Working With React Components

Video Transcript

Video 1 – Introduction To Styling With Bootstrap

As it turns out, styling is challenging enough that it's a career onto itself. And here at the outset, you might be wondering how can I add some styles without having to be an expert? Well, there are a number of frameworks that address precisely that need. Now, that doesn't mean you don't have to learn a little bit to use them, but the investment that you will make is a minimal one. And so, as we get going, we'll show you here how to use bootstrap. By far one of the most widely used frameworks that exist. But note that it is not the only one, there are many others. And in fact, you could write your own. It would be challenging, but tremendously rewarding to be able to take full control. But that's beyond the scope of what we're trying to do here. We're simply learning how to use Bootstrap in the context of front-end work.

Video 2 – Styling Components With Bootstrap

We will often find the need to style our components, and easy way to do that is to use one of these style frameworks. A very popular one, the most used by far is Bootstrap. So, let's go ahead and navigate to Bootstrap. And as you can see here, this is the landing page. And if we go to the documentation, you can see there that there are four broad categories, Layout, Content, Components, and Utilities. And we will use a component here to illustrate how to use Bootstrap. We will first create a standalone HTML page, and then we will create a one within React. So, let's go ahead and navigate to home simply to get a hold of the style sheets. And as you can see here, we could pull in the Barebones style sheet or we could do a lot more.

The difference being if you just want simply the look and feel of CSS, or if you want to use components that have JavaScript builtin into them. Now, let's go ahead and create a page. We will call it index. I'm going to go ahead, actually, let's call it 'bootstrap.html' and there is our page. Let's go ahead and put it in some of the boilerplate for HTML. Now, we're going to go ahead and add a style sheet. So, I will go ahead and enter here 'link'. And as you can see there, I get the placeholder. I'm now going to come here and copy the style sheet. Then I go ahead and add it to the page. And I believe it gave me the entire tag. So, I'm going to go ahead and view the entire line. And as you can see that, all of that is that, that link plus the tag that I got directly from Bootstrap. Now, at this point, we're ready to start adding content.

So, let's go ahead and go to the documentation. Then, I'm going to use one of the components. I want to use a card. I'm going to go ahead to this section, and I'm going to navigate and scroll down to page, to one I like. I'm going to go ahead and use this one that has a header and a body

here. And I'm going to go ahead and paste that onto my '< body >' of HTML. I'm then going to go ahead and drag and drop that file onto the browser. This is the 'bootstrap.html'. So, here is the file. And as you can see there we get our card. Now, we could use a container. So, we can enter a '< div >' here with the 'class="container" '. And we're going to wrap our '< / div >'.

This will simply give us a little padding around the container. And I'm going to go ahead and indent this just so that it's easier to see. And as you can see there, we have a little bit of padding around our div, around our card. Now, let's go ahead and do the same for React. And we'll start off with the simplest way. And react will have a standalone page, meaning not a full-blown app. Let's go ahead and go to the React documentation. I will go to React, add React to a website. And we're going to scroll down here to view the source code. And this is the way to create a standalone page, meaning that we're bringing in all of the plumbing of React by using script tags, much like the way we just use a tag to bring in styles. So, I'm going to go ahead and copy this. I will create a new file. I'm going to go ahead and save it first.

I will save it as 'react.html'. And then I'm going to go ahead and paste the content that I just got from the React documentation. Now, the very first thing that I'm going to do is I'm going to add the Bootstrap styles and go ahead copy from the first example that we did. I'm going to simply paste here. I will write myself a comment simply to be explicit. This will be 'bootstrap css'. Then below it, I'm going to add one more thing. I'm going to add 'babel' because I'm going to go ahead and create a component in a moment. And I will need 'babel'. And as you can see there, this is not the way to do it in production. This is simply to illustrate a very simple page that doesn't have all of this scaffolding that the projects do. So, I'm going to go ahead and add that line there that simply brings in 'babel'.

And then I'm going to go ahead and make a couple of changes. The very first one is I'm going to change this to ' "card.js" ' because I will call the file that we will create in a moment ' "card.js" '. And I will target this '< div >' here. But I want to call it ' "root" '. So, before we go any further, let's go ahead and review. What I did is I copied some HTML that I got from one of the React examples. Inside of this page, I have the Bootstrap styles, I have Babel. Then down here I have the libraries for React. In addition to that, I have one '< div >' that I'm going to go ahead and target, as you can see there, it's called ' "root" '. And then, now I'm going to go ahead and write this file that's called ' "card.js" '. So, go ahead and create a new file. Then go ahead and save. And I will call it 'card.js'. I'm first going to go ahead and add the card.

As you can see, I have very much the same syntax as I did before. I'm then going to go ahead and create an element from that 'card'. And I will then add this to the page by using the render method as you can see there, and 'getElementById' to target the ' 'root' ' '< div >' that we put in the HTML page. This one, here. Let's see where is it? It's over here. So, let's go ahead and drag and drop the file onto the browser. And I'll go ahead and open up my developer tools to see why I did not get the card. And as you can see there, I get some cryptic message. But it turns out that the reason why this error occurs is because you need to indicate to the browser what type of script

you're going to be evaluating. And in this case, we need to tell it that it's going to have 'babel', or syntax that 'babel' needs to interpret. And we also want to 'defer' until the document has loaded.

So, if we go ahead and reload the page, we now get a different error. And this has to do with serving the file directly from our file system. And it indicates that we now need to load it from an HTTP server. So, I'm going to go ahead and fire up my HTTP server now. And now, that I've gone ahead and done that locally, my address is 'localhost:8080/'react.html', which is the name of the file. I'm not going to go ahead and hit return. And as you can see there, now we get our nice card within a React page. Now, let's go ahead and create our React app with all of the tooling. So, for this, we'll go ahead and enter. Well, first let me move to my 'Desktop' here. And then I'm going to go ahead and 'create-react-app'. And I'm going to call it 'big', just simply to note that this one's going to have everything.

So, I'm going to go ahead and run that. Now, that that has installed, let's go ahead and clean up our work area. And then we'll go ahead and drag and drop our new application here. And as you can see there, we have the typical structure of a React app. And now, we need to go ahead and add Bootstrap to this project. And so, we're going to do that through the command line now. I'm going to go ahead and close that. And I'm going to go ahead and open up my command line. We're then going to go ahead and use 'npm' to 'install bootstrap'. And as you can see there, that's what I'm doing. Now, by doing that, we have the files. Now, let's go ahead and open up 'index.js', and we will add in 'index.js', the CSS file. And we do that by entering 'import'. And then we're going to enter the path to our file.

So, we'll start off with 'bootstrap', then distribution '/dist'. And if you dig in through the node modules, you will be able to see this. And here's '/css', that is the directory that holds this. And then inside of it, it's 'bootstrap.css'. Now, by doing that, CSS styles are now available inside of this application. And I'm going to go ahead and navigate to the 'App.js'. And I'm going to go ahead and change here. This is your default landing page when you create a project. And so, when you create a React application. And I'm going to go ahead and replace all of this here with the syntax for my tag. So, here is the HTML or the ' "card" '. And note that I'm using a ' "container" ' as well. Now, there's one more change we need to make, and that is that we need to change 'class' to 'className'. Now, let's go back to the command line and move into our application.

And then we can 'start'. And as you can see there, we get as expected our little card. So, as you can see, adding Bootstrap styles is pretty easy. I showed it originally in the HTML file standalone. It's just one line. Then we just use the syntax for the components on Bootstrap. Then we added in a standalone page on React. Then we did with the full React application. So, go ahead and try this. There are some twists. As you can see there, you know, for example, you got to use classNames once you move into React because it is a reserved keyword. And as you move on to the full application with all the tooling, you got to do a few more steps. But ultimately, once you get through those, then you're working once again with Bootstrap, which easily gives you access to all of Bootstrap's tooling.

Video 3 – Introduction To Components - HTML To React

Creating blocks of functionality, whether we call them a function, a class, a library, a module. We have many names for them, is something that has existed since the first days of computing. And when it comes to the browser, we've had a similar evolution. Over time, we've needed that grouping of functionality, that ability to bring in bigger blocks without having to see all of the details and being overwhelmed in the same way. So, in this case, we're going to take a look at an example, a simple example, creating a list of books. And figuring out how we can lay this out using components, we will look at the progression of the history of the browsers, starting from pure HTML all the way to the latest front-end frameworks.

And within this history, we will look at how do we do this with pure HTML. In the example that we will look at will be one of the styling frameworks where you can take a block of code that they have defined and style in a very elegant way, your component. Now, that certainly works, but it can be a tremendous amount of work to do it that way. So, the next step that we will take will be to do the same but do it using JavaScript, which certainly works. However, the browser has provided some of this capability and can give you custom tags. That is logic that is encapsulated within these tags that you define using JavaScript to do the exact same thing.

And we'll see how we can do that in conjunction with modules and do this same thing, but do it programmatically if we have a list of content and minimize the amount of work that we have to do and yet achieve the same goals of encapsulation and being able to separate our concerns. Now, as a last step, we'll take a look in how to do the exact same things that we have done, but do it using React. And then finally, how to mix both of them. That is native web components that are understood by any browser without any frameworks. And then add the capability of being able to use them within the front-end framework that we're using, which is React. So, this would be a good one to see that spectrum and get some firm footing on how to use components across the different types of ways in which you might potentially have to do so.

Video 4 – Web Components With Bootstrap

Say you wanted to create a webpage to list your books. And furthermore, say you wanted to leverage some of the open-source projects that provide you with free components. These are components that have no licensing restrictions. And so, in this case, we're going to do precisely that and we're going to navigate to their starter template. We're going to scroll down the page and you can see it here. And we're going to go ahead and copy this HTML onto our page, which is books.html as you can see. And then we're going to go ahead and drag and drop that file onto

our browser. And as you can see there, we get our Hello, world! with some minimal styling. We only have one '< h1 >'. And I'm going to go ahead and remove here the JavaScript since we will only be using the styles in this case.

Now, as mentioned, we want to go ahead and use some of the components so we can scroll on the page here to the Component section. We can select the Cards. And I'm going to go ahead and scroll down till I find a card that fits the styling that I want to use. And this one here looks pretty good like we could use it to list our books. And I'm going to go ahead and copy this here and I'm going to paste that onto the page. I'm going to replace the 'Hello, world!'. And then we're going to go ahead and reload our page. And as you can see there, we have a card. It's not styled very well within the page. So, next thing we're going to do is that we're going to add a container. And this is a '< div >'. And the 'class=' is as mentioned ' "container" '. Then inside of that ' "container" ', we are going to put our ' "card" '.

So, let me go ahead and make myself some space. I'm going to go ahead and grab the ' "card" '. And I'm going to go ahead and paste it onto the page. And I'm also going to go ahead and add a style or a header. I'm going to go ahead and add in an '< h1 >' header, and I will call it 'Books'. And before we show that on the page, I'm going to go ahead and add some local styles. These are some styles that are going to extend the default styles for Bootstrap. So, I'm going to go ahead and create a new file. I'm going to paste some styles that I wrote beforehand. I'm going to write, I'm going to go ahead and save that page, styles.css. And then I will add it to my page.

So, I will enter here 'link', and then the name of our file, which is ' "styles.css" '. And now, if we reload the page, as you can see there, we get some nicer styling. And you can see there that the card looks like something we might want to put a book inside of. And so, what I've done beforehand is that I work with the styling that the card gave me. I added a table inside to structure some of the book information. And let me go ahead and show you now what I created. So, here I have a sample book and I'm going to go ahead and remove the one we have. And I'm going to go ahead and paste the sample book. And I'm going to go ahead and reload the page.

And as you can see there, we get something that we might expect when we get a list of books, we have the Title: we have the Subtitle: the Author: the Publisher: and some Description: about the book. You can see here how that is created. We have the ' "card" ', the ' "card-header" '. We have an '< h5 >' heading five for the 'Subtitle:'. Then we have a table, which is an HTML table of information we're leveraging some of the styling from Bootstrap. You can see here the green text that you see on the card and that continues all the way to the 'Description:' and then we close most of our tags. Now, I'm going to go ahead and add the rest of the books that I prepared so that you can see a full listing.

And I'm going to paste them here at the end of my book. And then I'm going to go ahead and reload the page. And as you can see there we get a listing of books. Now, this was pretty easy to do. Bootstrap took us a long way. However, when it comes to the maintenance of this text, you can see here that even enlisting a handful of books, There's a great deal of syntax to get right, to maintain, to keep track of. And if you're writing it by hand, this is pretty exhausting and it's hard to

read through and find things. So, although this is pretty good, it certainly looks good. In subsequent lessons, we're going to see how to simplify this.

Video 5 – Create Custom Element - Web Component

In a previous lesson, we created a listing of books using components from Bootstrap. And you can see here an example of one of them. And if I scroll down here on the page, you can see the HTML syntax for that element. We have a ' "card" ', we have a ' "card-body" ', a 'header', some of the body tags, and you can see here some tables that are also laying out the content inside of the card. And we're even leveraging some of the styles for the coloring when it comes to the categories. So, this worked pretty well. But as you can see here, there's a great deal of HTML that needs to be written. And if we have a long collection, this can become troublesome and difficult to maintain.

It would be much better to have something that was pretty trim and that simply focused on just the attributes and created a custom element, something like the following. As you can see here, we have a custom element called 'mit-book', and the attributes are the bare minimum that we need for that card. And somehow all the other syntax would be created. Now, it turns out that we can do precisely that with native web component, and that's what we're going to do next. I'm going to go ahead and delete this custom component because we have not created it yet. And I'm going to start off by creating a new file. We're going to go ahead and save it. And we will save it as books.js.

And inside of it, we're going to make use of the classes that have been built into the language in order for us to build a custom web component. So, we're going to start off here by creating a 'class' called 'Book' that 'extends HTMLElement'. Then inside of our element, we're going to call our 'constructor()'. This is all boilerplate that is required when you're creating a native web component. Inside of it, I'm going to call 'super();'. We're not going to do anything with that. We're simply going to create a very simple web component that simply renders our card. And that's going to be little more than an HTML literal template that has placeholders for the variables in the attributes that we're going to pass in.

So, we'll start out precisely with that. I'm going to go ahead and create a variable for the 'title'. And we're going to get access to the attribute, through 'this.getAttribute'. Then we're going to enter the name of the attribute and we're going to call it '('title');', as you saw in that mock definition that we had. And then I'm going to do the exact same thing for the rest of those attributes. You can see here that I've done precisely that I had taken the same approach for a 'subtitle', 'author', 'publisher', and 'description'. I'm now going to set the innerHTML of this element. And we can do that by entering 'this.innerHTML'. And then we're going to create a template literal. And I'm going to enter my backticks.

And inside of it, I'm going to put in the HTML code that looks very much like the HTML definition that we use to create the card that you see in the browser. You can see here all of that syntax. However, instead of it having the values for one of the books, you can see there that I have variables for '${title}', '${subtitle}', '${author}', '${publisher}', and '$(description}'. Now, that's all we need to do in order to create our class. However, we now need to define this custom component. And we do that using 'customElements.define', we enter the name that we're going to give this custom component and I'm going to call it '('mit-book', '. And then we're going to reference the class that we just wrote, which is 'Book');'. That's it. That's all we need to do.

And as mentioned the structure that I'm following has to do with the standard on how to create a native web component, an additional tag that we can use inside of the browser. So, we are extending the amount of tags that the browser will recognize. Now, I'm going to go ahead and add this to the page and I will do it here at the end. I will enter '< script >', and then I will enter 'src='. And inside of it, I will write ' "books.js" ', and that adds it to the page. So, let's go ahead and review. We have very much the same syntax as before. We are defining a HTML card using Bootstrap attribute. So, the Bootstrap component to be able to create a card.

And now, we're going to include a custom one that should create the exact same thing with the exception that this is a custom components. So, you can see that that's much trimmer, leaner, simpler to read. And let's go ahead and reload that in the browser. And as you can see there, we now have two components that look exactly the same. Both of them have the same information. Yet the one that we just entered, as you can see, is so much easier to read. And if you were writing many of these, as sometimes you might have to. This is a simple technique to be able to clean up your code. And if you saw a long collection of these, that was all web components as opposed to, or native web components, a custom element, as opposed to what we would have to create if we simply use Bootstrap alone, you would see that this is a much cleaner definition for this list of books.

Video 6 – Create Web Components Programmatically

Previously, we created a list of books using Bootstrap components. We then took that syntax'.html', and we encapsulated it in a custom web component. And you can see here that we created an element that was much leaner, simpler to read and to compose. And that we captured all of that syntax in the custom web component, as you can see here. We read the attributes, and then we use the syntax'.bootstrap' defines. And we created a template literal into which you have placeholders, which are, which has the attributes that are being read from the component. Now, in this case, we're going to consider the scenario where we have the structured data, in this case, JSON. And we have the list in a way that we can read programmatically. In that case, we want to create the instances of the custom web component programmatically.

And in order to do so, we're going to make a few changes. So, the very first thing that we're going to do is we're going to delete the web components from the HTML. That is, we're not going to be

writing them by hand. And instead, we're going to create a '< div >', and we're going to give it an 'id' of ' "books" '. And we will later use this '< div >' to target it and populate it with the elements that we're going to construct programmatically. Now, we're not going to be accessing ' "books" ' directly. Instead, we will create a file that we will call ' "main.js" '. Let's go ahead and create that next. I'm going to go ahead and save it, and I will call it 'main.js'. And then the next thing that we're going to do is that we're going to modify our custom element. We created it to be authored from the HTML file, to be authored directly.

But in this case, we want to do that programmatically. So, we're not going to be reading the attributes. Instead, we're going to be setting or we're going to be passing an object through which you can read all of its properties, which will be the data about the books. So, I'm going to close this constructor here. Then I'm going to create a new method. And that method is going to have a parameter to take the books. And we're going to make that a 'set book()' and take the parameter I mentioned, which is '(book)'. And then we're going to go ahead and wrap that up in the '< body >'. And so, as you can see there, that's a pretty minimal change.

However, we can now assign or pass in a book, which means that we will be accessing properties. And we need 'book.' for each of those data fields, properties, in this case, since we're reading them from an object. So, I've gone ahead and made all the changes there. That last line is the line that adds the definition to the browser. And now, we're going to go ahead and start writing our 'main.js'. So, just to recap before we move forward, we have an HTML file that has a '< div >' that we're going to target with an 'id' of ' "books" '. Then that page itself is going to call in 'main.js', which is what we're about to start writing. And the very first thing that we're going to do in this page is that we're going to 'import' the file with our custom web component. So, we're going to enter here ' './books.js' '.

Next, we're going to add an 'addEventListener' to the page. And the event that we're going to be watching for is for the document the page to '('load')'. When that does, we're going to write, run a function, a callback. And I will make this simply or not simply, I will use the '=>' notation to do this. Let me go ahead and enter the '< body >'. And here all we're going to do is that we're going to call a method called 'createBookList();' that we will write in a moment. Go ahead and close here. And now, we're going to write that method. So, this is going to be an 'async function' because we're going to be reaching out to that JSON file. And this is 'createBookList()'. Now, inside of here, we're going to reach out.

And the response is going to come back to a variable called 'response'. And we're going to 'await' for the 'fetch' of that JSON file. And the JSON file is '('./books.json')', which is this file here. Now, the next thing that we're going to do is that we're going to pull out the 'json'. And this is part of the language that is part of the functionality of what the 'response' is. And so, here we're going to pull out 'json'. And so, that gets us a handle on the data that is in that file. Next, we're going to get a handle on that element in the HTML file, which is here, the '< div >' with an 'id' of ' "books" '. And we're going to call this 'books'. And here we will enter 'document.getElementById' And the 'id' that

we're looking for is '('books')'. And so, at this point we have the data, we have a handle on the '< div >' that we're targeting.

And now, we're going to loop through all of the entries in that JSON file. So, as you can see here, we have ' "books" ' as a property, and then we have an array of objects, as you can see here, that have the data about the books. So, let's go ahead and loop through that. I'll enter, 'json.books.forEach()'. And we're going to use once again the arrow notation. The parameter I'm going to call '(book)'. Then I'm going to enter my '=>'. And now, the body of the function. The very first thing that we're going to do is we're going to create an instance of our custom web component. And we're going to call that an 'element'. And we can create that instance by entering 'document.createElement'. And then we will enter the name, which is '('mit-book')'. And once we have that element, we can set the 'book'.

That is the set method that we wrote here. And we are going to 'set.object'. And now, the only last thing that we need to do is we need to append to books, a new element. And so, we're going to use 'appendChild'. And that is going to be our new element called '(element);'. So, once again, what we're doing is that instead of creating each of these entries in the HTML file by hand, we're now doing it programmatically or leveraging the data that we have in a JSON file. We have modified slightly our custom web elements so that we can create it programmatically and set each of those objects. And all of that is now in, being reference through this ' "main.js" '. Now, I'm going to go ahead and make this a module. And all I have to do that in order for us to be able to do this is to type 'type' and then the ' "module" '.

So, if I didn't make any mistakes, we should be able to see now, the list of books. So, let's go ahead and reload the page. And as you can see there, I do not have any books. And if I took a look at the developer tools, you can see there that I have a permission problem. And that is because the browser is not supposed to be rifling through the file system. And instead, in this case, we need to run it from a server. The assumption is that you own the domain of your server and that is okay to reference. So, I'm going to go ahead and spin up a small HTTP server that is part of, that is an 'npm' package. So, let me go ahead and do that. So, I will enter that here, 'http-server'. As you can see there, it tells me where I can go. I'm going to go ahead and enter that. Oops! I'm typing in the wrong window here.

Let me go ahead and get rid of that comment to avoid any errors. And now, I will go ahead and enter 'localhost'. And I will click on 'books.html'. And as you can see there, we have all of our cards neatly laid out. So, in this case, not only are we taking advantage of the custom web component to simplify our creation of these cards for each of these books. We're doing that programmatically in 'main.js', as you can see here. And it is pretty clean syntax. So, this is a good way and a good practice when it comes to creating your UI. Most of the frameworks follow similar approaches, even if they are doing so with proprietary code. In this case, we're using standards that are part of the browsers.

So, this approach will work everywhere. And because I made this a module, not only am I leveraging the mechanism that we've just been over. This also gives us some added protections

when it comes to not polluting the global space variables. So, this is an example that brings together many different things. It brings together Bootstrap components. It brings together custom web components. It brings together modules, and it creates content programmatically. So, it's a good one to walk through. It's a good one to experiment with. It's a good one to play with variations and test some of the scenarios that you might want to investigate. This is a pretty simple use, but you can take a look at how to handle eventing and many of the other things that you may want to do.

Video 7 – React Components

We have been considering the scenario where we want to render a list of books, as you see before you. Initially, we did it by hand, writing HTML and using Bootstrap styles. Then we took that general template, that component, and created a browser in HTML native web component. And we saw how much cleaner that could be. Where we created a set of custom tags with custom attributes. And how we can make a very clean trim definition as opposed to just pure HTML. We then took a step further and created all of this content programmatically. And we're able to get great efficiency when it came to creating all of the components that we have been considering, which are really just representing a set of books.

Now, we're going to take the same approach, but we're going to do it in React, React broke new ground initially by defining a very tractable set of components at a time when the browser really hadn't gotten that far. And so, let's go ahead and import our code that we had been working on. Imported this time to create React components. We will start out by renaming some of our files so that they're more consistent with the naming for React that we have been using. And so, I'm going to go ahead and rename my 'books.html' to 'index.html'. Then I will rename the 'main.js', and I'm going to call this one 'index.js'. As before, we have our styles and then we have our data, which is the list of books. And then we have 'books.js', which we will keep with the same name.

So, now let's go ahead and move to 'index.html' and see what kind of changes we want to make. And one of the first ones is pointing to our local styles. This is our ' "styles.css" '. The next thing we will remove the header that we have there. And instead of having ' "books" ' as a target, we will call this ' "root" '. We will then load our React libraries, and then our components. As you can see there, I am pointing to ' "books.js" ' and ' "index.js" '. And in fact, I'm going to go ahead and remove the ' "container" ' for now so we can have a trimmer page. Now, let's go ahead and save our file and move to 'index.js'. So, I'm going to go ahead and start working here.

And in this case, we're going to simply return a component or a top component that we're going to call 'App'. And so, I'm going to go ahead and remove the code that we have at the moment. I'm going to go ahead and enter the top component that I mentioned, and then 'return();', which will be empty for now. And below it, we're going to add that component to 'ReactDOM' using the

'render()' method. And we're going to pass to it, a tag called '< App />'. Then we're going to target the element on the HTML page as usual called '('root')'. And so, for now, let me go ahead and enter an empty fragment here simply to get rid of the red underline. I'm going to start off here by creating a couple of state variables. And I'm going to go ahead and start with one called 'data,'. Then we're going to set it using 'setData'.

And we will 'useState()' to initialize. And shortly, here we will use that variable to load our data into. Now, we're going to have an additional variable. This one will be called 'loaded,' and 'setLoaded'. And this one we will use to hold up the UI while the data loads. So, just as before, 'React.UseState()'. And in this case, we're going to initialize to '(false);', meaning the data is not yet loaded. Meaning, it is false. Next, let's go ahead and use the 'React.useEffect' method. And you can see there the signature. I'm going to go ahead and clean that out. Because at this moment, I simply want to note that we will be running this event only once. And you can see there that inside of it, we're going to now write the body of the function. We'll start off here with an asynchronous method.

And we're going to use this to make a call and get the data. And so, I'm going to go ahead and call it 'getData()'. Then inside of the body, we're going to go ahead and use the usual notation when one uses a fetch call, that is we are reaching out for data. And in this case, I'm going to go ahead and write that. So, we're going to go ahead and reach out for a local file, and this is simply '(' ./books.json');'. Now, we're going to go ahead and extract our 'json' as we have been doing. And this part, is now new, where we're setting our data, and we're going to set our '(json);' into our local variable here. And then once we do that, we're going to set our loaded to '(true)', meaning we can now see the UI, and we will write that logic shortly.

So, as you can see there that is very much the same logic we have used in the past, with the exception of setting data and setting loaded, the variables that we have locally. So, after that, I'm going to go ahead and call that function. And now, let's go ahead and put in a 'console' right below that simply to see what is being loaded and when it's being loaded, once we start to render the UI. And inside our JSX, we're going to start off with an expression. And that expression is simply going to have 'loaded' first to see if that has been loaded, as I've mentioned a few times. And next, we are going to, if that is, in fact, loaded, then we're going to go ahead and loop through that data. So, that's going to be 'data.books'. This is the same thing we've done in previous components when we did it within the native web components and then with an HTML as well.

And then we are going to 'map()' the following function. We're going to go ahead and take as parameters the item which I will call 'book', and then the index 'i'. And I'm using ES6 here notation. And then we're going to go ahead and instantiate the component that we will write shortly, which will represent each of the books. So, this is going to be 'Book'. Then I'm going to pass the property which I will call 'data'. And that will be the '{book}', and then a 'key'. So, that React doesn't complain about me making a list that does not have a key. And so, I will enter for the 'key={i}', and then close that tag. So, now let's go ahead and save our file. Let's go ahead and open up our 'books.js'. And as you can see there, this is our HTML web component.

Now, we're going to go ahead and write a React component. And so, this first part will change here to 'function Book'. And we're going to take one parameter. That parameter will be the '{data}'. Each of those will be a book. And then we no longer need the 'constructor();'. And the only method that we're going to have is going to be the 'render'. Not the 'render' but the 'return'. And since React allows us to have declarative code within our JavaScript, we no longer need a string. And so, this is the start here. And I'm going to go ahead and close it here. Now, I have one-to-many curly braces. So, let me go ahead and clean that up. And now, since we've decided to call it 'data', we can go ahead and remove this part, which was what we were using before in our literal template.

And now, I'm going to go ahead and change all of those to simply be curly brace, I need to remove the old one. So, '{data.'. Okay, so that is all for this one. Let's go ahead and save this file. Oh, and I'm glad I caught this one. We no longer need that either. And so, just to review, we've rewritten our 'index.js'. It is now a parent component that has books. And we load our data and 'useEffect' here. Then we set it into our data variable. And we have a variable, as mentioned before, to track whether the data has been loaded or not. Once that is loaded, then we loop through all of those books and we create an entry in a tag for each of those books. So, let's go ahead and load this now on the browser. We don't have that path anymore. So, let me go ahead and reload, and looks like we have an error.

And the very first one is that I no longer need 'main'. We have renamed that file. And it looks like we have something else missing. Oh, I've just noticed that I've entered the styles twice. Okay, removed our transpiler. Let me go ahead and add that in. Let's go ahead and reload the page. Now, we have some errors. That is our classNames and if we scroll down, we can see that now we have all of our components. So, let's start to clean up. Let's go ahead and move here. And all of our classes. Since we're now living in React land, have to be called name, 'className' as opposed to just 'class'. Let's go ahead and reload the page. And it looks like we are having some problems when it comes to parent-child. The reason for that is because React is a lot stricter when it comes to HTML.

Then the browser, I'm going to go ahead and remove this paragraph from here. I'm also going to go ahead and enter a tag for table body '< tdody >'. So, I'm going to go ahead and wrap the table that we have here, or the table elements that we have in it. Let's see how far that took us. And it looks like that took care of the problems that we had. Now, let's go back to 'index.js', and note here that the very first time we write to the 'console', you can see here that the value of 'data:' is 'null'. Let me go ahead and not make this, but make the console a little bit bigger. You can see the very first time that is printed to the console, 'data:' is 'null', but then that the following time already has content, although it has not fully loaded, you can see here 'false'.

And then by the third time, we have loaded is 'true', and then our array as well. So, I'm going to go ahead and close here the console. And I'm going to go ahead and add some formatting element simply so we can see it as we did before. So, let's start off here by adding a ' '. And this '< div >' is going to have a class, which here we will call 'className'. And that's going to be the '

"container" ' we have been using. And I'm going to have all of the book elements inside of that. We're also going to add a header. In this case, let's go ahead and add an '< h1 >', and we will call it 'React Components'. And let's go ahead and reload our page. And you can see there that we have something much closer to what we've had in the past.

So, to review, we rewrote some of the names of our file simply to have something closer to your prototypical React application. We went ahead and added our styles. We included Bootstrap as we have been doing on all along. We have our transpiler, the element that we're targeting, React libraries, and then our content for React, which is the books component and also 'index.js'. Inside of it, we have a parent component that builds everything else. And as you can see there, we have a state variable that we call data that held the books. Then another one that we use to render our UI conditionally. And then down here, we have the loading of all of those book components.

And the book component was defined in our 'books.js'. Very close to what we had previously when we had web components. You can see there that it's a very similar pattern. So, this had a lot of things. It had bootstrap. It had a pattern that was very similar to web components, and have programmatic loading of the data. And then, it had a looping and an injection of the components into the page. So, once again, a lot of things going on here. Go ahead and test your assumptions. Write the code that I have written. Replicate the functionality, and test your assumptions.

Video 8 – Web Components In React

In this lesson, we're going to take a look at how to use web components with React. Initially, we took a look at plain HTML with Bootstrap by hand. Then we wrap that up in web components. And then we did a similar approach but in React. In this case, we're going to combine both. We're going to be using web components native to the browser and React components at the same time. So, you can see here the code that we have been using. And in this case, we're going to change very little. All we're going to do is that because we're using declarative statements within React, we're passing data onto pure HTML components. Then we're going to pass each of those attributes one at a time. The properties that we normally use within React components. In this case, we're going to pass them one by one. So, I'm going to start off here by entering 'title='.

And the value of this will be '{book.title}'. Then we're going to have 'subtitle=' and the rest of our attributes. Now, we no longer need 'data' as we did before. And we will leave the 'key='. Now, let's go ahead and move to books.js. And here, I'm going to go ahead and delete all the data that I have. And I'm going to go back to an earlier version of web components that I have, that I had, and you can see here that I have slightly different syntax. The definition is the same, 'class Book extends HTMLElement' 'constructor()' is the same. However, I'm now using an event 'connectedCallback()'. That event runs each time the element is added to the DOM. And so, in this case, that is precisely what I want. You can see here that below that I am getting a handle in all of the attributes that we have on the page.

So, the 'title' 'subtitle', and so on. And then I am constructing, as I have before, my string literal template, where I have place folders for each of those values. The bottom of the page then is the definition we have done. And now, when we go back to our code within index.js, we can enter the

name of our component and the name of our component, as you can see here is ' 'mit-book' '. And we're going to change here 'Book' to 'mit-book'. So, let's go ahead and save that file, save the other file. And let's go ahead and reload our page. And as you can see there, we have the exact same thing. Let's go ahead and remove this line simply to show that we're looking at the data and we're not loading some by mistake, some previous version.

So, very small changes as you can see there. We have to pass our arguments in a different way, but we can then leverage native web components. The advantage of native components is that they will work on all platforms and all browsers natively without any libraries. And in this case, we can leverage them from React. And another reason you might want to do it is that some third-party vendors that create some of these components target in neutral platform simply to avoid writing many different versions for many different types of frameworks. So, now it's your turn. Go ahead and make the changes that I did. Go ahead and write a custom web component just to see that you can in fact swap this out and have the expected functionality.

Video 9 – Introduction To Routing And The Single Page Application

Routing within the same page turns out to have a number of considerations that are important to recognize. The so-called SPA, the single-page application. It's a great experience to the user. It's very close to what we used to have with native desktop applications. And in the case of the browser, we're doing the same thing. We load one page, never navigate away from it. And all of the updates that we do, the updating of data, the updating of the UI, the navigation within the page. It's all done by manipulating that one page. And so, because of it, we wanted to give you here an experience or a walk-through in creating this type of framework, this type of routing within a single page with no front-end framework, you could do it all from the ground up, simply using HTML and JavaScript.

And so, we will be looking at how to route using buttons and fetch the capability of browser to reach out and take content or bring content in. Some of the same page linking, we will be looking at browser events and URL access within JavaScript. How do we route using location hash and how to provide the right history to the browser so that if you're moving back and forwards, the history works as users would expect it. We'll also look at some of the optimizations that you can do and some of the bumps that you might find if you're trying to create this for yourself. If, say you were trying to create a library, again, once again, we are doing this simply to illustrate some of the design decisions that those in the front-end frameworks have had to do. So, enjoy seeing the routing from the ground up, and dig in, you'll see and get to appreciate some of the solutions that have been done.

Video 10 – Overview of Frameworks And Routing

Let's consider a scenario where you wanted to share information about your organization. And say that after careful consideration, you came up with an information architecture where you had 20 pages or so. Pages like about team contact, and a number of products and services. Historically, each of these would have been an independent page, and you would have created a linking mechanism between these pages. Sometimes, you would have even been given a sitemap to be able to overview all of the information on the site. And here on the screen, I have an example of this type of pages. You would have had some type of sign-in. You would have had potentially some blog, some pricing information, and many other pages.

A very reasonable approach. However, when it comes to round-tripping to the server, there's a great deal of delay that the user may experience. And although it might not seem like it. Even tenths of a second of delay can have a big impact on the user experience. There are additional issues as well. I'm going to go ahead and add a navigation bar to each of these pages. In frameworks such as Bootstrap, which is what I'm using for styling between these pages, gives us some great components that can add great blocks of functionality to the page. As you can see here I have on the editor, all three of these pages may also have a navbar that is ready to go. I'm going to go ahead and expand it.

And as you can see there, there's a lot of HTML, and you will see what that looks like in a second. I'm going to go ahead and copy this and I'm going to put it into each of the pages. Here's the first one, then the second one. Then the third one. I'm going to go ahead and reload the page here. And as you can see there, I get this nice bar, navigation bar across the top. If I reload the next page, I will do the same thing. And somehow, I'm not getting my update there. And there it goes. It looks like the page was cached. Going to go ahead and reload pricing. And you can see there that we get the navigation bar as well. Now, this is great functionality. We can bring in these components, and this has certainly moved forward. These open-source libraries are ability to construct pages.

However, we are repeating information amongst all these pages. And on a large side, you could potentially, have thousands of pages. Imagine having no update, each one of those navigation bars. So, it would be much easier to have a centralized resource, or if we made one change, we could change all of the pages. And parts of this are already hinting towards a page where we can load components into, and we can make these transitions without having to roundtrip to the server. And in order for us to do that, we need to come up with some mechanism that replaces the navigation between the different pages, and that is called routing. And in this case, we're talking about client-side routing, where we are adding and removing pages.

And in this section, we will go ahead and walk through some of the implications and some of the design considerations when constructing these routers. But more at large, we are touching on many of the same design considerations when it comes to JavaScript frameworks at large. We will look at this in a simplified example. But many of the large frameworks are addressing many of the same design issues when it comes to routing; when it comes to state management; when it comes to how you organize your code. So, ultimately, we're trying to provide a better experience

to the user. And we're trying to create a more efficient design on our backhands as well. Even if in this case, our backend is, in fact, the front end.

Video 11 – Route Using Buttons

There are many ways in which you could load content and the background and then update your UI. In this example, we're going to reach out and grab additional HTML pages and use them to populate the page that we're looking at now, the index page. And as you can see here in the directory, we have also a blog page, we have a pricing page, and then we have a sign-in page. Now, the index page, the one that gets served by default, the one we're looking at here in the browser has a number of things, as you can see here. I'm going to go ahead and open up the '< head >'. And as you can see there, I am making a reference to Bootstrap styles. This is something you can get directly from the Bootstrap site. I then have a navigation bar.

And this navigation bar, because we have a central page that will be loading onto it. Additional pages will be shared by all. However, it's not being repeated. It is only being defined in one page which will function as a hub, as a router of content. And so, the content that we put on this page will be shared by all. Then in addition to that, I have here three '< buttons >' as you can see, and each of them call a function called 'loadpage' and pass a parameter. And the parameter that will be passed will be the name of the page that we're trying to load. Then in addition to that, on line 54, I have a '< div >' with an 'id=' of ' "content" 'onto which we will load the HTML that we will get from those pages. So, let's go ahead and write our function.

We're going to use an asynchronous function. That is because we're going to be making calls, which we do not know when they're going to come back. And so, then we're going to call the function called 'loadPage', just as indicated there. We're going to take the '(page)' in a parameter called '(page)'. And inside of there, we're going to first make a call. And we're going to catch it on a variable called 'response' And we're going to 'await' the completion of that call. And we're going to use 'fetch' to reach out for that file. And 'fetch' is part of the JavaScript language. And it's going to take as a parameter, the '(page);' that we're trying to find. Then we're going to extract the content. And we're going to once again use 'await'.

And we're going to extract the 'text();' from that 'response'. Now, once we have that, we're going to get a handle on the element. And the element name is, or the element id is content. We're going to reach for that using 'document.getElementById'. Here, we're going to enter the id which is '('content');'. Once again, this one. And then as a last step, we're going to set the 'innerHTML' of that 'element' with the 'content;'. that we have brought in from the page, this one here. So, let's go ahead and reload the page. We're going to go ahead and reload. And let's go ahead and click on Sign in. And as you can see there nothing happened. And when we take a look, the developer tools at the Console. We can see there that we have a problem accessing the files.

And that has to do with the sandboxing of the browser. Now, if we run this on a server, then the browser or the web considers that to be a particular set of content that you own. And because of it, on the server, it is permitted to do fetching onto your domain. And so, let's go ahead and run a small server. If you are familiar with some server already, you can go ahead and use that. I'm going to go ahead and use one that is an npm package that is called HTTP server. And as you can see here, I'm going to enter that command. And you can see there that it's pointing me to an address. And I'm going to go ahead and now access that on the browser. So, that's localhost:8080. And as you can see there, the page looks the same. So, let's go ahead and click on Sign in and nothing happened. Let's go ahead and see if we see an error.

Load page is not defined on, that is because the page is cached. Let's go ahead and reload the page. And as you can see there, now we get the Sign in. So, let's go ahead and click on Blog. Let's go ahead and click on Pricing. We can come back to the Sign-in page, and you can see there that we are loading dynamically the content from these pages. And then we are extracting the text, which is the HTML, and then loading it onto the place holder that we have on that < div >. Now, this mechanism of routing certainly works. We're using buttons, however, we're not making full use of the web conventions. We're not using true URL linking, for example. And there's other, a number of other shortcomings that we will see in the lessons to come.

Video 12 – Same Page Linking

I have a simple page on the browser here. As you can see, it's mostly text separated by a few headings. And here, at the outset, we have three links. I have the source for the page here in the editor. And you can see that each of those links point to '>w3c<', '>MDN<', and '>Ecma<'. And if we click on the link, as expected, we navigate to w3c. Now, there is a different type of linking that takes place within the same page. And we can see that if we navigate to many Wikipedia pages and we click on the navigation. So, say we went to Semantic HTML we're further down on the same page. And if we come back, we're back at the point that we were within that navigation menu.

Now, a feature to note within same page linking is that we are not going to the server when we click on that link. This is internal to the page, and because of it, it forgoes that expense of making that connection out to a server. At the same time, the navigation is indexed and it is part of the web machinery. So, we could take a link directly to that section, you know, if we were to open up a new page and we pasted that address, we would navigate directly to that section. So, it leverages the mechanism, the machinery, the infrastructure of the web. And at the same time, it provides us a way to link within the page without having to go to the server. So, this is something that we will later leverage when it comes to routing or creating a router within our page.

Now, let's go ahead and create an example of that within our page. And all we need to do in order to do this linking within the page is to provide an element with an id that we're pointing to. Let's go ahead and add that to each of these. And so, I'm going to go ahead and add at the same place where the headings are, I'm going to provide an 'id=' and I'm going to call this one ' "about" '. Then the next one I'm going to call and notice that it doesn't matter which type of element it is. In this case, I'm using < h1s >, but it could be using < divs > or it could be using anything else. And so, I'm going to call this one ' "mission" '. And then the last one is going to be ' "history" '. Now, I'm going to go back to the top and I'm going to replace these with a link that is within the same page.

So, note that I use the '#', which indicates to the browser that this is a link within the same page. So, here I'm going to first go to ' "#about" ', and we'll enter '>about<'. Then I'm going to replace the next one, which is ' "mission" '. Again ' "#mission" ' Then I'm going to enter '>mission<' here. Then the last one is going to be ' "#history" '. And I need to add the '#' and enter '>history<' for the label. So, I'm going to go ahead and save that page, reload the page. And as you can see there now we have about mission and history. And so, if I click on history, you can see there that I have navigated all the way to History. And I'm not exactly at the same point because the page ends.

But if I reload the page now, well, let me go ahead and go back to the beginning and restart the page. And if I click to history, you can see there that I click to the section as expected. If I click on mission, the same thing happens. And if I click on about, I navigate to that part of the page. So, as mentioned, this follows the same approach of linking that we would do if we were linking on to other pages or if we're linking onto other pages, navigating through our own server. However, in this case, later, we're going to use this mechanism of not leaving the page that is not going to the server and using that to create a routing mechanism.

Video 13 – Listening To Browser Events

If we're using inpage linking to route to content that is not on the page as in this example here, but to content that we will be loading dynamically, we want to know about those clinks, those events in the browser. Now, if we look at the eventing information within the browser, as you can see here, this one is MDM documentation. You can see there are a multitude of events that are flagged by the browser that is that you have access to in your code. In this case, we're looking for HashChangeEvent, in particular, and you can see here that there is some information about it.

So, let's go ahead and go back to our page and let's write some code that will detect that hash change, that click by the user. We'll start off here by adding a 'window' listening event. And the function that we want to hook onto is called 'addEventListener', then the specific event is '('hashchange','. And the function that we will call, the callback, I'm simply going to call, 'callback)' here. And then we're going to write our function. And we're going to catch that event, as a parameter. And then inside of that function, I'm going to simply write to the 'console' the event. And then second, I'm going to write the, just a simple message to myself that says '('location has changed');'.

Okay, now let's go back to our code. Let's reload the page, and let's go ahead and open up our developer tools. And I'm going to go ahead and increase my font size so we can see our messages. I'm going to go ahead and click on Mission. And as you can see there, we get the event rich of all the information relating to the event and relating to the paths and relating to the information that was clicked on, as well as a message saying location has changed. And so, you can see there that we would get the same for the other links on the page. Now, this is information that we can leverage within a routing mechanism.

Video 14 – URL Location Access With JavaScript

In addition to being notified of the '("location Hash Changed");' event, we can also get access to the location. Now, I'm going to go ahead and show you how we can do that. I'm going to go ahead and simply write to the 'console' here. And I'm going to access that through the '(location.' object that is available on all pages. And in this case, I'm going to look at the full address. Then I'm going to do a couple more properties. The second one will be the 'pathname);'. The third one will be the 'hash);', the hash address part of the URL of the location. So, let's go ahead and reload this page. Now, I'm going to go ahead and open up my developer tools.

Go ahead and increase my font so we can see it going to go ahead and click on Mission. You can see there that we have the event information, but in addition to that, we have the location, the full address. And right now I'm loading directly from the file system so you can see that. You can also see the pathname without the protocol and also the hash part of the address. Now, if we did that here for History, the last one would be #history. And then About as well. Now, if we went to a side that had a much richer URL address, like Amazon, if you've ever paid attention, you know that they create very long URL's.

And I'm going to go ahead and open up our developer tools here. And I have some code that accesses more of those properties. And so, I'm going to go ahead and paste it on here. And you'll see there that that address is much longer and is looking at the different parts of it and writing them there to the console. So, as you can see there, we can get access to the event that something has changed. We can then access the location. And in this case, the path that is part of that change.

Video 15 – Route Using Location Hash

Earlier we routed content within the page using buttons and onClick events. Here we're going to repeat the exercise, but this time we're going to be using links. As you can see here on the code, we have three links. One of them is for '>Sign in<' the other one is for '>Blog<', and the third one is for '>Pricing<'. And we're loading all of this through a page called index.html, which is being loaded by default through the server. Now, as you can see there, the last line there on that block is for ' "content" ', and that is the place holder into which we will put the HTML that we will load.

So, we will start out here by getting a handle on the event that will notify us of changes in navigation. And we're going to use the 'window' object, then 'addEventListener'.

The event listener that we're going to add is '('hashchange','. And then we're going to 'loadpage);', which is a function that I will write next. Now, I'm going to make this an 'async function' because we're going to wait for that content to load when we reach for it using fetch. And the name of the function is going to be 'loadpage()'. Inside of it, the very first thing that we're going to do is that we're going to write to the 'console'. We're going to write the '(location.' of the 'hash);' address. We're going to do that first, and let's go ahead and reload the page and open up our developer tools, and loadpage. it says, is not defined, and that's because I did not get the right casing. Go ahead and reload the page and go ahead and click.

And you can see there that we are getting the paths as expected. Next, we are going to clean up the address to be able to get the name that we will use to reach for the page. And we're once again going to use the 'location.' object, then the 'hash.', and then the string 'replace'. And we're going to remove that hash. Next, we're actually going to reach out and put the 'response' into a 'response' variable. And then we're going to 'await' the 'fetch'. And the thing we're looking for is the '(page)'. Then we're going to get the content of the page. We once again use 'await' then 'response.'. And then we are extracting the 'text();'. Next, we're going to get a handle on the 'element'.

And we do that by using 'document.getElementById', the name of the element we want is '('content')'. And you can see it there, here. ' "content" '. And once we do that, we're going to set the 'innerHTML' of content and we're going to set that equal to the text that we got from the response. So, let's go ahead and save that page. Let's go ahead and reload the file. And let's go ahead and close to the developer tools and click on Sign in. And you can see there that we get the page of Sign in, which is this one, then Blog, then pricing. And I have collapsed there, the HTML, just so you can see what's on the page. And that was Sign in, then we get Blog, then we get the Pricing.

Now, as opposed to what we had before where we had buttons and clicking, now we're using linking, which is the way the web is stitched-together, which means that we can go back to the Blog, and then we can go back to Sign in. So, now we are getting closer to the functionality that you had before we started to load things dynamically on the background when you would've had linking between pages. Except now we are catching those events within the page before they go to the browser. And we're dynamically loading content onto the UI.

Video 16 – Building A Router

Let's write a router that brings together all of the different points that we have covered. As you can see here, I have a simple page that has a number of links. And you can see those here in the source. They are links within the same page. They start with a hash. And if we open up our

developer tools at the moment, if we click on any of them, nothing happens because we don't have any logic to handle those events. So, I'm going to go ahead and close here the events. And I'm going to go ahead and move to the script part of my file. And we're going to get started with catching that event. We're going to do that through the 'window'. Then we're going to go ahead and 'addEventListener'. Then we're going to listen for the '('hashchange', ' event. And then we're going to call a function called 'router);'.

That is our callback. Going to go ahead and start on that 'function' next. That's the 'router()' function. Inside of it, all we're going to do is we're going to confirm that we are catching the right path. And so, in this case, '(location.hash);' will write to the console. So, let's go ahead and take a look at that. Let's go ahead and reload the page. I'm going to go ahead and open up our developer tools, as you can see there. And now, I'm going to go ahead and click on Sign in. And you can see there we get Sign in, then Blog, then Pricing. Next, let's go ahead and create an object that holds the 'routes'. And as you can see there, I have accounted for the case where no route is given, that should be sent or that should load the ' 'home.html' ' file.

The same thing when only a slashes provided as well as home and some other cases in more broader scenario, you would have much more information. But here, we're going to start off pretty simple. Now, inside of the 'router()' function, I'm going to go ahead and build the link to get the requested page. So, '// get requested page'. And we're going to call that variable 'link'. And that is simply going to have the 'location.hash'. Next, we're going to get the path, the route for the page. So, let me write a comment here, '// get path (route) for page', and we're going to assign that to a variable called 'route'. Then we will reference our 'routes[]' object, and we will use our hash as the key, as the property name. Then we're going to create a condition that says, '// if route exists,' then 'load page'.

And we're going to test in the following way, 'if (route)', then 'loadPage', and we're going to write that 'loadPage' function in a moment. This is going to take the path which is the '(route);'. Now, we're going to address the very first loading of the page by running 'router();'. You can see there that we have accounted for the case where there is no address beyond know hash address within the path. And so, that will take care of that scenario. Now, we're going to write a function to load the page. This is very much the function that we've written before. So, I'm going to go ahead and paste that into our code here. So, I'm going to go ahead and do that. You can see there that is an 'async' function. Once again, we need that because we're loading, we're going to be using 'fetch' and 'await'.

You can see there that the very first line reaches out for the file. The second one extracts the text. The third one here gets a handle on the < div > within the page that we're going to use to populate the content. And then we set the 'innerHTML' with the 'content' that we extracted. So, let's go ahead and reload the page. And then go ahead and do that a few times. And you can see there that by default we are loading now the home well, I have it on pricing from the previous navigation, so let me go ahead and remove that. And you can see here that we are loading a homepage by default. If I go to Sign in, we get a different page, then Blog, then Pricing.

And I can go ahead and remove once again the path. And you can see there that we have a default page. So, this has brought up everything that we've done previously. You can see here that we have rounded out by creating a number of paths that can be defined independently of the code. We then have a routing function now that gets that path, that location, hash location from the object that is part of the page. Then we use that to query our object and extract that property. And if we in fact have a path for it, we go ahead and load the page as we have done in the past.

So, this is a good exercise to walk through to get comfortable with because it can be of practical use. There's some additional concerns that are refinements on this idea. But it's important to note that we're not building a commercial product, we're simply exploring the design considerations. If you were to set out to build a router, this is not meant to be a replacement, and it's meant to be for academic use. That is, for you to think through some of the issues. So, go ahead and try this, try some variations, see what additional shortcomings you can identify. And what extensions you can add.

Video 17 – Same Page Linking Limitation

One of the disadvantages of the router we have created is that since we have hijacked linking within the page, what happens if you wanted to have linking within the page? So, let's take a look, here I am at the homepage, but if we move to sign in, which I have replaced the content on, you can see that I have three sub lengths inside of the page. The link to sign in on one, sign in on two, sign in three. And each of these, as you can see the source here on the left-hand side inside of the editor, have that linking within the page, right? And so, if we were to link to any of these, this works. However, if we were to reload the page, that would not work. You can see there because this is not being handled by a router.

We are routing on the pages that we're loading onto our host page. However, we're not dealing with the links within those pages. And so, if someone wanted to handout this address to a friend, they will not be able to link directly to that section. So, there's a number of ways that we could address this. The one way that I'm going to use is that I'm going to add a hierarchy to these links. That is, I'm going to give myself the necessary information to understand that this is a link within a page. And as you can see here, I have the name of the page that I'm loading. So, that is if we mouse over here, if you look in the address, you can see there that sign-in is the information that my router has to be able to load this page.

So, that is 'signin.html'. And then in addition to that, I am providing the link that is within that page, the page that I have loaded. So, let's go ahead and see what changes we need to make to our router to accommodate that. So, I'm going to go ahead and move to the router. And remember once again, what I'm trying to do is I'm trying to load the same page again. However, provide myself some information so that I can scroll to the right link within that dynamically loaded page. So, I'm going to, ultimately, all of these, as you will recall when you're linking within the page, our

elements, and we're linking to those ID. So, in this case, you can see here the 'id' as ' "SignInOne" ', and I have the same naming convention for ' "SignInTwo " ', and then ' "SignInThree" '.

And so, what we want to do is we want to get the name of the element that we're linking to within that dynamically loaded page. So, I'm going to call that 'var innerElement', and I'm going to initialize it as an empty string. Then I'm going to take a look at that location hash. And if I have more than one parameter in the link, then I know that I am targeting an element in the page. And the code that I'm going to use is the following. You can read through it in detail. For now, all I'm going to say is that I am counting the number of parameters on the hash string. And if I have more than one, then I'm going to pick up the 'innerElement'. Then the rest of the code is the same that we had before, except that I'm now going to pass 'innerElement' to my 'loadPage' function.

So, I will add that parameter here as well, 'innerElement'. And then I'm going to consider here an additional condition. And that condition is going to be 'if(innerElement)'. So, I am taking a look and seeing if I have an 'innerElement', and if I have an 'innerElement', then I'm going to 'scroll' this is going to be the name of my function, 'IntoView'. And I'm going to pass that element. I'm now going to write that function. As you can see here, I've given myself a comment here. And I will enter 'scrollIntoView'. That's going to take the '(id)', the 'innerElement' id is what I'm passing in the parameter. And then inside of it, I'm going to use 'document.getElementById()'. This will give me a handle on that element. And then I'm going to 'scrollIntoView'. So, this is going to be 'scrollIntoView'. And that will take us to that link, to that element within the page.

So, let's go ahead and reload our page. Going to go ahead and go home here, reload it and make sure that I'm going to go ahead and open up my developer tools simply to disable the cache here. And then I'm going to load that page again. And now, I'm going to go ahead and load to the third one. You can see it there. And now, if I open up a new page, and we paste that address, you can see there that, that works as we would expect it. So, now we have made a fix to our router that deals with in-links, that deals with pages that have links within their own content. So, we have routing at two levels, if you would.

One of them just leveraging the ability of HTML pages to have links within it. And the other one where we are routing ourselves. So, that is at the 'index.html' we have created our router and we come up with a mechanism to pass enough information so that we can scroll to the right place. So, now it's your turn. Go ahead and take a look at the code, make sure you understand all of the steps that are being taken. Perhaps explore an additional way to deal with this type of additional requirement and see what type of possibilities there are within the features of JavaScript and within the design that you can come up with.

Video 18 – How To Avoid Reloading The Page

We address linking within the dynamically loaded page by adding information to the link. You can see here if I mouse over, I have first the name of the page which is Sign in, and then a slash for

the name of the element I'm linking to within the page. However, what might not be apparent in that approach is that we are reloading the page every time. And for text, that's not very costly. But you can imagine a page that has a large number of videos which take a few seconds to load when you load the page. And if you're linking within that page, between the different videos and you have to reload that page every single time in order to go to the section that you're linking to that could become quite expensive and lead to a bad user experience.

So, we want to avoid doing that. In order to do that, one way would be to remember to create some state. And here you can see that I have created a new variable that is a global variable, called 'history'. And inside of our router, which is unchanged, we're going to add some logic. And this is going to take a look at the following. We're first going to take a look and see if our 'history' is the same as our 'link'. And if that is true, we also want there to be an 'innerElement'. That is, we're checking, are we in the same place? And if we are, are we navigating somewhere within that dynamically loaded page to a certain element. If that is true, then we'd want to skip loading the page and we can go straight into scrolling into the view.

That is, we are within the same page and we are navigating to a certain element within that page. And so, then we simply pass the '(innerElement);'. And then we're going to reset our 'history' to our new 'link'. And then we're going to 'return;'. We're not going to do anything else. Otherwise, we're going to set our history to the link for the next navigation that the router will handle. So, by making that change, and I'm going to go ahead and open up my developer tools here to avoid cashing. Within the page you can see there I have select the Disable cache and I'm going to load it a couple of times to make sure that is gone. Now, when we load within the page, it is much smoother because we're simply moving within that dynamically loaded page as opposed to loading the page dynamically and then moving to the element that we're targeting.

So, once again, with text here, that doesn't matter that much because it's pretty cheap. And also, I'm loading things locally. However, if you have content that takes a second or two to load, that could lead to a pretty bad experience if you're reloading the page every time. With this approach, we're simply scrolling to that element that we're targeting. So, you can see here that regardless of which way you go, because you're trying to create your own custom client-side routing, there are a number of considerations to take into account. And that within this simple router that we have created, we have already seen a couple of issues. So, now I invite you to try some of these considerations yourself to see if you can come up with a cleaner router to see if you can identify other shortcomings.

It's very good exercise to go through because many other considerations are the same ones that the bigger, broader, more commercial offerings and routers have. And so, this is one that you can play with. You can see that we started to creep into state here with taking, remembering something throughout the different navigations. So, we're looking at state management and directly, we're trying to consider some of the ways of organizing our content. Once again, many of the same considerations that routers take into account, and more broadly, that the frameworks need to address the large frameworks that attempt to solve many of these issues for you.