Tiered Bad Bank

Video Transcript

Video 1 – Introduction To Bad Bank

Bad Bank will be one of the most in-depth modules of this course. We will start off by creating a simple application that is simply composed of HTML and JavaScript. We will then take that same code and refactor it to make use of one of the widely used frameworks that exist and that is React JS. We will then consider how we can extend this functionality, not just keep the state in the front end and the application solely on the front end but how do we extend that to have server-side component and server-side functionality that take some of what was taking place in the front end and moves it to the back end. Likewise, when we consider the state of this application, we will move all of the data management of it onto a database server.

We will look at some of the fundamentals of databases there, and we will look at some of the design considerations. Ultimately, we will then refactor that front end to be able to consume those new API routes to be able to make use of storage in the database. And then we will finalize by giving you the challenge of considering how you could secure this application. We started off with the notion of a bad bank because it didn't have security. And we will challenge you to add those components to be able to secure this application and walk through some of those design decisions and consider the case that you are tackling when it comes to security. It can be quite large and that will require some judgment on your part. So, this one is one where you can learn a tremendous amount and you can walk through a number of technologies.

Video 2 – Bad Bank Exercise Overview

In this exercise series, we're going to be building an application we call Bad Bank. Bad Bank because it has no security. Now, as you can see here, we have a portion of a user interface. And you can see here the common elements that you might expect to see in a banking application. Create Account, Login, Deposit, check your balance, and so on. And you can see also here a few first indication of what that design might be, a Container, a navigation bar, and some components, some modules that we're swapping in and out. Now eventually, we will build a full three-tier application, as you can see here. But initially, we will start off with a simpler case. We will construct everything within the front end. We will capture the state, the information, the data, and keep everything local. That will help us to get our arms around this application before we add more complexity before we add a back end and then a datastore.

So initially, as mentioned, we will simply be using HTML and JavaScript. We will use no JavaScript frameworks. However, because styling can be challenging onto itself, we will bring in a Styles Framework. And that Styles Framework, it's Bootstrap. So, instead of writing our own .CSS files, we will simply reference those from Bootstrap. Now, as you can see here, we, I am calling out here modules and components, and we will use these terms interchangeably. And they correspond to a block of functionality that is represented in the user interface. And you can see here a number of them being called out. Now, conceptually, you can think of them as independent files, as you can see here, a navbar would have a representation within modules in JavaScript than in API which would be what communicates eventually with the back end.

But initially would simply simulate that and keep the state local. And you would have something equivalent for the rest of the modules for createaccount, for balance, for deposit, and so on. However, as you can see there, those long bars indicate that really all of this is being captured in a modules.js file, and then above it in the green in an api.js file. Now, all of this is being brought together into one container, an HTML file called index.html, which will load the module, which will load the API, which will load the styles and control the application. So, as you can see here, in the file system, we will have three files. However, you can think of the conceptual organization in the following way, that you have this multitude of components that we are coordinating. So, with that in mind, let's go ahead and start building this application.

Video 3 – Bad Bank Index Page

Let's build our index.html file. The file that will bring in all of the additional files in the application. As you can see here, I have an overview of the code where I am highlighting that in bringing in 'bootstrap', I'm also highlighting that I will have a 'div' which will be a ' "target" '. We will load components in and out of that ' "target" '. And then there at the bottom of that file. We have the two files that we have referred to previously, one for the definition of the 'modules' and one called ' 'api.js' '. So, let's go ahead and build that page. As you can see here, on the left-hand side, in my editor, I have the files that I overviewed when I looked at the code on the slides, I have api which is empty, I have modules which is empty, and I have index that just has some boilerplate placeholders for my HTML. Now, I'm going to start off by adding the styles that is from Bootstrap.

And as you can see here, I've put in that link there that will bring in that file and give us access to the Bootstrap styling within this application. Next, I'm going to go ahead and add some HTML tags. And as you can see here, I have a 'div' called ' "navigation" '. I will use that to add our navigation bar once we write it. And then below that, I have the ' "main" ' area of the application. And I have also inside of it a 'div' called ' "target" ' that we will use to load the modules that we will be creating later. Now, the styling that I'm using here is from Bootstrap. And if you want to learn more about it, you can navigate to Bootstrap and take a look at the classes and styling that is available. Now, the last thing that I'm going to do is I'm going to go ahead and add those JavaScript files.

You can see there my reference to ' 'api.js' ', and ' 'module.js' '. So, I'm going to go ahead and save that. And now, I'm going to go ahead and drag and drop that file into the browser. And as you can see there, there's nothing to take a look at because we have no user interface elements. But if I look at the code behind, you can see there that the code that we wrote is now on the page. So, this is just the first definition of the container that brings in the styles that references those JavaScript files, API, and modules and define some initial elements for the UI that we will be targeting once we moved to building components.

Video 4 – Navigation Bar Module

Let's start building our user interface components. We'll start off by building the navigation bar, that is that bar across the top of the page that has the links that we will be using to generate events and load components. In this case, we're going to go ahead and use the styling from Bootstrap. And given that we will be using quite a bit of functionality, a lot of the components, the style components from Bootstrap, it's worth seeing where that code comes from. So, if we select Documentation within the Bootstrap project and we entered Navbar, you'll see there. And we scroll down, you'll see code very similar to the one that I'm going to be writing in a moment. And as you can see here, the styling is already pretty close to what we have on the page.

So, with that in mind, and understanding the source of where this styling comes from, let's go ahead and write our code. We'll start out in the blank module.js file. And I'm going to go ahead here and start off by creating a 'ui' object. And this 'ui' object is going to contain all of the modules on the page, initially just the navigation one. So, I'm going to go ahead and show you that next. And as you can see here, this code is pretty close to the code that we see within the Bootstrap project. As we move through it, you will be able to see here different parts of that 'ui', 'Create Account', 'Login', 'Deposit', 'Withdraw', 'Balance', and so on. And there's some additional calls there that we will return to when we're looking at the logic that being connected to these user interface elements.

Now, let's go ahead and add a couple more lines. And as you can see here, we're getting a handle on the element id here, which is 'navigation'. And then we're simply adding all of this, all of this string, which is really HTML, onto that component 'navigation.innerHTML' as you can see. So, I'm going to go ahead and save it. And I'm going to go ahead and show you what this is referencing, as you can see here. Just worth repeating what we covered before. And now, let's go ahead and load this into the browser. We're going to go ahead and reload the index.html as previously. And as you can see there we get this nice navigation bar across the top. I'm going to go ahead and make it bigger.

Not so big that it wraps, but you can see there the elements that I called out when I was overviewing the code, Create Account Login, and so on. So, this is our first module that we're adding onto the page. The source came, the source styling came from Bootstrap. There's some additional code that we are adding, but we will review that later. But you can see here how that is

constructed and programmatically injected into the HTML page. That is the code that we showed before and how it is added, as you can see here at the end. So, with that in mind, now it's your turn. Go ahead and create a component of your own before you go any further and inject it as well into the page.

Video 5 – Additional Modules With Bootstrap Cards

In our previous lesson, we added a module to our Bad Bank application. The module that we added was one for a navigation bar. And we used the component styles from the Bootstrap framework. In this lesson, we're going to add the remaining modules that you see here on the screen, Create Account, Login, and so on. Now, for this type of module, we're going to use a different Bootstrap component and the component that we're going to use to surround or wrap our module is going to be the Card component. To look up the documentation of Cards, we can navigate to Bootstrap, select Documentation, go ahead and enter Cards. And you can see here that this is the documentation. If we scroll down the page, we can see there a sample card. And if we were to go further down, we see some of the ones that are more richly colored like the ones we will use on the page.

You can see here the strong primary colors. And if we scroll past them, you can see here the syntax that we can use to wrap some content in a card. So, now back in the editor, we can go ahead and make those additions for those modules on the page. And as you can see here, we have the one we wrote previously for navigation. And now, we're going to go ahead and add some additional ones for our additional modules. So, as you can see here, I have one for create 'ui' or 'createAccount', and this is going to be the module for that. The next one is going to be one for 'login', the next one after that 'deposit', and 'withdraw'. And the rest of the modules, as you can see there being defined. The wrapping syntax is that of the cards, and then internally we have some element input elements or buttons depending on which of these modules we are creating.

Now, once we get to the end of this, you'll see that I'm also defining functions that can be called, that will draw or call that definition. And you can see there that they're being added to the user interface. If you remember, the front end has a module, a module element here, or an HTML element this you can see here called ' "target" '. And we're setting each time we call one of these functions the inner HTML of that 'target'. So, you can see here the definitions of all of these functions, minimal functions that are just simply going to go ahead and assign the content of that element, the 'target' element. With one of these modules that we have predefined in the code, one of these long strings, as you can see here. Now, at the end of all of this, we're going to have a 'defaultModule();' which is defined here.

And you can see here that this is going to be the 'ui.default'. Then if we scroll up, you will see here the default, which will be the homepage. So, let's go ahead and save the page. Move to the browser, and let's go ahead and reload that page. And as you can see there, we have the navigation bar that we had previously. And in addition to that, we have our new module, the default

module, that is being brought into the page. So, as you can see at this point, we have created a structure that we can use to be able to load content dynamically. We have created a navigation bar and then in addition to that, we have a target element that we will be populating depending on which module we are loading at that point on the UI. So, at this point, go ahead and replicate that structure. Go ahead and add a default module and make sure that your user interface is behaving as it should.

Video 6 – Dynamic Interface

So far, we have created an HTML page into which we can load components, the modules we have constructed. But we want to go further than that. We want to be able to have a dynamic user interface that can respond to user events. For example, clicks on the navigation bar and that the right components, you can see here our list of modules can be brought into the user interface dynamically. Now, to illustrate this point, let's go ahead and make Create Account the default component, the default module that gets loaded instead of the landing module that you see here, the BadBank Landing Module. So, let's see how we would do that in the code. As you can see here, I have my modules.js file, where I have the navigation, create account, and many of the other modules we have defined.

Now, the one we want is 'createAccount', 'ui.createAccount'. So, this is the one we want definitely. And if we look further down on the page, we will be able to see the 'CreateAccount' that is being loaded into 'target', which is our user, our each HTML element that we have defined within index.html. So, we want to be able to run this function as a default. So, instead of running here 'defaultModule();', which is what we had originally, we're going to go ahead and write here, 'loadCreateAccount();'. Go ahead and save that. Now, we're going to go ahead and reload our HTML page. And as you can see there now the default module that is being loaded into the UI is the Create Account. And so, you can see there that there is that dynamic loading of components onto that HTML element on the page, the target element.

And we can do so at any point that we want programmatically, but we also can do it as a response to interactions with the user as mentioned, to events and I mention specifically there, click. And so, that is precisely what we have. And if we scroll up here, you'll see here that the navigation, in fact, has a lot of 'onclick' components that will respond to the user clicking on that navigation bar on that link. And that it will call to each of those components that we have defined and it will load the right module. And so, you can see here we have ' "loadLogin()" ', 'Deposit()', 'Withdraw()', and so on. So, let's go ahead and take a look at what that looks like in code, in code within the browser. And so, at this point, actually, let me go ahead and put back the default. And we will remove this and enter here 'defaultModule();'. Save it, reload the page.

And as you can see there, we're back to our Landing Module. And now, if we click on any of these links in the navigation bar, you can see there that we're now getting each of those modules that we defined, but being loaded dynamically onto the page. So, we have Withdraw, Balance, and

AllData. So, as you can see there, we have achieved our goal. We have a dynamic user interface that is loading modules based on events, in this case, click events. So now, it's your turn. Go ahead and replicate what you saw. You should have most of it already, but make sure that the functionality is in fact what you think it is, that you are, in fact, being able to load those components based on clicks on the navigation bar.

Video 7 – Local State

Now, that we have a container that we've added styles, that we've added modules, that we have a dynamic user interface. Now, we need to add the logic, we need to add state for our data. And we will do that in our api.js file. Conceptually, the same organization that we use with modules we will use with APIs. You can see here that a corresponding block of code, that is the logic, will accompany each of those modules. So, let's go ahead and write some code. We will start off by creating a 'context'. This will be the state of the data. We will be keeping the state local. And simply to emphasize here, I will say that this is being 'globally shared', and we will create a context and I will call it 'ctx' for short. Inside of it, we will have an array, and that array will be of 'accounts:'. And I'm going to go ahead and enter the brackets there for the array. Then I'm going to go ahead and make sure that I use the, I use strict JSON here.

So, I'm going to go ahead and give that the double-quotes. Now, I will initialize that array for ' "accounts":' with an object, our initial object, this will be simply seeding the application with one user. This user will be ' "peter" ', and you can see there that ' "peter" ' has an ' "email" ', a ' "balance" ', and a ' "password" '. Next, let's go ahead and write the first block of code that will accompany a module, and this one will be for create account. And so, I will enter here 'function create()', then the body of the function. Inside of it, what I'm going to do is that I'm going to get a handle on the elements of the card. And I'm going to go ahead and show you that right now. So, if we go to modules, and we scroll up to the 'createAccount', you'll see here that we have some input elements. You can see here 'id', ' "name" ', ' "email" ', ' "password" ', and so on. And so, here we want to get a handle on those programmatically.

So, I'm going to go ahead and enter here 'name', then 'document.getElementByID' and we will be using the IDs that we gave it. And so, the very first one will be ' ('name'); ' and I will do the same for the rest. So, I'm going to go ahead and change here to 'const' and I'm going to go ahead and line up my code. So, as you can see there, the IDs that we are referencing is ' ('name'); ', ' ('email'); ', ' ('password'); ', and ' ('createStatus'); '. And so, you can see here that we have 'id' of ' "name" ', ' "email" ', ' "password" ', and ' 'createStatus' '. Now, at this point, programmatically, when this is executed, we would have gotten a handle on each of those elements on the page. And then we're going to go ahead and extract their value and add one more user to accounts. So, as you can see here I have 'ctx.accounts.push', and then I have here the names matching the pattern that we set up. Therefore, a user of 'name:', 'email:', 'password:', and 'balance:'.

And I'm referencing here the value that corresponds to each of those elements that we got a handle on, 'balance:', I'm going to initialize at '0', as you can see. So, now that we've added the user, the only thing that's left to do is to clear that form in case there needs to be another user that needs to be created. So, that is this last part here. I'm going to update the status. So, that is to clear out the form and then set a status message of account being created. So, I'm going to go ahead and save this. So, now let's go ahead and move to the browser and see if that functionality matches what we expect. I'm going to go ahead and navigate to Create Account. Well, actually let me go ahead and reload the page to make sure we get the new code. I'm then going to go ahead and select the Create Account. And I'm going to go ahead and enter clark, clark@mit.edu, then the password is going to be secret.

I'm going to go ahead and create the account. And as you can see there, the fields were cleared. And we have that status message of account being created. And at this point, we can take a look at the Console. And we can go ahead and let me go ahead and make this a little bit bigger, make my font a little bit larger. And we can take a look at the context, which is global. And as you can see there, we have now two accounts in our 'Array(2)'. And if we expand it you can see there that we have ' "peter" ', which was our initial user, and then we have ' "clark" '. And you can see there the information that we've entered for ' "clark" '. So, this was successful. Now, let's go ahead and do one more. This is the one that the UI element that will simply return all of the data. So, if we scroll down, we should be able to see it. And you can see here it's a simple button that is simply going to return all of the data and it's calling on a function called 'allData'.

So, let's go ahead and go back to the UI, and let's go ahead and write that function. We're going to go ahead and enter here 'allData'. And then we're going to write the body of the function. This is simply going to be here 'status'. And we're running, go ahead and get a handle on the 'ElementById', as we did before. And the name of this element that we're trying to get a handle on is called ' ('allDataStatus'); '. Then once we get it, all we're going to do is that we're going to set the 'innerHTML' as we did previously. Then we're going to go ahead and enter 'JSON.stringify()' so we can create a string from our object. And we're going to go ahead and enter here 'ctx.accounts'. And then we're going to go ahead and assign that to the interface when this button is clicked. So, let's go ahead and save. Go back to our UI, and let's go ahead and reload the page to get the new code.

Let's go to AllData and Show All Data. At this point, "peter" has gone away because our status local. And so, if we reload the page, we lose that context. But you can see there that we can see the default user. And if we were to create clark again. We'll enter clark. Now, the user is cached. So, I'm going to go ahead and enter secret. You can see there that that account was created. And if we select Show All Data, you can see there that "clark" is now part of our list of users. So, those were a couple of the accompanying API logic. This is the logic for those modules that we created. There are more to be done, Login, Deposit, Withdraw, Balance, and you could add your own. So, we will leave those to you. Now, it is your turn. Go ahead and write a few more components. Add

something else that you think might be nice to have, and work a little bit with the UI and make it your own.

Video 8 – Bad Bank With React

In our pure HTML and JavaScript version of Bad Bank, we had the following design. We had a general container, which was the page, and then a number of components that we loaded in depending on what part of the application we were using. Now, this arrangement or this organization was conceptual. That is, we had strings of HTML inside of functions. We then had some API that was really mocking the functionality of that page as well. And then all of it was under this index.html container. However, as we move forward and we continue to add functionality, one of the issues that we did not discuss was that we were starting to really walk into some of the functionality that a lot of the JavaScript frameworks give you.

And so ultimately, as your application gains more sophistication, gains more complexity, there really starts to be a strong argument for using one of these frameworks. And you can see here some of what we did. We addressed routing, and we did that with an event-based mechanism that when the navigation bar, item, or label was clicked on, we would load a function then we were looking at state management. How do we keep state management within the page? We defined components, although those were our own definition of components. We had data bindings to some of those elements as well. And we really didn't do a declarative definition. That is the HTML that we embedded inside of our JavaScript was simply treated as strings, meaning it was not understood, which meant that it was hard to debug. Now, there were a number of other decisions, design decisions that we needed to address.

And you can see here that some of those were captured in that previous diagram and some other ones that really need to be addressed as you scale up your application. So, that is precisely what many of the JavaScript frameworks set out to tackle, a lot of that functionality that is common to most applications. And one of the most widely used frameworks that is in the market today is that of React; the React JavaScript framework. And it was created precisely to do front end applications. And so, we're going to do a redesign of our application, the pure JavaScript and HTML application to recapture the functionality using the JavaScript, the React JavaScript framework now. So, we will take a similar approach. However, we will now be using standardized components by the React JavaScript framework, as you can see here. And you can see there the list of components that we're setting out to create as well. You can see here the architecture, however, now we will be creating individual files to hold each of those components.

You can see here that we will have in contexts that is also part of the framework. So, several of the things that we were doing by hand, we were making it up for ourselves. We're now relying on the framework to do it for us. And we will also be breaking up, all of that up into individual files as opposed to have it all cuddled up together in one file. We will be writing more code, and so it makes sense to start to break apart that code to be able to have cleaner debugging and more maintainable code. The other thing that we will address through the framework as well is routing. And finally, the last thing that we will do is that we will have a shared context that is part of the framework as well. Now, we will continue to use Bootstrap Styles globally as we did previously. So, with that in mind, this is an initial overview of what we're going to do. That is we're going to

create the same application that we had created previously. However, we're now you, we're now going to be relying on one of the JavaScript frameworks.

Video 9 – Create App Files

Let's create the files for our application. We will be using the following architecture. And as you can see here, the majority of the files that we're going to be creating are files that are going to contain the modules, the React modules in our case. Now, the ones highlighted in yellow, there are the ones that we will be creating. We will leave the rest up to you. So, we will create these here in yellow and the ones that are not highlighted, those are the ones that you are going to be doing. We will do all of that in index.html, which is the container where we will bring in all of the files that we're creating. So now, let's go ahead and move to the file system. Let's move to the editor in the terminal window. And let's go ahead and create these files.

Let's start by creating the index.html file. And inside of it, let's paste some starter code. As you can see here, this is a pretty simple HTML file, it makes a reference to our 'bootstrap' styles which we will be using globally. Then we are referencing 'babel', which is required for React, the React framework. Then when it comes to the body, we are noting here the element that we will be targeting from our React code to be able to load all of our components into. The next block is the React library, as you can see there. And this bottom part here is the files that we're going to be creating in the terminal window in a moment. Now, as mentioned, we are not going to be creating all of the files. We're simply going to do the ones that are not highlighted there.

So, I'm going to go ahead and delete these, save the file. So, let's go ahead and create those files. And so, as you can see when we come back to the editor, we can see all of those. Although as you can see, they are all empty at the moment. So, now that we have the files, let's go ahead and run a small HTTP server to look at our application. As you can see here, I have that page opened up at npm, the npm registry, and the command to install is right there before you 'npm i http - server'. I have already installed that, so I'm going to go ahead and open up my console. And here are the files as you can see. And so, I'm simply going to enter, I have installed it globally.

So, I can simply enter 'http-server'. And as you can see there, I have a listener in that directory and I can simply hit the index route. And the port that I'm being given there, as you can see is '8080'. So, let's go ahead and enter that address. That would be localhost:8080. And although we can't see anything, because we don't have anything on the page. You can see here that we get the title BadBank React, and if we were to look at the code behind, we would see that the page is now being served. So, now it's your turn. Go ahead and create all of the files that we have created, and don't forget to add the additional ones for you. That is login, deposit, withdraw, and balance.

Video 10 – Navigation Bar Component

Let's add our first component and let's make that component the navigation bar. If you recall the design that we're using for BadBank, we have a container and inside of that container, we have a number of components. The navbar component is the one that is across the top. As you can see, they're noted in the arrow. So, let's go ahead and write our code. And let's open up our navbar.js file. And inside of it, I'm going to go ahead and paste in the code for this component. Now, if you recall, when we wrote our HTML only in JavaScript version of BadBank, we had very much the same syntax. However, as you will see now, this is recognized by the editor. It is highlighted, it is color-coded. It understands what the structure is. And so, this can help us a great deal when it comes to debugging and given us some hints about what could be wrong when we make mistakes.

Now, also note that this is a pretty simple approach to writing a component. It's simply a function and there are few rules you can read more about them within the React documentation. But here I'll point out some basics. One is that the name of the function must be written in uppercase. So, you can see there that the 'N' is uppercase. And then you simply need to return valid HTML. There's more to it as mentioned, you can read about it, and we will talk about some of that here. But I'm giving you here just the basics. Now, the other thing is that the topmost tag that we have within that 'return' must encompass the rest of what we have. And at times you'll see us write some fragments and then we will write empty tags like this, which will encompass the rest of the HTML.

Now, here, you can see here what I mentioned previously about the editor in some of the features that are built into the parsing that allow us to see our mistakes. As soon as I wrote that fragment of HTML, everything else broke because now we have two. When I remove this 'nav', encapsulates the rest in those tags, and so we are once again compliant. Now, the rest of the syntax that you see here is the syntax that is directly from Bootstrap. And so, you can see here the same thing that we had before, but we're now encapsulating in using the functionality that React gives us. I'm going to go ahead and save this. And now, we've created and defined our 'NavBar' component. So, now let's go ahead and move over to index, which will be coordinating a lot of the work that we do within this application. In this case, we are going to define our topmost component, and we will call that component 'Spa' for a single-page application.

Let me go ahead and enter this 'Spa()' there. Inside of it, we are going to do very little. And especially here at the beginning, I'm going to go ahead and put parentheses around everything that we're going to do. And for now, all we're going to return is simply the 'NavBar' component, ' < NavBar >'. And if you recall that simply our navigation bar across the top of the page. Now, we will close that tag there. And then the next part is the 'ReactDOM'. And this will 'render()', and this is just part of the language. You will always have this when you are using the structure that we're using and when you're targeting one element within the HTML page. And you can see here that that element is the ' "root" '. And so, we're going to be rendering into that component. And in this case, we are simply going to be including the React component that we just created '< Spa/ >', which within it contains the 'NavBar'.

And you can start to see here some of the hierarchy that we will have. And then in addition to that, we are simply going to target the component or the element, HTML element that we will be using in index.html. And so, in this case, we are going to use 'document.getElementById'. And if you recall, the ID of that element is ' 'root' '. So, I'm going to go ahead and enter that here and then close my syntax. But simply to note that we are using that ID. And so, at this point, you can see here that we have completed the basics of what we set out to do. We created one component, 'NavBar'. Then we created a second component inside of index, which is 'Spa', which will contain a lot more as we go forward, but at this point simply has 'NavBar' within it. And then we render into the 'root' element on the page, the 'Spa' component. So, let's go ahead and save. Let's then go ahead and run our 'http-server'. And as you can see there, it's listening in port '8080'.

And now, let's go ahead and reload our page. And let's go ahead and take a look at the console. And we do not see any errors. So, let me go ahead and reload the page. And as you can see there, we now have BadBank. A lot of times it's good to reload the page because some of the pages might have been cached within the browser. And another tip is that you can always go to the Network and disable the cache. And that will make sure that you're not loading pages that were cached. So, let's go ahead and make the page a little bit smaller here so we can get the full bar. And as you can see there, we get that bar across the top, and it is using the styling from Bootstrap. So, now it's your turn. Go ahead and add the navigation bar as you're working on that, go ahead and add one more or two-component. And for now, you can simply have them say hello world. Don't worry about adding anything else.

Video 11 – Routing

All applications need a mechanism to route users to the different parts of the application. Often we call this routing. And there's a number of ways in which we could do routing within an application. When we wrote our pure HTML and JavaScript version of Bad Bank, we did this using event-driven clicks by the user, by clicking on the navigation bar, and loading modules onto the page this way. Now, that has a number of drawbacks. One of the most important being that it bypasses the history of the browser. And so, as this application grows, and you get more expectations from users, you want to be closer to the functionality the users expect. And that is keeping the history and the browser having some functionality that mirrors navigation through pages, although in this case, we're loading everything into a single page.

And so, for that, we have libraries within the React framework that allow us to do that. And in this case, we will be adding navigation to our application. So, let's go ahead and move to the browser, not to the browser but to the editor. And we will start off by making some additions to the context. The idea of the context is something that I shared throughout the application. In this case, we will make minimal additions to it, so you could really put it into index. But in this case, I wanted to make the point of the separation of concerns and having a file that contains within it things that are shared throughout the application. So, in this case, as you can see here, I'm going to make

use of the 'ReactRouterDOM', that 'Route', 'Link', and 'HashRouter' as you can see. So, I'm going to go ahead and save this file.

I'm then going to go back to index, and within index, we're going to go ahead and add a code to route through all of our components. And so, I'm going to go ahead and paste the code that I've written previously. And as you can see here, we still have '< NavBar >', but now is included inside of our '< HashRouter >'. And so, the code to note here is '< HashRouter >', which we defined in contexts. This is styling, This is the 'NavBar' that we had previously. And then the routes that we are defining, as you can see there, we have the root route, which would be the 'Home' component. Then we have the 'Route' for 'CreateAccount', and then the one that will show us all the data. So, for now, let's go ahead and create some placeholders, some placeholder content on each of those components. We'll start off with the home component.

We're going to go ahead and enter here 'function Home()'. Then we're simply going to 'return', and we will return an empty string or well, actually let's go ahead and enter an '< h1 >'. And inside of it, we'll say here, 'Home Component'. Now, let's go ahead and save, and we'll take the signature there to use also on createaccount. Except this will be 'CreateAccount()'. And we'll say here the same 'Create Account', save. And then we have one more. Let's go ahead and take a look here. The last one that we need to do is alldata. So, let's go ahead and enter here 'AllData()'. And I believe I entered the message in the wrong place. Yep. So, we can just enter 'Create Account'. Now, we have home, we have alldata, we have index.

And let's go ahead and save this, or start our server. Okay, now let's go ahead and reload our page within the browser. And as you can see there, we get the Home Component. If you do not see it, remember about caching within the page. And if we click on Create Account, we get Create Account. And otherwise, if we select AllData, we get AllData. Now, notice that the navigation is updating, the URL is updating as well. And so, all of this would be recorded into the history of the browser, and the navigation between components or pages as they may be perceived by the user is as expected. So, now it's your turn. Go ahead and add routing to the application. Make sure that you can move between components. Go ahead and add an extra component or two. And for now, simply add some placeholder message, just like we did.

Video 12 – Shared Context

Applications often have the need to share information, to share functionality. And so, the concept of a shared context is one that you see frequently in applications. In our previous version of Bad Bank, the one that was JavaScript and HTML only. We created an object that we shared globally throughout the application. Well, as you might imagine, JavaScript frameworks have something analogous. And in this case, we're going to use that, that is part of React to do precisely that, to have a shared context. Now, the namespace that we will be using, as you can see here is React.createContext, and we're going to be creating, as you can see there, a variable that we will use throughout the application called UserContext.

So, let's go ahead and move to our editor. And let's start out by adding that within the context file. As you can see there, we're calling it 'UserContext'. Now, let's go ahead and use that within index. We're going to go ahead and wrap the part of the application that we want to have access to this shared context. And so, I'm going to go ahead and enter here. As you can see, a line that is going to be adding something to that context initially. And as you can see there, it's one user. I'm going to initialize the application with the user of ' 'abel' '. And you can see there that we have a 'name:', 'email:', 'password:', and 'balance:'. Then we're going to go ahead and close that tag here. So, let's go ahead and line these up.

Now, that we've lined that up, let's go ahead and add or let's go ahead and use that context in one of our components. We're going to go ahead and do it in 'AllData()'. Because as the name implies, all we're going to do there is show all of our data. And so, we're going to create here a variable. And we're going to go ahead and access the 'useContext', and then '(userContext)'. And now, we're going to go ahead and modify our output here. I'll indicate that we're going to be writing a fragment here. And once I enter the matching tags, you can see there that the red that goes away. And then in addition to this, we're going to go ahead and write the context of our 'userContext' by entering or stringifying our object.

And so, I'm going to go ahead and enter '{JSON.stringify}', and then simply the '(ctx)'. So, I'm going to go ahead and save that. Then run our server, reload our page, and go ahead and do it a couple of times. Then go to AllData, let me go ahead and reload that. And let me open up my developer tools to see if this is by chance caching. Let me go ahead and, yes, it was. And as you can see here, I am now getting access to the data that is in the 'userContext'. In this case, the only user that exists is the one that I created initially. But once we start to add functionality to each of our components, we will be able to read and write to that context. So now, it's your turn. Go ahead and add the shared context, and make sure that you can read that context from one of the components that you already have.

Video 13 – Card Component

We're now going to be adding user interface components to our application. And we're going to be leveraging Bootstrap styles. So, if you look here at the layout inside of the overall container, we have a smaller box, The one that says BadBank Landing Module, and that card is one that is defined within the Bootstrap styles framework. If you want to learn more about it, you can navigate to Bootstrap, look at the Documentation, search for Cards, and scroll down on the page and you will be able to see the very colorful cards that we're going to be using. They're coming up here, as you can see. And if we scroll down, you can see there the syntax that defines that card. And to avoid repetition within our code, which is something that is always a good design consideration.

We're going to go ahead and add one card component that we're going to be sharing throughout all of the rest of our components within Bootstrap here. And to avoid repetition, which is always a good design consideration. We're going to be creating one card component that is going to be

shared by the rest of our modules. You can see here what the general component looks like. And we're going to be styling as needed the class style, the body, and the title as we need to. So, we will set those programmatically. And you can see here the type of colors that we will be using. Once again, all of these are defined by the Bootstrap framework. So, let's go ahead and navigate to the context. And as mentioned, this is a good place or this is a good naming convention for some of the things that you're going to be sharing across the application.

In this case, I'm going to go ahead and add a new component. You can see here this is defined by the bootstraps syntax. But I'm setting several of the property, several of the options, several of the styles within this component programmatically. So, I'm going to be passing in that through the properties when I make use of this component. Let's go ahead and save that. And let's make use of this new component within one of the ones that we already have. So, as you can see here, I have the 'Create Account' that for now is simply returning a placeholder. And so now, I'm going to go ahead and replace that with our new component. And as you can see here, I am setting some of the attributes I have here. The 'bgcolor', the 'header', the 'status'. Then I'm going to set an empty in the 'body' which is going to be empty as well. So, let's go ahead and save this. Let's restart our server. Let's go ahead and reload our root page.

Let me go ahead and open up my developer tools so that I can be sure that I am removing, and I think I had a breakpoint there, not sure why, but it's kicking in. Let's go ahead and move this. Let's go ahead and move to network. Let's reload the page once again. Now, let's go ahead and click on Create Account. And as you can see there we get the stub. There is nothing else within it right now. There's no body, no fields, nothing else. But we are getting now the card, and we're getting the styling that we want. So, at this point, we have added a generalized card that we're going to go ahead and set programmatically when it comes to the styles and the fields, and the body, the status, and so on. So, now it's your turn. Go ahead and add the new card component, and let's go ahead and go ahead and leverage that new card component from one of the additional components that you have that for now only have a place holder.

Video 14 – Create Account

It is finally time to create the logic of the application. And here we will get started with our Create Account component. As you can see here, we will show a form, we will show a number of fields, and then there will be a number of actions that can be taken in some state that needs to be managed. So, there will be some code blocks we need to keep track of the state of the variables like name, email, and password. But also of status, what are we showing to the user, and the user contexts that we will use to be able to store the information. Then we need to show this depending on whether we are entering or we are clearing, we need to enter a certain amount of information. And you can see there that we have some conditionals. In addition to that, we need to handle the create and the validate. And in this case, we will do a simple implementation. And depending on what we select or what step we're on, we will show one bit of code or the other.

So, let's go ahead and get started. We'll go ahead and navigate to createaccount. And we will go ahead and replace our placeholder here with the following. The, when we look at what we are returning, the first part we know well, which is simply a 'Card', then we're referring to some new functionality that we haven't seen before. And we need to refer to what's written at the beginning of this function here. As you can see here, we are setting state on two variables. One of them says 'show' and the other one sets status. And we will use 'show' to be able to decide whether we show the 'CreateForm' or the 'CreateMsg'. And we will use 'status' to keep track of the message that we're setting within the user component. Now, when it comes to 'setShow', to 'CreateForm' and 'CreateMsg', those are two new components.

And so, I'm going to start off with the simpler one of the two. As you can see here, this is the 'CreateMsg'. So, when we submit a new user and that is created, we're going to show the 'CreateMsg'. You can see here that it has 'Success' across the top. Then it has a '< button >'. That button is given the user the option to create in other user account. And if that option is selected, then we set 'setShow' to '(true)'. Which means that we will load the 'CreateForm' or 'CreateCount' component once again. Now, the create component it's much bigger because it holds the form. And if we walk through what it contains, you can see here that we have 'name', 'email', 'password', and they were making reference to the context as we have done before. But in this case, we are going to be adding to that context.

So, we will have a 'function' call 'handle()'. And as you can see there, this will be pushing a new user into the context. And once again, we are handling here whether we show or not. So, when we create a new user, we 'setShow' to '(false);', which will then show us the 'CreateMsg' component as a we're shown here in this part of the topmost components. So, notice that we have here a parent which is called 'CreateAccount()' component, that then uses 'CreateMsg' and 'CreateForm'. Now, when it comes to the, what the 'CreateForm' component is returning, you can see here that this is a pretty traditional form. It has some input elements. And you can see here one for the 'Name', the other one for 'Email', the last one, therefore, the 'Password'. And then it has a button that 'onClick' calls the 'handle()' function. So, let's go ahead and load, all of this into the browser.

And so, as you can see here, this is the interface we had before. It's worth noting that there are a couple of options on the server and depending on which one you're using to prevent the server from caching on the server-side. Now, there's also caching on the browser side, which you can address with the option that I showed here to disable cache. So, for now, we're going to go ahead and use the one on the browser. I'm going to go ahead and reload the page. And it looks like I'm going to go ahead and start that, that'll be 'http-server'. Let's go ahead and reload the page and as you can see there, we now get the form with field. So, I'm going to go ahead and enter a new user here, peter@mit.edu, Password is going to be secret and then I'm going to go ahead and create the account. And as you can see there, we get Success, which is the other component, the component that we show when we have created the account.

Now, let's go ahead and navigate to AllData. And as you can see there, the date, the information that is being held in the context now has one more user, which is "peter", and the "email", and the "password". Now, I did not set the balance, so we do not have the balance, but I will leave that to you. So, now it's your turn. There's a lot more modules there to be completed. You can walk through the design, make some decisions. And as you saw here in the component that we created together, there are a number of ways in which this could be designed, in which you could lay out the logic of your component. You could decide to have it all in one. You could break it further down than I have. You could decide to make other, another design. So, it is your turn. Go ahead and add some components and make it your own.

Video 15 – Challenge: Your Turn To Refactor

The more you work with an application, the more opportunities you see to refactor the code that is to simplify your design to integrate more functionality. In this case, you can imagine that instead of using separate forms for each component, you could potentially create one for them all. We did that when it came to the card itself, the look and feel. But could you extend that say to have one form, say, defined in the context, that was common to all of the other, to the other module, to the other components in the design. You can see here a hint that what something like this could be, you know, much more compact functionality, better encapsulation. Perhaps this is the best way to do it but now comes your judgment. What changes would you make? How would you improve the design that you have up until this point? Would you break down further? Would you integrate at a different level? So, this is the opportunity for you to change the application, to simplify it, and make it better for you.

Video 16 – Bad Bank With Back End Overview

We have been working on an application called Bad Bank, so-called because it doesn't have security. And so far we've only written the front end. First, we did it using JavaScript and HTML only plus Bootstrap styles. And then we rewrote the application using a more formal framework, the React JavaScript framework. Next, we're going to extend beyond the front end to have a server and to have a data store. We're going to be going across all the three tiers. We're going to be serving files as traditionally done for HTTP. We're going to have some logic running server-side, as is often the case. And then we're going to have storage, a data store that is going to maintain the state of our application. To do that, we will be restructuring our application. We will move all of our front end code to a directory called public.

And we will create a new Node.js application to host our code for the server and the connectivity to the data store. We will serve our static files using Express, as mentioned, out of that public directory. We will then create a number of routes in the Node.js application using the Express package. And you can see here that those will be equivalent to the user interface elements that

we have in our Bad Bank application, that is Create Account, Login, and the rest of the functions that we have in our application. We will then make updates to the front end and all of the updates to the state of our application. That is the number of users that had been created, the authentication that is done will all now be held in the state in the database.

Which means that whenever we're creating an account, we are logging in, we are confirming some other functionality you'd like returning all of the users. We will be making a call to the back end which will then interface with the data store. And so, the data store as well will have the information that is required in order to create new users, in order to verify login or return the data that will contain the information with which we can verify the login in the server-side logic. Once we have built all these different pieces and verified their functionality, we're then going to bring everything together, connecting the front end to the back end, to the data store, and having all of the pieces working together.

Video 17 – Node Server

Let's write our application server. We will write it using the Node.js platform. And using one of the most widely used packages, which is Express, we will restructure our application. That is, we're going to reuse the code that we wrote for our Bad Bank application using the React framework. And we will restructure. As mentioned previously, we will move all of that front end code into the public directory, as you can see here. And then we will write a new Node.js application that will hold the server-side logic. Then we will serve all of that content out of that public directory and the static front end code using Express as you can see here. And then finally, we will write some placeholder, some placeholder routes in our server to match with those function UIs that are going to be manipulating the content that we see.

That is when we create an account when we do a login, we're going to be calling to the server and we're going to write some placeholder routes for now in the server-side. So, let's go ahead and get started with our code. As you can see here, I have an application called BADBANK_THREE_TIERS. And we're going to start off here by creating a new folder. And we're going to call that folder public. And inside of it, I'm going to drag and drop all of the files that we wrote previously for our Bad Bank application. So, I'm going to go ahead and drop into that folder. And as you can see there, we have all of those modules that we wrote, all of those components that we defined using React, we have their alldata, balance, context, createaccount, and so on. Next, we're going to write our Node application. Now, let's go ahead and open up the console to do that. And I'm going to run 'npm init' here.

And I will use the '-y' option to go with the defaults for now. So, I'm going to go ahead and hit return. And as you can see there, we have our basic package. We now need to install Express. So, I'm going to go ahead and enter 'npm install express'. And I'm also going to go ahead and install the 'cors' package, and this is to avoid conflicts once we start using more than one tier. Now, let's go ahead and add our index file, or what we will call index file for our root application

here in the Node.js side And I'm going to go ahead and enter here index.js. And inside of it, we're going to start off by requiring the packages that we just installed. So, I'm going to 'require ('express');', then create an instance of that, which will be our 'express' app, and then requiring '('cors');'. Next, we're going to denote the folder that we're going to be using for our static content.

And that is mentioned previously as our 'public directory'. And then that we're going to be using '(cors());' as well. Now, let's go ahead and add our first, our first definition for a route. And I'm going to go ahead and close the console here. And as you can see there, this is a route to be able to create new users. I have '/account/create', and then I'm going to take three parameters, a '/:name', an '/:email' and a '/:password:'. And at this point, all I'm going to do is I'm simply going to echo that to the calling client. I'm then going to enter a couple more routes. One for logging in a user, and the last one for creating all accounts. Sorry, that's not creating all accounts, by returning all accounts, and it looks like I have an error. Oh yes, I'm missing a comma.

And so, for now, that's all the routes that we're going to go ahead and enter. The only last thing to do is to start our listener. And as you can see there, we're going to be running on 'port = 3000;', and we have a 'console.log' message. They're simply saying that. So, let's go ahead and save our file, open up our console. Let's go ahead and run our new file. And as you can see there, we're 'Running on port: 3000'. So now, let's go ahead and test if our server is doing what we think it's doing. So, we'll start off by simply hitting our localhost:3000. And as you can see there, this file is being served as we expect that is the static content on the default route of the application is being served. And as you can see here, we get what we expected, our Bad Bank application with all of our tabs, we can click on Create Account, Login, Deposit, Balance, and AllData.

And we have the functionality that we have built into our React application. Now, let's go ahead and hit one of those routes that we created and that are simply placeholders at this point. And as you can see here, I have on the address, on the address box there, localhost:3000 /account/create /bruce/bruce@mit.edu/secret. So, I'm going to go ahead and enter that. And as you can see there, I get echoed back that information that I just sent. So, now it's your turn. Finish writing the placeholder routes on the server for the remaining user interface elements. So, that is Deposit, Withdraw, Balance. And if you've written any additional ones yourself, some additional user interface elements, go ahead and write the placeholders for those as well.

Video 18 – Database Introduction

Data is an important part of applications. In fact, it's hard to think of a computer program that runs without data. And this has been true through history. The way to represent the data, the way to store it, and how we manipulate it has been something that is challenging. And so, over time, a different architectural component, a data store to store and manipulate the information was created. And there are many types of data stores. But here we will be discussing two. The first one is relational, which is by far the most popular one historically, where data's represented as tables. The second one is a document data store, which is a great fit for JavaScript components

and JavaScript data. In fact, a document, one of the core units of this representation is really a JavaScript object, and we will see more about that in the latter part of this discussion.

For now, let's think about some of the database design elements. When it comes to the document data stores, you have collections of documents which would be a collection of these JSON objects. Then we have fields and properties, that is the properties that we have inside of those objects. And then how do these collections and properties interact? And we care about this because this is the way in which we will answer queries that are necessary within the applications that we're writing. Now, when it comes to the relational side of the conversation, we then talk about tables and we talk about columns. And ultimately, when it comes to these relational systems, we're talking at a high level about tables, the entities that they represent, and the relationship between them. So, that is the entities would be the objects, the attributes would be the properties.

The relationship is how do they connect between these different types of entities, and ultimately, one of the last one is cardinality ratios. That is, for example, here we have departments of which you might have one in a university, and a department may have many programs or professors. Here is another one that might be useful to look at that represents media. And you can see there the relationships between them. But we will be using a document database. And one of the advantages of document data stores is that the documents are independent units. That is, you have these independent JSON objects. And because of it, the application logic is easier to write. Now, there are some advantages of relational and many other types of databases depending on what you're trying to do, as tends to be the case.

But one of the big advantages of data stores, of document data stores, is that we can store data in a way that is difficult to fit into the relational model that is, unstructured data can be stored easily just like you can put into a JSON object, any structure you like without having to conform to a certain schema. So, you can see here a representation of a document. And as mentioned before, as you can see, this is very much a JSON object. You can see also that a collection is a number of these, as you can see. And here you have some of the sample operations that you might do with this data store. You can see here how to create, how to find, and how to update and delete. We will also be using a visual manager to be able to interact with the database simply to confirm that the transactions we perform in our code did in fact take place.

Video 19 – MongoDB Standalone

The document database that we will be using is MongoDB and we will be running Mongo inside of a Docker container. You can see here the instruction that we're going to run at the command line. You can see the port, you can see the name that we're giving it. And so, now let's go ahead and move to the terminal and run this command. Let's go ahead and open up the console. Let's paste that instruction there. Let's go ahead and run it. And now, we can confirm if that is, in fact, up and running by entering 'docker ps'. And as you can see there, we have our mongo instance

up and running. And you can see there the information that we passed in the name of 'mongo' and also of 'badbank', which was the pretty name if you would, that we gave it.

Now, before we integrate our full Bad Bank application onto the database, we're going to go ahead and write a small standalone client that will run a Node on its own without Express or anything else, and make the connection to MongoDB simply to confirm that we have the access and we have the permissions to be able to write onto this database. The npm package that we will use is called the same thing as Mongo 'mongodb.' And you can access more information about it by going to the npm registry. So, now let's go ahead and install that package. We will enter 'npm install mongodb'. Now, let's go ahead and add a new file. We're going to go ahead and call it 'mongo_test.js', and inside of it, I'm going to go ahead and close the terminal. Inside of it, we're going to go ahead and require that package.

And as you can see there, I've done precisely that. The next line is the connection, the address for the path to that database. And as you can see there, that is the default port that we entered when we created the container. Next, we're going to go ahead and connect. And the options that you see here are the ones that are specified by the package. And if you want to read more about it, as mentioned before, you can navigate to npm and look at the documentation. Here, initially, we're simply going to test the connectivity that we, in fact, connected to the database. So, I'll write a message here saying ' 'Connected!' '. Let's go ahead and save that. Let's go ahead and open up our terminal. I'm going to go ahead and clear here. Let's go ahead and run that file. We'll enter 'node mongo_test.js'. Let me go ahead and move that message away. And as you can see there, we connected successfully to MongoDB.

Now the reason that hangs is because I never closed the connection, but I'm going to go ahead and simply close the application. I'm now going to go ahead and write, that is insert a document into this data store. I'm going to go ahead, and I'm going go ahead and close the terminal. And I'm going to paste here some code to do that. As you can see here, on the very first line, we are defining the name of the database, and if that database does not exist when we connect to it, it creates it. You can see here the first user that I'm going to insert, I'm creating some information here randomly. Then I try to insert that into the collection as you can see here. And so, let's go ahead and run this code.

And then we're going to go ahead and navigate to the visual manager that I mentioned when I overviewed MongoDB. And we will be able to see that through that tool as well. So, let's go ahead and save it. And let's go ahead and run it. Go ahead and 'clear' the console here and run 'node mongo' once again. And as you can see there, we're able to insert that document. Now, as you can see here, I have a pop-up. I've gone ahead and launch Robo 3T, which is the visual manager for MongoDB. And I'm going to go ahead and connect using the default information. So, as you can see here, I am connected to the database. I have the config and System, which are the fault. And then I also have myproject, which was the database that we just created. Now, if I move inside of it, and I move into Collections, and I take a look here at customers. I'm going to go ahead and right-click and View Documents.

You can see here that I have one user. And you can see there also the information that was passed in, which is the randomly generated information. However, it is, it is using the @mit.edu that we passed in as well. Now, let's go back to our code. I'm going to go ahead and cancel, 'clear'. Let me go ahead and minimize that. And I'm going to go ahead and add some more code to read back the contents that we have on the database. I'm going to go ahead and add one more block here at the end. And as you can see there, I am going to read from that customer's database, and at the very end of it. I'm also going to close that connection. So, I'm going to go ahead and save. I'm going to go ahead and open up the console. Run once again, 'node' at 'test.js'. And I should note before I run it that the randomly generated user will be created again.

So, we will write to the database once more, and then we will read from it as you can see here at the very end. So, let's go ahead and run that. And as you can see, they're coming back. We get two records. The first one that we saw when we were looking at the visual manager of the database. And then the second one that we just created now. So, as you can see here, we were able to create a small node application using the MongoDB package that allowed us to make a connection. We didn't use Express or anything else that we had created already. We created an independent small application, which is always a good idea to be able to verify functionality, to do it in a subset of what you're building simply to make sure that that will work as you think it will. And so, we're able to connect, we're able to write, and we were able to read. So, now it's your turn. Make sure you can do it as well.

Video 20 – Connect Express And MongoDB

It's time to connect our Node Express application to the database. That is, we're going to take all of our Bad Bank code and connect it to our data store. We're going to write a new package, and we're going to pass all of our calls to the database into this new package. And we're going to call this new package a data abstraction layer. Now, this is a good design practice because database-specific code can be separated into a separate package. And if you ever decide to change your datastore, say, in this case, for MongoDB to Postgres or Redis, then you can make all of your changes within that database, within that package without having to change your Node Express application.

And this is a nice separation to have. Now, because we're calling the database and that is an asynchronous mechanism. We are calling a separate server. We don't know when that server is going to respond. It could be slow or it could be down. In this case, then we're going to wrap that call in a 'Promise'. And as you can see here, this is an example of creating new users. And we're going to use the syntax of promises to 'reject' or 'resolve' that call. Now, from our Node Express application, this, this previous one is going to be inside of the DAL, the data abstraction layer. From our node application, we're going to handle that call to a function that returns a Promise. So, when we enter here, as you can see, 'dal.create', and we pass those parameters.

We're then going to use the then, '.then' notation to be able to handle that response and then in turn respond to the front-end, as we usually do with 'res.send'. So, now let's go ahead and move to our editor and write some code. And we're going to go ahead and start here by adding a new file. And we're going to call that file dal, or data abstraction layer, as mentioned. And inside of it, we're going to start off by making some requirements so, to the packages that we need to connect to the database. So, as you can see here, I've required ' 'mongodb' '. Then I am setting the 'url' that we're going to use to connect to this docker container that we're running locally. We're then going to connect to Mongo, and we're going to assign the connection, as you can see here, to that database, to this variable called 'db', so that is accessible throughout all of our subsequent calls.

Next, we're going to add a function to create a user. As you can see here, we're calling the function 'create', it's going to take three parameters. We're then going to wrap it in a 'Promise' as mentioned. And the rest of this code is the one we overviewed when we were looking at the slides. That is, we're going to 'insertOne' document, and we're going to pass in that document that was just created in the line before. So, this is a, this is in a JavaScript object, as you can see there, and we're passing it in as a document. Now, notice well that the 'collection' is going to be called users. And we will have then, as time goes on and more users are created, a collection of users. Now, next, we're going to add one more function, as you can see here in this one is simply going to return all of the users.

You can see here that once again, we're wrapping with a 'Promise'. And the rest of this is simply returning all of the users that exist in the user's collection. Now, note once again that we're using 'reject' and 'resolve', the keywords of the promise to return from the call, to return that content from the database to the calling function, which will be in our Express application. So, now the only thing that is left for us to do is to export this package. As you can see there, we will be exporting those two functions so that we can use them from the Express application. I'm going to go ahead and save. Now, let's go ahead and move into the index file that has our express application. And the very first thing that we're going to do is that we're going to go ahead and add that package that we just wrote.

So, as you can see there, we are requiring ' './dal.js' ' the data abstraction layer. Next, I'm going to go ahead and replace the 'create user account' that was just a place holder with one that called the 'dal'. As you can see there I have 'dal', data abstraction layer '.create'. And then I'm using the '.then' notation to be able to handle that returning promise. And then after that, as mentioned previously, I'm simply handling that response and sending that back to the front-end, which will be invoking this route. I'm now going to go ahead and remove 'login' because we will not be using it, and then add in additional method or 'all accounts'. And that was the second method that I wrote in the 'dal'. And as you can see there, here I am doing the same thing of handling that promise. And I have a response back to the invoking client.

So, now let's go ahead and save that file, open up our console, run our 'index.js', which is our Express application. And as you can see there, we get the notice saying that we have successfully

connected to the database. And we're using now the 'dal' package inside of our Express application, which is the 'index.js' file. So, now let's go ahead and move to the browser and try to invoke those routes. And so, as you can see here, I have our application, but for now, I'm going to go ahead and hit those routes directly on the API. And as you can see here, I am hitting the /account/create/ route, and I'm passing in /bruce/bruce@mit.edu/, and a password of secret. So, I'm going to go ahead and hit return. And as you can see there we get back the echoed user that we just created.

Now, there's an additional ID, and this is coming from the database. And I'm going to go ahead and create one more user. And as you can see there, I have created "peter" now as well. And so, now that we have created a couple of users, let's go ahead and access the path that will return all. And let's see if that works. And as you can see there we have our two users that we have created. We have both "bruce" and "peter". So, now it's your turn as you saw, we simply created the Create Account route and the equivalent function within the DAL, the data abstraction layer. There's a lot more to write to match the functionality of the UI. So, this is now up to you. It is your turn to do that. Write the functions in the DAL, then write the API routes, and make sure that you can perform the operations by testing them in the UI routes within the browser.

Video 21 – Connect Front End

It's now time to connect the front-end. We have created a Mongo database. We ran that in a Docker container. Then we connected our Express application that is serving static content plus running logic on the backend and making a connection to the database. It is now time to write the front-end code that will connect to those routes in our Express application, which in turn are connected to the database. We will do it for the two routes that we have been building. One of them is create account, and the next one is the one that simply gives us everything back in the UI. That is, all of the accounts that had been created. And as you can see here on the left-hand side, we had the original code that we wrote in React that simply stored all the content to the context.

You can see there that the 'function handle()' has three statements. The first one is 'console.log' is simply writes the parameters that we're going to be using to create a new account. It then updates the context. It adds a new user. And the last one is simply a UI instruction. On the right-hand side, we have the code that we will be writing. We will be updating that function with this code. And you can see there that the very first line and the last one, 'console.log' and 'props.setShow' are the exact same ones. And there in the middle, we are making the call to the backend route and simply writing to the console the result of that.

Now, we will have one more, and this corresponds to the component in the UI that simply shows us all of the user accounts that we have. And you can see there that on the left-hand side on the original React code, we're simply reading the context data and then returning it as part of our syntax that was going to be displayed. Now, on the left, on the right-hand side, our new code has

similar components, but it is organized in a different way. We're going to have, go ahead and use data and 'setData' in new state. Then we're going to go ahead and use the 'useEffect' hook to simply call that route and then update the UI. You can see there on that line 'setData' that is going to do something similar by stringifying our JSON that returns from the backend. And then we're going to go hand here on the, on the elements, the tags that we're going to send back to the UI.

You can see here that we have '{data}' and '{data}' simply once again, the 'data' and 'setData' that we have set the 'useState'. And we're going to go ahead and set it once we call that backend. So, we're going to go ahead and make a call to the backend, bring back those users, 'stringify' that into data. And then '{data}' will be evaluated here on the tags that we will show on the front-end. So, let's go ahead and write our code. Let's go ahead and expand their public directory. We will start off with alldata because it's a simpler file. We're no longer going to be using here contexts. I'm going to go ahead and remove it. In its place, we're going to add the code that we had in the slides. So, that is the '[data, setData]' used in 'useState'. We're then going to go ahead and use the 'useEffect()' hook. And as written there, we'll execute only once when the page is loaded.

And that will be enough to be able to set that value for data, which will be distinct stringified value from having called the backend route that returns the users we have created. And the last one, the last block will be what we 'return'. And as you can see there, we have updated that block as well. Now, it is simply the value of data that will be sent to the front-end. So, let's go ahead and save this. And let's see what that looks like in the browser. I'm going to go ahead and enter here localhost:3000. And it looks like I'm not running the server. So, I'm going to go ahead and open up the console. I'm going to go ahead and 'clear', and then enter 'node index.js'. And as you can see there, we have connected to the server and are connected successfully to the database server as well.

So, let's try that address again. And as expected now we get the server serving that content, the UI. So, let's go ahead and click. At this point, we have only updated one of the components. So, let's go ahead and take a look and see if we see anything in All Data. Great. We can see the two users that we had created previously. You can see they're "bruce" and "peter", the rest of the information, the "balance", and you can see there the "_id", which is the identifier for that record within the database, the Mongo database. So, now let's go ahead and add the code for the other component, which is create account. Let's go ahead and open up that file. And as you can see, this is the original create, create JavaScript framework code that we have written in the past.

And now, we simply have to update that handle function. So, I'm going to go ahead and remove the context since we don't need it anymore. Now, I'm going to go ahead and replace this function with the new function that includes the call to the backend. You can see here, we're simply echoing the parameters that we're going to pass to the backend. Then we set the address that we're going to head on the backend, that is the route that we are reaching out to. Then we use 'fetch' to reach out to make that call. And then we write whatever we get back from the server to the console. Then we update the UI simply to show that we have created a new user. So, I'm going to go

ahead and save. Now, let's go ahead and move to the browser. And I'm going to go ahead and open up the Developer Tools to make sure that my page is not caching.

And I'll go ahead and move to the root of the application. Now, I'm going to go ahead and select Create Account. Let me go ahead and reload one more time just to make sure that my code is not cached. And I'm going to go ahead and create a new user. I'm going to go ahead and enter clark as a name. clark@mit.edu, password is secret And I'm going to go ahead and create the account. Now, I'm going to go ahead and take a look at the Console. Let's see if something came back. And as we can see here, we do get a user that was created. We can see the ID from the database and the rest of the information that we just entered. And so, now if we go to AllData, we can see there that we now have three users. So, we're now connecting across all tiers. And if I bring the visual manager tool, we can go ahead and take a look here at the documents that we have and that is the old collection.

So, let me go ahead and refresh here. And we should see the one for users. So, let me go ahead and view that one. And as you can see here, we have three records. The last one being clark, the one before that peter, and the very first one, bruce. So, now it's your turn. We went ahead and connected to the routes for create an AllData, Create Account, and AllData. Now, it's your turn to connect for the rest of the routes for Login, Deposit, Withdraw, and Balance. Make sure you can do the operations on the UI, make sure that you can handle the changes in state, and that you are connecting all the way to the database. If needed, you can go ahead and download the Robo 3T tool so that you can take a look at the database in an easier way.

Video 22 – Final Challenge: Can You Add Authentication?

As a final challenge, consider the case of making your Bad Bank, a good bank. That is adding authentication. So, that when you navigate through the site, you would only be able to see the options that make sense. That is, when you arrived at decide on the landing page, you should only be able to see Create Account and Login. Likewise, once you authenticate, you should be able to then see other options such as Deposit, Withdraw, Balance, and AllData. Now, more than that, you should only be able to see your data. And as you'll see, as we discussed, once it comes to the security considerations, your scope can grow quite a bit.

Now, when it comes to being able to identify who you are and what you should see. We could use authentication. We could use, for example, Firebase authentication. However, that would only, that would only authenticate the front end, what about the routes on the back end? How would you secure those? Well, it turns out that as you might recall from our serverless section, you can pass that token that is given to the client on Firebase and then verify that token on the back end using a Firebase package. So, that is something that you may want to do. Now, once again, the scope of these considerations can grow quite a bit.

And one of the questions you might want to address is if anyone can create an account, is it secure? If it's a bank, most likely you would want to turn some of those down. So, you would have to have some sort of pending scenario. What about roles? Not everybody will be a bank user. Some of them will be simply guest, other will be administrators. And so, those are a little bit of the hints of how large this application could get, simply the security considerations. So, now it's your turn. It's up to you to determine how much of the security scenario you want to provide functionality for and make those design decisions as you take this last challenge.