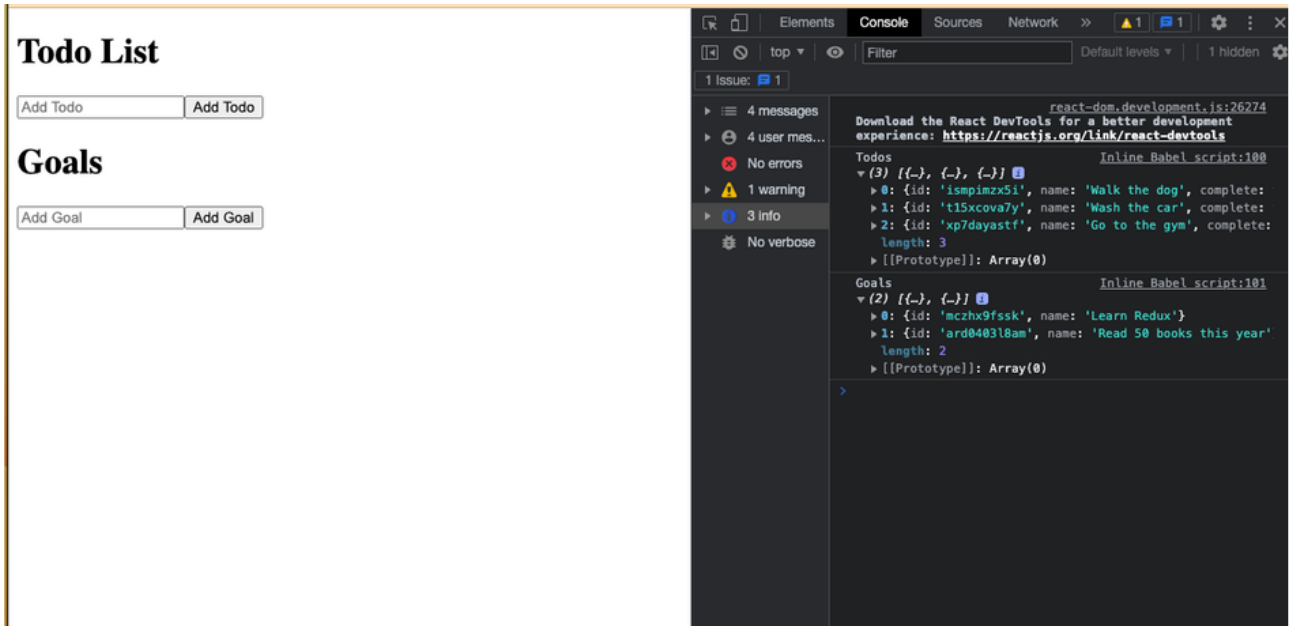
🔨Exercise Task

Add the following behaviour to the project:

- When the app loads, `console.log` all of the todos and all of the goals that reside in our fake database.

Solution Code

```
const App = (props) => {
  // ..
useEffect(() => {
Promise.all([API.fetchTodos(), API.fetchGoals()]).then(([todos, goals]) => {
console.log("Todos", todos);
console.log("Goals", goals);
});
}, []);
// ...
};
```



Promise-Based API

The methods in the provided API are all Promise-based. Let's take a look at the `.fetchTodos()` method:

```
API.fetchTodos = function () {
  return new Promise((res, rej) => {
    setTimeout(function () {
      res(todos);
    }, 2000);
  });
};
```

See how we're creating and returning a new `Promise()` object?

In the task above, you could've just fetched all of our todos and then all of our Goals, but that's serial and is just making the user wait an unnecessarily long amount of time. Since the API is Promise-based, we can use `Promise.all()` to wait until all Promises have resolved before displaying the content to the user.

Promises are asynchronous, and this lesson is all about working with asynchronous data and asynchronous requests. If you're feeling a little unsure about Promises, check out the Promise documentation on MDN.

Handling Initial Data

At this point, we need to actually tell our Redux store about this data over here. Right now we actually don't have any actions that handle adding multiple todos as well as multiple goals. To solving this problem, we will add a new action that will be called receiveDataAction. This function is going to take in the todos as well as the goals, and it'll return an object with three properties, one for the type which we can call receive_data, the second thing will return that is todos and finally, it also return the goals.

```
function receiveDataAction(todos,goals) {
    return {
        type: RECEIVE_DATA,
        todos,
        goals,
    }
}
```

If we did like this, all we need to do is actually just pass the goals as well as pass the todos. Now we will add a new constant called RECEIVE_DATA.

```
// App Code
const ADD_TODO = "ADD_TODO";
const REMOVE_TODO = "REMOVE_TODO";
const TOGGLE_TODO = "TOGGLE_TODO";
const ADD_GOAL = "ADD_GOAL";
const REMOVE_GOAL = "REMOVE_GOAL";
const RECEIVE_DATA = "RECEIVE_DATA"
```

Now how do we actually want to change the state of our store based on specific action? To start off, in regards to our todos portion of our state, whenever the receive_data action is dispatched, we will return action.todos. Because at this point, the todos array is actually going to be an empty array. Instead of an empty array, what we want to return is action.todos.

```
function todos(state = [], action) {
    switch (action.type) {
        case ADD_TODO:
            return state.concat([action.todo]);
        case REMOVE_TODO:
            return state.filter((todo) => todo.id !== action.id);
        case TOGGLE_TODO:
            return state.map((todo) =>
                todo.id !== action.id ?
                todo :
                Object.assign({}, todo, {
                    complete: !todo.complete
                })
            );
        case RECEIVE_DATA:
            return action.todos;
        default:
            return state;
    }
}
```

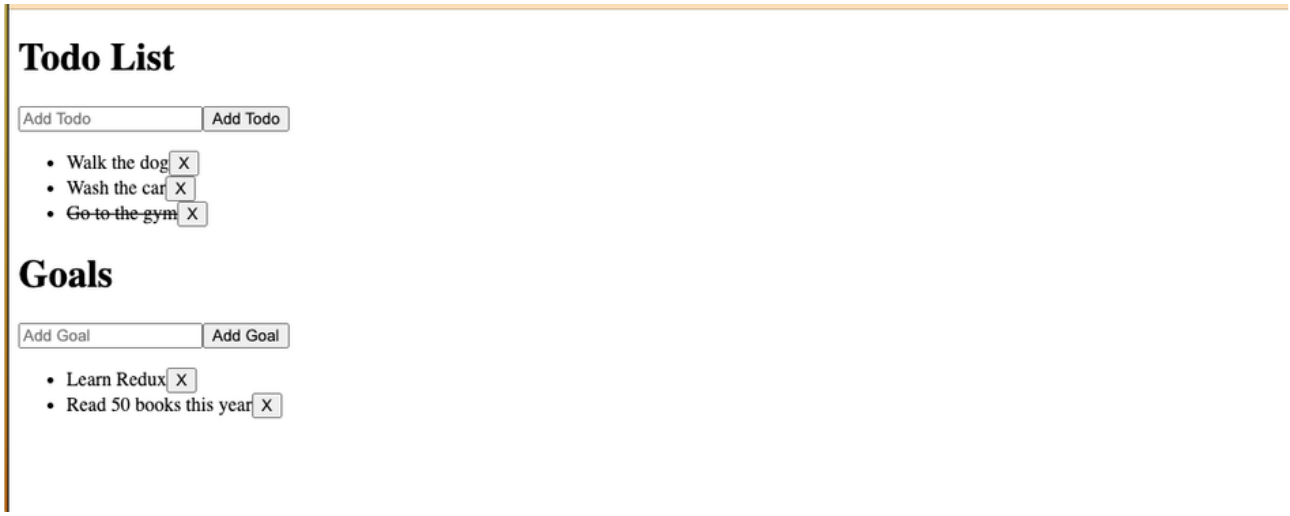
We will do the same with goals as well.

```
function goals(state = [], action) {
    switch (action.type) {
        case ADD_GOAL:
            return state.concat([action.goal]);
        case REMOVE_GOAL:
            return state.filter((goal) => goal.id !== action.id);
        case RECEIVE_DATA:
            return action.goals;
        default:
            return state;
    }
}
```

In other words, what we just did was that we had a single action type and we're leveraging this single action type to affect multiple different parts of our store. In this case, when RECEIVE_DATA is dispatched, not only are we updating the todos portion of our state, but we're also updating the goals portion of our state. Now go back the app component, what we want to do is call store.dispatch. Then we want to dispatch is our new receiveDataAction creator that we just made passing, we will passing it through todos as well as the goals.

```
const App = (props) => {
    const [value, setValue] = React.useState(0);

    React.useEffect(() => {
        props.store.subscribe(() => {
            setValue((value) => value + 1);
        })
    },[])

    React.useEffect(() => {
        Promise.all([API.fetchTodos(), API.fetchGoals()]).then(([todos, goals]) => {
            // console.log("Todos", todos);
            // console.log("Goals", goals);
            props.store.dispatch(receiveDataAction(todos, goals));
        });
    }, []);

    const {todos, goals} = props.store.getState();

    return (
        <div>
            <Todos todos={todos} store={props.store}/>
            <Goals goals={goals} store={props.store}/>
        </div>
    )
}
```

So now go back the browser and hit “Refresh“, you’ll notice that for maybe about two seconds. Then what do you see now?



The List item was showed even we haven’t done anything. But once that data actually loads, we get all our initial todo list items and our initial goals list items as well.

Here’s the commit with the changes made in the discuss above.

Loading States with Redux

At this point in the application, when we refresh the browser, there’s about a two second delay or so before our data actually renders on a page, meaning that it’s pretty blank for the todo list and the goals list until about two seconds in which they actually finally make it on the page. However, it is not necessarily a good user experience for end users. So what we will do for this discussion? while the data is actually just about the load, we want to display a loading message to the user, just for those few seconds.

To do that, we can create a new reducer, just call it “loading”. It’s going to take in the state which we can default the true, and it also take in an action. Of course, with any reducer, we’ll do a switch statement, that’s which is on action.type.

```
function loading(state = true, action){
    switch (action.type){

    }
}
```

For the first case, we already know it, it is RECEIVE_DATA. In this case, if the data is already in there, we want to return false. Otherwise, if it’s not receiving any data just yet, but when the default case to return just the state itself.

```
function loading(state = true, action){
    switch (action.type){
        case RECEIVE_DATA:
            return false;
        default:
            return state;
    }
}
```

Now at this point, because we’ve made a new reducer, we just want to make sure our store knows about that reducer.

```
const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
        loading,
}),
Redux.applyMiddleware(checker, logger)
);
```

Then we also need to grab the loading state from the store.

```
    const {todos, goals, loading} = props.store.getState();
```

Finally, we need to also just check if loading is true. If the application is indeed loading, instead of rendering the normal UI, we just render an <h3> tag or something like it. We'll keep it pretty simple. We'll just have the header, just say loading.

```
const {todos, goals, loading} = props.store.getState();

if(loading === true){
    return <h3>Loading...</h3>;
}
```

Now, we go back the browser and refresh.

Loading...

Before the data shows up on the page, we see the loading message rendered on the page. Overall it's a small fix, but it ultimately provides a better user experience for the people who use our app.

Todo List

Add Todo | Add Todo

- Walk the dog X
- Wash the car X
- ~~Go to the gym~~ X

Goals

Add Goal | Add Goal

- Learn Redux X
- Read 50 books this year X

Here's the commit with the changes made in the discuss above.

Summary

In this section, we looked at how to work with an external API. We added a new action (`RECEIVE_DATA`), created a new action creator, and built a new reducer -- all to handle the different states our app can be in while getting our remote data:

- Before the app has the data
- While the application is fetching the data
- After the data has been received

In the next section, we'll look at how to optimistically update the UI based on the API actions that are performed.

Optimistic Updates

When dealing with asynchronous requests, there will always be some delay involved. If not taken into consideration, this could cause some weird UI issues. For example, let's say when a user wants to delete a to-do item, that whole process from when the user clicks "delete" to when that item is removed from the database takes two seconds. If you designed the UI to *wait for the confirmation from the server* to remove the item from the list on the client, your user would click "delete" and then would have to wait for two seconds to see that update in the UI. That's not the best experience.

Instead what you can do is a technique called **optimistic updates**. Instead of waiting for confirmation from the server, just instantly remove the user from the UI when the user clicks "delete", then, if the server responds back with an error that the user wasn't actually deleted, you can add the information back in. This way your user gets that instant feedback from the UI, but, under the hood, the request is still asynchronous.

Optimistically Deleting Items

Currently, we have the initial data from our application, which is coming from an API. Now the next thing we want to do is that when we actually delete an item, we don't just want to update our Redux store, but what we also want to do is update the database as well. Let's head over to our todos component.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();
        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addTodoAction({
                name,
                complete: false,
                id: generateId(),
            })
        )
    }

    const removeItem = todo => {
        props.store.dispatch(removeTodoAction(todo.id));
    }

    const toggleItem = (id) => {
        props.store.dispatch(toggleTodoAction(id));
    }


    return (
        <div>
            <h1>Todo List</h1>
            <input type="text" placeholder="Add Todo" ref={inputRef}/>
            <button onClick={addItem}>Add Todo</button>
            <List toggle={toggleItem} remove={removeItem} items={props.todos}/>
        </div>
    )
}
```

Now, as I just mentioned, when the remove item method is invoked, instead of just updating in our store, we also want to update the API. One way to approach this is that we can call API. deleteTodo, passing it the todos ID. Then that's going to return a promise that when it resolves, can call props.store.dispatch. With all these changes, ow instead of just calling dispatch, we also delete the item from the database and then we call it this patch only when we get confirmation that the item has been deleted.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();
        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addTodoAction({
                name,
                complete: false,
                id: generateId(),
            })
        )
    }

    const removeItem = todo => {
        return API.deleteTodo(todo.id).then(() => {
            props.store.dispatch(removeTodoAction(todo.id));
        });
    }

    const toggleItem = (id) => {
        props.store.dispatch(toggleTodoAction(id));
    }


    return (
        <div>
            <h1>Todo List</h1>
            <input type="text" placeholder="Add Todo" ref={inputRef}/>
            <button onClick={addItem}>Add Todo</button>
            <List toggle={toggleItem} remove={removeItem} items={props.todos}/>
        </div>
    )
}
```

Let's head back into the browser and refresh. Now when we get our initial data over here, we can try deleting an item.

Todo List

Add Todo | Add Todo

- Wash the car X
- ~~Go to the gym~~ X

Goals

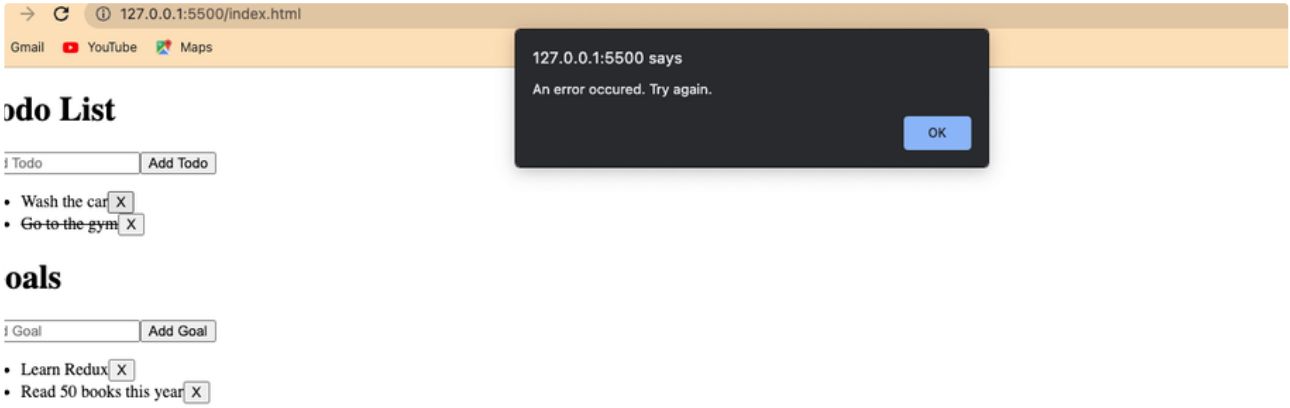
Add Goal | Add Goal

- Learn Redux X
- Read 50 books this year X

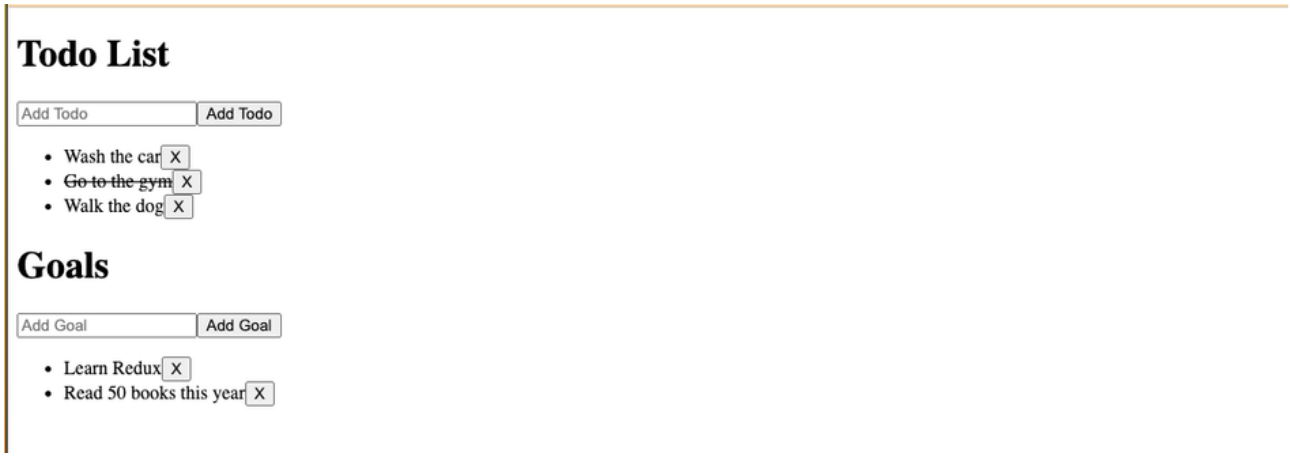
If you notice, it's pretty subtle, but there's actually a short delay between the time that you click on delete and when it's actually removed from the UI. It would be pretty nice if the user got instant feedback when they removed an item. In this case, what we can do is a little technique called optimistic updates. Basically, we don't want to wait until we get confirmation that an item has been deleted from the database before we call this patch. Instead, we want to do that immediately.

```
const removeItem = todo => {
    props.store.dispatch(removeTodoAction(todo.id));

    return API.deleteTodo(todo.id).catch(() => {
        props.store.dispatch(addTodoAction(todo));
        alert("An error occured. Try again.");
    });
}
```

Now what we're going to do now is we can first call it dispatch before we even make the API request. But moving forward, in case there's an error, we can run a catch function. What we want to do is called props.store.dispatch and Todo action. Passing the Todo. How we have it right now is that when we remove item, if there's an issue, we add the item right back. After adding the item back in, we can just come back in here and alert the user. In this case, we can say “An error occurred. Try again.”.



It's pretty still on the page, but what it says in this alert is that an error occurred, please try again. You'll notice if we hit okay, it actually adds the item back into our list over here.



We get the best of both worlds in that we're actually still updating our database, but now the user is getting instant feedback whenever they click on delete. Now let's do the exact same thing for our goals component.

```
const removeItem = goal => {
    props.store.dispatch(removeGoalAction(goal.id));

    return API.deleteGoal(goal.id).catch(() => {
        props.store.dispatch(addGoalAction(goal));
        alert("An error occured. Try again.");
    });
}
```

So now just to recap, whenever we remove an item from either our todo list or from our goals, not only does the user get instant feedback about that, but we're also updating our database with all that data.

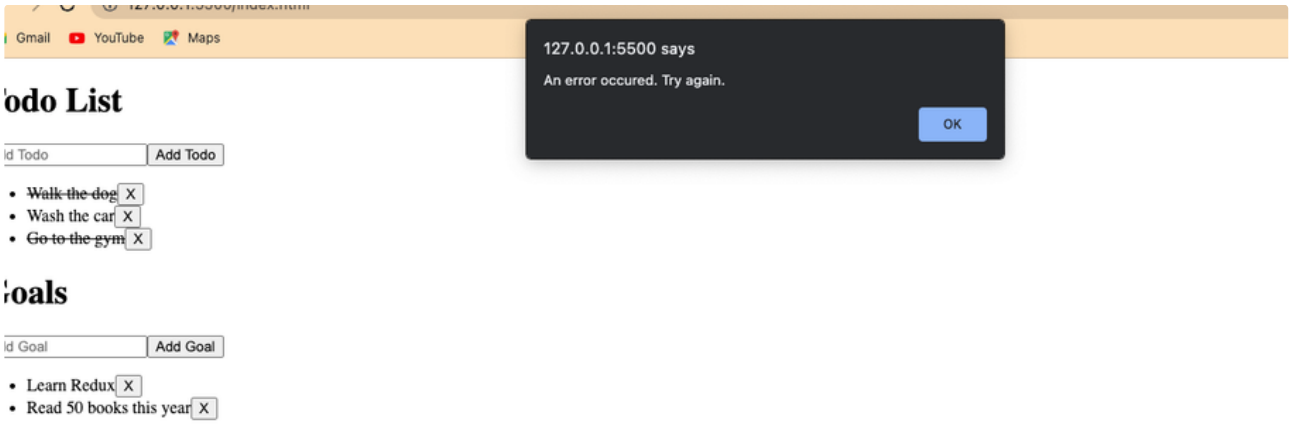
Here's the commit with the changes made in the discuss above.

Optimistically Toggling Todos

Back in our to-dos component, what we want to do now is just like we did for the remove item method. You want to do the same thing for toggling an item. Meaning that we want to make the change in the UI and also make the change in our database. We will use is the API.saveTodoToggle method. Pass in the ID. If it's an error, what we want to do here is toggle the todo back to its original state. We can call catch, then we'll call props.store.dispatch, invoke the ToggleTodoAction creator. Pass in the ID. Once that happens, we'll also alert the end-user that an error has occurred.

```
const toggleItem = (id) => {
    props.store.dispatch(toggleTodoAction(id));

    return API.saveTodoToggle(id).catch(() => {
        props.store.dispatch(toggleTodoAction(id));
        alert("An error occured. Try again.");
    });
};
```

Now we go back to a browser and hit "Refresh". Let's see if we can get the error to show up.



In this case, we've toggled it. Now we will see the line through. But there was an error that occurred, which you can see right over here in our alert. When I hit "Okay", we should actually see that todo item, walk the dog, toggled back to its original state. We will try it again. Seems like it works just fine.

What we did was first toggle the item in the UI, and then we also update our database. If there was an error, we toggle that back. Then in that case, we also alert the end-user that an error has occurred.

Here's the commit with the changes made in the discuss above.

Persisting Items

Now the only other functionality that we need to migrate to is our API is adding a new todo item as well as adding a new goal. Inside of add item function, what we can do is say API.saveTodo. Pass it the value of the new todo item which is pretty much just the name of the item which we can get by calling inputRef.current.value. Then when this function results we're going to get the new todo item. Ultimately, we can add that new todo item to the state of our store.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveTodo(inputRef.current.value).then((todo) =>{

        })

        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addGoalAction({
                name,
                id: generateId(),
            })
        )
    }
```

But first, we also have this code where on the client we're actually generating an ID and we have complete set in the false. It actually doesn't really make a whole lot of sense to generate IDs on the client for obvious reasons.

```
        props.store.dispatch(
            addGoalAction({
                name,
                id: generateId(),
            })
        )
    }
```

What we'll actually do is assume that when we invoke say, todo from an API, we'll call props.store.dispatch, pass in addTodoAction and pass into that the new “todo” item that we got from the server. Then from here what we can do is also reset the input field for the user. Finally, we can also add a catch here to just say, "There was an error, please try again."

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveTodo(inputRef.current.value).then((todo) =>{
            props.store.dispatch(addTodoAction(todo));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })

        const name = inputRef.current.value;
        inputRef.current.value = "";

        props.store.dispatch(
            addTodoAction({
                name,
                complete: false,
                id: generateId(),
            })
        )
    }
```

Now we can safely Delete all this other stuff.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveTodo(inputRef.current.value).then((todo) =>{
            props.store.dispatch(addTodoAction(todo));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })

        // const name = inputRef.current.value;
        // inputRef.current.value = "";

        // props.store.dispatch(
        //     addTodoAction({
        //         name,
        //         complete: false,
        //         id: generateId(),
        //     })
        // )
    }
```

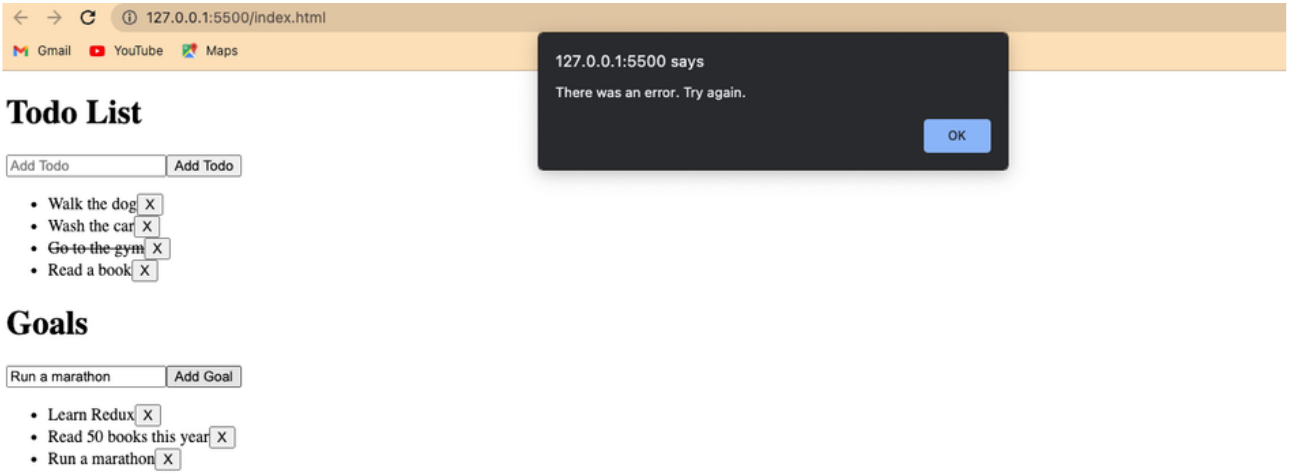
Delete all the comment line

Now, notice that we're not actually doing optimistic updates with adding an item. The only reason for that, that is with this code right here, the ID is actually being generated on the server for us. We could figure out a way to add it back to our Redux store if the request failed. But I think at this point you probably already got the idea of optimistic updates. For now, we're will Save this code, and when the request succeeds we'll just go ahead and add it back to our Redux store.

The same thing. We'll make the changes that we made in our Todos component and our goals component as well.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveGoal(inputRef.current.value).then((goal) =>{
            props.store.dispatch(addGoalAction(goal));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }
```

Now if we go back to our app, hit Refresh. We can add a new item over here let's say, "Read a book." It's pretty subtle but you might have noticed that it has a slight delay because we're actually making an asynchronous request over here and that request has to resolve before we get that updated in our UI. Then when we add a new goal we'll just say, "Run a marathon."



You can get an error here when you want add a goal or not, so if you will get an error, let's try it again.

You can now see that the item was added to our database but then when we got confirmation that it was actually added we went ahead and added it to our Redux store as well. Therefore, Save as updated,and therefore our UI is updated.

Here's the commit with the changes made in the discuss above.

Summary

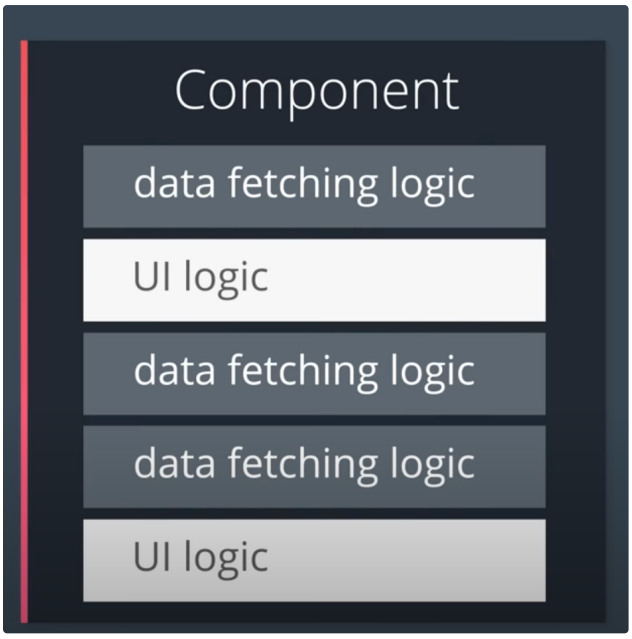
In this section, swapped more functionality over to using the API. We now use the database to:

- Remove Todos and Goals
- Toggle the state of a Todos
- Save a new Todo or Goal

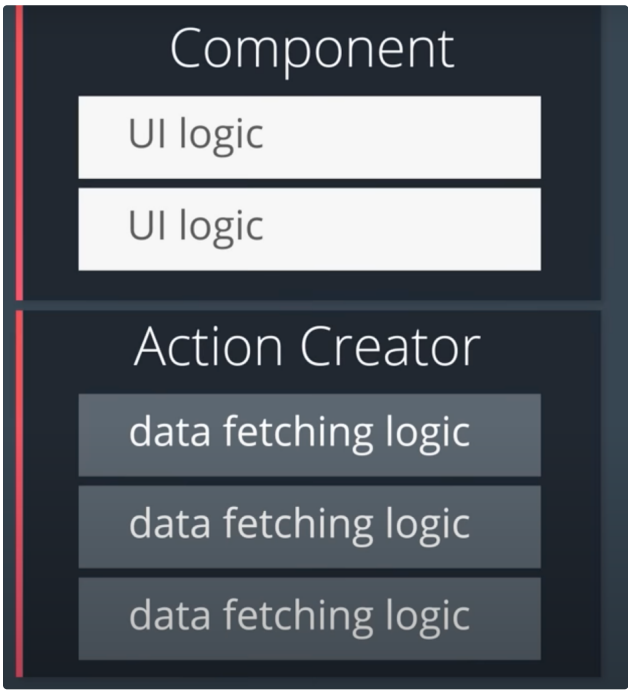
What's important is that for the removing and toggling, we're doing these actions *optimistically*. So we're assuming the change will succeed correctly on the server, so we update the UI immediately, and then only roll back to the original state if the API returns an error. Doing optimistic updates is better because it provides a more realistic and dynamic experience to the user.

Thunk

At this point, our app is working fine. We just updated it to work with asynchronous data coming from an external API. However, the way the code is organised right now, we've mixed all of our data fetching logic with our component UI logic.



Right now, our component that should just be focused on how the UI looks is also responsible for fetching data. It would be nice if we could keep those separate. Instead of calling our API and then passing that data to our action creator like we're doing now, what if we move the data fetching logic from the component to our action creator? By calling our API in an action creator, we make the action creator responsible for fetching the data it needs to create the actual action.



Moving the data fetching code here, we'll build a cleaner separation between our UI logic and our data fetching logic.

Currently, our code for removing a todo item looks like this:

Do you see how we are mixing our component-specific code with the API-specific code? If we move the data-fetching logic from our component to the action creator, our final `removeItem()` method might look like this:

This is much better! The `removeItem()` function only has one task: dispatching that a specific item needs to be deleted.

However, we need to make it so our `handleDeleteTodo()` action creator makes an asynchronous request before it returns the action. What if we just return a promise from `handleDeleteTodo()` that resolves with the action once we get the data? Well, that won't quite work; as of right now, every action creator needs to return an *object*, not a promise:

```
function asyncActionCreator (id) {
  return {
    type: ADD_USER,
    user: ??
  };
}
```

What if we used our knowledge of functional programming along with our knowledge of Redux middleware to solve this? Remember that middleware sits *between* the dispatching of an action, and the running of the reducer. The reducer expects to receive an action object, but what if, instead of returning an object, we have our action creator return a function?

We could use some middleware to check if the returned action is either a function or an object. If the action is an object, then things will work as normal - it will call the reducer passing it the action. However, if the action is a function, it can invoke the function and pass it whatever information it needs (e.g. a reference to the `dispatch()` method). This function could do anything it needs to do, like making asynchronous network requests, and can then dispatch a *different* action (that returns a regular object) when its finished.

An action creator that returns a function might look something like this:

```
function asyncActionCreator(id) {
  return (dispatch) => {
    return API.fetchUser(id).then((user) => {
      dispatch(addUser(user));
    });
  };
}
```

Notice that we're no longer returning the action itself! Instead, we're returning a function that is being passed dispatch. We then call this function when we have the data.

Now, this won't work out of the box, but there's some good news: we can add some middleware to our app to support it! Let's go ahead and see what that actually looks like.

We'll be adding the redux-thunk library in the following the discussion, so you'll need this:

```
<script src="https://unpkg.com/redux-thunk@2.2.0/dist/redux-thunk.min.js"></script>
```

Custom Thunk (Part 1)

As of right now, what we've done is we've implemented throughout our application different invocations of our API. For example, we invoked it somewhere like this.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveTodo(inputRef.current.value).then((todo) =>{
            props.store.dispatch(addTodoAction(todo));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }

    const removeItem = todo => {
        props.store.dispatch(removeTodoAction(todo.id));

        return API.deleteTodo(todo.id).catch(() => {
            props.store.dispatch(addTodoAction(todo));
            alert("An error occured. Try again.");
        });
    }


    const toggleItem = (id) => {
        props.store.dispatch(toggleTodoAction(id));

        return API.saveTodoToggle(id).catch(() => {
            props.store.dispatch(toggleTodoAction(id));
            alert("An error occured. Try again.");
        });
    };
```

Generally it works pretty well because not only is our data living locally, but it's also living and being modified in a database somewhere. Well, the problem is that once we actually started adding in all of this API code, all of our component logic got quite a bit messy. In fact, what we're doing now is that we're actually mixing our data fetching code with our API along with our component code, that is the UI code.

Now, it's not the worst thing in the world, but it would be pretty nice to keep those two things more separate. It would be pretty nice in that, in one part of our application, we have all of our data fetching logic. In our components, we'd have all our UI logic.

Let's focus on our removeItem method first. If you remember what it looked like before we added all this API code over here.

```
const removeItem = todo => {
    props.store.dispatch(removeTodoAction(todo.id));

    return API.deleteTodo(todo.id).catch(() => {
        props.store.dispatch(addTodoAction(todo));
        alert("An error occured. Try again.");
    });
}
```

For the most part it was just this line up over here. I mean the old version we did before without API.

```
const removeItem = todo => {
    props.store.dispatch(removeTodoAction(todo.id));
}
```

Again, what we had before was that when removeItem was invoked, we dispatched the removed todo action, action creator. This code was pretty easy to read and reason about, when removeItem was invoked, all we did was just remove an item from the store.

But now we have a little bit more than we have to handle here, because we're now dealing with an API. Well, what if there was a way to make our action creators a little bit more powerful? In other words, what if there's a way to encapsulate all of our API data fetching logic into action creator itself. Instead of invoking the removeTodoAction creator, but if we went ahead and just created a new action creator. Let's say if we called props.store.dispatch, and such action creator doesn't exist yet, but for now we'll call it handleDeleteTodo. Then, we just pass the todo within that.

```
const removeItem = todo => {
    props.store.dispatch(handleDeleteTodo(todo));

    props.store.dispatch(removeTodoAction(todo.id));

    return API.deleteTodo(todo.id).catch(() => {
        props.store.dispatch(addTodoAction(todo));
        alert("An error occured. Try again.");
    });
}
```

Now, what if we actually just take all of this logic like this, cut it out.

```
const removeItem = todo => {
    props.store.dispatch(handleDeleteTodo(todo));

}
```

Now our code is actually back to what it looked like before, in which all we did was just this patch of single-action. It looks pretty clean, but now the question is, how do we actually make this work? Let's go back to our action creators and create a brand new one called handleDeleteTodo.

```
function receiveDataAction(todos,goals) {
    return {
        type: RECEIVE_DATA,
        todos,
        goals,
    }
}

function handleDeleteTodo(todo){

}

function todos(state = [], action) {
    switch (action.type) {
        case ADD_TODO:
            return state.concat([action.todo]);
        case REMOVE_TODO:
            return state.filter((todo) => todo.id !== action.id);
        case TOGGLE_TODO:
            return state.map((todo) =>
                todo.id !== action.id ?
                todo :
                Object.assign({}, todo, {
                    complete: !todo.complete
                })
            );
        case RECEIVE_DATA:
            return action.todos;
        default:
            return state;
```

Again, we'll pass in the todo, and if you recall what our action creators did before, all they really did was actually just return a single object which describe the specific event that is actually occurring. We will paste in what we cut earlier, now our action creator has to do quite a few

things. First it has a dispatch, that it has to make an API request, then it has to dispatch again if it needs to.

```
function handleDeleteTodo(todo){
    props.store.dispatch(removeTodoAction(todo.id));

    return API.deleteTodo(todo.id).catch(() => {
        props.store.dispatch(addTodoAction(todo));
        alert("An error occured. Try again.");
    });
}
```

Now, what we need to do is first get access to that dispatch. Instead of returning an object like we did over here.

```
function receiveDataAction(todos,goals) {
    return {
        type: RECEIVE_DATA,
        todos,
        goals,
    }
}
```

What we can do is return a function and this function can be passed dispatch.

```
function handleDeleteTodo(todo){
    return() => {
        props.store.dispatch(removeTodoAction(todo.id));

        return API.deleteTodo(todo.id).catch(() => {
            props.store.dispatch(addTodoAction(todo));
            alert("An error occured. Try again.");
        });
    }
}
```

Now instead of saying props.store, we can just say dispatch, because now we'd be getting it from the arguments. This would actually be pretty great because this would actually let us do what we're going to do in removeItem function, where we just dispatch the single action, or the single action creator and then it'll handle all the logic for us.

```
function handleDeleteTodo(todo){
    return(dispatch) => {
        dispatch(removeTodoAction(todo.id));

        return API.deleteTodo(todo.id).catch(() => {
            props.store.dispatch(addTodoAction(todo));
            alert("An error occured. Try again.");
        });
    }
}
```

Well, what if we actually created a new custom middleware in order to support this functionality.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveTodo(inputRef.current.value).then((todo) =>{
            props.store.dispatch(addTodoAction(todo));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }

    const removeItem = todo => {
        props.store.dispatch(handleDeleteTodo(todo));
    }

    const toggleItem = (id) => {
        props.store.dispatch(toggleTodoAction(id));

        return API.saveTodoToggle(id).catch(() => {
            props.store.dispatch(toggleTodoAction(id));
            alert("An error occured. Try again.");
        });
    };
```

For example, whenever an action is dispatched, if an action is a function, then we would invoke it passing a dispatch, and if it's not, then we just dispatch as we normally would.

Custom Thunk (Part 2)

We will go to our middleware section.

```
const logger = (store) => (next) => (action) => {
    console.group(action.type);
    console.log("The action: ", action);
    const result = next(action);
    console.log("The new state: ", store.getState());
    console.groupEnd();
    return result;
}

const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
        loading,
    }),
    Redux.applyMiddleware(checker, logger)
    );
```

Right here we can say const thunk, because technically that's actually what it is. Then we'd have a weird little pattern. It'll take in a store, next, as well as an action. Then inside of this function, we can just say if action equals a function.

```
const logger = (store) => (next) => (action) => {
    console.group(action.type);
    console.log("The action: ", action);
    const result = next(action);
    console.log("The new state: ", store.getState());
    console.groupEnd();
    return result;
}

const thunk = (store) => (next) => (action) => {
    if(typeof action === 'function'){

    }
}

const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
        loading,
}),
Redux.applyMiddleware(checker, logger)
);
```

Which actually will be when we do something like this

```
function handleDeleteTodo(todo){
    return(dispatch) => {
        dispatch(removeTodoAction(todo.id));

        return API.deleteTodo(todo.id).catch(() => {
            props.store.dispatch(addTodoAction(todo));
            alert("An error occured. Try again.");
        });
    }
}
```

Where instead of returning an object as the action, we can return a function as the action. When an action is a function, what we want to do is invoke that action and pass it stored at dispatch. Let's make sure we also return it.

```
const thunk = (store) => (next) => (action) => {
    if(typeof action === 'function'){
        return action(store.dispatch);
    }
}
```

In the case that the action is not, then what we want to do is invoke next, then just pass in the action.

```
const thunk = (store) => (next) => (action) => {
    if(typeof action === 'function'){
        return action(store.dispatch);
    }

    next(action);
}
```

Because we actually have this new middleware, we need to make sure redux knows about it. So let's go to apply middleware. Then just add it right here as the first middleware interchain.

```
const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
        loading,
}),
Redux.applyMiddleware(thunk, checker, logger)
);
```

Now what happens is that when we remove a todo item

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveTodo(inputRef.current.value).then((todo) =>{
            props.store.dispatch(addTodoAction(todo));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }

    const removeItem = todo => {
        props.store.dispatch(handleDeleteTodo(todo));
    }


    const toggleItem = (id) => {
        props.store.dispatch(toggleTodoAction(id));

        return API.saveTodoToggle(id).catch(() => {
            props.store.dispatch(toggleTodoAction(id));
            alert("An error occured. Try again.");
        });
    };
```

We will also dispatch handled leaked todo and then that function encapsulates all its functionality inside of it. This way, we're no longer mixing our data fetching logic with our UI logic.

If We go back to our browser, refresh, then we go ahead and delete todo item. Everything is actually still working normally. But again, instead of having that logic living inside of our component, it's now tucked away inside of our new action creator. As it turns out, this pattern is actually so common that there's a library we can include right here in our script tag. Its called a Redux Thunk.

```
<head>
    <title>UI Todos Goals</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/redux/4.1.2/redux.min.js"></script>
    <script src="https://unpkg.com/react@17.0.2/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@17.0.2/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone@7.17.6/babel.min.js"></script>
    <script src="https://tylermcginnis.com/goals-todos-api/index.js"></script>

    <script src="https://unpkg.com/redux-thunk@2.2.0/dist/redux-thunk.min.js"></script>
</head>
```

Now what we can do is instead of using our own thunk middleware that we just created, we can actually go ahead and remove everything we just typed.

```
const logger = (store) => (next) => (action) => {
    console.group(action.type);
    console.log("The action: ", action);
    const result = next(action);
    console.log("The new state: ", store.getState());
    console.groupEnd();
    return result;
}

// const thunk = (store) => (next) => (action) => {
//     if(typeof action === 'function'){
//         return action(store.dispatch);
//     }

//     next(action);
// }

const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
        loading,
}),
Redux.applyMiddleware(thunk, checker, logger)
);
```

Delete all the comment line

Then we go to apply middleware and instead of using our own thunk, we can just say Redux Thunk, then default, hit "Save".

```
const store = Redux.createStore(
    Redux.combineReducers({
        todos,
        goals,
        loading,
}),
Redux.applyMiddleware(ReduxThunk.default, checker, logger)
);
```

Now our application is going to leverage the new Redux Thunk script. Going back to our browser, hit refresh. Let's go ahead and delete an item.

Todo List

Add Todo | Add Todo

- Wash the car X
- ~~Go to the gym~~ X

Goals

Add Goal | Add Goal

- Learn Redux X
- Read 50 books this year X

Everything works pretty much exactly the same. But now instead of having our own thunk, we're just taking an advantage of the official Redux Thunk package.

Here's the commit with the changes made in the discuss above.

🔁 Remember that middleware executes in the order of the arguments passed into the `applyMiddleware()` function.

Benefits of Thunks

Out of the box, the Redux store can only support the *synchronous* flow of data. Middleware like **thunk** helps support *asynchronicity* in a Redux application. You can think of thunk as a wrapper for the store's `dispatch()` method; rather than returning action objects, we can use thunk action creators to dispatch functions (or even or Promises).

Without thunks, *synchronous* dispatches are the default. We *could* still make API calls from React components (e.g., using the `useEffect` hook to make these requests) -- but using thunk middleware gives us a cleaner separation of concerns. Components don't need to handle what happens after an asynchronous call, since API logic is *moved away* from components to action creators. This also lends itself to greater predictability, since action creators will become the source of every change in state. With thunks, we can dispatch an action only when the server request is resolved!

Summary

If a web application requires interaction with a server, applying middleware such as **thunk** helps solve the issue of asynchronous data flow. Thunk middleware allows us to write action creators that return *functions* rather than objects.

By calling our API in an action creator, we make the *action creator* responsible for fetching the data it needs to create the action. Since we move the data-fetching code to action creators, we build a cleaner separation between our UI logic and our data-fetching logic. As a result, thunks can then be used to delay an action dispatch, or to dispatch only if a certain condition is met (e.g., a request is resolved).

Further Research

- Redux Thunk on GitHub
- Async Flow from the Redux docs

Leveraging Thunks in our App

Thunkify Goals

Now that we have the ability to return functions from reaction traders, let's go ahead and thunkify our goals component. We're going to move all this data fetching logic as well as all the data fetching logic into their own action creators.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        return API.saveGoal(inputRef.current.value).then((goal) =>{
            props.store.dispatch(addGoalAction(goal));
            inputRef.current.value = "";
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }

    const removeItem = goal => {
        props.store.dispatch(removeGoalAction(goal.id));

        return API.deleteGoal(goal.id).catch(() => {
            props.store.dispatch(addGoalAction(goal));
            alert("An error occured. Try again.");
        });
    }

    return (
    <div>
        <h1>Goals</h1>
            <input type="text" placeholder="Add Goal" ref={inputRef}/>
            <button onClick={addItem}>Add Goal</button>
        <List remove={removeItem} items={props.goals}/>
    </div>
    )
}
```

Let's start off with adding an item. We're going to go ahead and remove all this.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        // return API.saveGoal(inputRef.current.value).then((goal) =>{
        //     props.store.dispatch(addGoalAction(goal));
        //     inputRef.current.value = "";
        // }).catch(() => {
        //     alert("There was an error. Try again.");
        // })
    }

    const removeItem = goal => {
        props.store.dispatch(removeGoalAction(goal.id));

        return API.deleteGoal(goal.id).catch(() => {
            props.store.dispatch(addGoalAction(goal));
            alert("An error occured. Try again.");
        });
    }

    return (
    <div>
        <h1>Goals</h1>
            <input type="text" placeholder="Add Goal" ref={inputRef}/>
            <button onClick={addItem}>Add Goal</button>
        <List remove={removeItem} items={props.goals}/>
    </div>
    )
}
```

Remove all this comment

What we eventually want to do is to be able to call props.store.dispatch. Within this dispatch, we will invoke our new handleAddGoal action creator. Again, that function hasn't been defined just yet, but we will create a little bit later. What we want to pass into handleAddGoal are two parameters, our two arguments. First, we want the name of the goal, which we can get from inputRef.current.value. Next, what we want to do is also give it the ability to reset the input value to an empty string. To do that, what we can do is pass it a callback function. It will just be one statement in there or one expression in there. We can probably just use regular parentheses. What we want to return is inputRef.current.value, setting it to an empty string.

```
const addItem = (e) => {
    e.preventDefault();

    props.store.dispatch(
        handleAddGoal(
            inputRef.current.value,
            () => (inputRef.current.value = "")
        )
    )
}
```

Great. Let's go to create this new action creator. We will call it handleAddGoal. Again, it takes in the name of the goal as well as a callback function. Just like we did earlier, we're going to make sure that this action creator function returns a function. We're going to pass into this function dispatch. Within that function, we're going to make our API call pass in the name of the goal, and then, when that API request resolves, we're going to get our goal. What we want to do with that goal is first called dispatch. Invoke our addGoalAction creator. Finally, passing the goal. Just for good measure, also create a catch function over here so in case there's an error, we can just alert the user. Hit save. This all should work.

```
function handleAddGoal(name, callback){
    return (dispatch) => {
        return API.saveGoal(name).then((goal) => {
            dispatch(addGoalAction(goal))
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }
}
```

We'll test it later. But let's go back removeItem first.

```
const removeItem = goal => {
    props.store.dispatch(removeGoalAction(goal.id));

    return API.deleteGoal(goal.id).catch(() => {
        props.store.dispatch(addGoalAction(goal));
        alert("An error occured. Try again.");
    });
}
```

Next one we want to do is update our remove item method. Take out all this API logic. Now, instead of saying dispatch removeGoalAction creator, we're going to create a new function called handleDeleteGoal. In this case, we'll pass in the entire goal.

```
const Goals = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        props.store.dispatch(
            handleAddGoal(
                inputRef.current.value,
                () => (inputRef.current.value = "")
            )
        )
    }

    const removeItem = goal => {
        props.store.dispatch(handleDeleteGoal(goal));
    }

    return (
    <div>
        <h1>Goals</h1>
            <input type="text" placeholder="Add Goal" ref={inputRef}/>
            <button onClick={addItem}>Add Goal</button>
        <List remove={removeItem} items={props.goals}/>
    </div>
    )
}
```

Next one, we will make a new function called handleDeleteGoal. We'll pass on the goal. Again, this action creator function is going to return a function itself. This function takes in a dispatch. What we're dispatching is, removeGoalAction. This takes in the goal.id. Actually, what we're really returning is the promise. We can say return API.deleteGoal, pass in the goal.id. In this case, if it's something that went wrong, if there's an error, we'll catch it. Within this catch, we're going to do is dispatch addGoalAction to add the goal back in. Also do that. If there is an error, let's just alert the user.

```
function handleDeleteGoal (goal) {
    return (dispatch) => {
        dispatch(removeGoalAction(goal.id));

        return API.deleteGoal(goal.id).catch(() => {
            dispatch(addGoalAction(goal));
            alert("There was an error. Try again.");
        })
    }
}
```

Let's go ahead and test everything out now. Heading back to our browser. Then, inside of our goals component, we should be able to add a new goal.

Here's the commit with the changes made in the discuss above.

Thunkify Todos

Just like we did with our goals component, we also want to thunkify our todos component. In other words, we're going to move all this data fetching logic, and move all that to their own action creators.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        // return API.saveTodo(inputRef.current.value).then((todo) =>{
        //     props.store.dispatch(addTodoAction(todo));
        //     inputRef.current.value = "";
        // }).catch(() => {
        //     alert("There was an error. Try again.");
        // })
    }

    const removeItem = todo => {
        props.store.dispatch(handleDeleteTodo(todo));
    }


    const toggleItem = (id) => {
        props.store.dispatch(toggleTodoAction(id));

        // return API.saveTodoToggle(id).catch(() => {
        //     props.store.dispatch(toggleTodoAction(id));
        //     alert("An error occured. Try again.");
        // });
    };


    return (
        <div>
            <h1>Todo List</h1>
            <input type="text" placeholder="Add Todo" ref={inputRef}/>
            <button onClick={addItem}>Add Todo</button>
            <List toggle={toggleItem} remove={removeItem} items={props.todos}/>
        </div>
    )
}
```

Instead of all this, we can delete all of it, and instead say, props.store.dispatch. Then invoking a new action creator under name handleAddTodo. Again, this function doesn't exist just yet. We're going to create it later. We'll pass it two arguments. One will just be the name of the todo. We can grab that from inputRef.current.value. The second will be a callback. Kind of the same thing as earlier, this callback is going to allow us to reset the user input field. Actually, make sure this is set to an empty string.

```
const Todos = (props) => {
    const inputRef = React.useRef();

    const addItem = (e) => {
        e.preventDefault();

        props.store.dispatch(handleAddTodo(
            inputRef.current.value,
            () => inputRef.current.value = ""
            ))
    }

    const removeItem = todo => {
        props.store.dispatch(handleDeleteTodo(todo));
    }
```

Now we need to create this new function, handleAddTodo. This function takes the name and a callback function. Just like we did earlier, we're going to make sure that this function returns a function. We're going to pass in dispatch into this function, make our API call, pass in the name to it. Then, when this request resolves, we're going to make sure we call it dispatch within it, invoke addTodoAction. In that function when it's invoked, we're also going to be passing in todo which we get from our response. Then, the last thing we need to do is make sure we reset the user input field as well. Save it and this work will be worked. But just in case there's an error, let's also make sure the user is alerted.

```
function handleAddTodo(name, callback){
    return (dispatch) => {
        return API.saveTodo(name).then((todo) => {
            dispatch(addTodoAction(todo));
            callback;
        }).catch(() => {
            alert("There was an error. Try again.");
        })
    }
}
```

AddItem looks pretty good for now. Let's go to toggleItem and change our toggling. We can clean things up by delete it.

```
const toggleItem = (id) => {
    props.store.dispatch(toggleTodoAction(id));

    // return API.saveTodoToggle(id).catch(() => {
    //     props.store.dispatch(toggleTodoAction(id));
    //     alert("An error occured. Try again.");
    // });
};
```

Delete all the comment line

Instead of dispatching toggleTodoAction, we'll just dispatch a new function, handleToggle.

```
const toggleItem = (id) => {
    props.store.dispatch(handleToggle(id));
};
```

Let’s create a new function with name handleToggle. Make sure we pass in the id. All we're going to do is return a function. This function takes in dispatch. We'll first make the dispatch, invoke toggleTodoAction, that will take in the id. From here, we'll just make sure we make the API call as well. What's reflected on the backend. If there's an error with that request, we need to make sure we catch it. If there really is an error, we'll dispatch toggleTodoAction , has an id. This way we toggle back to its original state, and then we'll also alert the end-user.

```
function handleToggle(id) {
    return (dispatch) => {
        dispatch(toggleTodoAction(id));

        return API.saveTodoToggle(id).catch(() => {
            dispatch(toggleTodoAction(id));
            alert("An error occured. Try again.");
        });
    }
}
```

Now, let's go ahead and test in our browser. Now we should be able to add a new todo. On top of that, we should be able to toggle each todo as well.

Here's the commit with the changes made in the discuss above.

Thunkify Initial Data

The only other place where we're actually mixing in our data fetching logic with our component UI logic is actually in our app component.

```
const App = (props) => {
    const [value, setValue] = React.useState(0);

    React.useEffect(() => {
        props.store.subscribe(() => {
            setValue((value) => value + 1);
        })
    },[])

    React.useEffect(() => {
        Promise.all([API.fetchTodos(), API.fetchGoals()]).then(([todos, goals]) => {
            props.store.dispatch(receiveDataAction(todos, goals));
        });
    }, []);

    const {todos, goals, loading} = props.store.getState();

    if(loading === true){
        return <h3>Loading...</h3>;
    }

    return (
        <div>
            <Todos todos={todos} store={props.store}/>
            <Goals goals={goals} store={props.store}/>
        </div>
    )
}
```

Particular, it's actually when we fetch your initial data. Just we did in the past, what we also want to do here is move up all this logic into its own action creator. Just like we did before, we are going to delete it.

```
const App = (props) => {
    const [value, setValue] = React.useState(0);

    React.useEffect(() => {
        props.store.subscribe(() => {
            setValue((value) => value + 1);
        })
    },[])

    React.useEffect(() => {
        // Promise.all([API.fetchTodos(), API.fetchGoals()]).then(([todos, goals]) => {
        //     props.store.dispatch(receiveDataAction(todos, goals));
        // });
    }, []);

    const {todos, goals, loading} = props.store.getState();
```

Delete all the comment line

Now we can just say props.star.dispatch. We're going to invoke in here, handleInitialData.

```
const App = (props) => {
    const [value, setValue] = React.useState(0);

    React.useEffect(() => {
        props.store.subscribe(() => {
            setValue((value) => value + 1);
        })
    },[])

    React.useEffect(() => {
        props.store.dispatch(handleInitialData())
    }, []);

    const {todos, goals, loading} = props.store.getState();

    if(loading === true){
        return <h3>Loading...</h3>;
    }

    return (
        <div>
            <Todos todos={todos} store={props.store}/>
            <Goals goals={goals} store={props.store}/>
        </div>
    )
}
```

Then we will create this handleInitialData function. This function takes in no arguments, but it's going to return a function. This function is passed in dispatch, and when we paste our code from earlier like code in useEffect App, everything else pretty much stays the same.

```
function handleInitialData() {
    return (dispatch) => {
        Promise.all([API.fetchTodos(), API.fetchGoals()]).then(([todos, goals]) => {
            props.store.dispatch(receiveDataAction(todos, goals));
        });
    }
}
```

Except this time, instead of grabbing dispatch props.store, we're going to grab it from the dispatch that's passed in right over here.

```
function handleInitialData() {
    return (dispatch) => {
        Promise.all([API.fetchTodos(), API.fetchGoals()]).then(([todos, goals]) => {
            dispatch(receiveDataAction(todos, goals));
        });
    }
}
```

Let's go back over to our browser. Refresh. After about two seconds, we shall actually see all our initial data as well.

Just a recap. What we have now done is we've successfully moved all of our data fetching logic from our components into their own action creators.

Here's the commit with the changes made in the discuss above.

More Asynchronous Options

We've resisted including more content about advanced data-fetching topics with Redux because typically they bring in *a lot* of complexity, while the benefits aren't seen until your data-fetching needs become large enough. With that said, now that you have a solid foundation on Redux and specifically, asynchronous Redux, you'll be in a good position to read up on the different options to decide if any would work best for the type of application you're working on. We encourage you to read up on both of the other (popular) options.

- Redux Promise - FSA-compliant promise middleware for Redux.
- Redux Saga - An alternative side effect model for Redux apps
  - Initiating a side effect
  - Calling an API Stack Overflow thread
- Redux-Thunk vs. Redux-Saga - Which is better in your app?

Lesson Summary

In this section, we used the redux-thunk library that we installed in the previous section to make our code more singularly focused and maintainable. We converted the:

- Goals code to use thunks
- Todos code to use
- Initial data fetching to use thunks

On your own mental challenge

Read these articles: Redux Thunk, Why do we need middleware for async flow in Redux?, and Understanding how redux-thunk works.

1. Why do we use middleware to perform asynchronous tasks in Redux apps?
2. How do we use `redux-thunk` to make API requests in Redux apps?

(Doc) Week 7:

Why Testing is Important During Software Development

> As a software engineer, one big part of your responsibility is writing good quality and maintainable code. However, whether you're working as an individual or part of a team, we all make mistakes from time to time during the development process. The earlier we catch and fix these bugs, the more confidence your organization and customers will have in the software you build.

What are some of the benefits of testing your software during development?

- Improves the quality of the code it is testing.
  - The more tests you write against a piece of code, the less likely that code is to break or have bugs.
- Helps identify defects early in development.
  - Identifying bugs in your software before customers or users find them is crucial in software development. Finding bugs early in development saves companies time, money, and resources.)
- Helps with documenting how the system is expected to work.
  - Each test you write usually includes a brief description of the code you are testing and the behavior or output you expect after the test is done. If you're working on a large, complex application with code you're not familiar with, just reading these test descriptions can help you ramp up quickly on what the code should and should not be doing.
- Helps ensure logic does not break after code refactoring.
  - As technology and business needs drastically shift over time, you'll find yourself in many situations as a software developer where you need to go back to an old piece of code and refactor it. If this code already has good test coverage, you can simply rerun the tests after refactoring to ensure you did not break any existing logic.

Big picture: What is Jest

What is Jest?

- Jest is a Javascript testing framework developed by Facebook which is now known as Meta.
  - It has an MIT license so that it can be used for both private and commercial use for free.
- It can be used to test code written in React, TypeScript, Babel, Node, Vue, Angular, and more.
- Jest is used to create and run tests that ensure your React functions and components are working as expected.

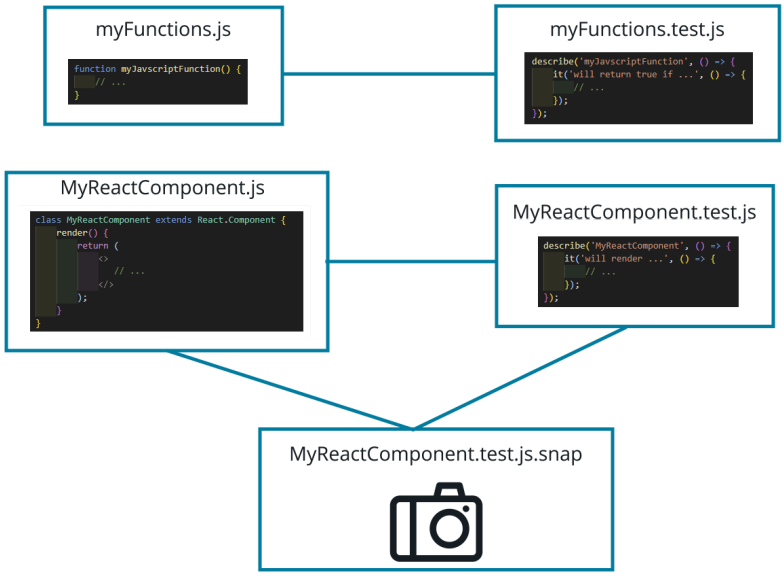
Alternatives?

The two big alternatives to Jest are Mocha and Jasmine. Each comes with its own advantages and disadvantages. For example, both Mocha and Jasmine require more configuration for testing React code.

> Of these 3 options, Jest is the most popular Javascript testing framework option since its both used and recommended by Meta which developed ReactJS.

What are the key features?

- No configuration is required!
  - Once you install Jest, you can start writing tests for your React functions and components immediately!
- Each test you write for a specific function or component will be in a separate file so that tests are isolated from the code and don't impact your React app's performance.
- Snapshot testing.
  - Jest can store a copy of how your React component will render in a browser and use it for reference when testing to ensure that no unexpected changes occurred.
- Simplicity and readability.
  - Jest has an API that gives you a lot of control over what you want to test in your code. You can also add titles and descriptions to each of your tests which allows you to document all the crucial functions in your code which you want to test.



Architecture of Jest Testing

Architecture of Jest Testing

1. On the left-hand side (`myFunctions.js` and `MyReactCompoennt.js`), you have your React functions and components which are typically in a .js file. Each file is then paired with a .test.js file on the right-hand side.
2. On the right-side (`myFunctions.test.js` and `MyReactCompoennt.test.js`), this is where the Jest tests are written to test the functions or components you've written.
3. At the bottom (`MyReactComponent.test.js.snap`), Jest will create a .snap file which is a snapshot of how your React component will look when rendered on the DOM. You can then write jest tests to verify that the snapshot is displaying DOM elements as expected.

Resources:

Jest Website: https://jestjs.io

Jest GitHub Repository: GitHub - jestjs/jest: Delightful JavaScript Testing.

How to Install and Run Your First Jest Test

Prerequisite:

- Have NPM already installed on your machine.

Installing Jest:

`npm install --save-dev jest`

Package.json File:

After installing Jest, you need to update your package.json file to include a script section so that you can run the Jest tests through npm:

```
{
  "devDependencies": {
    "jest": "^27.4.5"
  },
  "scripts": {
    "test": "jest"
  }
}
```

multiply.js File:

```
// multiply.js

function multiply(x, y) {
    return x * y;
}

module.exports = multiply;
```

multiply.test.js File:

```
// multiply.test.js

var multiply = require('./multiply');

describe('multiply', () => {
    it('will return the product of both numbers passed', () => {
        expect(multiply(2, 3)).toEqual(6);
    });
})
```

Running Jest:

`npm run test`

Demo script

Using Visual Studio Code

1. (**multiply.js**)

2. [LINES 1-3] Take a look at this file called multiply.js. It contains one javascript function called multiply which simply takes two numbers as arguments and returns the product of both.
3. [LINE 5] There is also an export function at the bottom so that this function is accessible to other files that want to use it.
4. The first step will be installing Jest in this directory. Let's do that now. I'll enter the command "npm install --save-dev jest" in a terminal and run it.
5. (RUN `npm install --save-dev jest`)
6. You'll see we now have a package.json file created. We need to modify this file to include a script command for running jest.
7.
```
1
```
8. "scripts": {
9. "test": "jest"
10. }
11.
```
1
```
12. With this script command added, we'll now be able to run jest tests by entering the command NPM RUN TEST. Let's look at a sample Jest test written for the Multiple function we just looked at.
13. (**MULTIPLY.TEST.JS**)
14. [LINE 1] We start by importing the multiply function into this file by entering "const multiply = require('./multiply');"
15. [LINE 3] Next, there is a DESCRIBE function. This is where you'll want to enter the name of the function or component you are testing.
16. [LINE 4] Within this describe function, we'll enter an IT function. Here, you'll enter what you expect the test to do. In our case, we are expecting that the function "will return the product of both numbers passed"
17. [LINE 5] Now within the It function is where we'll enter our test. We start with the EXPECT keyword. This allows you to check the value passed to it against a certain condition.
18. So what do we expect 2 * 3 to return? That would be 6, so we extend the EXPECT function with .TOEQUAL(6)
19. (RUN `npm run test`)
20. Take a look at the output. You'll see that it states 1 test passed across 1 suite.
21. If you want to see what the test would look like if it failed, let's change .TOEQUAL to 0 and rerun the test
22. (CHANGE TOEQUAL to be 0 and run `npm run test` again)

Jest Matchers Part 1

What are Jest Matchers?

Jest Matchers compare two values:

- Expected
  - *What should be the result*
- Actual
  - *What the actual result is*

How do they work?

- If the two values *match*
  - the matcher returns `true`
  - test continues
- If the two values **don't match**
  - the matcher returns `false`
  - test fails

Glossary of Matchers Covered:

Numbers

- `expect()` allows you to check the value passed to it against a certain condition. These conditions are Jest Matchers.
- `toEqual()` will verify that the value of an object is equal to what you expect.
  - **Examples**: expect(2 3).toEqual(6); -> *pass* | expect(2 3).toEqual(0); -> fail
- `toBeLessThan()` will verify if the expected number is smaller than the actual number.
- `toBeLessThanOrEqual()` will verify if the expected number is smaller or equal to the actual number.
  - **Examples**: expect(5).toBeLessThan(10) | expect(5).toBeLessThanOrEqual(5)
- `toBeGreaterThanOrEqual()` will verify if the expected number is greater or equal to the actual number.
- `toBeGreaterThan()` will verify if the expected number is smaller than the actual number.
  - **Examples**: expect(5).toBeGreaterThanOrEqual(5) | expect(5).toBeGreaterThan(1)

Not

- `not` is used in the case when you don't expect two variables to match
  - **Examples**: expect(2 * 3).not.toEqual(0) | expect(5).not.toBeLessThan(1)

For a full list of all of Jest's expect matchers, take a look at Jest's official documentation HERE.

Jest Matchers Part 2

Glossary of Matchers Covered:

Strings

- `toMatch()` will verify if a string matches a regex expression passed.
  - **Example**: expect('I enjoy using Jest.').toMatch(/enjoy/);

Arrays

- `toContain()` will verify if an array contains a specific object. This includes numbers and strings.
  - **Examples**: expect([1, 2, 3, 4]).toContain(3); | expect([‘cat’, ‘dog’, ‘bird’]).toContain(‘bird’);

Null

- `toBeNull()` will verify if the expected value is null.
  - **Example**: var test = null; | expect(test).toBeNull()

Functions

- `toThrow()` will verify that we are expecting a function passed to throw an error. **Example**:

```
function throwAnError(text) {
  throw new Error('Error: ' + text);
}
expect(throwAnError(‘my error’)).toThrow();
```

For a full list of all of Jest's expect matchers, take a look at Jest's official documentation HERE.

Testing Async Functions

What is an asynchronous function?

> A function which depends on events outside of the code in order to complete
> Used for performing CRUD operations to the server-side and then awaiting a response

Examples of asynchronous patterns in JavaScript:

- Callbacks
- Async/Await
- Promises

isUtensilAvailable.js:

```
1  // isUtensilAvailable.js
2
3  var utensils = ['fork', 'knife', 'spoon'];
4
5  function isUtensilAvailable(utensil){
6      return new Promise((resolve, reject) => {
7          setTimeout(() => {
8          utensils.includes(utensil)
9              ? resolve(true)
10             : reject('No utensils found.')
11         }, 2000);
12     });
13 }
14
15 module.exports = isUtensilAvailable;
```

isUtensilAvailable.test.js:

```
1  // isUtensilAvailable.test.js
2
3  var isUtensilAvailable = require('./isUtensilAvailable');
4
5  describe('isUtensilAvailable', () => {
6      it('will return true if the utensil is found', async() => {
7          var utensil = 'fork';
8          var result = await isUtensilAvailable(utensil);
9          expect(result).toEqual(true);
10     });
11
12     it('will return an error if the utensil is not found', async() => {
13         var invalidUtensil = 'tree';
14         await expect(isUtensilAvailable(invalidUtensil)).rejects.toEqual('No utensils found.');
15     });
16 })
```

Demo script

1. (**isUtensilAvailable.js**)
2. Let's start by writing a test which will verify that isUtensilAvailable will return true if the utensil passed to it is present.
   a. [LINES 1-4] First, we import the function to the test file and add the describe and IT functions.

    b. [LINE 5] Next, we create a variable called “utensil” and set it to “fork” which we know is present in the utensil array.
    c. [LINE 6] Then, we create a variable called result which calls isUtensilAvailable and passes utensils as an argument.
    d. [LINE 7] Lastly’ we expect result to equal true.
3. RUN `npm run test`
4. Notice that the test actually *failed*. The expect method was expecting the result to equal true, but instead it was an empty object. This happened because isUtensilAvailable is an async function and Jest is unaware of it.
5. If Jest does not know that the function its testing is async, it will continue executing its test before any asynchronous code is completed. That’s why the result variable is empty instead of being set to true. We need to let Jest know that it needs to wait for the async function to resolve or reject before moving on.
6. We can do this with two simple changes.
7. The first change is to add the word “async” before the function passed to the test. This let’s Jest know that there will be at least one async function called in this test.
    - (ADD “async” before () ON LINE 4)
8. The second change is to add “await” before the call to isUtensilAvailable. This let’s Jest know that this is the async method we are testing
    - (ADD “await” BEFORE FUNCTION CALL ON LINE 6)
9. (RUN `npm run test`)
10. The test will now pass! Let’s now add one more test to verify that an error is returned if we pass a utensil that does not exist.
11. (Use THE FOLLOWING code)
12. it('will return an error if the utensil is not found', async() => {
13. var invalidUtensil = 'tree';
14. await expect(isUtensilAvailable(invalidUtensil)).rejects.toEqual('No utensils found.');
15. });
16. We’ve added the IT description stating that it “will return an error if the utensil is not found”
17. Next, we’ve added the async key word so that Jest knows its testing an async method.
18. Then, we created a new variable called invalidUtensil and set it to “tree” which we know is not present in the utensil array.
19. Finally, we’ll AWAIT and then EXPECT isUtensilAvailable along with the invalidUtensil dot REJECTS dot TOEQUAL the string “no utensil found”
20. If we run this test, it will pass. This is basically verifying that isUtensilAvailable is returning the reject method when the utensil is not found.
21. (RUN `npm run test`)
22. We’ve now written our first two Jest tests against asynchronous code! Take your time and review the code we’ve written here.

Introduction to React Testing Library

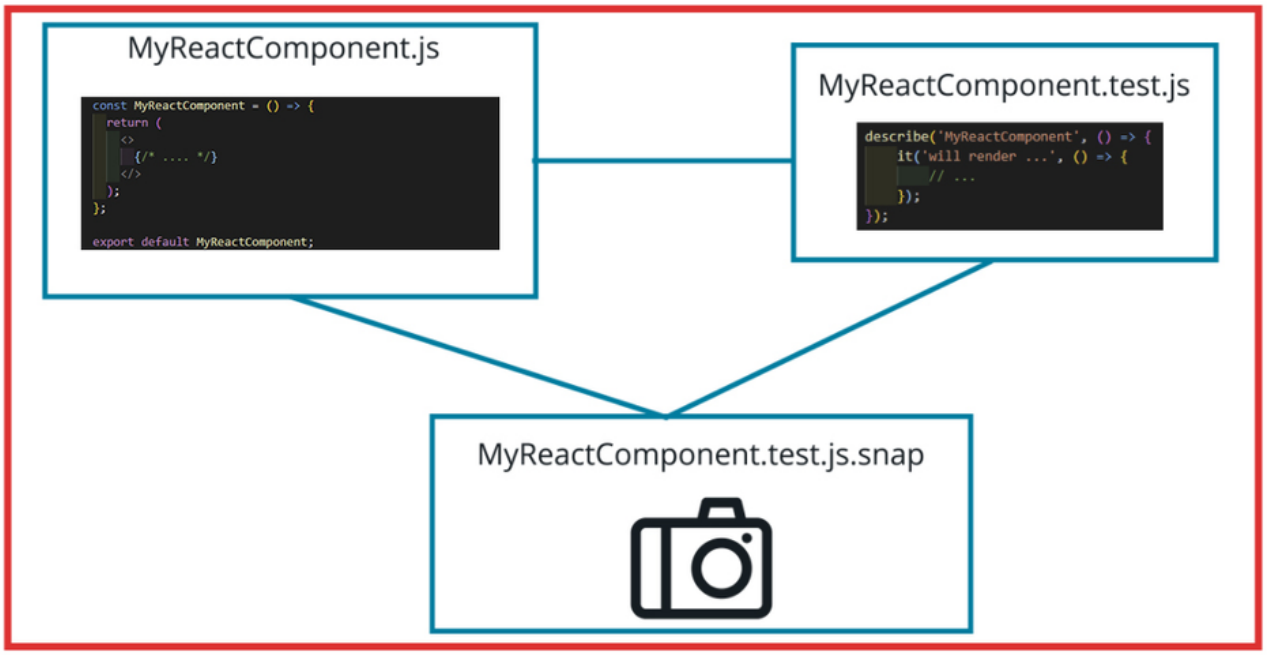
What is 'React Testing Library'?

A JavaScript testing utility for rendering and testing React Components

- Great for snapshot testing
- Allows you to write tests that interact with the React DOM
- Examples:
  - Typing into an input field
  - Clicking a button
  - Selecting an option from a dropdown

> One common misconception is that React Testing Library is an alternative to Jest. However, that is not true. React Testing Library is meant to be used with Jest. If you use React Testing Library on top of Jest, React Testing Library will allow you to interact with the React DOM and then you can use Jest to validate whether or not a change occurred on the React component. In addition, you would use Jest to run the test and see if it passes or fails.

Creating Jest test files for testing both Javascript and React functions.



Creating Jest Tests for testing React Components.

React Testing Library

- Install React Testing Library and use it to render a React component within a Jest Test.

What's next?

1. Generate and test snapshots.
2. Using React Testing Library to interact with the DOM which is rendered by the React component and using Jest to validate changes that happen on the DOM.

Resources:

React-Testing-Library Website: https://testing-library.com

React-Testing-Library GitHub Repository: GitHub - testing-library/react-testing-library: 🐐 Simple and complete React DOM testing utilities that encourage good testing practices.

Rendering a React Component for Testing

Prerequisite:

- Have NPM already installed on your machine.
- Have the latest version of NodeJS installed on your machine.
- Have Jest installed (if not using Create-React-App to create React App).

Create-React-App Command:

```
npx create-react-app demo
```

App.js

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

App.test.js:

```
import { render, screen } from '@testing-library/react';
import App from './App';

test('renders learn react link', () => {
  render(<App />);
  screen.debug();
});
```

Demo script

1. Select a directory where we will create our react app.

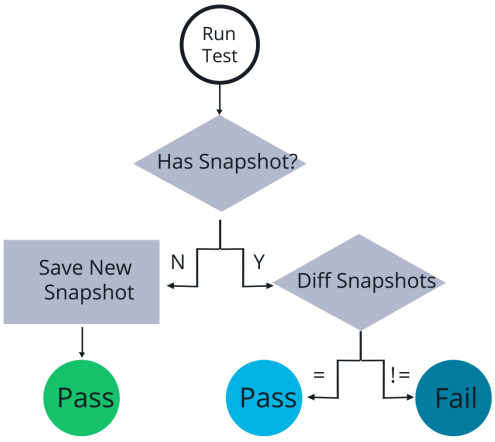
2. Open a terminal in this directory and run the following command: npx create-react-app along with the name you want to give the project. Let's just call it "demo"
3. (ENTER `npx create-react-app demo`)
4. You can see that create-react-app will create all the files we need to get started. Let's start off by looking at package.json
5. (**package.json**)
6. [LINE 7] Notice that React-Testing-Library is already included by default with create-react-app. Let's now take a look at App.js
7. (**APP.JS**)
8. app.js is rendering a simple component which contains an image, a short paragraph tag, and a link to React . Now let's look at the test file that was generated as well
9. (**APP.TEST.JS**)
10. [LINE 1] At the top of this test, we can see two functions being imported from React Testing Library: render and screen. We'll use both of these to render the App.js component in our test.
11. [LINE 2] Next, we are importing the App component from the App.js file.
12. [LINE 5] Now let's look at the test. The first thing it does is call that render method from React Testing Library in order to render the App component.
13. Remove the last 2 lines since they are covering a future topic.
14. (REMOVE LAST 2 LINES and ADD: screen.debug();
15. Replace these lines with screen.debug(). What this will do is print the HTML output of the App component. To see it in action, let's save this change and run npm test
16. (RUN TEST)

Snapshot Testing

What is a snapshot test?

- Allows you to verify that your component rendered as you expect.
  - A snapshot file is generated the first time the test runs.
  - On future runs, the component rendered in the test will be compared to the snapshot.

Snapshot Testing Flow



The Snapshot Testing Flow

Snapshot testing flow

Starting at the top, the first time a snapshot test runs, Jest will first check if the Snapshot exists.

- If a snapshot is not present, we save a snapshot of the React component and pass the test.
- If a snapshot is present, Jest will perform a Diff comparison between the existing snapshot and the component being rendered by Jest.
  - If the snapshot matches, the test will pass.
  - If the snapshot does not match the rendered component, the test will fail.

Jest Matcher to Verify Component and Snapshot Match:

```
expect(component).toMatchSnapshot();
```

Demo script

In the **GREETINGS.js** file, you can see in this demo, this component simply renders an h1 tag with the text “Welcome”. Let’s look at how to create a snapshot test for this component

1. (**GREETINGS.TEST.JS**)
2. [LINE 7]In this test, we start by declaring a variable called component and making it equal to the rendering of the Greetings component.
3. [LINE 8] The next line introduces a new Jest matcher called toMatchSnapshot(). This essentially makes this test a snapshot test. What we are doing here is expecting the component to match the snapshot file. However, we currently don’t have a snapshot file, so let’s generate one by running the test just once.
4. (RUN npm test)
5. Notice that after running the test, we see a message in the terminal stating ONE SNAPSHOT WRITTEN Also there is a new directory which was generated in our solution folder called _snapshots. Let’s open that up.
6. (**GREETINGS.TEST.JS.SNAP**)

7. We see here a new .SNAP file which shows the HTML output of the React component we rendered. This is the snapshot file which will be used for reference each time this test runs.
8. (RUN npm test)
9. Notice that the test still passes. This is cause the rendered component still matches the snapshot. But now, what happens if we modify the Greetings component slightly?
10. (**GREETINGS.JS**)
11. Change the test “WELCOME” to be “GOODBYE”
12. (RUN npm test)
13. Change text to be GOODBYE
14. (RUN npm test)
15. Notice that the test now failed. This is because the snapshot file no longer matches the rendered component. In the snapshot file, the h1 tag states “WELCOME” but the component was updated to state “GOODBYE”
16. Snapshot tests are very helpful for testing pages which you don’t expect to change frequently. They can help you find bugs where you made a change in the code, but do not want or expect the UI to change in any way.

What if the text change from Welcome to Goodbye was intentional?

There are two ways to update the snapshot so that the test passes.

- Simply delete the snapshot and rerun the test. This will cause a new snapshot to generate, replacing the old one.
- When you run the test and see that the snapshot failed, jest will actually give you the option to enter the character ‘u’ which will update all of the snapshots and rerun the test. Either solution is fine.

Type of Queries Table

	No Matches	1 Match	Multiple Matches
Single Element			
getBy	Throws Error	Returns Element	Throws Error
queryBy	Returns NULL	Returns Element	Throws Error
Multiple Elements			
getAllBy	Throws Error	Returns Array	Returns Array
queryAllBy	Returns Empty Array	Returns Array	Returns Array

Type of Queries Table

Table description and explantion

Single Element

GetBy

When the getBy query is used, we want React Testing Library to search for an element that we know should be present in the React Component.

- If no match is found an error is thrown and the test fails
- If 1 match is found then the element is returned and the test will continue
- If multiple matches are found, an error will also be thrown. This is because getBy is only used for querying single elements and is not expecting more than 1 element to be found.

QueryBy

When the queryBy query is used, we want React Testing Library to search for an element that may or may not be on the page. This can be useful for testing an edge case where we know an element should be present in certain cases, but we want to verify that it is hidden in other cases.

- If no matches are found, NULL is returned instead of an error. The test would continue and the next step in the Jest test could verify that the element is not supposed to be present
- If 1 match is found, we just return the element and the test will continue
- If multiple matches are found, an error will also be thrown. Again, this is because we are only querying single elements and are not expecting more than 1 element to be found.

Multiple Elements

getAllBy

We use getAllBy when we want to retrieve multiple elements that we know should be present in the React Component.

- If no matches are found, an error is thrown and the test fails
- If 1 match is found, an array is returned with that single match and the test continues
- If multiple matches are found an array is returned with all the elements found.

queryAllBy

When queryAllBy is used, we want React Testing Library to search for all elements that may or may not be on the page. Therefore, it will never throw an error.

- If no matches are found, an empty array is returned and the test will continue
- If 1 or multiple matches are found, an array is returned with any of the matches found.

When to use Get vs Query

- **Get**
  - When you want your test to verify an element is present
- **Query**
  - When you want to search for an element and verify that it is not present

React DOM Testing - Selecting Elements

Common Query Selectors

- `Text` will select an element by an exact string or regex that appears in an element.
  - getByText( )
  - queryAllByText( )
- `TestId` will select an element by an HTML attribute called data-testid.
  - queryByTestId( )

Glossary of Matchers Covered:

`toBeInDocument()` will verify whether or not a component is present in the document. If it's not, the test will fail.

App.js

```
// App.js

function App() {
  return (
    <div>
      <form>
        <label for="fname">First name:</label>
        <br />
        <input type="text" name="fname" data-testid='first-name-input' />
        <br />
        <label for="lname">Last name:</label>
        <br />
        <input type="text" name="lname" data-testid='last-name-input' />
        <br />
        <input type="submit" value ='Submit'/>
      </form>
    </div>
  );
}

export default App;
```

App.test.js:

```
import { render } from '@testing-library/react';
import * as React from 'react';
import App from './App';

describe('App', () => {
    it('will have all expected fields', () => {
        var component = render(<App />);

        var labels = component.getAllByText(/name/)
        expect(labels.length).toEqual(2);

        var firstNameInput = component.getByTestId('first-name-input')
        var lastNameInput = component.getByTestId('last-name-input')
        expect(firstNameInput).toBeInTheDocument();
```

```
15          expect(lastNameInput).toBeInTheDocument();
16
17          var submitButton = component.getByText('Submit')
18          expect(submitButton).toBeInTheDocument();
19      });
20  })
```

For a full list of all of React Testing Library queries and selectors, take a look at React Testing Library's official documentation HERE.

Demo script

1. (**APP.JS**)
2. Here in our App file, we have a simple component which consists of two labels, two input fields and one submit button. Let's run it so that you can see how the component renders
3. (RUN `npm start` and show the browser)
4. Look at a test using Jest and React Testing Library which will verify each of these components are present.
5. (**APP.TEST.JS**)
6. The objective of this test is to query each field we just looked at and verify that they are all displayed on the page.
7. [LINE 7] First, we are declaring a variable called component which is rendering the App component.
8. [LINE 9] Next, we declare a variable called labels. This is calling component.getAllByText and we are passing regex with the string "NAME". This will retrieve all of the labels which have text that contains name. Therefore, it will return both the First Name and Last Name labels.
9. [LINE 10] Next, we EXPECT labels.length to equal 2. This verifies that both labels were successfully found from the DOM.
10. Next, we want to retrieve the first and last name input fields.
11. [LINES 12-13] We declare both a firstNameInput and lastNameInput variable. For each of these, we call component.getByTestId and pass a string id. Notice that the string IDs are first DASH name DASH input and last DASH name DASH input. Let's go back to App.js
12. **(APP.JS**)
13. [LINE 7 and 11] Notice that for both input fields, I've added a data DASH testId attribute and set them to be equal to the test ids which I'm using for my test. Adding attributes like this to your React component makes it much easier for you to retrieve and verify them in your tests. Let's go back to the test now
14. **(APP.TEST.JS**)
15. [LINE 14] We expect firstNameInput toBeInTheDocument. This will verify if the input field is present in the document. If it's not present, an error will be thrown and the test will fail.
16. [LINE 15] Line 15 repeats this logic for the lastNameInput field
17. [LINE 17] Lastly, we want to verify that the submit button is present. We query it with getByText and add the text 'Submit' as a parameter. This will look for a single element that contains the text "Submit".
18. [LINE 18] We then verify its presence by EXPECTING submit button to be in the document
19. (PRESS `CTRL C`)
20. (RUN `NPM TEST`)

React DOM Testing - Firing Events

Fire Events Covered:

- `fireEvent.click`
  - Simulates clicking on a React DOM element
  - **Example** var submitButton = component.getByText('Submit') fireEvent.click(submitButton);
- `fireEvent.change`
  - Simulates modifying the value of a field
  - **Example** var nameInput = component.getByTestId('name-field') fireEvent.change(nameInput, { target: { value: 'Jean' } });

App.js:

```
import { useState } from "react";

export const NameForm = () => {
    const [success, setSuccess] = useState(false);
    const [error, setError] = useState(false);
    const [name, setName] = useState(null);

    const handleNameChange = (e) => {
        setName(e.target.value);
    };

    const handleSubmit = (e) => {
        e.preventDefault();

        if (!name) {
            setSuccess(false);
            setError(true);
            return;
        }

        setSuccess(true);
        setError(false);
    };

    return (
        <div className={"App"}>
            {success &&
                <h1 className={"Success"} data-testid="success-header">Name Submitted!</h1>
            }
            {error &&
                <h1 className={"Error"} data-testid="error-header">Please enter a name.</h1>
            }
            <form className={"Form"} onSubmit={handleSubmit}>
                <div>
                    <label>Name:</label>
                    <br />
                    <input
                        data-testid="name-input"
                        type="text"
                        value={name}
                        onChange={handleNameChange}
                    />
                </div>
```

```
                <input
                    data-testid="submit-button"
                    type="submit"
                    value="Submit"
                />
            </form>
        </div>
    );
}
```

App.test.js:

```
import * as React from 'react';
import { render, fireEvent } from '@testing-library/react';
import { NameForm } from './App';

describe('NameForm', () => {
    it('will display an error if the name is not provided.', () => {
        var component = render(<NameForm />);

        var submitButton = component.getByTestId('submit-button');
        fireEvent.click(submitButton);
        expect(component.getByTestId('error-header')).toBeInTheDocument();
        expect(component.queryByTestId('success-header')).not.toBeInTheDocument();
    });

    it('will display a success message if the name is provided.', () => {
        var component = render(<NameForm />);

        var input = component.getByTestId('name-input');
        fireEvent.change(input, { target: { value: 'Mike' } });
        var submitButton = component.getByTestId('submit-button');
        fireEvent.click(submitButton);
        expect(component.getByTestId('success-header')).toBeInTheDocument();
        expect(component.queryByTestId('error-header')).not.toBeInTheDocument();
    });
});
```

For more information on React Testing Library's fire event actions , take a look at React Testing Library's official documentation HERE.

Demo script

1. **(APP.JS**) Scroll down to render section)
2. Here we have a React component which consists of 4 fields
    a. A header for a success message
    b. A header for an error message
    c. An input field for the user to enter a name
    d. And a submit button
3. Note that both the success and error headers are only visible when the success or error state is set to true. Otherwise it’s hidden.
4. (Scroll to handleNameChange and handleSubmit method)
5. If we look at handleNameChange, you can see it is simply setting the name state to be whatever the user has entered into the input field.
6. In handleSubmit method, we check whether or not the name state is set. If it is not set, we set the error state to true and the success state to false. This will display the error message. However, if the name is populated, success is set to true and error is set to false, displaying the success message.

7. (RUN `npm start` and show browser)
8. So we see the input field and submit button rendered here. If I click Submit without entering anything into the input field, I'll see an error message
9. (click submit button)
10. But if I enter a name and then click submit, I'll see the success message
11. (Type Mike and click Submit)
12. Let's now look at two unit tests which will perform the same actions we just did to verify that the React component is working as we expect it to.
13. (**App.test.js**)
14. The first test we'll do is to make sure an error message is displayed if no name is provided.
15. [LINE 7] We start with rendering the NameForm
16. [LINE 9] Next, we get the submit button by its test id.
17. [LINE 10] After that, we do fireEvent.click on the submit button to simulate clicking the button
18. [LINE 11] We then expect that the error message will be in the document, so we get it by testId
19. [LINE 12] However, we also want to verify that the success message is not displayed either, so we add the last line which will do a query by the test Id for the success message and verify that it is not rendered in the document
20. For the second test, we need to verify that the NameForm will display a success message if the name is provided.
21. [LINE 18] After rendering the component, we get the the input field by its testId
22. [LINE 19] After that we call fireEvent.change to the input field and set its target value to "Mike"
23. [LINES 20-21] We then get the submit button by its ID and perform a fireEvent.click on that button
24. Two expect methods.
    a. [LINE 22] One which will verify that the success message is rendered in the document.
    b. [LINE 23] The second one is to verify that the error message is not rendered in the document.
25. These tests will pass if we run them
26. (CTRL C and RUN NPM TEST)

React DOM Testing: Redux

Create React Redux App Command:

```
npx create-react-app demo --template redux
```

Edgecase for Redux Provider:

Just like how you wrap the App with a Provider and Store in index.js when building a Redux application, you need to use the same wrapper for your Jest tests that use Redux as well. If your Jest test is failing with the following error:

Could not find react-redux context value; please ensure the component is wrapped in a

Then, you need to import the Provider from react-redux like so:

```
import { Provider } from 'react-redux';
```

And then wrap your rendered React Component with `<Provider>` and pass either the actual store or a mock of the store.

Edgecase for React Router:

If your React component uses react-router or react-router-dom, you may need to import MemoryRouter like so:

```
import { MemoryRouter } from 'react-router';
```

And wrap your component with `<MemoryRouter>` in order for the test to pass.

Resources:

1. For more information on **Redux** testing with React Testing Library, check out Redux's official documentation HERE.
2. For more information on **React Router** testing with React Testing Library, check out React Testing Library's official documentation HERE.

Demo script

1. (RUN `NPM Start` and show browser)
2. Here we see a simple counter application. You can click Add Amount and it will add 2 to the total count
3. (CLICK ADD AMOUNT)
4. And you can click the minus and plus buttons to subtract or add 1 to the counter.
5. (CLICK MINUS AND PLUS)
6. Now let's look at the Jest test provided by the starter code.
7. (**APP.TEST.JS**)
8. Here we can see a text that simply renders the App and then verifies that the text "learn" is present in the document.
9. However, you may have noticed something different within the render method.
10. [LINES 9 and 11] Note that the App component is wrapped by another component called Provider on line 9. You'll also see that the store prop is being passed to it. Let's see what happens if we remove the Provider wrapper and rerun the test
11. (REMOVE PROVIDER TAGS AND RERUN TEST)
12. Note that the test failed. Let's scroll up and see the error
13. (Scroll to error)
14. The error states "could not find react-redux context value; please ensure the component is wrapped in a "
15. This is just an edge case you need to be aware of when testing a Redux application. Let's look at index.js
16. (**index.js**)

17. [LINES 11 and 13] Just like how you wrap the App with a Provider and Store in index.js when building a Redux application, you need to use the same wrapper for your Jest tests as well. If we add the Provider back to the test, it will pass.
18. (**app.test.js**)
19. add provider back, rerun test and show it pass)
20. One last edge case with Redux testing I'd like to cover is when you're testing a component which is using react-router or react-router-dom. For cases where your component is using a router, you may need to wrap the component with MemoryRouter in order for the test to pass like so paste

```
import { MemoryRouter } from 'react-router'; )
(wrap Provider with MemoryRouter)
```

(Doc) Week 8:

Introduction to Backend Development with Node.JS

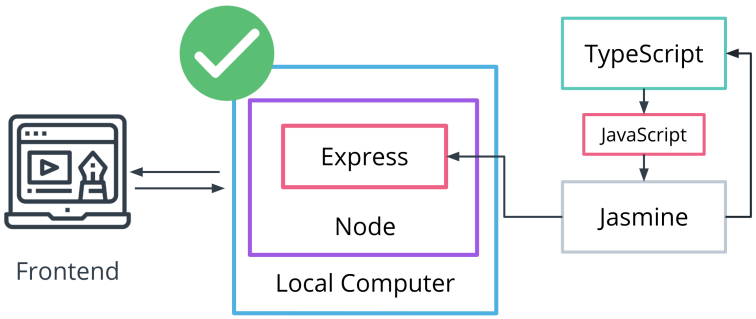
A Backend Consists of Three Parts:

- **The server:** the computing resource that listens to requests from the frontend
- **The application:** code that runs on the server to process requests and return responses
- **The database:** the part of the backend that is responsible for storing and organizing data

The backend is responsible for processing the requests that come into the app and managing its data. That can mean different things for different apps. In a simple single-page application, the backend may only be needed to host the website. In other cases, the backend is also used to store, organize, and serve data. The backend also plays an important role in authentication, security, and scalability to ensure that the system has the capacity to handle all of the incoming requests.

In This Course We'll Use:

- Node.js as our runtime
- The Express framework to initiate our server and build out the routes necessary for our application
- TypeScript for server-side code
- The testing framework Jasmine, to ensure that we're writing performant code and catching errors and edge cases before making it to production



Our Focus is on Node and Express

This focus will set you up to build scalable applications using professional tooling to create error-free, maintainable code that can be developed across a full stack team.

Stakeholders

Stakeholders in development teams include team leads that typically make service level architecture decisions, a quality assurance team that will likely work with you with testing the application, and a product owner or application manager that will make the higher level decisions and ensure that the application maintains a cohesive structure between services. Other stakeholders are management and owners above them that may be from marketing or engineering. Last and probably the most important are the customers and users who rely on their application to work as expected.

Stakeholders for Backend Development

As a developer, you are also an important stakeholder. By learning how to do backend development with Node.js, you open yourself up to being able to work across an entire stack with just JavaScript.

Term	Definition
Stakeholder	Any individual or group of individuals with an invested interest in something

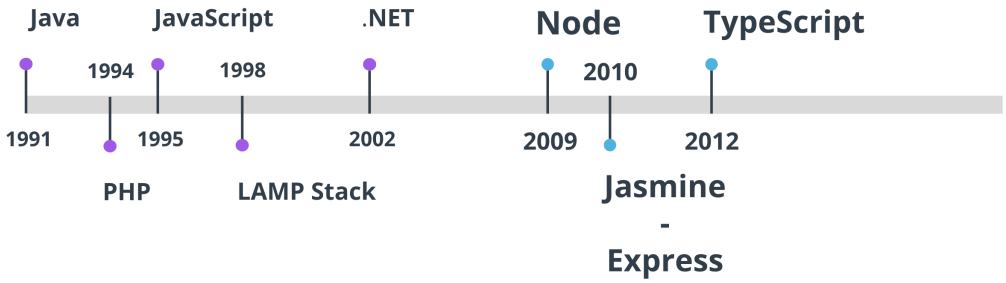
Further Reading

- This is a great article on building a development team from a stakeholder's perspective: The Web Development Team Structure You Should Know as a Stakeholder
- Learn more about the breakdown of different development team structures and who is on them: 6 Tips on How To Structure a Development Team

History of JavaScript for the Backend

- Scripting languages first appeared in the 1990s
- The LAMP stack (standing for Linux, Apache, MySQL, and PHP) was introduced in 1998
- In 2009, Node.js was created as a javascript runtime that could be used server-side
- In 2012, Microsoft introduced TypeScript to correct Typing issues with JavaScript and reduce typing errors

What's next? That's one of the most exciting and challenging parts of development! There's always something new to learn.



History of Backend Development

New Terms

Term	Definition
Runtime	The final phase in an application where the code is run

Further Reading

- What are Scripting languages?
- If you think 2009 to 2012 was an important time frame for backend development, check out this article from Log Rocket and see how much changed in frontend development in the same timeframe, much of which was made possible by Node.js: History of front-end frameworks
- Learn about the history of the internet from the place that created the internet, CERN: Where the Web was born.

Why Use Node.JS

The Node.js Advantage

- Node.js allows for JavaScript to be used on the frontend and backend.
- Node.js allows for easy application scaling and maintenance.
- Node.js is easy to learn.

Good Use Cases For Node.js

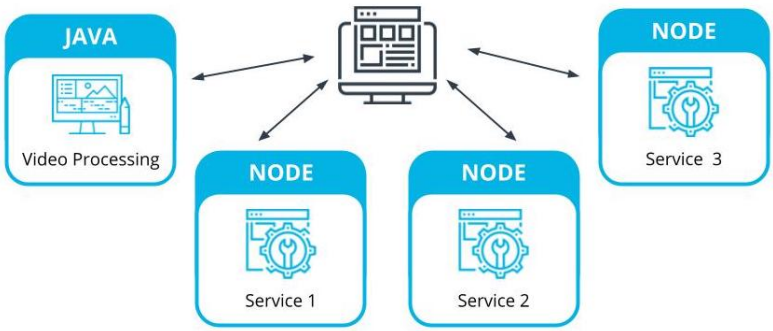
- eCommerce
- Blogging
- Chat Apps
- Social Networking
- Simple Games
- Content Management Systems

Good Use Cases for Node.js

Don't Use Node.js When Heavy Processing Is Required

Node.js is ***not*** well-suited for applications that require heavy processing and computation, like video processing, 3D games, and traffic mapping.

In these cases, you can use a microservice structure to use a different language for the services that require heavy compute power and use Node.js for the rest.



Use Node.js in the Appropriate Modules

New Terms

Term	Definition
Microservice	A piece of a larger application. In Microservice applications, the application is broken down into encapsulated microservices that can be maintained individually.
Monolithic Architecture	A unified architecture where there is no separation between services or components of the application.
Multithreaded language	The ability of a language to run 2 or more concurrent tasks on what are known as threads. A CPU has 2 threads. If you are on an 8 core machine, you could run 16 concurrent tasks.
Runtime	A runtime is an environment that is used to interpret and run a programming language.

Further Reading

- Check out the official documentation from Node.js: Node.js documentation
- Google's official documentation on their v8 engine: What is V8?

JavaScript with Node.js

Getting Started

It's easy to get Node.js installed on any system. The fastest method is to download and install the most recent LTS (long-term supported) version directly from https://nodejs.org/.

NodeJS LTS version download page for all platforms: https://nodejs.org/en/download/

Once Node.js is installed, you can open up your system's command-line utility to get started running JavaScript outside of the browser.

To test that Node.js is working, check your current version by running `$ node -v`.

More About Node.js

Updating Node.js

If you have projects using older versions of Node.js, updating can introduce breaking changes. Always check the changelog to see what the newest version contains and if it has the potential to break your project. The safest way to update is to backup any projects using Node.js to a repository. Install the latest version of Node.js through nodejs.org, then update all of the project's dependencies. If the risk of introducing breaking changes seems high, Node Version Manager makes it possible to run multiple versions of Node.js on the same system.

Containers

Containers include the runtime, all configurations, and files needed to ensure that all individuals working on a project have the same environment regardless of the operating system or software installed globally. There are several choices to use to create a container for your application. Typically when working on an enterprise project, you will install a container software and run your project within that container which will also include a version of node.js that won't interfere with the version installed globally on your system.

Running JavaScript with Node.js

Two Options for Running JavaScript Code with Node

et's have a look at our two options for running JavaScript code in files with NodeJS. The first method is through accessing nodes REPL environment, which stands for read, evaluate, print, loop. REPL runs JavaScript directly in your console application. You can use Command D to exit.

Using REPL

The first method is through accessing Node's **REPL** environment. **REPL** stands for Read, Evaluate, Print, and Loop.

To access the Node.js REPL environment, run:

```
$node
```

To exit the REPL environment, use ***cmd+d*** on a Mac or ***ctrl+d*** on a Windows or Linux machine.

Using the `node` Command

The second method is by running JavaScript files using the `node` command in your command line tool, followed by the path to the file.

You can run

```
$ node src/index.js
```

or

```
$ node src/index
```

or

```
$ node src/.
```

or

```
$ node src
```

or

```
$ node ./src/index.js
```

To run other files use:

```
$ node src/filename.js
```

or

```
$ node src/filename
```

or

```
$ node ./src/filename
```

New Terms

Term	Definition
Container	A tool used to encapsulate the entirety of an application including runtime, libraries, and files to run independently of global configurations on an operating system
Node Version Manager (nvm)	A software package that allows a user to run a different version of Node.js for each project on the same machine
REPL (Read, Evaluate, Print, Loop)	An environment used for running programming languages

Further Reading

Learn more about containers from the most popular container application Docker: What is a Container?

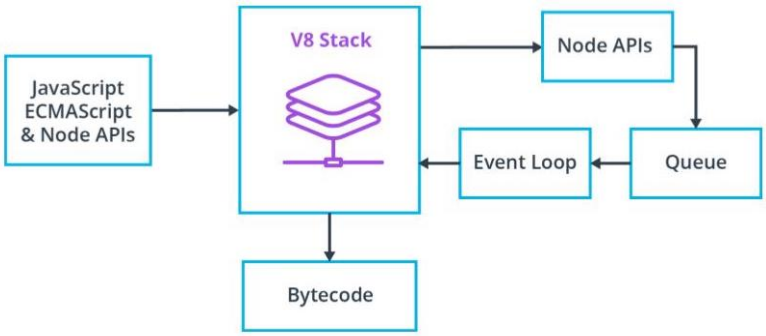
Node.JS Globals

Node.js and the V8 Engine

Node.js uses Google's V8 engine as a runtime to process JavaScript.

JavaScript is sent to the V8 engine in Node.js. Any asynchronous code is:

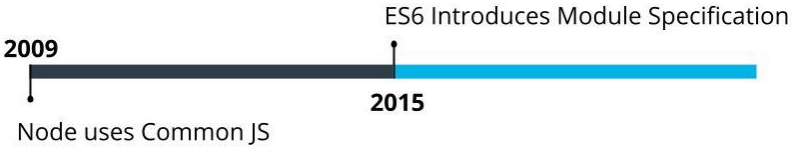
1. Sent off to be processed by the Node APIs
2. Added to the queue
3. Processed by the event loop
4. Sent back to the V8 Engine
5. Processed to bytecode.



How JavaScript is Processed

There are some APIs that overlap between the browser and Node.js, like Timers and Console, but others that are exclusive to the browser or Node.js like File System, which is exclusive to Node.js.

Node.js needed a module system as far back as 2009, 6 years before the ES6 Module system. Node.js used the Common JS module system to break code into smaller chunks; It's still used today. TypeScript (a JavaScript superset) compiles to the Common JS Module System.



Node Started Using Modules in 2009

The Module System

Common JS Module System

The module system creates the ability to export and import JavaScript from separate files.

Export

```
// working file = util/logger.js

// exports as object
module.exports = {
```

```
    myFirstFunction: myFirstFunction,
    mySecondFunction: mySecondFunction
}

// using ES6 shorthand property names
module.exports = {
    myFirstFunction,
    mySecondFunction
}
```

It is most common to see shorthand property names in use, however, some developers choose to use very specific function names when developing their libraries, and then create short names for the functions when exporting choosing to document the short names for ease of the library user, but adding to the maintenance overhead for the library developer.

Require

```
// working file = index.js
// all functions in util/logger.js are available
const logger = require('./util/logger.js');

// using ES6 object destructuring, only myFirstFunction is available
const { myFirstFunction } = require('./util/logger.js');
```

It is standard to name the const the same as the file or module name. Omitting the file extension is safe and common practice. Destructuring is often used when only requiring one function from an otherwise large library.

When using require, a preceding slash must come before a locally created module name; otherwise, Node.js will search the core modules and then node_modules (discussed later).

`dirname` **and** `filename`

The scope of the module specification includes access to both the directory of the module as well as access to the absolute path of the filename of a file within a module. The ability to access directories and filenames becomes especially useful when working with the path and filesystem modules discussed later.

```
// working file = /app/util/logger.js

console.log(__dirname);
// prints /app/util

console.log(__filename);
// prints /app/util/logger.js
```

Timers

The Node.js timers API is similar, but not identical to the browser API. Node v11 introduced what would be for some projects, a breaking change in that it made timers behave even more similar to the browser API.

You're likely familiar with `setTimeout();`, `setInterval();`, `clearInterval();` and `clearTimeout();` already; Node.js introduces `setImmediate();` and `clearImmediate();` which get's its own phase in the Node.js event loop which we will look at in the event loop lesson.

Console

Like Timers, the Console API is likely familiar to you as part of the browser API. We typically learn to use `console.log` as our first debugging method with JavaScript and that it only serves to debug and should be removed in shipped code.

The options available within the Node.js console are generally the same, and console.log serves well as an initial debugging tool. However, `console` can serve as an incredible tool for giving server-side feedback, such as letting you know that you have successfully connected to the server or creating a node module that takes in information from the user through a console application. It can be used in development for logging events, but it is best practice to log to a file to save the data rather than just logging to the terminal. The use of `console` is typically blocking (though there are some inconsistencies between versions and operating systems.

Examples

```
console.log('Server ready');
```

- Prints to the console using standard output (stdout)
- Creates a new line for each statement.

```
console.error('Server failed');
```

- Shows in the console using standard error (stderr)
- Creates a new line for each statement.

In the browser API, it is clear when `console.error();` is used. In Node.js, there may be little to no distinction visually between which is used.

More Options for `console`

There are many more options available and it is worth it to see all that console is capable of.

Note: using `console` should never take the place of proper error handling. Any console statement that remains in your shipped code should serve a specific purpose outside of general debugging.

Further Reading on Error Handling

- Read more about Node.js's error class from the Node.js documentation.
- Read more on error handling in Node.js from Smashing Magazine's Kelvin Omereshone: Better Error Handling In NodeJS With Error Classes

Process Module

Not found in the browser APIs, Process relates to the global node execution process which occurs when you run a js file through Node.js. Process has many options available--we will focus on the most commonly seen.

Examples

`process` events

The Process module contains the ability to perform tasks immediately before the process exits, and when it exits. `beforeExit` allows for asynchronous calls which can make the process continue whereas `exit` only happens once all synchronous and asynchronous code is complete.

```
// create conditions for exit code options
// example: 0 typically implies without errors, 1 with.

process.exitCode = 1;

process.on('beforeExit', () => {
  console.log('beforeExit event');
});

process.on('exit', (code) => {
```

```
  console.log(`exit event with code: ${code}`);
});
```

`process.env`

```
console.log(process.env);
```

Process.env gives you access to the environment information of your Node.js application. It also allows you to add environment variables that can be used if your code is dependent on the environment it is run in. With the use of a module like dotenv you can easily control your project's configuration in separate .env files based on what environment you are using (ex. production vs test vs development).

Common reasons include: changing the port or IP, accessing static files, or access to the production vs development databases.

`process.stdout`

A lesser-known fact is that console.log actually utilizes `process.stdout` in order to log to the console.

`process.stdout.write('Hello, world.');` and `console.log('Hello, world.');` are nearly identical except for one very important difference, `process.stdout` does not force a new line break. This allows you to create helpful tools like progress bars.

`process.argv`

An array containing your console arguments information for your executed process.

```
// index.js
// When run, will output an array of all arguments supplied to the node process.

console.log(process.argv);
// Terminal

$ node index.js argument1 argument2
[
  '/usr/local/bin/node',
  '/Users/user/Desktop/app/index.js',
  'argument1',
  'argument2'
]
```

When index.js was run through Node.js, the entire command contained four arguments, so the array has four values, the location of Node.js on the system, the location of the file run, and then two additional arguments.

`process.argv` allows you to pass in arguments to your application which can be a common occurrence when needing to parse data from files.

`process.nextTick`

Allows you to run JavaScript between the different phases of the event loop. `process.nextTick` will be described in detail when discussing the event loop.

New Terms

Term	Definition

Interpreted Language	The language is read by a runtime and executed on the spot and errors are found on execution
Superset	A language that extends or builds on top of another language or standard.

Further Reading

More on Node.js globals from the Node.js documentation.

Node.JS Core Modules

Path Module

Windows

```
filePath = 'app\\src\\routes\\api';
```

Mac/Linux

```
filePath = 'app/src/routes/api';
```

In the above examples, if you were working exclusively in a Windows environment, then you could safely use that file path structure, but if a member of your team who uses a macOS or Linux system were to join the project, the filePath would no longer work for them since the syntax is different for macOS/Linux systems. Using the path module allows us to normalize paths to work across platforms.

The path module must be imported via `const path = require('path');`. Once imported, there are three commonly used options that you should know.

`path.resolve`

Enables you to get the absolute path from a relative path.

```
console.log(path.resolve('index.js'));

// prints /Users/user/Desktop/app/index.js
```

`path.normalize`

Normalizes any path by removing instances of `.` , turning double slashes into single slashes and removing a directory when `..` is found.

```
console.log(path.normalize('./app//src//util/..'));

// prints app/src/util
```

`path.join`

Used to concatenate strings to create a path that works across operating systems. It joins the strings, then normalizes the result.

```
console.log(path.join('/app', 'src', 'util', '..', '/index.js'));

// prints  /app/src/index.js
```

File System Module

The File System Module is highly sophisticated and must be imported using the module system's `const fs = require('fs');`. File system (fs) allows for reading and writing to files with many options.

We will dive into the file system module in the last lesson. Until then, if you would like to do your own investigation work, check out the Node.js documentation on the file system module.

Other Core Modules

- **HTTP/HTTPS** is used to transfer data. Later on, we'll be using Express, which builds on top of this module, to create our server
- **URL** is used for parsing and resolving URLs

- **TLS/SSL** implements security protocols on top of OpenSSL There are more core modules worth checking out in the Node.js documentation

Further Reading

Learn more about Node.js Core Modules from the Node.js documentation.

The Event Loop

Event Loop

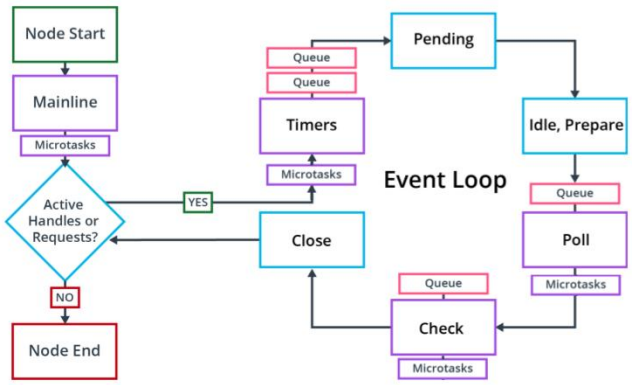
Nearly every Node.js feature is considered to be asynchronous (non-blocking). This means that we can request an API using promises and have our application continue running while that request is being waited for. But how does Node.js process that asynchronous request? Both the Browser and Node.js take advantage of something called the Event Loop. The Event Loop isn't an API or language; it's a process that runs anytime you have asynchronous code.

The Event Loop controls the order in which results (output) of asynchronous tasks (input) are displayed. Think of the Event Loop as the person working the door at an exclusive venue. That person lets people in based on a set of information provided by the venue. Your application is the venue, your asynchronous tasks are the people trying to get in, and it's your job to tell the door person how to do so. Once you become familiar with the Event Loop and the order in which Node.js handles tasks, you will control when those tasks occur in your application.

Six Phases of The Event Loop

1. **Timers** - executes callbacks using timers. If there are timers set to `0 ms` or `setImmediate(),` they will run here. Incomplete timers will run in later iterations of the loop.
2. **Pending** - *internal phase*
3. **Idle/Prepare** - *internal phase*
4. **Poll** - process I/O callbacks
5. **Check** - execute any `setImmediate()` timers added in the Poll phase
6. **Close** - loop continues if there are more timers or I/O calls. If all timers and I/O calls are done, the loop closes and the process ends.

NOTE: `process.nextTick();` will always run at the end of whichever phase is called and before the next phase.



The Event Loop

A Bit More About the Polling Phase

The polling phase is a bit more complex than just adding to the poll queue. If there are no timers left to execute when the polling phase is reached, the poll phase will wait for input/output callbacks. If the I/O contains synchronous code, this code will not be added to the call stack till the polling phase is reached. If setImmediate() is reached, the polling phase will end and the check phase will begin. If setImmediate() is not called, the polling phase will continue to wait for a bit, and then move on through the next phases to execute additional timers and so forth.

Let's walk through a demonstration to help you visualize the loop. The code you see will be unfamiliar at this point, but by the end of this course, you'll fully understand it.

Further Reading

Read the official Node.js documentation on the event loop: What is the Event Loop?.

Node Package Manager

What Is Node Package Manager (NPM)?

NPM is both a **tool for managing project dependencies** via command line and a **website hosting more than 1 million third-party packages** that can be used for your project.

- Modules are shared as packages
- Packages extend the functionality of your app
- Modules are stored in the app's `node_modules` folder
- Core modules include `path`, `Filesystem`, and more

Initializing `npm` and Creating a `package.json` file

Initializing npm will create a package.json within the root of your application folder containing general information about the project.

To initialize `npm` and go through all of the settings use:

```
npm init
```

To automatically select all defaults use `-y`

```
npm init -y
```

Adding Dependencies

Applications will either include both dependencies and devDependencies or just dependencies. It is dependent on the team setting up the project. `devDependencies` are thought of as dependencies that are only necessary for development whereas `dependencies` are those dependencies used in both development and production. An example would be needing TypeScript added as a dependency for development, but since it compiles to standard JavaScript to be used in production, TypeScript is not needed for production and therefore could be just a `devDependency`. Many teams find little use in separating but when learning, it can be a helpful practice to determine which dependencies are only being used in development vs which are also needed for production.

```
npm i module-name // install module to dependencies
npm i --save-dev module-name // install to dev dependencies
npm i --save-dev module-name@1.19 // install a specific version (1.19 here) of module
```

Installing dependencies adds the dependency to your package.json file in the format:

```
  "devDependencies": {
    "prettier": "^2.2.1"
  }
```

Pay special attention to the version listed. The format is as follows.

- First number = major version
- Second number = minor release
- Third number = patch

The version states what was installed, but it also clarifies how it can be updated should you remove the node_modules and package-lock.json files and reinstall all dependencies with `$ npm install`.

The additional included characters (or lack thereof) tell npm how to maintain your dependencies.

- `*` means that you'll accept all updates
- `^` means that you'll only accept minor releases
- `~` means that you'll only accept patch releases

- `>`, `>=`, `<=`, `<` are also valid for saying you'll accept versions greater/less/equal to the listed version
- `||` allows you to combine instructions `"prettier": "2.2.1 || >2.2.1 < 3.0.0"` which says use prettier greater than 2.2.1 and less than version 3.0.0
- You can also leave off a prefix and only accept the listed version

package-lock.json

`package-lock.json` contains all of the information for the dependencies of the modules you have installed.

It is best practice to add `package-lock.json` as well as `/node_modules` to your `.gitignore` file when using a repository. The `node_modules` folder can grow rapidly, containing thousands of files. It is best to clone a repository without `node_modules` and run `npm install` to reinstall all dependencies of the project directly from npm.

npm update

running `npm update` will update all of your dependencies based on the specifications given in your package.json file.

Scripts

To run a script that you have added to your package.json file, simply run $ npm run argument with the name of the script as the argument.

```
npm run prettier
```

Using Prettier

Prettier is a code formatter that will ensure you're keeping your code consistent. It's commonly added to projects to ensure all members on a team are formatting in a consistent way such as always using semicolons, trailing commas, and single quotes. It can be configured to the preferred settings of the team and works well with additional tools like linting.

We are able to add it to a project with NPM by doing the following:

- Locate prettier on npm | Home to get the install script and other information.
- Run the install script `npm i --save-dev prettier`.
- Add a prettier script to your `package.json` file. The script you choose can vary dramatically depending on the project. The one below will only overwrite files located in the src directory that are js files. You may need a different script depending on the project.

```
// example config file, path structure to check, and write fixes
"prettier": "prettier --config .prettierrc 'src/**/*.js' --write"
// or
"prettier": "prettier --config .prettierrc \"src/**/*.js\" --write"
```

- Create a `.prettierrc` file for any custom configurations.
- Run `npm run prettier` to run prettier (or whatever you named your script).

> NOTE: It's common to encounter deprecation warnings when working with NPM packages. Packages may have multiple dependencies. If one updates before the other, you may encounter one of these warnings. They are typically taken care of within the next 2 updates of the package. It's best to look them up when you find them to see if someone is actively working to repair the issue or to see is a better solution.

New Terms

Term	Definition

dependencies	Dependencies used in both development and production
devDependencies	Dependencies that are only necessary for development
Node Package Manager (npm)	A tool for managing project dependencies via command line as well as a website hosting more than 1 million third-party packages that can be used for your project
package-lock.json	A JSON file that contains all of the information for the dependencies of an app's installed modules
package.json	A JSON file that acts as a manifest for your project including name, author, version, description, license, dependencies, scripts, etc.
Prettier	A code formatting package that can be integrated into projects to improve code consistency and readability

Further reading

Check out the full documentation on Prettier to see how it's capable of improving your projects.

TypeScript

Intro to TypeScript

JavaScript is Weakly-Typed

This means that types are assigned by the interpreter based on the data and makes an educated guess when the code's intention is ambiguous. This can lead to unintended results.

Example:

```
2 + '2' = '22'
```

TypeScript Adds Typing to JavaScript

In short, TypeScript is a static and strong typed superset of JavaScript. When we're done with our TypeScript code, it compiles to JavaScript.

TypeScript may not be needed if you have a lot of code quality measures in place. Still, with how easy it is to learn and implement, it's generally worth it to reduce coder errors, and TypeScript offers the developer the ability to state their intentions clearly.

TypeScript Is A MUST for Large or Growing Teams!

As a project/team grows larger, the room for errors in the development flow grows.

- TypeScript is a way to reduce developer error efficiently
- Typing is familiar to developers who have worked with strictly typed languages like Java, C#, and C++
- TypeScript is easy to learn
- Used in popular front-end frameworks including Angular, React and Vue
- Supported by all major IDEs and code editors including Visual Studio Code, Sublime and Atom

Installing TypeScript

TypeScript can be installed locally or globally. If interested in working with TypeScript outside of a node.js project, you would install TypeScript globally to have access to the compiler globally. We will be working with TypeScript locally. To install locally, use one of the following 2 commands based on need. Again, using devDependencies when you begin working with Node.js can be helpful in learning which modules are required only for development purposes and which are required for production.

```
$ npm i typescript // save to dependencies
$ npm i typescript --save-dev // save to devDependencies
```

NPX and Creating your package.json Script

To use TypeScript, you need to add a script to your package.json file to compile TypeScript to JavaScript. This is generally called your "build" script but could be named anything.

```
"scripts": {
    "build": "npx tsc"
  },
```

npx comes packaged with npm by default. **npx** first checks that a package is in your project; if it is, it executes the package; if not, **npx** installs then executes that package. The command **npx tsc** in a project will transpile TypeScript to JavaScript.

To execute your "build" script use the following:

```
$ npm run build
```

TypeScript Basics

Basic Type

string - used for string types, textual data

```
let studentName:string;
studentName = 'Dae Lee'
```

number - used for number types including integers and decimals

```
let studentAge: number;
studentAge = 10;
```

boolean - used for true/false types

```
let studentEnriched: boolean;
studentEnriched = true;
```

Union Types - used when more than one type can be used

```
let studentPhone: (number | string);
studentPhone = '(555) 555 - 5555';
studentPhone = 5555555555;
```

null - used when an object or variable is intentionally **null**, typically only functionally found in union types

```
const getCapitals = (str:string):string[] | null => {
  const capitals = str.match(/[A-Z]/);
  return capitals;
}

console.log(getCapitals('something'));
// returns null
console.log(getCapitals('Something'));
// returns ['S']
```

undefined - used when a variable has yet to be defined

```
const myFunc = (student: string | undefined) => {
  if ( student === undefined ){
    // do something
  }
};
```

void - used as a return type when the function returns nothing

```
const myFunc = (student: string): void => {
  console.log(student);
};
```

never - used as a return type when the function will never return anything, such as with functions that throw errors or infinite loops

```
const myError = (err: string): never => {
  throw new Error(err);
}
```

any - should be avoided. Used when the type of the item being typed can be anything

```
const myFunc = (student: any): any => {
  // do something
};
```

unknown - used when the type of the thing being typed is unknown. Used heavily for type assertion

```
const myFunc = (student: unknown): string => {
  // do something
}
```

Objects and Interfaces

Objects are easily created in JavaScript due to JavaScript's weak typing. With TypeScript, they take a bit more work. It is possible to create an object in TypeScript, but TypeScript offers better tools for doing so.

Object - creating an object requires defining the object before setting values. Once you have defined the object, additional properties cannot be added to the type definition, making it unhelpful when you need to add more properties after creation.

```
let student:{ name: string, age: number, enrolled: boolean} = {name: 'Maria', age: 10, enrolled: true};
```

interface - Interfaces are a concept not native to javascript, but similar concepts exist in other languages like Java, C++, and Python, where you create an abstract class as an interface for creating classes. With TypeScript, interfaces are simply used as the blueprint for the shape of something. Interfaces can be used to create functions but are most commonly seen to create objects.

Interfaces have the ability to be added too without the need to be extended. Meaning, if you have an interface, you can declare that interface a second time and add additional properties to it. This allows you to easily work with third-party interfaces that may need additional properties.

Use PascalCase for naming interfaces.

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean
};
let newStudent:Student = {name: 'Maria', age: 10, enrolled: true};
```

Optional and Readonly Properties

Typescript gives the ability to create both optional and read-only properties when working with object-like data.

Optional - use when an object may or may not have a specific property by adding a ? at the end of the property name.

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean,
  phone?: number // phone becomes optional
};
```

readonly - use when a property should not be able to be modified after the object has been created. Keep in mind that this will only produce TypeScript errors and that the actual properties can still technically be changed as read-only does not exist in JavaScript. The closest thing in JavaScript is **Object.freeze** which will make all properties of the object unable to be modified.

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean,
  readonly id: number // id is readonly
};
```

Classes

TypeScript classes look and behave very much like the classes introduced in ES6. A class in programming is made up of member variables and member functions/methods. The same goes for TypeScript, with the big difference being our variables (properties) are typed, as are the parameters and return types for our constructor and methods.

```
class Student {
  studentGrade: number;
  studentId: number;
  constructor(grade: number, id: number) {
    this.studentGrade = grade;
    this.studentId = id;
  }
}
```

Factory Functions

If Factory Functions remain your preferred way of creating JavaScript objects, they are still useable within TypeScript. To create a factory function with explicit typing, create an interface with the object's properties and methods and use the interface as the return type for the function.

```
interface Student {
  name: string;
  age: number
  greet(): void;
}

const studentFactory = (name: string, age: number): Student =>{
  const greet = ():void => console.log('hello');
  return { name, age, greet };
}

const myStudent = studentFactory('Hana', 16);
```

Access Modifiers

The biggest addition to TypeScript classes is the addition of access modifiers. Access modifiers are used in most object-oriented programming languages to declare how accessible a variable should be.

public - by default, all properties of a TypeScript class are public, and it's commonplace to leave off the modifier to denote that it's public. Public properties can be accessed outside of the class.

private - private properties can only be accessed and modified from the class itself. TypeScript uses the keyword private, but you can also use the # symbol now available for privatizing fields in EcmaScript. Remember that private properties are only private in TypeScript; there are no true private properties in JavaScript classes, so a private property can still be changed if you ignore the error.

protected - protected properties can be accessed by the class itself and child classes.

```
class Student {
  protected studentGrade: number;
  private studentId: number;
  public constructor(grade: number, id: number) {
    this.studentGrade = grade;
    this.studentId = id;
  }
  id(){
    return this.studentId;
  }
}

class Graduate extends Student {
  studentMajor: string; // public by default
  public constructor(grade: number, id: number, major: string ){
      super(grade, id);
      this.studentId = id; // TypeScript Error: Property 'studentId' is private and only accessible within class 'Student'.
      this.studentGrade = grade; // Accessable because parent is protected
      this.studentMajor = major;
  }
}

const myStudent = new Graduate(3, 1234, 'computer science');

console.log(myStudent.id()); //  prints 1234
myStudent.studentId = 1235; // TypeScript Error: Property 'studentId' is private and only accessible within class 'Student'.
console.log(myStudent.id()); // prints 1235
```

Introduction to TypeScript

What Is Typescript?

TypeScript is a free and open source high-level programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript and adds optional static typing to the language. It is designed for the development of large applications and transpiles to JavaScript.

Why Use TypeScript?

TypeScript builds on JavaScript to effectively fill in the gaps and give developers better tooling at any scale. Weighed against its competitors, TypeScript is both easier and more efficient.

Ease of Use

One of the chief advantages of TypeScript is its ease of use. If you are at least a little familiar with JavaScript, it will require very little effort to get started with TypeScript. This is because all TypeScript code is converted into its JavaScript code equivalent for execution.

Conversely, any JavaScript (.js) file can be renamed to a TypeScript (.ts) file for compilation with other TypeScript files.

Portability

In many ways, TypeScript *is* JavaScript, i.e., TypeScript code converts to JavaScript to run anywhere JavaScript runs. Consequently, users can have confidence that TypeScript can run on any environment that JavaScript runs on — browsers, devices, and operating systems.

This starkly contrasts with many TypeScript competitors that require a dedicated VM or specific runtime environments for execution.

Robust Developer Tooling Support

Overall, TypeScript aims to improve developers' efficiency and productivity by mitigating errors, aiding problem-solving, and delivering better tooling at scale.

TSC is also advantageous for developers because it can run as a background process to support compilation and IDE integration.

Installing and Configuring TypeScript

Installing TypeScript

TypeScript can be found on npm's website here. TypeScript can be installed locally or globally. If interested in working with TypeScript outside of a node.js project, you would install TypeScript globally to have access to the compiler globally. We will be working with TypeScript locally. To install locally, use one of the following 2 commands based on need. Again, using `devDependencies` when you begin working with Node.js can be helpful in learning which modules are required only for development purposes and which are required for production.

```
$ npm i typescript // save to dependencies
$ npm i typescript --save-dev // save to devDependencies
```

NPX and Creating your package.json Script

To use TypeScript, you need to add a script to your package.json file to compile TypeScript to JavaScript. This is generally called your "build" script but could be named anything.

```
"scripts": {
    "build": "npx tsc"
  },
```

`npx` comes packaged with npm by default. `npx` first checks that a package is in your project; if it is, it executes the package; if not, `npx` installs then executes that package. The command `npx tsc` in a project will transpile TypeScript to JavaScript. You can learn more about `npx` from the npm documentation.

To execute your `"build"` script use the following:

```
$ npm run build
```

Configuring TypeScript

`tsconfig.json` can also be named `jsconfig.json`.

To install the config file, run

```
$ npx tsc --init
```

You should always check your compiler options to note what you are transpiling to as well as your output directory. Common output directory names include `dist`, `build`, `prod`, and `server`.

This config file is also where you can tell TypeScript how strict it should be while checking your code and what to ignore. If you're moving a project to TypeScript, you can gracefully integrate TS by working with the settings in this config file.

ES6 Modules

Now that we are using TypeScript for our application, we can also easily utilize the ES6 module system instead of the CommonJS module system. Destructuring should only be used when you are exporting the functions individually. If choosing to use export default, you must import the entire default as a module.

Import

For importing modules, use the following syntax

```
// Rename the module
import 'name' from 'module';

```

```
// Use destructuring to pull in specific functions when they are exported individually
import {function, function} from 'module';
```

Export

```
// Export an individual function or other type of object in code
export const myFunction = () => {};

// Export a single item at the end
export default object;

// Export a list of objects
export default {object1, object2};
```

More TypeScript Configurations

Helpful configurations to note:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "lib": ["ES2018", "DOM"],
    "outDir": "./build",
    "strict": true,
    "noImplicitAny": true,
  },
  "exclude": ["node_modules", "tests"]
}
```

You will see many more options available than what is above. Your application may require additional settings to be configured, but these are typically the main settings to start with.

- `target` - sets what version of JS TypeScript will be transpiled to.
- `module` - sets what module system will be used when transpiling. Node.js uses the common.js module system by default
- `lib` - is used to say what libraries your code is using. In this case, ES2018 and the DOM API
- `outDir` - where you want your src code to output to. Often named build, prod, or server (when using it serverside)
- `strict` - enable strict typing
- `noImplicitAny` - disallow the `"any"` type (covered in TypeScript Basics)
- `exclude` - directories to exclude in compiling

Further Reading

- Catch up on ES6 modules if you haven't had the opportunity to work with them yet.
- Official documentation from Microsoft on installing TypeScript.
- Explicit instructions from Microsoft on Installing TypeScript and working with NPM and a text editor.

How to compile a Typescript file?

TypeScript file

Now, run the following command in the command prompt to compile the typescript file. This will create a javascript file from typescript automatically with the same name.

Compiling TypeScript file: `tsc fileName.ts`

Run the JavaScript file: `node fileName.js`

Output: `"Welcome to GeeksforGeeks!"`

TypeScript Basics

Implicit Typing and Explicit Typing

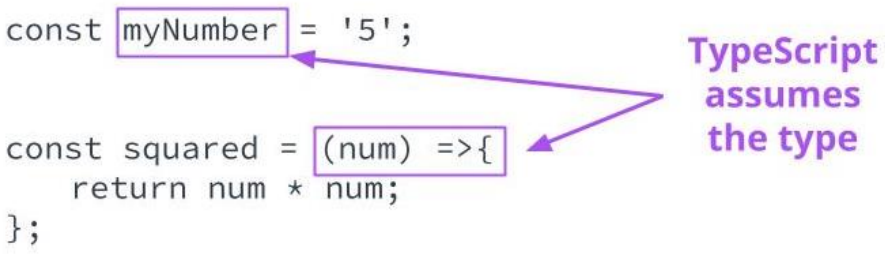
TypeScript offers two types of typing:

Implicit Typing

TypeScript will automatically assume types of objects if the object is not typed. It is best practice to allow TypeScript to type immutable variables and simple functions implicitly.

```
const myNum = 3; // TypeScript implicitly types myNum as a number based on the variable

```

Implicit Typing is a best practice when the app is self-contained (meaning that it does not depend on other applications or APIs) or variables are immutable.

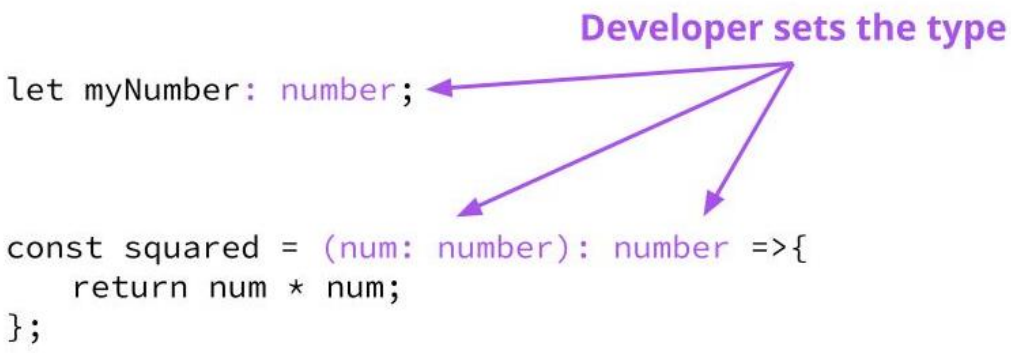


Implicit Typing

Explicit Typing

The developer does explicit typing. The developer explicitly applies a type to the object.

```
let myVar: number = 3; // myVar has been explicitly typed as a number

```



Explicit Typing

Basic Types

`string` - used for string types, textual data

```
let studentName:string;
studentName = 'Dae Lee'
```

```
3
```

`number` - used for number types including integers and decimals

```
let studentAge: number;
studentAge = 10;

```

`boolean` - used for `true` / `false` types

```
let studentEnriched: boolean;
studentEnriched = true;

```

Union Types - used when more than one type can be used

```
let studentPhone: (number | string);
studentPhone = '(555) 555 - 5555';
studentPhone = 5555555555;

```

`null` - used when an object or variable is intentionally `null`, typically only functionally found in union types

```
const getCapitals = (str:string):string[] | null => {
  const capitals = str.match(/[A-Z]/);
  return capitals;
}

console.log(getCapitals('something'));
// returns null
console.log(getCapitals('Something'));
// returns ['S']

```

`undefined` - used when a variable has yet to be defined

```
const myFunc = (student: string | undefined) => {
  if ( student === undefined ){
    // do something
  }
};

```

More Basic Types

`void` - used as a return type when the function returns nothing

```
const myFunc = (student: string): void => {
  console.log(student);

```

```
};

```

`never` - used as a return type when the function will never return anything, such as with functions that throw errors or infinite loops

```
const myError = (err: string): never => {
  throw new Error(err);
}

```

`any` - *should be avoided*. Used when the type of the item being typed can be anything

```
const myFunc = (student: any): any => {
  // do something
};

```

`unknown` - used when the type of the thing being typed is unknown. Used heavily for type assertion

```
const myFunc = (student: unknown): string => {
  // do something
}

```

Type Assertions

Type Assertions are used to tell TypeScript that even though TypeScript thinks it should be one type, it is actually a different type. Common to see when a type is `unknown`

```
const myFunc = (student: unknown): string => {
  newStudent = student as string;
  return newStudent;
}

```

`typeof`

If you run into a situation where you have an ambiguous function, and you don't know exactly what it's doing, or you're working with a third-party library, and type definitions are missing, and you quickly want to access the type, one way of doing so is using `typeof`. This won't work for every type, such as null returning an object, but it will work for most.

```
console.log(typeof myFunc(param));

```

Further Reading

Official documentation from Microsoft on TypeScripts basic types.

A cheat sheet for TypeScript, from SitePen, the founders of TF Conf, a conference for TypeScript.

New Terms

Term	Definition
Explicit Typing	When type is declared by the developer
Implicit Typing	When type is Inferred by the compiler
Self-contained application	The use of self-contained implies the application receives data from no external sources

Object-Like Types

Arrays are critical data structures to JavaScript, so it makes sense that with TypeScript, there are multiple ways of creating them depending on the content that goes inside. TypeScript offers 2 ways of working with arrays and a third that can feel more like an array or an object depending on how it is used.

Array - to type as an array, use the type, followed by square brackets. Union types can be used to allow for multiple types in an array.

```
let arr: string[]; // only accepts strings
let arr2: (string | number)[]; // accepts strings or numbers

```

Tuple - tuples are not native to JavaScript. When you know exactly what data will be in the array, and you will not be adding to the array or modifying the type for any value, you can use a tuple.

```
let arr: [string, number, string]; // ['cat', 7, 'dog']

```

`enum` - enums are not native to JavaScript but are similar to enumeration used in other languages like C++ and Java. You use an enum when you have a constant set of values that will not be changed. By default, the values in an enum are also given a numeric value starting at 0. However, the numeric value can manually be set to any number explicitly or by calculation. Uses PascalCase to name the type.

```
enum Weekend {
  Friday,
  Saturday,
  Sunday
}

```

Working With Objects in TypeScript

Objects and Interfaces

Objects are easily created in JavaScript due to JavaScript's weak typing. With TypeScript, they take a bit more work. It is possible to create an object in TypeScript, but TypeScript offers better tools for doing so.

Object - creating an object requires defining the object before setting values. Once you have defined the object, additional properties cannot be added to the type definition, making it unhelpful when you need to add more properties after creation.

```
let student:{ name: string, age: number, enrolled: boolean} = {name: 'Maria', age: 10, enrolled: true};

```

`interface` - Interfaces are a concept not native to javascript, but similar concepts exist in other languages like Java, C++, and Python, where you create an abstract class as an interface for creating classes. With TypeScript, interfaces are simply used as the blueprint for the shape of something. Interfaces can be used to create functions but are most commonly seen to create objects.

Interfaces have the ability to be added too without the need to be extended. Meaning, if you have an interface, you can declare that interface a second time and add additional properties to it. This allows you to easily work with third-party interfaces that may need additional properties.

Use PascalCase for naming interfaces.

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean
};
let newStudent:Student = {name: 'Maria', age: 10, enrolled: true};

```

Duck Typing

Duck Typing is a programming concept that tests if an object meets the duck test: "If it walks like a duck and it quacks like a duck, then it must be a duck."

TypeScript uses duck typing for interfaces, meaning that even though you may say a function takes in an argument of interface A, if interface B has the same properties of A, it will also accept B. Interface A is the duck, and Interface B walks and quacks like a duck, so we'll accept it as a duck too.

Optional and Readonly Properties

Typescript gives the ability to create both optional and read-only properties when working with object-like data.

Optional - use when an object may or may not have a specific property by adding a `?` at the end of the property name.

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean,
  phone?: number // phone becomes optional
};

```

`readonly` - use when a property should not be able to be modified after the object has been created. Keep in mind that this will only produce TypeScript errors and that the actual properties can still technically be changed as read-only does not exist in JavaScript. The closest thing in JavaScript is `Object.freeze` which will make all properties of the object unable to be modified.

```
interface Student {
  name: string,
  age: number,
  enrolled: boolean,
  readonly id: number // id is readonly
};

```

Type Aliases, TypeScript Classes and Factory Functions

Type Aliases

Type aliases do not create a new type; they rename a type. Therefore, you can use it to type an object and give it a descriptive name. But like the object type, once a type alias is created, it can not be added to; it can only be extended. Meaning, if you wanted to create an object from a type alias and then a second with additional properties, you would need to extend the type alias and make your second object with the extended alias. This makes interfaces the preferred method for creating objects.

Type aliases become very useful when you would like a shorthand for something like a specific union type or a tuple with a specific meaning. For example, if I needed to create multiple arrays of coordinates, I could create a tuple that accepts 2 numbers, call it Coordinate and create multiple arrays of type Coordinate.

```
type Student = {
  name: string;
  age: number;
  enrolled: boolean;
};

let newStudent:Student = {name: 'Maria', age: 10, enrolled: true};

```

Classes

TypeScript classes look and behave very much like the classes introduced in ES6. A class in programming is made up of member variables and member functions/methods. The same goes for TypeScript, with the big difference being our variables (properties) are typed, as are the parameters and return types for our constructor and methods.

```
class Student {
  studentGrade: number;
  studentId: number;
  constructor(grade: number, id: number) {
    this.studentGrade = grade;
    this.studentId = id;
  }
}

```

Access Modifiers

The biggest addition to TypeScript classes is the addition of access modifiers. Access modifiers are used in most object-oriented programming languages to declare how accessible a variable should be.

`public` - by default, all properties of a TypeScript class are public, and it's commonplace to leave off the modifier to denote that it's public. Public properties can be accessed outside of the class.

`private` - private properties can only be accessed and modified from the class itself. TypeScript uses the keyword private, but you can also use the # symbol now available for privatizing fields in EcmaScript. Remember that private properties are only private in TypeScript; there are no true private properties in JavaScript classes, so a private property can still be changed if you ignore the error.

`protected` - protected properties can be accessed by the class itself and child classes.

```
class Student {
  protected studentGrade: number;
  private studentId: number;
  public constructor(grade: number, id: number) {
    this.studentGrade = grade;
    this.studentId = id;
  }
  id(){
    return this.studentId;
  }
}

class Graduate extends Student {
```

```
  studentMajor: string; // public by default
  public constructor(grade: number, id: number, major: string ){
      super(grade, id);
      this.studentId = id; // TypeScript Error: Property 'studentId' is private and only accessible within class
      this.studentGrade = grade; // Accessable because parent is protected
      this.studentMajor = major;
  }
}

const myStudent = new Graduate(3, 1234, 'computer science');

console.log(myStudent.id()); //  prints 1234
myStudent.studentId = 1235; // TypeScript Error: Property 'studentId' is private and only accessible within clas
console.log(myStudent.id()); // prints 1235

```

Factory Functions

If Factory Functions remain your preferred way of creating JavaScript objects, they are still useable within TypeScript. To create a factory function with explicit typing, create an interface with the object's properties and methods and use the interface as the return type for the function.

```
interface Student {
  name: string;
  age: number
  greet(): void;
}

const studentFactory = (name: string, age: number): Student =>{
  const greet = ():void => console.log('hello');
  return { name, age, greet };
}

const myStudent = studentFactory('Hana', 16);


```

Further Reading

More information on working with classes and objects in TypeScript from *SitePen*. Advanced TypeScript 4.0 Concepts: Classes and Types.

New Terms

Term	Definition
Access Modifier	Used in classes to declare how a property or method can be accessed from the application
Duck typing	A programming paradigm where if two or more structures (functions, interfaces, objects) have the same properties, they can be used interchangeably regardless of any type declarations
Enumerated type	A set of constants that are automatically indexed and can be called by their name or index

Interface	Used as a blueprint to declare the shape of something reuseable such as functions, objects, and classes
Tuple	A data type of an array with a set number of values where all value types are known

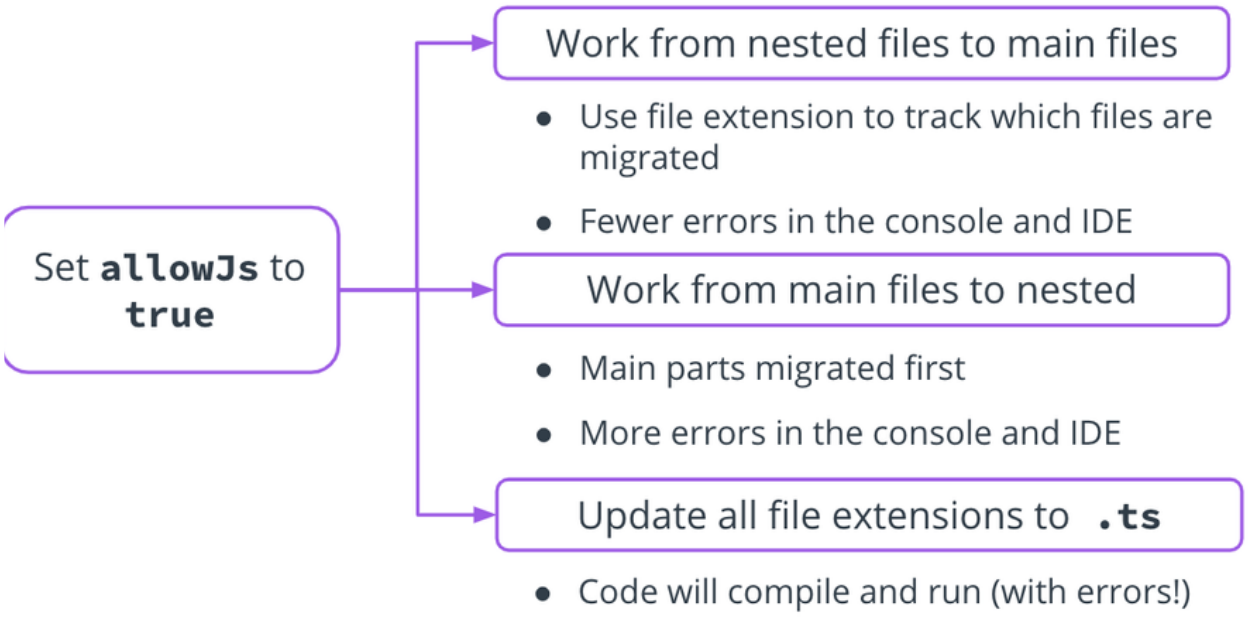
Migrating to TypeScript

Migration Strategies

- Look at the project structure
- Decide whether to migrate all at once or file-by-file.
- Add Typescript to *each* service if project uses microservice architecture.
- For monolithic architecture, move to a `src/dist` to keep working files separate from complied Javascript.
  - Check if this affects any of the other paths within the project, as they might not be automatically updated (although most IDEs do).
  - If it doesn't automatically update, you can use a `path` module.
- To exclude folders you don't want to be migrated, utilize the configuration file.

Third-party Module Type Definitions

- To find the definitions, search through `dependencies` and `dev-dependencies` going through each dependency and adding definitions for each. If a dependency doesn't have definitions, you can create your own.



Typescript Migration Strategies

By setting `allowJS` to `true` in the `config` file, you. can follow the following approaches:

- Work for nested files to main files
  - Use file extension to track which files are migrated
  - Fewer errors in the console and IDE
- Work from main files to nested:
  - Main parts migrated first
  - More errors in console and IDE
- Update all files to `.ts`
  - Code will compile, but run with errors.

Why Use Jasmine?

Testing is important !!!

When developing, our goal is to develop in a way that if someone new reads our code the code is clear, concise, and bug-free. Debugging and refactoring are as important as writing the initial code. It's common for new developers to learn development in a way that you code and use tools like `console.log` as your method for debugging. However, there's a better way.

Introducing testing into your project as a priority and first action allows you to code in a way that writing concise and accurate code that takes into consideration edge cases right from the beginning when the code is easiest to correct. Running our tests is dependent on the developer and can be set to run when the project is saved, compiled, or on-demand. Tests are also incredibly helpful during deployment and can find conflicts between different developers' code, making sure that deployed code is error-free.

The majority of tests written for an application are unit tests. Unit tests test individual pieces of code.

Behavior Driven Development

Jasmine is recognized as a Behavior Driven Development testing framework. This makes sense since Jasmine was originally developed for front-end development testing which focuses on user behavior. With Behavior-driven development, tests are focused on how the user interacts with the application, and stakeholders are included throughout the entire process. However, Jasmine can also be used on the backend where we are less concerned with the user's behavior.

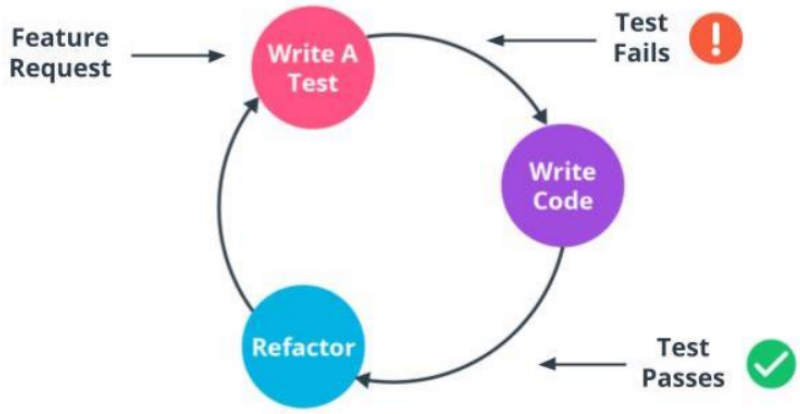
Test-Driven Development

Test-Driven Development is a development style well suited for backend development. It focuses on writing unit and integration tests that produce expected results.

Test-driven development follows a development cycle:

- A feature request comes in
- Tests are written for the most simple functionality of the feature that includes edge cases and failure expectations
- Tests fail due to lack of code
- Code is written to make tests pass
- Code is refactored to be most concise and easy to read

This cycle continues until the feature is complete. The tests remain in the codebase and as the feature is built upon or other features are added, the tests will ensure the feature continues to work as expected and will quickly alert the development team to any potential conflicts or bugs.



Test Driven Development Cycle

Further Reading

- There are a lot of different styles of testing available that are useful across a team. Agile Alliance (an organization focused on agile project management) has produced a great resource general understanding of Test Driven Development not specific to any language or framework: What is Test Driven Development?
- IBM provides a great deal of information on using Test Driven Development with JavaScript. Check out this article by Grant Steinfeld: 5 Steps of Test Driven Development.
- Learn more about how Jasmine can be used as both a Behavior Driven Development framework as well as a Test-Driven Development framework from Testio: Is Jasmine BDD or TDD? Here's What You Need to Know.

New Terms

Term	Definition
Behavior Driven Development	A development style built on Test Driven Development where the focus is user interaction and stakeholders.
Test-Driven Development	A development style where tests are written before development

How Experts Approach Unit Testing

Example: Testing a Coffee App

Let's Build a Coffee Ordering App

- Only offers coffee
- Can add cream
- Can add sweetener

What Information Do We Need From the User?

Size

Regular or Decaf

Cream

Sweetener

What Should We Test to Confirm Success?

- Is coffee hot enough to serve?
- Is there enough coffee for that size cup.
- Do we have enough of the requested ingredients to fill the order?

How Can We Confirm Graceful Error Handling?

- "Waiting..." message when the coffee isn't hot?
- Out of stock message delivered before payment?

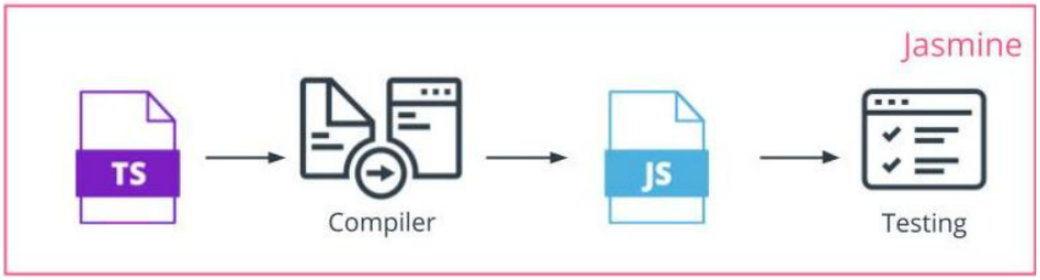
What if We Add New Options?

- Model new tests on existing tests
- Continue running existing tests to ensure that new options don't break the existing system

Expert Approach to Testing

- Start with the simplest tests
- Add tests for more complex situations and edge cases
- Test for graceful error handling
- Add more tests as needed

Configuring Jasmine



Running Jasmine after Code Has Compiled to Javascript

Install Jasmine:

1. To install Jasmine run:

```
npm i jasmine
```

1. Add a reporter for outputting Jasmine results to the terminal:

```
npm i jasmine-spec-reporter
```

1. Add type definitions for Jasmine with :

```
npm i --save-dev @types/jasmine
```

Add Testing Scripts:

- Find the `scripts` object in the `package.json` and add the following to run jasmine:

```
"jasmine": "jasmine"
```

Set Up the File Structure:

1. In the root directory of the project, create a folder named `spec`.
2. In the `spec` folder, create a folder named `support`.
3. In the `support` folder, create a file named `jasmine.json` to hold the primary configurations for Jasmine.
4. In the `src` folder, add a folder named `tests`.
5. In `tests` add a file named `indexSpec.ts` to hold the tests for code in the `index.js` file.
6. In the `tests` folder, add another folder named `helpers`.
7. In `helpers`, add a file named `reporter.ts`. This will be the primary configuration for your spec reporter.

File Structure

```
├── node_modules
├── spec
|      └── support
|           └── jasmine.json
├── src
|     ├──  tests
|     |     ├── helpers
|     |     |      └── reporter.ts
|     |     └── indexSpec.ts
|     └── index.ts
├── package-lock.json
```

```
├── package.json
└── tsconfig.json

```

Best Practices For File Naming

When creating files for tests, a best practice is to name the `.ts` file the same as the `.js` file to be tested with `Spec` appended to the end. The more tests needed to be run, the more test files will need to be created. Be sure to follow this best practice to keep track of the `test` file that contains the tests for each `.js` file.

In `reporter.ts`, add the following code from the jasmine-spec-reporter TypeScript support documentation to configure the reporter to display Jasmine results to your terminal application. These are default settings and can be adjusted to meet your specific needs. The documentation on GitHub provides more options available.

```
import {DisplayProcessor, SpecReporter, StacktraceOption} from "jasmine-spec-reporter";
import SuiteInfo = jasmine.SuiteInfo;

class CustomProcessor extends DisplayProcessor {
    public displayJasmineStarted(info: SuiteInfo, log: string): string {
        return `${log}`;
    }
}

jasmine.getEnv().clearReporters();
jasmine.getEnv().addReporter(new SpecReporter({
    spec: {
        displayStacktrace: StacktraceOption.NONE
    },
    customProcessors: [CustomProcessor],
}));
```

In the `jasmine.json` add the following configurations for a basic Jasmine configuration:

```
{
    "spec_dir": "dist/tests",
    "spec_files": [
        "**/*[sS]pec.js"
    ],
    "helpers": [
        "helpers/**/*.js"
    ],
    "stopSpecOnExpectationFailure": false,
    "random": false
}
```

In the `tsconfig.json` file, add `"spec"` to the list of folders that we want to exclude.

```
  "exclude": ["node_modules", "./dist", "spec"]
```

Write a Basic Test

We'll start with a simple test:

index.ts ```typescript const myFunc = (num: number): number => { return num * num; };

```
export default myFunc;
```

We can write a simple test for the function:

indexSpec.ts ```typescript import myFunc from '../index';

```
it('expect myFunc(5) to equal 25', () => {
  expect(myFunc(5)).toEqual(25);
});
```

To test this we'll need to first run the build script and then the test script:

```
npm run build
npm run jasmine
```

Or we can combine the two into one script in our `package.json` file:

```
"test": "npm run build && npm run jasmine"
```

Troubleshooting

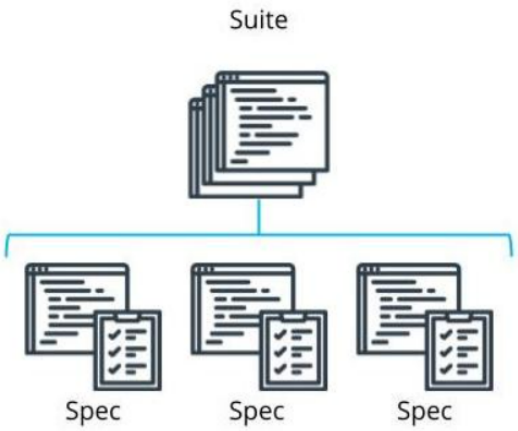
It is not uncommon for conflicts to arise between NPM packages as authors update or add functionality or as the packages that these packages depend on are updated. When you get an error, you can either look for the error to see if it has been reported and follow a solution offered, use older stable versions, or attempt to remedy the issue and submitting a solution.

Writing Unit Tests

Writing Basic Unit Tests with Jasmine

Jasmine uses Suites and Specs

- **Spec**: an individual test
- **Suite**: a collection of similar tests related to one function
- Tests should cover ***all*** intended behaviors.
- Error handling should also be tested



Suites and Specs

Jasmine Syntax

- Use the **describe** keyword followed by a short description of what the suite is testing and one or more specs.
- A best practice is to start a sentence with “it” and then complete the sentence with the description of what the suite is testing.

```
describe(“suite description”, () => {
    it(“describes the spec”, () => {
        const myVar = true;
        expect(myVar).toBe(true);
    });
});
```

Comparisons

- Can compare strings, numbers, objects, or arrays
- `.toEqual(expected value)` checks if the tested value is ***equal to the expected value***
- `.toBe(expected reference)` checks if tested object is ***the same object***

Test Types

Truthiness

- `.toBeTruthy()` passes when
  - The expectation has any non-zero value
  - The expectation evaluates to `true`

- `.toBeFalsy()` passes when the value is:
  - `0`
  - `''` (an empty string)
  - `undefined`
  - `null`
  - `NaN`
- If you just need the Boolean value of `false`, use `.toEqual()`

Numerical Matchers

- `.toBeCloseTo(expected value, precision)`
  - Passes if a value is within a specified precision of the expected value
  - Precision is optional and is represented the number of decimal points to check (defaults to 2)
- `.toBeGreaterThan(expected value)`
- `.toBeLessThan(expected value)`
- `.toBeGreaterThanOrEqual(expected value)`
- `.toBeLessThanOrEqual(expected value)`

Negating Other Matchers

- In JavaScript, use the `!` to negate
- In TypeScript, use `not`
- In Jasmine, use `.not`

Exceptions

- `.toThrow(expected value)`
- `.toThrowError(expected value, expected message)` (expected value and expected message are optional)

Other Matchers

- `.toContain(expected value)`
- `.toMatch(expected value)`
- `.toBeDefined()`
- `.toBeUndefined()`
- `.toBeNull()`
- `.toBeNan()`
- Custom matchers

New Terms

Term	Definition
Comparison Test	A type of test that compares strings, numbers, objects, or arrays
Numerical Matchers Test	A test of numerical values within a specified range of the expected value
Spec	An individual test

Suite	A group of related tests
Truthiness	When a conditional proves to be truth-like such as the boolean true or a condition being `true`, or a value not equal to `0`, `NaN`, `undefined`, `null`, or empty.

Further Reading

Check out Jasmine's full documentation for working with matches.

Testing Asynchronous Code

Testing Asynchronous Code

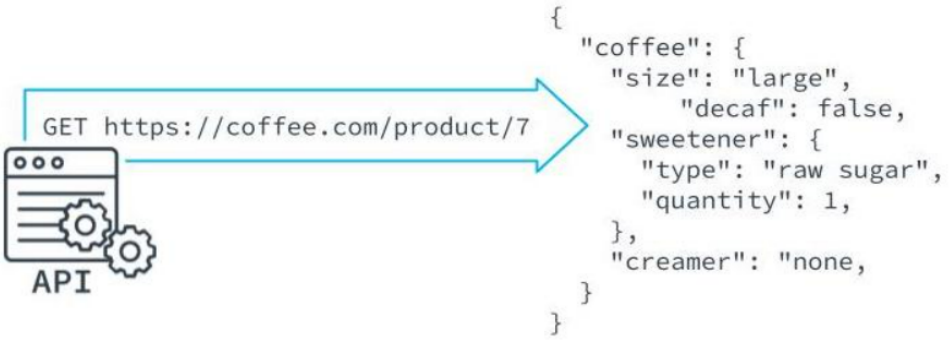
The key to testing async code is letting Jasmine know when it’s ready to be tested.

- Using **async/await** syntax makes testing easier
  - Jasmine syntax mimics JavaScript syntax
  - Add **async** before the asynchronous function call
  - Add **await** before the return
  - Testing occurs after the return
- Using p**romise** syntax with Jasmine
  - **Promise** values are included in the **return** statement
  - Test is run in the **.then()** statement that is chained to the return value
- Testing promise resolution and rejection with ES6 Promise Matchers Library
  - **.toBeResolved()** tests if a promise is resolved and will return true if the promise is resolved
  - **.toBeRejected()** tests if a promise is rejected and will return true if the promise is rejected
  - **.toBeRejectedWith(expected value)** tests if the expected error is returned

Endpoint Testing

Defining an Endpoint

An **endpoint** is the **URL** of the **REST API** with the method that *gets*, *adds to*, or *modifies* the *data* of an API in some way.



An Endpoint is the URL of a REST API

Benefits of Endpoint Testing

- Confirms that the server is working.
- Confirms that endpoints are configured properly.
- More efficient than manual testing.

Adding a Framework for Endpoint Testing

Endpoint testing is not native to Jasmine and requires a third-party framework, like Supertest to test the status of responses from servers.

Setting Up Endpoint Testing

- Install Supertest as a dependency.

```
$ npm i supertest
```

- Add type definition to allow the code to compile without TypeScript errors.

```
$ npm i --save-dev @types/supertest.
```

- Import SuperTest in the spec file.

```
import supertest from 'supertest';
import app from '../index';

const request = supertest(app);
describe('Test endpoint responses', () => {
    it('gets the api endpoint', async (done) => {
        const response = await request.get('/api');
        expect(response.status).toBe(200);
        done();
    }
)});

```

- Create and Run Tests

```
$ npm run test
```

New Terms

Term	Definition
Endpoint	An endpoint is the URL of the REST API with the method that gets, adds to, or modifies the data of an API in some way

Setup and Teardown

Performing Tasks Before and After Tests

Setup and Teardown of Suites

These Jasmine features allow you to

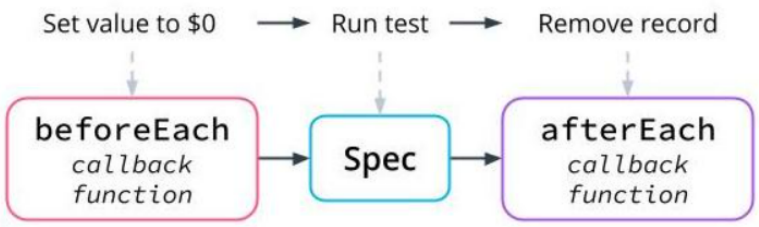
- Connect to a database before a test
- Connect to a different database for specific tests
- Run only a specific test
- Skip one or more tests

`beforeEach` and `afterEach`

- `beforeEach` takes a callback function where we can tell the test to perform a task ***before each test is run.***
- `afterEach` is used if there is a task to be run ***after each test is complete***.

Example: ```javascript describe("", () => { beforeEach(function() { foo = 1; });

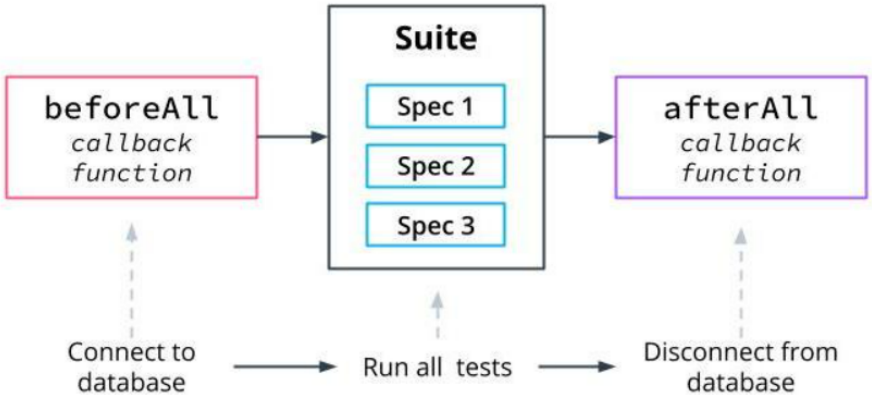
```
  it("", () => {
    expect(foo).toEqual(1);
    foo += 1;
  });

  it("", () => {
    expect(foo).toEqual(2);
  });
});
```



`beforeEach` runs ***before each*** test and `afterEach` runs ***after each*** test

`beforeAll` and `afterAll`

- To perform an operation ***once before*** all the specs in a suite, use `beforeAll`
- To perform an operation ***once after*** all the specs in a suite, use `afterAll`.



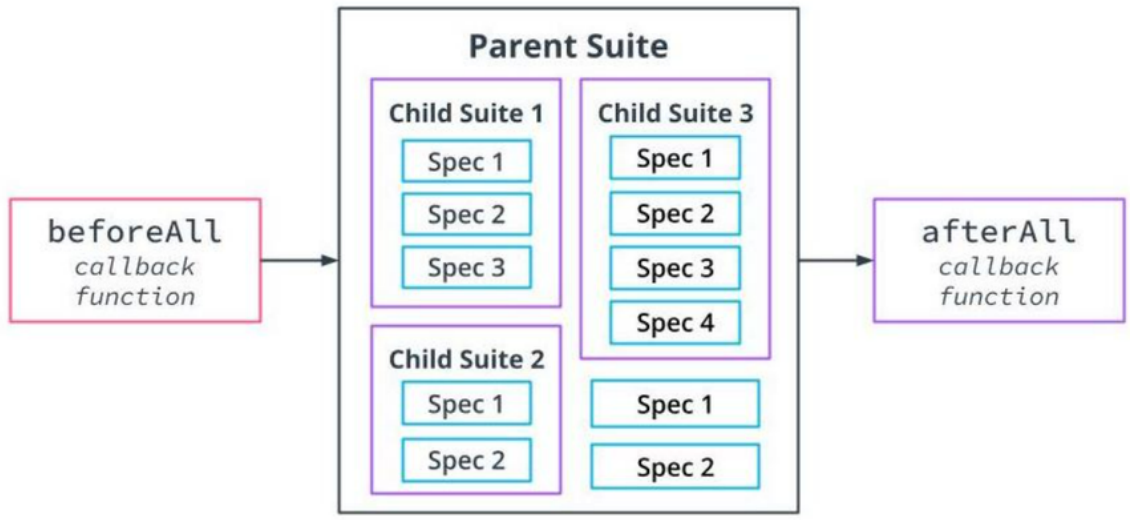
`beforeAll` runs ***before*** the tests and `afterAll` runs ***after*** the tests

Handling More Than One Suite

Jasmine gives us the ability to use set up and teardown for more than just one suite. Whatever action is performed as setup or teardown for the parent suite, all sub-suites will also have access to the repeated or one-time setup or teardown.

Example: ```javascript describe("A spec", function() { beforeEach(function() { foo = 0; });

```
  it("is just a function, so it can contain any code", function() {
    expect(foo).toEqual(1);
  });

  describe("nested inside a second describe", function() {
    var bar;

    it("can reference both scopes as needed", function() {
      expect(foo).toEqual(bar);
    });
  });
});
```



Handling Multiple Suites with `beforeAll` and `afterAll`

Skipping or Specifying Tests

- To skip a test or suite, add `x` in front of `describe` or `it`. This can be helpful to avoid a time-consuming test.
- To focus on one test or suite, add `f` in front of `describe` or `it`. This reduces clutter in the terminal.

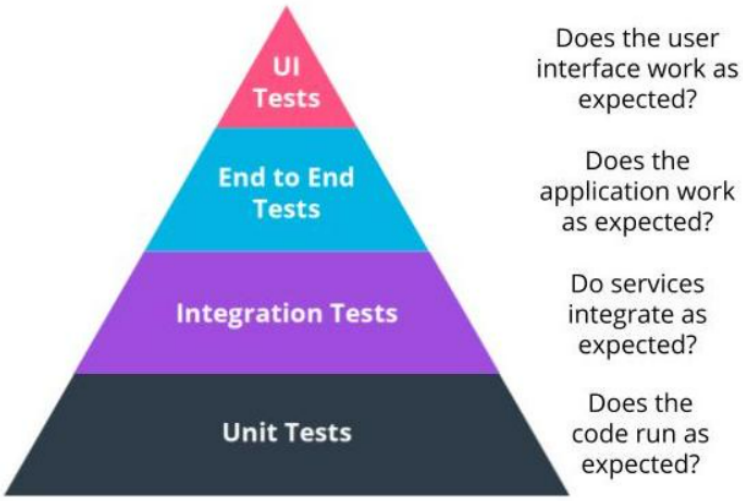
Example: ```javascript xdescribe("A spec", function() { it("is just a function, so it can contain any code", ()=> { expect(foo).toEqual(1); }); });

```
fdescribe("A spec", function() {
    it("is just a function, so it can contain any code", ()=> {
        expect(foo).toEqual(1);
    });
});
```

Beyond Unit Testing

The Testing Pyramid

- **UI Testing:** Does the user interface work as expected?
- **End to End Testing:** Does the application work as expected?
- **Integration Testing:** Do services integrate as expected?
- **Unit Testing:** Does the code run as expected?



The Testing Pyramid

Jasmine and Testing:

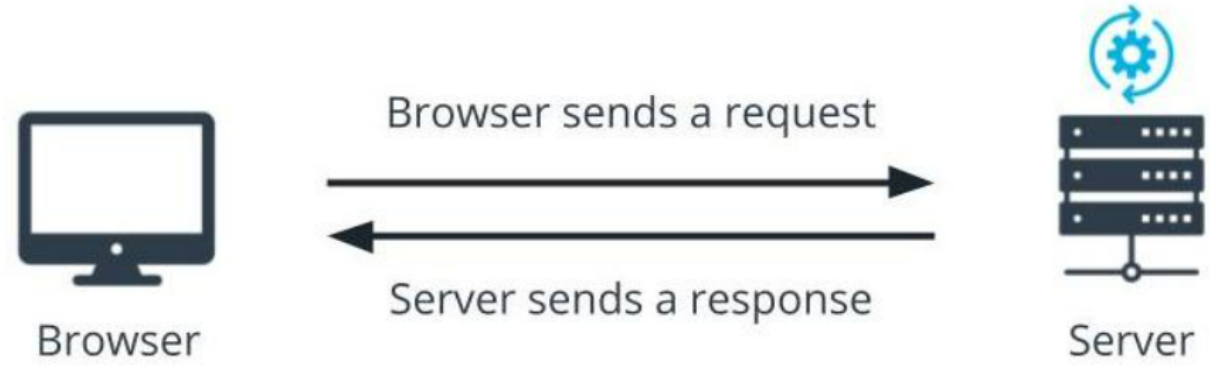
Jasmine works well with **Unit Testing** and **Integration Testing**. Remember, the difference between Unit Testing and Integration Testing is the use of third-party integration. An example would be function that creates an endpoint. This requires a Unit Test. However, if the use case requires testing of the response from the endpoint and requires a third-party tool to do so, this becomes an integration test.

Jasmine can be used for **End-to-End Testing** with a tool call Selenium to emulate user interactions.

For **UI Testing**, Jasmine is simply not helpful.

Why Express?

How Does a Server Work?



How a Server Works

HTTP Requests

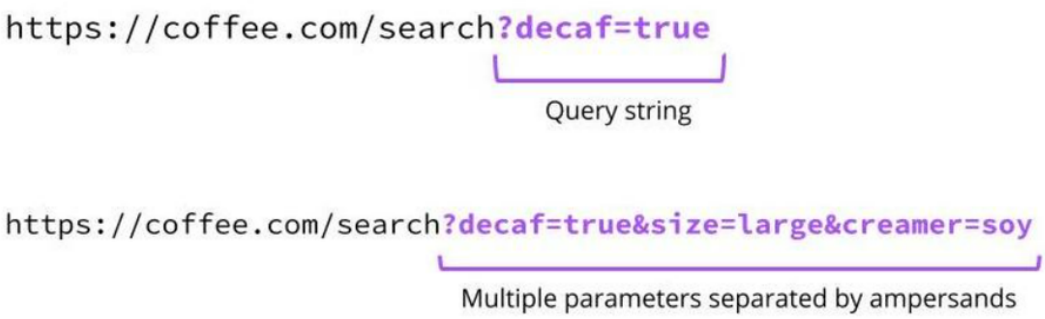
```
POST /drinks
Host: my-coffee-app.com
Accept-Language: en-US,en;q=0.9
Content-type: application/json
Content-length: 20
Request body: {"size": "large", "decaf": false, "sweetener":
{"type": "raw sugar","quantity": 1,},"creamer": "none,}
```

Common HTTP Requests

- **GET** - retrieves data from the server
- **POST** - sends data to the server
- **DELETE** - removes data from the server
- **PUT** - replaces data on the server
- **PATCH** - updates data on the server

Query parameters

- Query strings are parameters in the URL, identified by a '?'
  - Ex: https://coffee.com/search?decaf=true
- To chain multiple parameters together in a query string, use '&"
  - Ex: https://coffee.com/search?decaf=true&size=large&creamer=soy



Query Parameters

HTTP Response Status Codes

Status Code Range	Example Code
100-199: information	100 Continue
200-299: request was successful	200 OK 201 Created
300-399: request was redirected	301 Moved Permanently 307 Temporary Redirect
400-499: client-side error	400 Bad Request 401 Unauthorized 405 Method Not Allowed
500-599: server-side error	500 Internal Server Error

Idempotency

- Definition:
  - “A request is said to be idempotent when making multiple requests to the API that are identical produce the same result.”
- Idempotency and API Methods:
  - The only method not considered idempotent is POST.
  - POST adds a new resource each time; on the other hand, GET, DELETE, PATCH, and PUT act on the same resource each time with the same result.
- Idempotency and Security
  - Get:
    - Safe because the database doesn’t change
    - Endpoint is stored in session history
    - Can be cached
    - Often logged
  - Post:
    - Endpoint **not** stored in session history
    - Protects user data from being inadvertently exposed

Why Use Express?

- **Express is a framework used to**:

    - Set up the server
    - Work with routes
    - Apply middleware
- **Express solves problems, because it**:
    - Builds on HTTP module
    - Handles requests and parses data with minimal configuration
    - Makes it easy to add middleware

An easily recognizable acronym is **MEAN**, which stands for: **MongoDB**, **Express**, **Angular**, **and Node.js**. Express is incredibly popular, and is one of the most downloaded packages from NPM.

How Experts Approach Express

Express Makes It Easy to Structure an App

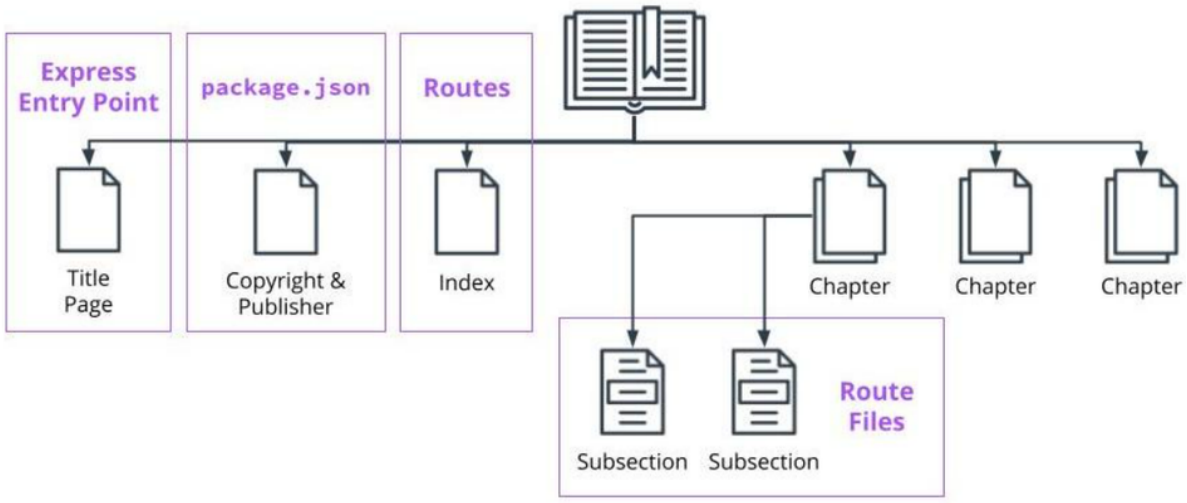
Application architecture is critical for building scalable applications. Fortunately, express offers a fantastic framework for building applications.

For every application, you have your entry point to the application which is the file that creates the server. This is the page that directs you to all of the other pieces of your application.

With your project, you have a file that is critical for the application but is independent of your application, this is your `package.json` file compliments of NPM.

From the server file, you direct your application to a routes folder with the main index directing to all of the paths used in your project. Each path getting its own file and each file filled with endpoints.

An Express Project Is Organized Like a Book



Express is Like A Book

The structure of an Express project is very similar to the structure of a book:

- The server file acts as the entry point or title page to the book
- The package.json file includes the publisher/copyright information
- The main route file acts as an index to the different chapters in your book
- Each route file contains the endpoints for available actions

Using Express

Express Basics

Application

The Application Object

Every Express application requires the creation of what is known as the application object. All of the core functions of express take place on the application object including endpoint methods. After importing express to use in your application, the first thing you do is create your application object most commonly using the name app.

To create the object, you set your application name = to the top-level Express function.

```
const app = express();
```

Core Methods

- `.listen()` - listens for connections to a specified host and port
- `.get()` - used to get a route and takes a route and a callback function as arguments. The callback function takes two arguments, the request from the browser and the response from the server. Additionally, middleware can also be passed in as an argument which will be covered in the middleware section.
- `.post()`, `.put()`, `.delete()` - the other app methods that make up endpoints. They require having the ability to store data. `.post()` is used to post a new item, `.put()` used to edit an item already in existence, and `.delete()` to remove an item from the data. Like get above, all three methods will take a route.

Creating a Server

Request and Response

Request

The request object is the HTTP request to the server. Request has many properties and methods available to it for getting information about the request from the browser.

ip - a property to get the ip of the request.

```
console.log(req.ip);
// prints '127.0.0.1'
```

cookies - a property to access cookie information of a request.

```
console.log(req.cookies.name);
// prints 'cookie name'
```

path - a property to get the path of the URL request.

```
https://website.com/students
console.log(req.path);
```

```
// prints '/students'
```

subdomains - a property to get the subdomain of a URL request.

```
https://students.website.com
console.log(req.subdomains);
// prints 'students'
```

params() - a method to get the param values from a request URL.

Response

The Response object is returned by the server after a request. It is the response from server back to the browser. Like request, it has many properties and methods available to it.

send() - sends a response from the server to the browser.

```
app.get('/students', (req, res) => {
  res.send('Hello, world.');
});
```

status() - set's an HTTP status.

```
app.get('/students', (req, res) => {
  res.status(400).send('bad request');
});
```

cookie() - set's a cookie for the route.

```
app.get('/students', (req, res) => {
  res.cookie('admin', { expires: new Date(Date.now() + 1000000));
});
```

Setting Up a Server (code demo)

Walking through this preconfigured express server we can see the following

package.json

- Express is installed through NPM along with express type definitions
- Nodemon is installed and a "start" script is created to run nodemon on the application entry file

index.js

- Express is imported into the app
- the application object is created and a port is defined
- an endpoint is created that gets the route API and sends a message back to the browser
- the app listens on the defined port and starts a server on localhost then sends a message to the console that the server has started

Middleware

What Is Middleware?

Middleware is a function that is applied between the request and response. Meaning you get the request, do something with it, and then send the response. Common uses of middleware include checking the authentication status of a user before sending a response or logging the request before sending the response. There are many different uses for middleware and multiple types that you can use.

How Middleware Works

Types of Middleware

Built-in Middleware - Express contains 3 builtin middlewares:

- `express.static` - for serving static files
- `.json` - for parsing incoming JSON
- `.urlencoded` - for parsing incoming urlencoded data

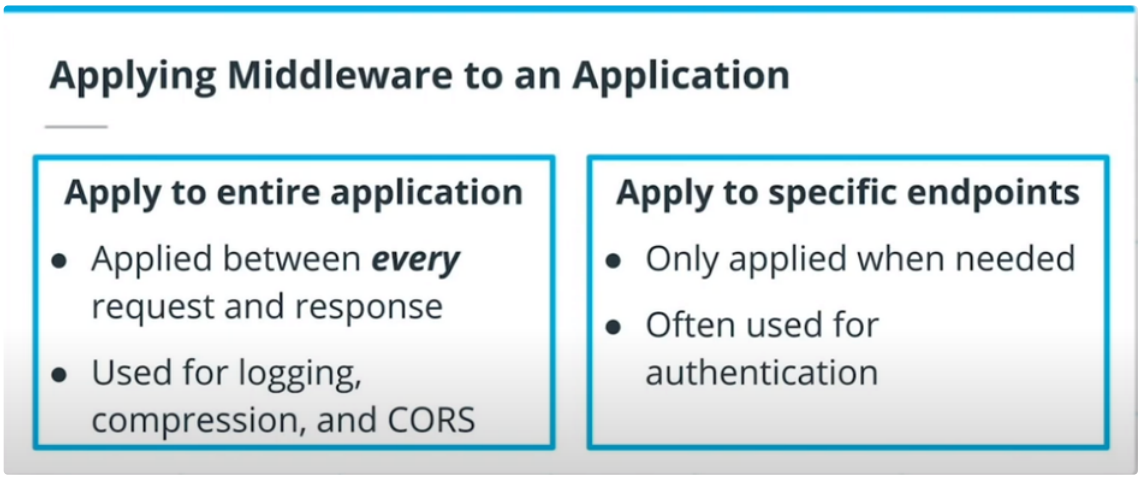
3rd Party Middleware - Middleware that's installed through NPM

Custom Middleware - Middleware that you create specifically for your own project.

Using Middleware

There are two ways of applying middleware:

Application/route level



Applying Middleware to an Application

`.use();`

The `.use();` method is a method that can be applied to the application object or to route objects. It is used for applying middleware and can take in a route, and middleware as arguments

```
app.use(middleware);
```

Endpoint level

Applies middleware to a specific endpoint.

```
students.get('/', middleware, (req, res) => { // do stuff });
```

Applying Multiple Middleware

It's possible to apply multiple middlewares to an application/route or endpoint.

Using an Array

Create an array of the names of the middleware and apply that on `app/route` or endpoint level

```
const middleware = [cors, logger];

app.use(middleware); // app level
students.get('/', middleware, (req, res) => { // do stuff }); // endpoint level
```

Listing Middleware

List out the middleware

```
app.use(cors(), logger); // app level
students.get('/', cors(), logger, (req, res) => { // do stuff }); // endpoint level
```

Writing Middleware

Middleware is really just a function that is applied between the request and response. As such, if writing middleware, you create a function.

A middleware function takes at least 3 arguments (req, res, next); a 4th is also available of err (err, req, res, next) for use in writing error-handling middleware. Then you write the necessary code to complete your functionality followed by calling the next(); method.

`next();`

The next method is a method from the express router. next() calls the next middleware in a chain of middlewares. Without adding next to your middleware function, your application will get stuck on the middleware.

```
const myMiddleware = (req, res, next) => {
  // do stuff
  next();
};
```

New Terms

Term	Definition
Middleware	Functionality that runs between a request to the server and the response from the server

Further Reading

- Read the official documentation from Express on working with middleware.
- Here is a great blog post from Ashutosh Singh from Log Rockett on working with middleware: Express Middleware: A Complete Guide.

Working with Routes

Router Object

When building an express application, it's best practice to keep the server and application endpoints and functionality separate. With the router object, you're able to create a directory of routes and separate the functionality of each route onto its own file.

Router();

The router method is applied to the top-level express object. With this method, you are able to create a routes object that you can apply your endpoints to rather than the application object.

```
import express from 'express';
const routes = express.Router();

routes.get('/', (req, res) => { //do something });

export default routes;
```

Using the Router

To use the router you have created, you must first export the router. Then on your main application entry point, you can import your routes module. Then use `app.use()` ; to use your routes module as middleware.

Setting Up a Router

File Structure

- Create a directory for routes under /src
- Create a directory for your individual route files
- Place a main route index file in your routes directory

Main Route Index

- Import express from Express
- create your routes object from the `express.Router();` method
- Create your root endpoint for getting the root path of your app using your routes object
- Export your routes object

Main Application Entrypoint

- Import your routes object
- Use the use method on your application object to apply your router as middleware

```
import routes from './routes/index';

app.use('/', routes);
```

Setting Up a Router Part 2

File Structure

- Create individual files for each different route your application will contain under your individual routes folder `./routes/api`

Individual Route File

- Import express from express
- Create your individual route object with `express.Router();`
- Use the new object for all of your API endpoints pertaining to that route

- Export your route object

Main Route File

- Import your individual routes from their paths
- Use `routes.use();` to apply your individual routes as middleware setting the path to use and then which middleware to use for that path

```
import route1 from './api/route1';
import route1 from './api/route2';

routes.use('/route1', route1);
routes.use('/route2', route2);
```

New Terms

Term	Definition
Router	Middleware that directs your application to different routes

Further Reading

The official guide to using Express Routing.

A tutorial and guide from MDN on working with the express router: Routes and Controllers.

Introduction to Postman

Using Postman for Endpoint Testing

Postman is a free tool that's extremely helpful for building out APIs. It is an incredibly sophisticated application but can be used for very basic functionality getting started. You haven't learned much about creating APIs at this point, but it's helpful to know a tool for working with APIs when you first get started, it makes the process easier being able to have a tool for visualizing the interactions with your server when you don't yet have a frontend.

You do not need Postman to build APIs and there are other tools out there that you may find you prefer, so don't feel obligated to use it, just know it's available for you.

Installation

Postman is available for download from Download Postman | Get Started for Free it is available for Windows, Mac, and Linux machines.

Using Postman

The left menu provides several menu options, the two most important for getting started include Collections and API. Collections are for you to create a collection of endpoints for your application and save them for testing. API allows you to add an API and begin documenting it. The API page will allow you to add collections as well as other tools for your API.

Above the central workspace, there's a `+` sign. Clicking that `+` sign allows you to add and endpoints and save them to your collections. You can send the endpoints that you create to your server for testing.

Upon sending a request to your server, Postman will give you the status information as well as any response from the server, whatever that may be. Each request has several options including working with the headers and body of the request.

Thus far we've learned about `get` requests which require no additional settings. Sending a get request on an external API will give you the API JSON response. Sending a `get` request to your server that has no response set up for the root get endpoint will just return a status of `OK` if it was successful.

Further Reading

A list of alternatives to Postman from PCS*tacks*.

Reading and Writing with File System

Working With Data

File System is one of the core node.js modules. It's a larger module and requires import for use. Working with the file system allows you to access files within your system, and then on the server once the project is launched. It only allows access to the local file system, not users' file systems.

The File System Module also allows us to work with data by creating, delete, reading, and writing to files.

File System is almost entirely asynchronous by default, but there are some methods that are synchronous and should only be used when first opening a file such as wanting to have a file read before the rest of the code runs. Otherwise, the remainder of the file system module is asynchronous. To avoid using callbacks, we can use the **File System Promises API** which allows the asynchronous methods to return promises.

```
import {promises as fsPromises} from fs;
// or
import {promises as fs} from fs;
```

Where Do We Get Data?

With an application, there are several ways of working with data. The most basic is application memory, where you save data to variables. You're also likely familiar with external APIs where you can read external data from someone else's API. If you've worked with frontend frameworks, you're also likely familiar with local storage which is data saved to your browser across multiple sessions and cleared with clear cache.

The more complex way of working with data involves saving data to an external file. With Node.js there are two ways of doing this, through File System, or through a database. We'll be using File System for data storage for the remainder of this lesson.

File System vs Database

Technically, databases are just files. If you've ever tried to look at an SQL file, you may have seen this in action. They can be single files containing gigabytes of text data. So why use a database when it's ultimately a file as well?

With Databases the content is structured, can be relational, and indexed. With File System, you can only control where you write to the file, and where you read from the file, so File System is only good for simple data storage.

File System vs Database

Using File System to Open and Write Files

Writing files is one of the most useful tasks when working with File System. Not only can you write to a file, but you can also create a file. There are two ways of writing files. You can open the file first, then write to it which is useful when you need to perform other operations on the file as well. You can also just write the file without opening it which is useful when writing is the only function you wish to perform, but you're not concerned about overwriting data that is already there.

File System Flags

File System Flags are used for identifying read/write operations available when opening a file.

- `r` - allows for the reading of a file
- `r+` - allows for the reading and writing of a file, will overwrite content in the file
- `w+` - allows for the reading and writing of a file, will create a file if it does not yet exist
- `a` - allows for reading and writing of a file and will append new content to the end of the file, not overwriting current content
- `a+` - allows for reading and writing of a file, will create a file if it does not yet exist, and will append new content to the end of the file, not overwriting current content

Writing to a File

- `.open()` - Used to open a file. Takes a filename and flag as arguments.

```
const writeData = async () => {
  const myFile = await fsPromises.open('myfile.txt', 'a+');
}
```

- `.write()` - Used to write to a file that is already open. Takes data, and options as arguments.

```
const writeData = async () => {
  const myFile = await fsPromises.open('myfile.txt', 'a+');
  await myFile.write('add text');
}
```

- `.writeFile()` - Used to write to a file, overwriting any content that may already exist in the file. Takes a filename, data, and options as arguments.

```
const writeData = async () => {
  const myFile = await fsPromises.writeFile('myfile.txt', 'add text');
}
```

Reading, Moving, Renaming and Deleting Files

- `.read()` - Used to read a file. The file must be opened first. Allows for reading only a portion of a file, but requires the creation of a buffer to do so. Takes a buffer and options as arguments.

```
const readData = async () => {
  const buff = new Buffer.alloc(26);
  const myFile = await fsPromises.open('myfile.txt', a+);
  await myFile.read(buff, 0, 26);
  console.log(myFile);
}
```

- `.readFile()` - Used to read the entire contents of a file. Takes a path and options as arguments. Is the preferred method for reading files when the entire content needs to be read.

```
const readData = async () => {
  const myFile = await fsPromises.readFile('myfile.txt', 'utf-8');
```

```
  console.log(myFile);
}
```

- `.rename()` - Used to rename or move a file. Takes the old file path and new file path as arguments.

```
const moveData = async () => {
  await fsPromises.rename('old-name.txt', 'new-name.txt');
}
```

- `.mkdir()` - Used to make new directories. Takes a directory path as an argument.

```
const makeDir = async () => {
  await fsPromises.mkdir('src');
}
```

- `.unlink()` - Used to remove a file. Takes a file path as an argument.

```
const removeFile = async () => {
  await fsPromises.unlink('myFile.txt');
}
```

- `.rmdir()` - Used to remove an empty directory. Takes a directory path as an argument.

```
const removeFile = async () => {
  await fsPromises.rmdir('src');
}
```

For removing directories that contain files without needing to remove the files first, it's easiest to use a third-party module such as rimraf.

Further Reading

Official documentation from Node.js on the File System module.

A tutorial on using the file system from Digital Ocean: How to Work With Files Using the FS Module in Node.js.

When To Use Express

When Express Isn't The Right Choice

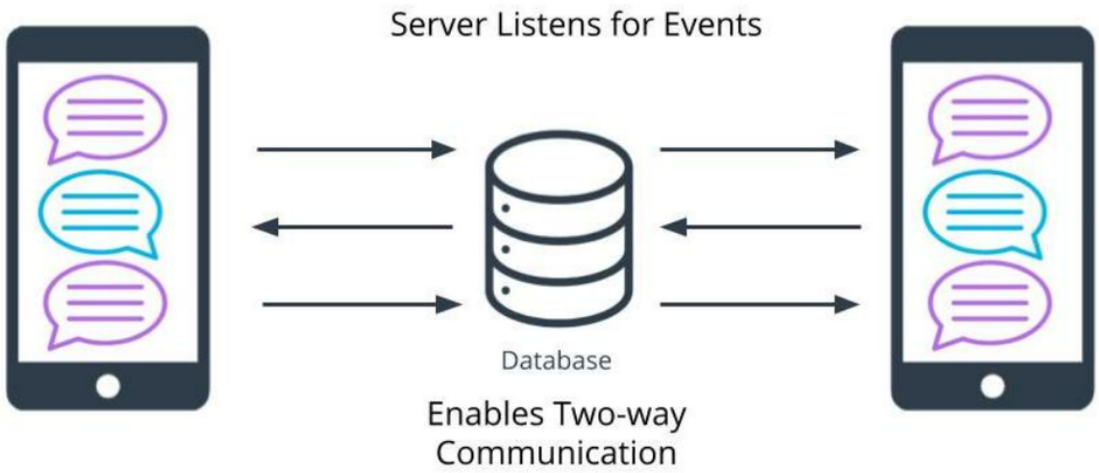
Everything we've done in this lesson so far has been the foundation of what's known as a REST API (Representational State Transfer). It's an architectural standard for APIs and currently the most popular type of API in use. RESTful APIs are considered to be stateless meaning the user/browser is independent of the server and they don't care what the other is doing.

But what if you need the interactions between the server and the user to be stateful meaning the server is aware of what the user is doing. Think about when you are waiting for someone to send a text message and you see the dots pop up letting you know they are working on a response--that's stateful. So what do we do when we need a real-time application? Instead of a REST API, you create a **WebSocket API.**

WebSocket APIs

Websocket APIs are stateful and allow for real-time communication between the user and the server allowing for one user to know what another user is doing. Websockets do not use the HTTP protocol and have their own WebSocket protocol.

The most popular library for working with WebSockets in Node.js is socket io.



WebSocket API

A note on GraphQL

GraphQL isn't specifically a type of API, although it is generally referred to as one. GraphQL is a query language for working with APIs. It's becoming increasingly popular and does work well with Express. Once you're familiar with REST APIs it's very easy to learn GraphQL.

New Terms

Term	Definition
GraphQL	A query language used for working with APIs
REST	Representational State Transfer, a method for working with data.
RESTful	Term to describe an API that is implemented following REST principles. Often used interchangeably with REST (e.g. REST/RESTful APIs).

Websocket API	A type of stateful API that allows the server to know what the user is doing and vice versa

Further Reading

MDN Resource on the WebSocket API.

An in-depth article from Red Hat on graphql: What is GraphQL?

(Doc) Week 9:

Creating an API with PostgreSQL and Express

Course Outline

Welcome to Building an API with Postgres and Express! I can't wait to walk through this course with you as we explore how to use databases and make them accessible to frontend applications.

Course Goals

By the end of the course, you will be able to:

- Create a RESTful Node API with Express
- Write SQL queries for Postgres to power an API
- Design a clean Node API with well organized logic, full test coverage, and database management through migrations

Course Topics

Here are some of the concepts to look forward to in this course:

- Different kinds of databases and what they're for
- Relational Database structure, queries, and common commands
- Relational Database joins
- Database migrations
- CRUD for Node models
- REST structure for Express Endpoints
- CORS enabled API
- API Testing
- Password hashing for security
- JWT's for authentication and protected endpoints

Course Outline

Welcome to Building API's with Postgres and Express! Here's a look ahead at the course.

Databases and SQL

Gain a general understanding of databases, their purprose and use cases for different types and structures. This course uses Postgres, a relational database, so we will spend the majority of the lesson learning more about Postgres and experimenting with some of the most common and useful SQL commands and concepts.

API with Postgres

Learn all the ins and outs of creating a Node API backed by a Postgres database. This lesson includes concepts and practices you are likely to use every day in a web development or software engineering job, such as database migrations, environment variables, and integration testing,

REST API with Express

To complete the API, we turn our focus to the client-facing Express logic. We will cover what it means to create a RESTful API and you will learn good practices for clean code architecture in a Node application. We also touch on important concepts like CORS and Express middleware.

Authentication and Security in a Node API

Password hashing, salt and pepper, JWTs, and more - this lesson is all about data security and authentication. You will be introduced to the best industry standard libraries for these security measures and pick up some good habits along the way.

SQL for advanced API functionality

In the last lesson we pick up right where we left off with SQL and learn queries and database concepts to extend our API in powerful ways. We cover joins and database relationships, and use them to create new concepts in our API like cart functionality and how to create dashboard endpoints filled with useful information.

Local Environment Setup Docker

How to install Docker Desktop?

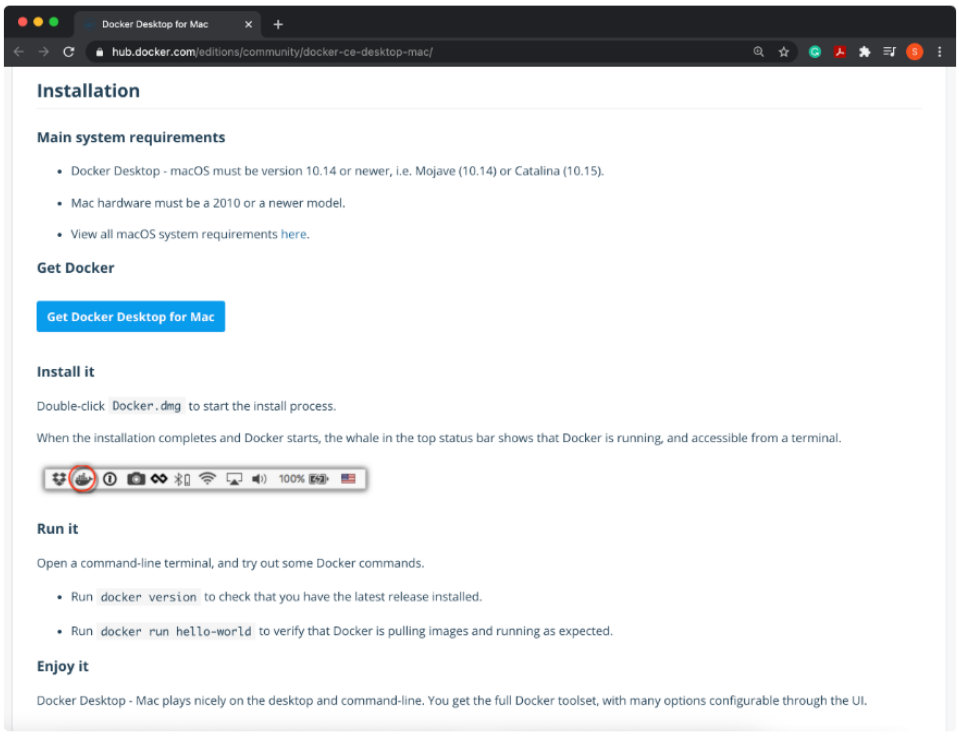
Installing Docker means installing Docker Desktop, a command-line utility. There are installers available for all the major operating systems: Linux, OSX, and Windows. You can find installers at either of the below links which are part of the official Docker documentation:

- Get Docker
- Docker Desktop overview

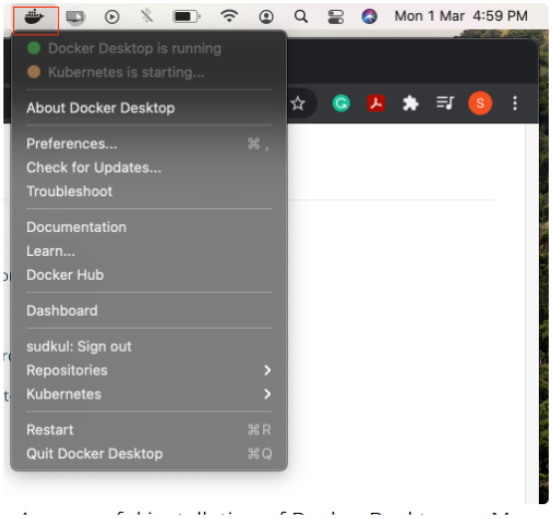
You should install Docker now so you will be ready to continue with the rest of this lesson.

1. Docker for Mac

Upon clicking on either of the links above, you will be redirected to Docker Desktop for Mac page. The requirement is that your macOS must be version 10.14 or newer. This page also has the installation instructions available for you.



Installation instruction on the Docker Desktop for Mac page



A successful installation of Docker Desktop on Mac

2. Docker for Windows 10 Professional or Enterprise (64-bit)

Upon clicking on either of the links above, you will be taken to the Install Docker Desktop on Windows page. Be sure to fulfill the System Requirements mentioned there. Next, you will be redirected to Docker Desktop for Windows page to download the *.exe* installer.

Run the *.exe* file as an Administrator, and follow the prompt. **Importantly, ensure to check the "Enable Hyper-V Windows Features" option on the Configuration page.**

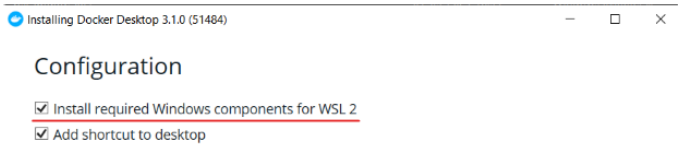
3. Docker for Windows 10 Home (64-bit)

In this case, you will be redirected to Install Docker Desktop on Windows Home page. But, it requires one additional step - you must install and enable the Windows Subsystem for Linux (WSL) 2 before installing the Docker.

After installing WSL2 on your machine, you can follow the regular process of downloading and running the installer from the Docker Desktop for Windows page.

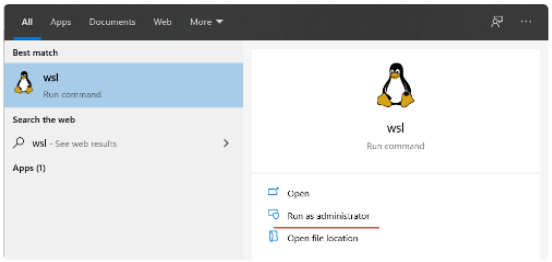
Installation instructions on the Docker Desktop for Windows page

Importantly, ensure to check the "Enable WSL 2 Features" option on the Configuration page.



Configuration page while installing Docker Desktop on Windows

Restart your machine allowing the Docker Desktop to take effect, and launch the WSL from the Start menu.



Launch the WSL from the Start menu

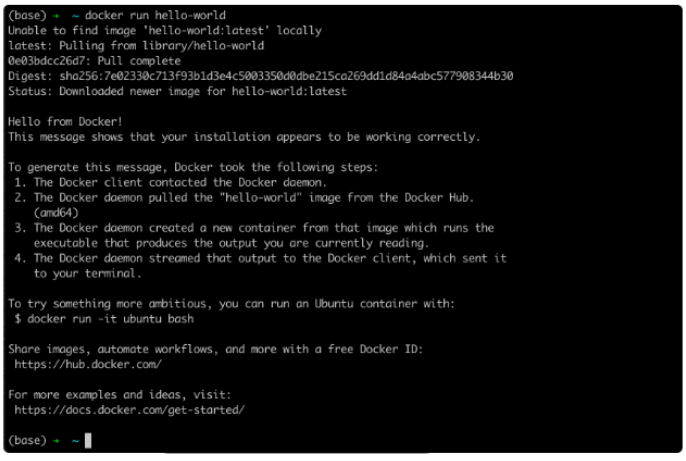
Tip: Windows 10 Home users may refer to this blog that suggests having your code inside a Linux distribution while developing with Docker and WSL 2.

4. Verify the Docker installation

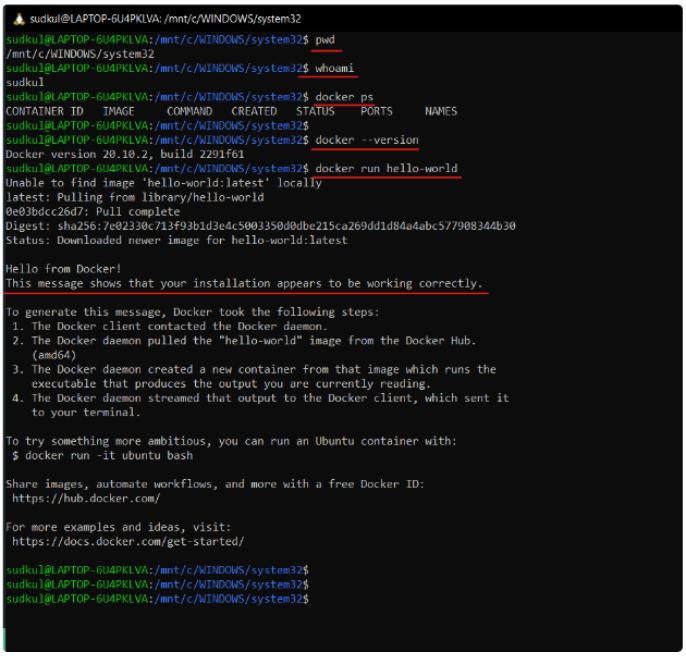
You can run either of the following commands in your Mac terminal / WSL terminal:

```
# to check the version
docker version
# to verify that Docker can pull and run images
docker run hello-world
```

Verify using `docker version` in your Mac terminal



Verify using `docker run hello-world` in your Mac terminal



Verifying Docker installation on Windows 10 Home - WSL2 terminal

Some important docker commands ```bash #List all containers docker ps --all # Stop docker stop <container_ID> # Remove docker container rm <container_ID>

```
1 ```
```

If you want to run one or more containers simultaneously, we use docker-compose.yml.

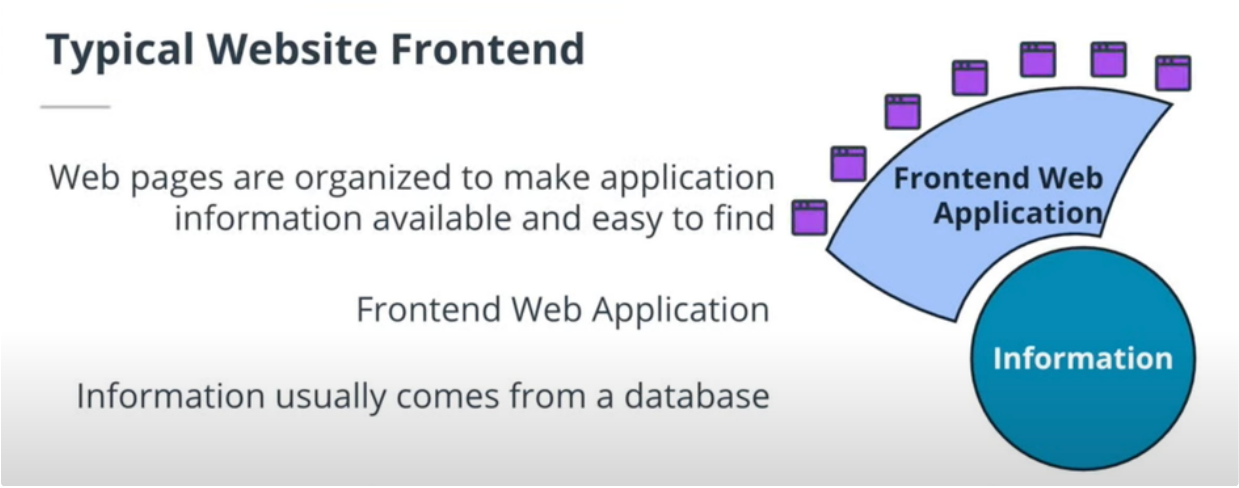
The docker-compose.yml file included in the Github repo should allow you to run a docker container locally if you have Docker installed.

After installing Docker and Docker Compose, run `docker-compose up`

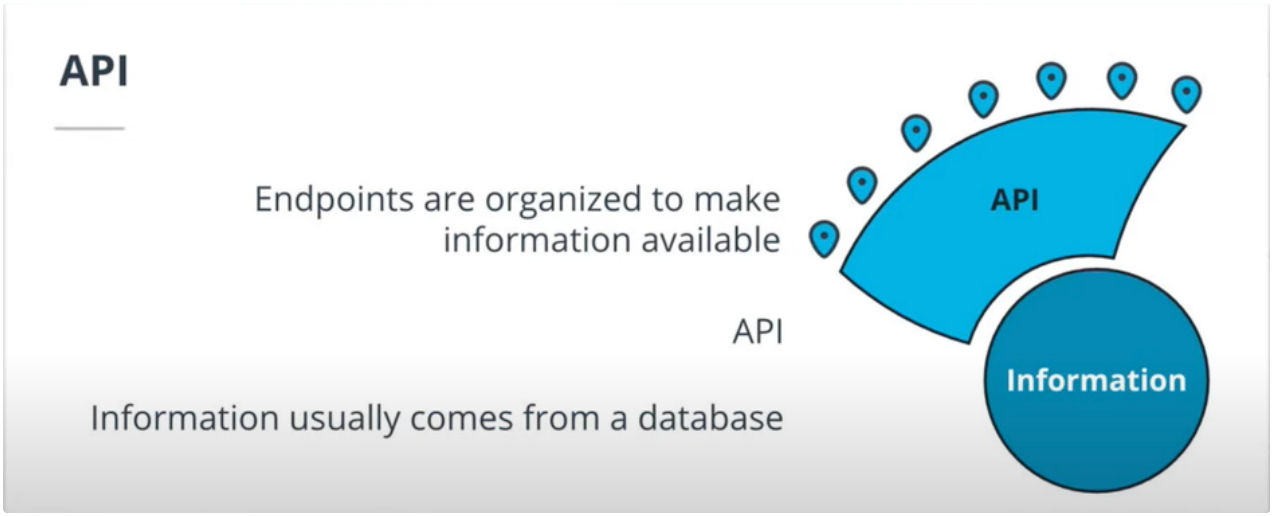
What is an API?

Building an API

It is important that we start on the same page with a solid understanding of what an API really is, because in this course we will go over all the skills and concepts required for you to architect your own API with Postgres and Javascript!



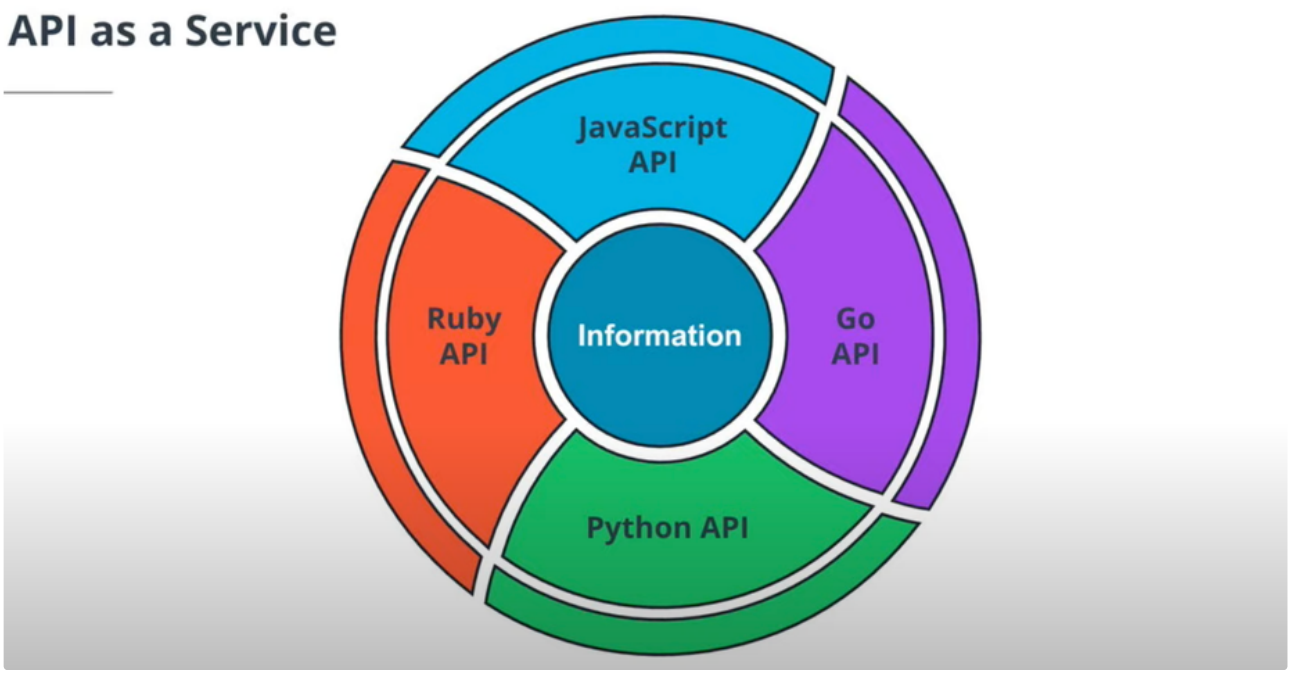
In very general terms, a website gives information to users who are people from a database. Web pages supply information or functionality. But these web pages are for people, so we wrap that information and functionality in HTML and CSS. We give it the color, animations, images, and everything else we need to make that information appealing and easily understandable to humans. The front-end or user-facing website is not itself the data we want to interact with; it is the means or interface we use to interact with the data.



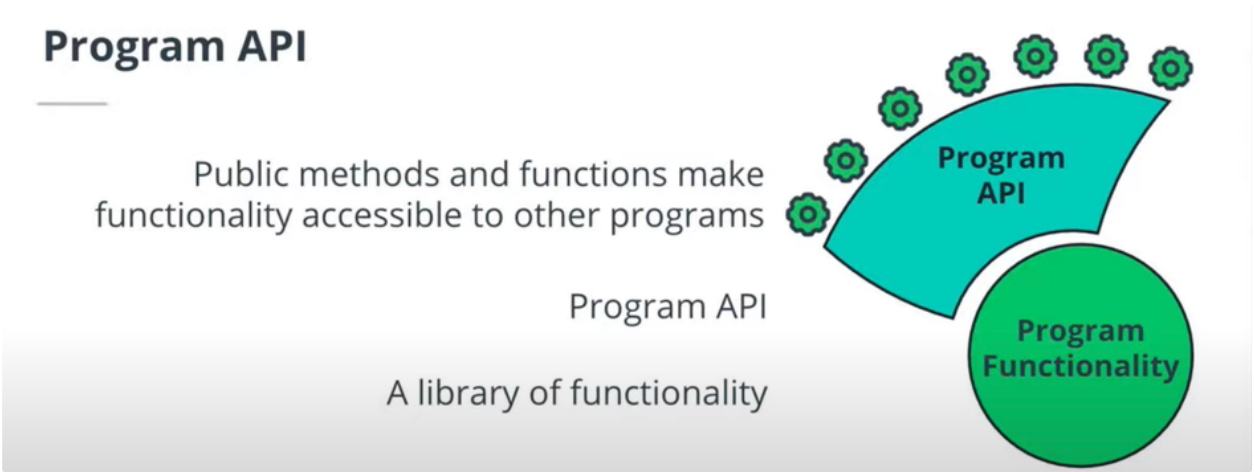
Now let's take a look at what an API does. Essentially, it is exactly the same as a website. It is an interface through which we can interact with information in a database. There's only one real difference, the audience. Where a website is primarily for people, an API is designed to supply other programs with data. This could be like a website, or mobile app, desktop app, etc. Because programs don't care about the aesthetics of a web page, APIs don't need to use HTML, CSS, or anything else design-related. Instead, APIs send back information in a computer-readable way. The API we're going to build will be a JSON API, meaning it supplies information in JSON format. Where a website has webpages that a user can visit, with APIs, we call them endpoints so there's really no difference between the two, except that a web page is designed to look nice and an endpoint is designed to display straight data.

So why is it sometimes confusing to talk about APIs?

Because, though the description above is the most common example of an API, it's not the only way you hear the word used.



Some businesses supply APIs as a service, meaning their data is their product and they make that product available to people via programs they've created. Often, you access these via a developer SDK, an npm library, or a script. These are APIs as a product so they look a bit different but perform the same function to be an interface that allows us to access information.



An API as the interface that makes information or functionality accessible to another program, it begins to be understandable. See, Fetch is a program that makes HTTP requests. The interface that allows a JavaScript program to use that functionality is the Fetch method, so a set of methods I can call in my program. Methods that allow us to use functionality we didn't otherwise have access to can be called an interface or an API.

What is an API anyway?

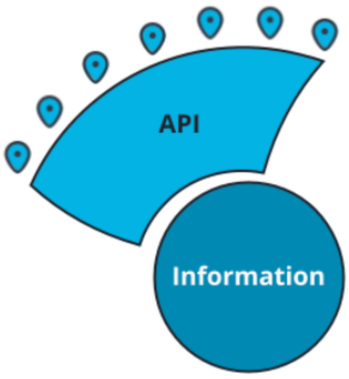
An API is the interface or gateway through which we interact with an external program or data set. Many times, you'll see code bases where the backend of an application is an API for the main database the application utilizes.

API

Endpoints are organized to make information available

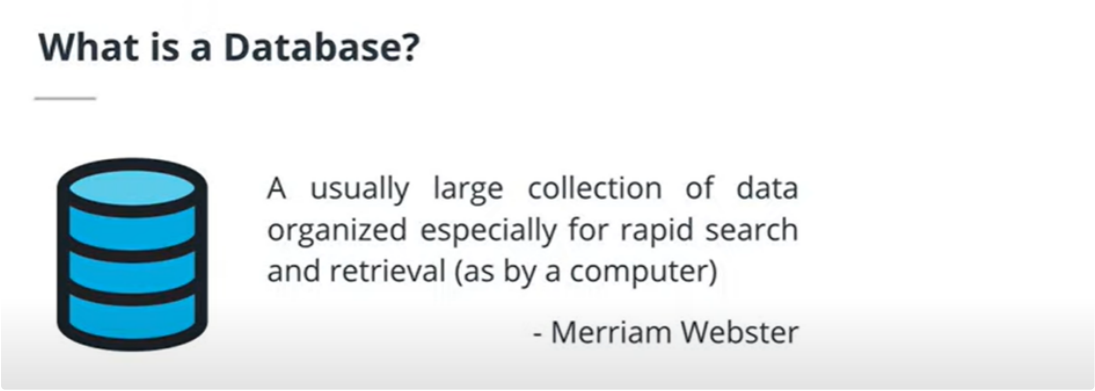
API

Information usually comes from a database



Database Types and Relational Databases

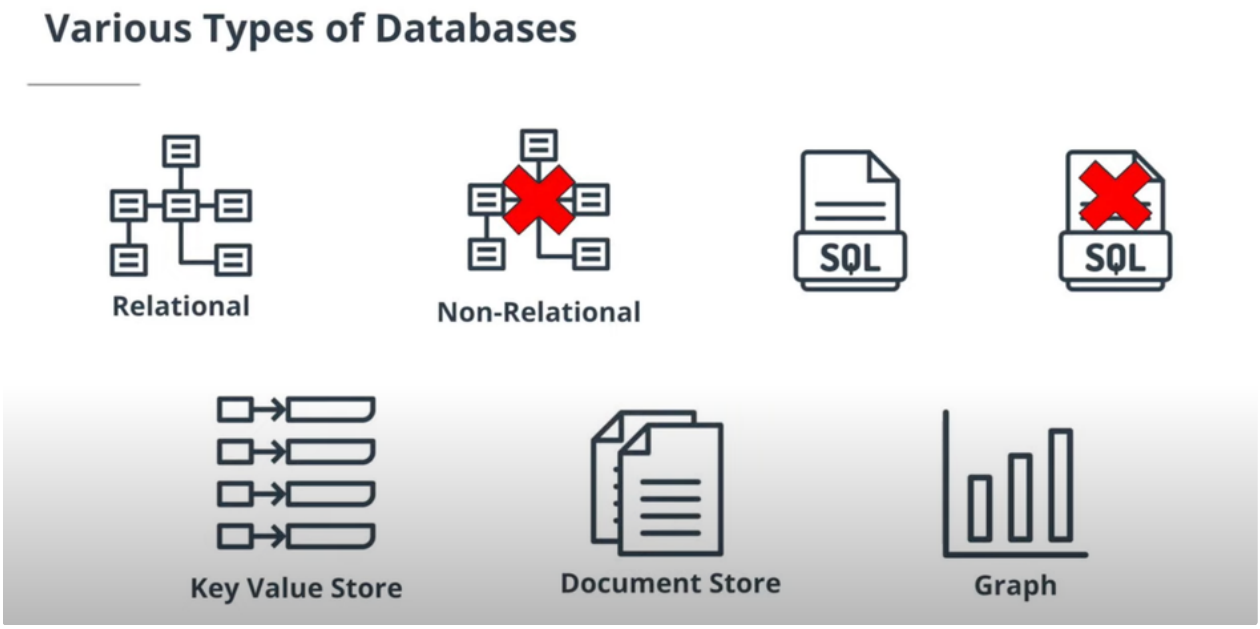
What is a database?



By definition, a database is a collection of data organized in order to be useful for various tasks. By this definition, a digital database that is consumed by an application is only one form of a database.

A library, physical ledger book, or text file of your rock collection could be considered databases because they store information organized in a way that makes data useful. A digital database is no different.

What makes it hard to talk about databases is how many different forms they can take. This is only getting harder because as apps generate and consume more and more data, there are new database types emerging to meet more specific needs.



So now do we need the database to hold billions of entries in a stable way, or do we care most about the speed of saving new information? The security of the information being stored could also be a concern. In this lesson, we will inspect eight of the most common databases to see how they are organized and the intended use case for each database type.

Summary:

- Databases are organized data storage (often, but not exclusively for computers)
- How a database is organized is dependent on how the information it stores will be used
- What makes it hard to talk about databases is how many different forms they can take

Different Databases for Different Tasks

There are tons of different types of databases, each with strengths and weaknesses for different purposes. Choose the right one for your situation, and it will help you move quick and smooth. Choose the wrong one and you might fight against it while building out functionality. For a practical overview, let's take a look at ten of the most common database technologies and divide them up by type. This isn't intended to be an exhaustive list of types or databases, it's just to set the scene for the rest of the database work we will do in this section.

SQL/Relational Type Databases

SQL type databases are organized to be query-able using SQL (Structured Query Language) and organize information in tables. These are pretty much like giant spreadsheets, where an item stored in the database is a row in the table, and columns hold data points on each item.

Ideal Use Cases:

Repeating, structured data, such as:

- user information
- product inventories
- blogs

Common SQL/Relational Database Technologies:

- MySQL
- PostgreSQL
- MariaDB
- Microsoft SQL Server

NoSQL

As you might have guessed, a NoSQL database ...doesn't use SQL. Really this means it isn't set up like a spreadsheet. These databases can take a few different forms and are used for large sets of distributed data (like for use in micro service architectures).

Ideal Use Cases:

Partially structured or un-structured data: really big collections of complex data, caches

Types of NoSQL Databases:

- **Key-Value store**
  - A key-value store is a non-relational, noSQL database type that stores data in key-value pairs (exactly like objects or dictionaries in programming). These databases are fast because the keys are unique and easily searchable, and they are flexible, because these key value pairs can store any combination of data types required.
- **Document store**
  - A document store is a non-relational, noSQL database type that organizes data into documents. Documents can hold any shape of data, which means document stores can easily handle data with no structure or that is arbitrarily nested, which can be a headache to account for in a relational way.
- **Column-oriented**
  - Data organized by column instead of by row. This architecture scales easily and makes fast, efficient queries. I'm including this architecture as a NoSQL type dbms, but this architecture can actually be used with SQL as well.

Common NoSQL Database Technologies:

- Redis [Key Value store]
- MongoDB [Document store]
- Elasticsearch [Document store]
- Apache Cassandra [Column-oriented]

Postgres

Relational databases are where we will spend the rest of the course. We will be creating and connecting to a PostgreSQL (mostly known as just Postgres) database, which is a relational, SQL type database.

Why Postgres?

Here are some reasons why we chose Postgres for this course:

- **Availability** - It is open source and free to use with any project
- **Common** - It is a relational, SQL database which is the most common type of database right now
- **Popular and well tested** - Postgres is a very popular database, and a common choice among enterprise software
- **Transferable skills** - Because Postgres uses SQL, what you learn with Postgres is entirely transferable to working with a MySQL database or any other SQL database

SQL and Creating a Postgres Database

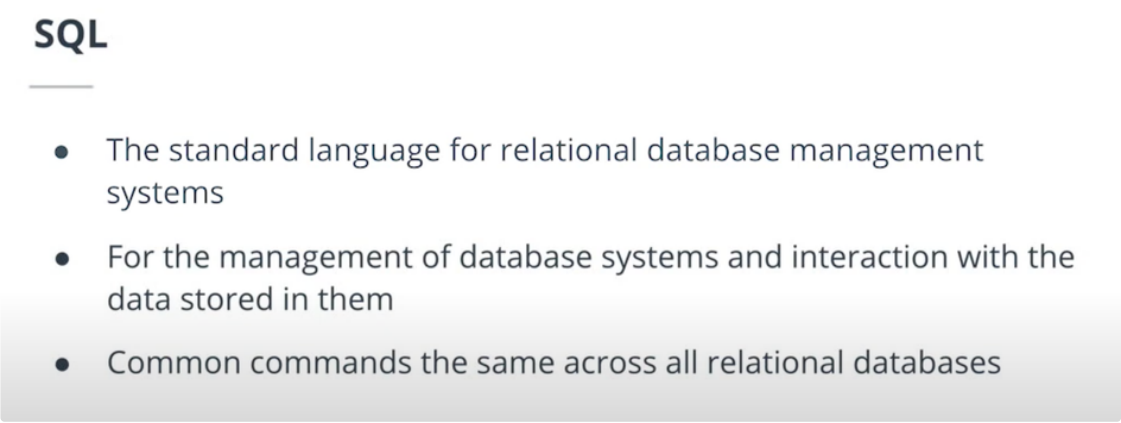
What is SQL?

Because SQL is associated with a type of database, people sometimes mistakenly think SQL itself is a database, but SQL is actually a language or syntax. SQL stands for Structured Query Language, and it is the syntax that allows us to interact with a SQL type database. Like using bash in a terminal to operate a computer instead of a mouse, SQL is the language that allows us to operate on a database without any extra graphical tools.

For example, in a spreadsheet, we use a mouse and graphical user interface (GUI) to click on columns, update values, copy values, add rows, filter views, etc... but a database doesn't always have a GUI so we to use SQL commands in the terminal to perform these actions.

Introduction to SQL

Before we learn to write SQL commands, we need a database to run commands on! We will create and connect to the database via `psql`.

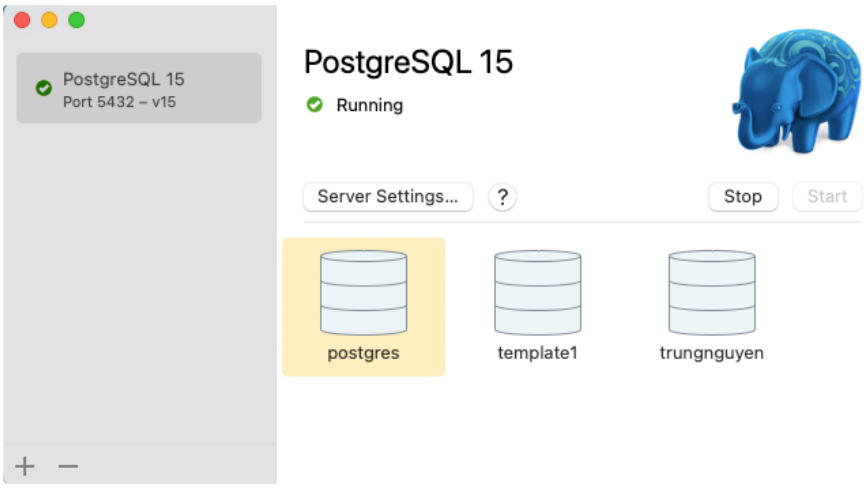


SQL is the language that allows us to interact with information stored in a SQL-type database. Any action we need to take on a database or information stored in it can be achieved with the SQL syntax. One remarkable thing about SQL is that many of the commands will work on any type of relational database. So everything we'll learn in this lesson for Postgres will be the same on a MySQL or other relational database.

Before you run the command below, you need to install Postgres on your machine.

Link download for Windows: https://www.postgresql.org/download/windows/

Link download for Mac: https://postgresapp.com/downloads.html



After installing Postgres on your machine, open the app and choose Postgres database.

```
(base) trungnguyen@Trungs-MBP ~ % /Applications/Postgres.app/Contents/Versions/1
5/bin/psql -p5432 "postgres"
psql (15.2)
Type "help" for help.

postgres=#
```

The terminal will open automatically for you; then you can see the terminal look like the above.

In SQL, we can create a new database like this. In SQL, it is convention to write command words in all caps and user input values in lowercase. But this is only for developer convenience. Postgres doesn't care. User input, like database names or other values, are also written lowercase and with underscores rather than camelCase or with dashes. (Please do not hit “Enter” this command)

```
postgres=# CREATE DATABASE my_new_database
```

Let's make a new database. Database names are usually plural and identify the environment that they are for. Usually production, development, testing, etc. In this case, I'm going to say this database is going to hold plants and it's for my development environment.

```
postgres=# CREATE DATABASE plants_dev
postgres-#
```

When I run this command, you can see nothing happened. Why did nothing happen? Well, one of the things to watch out for in SQL is that you have to end your commands with a semicolon. I didn't end my command with a semicolon, and therefore my line has not ended. Postgres tells me that my line is continuing by changing the beginning of the line from the equal sign to the dash.

You have to pay attention when this is happening because it can cause bigger problems later on. But thankfully, if you catch it now, it's not that bad and it's really easy to fix. What we can do is even though this is a new line, just add a semicolon, hit "Enter", and it will tell us that the database was created and everything is good.

```
postgres=# CREATE DATABASE plants_dev
postgres-# ;
CREATE DATABASE
postgres=#
```

We have a database and the beginning of our terminal line is back to normal. But we can't do anything with our database until we connect to it. We connect to a database with the \C command and then the name of the database. Now we are connected to the plants_dev as our user. This just happens to be my user.

```
postgres=# CREATE DATABASE plants_dev
postgres-# ;
CREATE DATABASE
postgres=# \c plants_dev
You are now connected to database "plants_dev" as user "trungnguyen".
plants_dev=# \
```

Each user will be different for your environment. You'll notice that in PSQL, system-level commands that allow you to move amongst, manage, and view information about databases on the system use the backslash character before them. These commands are known as meta commands and they are the only ones that don't need semi-colons after them.

We have one more really important meta command. A common question is, how do you get out of PSQL? You might feel trapped because the usual tricks won't work. Command C, Escape, Q, etc. So what do you do? The meta command you need will be \Q for quit. This will get you back to the normal terminal.

```
postgres=# CREATE DATABASE plants_dev
postgres-# ;
CREATE DATABASE
postgres=# \c plants_dev
You are now connected to database "plants_dev" as user "trungnguyen".
plants_dev=# \q
(base) trungnguyen@Trungs-MBP ~ %
```

Now we have a database and we can connect to it. If you want to PSQL directly into a certain database, you can actually just type PSQL and then the name of your database as a small shortcut.

```
(base) trungnguyen@Trungs-MBP ~ % psql plants_dev
psql (14.7 (Homebrew), server 15.2)
WARNING: psql major version 14, server major version 15.
         Some psql features might not work.
Type "help" for help.

plants_dev=#
```

Now that we have a database, but it's still empty, we have decisions about how to structure the information we want to store. We will discuss it later.

Summary:

- SQL is the standard language for relational database management systems
- psql in the terminal allows us to execute SQL commands
- Non-meta psql commands must end in semicolons
- executing common psql commands

Common psql commands

- open psql: `psql postgres`
- connect to a database: `\c <database_name>`
- create a new database: `create database <database_name>`
- get out of psql: `\q`

Helpful Resource

A good list of helpful meta-commands can be found from Chartio here.

The Database Schema - Tables and Columns

We have created a database! But... what does that mean exactly? It is now time to discuss the structure of a Postgres database.

In the discussion above, we created a database to do stuff with it. A relational database is like a spreadsheet document. In this discussion, We'll walk through its parts and then go into some of the PSQL commands. A Postgres database is made up of many tables. Tables have the same structure as individual spreadsheets with the common columns and row format.

It's also worth noting that there can be multiple databases in one PSQL system. Would you still be able to remember what we did above? Now we are back at the database from the discussion above, the database is empty now and we want to add some tables or spreadsheets

to it. We can do that using the CREATE TABLE command. The table that we want to create is called plants. (Please do not hit “Enter” this command)

```
plants_dev=# CREATE TABLE plants (column_name column_type)
```

Notice that the name is again plural and it's lowercase because this is a name that we are choosing instead of a command word, like CREATE. Now we want to add columns to this table. I can do that by defining the column name followed by the column type. Spreadsheets don't really talk about a column type. When we create a new column in a spreadsheet, we normally just have to give it a name. But this type thing is required by Postgres to know what kind of thing is going to be stored in that column. We need to explore some of the types that we can use in Postgres. Plants have names. Our table should have a column for names for plants.

But what type should that be? Let's explore some of the types we can use in Postgres by adding columns of each type. First we add a name column. Name will be the title of the column. But how do we portray that this is a string for Postgres? Well, we do that by specifying VARCHAR. VARCHAR stands for variable character, where character is the computer science term for just a letter, and variable, meaning that we don't know exactly how many characters are going to be held in this column. (Please do not hit “Enter” this command)

```
plants_dev=# CREATE TABLE plants (name VARCHAR)
```

Some people will stop here and just say VARCHAR, in which case, Postgres defaults to understanding that this string, this word, will have a limit of 255 characters. Now that's fine. But there's an easy optimization that we can also do here, where if we know that the string we are storing is less than 255 characters, we can specify our own number.

We have looked at it and not very many plants have names of over 100 characters, so we can, with a good degree of confidence, we can put like this.

```
plants_dev=# CREATE TABLE plants (name VARCHAR(100))
```

Now it's also worth saying that if in doubt choose the higher number, because it's better to error on the side of caution and use a little bit of extra space than it is to run out of memory.

Now, what if we wanted to add another column to this table for description? Well we know that a description is likely to be longer than 255 characters. What can we do with VARCHAR to allow a longer sequence of letters? In fact, We can't, and there's a different Postgres type for this called text. Please note that you shouldn't use text if you can use VARCHAR. (Please do not hit “Enter” this command)

```
plants_dev=# CREATE TABLE plants (name VARCHAR(100), description text)
```

But when needed, this is for long chunks of text. Now let's say this plants table is going to hold nature recordings from hikers and plant enthusiasts, and they want to record how many plants they saw at a particular time. To do that, we would need to keep the number of plants in the viewing. Now since we can be sure that we're talking about a positive whole number of plants, we can safely use type integer.

Now let's say that this plants table is going to record plant sightings in the wild, and we want to record how many individual plants were recorded in that one sighting. To do that, we could add a column, like individuals, and we would want it to store the number of plants seen. Except, how do we store a number in Postgres? Well we can be sure that the number of plants is a positive number and it's a whole number. (Please do not hit “Enter” this command)

```
plants_dev=# CREATE TABLE plants (name VARCHAR(100), description text, individuals integer)
```

We can use a type called integer to record that. There are other types that you can use for numbers in Postgres, but this is the one that we need in this case.

Another thing we might want to add is the date of the sighting. How do we store a date in Postgres? Well, thankfully, Postgres makes it really easy for us and there is actually a type date already made. There's also a date-time which records the timestamp as well as the calendar date. But date will only record the calendar day.

We've covered types VARCHAR, text, integer, and date, and we have the beginnings of a good table. Now we will hit “Enter“ and we can see that our table was created.

```
plants_dev=# CREATE TABLE plants (name VARCHAR(100), description text, individuals integer, sighting_date date);
CREATE TABLE
plants_dev=#
```

Now if we want to list all of the tables on a database, you can use the Meta command dt, which stands for display tables.

```
plants_dev=# \dt
           List of relations
 Schema |  Name  | Type  |    Owner
--------+--------+-------+-------------
 public | plants | table | trungnguyen
(1 row)

plants_dev=#
```

When we do that, we see that we have our plants table and that it is the only one in this plants dev database. Here we can see the list of tables on this database and that the plants table we just created. Now the tables of a database and the columns and column types of all of those tables can collectively be referred to as the schema. The schema just describes the structure of a database.

Now for one last really important thing about creating tables and adding columns. It's important to know that data stored in a Postgres database doesn't have an order, it's not sorted, the data is just essentially in whatever order it was inserted into the table.

If we add a new row for a plant we found, how could we be sure to get back to that exact row later on? We need at least one piece of information, one column that is unique to each row, so that we can tell them apart even if they are otherwise identical. This unique column is important and has a special name, it's called the primary key column.

If we go back to the create table command we had earlier, I'm going to add one new column, and that's going to hold an ID. The ID is going to be a special Postgres type called the SERIAL PRIMARY KEY. It is most common to have an ID column be the serial primary key for a table. The word serial means that these values will be auto-incrementing and the ID of each row will be one greater than the row previous.

```
plants_dev=# DROP TABLE plants;
DROP TABLE
plants_dev=# CREATE TABLE plants (id SERIAL PRIMARY KEY, name VARCHAR(100), description text, individuals integer, sighting_date date);
CREATE TABLE
plants_dev=#
```

Also it means that we cannot leave this value empty, so every row is guaranteed to have an ID. Now we have a table with columns, and the only thing left is to save some data into this database.

Summary

- Structure of a relational database table is like a spreadsheet with columns
- Command to list database tables is '\dt'
- The Primary Key is a piece of information unique to each row, often an ID.
- Command to create a new table:

```
CREATE TABLE [IF NOT EXISTS] <table_name> (
 column1_name column1_datatype,
 column2_name column2_datatype,
 column2_name column2_datatype
);
```

Data in the Database and CRUD Operations

Storing and accessing entries in a database

We've created a database and know how to structure it. But now we need to know about the stuff we're going to store in the database. How do we store information in a database? And once we've stored it, how do we get it back out again?

In the last discussion, we created a plants table that is waiting for us to save values into its empty columns. In this lesson, we're going to do just that and take a deeper look into what things we do with databases. Because it turns out that though database queries can get very complicated, the basic things we do are pretty simple and can be boiled down to an acronym CRUD.

CRUD stands for create, read, update, and delete. It turns out that these four actions are really the only actions we take on information stored in a database. For create, we can add new things in or read means to get things out. Update means to edit existing things and delete, obviously means to get rid of things.

We'll go through these one by one and show the PSQL commands that carry each of them out. We're back with our plants table and we want to do a create action. Like it sounds this is storing new values into the database. A create action happens with the INSERT INTO command. When we say insert into, we first have to specify the table name, or in our case, plants. (Please do not hit “Enter” this command)

```
plants_dev=# CREATE TABLE plants (id SERIAL PRIMARY KEY, name VARCHAR(100), description text, individuals integer, sighting_date date);
CREATE TABLE
plants_dev=# INSERT INTO plants (columns) VALUES ()
```

This is the destination for the information that we're going to save. The next thing we have to do is specify what columns we're going to fill, followed by the values that we want to put into those columns. Now the only thing that matters is that the order of columns matches the order of values so that they get mapped correctly.

To add a new plant, all we would have to say is name, description, individuals, and sighting date. We don't have to specify the ID because the ID was serial and will be automatically incremented and created for us.

```
plants_dev=# INSERT INTO plants (name, description, individuals, sighting_date) VALUES ()
```

To add the values into each of these columns. We can say that the name is going to be “Dandelion”. The description is going to just be a short description. Let's say that we saw five individuals and that the date, which will be in string format, was the 1st of January 2023.

```
plants_dev=# INSERT INTO plants (name, description, individuals, sighting_date) VALUES ('Dandelion','Fuzzy yellow flowers', 5, '2023-01-01');
```

Now, when we run this, we will see that there was one row inserted into the plants table.

```
plants_dev=# INSERT INTO plants (name, des
INSERT 0 1
plants_dev=#
```

We've now added a row into the table.

Now we can move on to the second CRUD action, which is read. Reading is accessing information stored in the database. To read, we use the Postgres command SELECT, and we can select all columns from the plants table. When we do that, we get back the new entry that we just created.

```
plants_dev=# SELECT * FROM plants;
 id |   name    |     description      | individuals | sighting_date
----+-----------+----------------------+-------------+---------------
  1 | Dandelion | Fuzzy yellow flowers |           5 | 2023-01-01
(1 row)
```

We can put information into the table and we can read that information back out. We can see that all of the values were correctly put into their corresponding columns.

Now we can move on to the third CRUD action, which is update. That would be editing this row that already exists in the database. The update command is UPDATE in SQL. When we use the update command, we first specify the table where the row is located, and then we use the SET command to define the new value. Let's say that we want to update the number of individuals recorded in this sighting from 5-8. To do that, we would set the individuals column, which is the column we want to change to equal eight. But how do we know which row to update? Well, in this case, we know that the unique identifier for this row is the ID and that the ID of this row that we want to change is one. We can use a new SQL command called WHERE to say id equals 1. This will go to the row where id equals 1 and update the individual's column to now be eight. When we run this, we can see that one row was updated.

```
plants_dev=# UPDATE plants SET individuals = 8 WHERE id=1;
UPDATE 1
```

If we run the select command again, we can see that the individuals for this dandelion citing has been updated to eight.

```
plants_dev=# SELECT * FROM plants;
 id |   name    |     description      | individuals | sighting_date
----+-----------+----------------------+-------------+---------------
  1 | Dandelion | Fuzzy yellow flowers |           8 | 2023-01-01
(1 row)
```

Fantastic. We can add new information in, read information out, and update information in our table. The last CRUD action is the D for delete. Like we probably inferred, delete means to get rid of an item. In Postgres, delete can be accomplished with the DELETE command and we need to specify the table from which we are deleting the row. Now again, we need to be able to specify exactly which row or rows we want to delete. We can use the where word again and specify the ID. This will delete the row where the ID is equal to one, which will only be this single row. Even if we had lots of rows in our database, the fact that the ID is unique and serial would mean that this command will only delete the single specific row where id is one.

```
plants_dev=# DELETE FROM plants WHERE id=1;
DELETE 1
```

If we run this, you can see that one row was successfully deleted. If we try and select things back out from the plants table, you can see that there is nothing here. There are zero rows.

```
plants_dev=# SELECT * FROM plants;
 id | name | description | individuals | sighting_date
----+------+-------------+-------------+---------------
(0 rows)
```

So we've come full circle. We've gone through the full set of CRUD actions. We will build on these concepts throughout the course.

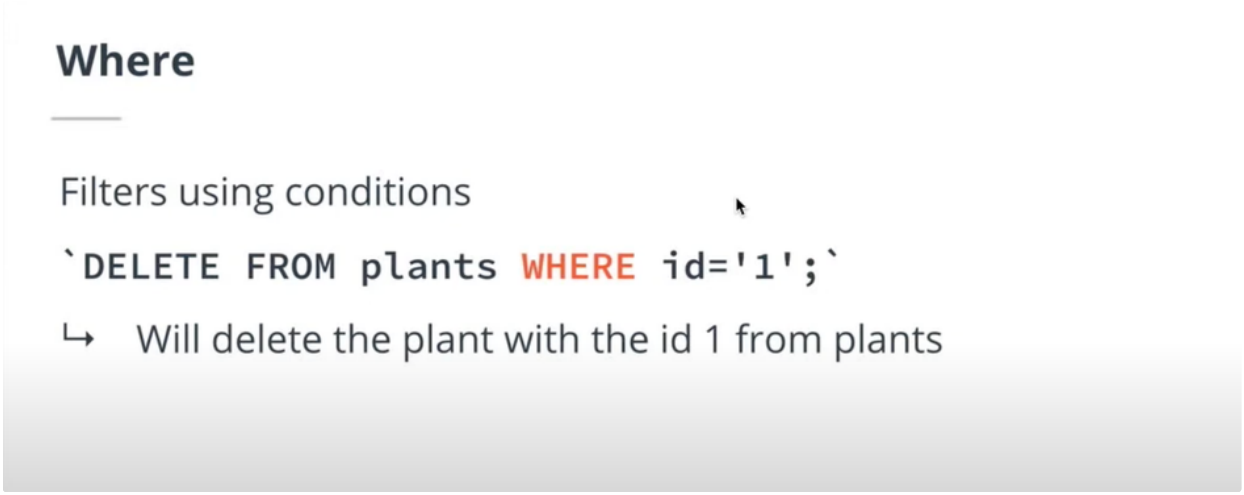
Summary

- SQL commands to interact with rows in a database table
- CRUD stands for Create Read Update Delete and represents the types of actions we often take on information in databases

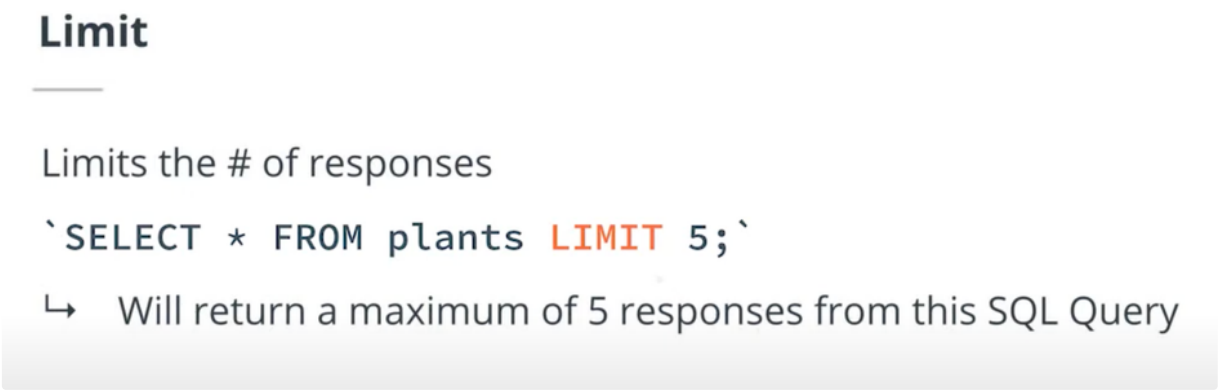
SQL Filters

In the section above we used the word "where" to refine some of our crud queries. Now, "where" is actually a special type of Postgres word known as a filter. In this section, we'll take a short tour through some of the other SQL filter words. We'll start with the "where" filter.

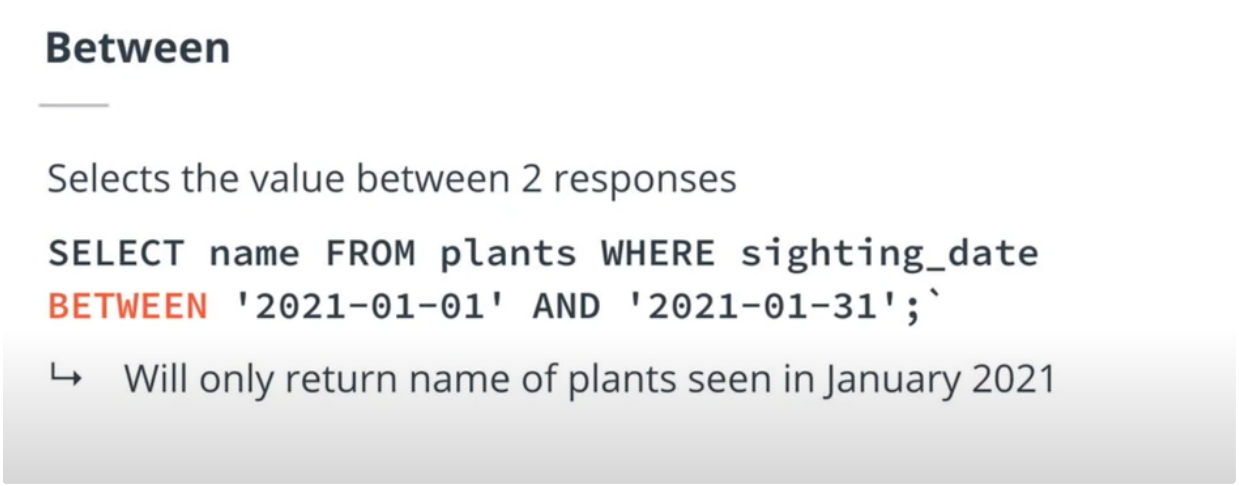
"Where" filters rows based on a specified condition. This filter is awesome because it can be used to get a broad range of information or a very specific set of information, like think if dandelions are a common entry in our plants table. We could grab all of those entries at once simply by looking for where the name is dandelion. Or we can be as specific as grabbing the ID of a single row as we did in the section above. "Where" is a super common and helpful SQL filter word.



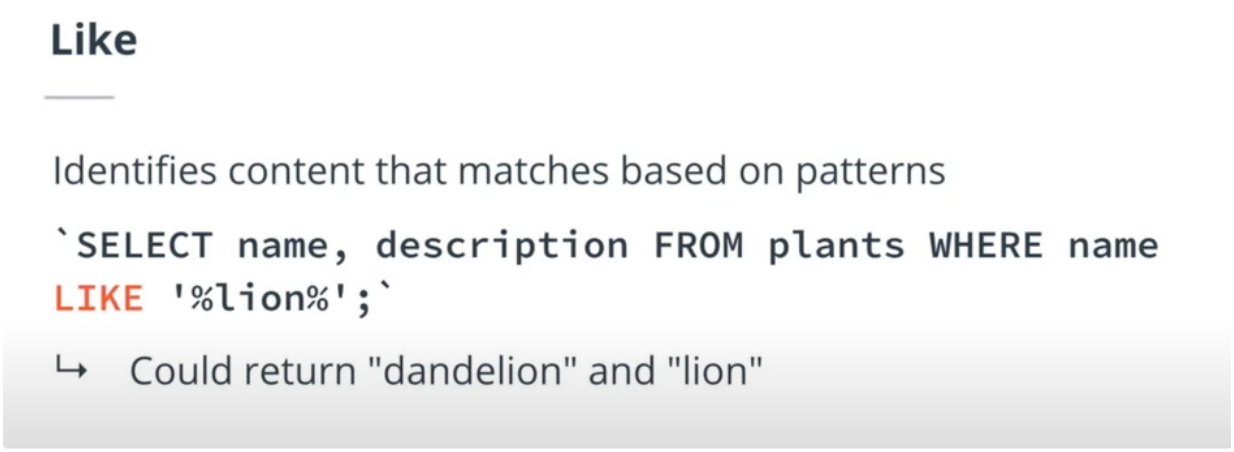
Another is limit. Limits the number of responses from a query. Now, we can run a command like this that would limit the number of responses to five, even if our plants table eventually had hundreds of rows. Now, this won't be super-helpful because again, rows in a table aren't ordered. So limit is usually used with an ordering word, which we'll cover later. But the important thing is that this query will return a maximum of five rows.



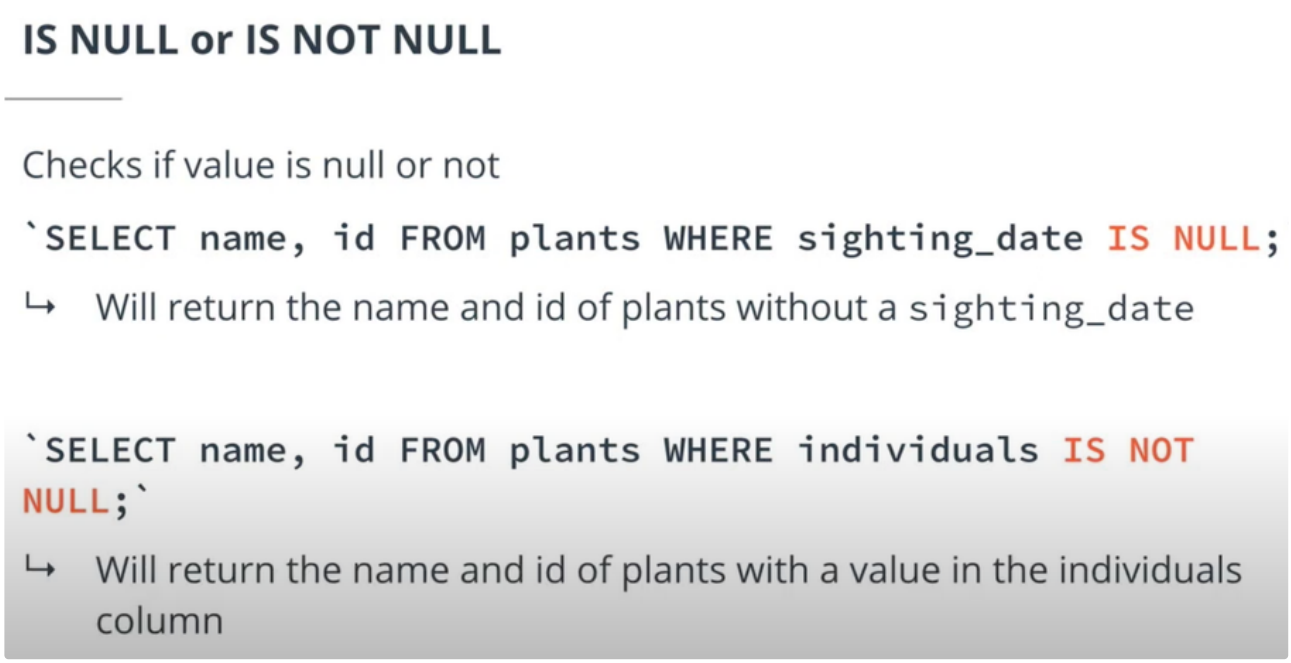
Another SQL filtering word is "between". "Between" selects data that is between two values. In this query, we're going to get resulting rows where the citing date was between the first and last of the month. This is a super common filter to see because it allows us to get just a subsection of data according to our own custom parameters.



Another really common and helpful SQL filter word is "like". "Like" allows us to look for patterns in the content of a cell. Let's take this query for example. We are going to search the plants table for any rows where the name contains the string "lion". Now, this could return dandelion like we input earlier, but it could also return other plants like lion's mane. This filter is especially helpful if we don't know exactly what we are looking for or we want to find all rows with a shared partial piece of information.



For our last filter word, we have "is null" or "is not null". This checks if a value is or is not null. This is really helpful for finding gaps in a database. For example, in this query, we can look for where the citing date is null and we will get a result of all the rows where someone forgot to add the citing date. On the other hand, we can also look for where individuals is not null, and this would get us a list of valid rows where the number of individuals was successfully recorded.



It's important to note that "is" and "is not null" are often used in conjunction with another filter word. So here, we're using the "where" to find a specific column to run the "is" or "is not null" on. There are many more filter words out there for SQL. These are just a few to get we started. As we get more comfortable with them, we can use multiple filters at once to make even more powerful queries.

Summary

- SQL filtering words can be added to refine and hone commands
- Common filter words are WHERE, BETWEEN, LIKE, IF NULL, IF NOT NULL

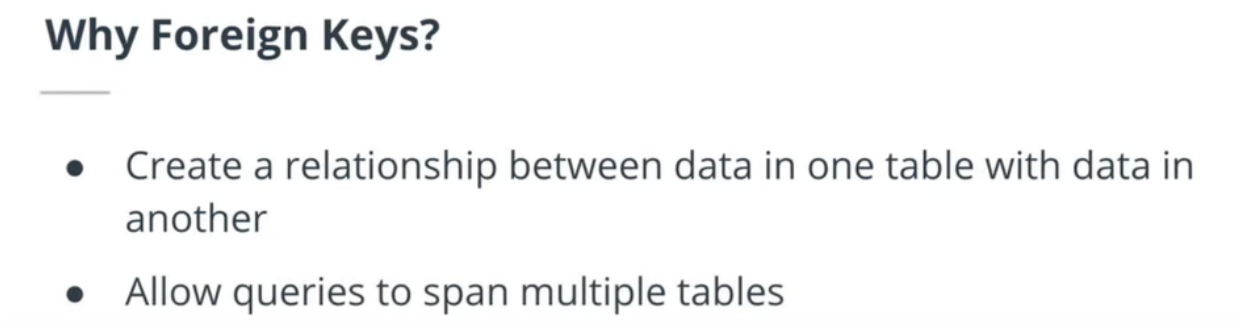
Watch Out! Common SQL Mistakes

There are some really easy to make and easy to miss mistakes that can trip us up in SQL. Here are two of the most common ones - both have gotten me many times.

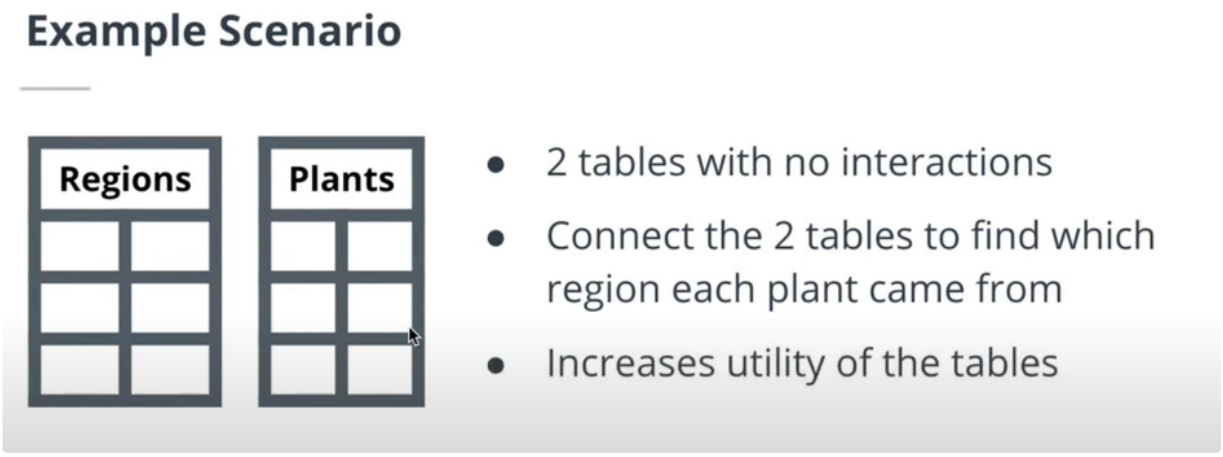
- **Double quotes instead of single quotes**. Double and single quotes are used for different tasks in SQL. For common strings like finding a name in a WHERE statement you must use single quotes. This is really easy to mix up, especially if you are copy/pasting SQL queries, as the formatted special or back-tick quotes also might not work. So if a SQL command isn't running, but you know the syntax is good - double check for single quotes. Here is a good article about the differences in uses of double and single quote in Postgres from Prisma.io.
- **Missing a semicolon**. So easy to miss, the semicolon at the end of a query is one of the most common mistakes to make. Thankfully its easy to fix by just adding a semi colon on the next line - but that fix only works if you notice it soon enough. This is why its a good idea to keep an eye on the beginning of the terminal line, to make sure it ends with this `=#` and not something like this `-#`. Another good practice for catching SQL syntax mistakes early is to pay attention to the result output after a command. If you make a new table you should get a message saying there's a new table, if you added an item you should see an insertion, if you do not get a response after running a command, something is wrong and you should stop and fix it.

Relating Tables with Foreign Keys

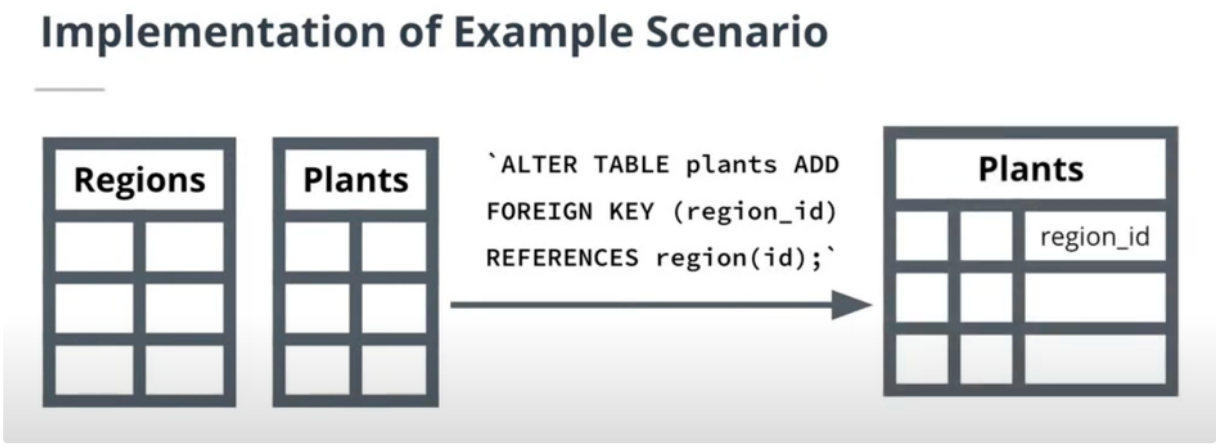
Storing information in tables is great now, but we can only get so far with standalone tables. The power of data comes from being able to create and find relationships in the information we collect. But so far, we have no way to connect information in one table to information in another. In this section, we will cover foreign keys, which allow us to connect information from different tables.



Let's take a moment to think about this conceptually before we jump into the syntax. For example, our database had two tables: regions and plants. But right now, those are pretty boring tables. It's just a list of regions and the list of plants, and they have no interactions with each other or anything else.



But what if we wanted to track which region each plant came from? That would be useful information, right? It would allow us to sort the plants by region and do a lot more. Let's take a look at how we can do this. We will add a column on the plants' table called region_id, that references the id column on the regions table.



Now, let's take a look at this in PSQL. We can see that I've already added the region_id column to the plants' table. Now, each plant has a relationship with a region. We could say that every plant belongs to a region. Pay attention to that belongs to because it's a type of database relationship, and we will discuss those in greater detail later.

```
plants_dev=# ALTER TABLE plants ADD region_id integer;
ALTER TABLE
plants_dev=# SELECT * FROM plants;
 id | name | description | individuals | sighting_date | region_id
----+------+-------------+-------------+---------------+-----------
(0 rows)

plants_dev=#
```

This region id column is the foreign key and its presence opens up a whole new set of things we can do with these tables. Now, we can see what plants are most common across all regions. We can see which regions have the largest and smallest varieties of plants. We could figure out what regions we would need to travel to in order to collect a list of plants.

This is where databases start to get interesting and take on depth and value for data analytics.

Summary

- The Foreign Key is a column that relates each row to in the table to the primary key of another table
- Having a foreign key allows us to query for relationships between two tables of data

Designing a Database

Turning Ideas into Reality

Imagine this scenario:

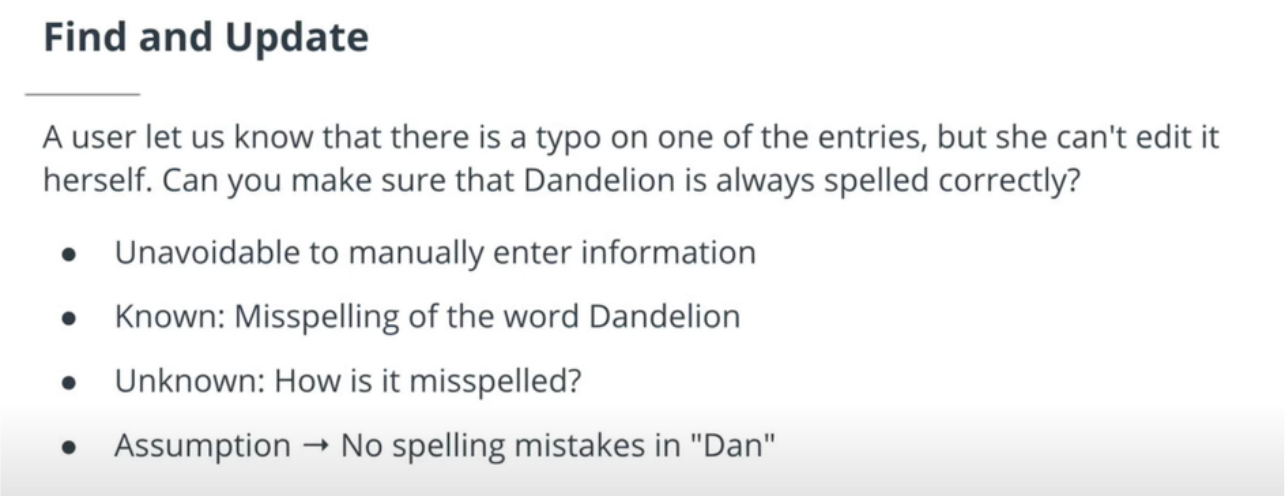
My app is going to help people track their home electrical usage using bluetooth enabled electrical socket extensions, allowing users to view their most power hungry appliances, note which appliances have been running for an extended amount of time, know which outlets they use the most, and track their electrical usage throughout the year! Its going to be awesome! I have a front end in the works with a stunning user interface that everyone is going to love, and I need you to create the API to provide the data.

...Easy right? Or perhaps not. Part of being a software developer (and this really applies for both front and back end developers) is being able to translate abstract ideas for features or apps into concrete data structures. Nowhere is this more true than in database design. It can be hard to take abstract ideas, identify their data needs, and meet those needs with tables and columns. In the section, we take an abstract idea in the form of a task, discuss what data needs are implied, and update the database to complete the task.

Task 1 - Find and Update

A user let us know that there is a typo on one of the entries, but she can't edit it herself. Can you make sure that Dandelion is always spelled correctly?

In this task, a user has alerted us to a typo in one of our entries that she can't fix. So we're going to go into the database and fix it. Making manual updates to the database is something we really try to avoid. But let's say that in this case, there's just no way around it. The issue is that we don't know what the typo is. We only know that it was a spelling mistake in the word "Dandelion". How can we find the row with the misspelled name? We are going to make the assumption that the spelling mistake happens after the initial letters D-A-N.



Making that assumption allows us to write a query like this where we can search the entire name column for any entries that start with D-A-N.

```
plants_dev=# SELECT name, id FROM plants WHERE name LIKE 'Dan%';
```

In Postgres, the percent sign is a wildcard, meaning that any set of characters of any length could come after this N and still be returned from the query. Here, we can see that there are four rows that share the beginning of their name D-A-N. It also allows us to visually pick out the one with the spelling mistake fairly quickly.

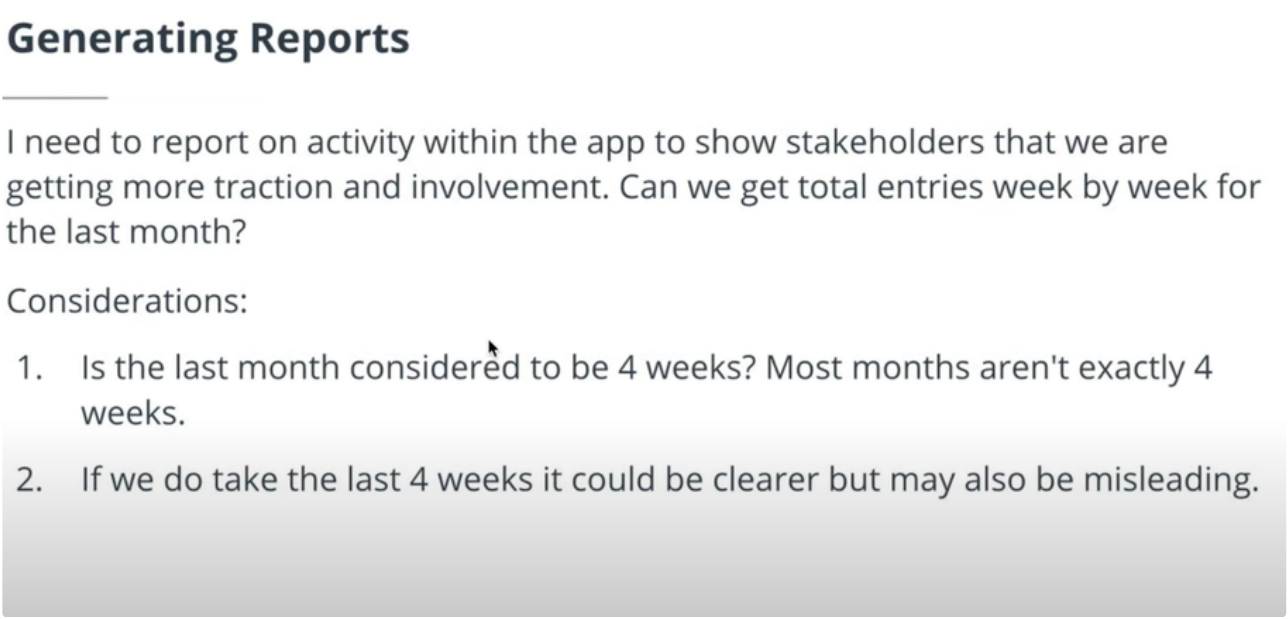
```
plants_dev=# SELECT name, id FROM plants WHERE name LIKE 'Dan%';
    name    | id
------------+----
 Dandelion |  2
 Dandelion |  3
 Dandelion |  4
 Dandalion |  5
(4 rows)
```

Now that we know the ID of the row with the misspelled name, it's a fairly simple fix of updating that row with the correct spelling. we can run this query to update row 5.

```
plants_dev=# UPDATE plants SET name = 'Dandelion' WHERE id='5';
UPDATE 1
plants_dev=# SELECT name, id FROM plants WHERE name LIKE 'Dan%';
    name    | id
------------+----
 Dandelion |  2
 Dandelion |  3
 Dandelion |  4
 Dandelion |  5
(4 rows)
```

Once we do that, we see that one row was updated. If we re-run the initial search for "Dan", we can see that now all of them share the correct spelling. This task is complete.

Task 2 - Generating Reports



In this task, we're being asked to create a report on activity within the app week by week for the past month. This is usually to see positive or negative trends in engagement with an app or website. Now, requests for reports often sound simple, but hide complexity because tiny ambiguities or small differences in interpretation can skew the results. This one is pretty straightforward, but there's still some snacks. First off, are we counting the entire last calendar month? If so, it almost certainly doesn't break into four perfect weeks. On the other hand, we could take the latest four calendar weeks and that might lead to a cleaner query. But it also might be misleading if these results are being titled January progress or some other label.

Now, let's say that we cleared that up with the person asking for the report and we've decided to make a query for the last four calendar weeks, starting January 4th and ending January 31st. We can write a query like this to help us with this task, where we're looking for any sighting date between the 4th and the 10th or the first week of January.

```
plants_dev=# SELECT name, sighting_date FROM plants WHERE sighting_date BETWEEN '2022-01-04' AND '2022-01-10';
    name    | sighting_date
------------+---------------
 Dandelion | 2022-01-10
(1 row)
```

When we run this you can see that we get the sighting date and name and there was one entry in the first week of January.

Now, the report doesn't actually ask for the name or the date of any of the sightings, it only cares about engagement. They really only need the number of sightings in any given week. To make this query more specific we want it to return just the count or just the number of sightings seen in each week. To do that we can run a query like this.

```
plants_dev=# SELECT COUNT(*) FROM plants WHERE sighting_date BETWEEN '2022-01-04' AND '2022-01-10';
 count
-------
     1
(1 row)
```

Where we include a new word Count, to count the number of results from the plans table where the sighting date is the first week of January. When I run this you can see that instead of getting back and name and sighting date, both of those are counted and just the number is returned.

```
plants_dev=# SELECT COUNT(*) FROM plants WHERE sighting_date BETWEEN '2022-01-04' AND '2022-01-10';
SELECT COUNT(*) FROM plants WHERE sighting_date BETWEEN '2022-01-11' AND '2022-01-17';
SELECT COUNT(*) FROM plants WHERE sighting_date BETWEEN '2022-01-18' AND '2022-01-24';
SELECT COUNT(*) FROM plants WHERE sighting_date BETWEEN '2022-01-25' AND '2022-01-31';
 count
-------
     1
(1 row)

 count
-------
     0
(1 row)

 count
-------
     0
(1 row)

 count
-------
     1
(1 row)
```

Now, we can just run this query four times and update the dates for each query. When we do that, we can see that the first week of January had one. The second week of January had zero followed by zero and ending with a one in the last week of January, so our stakeholder can see a positive trend in engagement over the last month. This is pretty cool and also satisfies the report we were supposed to generate.

Lesson Conclusion

We will learn more SQL queries and database actions in future lessons, but what you have just learned will allow us to build an API with NodeJS and Express in the next lesson. You have a great foundation to build on.

Topics we covered

- Explore popular databases and their use cases
- Get started with SQL
- Create a database with PostgreSQL
- CRUD operations on a database
- SQL filters
- Connect database tables via foreign keys
- Translate data requirements into a database schema or SQL commands

Glossary

- **CRUD** - Acronym for CREATE, READ, UPDATE, DELETE
- **Meta Command -** a management command starting with `\` like in `\c` for connect
- **Primary Key -** A column with unique identifiers for each row in a table
- **Foreign Key -** A reference to the primary key of another table
- **SQL Filter -** A SQL command for filtering or narrowing the result rows
- **Database Schema -** The high level structure of a database, you can think of it as the general blueprint of a database with information about what tables it holds and what columns exist on those tables

Other Helpful Acronyms

- DBMS - Database Management System
- RDBMS - Relational Database Management System

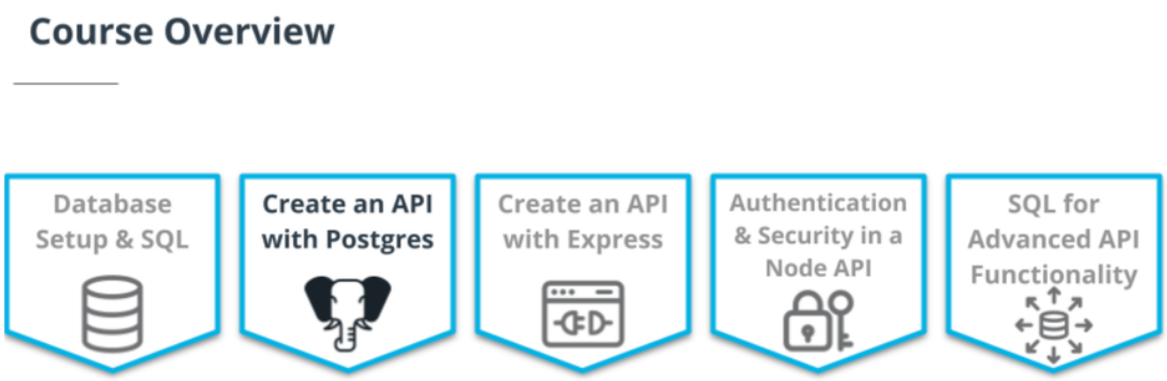
Going Further

Here are some resources help you explore more queries and take you deeper into database land:

- Chartio's database blogs are an awesome source of documentation and tutorials
- TutorialsPoint has extensive documentation on Postgres commands
- Database Guide has all sorts of database-related information, linked is the section for SQL
- Intro to Postgres from Prisma has good entry info about Postgres

Create an API with a PostgreSQL connection

Introduction & Lesson Overview



What we'll do

This lesson is all about the database facing side of our API. We'll cover the following topics

- What it means to connect to a database
- The role of environment variables
- Database migrations
- Models in NodeJS
- Testing models

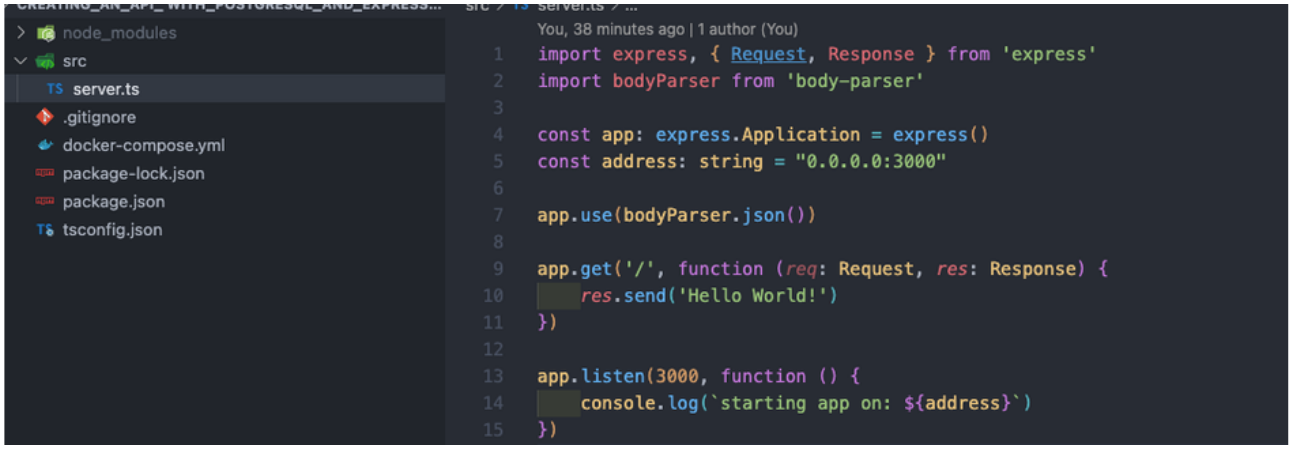
Helpful Preparation

- If you need a refresher on Express, check out the Mozilla docs.
- It will be greatly beneficial, though not strictly necessary, to understand MVC! This article is written for Ruby on Rails, but its a good breakdown of what MVC means and how it fits into the flow of requests and responses.

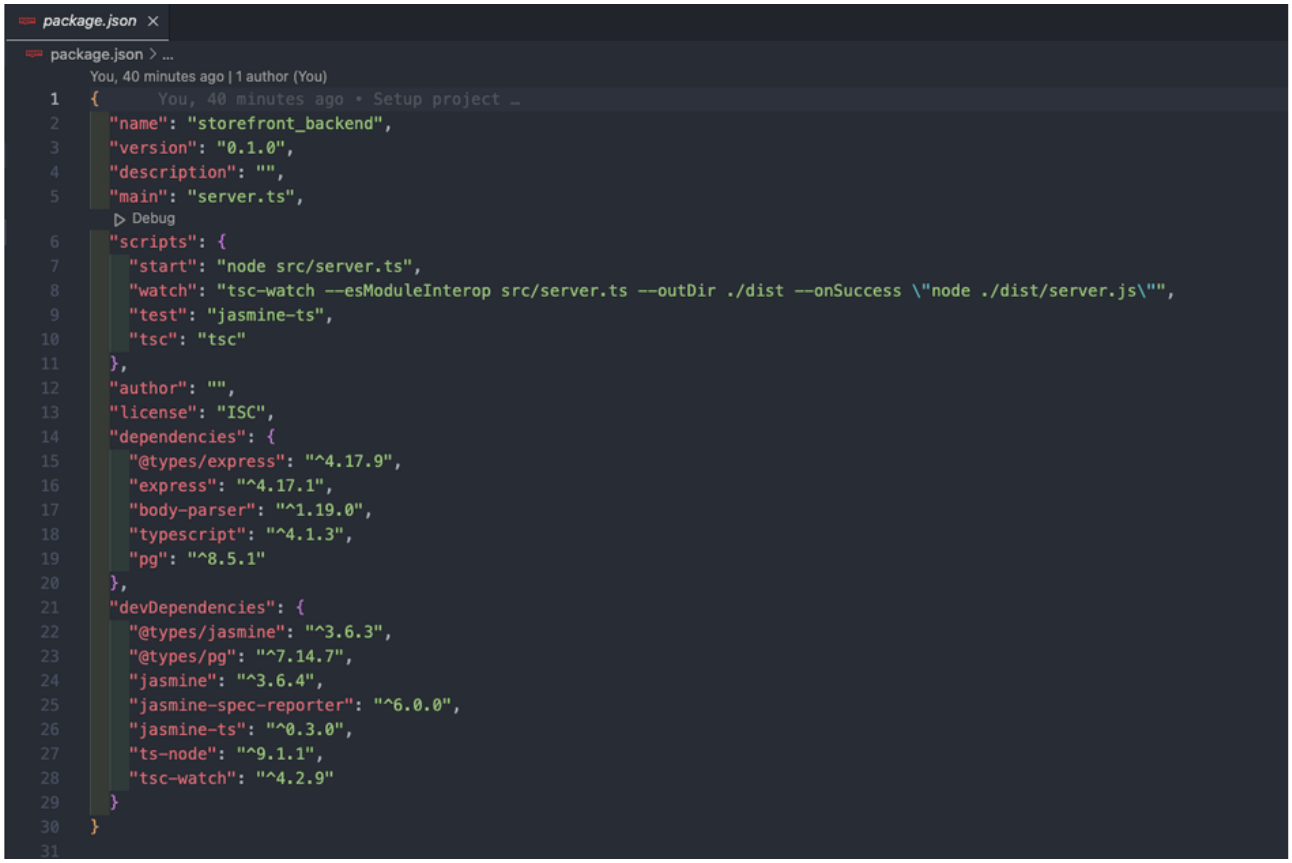
Connecting Node to a Postgres Database

A Tour of Our Node Application

In this section, We are going to give you a little tour of the starter node application we'll use in this lesson. It is the same starter application that you'll get for the project. At this point in time, it's really simple. We just have one server file, and it imports express and body-parser and has one test drought, and starts the server on port 3000.



In the package JSON, you'll see some of the dependencies I've already added, which are expressed in its types, TypeScript and Postgres. We also have Jasmine for testing and TSC watch for compiling and watching your TypeScript files. In the script section, I've provided an npm watch command that will allow you to start the compiler that will watch for any changes that you make in your TypeScript files.



Documentation

Here are links to the documentation for the libraries I talked about in discussion above

- NPM Postgres
- Typescript watch

Where Databases Run

Since we are talking about connecting and running databases, I want to touch on where these databases are, because that can be confusing and there are lots of options.

- You can run a database on your computer and connect to it **locally** kind of like localhost.
- You can run a database on your own computer in a **container system** like Docker.
- Container systems are also common remotely, you can connect to a database running in a **virtual machine**.
- Services like AWS and Azure provide databases in the **cloud**.

In each of the situations above, the process of connecting to a database will look a little different, but you will still be connecting to a database as a user. Generally speaking, the more technologies or layers between you (or your application) and the database you want to use, the more complex it will be to connect to that database. To keep things simple for this course, the Node application and Postgres database will run in the same VM.

Environment Variables

Working with sensitive information can be hard, especially when your application relies on keys and passwords in order to connect to and access databases or APIs. The instructions below will walk you through adding a library for environment variables in Node so that we can safely store information away from public eyes without moving it out of reach.

1. The library we will use for environment variables is called dotenv. You can add it via npm or yarn like this: `npm add dotenv`
2. Once we have dotenv listed in the package.json dependencies, we need to create one file. Make a new file called `.env` in the root of the project. In that file, add this: `TEST_VAR=testing123`. This is our first environment variable!
3. One last, **super** important step. The .env file hides sensitive information and makes it available to our application via a variable, so it holds a lot of really important, secret information. Information we don't want shared even in a respository. If a gitignore file exists in your project add the .env file there. If there isn't a gitignore, add a file called `.gitignore` to the root of the project and add a single line in the file that says just `.env`. **If you include your env file in a public repository, you have completely negated the purpose of adding environment variables.**

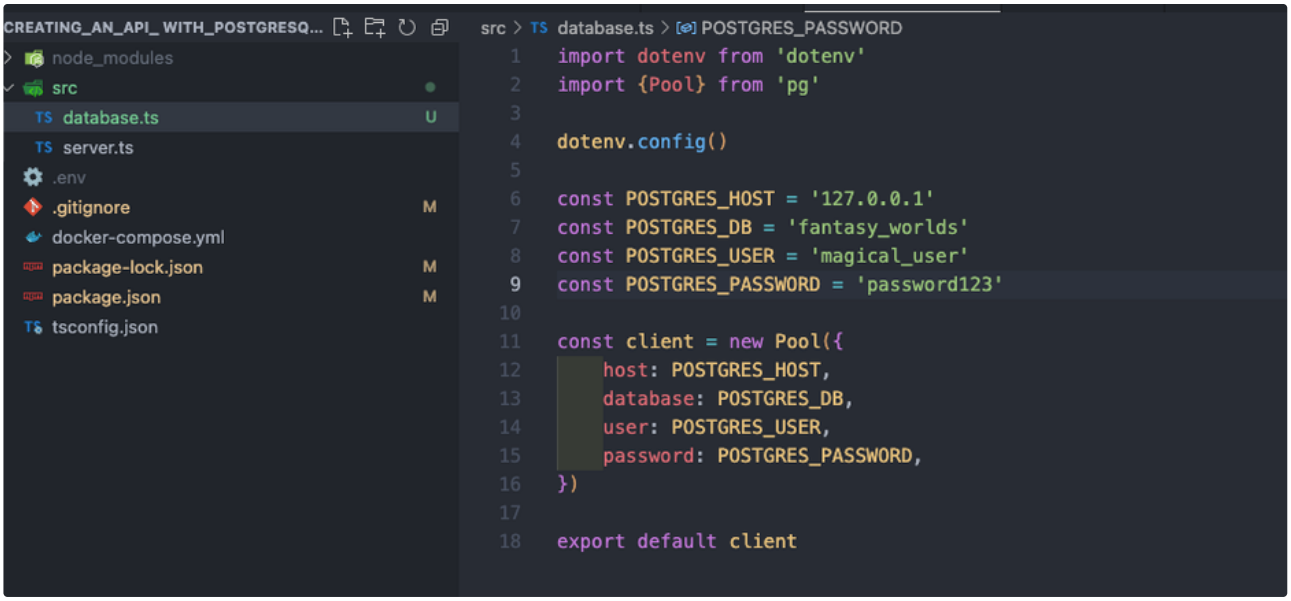
The dotenv library documentation lives here.

Note: your app should be agnostic to any environment so it can run on a local computer with any operating systems, docker, or the cloud. Other than storing the secret credentials, you can also use the `.env` file to set up portable environment variables, such as assigning a PORT number and determining development/test/production environment. To set up environment variables using `dotenv` library, this blog post provides some tips and tricks to store, load, and organize NodeJS environment variables.

Connecting the database

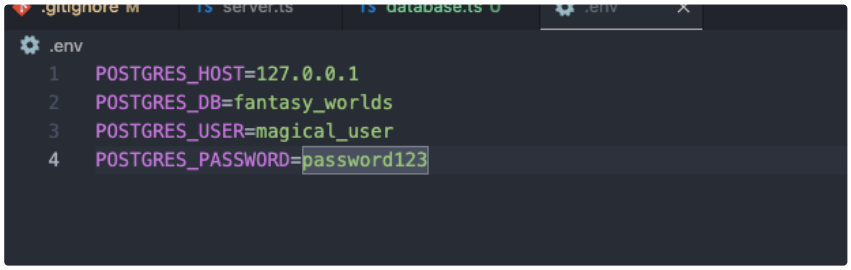
When we write the command `psql <username>` we enter the interactive terminal for working with Postgres. But with a backend application like the Node API we are creating in this course, **we** won't be the ones using psql and connecting to a database. The only one making changes to the database will be our Node app. The Node application is going to run the same psql command as we do to connect to the database, but we need to add some additional information.

In this section, we are going to connect our node application to a Postgres database. We will run our database locally in a Docker container. To connect this app to Postgres, we're going to make a new file called Database.ts in the source folder next to the server file. The job of this file is to provide the information that our application needs in order to connect to a Postgres database.

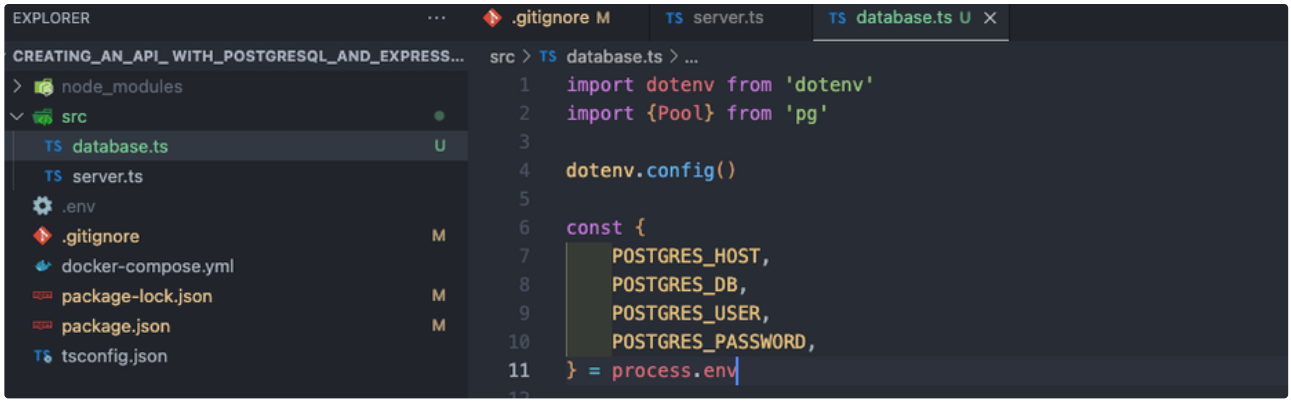


First, the information we need is the host or location of the database we're trying to connect to. In our case, that's 127.0.0.1 or my local server. The next piece of information that we need in order to connect to a database is the name of the database. In my case, I've made a database called fantasy_worlds. The third piece of information that we need is the name of the Postgres user. When we used PSQL Postgres, we connected to the database as a user called Postgres. But we don't do that when we're connecting as an application. Instead, we create a user specifically for our application. I've created a user just called magical_user. The last piece of information that we need is the password related to the user. We didn't have this for Postgres because Postgres is only for development. But in this case, I've added a simple password of password123.

Because I just hard coded a password into my file. As we learned with environment variables, this is something we should never do. This password should be stored securely in an environment variable.



That's what we're going to do. In my .env file, I have conveniently created all four of these as environment variables, which means that here in my database file, I'm going to instead reference the environment variables instead of creating these constants. The `dotenv.config()` line initializes the environment variables. You can't access the env vars unless this line exists in your code, it typically goes as close to the beginning of the program as possible.



Now, I'm getting all of these values from the environment variables file and passing them to a new thing called Pool.

```
const client = new Pool({
    host: POSTGRES_HOST,
    database: POSTGRES_DB,
    user: POSTGRES_USER,
    password: POSTGRES_PASSWORD,
})

export default client
```

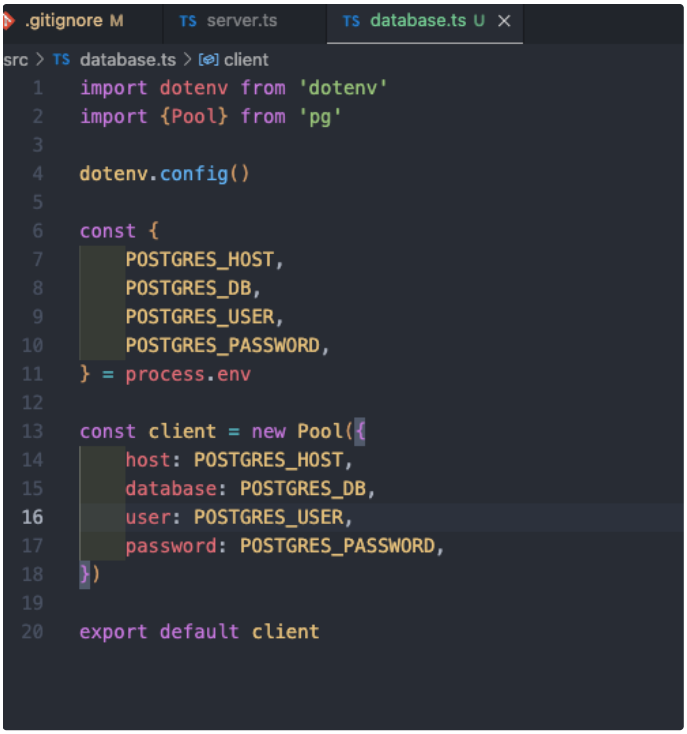
Now, Pool is the actual connection to our database and it is a method that comes from the Postgres library that we imported. Think of Pool as just one or many connections to a database. For this application, I've called it client, but you can also call it Postgres or database as you wish. Now I'm set to use these values from the environment variables file instead of using constants here in my code.

To finally make this connection, we take Pool from the Postgres NPM library that we imported. Think of Pool as a connection or set of connections to the database. We instantiate it and we'll call this client, though you can also call it Postgres or database as you wish. Then all we need to do is pass it the parameters that it needs to connect to the database.

If any of these steps feel hard to follow or abstract, it's worth remembering that all of these details are part of implementing the Postgres library. No developer knows to use Pool automatically. The code here is all from the documentation for this library. Now, we have a connection to the database, but we have no way to prove it.

For that, we need some code and in the next section, we'll get started with migrations and start using our app to make changes in our database.

The final code looks like this:


.gitignore M
server.ts
database.ts U
src > database.ts > client
import dotenv from 'dotenv'
import {Pool} from 'pg'
dotenv.config()
const {
POSTGRES_HOST,
POSTGRES_DB,
POSTGRES_USER,
POSTGRES_PASSWORD,
} = process.env
const client = new Pool({
host: POSTGRES_HOST,
database: POSTGRES_DB,
user: POSTGRES_USER,
password: POSTGRES_PASSWORD,
})
export default client

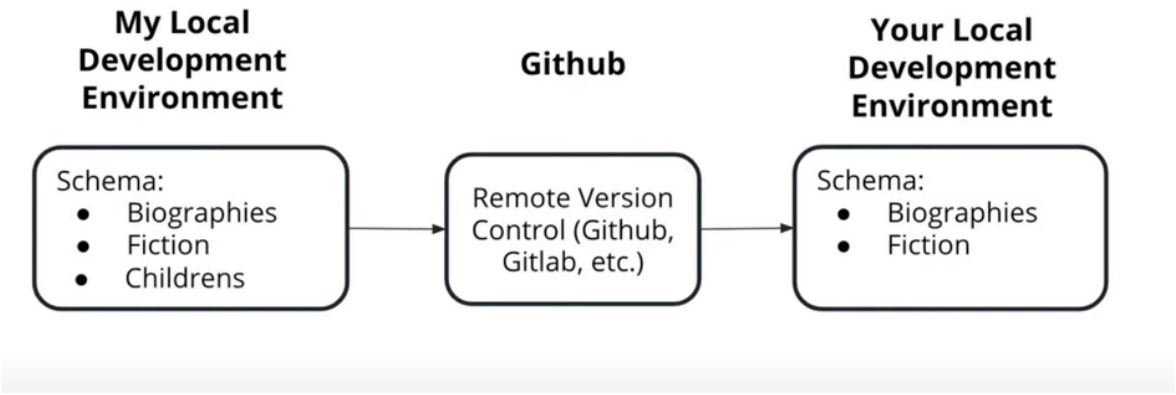
Introduction to Migrations

Migrations

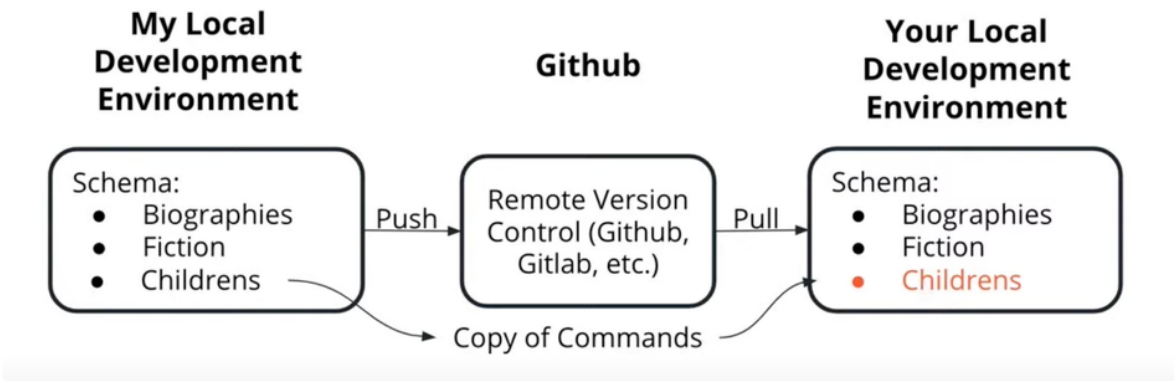
Before we get started, I feel it's important to say that database migrations could be a **big** topic, with much more time devoted to it than I am going to spend here. That being said, the complexity level of migrations can easily go beyond what we need for the application we are building in this course, so I have tried my best to contain the topic of migrations to just what is needed to implement this kind of project well, without losing any of the most important concepts. There will be resources available at the end of this lesson to help you go further with migrations, and I hope you find this section to be a helpful start!

That is out of the way; it's on with the show. In this section, I build a case for why migrations are helpful and introduce the foundational concepts of how they work.

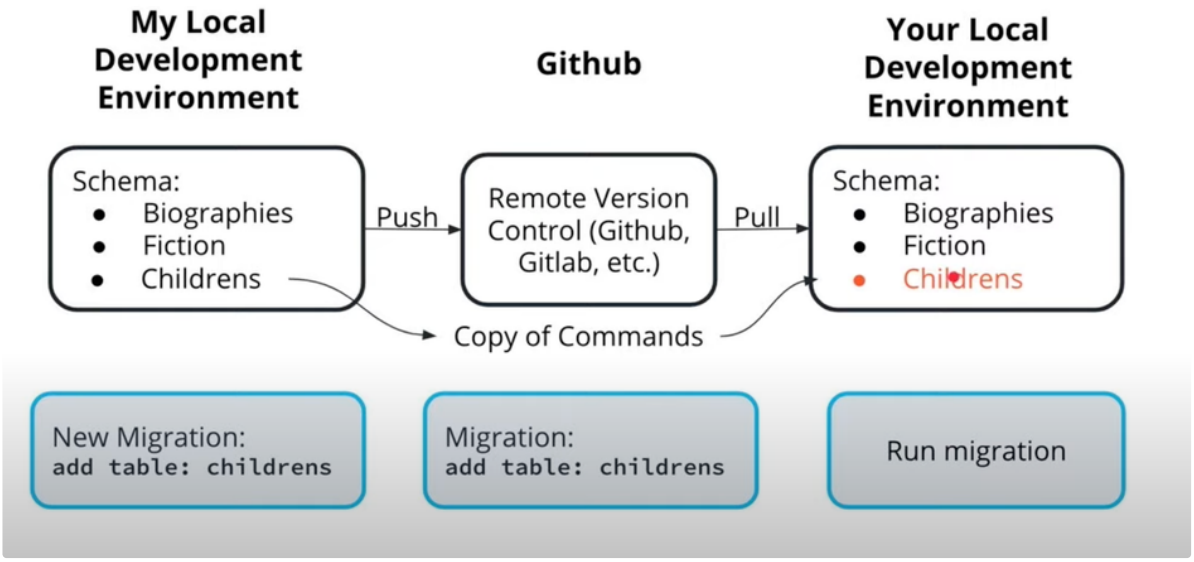
In the last lesson, we learned a lot of database commands. Now, I want to do a bit of a thought experiment with you. Imagine that we are coworkers, and we're both working in our own local environments with separate copies of the database to build an app. If I work on a ticket that adds a table to the database and updates the code, I will push that code to the repo, and you could pull the changes down.



But now, our databases are out of sync. My code changes expected table to exist, but your database does not have that table. What do we do? I would have to send you a copy of the database commands I ran on my machine and you would have to manually run those commands on your copy of the database. Then, we would have to run those commands manually again on the production environment when we push these changes live.



But that's just not an adequate solution. Imagine trying to then roll back some of the database changes we made this way. There's a need for a unified place to talk about all of the changes that we make to a database. The way we do that is with migrations. With migrations, what I would do is create a migration as part of my task to add a table.



That migration would state what table I am adding and what columns it has. Then that migration is part of the code repository. When I push it to version control, it stays there as a record of the change that I made. When you pull these changes down, you will have the migration that is the record of the change that I made on the database. You can run that migration to get the necessary table into your local database. Migrations are the topic of this lesson and we'll go into further detail about how to implement them in the next section.

Documentation

- This is the npm library we will use for database migrations: DB-migrate

Instructions to install db-migrate

1. Install the global package `npm install -g db-migrate`
2. Install the package to the project `npm install db-migrate db-migrate-pg`
3. Add a database.json reference file in the root of the project. Later, when we are working with multiple databases - this will allow us to specify what database we want to run migrations on. Here is an example **database.json**, you will just need to change the database names:

```
{
  "dev": {
    "driver": "pg",
    "host": "127.0.0.1",
    "database": "fantasy_worlds",
    "user": "",
    "password": ""
  },
  "test": {
    "driver": "pg",
    "host": "127.0.0.1",
    "database": "fantasy_worlds_test",
    "user": "",
    "password": ""
  }
}
```

1. Create a migration `db-migrate create mythical-worlds-table --sql-file`
2. Add the SQL you need to the up and down sql files
3. Bring the migration up `db-migrate up`
4. Bring the migration down `db-migrate down`

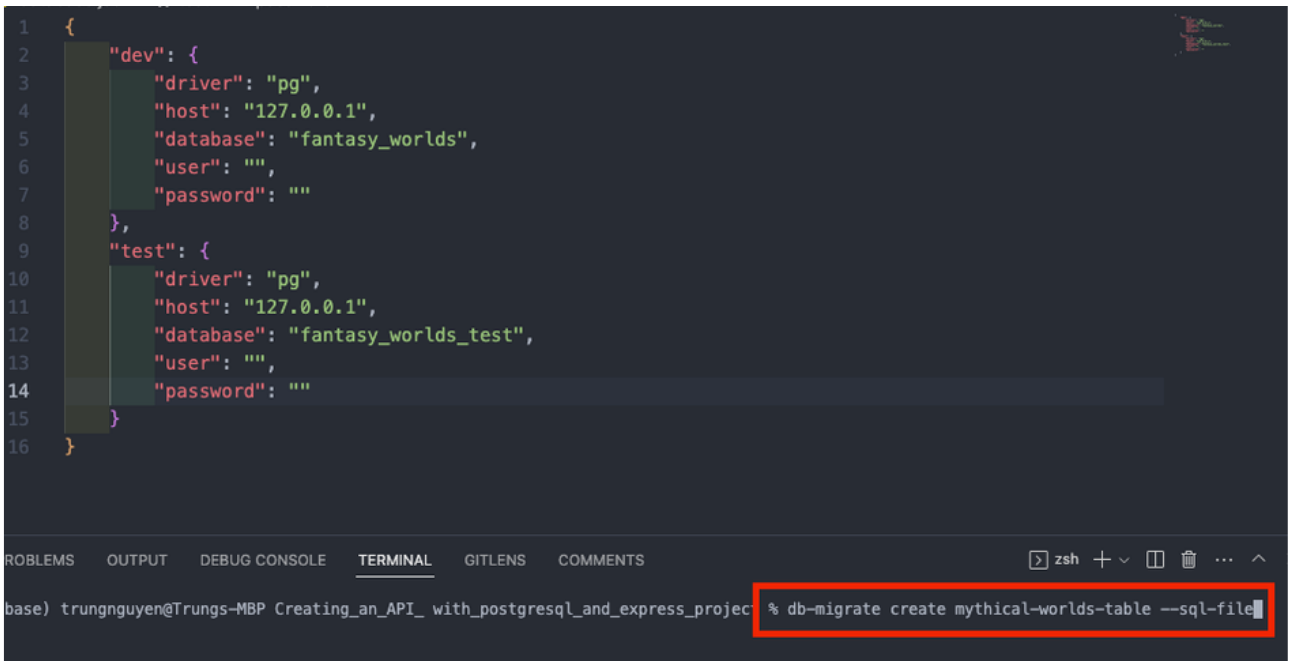
Explanation of all the step above

The situation I described in the last section is a picture of the kind of problems database migration solve. Essentially, migrations are documents outlining the changes made on a database over time. These changes can be applied to the database or rolled back and removed from the database. Migrations are only for tracking changes to the database schema, never to individual rows in a table. If you remember, the database schema refers to its structure, its tables, columns, and the column types. For this course, we'll be using the npm library, db-migrate.

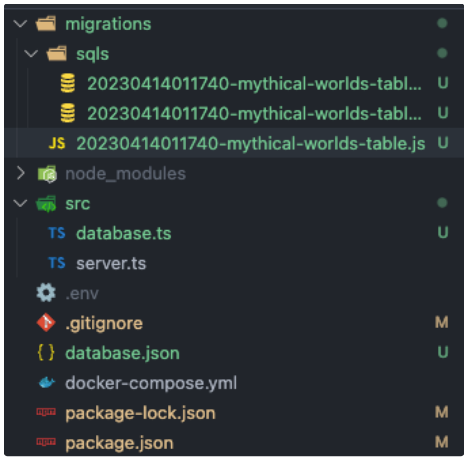
To install db-migrate first run npm install -g db-migrate. Adding this library globally will allow you to use the terminal commands that it provides.

Next, we'll run npm install db-migrate db-migrate-pg. These will create the dependencies in our project and record them in package.json.

Now that we have those installed, let's get a better understanding of what a migration is by creating one. Db-migrate gives us a terminal command to create a new migration, which looks like this.

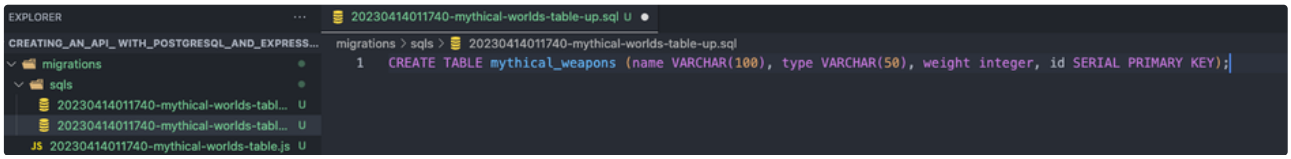


If I run this command, you'll see that I get a new folder called migrations.

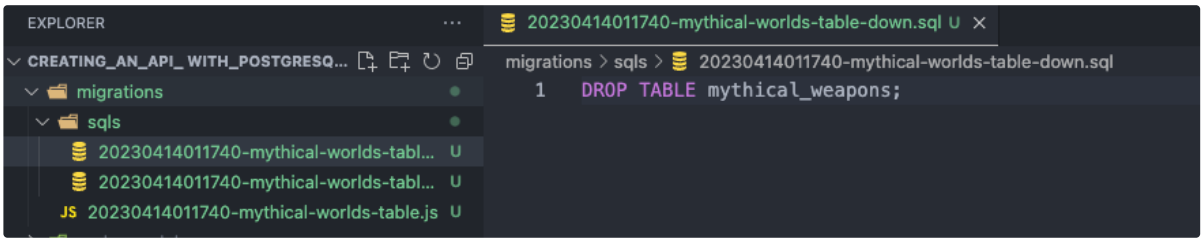


In the Migrations folder, you see that we get one generated file which you won't need to update. But we also get some SQL files. This is where things get really cool because we'll actually get to use what we learned in the last lesson and write SQL in our node application.

You can see that this file has some space for SQL commands and its name is mythical-weapons-table-up.

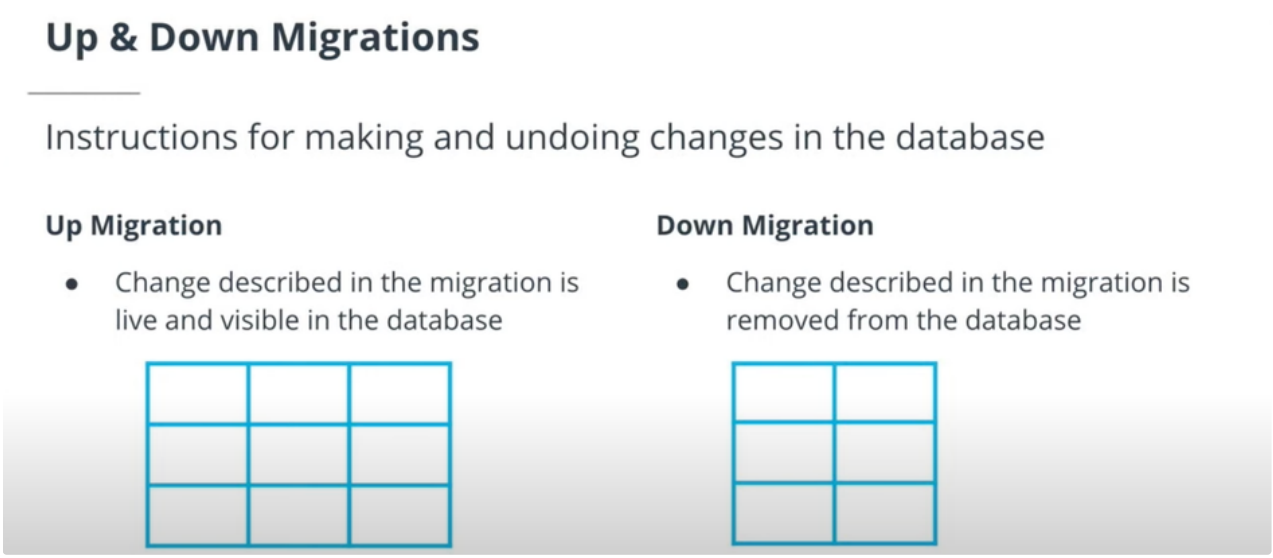


This SQL should hold the command for creating a table called mythica_weapons. Using what we learned from the first lesson we can create a table called mythical_weapons with an underscore because this is Postgres and we can define the columns for this table. In this case, we're going to have name as a string, type as a string, weight as an integer, and we'll also add an id that is a serial primary key. Now if we save this migration, we also have a second SQL file to take a look at. This one is called mythical-weapons-table-down.



A down migration actually holds the instructions to remove whatever change or undo whatever change was made in the up migration. Here I'm just going to type DROP TABLE or REMOVE THE TABLE called mythical_weapons. The important point here is that for every change that we make to the database, every migration that we create, we record not only how to make that change, but how to undo that change. That's one of the big benefits of using migrations to keep track of the changes that you make to your database.

For a quick visualization of what up and down migrations mean, that is if we have a migration for adding a table to a database, the up part of the migration means that this change is enacted on the database. We can see this change is live in the database. The down migration refers to removing or undoing that change that we made on the database, in this case, removing the table. With migrations, we get the power to create changes on a database and roll back those changes, and we get the ability to keep multiple databases in sync with each other by referencing common files.



Back in the code, the very last thing that we need to do is actually run this migration on the database. I do that with a command from the db-migrate library called up. When I run this, you can see that CREATE TABLE called mythical_weapons was actually run on my database as part of the migration called mythical-weapons-table and that it completed.

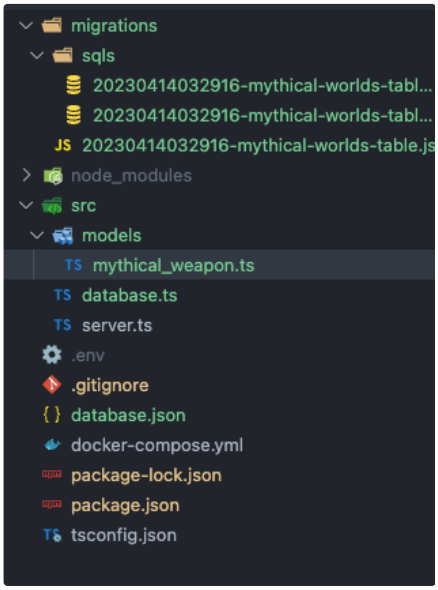
```
(base) trungnguyen@Trungs-MBP Creating_an_API_ with_postgresql_and_express_project % db-migrate up

received data: CREATE TABLE mythical_weapons (name VARCHAR(100), type VARCHAR(50), weight integer, id SERIAL PRIMARY KEY);
[INFO] Processed migration 20230414032916-mythical-worlds-table
[INFO] Done
```

This tells me not only that my node application has a successful connection to my database, but also that I'm able to use the db-migrate library to update the schema of that database.

Introduction to Models

In the previous lesson, we spent much time on tables and CRUD, but now our Node application will trigger all those actions. The focus of this session is performing CRUD operations on the database from within a Node program.

In the last section, we created a migration for the mythical weapons table. Now, we're going to add support for crud actions on that table. First things, let's set up our folders and files. You can see that I have a new folder called models inside of the source, just above the database and server.



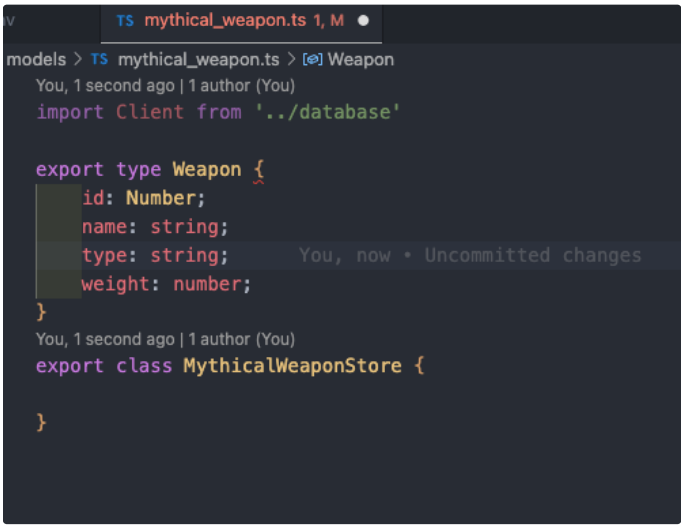
In this folder, I have a file called mythical weapon.ts. We will talk about this file for just a moment because it's really cool. A database table holds many items of the same type. The rows all share the same columns or properties.

In code, when we want to create many items, all of the same type or from the same template, what do we use? We use a class. A table in a database can be represented in code as a class, in which case each row can be an instance of this class. Also notice the naming of the file. Database tables are always plural because they hold a list of many things. But model names are singular because they are a single template, the mold, if you will, the class that will be used over and over to make each new item. Each instance of this class will become a new row in the database table.

In this model file, the first thing I do is import the database connection. Now let's take a look at what we'll build. The table in the database has columns, name, type, weight, and an ID.

```
You, 1 second ago | 1 author (You)
CREATE TABLE mythical_weapons (
    name VARCHAR(100),
    type VARCHAR(50),
    weight integer,
    id SERIAL PRIMARY KEY
);      You, now • Uncommitted changes
```

Since we're using TypeScript, we should make this into its own TypeScript type, which can be represented like this.



Great. We have our TypeScript type. Now let's get ready to write the crud functionality. First, you'll notice that this class is called the mythical weapons store. Store is often synonymous with database. But we use the word store here because within the context of our node application, this class is going to be the representation of the database, the Postgres ambassador in JavaScript land.

Let's add a method. Since read requests are the simplest, we'll start there. We'll make a method called index. Index to mean that this is going to get a list of all the items that we have in the database. This method needs to be asynchronous because all calls to the database will be promises. Next, remember that everything after the colon in TypeScript talks about what type of thing will be returned from this function. In this case, our function returns a promise of a weapons array rather than the weapons array itself.

```
9  export class MythicalWeaponStore {
10     async index(): Promise<Weapon[]>{
11         const conn = await Client.connect()
12         const sql = 'SELECT * FROM mythical_weapons'
13         const result = await conn.query(sql)
14         conn.release()
15         return result.rows
16     }      You, 1 second ago • Uncommitted changes
17 }
```

The first thing that our method is going to do is use the connection to the database to open a connection so that we can actually talk with the database and the table that we want. This means that next, we can write a SQL command in our node application that will get run on the table. In the case of the index method, I'm going to write SQL, select all from mythical weapons. This is super cool because we get to use what we learned about queries in the first lesson, use them in a node application with TypeScript, and have them run on a Postgres database.

The next thing we will do that is run this query on the database. Here you can see that the SQL that I stored in this variable is getting passed to our connection as a query. This query will get run and the resulting rows will be saved as the result. Now we have a place to store the resulting rows from our database query. We have two last things to do. When you open a database connection, you also need to make sure to close that connection when you're done. This release word is going to close the database connection that we opened on line 11. Finally, all we do is return the rows contained in the result from the database query.

We wrote SQL in a JavaScript file, made that query on a database, and got a response. There's one last thing we should do to protect ourselves from any errors. We're making a connection to an external database. This could fail. So it's best if all of this code we're actually encapsulated in a try-catch statement.

Here I'm going to wrap all of the code in a try, and add a catch just in case anything goes wrong. Here I'll have an error. I'll throw a new error with a message, and I will pass the error. Now we can make the request and query on the database with the peace of mind that if anything did go wrong, the error is still caught.

Now we can make a request to the database with the peace of mind, knowing that if anything did go wrong, the error is caught.

Summary

- Walkthrough for creating the file and folder structure for models in the Node app
- Walkthrough creating a model file with methods
- Tables hold a list of items that share properties (columns), models are a class in our code that can be used as a template to create items that are stored as rows in the table.

Documentation

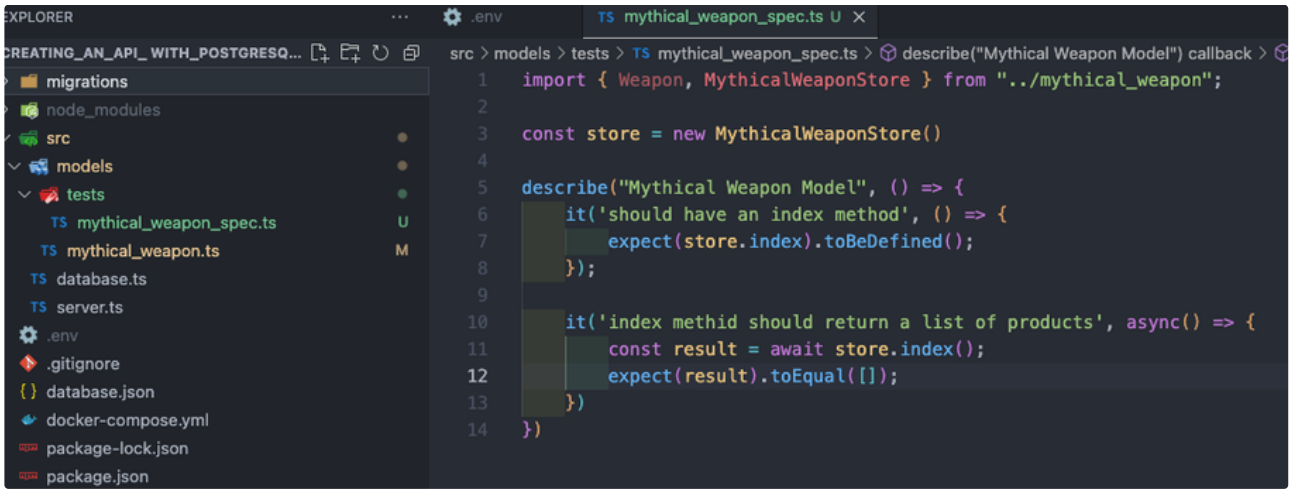
- Express documentation

Testing Models

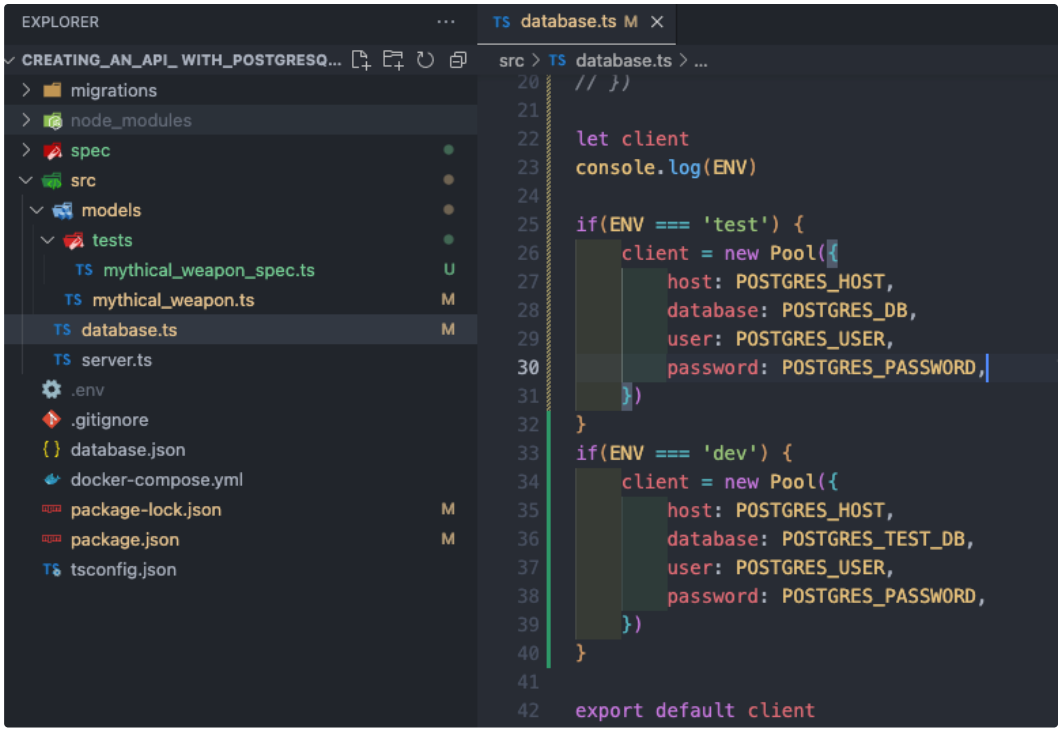
Models are a main component in our API, therefore they should be well tested so that developers can sleep well at night. The focus of this section is installing the testing library Jasmine and creating unit tests for the models we created in the last section.

In this section, I'm going to cover how to use Jasmine to create tests for our models. Normally, you would write unit tests to cover classes in your code. But these tests are going to go a little bit beyond regular unit tests because they're going to test an integration with the database. We don't want to just know that code runs; we want to know that the model methods have their intended effect on the database.

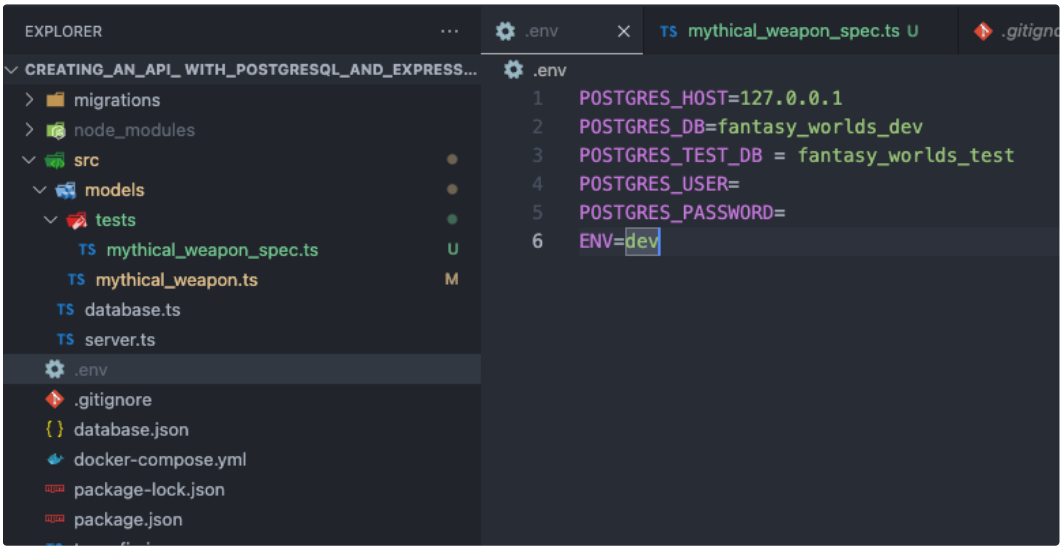
Jasmine will already be set up for you in the workspace and later on in the project, so you can get straight to testing. First off, I've created a tests folder inside of my models folder. Where you put tests, whether they're in a spec folder at the root of the project or in folders alongside the code they test, it's really just a matter of personal preference.



Personally, I like to put my test files close to the code that they test simply because I feel like I'm less likely to forget to create or update them. In this test file, I'm importing the type and class that we created in the model file. I'm also setting up a basic describe block with some it statements. The first test is fairly straightforward. It's just checking that the method actually exists. The second test though is checking for a specific array result from running the index method. The question is, how can we be sure what this array will be? If I'm running this test with my development database, there's no way for me to be sure what this array will contain and it will probably change over time. So in order to run this test, we need a clean, isolated database for these tests to run on.

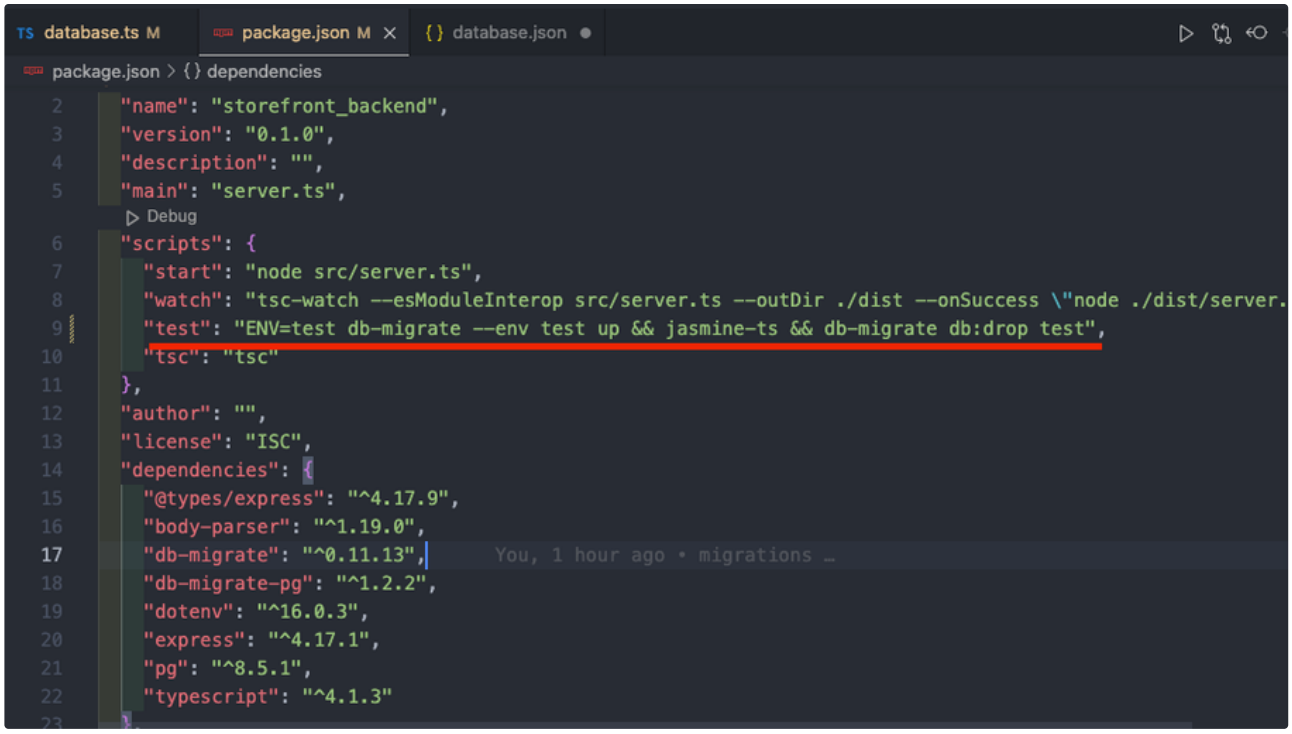


In order to do that, I've set up a testing database with some new environment variables. By appending the word test, I just make it more clear that this is a database only for testing and should never be updated at any other time. I've also added a new ENV tag with dev as the default. This is because most of the time that I'm running my database, it will probably be in development mode. But when I run my tests, I want to overwrite this ENV variable to contain test instead of dev.

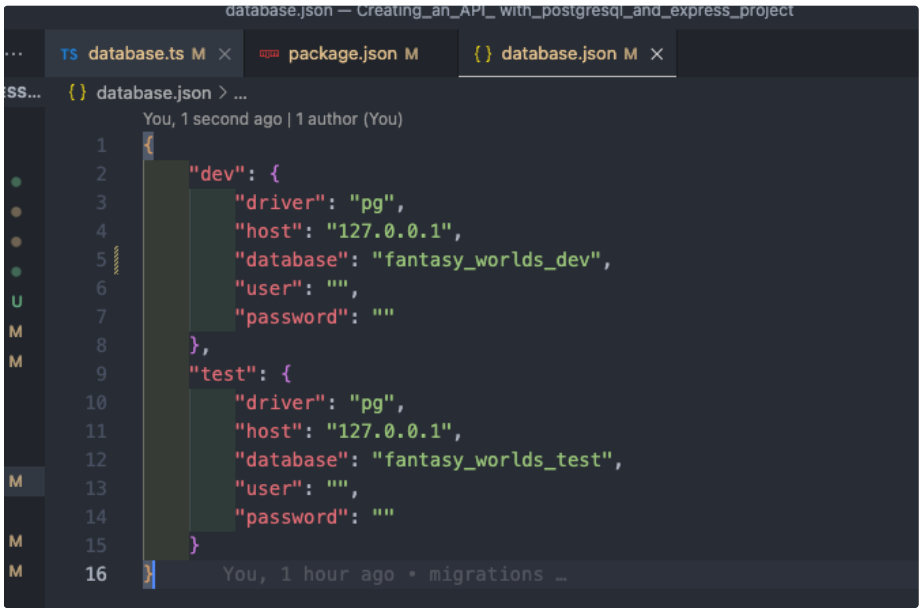


In json script, you can override an environment variable at runtime. Here, when I run npm test, the environment variable is overwritten to contain the word test. This will cause the database to look for the testing version of our database instead of our development version. There's another thing that this script needs to do though.

This new testing database doesn't have any tables. We have run the migrations on our development database but not on this new testing one. So we need to run our migrations on this test database. To tell db-migrate which database to use, I have to update the list of databases in the database JSON file, which is what db-migrate looks at to see the available environments.



Once I have this setup with the new test database, I can add that to the command to run the migrations before each test. There's one last really important thing the script needs to do, and that is to clear the database after every set of tests. That way, the environment is clean and fresh for each new round of testing that we do.



Installing Jasmine

- Add globally for CLI commands `npm install -g jasmine`
- Add Jasmine and its Typescript types locally to package.json `npm install jasmine @types/jasmine`
- Run Jasmine initialization to get test structure `jasmine init`

Integration vs Unit Testing

I touch on this topic in the section above but it deserves a little more explanation. This isn't a course on testing so I'm not going too far into it, but understanding the main differences between these is important.

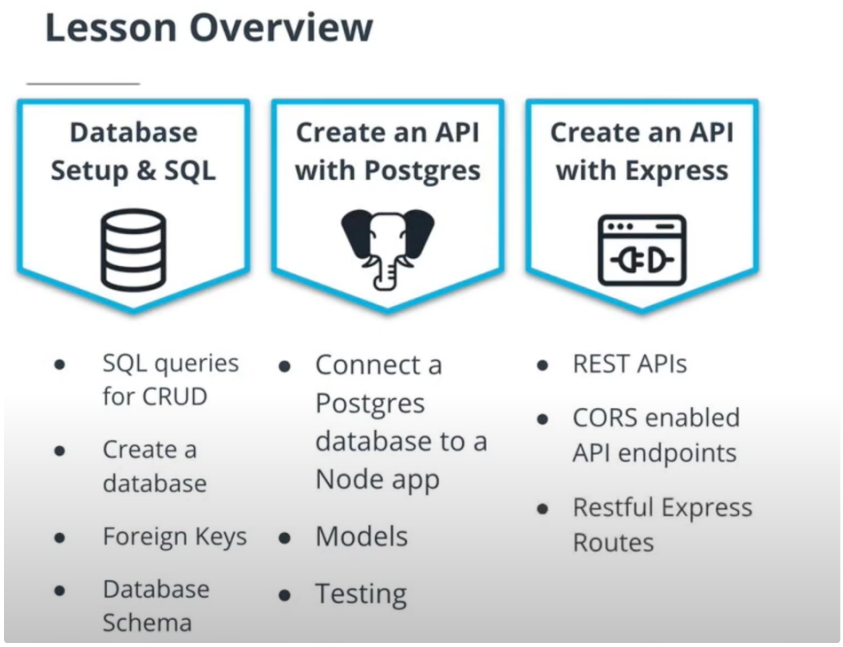
A **unit test**, tests a small chunk of code in your application. Unit tests are for the individual functions or classes in your code, to make sure each one is doing what you need. These are small, typically fast to write, and can really save you time when trying to understand where a problem is happening in the program.

An **integration test** checks how the individual pieces of your application logic work together. The span of one integration test will cover multiple chunks of code (that can and should each have their own unit tests) and make sure that working correctly together in a flow or process.

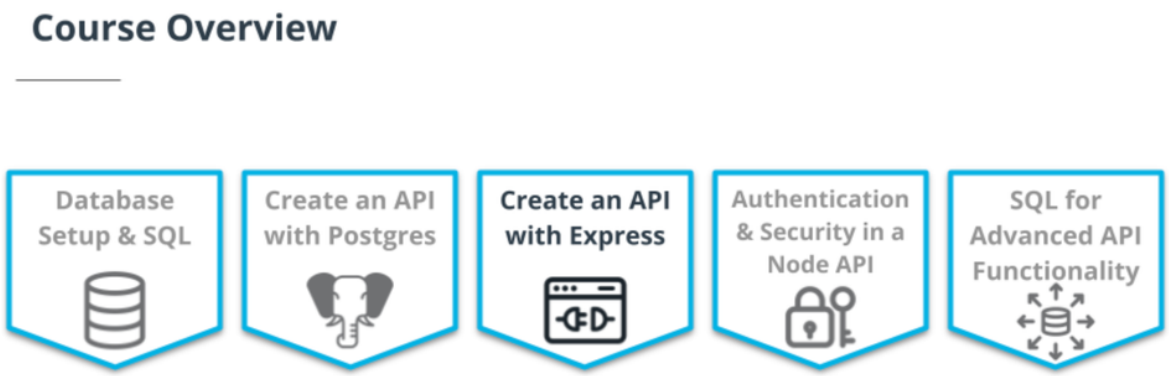
I refer to the tests we wrote in this section as integration tests because they test a flow in the application from model to database and back. Its not a large process, so they won't look very different from unit tests, but it is a good opportunity to talk about the differences.

Introduction

This lesson will cover how our API communicates with the outside world. We talk about RESTful APIs and how to create a RESTful architecture in our API. We'll also create routes and use CORS with the help of Express.



Course progress



You are here: Create an API with Express

What we'll do

This lesson is all about the client-facing side of our API. We'll cover the following topics

- RESTful API structure
- Express for incoming requests
- Breaking Express routes into separate files
- Mapping RESTful routes to model methods
- Testing routes

Helpful Preparation

- For this lesson it will be helpful for you to brush up on your HTTP basics - specifically requests and the HTTP verbs. Here is the Mozilla docs for reference.
- MVC. In this course I won't be discussion more a strict MVC design, but there are times that we will borrow from it. If you aren't already familiar with the Model View Controller design pattern, its a good one to be familiar with. Here is an intro article to some of these concepts.

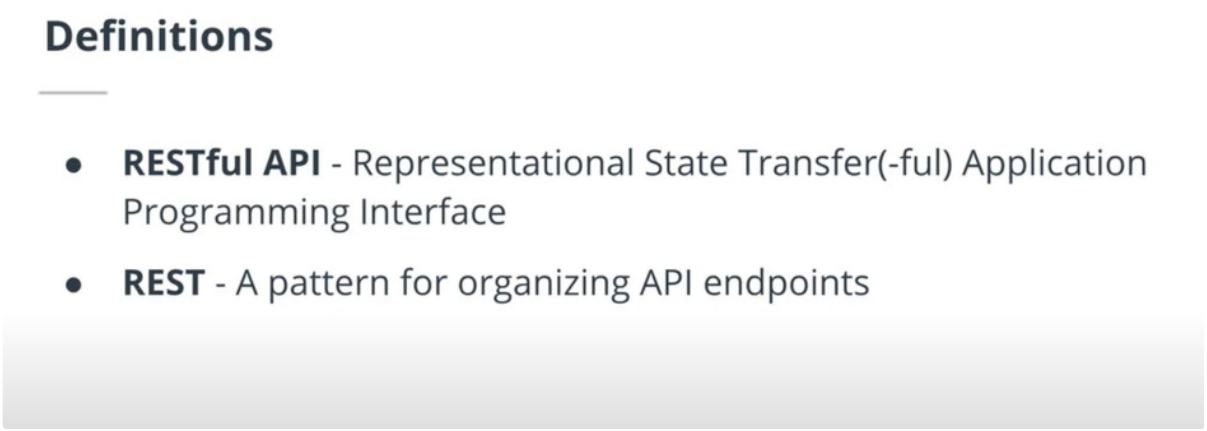
Intro to RESTful APIs

RESTful APIs

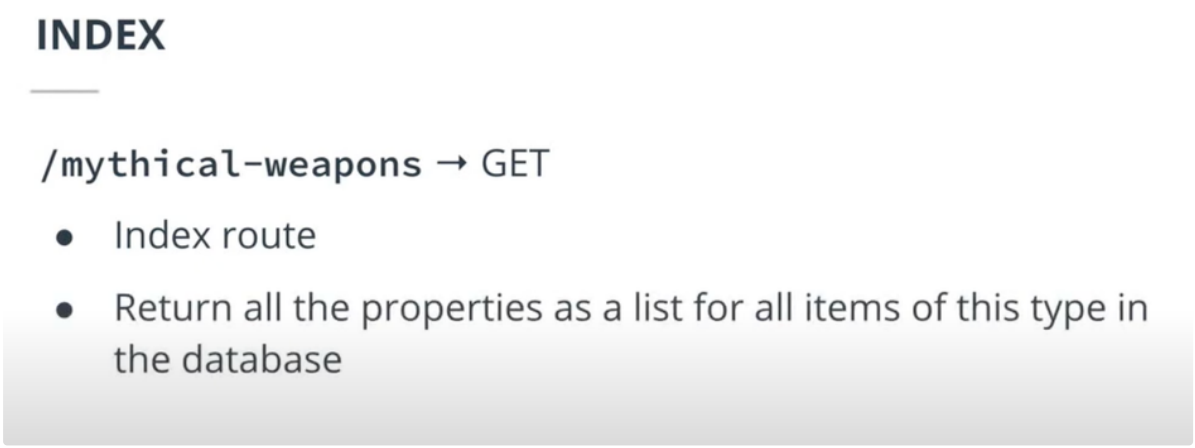
To get us started in this lesson, we are going to look at what happens when a client-side (aka from a browser or end-user) request reaches our API. This section is going to focus on what it means to be a RESTful API.

This course teaches you to build a RESTful API. But what does that really mean? It's a lot of acronyms to handle it once, and if you read them all together, that would be a Representational State Transfer Full Application Programming Interface. But not only is that a mouthful, it's also frustratingly unhelpful.

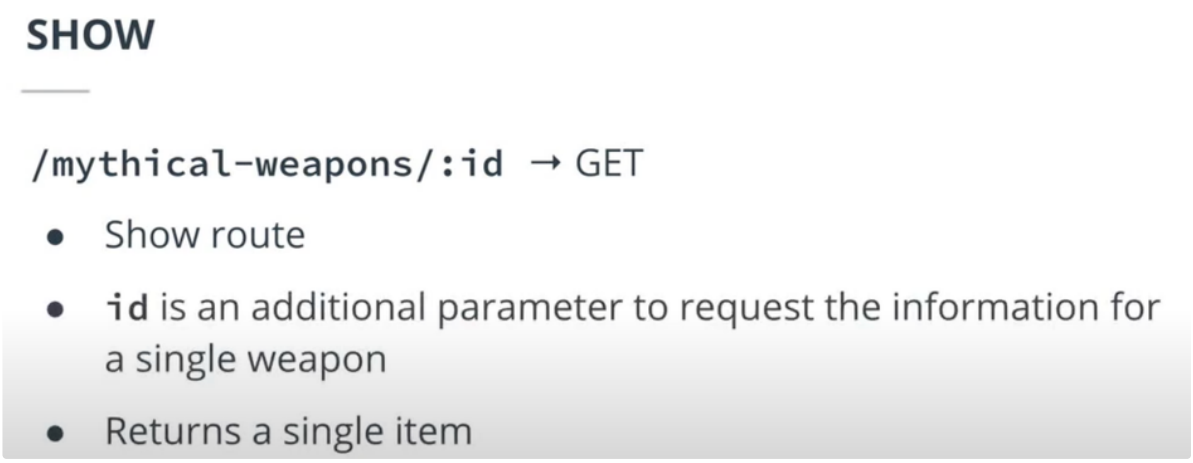
Rest is easier to understand as an organizational pattern for your API. It refers to how you structure your API endpoints, and how we organize API endpoints is exceptionally important. Just like it's hard and frustrating to browse a website if the pages aren't organized in a meaningful or predictable way, it's hard to get information from an unorganized API.



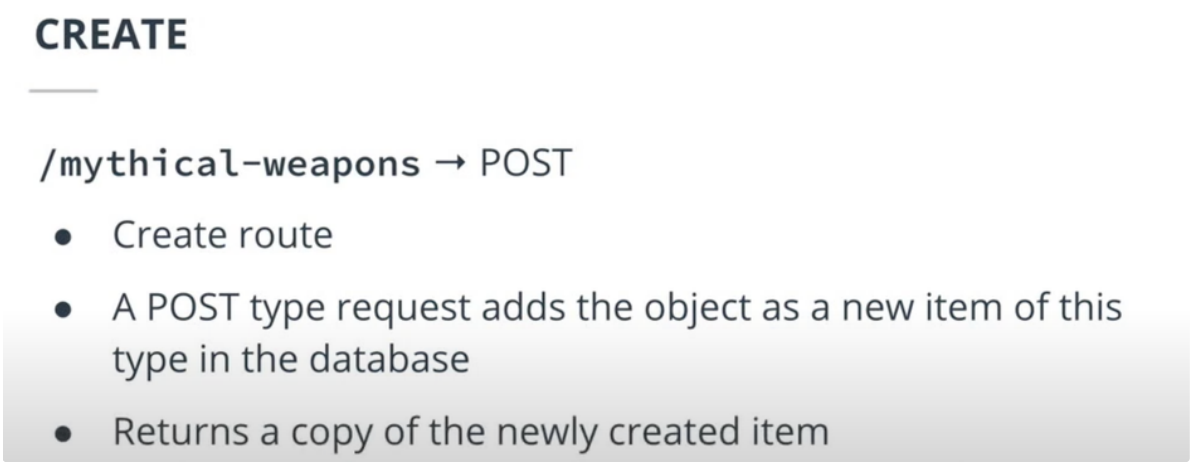
REST is great because it is a predictable pattern. If all I was told about an API was that it was RESTful, I can write most of my Fetch requests without much more than a glance at their API documentation. In this section, we'll go over the RESTful routes for the mythical weapons model we worked on in the last lesson.



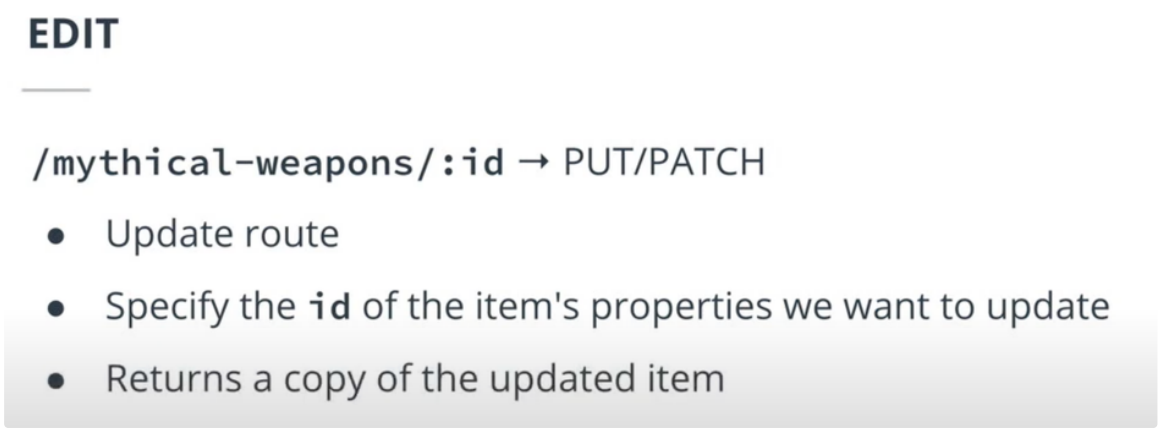
This is called the Index route. In a RESTful architecture, this route will return a list of all of the items in a database table and all of their properties.



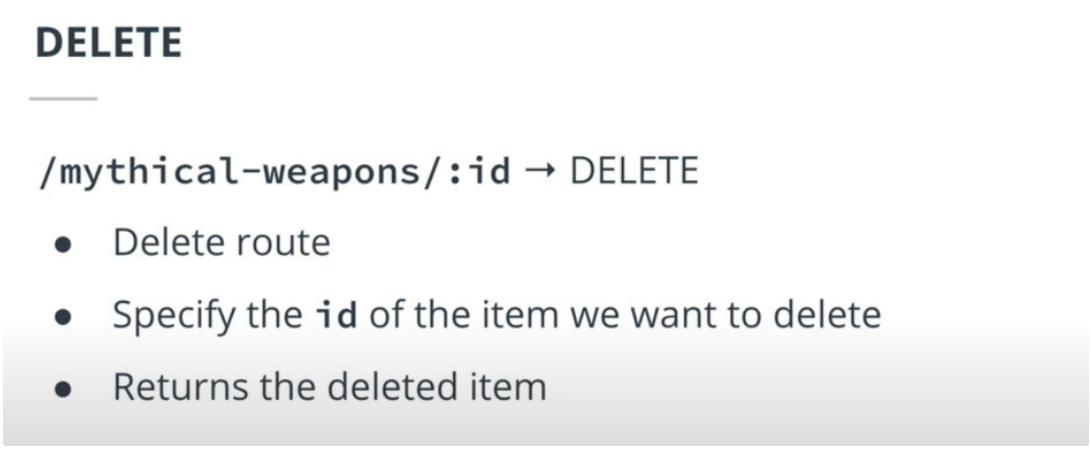
This route is called the Show route. It shares the URL path and HTTP verb with the index route. But by adding an ID parameter, we request the information for a single weapon. This route returns an object.



Next is the Create Route. It uses an identical URL path as the index route. But instead of a GET HTTP verb, this uses a POST HTTP verb. A POST type request to this URL, will add the object carried in the request body as a new item or new row in the database. This route typically returns a copy of the newly created item if successful.



This is the Update route. Just like the GET route, it specifies the ID of the single item it wants to operate on. The properties to be updated are in an object carried in the request body. This route typically returns a copy of the updated item.

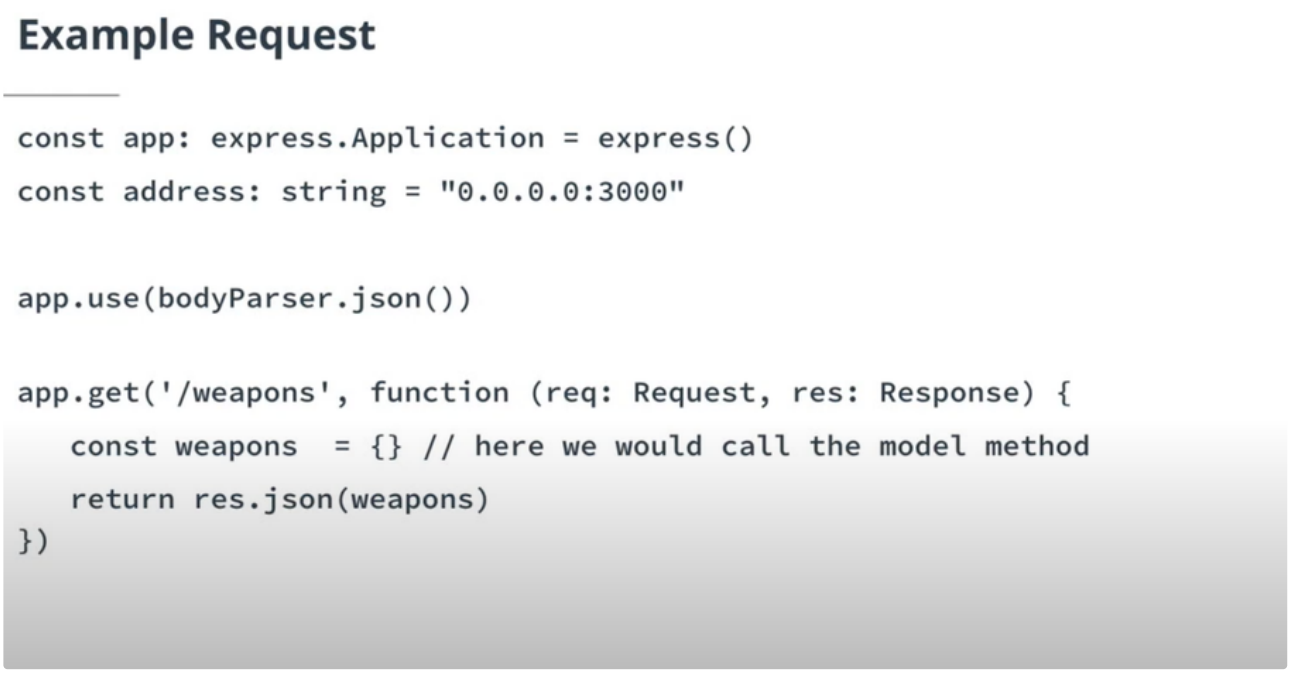


Last, we have the Delete route. Here we specify the ID of the single item we want to delete, and usually this route returns a copy of the deleted item.

Now you've seen all five of the RESTful routes for an API. We had the Index route that gets a list of items.

- Show that gets a single item
- Create, that creates an item
- Update to update a single item
- Delete to delete a single item.

Now let's take a look at the Express code for one of these RESTful routes.



In this code, you can see that Express is waiting for a GET type request to the /weapons endpoint. This means that we're looking for the index route on the weapons model. What we need to do, is call the Index method that we created on the weapons model here in this Express route. Then all we need to do is return the response from the model as JSON. Hopefully the puzzle pieces are starting to come together now, how the RESTful Express routes we build will fit with the model methods we created earlier.

Index Model Method Matches Index RESTful Route

```
export class MythicalWeaponStore {
 async index(): Promise<Weapon[]> {
   try {
     //@ts-ignore
     const conn = await Client.connect()
     const sql = 'SELECT * FROM mythical_weapons'

     const result = await conn.query(sql)

     conn.release()

     return result.rows
   } catch (err) {
     throw new Error(`unable get weapons: ${err}`)
   }
 }
```

There is actually a bit of foreshadowing and what we named the model methods because they match the RESTful routes. This was just some example code to show you how we can take in a request with Express, map it to a model method, get information from the database and return that information to the client.

In the next section, we'll go over creating the file and folder structure and the logic to make all of this work in our application

Summary

- REST implies a specific set of API endpoints for every entity in the app
- There are five actionable RESTful routes for APIs:
  - INDEX
  - SHOW
  - CREATE
  - EDIT
  - DELETE

CORS for API Endpoints

Adding CORS to our API is a small but essential step. This section discusses CORS, why we need it, and how to enable it.

CORS has been a riddling thorn in the side of most web developers at some point. But its purpose is to keep our apps secure and make the Internet a little bit safer place to be. What is it? CORS stands for Cross-Origin Resource Sharing. It states that a browser can only make requests to an API if they share the same URL domain. If they don't share a domain, the client-side domain must be white-listed by CORS in the API in order to have access. Adding CORS is super important if you plan to make your API publicly accessible. But you don't need it if your front-end and back-end are going to have the same domain. CORS is now used by all major browsers.

It really comes down to the fact that if we want a client-side application from another domain to be able to consume the API we're building, we must support CORS. Thankfully, the makers of Express knew that we might need this and included it in the Express library, which makes adding it quite painless. In this section, I'll walk you through the steps for adding CORS.

First, we need to install the npm package CORS by npm install cors and npm i --save-dev @types/cors. Then we need to create some CORS configuration options in the server.ts file. This comes straight from the Express documentation, and there's a lot more configuring that you can do. But the important thing here is that you can whitelist the foreign domains you want to allow. Once we have set these options, we can tell Express to use the CORS library and pass in the options that we created, and for our use case, that's it.

```
import express, { Request, Response } from 'express'
import cors from 'cors'
import bodyParser from 'body-parser'


const app: express.Application = express()
const address: string = "0.0.0.0:3000"

const corsOption = {
    origin: 'http://someotherdomain.com',
    optionsSuccessStatus: 200
}

app.use(cors(corsOption))      You, 1 second ago • Uncommitted changes
app.use(bodyParser.json())

app.get('/', function (req: Request, res: Response) {
    res.send('Hello World!')
})
```

We're done. It's this easy because we're going to apply the CORS capability to all of our routes. But we can also apply CORS on a route-by-route basis and to do that, all we have to do is call a CORS middleware on each route that we want to support CORS. That looks like this.

```
const app: express.Application = express()
const address: string = "0.0.0.0:3000"

const corsOption = {
    origin: 'http://someotherdomain.com',
    optionsSuccessStatus: 200
}

app.use(cors(corsOption))
app.use(bodyParser.json())

app.get('/', function (req: Request, res: Response) {
    res.send('Hello World!')
})

app.get('/test-cors', cors(), function (req, res, next) {
    res.json({msg: 'this is CORS-enabled with a middle ware'})      You, 1 se
})
```

Where I'm calling CORS with the CORS options specifically for this test CORS route. I point this out for the opportunity to say a quick word about middleware. Middleware generically refers to any function that runs between two other functions in a progressive chain. In this route, CORS is the middleware that we want to run between request and response, and adding the parameter next allows us to use this middleware.

This knowledge of middleware and how they work might come in handy sometime. But that's all that we have for CORS. I'll see you in the next lesson.

Summary

- CORS stands for Cross Origin Resource Sharing and is an important factor in making and handling API requests
- Express provides a CORS integration that is pretty straightforward to set up
- We can use CORS as a middleware on a route by route basis

Routes to Models

We're getting close to a fully operational API! In the last lesson, we began to write some Express handlers for incoming requests, and now, we need to connect the last of the plumbing and make sure those handlers can interact with the model methods we created in the last lesson. This session walks through how to call model methods from handlers.

In the last session, we got a sneak preview of an ExpressRoute calling a model method. Now, we get to add that functionality to the app. The first thing we'll do is add a folder called handlers inside of the source as a sibling of models. In a RESTful architecture, each entity or model in the app gets its own set of RESTful routes, so it's convenient and intuitive if our file structure follows the same pattern.

We will have one handler file per model file, and these files' job is to hold all of the handlers functions for the RESTful routes associated with this model. In this handlers file, I imported express and the type and class we created for the model.

```
const index = async (_req:Request, res:Response) => {
    const weapons = await store.index()
    res.json(weapons)
}
```

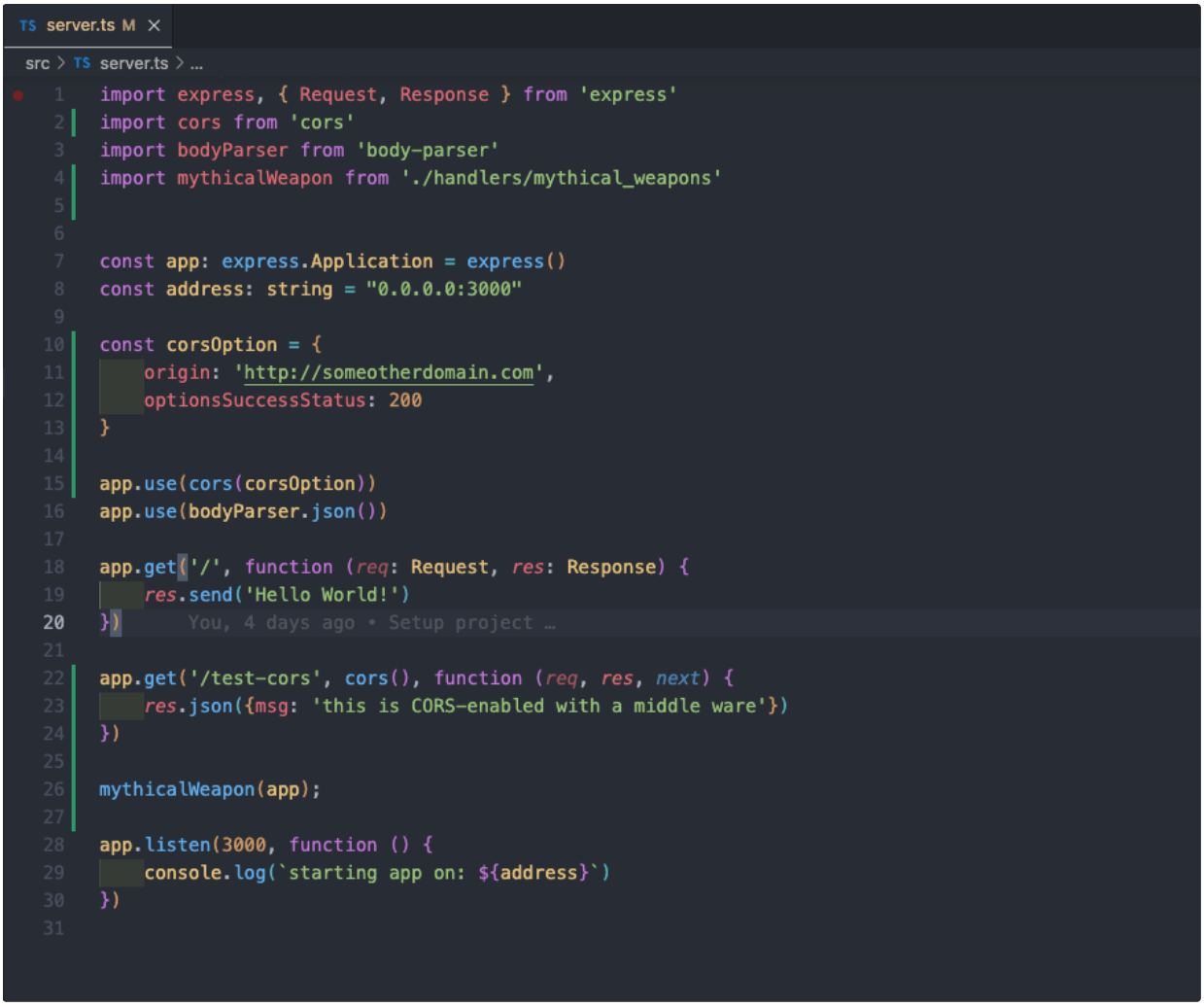
This function is the express handler function. But we need to make sure that this file has access to all of the express methods. To do that, I created a mythical weapons routes function that takes in an instance of express. Once we have that, we can call express methods that match two routes and calls the RESTful route handler to create a response.

```
const mythical_weapon_routes = (app: express.Application) => {
    app.get('/product', index)
}
```

To walk through how all of that's connected one more time; in the server file, app is the express object through which we use the express methods and map incoming requests of a certain verb to an endpoint.

```
app.get('/', function (req: Request, res: Response) {
    res.send('Hello World!')
})
```

Then we call a function to create the response. Once we have this function, all we have to do is make sure that it gets called in the server file. Here, I'll import mythical weapons routes from the handler file. Then I call the function and pass it, this instance of the express app. This function is all I need to be able to use the express routes here in my handler file.

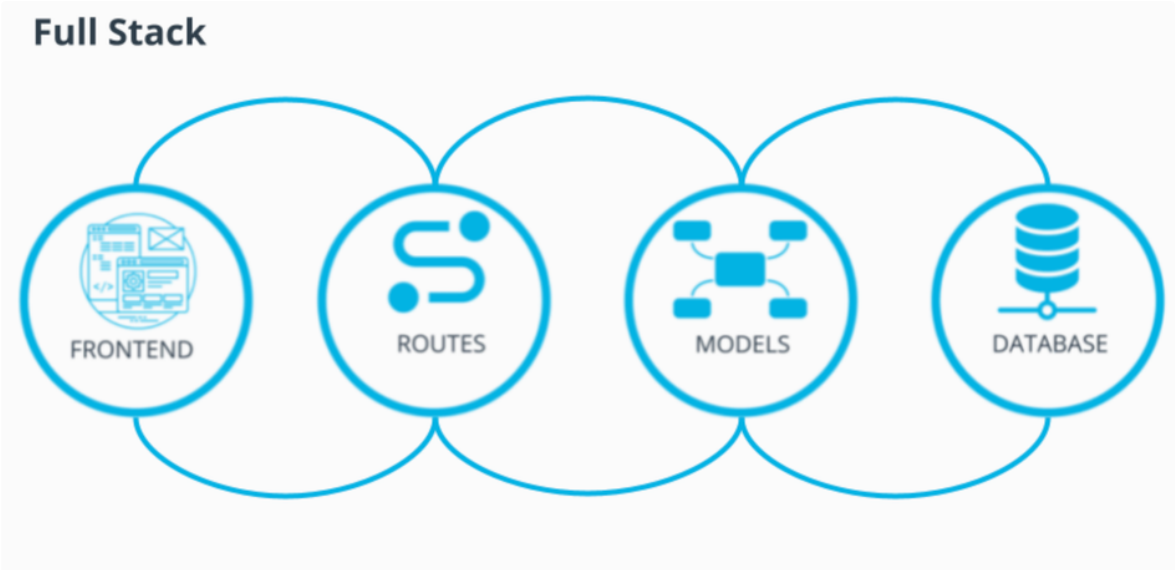


That's all for this session.

Summary

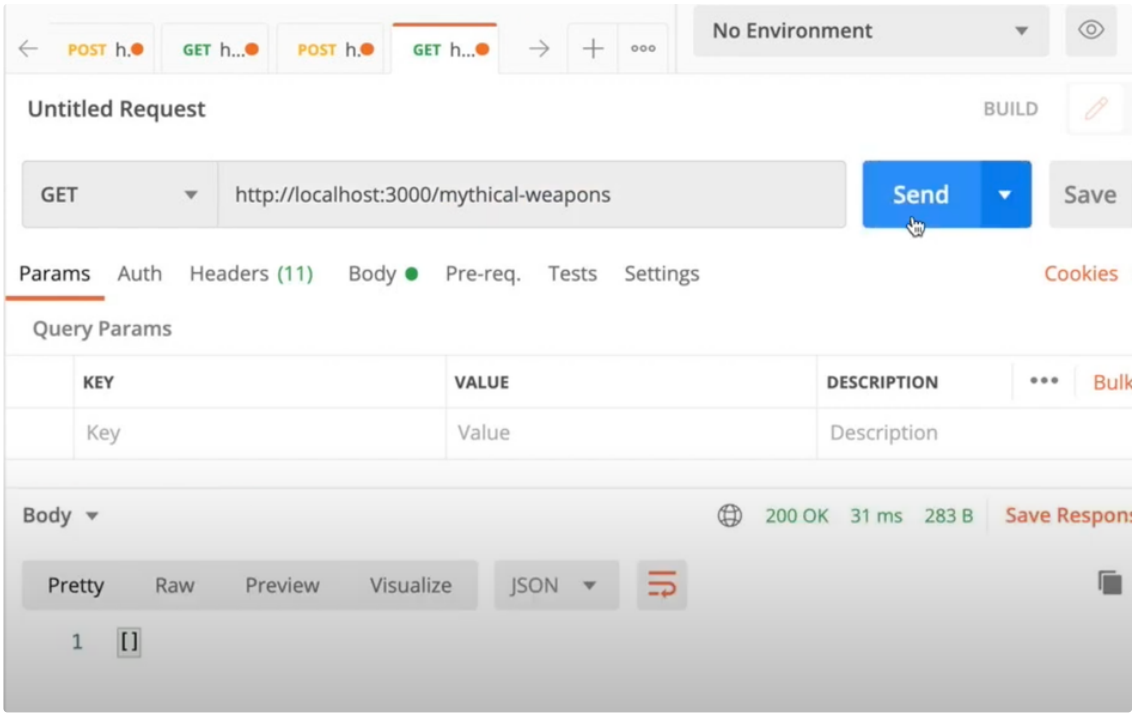
- We tie everything together by calling model methods in the handler functions

Fullstack Big Picture - CRUD to REST to HTTP Requests



This is going to be a fun session. There won't be any new content, but we're going to take one satisfying moment to step back and see everything we've learned working together as part of one seamless full-stack machine. Let's imagine a front-end application that makes a fetch request to our API.

Expressible handle the incoming route and call the associated model method. The model method contains a SQL query, which is run on the database and responds with a series of rows. Those rows are parsed by the model and passed back along the same chain to the routes. The model parses that response of rows and passes the response to the handler function. The handler function packages the result sent from the model method into an HTTP response, which is then sent as JSON to the front end. To see all of this in action, I am running the API locally.



Here, we're going to use Postman to create a GET type request to our local mythical weapons model. When I send this request, Express picks up this request and calls the correct handler method.

```
const index = async (_req:Request, res:Response) => {
    const weapons = await store.index()
    res.json(weapons)
}
```

This handler has a connection with the model that we created and calls the associated method on that model, which takes us to our model file.

```
export class MythicalWeaponStore {
    async index(): Promise<Weapon[]> {
        try {
            // @ts-ignore
            const conn = await Client.connect()
            const sql = 'SELECT * FROM mythical_weapons'
            const result = await conn.query(sql)
            conn.release()
            return result.rows
        } catch (err) {
            throw new Error(`Cannot get weapons ${err}`)
        }
    }
}
```

Where in this method, we create a SQL query and run that SQL query on the database. The database then responds with a set of rows which we parse and pass back to the handler function. The handler function then takes that response, turns it into JSON, and sends it back to the client. In this way, we've traversed the full stack.

Before finishing this lesson, I'll address one more thing. As we walked through the flow, you might have been wondering, why are there so many middlemen in this process. Why can't the Express routes in our handler files, perform crud operations on the database. Why can't we cut out some of these extra steps? It is tempting, but the separation of concerns is a good idea.

Talking to the database and handling requests are two completely separate jobs. So it's best to let them be separate. It's easy encode to take a shortcut and combine two distinct processes, but later when you need a more robust solution, untangling that mass to make them separate again, will require significant effort. One of the most common code smells I come across and one of my least favorite to deal with is debugging a function that is trying to do too many things at once. Leave your app room to grow by separating things with different jobs or purposes, if not in different files, then at least in their own functions.

That is the end of this lesson. In the next section, we'll be taking a look at security and authentication in an API.

Summary

- Express handles incoming HTTP requests to the API and the handler functions call model methods
- Model methods query the database and send the information back to the handler, which parses it into json and sends the HTTP response

Lesson Summary

In this course, we've built the client-facing sections of an API; here's a look back at what we did:

- Implemented RESTful API structure
- Created Express functions for incoming requests
- Organized Express routes into handlers
- Mapped RESTful routes to model methods
- Added endpoint tests

Glossary

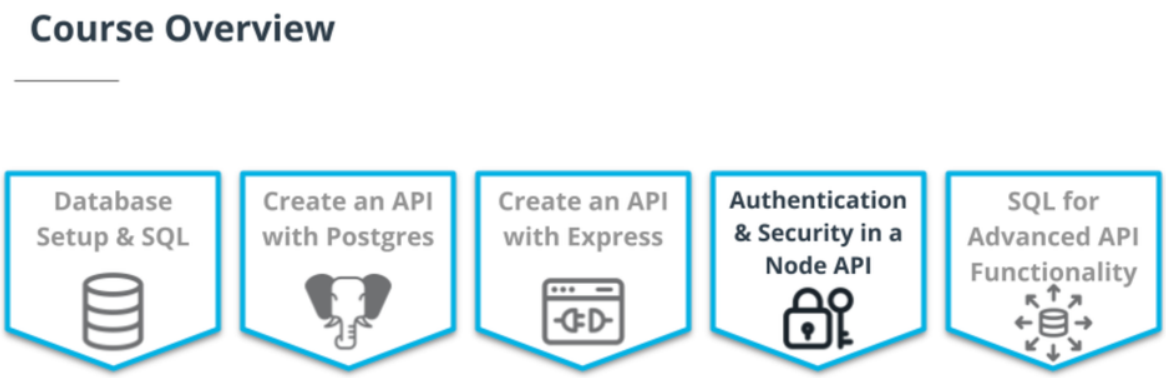
- **CORS -** Cross Origin Resource Sharing is required by browsers in order to access an API

Going Further

- For more about **REST**, here is a good article to get you started.

Lesson Introduction

Course Progress



You are here: Authentication and Security/Authorization in a Node API

What we'll do

This lesson is all about security. We need to protect the information in our database and handle things like user authentication. We'll cover the following topics

- Password hashing and salt
- Implementing the **bcrypt** library for password hashing
- Introduction to JWT's
- Implementing JWTs with the library **json-web-token**

Helpful Preparation

To get ready for this lesson, and hopefully drum up some curiosity, you can visit jwt.io to view their excellent docs and introduction to JWTs.

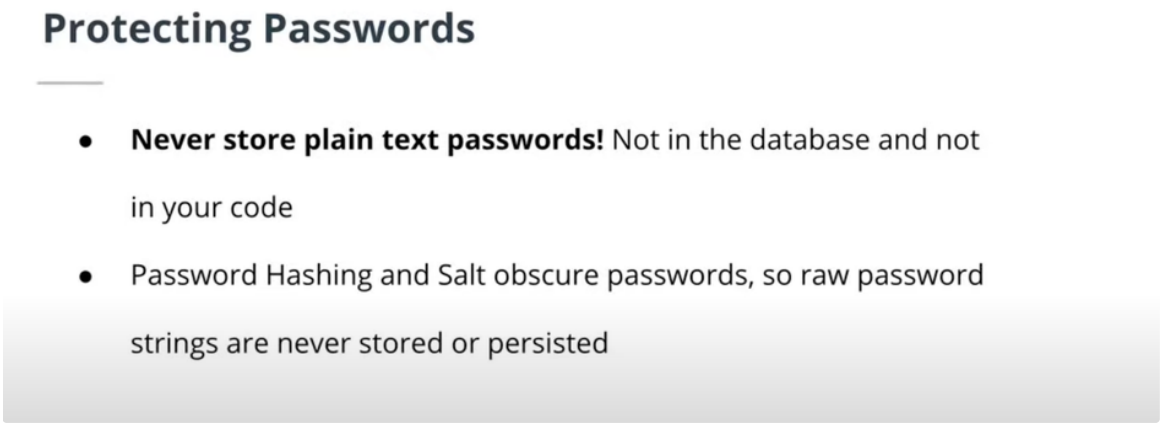
Database Security - SALT and password hashing

Database Security

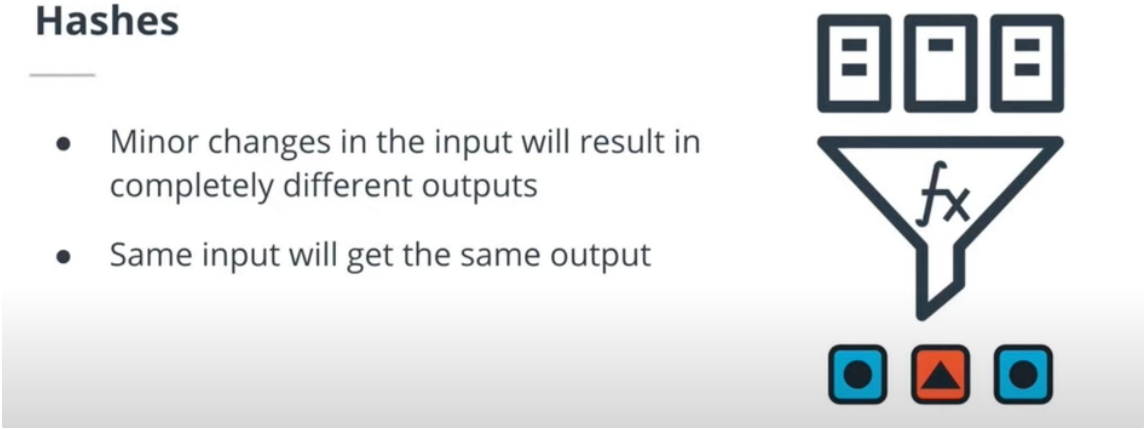
We have learned how to store data in a Postgres database, but there are a few topics we haven't covered. One of the big concepts we've side stepped up to this point is **data security**. When we're storing information like worlds, plants, or weapons, none of that information is sensitive. If someone got access to our database, they could make a mess, but the bad things they could do with that information are limited. Other tables, though, for example, a user's table, have information that needs to be protected like passwords, IDs, even emails, or credit card information - there are lots of data points that an attacker could use maliciously if they got a hold of it. This section introduces the important concepts behind **protecting passwords** in a database.

In this section, I will walk through how to securely store passwords in a database. With passwords, I have one rule, and that is never should there ever be a plain text readable password stored anywhere, not in your code and not in your database. But we do need a way to store our passwords. They just won't be in plain text. Imagine your database is like a vault. It should be hard to get into in the first place, but if someone did get in, you don't want them getting all your important notes. In our case, the passwords. Hashing and adding salt to your passwords is like writing your notes in a secret code, then storing them in the vault.

If someone did gain access to your database, it wouldn't be the end of the world because our truly sensitive information would be protected even within the database. Password hashing and salt are methods for obscuring the contents of a column in a table, often used for passwords. These are the two new terms we'll introduce in this lesson, hash and salt.



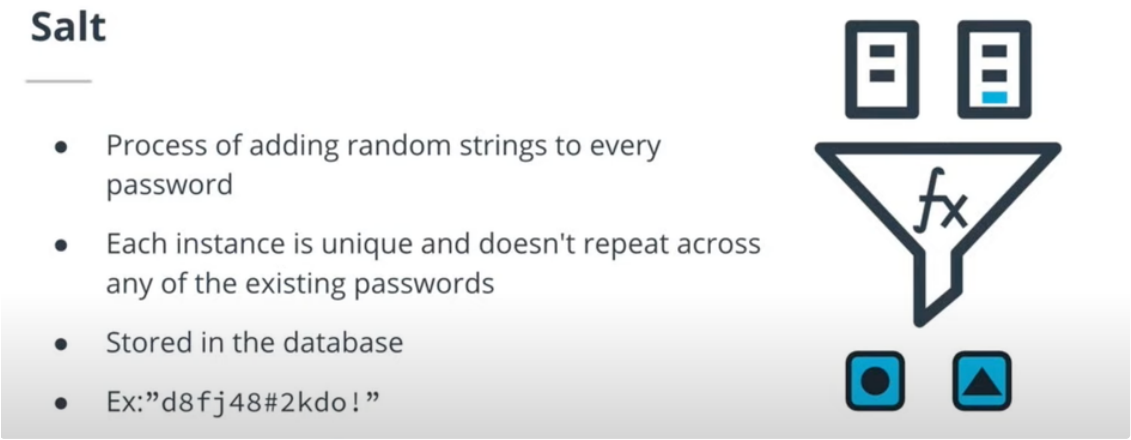
Both are part of the process for protecting passwords. Let's see what they do.



We can protect passwords by hashing them. Hashing means to run the password through a function that will spit out a sequence of seemingly random letters and numbers on the other side. A few things are important here. First, that even a small change in the string passed to this function will result in a completely different result on the other side. Second, that the same input to this function will get the same output, because for things like passwords, we need to be able to check an incoming password against the original every time a user

logs in. The last thing is that these functions only work one way. Meaning you can't put in a hashed password and get back the original. You can only put in a string and get out a hash. These are hashes. They do a good job of hiding the original password. But because the same input always returns the same output, they have a weakness to things like reverse lookups, where a hash can be compared against a dictionary of hashes and their associated passwords.

We need some other way to further obfuscate these passwords. This is where salt comes in.



Salt is an extra bit of information that you add to the original password before hashing it. With salt added to the password, the most common brute force attacks to guess your passwords won't work with this random string appended. However, it's important that you keep your salt secret, usually in an environment variable or your database. Password hashing and salt and pepper are really interesting topics, but to go into them further is the job for a security or cryptography course.

In the next lesson, we'll be installing bcrypt, a very common library for password hashing with salt in our Node API.

Summary

- A hashed password has been run through a function that generates a long encrypted string from the original password.
- The same password run through the same hash function will generate the same response, this is how we can match passwords when users log in
- Simply hashing passwords though isn't enough, adding **Salt**, an extra string sequence to the beginning or end of a password before hashing it makes it much harder for attackers to decrypt passwords
- **Bcrypt** is a very common library for password hashing in web apps

Password hash creation and validation with Bcrypt

Installing Bcrypt

In the last section, I introduced you to the concepts of password hashing and salt. Bcrypt is a very common library for implementing password encryption, and in this section, I'm going to walk you through password protection and validation in a Node API.

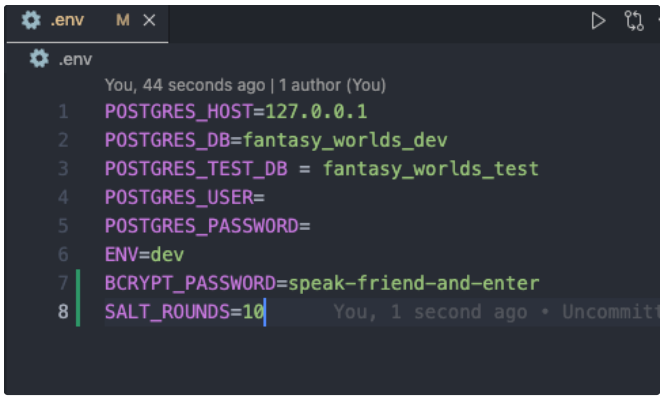
Steps to install Bcrypt

1. Add the dependency: npm install bcrypt
2. Import bcrypt into the user model: import bcrypt from 'bcrypt'
3. Create the necessary environment variables: BCRYPT_PASSWORD = your-secret-password SALT_ROUNDS = 10
4. Use the hash method inside the create method to hash, salt, and pepper the password and save the resulting value to the password_digest column on the user's table:
   const hash = bcrypt.hashSync(u.password + pepper, ParseInt(saltRounds));

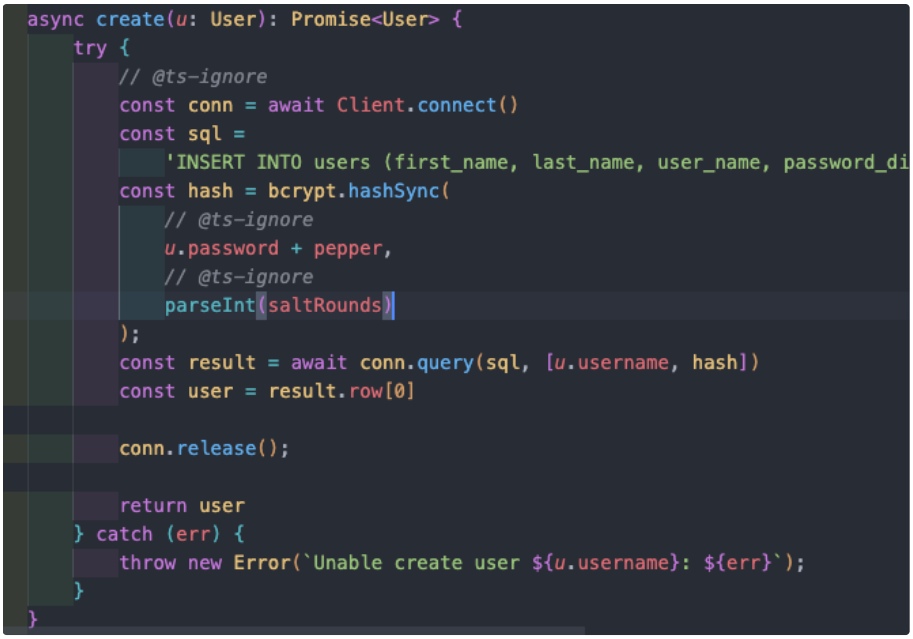
Hashing passwords at user account creation

In this section, I'll show how to add bcrypt password hashing when a user is added to the database.

Once BCRYPT is installed, we need to implement password hashing. This step should happen in the models create method before anything is saved to the database. That way, we avoid ever storing plain text passwords in the database. In this section, I'll walk you through how we're going to implement BCRYPT in this project.



I'm going to use SALT like from the last section, as well as an additional peppering step for added security. First, what we need to do is add two new environment variables to the project. In this version of BCRYPT, SALT is a number of times the password will be hashed. I'm saving that number as an environment variable called SALT ROUNDS. I'm also saving an extra string. We'll use this later in the peppering step.



This is the function will use to hash the password from the user. Here, the user's password is being concatenated with pepper, the extra string that I added in the environment variable. We also pass in the environment variable for saltRounds. The constant hash now contains the hashed password. It isn't that no one could ever hack this password, they definitely could. But what we've done by adding these protective measures is ensured that no one can take a password stolen from another service and have it work here in our application as well. The hashed password will also be long enough that cracking it with a brute force attack would take a long time, too long to be worthwhile. We save the hashed password into the database as the password digest. We save the hashed password into the database in the password digest column and we have done a pretty good job at keeping things secure.

Summary

Here's an example of the bcrypt hashing method with salt and pepper:

```
1   const hash = bcrypt.hashSync(
2      u.password + pepper,
3      parseInt(saltRounds)
4   );
```

Validating passwords at user sign in

In this section I'll show how to add a custom authentication route to our API and how to validate hashed passwords at sign in.

```
async authenticate(username: string, password: string): Promise<User | null> {
    try {
        const sql = 'SELECT password_digest FROM users WHERE username=($1)';
        const conn = await Client.connect();

        const result = await conn.query(sql, [username])

        console.log(password+pepper)

        if (result.rows.length) {
            const user = result.rows[0]

            console.log(user)

            if (bcrypt.compareSync(password + pepper, user.password_digest)) {
                return user;
            }
        }

        return null
```

For sign-in functionality, we're going to do one new cool thing. So far, all of our models have only had methods that translate directly to the RESTful routes we learned. But sign-in doesn't really fit a create, show, edit or delete action. So we're going to create a custom method called authenticate. When a user signs in, they will give us their password. To see if it matches the password in the database, we could hash the password in the same way we hash the password the first time in the create method. But bcrypt actually doesn't work that way, and we will use their provided method, compare, to check the incoming password concatenated with pepper against the contents of the password digest column.

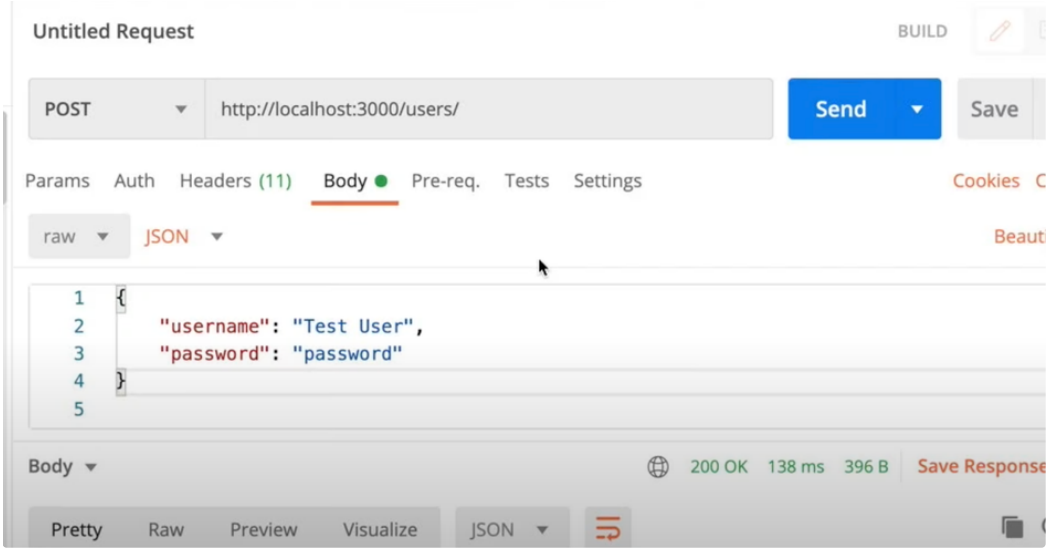
```
if (bcrypt.compareSync(password + pepper, user.password_digest)) {
    return user;
}
```

First we look up the user by username in the database. Then we take the password digest from that user and pass it in as the second argument to the compare function. If there's a match between the incoming password plus pepper and the password digest column on the user, we will return the entire user object.

One last consideration is that we need to check if a user account exists with the requested username. Because sometimes people get mixed up between sign-up and sign-in forms, or they forget what username was used at account creation. We want to be able to tell them if they're trying to log into an account that doesn't exist.

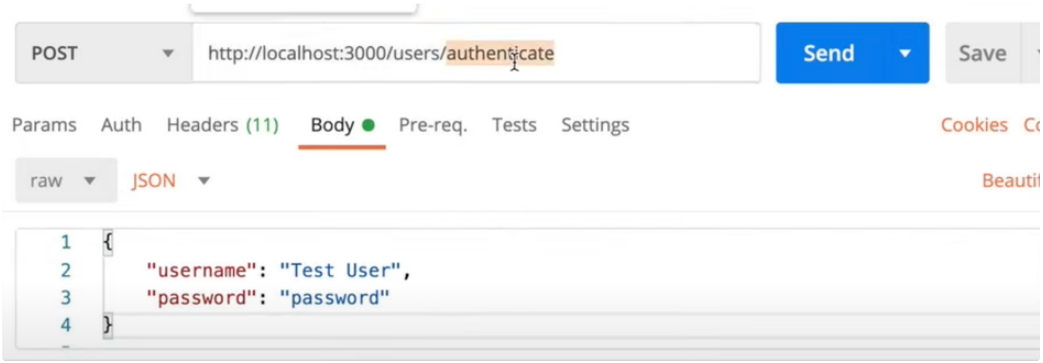
```
    if (result.rows.length) {
        const user = result.rows[0]

        console.log(user)
```

To do this, we add an if statement to check that we get a response from our SQL query. If we don't, the whole function will return null and we can pass an error along to the user. Now that we have all this logic set up, let's see it in action. Here I'm running my API locally and accessing it with Postman.



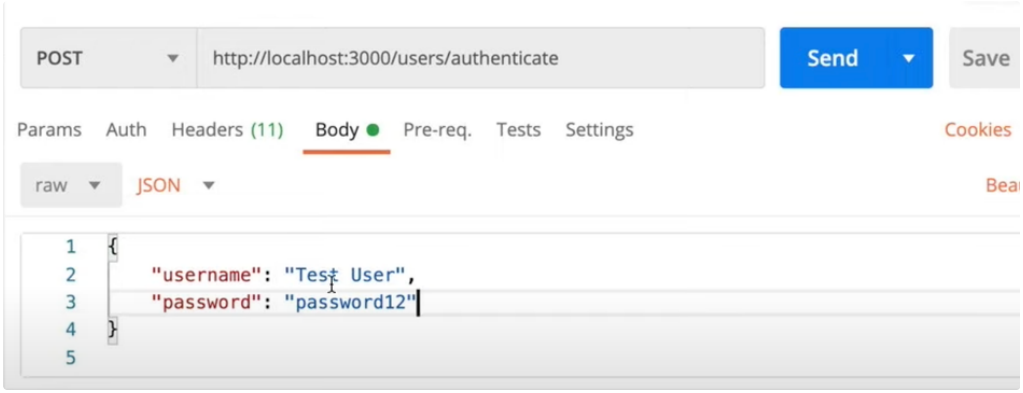
When I send a request to create a new user, I see that the response contains the user information as well as a password, digest.

Then if I go to the authenticate route instead and pass in the same username and password.



We see a successful response with the password digest.

But now if I change the password so that it doesn't match what we have for this user and send the request.



You can see that I get back null, meaning that my password didn't match what was in the password digest field.

We have working authentication for our application using bcrypt, password hashing, salt, and pepper.

Summary

Here is an example of the bcrypt compare method that checks an incoming password for a match against the hashed password stored in the database

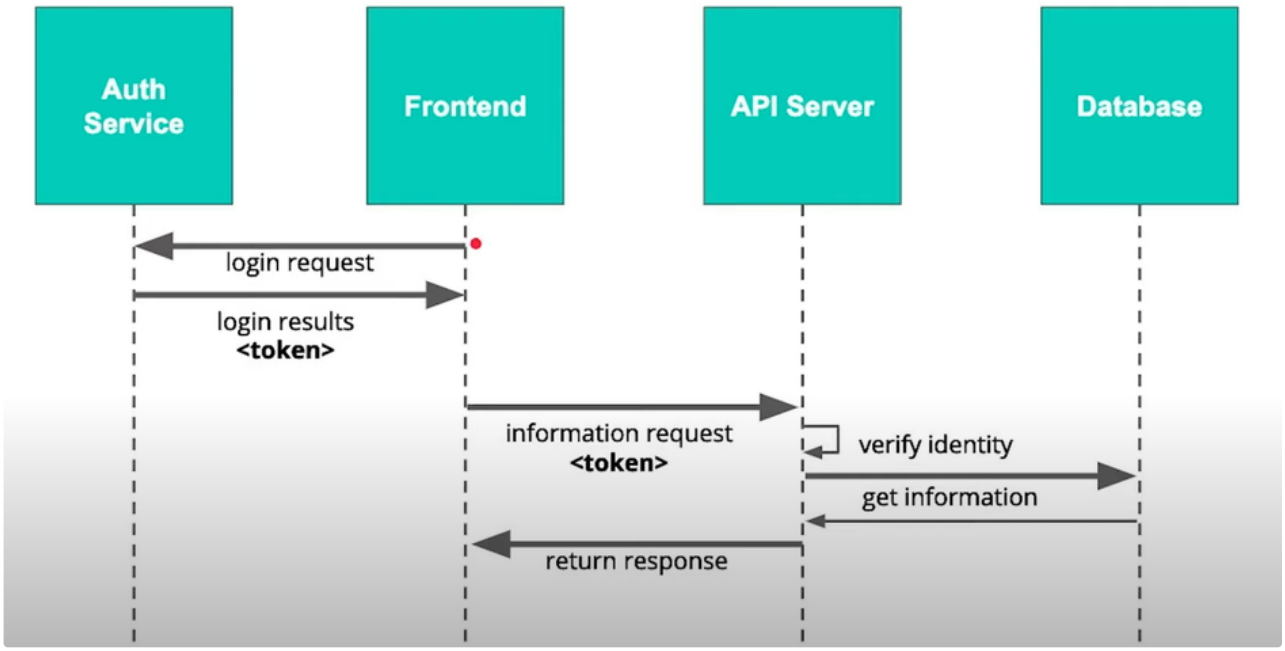
```
bcrypt.compareSync(password+pepper, user.password_digest)
```

Introduction to JSON Web Tokens

JSON Web Tokens (JWTs)

In the next three sections, you'll be introduced to JSON web tokens. JWTs are the most common means for authenticating users in decoupled (meaning you have a separate front and back end) web applications. They are secure digital tokens that can be passed between your front-end and back-end applications to authenticate users and even store important user information. We will be integrating a JWT authentication flow into this API.

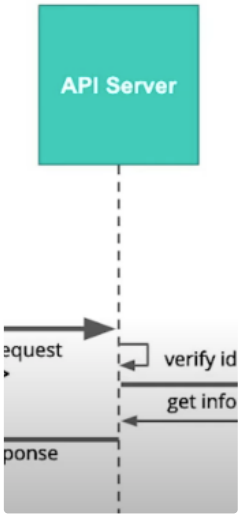
Let's take a closer look at our sequence diagram for a third-party authentication flow.



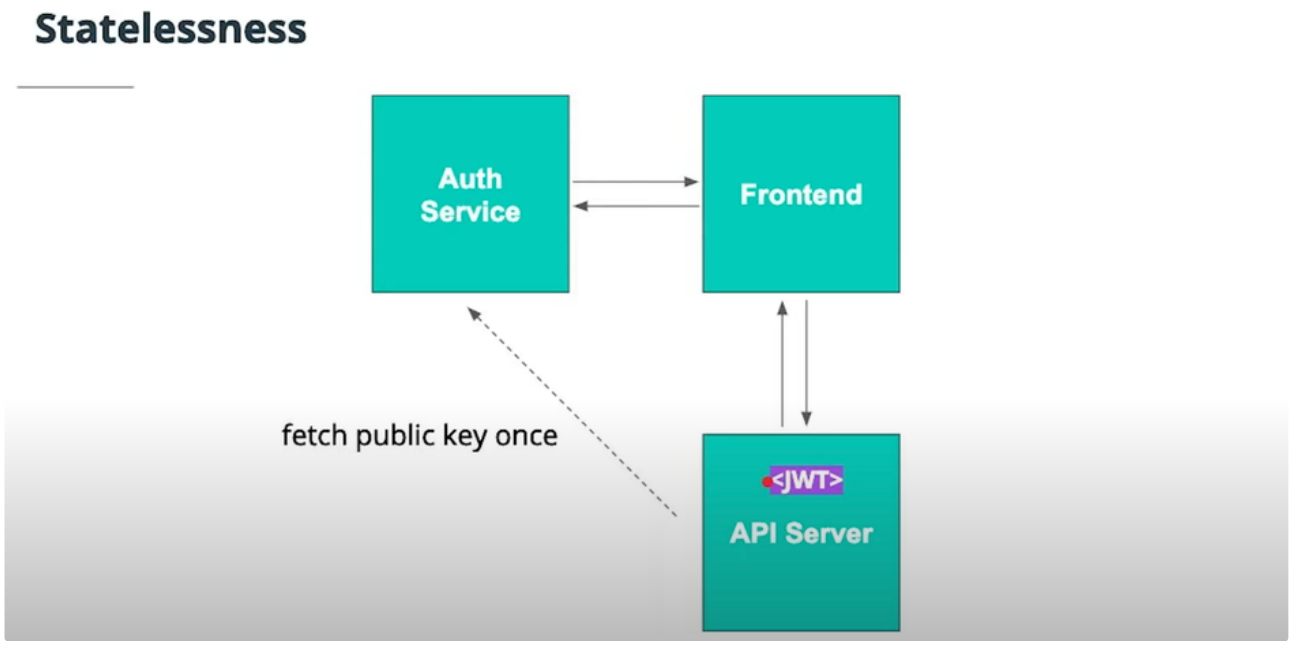
We recall our user will submit some kind of information from our front end to an authentication service. If the login attempt is successful, the authentication service will return a successful result along with something we called a token.

This token would be used in subsequent requests to send some kind of credential to whatever service is requesting it to verify the identity on that service. Traditionally, servers validated authentication using something called a session table. This was a literal table within our database that included a session ID and the user ID pair. On every request, the client would provide that session ID to the server. The server would make a request to the database to see if that session ID was still valid, and if it was, it would continue. But using a microservices architecture, we might have one or hundreds of services who need to maintain that state across entire systems that might be located in different parts of the world.

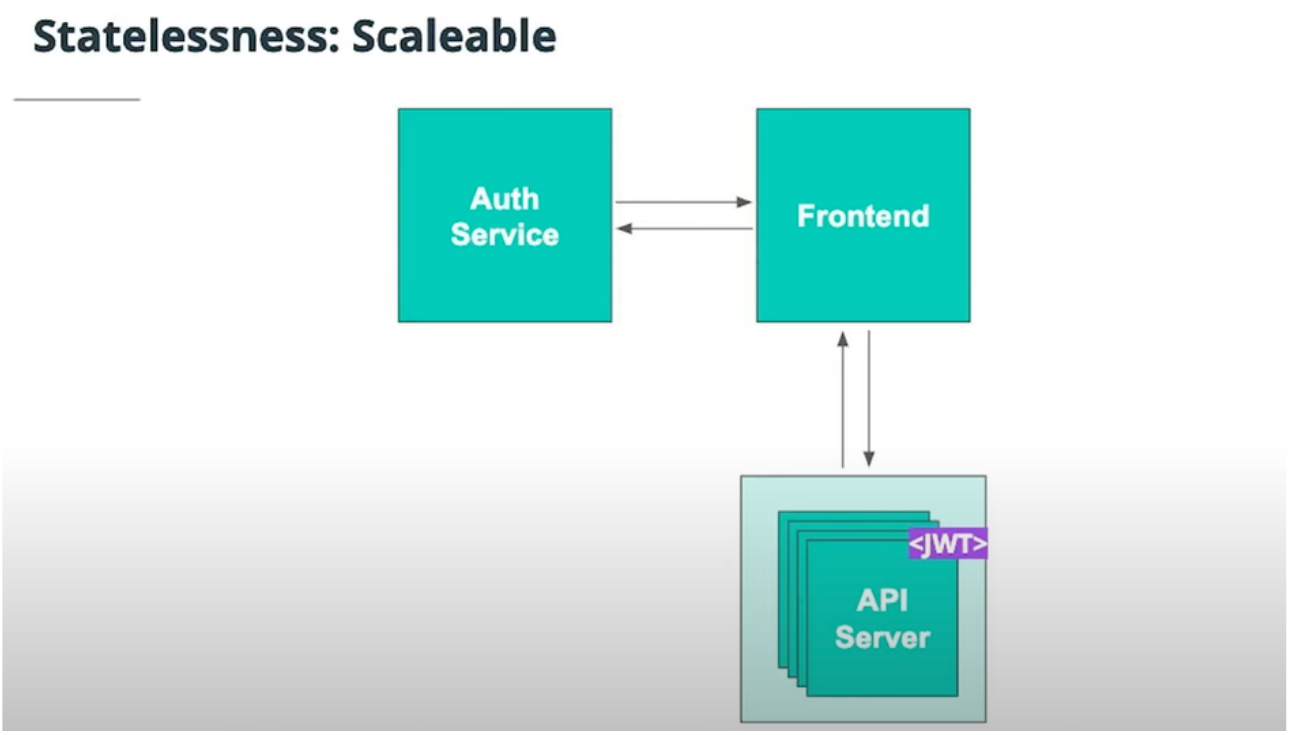
That becomes a problem to maintain and manage. There's a certain amounts of latency and time that's involved in making that request to check a session, and sessions might change across different parts of the stack, and it takes time for that change to propagate. Instead, we would like something that is stateless.



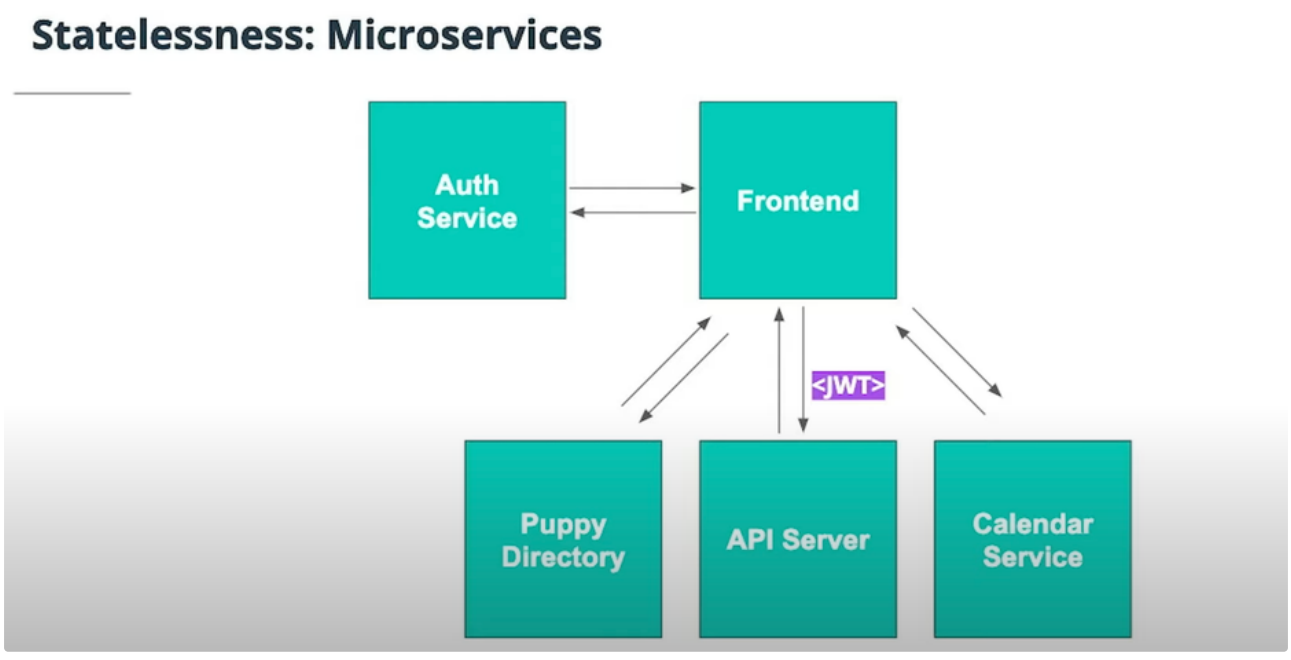
That means our server just knows that this token is valid and works.



JSON web tokens are intrinsically stateless. When they're sent to a front end and then to a server, that server only has to fetch something called a public key one time from the authentication service. This authentication key will then be stored within the API server, allowing us to verify that this JWT is indeed valid and we can trust who it is.



Statelessness also solves the problem of scalability. Let's say we have our API server that starts to have a tremendous amount of demand. In this case, we'll be spinning up multiple instances over the same service. Now, our JWT could be hitting any one of these servers within the stack, and since it's stateless, each of those servers can be confident in the identity provided.



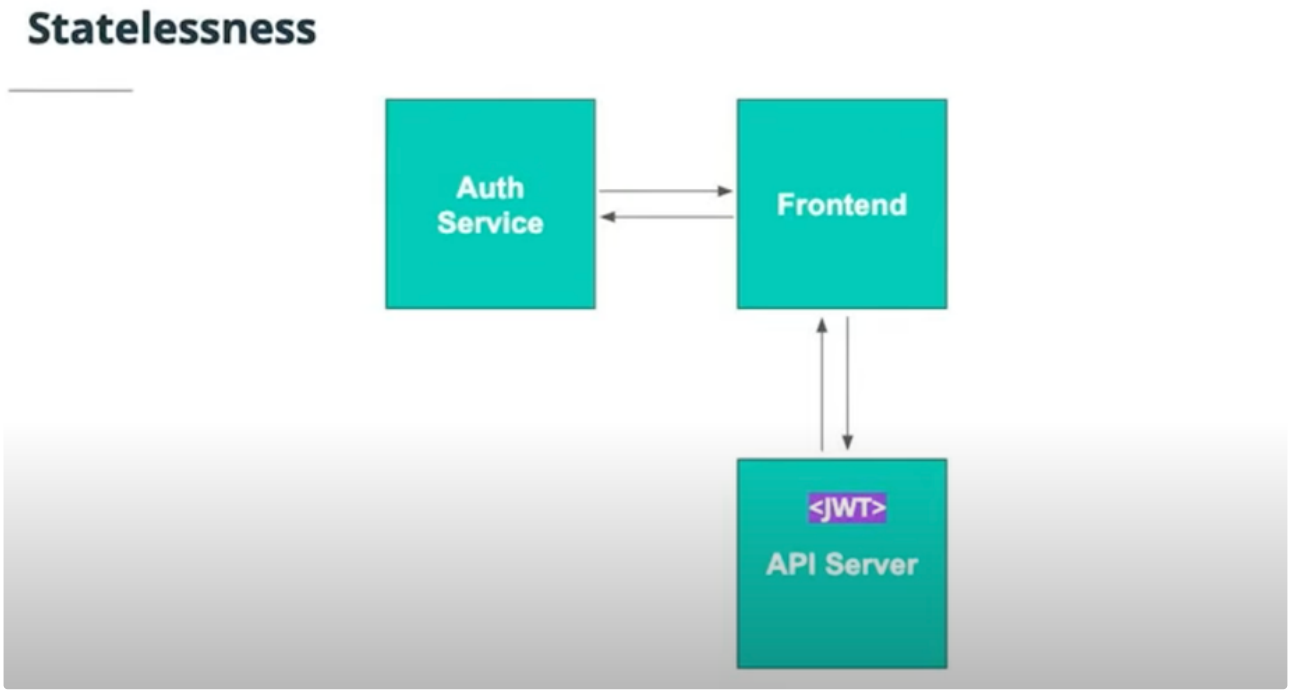
As we discussed, statelessness works wonders in microservices architecture. Now, no matter where our JWT ends up, each of our services can be confident in the authentication provided.

Storing Data in JWTs

JWT - Data Structure

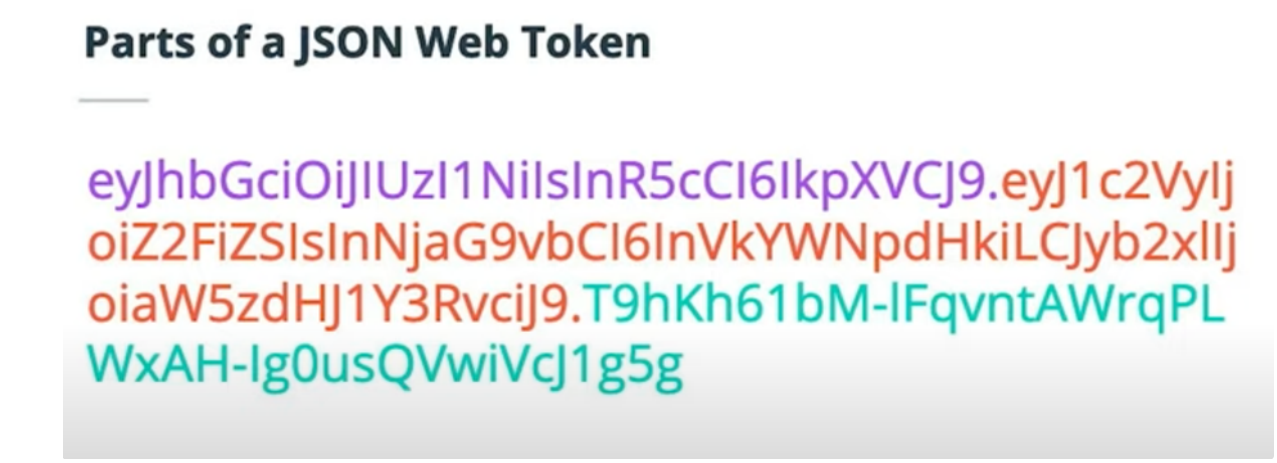
Parts of a JSON Web Token

Now that we understand the basic flow of how a JWT moves from an authentication service upon a successful login to an actual server that will use it during a request to verify identity.

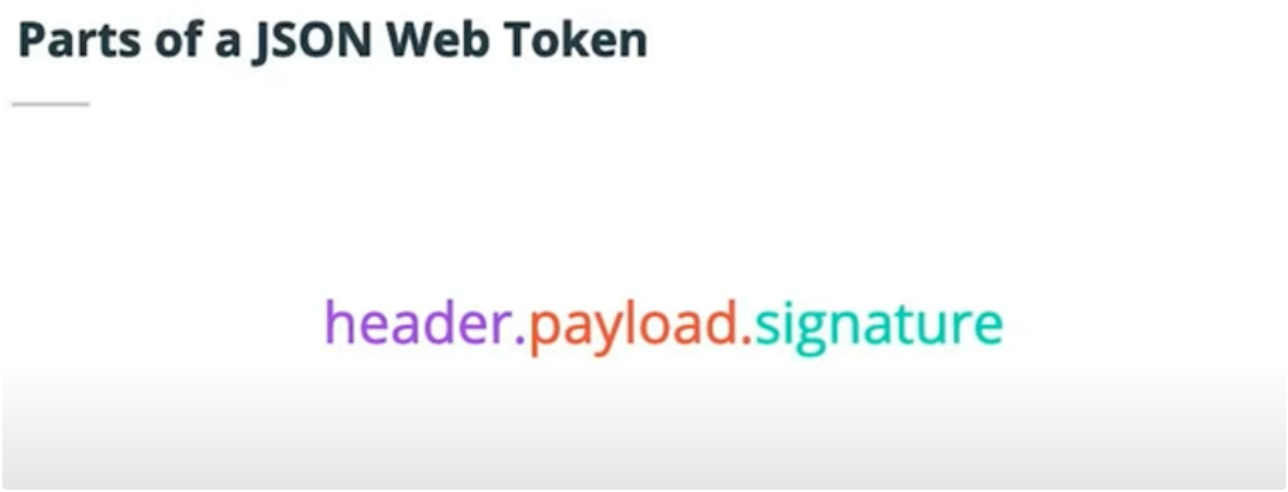


we'd like to answer the questions of how we know who is actually making the request and we'd like to know how can we trust that that information is indeed valid.

In its raw form, a JSON Web Token or JWT is basically just a string. Looking at it, you don't get a lot of information just natively. It's jumbled-up mess that just looks like a bunch of random letters and numbers but hidden within is actually some pretty intuitive structure.

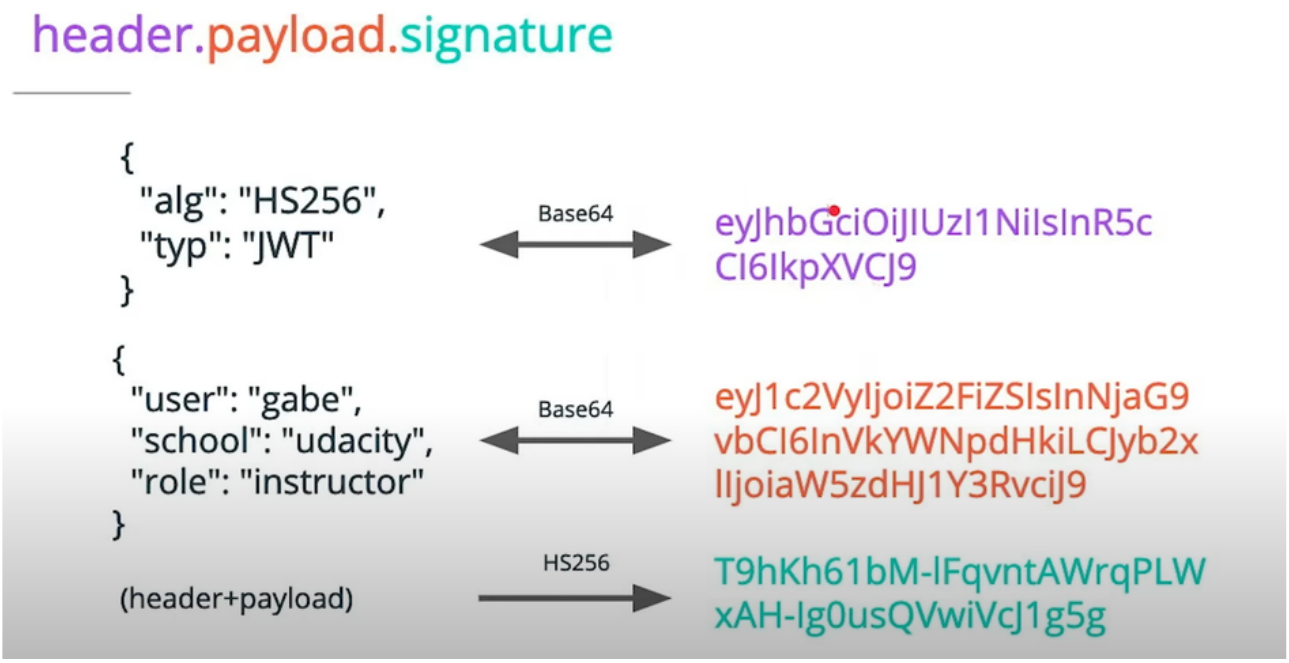


It's broken up into three main parts, a header, payload, and signature.

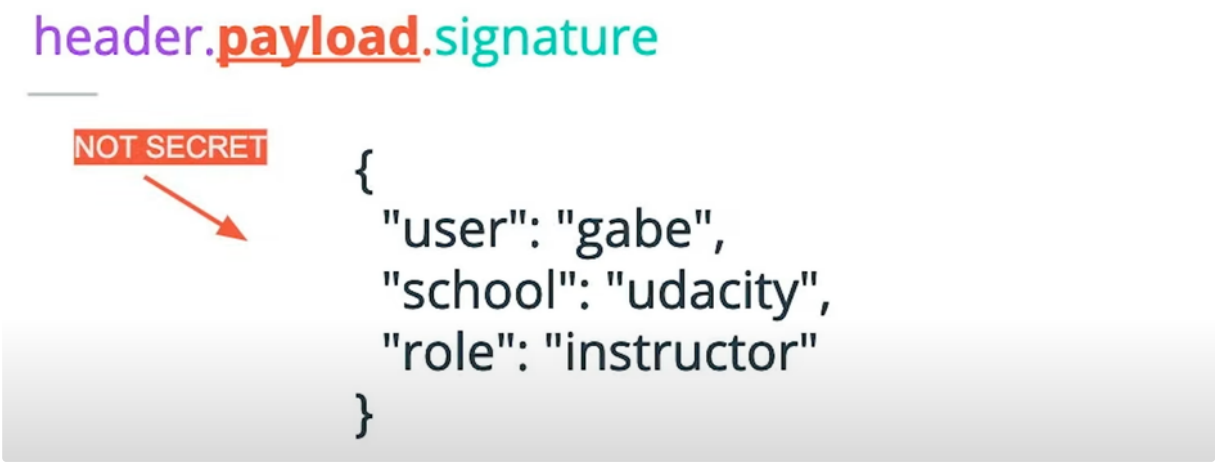


These three parts work together to ensure that the information within the JWT is consistent and that we can validate that that information has not been changed.

Including Data in Our JWT Payload



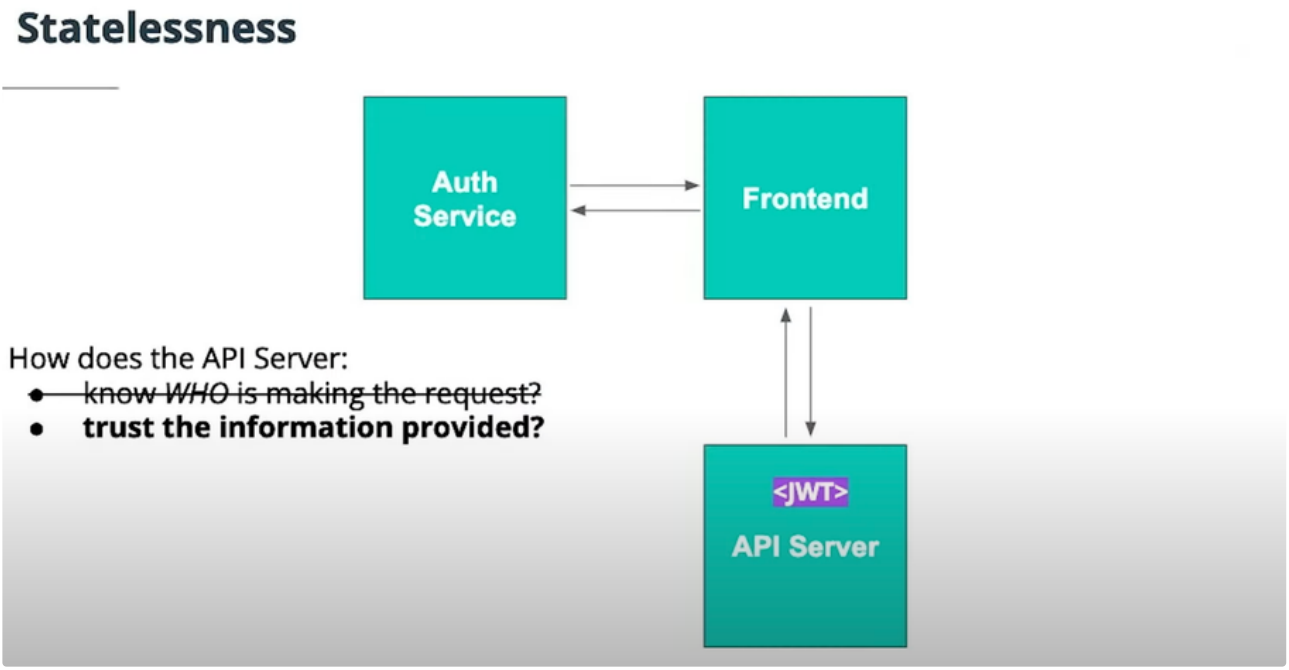
Centrally, the jumbled up strings contain information using a very simple algorithm called base-64 encoding. This is a two-way transformation that goes from some string or some text to a jumbled up looking text. What's important for us is that these three parts of the JWT do the work together to answer our questions of who is making the request and do we trust that that information is indeed valid.



Let's take a close look at each of these parts individually. We'll start with the payload. After the base-64 decoding, we're left with a user object. This includes information like our username, our school and the role we play within that school. Now it's important to remember that this information is not necessarily secret. Since the JWT base-64 encoding can be easily decoded without any additional information, this data is easily accessible by anyone who has the JWT. For that reason, you should never store sensitive information like passwords, or things like social security numbers, or even phone numbers within this data object.

Ultimately, this is what will be answering the question of who is making the request. We'll use either our username or more commonly a user ID within this payload on our server to fulfill certain actions for that particular user.

Most commonly, the header includes something like an algorithm such as H moksha 256, which will be using in just a second.



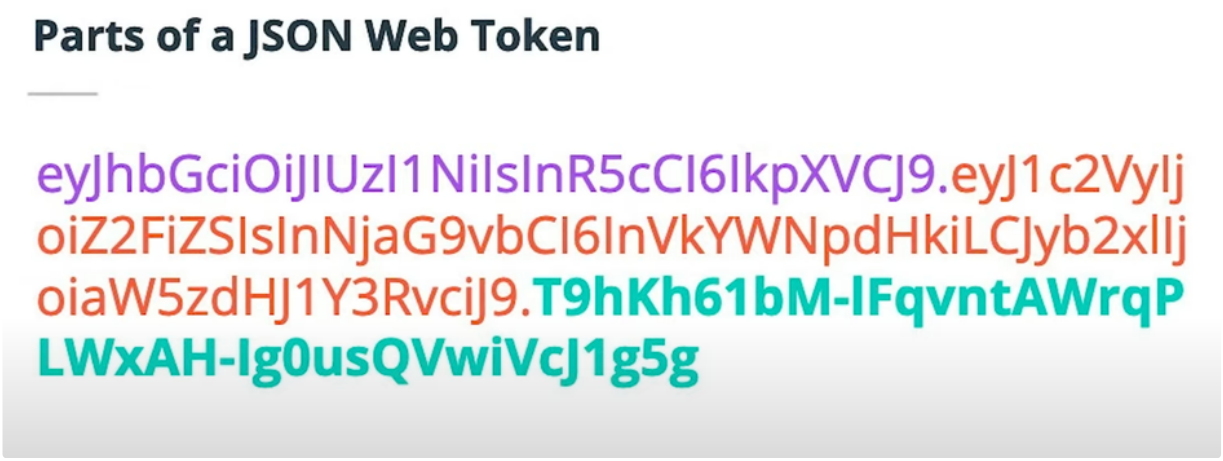
By decoding our payload we answer that question of who. But we still have the question of do we trust this information. The information within the payload ultimately answers our question of who is making the request. But since the base-64 encoding scheme is so easy to use, anyone can just create their own JWT tokens. You can create a JSON object, pass it in to the base-64 encoding, and send that token to our server. We now need to answer the question of if that JWT was indeed generated by a system that we trust and that the JWT itself is containing the authentic identity of the individual making that request.

Validating JWTs

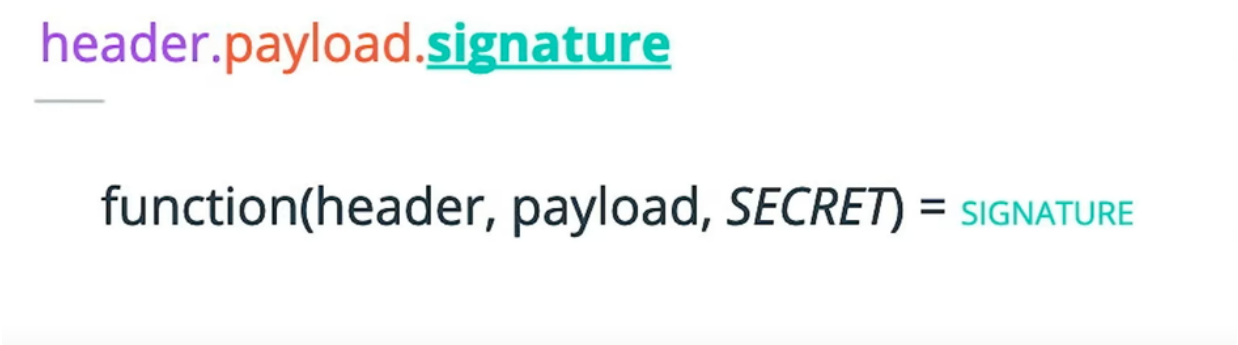
JWT - Validation

Validating JWT Authenticity

The information contained within the payload answers the question of who is making the request. But since we've learned that the base-64 algorithm just jumbles up text in a standard way, anyone can produce a JWT like Token very easily. We now need to understand how we can trust that this JWT is indeed authentic and has not been tampered with.



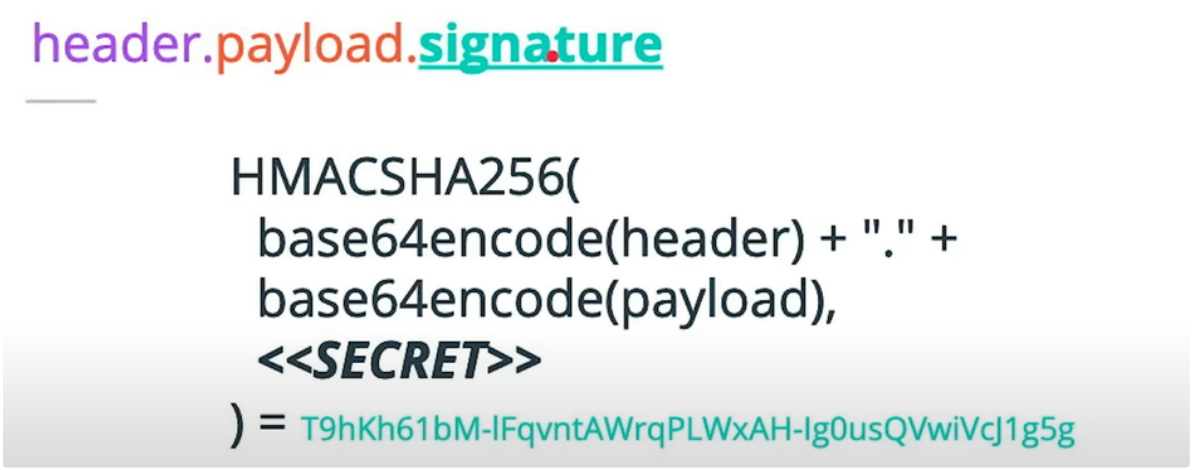
This is where the last part of the JWT will do the work, this is called the signature.



The goal of our signature is to verify that the information within the JWT has not been tampered with and came from a trusted source. To achieve this goal, we really need a function that will output a signature that depends on our header, our payload, and something we'll be calling a secret. A secret is essentially just a string that we store on our authentication service, and on this server that we'll be validating the JWT. If the secret is not known by a third party, they cannot sign the information within their payload or header. If the payload or header changes within a JWC signed by our authentication service, but the secret remains the same, our signature will still change.

Therefore, if a JWT that is signed on are Auth service does not contain the same signature when it assigned on our consuming API server, we know that that data has been tampered with in transit.

Thinking back to the first part of our JWT or the header, we included something called an algorithm HMACSHA256.



In practice, this is a commonly used algorithm to perform this method of signing. Essentially, we take our header and our payload base-64 encode them both, concatenate them with a dot, pass in our secret to generate some kind of signature string output that is only valid for the secret payload and header provided. The same algorithm is used to validate the JWT on the other side. We take the header and payload from our JWT token, we use that same secret, and we compare the string signature outputted from this algorithm.

Additional Resources:

- JWT.io a useful guide and list of popular JSON Web Token implementations.
- Base64 Encoding
- HMAC keyed-hash message authentication code

Authentication with JWTs

Now that you have a good sense of what JWTs are and what they can do, we are back to implementing JWT authentication in a Node API. The first thing we need to do is set up the tools we'll need.

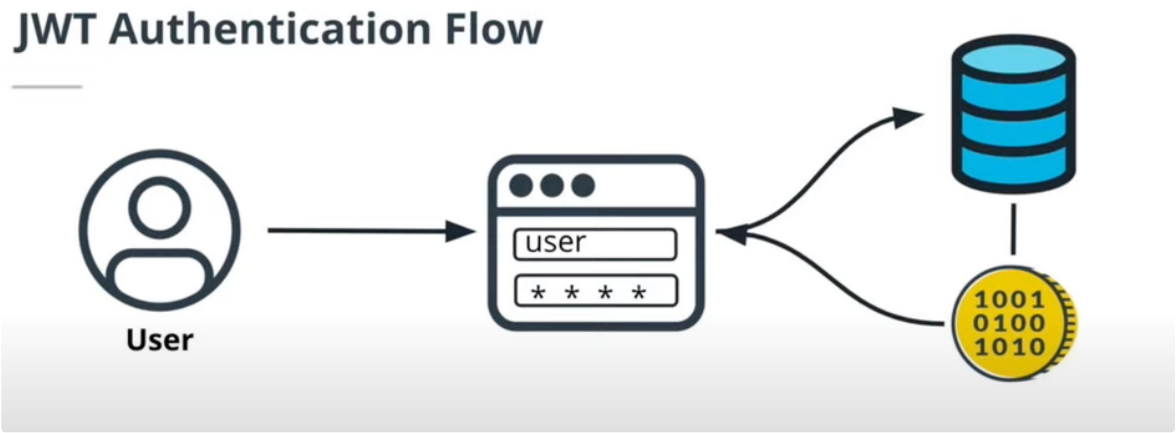
Add the NPM library JSON web token

1. Install the dependency: npm install jsonwebtoken
2. Import the library: import jwt from 'jsonwebtoken'
3. Create a token jwt.sign()
4. Check a token jwt.verify()

Create JWT at user sign up

In this section, I'll go over how to create a token using the jsonwebtoken library and add that step to the user creation flow.

Now that you're familiar with what JWTs are, let's talk about how they fit into the authentication and authorization flow of our app. A user is going to sign up or sign in to the app. Once they successfully create an account or authenticate, the API is going to give them a JWT. The client then stores that JWT, and until that token expires, is revoked, or is removed from the client, the presence of that token means that the user is signed in.



Anytime a user requests a web page or a resource that requires authorization, they will pass the token with the HTTP request. The API will check the token and send back the requested information if it is valid or an error if the token isn't valid. What we need to do first is write the logic to create a token and pass it back to the client after a successful registration or sign-in.

The JSON Web Token library gives us a method called sign that creates a token. Sign takes two arguments, an object of the information we want to store in the token, and a string to sign the token with. The string to sign the token should be kept secret.

```
const create = async (_req: Request, res: Response) => {
    const user: User = {
        username: _req.body.username,
        password: _req.body.password,
        id: 0,
        firstname: '',
        lastname: ''
    }
    try {
        const newUser = await store.create(user)
        var token = jwt.sign({ user: newUser }, process.env.TOKEN_SECRET);
        res.json(newUser)
    } catch(err) {
        res.status(400)
        res.json(err)
    }
}
```

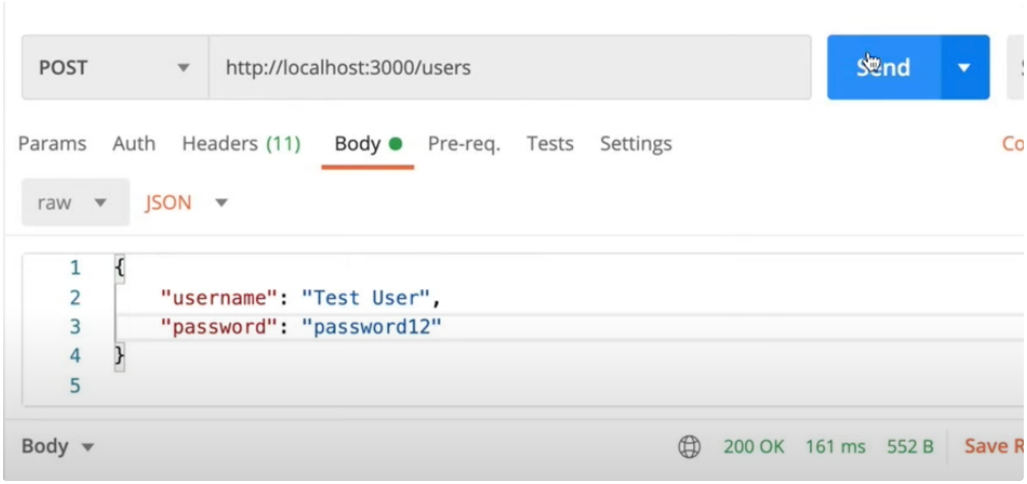
I've created it as a new entry in my environment variables file.

```
POSTGRES_HOST=127.0.0.1
POSTGRES_DB=fantasy_worlds_dev
POSTGRES_TEST_DB = fantasy_worlds_test
POSTGRES_USER=magical_user
POSTGRES_PASSWORD=password123
ENV=dev
BCRYPT_PASSWORD=speak-friend-and-enter
SALT_ROUNDS=10
TOKEN_SECRET=alohomora123!
```

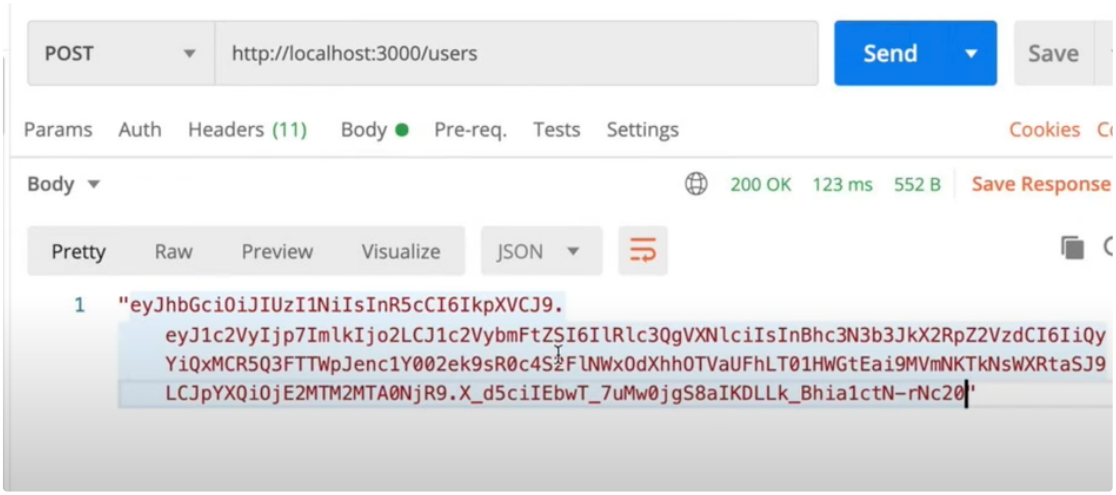
Now that we're creating this token after a new user is created, instead of passing back the new user as JSON, we're going to pass back the token so that the client can store the token and use it for future HTTP requests.

```
try {
    const newUser = await store.create(user)
    var token = jwt.sign({ user: newUser }, process.env.TOKEN_SECRET);
    res.json(token)
```

Let's see what this looks like in action. Here in Postman with my API running locally, I'm going to make request to create a new user.



We can see in the response that I get back a JWT.



Now, the really cool thing, if I take this JWT and decode it here in the JWT website.

You can see that I have stored the information about my user successfully. This is super exciting because now we have a token that can be stored on the front end and used for future authorizations with our API.

Add JWT validation to an endpoint

After the last section, users have tokens after they sign in or sign up. We can start requiring tokens in order to access certain pages. Let's say that anyone in the app can see the list of mythical weapons, but only logged in users should be able to create new entries. To write this requirement into our logic, we need to use the verify method from the JSON Web Token library.

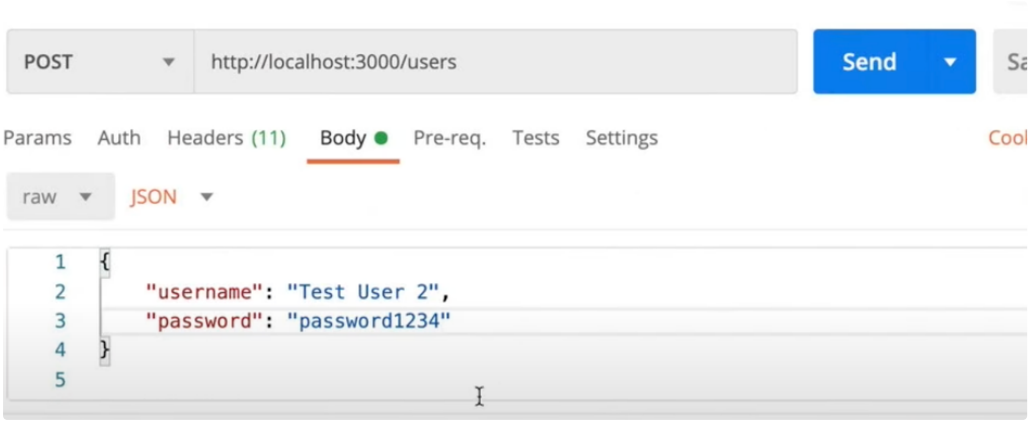
```
const create = async (_req: Request, res: Response) => {
    const weapon: Weapon = {
        name: _req.body.name,
        type: _req.body.type,
        weight: _req.body.weight,
    }

    // @ts-ignore
    jwt.verify(_req.body.token, process.env.TOKEN_SECRET)

    try {
        const newWeapon = await store.create(weapon)
        res.json(newWeapon)
    } catch (err) {
        res.status(400)
        res.json(err)
    }
}
```

The verify method takes two arguments. First, the token that is passed with this request, and the token secret which has to be the same as the token secret that I used to create this token in the first place. The verify method will return true if the token is valid, and false if it is not. We will wrap this in its own try and catch.

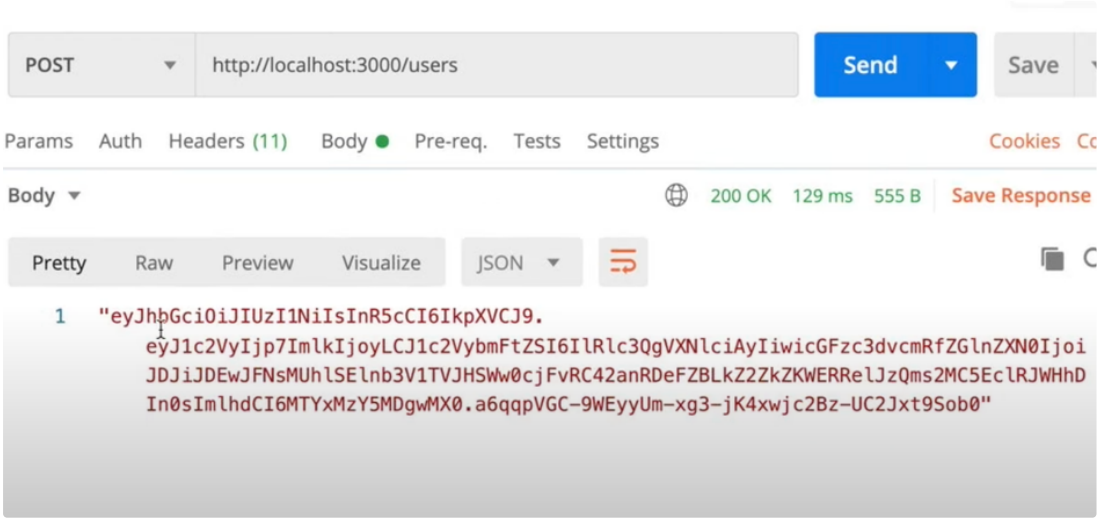
```
try{
    // @ts-ignore
    jwt.verify(_req.body.token, process.env.TOKEN_SECRET)
} catch(err) {
    res.status(401)
    res.json(`Invalid token ${err}`)
    return
}
```

If the verify method fails, we want to return a 401, which is the status code for an invalid authentication. We can also send a message with that status code. We make sure to add a return, and this logic should be good to go. Let's take a look at this in Postman.

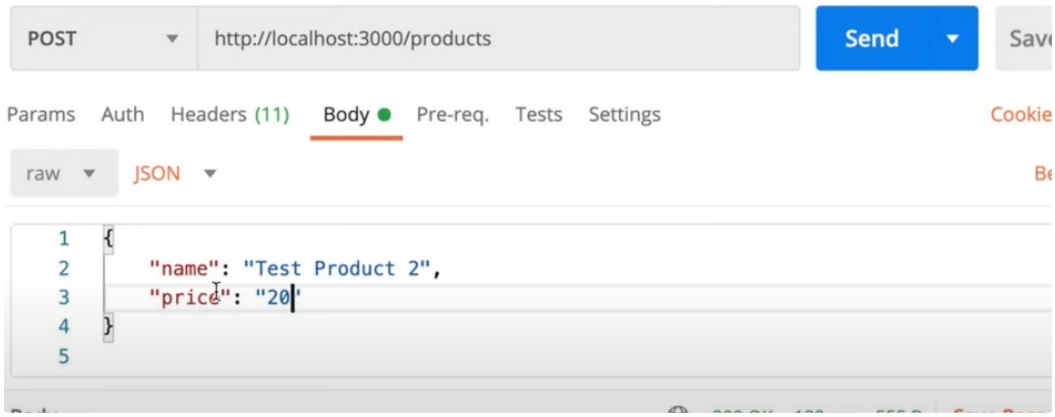
If I create a new user.



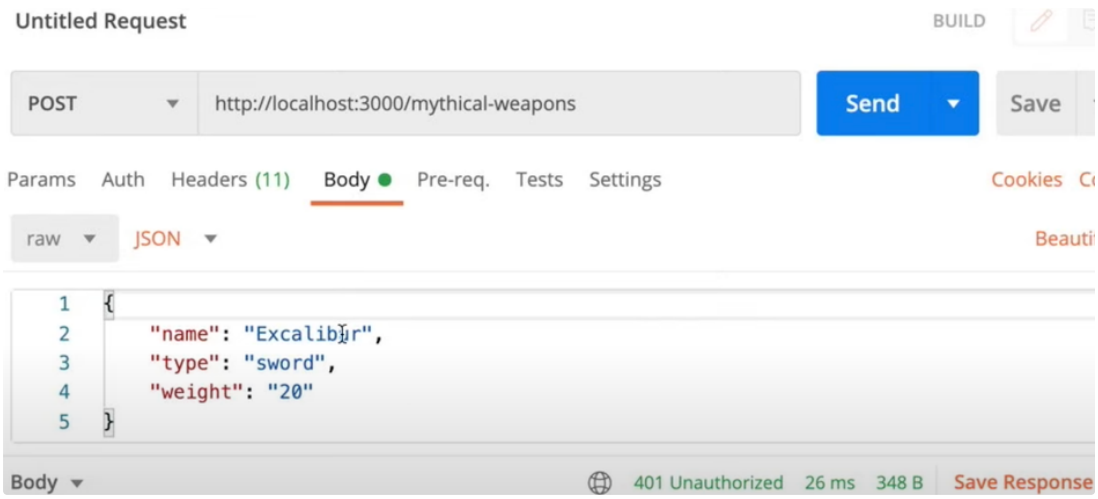
I get a response back that is a token.



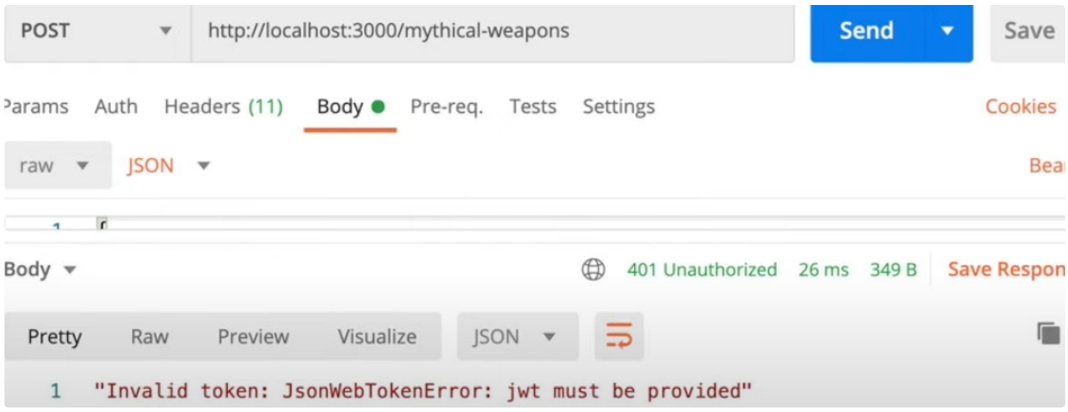
I'm going to save that token, and then I'm going to try to create a new product.



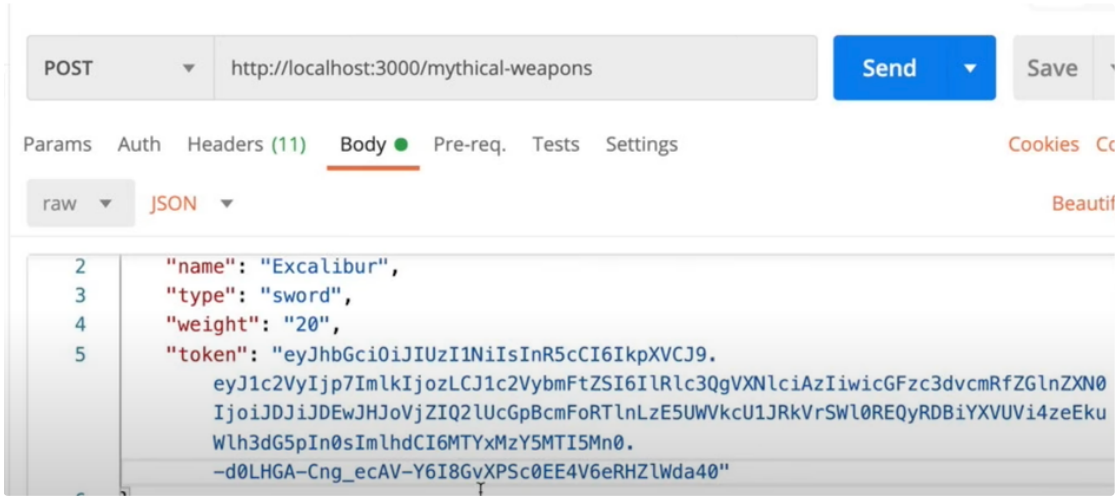
Notice that here, I am not supplying the token that I received at sign up.



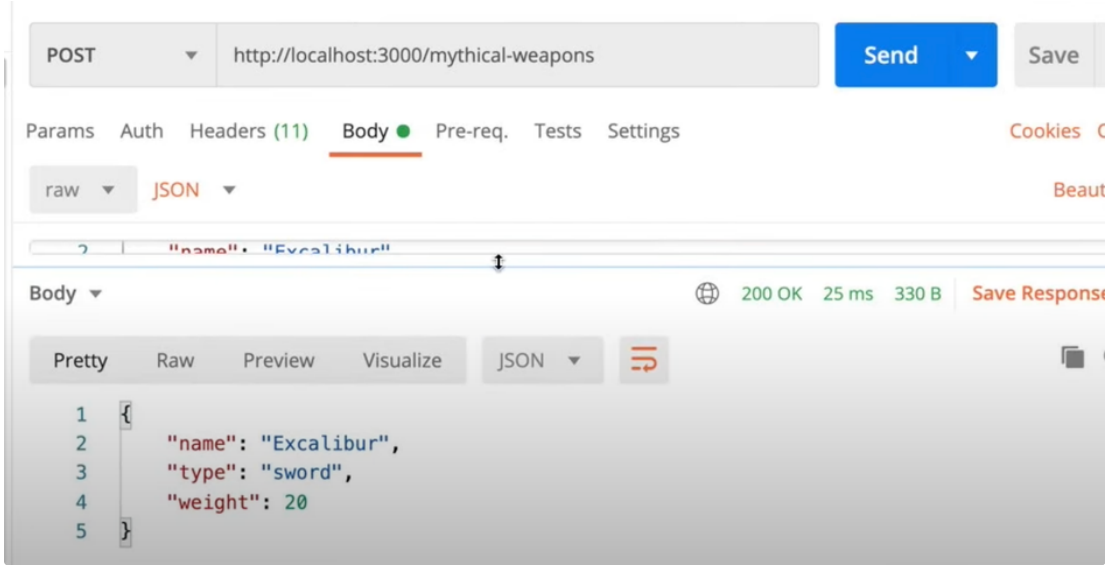
Here I'm writing a request to create a new mythical weapon, but I'm not supplying the token. When I send this request, you can see that I get back an error, invalid token, JWT must be provided.



Now if I supply the token that I received at sign up and send the same request again.



You can see that my mythical weapon was added.



If we look at the index route for this, you can see that Excalibur has been added to the list of mythical weapons. You can see that we were blocked from creating a new weapon until we were authorized by having a JWT.

We are requiring authentication on a route by route basis. If we wanted to protect all endpoints of a handler, we could also create a custom express middleware.

Implementing JWTs in a real application

The discussion above shows in theory how to require a token to be present in order to perform an action. However, there is one big way that the solution above would not be sufficient for a real app, and that is how the JWT is passed to the API. In the discussion, you may have noticed that I get the token from `req.body.token`. And this technically works and is easy when testing with Postman and other tools. But in real life, the token will not be part of the request body. Instead, tokens live as part of the **request header**.

There are many reasons for this, like added security. But that discussion is a bit outside the scope of this course, what we will focus on instead is how to get the token out of the header and use it in our logic. When we use JWTs, we pass them as a special header called the **Authorization header** using this format:

> `Authorization: Bearer <token>`

Where `Bearer` is a string separated by the token with a space.

Getting the header

In Node, we can locate the authorization header sent with a request like this:

> `const authorizationHeader = req.headers.authorization`

Parsing the header

Then, to get the token out of the authorization header, we need to do a little bit of Javascript string parsing. Remember that the word "Bearer" and the token are together as string, separated by a space. We can separate them with this logic:

> `const token = authorizationHeader.split(' ')[1]`

Where we split the string by the space, and take the second item. The second item is the token.

Putting it all together

Now we have a way to get the token from its correct location in the authorization header, so the code from the discussion could be revised to look like this:

```
const create = async (req: Request, res: Response) => {
    try {
        const authorizationHeader = req.headers.authorization
        const token = authorizationHeader.split(' ')[1]
        jwt.verify(token, process.env.TOKEN_SECRET)
    } catch(err) {
        res.status(401)
        res.json('Access denied, invalid token')
        return
    }

    ....rest of method is unchanged
}
```

And this would work. But to be even more professional about this, let's make this process of requiring token verification easily replicable by turning it into a function.

Making a custom Express middleware

In the handler file, we are going to add a new function called `verifyAuthToken`. I'll first show you the function, most of the logic is a direct copy from the create method above:

```
const verifyAuthToken = (req: Request, res: Response, next) => {
    try {
        const authorizationHeader = req.headers.authorization
        const token = authorizationHeader.split(' ')[1]
        const decoded = jwt.verify(token, process.env.TOKEN_SECRET)

        next()
    } catch (error) {
        res.status(401)
    }
}
```

Things to note:

This function takes in three arguments, `req` and `res` (exactly like a route handler) and another called `next`. This is how we create a custom Express middleware.

We complete the function, not with a **return** but by calling **next**. If the token could not be verified, we will send that 401 error.

Now, we can tell Express to use this middleware, like this:

```
const mount = (app: express.Application) => {
    app.get('/users', index)
    app.get('/users/:id', show)
    app.post('/users', verifyAuthToken, create)
    app.put('/users/:id', verifyAuthToken, update)
    app.delete('/users/:id', verifyAuthToken, destroy)
}
```

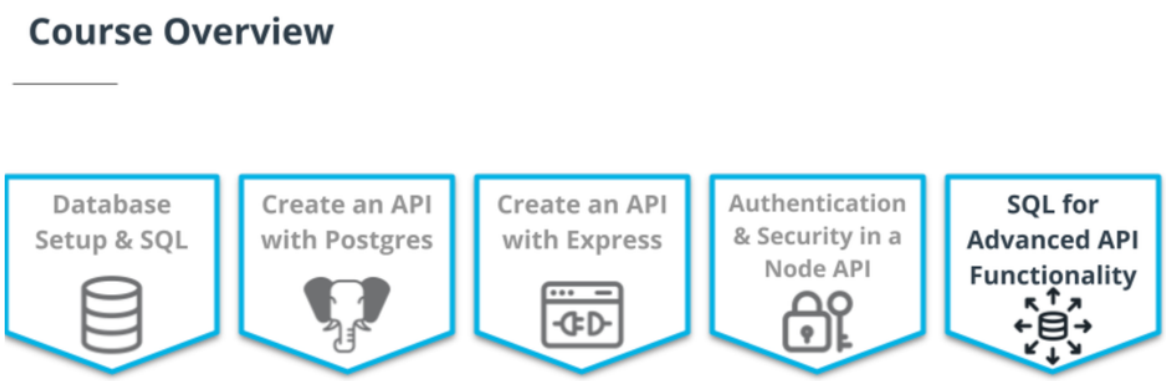
So, for the CREATE route, you can see that the request will come in and `verifyAuthToken` will be called before the handler's `create` method.

And that's it! You've created a custom Express middleware!

SQL for advanced API functionality

Lesson Overview & Introduction

Course Progress



You are here: SQL for Advanced API Functionality

What we'll do

You have created an API to power a full stack application - great job! The main goal of this course is complete. But what you have learned so far is truly just the beginning of all you can do with these skills. Right now, you can create a RESTful API that supports CRUD for all entities in the database, but what's beyond that? In this lesson we'll explore advanced SQL queries to support a wider variety of API endpoints, we'll cover these topics:

- Database relationships
- SQL Joins
- RESTful endpoints using with joins
- RESTful endpoints with params

Helpful Preparation

To get ready for this lesson, make sure you're comfortable with all the SQL topics we covered earlier in the course, and feel free to peruse these resources:

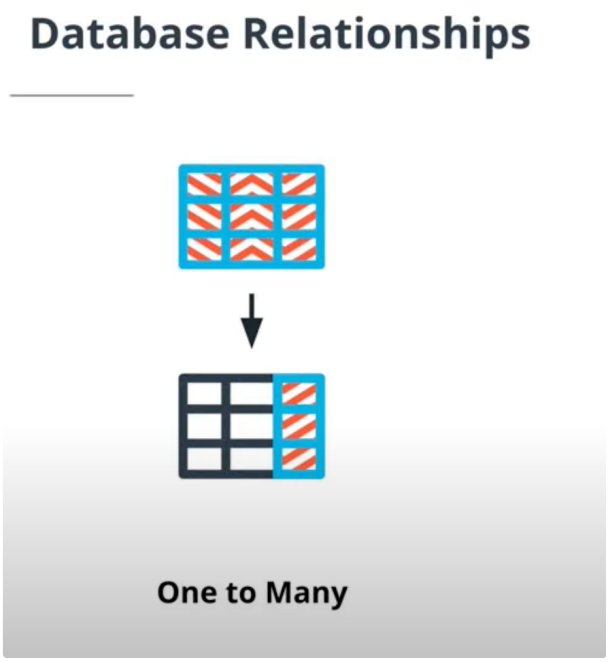
- Introduction to database relationships article from Lifewire.
- For a visual representation of SQL joins, take a look at this blog post.

SQL Relationships - Has Many, Belongs to

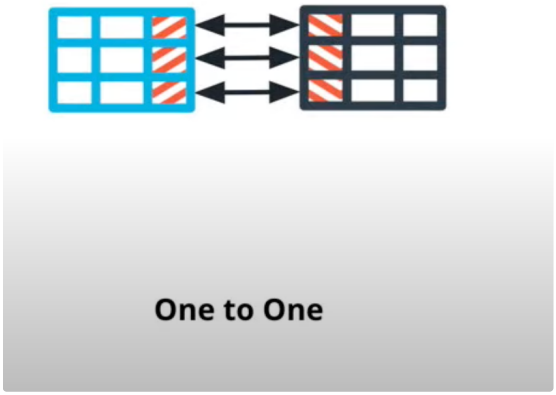
SQL Relationships

Database tables can be related to other tables in the database. In the SQL lesson, we used foreign keys to relate information from one table with another. We had a list of herbs, and each row on the herbs table had a column world_id that held the id of a row in the worlds table. Because of the presence of this foreign key, the herbs table is related to the worlds table and we can make more interesting queries of this relationship.

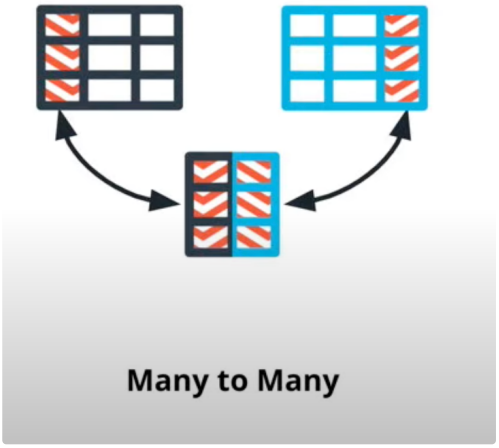
Relationships between tables take different forms, and there is some language to describe these relationships. In this section, we'll discuss the different types of relationships and how to create them.



In this section, I'm going to show you three of the most common database relationships. One of these relationships you actually have already come across, which is the one-to-many. When we had the relationship between the plant's table and the region's table in the very first lesson in this course, we created a one-to-many relationship. Where one table is associated to another with a foreign key, where only one of the tables holds that foreign key. In this case, plants belonged to a region, and the plants table contained a foreign key that was the region ID or the primary key of the regions table. This is one of the most common database relationships that you'll find.

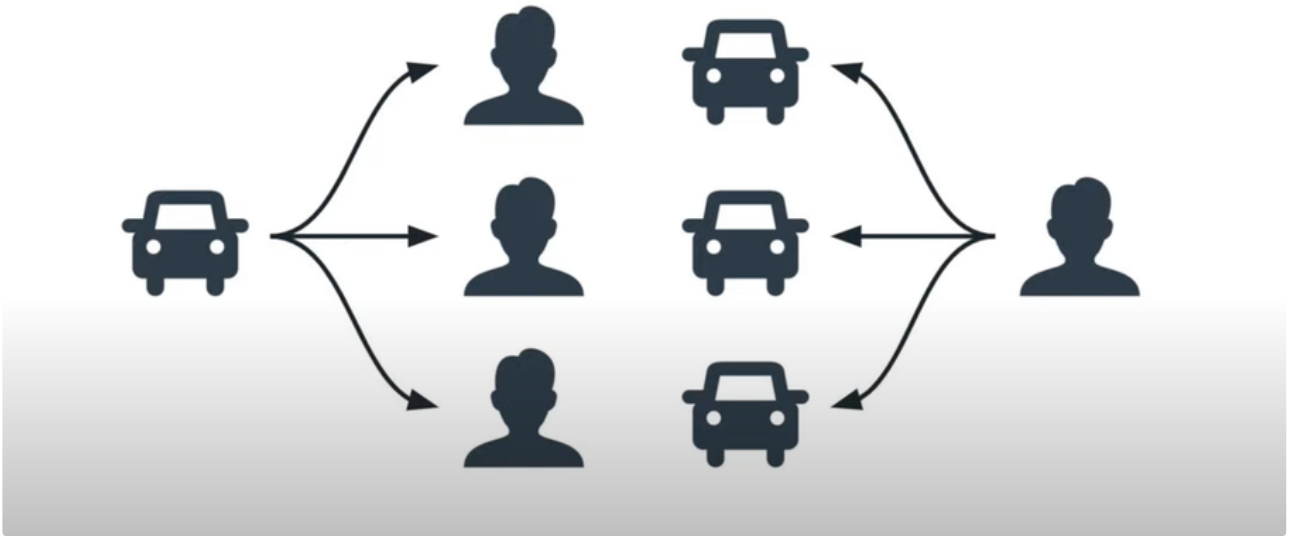


Another is the one-to-one relationship, where rows between two tables are locked in sequence as they share the same ID.

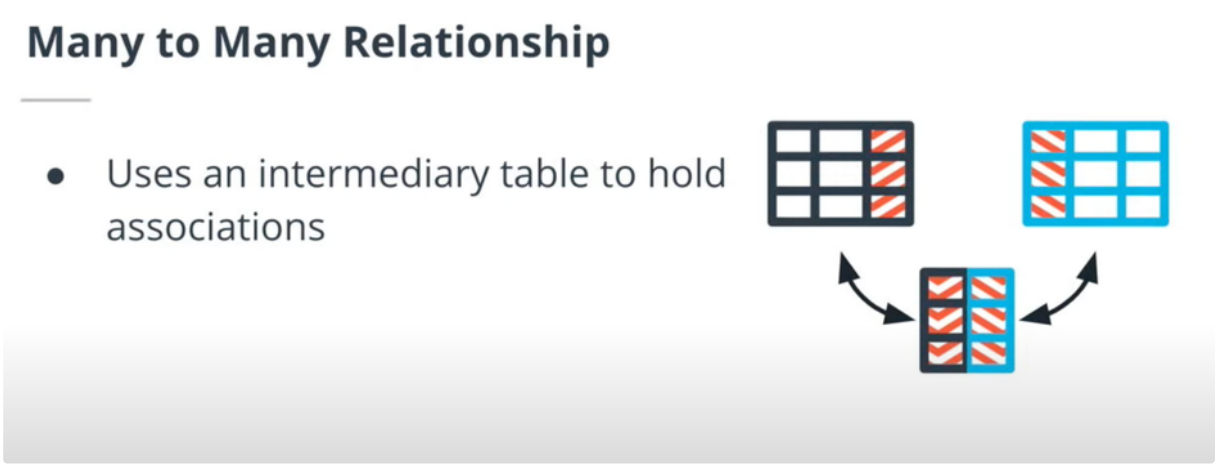


In a many-to-many relationship, two tables are given a special relationship through an intermediary table. This many-to-many relationship is what we'll discuss for the rest of this section.

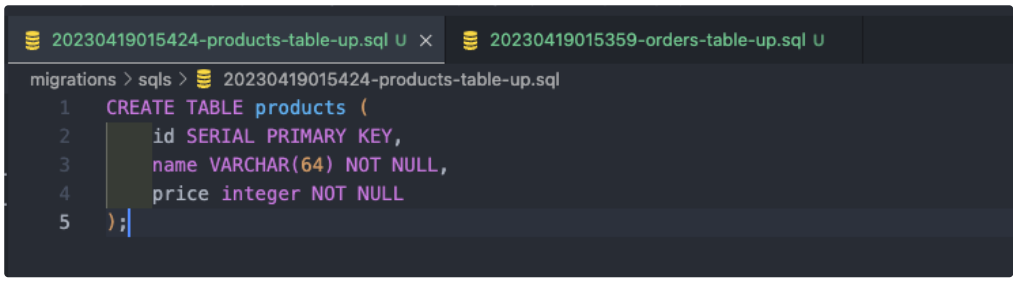
Example Relationship



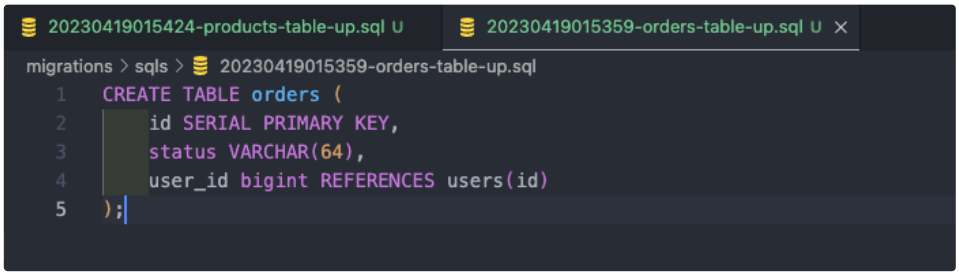
Imagine the relationship between people and cars. A car can be associated with many people, like a family car that's used by multiple members of the family. But one person can also be associated with multiple cars. This would be like if one person owns many cars. How could we represent this relationship in the database? Well, it's not obvious at first because we can't achieve this relationship with some of the strategies that we've used so far.



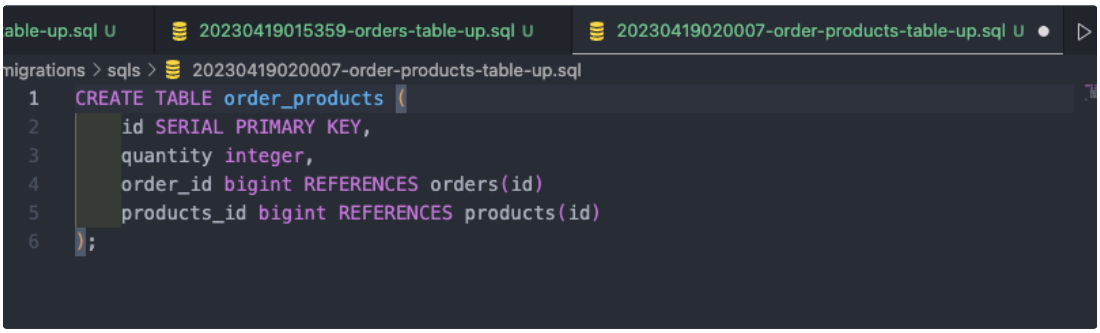
One table isn't enough to hold this two-sided relationship. We need another place or a new table to hold the associations between cars and people. We will do this in what is called a join table. I'm going to walk through the process of creating a many-to-many relationship between products and orders in our API using migrations.



First, I've created the migrations to add the products and orders table, respectively. Products have an ID, a name, and a price.



Orders have an ID, a status, and belong to a user ID. This is a foreign key that references the ID column of the user's table because an order belongs to a person. This is an example of a one-to-many relationship.

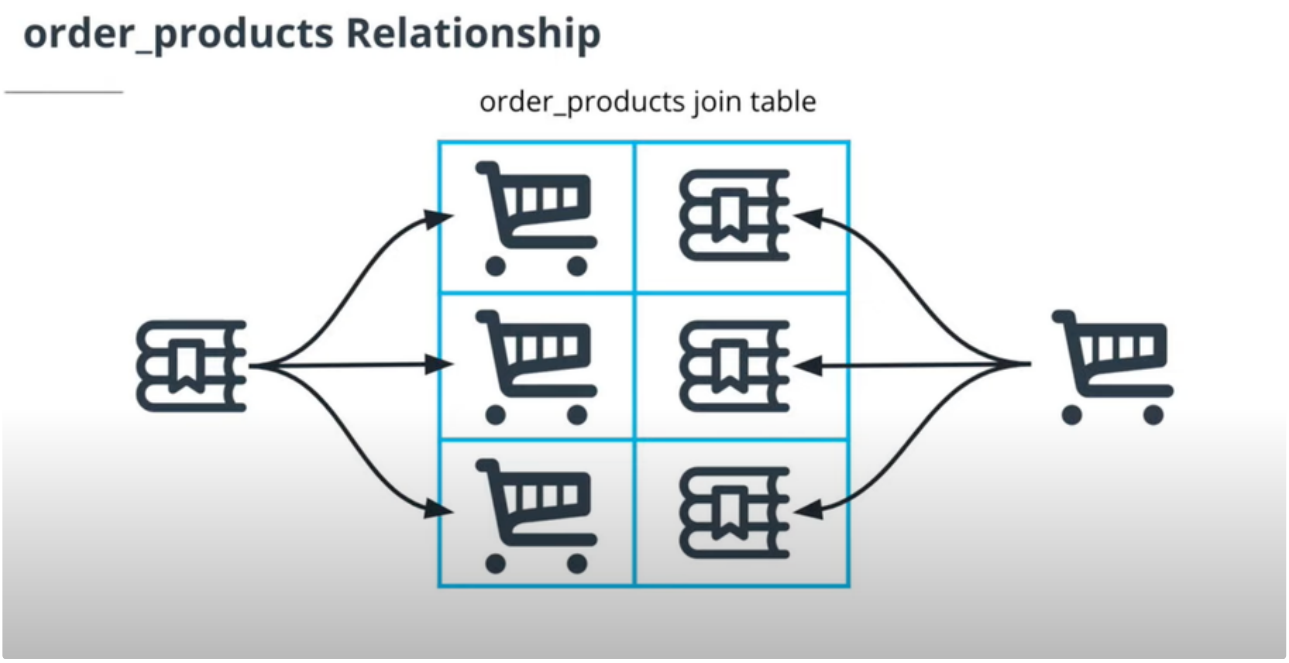


But we're talking about a many-to-many relationship. We need to create the join table between orders and products. I've created a migration to add that table here. The join table will be called order products to talk about the two tables that will be joined through this table. Each row will have an ID like normal. But the important thing to note here is that there will be two foreign keys. One column holds the order ID, which

is a foreign key to the ID of the orders table. There's another column for the product ID, which is a foreign key to the ID of the products table. This join table gives us the ability to create relationships in any direction that we want between orders and products.

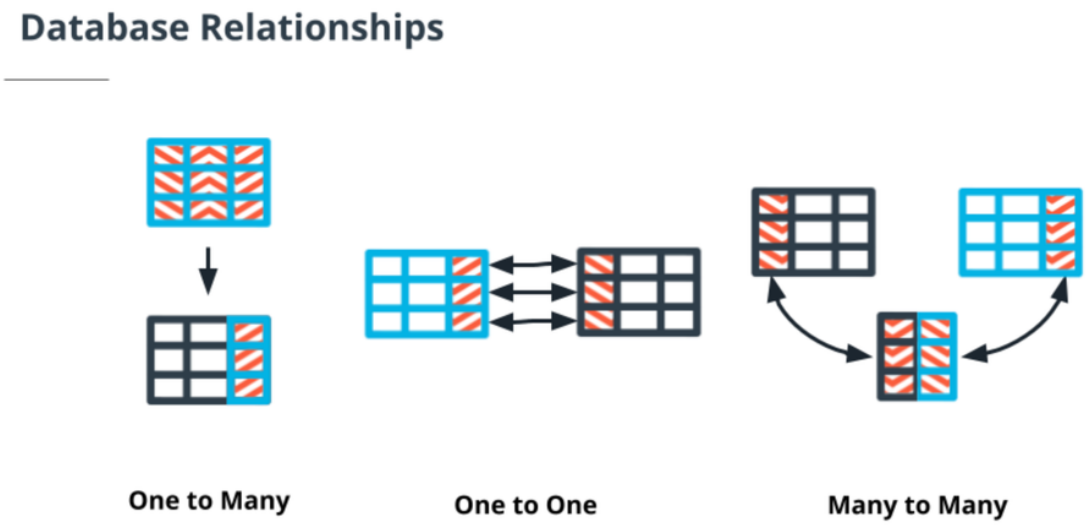
We'll also keep track of the quantity because a product being ordered can be ordered in a specific quantity. It's important to note that the foreign keys between these two tables live here in the join table and that the orders table has no foreign key directly to the products table, and the products table has no foreign key directly to the orders table. Both of those foreign keys only live here in the order products table.

We've just created the order products join table, where products can be related with many orders, and orders can be related with many products.



Summary

In this section we went over a few different types of database relationships. Here is a summary of the types.



Database Relationships

One to Many

One to many is like the relationship with plants and regions at the beginning of the course where many plants could be associated with one region by adding a foreign key on the plants table - the belong to side.

One to One

In a one-to-one relationship, one row in a table is associated with one row in another table - just one row. Where in the one to many, many plants could be associated with one region, this would be if there could only be one plant per region.

Many to Many

Many to many was the focus of this section and describes a more complex relationship between where rows on both tables can be associated with many rows on the other. This relationship is achieved by an intermediary table that stores each relationship as a row, this is called a join table.

Creating A Cart - Models and Requests

In the last section, we added a product table so we can store and create products. We also added an orders table so that we can store and create orders. Then we added an order products table so that we could add products to orders. The focus of this section will be to use these new tables to add a cart functionality to our API.

The idea of a cart in an e-commerce website is really just an open order that products can be added to. We're going to create the endpoints and logic required to provide this. First, we need model files for the two new tables. The join table will not have its own model. These models already contain the logic for creating new orders and products, but we're going to focus on creating the logic for adding products to existing orders.

```
export type Product = {
    id?: string;
    name: string;
    price: string;
}

export class ProductStore {
    async index(): Promise<Product[]> {
        try {
            //@ts-ignore
            const conn = await Client.connect()
            const sql = 'SELECT * FROM products'

            const result = await conn.query(sql)

            conn.release()
            return result.rows
        } catch (err) {
            throw new Error(`Unable get products: ${err}`);
        }
    }
}
```

Where do you think that logic should go? It belongs on the order model because in this case, the order is the primary entity, and it will have many products associated with it.

```
async addProduct(quantity: number, orderId: string, productId: string): Promise<Order> {
    try {
        const sql = 'INSERT INTO order_products (quantity, order_id, product_id) VALUES($1, $2, $3)'
        //@ts-ignore
        const conn = await Client.connect()

        const result = await conn.query(sql, [quantity, orderId, productId])

        const order = result.rows[0]

        conn.release()
        return order
    } catch (err) {
        throw new Error(`Could not add product $product ${productId} to order ${orderId}: ${err}`);
    }
}
```

In the orders model, I've created a new method called addProduct that takes in the necessary information for the join table, that is, the productId, the orderId, and the quantity of products in the order. When we write our SQL query, you'll notice that we're inserting a new row into the order products table, not the order table, like most of the other methods in this model. The rest of this method should look pretty familiar. It just adds a new row in the same way as the create method. The next thing we need to create is the handler function to create this endpoint.

```
import express, { Request, Response } from 'express'
import {Order, OrderStore} from '../models/order'

const orderRoutes = (app: express.Application) => {
    app.get('/orders', index)
    app.get('/orders/:id', show)
    app.post('/orders', create)
}

const store = new OrderStore()

const index =async (_req: Request, res: Response) => {
    const orders = await store.index()
    res.json(orders)
}

const show =async (_req: Request, res: Response) => {
    console.log(_req.params)
    const orders = await store.show(_req.params.id)
    res.json(orders)
}

const create = async (_req: Request, res: Response) => {
    const orders: Order = {
        id: _req.params.id,
        status: _req.params.status,
    }
}
```

We will add this handler function to the orders handler, but we need to figure out what HTTP verb to use and what the URL path for this endpoint should be. Because we're adding a new product to the order, this route will use the post verb. Even though both the order and product already exist in their own tables, we are creating a new relationship between them. We're adding a row to the join table. Because this endpoint will result in a new row being added to a table, it's a good clue that post is the right HTTP verb to use. Now what is the correct RESTful route when we need to show a relationship between these two things? Because the product is being added to a specific order, we're going to expose the following endpoint: orders id.

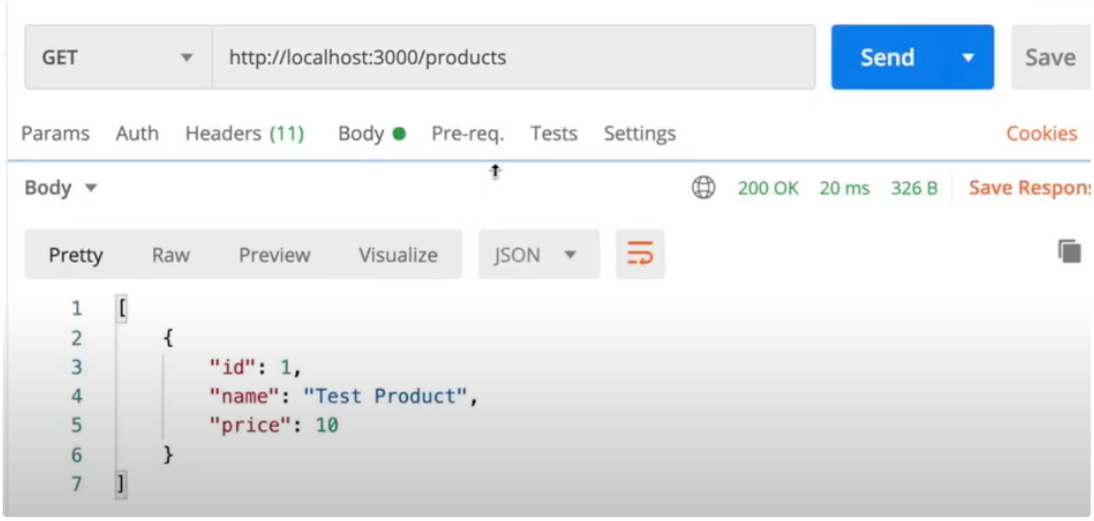
```
const orderRoutes = (app: express.Application) => {
    app.get('/orders', index)
    app.get('/orders/:id', show)
    app.post('/orders', create)
    //add Product
    app.post('/orders/:id/products', addProduct)
}
```

This would get us a particular order, but now it is the products belonging to this specific order. We'll set this route to call the addProduct method, which I've added here.

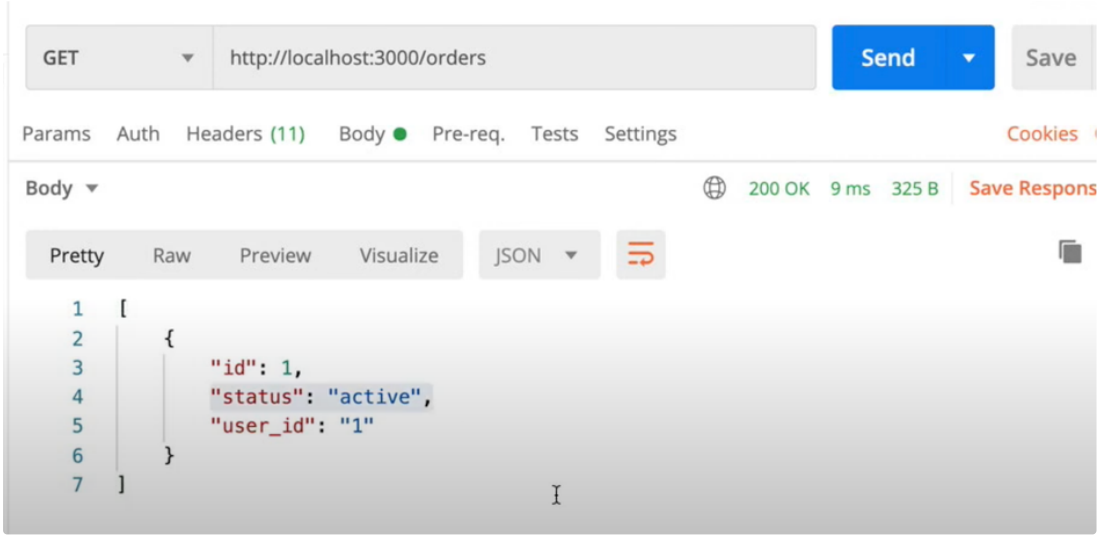
```
const addProduct = async (_req: Request, res: Response) => {
    const orderId: string = _req.params.id
    const productId: string = _req.body.productId
    const quantity: number = parseInt(_req.body.quantity)
    try {
        const addedProduct = await store.addProduct(quantity, orderId, productId)
        res.json(addedProduct)
    }catch(err) {
        res.status(400)
        res.json(err)
    }
}
```

The addProduct route gets the orderId, productId, and product quantity out of the request and passes them to the addProduct method on the model.

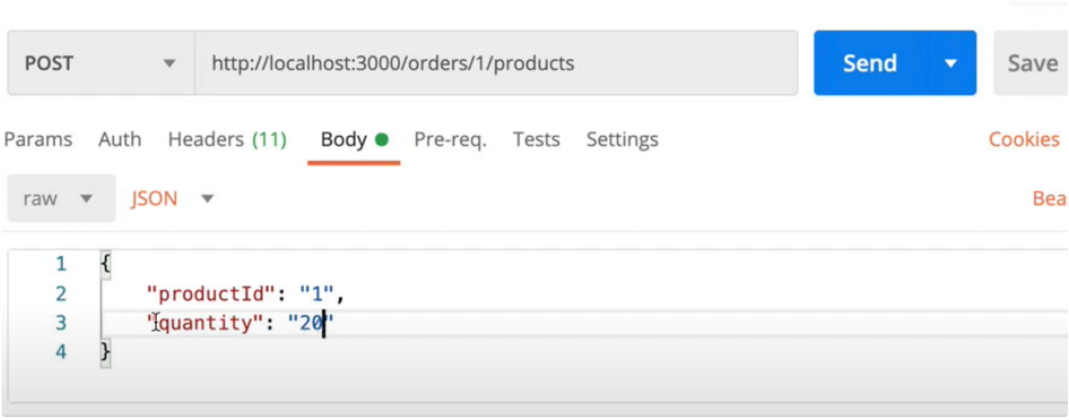
Now let's see this in action with Postman. Here you can see that I have a list of products with a single product already created.



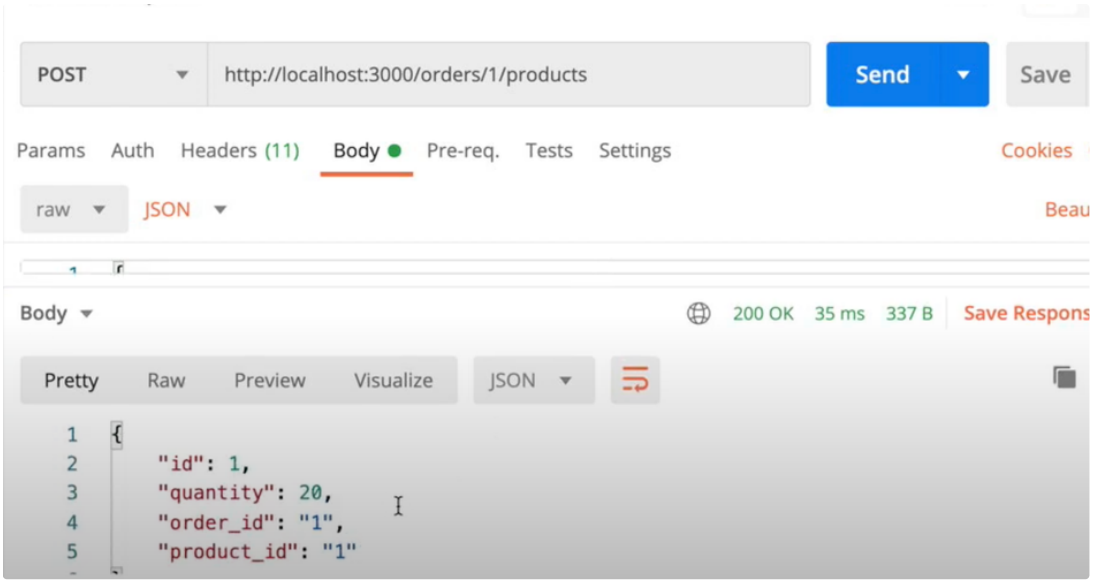
And orders with a single order already created.



Now what I want to do is test the logic that we've just written to see if I can add a product to this order. I'll use the post method and specify the ID of the order and products.



I need to pass in the required parameters. ProductId, which will be one, and quantity, we'll say that's 20. I don't need to pass the orderId here in the body because the orderId is available from the URL path. When I send this request, you can see that we get back the row that was added to the order products table that has an id, a quantity, the order_id foreign key, and the product_id foreign key.



We just added a many-to-many relationship in our database and carried that functionality through all the way to an endpoint. This would give us the ability to create a functioning cart in our API, allowing users to create orders and then add products to those orders. You can easily see how adding other cart functionality would just build out this cart to be more useful for users.

Summary

This lesson shows how to create endpoints for the many to many relationship we created in the last section to give the API cart functionality.

- Nested REST routes to show relationships

More SQL: Sorting and Joins

SQL commands for sorting

There are a few more SQL commands that will really come in handy; all of these are for ordering the responses you get back from a query.

Order By, Ascending, and Descending

These commands allow you to order responses alphabetically or by a number. First, you choose the column that you want to use to order the rows, then you can choose the direction - ascending or descending. Here's an example:

```
SELECT * FROM products ORDER BY price DESC;
```

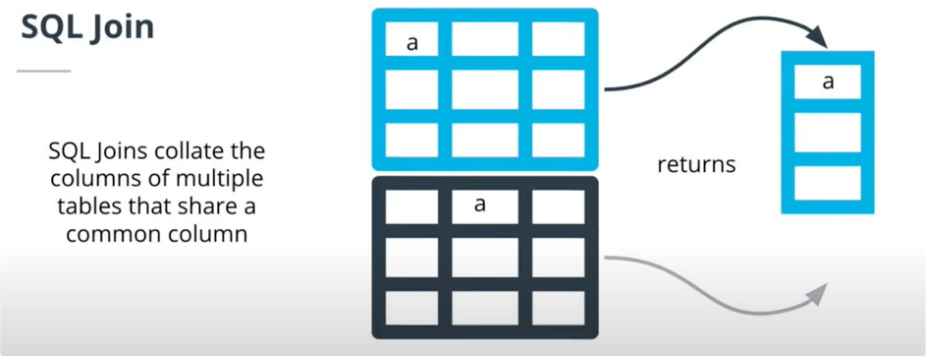
This query would get you all rows and columns from the products table. We have chosen to use the price column as the piece of information we want to order by, and chosen DESC as the direction, so the response we see will be a list of all products where products with the highest prices would come first and get smaller as you scrolled down the list.

```
SELECT * FROM users ORDER BY name ASC;
```

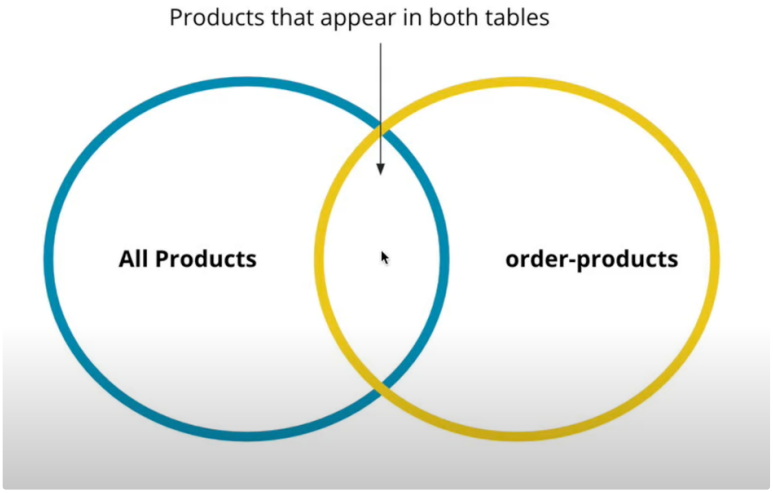
Same thing but with alphabetical order instead of numerical. This query would get all the rows and columns from the user's table and sort the rows by name, starting with A and going up to Z.

SQL Joins

This section will cover SQL joins. These allow us to collate and analyse related data in a Postgres database.



I'm going to walk through creating a join in the code. But think of the products, orders, and order_product tables from the last section. What if I wanted to find a list of all of the products that have been included in an order. It's easiest to envision joins as a Venn diagram, where each table involved is a circle and the rows that show up in both of those tables are the overlap, and that's what we want our query to display. This kind of join is the most common and is called an inner join, meaning we only want to see the overlap in the Venn diagram.



Now let's look at running this in the database. I want to display a list of the products that have a related order, but to do that, I need a piece of information that's shared between the two tables. In this case, we have an obvious one, the product_id. To start this join, I'll use the familiar SELECT word and specify that I want my end result to contain the name column FROM the products table and here we specify that we're making an INNER JOIN with the order_products table and the piece of common information that we'll use between them, we specify with the word ON and we will say that the ID column of the product table is the same as or maps to the product_id on the order_products table.

```
fantasy_worlds=# SELECT name FROM products INNER JOIN order_products ON products.id = order_products.products_id;
```

When I run this query, you can see that I get two response which says that Orange and Mango, whatever its ID is, shows up in both the product and order_products table.

```
 name
--------
 Orange
 Mango
(2 rows)
```

What this query is essentially doing is taking the two tables and combining them into one giant table where we only see the columns in the response that we included in the select. I can show this by adding more columns to the SELECT statement.

```
fantasy_worlds=# SELECT name, price, order_id, quantity, products_id FROM products INNER JOIN order_products ON products.id = order_products.products_id;
```

Here you can see the joining of the two tables much more clearly, where I have the name and price coming from the products table but I also have the order_id, quantity, and product_id coming from the order_products table.

```
  name  | price | order_id | quantity | products_id
--------+-------+----------+----------+-------------
 Orange |    15 |        1 |       10 |           2
 Mango  |    10 |        2 |        5 |           1
(2 rows)
```

They're both showing up here in my response next to each other. What the join has allowed us to do is find the relationship between rows into different tables by knowing that they have a common column, in this case, the product_id. There are tons of other types of joins that allow you to get exactly the subset of information that you want to.

But to be honest, this type of INNER JOIN is by far the most common. That's it for now with joins.

Summary

This session explains and implements an inner join in psql.

Example join syntax:

```
SELECT * FROM products INNER JOIN order_products ON product.id = order_products.id;
```

Create a Dashboard Endpoint

API Routes for Dashboards

The API routes we have looked at so far in this course have all been around action - CRUD actions. But that isn't always the case; sometimes, a web app might just want specialised information from our API. An easy example of when this might be useful is for creating a dashboard page for admin or for a user profile in a front-end application.

Here is a copy of the SQL Join that I created in the last section:

> `SELECT * FROM products INNER JOIN order_products ON product.id = order_products.id;`

The question now is ... what model/handler should this query belong with? Being a join, it involves two tables and not in a belongs-to relationship like orders and products. Even harder is that one of these tables is a Join table and not associated with a model. So what should we do?

Creating a Service

At this point, I would say that our needs for this application have grown beyond our simple model - handler architecture. It would not make sense to cram this query onto the products table or any of the other options discussed above. This `JOIN` query is **business logic that does not belong in any model or handler**, so we are going to put it in a new place called a **service.**

I will add a services folder as a sibling of models and handlers. `Services` will have a file called `dashboard.ts`. Here, we can add various methods that get information from the database in the form of specialized select queries or joins. One thing is very important - the dashboard will run SQL queries to **READ** information from the database, but **any actions on the database should be done through a model**. This dashboard file is simply allowing us to isolate our informational queries together in one place, rather than spread them out across all the models. Since what information is shown in the dashboard is likely to change often, this will cut down time to edit dashboard queries when needed. This also fits conceptually, because a model is supposed to be the representation of your database table in the Node application, it should not be concerned with getting the 5 most recently added products, for example.

A service file is a place to write extra business logic that does not belong in a handler or a model or orchestrates changes with multiple models.

For another example, as the complexity of our logic for authorizing which users can see various pages grows, the logic to check JWTs for authorization rights would become its own service as well.

The Code

src/services/dashboard.ts --> orderedProducts

```
import Client from '../database'

export class DashboardQueries {
  // Get all products that have been included in orders
  async productsInOrders(): Promise<{name: string, price: number, order_id: string}[]> {
    try {
      //@ts-ignore
      const conn = await Client.connect()
      const sql = 'SELECT name, price, order_id FROM products INNER JOIN order_products
                    ON product.id = order_products.id'

      const result = await conn.query(sql)

      conn.release()

      return result.rows
    } catch (err) {
```

```
      throw new Error(`unable get products and orders: ${err}`)
    }
  }
}
```

Things to note from this:

- We import the database client and create a connection in the method just like a model, because this service is running queries on the database, they will just be READ-ONLY queries, instead of updating tables, so this is ok.
- `productsInOrders` - sometimes, it is really hard to give a method a clear name, especially in situations like this. If you can't find a name that describes precisely what is going on, leave a comment like I did to explain what the name fails to convey.
- Notice the return type from this typescript method -- it isn't a product, an order, or any other type we created in the models. This is another sign that we were right to put this method away in its own service rather than in products, it is returning a hybrid of two tables, and that would be messy to implement in any model file.

Now for the handler

We will create a separate handler file for these methods.

```
import express, { Request, Response } from 'express'

import { DashboardQueries } from '../services/dashboard'

const dashboardRoutes = (app: express.Application) => {
    app.get('/products_in_orders', productsInOrders)
}

const dashboard = new DashboardQueries()

const productsInOrders = async (_req: Request, res: Response) => {
  const products = await dashboard.productsInOrders()
  res.json(products)
}

export default dashboardRoutes
```

Things to note here:

- This looks mostly like any other handler we created, but we aren't importing a model type, instead we are importing `dashboard` from services.
- RESTful routes are great for describing actions taken through the API, but they begin to break down for informational routes like this. Most of this comes down to personal preference, but I try to stick with REST as long as I can, and then name routes in the most descriptive way I can think of, and leave comments. A good pattern for naming these routes in your application may emerge as you build out more of them, so pay attention situationally to what the best options are for naming your routes.

Lesson Conclusion and Research Resources

Well done! In this lesson, we've uncovered more SQL goodness and followed through to how those more advanced queries can power new endpoints for our API:

- Database relationships
- SQL Joins
- RESTful endpoints using joins and relationships

Going Further:

- A good reference for join syntax from DoFactory.
- A cool tool for visualizing SQL joins with syntax.
- Another tool for visualizing SQL joins syntax for good measure.
- A resource for nested REST routes.

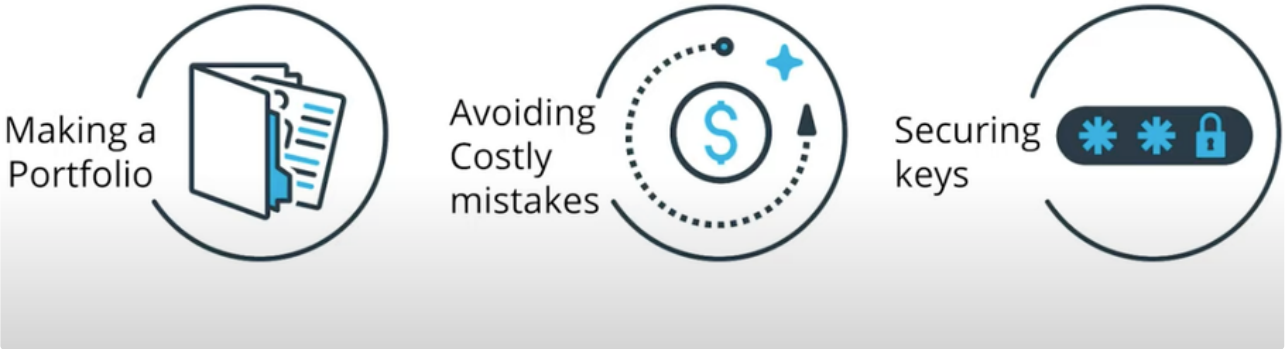
(Doc) Week 10:

Deployment Process

Foundation of Deployment Process

The Deployment Process Is Important

Why Is the Deployment Process Important?



Knowing how to code is not enough to get your application in the hand of users. Understanding the deployment process is important in order to:

- Make your portfolio of projects available online to recruiters and users.
- Avoid overpaying for services you don't need.
- Secure the private keys of APIs. Often these APIs might be paid services and you need to ensure you are not sharing these keys in public.

The great thing about learning deployment tools is that the knowledge transfers well to different cloud providers.

Further Reading

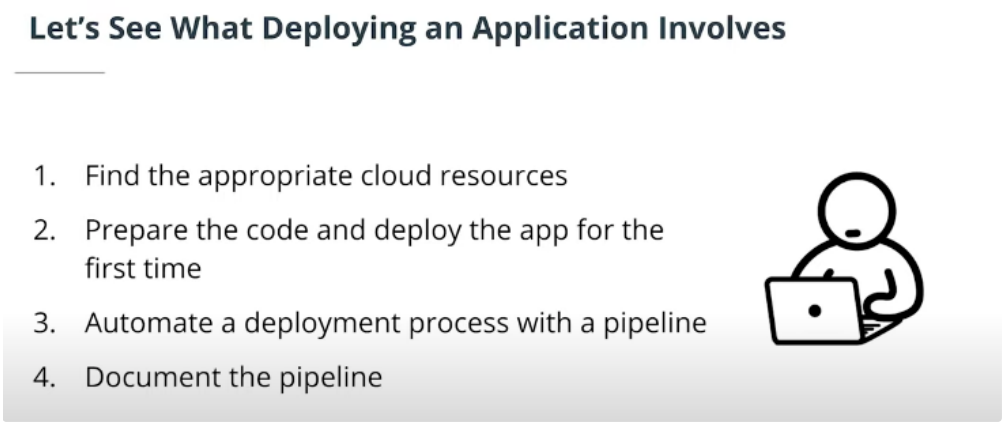
- A Beginner's Guide to AWS Cost Management. A great overview of the cost management tools offered in AWS.
- What is DevSecOps? Great overview of taking a security approach during the deployment process.
- List of cloud providers this is a great list if you want to explore different cloud providers. We will cover AWS in this course but there are a lot of alternatives.

Interested in learning more? These terms might be useful topics to explore!

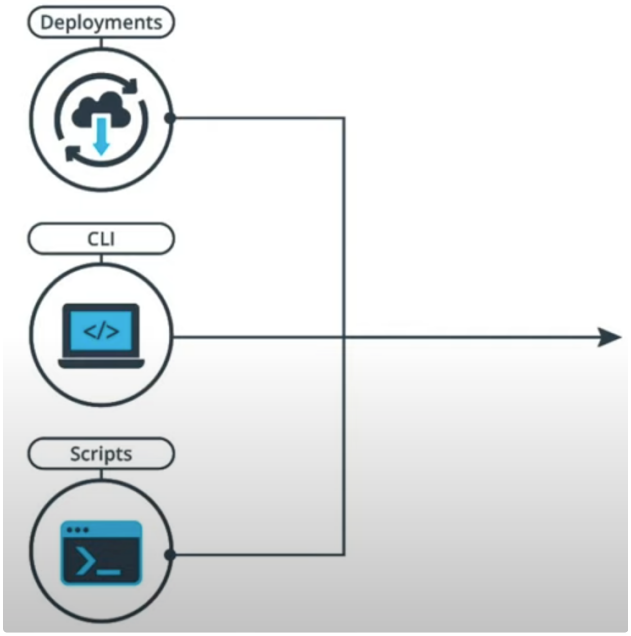
- **DevSecOps:** Adding Security to the DevOps keyword aims at promoting the importance of taking an active approach to security.
- **Cost Operations:** Taking an active approach to controlling cost saves a lot of money in the long run.

Introduction to the Deployment Process

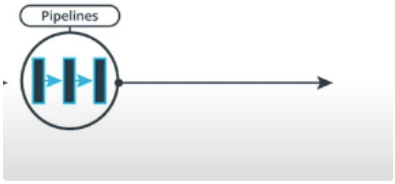
Let's now introduce the deployment process. What does deployment mean? Let's dive. Deploying applications involves a lot of things. First, it involves finding the appropriate cloud resources where we will host our application. It also means that as a second step, we will prepare our code and deploy the application for a first-time. This is important in order to gain a good understanding of the process. After we have gained this understanding, we will automate the deployment process with the use of a pipeline. Lastly, once we're happy with our pipeline, we will document it, in order to have good documentation when we come back to it.



Let's see a little bit what will be in this course. Let's have a look at this diagram which should explain a little bit better. We mentioned the word deployments. We mentioned also the word command-line interface, as well as scripts. All of these are skills that are really important when it comes down to deploying an application.



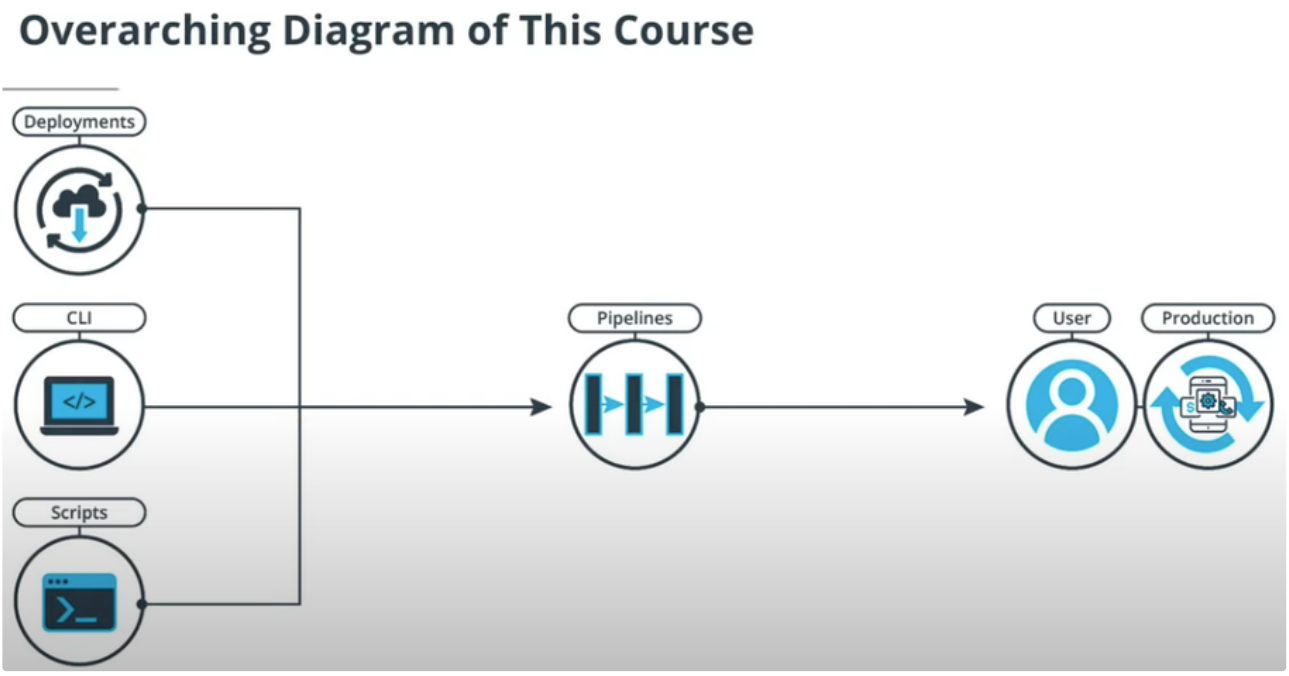
The point and they really go into making your pipeline.



Pipelines are really a use of all those skills that you will learn in the first lectures and in the end, the goal of a pipeline is to send your application to production so that it can be consumed by a user.

So at the core of everything that we do when we deploy an application, is the user using your code and using your application.

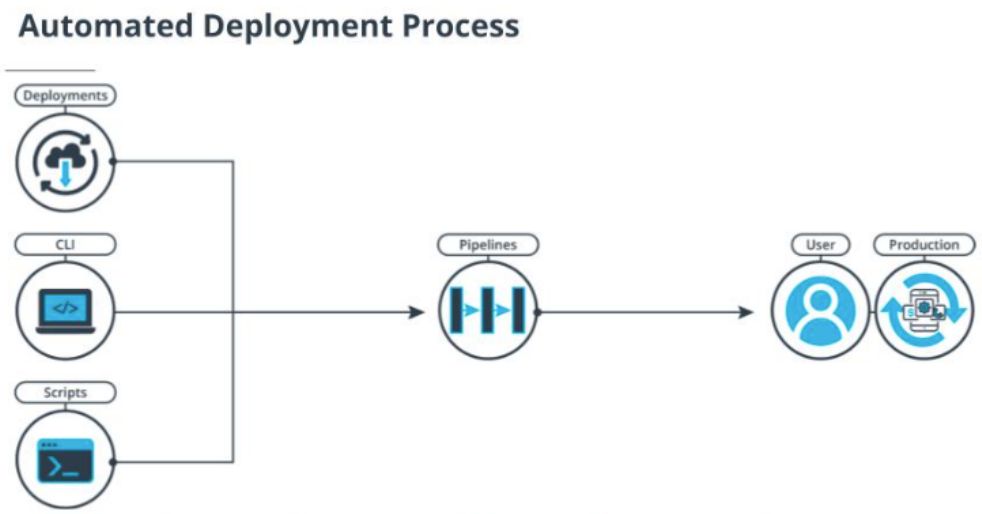
So never forget, we are doing all of this so that we can have a nice set of features for our users and make everybody happy.



Deploying an application includes the following steps:

1. Determining what cloud resources are needed.
2. Preparing the application code for production and deploying it for the first time.
3. Creating a deployment process that you will automate using a CI/CD Pipeline.
4. Documenting your process and iterating on it.

The image below illustrates how the main components of an automated deployment process are related. We will cover each of these components in-depth in this course.



Automated Deployment Process

New Terms

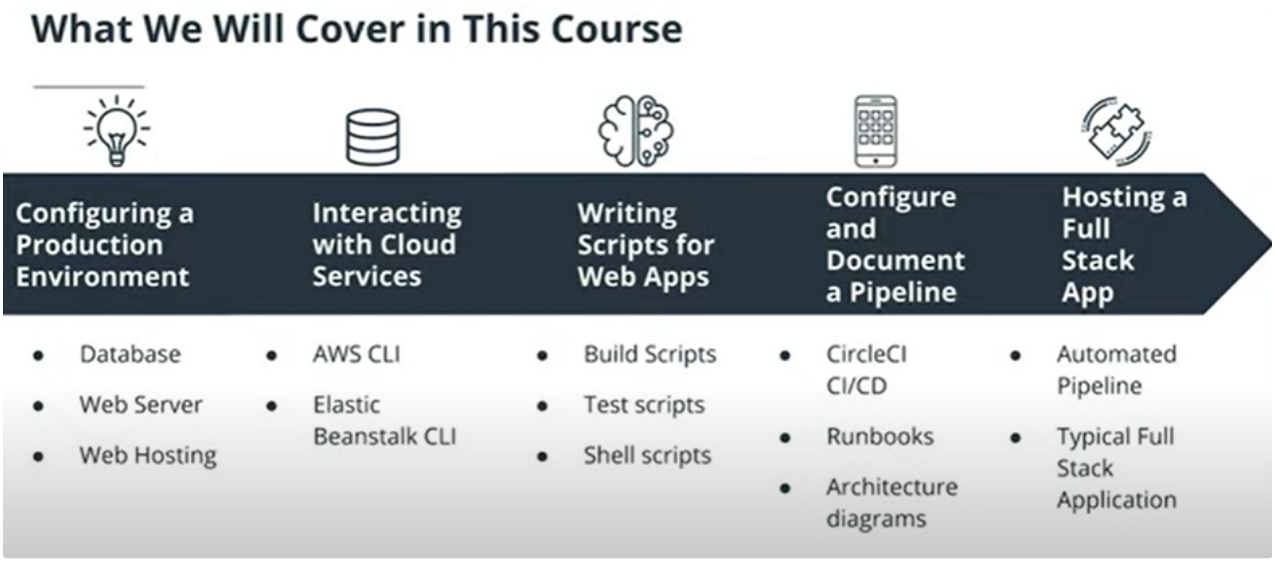
- **Pipeline:** A series of automated steps (Command-Line Interface commands) that simplify the testing, building, and deployment of code. While this might seem abstract for the moment, it will make more sense as we progress in this course.
- **Cloud Provider:** Companies that offer servers for rent and hosting services that provide flexible computing.
- **AWS:** Amazon Web Services is the dominant player in the Cloud Provider industry.

Further Reading

- Between Continuous Integration, Continuous Deployment and Continuous Delivery? Great blog post to understand the difference.
- How To Become a DevOps Engineer. This article covers more than needed for JavaScript apps but this gives a great overview of the landscape. The article mentioned the term **DevOps**, which is a collection of techniques trying to bridge the gap between development and operations.

Course Outline - What We Will Cover In This Course

What We Will Cover In This Course



In this course, we will cover the basics of deploying a web application and automating this process. Specifically, we'll talk about:

- **Setting up a production environment.** We will learn how to secure and set up a production server with environment variables through the AWS console.
- **Interact with cloud services.** By using a Command Line Interface (CLI), we will update the application code and deploy it to cloud services.
- **Write scripts for web applications.** We will learn how to automate with scripts most of the manual steps involved in deploying an application.
- **Configure and Document a CI/CD pipeline.** With the help of CircleCI, we will create a well-documented deployment Pipeline.

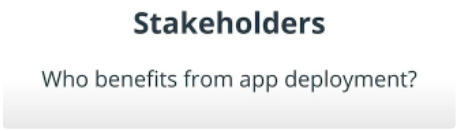
Learning Objectives

By the end of this course, you will be able to:

- Secure and set up a production environment and environment variables through the AWS console.
- Create package.json scripts to install and bundle a Javascript application
- Interact with cloud services to host a JavaScript application through a CLI.
- Automate and document the deployment process of a JavaScript application through CircleCI

Deployment Process Stakeholders

Deployment Process Stakeholders



Let's understand who the stakeholders are. Who really benefits from deploying an application? In my own opinion, everybody benefits from it; designers and product managers, you as a developer, as well as the customer.

As a developer, I was really happy when customers could use my application. But also, I sometimes felt like I was far from my customers. This was when I realized that **communication** is so important, and the deployment process plays a really important part in this. Mostly when you deploy an application, the feedback between you, your colleagues such as designers and product managers, and your customer gets shorter.

You get faster reactions to your code when you can deploy your application often, and thus, deployment really increases communication. So stakeholders are really everybody. Everybody benefits from getting a developer's work out in the open faster.

Anybody working on an application or consuming it is a stakeholder. Having a strong automated deployment process benefits all of these people:

- **Product managers and designers** benefit from a faster feedback loop with customers.
- **Developers** benefit from a more streamlined workflow with fewer manual interventions on their part.

- **Customers** benefit from receiving features faster and getting up-to-date software.

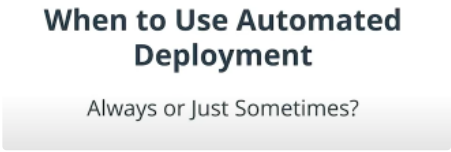
New Terms

- **Production:** The live environment our customers use to connect to our application.
- **Feedback loop:** A generic reference to the time it takes to receive customers' feedback on a new feature.

Further Reading

- Do not deploy on Friday! This is a great blog post challenging the pre-conceived idea that we can't deploy before the weekend in order to have a worry-free weekend.
- DevOps for Designers. This is a great post making the argument that we can all benefit from having a DevOps mentality.

When To Use Automated Deployments



Let's try to understand when we need to use automated deployments. Do we always want to be deploying automatically or just sometimes? Well, almost all of the time, we want to deploy automatically. However, it's not always possible, so let's be careful. Before we deploy an application, we want everything to be well-tested. We need to be confident that the app we're deploying is ready to be deployed. But also, we must ensure that we comply with regulations. Some industries really have a lot of regulations, so before deploying something automatically, we must be sure that we are complying with them. When this happens, it's useful to take a step back.

I really, really want to insist by saying that not everything needs automation. If you're thinking as industry such as legal or medical, you might be impacting people on parts of their lives that are really regulated by authorities. If you're thinking about medical records or legal records, you need to be sure that the code you're deploying in such industries complies with all the regulation.

But also, you might be executing a script that is doing a sensitive operation. Imagine you're migrating a database of credit cards, you might be doing this automatically and it would work, but it's better to take a step back and do this manually in order to have more confidence in what you're doing.

When automating deployments, almost anything can be done. However, we should take careful considerations in order to avoid potential errors. Think about the following questions:

- **Are you confident that the application is well tested?** If your test does not cover properly the core features of your application maybe you want to remediate that issue before moving to automated deployments.
- **Do you need to comply with regulatory procedures that could be impacted by a deployment?** Generally, it would be advisable to avoid automated deployments until you have a process in place to audit whether you comply with regulations in your industry.
- **Does everything need to be automated?** Automation is great at saving time. However, like other pieces of software, an automated deployment needs maintenance. It's good to slow down and automate where you need it instead of trying to automate at all costs.

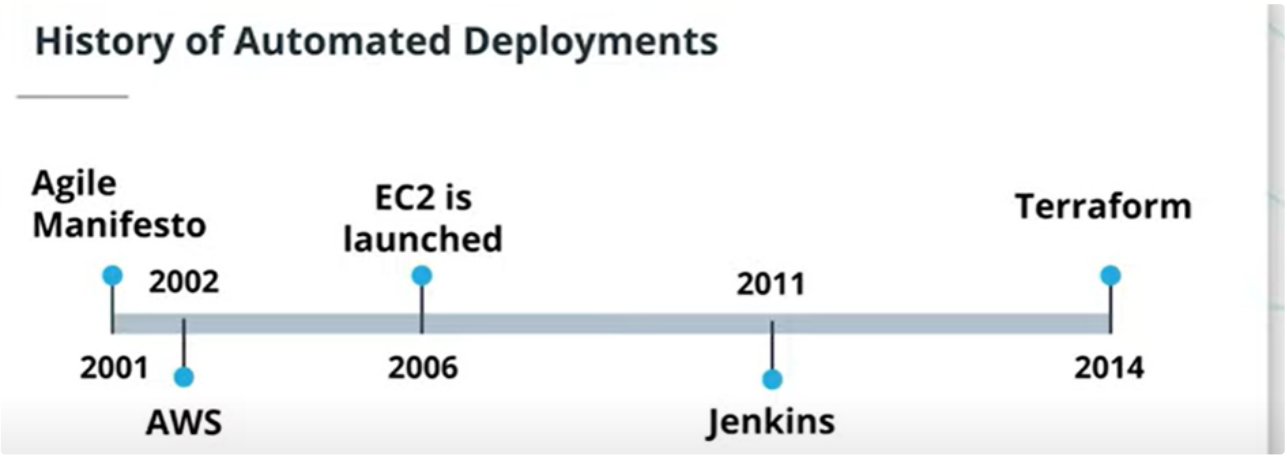
New Terms

- **Manual Check:** A step in a pipeline that requires human approval before going to the next step.
- **Regulations:** A series of rules that must be complied with.

Further Reading

- DevOps automation best practices: How much is too much? A gread read on striking a balance when automating.

History of Automated Deployments



The move to automated deployments happened gradually. These events contributed to the popularity of DevOps:

1. **2001 -** Agile Manifesto is published.
2. **2002 -** AWS platform is launched.
3. **2006 -** AWS EC2 is launched.
4. **2011 -** Jenkins is launched.
5. **2014 -** Initial release of Terraform.

New Terms

- **Agile:** A development methodology that prioritizes iteration over the heavy process.
- **EC2:** Amazon Elastic Compute Cloud, is the service amazon provides for renting servers on its cloud platform.
- **Jenkins:** One of the first CI/CD platforms and one of the most popular to this day.
- **Terraform:** A programming language that provides infrastructure as code capacities.

Further Reading

- Agile software development (Wikipedia). This Wikipedia entry goes in-depth, but you can get a good idea of the Agile methodology from reading the first few sections of the page.
- DevOps didn't exist when I started as a developer. Great blog post on why DevOps is not a single technology but a mindset.

Tools & Environment

Here are the tools you need:

- Git
- Terminal (Shell on Mac/Linux or PowerShell on Windows)
- Browser
- AWS CLI
- Elastic Beanstalk CLI

Glossary

New Terms In This Lesson

- **DevOps:** A collection of techniques trying to bridge the gap between development and operations.
- **Pipeline**: A series of automated steps that to simplify the testing, building, and deployment of code.
- **CI/CD**: Continuous Integration / Continuous Delivery (sometimes Deployments). Subparts of a pipeline dedicated to integrating code, delivering it, and deploying it.
- **Cloud Provider**: Companies that offer rentable servers and hosting services that provide flexible computing.
- **AWS**: Amazon Web Services is the dominant player in the Cloud Provider industry.
- **DevSecOps**: Adding Security to the DevOps keyword aims at promoting the importance of taking an active approach to security.
- **Cost Operations:** Taking an active approach to controlling cost saves a lot of money in the long run.
- **Production**: The live applications our customers use to connect to our application.
- **Feedback loop**: A generic reference to the time it takes to receive feedback on a new feature from customers.
- **Manual Check**: A step in a pipeline that requires human approval before going to the next
- **Regulations**: A series of rules that must be complied with.
- **Agile**: a development methodology that prioritizes iteration over the heavy process.
- **EC2**: Amazon Elastic Compute Cloud, is the service amazon provides for renting servers on its cloud platform.
- **Jenkins**: One of the first CI/CD platforms and one of the most popular to this day.
- **Terraform:** A programming language that provides infrastructure as code capacities.

Introduction-setting up a production environment

Lesson Introduction

In order to deploy an application online, we will first get familiar with Amazon Web Services (AWS).

In this lesson, we will learn about the different components of a production environment. We will cover the following topics:

- How experts approach a new application that they are tasked with deploying.
- Deploy a database using AWS RDS.
- Deploy a web server using AWS Elastic Beanstalk.
- Deploy a web UI using AWS S3.

Creating Resources in AWS Cloud

There are multiple ways to create AWS cloud resources, such as:

1. AWS web console
2. AWS CLI
3. Language specific AWS SDK

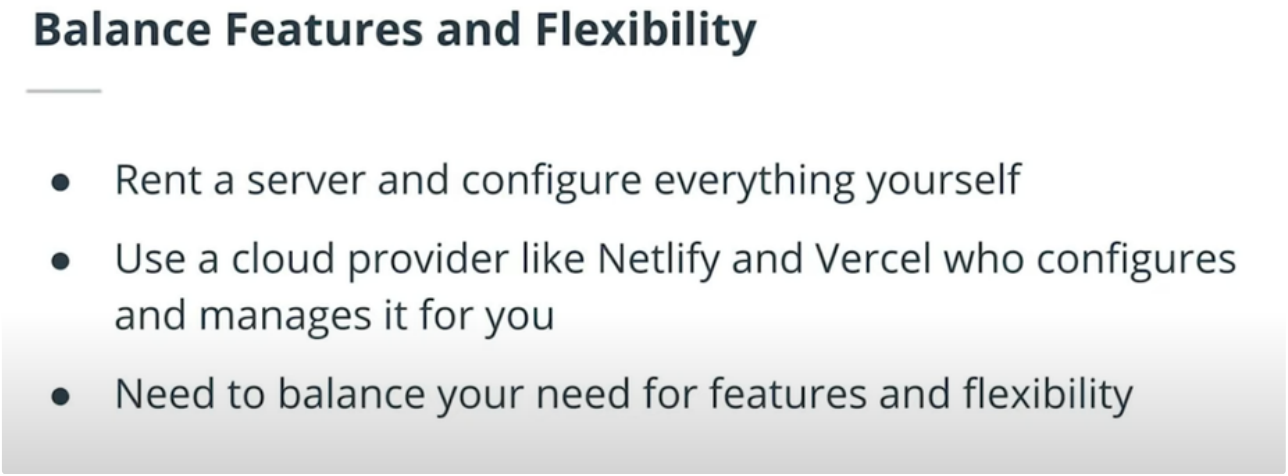
We will first learn to use the AWS web console to create and manage AWS resources in the current lesson. In the next lesson, we will learn to use AWS CLI.

Why Setting up a Production Environment?

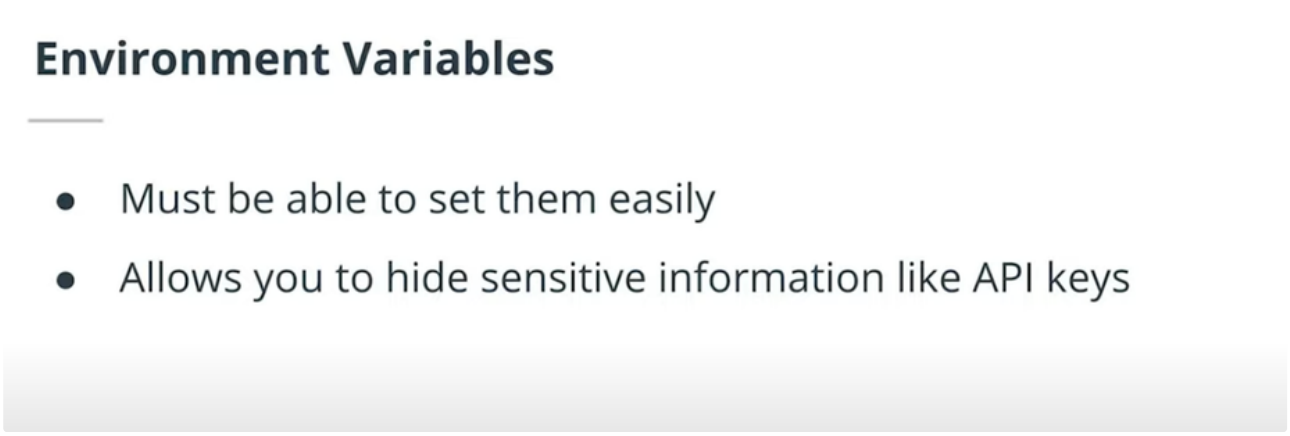
Why Does Your Production Environment Matter?

Production is the end goal. Your production environment is one of the goals of your deployment journey. The production environment is the complete set of services where your customers will consume your application. I often like to compare the production environment to a construction site where you will build your house. Before choosing your bathroom towels, you need to find a vacant land with a nice view where you will build your house. This is why as the person responsible for deploying the application, you have to choose your production environment with care.

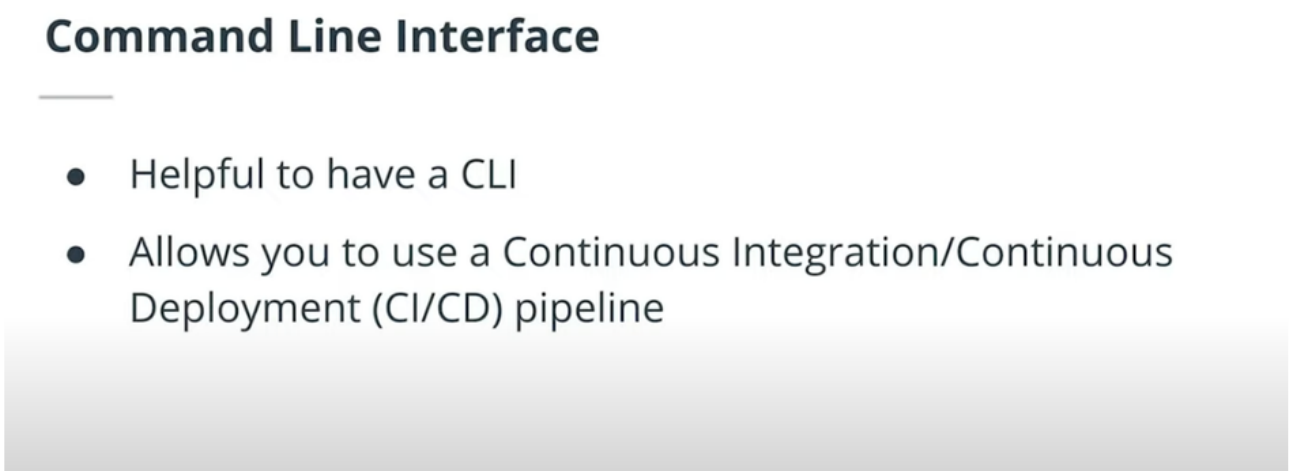
Now, when choosing a production environment, you are looking for a balance of features, and flexibility. On one end of the spectrum, you could decide to rent a server and configure everything yourself. This would mean that you would spend a lot of time configuring the server and all the tools that come with that. Or on the other end of the spectrum, you could use a cloud provider like Netlify, or Vercel, that configures, and manages everything for you.



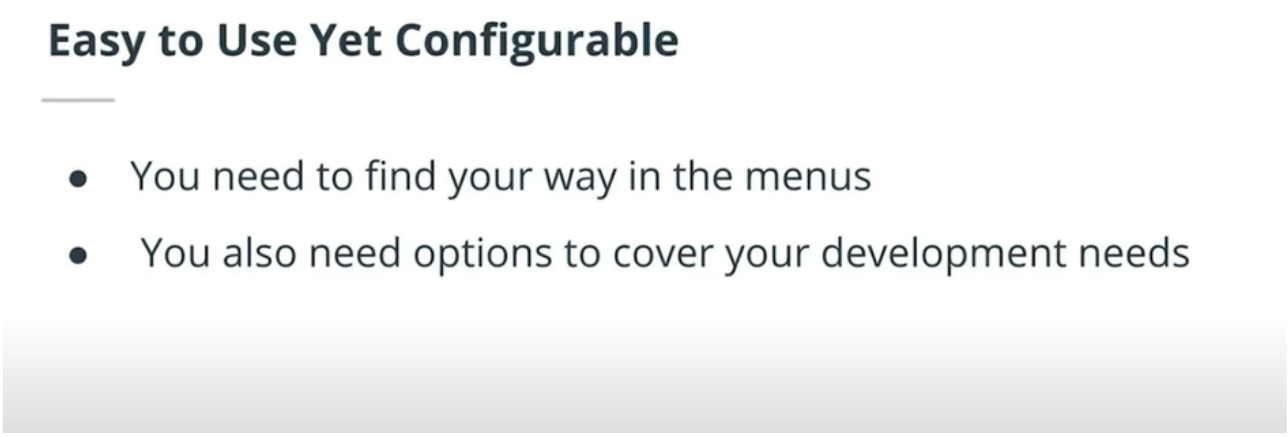
However, you would be losing a little bit of flexibility here. It is important that you balance the need for features, and flexibility, when you choose your production environment.



Another thing that we will be looking at when choosing, is the ability to set environment variables. You must be able to set them easily, as they are really important to hide sensitive information like API keys, or database connection strings.

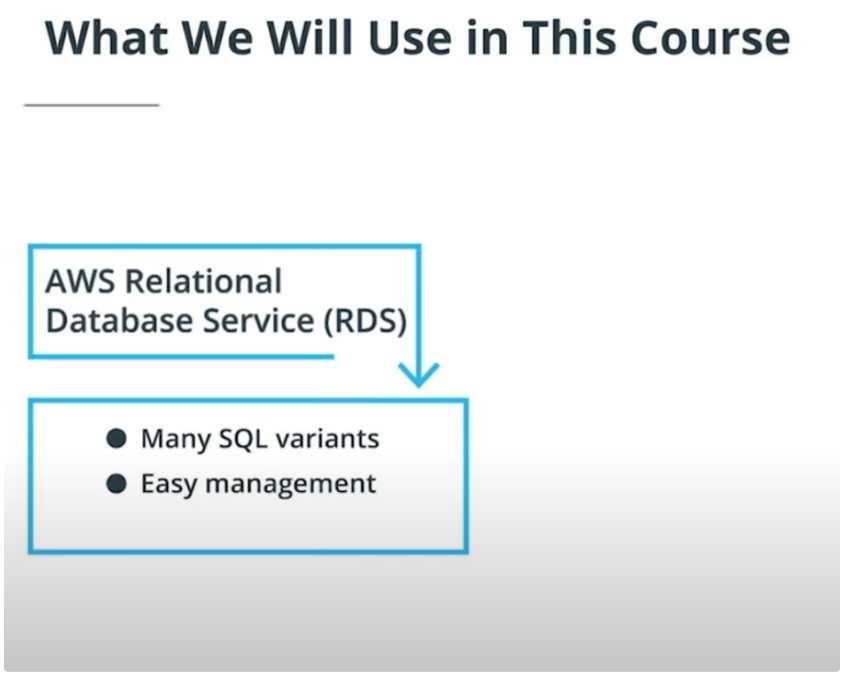


Another nice thing to have, is a command line interface that lets you manipulate the production environment. It is really helpful to have a CLI, because most of the tools that you will need to create a continuous integration, and continuous deployment pipeline, will be done through a CLI.

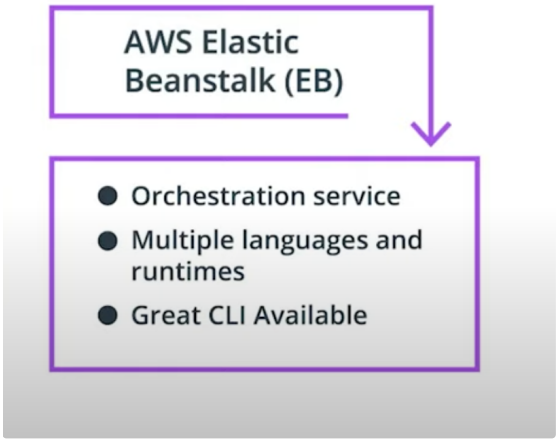


Lastly, one of the things we look for in a production environment, is something that is easy to use, yet configurable. As a developer, you are at your best when you're building up features, and not configuring the production environment. If you need to find your waste in the menu, and you have a difficult time doing this, it is time that is not well-spent. So you need to have easy menus that you understand, in order to really configure a prediction environment effectively. But you also need on the other end, and of options so that all of your development

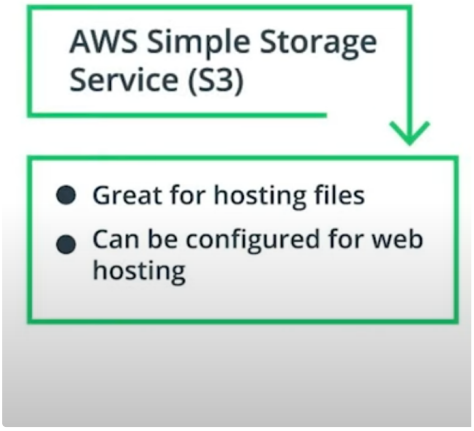
needs are covered. Like we mentioned, things like CLI, and environment variables are important for your application. What will we use in this course for our development environment? The first thing we will use is AWS Relational Database Service, otherwise known as RDS.



It is the offering by AWS, to have many databases in many SQL variance, and it's also provide for a really easy management tool for most production set-ups that you might need in your database.



For hosting your web server, we will be using AWS, Elastic Beanstalk, also known as EB. Elastic Beanstalk, is an orchestration service, that allows you to run servers in a multitude of language, and different runtimes, including NodeJS, Go, and C Sharp. It also provides a great command line interface, where you can manipulate your environment.



Lastly, we will be using AWS Simple Storage Service. We will be using it for configuring web hosting, and also for, in general, just hosting different files.

Summary

The **production environment** is the complete set of services that makes your application available to your end customers. This is where your customers will consume your application and where new features will be made available to them on a regular basis.

Your production environment is the **end goal of the deployment process.**

When choosing where we will host an application, we are looking for a mix of **features** and **flexibility.**

Important features of the production environment

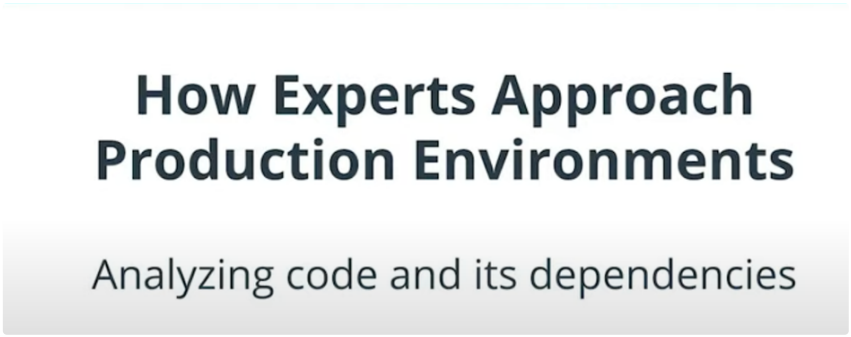
- **Environment variables:** The ability to set environment variables is very important, as it allows you to hide sensitive information like API keys. These values are dynamic variables that are used in your code.
- **CLI available:** It is helpful for us to be able to control the environment with a CLI, as it can automate a lot of operations on the environment.
- **Easy to use:** This is important to use since we are looking to spend more time working on features than configuring our environment in menus!
- **Configurable:** While we are looking for ease of use, we still need options to meet our development needs.

Tools used in this lesson

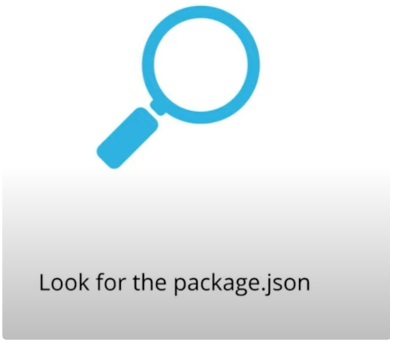
- **AWS Relational Database Service (RDS):** It has many databases and SQL variants and provides easy management.
- **AWS Elastic Beanstalk (EB):** This is an orchestration service that allows you to run servers in multiple languages and runtimes.
- **AWS Simple Storage Service (S3):** We will use this tool for configuring web hosting and for hosting files in general.

How Experts Approach Production Environments

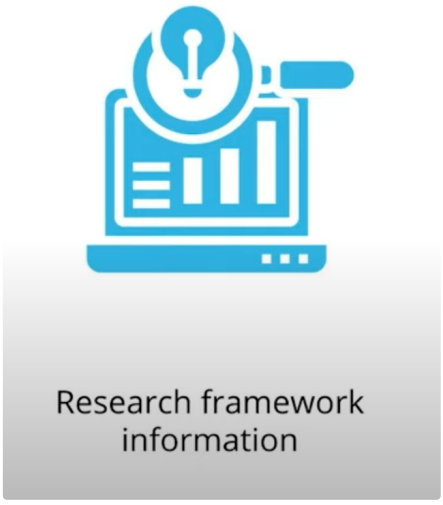
How Experts Think about Configuring a Production Environment



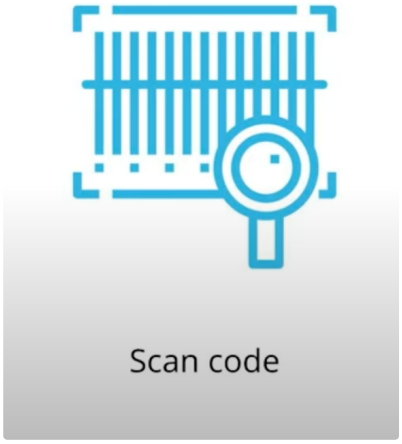
Let's see how experts approach configuring a production environment. It is not easy to set up a production environment, so we will learn how experts approach analyzing the code and its dependencies. There are a few steps that we can take in order to do that. The first one is that we will look forward to package JSON.



The package JSON will give you a lot of information about dependencies, scripts, as well as development dependencies.

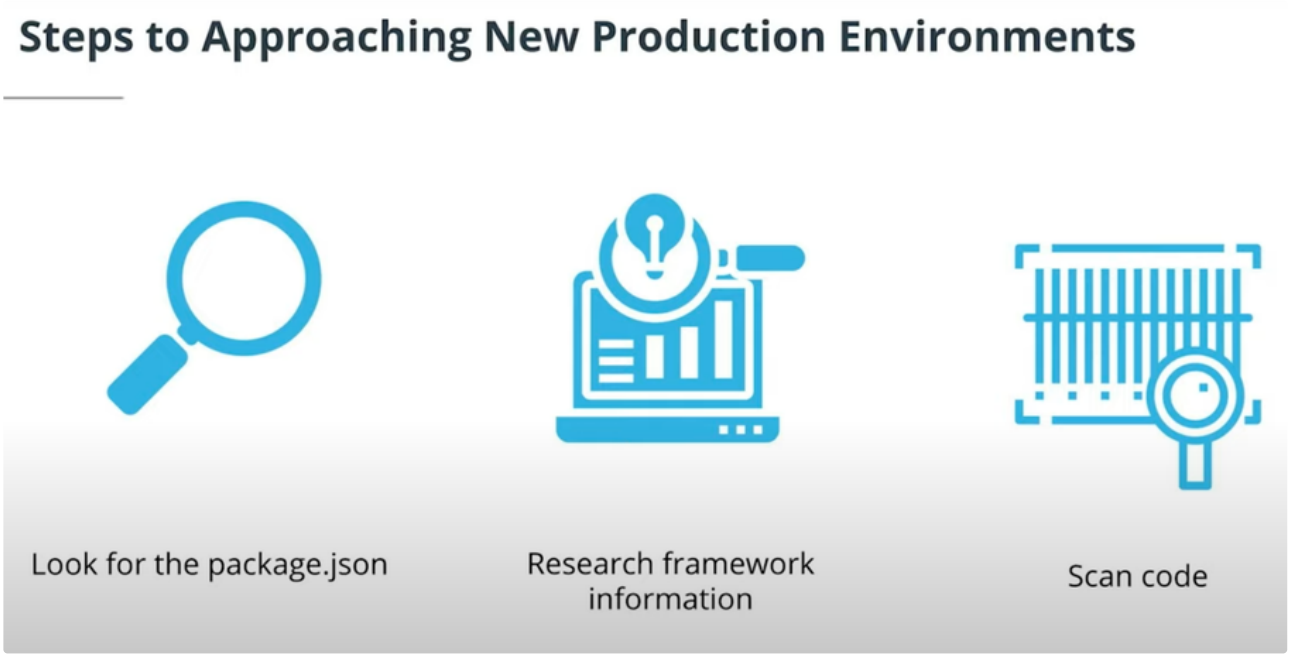


After we are done looking at the package JSON, we will research framework information. Especially if the framework is something like React or Angular, we can learn a lot about the production environment needs of our application by reading the documentation and doing a quick search engine research on it.



Lastly, we will be scanning the code. We will be looking for external communications such as APIs or databases.

Summary



Setting new production environment is not easy. This process gets easier the more you do it and experiment!

An expert would do the following in order to understand the application

- As an example, open the package.json in order to understand which framework is used.
- Research on that framework to understand at a high level the ideas behind the framework. It is important to read the documentation of the framework.
- Try to identify any communication to other applications in the code.

AWS Sign In

A step-by-step guide on how to sign in to your AWS account

1. Open your preferred web browser and go to the AWS Sign-In page at https://aws.amazon.com/.
2. Click on the "Sign In to the Console" button located in the top-right corner of the page.
3. Enter your email address or AWS account ID in the "Email address or mobile phone number" field and click on the "Next" button.
4. Enter your account password in the "Password" field and click on the "Sign In" button.
5. If this is your first time signing in, you may need to set up multi-factor authentication (MFA) for added security. Follow the prompts to set up MFA using your preferred method (such as SMS text messages or an authenticator app).
6. Once you have successfully signed in, you will be taken to the AWS Management Console, where you can manage your AWS services and resources.

That's it! You should now be signed in to your AWS account and able to start using the AWS Management Console. If you have any issues signing in or setting up MFA, you can refer to the AWS documentation or contact AWS support for assistance.

RDS Overview

Amazon Relational Database Service (RDS)

RDS (or Relational Database Service) is a service that aids in the administration and management of databases. RDS assists with database administrative tasks that include upgrades, patching, installs, backups, monitoring, performance checks, security, etc.

Database Engine Support

- Oracle
- PostgreSQL
- MySQL
- MariaDB
- SQL Server

Features

- failover
- backups
- restore
- encryption
- security
- monitoring
- data replication
- scalability

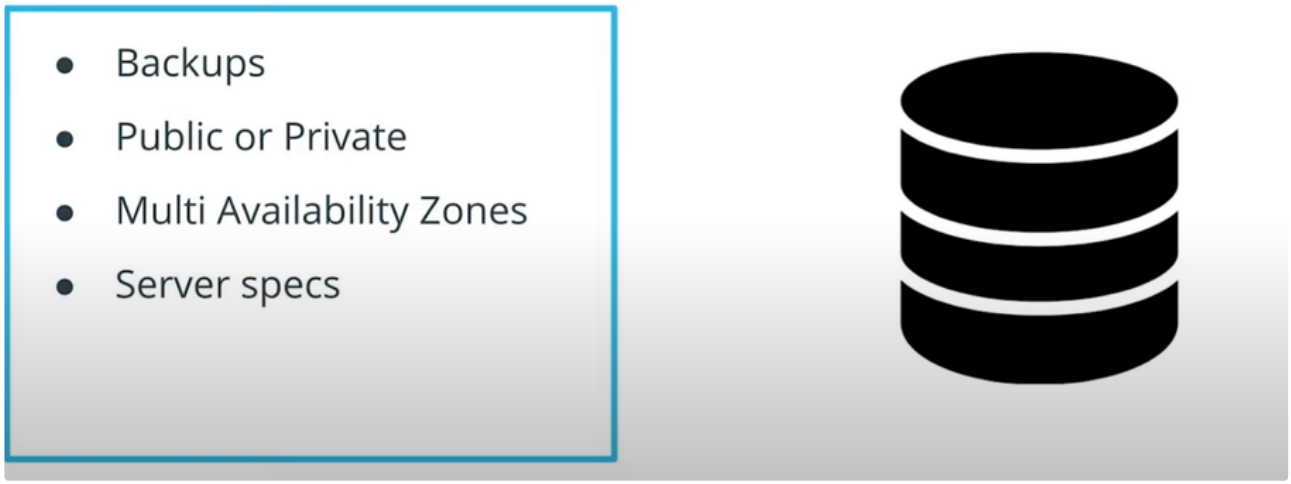
Summary

AWS provides many options for configuring databases. Its extensive configurability helps to deal with multiple different scenarios. However, a lot of those configuration options are not needed for a simple database. They will be useful when you start having a heavier workload!

How can we approach a complicated service?

1. Take a step back.
2. A quick Internet search for a simplified tutorial.
3. Experimenting to understand the features.

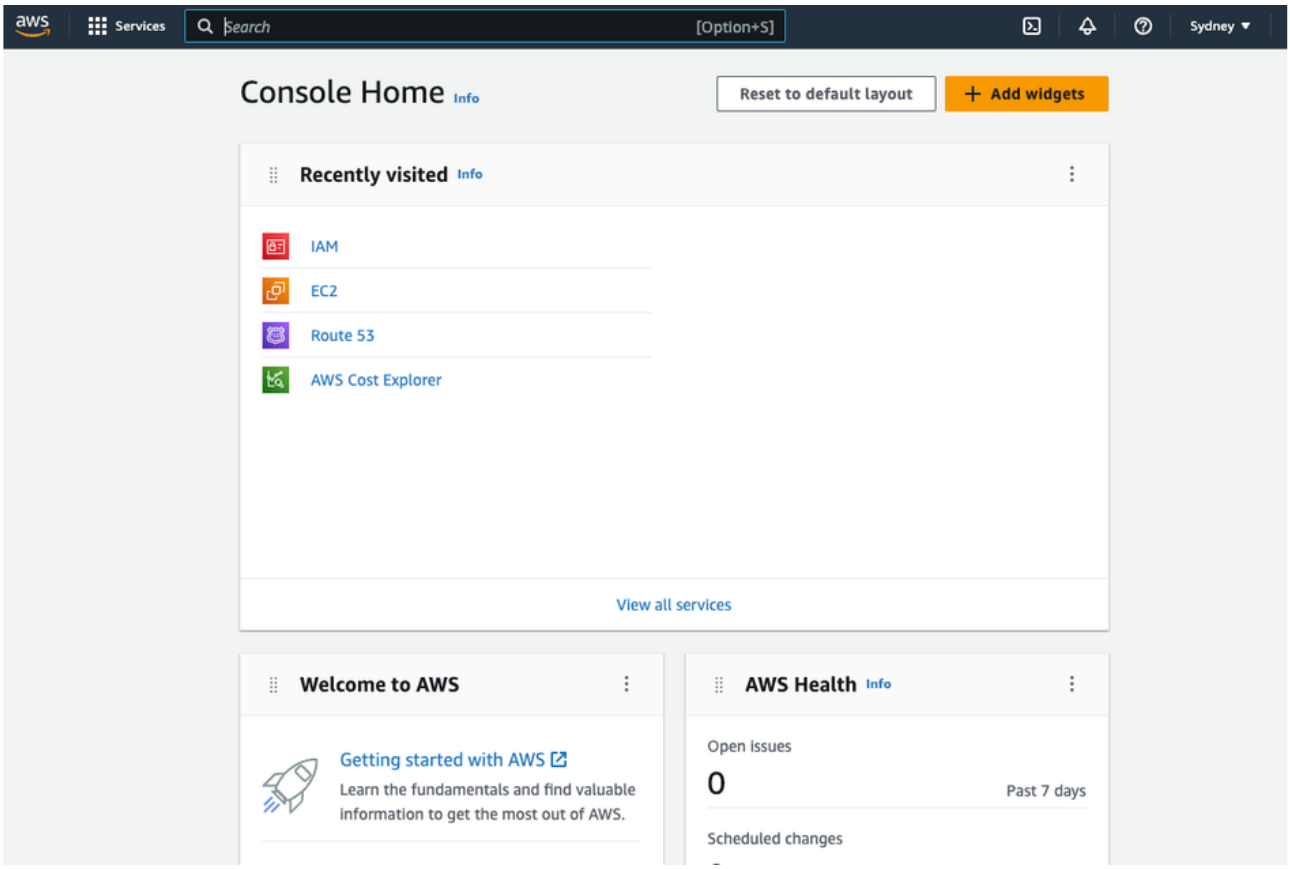
What do we need to configure?



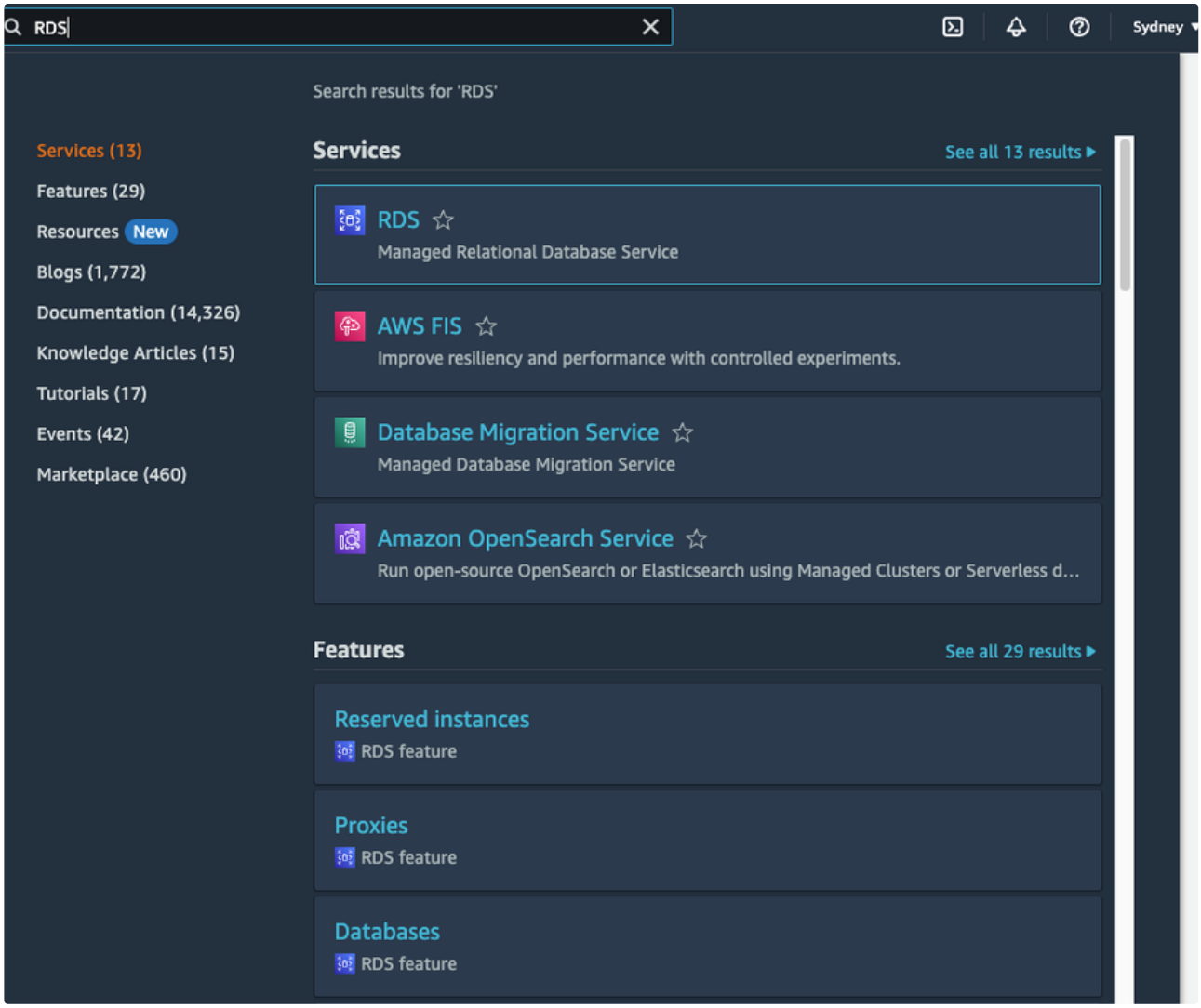
- **Backups**: A copy of your database can be made on a regular interval to avoid losing data if something goes wrong.
- **Public or private**: A database could be made available on the open web, or only within a private Internet network.
- **Multiple Availability Zones:** You can configure a database to be physically in multiple data-centers.
- **Server specs:** You can choose the size of the server.

Configuring a Database Demo

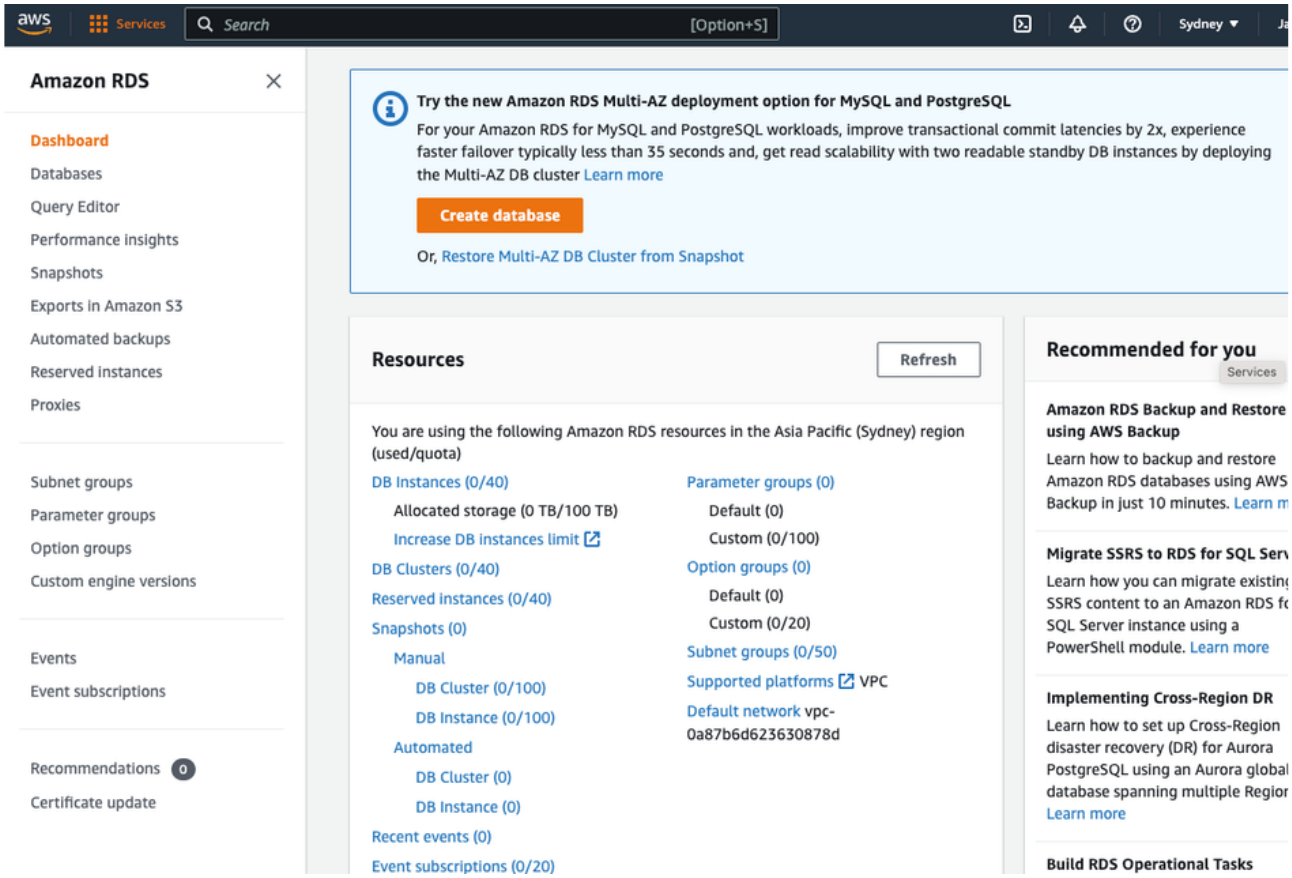
Let's dive a little bit into configuring a database. Let's try to understand this better and how we can do that. If this is the first time you are in the AWS console, take some time to understand your surroundings. AWS has a lot of services. Often, when I need to navigate to a specific service, I will use the search bar at the top.



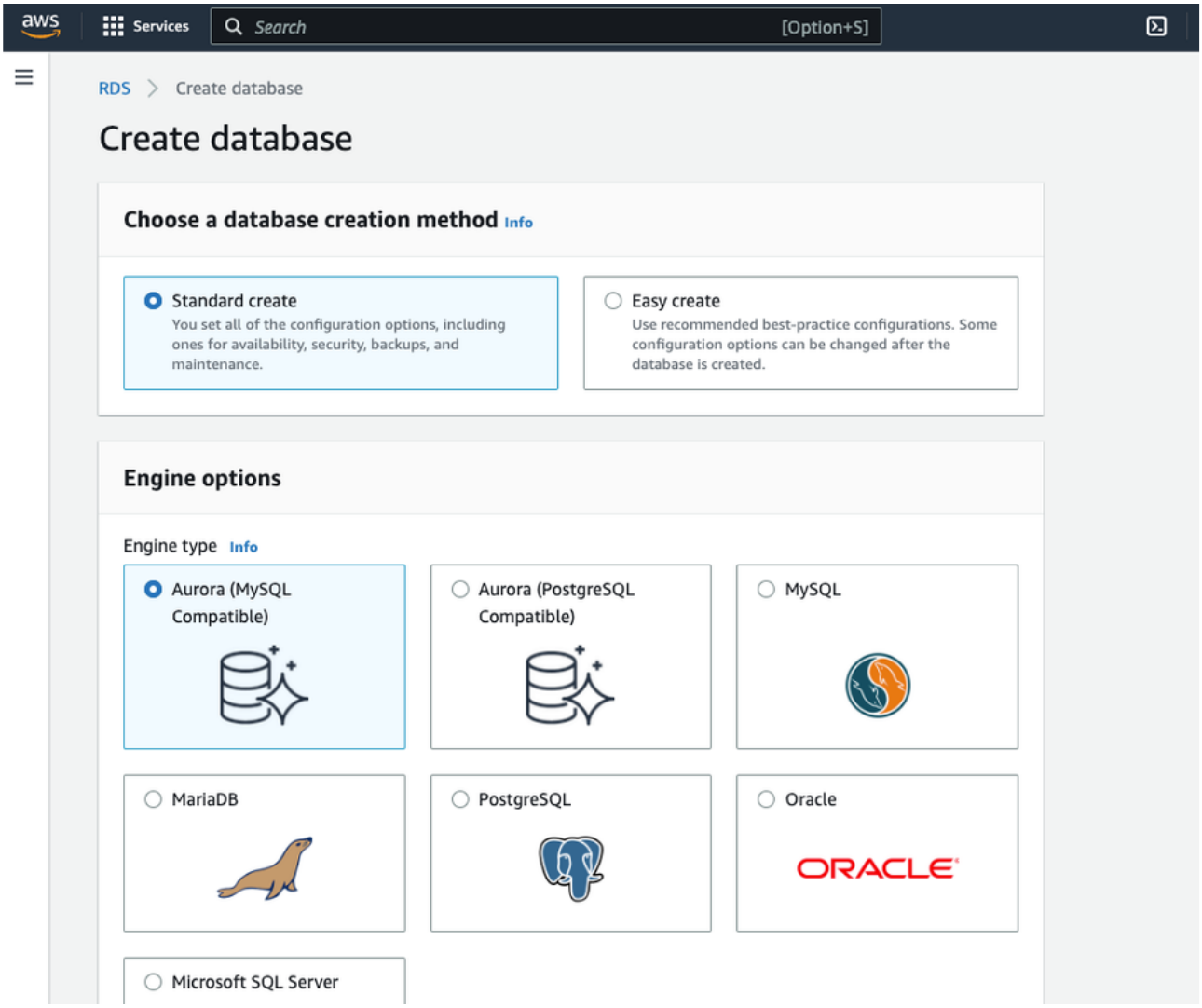
If you click on the Search bar and type, any acronym or name of the service, AWS does a really good job at finding it. I went ahead and typed RDS.



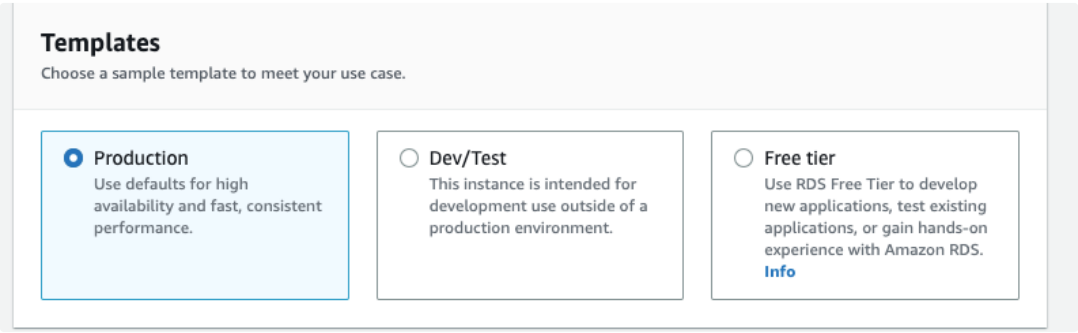
As you can see, the first service that I'm presented with is RDS. I will click this one.



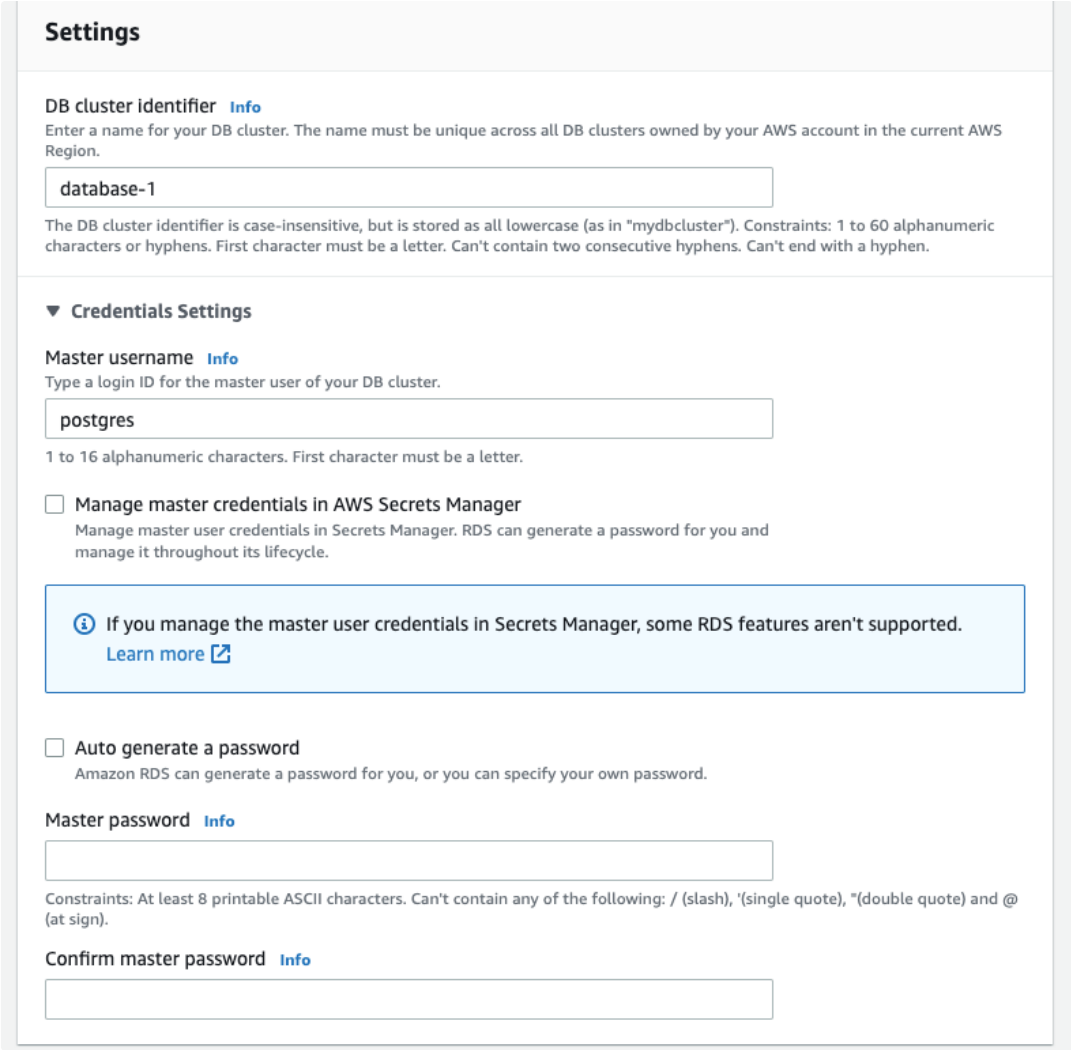
We are now taken directly to the RDS dashboard. Let's dive more into the topic of this section, which is how can we create a database? We will click this "Create Database" button.



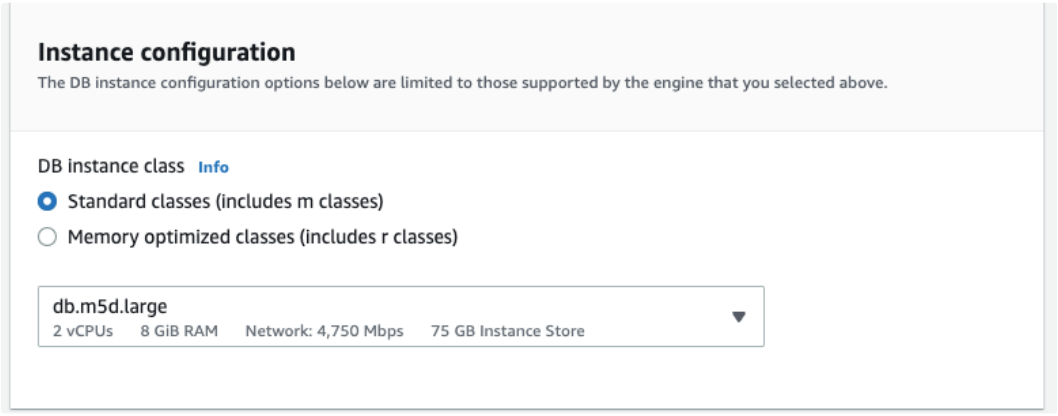
You have a couple of options when it comes to creating a database. You can choose the Standard Create or Easy Create. You can choose the different engines, which is the SQL-specific variant that you will be using. Let's scroll down a bit more.



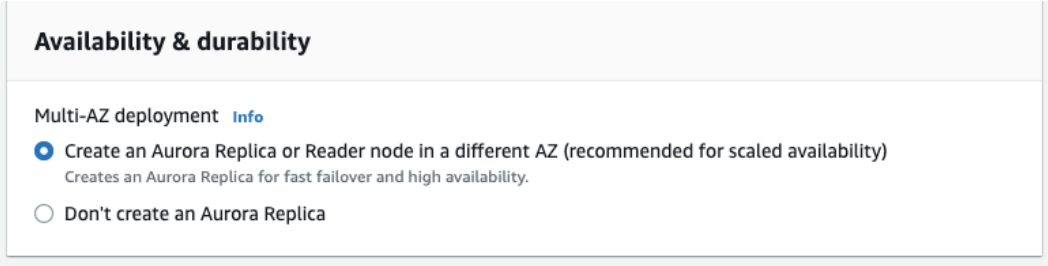
We can see also in the templates, the size of the server that we will use. You can use something that is big for production or something that is smaller for dev or testing. Let's scroll down again a bit more.



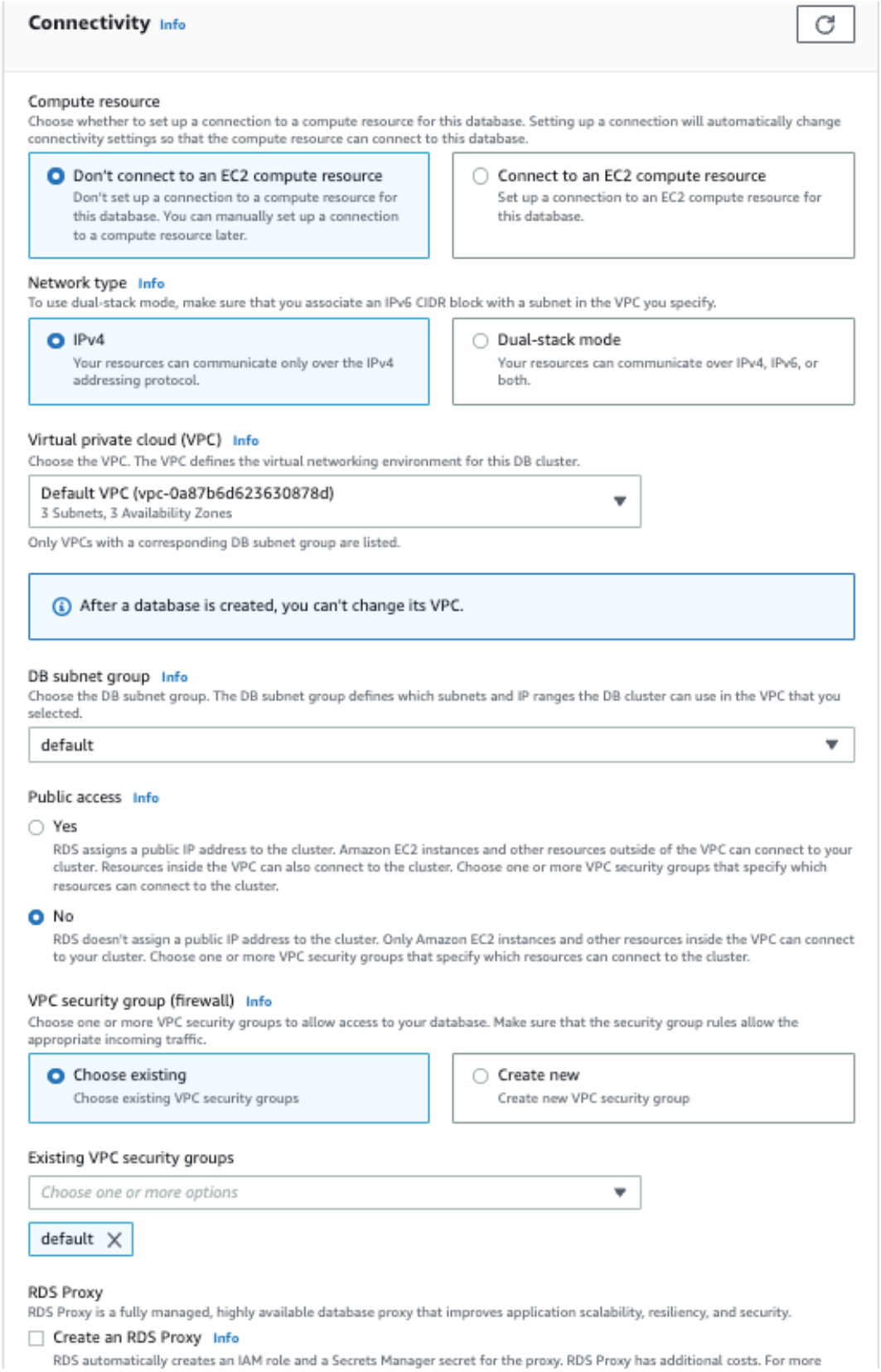
In the additional settings, we can also name the database server. We can add a master username, a master password. Let's go down a bit more. The other options we could configure it would be the database instance size. As your database server runs on AWS EC2, which is a on-demand server service by AWS, you can choose the exact size of the server.



Let's scroll down a bit more. We can also choose if we want to put it in multiple availability zones or one. Availability zones are different data centers that are independent from each other. We would normally try to put our database in two availability zones if we want to ensure some disaster prevention. Let's say if there's an earthquake in one of the data centers, our database would survive if it's in multiple availability zones. For most of the time, we will not need this. This is more for advanced production set-ups.



Let's go back down a little bit. We can also define the connectivity, which is the specific network on which your database is running. One thing to keep in mind is that you are seeing a lot of options here. Most of the time just taking the easy way and using the easy create option will do what you need.



But don't worry, if this feels overwhelming. We can make our database public or not. We can put it inside its own security group. Again, this is something aimed more at production setup. This was an overview of creating a database screen. Just again, I want to reiterate that this is a lot, so don't feel worried if you're seeing too much. With time, all of those options will become more familiar.

New Terms

- **Database backups:** A copy of your database taken on a regular interval to avoid losing data if something goes wrong.
- **Publicly available database:** A database could be made only available within a private Internet network or it could be available on the open web.
- **Availability zones**: Defines an area where AWS has a multitude of data-centers. You can configure a database to be physically in multiple Availability Zones.

Further Reading

- RDS resources: The official resource page of AWS RDS. This provides a great overview of everything you can configure.
- Postgres Documentation site: Great place to look for details on how you should manage and use a Postgres Database.

Exercise: Configuring a Postgres Database

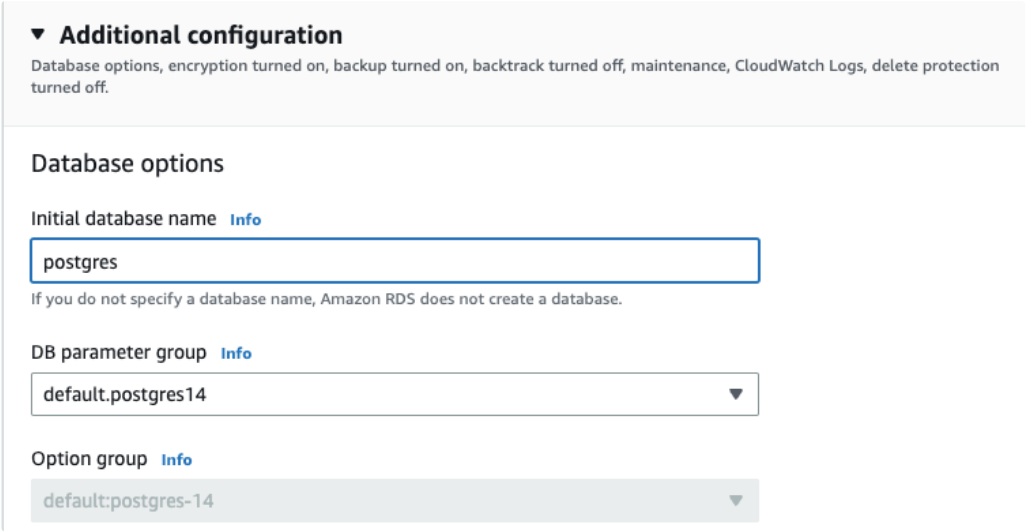
In this exercise, you will be tasked with configuring your own database in RDS and connecting it to a simple contacts API.

Step 1. Start a PostgreSQL Server instance in the AWS RDS console

1. Navigate to the RDS dashboard and create a PostgreSQL database with the following configuration, and leave the remaining fields as default.

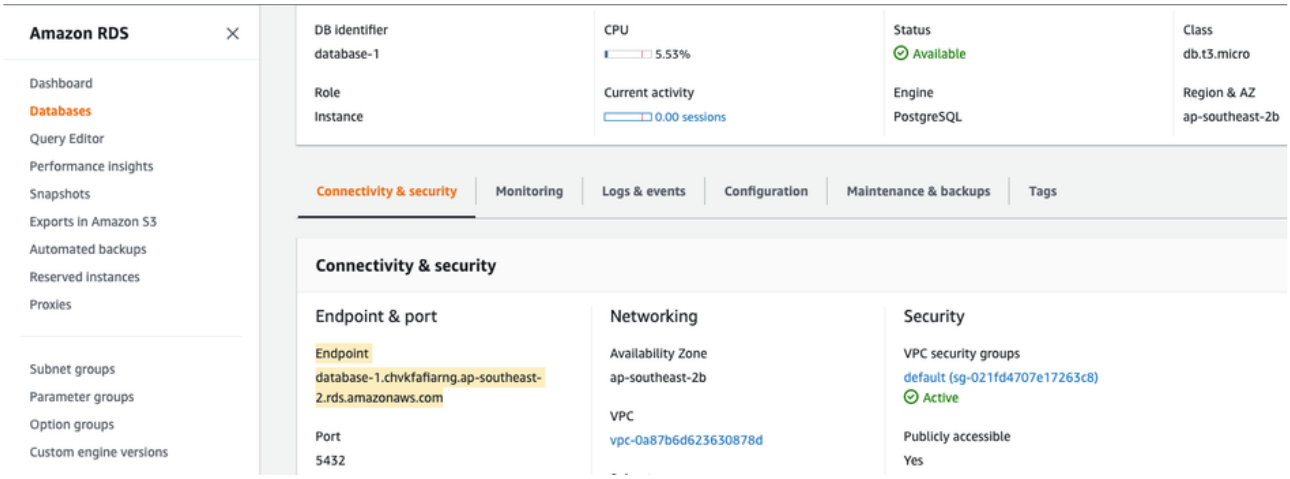
Field	**Value**
Database creation method	**Standard create**. Easy create option creates a private database by default.
Engine option	PostgreSQL 12 or 13, any release candidate
Templates	Free tier
Settings DB instance identifier, master username, and password	Your choice
Instance configuration DB instance class	Burstable classes with minimal size like db.t3.micro or db.t2.micro
Storage	Default
Connectivity VPC and subnet Public access VPC security group Availability Zone Database port	 Default YES Default No preferencce `5432`
Additional configuration Initial database name	 `postgres`

Leave the remaining fields as default and finish creating the database. * **Update from AWS**: Amazon RDS does not create a database if you do not specify a database name. Therefore, specify a database name in the **Additional configuration**, as shown in the snapshot below.

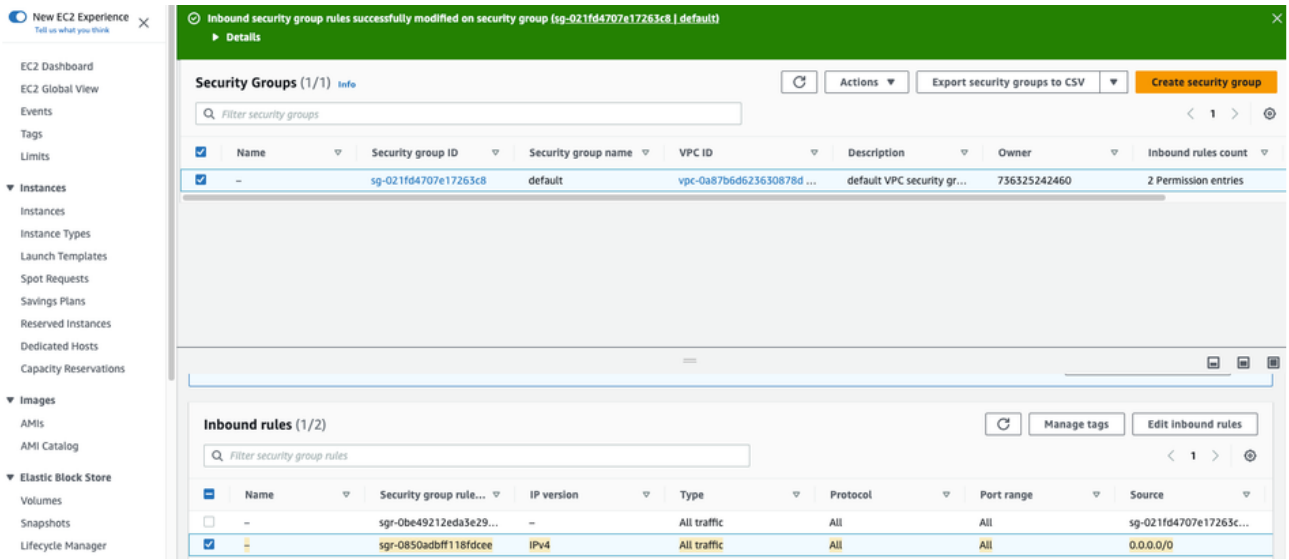


Go to the Additional configuration and provide a database name as 'postgres'

2. Once the database is created successfully, copy and save the database endpoint, master username, and password. It will help your application discover the database.



3**. Allow access to the database**: Edit the security group's inbound rule to allow incoming connections from anywhere (`0.0.0.0/0`). It will allow your local application to connect to the database.



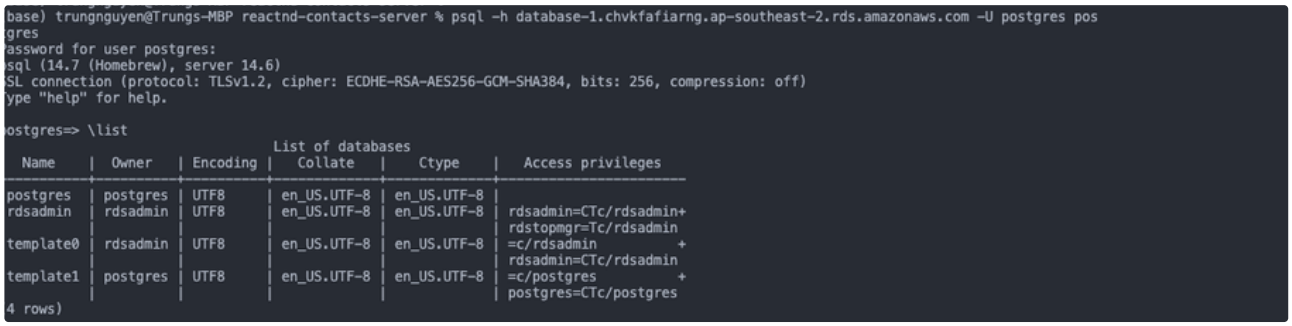
4. Test the connection using the PostgreSQL client.

```
# Assuming the endpoint is: database-1.chvkfafiarng.ap-southeast-2.rds.amazonaws.com

psql -h database-1.chvkfafiarng.ap-southeast-2.rds.amazonaws.com -U postgres postgres

# It will open the "postgres=>" prompt if the connection is successful.

# Provide the database password when prompted.
```

Later, when your application is up and running, you can run commands like:

```
# List the databases

\list

# Go inside the "postgres" database and view relations

\c postgres

\dt
```

or play around with some `psql` commands found here.



Step 2. Update the Connection String

Open the *reactnd-contacts-server/server.js file in your IDE*.

In this file, locate and replace the Sequelize connection string with the proper values to connect to your database. See an example below:

```
// Before
const sequelize = new Sequelize(
  "postgres://user:pass@example.com:5432/dbname"
);
```

```
// After
const sequelize = new Sequelize(
  "postgres://postgres:myPassword@database-1.chvkfafiarng.ap-southeast-2.rds.amazonaws.com:5432/postgres"
);
```

In the snippet above, we have assumed the following values:

Field	Value
database username	`postgres`
database password	`myPassword`
database name	`postgres`

database endpoint	`database-1.chvkfafiarng.ap-southeast-2.rds.amazonaws.com`

```
const sequelize = new Sequelize(
  'postgres://postgres:postgres@database-1.chvkfafiarng.ap-southeast-2.rds.amazonaws.com:5432/postgres'
);
```

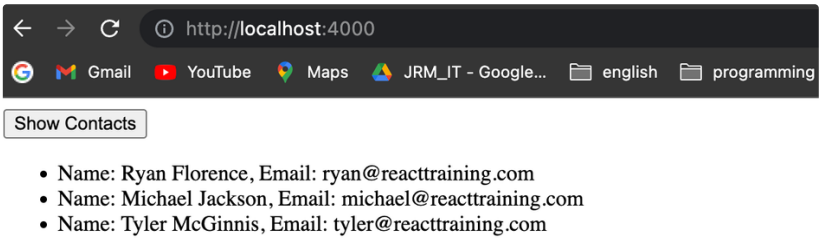
Step 4. Access the Application

Open CLI and enter

> npm install && npm run start

```
(base) trungnguyen@Trungs-MBP reactnd-contacts-server % npm run start

> start
> node server.js

Executing (default): SELECT 1+1 AS result
Connection has been established successfully.
Server listening on port 4000, Ctrl+C to stop
Executing (default): SELECT table_name FROM information_schema.tables WHERE table_schema = 'public' AND t
able_name = 'Users'
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.i
ndkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_inde
xdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid =
ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Users'
GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (ee0b1266-68b7-492d-91f2-4eab436899b3): START TRANSACTION;
Executing (ee0b1266-68b7-492d-91f2-4eab436899b3): SELECT "id", "name", "email", "avatarURL", "createdAt",
 "updatedAt" FROM "Users" AS "User" WHERE "User"."id" = '242dfef8-529a-11eb-ae93-0242ac130002';
Executing (ee0b1266-68b7-492d-91f2-4eab436899b3): COMMIT;
Executing (e1d981bf-945f-4694-a8b1-635fe30c421b): START TRANSACTION;
Executing (e1d981bf-945f-4694-a8b1-635fe30c421b): SELECT "id", "name", "email", "avatarURL", "createdAt",
 "updatedAt" FROM "Users" AS "User" WHERE "User"."id" = '299992c6-529a-11eb-ae93-0242ac130002';
Executing (e1d981bf-945f-4694-a8b1-635fe30c421b): COMMIT;
Executing (6e62e2b9-e337-453c-86d3-d8ecdd1c10d3): START TRANSACTION;
Executing (6e62e2b9-e337-453c-86d3-d8ecdd1c10d3): SELECT "id", "name", "email", "avatarURL", "createdAt",
 "updatedAt" FROM "Users" AS "User" WHERE "User"."id" = '307f0b0c-529a-11eb-ae93-0242ac130002';
Executing (6e62e2b9-e337-453c-86d3-d8ecdd1c10d3): COMMIT;
```

Results:



Supporting Materials

You can download the code from the link: database-code

Elastic Beanstalk Overview

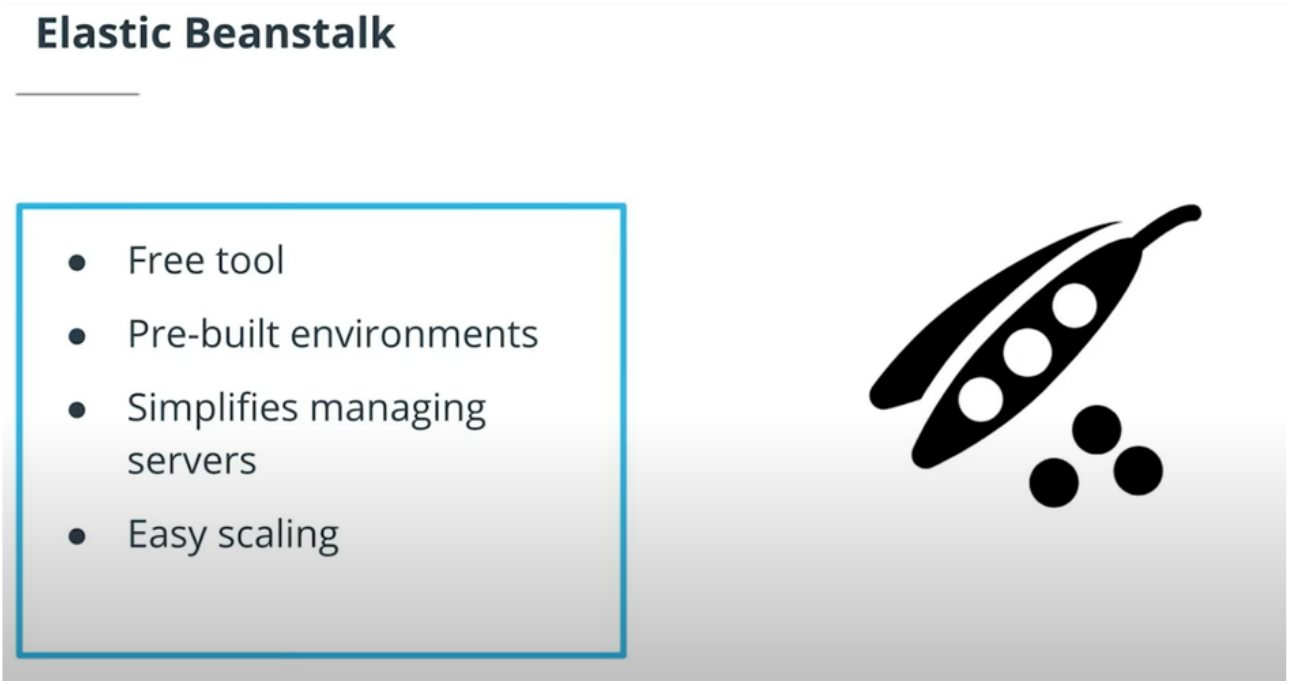
Elastic Beanstalk Overview

AWS Elastic Beanstalk is a service (platform as a service) that allows you to run your web application on the AWS cloud without worrying about scaling or configuring the underlying virtual machines (web servers).

AWS Elastic Beanstalk supports Java, .NET, PHP, Node.js, Python, Ruby, and Go platforms. You just need to upload an application zip file to the AWS Elastic Beanstalk and configure some settings to make the app run. We can either upload the zip file using the AWS UI (web console) or use the explicit commands in the local terminal.

After uploading the application zip file, Elastic Beanstalk will handle deploying the application to the (right-sized) EC2 VMs, load balancing, auto-scaling, and application health monitoring.

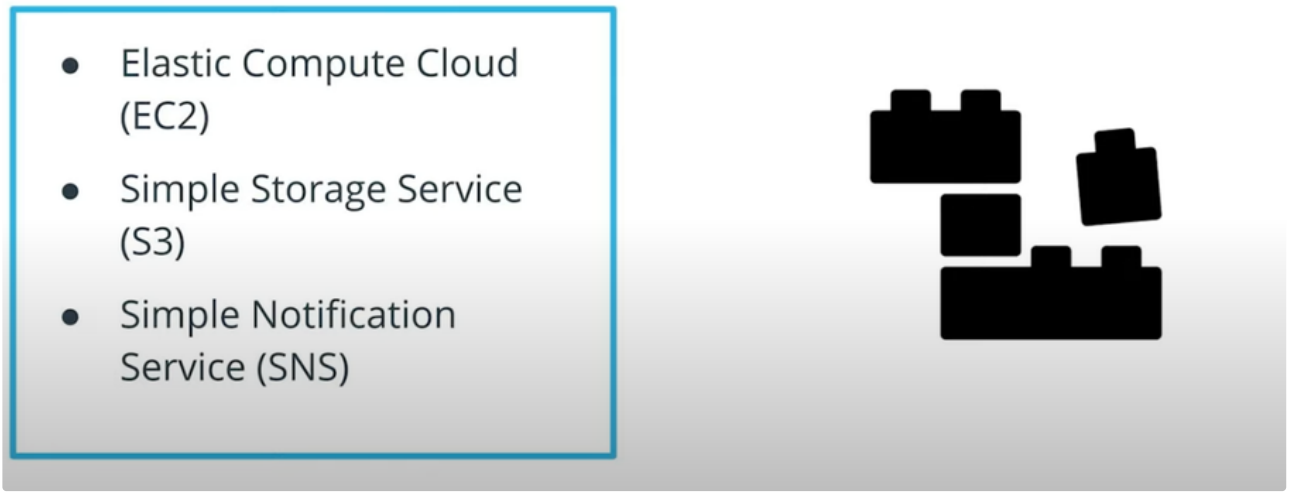
Ideally, industry standards recommend using the terminal, so we will see how to install and use EB CLI in the current section.



Elastic Beanstalk offers the following advantages:

- **Free**: You only pay for the servers that elastic beanstalk uses. The extra tools are free of charge.
- **Pre-built Environments**: Most major programming languages are supported out of the box.
- **Simple Server Management**: Security updates and system upgrades are done for you.
- **Easy Scaling**: If you need to provision extra servers, you can quickly change your configuration.

What does Elastic Beanstalk use?



- **Elastic Compute Cloud (EC2)**: Used for hosting servers.
- **Simple Storage Service (S3)**: Used for storing application code and sending it to other servers.
- **Simple Notification Service (SNS)**: Provides a way to notify you of events inside the environment.

New Terms

- **Elastic Beanstalk Environments**: Pre-configured servers that can be deployed easily with all the necessary software to run applications.
- **Elastic Beanstalk Application**: An application that you upload into the Elastic Beanstalk Environment.
- **Scaling:** The ability to provision more or fewer servers to fit the traffic on your application.

Further Reading

This is a great in-depth tutorial by AWS on how to use Elastic Beanstalk: Getting started using Elastic Beanstalk

Configuring Elastic Beanstalk Environment Properties

Configuring Elastic Beanstalk Environment Properties

Elastic Beanstalk provides an easy solution to hide some sensitive API keys or other secrets. You should always be sure to remove the following information from your application's source code:

- Private API keys
- Database connection strings
- Environment-specific information

Have a look at the AWS documentation provided in the section below for more details on how to create environment variables using Elastic Beanstalk.

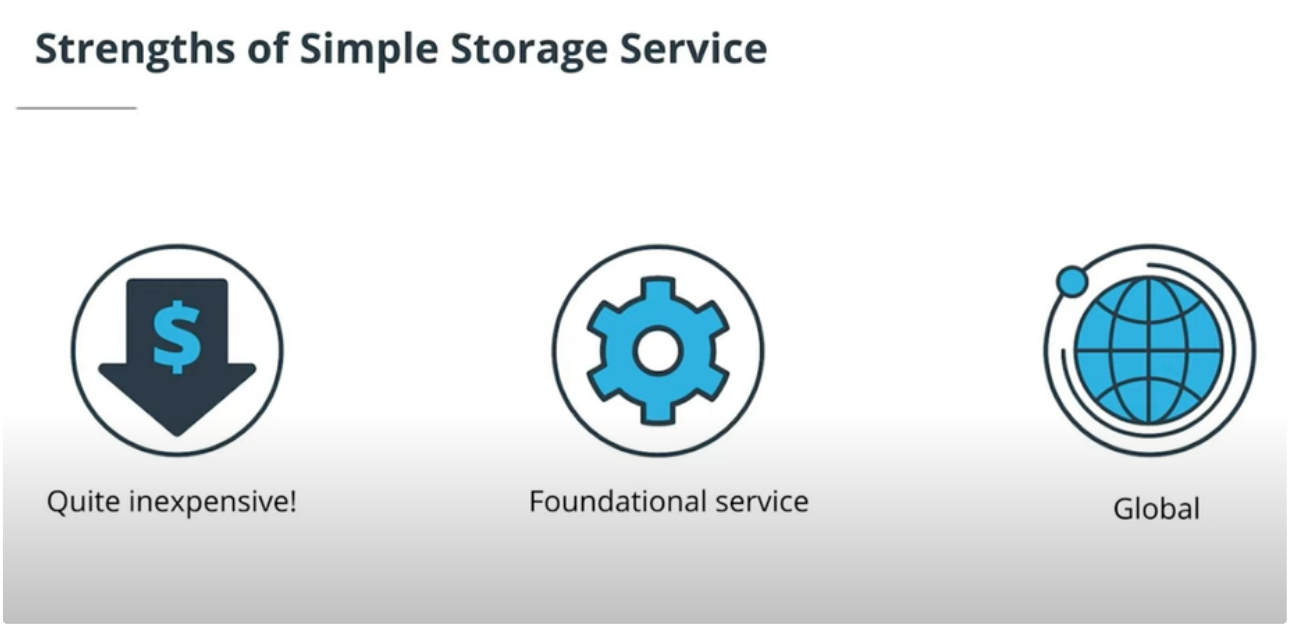
Let's Read Some Documentation about Environment Variables

This page shows different methods to pass the Environment Variables to the Elastic Beanstalk environment.

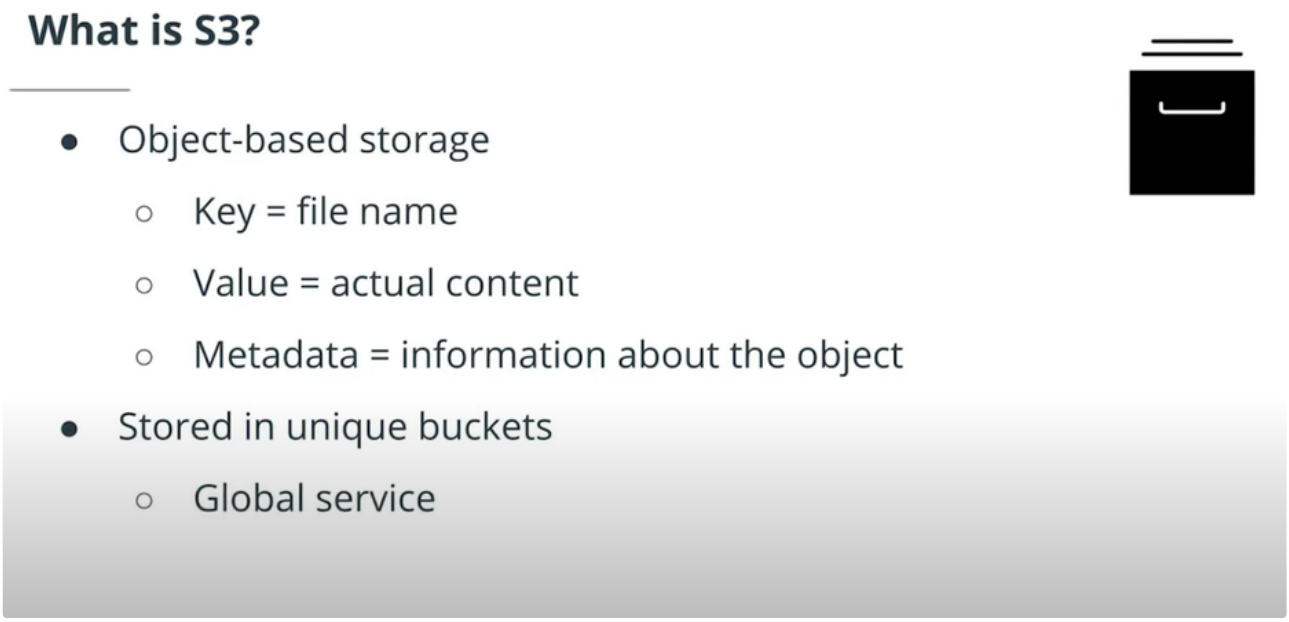
S3 Overview

Configuring S3 for Web Hosting

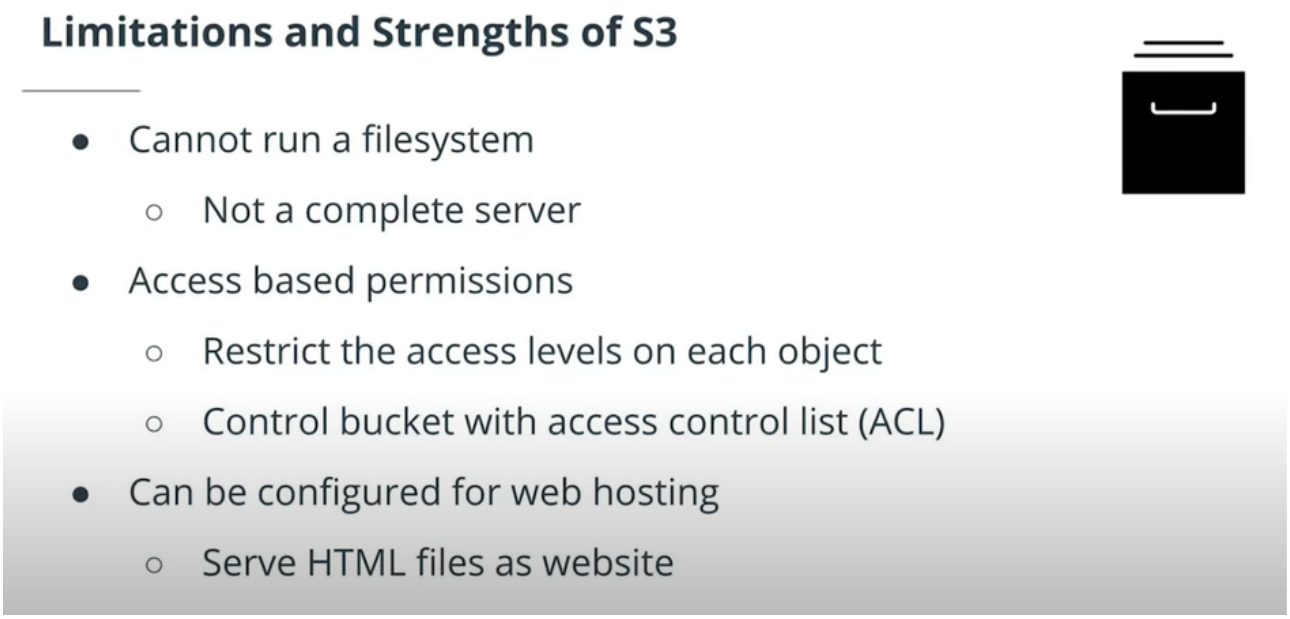
Let's see how we can configure S3 for Web Hosting. But first of all, let's dive into the strengths of history, otherwise known as Simple Storage Service. The first strength is that it's quite inexpensive. People really like it as it can provide inexpensive hosting of different types of files. It's also a foundational service. That means that other AWS service relies on S3 for storing different things and different objects, such as Elastic Beanstalk relies on it for storing the code itself. Lastly, it is a global service. That means that S3 is not restricted to a specific AWS region. It is a service that is available to all regions.



You might be wondering, what exactly is S3. S3 is an object-based storage. That means that the key is the file name and the value is the actual content of the object, the content of the file itself. We can also store metadata, which is information about your object, such as the last person who uploaded the object or the timestamp at which it was updated or uploaded. One thing to note is that S3 are all unique buckets. This means that the name must be unique. Since it is a global service, I think the same name would not work.



There are some limitations in S3. S3 cannot run a file system. When we mean a file system, we're referring basically to your Mac, Linux, or Windows file system. It cannot be a complete server. One of the strengths, however, of S3 is that it is managing permissions based on access. You can restrict access on each object, or you can restrict it also on the global bucket in itself, and this is done by using what we call an ACL, an access control list. One of the things we care the most, one of the strings we liked the most about S3, that it can be configured for web hosting, and being so cheap, it's one of the best ways to host a website. It can serve HTML files, which will act as your website after.



Summary

While we could use a web server that we set up on Elastic Beanstalk to serve HTML files, there are other options like AWS S3 for this purpose. It is possible that you have already heard about S3 since it is a really popular service. Here are some of its strengths:

- Inexpensive
- Foundational service on AWS upon which many other services are built
- Global and available in all regions

S3 Stands for Simple Storage Service. It is AWS's file storage service. S3 is different from a hard drive. It can be referred to as **object-based storage**.

By object-based, we mean that the file name is a key and the value of that key is the actual content of the file. Metadata, which includes information about the object, such as owner, date created, and other important information, is also stored.

Limitations and Strengths of S3

- **S3 can't run a file system:** S3 is just meant to serve files and cannot act as an operating system.
- **Fine-grained permission system:** We can control the access to the bucket with Access Control List (ACL) policy, which is a file written in JSON or yml.
- **Configurable for web hosting:** We can serve static files like HTML and CSS on S3.

New Terms

- **Access Control List (ACL) policy.** This is a file written in JSON or yml that can be used to grant or restrict access to an S3 bucket. This is also used in other AWS services.
- **Object Storage**: Storage that behaves differently than a file system organizing files as objects.
- **Metadata**: Data about data. Includes information about files such as owner, date created, and other important information.
- **S3 - Simple Storage Service**: AWS's file storage solution that drives many different connections between AWS services.

S3 - Create a Bucket

What is an S3 Bucket?

Let us start with a demo of using S3. Before that, let me give you a brief introduction to what S3 is. Please direct to the link aws.amazon.com/s3. S3 stands for Amazon's Simple Storage Service, S3. Here is a diagram showing the overall high-level picture of what S3 is.

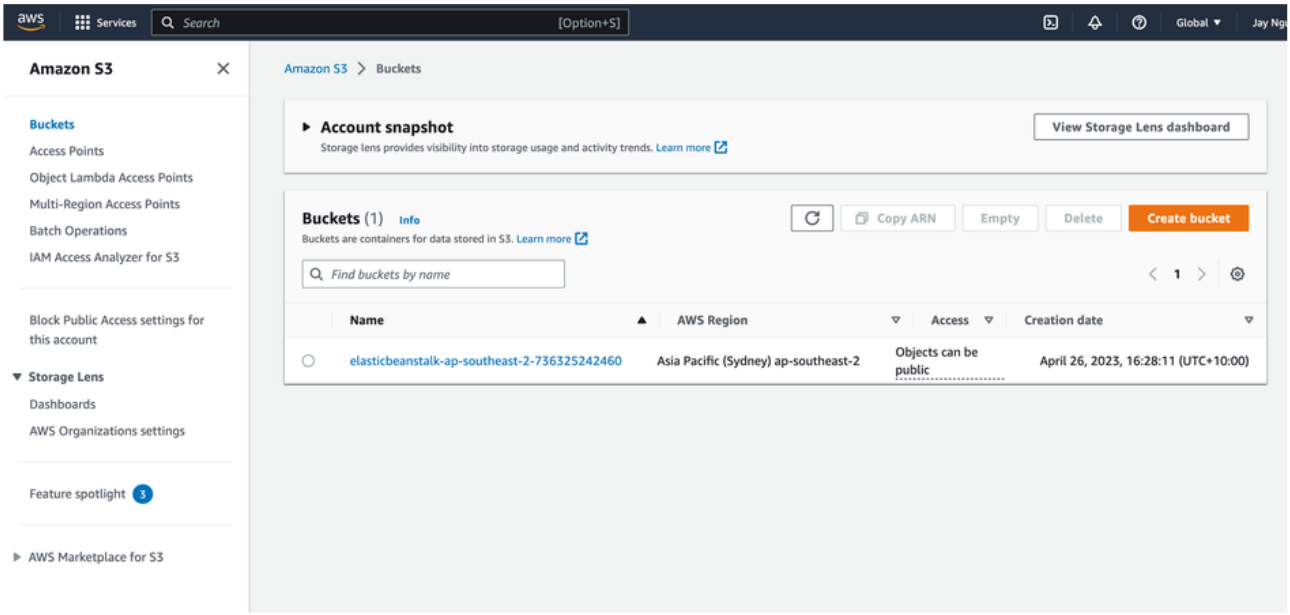
As you can see here, an S3 is like a bucket, or you can think of it as a file system in the Cloud where you can upload a variety of documents or files, for example, analytics data, text data, log files, application data, media files such as videos or pictures, backup, and archived files. You can upload all those files into a single bucket or into multiple buckets as you like. These files would be treated as individual objects. When you upload a file to an S3 bucket, it gets a unique URL that can then be used to access that particular file object.

One bucket can hold thousands or more objects. Once the bucket is created, we can define the policy for accessing. For example, who can access the content in the bucket, where the bucket resides, the various management options, and uploading options. For example, we can have the complete access control. Also, S3 provides different storage classes. If you have some data, which is least frequently used, we can store the data in a different class, for example, S3 Glacier class. We can replicate the data in the S3 across multiple regions, having multiple data centers. We can access the data in the S3 bucket from on-premises or Virtual Private Cloud. The data in the S3 bucket is protected, it's secured, and you can gain the visibility into your storage and the matrix.

Create an S3 Bucket

Steps to Create an S3 Bucket

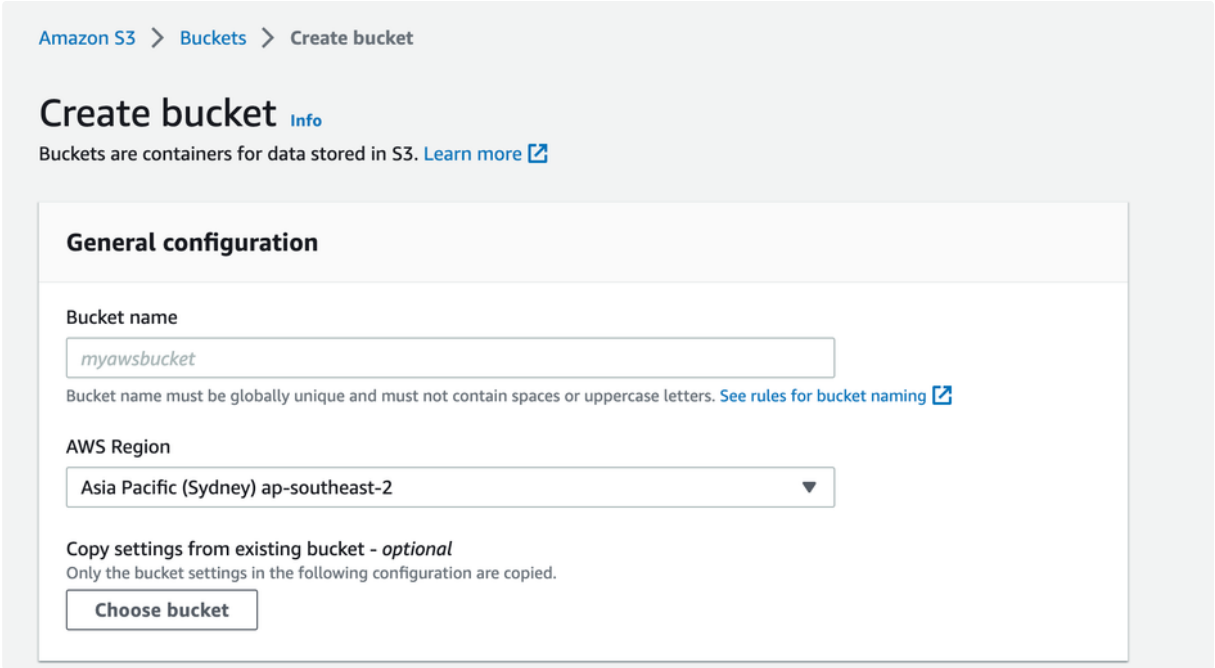
1. Navigate to the S3 dashboard, and click on the **Create bucket** button. It will launch a new wizard.



S3 service → Buckets dashboard.
View all of the S3 buckets in your account
(S3 is a global service, not a region-specific).

We create a bucket first, and later we upload files and folders to it.

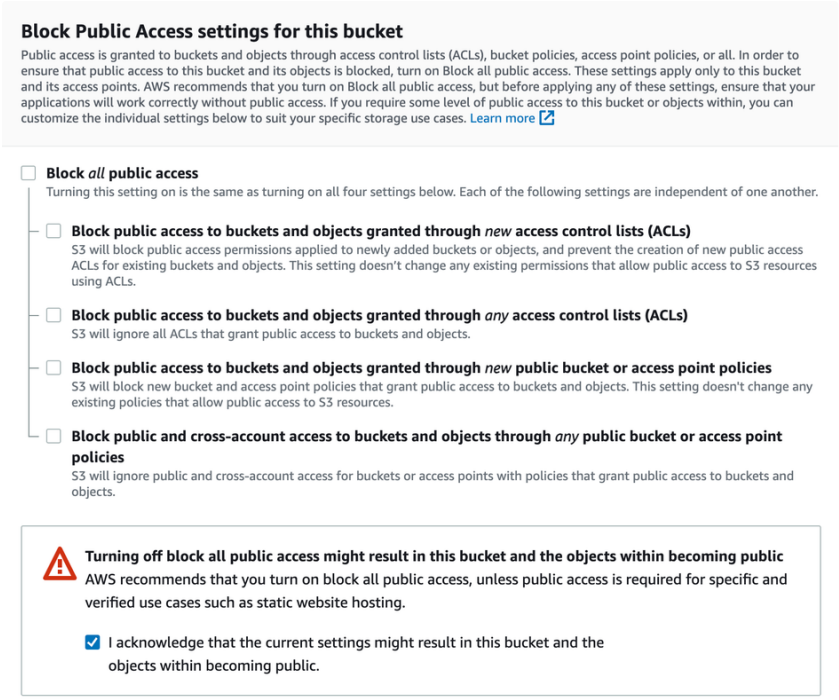
2. **General configuration** Provide the bucket-name and the region where you want to locate the bucket. The bucket name must be unique worldwide, and must not contain spaces or uppercase letters.



Create a bucket - Provide general details

3. **Public Access settings**

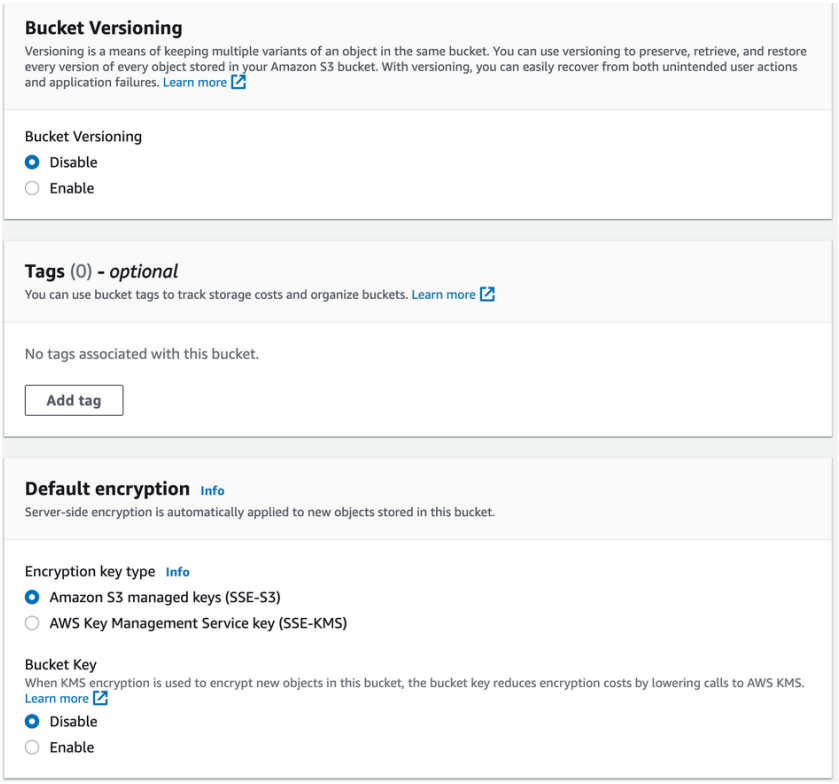
You can choose public visibility. Let's uncheck the *Block all public access* option.



Create a bucket - Make it public

4. **Bucket Versioning and Encryption**

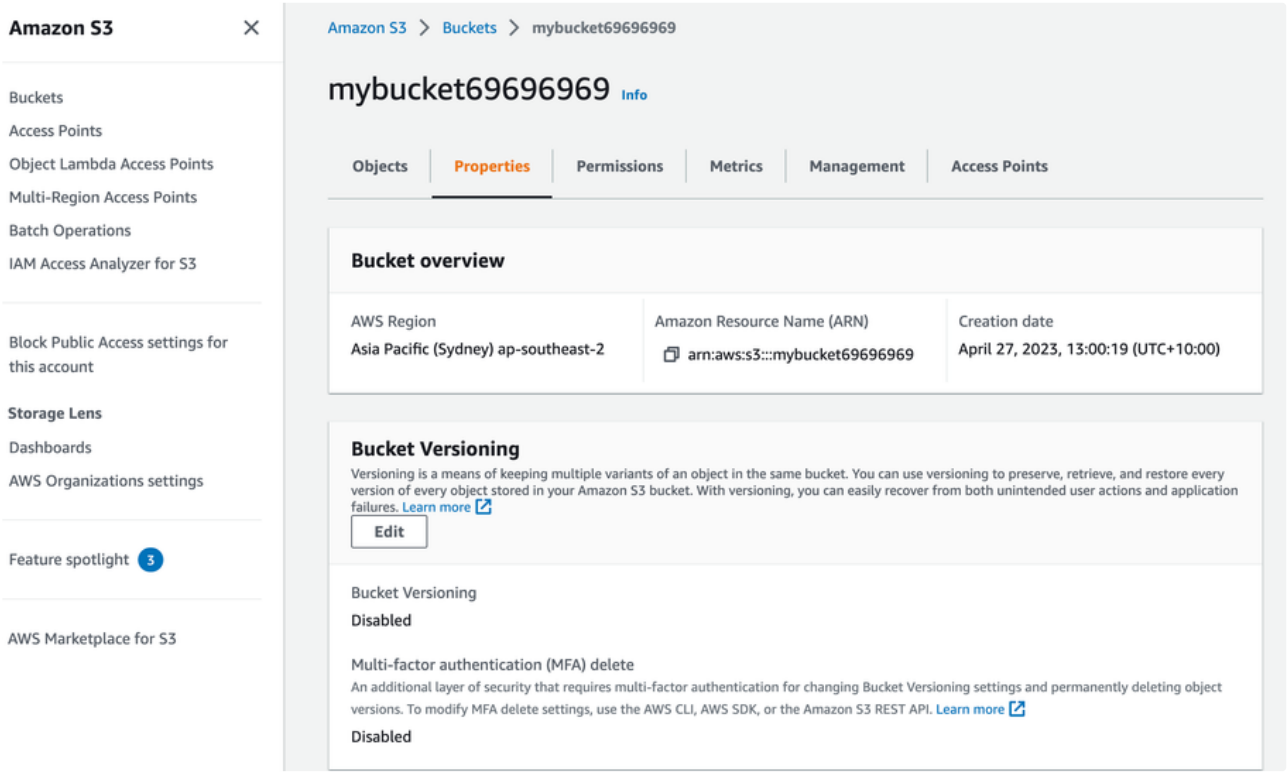
1. Bucket Versioning - Keep it disabled.
2. Encryption - If enabled, it will encrypt the files being stored in the bucket.
3. Object Lock - If enables, it will prevent the files in the bucket from being deleted or modified.

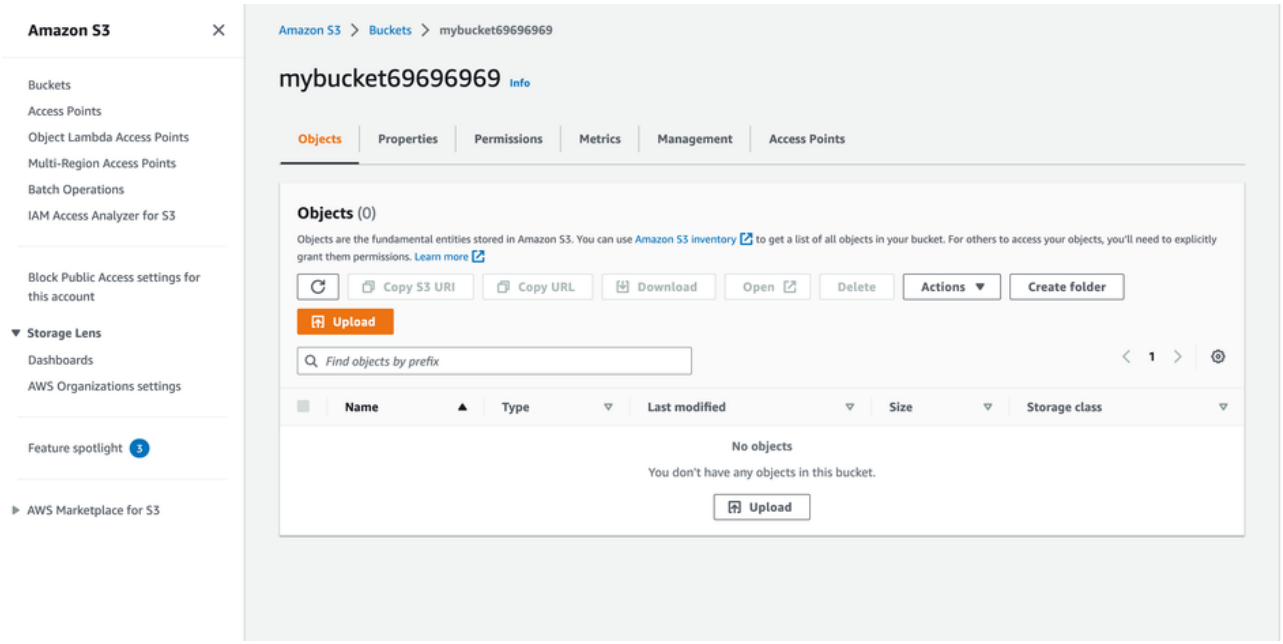


In the snapshots above, we have created a public bucket. Let's see **how to upload files and folders to the bucket**, and configure additional settings.

Upload File/Folders to the Bucket

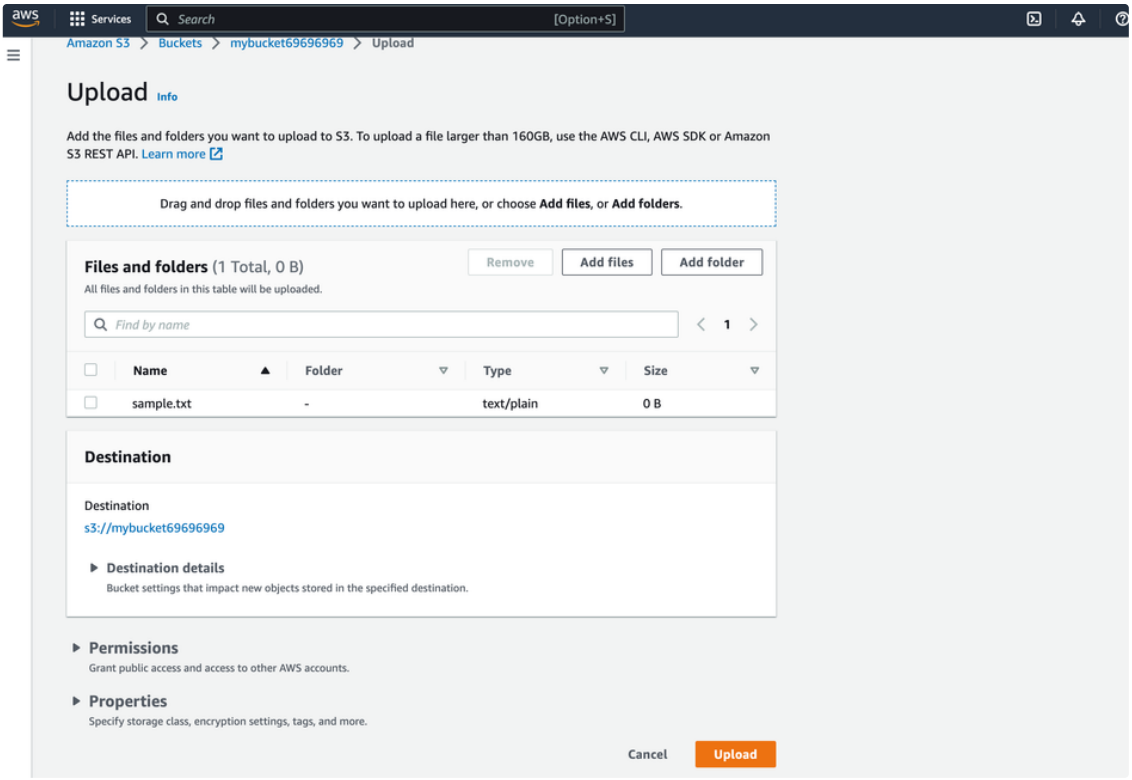
From the S3 dashboard, click on the name of the bucket you have created in the step above.



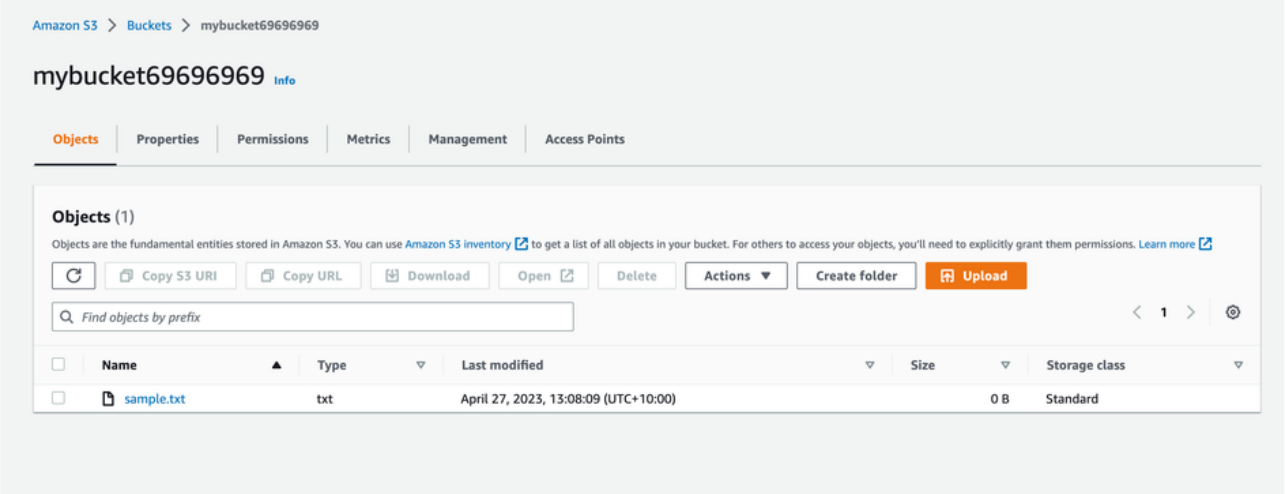


In the snapshot above, it shows that the bucket is in the Region: `Asia Pacific (Sydney) ap-southeast-2`, and it has a unique Amazon resource name (ARN): arn:aws:s3:::mybucket69696969. You can view more details of the bucket, in the tabs next to the bucket overview: **Objects, Properties, Permissions, Metrics, Management, and Access points**. Leet's upload a sample file to the bucket:

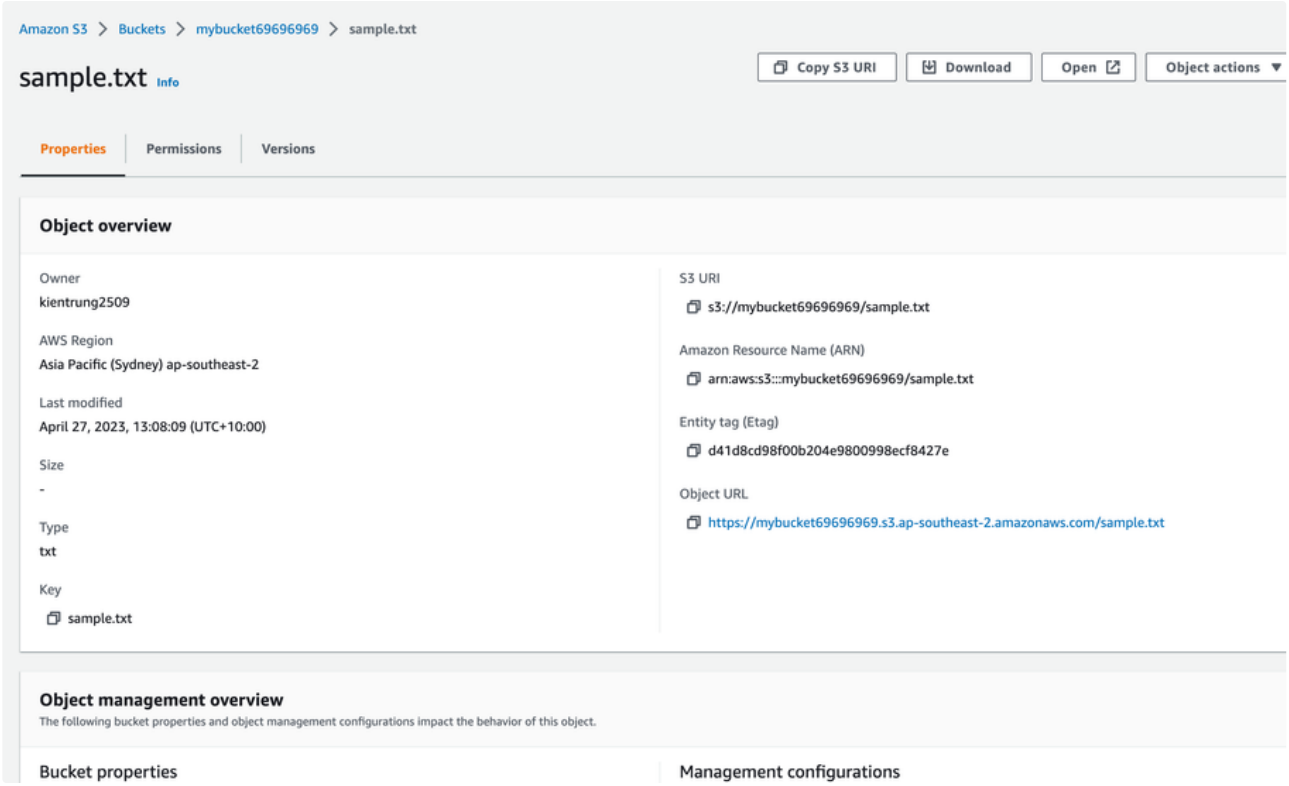
1. Click on the **Upload** button to upload files and folders into the current bucket. You can create and use a **sample.txt** file.



2. Click Upload button and you can see like this:



3. Click on the file name to view the file-specific details, as shown below.



Details of an individual file (object)

Details of an Existing Bucket

1. Properties

There are several properties that you can set for S3 buckets, such as:

- Bucket Versioning - Allows you to keep multiple versions of an object in the same bucket.
- Static website hosting - Mark if the bucket is used to host a website. S3 is a very cost-effective and cheap solution for serving up static web content.
- Requester pays - Make the requester pays for requests and data transfer costs.
- Server access logging - Log requests for access to your bucket.
- **Permissions**

It shows who has access to the S3 bucket, and who has access to the data within the bucket. In the example snapshots above, the bucket is public, meaning anyone can access it. Here, we can write an access policy (in JSON format) to provides access to the objects stored in the bucket.

2. Metrics

View the metrics for usage, request, and data transfer activity within your bucket, such as, total bucket size, total number of objects, and storage class analysis.

3. Management

It allows you to create life cycle rules to help manage your objects. It includes rules such as transitioning objects to another storage class, archiving them, or deleting them after a specified period of time.

4. Access points

Here, you can create access endpoints for sharing the bucket at scale. Using an endpoint, you can perform all regular operations on the bucket.

Setting up a Production Environment Recap

What We Have Learned

You have learned a lot in this lesson! The first step in deploying an application is always to layout a production environment. This is important because, in order to build scripts and a pipeline, we need to understand what we are working with and where we need to deploy.

You have also learned three important AWS services that we will use in order to deploy applications throughout this course:

- **AWS RDS** is useful for hosting our database, and we do not need to worry about scaling and backups.
- **AWS Elastic Beanstalk** is useful because it provides us with a set of tools that make it simpler to run a server.
- **AWS S3** provides fast and inexpensive web hosting for Front-End applications.

By using a combination of these services, you have deployed all the major portions of an application. You have done this with the help of the AWS console, but starting from the next lesson, we will learn how to do this with the CLI tools. As you gradually build your understanding of the services you just used, you will get more accustomed to them and will be able to automate all the operations that you need to execute.

Glossary-Deployment

New Terms in This Lesson

Database Backups: A copy of your database taken on a regular interval to avoid losing data if something goes wrong.

Publicly available Database: A database could be made only available within a private internet network or it could be available on the open web.

Availability zones: Defines an area where AWS has a multitude of data-centers. You can configure a database to be physically in multiple Availability Zones.

Elastic Beanstalk Environments: Pre-configured servers that can be deployed easily with all the necessary software to run applications.

Elastic Beanstalk Application: An application that you upload into the Elastic Beanstalk Environment.

Access Control List (ACL) policy. This is a file written in JSON or yml that can be used to grant or restrict access to an S3 bucket. This is also used in other AWS services

Object Storage: Storage that behaves differently than a file system organizing files as objects.

Metadata: Data about data. Includes information about files such as owner, date created, and other important information.

S3 - Simple Storage Service: AWS's file storage solution that drives many different connections between AWS services.

Interact with Cloud Services via a CLI

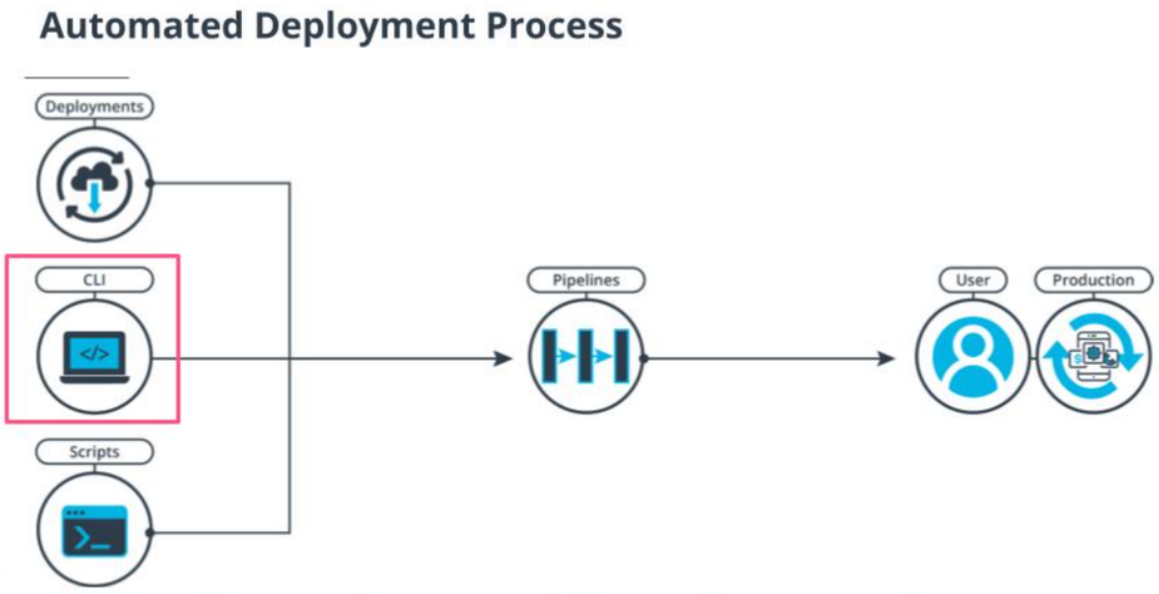
Introduction-Cloud Service

Lesson Introduction

Great job on making it so far! Deploying all those AWS services is quite an accomplishment in itself! We will now start to gain more confidence and manipulate our environment through the use of a command-line interface (CLI)! We will learn the following things in this lesson:

- How to use a CLI and read its documentation
- How to use the AWS Beanstalk CLI
- How to use the AWS CLI to update an S3 bucket

As shown in the image below, we will focus on the CLI throughout this lesson.



CLI in Automated Deployment Process

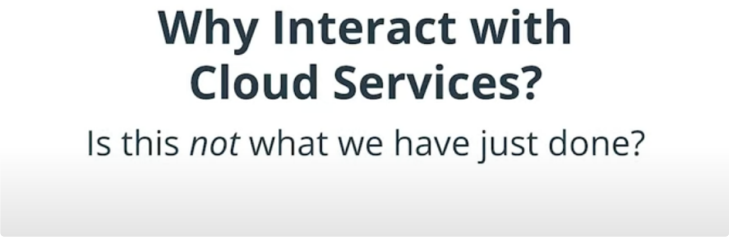
Creating Resources in AWS Cloud

There are multiple ways to create AWS cloud resources, such as:

1. AWS web console
2. AWS CLI
3. Language specific AWS SDK

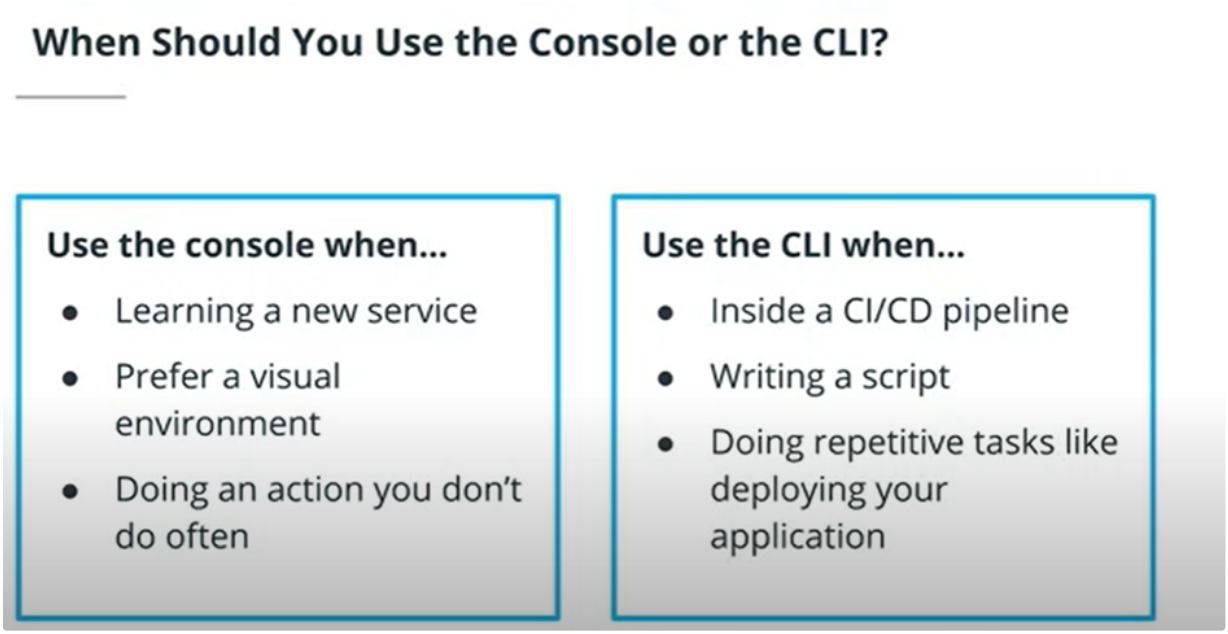
In this lesson, we will learn to use AWS CLI.

Why Interact with Cloud Services?



Let's see why we will interact with Cloud services. You might be wondering, is this not what we have just done? In a way, you are right. However, interacting with Cloud services with the AWS console is not normally the way we would go about it when deploying an application or really diving deeper into the department process. We will do this with a command-line interface. What can we do with the command-line interface? We can do a lot. But it does come back to what we were saying before. Mainly it's interacting with Cloud services. We can deploy code to Elastic Beanstalk or update the HTML to Simple Storage Service. We can also check the health of a service. We can see if it's responding properly, can navigate the logs, and we can get general information. Deluxe is an example, similar to checking the health, but we can also get, let's say, the latest version of our application. We could get a lot more information on this.

At this point, you might be wondering when you should use the AWS console or the command-line interface. Let's see a couple of examples. You should use a console when you are learning in your service. This is advisable because learning a new service through the use of a CLI, can be sometimes really intimidating as a lot of command coder. You should also choose a console when you want a visual environment, just because you prefer that environment and you feel that you're more competent in it, that is also a totally valid reason. Lastly, you should use a console when you're doing an action that you don't do often. For example, let's say you had been working in RDS for a couple of months and you're getting good, you're using the CLI with it, but your manager asks you to create a DynamoDB, which is another database offered on AWS. You should start by using the console just at the beginning. But let's contrast a little bit and see when we should use the CLI.



Basically, a pipeline for continuous integration or continuous deployment, cannot log in into the AWS console in the browser, so you need to use a command-line interface there. Also when you are writing the script, the same patterns apply there, or when you would do a repetitive task such as deploying your application, this is advisable to use a CLI at this point.

Cloud services are a great way to host your application and make it available to your users. It is your role, however, to monitor it and to be sure that the environment is in a healthy state so that your users can get your content! In this lesson we will understand how to do the following using CLI commands:

- Deploy code to cloud services
- Check the health of a service
- Get information about a service

Don't be afraid to jump into a terminal and use the AWS CLI! It is often the best way to interact with a cloud service.

Often we are also faced with the decision of using **the AWS console or the CLI**. While in most cases we can accomplish the same with both options, we should prioritize the console when we are learning new services. On the other hand, we must prioritize the CLI when doing repetitive tasks and when writing scripts.

There are some example for better to use the AWS console over the AWS CLI.

Scenario	Which is better?
Doing a repetitive task like adding new customer reviews from an Excel spreadsheet to your database	AWS CLI
Learning a new AWS service like AWS Kendra	AWS console
Preparing a CI/CD for a new project at your company	AWS CLI
Doing an action that you don't do often like creating a multiple AWS account setup for the first time.	AWS console

AWS - Install and Configure CLI

> The AWS Command Line Interface (AWS CLI) is a command-line tool that allows you to interact with AWS services using commands in your terminal/command prompt.

AWS CLI enables you to run commands to provision, configure, list, and delete resources in the AWS cloud. Before you run any of the aws commands, you need to follow three steps:

1. Install AWS CLI
2. Create an IAM user with Administrator permissions
3. Configure the AWS CLI

Step 1. Install AWS CLI v2

Refer to the official AWS instructions to install/update AWS CLI (version 2) based on your underlying OS. You can verify the installation using the following command in your terminal (macOS)/cmd (Windows).

```
# Display the folder that contains the symlink to the aws cli tool
which aws

# See the current version
aws --version
```

See the sample output below. Note that the exact version of AWS CLI and Python may vary in your system.

Mac/Linux/Windows: Verify the successful installation of AWS CLI 2

Step 2. Create an IAM user

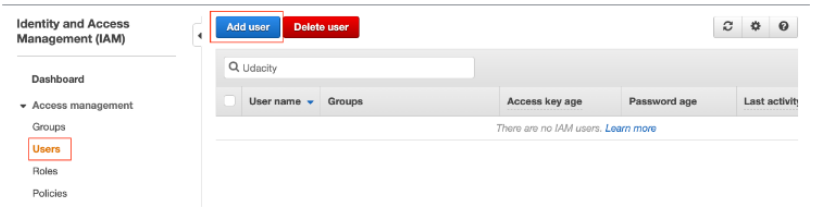
In this step, you will create an IAM user with Administrator permissions who is allowed to perform *any* action in your AWS account, only through CLI. After creating such an IAM user, we will use its **Access key** (long-term credentials)** **to configure the AWS CLI locally.

Let's create an AWS IAM user, and copy its Access key.

> AWS Identity and Access Management (IAM) service allows you to authorize users / applications (such as AWS CLI) to access AWS resources.

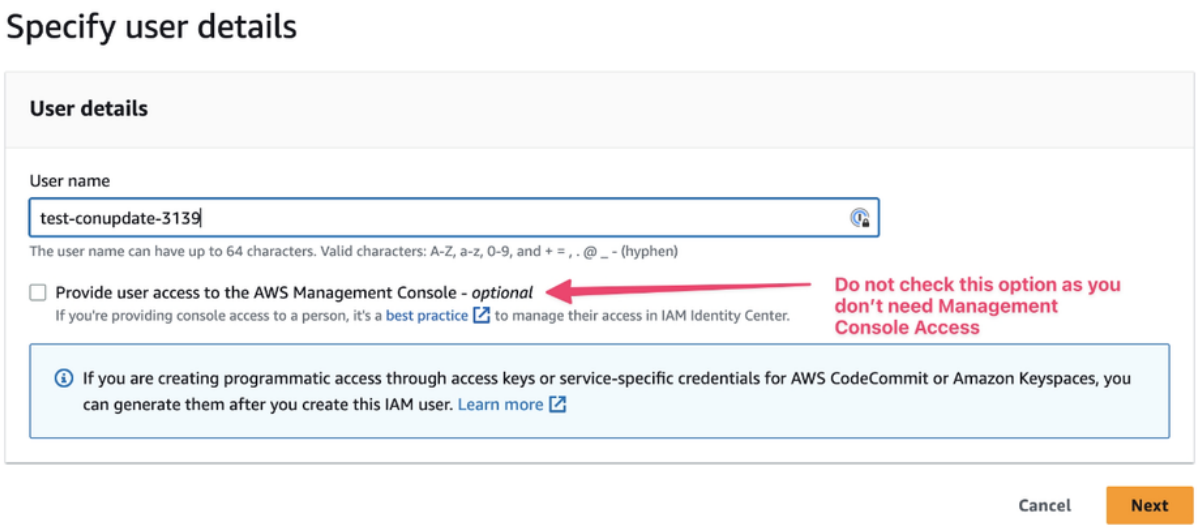
The Access key is a combination of an **Access Key ID** and a **Secret Access Key.** Let's see the steps to create an IAM user, and generate its Access key.

- Navigate to the IAM Dashboard, and create an IAM user.



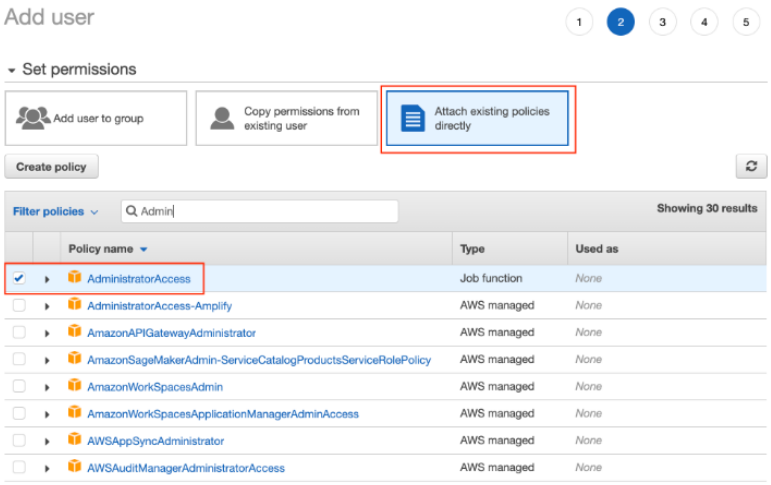
Add a new IAM user

- Set the user name, and click **Next**. **DO NOT** check **Provide user access to the AWS Management Console - *optional***.



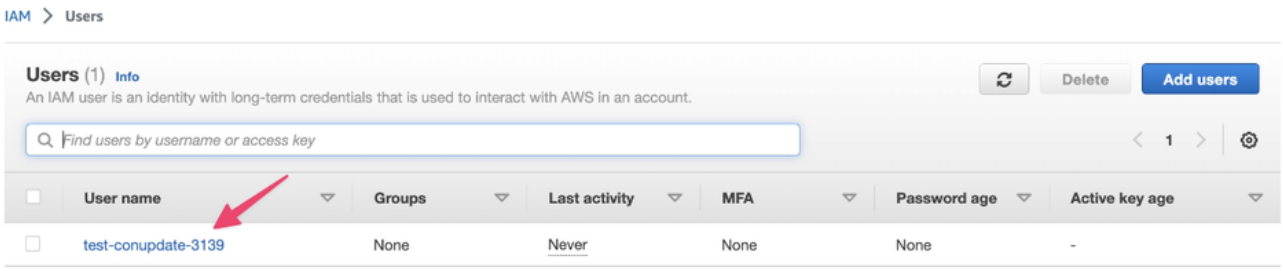
Set User name.

- Set the permissions to the new user by attaching the AWS Managed **AdministratorAccess** policy from the list of existing policies.



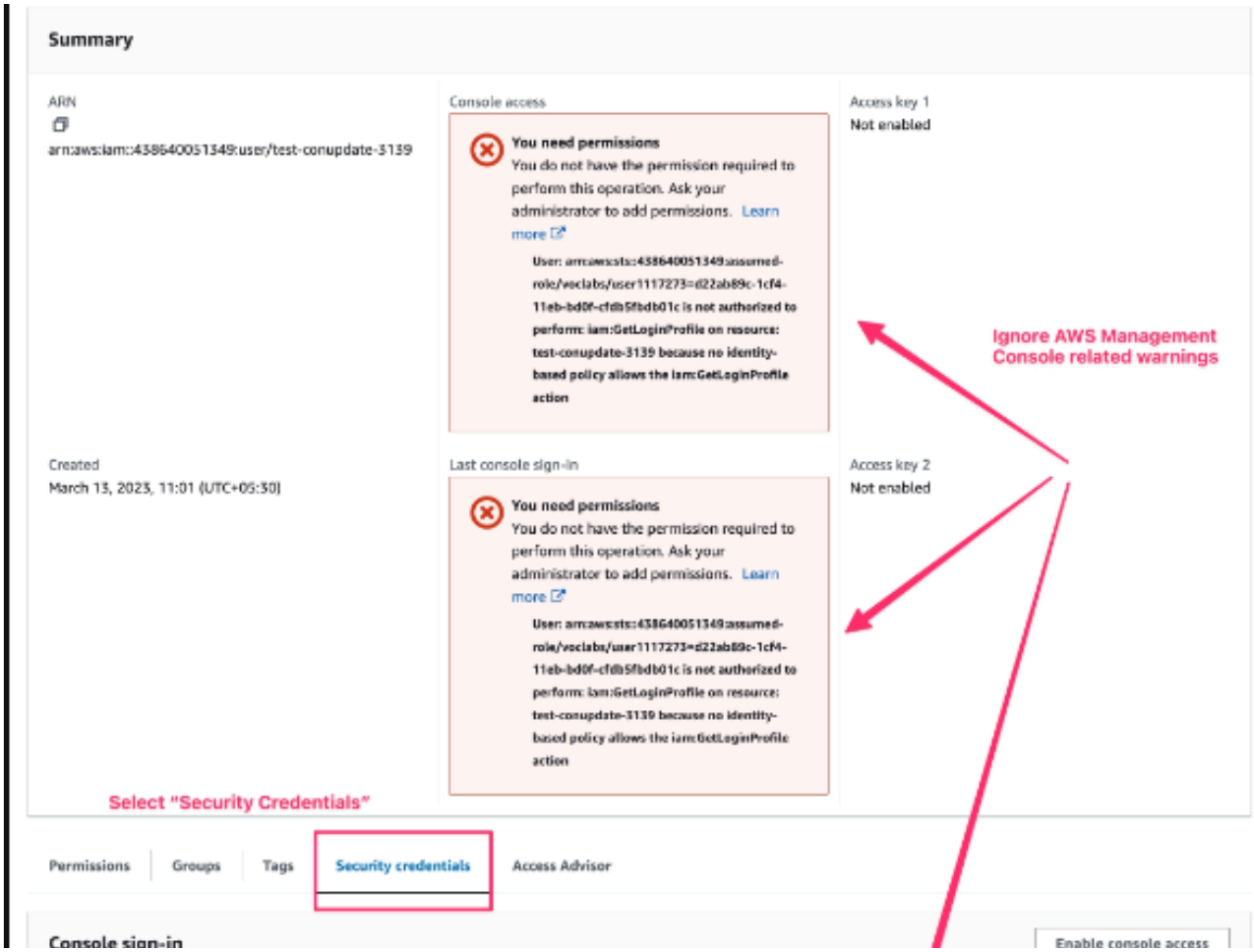
Attach the *AdministratorAccess* policy from the list of pre-created policies

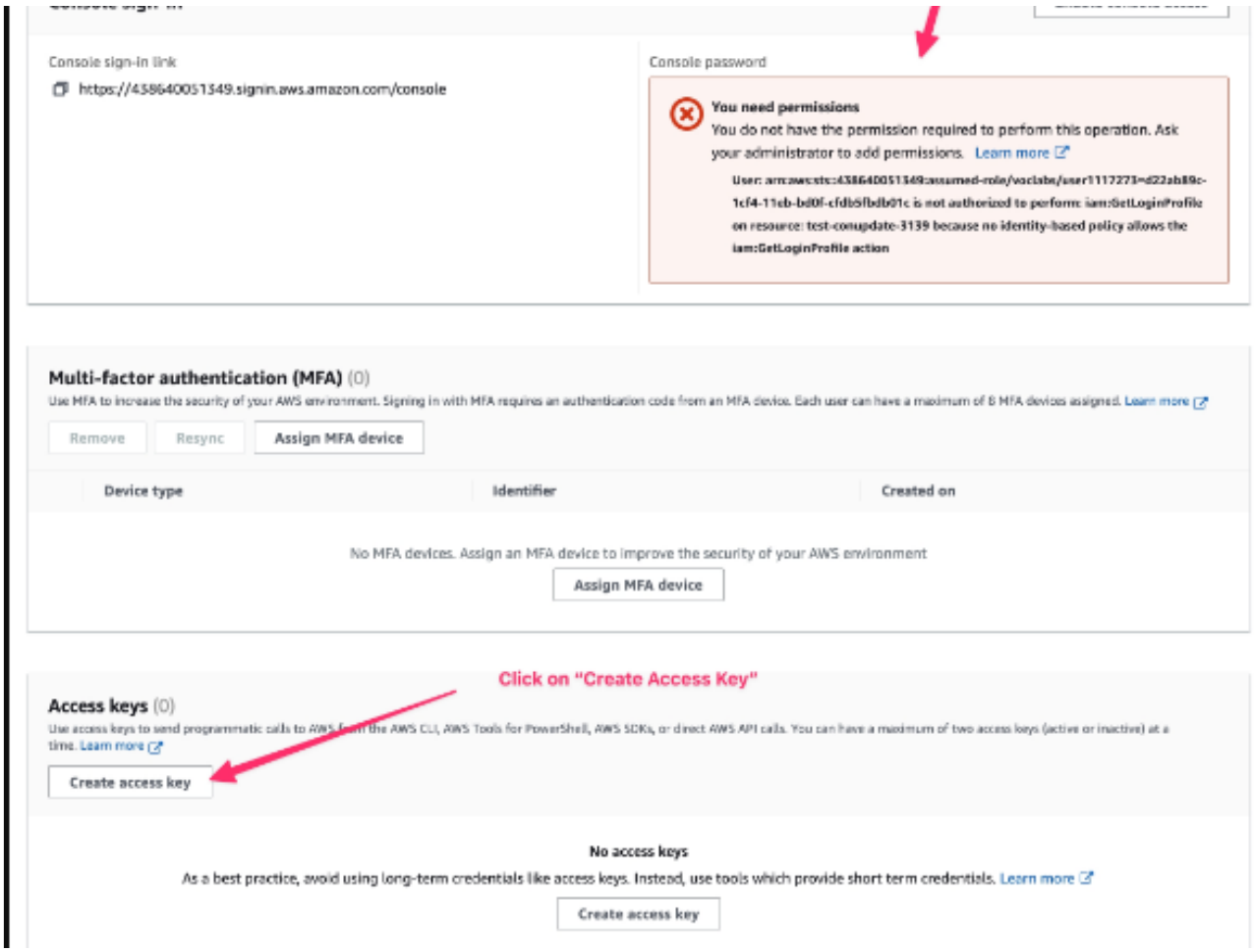
- Provide tags [optional], review the details of the new user, and finally create the new user.
- After a user is created successfully, click on the User name.



Select the created user.

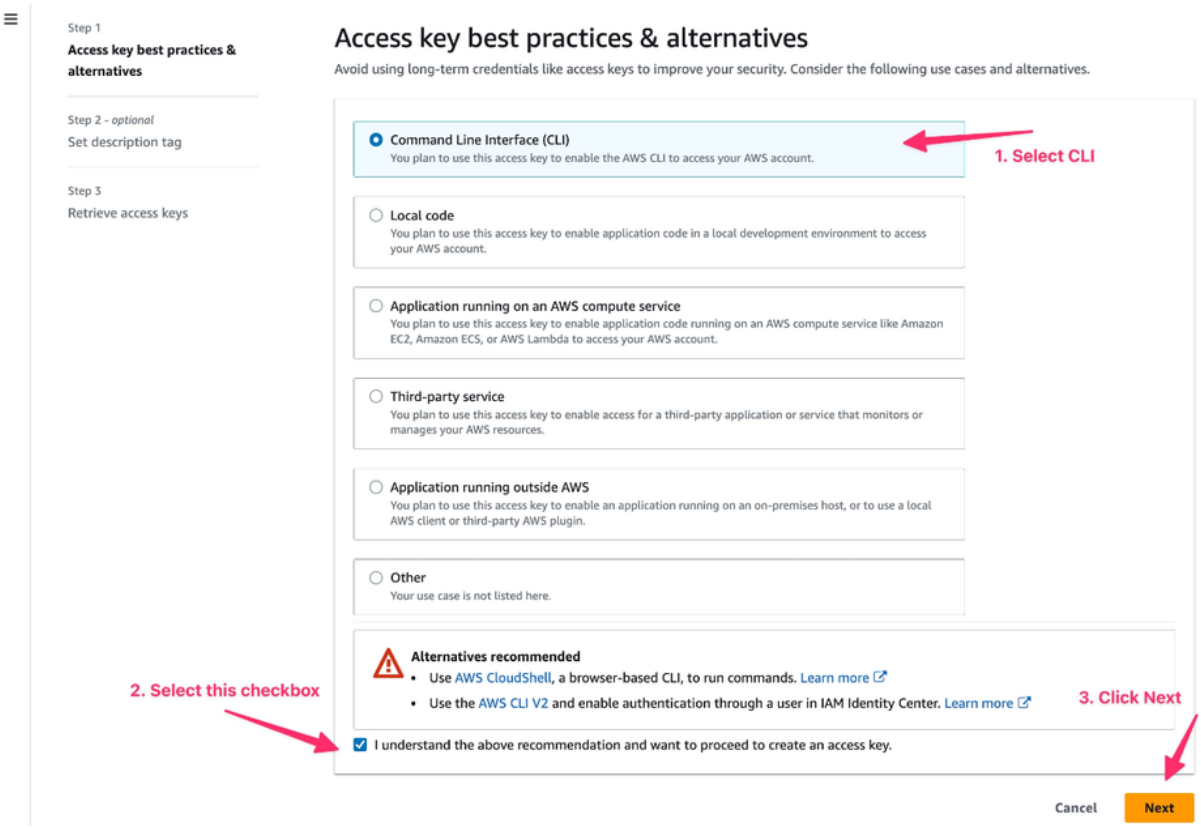
- Ignore **AWS Management Console** related warnings. Since you only need programmatic access, this can be ignored. Go to **Security Credentials** and select **Create access key**.


Summary
ARN
arn:aws:iam::438640051349:user/test-conupdate-3139
Console access
You need permissions
You do not have the permission required to perform this operation. Ask your administrator to add permissions. Learn more
User: arn:aws:sts::438640051349:assumed-role/voclabs/user1117273=d22ab89c-1cf4-11eb-bd0f-cfdb5fbdb01c is not authorized to perform: iam:GetLoginProfile on resource: test-conupdate-3139 because no identity-based policy allows the iam:GetLoginProfile action
Access key 1
Not enabled
Ignore AWS Management Console related warnings
Created
March 13, 2023, 11:01 (UTC+05:30)
Last console sign-in
You need permissions
You do not have the permission required to perform this operation. Ask your administrator to add permissions. Learn more
User: arn:aws:sts::438640051349:assumed-role/voclabs/user1117273=d22ab89c-1cf4-11eb-bd0f-cfdb5fbdb01c is not authorized to perform: iam:GetLoginProfile on resource: test-conupdate-3139 because no identity-based policy allows the iam:GetLoginProfile action
Access key 2
Not enabled
Select "Security Credentials"
Permissions
Groups
Tags
Security credentials
Access Advisor
Console sign-in
Enable console access


Console sign-in link
https://458640051349.signin.aws.amazon.com/console
Console password
You need permissions
You do not have the permission required to perform this operation. Ask your administrator to add permissions. Learn more
User: arn:aws:sts::458640051349:assumed-role/voclabs/user1117273=d22ab89c-1cf4-11eb-b80f-cfdb5fbdb01c is not authorized to perform: iam:GetLoginProfile on resource: test-coupdate-3139 because no identity-based policy allows the iam:GetLoginProfile action
Multi-factor authentication (MFA) (0)
Use MFA to increase the security of your AWS environment. Signing in with MFA requires an authentication code from an MFA device. Each user can have a maximum of 8 MFA devices assigned. Learn more
Remove
Resync
Assign MFA device
Device type
Identifier
Created on
No MFA devices. Assign an MFA device to improve the security of your AWS environment
Assign MFA device
Click on "Create Access Key"
Access keys (0)
Use access keys to send programmatic calls to AWS from the AWS CLI, AWS Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. Learn more
Create access key
No access keys
As a best practice, avoid using long-term credentials like access keys. Instead, use tools which provide short term credentials. Learn more
Create access key

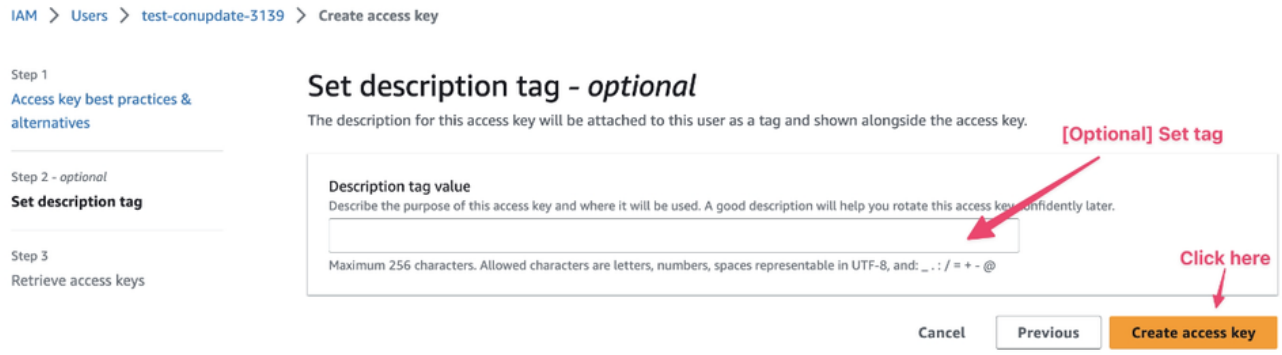
Create Access Key for the user.

- Select **Command Line Interface (CLI)** and click **Next**.



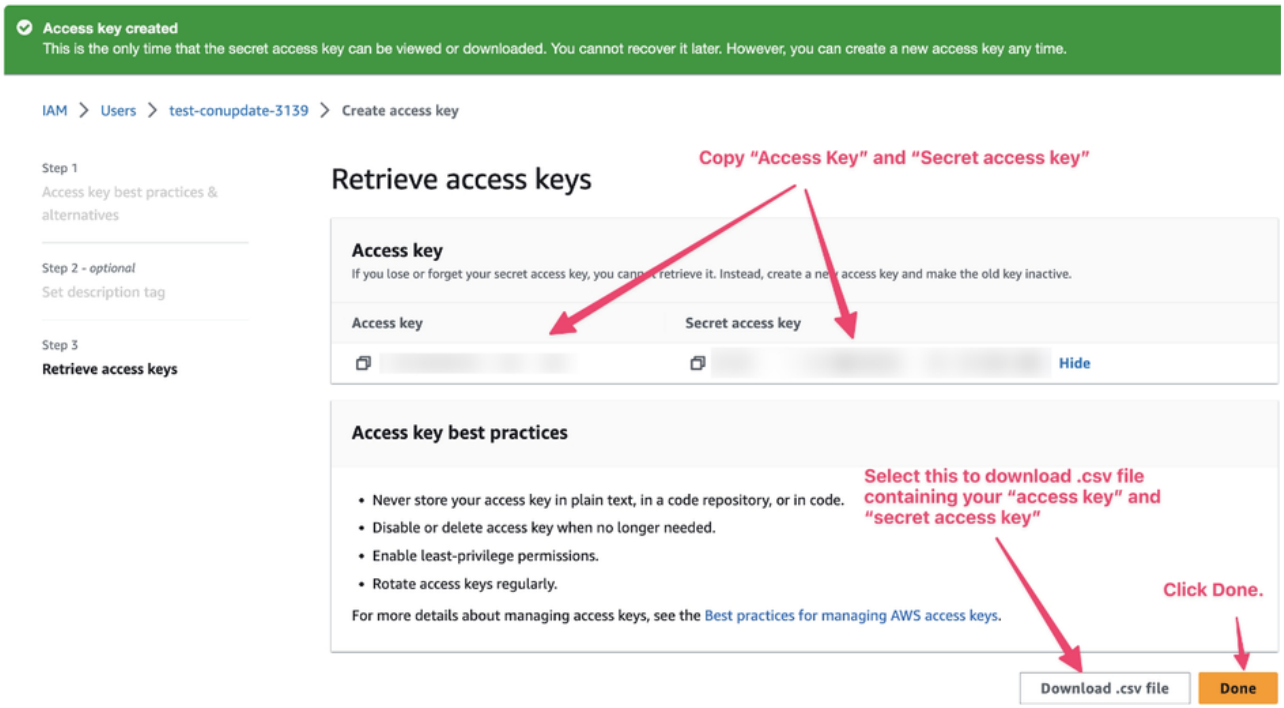
Select *Command Line Interface (CLI)*

- Optional - Set description tag and click **Create access key**.



Optional - Set description tag for the access keys

- Copy the created **Access key**, **Secret access key** and store it for later use. You can also download these as a *.csv* file.



Copy *Access Key* and *Secret Access Key*

Step 3. Configure the AWS CLI

You will need to configure the following four items on your local machine before you can interact with any of the AWS services:

1. **Access key** - It is a combination of an *Access Key ID* and a *Secret Access Key*. Together, they are referred to as *Access key*. You can generate an Access key from the AWS IAM service, and specify the level of permissions (authorization) with the help of *IAM Roles*.
2. **Default AWS Region** - It specifies the AWS Region where you want to send your requests by default.
3. **Default output format** - It specifies how the results are formatted. It can either be a json, yaml, text, or a table.
4. **Profile** - A collection of settings is called a profile. The default profile name is `default`, however, you can create a new profile using the `aws configure --profile new_name` command.

Here are the steps to configure the AWS CLI in your terminal:

- Run the command below to configure the AWS CLI using the *Access Key ID* and a *Secret Access Key* generated in the previous step. If you have closed the web console that showed the access key, you can open the downloaded access key file (.csv) to copy the keys later.

```
aws configure
```

If you already have a profile set locally, you can use `--profile <profile-name>` option with any of the AWS commands above. This will resolve the conflict with the existing profiles set up locally. Next, use the following values in the prompt that would appear:

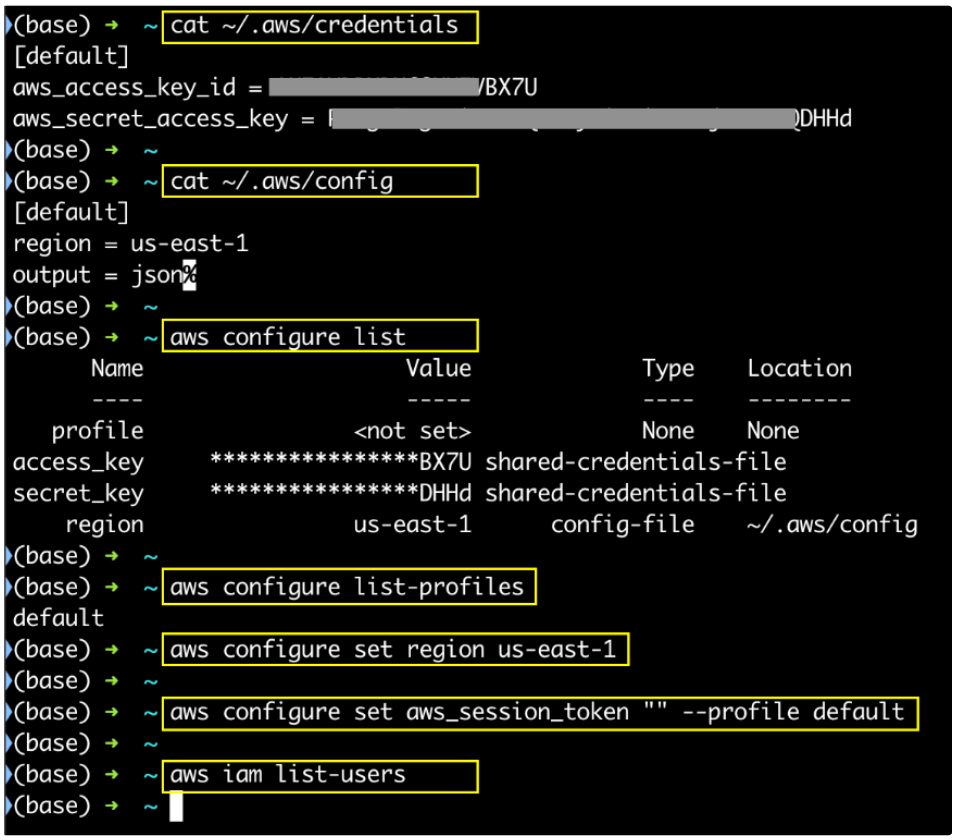
Prompt	Value
AWS Access Key ID	[Copy from the classroom]
AWS Secret Access Key	[Copy from the classroom]
Default region name	us-east-2 (or your choice)

Default output format	json

Important - ```bash # If you are using the Access key of an Admin IAM user, you should reset the `aws_session_token` aws configure set aws_session_token "".

- The commands above will store the access key in a default file `~/.aws/credentials` and store the profile in the `~/.aws/config` file. Upon prompt, paste the copied access key (access key id and secret access key). Enter the default region as `us-east-2` and output format as `json`. You can verify the saved config using:

```
# View the current configuration
aws configure list

# View all existing profile names
aws configure list-profiles

# In case, you want to change the region in a given profile
# aws configure set <parameter> <value>  --profile <profile-name>
aws configure set region us-east-2
```



Mac/Linux: A successful configuration

- Let the system know that your sensitive information is residing in the .aws folder

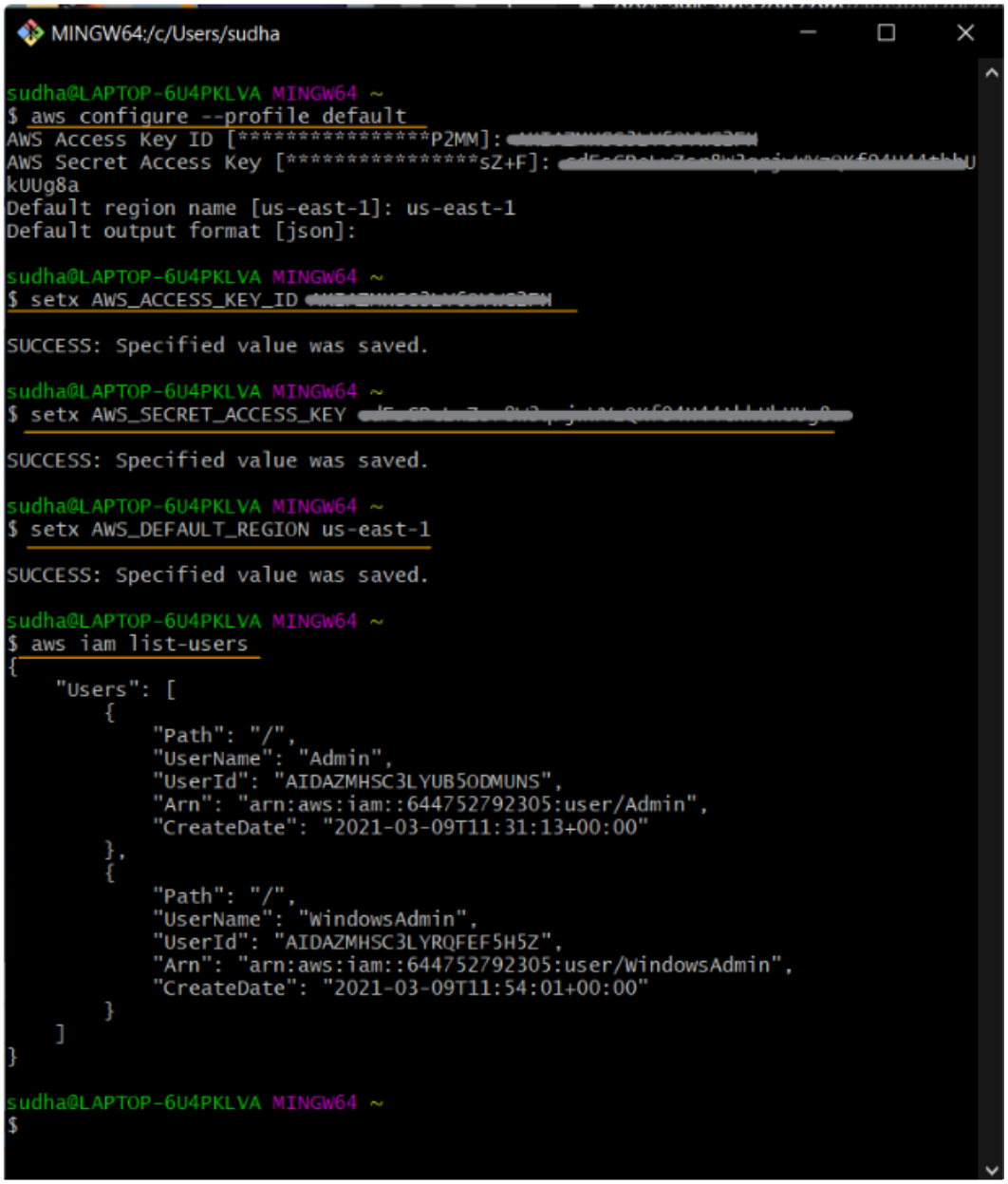
```
export AWS_CONFIG_FILE=~/.aws/config
export AWS_SHARED_CREDENTIALS_FILE=~/.aws/credentials
```

- **Windows users with GitBash only**
  You will have to set the environment variables. Run the following commands in your GitBash terminal:

```
setx AWS_ACCESS_KEY_ID AKIAIOSFODNN7EXAMPLE
setx AWS_SECRET_ACCESS_KEY wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

```
setx AWS_DEFAULT_REGION us-west-2
```

Replace the access key ID and secret, as applicable to you. Windows users using WSL do not need this step, they will follow all steps as if they are Linux users.



Windows: Successful configuration using the GitBash terminal

Step 4. Run your first AWS CLI command

- Check the successful configuration of the AWS CLI, by running either of the following AWS command:

```
# If you've just one profile set locally
aws iam list-users

# If you've multiple profiles set locally
aws iam list-users --profile <profile-name>
```

The output will display the details of the recently created user:

```
{
"Users": [
    {
        "Path": "/",
        "UserName": "Admin",
        "UserId": "AIDAZMXYZ3LY2BNC5ZM5E",
        "Arn": "arn:aws:iam::388752792305:user/Admin",
        "CreateDate": "2021-01-28T13:44:15+00:00"
    }
]
}
```

Troubleshoot

If you are facing issues while following the commands above, refer to the detailed instructions here -

1. Configuration basics
2. Configuration and credential file settings
3. Environment variables to configure the AWS CLI

Updating the specific variable in the configuration

In the future, you can set a single value, by using the command, such as:

```
# Syntax
# aws configure set <varname> <value> [--profile profile-name]
 aws configure set default.region us-east-2
```

It will update only the region variable in the existing default profile.

How Experts Approach Interacting with Cloud Services

One thing that can feel overwhelming at first is the insane amount of commands that you're going to see. But don't worry; if you read the documentation, spend time reading about each command, and try them, this will get a lot easier. Practicing is key here.

Don't forget to practice your commands, read your documentation, and don't worry too much.

CLI expertise comes with lots of practice! There is a lot you can do with the terminal and commands. The opportunities for practice are endless! Here are some ideas to get you started:

1. Read the documentation
2. Do everything with the CLI
3. Find things to automate

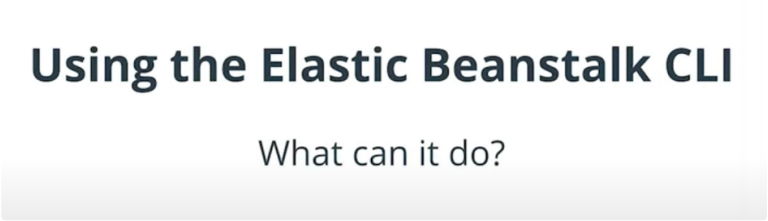
Over time you will start to feel more confident, and using CLI commands will start to feel like second nature!

Using the EB CLI

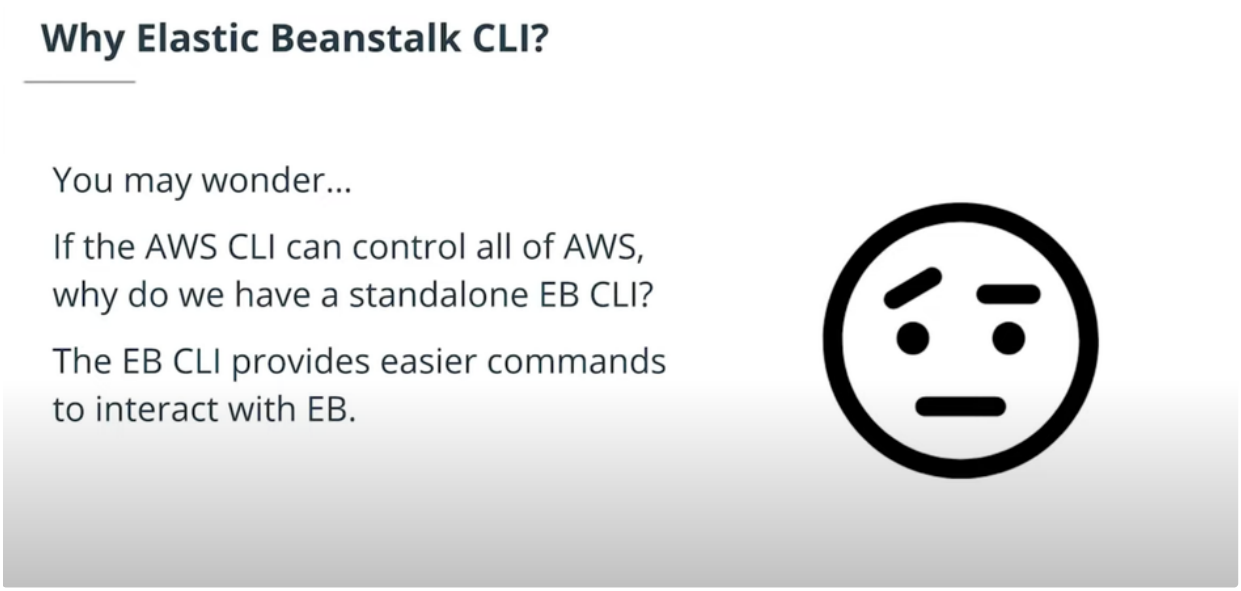
Using the Beanstalk CLI

While, AWS CLI can be used to perform almost any possible actions on the AWS platform, the commands to create and manage EB so are long.

Therefore, AWS created a dedicated Elastic Beanstalk (EB) CLI. The EB CLI is simple to use and provides a set of easy commands that let you control your application environment in a convenient way!

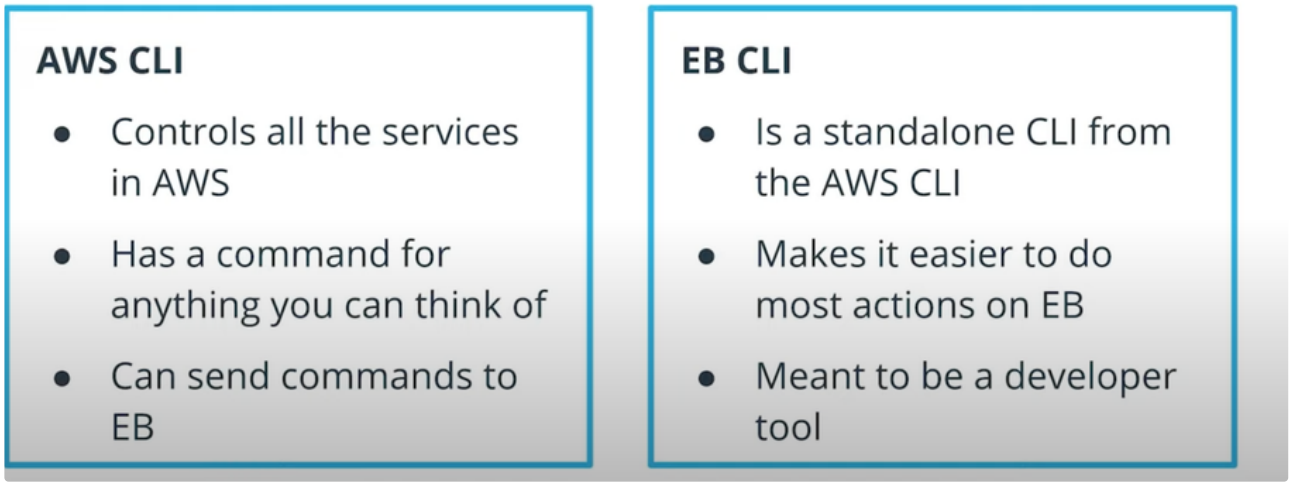


Let's see how we can use the Elastic Beanstalk CLI. But more importantly, what can it do? You might be wondering why there is an Elastic Beanstalk CLI, and that would be normal. We have an AWS CLI that can control all the services on AWS. Why do we need a standalone Elastic Beanstalk CLI? The reason is simple.



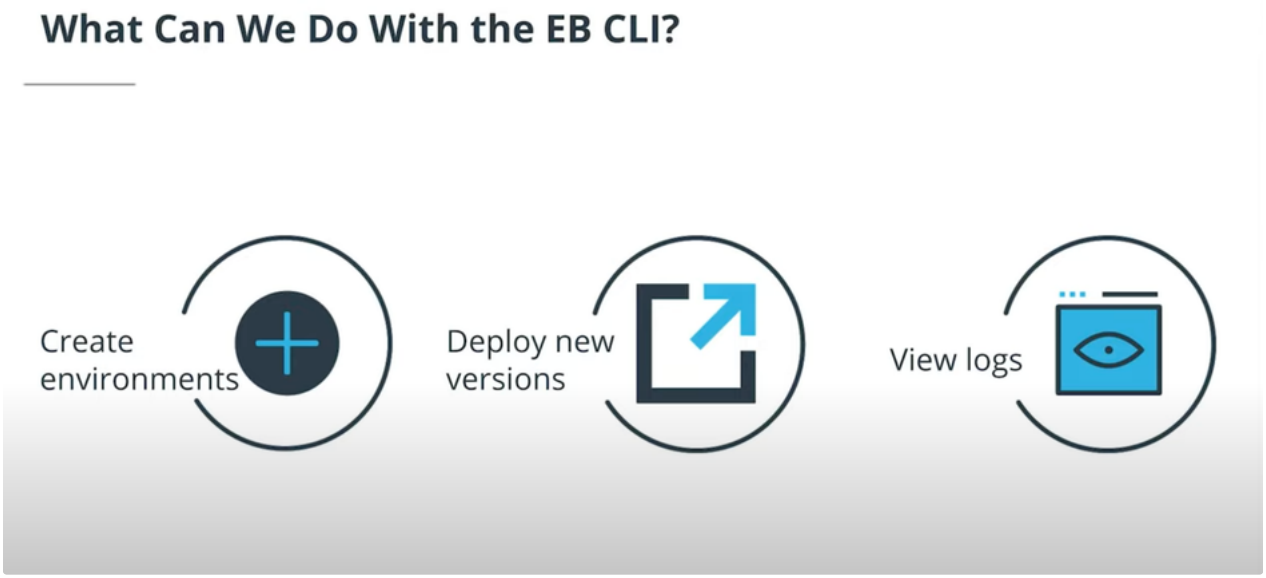
It's that the Elastic Beanstalk CLI is easier to use and the commands are much easier to interact with than using the full AWS CLI.

Here is the Difference



Let's look a little bit at the difference. The AWS CLI is really able to control all the services in AWS. It has a command for anything that you can think of, including commands that can be sent to Elastic Beanstalk. On the other hand, the Elastic Beanstalk CLI is a standalone. It's really created to make things easier when it comes to regard to Elastic Beanstalk. It was not that the goal was meant to make it a developer tool. Whereas the AWS CLI is really an operational tool for controlling all of AWS, the Elastic Beanstalk CLI caters to you, the developer.

What Can We Do With the EB CLI?



We saw that the Elastic Beanstalk CLI is a tool for developers. What does that mean? As a developer, you do actions pretty commonly that you must repeat and using a long command in your command line interface would feel unnatural. This is why Elastic Beanstalk has a fast command for a multitude of things, including creating environments, deploying new versions of your code, and checking the logs of what is happening on the environment.

Install EB CLI

We recommend installing the EB CLI using setup scripts, using the command like:

```
# Linux/MacOS
cd
```

```
python -m pip install virtualenv
git clone https://github.com/aws/aws-elastic-beanstalk-cli-setup.git
python ./aws-elastic-beanstalk-cli-setup/scripts/ebcli_installer.py
echo 'export PATH="/root/.ebcli-virtual-env/executables:$PATH"' >> ~/.bash_profile && source ~/.bash_profile
eb --version

# Windows users follow the instructions here:
# https://github.com/aws/aws-elastic-beanstalk-cli-setup
```

For other installation options, refer to Installing the EB CLI.

Deploy a sample NodeJS application

Here is the commonly used List of commands that are available with the EB CLI. We will use some of these commands in this course:

- Let's create a directory. The next few command will deploy a sample NodeJS application to EB.

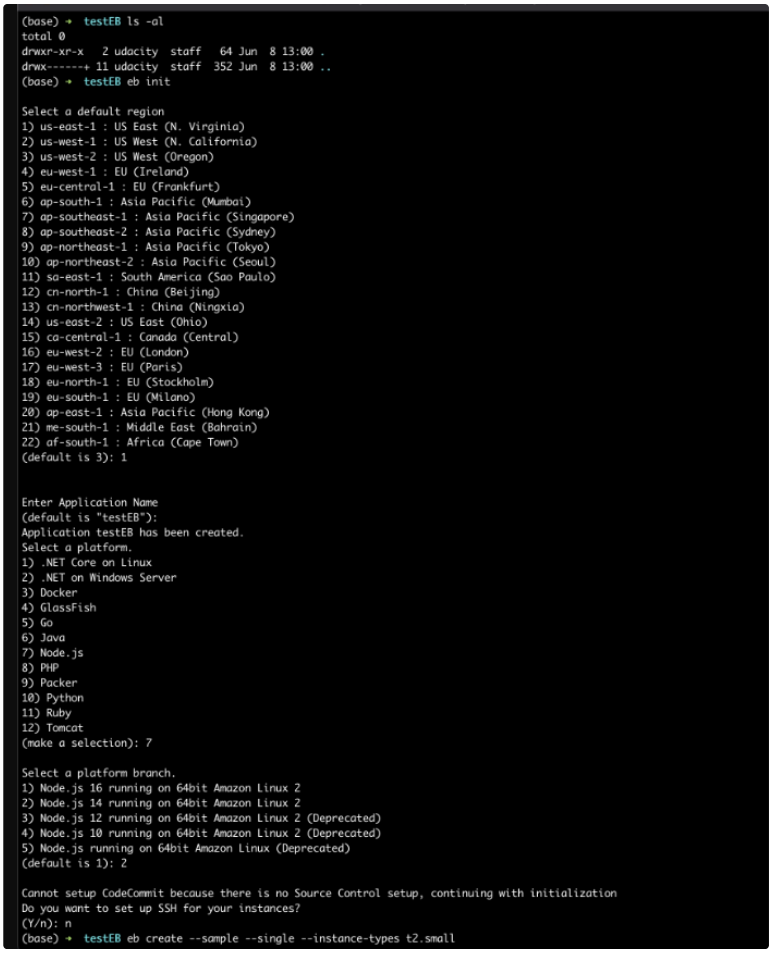
```
mkdir testEB
cd testEB
```

- Initialise an environment. Run this command in the root directory of the application you want to deploy. The eb init command will create ".elasticbeanstalk/config.yml" file in the current directory.

```
# Use the node.js 12 or 14 and the default region as applicable to you
eb init
```

> **What is an environment?**
> An environment is the collection of AWS resources and permissions to allow your web application to run smoothly. The Elastic Beanstalk service manages the environment for us.

The command above will prompt you for Application Name, runtime platform and its version (Node.js 14), and region. Choose "No" when it asks *"Do you want to set up SSH for your instances?"*.



Running the `eb init` command

- The eb create will bundle you application, if present in the current directory, and deploy to the EB. Otherwise, a sample application will be deployed. We can specify the `--sample` to be sure.

```
eb create --sample --single --instance-types t2.small
```

Provide you input for the prompts that appear, such as:

- Enter Environment Name: Default
- Enter DNS CNAME prefix: Default
- Would you like to enable Spot Fleet requests for this environment? (y/N): N
- Do you want to download the sample application into the current directory? (Y/n): Y

If you have chosen Yes to the question above, you will get these files in your local:

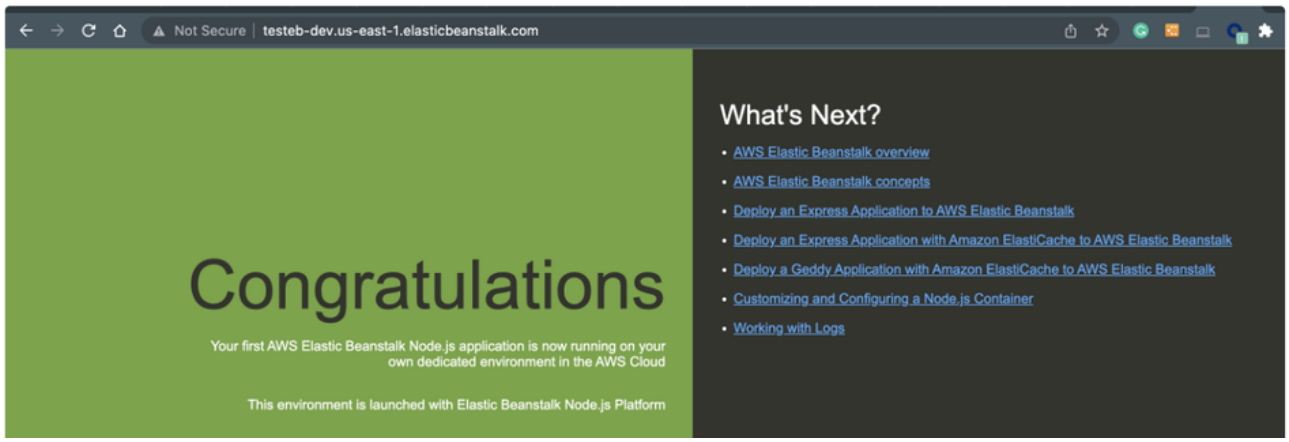
```
.
├── app.js
├── cron.yaml
├── index.html
└── package.json
```

The command above will take upto 15 mins to create the following resources as part of the environment:

1. An EC2 instance (size: *t2.small*) to host your application, and without a load balancer because we have use the *--single* option.
2. A security group (firewall rules) for the EC2 instance
3. An S3 bucket to store the application artifacts
4. A CloudWatch alarm for logging and monitoring
5. A domain name

```
(base) ➜  testEB eb create --sample --single --instance-types t2.small
Enter Environment Name
(default is testEB-dev):
Enter DNS CNAME prefix
(default is testEB-dev):

Would you like to enable Spot Fleet requests for this environment? (y/N): N
Do you want to download the sample application into the current directory? (Y/n): Y
INFO: Downloading sample application to the current directory.
INFO: Download complete.
Environment details for: testEB-dev
  Application name: testEB
  Region: us-east-1
  Deployed Version: Sample Application
  Environment ID: e-rjvaw9efim
  Platform: arn:aws:elasticbeanstalk:us-east-1::platform/Node.js 14 running on 64bit Amazon Linux 2/5.5.3
  Tier: WebServer-Standard-1.0
  CNAME: testEB-dev.us-east-1.elasticbeanstalk.com
  Updated: 2022-06-08 07:47:10.743000+00:00
Printing Status:
2022-06-08 07:47:09    INFO    createEnvironment is starting.
2022-06-08 07:47:10    INFO    Using elasticbeanstalk-us-east-1-751397240857 as Amazon S3 storage bucket for environment data.
2022-06-08 07:47:31    INFO    Created security group named: awseb-e-rjvaw9efim-stack-AWSEBSecurityGroup-R71W9PMI4IT5
2022-06-08 07:47:47    INFO    Created EIP: 52.21.157.108
2022-06-08 07:48:52    INFO    Waiting for EC2 instances to launch. This may take a few minutes.
2022-06-08 07:49:36    INFO    Instance deployment: You didn't specify a Node.js version in the 'package.json' file in your source bundle.
 The deployment didn't install a specific Node.js version.
2022-06-08 07:49:41    INFO    Instance deployment completed successfully.
2022-06-08 07:50:14    INFO    Application available at testEB-dev.us-east-1.elasticbeanstalk.com.
2022-06-08 07:50:14    INFO    Successfully launched environment: testEB-dev
```

The `eb create` will generate a domain name, for example, see the testEB-dev.us-east-1.elasticbeanstalk.com in the snapshot above.



Accessing the **sample** application at testEB-dev.us-east-1.elasticbeanstalk.com in the browser

- For troubleshooting, look into the logs from your terminal:

```
eb logs
```

- Editing and redeploying the application. You can edit you local application, and commit your changes. It is important to commit the code changes before you eb deploy it to the beanstalk environment.

```
git add -A
git commit -m "change log"
eb deploy
```

- If you have multiple environments running, you can associate the EB CLI with a particular one using:

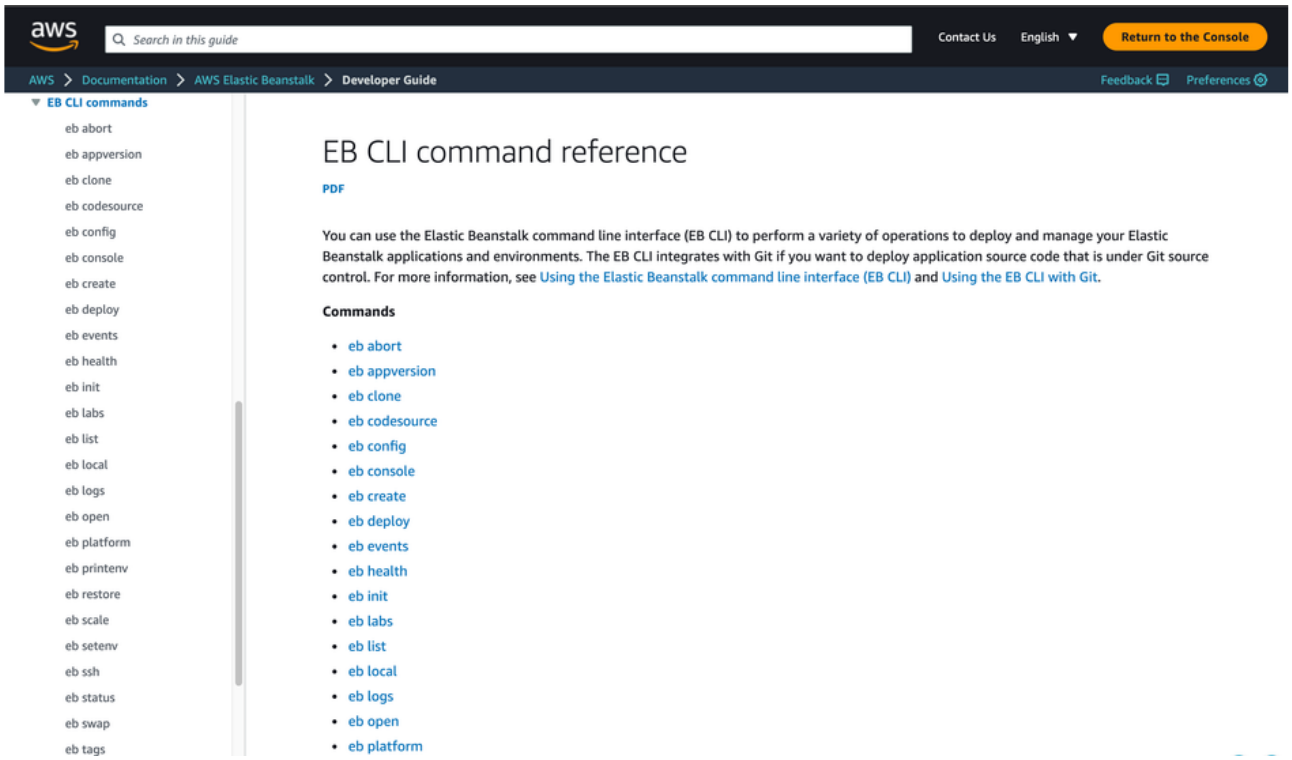
```
eb list
eb use [env-name]
```

- **Clean up** - Don't forget to delete your environment(s) if they are not in use:

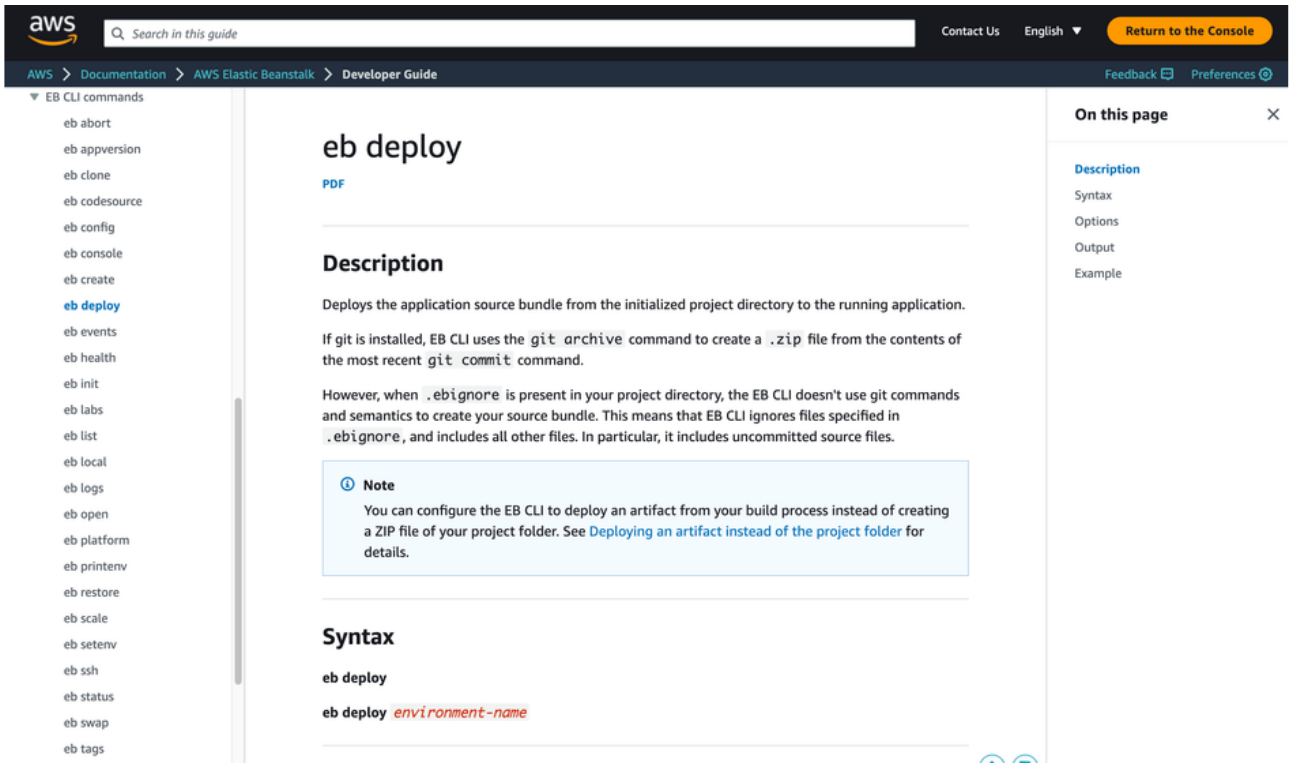
```
eb list
eb terminate [env-name]
```

The EB CLI command reference page provides a series of commands you can use. The video below explains how to read EB documentation.

Reading EB Documentation



Reading documentation is often something you will do throughout your career as a full-stack developer and this goes without saying that you will do the same when deploying applications. Let's look a little bit at how we can read documentation for command-line interface programs, such as the EB CLI one. When you come generally to one of those pages, what you will see is a list of commands and a general description about what the CLI commands are meant for. Let's dive into one of the commands that we have used already and see a little bit how this will play. Let's click on elastic deploy and parse a little bit this documentation page.



The first thing you are greeted with is a description of what this command will do. This description is normally a more human-readable format, trying to explain the intent of such a command. If you scroll down a little bit, you can also get information about the syntax.

Syntax

eb deploy

eb deploy *environment-name*

Options

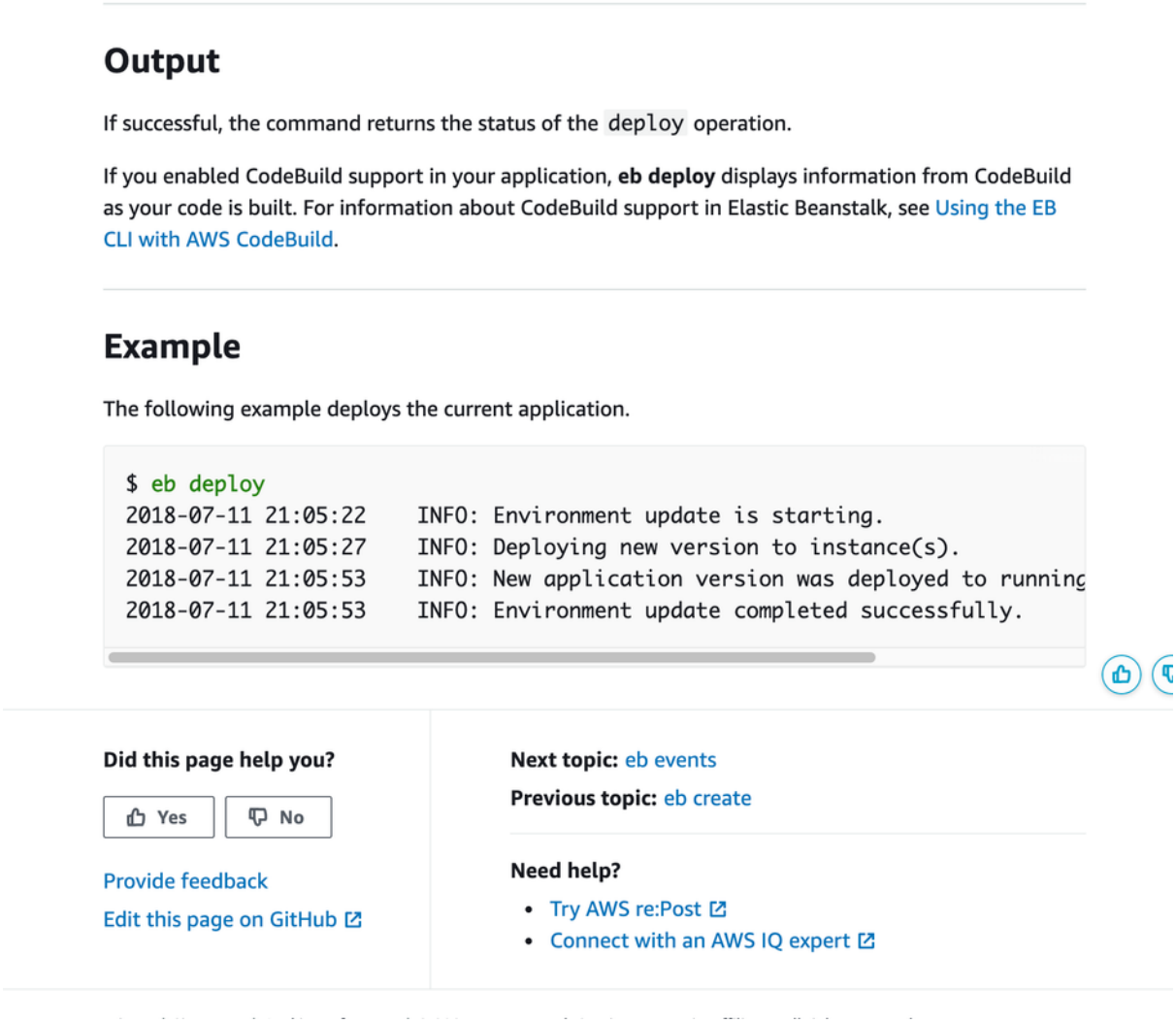
Name	Description
`-l` *version_label* or `--label` *version_label*	Specify a label to use for the version that the EB CLI creates. If the label has already been used, the EB CLI redeploys the previous version with that label. Type: String
`--env-group-suffix` *groupname*	Group name to append to the environment name. Only for use with Compose Environments.

`-m "`*version_description*`"`
or
`--message "`*version_description*`"`
	The description for the application version, enclosed in double quotation marks.
	Type: String

`--modules` *component-a component-b*	List of components to update. Only for use with Compose Environments.

`-p`
or
`--process`
	Preprocess and validate the environment manifest and configuration files in the source bundle. Validating configuration files can identify issues prior to deploying the application version to an

This one is pretty simple. If you are in a project where Elastic Beanstalk is already enabled, you can just call EB deploy or you could be specifying also the environment name. Under syntax, you will see the options table, which normally gives you additional options that you could add to this command and a brief explanation with a description on the right, of what they do.



Lastly, you will be presented with an output and an example so that this way you can understand how the exact command behaves and how you will use it in your own project.

This is of course, a varying type of documents. Not all the CLI's will have the exact same format. It is, however, good to learn how to read those and to practice as you will be using those a lot throughout your career.

Documentation Is Your Best Friend!

While the EB CLI is easy to use, the best way to learn it is to dive into the documentation and try using it to manipulate your environment. CLI documentation can vary in format but it will always communicate clearly what the commands can do. Below there is a cheat sheet to get you started, but take some time to explore the documentation in the further readings.

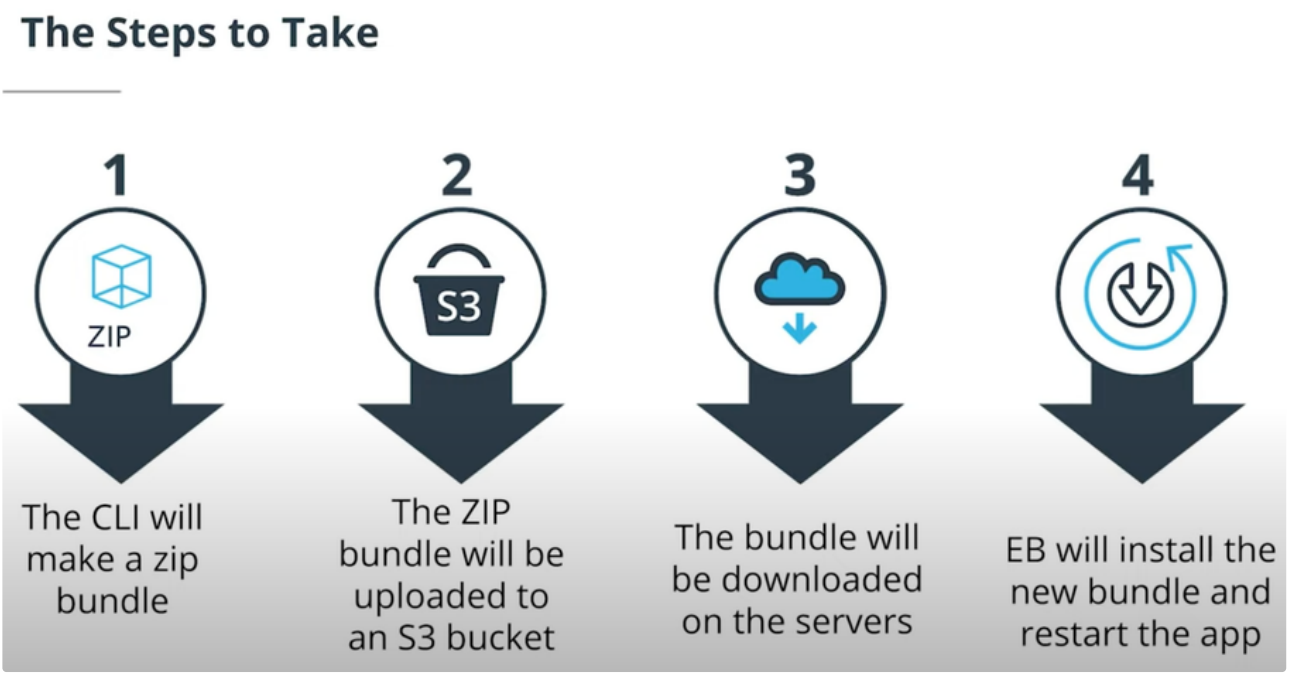
EB CLI Commands Cheat Sheet

- `eb create` allows you to create a new EB environment
- `eb deploy` will deploy your application to Elastic Beanstalk
- `eb use` will link a local repository to an existing EB project
- `eb health` will give you information about the health of your application
- `eb open` will open the EB console in your favorite browser

Deploying Code to EB

How can we deploy code to EB with the EB CLI?

One of the most common actions that you will do with the Elastic Beanstalk CLI is deploying applications. Let's look a little bit under the hood and understand what is happening when you're deploying an application with that CLI.



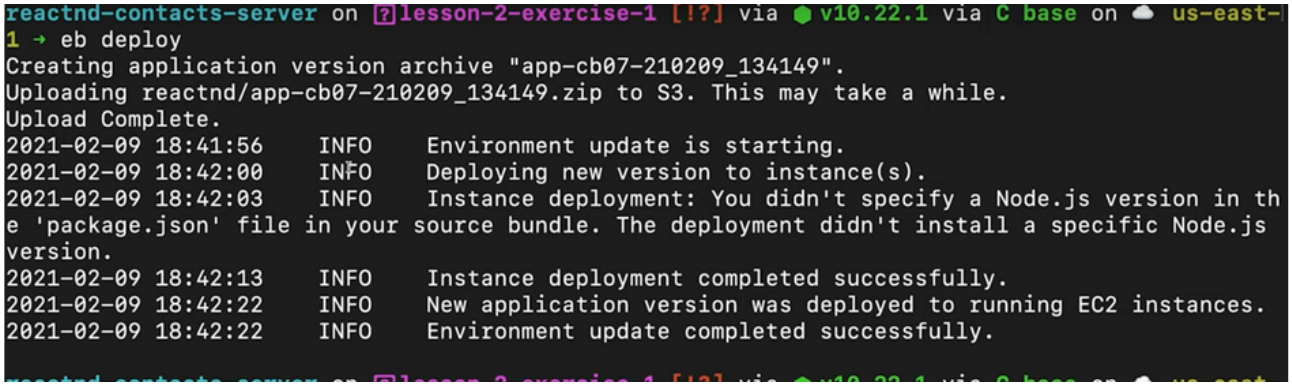
There are multiple steps to take when it comes to deploying applications. Using the command EB deploy is going to do all those steps totally for you. The first one will be to make a zip of the application code. The second step will be to upload the zip bundle into an S3 bucket where all the application versions are kept. Then just after the bundle will be downloaded onto the servers and everything will come in late into being installed on the servers and restarting the application. With a single command, this is what the Elastic Beanstalk does to deploy your application and update and update it.

Summary:

Using the Elastic Beanstalk to deploy an application makes things really easy for us. Under the hood a lot is happening:

1. The application is packaged into a zip file
2. The zip file is uploaded to an S3 bucket
3. The bundled zip is then downloaded onto EB servers
4. The servers are then updated with the new version of the code

Demo: Deploying Code using EB CLI



Let's see a little bit more in depth what is happening when you deploy an application using Elastic Beanstalk CLI. The first thing I will do, I will write eb, standing for Elastic Beanstalk. Then I will tell it to use the deploy command. It's a really simple command as you can see. This will take a while.

Now we will go to explain what is happening. The first thing I'd like to point out, is that this is creating the zip archive, which was the first step that we saw. The second step is uploading the zip archive of your code up onto history. After this is done, step 3 and 4, are happening. Deploying the new version to the instances is happening. Then the servers are actually restarting with the new version of the code. It took under a minute. It would be, of course, a little bit longer if your code was bigger, but in general, this is a fast action to do. What we've just done, would take any update to your code and make it available to your users, by sending this code on to Elastic Beanstalk, and updating your application.

Deploying is faster with the Elastic Beanstalk CLI!

As we have seen in this demonstration, deploying code to the Elastic Beanstalk requires only one command:

```
eb deploy
```

This command will package your code, upload it to S3 and proceed to update your environment with this new version of your code.

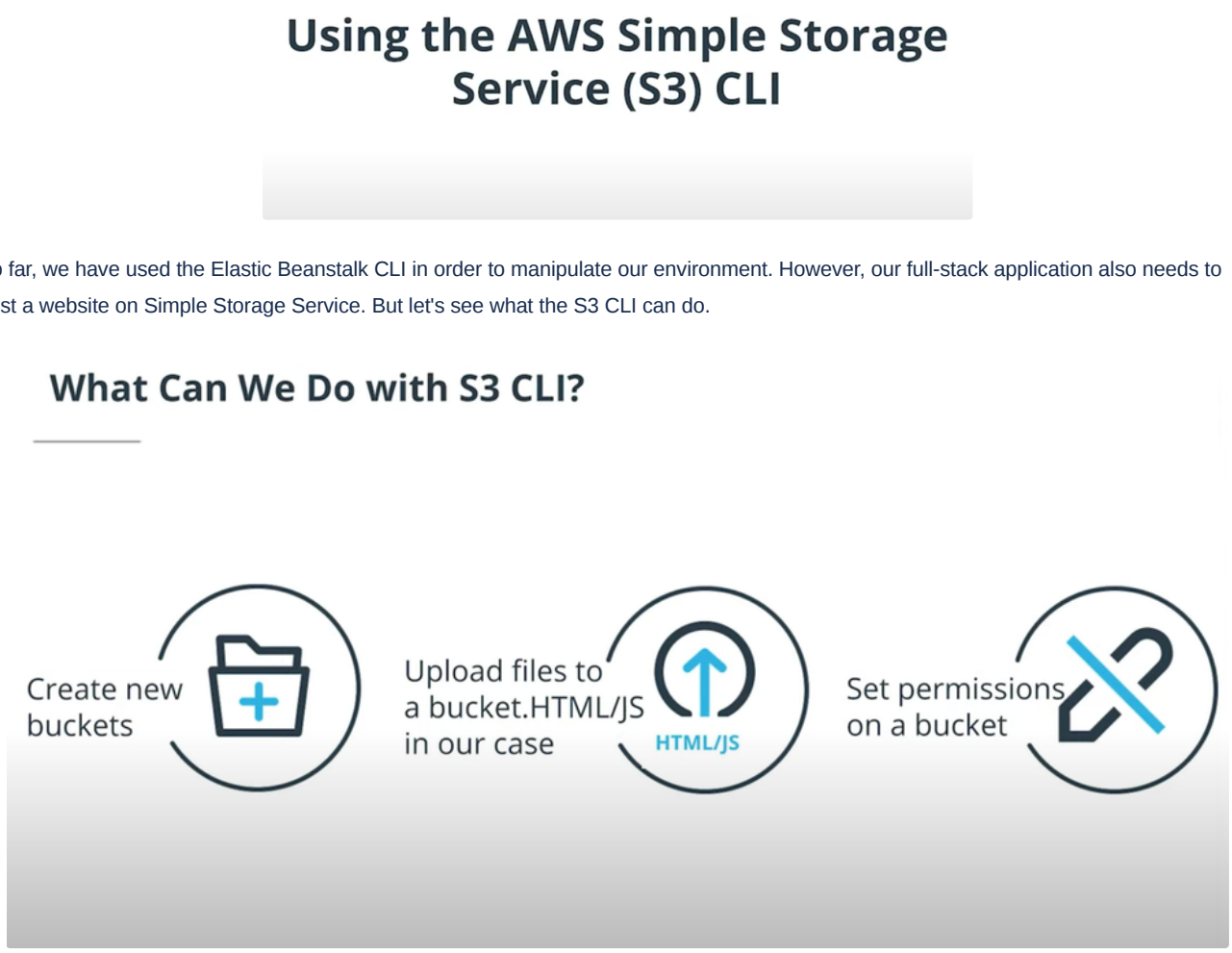
Further Reading

EB deploy documentation: This is the complete documentation going over what EB deploy does. The example given is for deploying a docker container, but the process is similar for a node application.

Blue/Green Deployments are a nice, advanced way to deploy an application to multiple EB servers.

S3 using the AWS CLI

Using the AWS S3 CLI



So far, we have used the Elastic Beanstalk CLI in order to manipulate our environment. However, our full-stack application also needs to host a website on Simple Storage Service. But let's see what the S3 CLI can do.

One of the first things that you might want to do is create a new bucket with it. This is useful because creating buckets can sometimes be repetitive, so you can do it with a CLI to save a little bit of time. One of the most common actions that we will do is also update the content and upload files to our bucket. In our situation, we will be using the CLI to update our HTML and JavaScript so that our website is updated. Lastly, we would want to set permissions on a bucket so that we decide if an object is public or if it is private.

The AWS S3 CLI is a set of subcommands of the AWS CLI that lets you perform actions on S3 buckets. While we can do a lot of actions, here are the ones that we will do the most often as developers:

1. **Create new buckets**: Creating a new bucket with the S3 CLI is quite fast!
2. **Upload files to a bucket:** The S3 CLI lets us copy local files to a bucket.
3. **Set permissions:** We can set access policies on a bucket via the CLI.

Since we will use the S3 CLI mostly to update a static website, we will be using the `aws s3 cp` command the most often!

`cp` is a Linux command that means **copy**. It is used to copy files from one location to another.

When it comes to hosting a static website and updating the content on it we face some challenges. Some browsers will effectively cache files (save them locally to load them faster). This means that, in some situations, to update the content of our website we will need to specify with S3 some cache HTTP headers. Cache-Control is beyond the scope of this course, but if you are interested in learning more, you can consult the "Further Reading" section.

New Terms

- **Cache-Control:** HTTP headers telling the browser how long it needs to cache certain files. To ensure browsers take the new content of our HTML files, we will sometimes need to force the browser to revalidate the content of the files in S3.

Further Reading

- Configuring the AWS CLI provides a quick way to set the AWS CLI if you have trouble linking it to your provided account.
- How cache control works: This is a detailed post explaining how cache-control works.
- Reference to the CP documentation: The full set of options available on the S3 cp command.
- Understanding access to S3 buckets: This is a more in-depth reading on managing S3 access through ACL and other more advanced techniques.

Edge Cases

`eb use` does says ERROR: NotFoundError - Environment "env-name" not Found.

If you are having issues having the EB CLI connect to your environment try running `eb init`. This will ask you a series of questions in order to connect the Elastic Beanstalk to the right application and AWS region. Read the documentation about the command if you are unsure what to answer

Connecting your AWS account to Elastic Beanstalk CLI

If you are having issues connecting your account to the Elastic Beanstalk CLI, try following this tutorial provided on the EB documentation site.

If you need assistance in generating AWS keys, take a look at this tutorial to generate a user with programmatic access.

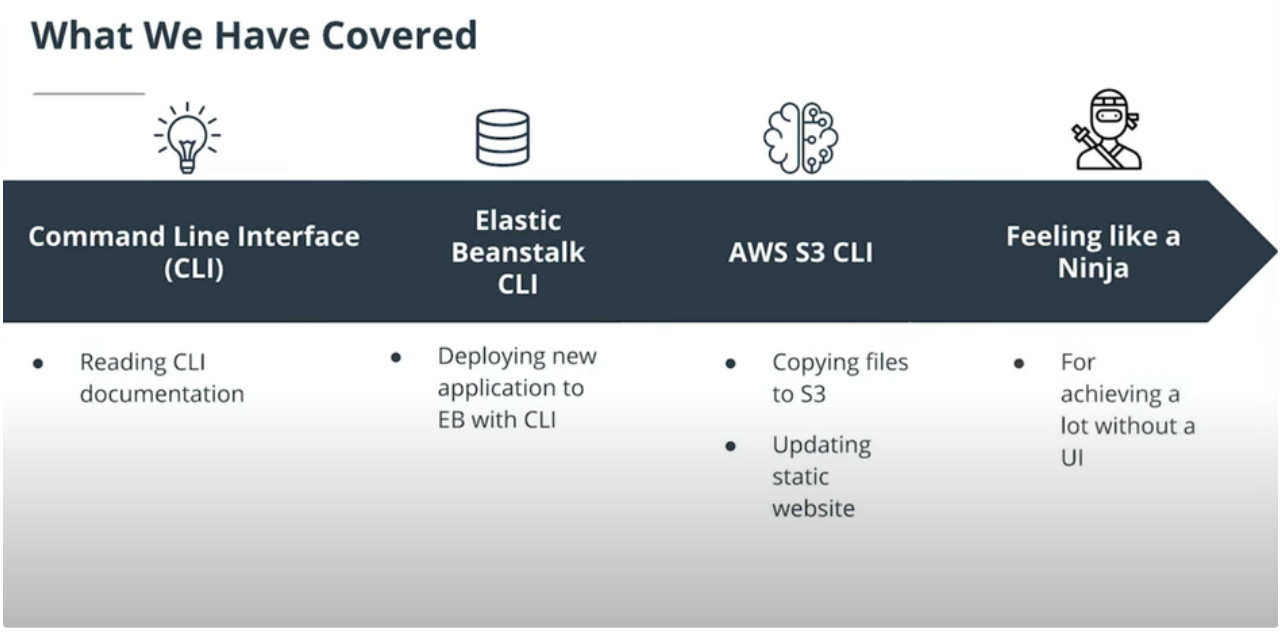
An S3 hosted website does not display updated content after deploying new application code

In some cases, it is possible that your browser is trying to cache the content of a webpage in order to provide you with a better user experience. This can cause some issues with displaying the updated version of a website. You can try running the following command in order to force websites to uncache and revalidate the content on a website. Make sure to replace BUCKET_NAME with the name of your bucket.

```
aws s3 cp --acl public-read --cache-control="max-age=0, no-cache, no-store,
must-revalidate" ./build/index.html s3://BUCKET_NAME/
```

Lesson Recap

What We Have Learned



Let's recap a little bit on everything that we've done. We have introduced the concept of a CLI, and we have understood how to read the documentation and how to find our way through it; we have also spent a little bit of time looking at the Elastic Beanstalk CLI, and we've deployed an application to it. After this, we learned the AWS S3 CLI, copied files to it, and we updated our static website. Lastly, again, we felt like a ninja because we have just built a bunch of CLI skills.

This lesson was a deep-dive into different command-line interfaces and how to use them to interact with AWS services! You have now gained some new skills that will empower you to deploy applications!

Throughout the deployment process, learning the specific commands you need to deploy your application to production is an integral part. With this knowledge in hand, we can now focus on writing reusable scripts. In the next lesson, we will learn to do just that.

Glossary-AWS

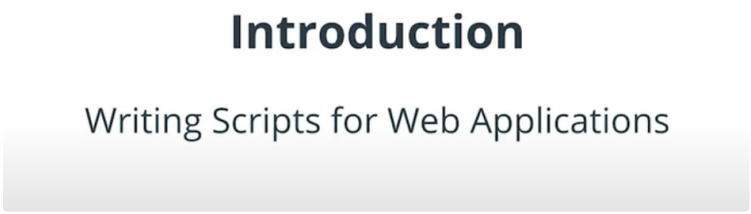
New Terms In This Lesson

- **IAM keys:** Identity access management keys can identify you to the AWS CLI. These keys hold associated permissions to give access to your account.
- **Runbook:** A simple step-by-step tutorial explaining how to do operations on an application.
- **Cache-Control:** HTTP headers telling the browser how long it needs to cache certain files. To ensure browsers take the new content of our HTML files we will sometimes need to force the browser to revalidate the content of the files in S3
- **ACL:** Access Control List is a way to control who can access S3 buckets.

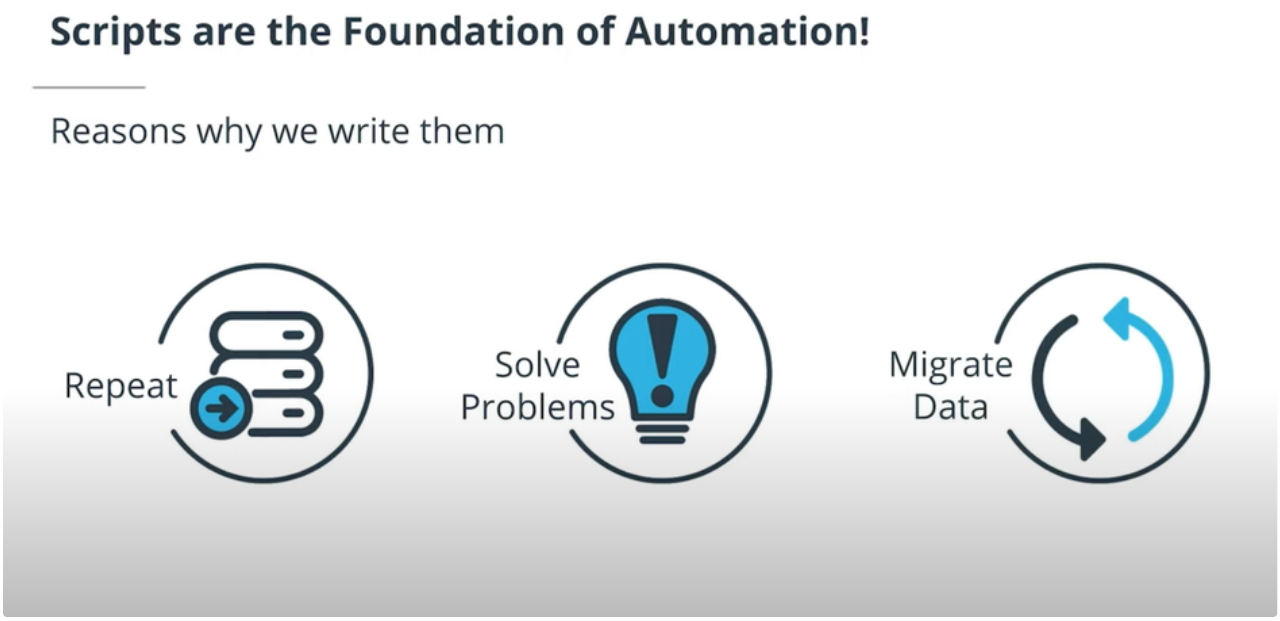
Write scripts for web applications

Introduction-

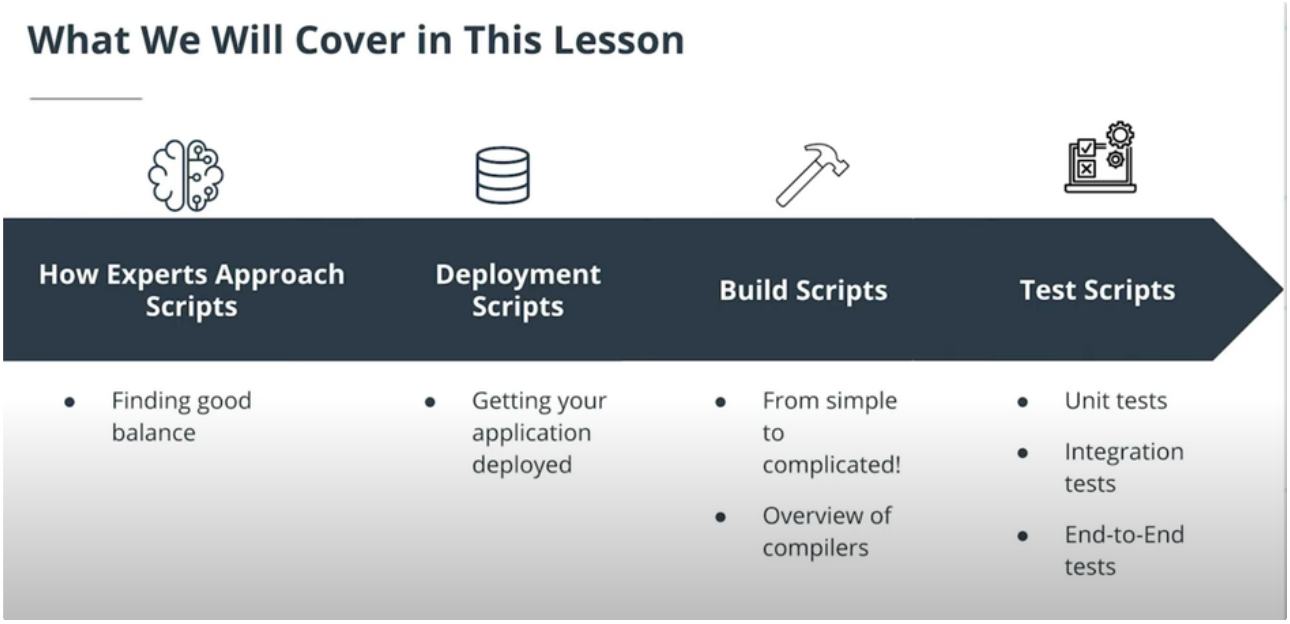
Lesson Introduction



Let's see how we can write scripts for web applications. But first of all, you might be wondering, why do we write scripts?



It is because scripts are the foundation of automation. Here are a couple of reasons. Scripts make everything repeatable. It means that you can make every action on your operations such as uploading, building, and testing. You can make all of those repeatable. They can also solve problems. If you have a complex operation that you don't want to be doing with the command line interface or that you don't want to be done directly in the AWS console, you can call the command line interface directly in the script. So that it becomes solvable more easily and you can call it with one command. Lastly, they are also really good at migrating data. Sometimes if you need to move from one database to another one, writing a script for doing this will be really useful. Let's see a little bit of what we will cover in this lesson.



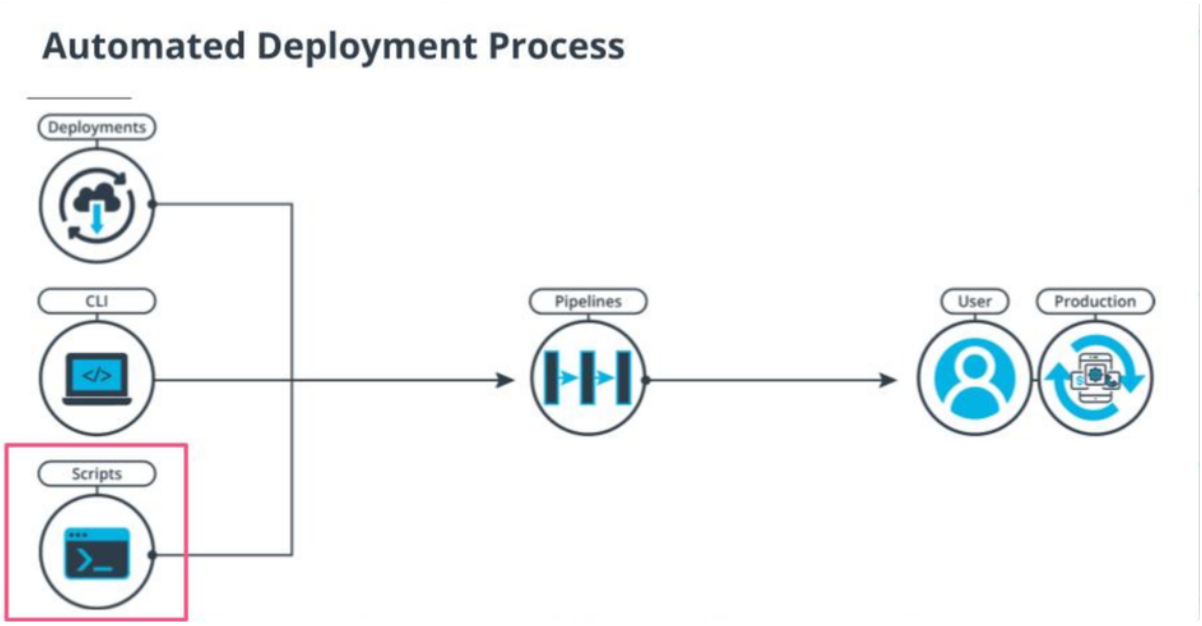
First of all, we will see our experts approach writing scripts, and it's all about finding a good balance, but we're going to dive on this a little bit later. We will learn also about deployment scripts. How we can get your application deployed. After this, we will go right into the build scripts. We will see simple ones as well as complicated ones. We will also get an overview of what is a compiler. We will end by explaining what our test scripts from unit tests, integration tests, up until end-to-end test. We will talk about test more from the standpoint of a script and deploying an application.

Summary:

In this lesson, we will learn about the different ways you can write scripts for your application. We will learn the following:

- How experts approach **writing scripts**
- **Deployment scripts** in order to deploy the application
- **Build Scripts** in order to package and build applications
- **Test Scripts** in order to catch potential bugs in an application

As shown in the image below, we will focus on writing scripts throughout this lesson.



Why write scripts?

Scripts are the foundation of automation. They enable you to **repeat actions** in a predictable way, **solve complicated problems** that would take multiple CLI commands to fix, and even **migrate data** from one database to another.

As an application grows and becomes more mature, you will gradually adapt your scripts and they will grow with the application. You will need to maintain them as you maintain your code in order to make your application well-organized and easy to work with.

How Experts Approach Writing Scripts

Playing the balancing game

Let's see how experts approach writing scripts. It's all a balancing game.

Balancing Customization and Maintenance

We need to balance customization and maintenance. On one hand, you don't want to be chained by maintaining your script and having a complicated one that is really difficult to maintain. But on the other hand, you also need to have all of your features. You will want to spend a good amount of time just making sure your script is easy to maintain and finds all your needs.

Steps to Follow When Creating a Script

1. Start from the commands you know and used in a CLI.
2. Is a pre-built script already available?
3. Simple scripts go in the package.json.
4. Complicated scripts should be in a script file (E.g., *.sh*).

Let's see a little bit, what this means step-by-step. Those are the steps I follow when creating a script. First, I start with the commands that I know and that I use in a CLI. This is always a good place to start because this is something you already know. You should also look if there is a pre-built script available. Something like angular build or create react app build would save you a lot of time because it's not something that you have to maintain yourself. You should also note that simple scripts should go into package.json. While you could write scripts somewhere else, in package json, it is where as a web developer, using JavaScript tools, you will be calling your scripts. Complicated scripts should be done in a script file. Here we're talking PowerShell or shell scripts but also please try to have a place where you can call the script from the package.json, because this is again, your main tool as a JavaScript developer.

Summary

A **script** consists of a command or a series of commands. Our end goal is to accomplish some tasks inside this script. This is why we will first try to look for some commands that accomplish our goal.

Experts will follow these steps in order to create a script:

1. Start from the commands they already know.
2. Look for pre-built scripts (like Angular build).
3. Add simple scripts directly in the `package.json`
4. Add complicated scripts inside a script file (for ex: deploy.sh) that is called from the `package.json`.

It is hard to draw a line on what makes a script simple or complex, but here are some indications that **your script is getting complex**:

- You are passing two or more options to the CLI commands
- You are using multiple CLI tools in the same script (for example: calling both `eb` and `s3`)
- You are using multiple subsequent commands
- The commands are long and hard to read

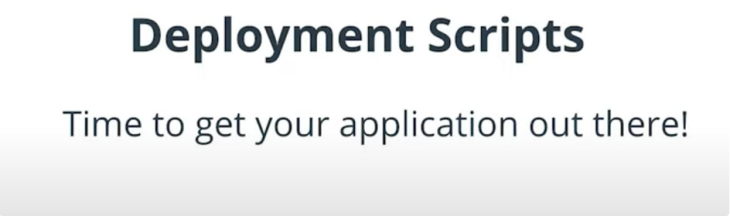
Scripts in the Context of a JavaScript Application

JavaScript developers mostly use **npm** or **yarn** as package managers. These two package managers also provide an easy way to execute scripts that are defined in the script section of the `package.json` file. This is why we will always add all our scripts directly into the script section of the `package.json`.

New Terms

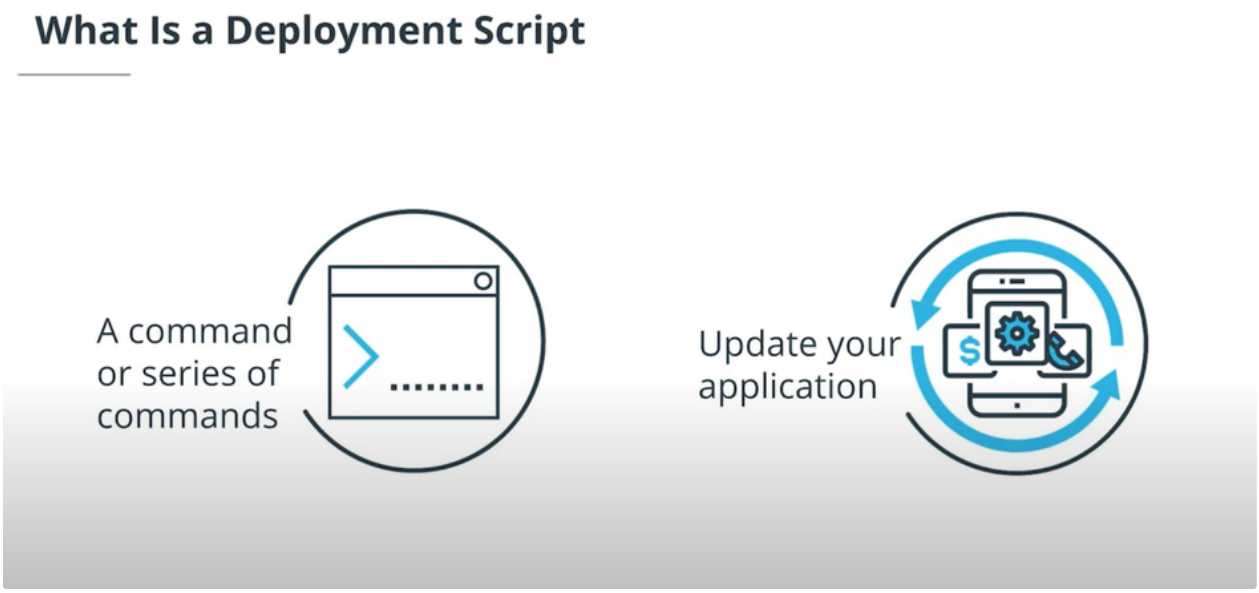
- **Shell** is a program that processes commands and returns the output. It is one of the most popular terminal programs out there.
- **Bash (Bourne again shell)** is similar to Shell, with more advanced features. It is available by default on most Mac and Linux systems.
- **Powershell** is a terminal program available on Windows. It has similar features to Shell and Bash but the syntax is different.

Deployment Scripts

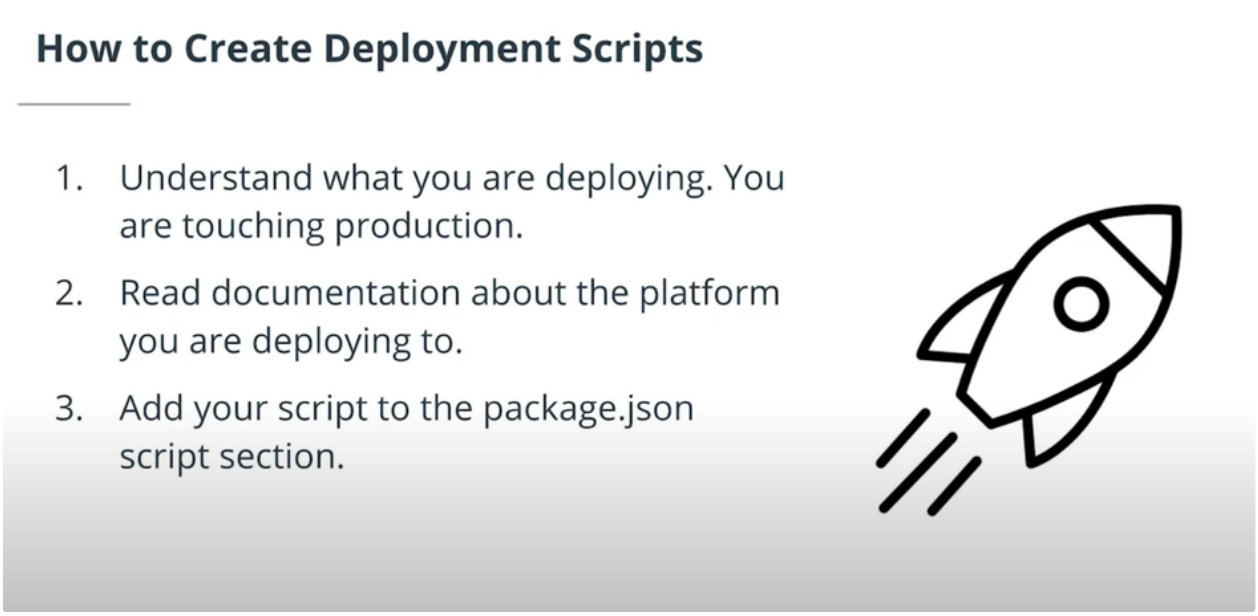


What Is a Deployment Script

Let's see how we can write deployment scripts. This is exciting because it's time to get the application out there and deployment scripts will help you do that. First of all, let's try to understand what is a deployment script. It is a command or a series of commands that will update your application in the production environment.



This is important to understand because that means you are touching what customers will see, what your users will actually be interacting with. Let's see how we can create deployment scripts.



First of all, it's important that you understand what you're deploying. If you're deploying an API, a UI, or a database, you must understand what this implies. As we mentioned in the last section, deploying a UI means your customers will see something different. But also declaring an API means that the data or the way that you send data to your UI will change. It's important to have an understanding of this. In order to do this, go ahead and read documentation about the platform where you're deploying. This could be the documentation about the AWS S3, about Elastic Beanstalk to get a good understanding of what you're doing. The last step after you're done writing your scripts is to be sure that your script is inside your package.json.

Key points:

A **deployment script** is a command or a series of commands that will update your application.

Deployment scripts are used to deploy your application. This can mean deploying your latest code to a platform like Elastic Beanstalk or updating the content of a static website hosted on S3. These scripts can call different commands such as `EB deploy` or `AWS S3 CP`.

These steps are useful when it comes to creating a deployment script:

1. Understand what you are deploying
2. Read the documentation about the platform you are deploying to. This will help you identify which script commands you can call inside your script.
3. Add your script to the `package.json` script section.

> Please remember that they should be included inside your `package.json` in order to be easy to call with npm or yarn.

New Terms

- **Yarn** is a package manager similar to npm.

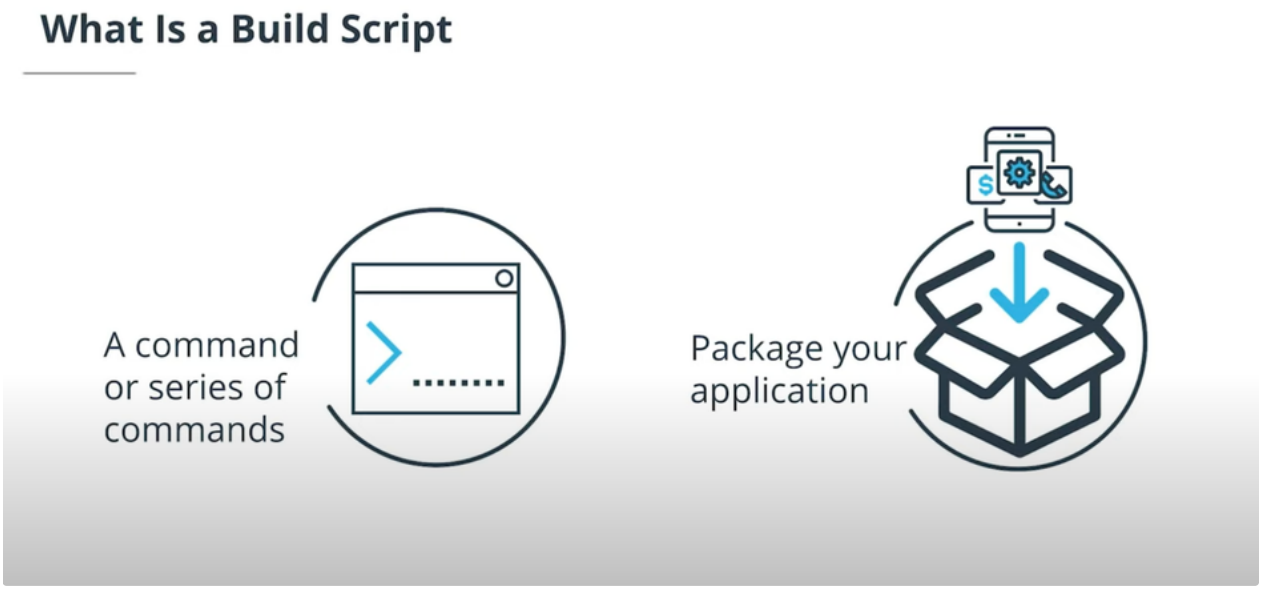
Further Reading

- Scripts inside package.json: This is a great small read to understand more in-depth how you can add scripts to your `package.json`.
- npm-run-all and concurrently are simple npm packages that give you additional tooling when you want to run more complex scripts from your `package.json`.

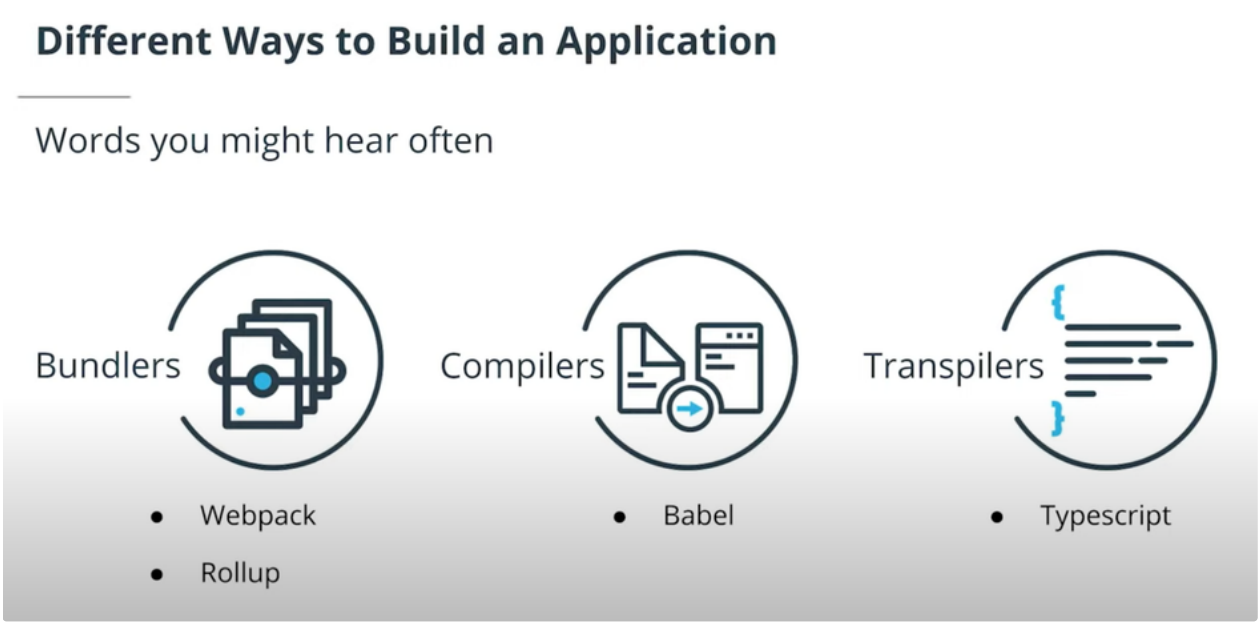
Build Scripts

What Is a Build Script?

Let's see how we can write build scripts. We have used so far in this course builds scripts really often we have called NPM run build to mostly deploy our front-end application. Let's understand a little bit how this works. Because one thing that is important to understand is that we cannot deploy something if you have not built it up before.

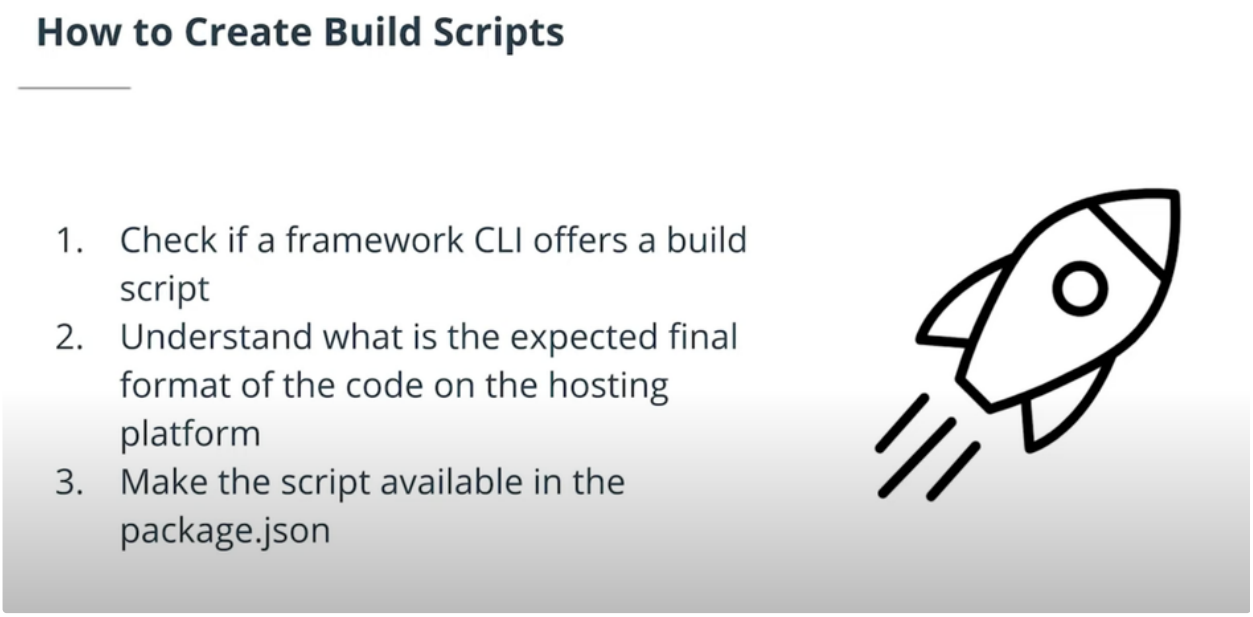


What is a build script can itself? It is a command or a series of command that will package your application. When you were using NPM run build. This was a pre-available script made available via the react command line interface. The same one would exist for angular as well. Let's dive a little bit and see what really happens in those scripts.



Building an application can mean different things depending on the type of application you are building. Let's see a little bit to common words you might hear when we're talking about a build script. You might have heard about Bundlers. These are responsible for taking source code such as JavaScript file HTML or CSS, and putting them into a smaller format that is more digestible for the server or the browser. Famous ones would include webpack and rollup. You might also have heard about compilers. Compilers are a little bit different. They will take code from more modern JavaScript, such as ES 6 or ES 7. They would transform that code so that it fits older versions of JavaScript, such as ES 4 or even earlier. Bundlers is a good example of this. Lastly, there are transpilers. Transpiler serve to take in another language to touch Typescript and change it into JavaScript. Typescript is a good example of what is a transpiler language. Normally, you would use a combination of those tools to build an application.

Let's see a little bit how we can create build scripts.



The first thing you might want to do to verify if the framework you're using, such as Angular or React offers a build script. This is often the easiest and best way to go because managing your own build scripts can be complicated. You need to understand the final format of the code that you're hosting on a platform. As I mentioned, hosting something on a browser will mean HTML JavaScript, and CSS so you want to have your final format to this. If you're simply hosting a server you can simply call node with the server file so the building becomes a little bit more simple. As with every script once you are done creating a build script, you want to make it available in the package.json.

Key Points:

A **build script** is a command or a series of commands that package your application.

You have been using build scripts really often throughout this course. Some frameworks like React and Angular make them readily available, making your life easier. While these frameworks are great and provide good scripts, it's still important to understand a little bit what they are doing. Here are some concepts that you would want to explore if you want to create your own build script:

- **Bundlers like Webpack and Rollup** are able to package your application code and all its dependencies. They are responsible for packing your code in a format that is more compact while still understandable by browsers and servers.
- **Compilers like Babel** let you use more advanced features of the latest JavaScript versions while maintaining compatibility with older browsers.
- **Transpilers like TypeScript** extend the base capacities of JavaScript by adding extra features not present in the base language.

You can take the following steps in order to create your build scripts:

1. Check if a framework script is available. Frameworks like Angular often offer pre-made build scripts.
2. Understand the final format of what you are building.
3. Make the build script available inside a `package.json`.

This is a lot to digest but it is within your reach!

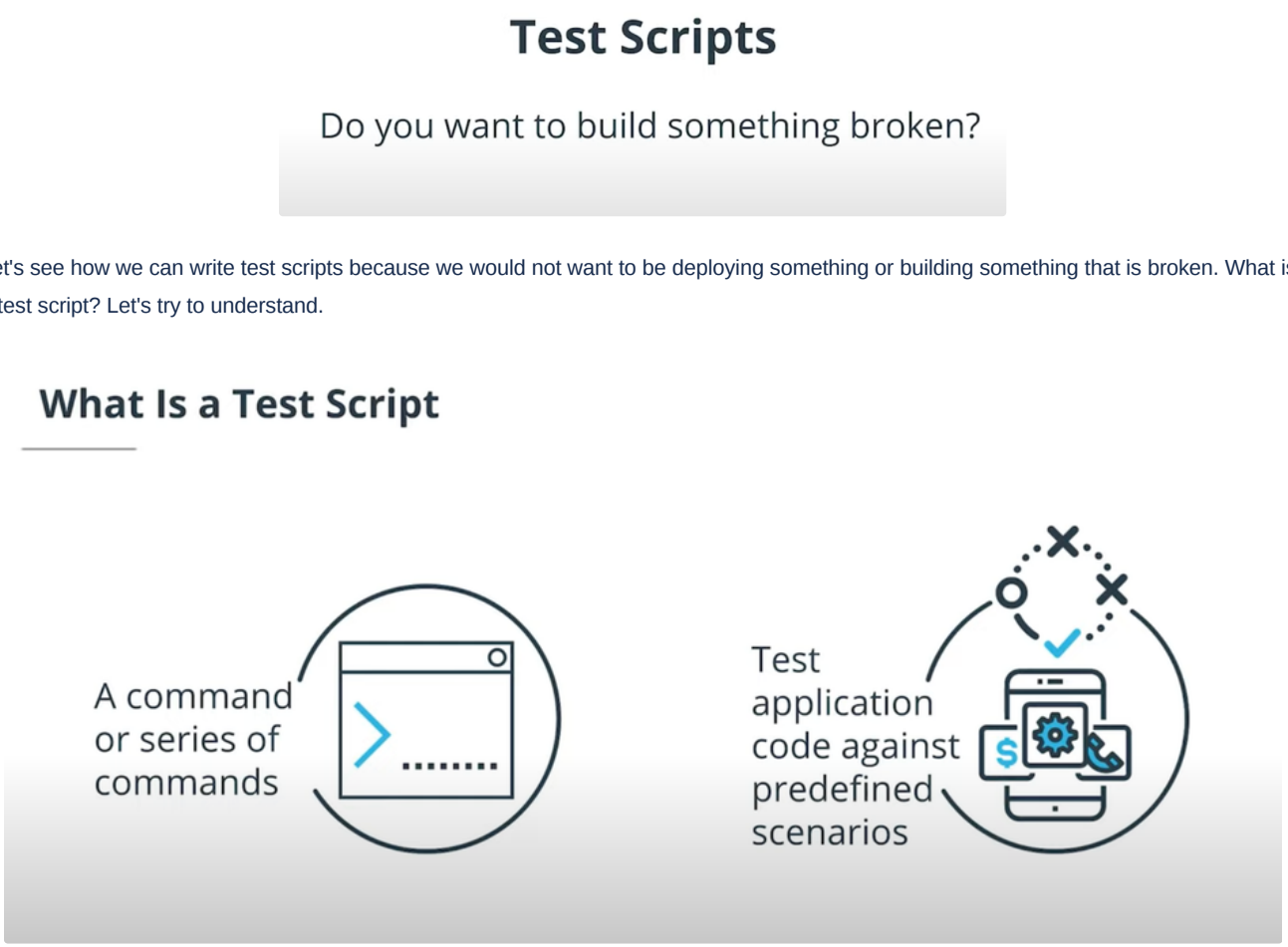
While it might seem like the world of bundlers, compilers, and transpilers is a complex one, it is one that you can understand! This course focuses on deploying an application, but if you want to learn more, try making your own Webpack config or setting up TypeScript by yourself in some practice projects!

Further Reading

- Webpack book by SurviveJS is a great online tutorial to understand Webpack in depth.
- What is a tsconfig: This documentation page on the TypeScript website offers a great explanation of how Typescript gets transpiled into JavaScript.

Test Scripts

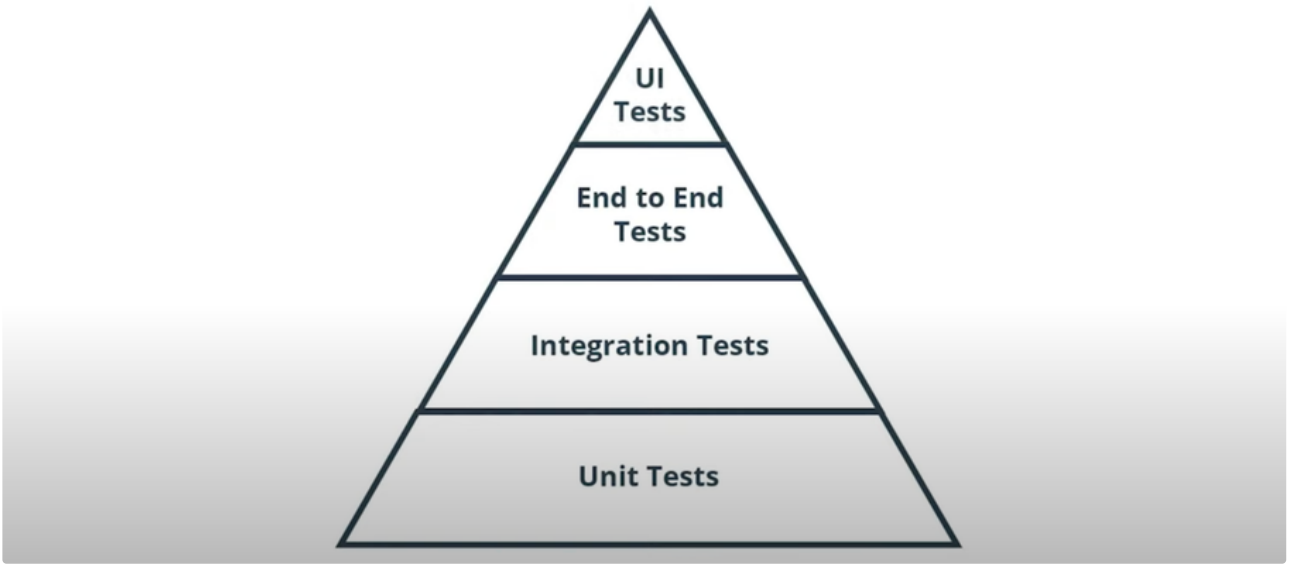
What Is a Test Script?



Let's see how we can write test scripts because we would not want to be deploying something or building something that is broken. What is a test script? Let's try to understand.

It is a command or a series of commands that will test an application code against predefined scenarios. Let's understand a little bit more about all the different types of tests.

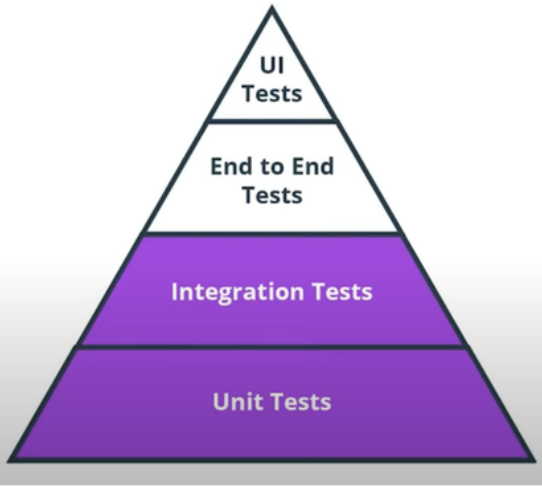
Different Types of Tests



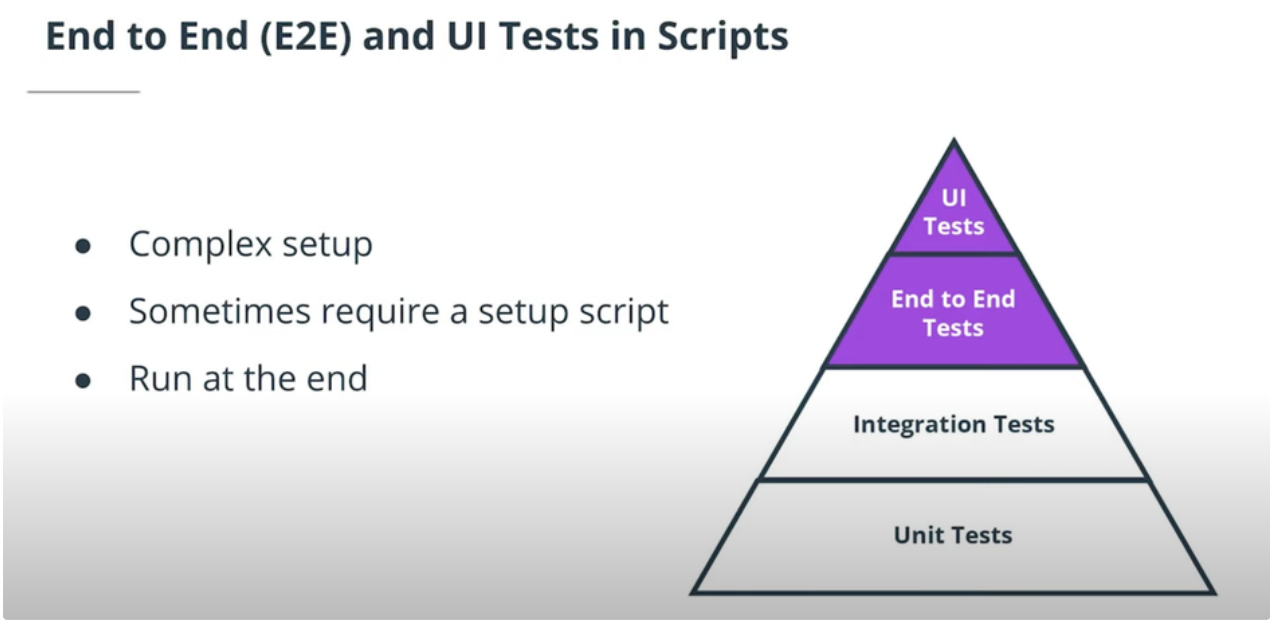
We can first start with the unit test, which tests normally simple things such as function code, and then we have the integration test, which will test different modules of the code together to find mistakes between them. We then have end-to-end tests and UI tests. Those two last categories are normally the most complex tests, but we'll dive into this a little bit later.

Unit and Integration Tests in Scripts

- Easiest tests to run with scripts
- Popular frameworks are Jest and Mocha
- We run them first



Let's talk first about the unit and integration tests. They are often the easiest test to run, so these tests run quickly and are normally pretty simple to write. There are popular frameworks such as Jest and Mocha that make it even easier. This is why we run them first because those tests are simple. Normally, we want to be sure that they all run before going on to the more complicated things first.



After this, we have the UI and end-to-end tests, which are normally more complex and require more setup. They often even need a setup script to have a dedicated test server, so this is why we will run them at the end. These tests are normally more complex, but they also complete the testing with more complicated use cases.

Key points:

A **test script** is a command or series of commands that test application code against pre-defined scenarios.

Testing is a prevalent topic in software development. When we think about deploying our application, we use tests in order to have confidence that we are deploying solid code that will not crash or introduce bugs.

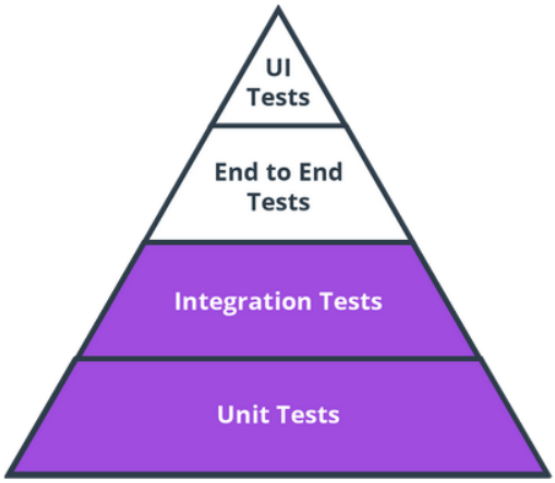
Bug-free software, however, does not exist, for this reason, we must do our best to test the application and build confidence in our automated process.

Tests are called via scripts most of the time. We will normally run our tests in this order before deploying an application:

1. **Unit tests** will be run first since they are the fastest and most simple tests.
2. **Integration tests** are a little bit more involved, so we will run them directly after unit tests.
3. **End to End (E2E) and UI** tests are often complex and involve some form of setup. For this reason, we will run them last.

Here is an example package.json script running unit tests with the Jest framework:

```

"test:ci": "jest --ci --coverage"
```



Testing Pyramid Diagram

New Terms

- **E2E tests or End To End tests** are tests that will ensure a complete workflow is working in your application. These workflows could be signing up a user or buying a product on a website.
- **Test Setup** is the step that some testing suites will need. For example, Cypress is a UI test suite that will start a development server and a mock browser to execute your tests. This setup is normally a CLI command called by test scripts.

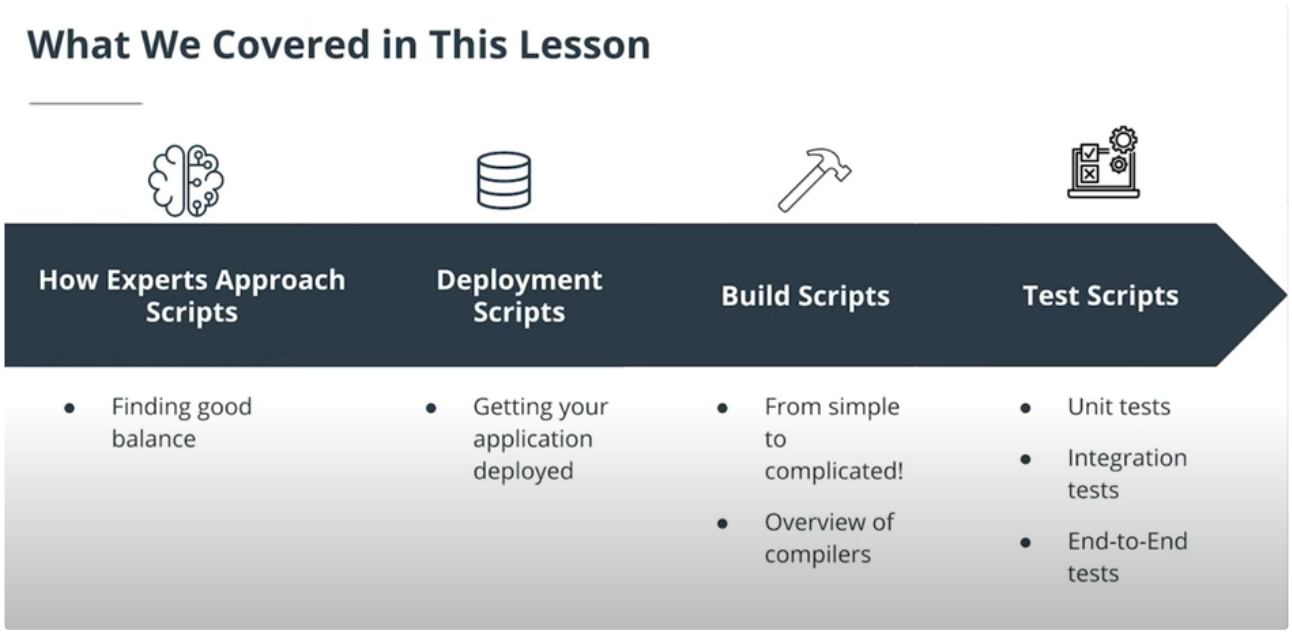
Further Reading

- Cypress in a nutshell is a great introduction video to Cypress and UI testing.

Write scripts for web applications - Lesson Recap

What We Have Learned

Let's see a little bit what we have seen so far in this lecture. We have covered a lot.



1. The first thing we covered was our expert approach scripts. We talked about finding a good balance between a complicated script and one that has all the features you want. But for now, let's really just focus on making simple scripts work well. Making more complicated scripts will come on later when you have a little bit more experience, and feel more confident in your tooling.
2. We have also learned about the permanent scripts, how to get your application deployed. We have learned about build scripts that can be simple or complicated. But again, we focused on for now the simple scripts that do what we need.
3. We also have an overview of the compiler that just went back in Rollup.
4. Lastly, we saw test scripts, we saw unit tests, integration tests, and end-to-end tests.

Let's now go to the next lesson where we will learn how to combine all of this into a continuous integration and continuous deployment outline.

That was a lot of information about all the possibilities that scripts offer! In the end, it is important to remember that scripts are just a collection of commands that are executed in a specific order. The more experienced you will become, the more you will start to see patterns inside the different scripts you use in your work. For now, we are concentrating on deployment, build, and test scripts.

These different categories of scripts will help us create an automated pipeline in the next lesson.

Glossary_

New Terms In This Lesson

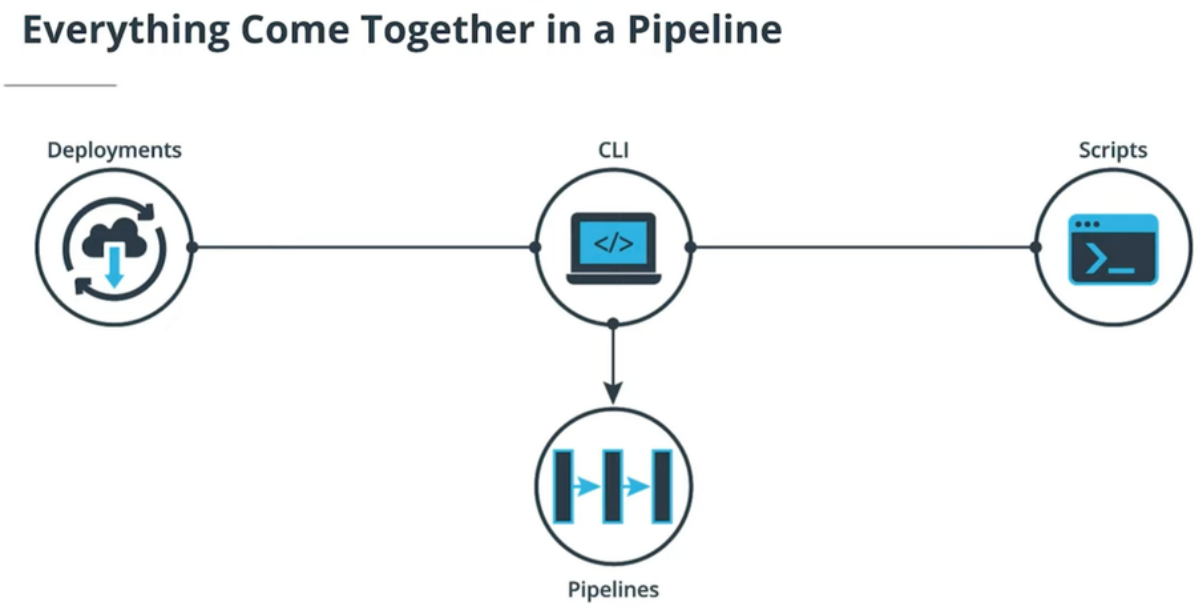
- **Shell** is a program that processes commands and returns the output. It is one of the most popular terminal programs out there.
- **Bash (Born again shell)** is similar to Shell with more advanced features. It is available by default on most Mac and Linux systems
- **Powershell** is a terminal program available on windows. It has similar features to Shell and Bash but the syntax is different.
- **Bundlers like Webpack and Rollup** are able to package your application code and all its dependencies. They are responsible for packing your code in a format that is more compact while still understandable by browsers and servers
- **Compilers like Babel** let you use more advanced features of the latest JavaScript versions while maintaining compatibility with older browsers.
- **Transpilers like Typescript** extend the base capacities of Javascript by adding extra features not present in the base language.
- **E2E tests or End To End tests** are tests that will ensure a complete workflow is working in your application. These workflows could be signing up a user or buying a product on a website
- **Test Setup** is the step that some testing suites will need. For example, Cypress is a UI test suite that will start a development server and a mock browser to execute your tests. This setup is normally a CLI command called by test scripts.

Configure and Document a Pipeline

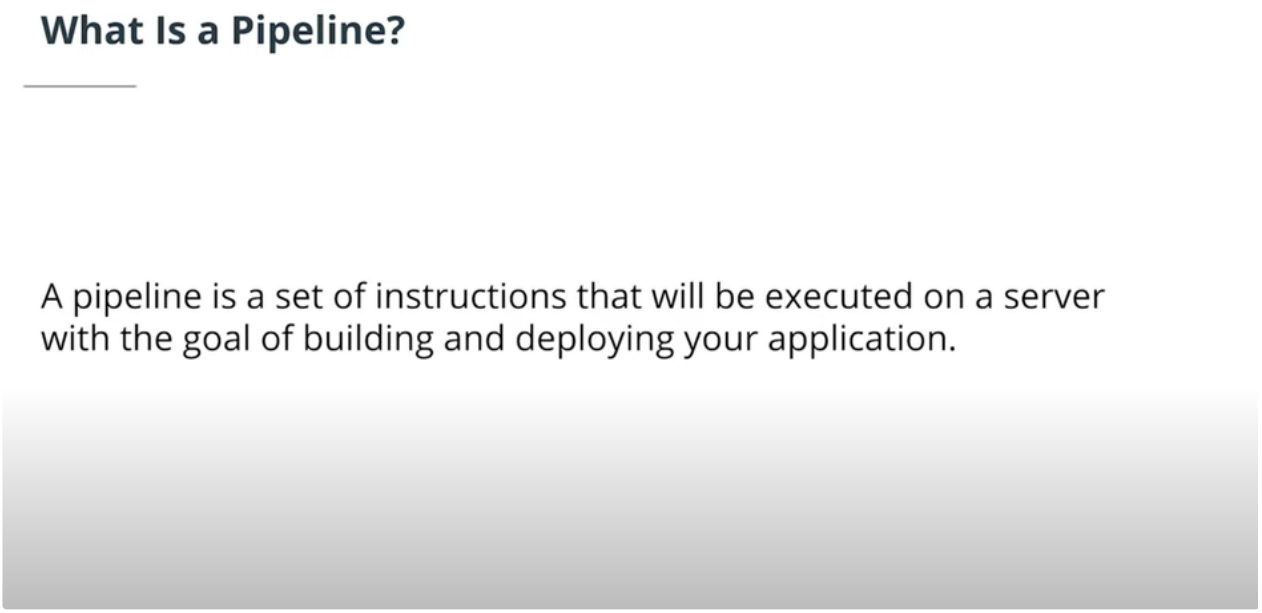
Introduction_

Lesson Introduction

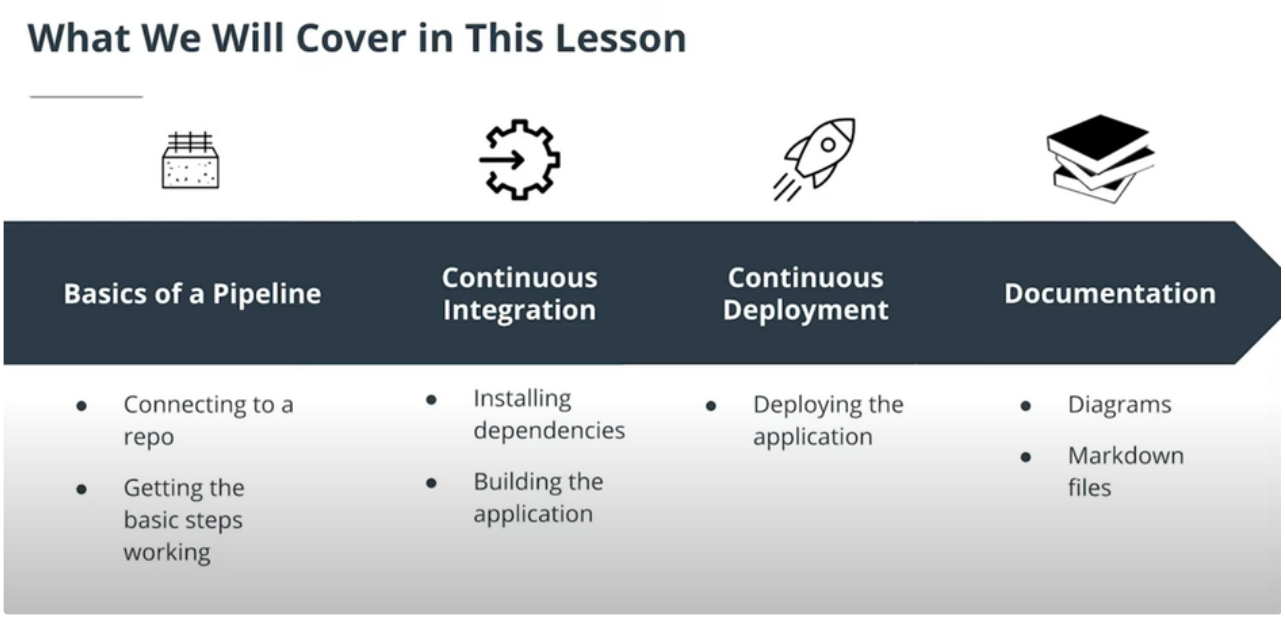
This is the final lesson. You have learned many skills so far. Separately, those skills are great, but when they come together, they make something even greater. That is a pipeline. The pipeline we will create in this lesson will really mean automation. It will mean the automatic deployment of your application. Finally, getting to your customers, it will empower you, it will mean a better product, and it also will mean happier customers. Let's see how heavy things come together inside a pipeline.



You have done so far a lot of things. In this course, you have learned how to deploy with a UI, with a command-line interface, and insight scripts. The beautiful thing about a pipeline is that all of these things happen inside the pipeline. All the knowledge that you have used will be inside one file where you do all of those in the same place, and you repeat it over and over automatically. But exactly what is a pipeline? Let's try to understand.



Let's understand deeper what we will cover in this lesson.



The first thing we will cover is the basics of a pipeline. We will learn how to connect it to a repo, and we will get the basic steps working to make a hello world with our pipeline. Right after this, we will be doing continuous integration. We will endorse steps, install dependencies, and build our application. Right after this, we will learn continuous deployment, which means deploying the application. Right at the end, we will learn about documenting your pipeline. We will be doing diagrams and using Markdown files in order to properly document what is happening in your pipeline.

All that we have learned so far will come in handy when it is time to create a pipeline! We could define a **pipeline** in the following way:

A pipeline is a set of instructions that will be executed on a server with the goal of building and deploying your application.

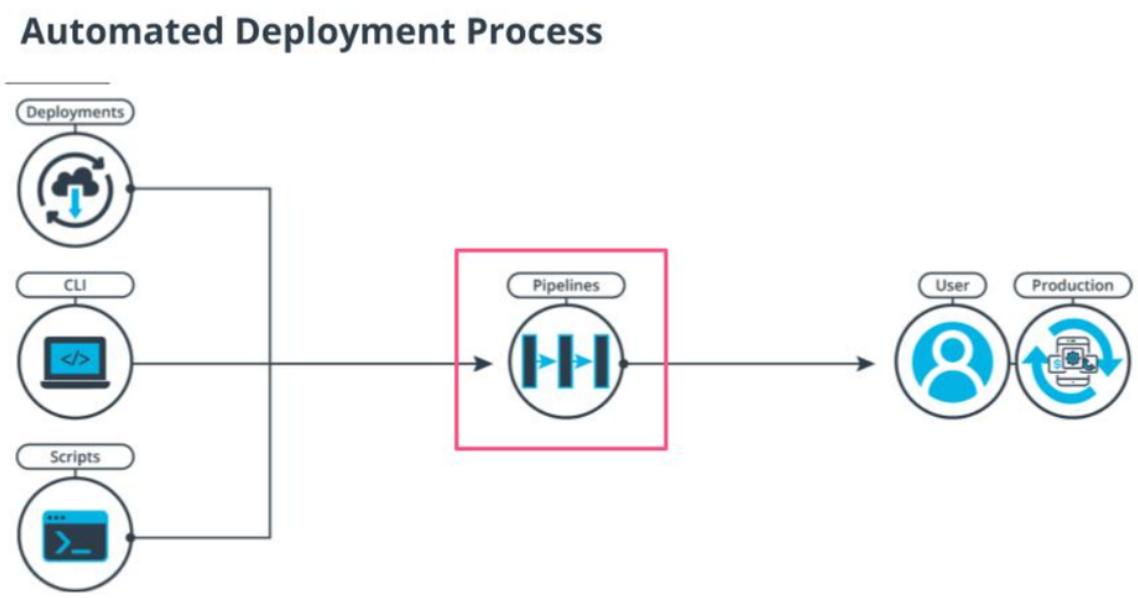
Those set of instructions really come together inside a pipeline in a way that allows us more flexibility! An efficient pipeline can benefit developers in the following ways:

- Faster feedback about the code
- Getting features deployed faster

To learn pipelines we will cover the following topics in this lesson.

- **Basics of a pipeline**: We will learn how everything comes together and enables automation around deployments.
- **Continuous Integration**: We will understand the different steps that a pipeline executes that form the CI portion of this.
- **Continuous Delivery**: We will learn how deployments can be automated after an application is built and tested.
- **Documentation**: We will learn to document a pipeline and the different operations around an application.

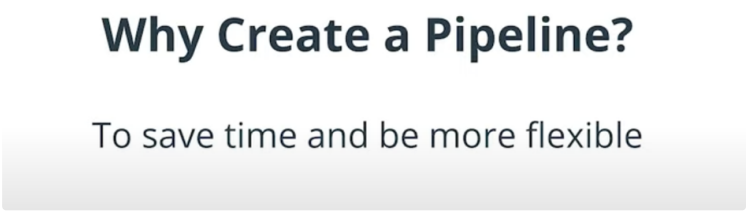
As shown in the image below, deployment, CLI and scripts come together in a pipeline. We will focus on pipelines throughout this lesson.



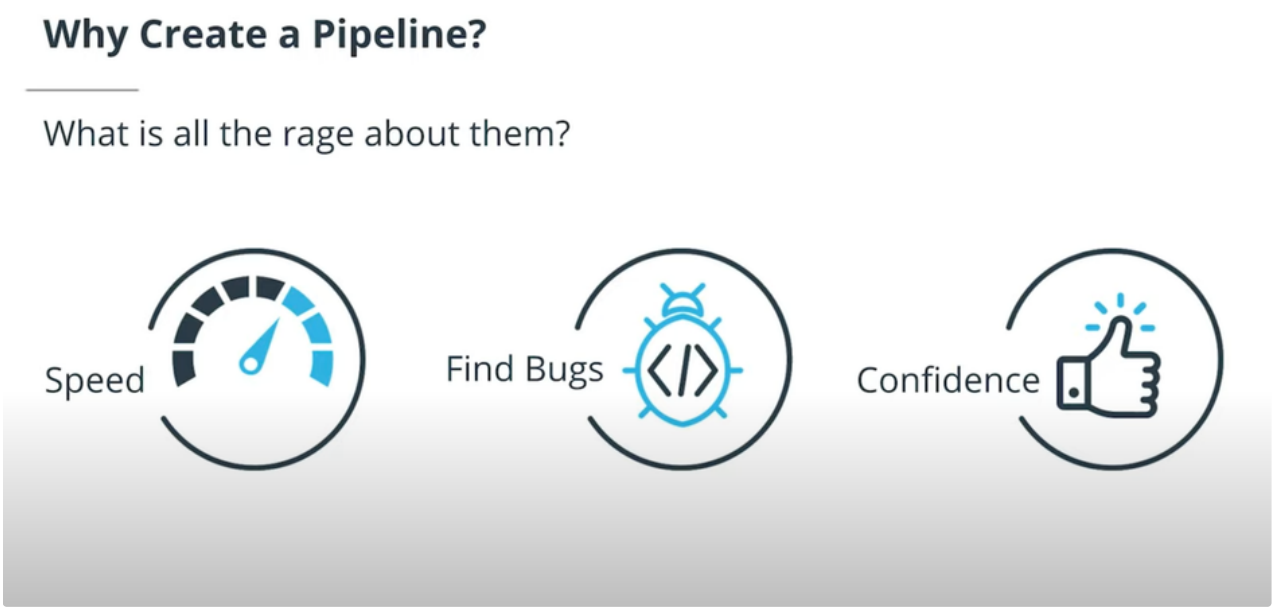
Pipelines in Automated Deployment Process

Why Create a Pipeline?

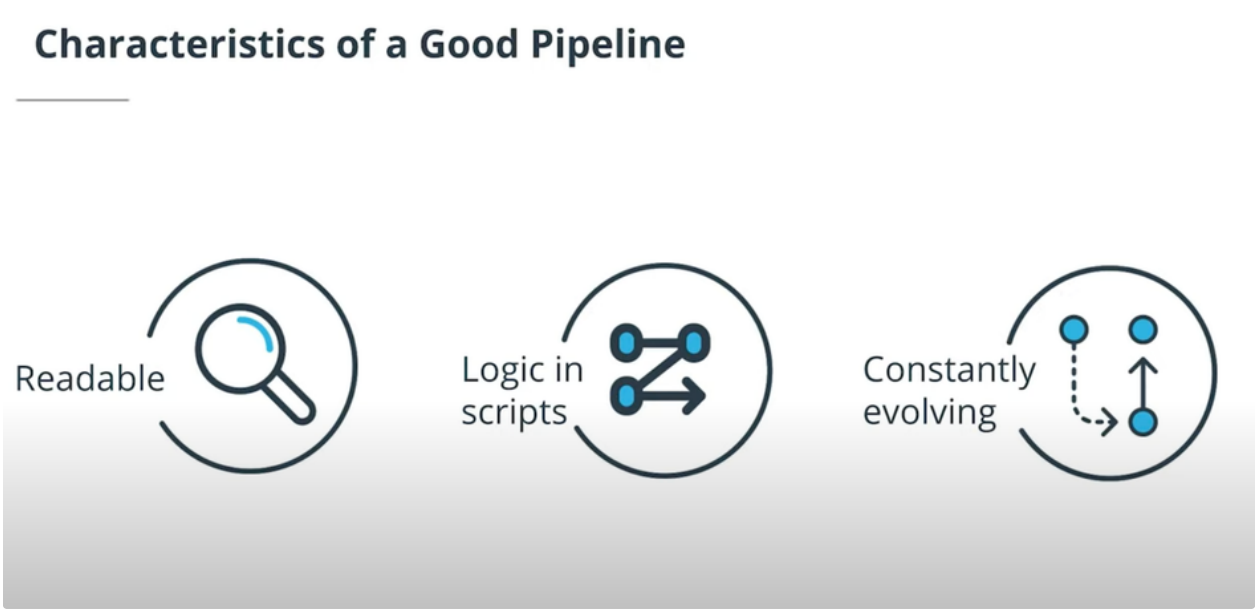
Why Creating Pipelines Matter?



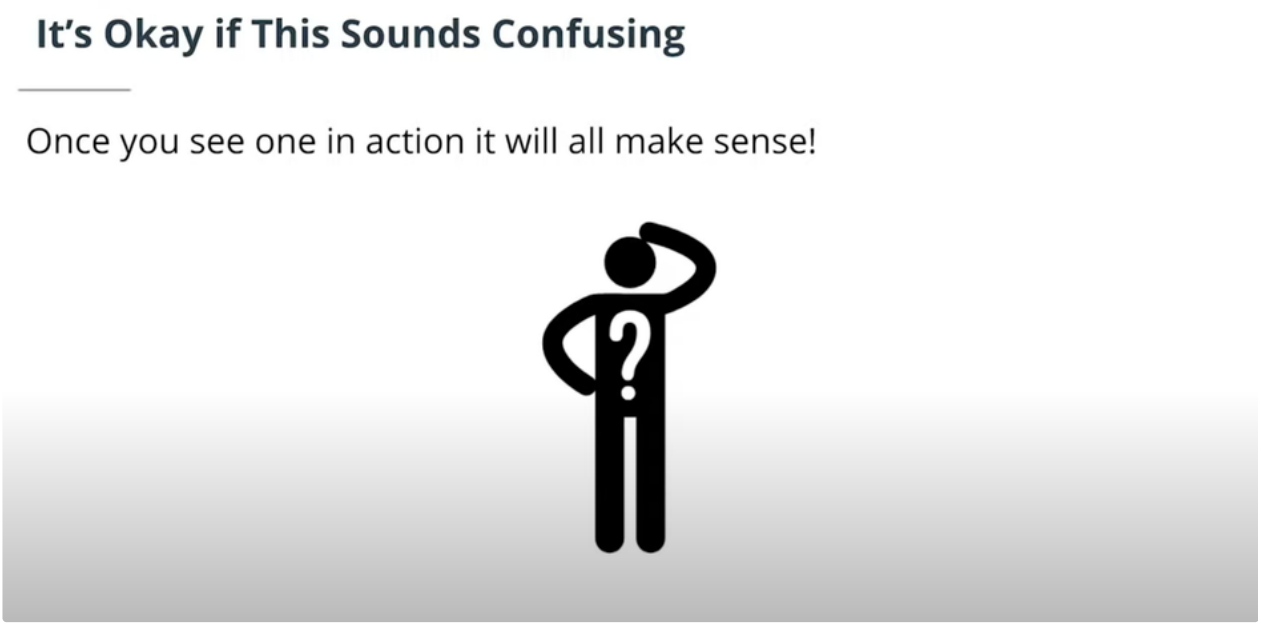
You might be wondering, why do we create pipelines? The answer is really that they save time and they allow you to be more flexible. But what is all the rage about them? Some people are really talking about the need for automation. Let's understand a little bit why they are so popular.



The first thing is the speedup. Instead of doing all those steps manually, they are all done in an automatic way, so that saves you a lot of time. They also allow you to find bugs early on by testing and building your application every time you're pushing to GitHub, this will help you ensure that everything is bug-free as best as you can do it. That will increase your confidence. Knowing that your build is working and passing inside the pipeline means that your application will be ready to deploy, so you can have more confidence in your code when you do this.



Let's a little bit think about what makes up a good pipeline. What are the characteristics we look for? We want it to be readable. Your pipeline should normally be in a configuration file so you need to make sure that you can understand it, have good comments, and read it quickly. We also want to have the complicated logic inside scripts. We don't want to be doing all of our complicated things directly in the pipeline file. Another characteristic is that pipelines don't stay the same, they evolve through time. As your application gets bigger and more complex, your pipeline will reflect this and will evolve to serve better the needs of your application.



This might sound a little bit confusing, but don't worry. Once you see it all in action, it will make sense, so the question marks you might be having right now, will all fade away.

Key Points:

Pipelines are widely used in the development landscape. Companies create them for various reasons, and they have the following benefits:

- **Speed**: Automatically performing all the steps of a pipeline is faster than doing it manually each time.
- **Finding bugs:** By running tests each time we are trying to deploy, we are able to find bugs earlier.
- **Building confidence in your release:** When you release software that has passed different quality steps, you can be more confident in its quality.

However, even if pipelines save a lot of time, it is important to treat them like an integral part of the software and build them with care. Here are some characteristics of a good pipeline:

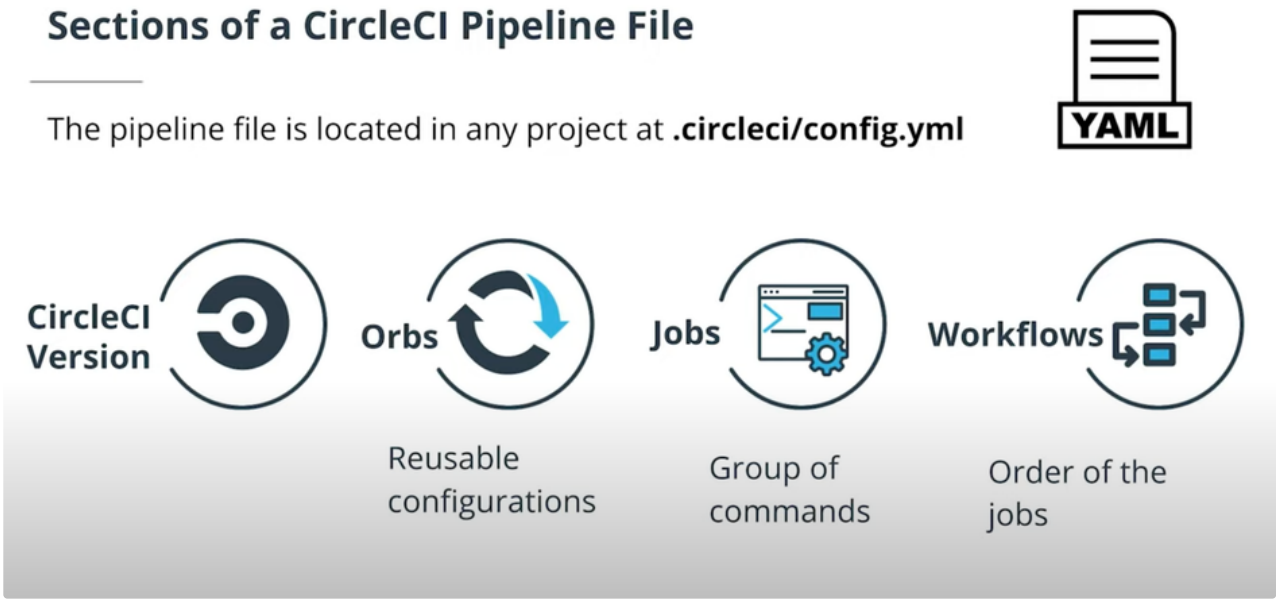
- **Readable:** You should try to make your pipeline concise and easy to read. Having multiple long commands inside the pipeline code will make it hard to understand.
- **Logic should be inside the scripts:** This allows for more portability to other pipeline providers if you ever need to move away from your current provider.
- **Constantly evolving:** This means that a pipeline should fit the need of the project and evolve as your project gains maturity.

Writing the Basic Pipeline

What are the Basics of a Pipeline?

Let's see how we can write the basic pipeline just to understand how this all works.

Many companies offer pipelines as a service. For this course, we chose to teach you CircleCI. It has all the features we need to deploy our application and it's also really popular in the industry. This knowledge will serve you in your career no matter where you work as you can transport your knowledge to also other type of companies other than CircleCI.



Let's see a little bit what is a pipeline file. First of all, it is a YAML file. This file contains instructions that will run your pipeline. With CircleCI it's normally located at.circleci/config.yml. There are many steps to this file. There is the version that you will use in CircleCI. Then there are orbs, which are a set of tools, reusable configurations could be the best way to describe them. They let you, lets say, set up a node or setup AWS CLI in a much faster way. Then there are jobs. These jobs are what is actually running scripts and calling CLI command. They're regrouping different types of command and then these jobs come together inside workflows, which are the order of the jobs and the sequence in which they're going to run.

Orbs

Orbs are useful to set up the pipeline server quickly and add tools to it.

Examples:

- Node.js
- AWS CLI
- Python
- EB CLI

Let's see a little bit more in depth what you can do with orbs. As we mentioned, they are useful to set up your server more quickly and add tools to it. A good example, like I mentioned, would be Node.js or AWS CLI. But you could also be setting up Python or the Elastic Beanstalk CLI. Orbs are really a set of precooked, pre made recipes that you can use on your pipeline server to have everything run smoothly and not have to set it up yourself. Let's talk about jobs more in detail.

Jobs

Jobs are groups of commands where you can take actions on your application.

Examples:

- Installing node_modules
- Calling a deploy script
- Calling a build script

The jobs, as mentioned earlier, are a group of commands that you can take actions on your application. Those actions would be installing your modules by running npm install, would be calling a deploy script or a build script. Basically, any script that you can think of can be called from here, inside jobs.

Workflows

Workflows let you dictate how jobs are called.

Examples:

- Executing a job without anything special
- Making a manual approval flow
- Making a job depend on another one

Now let's talk about workflows. Workflows let you dictate how you call the jobs and in which order. For example, you can just execute a job. You could just tell the workflow execute the build job. Or you could make a manual approval flow. This would mean that a human needs to go and click to approve, let's say, before deploying an application and they can also depend on one another. You could say that you cannot deploy without having built before. Workflows is really where you dictate how everything is going to go.

Key Points:

In this course, we will be using **CircleCI**. There are many other pipeline-as-a-service companies that offer a similar set of services, but for the purpose of this course, CircleCI presents a **great set of features** and has the advantage of being **popular** in the industry.

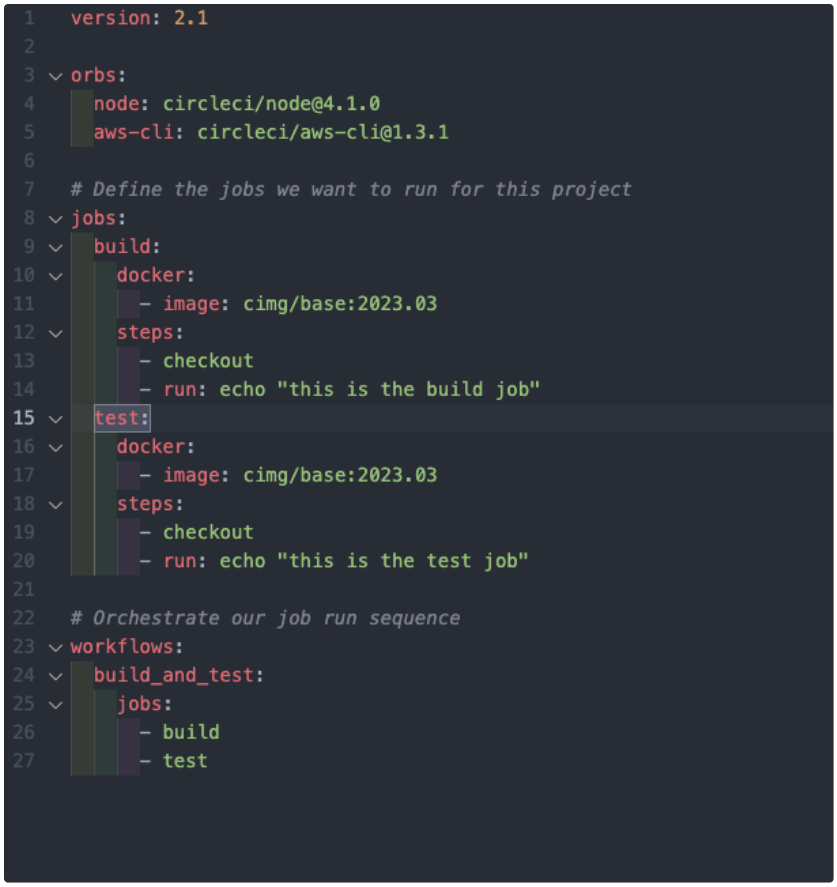
Pipelines are normally written **inside configuration files** as **a list of steps**. In the case of CircleCI, this file will always be located inside a `.circleci` folder and will be named `config.yml`.

It contains the following sections:

- **CircleCI version**: This is simply indicating which version of the platform our pipeline should use.
- **Orbs** are a set of instructions created by CircleCi that allow us to **configure the pipeline** on which we will run our actions. These instructions will instruct the server to setup specific software on the server executing our pipeline. We could use orbs to setup node.js or install the AWS CLI for example. Orbs are **not always present** in a pipeline.
- **Jobs** are groups of commands that we want to run. This is where we will run commands to **install, build or deploy our application**.
- **Workflows** are instructions about **the order of the jobs**. They allow us to create complex flows and specify manual approvals. Workflows are **not always present** in a pipeline.

Parts of a config.yml File

Simple configuration examples:



```yaml
version: 2.1

orbs:
  node: circleci/node@4.1.0
  aws-cli: circleci/aws-cli@1.3.1

# Define the jobs we want to run for this project
jobs:
  build:
    docker:
      - image: cimg/base:2023.03
    steps:
      - checkout
      - run: echo "this is the build job"
  test:
    docker:
      - image: cimg/base:2023.03
    steps:
      - checkout
      - run: echo "this is the test job"

# Orchestrate our job run sequence
workflows:
  build_and_test:
    jobs:
      - build
      - test
```

Each config.yml file will be unique depending on the project, but normally we can find some common sections:

- The **orbs section** will be responsible for setting up some basic recipes
- The **jobs section** will contain specific actions to take
- The **workflows section** will specify how the jobs should be handled

New Terms

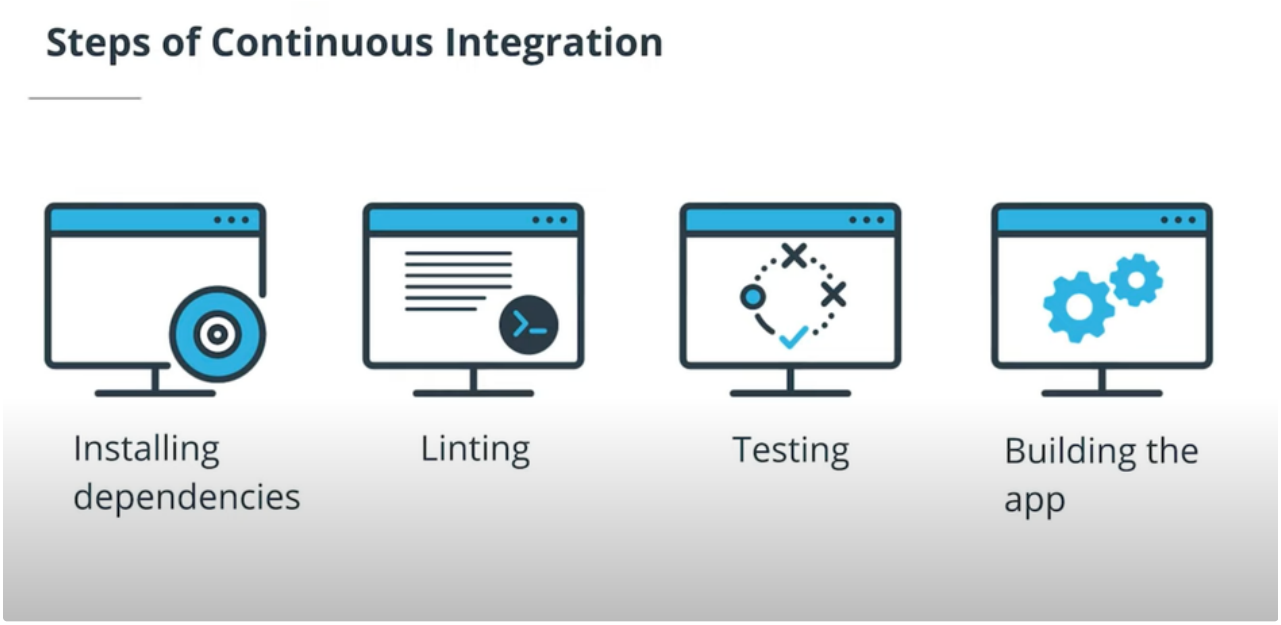
- **Pipeline:** A set of instructions that install, test, build and deploy applications.
- **Orbs:** Pre-made recipes offered by CircleCi to speed up setting up servers.
- **Jobs:** Commands that a CircleCI pipeline should run.
- **Workflows:** Information about the flow of jobs in a CircleCI Pipeline.

Further Reading

- CircleCi getting started: This is the intro tutorial from CircleCI.
- CircleCi Orbs: Search engine for all the orbs available on CircleCI.
- Understanding CI/CD in depth: this is a great post that explains CI/CD in-depth if you want to push the topics we will learn this lesson.

What Is Continuous Integration?

Now that we have built and for the first time our pipeline, let's understand the big steps that we take in a pipeline. You might often hear CI/CD. Let's focus first on the continuous integration, which stands for CI part.



The steps that we do in the continuous integration part would be installing dependencies. After this, we would lint our app to find if the code fits our standards, or how it looks, and how little commas were added at the red place. This is where, the step that will ensure that this is properly working. After this, we would have the testing step. Lastly, after everything has passed successfully before, we will build the application.

Installing Dependencies

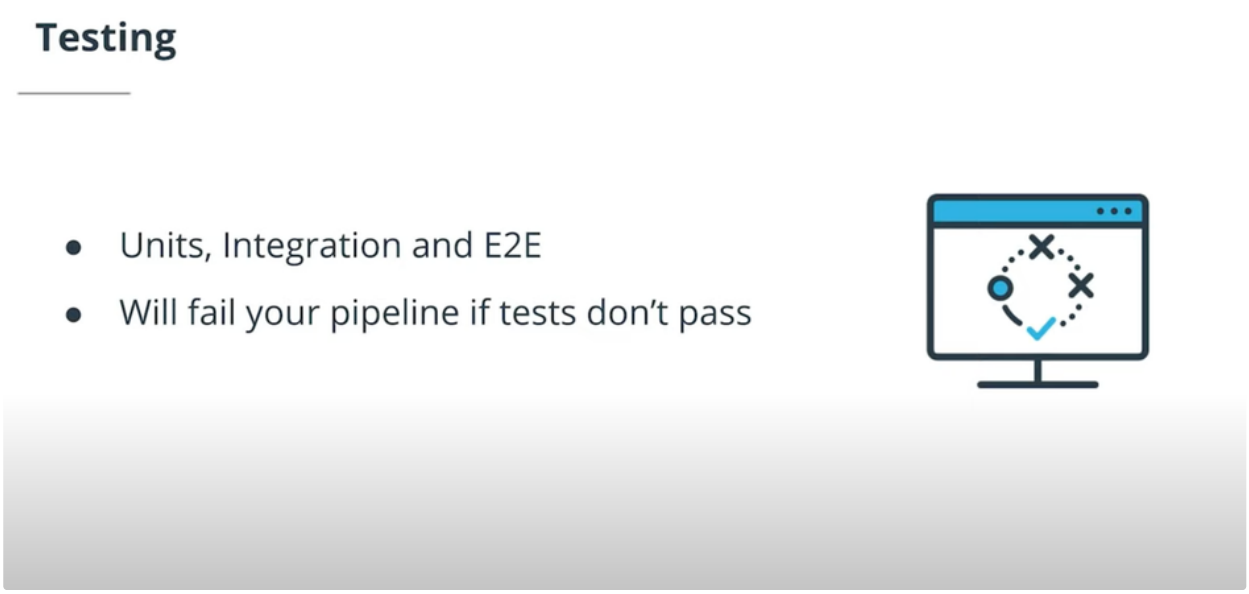
- Npm install
- Downloads node_modules on the pipeline server

Let's dive into each of those steps in more detail. The install dependencies step of continuous integration will go ahead and call Npm install. This will go and download the node modules directly to your pipeline server because we need them for running the next steps, and also for building the application.

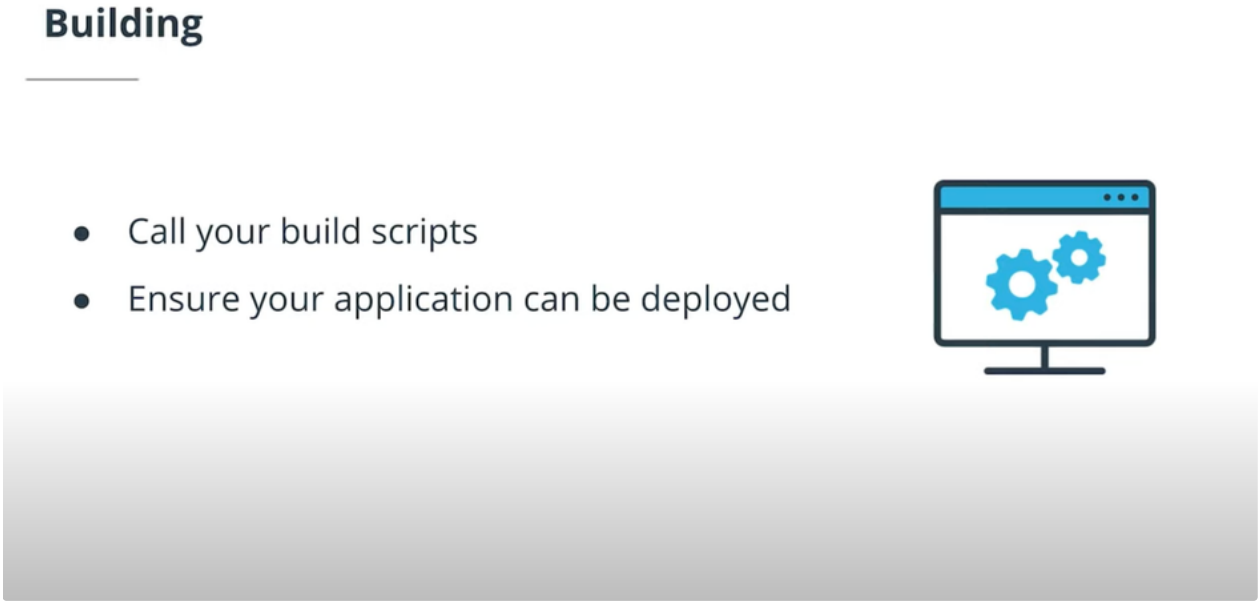
Linting and Code Formatting

- ESLint and Prettier
- Enforce consistency in the application

As for linting and code formatting, I'm thinking about tools like ESLint and Prettier. These tools enforce consistency and the application code, and really make sure you follow a team agreed-upon format for the code.



The next step would come via testing. We're calling here all the tests that we have in our application, units, integration, and end-to-end. If this test fail, then your pipeline will fell automatically because you don't want to be building an application that doesn't pass your test.



Right after you're done testing, you will start building the application. We will call here your build scripts. This will really ensure that your application is ready to be deployed. If you have installed lint and tested replication and you have a final bill that is passing, you're all ready to end the continuous integration step of your pipeline.

Key points:

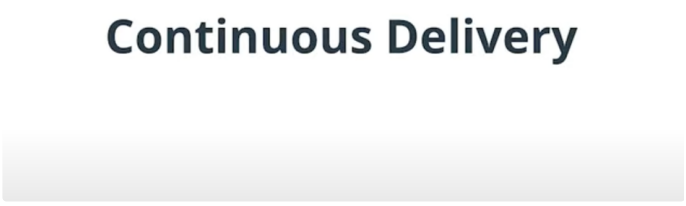
Continuous integration is a group of many steps in our pipeline. The goal of continuous integration is to verify if code is ready to be merged when a pull request is submitted or to see if code is ready and safe to be deployed. By installing dependencies and testing the code, we are building confidence that our application is ready to be deployed. To do so, we can include the following steps in our pipeline:

- **Linting** refers to verifying if the code follows certain standards of quality. This is the step responsible for calling lint scripts such as ESLint or Prettier.
- **Installing** is the step responsible for calling `npm install` to download node modules locally
- **Testing** is the step responsible for calling the different test scripts in our application
- **Building** is the step responsible for calling the build script of our application

> **Note:** Tools like Prettier have been gaining in popularity these last years. It should be noted, however, that they are different than linting. Prettier is a core formating tool that can only find stylistic errors in the code (ex: number of spaces after `if`), while linting can go further and also look at programatic errors (ex: a defined variable is never used).

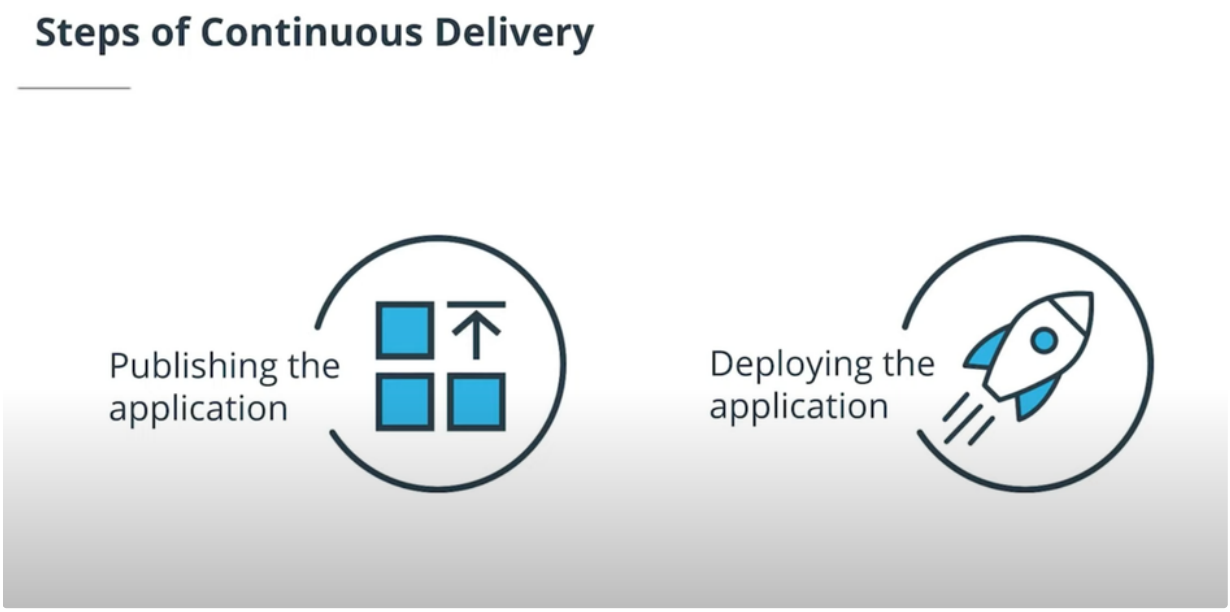
Further Reading

- What is Continuous Integration: Great post from Atlassian explaining the basics of Continuous Integration.
- Benefits of Continuous Integration: This is a great list of benefits that come with continuous integration.
- Prettier VS Lint: This is a great comparison to quickly understand the difference.

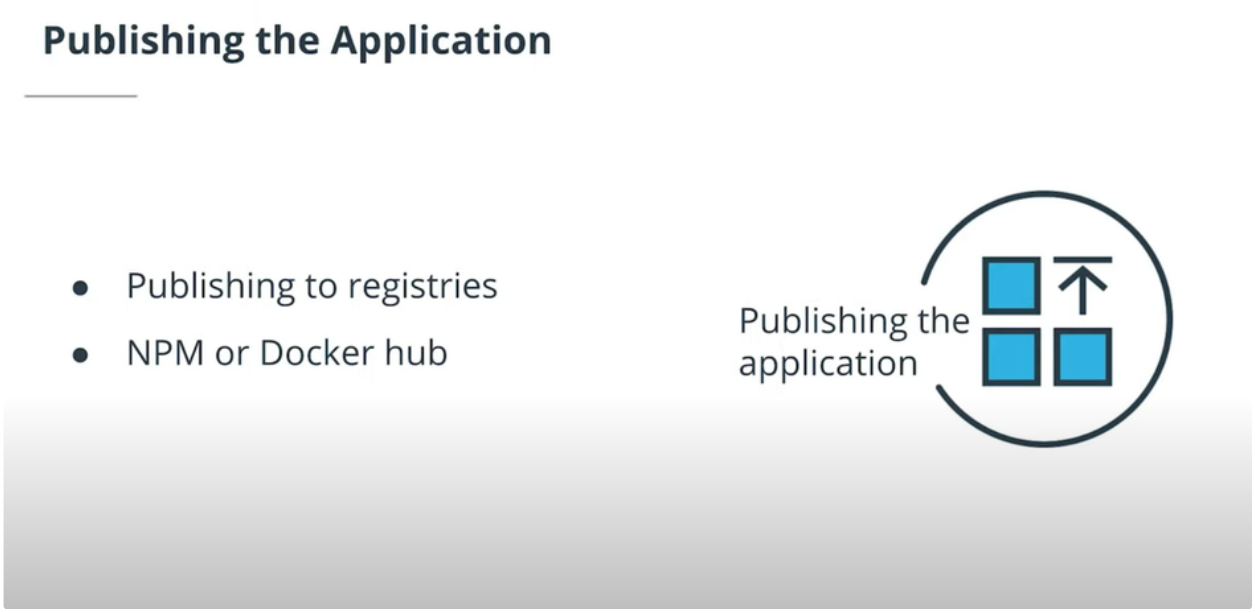


What Is Continuous Delivery and Continuous Deployment?

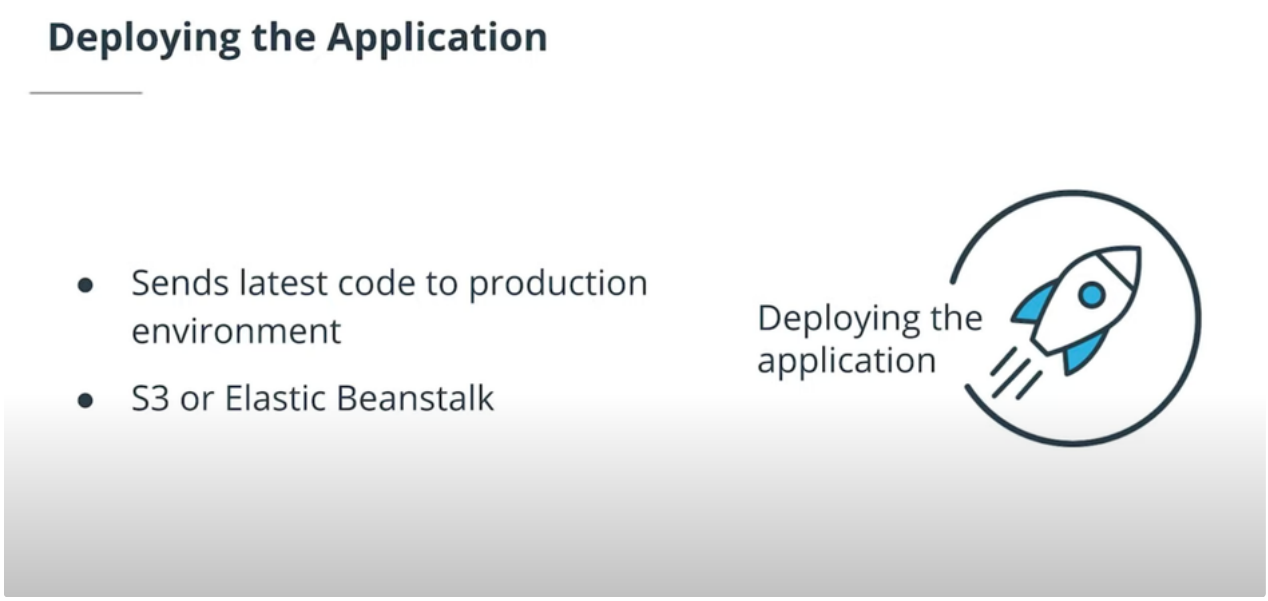
Let's jump into continuous delivery and understand a little bit about what is important at this step of the pipeline. What are the steps we're going to take?



First, we could be publishing the application or we could be deploying the application. Let's dive in and see a little bit of the difference between those two steps.



An application that we want to publish, would go somewhere to a registry. We're thinking about, let's say NPM or Docker Hub. This would be an application such as a JavaScript library that you might want to push or a Docker image.



In our case, however, we will be deploying our application. This is a little bit different. This means taking your code and sending it to your production environment. In our case, this will be AWS S3 and AWS Elastic Beanstalk.

Key points:

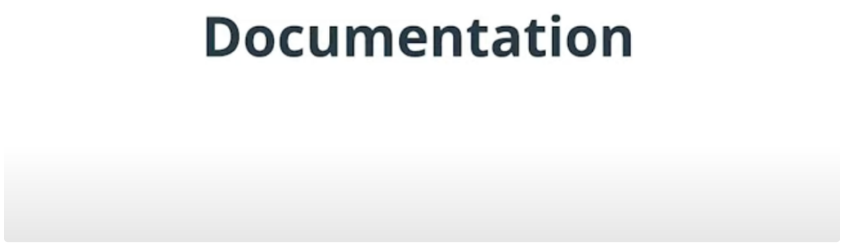
Continuous delivery or **continuous deployment** are both steps that we refer to when talking about the **D** of CI/CD. Both have as a goal to **get the application from the build stage** and **move it to its destination**. Let's explore a little bit the difference between each:

- **Continuous Delivery** aims at getting your application delivered to its final step before it is deployed. In this approach, code is manually approved for deployment.
- **Continuous Deployment** is similar to continuous delivery but goes one step further and makes the complete process automatic without human approval.

It is important to note also that some applications like NPM package or Docker images are meant to be published to a registry, while applications like servers or websites are meant to be deployed to services like S3 or Elastic Beanstalk.

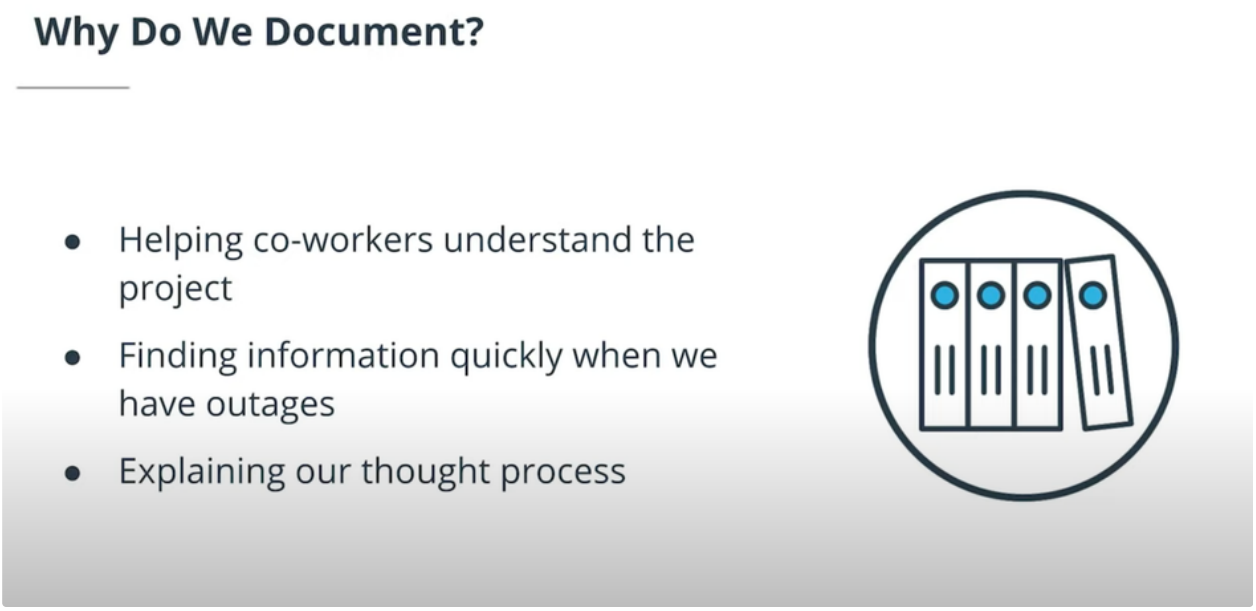
Further Reading

- Difference between Integration, delivery and deployment: This is a great post to explore the different terms that people often use in a pipeline.
- How to publish code to NPM: This explores the idea of publishing. While we don't cover it in this course, it is interesting to see how similar this process is to deploying an application.

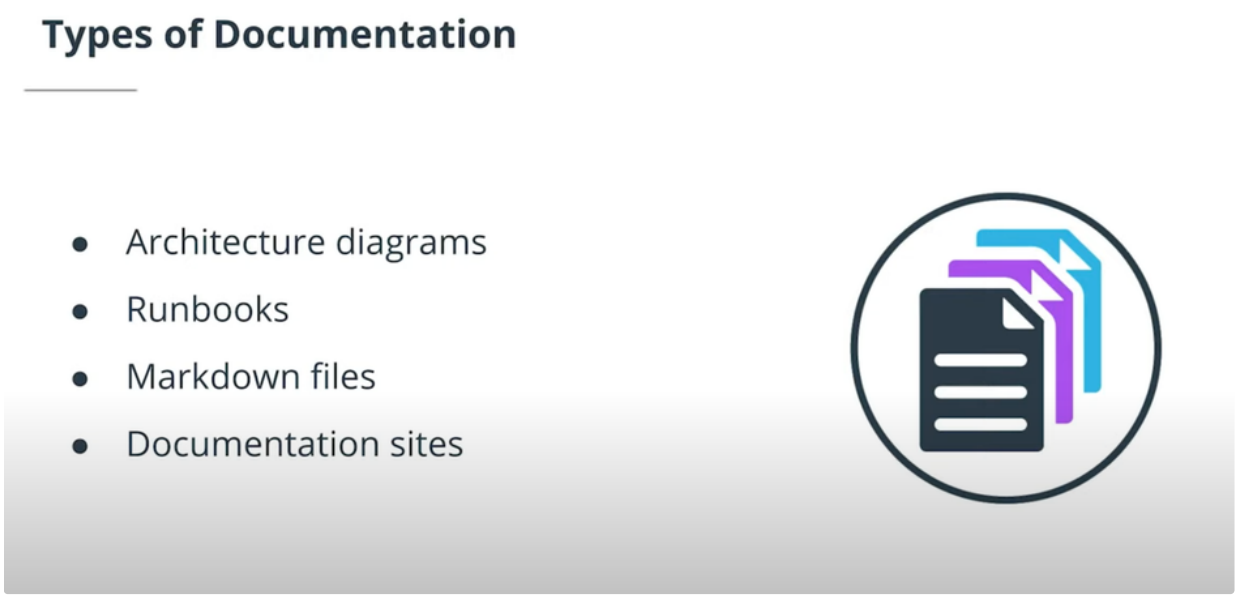


Let's have a look at how we can write good documentation about our application in general and also about our pipeline. Let's try to understand why we document.

Why do we document?



Documentation is really important it can help co-workers understand the project; not everybody has joined the project from the star,t so it's good to give it to your co-workers when they join your project; it's also really useful when you have outages or production bugs to find information quickly sometimes you want to do specific steps, but you don't remember quickly how to do. It so when you have documentation you can do this without having to remember and then explaining your thought process if you have complicated code, a complicated database documentation can help making this easier to explain.



Let's look at examples we could have architecture diagrams those would be done in tools like lucidcharts or draw.io, we could also create runbooks like we've done in lesson two explaining how to do certain actions on your application another type of documentation would be markdown files for example your README.md file at the root of your project is the best place to put onboarding instructions for new developers on the project and lastly you can do a full documentation site in html those are normally the best but they're also the most complicated to pull off and maintain.

Key Points

Documenting is important in order to properly communicate difficult parts of an application. Documentation also serves as a good reference when it comes to onboarding new developers on a project or diagnosing something that is going wrong.

There are multiple ways to document a project. Here are some forms of documentation that you might want to dive more into:

- **Architecture diagrams** are great at giving an overhead view of your application or production environment. These diagrams come in multiple forms but they all have a similar goal: give a visual reference to one part of your project.
- **Runbooks** are step-by-step guides that let you quickly remember how to accomplish certain operations on your website.
- **Markdown files** are great for giving a quick glimpse into the details of the project. The most common example is the README.md file that is present on most GitHub repo.
- **Documentation sites** provide a complete solution to host all documentation surrounding a project or company. Some are offered as a service like Confluence from Atlassian, while others are complete projects that live in a repo like Docusaurus.

Important sections of a README

In your README, you are trying to include general information about the project:

- How to set up the project
- A brief description of the project
- Any other useful information to communicate to new developers and give information at a quick glance

README Markdown Formatting

The first thing a developer will see when opening a new project is the README. You can use Markdown syntax to format the README. For example:

- Headings with `#`
- bullet lists with `-`

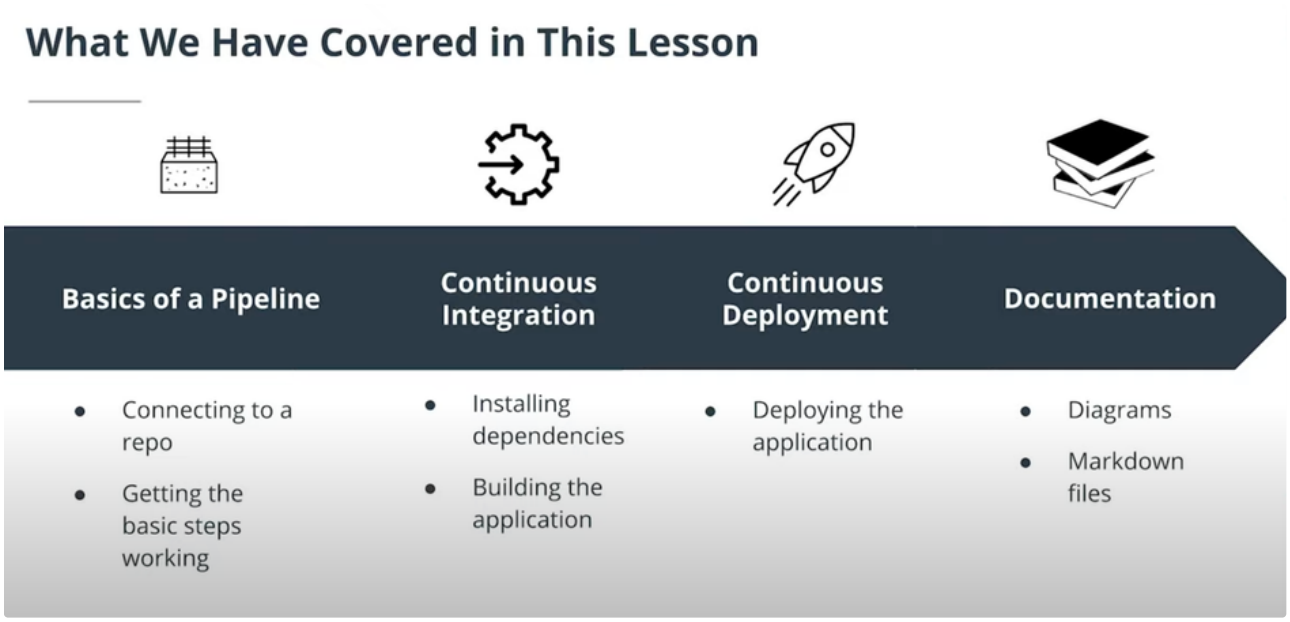
- numbered lists `1.`
- code blocks between triple backticks ```

The markdown format is great at quickly building documentation!

Further Readings

- Docusaurus is great documentation as a code framework built by Facebook.
- Confluence is documentation as a service product offered by Atlassian that is prevalent in most technology companies.
- Diagrams.net is a great tool to create architecture diagrams.
- Make a readme is a great site that explains how to make a good README. You can also find various examples of good readmes by looking at some big open source projects like Angular or Jest.

Lesson Recap_



You have learned a lot during this lesson! Creating pipelines and documenting your application are some of the most useful skills you can have as a developer. You will be adding value to any projects where you use those skills. Automating workflows can set you up for success in this fast-paced industry.

Glossary-

New Terms In This Lesson

- **Lint** refers to verifying if the code follows certain standards of quality. This is the step responsible for calling lint scripts such as ESLint or Prettier.
- **Installing** is the step responsible for calling `npm install` to download node modules locally.
- **Testing** is the step responsible for calling the different test scripts in our application.
- **Building** is the step responsible for calling the build script of our application.
- **Pipeline:** A set of instructions that install, test, build, and deploy applications.
- **Orbs:** Pre-made recipes offered by CircleCI to speed up setting up servers.
- **Jobs:** Commands that a CircleCI pipeline should run.
- **Workflows:** Information about the flow of jobs in a CircleCi Pipeline
- **Architecture diagrams** are great at giving an overhead view of your application or production environment. These diagrams come in multiple forms, but they all have a similar goal: give a visual reference to one part of your project.
- **Runbooks** are step-by-step guides that let you quickly remember how to accomplish certain operations on your website.
- **Markdown files** are great for giving quick documentation and passing along information. The most famous example is the README.md file that is present on most GitHub repo.
- **Documentation sites** provide a complete solution to host all documentation surrounding a project or company. Some are offered as a service like Confluence from Atlassian while some are complete projects that live in a repo like Docusaurus.

(Doc) Week 1: Python 01

Why Python Programming

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Introduction to Data Types and Operators

Data Types and Operators

Welcome to this lesson on Data Types and Operators! You'll learn about:

- Data Types: Integers, Floats, Booleans, Strings
- Operators: Arithmetic, Assignment, Comparison, Logical
- Built-In Functions, Type Conversion
- Whitespace and Style Guidelines

Assignment Operators

Operator	Example	Equivalent Expression (m=15)	Result
+=	m +=10	m = m+10	25
-=	m -=10	m = m-10	5
*=	m *=10	m = m*10	150
/=	m /=	m = m/10	1
%=	m %=10	m = m%10	5

Built-in Functions in Python

abs()	classmethod()	filter()	id()	max()	property()	str()
all()	compile()	float()	input()	memoryview()	range()	sum()
any()	complex()	format()	int()	min()	repr()	super()
ascii()	delattr()	frozenset()	isinstance()	next()	reversed()	tuple()
bin()	dict()	getattr()	issubclass()	object()	round()	type()
bool()	dir()	globals()	iter()	oct()	set()	vars()
bytearray()	divmod()	hasattr()	len()	open()	setattr()	zip()
bytes()	enumerate()	hash()	list()	ord()	slice()	__import__()
callable()	eval()	help()	locals()	pow()	sorted()	
chr()	exce()	hex()	map()	print()	staticmethod()	

Arithmetic Operators

Arithmetic Operators

Arithmetic operators

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `%` Mod (the remainder after dividing)
- `**` Exponentiation (note that `^` does not do this operation, as you might have seen in other languages)
- `//` Divides and rounds down to the nearest integer

The usual order of mathematical operations holds in Python, which you can review in this Math Forum page if needed.

Bitwise operators are special operators in Python that you can learn more about here if you'd like.

Examples

1. `print(3 + 5)` # 8
2. `print(1 + 2 + 3 * 3)` # 12
3. `print(3 ** 2)` # 9
4. `print(9 % 2)` # 1

Variables in Python

Variables I

Variables are used all the time in Python! Below is the example where we performed the following:

`mv_population = 74728`

Here `mv_population` is a variable, which holds the value of `74728`. This assigns the item on the right to the name on the left, which is actually a little different than mathematical equality, as `74728` does not hold the value of `mv_population`.

In any case, whatever term is on the left side, is now a name for whatever value is on the right side. Once a value has been assigned to a variable name, you can access the value from the variable name.

Variables II

The following two are equivalent in terms of assignment:

```
x = 3
y = 4
z = 5
```

and

```
x, y, z = 3, 4, 5
```

However, the above isn't a great way to assign variables in most cases, because our variable names should be descriptive of the values they hold.

Besides writing variable names that are descriptive, there are a few things to watch out for when naming variables in Python.

`1`. Only use ordinary letters, numbers and underscores in your variable names. They can't have spaces, and need to start with a letter or underscore.

`2`. You can't use **Python's reserved words, or "keywords,"** as variable names. There are reserved words in every programming language that have important purposes, and you'll learn about some of these throughout this course. Creating names that are descriptive of the values often will help you avoid using any of these keywords. Here you can see a table of Python's reserved words.

`3`. The pythonic way to name variables is to use all lowercase letters and underscores to separate words.

YES `my_height = 58 my_lat = 40 my_long = 105` **NO** `my height = 58 MYLONG = 40 MyLat = 105`

Though the last two of these would work in python, they are not pythonic ways to name variables. The way we name variables is called snake case, because we tend to connect the words with underscores.

Example

```
mv_population = 74728
mv_population = 74728 + 4000 - 600
print(mv_population) # 78128
```

Intergers and Float

There are two Python data types that could be used for numeric values:

- **int** - for integer values
- **float** - for decimal or floating point values

You can create a value that follows the data type by using the following syntax:

```
x = int(4.7)   # x is now an integer 4
y = float(4)   # y is now a float of 4.0
```

You can check the type by using the `type` function:

```
>>> print(type(x))
int
>>> print(type(y))
float
```

Because the float, or approximation, for 0.1 is actually slightly more than 0.1, when we add several of them together we can see the difference between the mathematically correct answer and the one that Python creates.

```
>>> print(.1 + .1 + .1 == .3)
False
```

You can see more on this here.

Python Best Practices

For all the best practices, see the PEP8 Guidelines.

You can use the atom package linter-python-pep8 to use pep8 within your own programming environment in the Atom text editor, but more on this later. If you aren't familiar with text editors yet, and you are performing all of your programming in the classroom, no need to worry about this right now.

Follow these guidelines to make other programmers and future you happy!

Good

```
print(4 + 5)
```

Bad

```
print(                    4 + 5)
```

You should limit each line of code to **80** characters, though **99** is okay for certain use cases. You can thank IBM for this ruling.

Why are these conventions important? Although how you format the code doesn't affect how it runs, following standard style guidelines makes code easier to read and consistent among different developers on a team.

Booleans, Comparison Operators, and Logical Operators

Examples

```
x = 42 > 43 # False
```

```
age = 14
is_teen = age > 12 and age < 20
print(is_teen) # True
```

Booleans, Comparison Operators, and Logical Operators

The bool data type holds one of the values `True` or `False`, which are often encoded as `1` or `0`, respectively.

There are 6 comparison operators that are common to see in order to obtain a `bool` value:

Comparison Operators

Symbol Use Case	Bool	Operation
5 < 3	False	Less Than
5 > 3	True	Greater Than
3 <= 3	True	Less Than or Equal To
3 >= 5	False	Greater Than or Equal To
3 == 5	False	Equal To
3 != 5	True	Not Equal To

And there are three logical operators you need to be familiar with:

Logical Use	Bool	Operation
5 < 3 `and` 5 == 5	False	`and` - Evaluates if all provided statements are True
5 < 3 `or` 5 == 5	True	`or` - Evaluates if at least one of many statements is True
`not` 5 < 3	True	`not` - Flips the Bool Value

Here is more information on how George Boole changed the world!

Strings

Strings

Strings in Python are shown as the variable type `str`. You can define a string with either double quotes `"` or single quotes `'`. If the string you are creating actually has one of these two values in it, then you need to be careful to assure your code doesn't give an error.

```
>>> my_string = 'this is a string!'
>>> my_string2 = "this is also a string!!!"
```

You can also include a `\` in your string to be able to include one of these quotes:

```
>>> this_string = 'Simon\'s skateboard is in the garage.'
>>> print(this_string)
```

```
Simon's skateboard is in the garage.
```

If we don't use this, notice we get the following error:

```
>>> this_string = 'Simon's skateboard is in the garage.'
```

```
  File "<ipython-input-20-e80562c2a290>", line 1
    this_string = 'Simon's skateboard is in the garage.'
                          ^
SyntaxError: invalid syntax
```

The color highlighting is also an indication of the error you have in your string in this second case. There are a number of other operations you can use with strings as well:

```
>>> first_word = 'Hello'
>>> second_word = 'There'
>>> print(first_word + second_word)

HelloThere

>>> print(first_word + ' ' + second_word)

Hello There

>>> print(first_word * 5)

HelloHelloHelloHelloHello

>>> print(len(first_word))

5
```

Unlike the other data types you have seen so far, you can also index into strings, but you will see more on this soon! For now, here is a small example. Notice Python uses 0 indexing - we will discuss this later in this lesson in detail.

```
>>> first_word[0]

H

>>> first_word[1]

e
```

The `len()` function

`len()` is a built-in Python function that returns the length of an object, like a string. The length of a string is the number of characters in the string. This will always be an integer.

There is an example above, but here's another one:

```
print(len("ababa") / len("ab"))
2.5
```

You know what the data types are for len("ababa") and len("ab"). Notice the data type of their resulting quotient here.

String Methods

In this part you were introduced to **methods**. **Methods** are like some of the **functions** you have already seen:

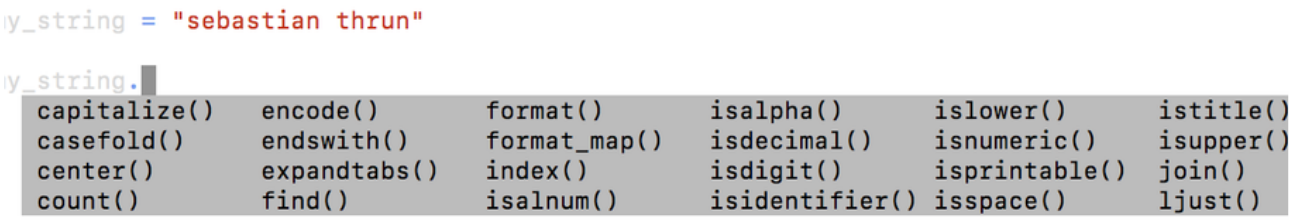
1. `len` ("this")
2. `type` (12)
3. `print` ("Hello world")

These three above are **functions** - notice they use parentheses, and accept one or more **arguments**. Functions will be studied in much more detail in a later lesson!

A **method** in Python behaves similarly to a function. Methods actually are functions that are called using dot notation. For example, `lower()` is a string method that can be used like this, on a string called "sample string": `sample_string.lower()`.

Methods are specific to the data type for a particular variable. So there are some built-in methods that are available for all strings, different methods that are available for all integers, etc.

Below is an image that shows some methods that are possible with any string.



Each of these methods accepts the string itself as the first argument of the method. However, they also could receive additional arguments, that are passed inside the parentheses. Let's look at the output for a few examples.

```
>>> my_string.islower()
True
>>> my_string.count('a')
2
>>> my_string.find('a')
3
```

You can see that the `count` and `find` methods both take another argument. However, the `.islower()` method does not accept another argument.

No professional has all the methods memorized, which is why understanding how to use documentation and find answers is so important. Gaining a strong grasp of the foundations of programming will allow you to use those foundations to use documentation to build so much more than someone who tries to memorize all the built-in methods in Python.

One important string method: `format()`

We will be using the `format()` string method a good bit in our future work in Python, and you will find it very valuable in your coding, especially with your `print` statements.

We can best illustrate how to use `format()` by looking at some examples:

Example 1 `python print("Mohammed has {} balloons".format(27))`

Example 1 Output `Mohammed has 27 balloons`

Example 2 `python animal = "dog" action = "bite" print("Does your {} {}?".format(animal, action))` **Example 2 Output** `txt Does your dog bite?`

Example 3 `python maria_string = "Maria loves {} and {}" print(maria_string.format("math", "statistics"))`

Example 3 Output `txt Maria loves math and statistics`

Notice how in each example, the number of pairs of curly braces {} you use inside the string is the same as the number of replacements you want to make using the values inside `format()`.

More advanced students can learn more about the formal syntax for using the `format()` string method here.

(Doc) Week 2: Python 02

Lists and Membership Operators

Lists!

Data structures are containers that organize and group data types together in different ways. A **list** is one of the most common and basic data structures in Python.

You saw here that you can create a list with square brackets. Lists can contain any mix and match of the data types you have seen so far.

```
list_of_random_things = [1, 3.4, 'a string', True]
```

This is a list of 4 elements. All ordered containers (like lists) are indexed in python using a starting index of 0. Therefore, to pull the first value from the above list, we can write:

```
>>> list_of_random_things[0]
1
```

It might seem like you can pull the last element with the following code, but this actually won't work:

```
>>> list_of_random_things[len(list_of_random_things)] #Index Error
```

However, you can retrieve the last element by reducing the index by 1. Therefore, you can do the following:

```
>>> list_of_random_things[len(list_of_random_things) - 1]
True
```

Alternatively, you can index from the end of a list by using negative values, where -1 is the last element, -2 is the second to last element and so on.

```
>>> list_of_random_things[-1]
True
>>> list_of_random_things[-2]
a string
```

> months = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']

> q3 = months[6:9]
> print(q3) # ['July', 'August', 'September']

> first_half = months[:6]
> print(first_half) # ['January', 'February', 'March', 'April', 'May', 'June']

> second_half = months[6:]
> print(second_half) # ['July', 'August', 'September', 'October', 'November', 'December']

> print(len(months)) # 12

> greeting = "Hello there"
> print(len(greeting)) # 11

Slice and Dice with Lists

You saw that we can pull more than one value from a list at a time by using **slicing**. When using slicing, it is important to remember that the `lower` index is `inclusive` and the `upper` index is `exclusive`.

Therefore, this:

```
>>> list_of_random_things = [1, 3.4, 'a string', True]
>>> list_of_random_things[1:2]
[3.4]
```

will only return **3.4** in a list. Notice this is still different than just indexing a single element, because you get a list back with this indexing. The colon tells us to go from the starting value on the left of the colon up to, but not including, the element on the right.

If you know that you want to start at the beginning, of the list you can also leave out this value.

```
>>> list_of_random_things[:2]
[1, 3.4]
```

or to return all of the elements to the end of the list, we can leave off a final element.

```
>>> list_of_random_things[1:]
[3.4, 'a string', True]
```

This type of indexing works exactly the same on strings, where the returned value will be a string.

Are you `in` OR `not in`?

You saw that we can also use `in` and `not in` to return a **bool** of whether an element exists within our list, or if one string is a substring of another.

```
>>> 'this' in 'this is a string'
True
>>> 'in' in 'this is a string'
True
>>> 'isa' in 'this is a string'
False
>>> 5 not in [1, 2, 3, 4, 6]
True
>>> 5 in [1, 2, 3, 4, 6]
False
```

Mutability and Order

Mutability is about whether or not we can change an object once it has been created. If an object (like a list or string) can be changed (like a list can), then it is called **mutable**. However, if an object cannot be changed with creating a completely new object (like strings), then the object is considered **immutable**.

```
>>> my_lst = [1, 2, 3, 4, 5]
>>> my_lst[0] = 'one'
>>> print(my_lst)
['one', 2, 3, 4, 5]
```

As shown above, you are able to replace 1 with 'one' in the above list. This is because lists are **mutable**.

However, the following does not work:

```
>>> greeting = "Hello there"
>>> greeting[0] = 'M'
```

This is because strings are **immutable**. This means to change this string, you will need to create a completely new string.

There are two things to keep in mind for each of the data types you are using:

1. Are they **mutable**?
2. Are they **ordered**?

Order is about whether the position of an element in the object can be used to access the element. **Both strings and lists are ordered.** We can use the order to access parts of a list and string.

However, you will see some data types in the next sections that will be unordered. For each of the upcoming data structures you see, it is useful to understand how you index, are they mutable, and are they ordered. Knowing this about the data structure is really useful!

Additionally, you will see how these each have different methods, so why you would use one data structure vs. another is largely dependent on these properties, and what you can easily do with it!

List Methods

Useful Functions for Lists I

1. `len()` returns how many elements are in a list.
2. `max()` returns the greatest element of the list. How the greatest element is determined depends on what type objects are in the list. The maximum element in a list of numbers is the largest number. The maximum elements in a list of strings is element that would occur last if the list were sorted alphabetically. This works because the the max function is defined in terms of the greater than comparison operator. The max function is undefined for lists that contain elements from different, incomparable types.
3. `min()` returns the smallest element in a list. min is the opposite of max, which returns the largest element in a list.
4. `sorted()` returns a copy of a list in order from smallest to largest, leaving the list unchanged.

Useful Functions for Lists II

`join` method

Join is a string method that takes a list of strings as an argument, and returns a string consisting of the list elements joined by a separator string.

```
new_str = "\n".join(["fore", "aft", "starboard", "port"])
print(new_str)
```

Output:

```
fore
aft
starboard
port
```

In this example we use the string `"\n"` as the separator so that there is a newline between each element. We can also use other strings as separators with .join. Here we use a hyphen.

```
name = "-".join(["García", "O'Kelly"])
print(name)
```

Output:

```
García-O'Kelly
```

It is important to remember to separate each of the items in the list you are joining with a comma (,). Forgetting to do so will not trigger an error, but will also give you unexpected results.

`append` method

A helpful method called `append` adds an element to the end of a list.

```
letters = ['a', 'b', 'c', 'd']
letters.append('z')
print(letters)
```

Output:

```
['a', 'b', 'c', 'd', 'z']
```

Try It Out!

In the beginning of the first video, you saw how the behaviour of variables containing mutable and immutable objects is very different and might even seem surprising at times! Experiment, use the print functions and double-check your work where you can, to make sure that your programs correctly keep track of their data. While you experiment with lists, try out some of the useful functions above.

Tuples

A tuple is another useful container. It's a data type for immutable ordered sequences of elements. They are often used to store related pieces of information. Consider this example involving latitude and longitude:

```
location = (13.4125, 103.866667)
print("Latitude:", location[0])
print("Longitude:", location[1])
```

Tuples are similar to lists in that they store an ordered collection of objects which can be accessed by their indices. Unlike lists, however, tuples are immutable - you can't add and remove items from tuples, or sort them in place.

Tuples can also be used to assign multiple variables in a compact way.

```
dimensions = 52, 40, 100
length, width, height = dimensions
print("The dimensions are {} x {} x {}".format(length, width, height))
```

The parentheses are optional when defining tuples, and programmers frequently omit them if parentheses don't clarify the code.

In the second line, three variables are assigned from the content of the tuple dimensions. This is called **tuple unpacking**. You can use tuple unpacking to assign the information from a tuple into multiple variables without having to access them one by one and make multiple assignment statements.

If we won't need to use `dimensions` directly, we could shorten those two lines of code into a single line that assigns three variables in one go!

```
length, width, height = 52, 40, 100
print("The dimensions are {} x {} x {}".format(length, width, height))
```

Sets

A **set** is a data type for mutable unordered collections of unique elements. One application of a set is to quickly remove duplicates from a list.

```
numbers = [1, 2, 6, 3, 1, 1, 6]
unique_nums = set(numbers)
print(unique_nums)
```

This would output:

```
{1, 2, 3, 6}
```

Sets support the `in` operator the same as lists do. You can add elements to sets using the `add` method, and remove elements using the `pop` method, similar to lists. Although, when you pop an element from a set, a random element is removed. Remember that sets, unlike lists, are unordered so there is no "last element".

```
fruit = {"apple", "banana", "orange", "grapefruit"}  # define a set

print("watermelon" in fruit)  # check for element

fruit.add("watermelon")  # add an element
print(fruit)

print(fruit.pop())  # remove a random element
print(fruit)
```

This outputs:

```
False
{'grapefruit', 'orange', 'watermelon', 'banana', 'apple'}
grapefruit
{'orange', 'watermelon', 'banana', 'apple'}
```

Other operations you can perform with sets include those of mathematical sets. Methods like union, intersection, and difference are easy to perform with sets, and are much faster than such operators with other containers.

Dictionaries and Identity Operators

Dictionaries

A **dictionary** is a mutable data type that stores mappings of unique keys to values. Here's a dictionary that stores elements and their atomic numbers.

```
elements = {"hydrogen": 1, "helium": 2, "carbon": 6}
```

Dictionaries can have keys of any immutable type, like integers or tuples, not just strings. It's not even necessary for every key to have the same type! We can look up values or insert new values in the dictionary using square brackets that enclose the key.

```
print(elements["helium"])  # print the value mapped to "helium"
elements["lithium"] = 3  # insert "lithium" with a value of 3 into the dictionary
```

We can check whether a value is in a dictionary the same way we check whether a value is in a list or set with the `in` keyword. Dicts have a related method that's also useful, `get`. get looks up values in a dictionary, but unlike square brackets, get returns None (or a default value of your choice) if the key isn't found.

```
print("carbon" in elements)
print(elements.get("dilithium"))
```

This would output:

```
True
None
```

Carbon is in the dictionary, so True is printed. Dilithium isn't in our dictionary so None is returned by `get` and then printed. If you expect lookups to sometimes fail, `get` might be a better tool than normal square bracket lookups because errors can crash your program.

Identity Operators

Keyword	Operator
`is`	evaluates if both sides have the same identity
`is not`	evaluates if both sides have different identities

You can check if a key returned None with the `is` operator. You can check for the opposite using `is not`.

```
n = elements.get("dilithium")
print(n is None)
print(n is not None)
```

This would output:

```
True
False
```

Compound Data Structures

We can include containers in other containers to create compound data structures. For example, this dictionary maps keys to values that are also dictionaries!

```
elements = {"hydrogen": {"number": 1,
                         "weight": 1.00794,
                         "symbol": "H"},
            "helium": {"number": 2,
                         "weight": 4.002602,
                         "symbol": "He"}}
```

We can access elements in this nested dictionary like this.

```
helium = elements["helium"]  # get the helium dictionary
hydrogen_weight = elements["hydrogen"]["weight"]  # get hydrogen's weight
```

You can also add a new key to the element dictionary.

```
oxygen = {"number":8,"weight":15.999,"symbol":"O"}  # create a new oxygen dictionary
elements["oxygen"] = oxygen  # assign 'oxygen' as a key to the elements dictionary
print('elements = ', elements)
```

Output is:

```
elements =  {"hydrogen": {"number": 1,
                          "weight": 1.00794,
                          "symbol": 'H'},
             "helium": {"number": 2,
                          "weight": 4.002602,
                          "symbol": "He"},
             "oxygen": {"number": 8,
                          "weight": 15.999,
                          "symbol": "O"}}
```

Conditional Statements

If Statement

An `if` statement is a conditional statement that runs or skips code based on whether a condition is true or false. Here's a simple example.

```
if phone_balance < 5:
    phone_balance += 10
    bank_balance -= 10
```

Let's break this down.

1. An `if` statement starts with the `if` keyword, followed by the condition to be checked, in this case `phone_balance < 5`, and then a colon. The condition is specified in a boolean expression that evaluates to either True or False.

2. After this line is an indented block of code to be executed if that condition is true. Here, the lines that increment `phone_balance` and decrement `bank_balance` only execute if it is true that `phone_balance` is less than 5. If not, the code in this `if` block is simply skipped.

Use Comparison Operators in Conditional Statements

You have learned about Python's comparison operators (e.g. `==` and `!=`) and how they are different from assignment operators (e.g. `=`). In conditional statements, you want to use comparison operators. For example, you'd want to use `if x == 5` rather than `if x = 5`. If your conditional statement is causing a syntax error or doing something unexpected, check whether you have written `==` or `=`!

If, Elif, Else

In addition to the `if` clause, there are two other optional clauses often used with an `if` statement. For example:

```
if season == 'spring':
    print('plant the garden!')
elif season == 'summer':
    print('water the garden!')
elif season == 'fall':
    print('harvest the garden!')
elif season == 'winter':
    print('stay indoors!')
else:
    print('unrecognized season')
```

1. `if`: An `if` statement must always start with an `if` clause, which contains the first condition that is checked. If this evaluates to True, Python runs the code indented in this `if` block and then skips to the rest of the code after the `if` statement.

2. `elif`: `elif` is short for "else if." An `elif` clause is used to check for an additional condition if the conditions in the previous clauses in the `if` statement evaluate to False. As you can see in the example, you can have multiple `elif` blocks to handle different situations.

3. `else`: Last is the `else` clause, which must come at the end of an `if` statement if used. This clause doesn't require a condition. The code in an `else` block is run if all conditions above that in the `if` statement evaluate to False.

Boolean Expressions for Conditions

Complex Boolean Expressions

`If` statements sometimes use more complicated boolean expressions for their conditions. They may contain multiple comparisons operators, logical operators, and even calculations. Examples:

```
if 18.5 <= weight / height**2 < 25:
    print("BMI is considered 'normal'")

if is_raining and is_sunny:
    print("Is there a rainbow?")
```

For really complicated conditions you might need to combine some `and`s, `or`s and `not`s together. Use parentheses if you need to make the combinations clear.

However simple or complex, the condition in an `if` statement must be a boolean expression that evaluates to either True or False and it is this value that decides whether the indented block in an `if` statement executes or not.

For Loops

Python has two kinds of loops - `for` loops and `while` loops. A `for` loop is used to "iterate", or do something repeatedly, over an **iterable**.

An **iterable** is an object that can return one of its elements at a time. This can include sequence types, such as strings, lists, and tuples, as well as non-sequence types, such as dictionaries and files.

Example

Let's break down the components of a `for` loop, using this example with the list `cities`:

```
cities = ['new york city', 'mountain view', 'chicago', 'los angeles']
for city in cities:
    print(city)
print("Done!")
```

Components of a `for` Loop

1. The first line of the loop starts with the `for` keyword, which signals that this is a `for` loop
2. Following that is `city in cities`, indicating `city` is the iteration variable, and `cities` is the iterable being looped over. In the first iteration of the loop, `city` gets the value of the first element in `cities`, which is “new york city”.
3. The `for` loop heading line always ends with a colon `:`
4. Following the `for` loop heading is an indented block of code, the body of the loop, to be executed in each iteration of this loop. There is only one line in the body of this loop - `print(city)`.
5. After the body of the loop has executed, we don't move on to the next line yet; we go back to the `for` heading line, where the iteration variable takes the value of the next element of the iterable. In the second iteration of the loop above, `city` takes the value of the next element in `cities`, which is "mountain view".
6. This process repeats until the loop has iterated through all the elements of the iterable. Then, we move on to the line that follows the body of the loop - in this case, `print("Done!")`. We can tell what the next line after the body of the loop is because it is unindented. Here is another reason why paying attention to your indentation is very important in Python!

Executing the code in the example above produces this output:

```
new york city
mountain view
chicago
los angeles
Done!
```

You can name iteration variables however you like. A common pattern is to give the iteration variable and iterable the same names, except the singular and plural versions respectively (e.g., 'city' and 'cities').

Using the `range()` Function with `for` Loops

`range()` is a built-in function used to create an iterable sequence of numbers. You will frequently use `range()` with a `for` loop to repeat an action a certain number of times. Any variable can be used to iterate through the numbers, but Python programmers conventionally use `i`, as in this example:

```
for i in range(3):
    print("Hello!")
```

Output:

```
Hello!
Hello!
```

```
Hello!
```

`range(start=0, stop, step=1)`

The `range()` function takes three integer arguments, the first and third of which are optional:

- The 'start' argument is the first number of the sequence. If unspecified, 'start' defaults to 0.
- The 'stop' argument is 1 more than the last number of the sequence. This argument must be specified.
- The 'step' argument is the difference between each number in the sequence. If unspecified, 'step' defaults to 1.

Notes on using `range()`:

- If you specify one integer inside the parentheses with `range()`, it's used as the value for 'stop,' and the defaults are used for the other two.
  e.g. - `range(4)` returns `0, 1, 2, 3`
- If you specify two integers inside the parentheses with `range()`, they're used for 'start' and 'stop,' and the default is used for 'step.'
  e.g. - `range(2, 6)` returns `2, 3, 4, 5`
- Or you can specify all three integers for 'start', 'stop', and 'step.'
  e.g. - `range(1, 10, 2)` returns `1, 3, 5, 7, 9`

Creating and Modifying Lists

In addition to extracting information from lists, as we did in the first example above, you can also create and modify lists with `for` loops. You can **create** a list by appending to a new list at each iteration of the `for` loop like this:

```
# Creating a new list
cities = ['new york city', 'mountain view', 'chicago', 'los angeles']
capitalized_cities = []

for city in cities:
    capitalized_cities.append(city.title())
```

Modifying a list is a bit more involved, and requires the use of the `range()` function.

We can use the `range()` function to generate the indices for each value in the `cities` list. This lets us access the elements of the list with `cities[index]` so that we can modify the values in the `cities` list in place.

```
cities = ['new york city', 'mountain view', 'chicago', 'los angeles']

for index in range(len(cities)):
    cities[index] = cities[index].title()
```

Building Dictionaries

By now you are familiar with two important concepts: 1) counting with `for` loops and 2) the dictionary `get` method. These two can actually be combined to create a useful counter dictionary, something you will likely come across again. For example, we can create a dictionary, `word_counter`, that keeps track of the total count of each word in a string.

Method 1: Using a `for` loop to create a set of counters

```
book_title = ['great', 'expectations','the', 'adventures', 'of',
'sherlock','holmes','the','great','gasby','hamlet','adventures','of','huckleberry','fin']

word_counter = {}

for word in book_title:
    if word not in word_counter:
        word_counter[word] = 1
    else: word_counter[word] += 1
```

Output: `{'great': 2, 'expectations': 1, 'the': 2, 'adventures': 2, 'of': 2, 'sherlock': 1, 'holmes': 1, 'gasby': 1, 'hamlet': 1, 'huckleberry': 1, 'fin': 1}`

Method 2: Using the `get` method

```
book_title = ['great', 'expectations','the', 'adventures', 'of',
'sherlock','holmes','the','great','gasby','hamlet','adventures','of','huckleberry','fin']

word_counter = {}

for word in book_title:
    word_counter[word] = word_counter.get(word, 0) + 1
```

Output: `{'great': 2, 'expectations': 1, 'the': 2, 'adventures': 2, 'of': 2, 'sherlock': 1, 'holmes': 1, 'gasby': 1, 'hamlet': 1, 'huckleberry': 1, 'fin': 1}`

Iterating Through Dictionaries with For Loops

Iterating Through Dictionaries with `For` Loops

When you iterate through a dictionary using a `for` loop, doing it the normal way (`for n in some_dict`) will only give you access to the **keys** in the dictionary - which is what you'd want in some situations. In other cases, you'd want to iterate through both the **keys** and **values** in the dictionary. Let's see how this is done in an example. Consider this dictionary that uses names of actors as keys and their characters as values.

```
cast = {
           "Jerry Seinfeld": "Jerry Seinfeld",
           "Julia Louis-Dreyfus": "Elaine Benes",
           "Jason Alexander": "George Costanza",
           "Michael Richards": "Cosmo Kramer"
       }
```

Iterating through it in the usual way with a `for` loop would give you just the keys, as shown below:

```
for key in cast:
    print(key)
```

This outputs:

```
Jerry Seinfeld
Julia Louis-Dreyfus
Jason Alexander
Michael Richards
```

If you wish to iterate through both keys and values, you can use the built-in method `items` like this:

```
for key, value in cast.items():
    print("Actor: {}    Role: {}".format(key, value))
```

This outputs:

```
Actor: Jerry Seinfeld    Role: Jerry Seinfeld
Actor: Julia Louis-Dreyfus    Role: Elaine Benes
Actor: Jason Alexander    Role: George Costanza
Actor: Michael Richards    Role: Cosmo Kramer
```

`items` is an awesome method that returns tuples of key, value pairs, which you can use to iterate over dictionaries in `for` loops.

While Loops

`While` Loops

`For` loops are an example of "definite iteration" meaning that the loop's body is run a predefined number of times. This differs from "indefinite iteration" which is when a loop repeats an unknown number of times and ends when some condition is met, which is what happens in a `while` loop. Here's an example of a `while` loop.

```
card_deck = [4, 11, 8, 5, 13, 2, 8, 10]
hand = []

# adds the last element of the card_deck list to the hand list
# until the values in hand add up to 17 or more
while sum(hand)  < 17:
    hand.append(card_deck.pop())
```

This example features two new functions. `sum` returns the sum of the elements in a list, and `pop` is a list method that removes the last element from a list and returns it.

Components of a `While` Loop

1. The first line starts with the `while` keyword, indicating this is a `while` loop.
2. Following that is a condition to be checked. In this example, that's `sum(hand) <= 17`.
3. The `while` loop heading always ends with a colon `:`.
4. Indented after this heading is the body of the `while` loop. If the condition for the `while` loop is true, the code lines in the loop's body will be executed.
5. We then go back to the `while` heading line, and the condition is evaluated again. This process of checking the condition and then executing the loop repeats until the condition becomes false.
6. When the condition becomes false, we move on to the line following the body of the loop, which will be unindented.

The indented body of the loop should modify at least one variable in the test condition. If the value of the test condition never changes, the result is an infinite loop!

Break, Continue

Break, Continue

Sometimes we need more control over when a loop should end, or skip an iteration. In these cases, we use the `break` and `continue` keywords, which can be used in both `for` and `while` loops.

- `break` terminates a loop
- `continue` skips one iteration of a loop

Zip and Enumerate

Zip

`zip` returns an iterator that combines multiple iterables into one sequence of tuples. Each tuple contains the elements in that position from all the iterables. For example, printing

`list(zip(['a', 'b', 'c'], [1, 2, 3]))` would output `[('a', 1), ('b', 2), ('c', 3)]`.

Like we did for `range()` we need to convert it to a list or iterate through it with a loop to see the elements.

You could unpack each tuple in a `for` loop like this.

```
letters = ['a', 'b', 'c']
nums = [1, 2, 3]

for letter, num in zip(letters, nums):
    print("{}: {}".format(letter, num))
```

In addition to zipping two lists together, you can also unzip a list into tuples using an asterisk.

```
some_list = [('a', 1), ('b', 2), ('c', 3)]
letters, nums = zip(*some_list)
```

This would create the same `letters` and `nums` tuples we saw earlier.

Enumerate

`enumerate` is a built in function that returns an iterator of tuples containing indices and values of a list. You'll often use this when you want the index along with each element of an iterable in a loop.

```
letters = ['a', 'b', 'c', 'd', 'e']
for i, letter in enumerate(letters):
    print(i, letter)
```

This code would output:

```
0 a
1 b
2 c
3 d
4 e
```

List Comprehensions

In Python, you can create lists really quickly and concisely with list comprehensions. This example from earlier:

```
capitalized_cities = []
for city in cities:
    capitalized_cities.append(city.title())
```

can be reduced to:

```
capitalized_cities = [city.title() for city in cities]
```

List comprehensions allow us to create a list using a `for` loop in one step.

You create a list comprehension with brackets `[]`, including an expression to evaluate for each element in an iterable. This list comprehension above calls `city.title()` for each element `city` in `cities`, to create each element in the new list, `capitalized_cities`.

Conditionals in List Comprehensions

You can also add conditionals to list comprehensions (listcomps). After the iterable, you can use the `if` keyword to check a condition in each iteration.

```
squares = [x**2 for x in range(9) if x % 2 == 0]
```

The code above sets `squares` equal to the list [0, 4, 16, 36, 64], as x to the power of 2 is only evaluated if x is even. If you want to add an `else`, you will get a syntax error doing this.

```
squares = [x**2 for x in range(9) if x % 2 == 0 else x + 3]
```

If you would like to add `else`, you have to move the conditionals to the beginning of the listcomp, right after the expression, like this.

```
squares = [x**2 if x % 2 == 0 else x + 3 for x in range(9)]
```

List comprehensions are not found in other languages, but are very common in Python.

(Doc) Week-03: JS 01

What is JavaScript?

History of JavaScript

Developer Tools on Different Browsers

console.log

JavaScript Demo

Intro to Data Types

Numbers

Comments

String Concatenation

Variables

String Index

Escaping Strings

Comparing Strings

Booleans

Null, Undefined, and NaN

Equality

Conditionals

Intro to Conditionals

Flowchart to Code

If...Else Statements

Else If Statements

More Complex Problems

Logical Operators

Logical AND and OR

Advanced Conditionals

Truthy and Falsy

Ternary Operator

Switch Statement

Falling-through

(Doc) Week-04: JS 02

Loops

Intro to Loops

While - Loops

Parts of a While Loop

For - Loops

Parts of a For Loop

Nested Loops

Increment and Decrement

Functions - JS

Function Example

Function Recap

Return Values

Using Return Values

Scope

Scope Example

Shadowing

Global Variables

Scope Recap

Hoisting

Hoisting Recap

Function Expressions

Patterns with Function Expressions

Function Expression Recap

(Doc) Week-05: JS 03

Arrays

Intro to Arrays

Creating an Array

Accessing Array Elements

Array Index

Array Properties and Methods

Length

Push

Pop

Splice

Array Loops

The forEach Loop

Map

Arrays in Arrays

2D Donut Arrays

Objects

Intro to Objects

Objects in Code

Objects - JS

Object Literals

Naming Conventions

Summary of Objects