React Fundamentals

Author

Drew Fierst

Notice of Rights

Disclaimer
Logical Imagination Group LLC makes a sincere effort to ensure the accuracy of the material described herein; however, we make no warranty, expressed or implied, with respect to the quality, correctness, reliability, accuracy, or freedom from error of this publication or the products it describes. Data used in examples and sample data files are intended to be fictional. Any resemblance to real persons or companies is entirely coincidental.

Notice of Liability
The information in this course is distributed on an 'as is' basis, without warranty. While every precaution has been taken in the preparation of this course, neither the authors nor Logical Imagination Group LLC shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Trademark Notice
Throughout this course, trademark names are used. Rather than just put a trademark symbol in each occurrence of a trademarked name, we state we are using the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Table of Contents

Table of Contents

Introductions

- Tell us your name
- What you're supposed to do at work
- What is your JavaScript experience?
- What is your web development experience?

Course Agenda

Day 1

- Introduction
- React Syntax and Basics
- Dynamic Content
- Styling Content
- Debugging

Course Agenda

Day 2

- Components
- Web Server Interactions
- Routing
- Forms

Course Agenda

Day 3

- Managing State with Redux
- Async Redux
- Testing
- Transitions and Animations

Course Agenda

Day 4

- Introduction to Hooks
- Side Effects
- Reducers and Context
- Custom Hooks

Lesson 1: Introduction

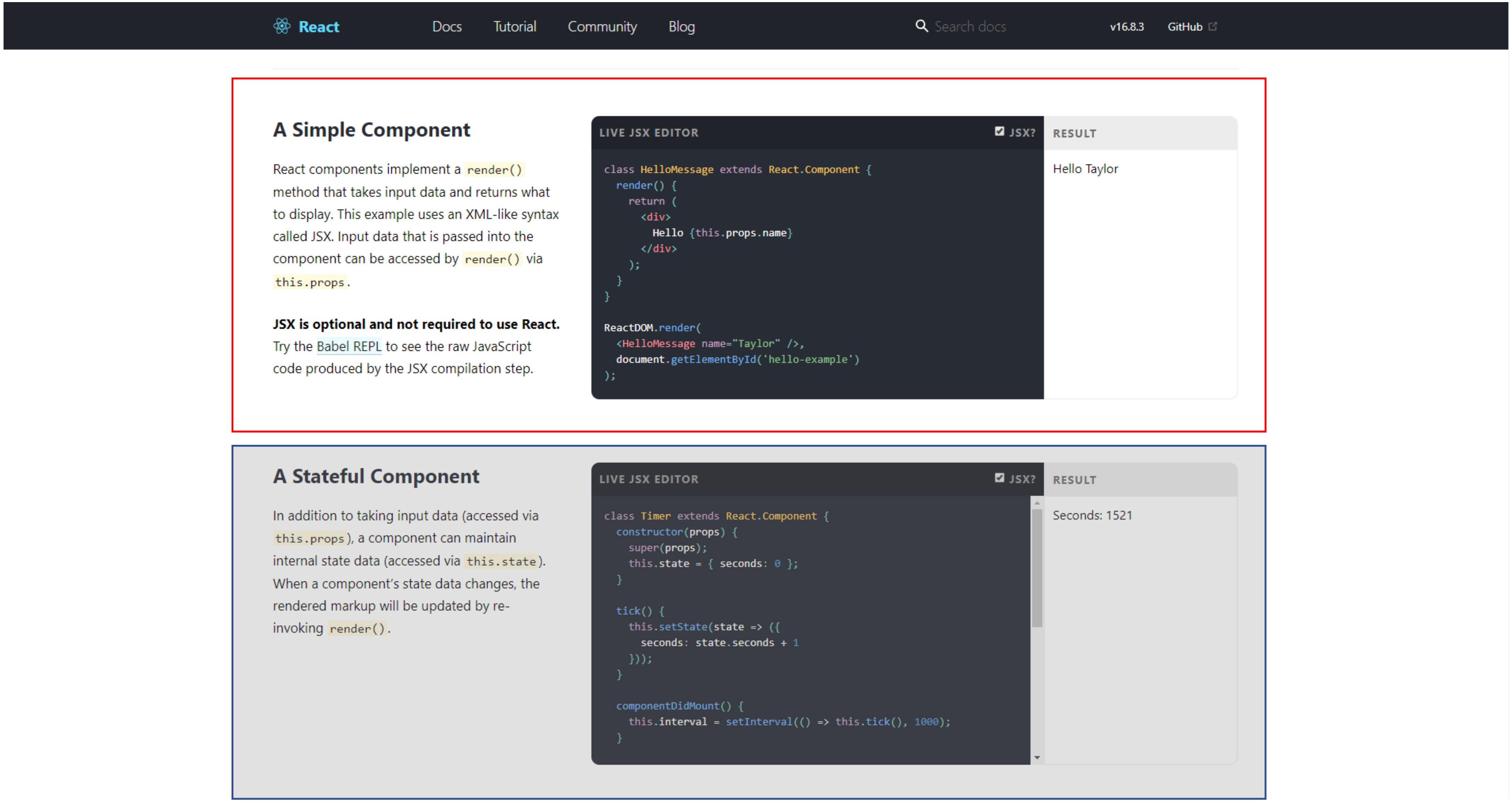
In this lesson you will learn about:

- What is React
- Why use React?
- SPAs and React Web Apps
- NextGen JavaScript Features

What is React?

- "A JavaScript Library for building User Interfaces" – reactjs.org
- React is a browser-based framework.
  - It implements application logic in the browser.
  - It quickly, efficiently updates the UI when data changes.
- React is a component-based framework.
  - It creates encapsulated application blocks.
  - Its architecture promotes reusability.
- React is a framework for creating custom HTML elements.
  - It facilitates building complex UI composed of smaller components.

Component Reuse



A Simple Component

React components implement a `render()` method that takes input data and returns what to display. This example uses an XML-like syntax called JSX. Input data that is passed into the component can be accessed by `render()` via `this.props`.

JSX is optional and not required to use React. Try the Babel REPL to see the raw JavaScript code produced by the JSX compilation step.

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  document.getElementById('hello-example')
);
```

Hello Taylor

A Stateful Component

In addition to taking input data (accessed via `this.props`), a component can maintain internal state data (accessed via `this.state`). When a component's state data changes, the rendered markup will be updated by re-invoking `render()`.

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  tick() {
    this.setState(state => ({
      seconds: state.seconds + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }
```

Seconds: 1521

Why Use React?

- It is scalable.
  - Use React as a library to build view layer only.
  - Use React as a framework to build a complete SPA.
- It has a shallow learning curve.
  - React is easier to manage UI state than plain JavaScript.
  - It lets you focus on application logic instead of low-level scripting details.
- React creates reusable components.
  - It is easy to extend and maintain ReactJS applications.
- React has fast rendering.
  - Diffing the virtual DOM minimizes actual DOM updates.
- There are available browser Dev Tools.
  - They provide useful debugging information is readily available.
- You can use React Native.
  - Leverage your React knowledge to build native iOS and Android apps.

SPAs and React Web Apps

Single Page Applications

- Only one HTML page sent to client
- JavaScript manages showing new content
- Content is rendered on client
- Smoother, app-like user experience
- React renders/manages the entire app

Multi Page Applications

- Each screen is a different HTML page
- Browser loads new HTML content pages
- Content is rendered on server
- Episodic, interrupted user experience
- React can render individual UI "widgets"

NextGen JavaScript Features

- ES2015 (ES6) introduces many new features, including:
  - "let" and "const"
  - Arrow functions
  - Exports and Imports
  - Classes
  - Class Properties and Methods
  - Spread and Rest operators
  - Destructuring
  - Array functions
- React applications typically use these features extensively.
- Transpilers such as Babel allow for use of these features when supporting older browsers.
  - The build process would include transpiling ES2015 features to older JavaScript syntax.

Declaring variables with "let"

- "let" is almost identical to "var".
  - It provides opportunity to use block-level scope for variables.

Code Bite

```
function myFunction() {
  if (true) {
    var count = 1;
    let message = 'Hello';
  }
  console.log(count);
  //count is usable here
  console.log(message);
  //error – message does NOT exist here
}
```

Creating Constants

- Constants are like a variable, but the value cannot be changed after being defined.
  - Most linters recommend constants when they detect a variable's value is never changed.

Code Bite

```
const daysInPeriod = 14;
const product = { name: 'Widget', color:
'blue' };

daysInPeriod = 15;
//error, cannot change value of a constant
product = {};
//error

product.color = 'red';
//OK – can change property values
```

Arrow Functions

- Arrow functions are essentially a shorthand syntax for an anonymous function.

```
      const myFunction = (parm) => { . . . };
```

- If exactly one argument is used, the parenthesis are optional.
- If the function body has only one line of code, the curly braces are optional.
  - If the curly braces are omitted, an implicit "return" is included.
- Arrow functions bind the "this" reference when they are defined.
  - Standard JavaScript functions bind the "this" reference when they are invoked.
  - This makes arrow functions more portable.

Function context example

```
const widget = {
      height: 200,
      createMeasureFunction: function() {
            return function() {
                  return this.height;
            }
      }
};

const getHeight = widget.createMeasureFunction();
const height = getHeight();
console.log(height);                          //undefined!!!
```

Function context solution

```
const widget = {
      height: 200,
      createMeasureFunction: function() {
            return () => {
                  return this.height;
            }
      }
};

const getHeight = widget.createMeasureFunction();
const height = getHeight();
console.log(height);                        //200
```

Exports and Imports

- Each JavaScript file becomes its own module.
  - Assets defined in a module are (by default) not visible to code in another module.
- Assets are declaratively exported from modules.
  - Any number of assets can be exported from a module (named exports.)
  - Only one exported asset can be marked as the default export.
- Exported assets can then be imported into other modules
  - This clearly defines dependencies between modules.
  - It allows for intelligent bundling or on-demand loading.
  - Non-default exports must be imported by their name (listed between curly braces.)
  - Default exports can be imported as any name desired (curly braces are omitted.)

Export and Import example

```
//inside customer.js
const customer = { name: 'Sally' };
export default customer;

//inside utility.js
export const taxRate = 8.5;
export const addTax = (amount) => amount * taxRate;

//inside app.js
import newCust from './customer.js';
import { taxRate, addTax } from './utility.js';
```

Import aliases

- Named exports can be aliased when importing.

```
      import { taxRate as percentage } from './utility.js';
```

- Everything exported from a module can be imported at once.
  - An alias is required.
  - Named exports are accessed by their name as properties of the alias.

```
      import * as taxHelper from './utility.js';

      const total = taxHelper.addTax(49.95);
```

Classes

- Classes are a new syntax for creating JavaScript object types.
  - It is merely syntactical sugar over function prototypes.

Code Bite

```
class Person {
  name = 'Carlos'
  sayHello = () =>
    console.log('Hi my name is '
      + this.name)
}

const p1 = new Person();
p1.sayHello();
console.log(p1.name);
```

Class inheritance and constructors

- Classes can inherit from other classes.

```
class Employee extends Person {
}
```

- Classes can have constructors.

```
class Person {
      constructor(nm) {
            this.name = nm;
      }
}
```

- Constructors in extending classes MUST invoke parent class' constructor.
  - Use the keyword "super" to refer to parent class.

Class Properties and Methods

- Properties are data attached to objects.

- ES6 syntax:

```
constructor() {
      this.myProp = 'value';
}
```

- ES7 syntax:

```
myProp = 'value';
```

- Methods are behavior attached to objects.

- ES6 syntax:

```
myMethod(p1, p2) { . . . }
```

- ES7 syntax:

```
myMethod = (p1, p2) => { . . . }
//benefit: binds "this" to the
//object instance
```

Spread and Rest operator

- The spread operator distributes array elements or object properties into a new item.

```
const newArray = [ ...oldArray, 42, 17];
const newObj = { ...oldObject, quantity: 5, discount: 0.05 };
```

- The rest operator accumulates individually provided parameters into an array.

```
function printSum(prefix, ...nums) {
        const total = nums.reduce((tot, num) => tot + num);
        console.log(prefix + ': ' + total);
}
printSum('Total is', 3, 4, 5, 6, 7);
```

Object Destructuring

- You can extract array elements or object properties to be stored in individual variables.
  - This allows for selection of specific elements/properties instead of using ALL.
- Array destructuring example:

```
let [a, b] = ['Sunday', 'Monday', 'Tuesday'];
console.log(a);           //Sunday
console.log(b);           //Monday
```

- Object destructuring example:

```
let { name } = { name: 'Gear', price: 4.95 };
console.log(name);        //Gear
console.log(price);       //undefined
```

Array Functions

- Each accepts a function as their parameter.
  - They loop over the array elements, executing the function on each element.
- Some examples:
  - forEach()
    - Executes some functionality for each element in the array.
  - filter()
    - Returns a new array consisting of elements for which the function returned true.
  - map()
    - Returns a new array consisting of the return value from each execution of the functions.
  - indexOf()
    - Returns the index of the first element for which the function returns true.
  - some()
    - Returns a Boolean indicating whether the function returned true for at least one array element.
  - every()
    - Returns a Boolean indicating whether the function returned true for all of the array elements.

Array Function - Reduce

- Reduce aggregates a value over the set of items.
- The first parameter is a function.
- The second parameter is the initial value.
- This function is given two parameters:
  - The current aggregate value.
  - The current element being iterated.
- This function should return new aggregate.

Code Bite

```
const arr = [1, 2, 3, 4, 5];

const total = arr.reduce(
  (tot, curr) => tot + curr,
  0);
```

React Basics

- React uses JSX – a preprocessor extension that adds XML syntax to JavaScript.
  - It requires the use of Babel or some other JavaScript preprocessor in application build steps.
- React applications require two libraries to be included.
  - react includes the core logic for React applications.
  - react-dom includes the DOM interaction for React web applications.
- The most basic syntax for a React component is a function.
  - The function name gets turned into a custom HTML element by React.
  - Attributes applied to that custom HTML element are given to the component instance as props.
  - Props are provided to the component function as a hashmap.
- ReactDom.render() is invoked to process the custom HTML element and its content.
  - A DOM node reference must be provided to tell React where in the document to place the content.

A simple example

```
<div id="results"></div>

function Person(props) {
  return (
    <div className="person-card">
      <h3>{props.name}</h3>
    </div>
  );
}

ReactDOM.render(<Person name="Qing" />, document.querySelector('#results'));
```

Exercise 1: Introduction

Skill Check

- Locate and complete the instructions for exercise 1 in your student files

Lesson 2: React Syntax and Basics

In this lesson you will learn about:

- Using Create React App
- Component Basics
- JSX
- Props and Dynamic Content
- State and Event Handling
- Two-Way Binding

Build flow

- Local applications typically go through a build process before running.
  - Transpile NextGen JavaScript features.
  - Optimize code.
  - Utilize linting for quality.
  - Execute CSS pre-processors.
- Will often use dependency management tools – npm or yarn.
- Employs a bundler – webpack.
- Includes a transpiler – babel.
- Uses a development web server.
- Setting up all of these could be done individually, but. . .

Using create-react-app

- create-react-app is a CLI for building new react applications.
  - It creates a standard folder structure.
  - It creates build flow.
  - It scaffolds initial content.
  - It configures a development web server.
- It is installed via npm.

```
npm install –g create-react-app
```

- create-react-app is used to create a new application.

```
create-react-app my-new-app
```

- It creates a directory using the application name as the directory name.
  - Then it installs react, react-dom and react-scripts.

Starting the application

- Use npm at the command prompt.

```
        npm start
```

- It builds the source files.
- It starts a development server.
- Then it loads the site in a browser window.
- Lastly, it monitors the source files to auto-rebuild and reload upon changes.
- Use Ctrl-c to close the application when done developing.

Tour of application files

- package.json – describes dependencies and application scripts.
- node_modules – contains all dependencies.
- public – root folder for content served by web server.
  - Contains index.html – the only html file served by a SPA.
  - Don't create multiple html pages here – a multi-page app would use create-react-app multiple times.
- src\index.js – start of the application.
- src\app.js – only component (currently) in the application.
- src\index.css – global styles for the application.
- src\app.css – styles for the app component (initially applied globally.)
- src\serviceWorker.js – base file for PWA behavior (initially disabled.)

Component basics

- ReactDOM can render any content onto the screen, including plain HTML.
  - Would not be dynamic, React wouldn't have anything to update as state changes.
- Typically render a React component.
  - React can monitor application state and update the screen automatically.
  - UI events can trigger React code to execute application code.
- ReactDOM.render() could be invoked multiple times to render multiple components.
  - Would create multiple "root" components.
  - Difficult to communicate between them.
  - Events in one component would not affect the state of other components.
- Typically render one root component.
  - Contains HTML and child components.
  - Child components can contain their own child components, etc.

Component syntax

- Can use class syntax to create components.
  - Functional syntax is the other option – more shortly.
- Class must extend Component (imported from React.)
  - Must also import React to be able to render anything to the screen.
- Class must have a render() method.
  - React will invoke when rendering this content to the screen.
  - Content (typically JSX) is returned.
  - JSX must be surrounded by ()
- Class is then exported as the default export from the file.

JSX

- Preprocessor step that adds XML syntax to JavaScript (JSX = JavaScript XML).
  - Invented by the React team.
- Allows for HTML and XML syntax to be mixed in with JavaScript.
- Combines rendering logic with content and other UI logic.
- Uses single curly braces to embed expressions into content.
- Content gets HTML-encoded.
  - Protects against content-injection attacks.
- Ends up being converted to React elements.

React element

- An object describing a React component or DOM node.
  - Can include property values and content.
- React.createElement(elt, config, childContent[, …n])

```
return React.createElement('div', { className: 'title' }, 'Hello World!');
```

- Can be expressed in JSX as:

```
return (<div className="title">Hello World</div>);
```

JSX restrictions

- "class" is a reserved word in JavaScript, so "className" must be used.
- Use initial upper-case letters for components, initial lower-case letters for HTML.
- Use double-quotes on attribute values.
- Use camel-case for props and enclose values in curly braces { }
- A single root element is required (React < 16)

```
return (
       <h1>My Title</h1>
       <h2>All about stuff</h2>
)
//the preceding is not allowed
```

Multiple root elements

- React 16+ allows multiple root elements by returning an array, with keys on each element.

```
return ([
    <p key="1">Multiple elements!</p>,
    <p key="2">Multiple elements!</p>
])
```

- React 16.2+ allows for fragments to be returned.

```
return (
    <React.Fragment>
        <p>One element</p>
        <p>Another element</p>
    </React.Fragment>
)
```

  - Also works with an empty element <> . . . </>

Functional component

```
function Person(props) {
  return (
    <div class="person-card">
      <p>{props.firstName} {props.lastName}</p>
      <p>{props.email}</p>
    </div>
  );
}
```

ES6 class component

```
class Person extends React.Component {
  render() {
    return (
      <div class="person-card">
        <p>{this.props.firstName} {this.props.lastName}</p>
        <p>{this.props.email}</p>
      </div>
    )
  }
}
```

Functional components*

- Are pure functions.
  - Depend solely upon their inputs.
- Have no state of their own.
- Do not have lifecycle hooks.
  - mounting, updating, unmounting, etc.
- Are easier to test and maintain.
- Could be optimized by React to perform better.
  - Avoid unnecessary change detection checks.
  - Minimize memory allocation.
- Best practices in design include:
  - Few "smart" components with state and lifecycle behavior – more like controllers.
  - Many "dumb" components with no state and lifecycle – more like views.

* - this conversation changes with React Hooks (version 16.8)

Props

- Allow a component to make use of data "owned" by an ancestor component.
- Are attribute values from component element, passed to component instance.
- Must not be modified by the receiving component.
    - Components must act like pure functions with respect to their props.

Code Bite

```
<Product name="Pulley" price="9.95" />


function Product(props) {
  return (<div class="product-tile">
      <h3>{props.name}</h3>
      <p>Price: {props.price}</p>
    </div)
}
```

Dynamic Content

- JSX content is treated as literal content.
- Expressions placed inside curly braces, { }, are dynamically evaluated.

```
<p>The current date is { new Date() }.</p>
```

- Parent components can put content between opening/closing child component tags.
  - React provides "props.children" to give the child component access to this content.

```
<Product>A very exciting product</Product>

function Product(props) {
      return (<div class="product">{props.children}</div>);
}
```

JSX cheatsheet

- ES6 uses "class" as a reserved word, use "className" instead.

```
<div className="instructions"></div>
```

- JS uses "for" as a reserved word, use "htmlFor" instead.

```
<label htmlFor="firstName">First Name</label>
```

- React will HTML encode content, use "dangerouslySetInnerHTML" to prevent.

```
<div dangerouslySetInnerHTML={{__html: props.postBody}} />
```

- JS logical operators can be used to conditionally render components.

```
<Fragment>
       {showDetails ? <DetailComponent /> : <SummaryComponent />}
       {showPopup && <MyPopup />}
</Fragment>
```

- Style properties can be directly set via a JS hashmap.

```
<div style={ { padding: '1em', marginTop: '20px' }}></div>
```

State

- Data "owned" by the component itself.
- Can be modified by the component.
  - Is "reactive" – i.e., monitored by React to propagate changes and re-render DOM.
  - Should not be directly modified – use setState() instead.
- A property of class-based components.
  - Usually initialized in the constructor.
- Constructor receives props.
  - Must be passed to base class constructor.

Steps

- To convert a functional component:
  1. Create a class that extends React.Component
  2. Give the class a render() method
  3. Move the function's return statement into the render() method
  4. Change "props" references to "this.props"

Logical Imagination

Event Handling

- React uses curly braces to bind DOM events to functions.
  - Functional components' event handlers are nested functions.
  - Class-based components' event handlers are methods on the class.
- React provides an event parameter whose type is SyntheticEvent.
  - Constructed according to W3C spec.
  - E.g., must call preventDefault() instead of returning false.
- Syntax differs from pure HTML syntax.
  - React uses camelCase instead of lower-case (e.g. onClick instead of onclick.)
  - React uses a function instead of a string (e.g. {handleClick} instead of "handleClick()".)

```
<div onClick={this.handleClick}></div>
```

Event handlers and context

- React event handlers often need to reference the component's state and props.
- They are not invoked from any particular context.
  - "this" does not refer to the component instance – typically it is "undefined".
  - There are several ways to fix the problem.
- "bind" the method to the component instance.
  - Accomplished in the constructor.
  - Historically, the most common approach.

```
this.handleClick = this.handleClick.bind(this);
```

Arrow function fixes

- Arrow functions bind the reference to "this" when they are defined.
  - Standard JavaScript functions bind "this" when they are invoked.
  - Newer, cleaner syntax.
- Component method can be defined as an arrow function.

```
handleClick = () => { . . . };
```

- Arrow function can be used for the event handler.
  - Generally, not recommended – a new handler is created each time the button is rendered.
  - Can also cause problems if the method is passed as a prop to lower components.

```
<button onClick={(e) => this.handleClick(e)}>Click Me</button>
```

Passing additional arguments

- Event handlers sometimes need additional data passed to them.
  - Especially within a loop, handlers may need the current element or id.
- Value(s) need to be passed when connecting the event handler.

```
<button onClick={(e) => this.editElement(item, e)}>Edit</button>
//or
<button onClick={this.editElement.bind(this, item)}>Edit</button>
```

- The arrow function needs the event parameter explicitly passed to it.
- "bind" will automatically pass along any additional arguments received.

Mutating state

- Constructors are the only place where state can be explicitly set.
- React needs to know about any changes to existing state.
    - Directly changing state outside a constructor will not re-render components.
- Each component has a method called setState().
    - Provide it with an object to be merged into the component's state.
    - React will schedule the update to occur within its processing sequence.

Code Bite

```
//WRONG – only works in constructor
this.state.message = 'success';

//CORRECT
this.setState({
  message: 'success'
});
//the new message value will be merged
//into any existing state for the component
```

Asynchronous updates

- State updates may be asynchronous.
  - If new state values depend upon current state or props, inaccuracies can occur.
- A second form of setState() exists.
  - Provide it with a callback function, which will be invoked when state is being updated.
  - The callback will be provided with current state and props.
  - The callback must return an object with the new state values.

Code Bite

```
this.setState((state, props) =>({
  counter: state.counter + props.delta
}));

//OR
this.setState(function(state, props) {
  return {
    counter: state.counter + props.delta
  };
});
```

Two-way Binding

- Form inputs are used to display and update data.
- State is updated by React, not directly by component code.
- Props should not be updated at all.
- Therefore, two-way binding is not used in React.
- Instead, input change events are used to setState.
    - To mutate props, child controls must emit their own event.
    - Parent control would then update state.

Form input example

```
class PersonForm extends Component {
  constructor(props) {
    super(props);
    this.state = { name: '' };
  }
  handleNameChange = (evt) => { this.setState({ name: evt.target.value }); }

  render() {
    return (
      <input type="text" value={this.state.name} onChange={this.handleNameChange} />
    );
  }
}
```

React Hooks (new with version 16.8)

- Allow functional components to utilize React features like state, context, refs, etc.
- Are completely optional.
- Will not replace class-based components.
- Can elegantly solve some challenges with class-based components.
  - It's hard to re-use stateful logic between components.
    - Current solutions use render props or higher-order components.
  - Code often duplicated between componentDidMount and componentDidUpdate lifecycle events.
  - componentWillUnmount often cleans up things done in componentDidMount.
  - componentDidMount often contains several unrelated actions.
- React team doesn't expect folks will re-write existing class-based components.
  - Do expect that going forward most (or all) components will be functional.

Exercise 2: React Syntax and Basics

Skill Check

- Locate and complete the instructions for exercise 2 in your student files

Lesson 3: Dynamic Content

In this lesson you will learn about:

- Conditionally Rendering Content
- Collection Content
- Updating State Immutably
- Collections and Keys
- More Flexible Collections

Conditionally rendering content

- Whether content renders can be based upon state or props data.
- JSX is ultimately converted into React.createElement() calls.
    - JSX is just JavaScript.
- Logical operations can be embedded in the JSX expressions.

Code Bite

```
render() {
  return (<div>
    //other JSX content here
    { this.props.showDetails &&
      <div>Some content</div> }
  </div>);
};
```

Using alternate content

- The ternary operator can be used to choose between alternative content.
- Standard "if" statements cannot.
  - Block statements are not allowed in JSX.

Code Bite

```
render() {
  return (<div>
    //other JSX content here
    { this.props.expanded ?
      <div>Some content</div> :
      <div>Other content</div>
    }
  </div>);
};
```

Simpler syntax

- Ternary operators in JSX can lead to convoluted, bloated code.
- Instead, other code can be executed inside the render() method, prior to returning JSX.
- Block statements such as "if" are allowed outside of JSX.
- This keeps the render method simpler.
  - Build chunks of content inside render.
  - Assemble the chunks in the return statement.

Code Bite

```
render() {
  let myContent = null;
  if (this.props.expanded) {
    myContent = (<div>Details here</div>);
  }
  return (
    //other JSX content here
    {myContent}
  );
}
```

Preventing a component from rendering

- In extreme cases, a component may wish to prevent itself from rendering.
- Return null from the render() method (class-based component.)
- Return null from the function (functional component.)

Collection content

- Often need to display content for each element of an array.
- Need to convert the array to JSX content.
- JavaScript array .map() function is helpful.
  - Receives a function as its parameter.
  - Iterates over the elements in the array.
  - Invokes the function for each element.
  - Collects the return values from each of the function invocations.
  - Returns a new array of the return values.

```
{myArray.map(elt => (<myComponent color={elt.color} size={elt.size} />))}
```

Collections and keys

- Each element in an array of content needs a key.
  - A unique string value added as an attribute called "key".
  - Must only be unique within the array, does not need to be globally unique.
  - Only needed on the top-level element (or component) for each array element.
- Allows React to determine which element(s) of the array have been modified.
- Avoids re-rendering the entire list each time a small change is made.
- Index of the element in the array should NOT be used.
  - Not a stable key, since elements may be re-ordered.
  - React may not be able to accurately detect changes in that case.

```
{myArray.map(elt => (<myComponent key={elt.id}
                          color={elt.color}
                          size={elt.size} />))}
```

Potential problem with state updates

- JavaScript arrays and objects are reference types.
- Care must be taken when updating state.
- Natural tendencies are to modify the current state.
  - Mutating the current state object(s) can lead to unstable apps.

```
const current = this.state.myCollection;
current.splice(idx, 1);          //directly mutates the existing state!
this.setState({ myCollection: current });

const obj = this.state.myObject;
obj.prop = newValue;             //directly mutates the existing state!
this.setState({ myObject: obj });
```

Updating state immutably

- Copies of reference types must be made before applying changes.
- Array .slice() method creates a new array with all the same members.
  - The spread operator would also work.

```
const current = this.state.myCollection.slice();
//or current = [...this.state.myCollection];
//changes made to current will NOT mutate the existing state
```

- The spread operator can also create a new object with all the same properties.

```
const obj = { ...this.state.myObject };
//changes made to obj will NOT mutate the existing state
```

More flexible collections

- Mutating state of an object in a collection is a bit more involved.
- Seems like a lot of unnecessary steps.
- But bad things can happen if state is directly mutated.
  - Optimized components will not detect the changes.
  - DOM will not be re-rendered.

Steps

1. A copy of the object must be created
2. The object copy is mutated
3. A copy of the array is created
4. The original array element is replaced with the mutated copy
5. The state can be set using the modified array copy

Exercise 3: Dynamic Content

Skill Check

- Locate and complete the instructions for exercise 3 in your student files

Lesson 4: Styling Content

In this lesson you will learn about:

- Inline Styles
- Dynamically Setting Styles
- Dynamically Setting Class Names
- Using Radium
- Using CSS Modules

Approaches to styling React components

- Global stylesheet
  - Legacy approach – still works, but difficult to maintain.
- Inline Styles
  - Hard-coded styles, specified in JavaScript code in component files.
- Dynamically setting styles
  - Style can change with state, difficult to maintain, no pseudo-class selectors or media queries.
- Dynamically setting class names
  - Style can change with state, relies on global stylesheet for pseudo-class selectors or media queries.
- Using Radium
  - Supports pseudo-class selectors and media queries.
  - Style rules are specified in JavaScript code in component files.
- CSS modules
  - Component-scoped styles stored in CSS files.
  - Requires webpack.

Inline styles

- JSX can include CSS style rules inline on the HTML elements.
- Style rules can be hard-coded on the elements.

```
return (<div style="border: 1px solid black">Hello World</div>);
```

- Style rules can be an expression evaluating to a JavaScript object.
  - Hyphenated style properties would be written in camelCase.

```
render() {
      const styles = { color: 'blue', backgroundColor: 'yellow' };
      return (<div style={styles}>Hello World</div>);
}
```

Dynamically setting styles

- Element styling often needs to change based on component state.
- Objects can be used to set inline styles on JSX elements.
- So properties of those objects could be dynamically set based on state.

```
render() {
        const styles = { color: 'blue' };
        if (this.state.backordered) {
                styles.color = 'red';
        }

        return (<div style={styles}>. . .</div>);
}
```

Using class names

- Setting styles directly violates Separation-of-Concerns.
- Styles should be set by CSS.
  - JavaScript code traditionally adds/removes classes to effect change.

Steps

- In the CSS file
  - Define class-based styles
- In the component's render() method
  - Assign desired class name to a variable
  - Databind to className property for desired elements
  - Component state and props can be used to make styles dynamic
  - Use an array to store multiple class names
    - Use .join(' ') to concat multiple class names

Code-based style difficulties

- Cannot use pseudo-class selectors (directly.)
  - :hover, :before, :after
- Cannot use media queries (directly.)
- These can be put in an external style sheet.
  - Applied in conjunction with a class or id.
- Those styles would be global.
  - Ideally, they should be localized to the content for a particular component.
- 3rd-party package called Radium can solve these problems.
  - Allows pseudo-class selectors and media queries with inline styles.

```
npm install radium
```

Radium

- Need to import Radium in any file where you want to use it.
- Need to export a higher-order component from the file.
    - Execute Radium() on the component class to create the HOC.

Code Bite

```
import Radium from 'radium';

class App extends Component {

}

export default Radium(App);
```

Higher-Order Components

- A pattern used in React programming.
- A function that receives a component and returns a new component.
  - Returned component encapsulates the original.
  - Returned component is called a Higher-Order Component (HOC).
- Often used to share functionality across multiple components.
  - HOC renders the wrapped component.
  - HOC maintains state and behavior modifying that state.
  - HOC passes that state to the wrapped component.
- Can be applied to functional components and class-based components.
- A component transforms props into UI.
- A HOC transforms a component into another component.

Radium and pseudo-class selectors

- Radium allows for pseudo-class selectors to be added as properties of a style object.
- Name of the property is the pseudo-class selector.
  - Must be wrapped in quotes, because of the ":"
- Value of the property is another object containing the styles to be applied.

Code Bite

```
const style = {
  backgroundColor: 'black',
  color: 'white',
  ':hover': {
    backgroundColor: 'blue'
  }
}

render(){
  return (<div style={style}></div>);
}
```

Radium and media queries

- Radium allows for media queries to be added as properties of a style object.
- Name of the property is the media query text.
  - Must be wrapped in quotes, because of the "@", "(", ")", and " ".
- Value of the property is another object containing the styles to be applied.
- Media queries and keyframe animation also require wrapping the entire application in the <StyleRoot> component.
  - Provided by Radium.

Code Bite

```
const style = {
  display: 'block',
  '@media (min-width:45em)': {
    display: 'inline-block'
  }
}

render(){
  return (<div style={style}></div>);
}
```

Problems with CSS

- Ideally, style rules should be stored in CSS files.
- Historically, CSS files tend towards tremendous bloat.
  - Almost always driven by fear and uncertainty.
  - Fear that changes will affect more than the desired elements.
    - So new style rules are added instead of updating existing rules.
  - Uncertainty whether a rule is still being used.
    - So outdated rules are never removed.
- Additionally, no code organization is enforced.
  - Style rules affecting the same (or related) elements can be separated by hundreds of other style rules.

CSS Modules

- An npm package.
- A process (build step.)
- Relies upon Webpack or Browserify.
- Changes class names and selectors to be scoped (similar to namespacing.)
- This way component styles:
  - Live in one place.
  - Apply to only that component.
- Minimizes use of global styles.
- Details at https://github.com/css-modules/css-modules.

Steps

- A JavaScript file imports a CSS file
- Receives an object ref for the style rules
  - Class selectors become properties of the object
- Apply classes to JSX elements
  - Uses the object ref's properties

React and CSS Modules

- Since version 2.0.0, react-scripts (used by create-react-app) has supported CSS Modules.
  - Nothing extra to install or configure.
- Naming conventions:
  - MyComponent.js
  - MyComponent.module.css
- *.module.css will automatically be processed by CSS Modules.
- Prior versions of react-scripts were compatible with CSS Modules.
  - Webpack configuration needed to be ejected and configured.

CSS Modules example

```
//in Button.module.css
.cancel {
  background-color: #aaa;
  color: #000;
}

//in Button.js
import styles from './Button.module.css';

render() {
  return (<Button class={styles.cancel}></Button>);
}
```

Exercise 4: Styling Content

Skill Check

- Locate and complete the instructions for exercise 4 in your student files

Lesson 5: Debugging

In this lesson you will learn about:

- Understanding Error Messages
- DevTools and Sourcemaps
- React Developer Tools
- Error Boundaries

Understanding error messages

- Eventually, something will go wrong in any application.
- Browser JavaScript error messages should be the first place to look.
- React development workflow usually gives good info right on web page.
- Open browser developer tools and check the console.
  - Extensions like Chrome's JavaScript Errors Notifier can help.
  - Scroll to the top to see the first error message.
  - Errors can cascade – deal with the first one first.
- Even if error message is cryptic, the location of the error is usually helpful.
  - The actual error might be a line or two earlier than indicated.
  - Sometimes problems don't surface until later .
  - E.g., you reference the wrong object but don't get an error until 3 lines later when using that object.

DevTools and Sourcemaps

- Browser developer tools have a JavaScript debugger.
  - Allows for breakpoints, stepping through code, inspecting variable values, etc.
  - Especially useful for identifying logic errors.
- However, the browser is not executing the code we wrote.
- Build process converts the source code.
  - ES6 is converted into ES5.
  - JSX is converted to JavaScript.
- Build process also creates sourcemap files.
  - Maps lines of transpiled code to source code.
  - Allows browser to execute modified code and still show source code in the debugger.
  - Can debug the code we wrote even though that is NOT what is executing in the browser.

React developer tools

- Browser developer tools are undeniably helpful.
- It is difficult to sift through all the JavaScript details to assemble a picture of the state of the React app.
- React team has created an extension for Chrome (and Firefox.)
  - Plugs in to the browser devtools.
  - Provides analysis of state, props, etc at the component level.
  - Available as a standalone app for use with Safari, IE and ReactNative.
- Has an Elements tab showing React components and their content.
  - Selecting a component displays its props, state, and event handlers in a side panel.
  - Some state values are even editable.

Error boundaries

- Despite the best design and debugging, runtime errors occur.
- Error screens seen so far are helpful for developers.
    - Are NOT something we want users to see in a live application.
- Error boundaries are typically used in a live application.
    - Components that wraps around other application components.
    - Catch errors in their child component tree.
    - Display fallback UI.
    - Like try-catch, but for declarative code.
- Application might have one or many such components.
    - General error boundary is all that is necessary.
    - Might create special-purpose error boundaries for use in different areas of the app.
- Implement the componentDidCatch lifecycle event to handle the error.

Details of error boundaries

- Error boundaries do NOT catch errors for:
  - Event handlers.
  - Asynchronous code.
  - Server-side rendering.
  - Errors thrown in the error boundary control itself.
- Errors not caught by any error boundary will unmount the whole React component tree.
  - As of React 16.
  - React < 16 left application running with a (possibly) corrupted component tree.
  - Upgrading existing app to React 16 could surface prior non-catastrophic errors to unmount the app!

Error boundary example

```
//inside the error boundary component
state = { hasError: false,
  //other state properties, as necessary
};
componentDidCatch = (error, info) => {
  this.setState({ hasError: true,
    //other state properties, as necessary
  });
};
render() {
  if (this.state.hasError) {
    //render error info here
  } else {
    return this.props.children;
  }
}
```

Exercise 5: Debugging

Skill Check

- Locate and complete the instructions for exercise 5 in your student files

Lesson 6: Components

In this lesson you will learn about:

- Creating Components
- Stateless vs. Stateful
- Component Lifecycle
- Pure Components
- Higher-Order Components
- Validating Props
- Context API

Creating Components

- What goes into its own component?
  - Components should be narrowly-focused.
  - Be guided by SRP and DRY.
- React components fall into one of two categories:
  - Container components.
  - Presentational components.
- Container components – "smart" components:
  - Usually manage data.
  - Render() is rather lean.
- Presentational components – "dumb" components:
  - Are usually given their data as props.
  - Are mainly all about render().

Organizing components

- React applications usually have MANY components.
- Organization of them is essential.
- Many developers organize them by their type:
  - One top-level directory for container components.
  - One top-level directory for presentational components.
- There is not one organization strategy that is recommended as a best practice.
  - Each dev team has their own preferences.
  - Different size projects have different needs.
- General best-practices:
  - Small number of "smart", stateful components.
  - Large number of "dumb", stateless components.
  - Each component narrowly-focused with clear purpose.

Stateless vs. Stateful

- Stateful components:
  - Also called container components, or "smart" components.
  - Historically implied a class-based component (React hooks changes this.)
  - Maintain internal state.
  - Pass state to children via props.

- Stateless components:
  - Also called presentational components, or "dumb" components.
  - Historically implied a functional component.
  - Have no internal state.
  - Receive data as props.
  - Vastly outnumber stateful components (in a well-designed app.)

Why the ratio?

- Best practices suggest creating few stateful components and many stateless.
- Provides predictable data flow.
- Application logic is focused in relatively few places.
  - Easier to debug.
  - Simpler to extend.
- UI is cleanly separated from logic.
- UI is granularly defined.
  - Easily reused.
  - Easily modified.
- UI is highly predictable.

Common State Mistake

- setState() should be used to mutate state.
  - Does not immediately apply state changes.
  - Schedules a state change.
  - React will apply the state change asynchronously – batched with other state changes.
- When new state values depend upon current state, race conditions can occur.

```
this.setState({
	counter: this.state.counter + 1
});
```

- Could result in erroneous state if multiple invocations like this get scheduled at once.

State Solution

- When new state values depend upon current state values, use functional form.
  - Callback function is scheduled instead of data being scheduled.

```
this.setState((prevState, props) => {
      return {
            counter: prevState.counter + 1
      };
});
```

- When state is actually updated by React, the current state is passed in.
  - Avoids race conditions.

Component Lifecycle

- Components go through a sequence of events as React manages each instance.
- Largely restricted to class-based components.
  - React hooks provide something loosely equivalent.
  - Not as precise or flexible as lifecycle.
- A few others exist, but are deprecated.
  - componentWillMount()
  - componentWillReceiveProps()
  - componentWillUpdate()

Code Bite

```
constructor()
getDerivedStateFromProps()
shouldComponentUpdate()
getSnapshotBeforeUpdate()
componentDidUpdate()
componentDidCatch()
componentDidMount()
componentWillUnmount()
render()
```

Why use lifecycle events?

- Lifecycle events are often seen as a last-resort:
    - Are inefficient.
    - Add complexity to the application.
    - Difficult to reason through.
- Typically, we would rather re-arrange components or pass state and props around.
- Lifecycle events are good for integrating non-React tools into a React application.
    - Many JavaScript libraries or modules are not constructed to integrate with React.
    - Lifecycle events provide hooks to invoke such modules' code at appropriate times.

Creation lifecycle events

- When a component instance is created.
- Only invoke side-effects (e.g., HTTP request or store data to localStorage) in componentDidMount().
    - Can block rendering process.
- Do not change state synchronously in componentDidMount().
    - Can trigger a re-render.
    - Can (and do) change state in an async callback from an HTTP request initiated here.

Code Bite

```
constructor(props)
      - ES6 feature
      - call super(props)
      – set up initial state
getDerivedStateFromProps(props, state)
      – rarely needed
      - sync internal state from props
render()
      - prepare and structure JSX code
      - child components will render
componentDidMount()
      - can invoke side-effects
```

Constructor

- A method called constructor().
- Receives a props object.
- Must invoke base class' constructor and pass props.
  - In order for Component to be properly initialized.
- Can initialize state.
  - The only place where state can be directly modified.
  - Do not use setState() in constructor.

Code Bite

```
constructor(props) {
  super(props);
  this.state = {
    name: 'Widget',
    color: 'blue',
    price: 4.95
  };
}
```

getDerivedStateFromProps

- Replaces componentWillReceiveProps().
- Used when a component has internal state that is dependent upon received props.
- Must be a static method.
- Receives both props and state.
- Should return updated state.
- Executes every time parent component re-renders.
  - Regardless of whether props have changed.
- Should be used rarely.
  - Each piece of data should be owned by only one component, and updated by that component only.

Code Bite

```
static getDerivedStateFromProps(props, state) {
  if (props.selected !== state.selected) {
    return {
      ...state,
      selected: props.selected
    }
  }
  return null;  //if state hasn't changed
}
```

Update lifecycle events

- When React is processing data changes.
- Only invoke side-effects (e.g., HTTP request or store data to localStorage) in componentDidUpdate().
  - Can block rendering process.
- Do not change state synchronously in componentDidUpdate().
  - Can trigger a re-render.
  - Can (and do) change state in an async callback from an HTTP request initiated here.

Code Bite

```
getDerivedStateFromProps(props, state)
      - rarely needed
      - sync internal state from props
shouldComponentUpdate(nextProps, nextState)
      - may cancel updating process
render()
      - prepare and structure JSX code
      - child components will render
getSnapshotBeforeUpdate(prevProps, prevState)
      - rarely needed
componentDidUpdate(prevProps, prevState, snapshot)
      - can invoke side-effects
```

shouldComponentUpdate

- Allows component to exit the update lifecycle.
  - React does not deeply compare props or state.
  - Any time either changes, components are re-rendered.
- Receives the new props and state values.
  - Typically compares them against existing props or state.
- Must return true or false.
  - False will tell React to omit re-rendering the component (and its descendants.)

Code Bite

```
shouldComponentUpdate(nextProps, nextState) {
  if (nextProps.myProp !== this.props.myProp) {
    //only proceed with updates if relevant
    //data has changed
    return true;
  }
  return false;
}
```

getSnapshotBeforeUpdate

- Invoked after render() but before changes are actually pushed to the DOM.
- Typically used to capture window state (e.g., scroll position) in order to restore it after DOM changes.
- Should return either null or a value (typically an object.)
    - This value will be provided to componentDidUpdate().

Code Bite

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  if (prevState.blocks.length <
this.state.blocks.length) {
    const isAtBottomOfGrid = window.innerHeight +
window.pageYOffset ===
this.grid.current.scrollHeight;


    return { isAtBottomOfGrid };
  }


  return null;
}
```

componentDidUpdate

- Runs after changes are pushed to DOM.
- Receives previous props and state as well as snapshot (if any.)
- Typically used to push changes to non-React libraries or restore window state from snapshot.

Code Bite

```
componentDidUpdate(prevProps, prevState, snapshot)
{
  if (snapshot && snapshot.isAtBottomOfGrid) {
    window.scrollTo({
      top: this.grid.current.scrollHeight,
      behavior: 'smooth'
    });
  }
}
```

Most-commonly used events

- componentDidMount() and componentDidUpdate()
  - Fetch new data from server (asynchronously!)
  - Interact with non-React libraries.
- shouldComponentUpdate()
  - Prevent non-necessary updates for performance.
- Should every component implement shouldComponentUpdate?
  - No, many components will always update when their parent does.
  - Those components would execute shouldComponentUpdate, always returning true.
    - This would actually hurt performance!
- Many components should implement shouldComponentUpdate.
  - Often will test ALL of their props for changes.
  - That can be tedious code to write, so. . .

Pure Components

- Often components want to filter their updates to only happen when a prop changes.
- Writing code in shouldComponentUpdate to test every prop is tedious!
- Inherit from PureComponent instead of Component for automatic checks of props.
    - No need to even implement shouldComponentUpdate!

Code Bite

```
import React, { PureComponent } from 'react';
class Products extends PureComponent {


  render() {
    //etc.
  }
}
```

Virtual DOM and React Updates

- render() does NOT directly update the browser's DOM.
  - DOM modifications are slow and expensive.
  - Rendering components can cascade and involve many areas of the DOM.
  - Rendering a given component sometimes results in no net changes.
- React maintains a virtual DOM.
  - In-memory only, not onscreen.
  - Modifications to it can happen quickly and cheaply.
    - The browser (and its UI) is not involved.
  - Only includes node properties relevant to React.

React rendering

- Minimizes interactions with the browser's DOM.
- shouldComponentUpdate() actually affects step 1, not step 3!
  - If a component returns false, its DOM fragment from the old virtual DOM is directly copied to the new virtual DOM.

Steps

1. Build a new virtual DOM
2. Compare old virtual DOM to new virtual DOM
3. Apply differences to the "real" DOM
   - If no differences, "real" DOM is not touched!

Rendering Adjacent Root JSX Elements

- React originally required components to return one root element.
    - Often led to extraneous <div>s wrapping multiple elements.
- React@16+ can return multiple root elements in the form of an array.
    - Provided each root element has a unique 'key' attribute.
    - Allow React to properly diff the new and old virtual DOM during updates.
    - Allows `this.props.someArray.map((elt) => (<div>{elt.name}</div>))` to work as root content.
- React@16.2+ can use a non-rendering wrapper element: Fragment.
    - Import Fragment from react.
    - Wrap multiple root content with <Fragment></Fragment>.
        - Technically has one root element.
        - Does not actually render to the DOM.
    - Variation: an empty element is an alias for Fragment.
        - Wrap multiple root content with <>. . .</>.
        - Do not need to import Fragment.

Custom Solution for Adjacent Root Content

- A custom functional component that simply returns its children could work.
- Does not even need to import React.
- Functions just like <Fragment>.
- A component like this is often referred to as a Higher-Order Component (HOC.)

Code Bite

```
const wrapper = props => props.children;


export default wrapper
```

Higher-Order Components

- Function that receives a component and returns another component.
  - A pure function with zero side-effects.
- Typically do little more than wrap around another component.
  - Don't usually add styling.
  - Don't usually add JSX content.
  - Often add some logic.
- Great way to implement cross-cutting concerns.
  - Error handling, logging, persistence, "global" constants, etc.
- Heavily used by 3rd-party React libraries.
- Somewhat of a convention to name them "withXXX.js".

Higher-Order Component Example

```
import React, { Component } from 'react';

const withGatewayToken = (Wrapped) => {
  class HOC extends Component {
    render() {
      return (<Wrapped {...this.props} gatewayToken={523090534693853} />);
    }
  }
  return HOC;
};

export default withGatewayToken;
```

Higher-Order Component Alternate Syntax

```
import React, { Component } from 'react';

const withGatewayToken = (Wrapped) => {
  return props => (
    <Wrapped {...props} gatewayToken={523090534693853} />
  );
};

export default withGatewayToken;
```

Using a Higher-Order Component

```
//the HOC must be imported where needed
import withGatewayToken from '../../hoc/withGatewayToken';

//the exported component must be wrapped in the HOC
export default withGatewayToken(Products);
```

Validating Props and PropTypes

- React can validate props (and their data types) being passed to components.
- Requires a package called prop-types.
  - Is part of React (developed by React team) but not part of React core (must be installed.)

```
npm install prop-types
```

- Passing invalid values to a prop will cause a warning in the browser console.

Steps

- Import PropTypes from 'prop-types'
- Add a "propTypes" static property to your component – value is a hashmap
  - Keys are the prop names
  - Values use PropTypes data values to define types and validation rules for the props

Available PropTypes

- PropTypes.array
- PropTypes.bool
- PropTypes.func
- PropTypes.number,
- PropTypes.object,
- PropTypes.string
- PropTypes.symbol
- PropTypes.node
- PropTypes.element
- PropTypes.instanceOf(ClassName)
- PropTypes.any

Code Bite

```
//after defining class, before exporting
Person.propTypes = {
  name: PropTypes.string,
  age: PropTypes.number,
  nameChanged: PropTypes.func
};
```

Special PropTypes

- PropTypes.oneOf(['Submitted', 'Reviewed', 'Approved', 'Rejected'])
  - Treated as an enumeration.
- PropTypes.oneOfType([PropTypes.string, PropTypes.number])
  - Allow for restriction to a few different types.
- PropTypes.arrayOf(PropTypes.number)
  - Strongly-typed arrays.
- .isRequired is a modifier that can be added to any type.
  - PropTypes.number.isRequired
  - PropTypes.any.isRequired
  - PropTypes.oneOfType([PropTypes.bool, PropTypes.number]).isRequired

Default props values

- PropTypes can create default values for props.
- Add a property called defaultProps to the component.
  - Hashmap with prop names and their default values.

Code Bite

```
//after defining class, before exporting
Person.defaultProps = {
  email: 'unknown',
  phone: 'none',
  restricted: false
};
```

refs

- Component code may need to interact with JSX elements.
- Most direct way is using state/props and databinding.
  - Some operations do not lend themselves to this approach.
- document.querySelector() methods do not work well with React components.
- React recognizes a "ref" attribute on JSX elements.
  - Value is a function that receives a reference to the element.
  - Typically used to store that reference in a component property.
    - Therefore, this only works with class-based components.
  - Other component code could then use that reference to work with the element.

```
<input type="text"
      ref={(el) => { this.nameBox = el }}
      value={this.props.name} />
```

String refs

- A string can be assigned as the value of the ref attribute.
- Component code has access to the JSX element through "this.refs".
  - Hashmap whose keys are the string values assigned to various elements' "ref" attributes.

```
<input type="text"
       ref="nameBox"
       value={this.props.name} />



this.refs.nameBox.focus();
```

The Problem with props

- React architecture encourages hierarchical component structures.
- Stateful components at the top manage data.
- Stateless components in the middle and bottom receive data through props.
- Frequently data managed at the top is only needed at the bottom.
    - Props passes from parent to child.
    - Intermediate layers often receive data they do not use, only to forward it to their children.
    - This is too tightly-coupled for best application maintenance.
- Context API can help avoid this situation.

Context API

- Context is a value (object, string, number) wrapped in a React-created object.
    - React creates a Provider and Consumer for the value.
- Created by invoking React.createContext().
    - Pass in default value as a parameter – it doesn't matter what is passed in.
    - Default content is typically passed in anyway (IDEs can use it for auto-completion.)
    - Usually defined in their own file.
- Imported into a (usually) stateful component.
- Provider element is wrapped around some JSX content.
    - Has a value attribute that sets the initial value (this is why default does not matter.)
    - This initial value is usually still tied to component state.
        - Changes to context data does NOT trigger re-render while changes to state does.
    - That content (and all descendants) have access to the context values.

Accessing Context API

- Descendant components that need access to contact data use the context Consumer.
- Component's render() should return the context Consumer.
- Consumer child content is a function.
  - Receives the context object.
  - Returns the content.
  - Content can use the context data and methods.

Code Bite

```
import MyContext from './myContext';
...
render() {
  return (<MyContext.Consumer>
    {
      (context) => (
        //JSX content here
      )
    }
  </MyContext.Consumer>);
}
```

Alternative Access to Context API

- Consumer only makes context available to JSX.
    - Other code in the component does not have access to context.
- React@16.6 added contextType.
    - Add a static property called contextType to the class.
    - Set its value to the imported context object.
    - Gives the component access through "this.context".
- No longer need the Consumer element wrapping the content.

Code Bite

```
import MyContext from './myContext';

class MyComponent extends Component {
  static contextType = MyContext;

  render() {
    return (
      <div>{this.context.someProp}</div>
    );
  }
}
```

Exercise 6: Components

Skill Check

- Locate and complete the instructions for exercise 6 in your student files

Lesson 7: Web Server Interactions

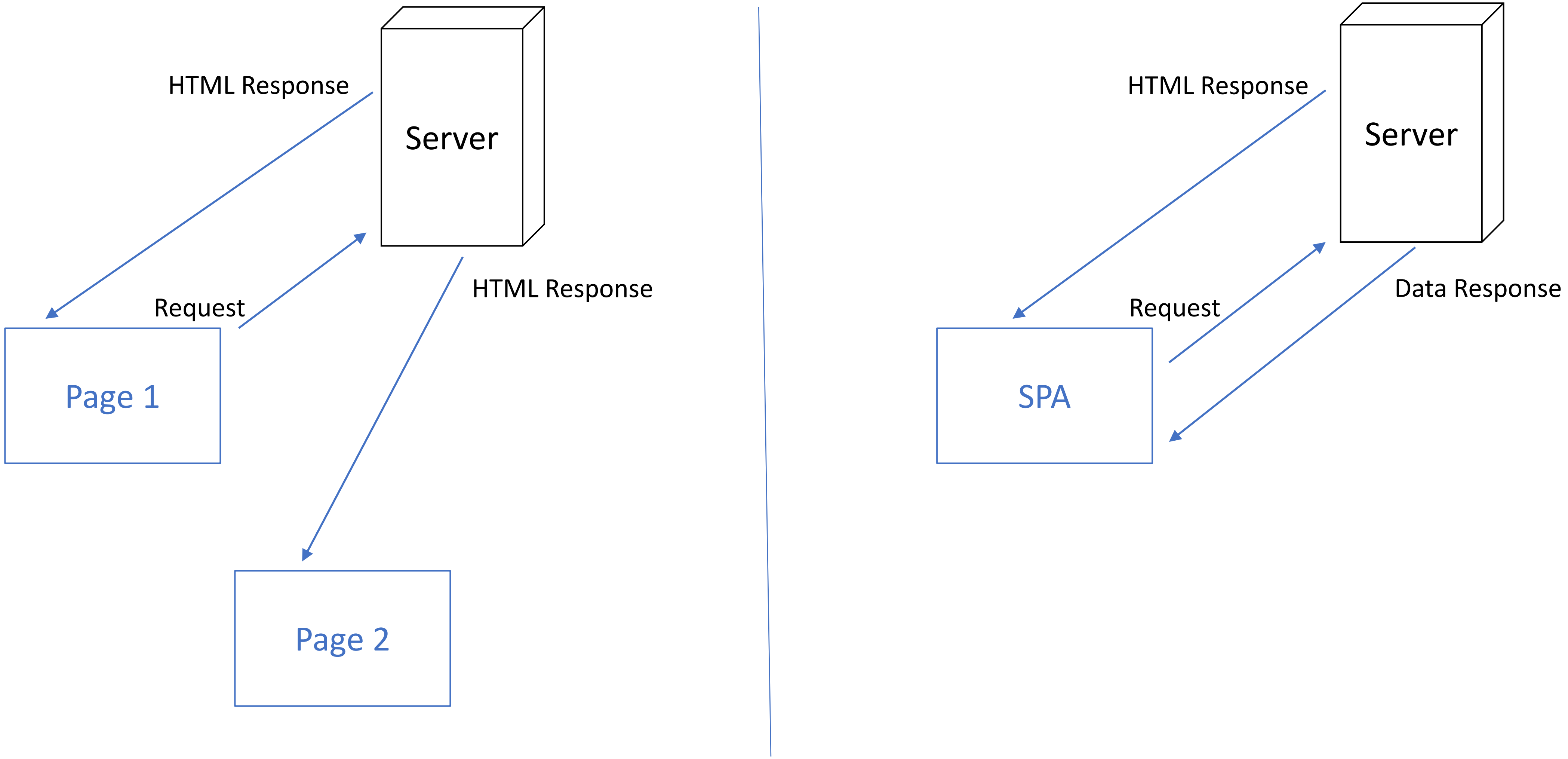
In this lesson you will learn about:

- AJAX Calls
- Using Axios
- Rendering Fetched Data
- Avoiding Infinite Loops
- POSTing Data
- Handling Errors Locally
- Interceptors

AJAX Calls

- React applications typically make AJAX calls to an API.
  - Asynchronously get data from or send data to the web server.
  - Update the UI after operation is complete.
- Used to enhance interactivity of individual web pages.
- Used to create a single-page application (SPA).
  - Only one HTML page loaded by browser.
  - ALL interactions with the web server happen via AJAX.
  - ALL UI updates happen through React.
  - More involved than initially appears.
    - What about URLs and deep links?
    - What about browser history and the Back button?

Full Page Postback vs AJAX



HTML Response

Server

Request

HTML Response

Page 1

Page 2

HTML Response

Server

Request

Data Response

SPA

Making AJAX Calls

- React is JavaScript, so applications can use XMLHttpRequest for AJAX.
  - Doing so is tedious, at best.
- 3rd-party packages make AJAX much easier.
  - Axios is one such package, commonly used.
  - Works well with React, even though not specifically designed for React.
- Need to install Axios to your application.

```
npm install axios
```

Using Axios

- Fetching new data for a component is considered a side-effect.
- componentDidMount() is the appropriate lifecycle event for side-effects.
  - Be sure to update state asynchronously.
- Axios has a get() method to make AJAX GET requests.
  - Requires a param with the URL being requested.
  - Has an optional param for additional configuration.
  - Returns a promise (so .then(), .catch() and .finally() are used to handle the response.)

Code Bite

```
axios.get('/api/products')
  .then(resp => {
    //work with the response here
  })
  .catch(err => {
    //handle any errors here
  })
  .finally(() => {
    //runs regardless of success or failure
  });
```

Rendering Fetched Data

- Data fetched via AJAX is typically put into state.
- State should be modified immutably.
  - Changes should not modify properties of existing state objects.
    - Create new state objects with modifications so reference is changed.
  - Changes should not modify membership in existing arrays.
    - Create new arrays with modified membership so reference is changed.
  - Changes to primitive (value) types can be directly made.
- State is sometimes passed to child components as props.
  - If state is modified properly, re-rendering will be triggered.
- State is sometimes used directly in component's own render() method.

Interactions

- Users often need to interact with elements/component rendered from collections.
- Event handlers need to know which data element should be used.
- Event handlers can be connected with two different syntaxes.
  - Directly connect a function to the event.

    ```
    onClick={handleClick}
    ```

  - Execute an anonymous function (or arrow function) in response to the event.

    ```
    onClick={(evt) => { handleClick(evt); }}
    ```

- The second approach is often useful when interacting with collections.

```
const content = this.state.products.map(product =>
      <Product key={product.id} onClick={(evt) => editProduct(product)}} />
);
```

Avoiding Infinite Loops

- Often when props change applications need to fetch additional data.
  - Remember that doing so is a side-effect.
- componentDidUpdate() is the appropriate update lifecycle event for side-effects.
- Updating state during componentDidUpdate can re-start the update lifecycle.
  - Infinite loop!!
  - Page will often look fine – check the network requests to see continual requests being made.
- Solution: carefully construct conditional checks inside componentDidUpdate().
  - Only load data if state is not in sync with props.
- Other solution: avoid the problem completely.
  - Do not create hybrid components.
  - A component either uses props or manages state, not both.

POSTing Data

- Axios has a post() method to send an HTTP POST request.
  - Params for the URL and the data being sent.
  - Optional parameter for any other configuration.
- Returns a promise.
  - .then() connects a success callback.
  - .catch() connects an error callback.
  - .finally() connects a completion callback.

Code Bite

```
const data = {
  name: 'Widget',
  size: 'medium',
  price: 19.95
};
axios.post('api/products', data)
  .then(resp => { . . . })
  .catch(err => { . . . })
  .finally(() => { . . . });
```

Other Request Types

- Axios has a put() method.
  - Works just like post().
  - Parameters for URL and data being sent.
  - Optional parameter for other configuration settings.
  - Returns a promise.
- Axios has a delete() method.
  - Works just like get().
  - Parameter for the URL.
  - Optional parameter for other configuration settings.
  - Returns a promise.

Handling Errors Locally

- Axios methods return a promise.
- catch() can be used to register an error handler.
  - Log the error.
  - Update the UI.
- Custom state properties could be created to store/display error messages.
  - Works well if further instructions are needed for the user.
- Modern front-end applications often use a toaster to display UI messages.
  - Works well if applications just need to display error messages.
  - react-toastify is one of many that work with React.

```
npm install react-toastify
```

Using react-toastify

- The toaster container and its CSS need to be included just once in the application.
  - The root component is usually used for this.

```
import { ToastContainer } from 'react-toastify'
import 'react-toastify/dist/ReactToastify.css'

//include somewhere in the JSX (it does not render visibly)
<ToastContainer />
```

- Any component can then display toaster messages.

```
import { toast } from 'react-toastify';

toast.error('Some error occurred');    //or .success() or .message(), etc.
```

Interceptors

- Handling errors locally in components makes sense.
  - Components often have unique error-handling needs.
- However there is often need to run specific functions on each AJAX request.
  - Setting common headers (authorization, etc.)
  - Logging responses.
  - Global error handlers.
- All Axios imports in an app share the same configuration.
  - Configuration changes at beginning of application will apply throughout.
- Axios maintains a collection of "interceptors".
  - A function that runs before a request is sent through axios.
  - A function that runs after a response is received through axios.
    - Before the promise is resolved or rejected.

Using a Request Interceptor

- Register a function with the request interceptors of Axios.
  - Receives the request config object.
  - Must return the (modified) request config object – or request will not be sent!
  - Can add HTTP header values, etc.
- Optionally register a second function that will act as a global error handler.
  - Must return Promise.reject() to forward the error info to the original requestor.
  - Will only handle errors occurring while sending the request (e.g., no internet connection.)

Code Bite

```
import axios from 'axios';

axios.interceptors.request.use(req => {
  //modify the request config

  return req;
}, err => {
  //process the error

  return Promise.reject(err);
});
```

Using a Response Interceptor

- Register a function with the response interceptors of Axios.
  - Receives the response object.
  - Must return the (modified) response object – or caller will get no data.
  - Can modify response data, if needed.
- Optionally register a second function that will act as a global error handler.
  - Must return Promise.reject() to forward the error info to the original requestor.

Code Bite

```
import axios from 'axios';

axios.interceptors.response.use(resp => {
  //process response data

  return (resp);
}, err => {
  //process the error
  return Promise.reject(err);
});
```

Global Axios Configuration

- Axios has a "defaults" configuration property.
- Set values on .defaults at application startup.
  - Will be used by all Axios requests.
  - baseURL – prefix for relative request urls.
  - headers.common – added to all requests.
  - headers.post – added to POST requests.
  - etc.

Code Bite

```
axios.defaults.baseURL = '. . .';
axios.defaults.headers.common['Auth']
  = 'MY AUTH TOKEN';
axios.defaults.headers.post['Content-Type']
  = 'application/x-www-form-urlencoded';
```

Custom Axios Instances

- Applications may need to communicate with multiple server APIs.
  - Global axios configuration will only help with one of them.
- Custom instances can be created and configured.
  - One for each API needed.
  - Have their own configuration.
  - Have their own interceptors.
- Have all the same methods as the default instance.
- Use the appropriate custom instance for the API desired.

Code Bite

```
import axios from 'axios';

const instance = axios.create({
  baseURL: 'https://myapi.com'
});

export default instance;
```

Exercise 7: Web Server Interactions

Skill Check

- Locate and complete the instructions for exercise 7 in your student files

Lesson 8: Routing

In this lesson you will learn about:

- Setting Up the Router Package
- Rendering Components for Routes
- Using Routing-Related Props
- Absolute vs. Relative Paths
- Nested Routes
- Route Guards
- Routing and Deployment

What is Routing?

- Replicating the feel of a multi-page application in a single-page application.
    - Want to give the user the same experience, only smoother.
- More than just changing one set of content for another.
    - URLs need to change to allow for bookmarking and emailing deep links.
    - Browser history needs to be involved so users can go back to a prior set of content.
    - Hyperlinks need to trigger loading of new content.
    - JavaScript code needs to be able to trigger loading of new content.
- URLs indicate different content to be displayed.
    - Do not represent actual files.
        - Browser cannot be in charge of loading the content.
    - Do represent specific chunks of content/behavior.
        - JavaScript must be in charge of loading the content.
- Routing is the infrastructure provided by React to make this happen.

Router tasks

- Parse the URL to determine what content/behavior should be loaded.
    - Reads a configuration to know what is available.
- Load appropriate component(s) and JSX.
- Render them in the designated portion of the existing content.
- Make query string parameters (and parameterized URL segments) available to code.

Setting up the Router Package

- The de-facto standard routing package is not created by Facebook.
  - It is the routing package that everyone uses.
  - Installed via npm.
  - Actually, it is two packages, just like React needs two packages.

```
npm install react-router react-router-dom
```

- Use <BrowserRouter> to wrap around the application JSX.
  - Imported from react-router-dom.
  - Router-based data and behavior will be available within this content subtree.

Route component

- Any content inside of <BrowserRouter> can use <Route>.
  - Even nested several components deep.
- Requires a "path" attribute.
  - path is a string.
- Has a "render" attribute.
  - render is a function that returns JSX.
- Is activated whenever the application's URL starts with the path string.
  - Optionally can contain an "exact" attribute.
    - Changes the criteria from "starts with" to "equals".
- Renders its JSX content whenever activated.

```
<Route path="/about-us" render={() => <h1>About Us</h1> } />
```

Rendering Components for Routes

- Route also has a "component" attribute.
  - Can be used instead of "render".
  - Is a reference to a component.
- Instructs Route to render a component instead of ad-hoc JSX.

```
<Route path="/about-us" component={ AboutUs } />
```

- Component can take a function as its value.
  - Avoid – causes a new instance with unmounting and mounting each render pass.
  - Interestingly, render does not suffer from this inefficiency.

```
<Route path="/about-us" component={() => <AboutUs />} />
```

Avoiding Postbacks

- HTML hyperlinks cause the browser to send a request to the server.
  - This causes a full-page postback.
  - React re-initializes and the user loses all application state.
- react-router-dom provides a Link component.
  - Renders like a hyperlink.
  - Invokes client-side navigation, without a postback.
  - Has a "to" attribute – used like the hyperlink's "href" attribute.
- "to" attribute can also be an object.
  - Allows for specification of path, querystring and hash.

```
<Link to="/">Home</Link>
<Link to={{ pathname: 'board', hash='#ceo', search: '?cat=title'}}>Board</Link>
```

Using Routing-Related Props

- Router adds props to every component that it loads.
- "history"
  - Object – has useful methods such as push(), replace(), goBack(), goForward(), etc.
- "location"
  - Object – has pathname, search and hash properties.
- "match"
  - Object – has path, url, params and isExact properties

Router Props in Child Components

- Router only adds props to components it directly loads.
  - Child components of those components do not get router-related props added.
- React router defines a higher-order component (HOC) called withRouter.
  - When applied to a descendant component it will add router props.

Code Bite

```
import { withRouter } from 'react-router-dom';


const myComponent = (props) => {
  return (
    . . . JSX here
  );
}



export default withRouter(myComponent);
```

Absolute vs. Relative Paths

- Absolute paths are appended to the domain name.
  - Regardless of what the current URL is.
- Relative paths are appended to the current URL.
- react-router Link will always create absolute paths.
- Relative paths can be built into Link.
  - Use the object syntax for the "to" attribute.
  - Use the "match" property added to props by router to build the desired url.

```
<Link to={{
      pathname: this.props.match.url + '/relative/path'
      }}>Go Somewhere</Link>
```

Styling the Active Link

- react-router provides a NavLink component.
  - Similar to Link but automatically adds a class to the current link.
  - Adds class="active" to any NavLink that matches the current route.
  - Uses "starts with" to determine if the link matches the route.
  - Adding "exact" switches to using "equals" to determine a match.
- Custom classes can be used with the "activeClassName" attribute.
- Inline styling can be conditionally applied with the "activeStyle" attribute.

```
<NavLink to="/" exact>Home</NavLink>
<NavLink to="/about-us">About</NavLink>
<NavLink to="/contact-us" activeClassName="current">Contact</NavLink>
<NavLink to="/products" activeStyle={{color: red}}>Products</NavLink>
```

Route Parameters

- Routes can be defined with parameterized URL segments.
  - Place a colon (:) in front of the desired parameter name as the URL segment.

```
<Route path="/products/:id" component={ProductDetails} />
```

- Any value in the corresponding URL segment will be considered a match for the path.
  - That value will be made available in router props.
- Routes are processed top-down.
  - Order they are defined can make a difference.
- E.g., an app needs "/products/add" and "/products/:id" to load different components.
  - They need to be defined in that order.

Linking with Parameters

- Links can be dynamically built.
  - Use databinding for the "to" attribute.

```
<Link to={'/products/' + product.id}>Details</Link>
```

- Router can be invoked from code.
  - Adds a "history" object to the props of loaded components.
  - Router navigation is basically a stack of pages.
  - The push() method will tell the router to navigate to a new URL.

```
this.props.history.push('/about-us');
```

Using Route Parameters

- Router keeps track of values of parameterized URL segments.
- Makes them available in props added to loaded components.
  - Also to components using the withRouter HOC.
- match prop contains a "params" property.
  - All values of parameterized URL segments are available on it.
  - Name of the property of "params" is determined by the route definition.

```
mysite.com/products/42

<Route path="/products/:id" component={ProductDetail} />

this.props.match.params.id       //would be "42"
```

Using Switch to Load Only One Route

- Occasionally more than one route pattern can match a given URL.
  - Sometimes that is desired.
  - Most often it is not.
- react-router-dom provides a Switch component.
  - Instructs the router to load only the first match out of its contained routes.

Code Bite

```
import { Switch } from 'react-router-dom';


<Switch>
  <Route path="/products/add"
    component={NewProduct} />
  <Route path="/products/:id"
    component={ProductDetails} />
</Switch>
```

Nested Routes

- Components loaded by router can themselves contain Router elements.
- Same router logic would apply once that component is loaded.
- Routes always use absolute paths.
  - Nested Route elements would need to specify the full path.
  - Can be a maintenance issue.
- Can avoid the maintenance issue by dynamically building the nested route path.

```
<Route path={this.props.match.url + '/:id'} component={ProductDetail} />
```

Route Guards

- Routes may have preconditions that need to be met before displaying.
  - Typically security restrictions.
  - Could be state-related as well.
- One option is to put code in the component(s) being loaded for those route(s).
  - Redirect or display different content if precondition is not met.
  - Works if each component has different preconditions.
  - Inefficient if the same precondition applies to many components.
- Another option is a route guard.
  - Not a function as in some other frameworks.
  - Use conditional statements to render Route components.
  - Fall through to a Redirect component if no Routes get rendered.

Route Guard example

```
import { Route, Switch, Redirect } from 'react-router-dom';


//in the returned JSX:
<Switch>
  {this.state.isAdmin ? <Route path="/products/add" component={NewProduct} /> : null }
  <Route path="/products" component={ProductList} />
  <Redirect from="/" to="/posts" />
</Switch>
```

Routing and Deployment

- React is currently handling all routes.
- For every request, we need the same HTML/JavaScript/CSS returned from the server.
  - React will load the appropriate content via client-side logic.
- Web servers don't usually behave that way.
  - Usually sends different content for "mysite.com/about" vs. "mysite.com/contact".
  - We want it to send the same content for all requests.
- Our dev server is configured to do that already.
  - create-react-app set it up that way.
- Web server hosting the React application needs to be configured this way.
  - Should return index.html from the "public" directory for every request.

Exercise 8: Routing

Skill Check

- Locate and complete the instructions for exercise 8 in your student files

Lesson 9: Forms

In this lesson you will learn about:

- Custom Dynamic Input Components
- Configuring a Form
- Handling Form Submission
- Custom Validation
- Showing Error Messages

Forms in React

- Often two-way binding is enough.
  - Application may not have many forms.
  - Application forms may be rather simple.
  - Minor validation can be written into component methods.

```
<input type="text"
       value={this.state.myVal}
       onChange={(e) => { this.setState({myVal: e.target.value}) } />
```

- Complex validation requirements often call for more.
- Need ways to reduce code duplication and increase reuse.
- Often useful to wrap an individual input in its own component.

Custom Dynamic Input Components

- A custom component is a nice layer of abstraction on the HTML input types.
  - Can render both a label and a form control.
  - Add flexibility/configurability by passing props to it.
  - The spread operator is useful for passing along any arbitrary props.
- Could encapsulate its own style.
- Provides a nice foundation to apply validation, error messages, other behavior.

```
<Input label="Name" placeholder="enter the username you desire"/>
<Input label="Email" type="email" />
<Input label="Password" type="password" />
<Input label="Notes" kind="textarea" />
```

Configuring a Form

- Adding another layer on the custom input components can abstract the form definition.
  - Allow forms to be data-driven.
- Describe the form fields and their characteristics in a hashmap.
- Create utility methods that consume the hashmap and generate JSX.
- Make sure that the onChange event is handled and passed back to base component.
- Allows highly customized solutions per application.
  - Reusable throughout the application.

Handling Form Submission

- HTML forms can be submitted via several triggers.
  - Submit button.
  - Pressing 'enter' while focus is within the form.
  - JavaScript code.
- Handling the click event of a submit button would miss some of those triggers.
- Best practice is to handle the onSubmit event of the form itself.
  - Any of the triggers would fire that event.

```
<form onSubmit={this.handleSubmit}>
```

Custom Validation

- Some new HTML elements and attributes trigger native browser validation.
    - <input type="email" />
    - <input type="url" />
    - <input type="text" required />
    - Etc.
- Most forms will still need some custom validation.
    - React has no built-in validation package.
    - Some 3rd-party packages exist.
    - It is not difficult to create custom functionality.
- Natural to put validation rules in data-driven form configuration.
- Evaluate and enforce the rules before submitting the form data.

Showing Error Messages

- Data-driven form infrastructure is a good foundation for custom error messages.
- Config data is the proper place for error messages to be stored.
- Validation evaluator is the proper place to add/clear messages.
- Custom form control component is the proper place to display messages.

Exercise 9: Forms

Skill Check

- Locate and complete the instructions for exercise 9 in your student files

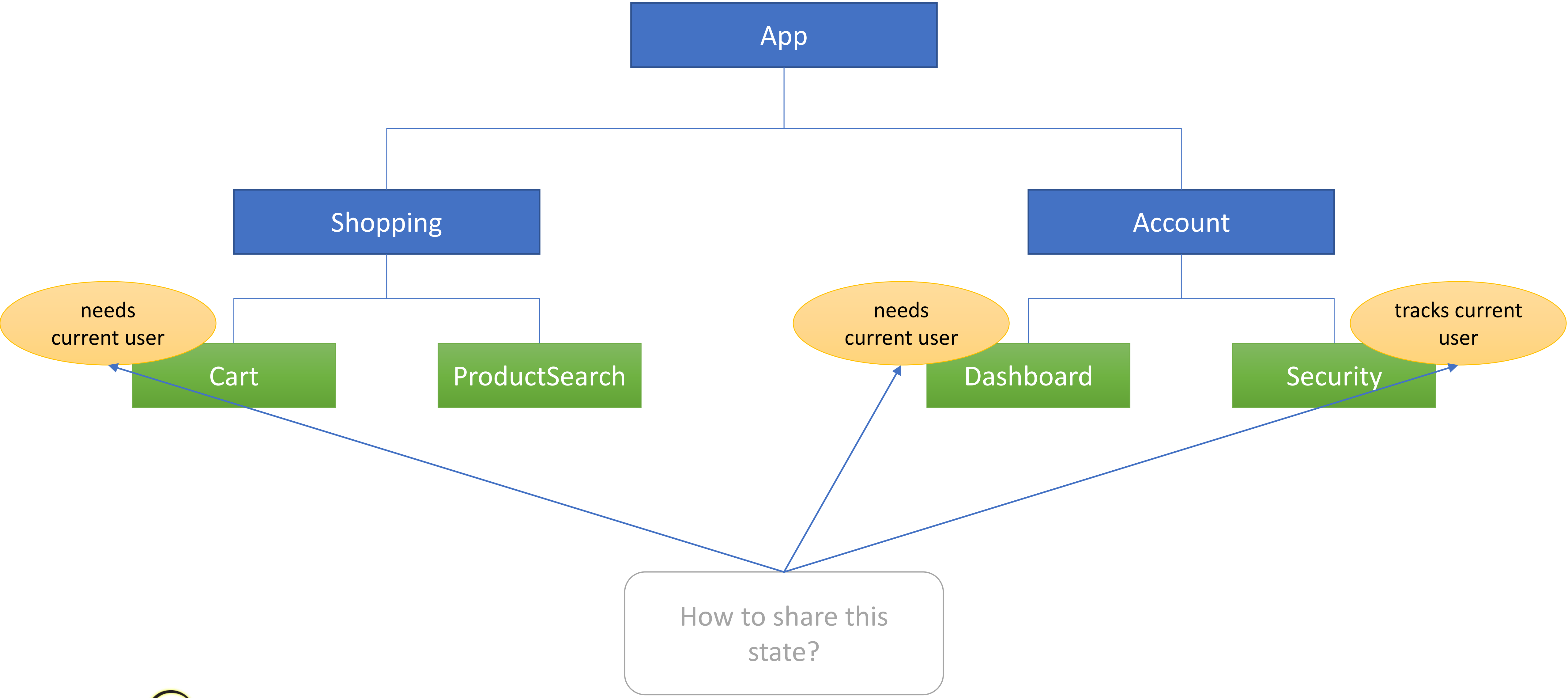
Lesson 10: Managing State with Redux

In this lesson you will learn about:

- Complexity of State Management
- How Redux Works
- Reducer Functions and State Store
- Dispatching Actions
- Creating Subscriptions
- Connecting React to Redux
- Dispatching Actions from Components

State Management

- Is often very difficult.
- Different parts of the UI (often on different screens) need to interact closely.
- Number of interactions between components multiply exponentially.
- Managing the interactions, preconditions and side-effects can be complex.
- React state property is often insufficient to the task.
  - Works within a component, but not between components.
- Can be helpful to move all of the state logic and data outside of the components.
  - Centralize it within one module.
  - Make it accessible to the application components.
- Packages exist to support this behavior.
  - Redux is the official package used by React.



©Logical Imagination Group LLC

How Redux Works

- Stores state data in a central store.
- State is only updated through one path.
- To change state, application dispatches an action.
- A reducer processes the action.
  - Evaluates preconditions.
  - Applies state changes, if appropriate.
- Interested parts of the application subscribe to portions of the state.

Redux Store

- Essentially a giant JavaScript object.
- Can be hierarchical.
- Stores all application state data.
- Is updated immutably.
    - What does that mean?!
    - When state needs to change, a new object is created with modified values.
    - Current state object instance is replaced with new state object instance.
- Application does not directly mutate state.
    - Would be too unpredictable.
    - Need only one update path.
    - State changes need to be predictable and easily maintained.

Redux Actions

- Predefined, named change request.
- A JavaScript object that represents one particular kind of state change.
- May or may not contain a payload.
  - E.g., "AddToCart" action would typically need a payload for the item being added.
  - E.g., "ClearCart" action would not need any payload.

Redux Reducer

- A pure function.
  - Depends only upon its inputs.
  - Always produces the same results for the same inputs.
- Embodies application logic related to state.
  - Evaluates preconditions to determine if action can be taken.
- Is the one place where state is mutated (immutably!)
  - Applies effects of actions to state values.
- Must execute synchronously.
  - Otherwise race conditions would occur.

Reducer Function

- Receives two parameters.
  - Current state – provide a default value for initialization.
  - Action being dispatched.
- Returns the updated state.
  - Return the current state if no changes are desired.
- MUST execute synchronously.
- Must not depend upon anything but its parameters.

```
const reducer = (state = initialState, action) => {
      const newState = { ...state };
      //modify newState appropriately
      return newState;
}
```

State Store

- A JavaScript object.
  - Managed by Redux.
- Created by Redux.
  - Redux has a "createStore()" method.
  - Requires the reducer function as its parameter.
  - Executes the reducer function to initialize the store.
- A container for data only.
  - Does not have any methods.

```
const store = redux.createStore(reducer);
```

Dispatching Actions

- Store has a "dispatch()" method.
  - Requires an action as its parameter.
- Action is an object.
  - Must have a "type" property.
  - Type values must be unique.
  - Convention is to use all-uppercase string for the type value.
  - Optionally can contain any payload needed by the action.
- React will invoke reducer function and pass the current state and the action.
  - Examines the "type" property to determine which action is requested.
  - Branches based on the "type" property to implement application functionality.

```
store.dispatch({ type: 'INCREMENT_COUNTER' });
```

Creating Subscriptions

- Store has a "subscribe()" method.
  - Receives a function as its parameter.
  - Returns a function that will unsubscribe the listener.
- Listener function will be invoked whenever state changes.
  - Has no parameters.
  - Queries state from the store.

```
const unsubscribeListener = store.subscribe( () => {
      console.log(store.getState() );
});
```

Creating the Store in React

- Typically need the store created when app is initialized.
- Index.js is the appropriate place.
    - Before ReactDOM.render() is invoked.
- Reducer function is typically defined in its own file.

Code Bite

```
import { createStore } from 'redux';
import reducer from './store/reducer';


const store = createStore(reducer);
```

Connecting React to Redux

- Need slices of state available for subscriptions in application components.
- Need to be able to dispatch actions from application components.
- A package is available to connect the Redux to React.

```
npm install react-redux;
```

- Wrap main application content with <Provider>.

Code Bite

```
import { createStore } from 'redux';
import reducer from './store/reducer';
import { Provider } from 'react-redux';

const store = createStore(reducer);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root'));
```

Subscriptions in React

- Usually do not manually create subscriptions.
- react-redux provides a method called "connect".
  - Function which returns a higher-order-component.
  - Allows for configuration to be passed in to connect().
- Configuration includes:
  - What slice of state is desired.
  - What actions will be dispatched.

Code Bite

```
import { connect } from 'react-redux';


//define component here

const mapStateToProps = state => {...};

const mapDispatchToProps = dispatch => {...}


export default

  connect(mapStateToProps,
mapDispatchToProps)(MyComponent);
```

Mapping State to Props

- First parameter to connect() is a function.
  - Receives the Redux state and returns a hashmap.
  - Keys of the hashmap become props on the component.
  - Values of the props come from state properties.
- Function effectively maps state properties to component props.
  - Sets up subscriptions behind the scenes.
  - By convention it is called "mapStateToProps".
- When state values change the props are automatically updated.

```
const mapStateToProps = state => {
       return { myProp: state.someProperty };
};
```

Dispatching Actions from Components

- Second parameter to connect() is also a function.
  - Receives the dispatch() method from Redux and returns a hashmap.
  - Keys of the hashmap become methods on the component.
  - Values of the hashmap are methods which typically invoke dispatch().
- Function effectively maps component props to action dispatches.
  - By convention it is called "mapDispatchToProps".

```
const mapDispatchToProps = dispatch => {
      return {
            onIncrement: () => dispatch({ type: 'INCREMENT' }),
            onAdd(amt): () => dispatch({ type: 'ADD', payload: amt })
      };
};
```

OwnProps

- Sometimes the component's props are needed when dispatching actions.
- An optional second parameter receives the component's props.
  - mapDispatchToProps
  - mapStateToProps
- Useful for navigating to a new route after dispatching an action.

Code Bite

```
mapDispatchToProps = (dispatch, ownProps)=>{
  onSelect: () => {
    dispatch({ type: 'SomeAction'});
    ownProps.history.push('/account');
  }
}

mapStateToProps = (state, ownProps) => {
  return {
    newProp: ownProps.prefix + state.name
  }
}
```

Constants and Action Creators

- Action types are used several places.
  - In the reducer function.
  - Wherever they are being dispatched.
- It can be useful to define them in one place.
  - Declare as constants in one file and export them.
- Actions that require payloads can benefit from action creators.
  - Method that creates the action object.
  - Creates the action with the correct type.
  - Can have parameters for any payload.
  - Creates the action with the correct payload property name.

Exercise 10: Managing State with Redux

Skill Check

- Locate and complete the instructions for exercise 10 in your student files

Lesson 11: Async Redux

In this lesson you will learn about:

- Adding Middleware
- Redux Devtools
- Action Creators
- Handling Async Actions
- Action Creators and GetState

Redux and Asynchronicity

- Actions must be synchronous.
  - Return value from reducer function is applied as new state.
  - Async code returns after initiating async action.
  - Callback for async code would get results but have no way to apply it to state.
- Application events often need to invoke async code.
  - Typically want to update state with the results.
- Could run async code in React components and dispatch action in the callback.
  - But the point of using Redux is to centralize state logic.
  - This would only centralize part of it, the rest would be spread throughout components.
- Redux provides a framework to solve this through middleware.
  - Functions that plug in to a process and are executed by that process.

Redux Middleware

- Executes between the dispatching of an action and the reducer execution.
- Are a function which returns a function that also returns a function.
  - Outer function receives the store as a parameter.
  - Middle function receives a pointer to the next middleware as a parameter.
  - Inner function receives the action as a parameter.
- Inner function must execute the next middleware for the action to reach the reducer.
  - Could change the action type or payload.

Code Bite

```
const myMiddleware = (store) => {
  return (next) => {
    return (action) => {
      //do middleware things here
      return next(action);
    };
  };
};

createStore(reducer,
            applyMiddleware(myMiddleware));
```

Redux Devtools

- redux-devtools-extension for the Chrome browser is a valuable tool.
  - Action and state inspectors.
  - Time travel capability.
  - Log monitor.
- Is a Redux enhancer.
  - Must be applied to the store.
- Has an enhancer combinator.
  - Middleware is another type of enhancer.
- compose is also an enhancer combinator.
  - Used as a backup for browsers where the extension is not installed.

Code Bite

```
import { createStore, applyMiddleware,
  compose } from 'redux';

const composeEnhancers =
  window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__
    || compose;

const store = createStore(reducer,
  composeEnhancers(
    applyMiddleware(myMiddleware)
  ));
```

Action Creators

- Recall: action creator is a function that returns the action object to be dispatched.
    - That returned object is typically immediately dispatched.

```
const mapDispatchToProps = dispatch => {
        return { onClick: () => dispatch(createSomeAction()) }
};
```

- Instead, the dispatch method could be passed to the action creator.
    - Async code in the action creator could then dispatch an action at a later point in time.
    - Disadvantage: component needs to dispatch async actions differently than synchronous actions.

```
const mapDispatchToProps = dispatch => {
        return { onClick: () => doAsyncAction(dispatch) }
};
```

Handling Async Actions

- Package called redux-thunk makes this easier.
- Adds middleware to Redux.
- Allows action creators to return a function that will eventually dispatch an action.
  - Function will receive "dispatch" from redux-thunk.
  - Can execute dispatch in async callbacks.
  - Be sure to dispatch a synchronous action to avoid infinite loops!

Code Bite

```
npm install redux-thunk


//when starting application
import thunk from 'redux-thunk';


...createStore(reducer,
      applyMiddleware(myMiddleware, thunk));
```

redux-thunk pseudocode

```
//redux-thunk is middleware that passes along any actions that are objects
//but executes any actions that are functions (without passing them along)

const reduxThunk = (store) => (next) => (action) => {
  if (action is an object) return next(action);

  if (action is a function) return action(store.dispatch, store.getState);
}
```

Combining reducers

- Nice to have multiple different reducer functions for different parts of state.
  - Easier to maintain the code.
- Redux allows only one reducer function.
  - Provides a way to combine several into one.
- combineReducers takes a config hashmap as its parameter.
- Store now becomes an object with each reducer's store being one property.

Code Bite

```
import { createStore, combineReducers }
  from 'redux';
import reducer1 from './store/reducers/one';
import reducer2 from './store/reducers/two';

const rootReducer = combineReducers({
  one: reducer1,
  two: reducer2
});

const store = createStore(rootReducer);
```

Action Creators and GetState

- Action creators may need to access current state.
  - Use state values in sending async requests.
  - Deciding whether to send async requests.
- redux-thunk will pass getState() as a second parameter in async action creators.
- This should not be overused.
- Well-designed actions will usually receive everything they need as parameters.

Code Bite

```
export const myActionAsync = () => {
  return (dispatch, getState) => {
    const currState = getState();
    //decide whether to dispatch or not


  }
}
```

Exercise 11: Async Redux

Skill Check

- Locate and complete the instructions for exercise 11 in your student files

Lesson 12: Testing

In this lesson you will learn about:

- Required Testing Tools
- What to Test?
- Testing Components
- Jest and Enzyme
- Testing Containers
- Testing Redux

Testing Tools

- Test runner:
  - Executes the tests.
  - Provides validation library.
  - create-react-app builds applications pre-configured with a test runner.
  - This course uses Jest.
- Testing utilities:
  - Simulate the React environment.
  - Mounts components.
  - Allows tests to interact with the DOM.
  - Could use React Test Utils.
  - Enzyme is newer and more powerful – recommended by React team.

What to Test? (or what not to)

- No need to test libraries – React, Redux, Axios.
  - E.g., do not test whether React can use props to emit an event to a parent component.
  - Do test whether a specific button click will emit a specific event on its props.
- Do not test interactions between components.
  - E.g., click a button in one component and verify the changes in another component.
  - These are integration tests.
  - They have their place, but that is not what we are covering here.
  - Do test whether updating props for a component triggers appropriate changes in rendered content.
- Do test isolated units.
  - Reducer functions.
  - Components – especially conditional content.

Test Syntax

- create-react-app test scripts look for files that end with ".test.js".
    - Convention is to put them in the same directory as the file being tested.
    - E.g., MyComponent.js would be tested inside of MyComponent.test.js.
- Jest defines a function called describe().
    - Used to create test suites.
    - First parameter is the name of the test suite.
    - Second parameter is a function that contains the tests included in the suite.
- Jest defines a function called it().
    - Used to create tests.
    - First parameter is the name of the test.
    - Second parameter is a function that contains the test.
- Can include multiple suites in a file and can nest suites.
- Can include multiple tests in a suite.

Test Syntax example

```
describe('my first suite of tests', () => {

  describe('a nested suite', () => {
    it('should do something', () => {     });

    it('should calculate correctly', () => {    });
  });

  it('should work properly', () => {    });

});
```

Why Enzyme?

- Need to test components in isolation.
    - Not within the framework of the application.
- Enzyme has utilities for this.
- Enzyme must interface with React.
    - Enzyme has a configure() function.
    - Uses an adapter to connect to React.
- Enzyme has two methods to create component instances.
    - shallow() – renders child components as placeholders and omits some lifecycle events.
    - mount() – fully renders children and triggers all lifecycle events.

Code Bite

```
import { configure, shallow } from 'enzyme';
import Adapter from
        'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```

Examining Rendered Content with Enzyme

- shallow() returns a ShallowWrapper, mount() returns a ReactWrapper.
  - Both implement a similar API for searching content.
  - .find(selector), .closest(selector), .first(), findWhere(predicate) all return wrappers.
  - .containsMatchingElement(JSX), .hasClass(className) , .every(selector) all return a Boolean.
  - .getElement(index) returns a ReactElement.
  - .props(), .prop(key) return props values.
  - .setProps(obj), .setState(obj).
  - many more
- Selectors flexible syntax can search the content.
  - by CSS selectors.
  - by component constructors.
  - by component display name.
  - by property values.

Assertions

- Jest provides the expect(value) function to make assertions.
  - Parameter is the value being tested.
- Has many matchers to define expectations.
  - .toEqual(value), .toStrictEqual(value)
  - .toBeNull(), .toBeDefined(), .toBeNaN()
  - .toBeTruthy(), .toBeFalsy()
  - .toBeGreaterThan(value), .toBeLessThan(value)
  - .toBeCloseTo(number, numDigits?)
  - .toBeInstanceOf(class)
  - .resolves, .rejects
  - .toHaveBeenCalled(), .toHaveBeenCalledTimes(number)
  - .toMatch(regex)
  - .not.
  - many, many more

Asynchronous Tests

- If the behavior being tested is asynchronous, Jest can handle it several ways.
- Return a promise from the test.
  - When the promise is resolved Jest will collect the test results.
- Receive a "done" callback function as a parameter to the test.
  - When the function is invoked Jest will collect the test results.
- Use the .resolves or .rejects matcher in the test's expect statement.
- Use async/await keywords in creating the test.

Testing Components

- Create the component instance with either shallow() or mount().
- Use setProps() to set any necessary props for the test.
- Use setState() to set any necessary initial state for the test.
- Use the wrapper API to find the content of interest.
- Use the assertions to verify the content is correct.
- Functional components are the most straightforward to test.
  - Depend only on their props.
  - Have no state.

Additional Jest Utilities

- Jest provides test suite initialization and cleanup functions.
  - Can handle asynchronicity the same ways that tests can.
- beforeEach(fn)
  - fn will be executed repeatedly, once before each test.
- afterEach(fn)
  - fn will be executed repeatedly, once after each test.
- beforeAll(fn)
  - fn will be executed once, before any of the tests.
- afterAll(fn)
  - fn will be executed once, after all of the tests.

Jest and Enzyme

- Full Jest documentation is available at https://jestjs.io/docs/en/getting-started.
  - Details about all matchers.
  - How to create asynchronous tests.
  - Setup and teardown.
  - How to mock dependencies.
  - Snapshot testing.
- Enzyme documentation is available at https://airbnb.io/enzyme/docs/api/.
  - Guides for working with different testing frameworks.
  - Guides for integrating with different versions of React.
  - API reference.
  - Both shallow and full DOM rendering.

Testing Containers

- More complex since they are often connected to Redux.
  - But Redux simply maps data and methods to props.
  - Do not need to test whether Redux works.
  - Therefore, we just need access to the class that is wrapped in Redux's connect HOC.
- Trick is to export the class from its file as a named export.
  - In addition to the default HOC-wrapped component.
- Create tests on the named export.
  - Tests the class only.
  - mapStateToProps and mapDispatchToProps will not apply when testing.

Testing Redux

- State is just data – no behavior to test.
- Reducer functions should be tested.
    - State logic should reside there.
    - Synchronous, so they are simple to test.
    - Are pure functions, so each action can be tested in isolation.
    - Complex chains of actions need not be tested.
- Action creators may need to be tested.
    - If there is any logic or data transformation there.
    - In most cases they should not contain anything needing testing.

Exercise 12: Testing

Skill Check

- Locate and complete the instructions for exercise 12 in your student files

Lesson 13: Transitions and Animations

In this lesson you will learn about:

- Using CSS Transitions
- Using CSS Animations
- React Transition Group
- Using the Transition Component
- Wrapping the Transition Component
- Animation Timing
- Transition Events

Animations

- More than just something that would be nice to add to an application.
- A critical part of the user experience.
  - Guide user's attention.
  - Help teach the user how to use the application (affordances).
- Provide a smoother, app-like experience to web applications.
- Add to the feel of quality of the application.
- Should be used purposely, not gratuitously.
- Should not provide false affordances.
  - Do not make a feature look like something the users might be familiar with if it does not behave that way.

Using CSS Transitions

- Can be an effective tool for animating.
- Has (almost) nothing to do with React.
- Instructs CSS engine to make changes to CSS properties gradually instead of instantly.
  - CSS rules specify which properties of which elements and how long.
  - When rules applying to those elements change properties, they are animated.
  - E.g., when adding/removing classes or hover state changes.

Code Bite

```
.nav-link {
  background-color: #eee;
  transition: background-color 1s ease-in;
}
.nav-link:hover {
  background-color: #ccc;
}
```

Using CSS Animations

- Also change style properties gradually.
  - Provides more control than transitions.
- Keyframes allow specification of beginning state and ending state.
  - Optionally any number of states in between.
  - Can animate any number of properties.
- Style rules then apply the animation over a duration.
  - Can play forwards or backwards.
  - Can repeat any number of times.
  - By default, plays forwards and then reverts to beginning state.

Code Bite

```
@keyframes slidein {
  0% { margin-left: 100% }
  100% { margin-left: 0 }
}

.my-div {
  animation: slidein 1s forwards;
}
```

Limitations of CSS

- Animated elements are still part of the DOM.
  - May be invisible, but they are present.
  - May mask other elements leading to need to adjust z-index.
- Can result in a very cluttered DOM.
  - In extreme cases it can impact performance.
- Conditionally rendering content is a possible partial solution.
  - Entrance animations would still play.
  - Exit animations would NOT play.
  - React removes the content from the DOM without waiting for any animation to play.
- Plugins are available to address these issues.
  - React Transition Group
  - React Motion
  - React Move
  - React Router Transition

React Transition Group

- A package created by the React community.
  - Not officially part of React.

```
    npm install react-transition-group
```

- Manages component states over time.
  - Including mounting and unmounting.
  - Designed with animation in mind.

Transition component

- Named export from react-transition-group.
- Manages internal states:
  - ENTERING
  - ENTERED
  - EXITING
  - EXITED
- Provides for specification of styles for each state.
- Requires an "in" prop to indicate whether its state should be ENTERED or EXITED.
- Requires a "timeout" prop to specify the transition duration.
  - Between ENTERING and ENTERED states.
  - Between EXITING and EXITED states.

Transition Component

- Content should be a function.
  - Receives the Transition (internal) state.
  - Returns JSX content.
- State received by the function is either.
  - entering
  - entered
  - exiting
  - exited
- Has mountOnEnter and unmountOnExit properties.
  - Add/remove the content to the DOM based on state.

Code Bite

```
import { Transition } from
    'react-transition-group';

<Transition in={this.state.someValue}
  timeout={600}>
  {state => <div>Some content</div> }
</Transition>
```

Using the Transition Component

```
<Transition
    in={this.state.showPopup}
    timeout={500}
    mountOnEnter
    unmountOnExit>
  { state => <MyComponent animState={state} clicked={this.handleClick} /> }
</Transition>
```

Wrapping the Transition Component

- Transition component can be used inside the JSX of a container.
  - Determines whether to include a presentational component.
- Transition could also be used inside a presentational component.
  - Presentational component would always be rendered in the container component.
  - Transition would be the top component rendered in the presentational component.
  - It would determine whether to render any other content for that component.
  - Transition component is considered to "wrap" the presentational content.
- Often just a stylistic choice of which approach to use.
  - Which component is viewed as "owning" the animation.

Animation Timing

- Timeout property specifies duration between "entering" and "entered" states.
  - CSS animations triggered by these states may take a different duration to play.
  - If timeout is too short not all of the animation may play.
  - If timeout is too long there will be a duration with no animation playing.
  - Often that is not a problem, but sometimes it may be.
- Different durations can be set for entering and exiting.

Code Bite

```
const timing = {
  enter: 500,
  exit: 1000
}




<Transition timeout={timing} . . . >


</Transition>
```

Transition Events

- Occasionally custom code needs to be coordinated with the animations.
- Transition component raises six custom events during its lifecycle.
  - onEnter(node, isAppearing)
    - Fires before "entering" status is applied. isAppearing indicates whether this is the initial mount.
  - onEntering(node, isAppearing)
    - Fires after "entering" status is applied. isAppearing indicates whether this is the initial mount.
  - onEntered(node, isAppearing)
    - Fires after "entered" status is applied. isAppearing indicates whether this is the initial mount.
  - onExit(node)
    - Fires before the "exiting" status is applied.
  - onExiting(node)
    - Fires after the "exiting" status is applied.
  - onExited(node)
    - Fires after the "exited" status is applied.

CSSTransition component

- Another version of Transition.
- Does not wrap around a function.
- Requires a "classNames" property.
- Adds classes to the wrapped element.
- Classes use the classNames as base text.
    - *base*-appear
    - *base*-appear-active
    - *base*-enter – removed after one frame
    - *base*-enter-active – during animation
    - *base*-enter-done – after animation done
    - *base*-exit – removed after one frame
    - *base*-exit-active – during animation
    - *base*-exit-done – after animation done

Code Bite

```
import { CSSTransition } from
    'react-transition-group';

<CSSTransition . . . classNames="slide-in">
  (JSX contents here
</CSSTransition
```

Animating Lists

- Can be difficult to animate adding and removing elements to lists.
  - Especially inserting elements in the middle.
- The TransitionGroup component was designed for this purpose.
- Render <TransitionGroup> instead of the list container element.
  - Can use "component" property to tell it to render as a particular HTML element.
  - Only works with <Transition> or <CSSTransition> child elements.
  - TransitionGroup automatically handles the "in" property in its Transition children.
  - As with any collection a "key" prop is required on each child for proper rendering.

Animating Router Transitions

- Can wrap <Switch> element with both <TransitionGroup> and <CSSTransition>.
- A few tricks are required:
  1. <CSSTransition> will be re-used, so it needs a key that changes with the URL.
  2. Exiting <CSSTransition> will change to the new URL (new content will show twice) so <Switch> needs its location prop explicitly set to "props.location".
  3. To make router animations smooth, the content usually needs to be absolutely positioned.

Exercise 13: Transitions and Animations

Skill Check

- Locate and complete the instructions for exercise 13 in your student files

Lesson 14: Introduction to Hooks

In this lesson you will learn about:

- What are React Hooks?
- Getting Started with UseState()
- Updating State
- Multiple States
- Rules of Hooks
- Passing State Across Components

What are React Hooks?

- Hooks are special functions added to React 16.8+
  - They extend to functional components state and other React features otherwise limited to class-based components.
  - They are ONLY usable within functional components and other hooks.
- Hooks allow you to create functional components in situations where you previously were limited to class-based components.
- Hooks basically provide a more direct API for React concepts such as props, state, context, refs, and lifecycle events.
- Hooks are completely optional features.
  - There are no plans to remove class-based components from React.

Why React Hooks?

- It is difficult to re-use stateful logic in React applications.
  - This complicates the implementation of cross-cutting concerns.
- Higher-Order Components (HOC) try to solve this problem.
  - But HOCs require you to change the component hierarchy and make code difficult to follow.
- Complex components often have fragmented code.
- Each lifecycle event can have a mix of unrelated code.
  - componentDidMount may load data as well as set up event listeners.
- Related code can be spread out across multiple lifecycle events.
  - componentDidMount and componentDidUpdate often duplicate the loading of data.
  - componentWillUnmount often cleans up resources established in componentDidMount.
- Hooks allow you to encapsulate related code and isolate it from unrelated code.

The Nature of React Hooks

- Functional components do not inherently have any persistence between rendering.
  - Each time they are rendered, they are executed anew.
  - Anything they define gets redefined each time they are executed (rendered.)
  - They do not inherently save (reuse) any state, data or functions across multiple renderings.
- The basic functionality of React hooks is to externalize something from within a functional component.
- This will allow that externalized resource to be persisted and re-used on subsequent renderings.
  - That opens the door to persist state, data, references, and function definitions across multiple renderings.

Getting Started with useState()

- The most commonly-used hook is useState().
- It extends the storage of state to functional components.
- useState() is a function that will create a state store.
- As with all hooks, it can only be used inside a functional component or within another hook.
- It should be invoked as the first action inside the functional component.
- It differs from class-based component state in two important respects:
  - It does NOT have to store an object – it can store a simple value.
  - Its updates do NOT merge – the entire state must be provided when updating.

Using useState()

- The parameter provided becomes the initial value of the state.
- It returns an array containing two values.
  - The first is the current value of the state.
  - The second is a function that will update the state.
- Array destructuring is typically used to extract the two array values into individual variables.
- Although it is executed each time the component renders, the parameter is only used on the first render.

Code Bite

```
import React, { useState } from 'react';


export default myComponent = (props) => {
  [counter, setCounter] = useState({
    value: 0, delta: 1
  });


  return (
    <div>Counter = {counter.value}</div>
  );
}
```

Updating State

- The second value in the array returned from useState() is an update function.
- When invoked, it will replace the stored state with the value it receives as its parameter.
  - This will trigger a re-render, just as does a state change in a class-based component.
- The provided value is not merged with any existing state – it replaces existing state.

Code Bite

```
const resetClickHandler = () => {
  setCounter({
    value: 0, delta: 1
  });
}

return (
  <div> Counter = {counter}
    <button onClick={resetClickHandler}>
      Reset</button>
  </div>
);
```

Updating State - continued

- As with class-based state, race conditions can occur during updates.
- When new state depends upon current state, the functional form of the update function should be used.
    - Provide a function instead of a data value when invoking the update method.
    - The provided function will be called when the state update occurs and will be given the current state at that time.

Code Bite

```
const incrementClickHandler = () => {
  setCounter(prev => {
    value: prev.value + 1, delta: prev.delta
  });
}

return (
  <div> Counter = {counter}
    <button onClick={incrementClickHandler}>
      Increment</button>
  </div>
);
```

Logical Imagination

Multiple States

- Hook-based state changes replace current state instead of merging.
  - This leads to including current values for non-changing attributes.
- Hook-based state can store data that is not an object.
- State properties that do not change together can be broken up into separate state.
  - With hooks, it is quite common to store each piece of data in its own state.

Code Bite

```
import React, { useState } from 'react';

export default myComponent = (props) => {
  [counter, setCounter] = useState(0);
  [delta, setDelta] = useState(1);

  return (
    <div>Counter = {counter}</div>
  );
}
```

Rules of Hooks

- In order for React hooks to work properly, a few rules must be followed:
- They can only be invoked within functional components or within custom hooks.
- They can only be invoked at the top level of the component or hook.
  - They cannot be invoked within a nested function.
  - They cannot be invoked within a loop or conditional block of code.
- There is a linter plugin available to alert you to violations of these rules.
  - https://www.npmjs.com/package/eslint-plugin-react-hooks

Passing State Across Components

- Whether state comes from a class-based component or from a hook, it can be passed between components the same.

Code Bite

```
const [counter, setCounter] = useState(0);

<button onClick={props.onSelected(counter)}
  >Click Me</button>
```

Exercise 14: Introduction to Hooks

Skill Check

- Locate and complete the instructions for exercise 14 in your student files

Lesson 15: Side Effects

In this lesson you will learn about:

- Sending HTTP Requests
- useEffect() and Loading Data
- useEffect() Dependencies
- Avoiding Infinite Loops with useCallback()?
- Refs and useRef()
- Cleaning up with useEffect()

Sending HTTP Requests

- Components often need to interact with HTTP APIs
- Axios is a common module used for this purpose
- The Fetch API is another viable alternative
- AJAX calls can be made within functional components.
- Beware – they will execute every time the component is rendered.
- If you update useState() with fetched data, this will force a re-rendering.
    - This can lead to an infinite loop!

The Fetch API

- The fetch() function is built into all modern browsers.
- It accepts a URL and an optional configuration object and returns a promise.
    - The configuration object allows specification of HTTP method, header values, CORS mode, and credentials, among other settings.
- The response object passed into the promise contains a json() method to parse the results as JSON.

Code Bite

```
fetch('https://example.com/myapi', {
  method: 'POST',
  body: JSON.stringify(myObj),
  mode: 'cors'
}).then(resp => {
  return resp.json();
}).then(data => {
  // process data here
}).catch(err => {
  // handle errors here
});
```

The useEffect() Hook

- React components often need to carry out side-effect actions.
  - e.g. fetching data, establishing subscriptions, manually changing the DOM.
  - Any operation that affects something outside the component and cannot be done during rendering.
- useEffect() is another hook provided by React.
- The same rules apply to its use as to the use of useState()
  - It can only be used at the top level of a functional component or another hook, etc.
- useEffect() requires a function as its first parameter.
  - This function will be executed AFTER the component has rendered.
  - This function will be executed each time the component renders, by default.
- useEffect() can accept an optional second parameter.
  - It expects an array of dependencies.
  - The function provided to useEffect() will only execute when any of these dependencies has changed.

useEffect()

- useEffect() functions without any dependencies will execute after each rendering.
  - This is similar to componentDidUpdate.
- useEffect() functions with dependencies will execute after any rendering where at least one of the dependencies has changed.
- useEffect() functions with an empty array of dependencies will execute only after the first rendering.
  - This is similar to componentDidMount.

Code Bite

```
const [isOpen, setIsOpen] = useState(false);

useEffect(() => {
  console.log('component has rendered');
});

useEffect(() => {
  console.log('isOpen has changed');
}, [isOpen]);
```

Loading Data with useEffect()

- Class-based components often load their data in the componentDidMount lifecycle hook.
- Functional components can do the same with useEffect().
  - Just be sure to include an empty array of dependencies to prevent an infinite loop!

Code Bite

```
const [items, setItems] = useState([]);


useEffect(() => {
  fetch('https://example.com/api')
  .then(resp => resp.json())
  .then(data => {
    setItems(data);
  });
}, []);
```

useEffect() Dependencies

- The array of dependencies will prevent the side effect from executing on every render of the component.
- This helps by at least making the component more efficient by not executing code unless it is necessary.
  - In extreme cases, it can prevent infinite loops from occurring.
- Strategic use of useEffect() can also provide opportunity to keep related code in one place rather than spread through several lifecycle events.

useEffect() Dependencies Scenario

- Consider a class-based component with search capabilities.
- In componentDidMount it may load the entire set of data to display.
- Its search text box will have an onChange handler, with a main effect and a side effect.
  - The main effect is to store the search term in state.
  - The side effect is to query the component's API for a subset of data to display.
- Thus, data-loading code is spread across two different event handlers.
- If either of those handlers set up a subscription, then componentWillUnmount needs to release that subscription.
  - This spreads the data-related code over yet another event handler.
- It would be more maintainable if that code were all in one place.

useEffect() Depencencies

- The effect executes the first time the component renders.
- The change event for the text box can focus solely on its primary effect.
- The changing of the search term triggers a re-render.
- The re-render will cause the effect to be evaluated, since the dependency (term) has changed.
- Data-loading code is centralized in one place (the effect.)

Code Bite

```
const [term, setTerm] = useState('');

useEffect(() => {
  // load (possibly) filtered data
}, [term]);

return (
  <input value={term}
    onChange={e => setTerm(e.target.value)}
);
```

Props as a Dependency

- Your effect may have a dependency on part of the props handed to your component.
- Listing "props" as a dependency for useEffect() can result in the effect executing much more often than required.
  - Changes to any of the props will force it to execute.
- The solution is to use object destructuring to extract only the specific props values that are dependencies.

Code Bite

```
const { valueOfInterest } = props;


useEffect(() => {
  // effect code here
}, [ valueOfInterest ]);


// instead of:
useEffect(() => {
  // effect code here
}, [ props ]);
```

Avoiding Infinite Loops - scenario

- Listing the dependencies for useEffect() can avoid most infinite loop situations.
- However, a more complex situation can give rise to infinite loops:
- A parent functional component defines an event-handling function that mutates its state.
  - It passes the event-handling function to a child component as props.
- The child component receives the event-handling function as one of its props.
- The child component invokes the event-handling function inside of useEffect().
  - The child component does list the event-handling function as a dependency in useEffect().
- When the effect occurs, the child component invokes the event-handling function.
  - The event-handling function executes in the parent component, changing its state.
  - The parent component re-renders because of the change in state.
  - It is a functional component, so it executes again, creating the event-handling function anew.
  - When the child component renders, it sees the change in the event-handling function, and executes the effect again, starting the infinite loop all over again.

Avoiding Infinite Loops with useCallback()

- In the parent component, a safe (memoized) callback function is created with useCallback().
  - It can then be passed to child components.
  - It will only be re-generated if any of its dependencies change.
- The use of useCallback() is considered a best practice.

Code Bite

```
import React, { useCallback } from 'react';

const somethingHandler = useCallback(() => {
  doSomething(a, b);
}, [a, b]);

return (
  <Child
    onSomething={somethingHandler}
  ></Child>
);
```

Refs and useRef()

- Functional components are re-created every time they are executed (rendered.)
- Effect code that wants to use a ref to a UI element will lose that ref, because a new ref variable will be created on each render.
- The useRef() hook externalizes the reference and updates it every time the UI is rendered.
  - The ref's "current" property will point to the desired element from the current rendering.

Code Bite

```
const elemRef = useRef(null);


useEffect(() => {
  const val = elemRef.current.value;
}, [elemRef]);


return (
  <div>
    <input ref={elemRef} type="text" />
  </div>
);
```

Other Uses for useRef()

- useRef() can also be used to emulate instance variables.
  - It simply creates a plain JavaScript object, albeit one that is managed by React.
- The objects created by useRef() can persist any desired data between renderings.
  - The "current" property is mutable.
- The useRef() object does NOT notify you when content changes.
  - Modifying it will NOT cause a re-render.

Code Bite

```
const myRef = useRef(0);

useEffect(() => {
  myRef.current += 1
}, [myRef]);

useEffect(() => {
  if (myRef.current > 10) {
    myRef.current = 0;
  }
}, [myRef]);
```

Cleaning Up from useEffect()

- Effects are often invoked repeatedly and are often asynchronous.
- If the previous invocation of the effect has not completed when it is invoked again, you may wish to cancel the previous execution.
- Effects can return a value.
  - If they do, it must be a function.
- This returned function will be invoked immediately prior to the next invocation of the effect.
- If the effect has [] as dependencies, the cleanup function runs when the component is unmounted.

Code Bite

```
useEffect(() => {
  const resource = setupResource();

  return () => {
    resource.unsubscribe();
  }
}, []);
```

Exercise 15: Side Effects

Skill Check

- Locate and complete the instructions for exercise 15 in your student files

Lesson 16: Reducers and Context

In this lesson you will learn about:

- Understanding State Batching
- Understanding useReducer()
- useReducer() and HTTP State
- Working with useContext()
- Performance Optimization with useMemo()

Understanding State Batching

- React does NOT apply state changes immediately – they are scheduled.
  - This applies to class-based component state update using this.setState().
  - This applies to functional component state update using useState().
- Invoking multiple state change functions in the same synchronous execution cycle (i.e. the same function) will NOT trigger multiple renderings.
  - Feel free to set as many different states as necessary in each function.
- New state values will NOT be available until the next component render cycle.

Code Bite

```
const [counter, setCounter] = useState(0);


const clickHandler = () => {
  console.log(counter);  //prints 0
  setCounter(12);
  console.log(counter);  //still 0
};
```

Managing State

- useState() allows functional components to hook into React stateful behavior.
- State batching reduces performance penalties for multiple modifications to state.
- This can lead to functional components that have many independently-managed bits of state.
- When multiple bits of state need to change together, it can result in complex code.
    - The same state values may be mutated in multiple effects or event handlers.
- useReducer() can help simplify the code in those situations.

Understanding useReducer()

- useReducer() allows the Redux pattern to be applied within a functional component.
- A reducer function is created OUTSIDE the functional component.
  - So that it does not get regenerated on each rendering.
- It receives the current state and an action and returns the new state.
- State should be changed immutably.

Code Bite

```
const myReducer = (state, action) => {
  switch(action.type) {
    case 'ONE_ACTION':
      return { new state };
    case 'TWO_ACTION':
      return { new state };
    default:
      return state;
  };
};
```

Using useReducer()

- Calls to useState() are replaced with useReducer()
- The parameter is the initial state for the reducer to manage.
- It returns an array containing a reference to the current state and a pointer to the dispatch function.
- To mutate the state, invoke the dispatch function and provide an action object.
  - The reducer will be invoked with the current state and your action object.
  - The reducer returns a new state object, which triggers rendering of the component.

Code Bite

```
import React, { useReducer } from 'react';

const MyComponent = props => {
  const [state, dispatch] = useReducer(
    myReducer, { // initial state });

  const handleClick = () => {
    dispatch({ type: 'ONE_ACTION',
      payload: 'some value' });
  }
}
```

useReducer and HTTP State

- Some components will have multiple asynchronous HTTP calls throughout their lifetime.
- The flow of activity during these calls can be thought of as state changes.
  - You may want to display loading icons or messages during these actions.
  - Error messages should also be displayed when HTTP calls fail.
- It can be helpful to use a reducer to track the HTTP state for the component.
- You would typically need to track whether a call is in progress and whether there is an error message, at a minimum.

Context API Review

- Remember props are passed from parent to child.
- When a component manages some data that a descendant needs, the intermediate layers would need to pass along the props even though they have no need of that data.
- The Context API provides a way to externalize that data and its management.
- Context is a value (object, string, number) wrapped in a React-created object.
    - React creates a Provider and Consumer for the value.
- Its Provider element is wrapped around some JSX content.
- All descendent components can make use of the Context.
    - The context's Consumer element can wrap around JSX content that wants to use the Context data.
    - The context data can be available to JavaScript code by creating a static property.
    - But only class-based components can have static properties!

Working with useContext()

- The useContext() hook makes context data available to JavaScript code in functional components.
- Its parameter is a context object (the return value from React.createContext)
- It returns the current context value for that context object.
- When the nearest provider above the component updates, this hook will trigger a render with the latest context value.

Code Bite

```
import React, { useContext } from 'react';
import { MyContext } from '../MyContext';

const MyComponent = props => {
  const myContext = useContext(MyContext);

  // all context properties/methods are
  // available through myContext

};
```

The Rendering Process

- Using functional components can sometimes lead to unnecessary re-rendering cycles.
- The React rendering process is as follows:
    1. When updating the DOM, React first renders each component (in memory.)
    2. The render results are compared with the previous render results.
    3. If they are different, React will update the DOM.
- Class-based components can inherit from PureComponent, which will skip steps 1 and 2 when none of its props or state have changed.
- Functional components can achieve similar results using React.memo().

Memoized Components

- Components wrapped in React.memo() will render the same content as the unwrapped component.
- However, React will memoize the result and re-use the same result on subsequent rendering passes unless the new props have changed.
- Used throughout an application, this can lead to a significant performance gain.

Code Bite

```
import React from 'react';


const MyComp = props => {
  // component body here
};


export default React.memo(MyComp);
```

The Problem with Memoized Components

- Hooks allow functional components to work in ways previously reserved for class-based components.
- They can manage state, define handler functions, and pass those to child components for updating state.
- Class-based components do not introduce performance issues for that behavior.
  - The class instance persists between rendering cycles, so the handler functions are the same each cycle.
- Functional components are executed anew on each rendering cycle.
  - The handler functions are re-created each cycle.
  - The child component will see a prop change for the handler function.
  - This destroys any advantage gained from using React.memo() on the child component.

Performance Optimization with useCallback()

- The useCallback() hook will memoize a handler function in a parent component.
  - This will prevent it from getting re-created on subsequent rendering cycles.
- The first parameter is the function to be memoized, the second parameter is an array of dependencies.
  - The function will be re-created and re-memoized when any of the dependencies change.
- The return value is the memoized function to be passed to the child component as a prop.

Code Bite

```
import React, { useCallback } from 'react';

const MyComponent = props => {
  const myHandler = useCallback(() => {
    // body of handler
    // (has dependencies of a, b)
  }, [a, b]);

  // rest of component code
};
```

Performance Optimization with useMemo()

- useCallback() memoizes a function.
- useMemo() memoizes a value returned from a function.
  - It is useful for expensive calculations.
- The first parameter is the function whose value is to be memoized, the second parameter is an array of dependencies.
  - The function will be re-executed and its value re-memoized when any of the dependencies change.
- The return value is the memoized value that is usually either rendered or passed to a child component as a prop.

Code Bite

```
import React, { useMemo } from 'react';


const MyComponent = props => {
  const someValue = useMemo(() => {


    return someComputedValue;
  }, [a, b]);
};
```

Exercise 16: Reducers and Context

Skill Check

- Locate and complete the instructions for exercise 16 in your student files

Lesson 17: Custom Hooks

In this lesson you will learn about:

- Getting Started
- Using a Custom Hook
- Sharing Data Between Hooks and Components

Getting Started with Custom Hooks

- Traditionally, React applications had two ways of sharing stateful logic between components, each with their own drawbacks:
  - Render props – can often require intermediate components to pass along props they don't care about.
  - Higher-order components – complicates the component tree.
- Custom hooks give you another option to share stateful logic between components.
- Custom hooks are a convention that follows from the design of hooks.
  - They are not a React feature on their own.
- Components using the same hook do not share any state.
  - Each call to a hook gets isolated state.
  - Calls to useState() and useEffect() in the hook behave as if they were directly called from the component using the custom hook.
- Custom hooks are simply functions used as hooks.
  - The really SHOULD be named starting with "use" just like built-in hooks.

A Simple Custom Hook

- Custom hooks are simply functions that are used as hooks.
- They often invoke useState(), useEffect() or useReducer().
- They typically encapsulate stateful logic that would normally be built into a component.
  - This allows for re-use of that logic.
- They SHOULD be named starting with "use".

Code Bite

```
import { useReducer } from 'react';

const useToggle = (initial = false) => {
  return useReducer((state) => !state,
    initial);
};

export default useToggle;
```

Using a Custom Hook

- The custom hooks are imported and used just like any other function.

Code Bite

```
import useToggle from '../hooks/toggle';

const MyComponent = props => {
  const [display, toggleDisplay] =
    useToggle();

  return (
    <button onClick={toggleDisplay}
      >Show/Hide</button>
    {display && <div>Hello</div>}
  );
};
```

Uses for Custom Hooks

- Custom hooks are a great technique to implement cross-cutting concerns.
  - Especially when there is data and behavior that need to be recreated across multiple components.
- Hooks are frequently used for:
  - Providing access to authentication state (who is logged in) and behaviors (login, logout, etc.)
  - Adding event listeners manually (and automatically removing on unmounting the component.)
  - Combining multiple hooks (such as the built-in router hooks) into one, for convenience.
  - Requiring authentication (or minimum security permissions) and redirecting if not met.
  - Encapsulating asynchronous method calls and handling state updates during execution.
  - Encapsulating status checks (is a resource available, or is a "friend" user online.)
  - Create convenience object for managing data in an array

A Router Custom Hook Example

```
const useRouter = () => {
  const params = useParams();
  const location = useLocation();
  const history = useHistory();
  const match = useRouteMatch();

  return useMemo(() => {
    return {
      push: history.push, replace: history.replace, pathname: location.pathname,
      params, location, history, match
    }
  }, [params, location, history, match]);
};
```

Sharing Data Between Hooks and Components

- Custom hooks use their input parameters to instantiate and configure themselves.
- Their return value can contain function pointers.
  - The parameters of these functions would be used to pass data to the hook from the component throughout the lifecycle of the component.
- The return value from the custom hook can also contain data generated or acquired by the hook.
  - e.g., the hook may make an AJAX request to a web service.
- These data values are used to pass data from the hook to the component.
  - When that data is updated by the hook, it will trigger a re-render of the component using the hook.
  - useEffect() is the primary way the component can respond to the mutation of hook data.

A localStorage Custom Hook Example

```
import { useState, useEffect } from 'react';
const useLocalStorage = (key, def) => {
  const [state, setState] = useState(() => {
    let value;
    try {
      value = JSON.parse(window.localStorage.getItem(key) || String(def));
    } catch (e) { value = def; }
    return value;
  });
  useEffect(() => window.localStorage.setItem(key, state), [state]);
  return [state, setState];
};
export default useLocalStorage;
```

Exercise 17: Custom Hooks

Skill Check

- Locate and complete the instructions for exercise 17 in your student files

Congratulations!

You are ready to build React web applications