WEEK 4 - REACT FORMS, FLOW ARCHITECTURE, AND INTRO TO REDUX

# WELCOME TO WEEK 4

Welcome to Week 4. In this week, you will first add to your growing knowledge of JavaScript with an introduction to several new programming concepts: the ternary operator, computed property names, and the spread syntax, as well as regular expressions.

Then you will learn about two different ways to use React with forms to gather input from the user. FIrst, you'll learn about Controlled Forms by creating a Contact Form. And you'll learn how to set up form validation as well. Then you will learn about Uncontrolled Forms by creating a Login form. With this, you'll also learn how to set up a Reactstrap Modal.

After that, you will learn about the MVC and Flux design architectures, as a precursor for learning about Redux, a JavaScript state management library commonly used with React. You will begin to incorporate Redux into your React application. And you'll convert your Contact form to use Redux to manage its form input state.

This week and every week, you will have multiple Code Challenges, and a Quiz, and you'll end the week with a workshop with your instructor and classmates.

Remember to stay focused, study every day, and don't forget the 20 minute rule.

Happy learning!

## 

# JavaScript - Ternary Operator

The ternary operator is also known as the conditional operator. It's called ternary, meaning 3, because it's the only JavaScript operator that takes 3 operands. The first operand is a condition to be evaluated, followed by a question mark. The second is an expression to execute if the condition is truthy, followed by a colon. The third and final operand is an expression to execute if the condition is falsy.

This operator is used as a shortcut for the *if* statement.

Let's look at an example in the console.

First, let's look at an *if* statement. Let's say we have a lightswitch object with a property that keeps track of the lightswitch's on or off state, and a method to turn it off or on.

const lightswitch = {

switchState: "on",

flipSwitch() {

if (this.switchState === "on") {

this.switchState = "off";

} else {

this.switchState = "on";

}

}

};

You can see that the current switchState is on : lightswitch

If I use the lightswithc.flipSwitch() method, then I can see that the switchState has changed to off.

And if I call it again, it'll switch back to on.

Let's try this now with the ternary operator. We can overwrite the switchState method:

lightswitch.flipSwitch = function() {

(this.switchState === "on") ? this.switchState = "off" : this.switchState = "on";

}

The first operand is the condition to evaluate, just like in an if statement. So we're evaluating here if this.switchState equals on. Then we have a question mark, and this code after it will execute if the condition in the first operand evaluates as truthy. If it evaluates as alsy, then this third operand, after this colon, will execute instead. So if this, then this, else this.

Let's test that out. The switchstate is currently this. I'll run lightswitch.flipSwitch again. Now you can see it has flipped to this. One more time - and you can see it's flipped back. So you can see it's working the same way as when we used the if statement.

We'll look at one more example. Here's a function.

function canVote(age) {

if (age >= 18) {  
 return true;

}

return false;

}

This function takes an age argument and returns true if it's 18 or over, and false if it's under 18.

Based on the previous example, you might think you can rewrite this with the ternary operator like so:

The condition, question mark, code to execute if condition is truthy, colon, code for falsy..

function canVote(age) {

(age >= 18) ? return true : return false;

}

This will not work. You cannot use the return statement within a ternary operation, you will just get this error. Because return is a statement, not an expression. However, you can take the result of the ternary operation and return that, like this:

function canVote(age) {

return (age >= 18) ? true: false;

}

[demo this working]

So that's a couple different ways you can use the ternary operator. Try the Code Challenge for ternary operators to practice this new syntax for yourself.

# 

# JavaScript: ES6 Computed Property Names and Spread Syntax

Computed property names and the spread syntax are both new to JavaScript in ES6. Though they are presented here in a single lesson, and they both involve working with objects, they are two separate concepts. I will first discuss computed property names.

Computed property names allow you to pass in variables for use as property names when initializing an object.

Let's look at an example in the console. Please follow along with me and type in the same commands yourself. We'll start by declaring a variable, myProperty, and storing a string inside it.

Then we'll declare an object, myObject, and we'll give it a property. For the property identifier, we'll use square braces, then that variable we declared earlier. And we'll give it a value.

Now if we look inside myObject, you can see that the property identifier was created using the string stored in myProperty.

const myProperty = "foo";

const myObject = { [myProperty]: "This is a test" }

So you can see how computed property names allow you to create objects more dynamically. You may have some code to create an object that doesn't know in advance what its property names will be, and will create them from variables that are passed to it.

JavaScript Spread syntax

The JavaScript spread syntax can be used with both arrays in objects. In both cases, they allow you to pull out a copy of what's inside the array or object. While there are various reasons to need to do this, in this course, it will be so that we can create new arrays and objects. Later, you will see how this can be used in Redux.

For example, here's an example with arrays. Let's say you have two arrays:

const landVehicles = ["sedan", "truck", "van"]

const waterVehicles = ["canoe", "sailboat", "submarine"]

And you want to combine them together into a new array. You can do that like this, using the spread syntax, which is these ellipses, the sets of three dots; - so here, the items inside of these two arrays are spread out, then recombined into a new array by putting square brackets around both of these to create the new array.

const vehicles = [...landVehicles , ...boatTypes];

And it doesn't always have to be two arrays. It could be an array and a string that you want to add to the array:

const moreLandVehicles = [...landVehicles, "tank"];

In fact, you could spread and recombine multiple arrays, along with any other kind of data that you can normally store in an array:

const mixNMatch = [...landVehicles, "banana", 3, 100, { someObjProperty: "someObjPropertyValue" }, ...waterVehicles]

As you can see, the contents of the two spread arrays were inserted into this new array along with the other string, number, and object items.

We can also use the spread syntax with objects, and it gives us a very easy way to take multiple objects and combine them together.

For example, let's say we have two objects:

const objOne = { color: "blue", height: 12 }

const objTwo = { material: "ceramic", width: 6 }

What the spread syntax does is pull out the key-value pairs of these objects, using three dots, so that you can recombine them into a new object, like this:

const objOneAndTwoCombined = { ...objOne, ...objTwo }

What if you try to combine two objects that have properties in common? For example:

const objThree = { color: "red", width: 7 }

Remember objOne? Let's combine objOne and objThree together using the spread syntax:

const objOneAndThreeCombined= { ...objOne, ...objThree }

objOneAndThreeCombined - so you can see that the color of objThree overwrote the color of objOne.

The only reason it did that is because of the order in which they were combined. If we switched them:

const objThreeAndOneCombined = { ...objThree, objOne }

Then here, you can see that the color of objOne overwrote the color of objThree.

You can also use the spread syntax to create a new object from an existing object, while updating one or more of its properties:

const objOneUpdatedHeight = { ...objOne, height: 300}

You see here, we spread out the properties from objOne, then updated the height, and put it all together into a new object, objOneUpdatedHeight.

You will encounter both computed property names and the spread syntax in this week of your React course, and hopefully this has prepared you to do so.

# OVERVIEW - CONTROLLED FORMS

You're familiar with forms from HTML. They're the standard way of obtaining input from users, whether via input fields, textarea fields, or selects.

SLIDE:

HTML Forms:

* Obtain input from users
* Input fields (with type text, email, date, checkbox, radio, etc)
* Textarea (for multi-line text fields)
* Select (for a drop-down with multiple options)

HTML form elements such as input, textarea, and select already naturally maintain some internal state, such as whether a checkbox is checked, the value, and so forth. This internal state is completely independent of React.

Controlled forms:

Controlled forms in React are a way of setting up an HTML form so that the form values are directly tied to the state of the React component that the form is in. In this approach, React reaches in and takes control of the state of your form elements, becoming what's called the "single source of truth" for the form state. This way, the HTML form and React won't ever conflict with each other on what the true state is; the form gets its true state directly from React.

That means any changes in the form inputs will be immediately reflected in the component's state. Then the component is able to easily respond to those changes using JavaScript, such as by validating the input and showing error messages.

Let me show you a basic example of a controlled form in Codepen.

Here, I have a form with just a single input.

You can see that the value of this input is set to a property of the component's state, which is initialized as an empty string in the constructor.

Then the onChange event handler is used to track any changes to the input's value, and we give it a class method, this.handleChange, to run when any changes happen.

Then here, we define that method. This 'event' object is passed in by default in JavaScript from the onChange event handler and it has some built-in properties that can be useful. The property event.target, for example, will always point to the element that the event originated from, so in this case, event.target is this input element. That means event.target.value is going to be whatever is in the text field. So using setState, the component state is changed to whatever is now in the text field. Then just to demonstrate, I have this text down here that then accesses the state and displays it.

The last thing I want to point out is this use of the bind method, which is a built-in JavaScript method. Whenever you need to use the "this" keyword inside an event handler method, you need to make sure that the "this" keyword is bound properly, and this bind method call does that. If I removed this binding, then this method would no longer work, because it wouldn't have the proper reference for "this".

Another way to make sure that the *this* keyword is bound properly is to use an arrow function for the method, then it will work without calling the bind method.

This Codepen is linked as an example in the written instructions, so you can look at it and experiment with it if you like. In the following exercises, you will be adding controlled form elements to your React website.

<https://codepen.io/minae/pen/KKKybVW?editors=0010>

# 

# EXERCISE - CONTROLLED FORMS

* Changed:
  + ContactComponent.js

In this exercise, we will be adding a contact form to the Contact component.

We will start by importing the necessary components into ContactComponent.js:

Component from React

And from reactstrap, we'll bring in Button, Form, FormGroup, Label, Input, and Col.

Then we will change the Contact component into a class component. So we'll replace the function keyword with the class keyword, then use extends Component.

Then we'll add a constructor, and in its state, we'll initialize state properties that will correspond to the contact form elements: firstname, lastname, telnum, and so on.

Then we'll set up two class methods, one to handle changes in form elements, and one to handle form submission.

The first one, to handle changes in form elements, we'll define as handleInputChange.

We'll create a couple of variables in here:

const target = event.target;

const name = target.name;

These are just making it easier for us to reference the properties event.target and event.target.name.

We'll also declare this variable called value. Here, we're using the ternary operator you learned about before. And we're saying, if the type of the form element that triggered event is a checkbox, then we get the value from the target's checked attribute. The checked attribute is a Boolean attribute, it can hold either true or false. And if it's not a checkbox, then we get the value from the target's value.

Then here, we use setState, and we use computed property names, which you also learned about earlier, to set the property based on the target's name. And depending on which form element triggered this event, that could be firstName, or lastName, or email, et cetera. Then for this property's value, we use what's stored in this value variable from above.

Then we need to bind the reference to the *this* keyword for this method in the class constructor, and we do that like this. This binding makes it so that we can use the *this* keyword inside handleInputChange and have it point to the correct object.

Next, we'll set up the handleSubmit method. For now, sWe will just set up this method to log the current state to the console. Console.log expects a string, not an object, and there's a handy global method that will help us make a string from a JavaScript object, called JSON.stringify.

And just for good measure, let's add an alert, so we'll get this in both the console and as an alert.

Then there's one more thing we need to do in this method. When you submit a form, it usually refreshes the entire page, and we don't want to do that. So we'll stop that here with this line: event.preventDefault();

Then we'll also need to bind this event handler in the constructor.

Next, we need to wrap our return statement in a render() method since we've switched from a functional component to a class component.

Then down here, we'll now be adding the contact form. The code for this form is in the written instructions for this exercise. Go ahead and copy and paste it here.

For the most part, this form is very similar to the form you had in Bootstrap. We're using Reactstrap Col component inside this form, which is actually equivalent to writing it this way with the div className="col". So this bit of code here, <Col md={10}> is equivalent to this one up here, <div classname="col-md-10". We could use them interchangeably, they render the same way into the browser DOM.

And it's the same here in the Label component, this md=2 in curly braces will render in the browser as class="col-md-2".

And remember, in HTML this label attribute would be the word 'for', but since the word for is reserved for *for* loops in JavaScript, in this JSX, we have to use the attribute htmlFor.

You'll notice there's a couple of Col components that look different, instead of just a number after md=, we're giving it an object that includes a size and an offset. So to define that object, we have to use a second set of curly braces. This allows us to set both the responsive column size and offset at the same time. This will render out like this: [annotation] <div class="col-md-4 offset-md-2">

and this one will render out like this: <div class="col-md-10 offset-md-2">

Then in the Input components from Reactstrap, we'll set up the usual type, id, name, and placeholder

Now, to make this a controlled form, we'll need to set the value of each input to {this.state. then the state property that we defined before to hold the state for this input. The exception here is the checkbox input, for which we use the checked attribute instead of value.

Each input component also gets an onChange event handler, set to the method handleInputChange that we created before.

And the entire form will also get an onSubmit event handler, set to the method handleSubmit.

We can now test this in the browser. Go ahead and start your app and make sure that the contact form looks like this.

Now I should be able to update these fields, then submit the form, and get an alert that shows the value of all the form elements.

Make sure that your contact form is working like this before you continue on to the next exercise. And don't forget, if you're using Git, this is a good time for a commit.

# EXERCISE - CONTROLLED FORM VALIDATION

* Changed:
  + ContactComponent.js

In this exercise, we will implement client-side form validation with the Contact form.

In ContactComponent.js, first import the FormFeedback component from reactstrap.

This will help us with showing error messages to the user.

In the Contact component's state, we'll need to set up a new property named touched. This property will keep track of whether these four fields, firstname, lastname, phonenum, and email, have been touched or not.

That is, has the user entered that field or not. And we'll track that with a boolean true or false. We will initialize each as false. This way, we know that if a field has not been touched, then we don't need to run validation on that field yet.

So how do we know when a user has touched a field? We'll use an event called Blur, which is fired when a user enters an input field then leaves it.

In the form, for those four inputs, we'll now add an onBlur event handler set to a method named handleBlur, which we'll create in a moment, and we'll pass a string with the name of this field. So now any time that a user enters, then moves away from this field, that will fire this event handler.

Make sure that you add this for each of the four inputs, firstname, lastname, phonenum, email.

We can set up the event handler method now, handleBlur.

Because we're passing an argument other than event, we need to do something a little different here. We need to wrap the handleBlur method in another function.

Also, we're using arrow functions here to define this method. I mentioned before that event handlers can be bound in React by using an arrow function in the method definition, instead of explicitly using the bind method /in the constructor.. That's what we're doing here, so we don't need to bind the *this* reference for this method in the constructor as we did with the others.

Inside this method, we'll use setState to change the touched object..

Now, we don't want to change the entire touched object. We only want to change one of the properties inside it. The way we can do that is by using the spread syntax here, as you learned in the previous lesson.

Using the spread syntax, inside this object literal, we'll spread out the this.state.touched object, then we'll update the property that corresponds to the field that triggered this event, whether that's firstname, lastname, et cetera. And we'll set the value of that property to true, so we know that it has been touched.

Next, we'll start to implement the form validation. Let's set up a class method called validate, and we'll pass in the four fields.

Inside it, we'll set up an object named errors, and we'll initialize four properties of this object that correspond to, but are not the same as, those four fields. These properties will hold the error messages for these fields. At first, they will be blank, so we're using empty strings here.

Next, we'll write some more JavaScript to handle form validation.

[Slide: Form Validation]

Here, we will add a new method named validate, passing in the values we wish to validate.

Inside this method, we'll first set up an errors object. The properties of this object will hold the error messages for the four fields, if there are any errors. We will initialize each with an empty string, which means there are no errors.

This if statement will check if the firstName field has been touched. If so, then another if statement nested inside it will check if the length of the value of that field is less than 2. If true, then we'll set up an error message for the firstName, first name must be at least 2 characters.

Then we'll set up an else if check to see if the first name might be over 15 characters. If true, then the error message will say that it must be 15 or less characters.

We'll use the same validation logic for lastName, so I'll just copy this, then change firstName to lastName.

Next, to validate the phone number, I'm going to use a regular expression, a regex. Regexes are outside the scope of this course to go into in detail, but there if you are interested, please try the optional code challenge for regexes. Basically, what this is saying is to check the phoneNum value and see if it only contains digits, that's what this d stands for. Then this if statement will check, has the phoneNum field been touched, and did it fail the regex test, that is, does it contain anything other than digits? If so, then an error message will be set for the phoneNum.

Finally, with the email, we will check to see if the email field has been touched, and if the email contains an @ symbol.

Then at the end, we will return the errors object. If there were any validation errors for any of the fields, then the corresponding property in the errors object should contain an error message. If not, they should contain empty strings.

Next, we'll drop into the render method, and we're going to declare a variable here named errors as well. Remember, variables declared inside methods are locally scoped, so the errors variable that we created above is not available here. So we are going to declare a new variable named errors, then call the validate method, which we have to do using the *this* keyword. Then we'll pass in the current values of the firstname, lastname, phonenum, and email fields, which we conveniently have stored in state. Then that method will validate those fields and return the errors object, which is then stored inside this errors variable.

Every time there is a change in the input fields, this component will be re-rendered, so this validate method will be called for every change.

We're almost finished. Down in the form, we need to now set an invalid attribute for each input that we're validating. The invalid attribute is a boolean attribute, so for its value, we'll set a conditional, is there an error message set for this field. As you know, an empty string would evaluate as false, but if the error message was not an empty string, it would evaluate here as true, so then the invalid attribute gets set to true.

Then below each of the four inputs that we're validating, we'll render the FormFeedback component, giving it the content of the error message for that input.

Make sure to add the invalid attribute for each of the four inputs, and the FormFeedback component underneath each.

As you can see I'm just going to copy and paste then change the name of the field.

Now we can test the form on the website.

Make sure that your contact form is working as expected. You should see error messages if the first or last name are under 2 or over 15 characters, if the phone number contains non-digits, or if the email does not have an @ symbol. And the error messages should go away once you've entered a valid input.

We are now finished with this exercise. If you are using Git, this would be a good time to make a commit.

# 

# OVERVIEW - UNCONTROLLED COMPONENTS

You're now acquainted with how to set up forms as controlled components in React, with the form component information directly tied to the state of the parent component. And you probably noticed that it took a bunch of setup to do so.

Still, controlled forms are the recommended way to set up forms in React. However, there is an alternative way to set up forms using uncontrolled components. This approach can be useful at times when you just need a quick and dirty way to to create a small form with not much code, or if you're trying to integrate non-React code with React code. For example, you might work on a project that was written without using React, then wish to add React components to it. Then using uncontrolled components for forms may make that integration easier.

As we've discussed, HTML form elements already naturally keep their own internal state. This state is tracked by the DOM, separately from your React app. Controlled forms take over control of this state. With uncontrolled forms, you will just let the DOM continue to handle the internal state of HTML form elements, then just use your React component to retrieve that data from the form. Then it's the form elements within the DOM that act as the single source of truth for the form data, instead of React.

The way that we can retrieve that data is by using a React object called a 'ref', short for reference. In Reactstrap, the word 'ref' was already being used to mean something else when React released refs, so Reactstrap uses a different keyword, 'innerRef', to use React refs, and that's what we will be using in the following exercise to create an uncontrolled form.

# EXERCISE - UNCONTROLLED FORMS

* Change:
  + HeaderComponent.js

In this exercise, we will implement a login modal in the Header component with an uncontrolled form. We'll start by setting up the modal before moving on to the form.

**[slide: Add a Modal]**

We'll need to add a Login button to the header navbar that will open the modal.

First, in HeaderComponent.js, we will make several new imports from reactstrap:

Button, Modal, ModalHeader, ModalBody

Then we'll render a Modal component below the Navbar.

We'll add a ModalHeader component with the tag "Login" inside the JSX tags.

Then the ModalBody component, where we'll add the form later.

Now for this modal to be able to be opened and closed, we will add a Boolean property within the component state called isModelOpen, initially set to false. This will keep track of whether the modal is open or closed. Don't forget the comma at the end of the line above it.

Next, we need to set up a class method named toggleModal, similar to toggleNav. And we'll also need to bind this. And let me just move the toggleNav binding below to group them together, just as a personal preference.

Now we can go back to our Modal component.

The Reactstrap Modal component comes with two built-in attributes that we will use here, isOpen and toggle. For the isOpen attribute, we'll give it the value this.state.isModalOpen.

Then when this state is set to false, the Modal will be closed, or hidden. If the state is set to true, the modal will be open, or visible.

Then for the toggle attribute, we'll give it the toggleModal method as this.toggleModal. We'll also add the same toggle attribute to the ModalHeader. That will automatically provide a way to close the modal when it's been opened.

Next, we'll need to add a button to the navbar. We'll do this just below the Nav component, but still inside the Collapse, so that the Login button will show up in the collapsed menu.

And we can use a JSX span element for this, and give it the classes of navbar-text and ml-auto, so this will make this button have the same text styles as the navigation links, and it'll also align it horizontally to the right. And we'll give it an onClick event handler set to the toggleModal method.

Let's test this in the browser. You can see that the button appears in the navbar, and if we shrink the viewport, then it goes into the collapsed menu. And if we click it, the modal comes up, and we have a way to close it as well.

Make sure that your modal is working in this way before you continue. Next, we will implement the uncontrolled form.

[Slide: Add Login Form]

First, we will need to import a few more classes from Reactstrap: Form, FormGroup, Input, Label

Inside the ModalBody component, we'll add a Reactstrap Form component, and we'll set up an event handler here for when this form is submitted, onSubmit= {this.handleLogin}. This handleLogin method, we will implement in a moment.

Inside the form, we'll set up three Reactstrap FormGroup components.

Inside the first one, we'll set up a Reactstrap Label component for a username input, with an htmlFor attribute of username

Then the input itself, giving it a type of text, id, and name as usual for form input elements. And this one will be for the login username.

Inside the second one, we'll set up a label and input for the password in the same way, but using the input type of password instead of text.

Then the third one will be a checkbox, and for this, Reactstrap requires us to add a check attribute in the FormGroup as well as for the label. Then the Input for this one will go inside the Label tags, and the type will be checkbox. we don't need an id for the input nor an htmlFor for the label because when you nest the input inside the label like this, it's already clear what input the label is for.

handleLogin(event) {

alert(`Username: ${this.username.value} Password: ${this.password.value} Remember: ${this.remember.checked}`);

this.toggleModal();

event.preventDefault

}

And we'll add a submit button at the bottom.

And oops, I forgot to change this text here to Password.

Next, let's go back up and set up the handleLogin method. We're not building a back end that can actually authenticate this login, so all we're going to do is have it alert us to the form values when it's submitted. We will close the modal using the toggle method. Then we'll also use event.preventDefault() to prevent the entire page from being re-rendered.

Then we will also need to bind this event handler in the constructor.

At this point, you may be wondering, where did these properties get set? this.username, this.password. this.remember - they aren't in the constructor, as you can see.

So that's the next part, we need to make sure that these properties are being set by pulling the values from the form correctly.. And that's where React refs come in.

Let's go back down to the form. To each of the inputs, we can add a innerRef attribute and set it to a callback function, where the value of the input field is passed, then we define a property for each one, this.username, this.password, this.remember, and set the value of of that property to the value of the input field.

Now when we submit the form in the browser, i's able to pull the form data with the innerRefs and load them into the component state, then echo it back to us in this alert.

So this is how we can access form values from a React component without having to do all the expensive setup of a controlled form. Again, controlled forms are still the recommended way to set up forms in React, but it's worth knowing about this approach also.

<https://reactjs.org/docs/uncontrolled-components.html>

<https://reactjs.org/docs/refs-and-the-dom.html>

<https://css-tricks.com/working-with-refs-in-react/>

<https://reactstrap.github.io/components/modals/>

# THE MVC DESIGN PATTERN

In the software development world, you will often read or hear people mention something called MVC, or one of its variants such as MVVM or MVA. You may also sometimes hear it called the MVC framework, but it's not a framework in the sense that Bootstrap or Angular is a framework.

Instead, MVC is what's known as a software design pattern. It is not a library, it doesn't consist of any actual code at all. What it is, is a reusable approach for how to structure an application's code.

The MVC design pattern, was first formally introduced back in the '70s for use with developing user interfaces for desktop applications. Typically, MVC is not used for the entire application, but for developing the presentation layer of the application, the UI. Since then, it has become very popular in web development as well.

A key concept of MVC lies in its separation of concerns. The separation of concerns in software development facilitates independent development of different parts of an app, not to mention independent testing and maintenance. With MVC, the presentation layer is divided into three parts: the model, the view, and the controller.

Briefly, the Model part of MVC is concerned with the application state and logic.

The View part of MVC is concerned with presenting information to the user.

The Controller part of MVC handles events and mediates between the Model and the View.

We'll go into more detail about each:

The Model stores the main application logic and data for the front end - think of it as not a database, but the data that's been loaded into the application from the database, and the front-end code that sends requests to and receives data from the database.

It will also respond to queries from the front end - such as queries about its state, which usually come from the view. And it will respond to requests to change its state. which come from the controller. It may also be set up to automatically send out notifications to the view or controller when any changes are made to it.

\*\*\*\*

The View renders the model into a form suitable for display, typically a UI element. The view can query the Model directly for information about the application state so that it can adjust itself.

The Controller receives information from the View, such as if a user submits data through an input element. Then it can process and relay that information to the Model, and instruct the Model to change its state. It can also send instructions to the View to make changes.

While it's good to know generally about the MVC pattern as a developer, we will actually not be using MVC for our project. React state management is most often handled with a newer design pattern called Flux. And for our app, we will be using a state management library called Redux, which owes some of its parentage to Flux. Let's discuss Flux next, then you will learn about Redux.

# 

# FLUX

In its beginnings, React developers tried to use React with the MVC pattern, with React used specifically for the View.

However, Facebook began to encounter certain issues. They found that MVC did not scale well for a complex, large application like Facebook, which has many thousands of components.

The main issues were that MVC became unpredictable at this scale, and it was very difficult to add new features, even very small ones, without causing unforeseen problems in other parts of the app.

This was due to the bi-directional nature of MVC's data flow. You can see in this diagram the kind of complexity that would happen with MVC, and data going every which way. A small change in one part of the app could result in cascading effects that would cause bugs in other parts.

So the engineers at Facebook came up with a new design pattern, or architecture, that they called Flux. Like MVC, it's an approach for how to structure your code. It can be considered as an evolution of MVC.

The key feature of Flux that sets it apart from MVC is that it has a strictly defined unidirectional data flow. Then instead of model, view, controller, it separates its concerns into Action, Dispatcher, Store, and View. When an Action of some kind enters the system, it goes to the Dispatcher. There's only one central Dispatcher, and it acts as a traffic controller. There can be multiple Stores, and they're all registered with the Dispatcher, and the Dispatcher will route actions to the appropriate stores. Stores are repositories for the application state and logic. They are similar, though not identical concepts to Models in MVC. Finally, the View updates whenever the Store says that something has changed.

Then from the View, another Action could enter the system, due to some user interaction with the UI. The Dispatcher makes sure that once an action enters the system, no other actions are processed until it has finished updating the Store, so cascading effects are prevented and the data integrity is protected.

In the next video you will learn about Redux, which is a popular JavaScript state management library influenced by Flux.

<https://facebook.github.io/flux/>

<https://youtu.be/nYkdrAPrdcw>

# 

# OVERVIEW - REDUX

In the last lesson, you learned that Flux is a design pattern that describes an approach to structuring the front end of an application. It does not provide us with any code on its own. There are various JavaScript libraries that do implement actual code written with the Flux pattern. Such as Flummox and Alt. One such implementation is the Redux library. Redux has some differences from Flux, which we will discuss. But despite those differences, Redux was originally influenced by the Flux approach and can be considered as an evolution of Flux.

Redux was initially created by a developer named Dan Abramov for use with React, who was soon thereafter hired by Facebook to work on React. It was co-written with Andrew Clark, author of another Flux implementation called Flummox. Aside from Flux, it was also inspired by the Elm functional programming language, and three other JavaScript libraries: Immutable.js, Baobab.js, and RxJs.

On Redux's website, it describes itself as "a predictable state container for JavaScript apps".

It provides the code that will store the state of your application, and a consistent way of being able to access and update that state from anywhere in the application. Note, by the way, that it says "for JavaScript apps" and not "for React apps" - while Redux was initially created for working with React, it is independent of React and can be used without it. It could be used with vanilla JavaScript, with Backbone, Ember, et cetera.

Now, the React application we have been developing is fairly simple. We have a container component called Main that has a handful of presentational components as its children. The application data is kept in the Main component's state, and any changes to that state data can only happen inside the Main component.

This approach works fine when we're dealing with a small number of components. But in the real world, applications can have hundreds, even thousands of components. Instead of just a single container component like Main, there may be many different groups of components organized around multiple container components. And communication between all the various components can get to be very tricky. So the approach we have been using so far, while fine for our small application, would run into issues at scale.

So in the following exercises, we'll implement the Redux library to handle managing the state of our application in a predictable, consistent way. I will warn you in advance - it will be difficult to understand at first, and it may seem like we're adding a lot of code for not a lot of value. But it will make more sense if you consider that Redux is meant to be helpful at scale.

Let's examine Redux in a little more detail, then in the following exercise, we will install Redux and integrate it into our application. We'll start with looking at Redux in theory:

**[Slide: Redux In Theory]**

There are three fundamental principles of Redux:

1. Single Source of Truth
2. State is Read-Only
3. Changes are made with Pure Functions

**Single Source of Truth:**

In the Flux architecture, there can be multiple stores, and the dispatcher is responsible for directing actions to the right one.

In Redux, there is only one store, and it holds a single state object tree that contains the whole state of your entire application as objects within that tree. The phrase 'object tree' refers to a tree-like data structure where every node contains an object. The DOM is another example of an object tree.

**State is Read-Only.**

The only way to change the state is with an *action*, an object that describes what happened. You will learn more about how actions work later.

**Changes are made with Pure Functions.**

Specifically, changes are made with pure functions that Redux calls reducers. Let me go into some detail about pure functions and reducers.

[sliA pure function is a general programming concept of a function that has no side effects and always returns the same output when provided with the same input. For example, Math.floor() is a pure function because if you give it the same number, it will always return the same result, and it doesn't have any side effects. Math.random() is not a pure function because it always gives a different result.

A Redux reducer is a pure function that takes the previous state and an action, and returns the next state. Reducers will always return new state objects instead of mutating the previous state.

This is similar in concept to how array methods like map work, where they never mutate the original array but create an entirely new one.

This is all linked to a programming concept called immutability, meaning data does not get mutated, only replaced by new data.

Replacing the entire state every time might seem like overkill, but objects are cheap to create in JavaScript, and doing it this way creates a lot of advantages, such as the ability to "time travel" - meaning you can step back to and forward from a previous state.

I have my own analogy for this: Think of it like an animation sequence, or an illustrated flipbook. Every frame in an animation, or page in a flipbook, is slightly different from the one before. But it is its own frame or page. Instead of editing or erasing and drawing over the existing frame or page whenever you want to make a change, you simply move to a new page and make the change there.

So those are the three fundamental principles of Redux. Now, how are these achieved in practice? We'll look at the technical details now.

**[Slide: Redux In Practice]**

[will need an illustration here]

In the Flux architecture, we had actions, a dispatcher, stores, and the view.

In Redux, we have Actions, Reducers, a single Store, and the View.

The dispatcher in Flux worked as a traffic controller that routed actions to different stores. With only one store, that role is no longer necessary. We do still have a dispatch method in Redux.

User interactions with the view trigger the dispatch of actions. The actions are sent to reducers. And the reducers create a new state, which is inside the Store. Then the View will change based on the state.

...

To use Redux with react, you will install the react-redux library. From this library, I'll briefly go over a few important functions that you will learn to use in the following exercises. There are other functions than these, of course, but we'll only look at the ones we'll be using this week for now.

This week, we will primarily be concerned with installing Redux and transferring our application state to the Redux store, and these are the functions that will help us do that. Next week, you will learn more about actions and reducers.

createStore() - this creates the Redux store which holds the state object tree

connect() - generates a container component that wraps around another component and subscribes it to the store

mapStateToProps() -

* this function must be created inside your component then passed as a callback in the connect() function
* called whenever the store state changes.
* receives the entire state tree, then returns an object that contains only the data needed by this component.

You will also use a React-Redux component named <Provider> that will wrap around the root component of your app, which is the App component. It takes the store variable as an attribute, and it makes the store accessible to all components that are connected with the connect() function.

<https://redux.js.org/introduction/prior-art>

# EXERCISE - INTRO TO REDUX

* Install:
  + yarn add redux@3.7.2
  + yarn add react-redux@5.0.7
* Add:
  + src/redux folder
  + src/redux/reducer.js
  + src/redux/configureStore.js
* Update:
  + src/App.js
  + src/components/MainComponent.js

To begin, we have two libraries to install. We'll need to install the redux library, using yarn add ~~or npm install.~~

Then we will also need to add the react-redux library, to make redux work specifically with React. Remember, Redux is not just for React, it can be used with any JavaScript application.

Next, we have a new folder to add inside the src folder. We'll call it redux.

In this folder, we'll add our first Redux-related code. Create a file named reducer.js.

We are moving the responsibility for the state from our Main component to Redux. And in Redux, we use reducers to create and update state. So here, we will import campsites, comments, partners, and promotions, all from the shared folder. Remember to use double dots here, since these source files are not inside the redux folder, so we need to back out of the redux folder then into the shared folder.

We will now create an object called initialState, and this will be, well, the initial state of our app, which we will get straight from the source data files. So we'll have a campsites property, which will use the campsites array, then a comments property, a partners property, and a promotions property.

Then, we will need to create a reducer function

const Reducer = (state = initialState, action) => {

return state;

};

As you can see, this uses default function parameters so that if there is no state passed in, then the state gets set to the initialState object that we just created. It also takes an action as a parameter, and for now, it just returns the same state that was passed in without making any changes.

Finally, we will need to access both these constants from other files, so we will need to make them into named exports. That is as simple as adding the word 'export' in front of each constant.

That's all we need for this file. Next, in the redux folder still, you will need to add a new file named configureStore.js.

Here, we will import createStore from redux, then the Reducer and initialState modules from the reducer file, and here we don't use a double dot, only a single dot because we're in the same folder as reducer.js.

Then we will create an arrow function here, and it will be a named export, and we'll call this ConfigureStore. Inside, we'll create our store variable, using the createStore function from Redux. And we'll pass it that Reducer function and the initialState. Then this function will return the store.

Next, we'll go into App.js and import the Provider component from react-redux, and the ConfigureStore function that we created.

Then remember, this function returns the Redux store,

so we'll capture that return value in a constant in App.js named store.

In the App component's return statement, we'll wrap everything inside with the Provider component, giving it the store as an attribute. This makes the Redux store available to all connected components that are children of App.

Finally, we'll open MainComponent. We will need two new imports here. First, the withRouter function from react-router-dom. Then, we'll import the connect function from react-redux.

Next, We are no longer storing the application data in the Main component state, we're transferring it to the redux store. That means we no longer need these imports, and we can remove the entire constructor from Main. Instead, in order to get the state from Redux, we need to set up the mapStateToProps function. We'll do that here, and it will take the state as an argument and return the campsites, comments, partners, and promotions arrays as props.

Then we need to change all these occurrences of state to props.

And finally at the bottom, we need to set up the connect method, like this. You will use a pair of parentheses after connect, the first will contain mapStateToProps, and the second will contain the Main component. All this together that we've done here now allows the Main component to take its state from the Redux store.

Then there's one more thing we need to do. Remember the withRouter function we imported earlier from react-router-dom? We need to wrap our export in this so that the React Router will still work with these changes to our export.

If you are using Git, this is a good time to make a commit.

# EXERCISE - REACT REDUX FORM

Now that we are using the Redux store in our React app, we will need to configure the Contact form to use the Redux store instead of the Contact component's local state.

All state data must now be stored in the Redux store, as it should be the single source of truth for our application state.

To help us with this, we will start by installing a library called react-redux-form~~,~~ using yarn ~~or npm~~. (yarn add react-redux-form@1.16.8)

~~This will help us to store the form state inside the Redux store.~~

~~Storing your form data in Redux is also helpful with form data persistence.~~

~~What I mean by form data persistence is, you could fill out a form partially, then move to a different view, maybe the About Us view,~~

~~then when you come back to the form, the fields would still contain the same data instead of being reset.~~

~~This week, we will use React-Redux-Form to create what it calls a LocalForm, which does not persist state data in this way, but next week, you will learn how to do that as well.~~

Then let's open Contactcomponent.js

We will be replacing the Reactstrap form components with ones from React-Redux-Form.

So we can get rid of these imports: Form, FormGroup, Input, and FormFeedback.

We will add a Row component import from reactstrap.

We will also need to import these components: Control and LocalForm, from react-redux-form.

Next, we can remove the handleInputChange and handleBlur methods, since we are now going to let react-redux-form take care of that behavior.

We will still need the handleSubmit method. But we will change the object we're passing in from event to values,

then we'll remove state from the console.log and alert, changing them to the values object.

And we can remove this line, event.preventDefault, as react-redux-form will handle this for us now.

We can also remove the entire validate method. Yes, the whole thing.

And we'll remove this errors variable in the render method that called that validate method.

Next, the Form component itself, we will change to the react-redux-form component LocalForm.

We will still keep the onSubmit method, but we will pass this values object to it now, using an arrow function.

All the FormGroup components we have below, we will now update to the Row component, with the className of form-group,

This is because we are no longer using the FormGroup component from reactstrap, since that only works with the reactstrap Form component.

This FormGroup with the check attribute will turn into a div with the className of form-check.

Make sure to go through and check that all the FormGroups have been converted, and the closing tags have been changed as well.

We can leave the Labels alone. But we will need to remove all the FormFeedback components.

All the Input components, we'll replace with the Control component, and we'll specify what kind of control like this, Control.text.

The phone and email will also be set up as Control.text.

This input with the type checkbox will be set up as Control.checkbox.

This input with the type select will become Control.select.

And this textarea will become Control.textarea.

(do each input)

Make sure to fix the closing tag the Select and textarea controls. For the textarea control, we'll turn it into a self-closing tag.

And I'm just going to fix the indentation here to make the self-closing tags on the other Controls more obvious.

Next, for each of these Control components, we will need to add a required "model" attribute.

This tells Redux that the value for this field will be stored in the state under the property name of firstName.

Be sure to add a dot in the beginning for each of the model attribute values. (do firstName all the way first now, including form-control, removing attributes)

We will also need to add the attribute of className="form-control" to each of these Control components.

Before, the Input component we were using from Reactstrap took care of adding this form-control class,

but we're not using that anymore, so we need to add it manually.

Finally, we will remove the rest of the attributes we were using for validation

- the value, invalid, onBlur, and onChange attributes can all be removed from each of these Control components.

So we'll do that for each of the Control components. We'll add a model attribute.

Then we'll add a form-control className. Then we'll remove the value, invalid, onBlur, and onChange attributes.

The exception is the checkbox control, for which we'll add a model as well, but the classname will be form-check-input.

Go ahead and test the app in your browser. We're not doing any validation at this point, so it shouldn't matter what values you put in, but they should all get echoed back to you in the alert when you submit the form.

We are at the end of this lesson, so if you are using Git, this is a good time to make a commit.

<https://davidkpiano.github.io/react-redux-form/docs/api/Control.html>

# 

# EXERCISE - REACT REDUX FORM VALIDATION

* Change:
* ContactComponent.js

We will now be adding validation back into the Contact component using React-Redux-Form.

In ContactComponent.js, we will first need to import Errors from react-redux-form.

Then, above the Contact component definition we will first define a few functions to help us with our validation.

[Slide: Error Checking Logic]

const required = (val) => val && val.length;

const maxLength = (len) => (val) => !(val) || (val.length <= len);

const minLength = (len) => (val) => val && (val.length >= len);

const isNumber = (val) => !isNaN(Number(val));

const validEmail = (val) => /^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(val);

This first function, required, receives a value as an argument.

And this will be a string value that it receives, we know that just because all form inputs are received as strings, even if they're numbers.

Then inside the function, it checks to make sure that there was a value received, it wasn't undefined or null,

And just this check for val will tell us that, because it would evaluate as falsy if it was undefined or null

then it checks and makes sure that the length of that string is greater than 0.

So basically, this function makes sure that a field has something in it, and it'll return true if it does, and false if it doesn't. If it returns false, that will mean it has failed this test and will create an error.

The next function, we will call maxLength. The way that this function will be called later requires us to wrap a function inside a function, so notice there's two arrow functions here, not just one. The first one takes the maximum length.

Then the second one takes the value, the input string.

Then from inside the inner function, we want to return true if the maxLength has not been exceeded.

So not val will return true, because if there's no value, the maximum length clearly has not been exceeded.

Or, we will also return true if the value's length is less than or equal to the maximum.

If both of these conditions are false, then this function will return false for maxLength, meaning it has failed the test for maxLength and will create an error.

This minLength function works similarly. It wraps a function in a function, then this inner function will return true if there is a value *and* the value is greater than or equal to the minimum.

And it'll return false if either of those conditions are false, and that will mean it has failed the test for minLength and will create an error.

Then the next function we will define is isNumber. So we want to check to see that the value is a number, so first we'll use the unary plus operator to turn this value into a number.

And if the value is not a valid number, the unary plus operator will turn this value into the special value NaN, which is Not A Number. Then we will check to see if this value is the opposite of isNan. So the logical not operator, the bang, is used in front of isNan.

So if this value is not a valid number, then this will ultimately return false, and if it is a valid number, then this will return true - basically, we're saying it's NOT not a number.

And then with this last one, we're going to use a Regular Expression to check for a valid email. And it'll be a little more thorough this time then just checking for an @ symbol.

I will go ahead and just copy and paste this in from the written instructions, as it's quite long.

So this is checking to make sure that the email address begins with A to Z, then contains only the characters that are valid in an email address: zero through nine dot underscore percentage plus and minus

and then we'll check to see if there is an at sign in there

then following the at sign, where the domain name would be, we'll permit characters that are A to Z or zero through nine any number of times,

then a dot is required, then the domain extension, which can be between 2-4 letters.

Then we'll use the built-in method from JavaScript called test here

This will test whatever value is passed in to see if it matches this regex pattern.

Now, to use these functions in form validation, let's go down to the form.

[Slide: Contact Form]

For each of the Control components in the form, starting with firstName, we will add a new attribute called validators.

For the value of that attribute we'll give it an object that contains the functions that are appropriate for that component. So for firstName, we want to give it the required function, then the minLength function,

and we'll specify that we want the minLength of 2, then maxLength of 15, like this.

It'll be exactly the same for Lastname, so I'll just copy and paste that over.

For the phoneNum control, we'll add the required validator, a minlength of 10, and a maxlength of 15, and we want to check for isNumber on this one.

Then for the email control, the only validator we want is the validEmail

Now we will use the Errors component from react-redux-form. Let's start with FirstName and work our way down again.

After the control component, we'll add the Errors component, and we're going to give it a few attributes.

The first attribute will be className="text-danger". That will make the error text color red.

The second attribute will be model. This just needs to match the model of the corresponding control component, so we give this one the value - dot firstName.

The third will be show=touched. This will cause the form field to only show error messages if it's been touched by the user. It's the same functionality as the code we set up before to do that in a previous exercise, except it's much simpler here because it's already built in for us.

Then we'll add component="div", and that just tells react-redux-form to wrap each error message in a div.

Then we'll add a "messages" attribute, and these will be the error messages that will be shown for these functions if they return false.

Again, we can just copy and paste this for lastName. Make sure that you replace the model though with lastName.

Then I'll copy it over for phoneNum, making sure to replace the model,

and for this we need to change the error messages to a minimum of 10 and a maximum of 15 numbers,

and we'll add an error message for if the isNumber test fails.

Similarly, for email, we need to change the model, and give this one an error message for validEmail

Then that's it! Now we should be able to test this in the browser

And if I touch each field and leave it without putting anything in, you can see that the errors come up

And if I go under the minimum or over the maximum, the appropriate error message comes up

And I can't submit this form until I fix the errors

And once the errors are fixed, I can submit the form, and an alert should come up to echo what I entered.

Make sure that your form validation is working in this way before you continue on.

<https://github.com/davidkpiano/react-redux-form/blob/master/docs/api/Errors.md>

# WEEK 4 WORKSHOP ASSIGNMENT

* Change (for all tasks)
  + CampsiteInfoComponent.js

SCRIPT:

For this assignment, you will only need to update one file: CampsiteInfoComponent.js.

You will be adding a form where users can submit their own comments. It will be located inside a modal.

This assignment is divided into three tasks:

For TASK 1:

* You will add a new class component named CommentForm.
* For now, all this component needs to do is render a Button, using the Reactstrap Button component, with the text "Submit Comment" on it.
* Then you will update the CampsiteInfo component so that it renders the CommentForm component.
* At the end of this task, your website should look like this whenever you are viewing a campsite's information. This button will not do anything just yet.

FOR TASK 2:

* Update the code so that clicking on the button will bring up a modal with the contact form.
* You will be using the Modal component from Reactstrap, along with the LocalForm component from react-redux-form.
* This form should include three fields: rating (1-5), author, and the- comment text.

FOR TASK 3:

* You will add validation for the author field only.
* You will validate it for a minimum length of 2 or more characters, and a maximum length of 15 or less characters.
* An error message should display if the input is invalid.

For this assignment, you are not being given explicit instructions on every step to take to get to the desired goal. You will have to fill in the blanks based on what you have learned, more so than you have done before.

Remember to work together with your workshop partners, and don't forget the 10 minute rule.

Good luck!

----

PAGE:

Objectives

* Update CampsiteInfoComponent.js to:
  + Include a new component named CommentForm.
  + Use react-redux-form to provide and validate a form for users to submit comments about a particular campsite.

Instructions

Note: Some of these instructions will not be detailed, step-by-step instructions on purpose. Attempt to refer back to your exercises during the week and infer on your own (and with your workshop partner) how to accomplish each task. But don't forget the 10-minute rule!

Task 1

* In CampsiteInfoComponent.js, create a new class component named CommentForm. For now, this class component will only render a Reactstrap button with the text "Submit Comment" on it, as shown in the screenshot below. Use the appropriate Font-Awesome icon, and the Reactstrap Button has a boolean attribute "outline" that can be used.
* In the RenderComments component, use the CommentForm component to display the button, as shown in the screenshot below.
* This button will be used in the next task to toggle a modal, but do not be concerned with this yet.

##### **Task 2**

* Construct a Modal that contains a React Redux form for users to submit their comments as shown in the image, as detailed below.
* The Add Comment button you set up in Task 2 must now toggle on a modal, using Reactstrap's Modal component.
* Inside the modal, set up a form as a LocalForm using react-redux-form with three Control inputs: rating, author, comment, as follows:
  + Implement the rating field Control.select and options 1-5, the author field with Control.text, and the comment text with Comment.textarea and six rows. For the model of the author, use ".author". For the model of the rating, use ".rating". For the model of the comment text, use ".text". The id and/or name for each should be based on the model. Similar to ContactComponent.js, you should be using the Control component from React Redux Form for each of these.
* For formatting: You do not need to use the Bootstrap grid for this. Format this in a way similar to the Login modal form, but instead of using the FormControl component (since that is not available for use with LocalForm), use a div with className="group".
* When you submit the form, it should echo back the form inputs to you in an alert as well as in a console.log.

##### **Task 3**

In this task, you will enable form validation as shown in the images below. You need to complete the following:

* The author field should be validated to be at least be two characters long.
* The author field should be validated to be less than or equal to 15 characters.
* The user should be alerted by showing the invalid message displayed at the bottom of the field.