Working with Simple Components

Video Transcript

Video 1 – Introduction to ATM Exercise

So, we've all been to an ATM machine and withdrawn money. And some of us, I'm sure, have deposited money. I'm not sure I've done it myself. So, in this exercise, we're going to keep track of the state of an ATM machine. So, we're going to need to know whether cash is being withdrawn or deposited. If it's being withdrawn, we need to check with the bank that there's sufficient funds to cover it. So, we're going to develop a simple user interface. It's going to have to web components. One is going to represent the bank, keeps track of the total amount of money that you've got. And the other is going to be the ATM itself. We're going to keep track of state. Is it a deposit or not via use state? And we're going to need to synchronize between these two components so that they understand what state each is in. So, this is a common thing that we need to do in React. It's really useful that you master it and so, we'll give you some exercise in that.

Video 2 – ATM with HTML

So, let's warm up our programming skills by doing a simple example. Here we've got an ATM and this would be the interface, for example, for depositing or withdrawing money. So, here we'll have a simple input element and then a submit element that when we click it, it will update our account. And here we'll write out the account balance. Now. In reality, the ATM interface will be at someplace in your town and will be connected to your Bank. And we'll probably if you're withdrawing, we'll check how much money you have in your account and if you can cover that amount, it will issue you the money. If you're doing a deposit, then it will communicate to the bank how much you've deposited and that will update your account. So, we'll have some separation of concerns here if you like. But we're going to program a very simple example of this.

So, let's take a look at the code. So, here we have the HTML, and below we've got the JavaScript. We've got a simple '< h1 >' tag and then a label. That inside that label we have an input that takes a number and then a submit button, the commits that transaction. And finally, an '< h2 >' tag that updates your account balance or that prints out your account's balance. We have to call back functions that when this input is changed, this 'onChange=' will call the function 'handleChange'. So, we're going to have to write that. And then when we want to commit, will click on this ' "submit" ' and that'll call ' "handleSubmit()" '. So, these are callbacks. So, let's see in our JavaScript where they're defined, so they're defined down here. Now, we'll notice that we've got some global variables hanging around here.

So, one is 'transactionState'. So, that's how much money we're trying to withdraw or deposit. And then 'totalState' will keep our running total of how much we have in our account. And then 'status'. We're going to use that to print out how much money we have. Now, the 'handleChange', will get past the number that you're entering into this input. And it will, as you change it, it will get called again and again. So, 'transactionState' gets updated by that '(value)'. So, when you finished inputting that your 'Number', then 'transactionState' will reflect that, will be that number. Now, when you click to submit, we'll update the 'totalState' of your account and print that out. So, now, if it was a deposit, you'll inform the bank now of that transaction and '${totalState}' would really reside at the bank and be communicated back to us.

But for now, we're just going to roll everything up into this one place. Then finally, we write out, we do 'document.getElementById'. To get this '< h2 >' tag. We get it by its 'id="total" '. So, that ' "total" ' matches there, it's an arbitrary name, can be anything you like. And then we go to the innerHTML and replace it with 'status;'. So, we write out. And notice here we're using the backticks. So, we can insert a variable here and insert its value. So, whatever '${totalState}' is, will get inserted here. And that $ sign in front is just, will get printed out as a dollar sign. But those backticks that different from quotes, remember these are ES6 additions to JavaScript and I really like them. Now, there's some issues here with our architecture that we've got global variables hanging around.

We've got functions that are detached from the element that needs them. So, we'd like to wrap this up a little better. And we'll see in React how to do that. So, we'll implement this next in React with web components. But for now, go through this, makes sure that you understand how it's functioning. So, for example, here, bring up your IDE. And then as you change the number here, notice that handleChange is being called and the numbers being printed out here. So, we can go into the sources in our IDE and put in breakpoints. So, I suggest you do that. And then when you submit, this gets updated. And if we put in, for example, another 10, then our total should go to 15, and it does. So, make sure you understand how this works, then we'll see how to code it in React, where it'll look a little different.

Video 3 – ATM Component

So, let's build a React standalone application. And we're going to simulate an ATM machine where you go and make Deposits or you can make a withdrawal. So, the way that this works is that, for example, our account balance now a zero, but we can Deposit, say, '16' into that. And it pops up, and it says it 'Account total' is '16'. And when we get rid of that, it shows us that it's 16. Now, we can go back, and for example, we can withdraw '-4', and so, now our 'Account Balance' is '12'. So, let's put a $100 in. We can type it in and Submit it. And we've now got a '112'. And of course, we can get rid of that pop-up. It's just there to tell us what's going on. So, that's what we're going to build, eventually. But first, we're going to build one that doesn't keep track of the running total. But we just want to submit something.

So, we want to, for example, put some money in. So, how do we do that? Well, we're going to need a form. And we've got an input here, and 'Submit'. And we're got a label here as well. So, let's go and take a look at our code. So, let's, I've got the 'standalone' here. And we've seen what that looks like. And we need to match up the JSX. Most of this, we don't really worry about its loading libraries for us, for React. But we do need to know the account or rather the. So, we do need to know the file that we're loading. And this one is ' "account.jsx" '. So, let's take a look at that. So, here's the outline we're going to have. This is our parent. We're going to call it the 'Account'. And we're going to have a form that we return. So, here we're going to add that form to the ' "root" '. So, now we're doing a component. So, we're going to do a web component called 'Account'.

Now, inside that, we want to 'return' a 'form'. So, let's take a look, what we want in here. We're going to have a 'form'. We've got the 'Account Balance', and we're going to store that into '{accountState}'. And then, we've got an 'ATMDeposit'. And that is going to be, so here, in there, we're going to have an 'input', and an 'input' of 'type="submit" '. So, this is going. The first one is the number, and when the number changes, we're going to call '{onChange}'. And we'll see that 'onChange' is being passed into us as an argument. So, here are the arguments. And we could put in here props. And, but here we're pulling out what we want from the props. So, if we look at ATM, let's have a look at 'ATM' here.

We'll see that we've got an 'onChange', and we don't have an 'id', so we certainly don't need that one. We've just got 'onChange' as part of the properties in that form, in the 'ATMDeposit'. Okay. Right. So, let's take a look now at what we've got here. We've got a 'label', so that's standard. And everything is going to be inside that 'label'. We've got a 'Deposit:' there for the 'label'. I'm putting a ' "label huge" '. And that's from Bootstrap. I've got Bootstrap loaded. So, we'll give you that. Now, the 'input', this is going to be where we actually enter our number. And then this is the ' "submit" ' button. And wait, that's of 'type="submit" '. So, that's quite straightforward. So, the only thing here is when we change the number that we type in, we want to call whatever this '{onChange}' function is.

And you see down here that 'onChange' is called 'handleChange'. So, we need to write 'handleChange' here. So, whenever we type in a number, we're going to have this event raised. And this is what we're going to do. So, here, let me just, I'm going to get rid of that for the moment. But we're going to pick up the 'event'. So, this is the 'onChange' event. We go to 'target' and 'value'. So, this will be the number. And now, what we're doing is in this 'Account', we're using 'useState'. We're setting it up with '(0)' as the initial value, and the 'accountState' will be kept. And we can change it by using 'setAccountState'. So, here we're using 'setAccountState', and we're getting the 'value' from that input. And we're saving it. So, we're continually saving that. I'm putting in a 'console.log' here.

So, we can see this being called. When we go into the IDE, we can see this 'console.log'. Now, we need to also, when we click on this ' "submit" '. So, here, when we click on submit, we need to catch that. And here on the 'form', we've got '{handleSubmit}' function. So, when the 'form' is submitted. So, when we click on the ' "submit" ', we need to write a 'handleSubmit' function here.

So, there's the outline of it. And what we're going to do is here, we're going to, let's get the total. So, in here we should just write out the 'accountState'. So, this would be present deposit. So, in this one, we can only do one deposit. We're not keeping a running total. But let's just make sure that we understand how that works. Okay, so let's go to here. I'm going to reload that. So, I just brought up the IDE, and I see that it's got the old code loaded.

So, be careful. You need to make sure that you clear browsing data. Let me reload, and just make sure I've got, yes. Now, I've got correct code. So, let's take a look at what happens. We're calling 'handleChange', and we're going to deposit '6'. There's '6'. Okay. Now, let's deposit some more. You'll see that we've still got, we've only got '10'. We deposited '10', and then we say the account balance is '10'. Let's deposit '14'. The account balance is just '14'. So, this is not keeping track of the running total. That we'd like to be able to keep the account balance and add money to it by depositing or withdrawing.

But in this case, it's not. It's just keeping the last transaction. So, what I want you to do is to take the code that I give you. So, that'll be the code here. And we'll call that the starter code for you. And I want you to figure out how do we keep a running total. So, I want this 'accountState' to be updated so that I can make multiple transactions. Now, I won't be able to, if I reload the page, this won't work. It's going to re-zero itself, obviously. But within one session. So, I'd like to be able to deposit one amount, then deposit another, and I should see the sum of them. Okay, so see if you can alter this code so that works.

Video 4 – ATM In React

So, let's now move the code that we developed as pure HTML code into React. So, here I've done a little and I'm going to give you this starter code, but we're going to need to change it to the React style and we're going to do that in two stages. So, first, let me just show you that it works. So, we submit deposit of 3, the account balance is $3. Then we submit 7 on the account balance is $10. So, this code works, and let me just show you your starter code, it's this. And the first thing that the starter code has done is that it's now using, we're going to load this web component called 'ATMDeposit' and that's being defined here. So, this is what we called a functional component. This is a fat function and we're naming it 'ATMDeposit'.

Remember you need to start with a capital letter for React web components. And inside it, we've got what we saw before, the HTML that we had for inputting the deposit, how large we wanted it to be. And then this would handle the changes if we were inputting it. And then when we clicked to submit and have the transaction executed, we click on this '{handleSubmit}'. These are the callback functions that are defined above. So, here are the two callback functions. And we've now got some global variables. Now, we want to move those inside a component. So, let me do that. We're going to focus now on this component. A moment. Don't worry about this one. We're going to get it to work just with the 'ATMDeposit'. So, let me copy those. I'm going to take all of this. I'm going to get rid of that.

We don't need that. And I'm going to put it inside here, inside 'ATMDeposit'. This should still work. Let's take a look. Okay, we start with zero, 3, 8. That still works. So, that's really good. Now, I'm going to make one change to this that I'm going to say that the 'value' of this is equal to ' "Submit" '. Now, what I want to do is I'm going to move a lot of this code into a new web component called 'Account'. So, this will simulate the bank, if you like. It's going to keep track of the total amount of money that we've got etcetera. So, let me now go one step further. We're going to now, instead of calling this web component 'ATMDeposit', we're going to start with the web component 'Account' and that's going to have as a child component, that's going to have the 'ATMDeposit'.

So, now I want to take everything that I've got at the moment in 'ATMDeposit', and I'm going to move it into a parent web component called 'Account'. So, 'Account' now is going to use the 'ATMDeposit'. So, I want to take all of this and move it one step further. Now, I want to take it out of our 'ATMDeposit', and I'm going to put it in here, and now, the 'ATMDeposit', we just need to handle the change there. So, now I need to make sure that when I call 'ATMDeposit', I'm passing in these properties. Here, in 'ATMDeposit'. I need to get hold of the properties, it's called '({onChange})'. And now, I want to change that to '{onChange}'. So, I have to do this because the scope now I can't access '{handleChange}' directly. I'm going to have to have it passed into me by the properties here in the 'handleSubmit'.

We don't need an event. But I do need to make sure I put this in and that should be everything we need. Let's save that. One more change we need to make that we've got this element. And at the moment, 'status;' doesn't have access to that so we need to put it in here. Okay, last one, we need to give this an id so that we can get hold of it as total. So, 'id= 'total' '. Okay, now we can write 'status;' into there and we're good. We don't have 'status;' anywhere else. Okay. Okay, one last critical thing, I need to change this down here to '< Account />'. We need to call the parent and not the child. So now, we should be in good shape. Let's see. Balance is zero. That's quite nice. We're stopping in the right place. Yes, it's 3 add $6 more, it's 9, takeaway 2, 7. So, our ATM machine is working. That's great. Take a look at this code. So, go through the steps, copy it from the video and we're almost there. We're going to make one more change to this and that's going to be to do with the way we store state. And that will then be fully operational in React.

Video 5 – ATM State

So, we got our solution working for stage 1. Now, I want to change it just a little so that it works in the way that React needs things to work. So, let's go and take a look at the code. So, this is the code that we ended up with at the last stage. I'm going to close that the 'ATMDeposit'. We're not going to change that at all. But here we've got a problem in that we're using global variables of state, and React will break if we continue to do that. What we need to do is to make some changes. We need to make sure that we write the 'status;' and we shouldn't be doing it into the DOM. That's also not a good thing to do because the React Shadow DOM doesn't know what we're doing in the actual DOM.

Later on, it will compare its version of the DOM with what's actually in there and will detect the error. But we shouldn't be doing this. Okay, so let's get rid of that statement. So, 'status' now, let's cut that out here. We won't write it there. We'll write it up here. Okay. Now, we've got to keep the state somewhere. So, we want to get rid of this statement here and replace it with React's hook called useState. Now, useState has a fixed format. We can make these variables anything we like. So, we're calling this '[totalState,'. But the convention is now that to set it, we 'setTotalState]' and we call 'React.useState'. And we give it the initial state. So, our account has '(0)' in it to begin with. We'll come back in more detail and go over this.

But now we have a way of storing local state. So, here, we shouldn't do this. What we should do to store the state here. Use 'setTotalState'. And that's equal to or rather we pass it '(totalState +' the increment. So, now that's stored and this will fire a re-render actually. So, now what we do is we rewrite it out here. Now, this initialization will only occur the very first-time 'Account' is rendered. I'm going to put a write statement in here to see when the re-render occurs. And we want to write out, or we'll just say '('Account Rendered')'. That'll do. Let's take a look at this now. We should be working fine. Let's see. Here we are, Account Rendered. Now, let's put $1 in there and we've got $1. And notice the account was re-rendered.

Now, let's put $2 more dollars in there. So, we've got that handler. Now, we submit and the account is re-rendered. Let's put $7 more dollars in and we get a total of 10. So, this is working. We just have to be careful about the order that we do things in. I'm just going to show you what would happen if we tried to render 'status'. So, let's have 'status' here. And I'm just going to put it to ' 'zero'; '. But suppose I try to render 'status' down here. So, I put 'status = `Account Balance $ $ {total state}`', we just reset '{total state}'. But let's see what happens here. I need to put the extra, okay. So, this, I believe will break it. So, let's re-render it. We get the zero rendered. I put 1 in, handleChange, the account re-rendered.

So, it's picking up these changes but then not being rendered. Make sure you understand what's going on. So, what I'm saying is that when we set the useState, it causes this component to be re-rendered. So, that means all of this will get executed again. And we set 'status' to ' "zero"; ' there. And that's the last thing that will be done because these won't be triggered again. So, the last thing is that we will have a render. So, what we need to do is to make sure we take the 'status' and put it up here. '`Account Balance $ ${totalState}`; '. Down here we don't need to do anything with the 'status'. Let's just check that that runs. Account balance is 0. Account balance is 1. Account balance 5, 7. Okay. So, this is running in React, and we're beginning to see some of the complexities that we need to be aware of.

Video 6 – ATM Deposit and Cashback Buttons

So, let's polish our ATM machine so that it looks like this. So, what we'd like to do is handle both 'Deposit' and 'Cash Back'. And we would like to be able to 'Submit'. So, for example, 'Deposit' here. We add in '15', and we have a total of '24'. Now, we choose 'Cash Back'. It tells us we're taking cash back. Let's get back '10', and we end up with '14'. So, let's take a look how we handle this. We've added two buttons. We've still got the input and the 'Submit' as we had it before. And we've still got the total. So, let's take a look at the code and how we change it. So, here's the code that we ended up with last time. And what we want to do is we want to put in some buttons. So, let's add those. So, we're going to add a '< button >'.

Okay. And we need to have that labeled. So, let's suppose this is the deposit one. So, we'll just put 'Deposit' there. And we'll certainly need an 'onClick'. So, when we click on it, we want to choose that this transaction is a deposit. Let's create a variable, 'isDeposit'. So, up here, let's use 'useState' to create '[isDeposit, setIsDeposit]'. And this is going to be true or false. 'useState', and let's set it to 'true' that we're doing a deposit. That's the default. Okay. So, now we've got 'isDeposit', and on the 'onClick', we can fire a function. So, we need to put it in brackets. And the function '=>', and we want to 'setIsDeposit' to '(true)'. That seems reasonable. And now, let's put another '< button >'. And this time we want to 'setIsDeposit' to '(false)'. And let's call this 'Cash Back'.

So, it's withdrawal, basically, cash back. So, when we click either of these, we're going to set 'isDeposit' up in 'useState' here. Now, let's handle what do we do. Well, when we submit. Now, it depends, what is, is it a deposit? Is that true? Well, if it is, let's see. We want to let, let's call it 'newTotal ='. We're going to check 'isDeposit'. And if it's true, the first choice is we're going to set the 'newTotal', whatever we've got already. So, that's a 'totalState + deposit'. Yeah, we're, remember we're putting that into a local variable called 'deposit'. And now, if it's not, then we want 'totalState - deposit'. That looks good. And now, we don't need this line. And we're setting 'setTotalState'. That looks good. Now, let's see up here, do we need to change anything up here?

Yes. We need to know whether it's a deposit, and I've already put that in there. So, when we create 'ATMDeposit', we need to pass that in. So, here I've added this, 'isDeposit', and we get the value of whether it's a deposit or not. So, that's up here. We'll know if it is deposit or not. Now, we need to write out. I've added this, and I've put in an array 'choice', whether it's a deposit or not. And I'm getting the 'Number' of '!isDeposit'. So, it's going to choose either zero or one, true or false. I'm casting that to a number. Let's see if that works. Yeah, it looks like we're doing okay. We've got 'Deposit', 'Cash Back'.

Choose 'Deposit', stays as deposit. Choose 'Cash Back'. Okay, so let's deposit, let's deposit '4'. And let's get cash back of '2'. Seems to be working. Let's do a couple of deposits. Deposit '9'. Let's keep another deposit. '13', another deposit, '46'. 'Cash Back'. Let's get cash back of '59'. We're bankrupt. We should have stopped that. But that's fairly easy. Why don't you do that? Why don't you make sure that we refuse any transactions if the account balance can't cover it? Okay, so that's our simple ATM machine interface. And I think we've learned a lot about how to pass data around. As I say, why don't you implement it so you stop any negative balances, okay? So, that if cashback exceeds the account balance, it shouldn't allow it.

Video 7 – Fetching Data Exercise Overview

One common operation is to go and get data. And so, fetching data from a URL is often used in React. We're going to cover that in this exercise. We're going to fetch data first from the local machine, a file on the local machine. But then we're going to go out to a URL and get data from a website called Hacker News. It's going to return to us lots of data and we're going to need to handle that. Going. And getting data is often called a side effect in web components. So, it doesn't actually affect what's rendered. I mean, ultimately it well, obviously, we're not getting the data for no purpose at all. It's ultimately probably going to be used and rendered. But the actual process of getting data is part of the life cycle and can be controlled in useEffect. So, we're going to embed fetching data in useEffect. We're going to keep track of state and in fact of multiple states. We're going to use, useReduce to do that, and this is also going to use, obviously, useState as well. So, we're going to be looking at controlling the rendering and handling the life cycle of components. I hope you enjoy this. I've enjoyed making it and we'll be talking to you later.

Video 8 – Fetch Data Via A URL With UseEffect

So, we've seen how to render lists in React. And we looked at the example where we were getting data from a local variable. Now, let's go and get the data from the web. So, we're going to hit a URL and pull in data. Let's take a look at the code. So, the data that we're going to get is going to be in our local file system. And it's kind of consist of ' "Apples:" ', ' "Oranges:" ', et cetera. And we've got a few more attributes here in our object. That we've got the ' "country":' where the produce was grown. We've got the price. We've got the number of stock items. We've got some inflammation of the ' "url" ', and ' "objectID" '. So, the data is a little more complicated. And as we go further out on the web, we'll see that this will grow. But, for now, this is enough, and we see that it's called. It's an object, and inside the object is an array called ' "hits":'.

Okay, so this is the data we're going to get. Let's take a look at our code. So, here's our app that we're going to load. It's going to return a '< Container >'. Let's see if I can collapse this a bit. So, this is the main thing that we're returning, and then we're rendering the 'App' and the 'DOM'. So, that's fairly straightforward. Now, we can take a quick look at where it's rendered. So, once we've got the data back, we'll see that it's going to be rendered down here where we're going to get an unordered list '< ul >'. And we're going to put into that, we're going to use 'data.hits'. So, remember our structure was that we had hits in our data, and we're going to 'map' that. And we're going to have an unordered list '< ul >', where we're going to write out '{item.url}' and the '{item.title}'.

Now, we may want to do a query and set that we go to another website. For now, we're going to hardwire this. And we're going to use up here, we'll have 'localhost:8080' and the 'date'. So, let's take a look where we make the request on the web. We're going to make it inside 'useEffect()'. Now, let's clean up. We're repeating all these Reacts here. Let's clean these up a little. So, I'm going to delete those. And since I've done that, I need to make sure we have access to them. And we can do that with 'useState' and 'useEffect'. And we'll get those from React. So, we'll use destructuring to go and get. Okay, so now we've got access to 'useState' and 'useEffect'. Let's take a look at 'useEffect', this is new. So, we've seen that whenever we change a 'useState', so when we do 'setState', for example, or 'setData' here, we're going to get a re-render of this component.

Now, anything inside 'useEffect' doesn't cause a re-render. Let me just collapse this down a little. So, 'useEffect' takes a function. So, here's the function. And it takes another argument. Now, depending on what we specify for this argument, it may or may not be re-rendered. But 'useEffect' itself, like this, will only cause a re-render if this variable changes. So, it's kind of as if 'useEffect' is watching whatever is written in here. And we can put a list of things in here. And if any of them change, then 'useEffect' will cause a re-render. But if we just, for example, if we don't want it to re-render all, we can just put a blank array in there. So, I can put that because we're not going to need that. Now, the 'fetchData();', we're going to use 'axios'. So, in the standalone, I'm loading 'axios'. It's a library that helps fetch data.

We could use the built-in JavaScript fetch, that's now part of the standard, 'axios' is perhaps a little easier. So, what we're saying is, we're going to make an asynchronous call. This is an asynchronous function. And here is where we make the call 'axios(url);' goes out to whatever a URL we've specified, and we'll await it bringing data back. We're setting a variable here, 'setIsLoading' to '(true)'. So, 'isLoading', is, yeah, 'setIsLoading' and was '(true)'. So, 'isLoading' is going to be 'true'. And then once it's loaded here, when we, when the promise comes back and we satisfy that. And so, we now have result; it's come back. Then we can set 'result.data'. So, we happen to know that inside 'result' is something called 'data', that's just part of the standard. And it'll either be now or it will have data. And here we're using 'setData'.

And then we're setting 'setIsLoading' to '(false)' because it's loaded. We're calling that 'fetchData();'. So, here we're just defining 'fetchData();'. Now, with firing it. So, we'll go and fetch the data. Okay, and once we've got the data, then we can get access to it here with 'useState'. And we know it has an object called 'hits' that happens to be an array, and then we can map it and render it out. So, that's how this works. And as you saw, let me save this. And we can go out and we can rerun it. And you see it goes out and fetches the data. So, we're going to improve on this and build it out more. But essentially, the new part is this 'useEffect'.

So, we see that 'useEffect' is important because it allows us to take actions while the object is being loaded, while it's actually is loaded. And various side effects are happening, like going out and getting data, and pulling back that data and storing it. And then finally, when our component is unloaded that we can clean up. As I said, that useEffect itself does not cause any re-renderings. However, if we're doing things inside of 'useEffect' that are using you 'useState' like here, we're setting the data that will cause a re-render of the component. So, we'll go into more detail on how to use 'useEffect', and how to build out this app so that we can handle getting data.

Video 9 – Render Data From URL

So, we saw how we could use a URL and fetch data from it and render that data. Let's take a look at the code because we need to understand how various parts are being used. So, where rendering our 'App'. So, this function is going to be fired by React. And it's going to be fired, not only once but many times, and we need to understand when it's rendering. So, I've put a 'console.log("Rendering App");' here. So, we can see when this is being rendered. Now, we're using useEffect to go out and get the data. This getting of data is called a side effect because it will affect the rendering ultimately, but actually getting the data doesn't. It's only when we actually render it that it's necessary to re-render.

So, we put it in a 'useEffect', and 'useEffect' takes a function. So, here's the function. It goes from there to there. Now, it's a strange function in a way because inside it we have a function 'fetchData' and it's being fired. So, it kind of fires itself. So, that's the first argument. But the second argument is this little blank array. Now, that's being put in there because we want to stop useEffect being called multiple times. So, we're saying to useEffect actually, track this variable or this number of variables in this array. So, we could, for example, track the URL. And if the URL changes, now useEffect will change. Now, what I'm doing here is stopping useEffect being called multiple times at all. So, let's take a look now at when we run this.

So, here we are, and I'll show the console. So, I'll reload this. Notice that it renders the App. Then it executes what we had in useEffect. So, it fetched the data. Now, that data was rendered in some elements, and Rendering App needed to be called again. So, then it actually rendered this data. So, it started Rendering the App, saw that it needed to fetch data in useEffect, fetched the data, and then re-rendered. Let me just show you what happens if we don't stop useEffect. So, useEffect now, I'm not putting in that argument. So, let's rerun it. So, you see that it's just fetching data all the time. Let's stop this.

Let me put that back in and reload this. Okay, now we've got control again. So, it's very important to understand what's happening here. So, we fire this function in useEffect, and that function has another function inside defined that it fires itself. Then we're done with that, and we've got our data rendered. Okay, so we saw that. Now, we're going to look in more detail at what happens when we have different stages inside useEffect. So, for example, we're saying fetching data here, but we'd like to know when it's got the data. Do we do something? If it fails to get the data, we need to do something else.

Video 10 – Fetch Data from Hacker News

So, let's take a look now at an update to this app. Here, we're going out to Hacker News and getting MIT's news events. Now, we see here that we've fetched in data and we've got a number of renderings. Let's go take a look at what's happening now. Let's take a look. This is our app function and we're keeping track now of data that's coming back, we're keeping track of the query. So, it's initially set to MIT as we saw, but we can reset that query so we can query different parts of Hacker News. Now, we're keeping track of if there's an error and we're also keeping track of the '[url]'. Let's take a look at 'useEffect'. So, here's 'useEffect'.

Now, remember it takes a function. So, here, let's get the function, there it is. And we're specifying now that we want useEffect to track the '[url]', if the '[url]' changes, then 'useEffect' is going to change. It's going to fire. This will fire, and it will go and fetch data again because now we're hitting a different '[url]'. Yep, we're going to hit instead of MIT, we're going to hit some other part of Hacker News. So, that's the first thing. So now, when '[url]' changes, we're going to get new re-renders. Lets us see that. So, here we are. Let's, we've rendered four times, and we fetched data just once.

Now, I'm going to delete the MIT, watch that we render every time I input. Notice there's a little counter here. If it knows that I'm updating the same thing, it just starts counting here. So, instead of MIT, Let's see what Harvard has got. Say you'll see now there's 9 re-renders there, and let's search now in Harvard. Now, we go through our whole update again. So, this is Harvard's news. Okay, so we need to really understand what's going on in our app that we're having things fired in useEffect because things are changing elsewhere in our app. Okay, so that's fetching the data.

We're not at least fetching it every time that when we, remember when we had this as blank, it would go out every time and useEffect because it would re-render, call useEffect, and useEffect would cause another re-render because things elsewhere we're changing. Let's go back and we see now it's getting kind of crowded up in here, that in this useEffect there's a lot of states changing here, it's loading. Here, it has loaded, and we've got the data back. And possibly we might have an error. So, these are lifecycle events. And now, we'd like to handle them better. And we'll take a look next at grouping some of these together, and using useReducer.

Video 11 – UseReducer For Fetching Data

So, we've seen that we can go to a remote URL and retrieve data. So, here we're going to hack a news, and we're requesting data about 'MIT'. We can change that, and for example, ask about 'React'. And it goes and gets the data. But let's go and look at the code because the code was getting fairly bloated. So, here's the code for 'getData2.jsx' that we were using before. And we see that our 'App()' function is getting quite bloated in the sense that we've got a lot of states here that we're keeping separately. And we saw how to use, 'useEffect'. So, 'useEffect' to handle 'Lifecycle Events'. So, inside here, we're waiting for the data to come back. If it comes back, then we have success and we set the data. We use a 'useState' to set the data. And if there's an '(error)', we also catch that.

So, 'useEffect' is handling the 'Lifecycle Events' for what we call side effects. So, these are not directly related to rendering. Now, one thing to notice is that 'useEffect' fires getting the data in a strange way, in the sense, that if 'useEffect' is called, then here it fires this function which we've just defined. So, we don't fire it directly. Every time 'useEffect' is called, it will fire. And 'useEffect' will be called, remember, if the 'url'changes. So, now we have another step backwards, in the sense, that all we have to do is change the URL, and the new data is going to be fetched. And because we're setting the data, that will cause the 'App' to be re-rendered. And we're stopping it being re-rendered.

If we haven't changed the URL, that 'useEffect' won't be called. That, remember that 'useEffect' is called after a rendering or a re-rendering. And that re-render checks to see if this 'url' has changed. In fact, we can put any number of variables in there, and it will check whether they've been changed. If they have 'useEffect' will be called. If they haven't, 'useEffect' won't be called. Now, let's see how we can clean this up. So, let's go to 'getData3.jsx'. Now, 'getData3.jsx', you will notice that the 'App' now is quite small. Here's the 'App()', and here's its 'return()', where it's rendering everything, the fragment. Let's collapse this down. Now, what we've done is we've written what's called a custom hook.

So, above, we've defined 'useDataApi'. It takes two arguments here, that it takes the new query and it takes the data, which is going to be 'data.hits' as an empty array. So, this is initialization. So, it's only going to be called once. Now, this returns an array of, this is states. So, here we've got a number of states which is an object. And then we've got, it returns to us also a function. It turns out that that function will set the URL. So, down here, 'doFetch', sets this URL. And we know that setting a 'useState' causes everything to be re-rendered. The 'App' will be re-rendered. And we know that here, this is actually setting the URL. And we know that 'useEffect' has been monitoring the URL. It will monitor that URL and that'll cause this to be re-fired Now, let's take a look at the details of this 'useDataApi', this custom hook.

So, we said that it took two variables. We can have it take as many as we want. But here we're using these to initialize the state, and in fact, initialize the data state down here. Now, we're using a React function called 'useRenderer'. So, it's like 'useState', 'useEffect'. We've got 'useRenderer'. And we supply to it a function and an initial state. What it's going to do is bundle together a number of states. So, we don't need the separate 'useState's. We're going to handle 'isLoading:', 'isError:', and we're going to handle the 'data' all in one function. So, ultimately, what's going to be passed back by 'useReducer' is this state. And it's going to be modified. And also a function called 'dispatch', which we'll see is going to be used in 'useEffect'.

But let's take a look first at the 'dataFetchReducer'. So, let's expand that out. So, this is the function that we're passing into our 'useReducer'. And what it does, it takes the 'state,', and it takes an 'action'. And inside that 'action', needs to be an object with a 'type' because we're going to use '(action.type)', and check it against if that's ' "FETCH_INIT": '. We're going to do, we're going to 'return' this state. So, this is the three states that are being modified. So, the '...' notation is spreading 'state,'. And then these are being overwritten. So, state has got three variables, and we're overriding two of them. Down here, we're overriding more three. We're adding the 'data:' and the 'action.payload'. And down here, we're again, we're just overriding two of them. So, this then will be used in 'useEffect'.

And it returns 'dispatch', and now we use it and pass in that 'type:' that we need down here that this data fetch needs is this type. And we can also pass in. Here we've got 'payload:', which is a 'result.data'. So, 'dispatch' always takes an object, and it can have one or two attributes. It must have a 'type', but it can also have a 'payload'. And we saw down below here that 'action.payload' was being used, and 'data:' was being set to that. So, it seems like a little tricky, but you'll get used to using it. And the main thing to realize is our goal here is to separate out the issues so that the 'App' keeps track of the things that it needs. But now, it can offload this custom hook and we can use this custom hook in other components. So, that's the idea. Take a look at how it works. I'd like you to track through exactly what's going on when it's running by going into the IDE, and go through and put breakpoints in, to see what's happening. Also, put write statements in, do console.logs in the code that we're using. So, in here, put 'console.logs' in the App, and also in useEffect to see when it's being called. So, that you get a good understanding of exactly what React is doing.

Video 12 – Paginate Data

So, we saw last time that we retrieved data from the Hacker News API. And we had quite a lot of items retrieved. What we'd like to do now is paginate them. So, we'd like to choose how many we have per page. And then we can step through the pages using these buttons so that's stepping forward and we can step back. So, we'd like to add this to our user interface. Let's take a look at the code. So, we remember last time we had our own custom hook and it was called 'useDataApi'. And that hook was used to manage a number of state variables. Now, we're going to add to this a 'Pagination' web component. And we've got a couple of helper functions. One is called 'paginate' and one is called 'range'. Now, 'range' we call it, and we call it with a '(start,' and an 'end)'.

And it'll return an array that has, for example, 1, 2, 3 in it. That if it's an array with three elements, then we're going to initialize them to 1, 2, 3. But we can give the start. So, we could start at 2, for example, and have 2, 3, 4. So, this is the little helper function that we're going to need. So, take a look at that because you've seen the dot notation. So, we get an 'Array' and then '.fill' is a function that we call, and then that returns an 'Array' and we call '.map' on that. So, we get returned to us, map returns to us another 'Array' that's been modified. So, that's a neat little one. You can do this in several ways with a for loop, for example, but this is quite neat. It actually returns it in basically one line of code.

And we're going to need something to 'paginate' that we're going to feed it, the number of items that we've got in total. So, suppose we've got 24 items and we've got a 'pageSize' of 10. We're going to put 10 items per page, then this given the page number that we're on will return to us the items for that page. So, they could be through 0 to 10, or 10 to 20, or 20 to 24. So, this function will do that for us. Now, let's take a look at the 'App'. What we're going to do is we're going to add to our '< Fragment >' here. At the very bottom, we're going to put our 'Pagination'. And we're going to pass in the data that we've retrieved. We're going to pass in the 'pageSize'. I'm hard-coding it here to be 10, but we could, for example, have an input that allows the user to choose the 'pageSize'.

And we're going to call 'onPageChange'. We're going to have a function that handles the page change. Let's just take a look at that function. So, when we click on one of those blue buttons, we're going to pick up which button it was going to get '(e.target.textContent)'. And we're going to convert that to a number. And so, if the button had number 2 on it, we'd set the current page number 2. So, the button that was clicked on, that's going to be page that we're going to go to. Okay, so that's pretty straightforward. Let's take a look at 'Pagination'. So, here we're going to pass in the list of items that we've retrieved. We're going to pass in the 'pageSize', and then that's the handler that we just saw. So, here's 'Pagination'. We're going to get the number of items that have been returned. If they're less or equal to 1, then we don't need any buttons so this will return.

We're not going to do any 'Pagination'. We're not going to put the buttons up if we don't have more than one item returned. Now, we're going to calculate the 'pageSize', the total number of pages we need We're going to get the item's length and defined by 'pageSize'. And then we're going to round up. So, for example, if we've got 11 items, we're going to divide 11 by 'pageSize'. It's going to come to 1.1, 'pageSize' is 10. And we want to round that up to 2. So, we're going to get 2 pages then. Now, we get an 'Array' that's got the numbers 1, 2 in it if we've got 2 pages. And now, what we're going to do is we're going to return those two blue buttons. So, this is where they're created. And now, we put them into an unordered '{list}'.

So, we've got buttons in the '{list}' and we 'return' that between two < divs >, just '< nav>'. So, that's basically it. Let's just go and take a look. Now, at this, we need to pass in these properties. So, we've lined those up. Now, remember one of the things is that we're going to pick those properties up in the 'Pagination' web component. So, usually, we don't have to match these arguments. But here we do because we're using decomposition here, that whatever object is being passed to us, and that's usually called props.

We're going to pick out props.items, prop.pageSize, and props.onPageChange.Okay, so that's one little thing to watch out for. Okay, so that's the code. And as an exercise, why don't you put in an input so we can input the number of items per page, okay, that we want they'll set the 'pageSize'. Put in an input, I don't mind where you put it, but we should be able to set the 'pageSize' so that, for example, we can have 5 items instead of 10. As I say, I hard-coded. Okay, so, that's been a long journey with getting data, but now I think we've mastered it.