NextStep IT Training
powered by Smallrock Internet

React Programming

NS60601

Table of Contents

Copyright

Copyright 2018 by NextStep IT Training (Publisher). All rights reserved. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN:

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

Information has been obtained by Publisher from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Publisher, or others, Publisher does not guarantee to the accuracy, adequacy, or completeness of any information included in this work and is not responsible for any errors or omissions or the results obtained from the use of such information.

Node.js is a registered trademark of Joyent, Inc. and/or its affiliates. JavaScript is a registered trademark of Oracle America, Inc and/or its affiliates. TypeScript is a registered trademark of Microsoft Corporation and/or its affiliates. Webpack is a registered trademark of JSFoundation, Inc. and/or its affiliates. All other trademarks are the property of their respective owners, and NextStep IT Training makes no claim of ownership by the mention of products that contain these marks.

TERMS OF USE

This is a copyrighted work and NextStep IT Training and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without NextStep IT Training's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." NEXTSTEP IT TRAINING AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. NextStep IT Training and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither NextStep IT Training nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. NextStep IT Training has no responsibility for the content of any information accessed through the work. Under no circumstances shall NextStep IT Training and/ or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

NextStep IT Training
powered by Smallrock Internet

999 Ponce de Leon Boulevard, Suite 830

Coral Gables, FL 33134

786-347-3155

http://nsitt.com

Chapter 1 - React

Project Structure
Application Structure

Objectives

- Build a new React project using create-react-app
- Explore the structure built by create-react-app
- Investigate the React application flow

Overview

create-react-app has greatly simplified the work to build a React application. There really is no longer a need to create or modify the WebPack configuration, which includes things like the compilation into JavaScript 5, bundling css and image files, etc. *create-react-app* sets up all of the basic dependenices that the application will need.

Project Structure

Project Structure
Application Structure

React and JSX

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1 className="App-title">Welcome to React</h1>
        </header>
         <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App
```

- React is all about the presentation
- JSX notation allows tags defining the presentation to be embedded in the JavaScript code

Pure React Code

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return React.createElement('div', { className: 'App' }, [
      React.createElement('header', { className: 'App-header' }, [
        React.createElement('img', { src: logo, className: 'App-logo', alt: 'logo' }, null),
        React.createElement('h1', { className: 'App-title' }, 'Welcome to React')
      ]),
      React.createElement('p', { className: 'App-Intro' }, [
          'To get started, edit ',
          React.createElement('code', null, 'src/App.js'),
          ' and save to reload.'
      ])
    ])
  }
}
```

```
export default App
```

- JSX is just a convenience - JSX is compiled into calls to React.createElement
- A React component builds a DOM tree of React elements; these will be rendered into the browser DOM

Programmatically Decide What To Render

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

class App extends Component {
  render() {

    let result;

    if ((new Date()).getHours() > 17) {

      result = 'Sorry, we're closed!';

    } else {

      result = 'Welcome, please place your order.';
    }

    return (
      <div className="App">
        <p className="App-intro">
          { result }
        </p>
      </div>
    );
  }
}

export default App
```

- The presentation is dynamic, what created for depends on the current application state
- Logic decides what will be rendered; *{ result }* is an expression replaced with the value of *result*
- If the state changes, the user interface will be re-rendered and the user will see the change

Node.js

- React depends on Node.js to for project dependencies and development tools
- Node.js is stand-alone environment for executing JavaScript programs

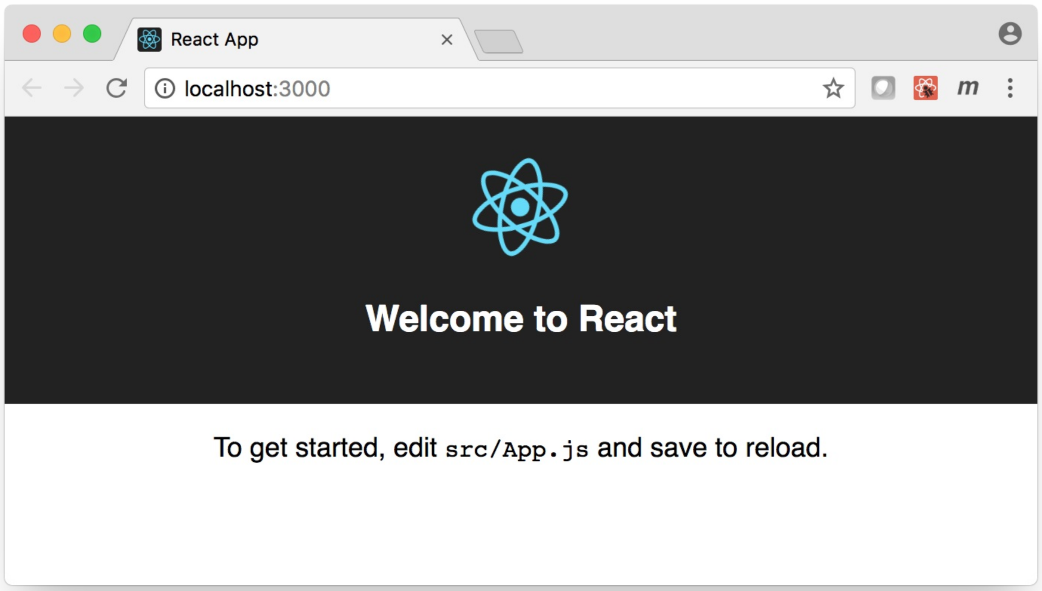
npm and *yarn*

- *npm* (Node Package Manager) is used to manage Node packages for a project
- Configuration is maintained in the file *package.json*
- *yarn* is an alternative package manager that uses the same npm registry

Local vs Global npm Packages

- Local packages are modules required by the application
- Installed into the folder *node_modules*
- Global packages are cross-project tools and commands

create-react-app



To get started, edit `src/App.js` and save to reload.

```
$ npm install -g create-react-app
$ create-react-app caribbean-coffee-shop
$ cd caribbean-coffee-shop; npm start
```

- create-react-app will generate a basic React project
- The tool is installed global using *npm*

package.json

```
{
  "name": "caribbean-coffee-company",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "react": "^16.4.0",
    "react-dom": "^16.4.0",
    "react-scripts": "1.1.4"
  },
  ...
```

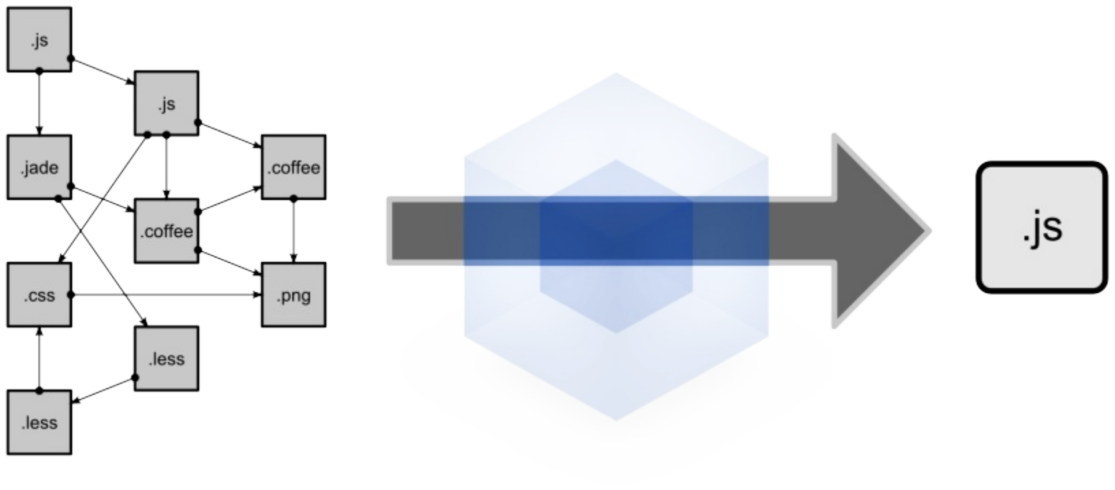
- Dependencies are packages bundled with the application
- Dependencies have dependencies; *react* depends on *object-assign* and other packages

Project Scripts

```
"scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject"
}
```

- *npm start* launches the application in a web server
- *npm run build* creates the bundle script from the sources
- *npm test test* runs the Jest unit tests
- *npm run eject* separates the Webpack configuration

Webpack



- Webpack controls compilation and bundles the project
- JavaScript, CSS, and other resources are bundled together

Babel

- Webpack uses *Babel* cross-compile JavaScript
- *Babel* Source maps are used for browser debugging
- The *babel-preset-react* plugin compiles JSX into JavaScript

Application Structure

create-react-app
Application Structure

public/index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <title>React App</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
  </body>
</html>
```

```
  <script type="text/javascript" src="/static/js/bundle.js"></script></body>
</html>
```

- Static files are published in the *public* folder
- The *public* folder contents is compiled by Webpack into the *build*
- *index.html* is a template, the bundle is merged in by webpack-dev-server to bootstrap the application

webpack-dev-server

- *webpack-dev-server* serves the index.html, the bundle, and other assets in the build
- Servers are necessary to publish assets and handle AJAX requests
- Default port is 3000

src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));
registerServiceWorker();
```

- Entry point to the application
- Bundle is created by following all the imports
- Uses ReactDOM to render the top React component

src/index.css

```
body {
  margin: 0;
  padding: 0;
  font-family: sans-serif;
}
```

- Bundled CSS file, not loaded as a separate asset

src/App.js

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
      ...
```

- Top of the React component tree
- *render* starts to produce the component tree
- "HTML" tags are JSX, React components which also *render*

React Elements

- React elements parallel HTML tags
- These are the smallest components of a React application
- They are the objects that will create HTML DOM elements

React Components

- Complex React components render other React components
- The React elements produce a DOM tree

Checkpoint

- React is used to programmatically create a DOM tree
- create-react-app builds a skeleton React application
- Babel and webpack are used to bundle the application

- webpack-dev-server injects the bundle into index.html
- React components *render* other components
- The component tree produces a DOM tree for the browser to display

Lab 1 - React

Objectives

- Introduce the basics of a React project.
- Explore using JSX to render basic React elements.
- Use conditional logic to decide what is rendered.

Estimated Completion Time

60 minutes lab and group discussion

Synopsis

The first lab is an instructor-lead, group-project that first introduces the roles of NodeJS, npm, and WebPack in a React project. The long-term goal of the project is an order-entry system for a cafe, the Caribbean Coffee Company. For the first project iteration the group will create a new project using the *create-react-project* command line tool, and explore the structure of the application, how React elements are rendered using JSX, and making simple changes to the JSX.

Requirements

- Start by using the create-react-app utility to generate a basic project in the Labs folder named "caribbean-coffee-company."

Project Steps

1. Work in the Labs. Run the command *create-react-app caribbean-coffee-company* to initialize a new project.
2. Explore the structure of the application, paying attention to:
   i. Explore how the application will launch with index.html, which will run index.js, which will render the App.js component.
   ii. Look at the CSS files corresponding with the different parts of the application.
   iii. Identify the test files that were created as part of the project.
   iv. Start the webpack-dev-server with *npm start*
   v. Verify the application is running.
   vi. In the browser look at how webpack-dev-server rewrites the index.html to deliver the JavaScript code to the browser.
   vii. Again follow the flow flow of the application from the index.html to the index.js which renders the top of the React DOM tree, to the App.js which renders the initial page content. Use the debugger as a way to watch what is happening.
   viii. Look at the React DOM using the react tools plugin in the Chrome or Firefox browser and compare it agains the browser DOM. Notice the difference: App is included in the React DOM.
   ix. Stress how React programmatically builds the React DOM tree, which is rendered into the browser DOM.

    x. Change what React is serving; change the message to "Hello World!" and see WebPack rebuild and deliver the application on the fly.

3. Discuss the differences between *npm start* to launch the development server and *npm build* to create the bundle files.
4. Look at *npm eject* and what the benefits and consequences are.

Results

This lab accomplished getting a React project set up, running, and understanding how it is put together. Now you are ready to start work on the cafe project.

Congratulations, you have completed this lab!

Chapter 2 - Elements and Components

React Elements
React Components

Objectives

- Identify the difference between React Elements and Components
- Build React Components
- Understand the React Document Object Model

Overview

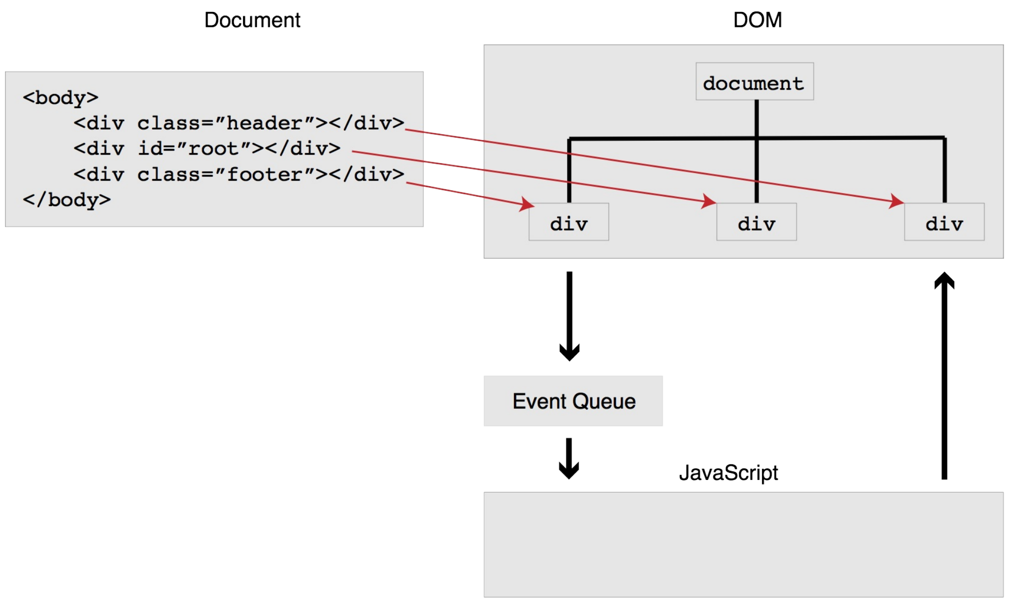
React elements are basic elements that map to the output environment, normally the HTML elements in the browser. Browser DOM elements are heavy-weight; there is significant overhead in creating and managing them. React quickly builds a tree of light-weight elements, and then renders only the new heavy-weight Browser DOM elements or property changes that are discovered as necessary from it.

React components are complex objects, rendering a hierarchy of other components and elements. Components should be designed as any other classes: modularization and reuse are the primary focus. Looking at the application interface the user sees, break it down into individual components that modularize the structure.

React Elements

React Elements
Components

Browser DOM and Event Loop



- The browser builds a document object model from the HTML document, and renders the screen from the DOM
- Then the browser listens for events occurring in the DOM (and elsewhere), and can execute JavaScript code in response
- JavaScript code may manipulate the DOM, changing what the user sees, which may trigger more events

React Elements

```
import React from 'react'

let re = React.createElement('p', null, 'Hello, World!')
```

- The basic form of creating an element is to use React.createElement
- The first parameter is the element to create, the second is properties, and the third is the content
- The result is a React element, a light-weight JavaScript object that can be added to a tree of elements

Rendering to the DOM

```
import React from 'react'
import ReactDOM from 'react-dom'

let re = React.createElement('h1', null, 'Hello, World!')
```

```
ReactDOM.render(re, document.getElementById('root'))
```

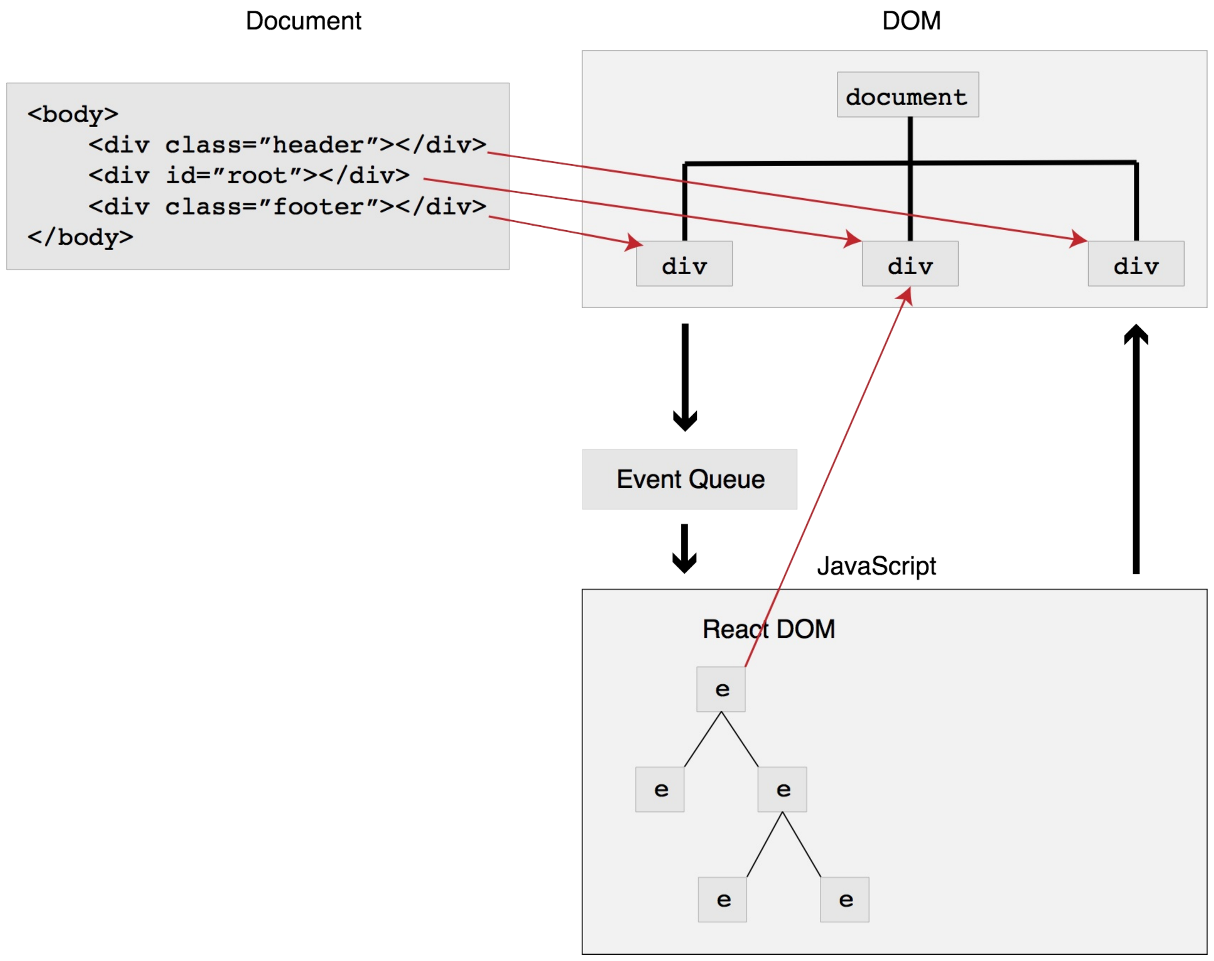
- A React element, or a tree of elements, is linked to the DOM at a particular point using ReactDOM.render
- React syncs the browser DOM to match the element, or tree, linked to it
- Syncing produces minimal changes to the browser DOM, speeding up the presentation

Create Element Trees

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

import logo from './logo.svg';
import './App.css';

class App extends Component {
    render() {
        return React.createElement('div', { className: 'App' }, [
            React.createElement('header', { className: 'App-header' }, [
            React.createElement('img', { src: logo, className: 'App-logo', alt: 'logo' }, null),
            React.createElement('h1', { className: 'App-title' }, 'Welcome to React')
            ]),
            React.createElement('p', { className: 'App-Intro' }, [
                'To get started, edit ',
                React.createElement('code', null, 'src/App.js'),
                ' and save to reload.'
            ])
        ])
    }
}

export default App
```

- A tree is created when additional elements are created and inserted as the content of another element
- Content may be an individual element, or an array of adjacent elements

React Document Object Model



- React elements are "rendered" to create the React DOM
- React syncs its DOM with the browser DOM, changing what the user sees
- Basic React elements are all that are synced, because they correspond directly to browser DOM elements

Factories

```
/* react-element-factory.js */

import React from 'react'

export const code = React.createFactory('code')
export const div = React.createFactory('div')
export const h1 = React.createFactory('h1')
export const header = React.createFactory('header')
export const img = React.createFactory('img')
export const p = React.createFactory('p')
```

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

import logo from './logo.svg';
import './App.css';
import { code, div, h1, header, img, p } from './react-element-factory'
```

```
class App extends Component {
    render() {
        return div({ className: 'App' }, [
            header({ className: 'App-header' }, [
                img({ src: logo, className: 'App-logo', alt: 'logo' }, null),
                h1({ className: 'App-title' }, 'Welcome to React')
            ]),
            p({ className: 'App-Intro' }, [
                'To get started, edit ',
                code(null, 'src/App.js'),
                ' and save to reload.'
            ])
        ])
    }
}

export default App
```

- Factories are simply functions created by *React.createFactory* that call *React.createElement*
- So, they are just a shortcut to simplify the code
- Not frequently used anymore

JSX is preferred to create elements

```
render() {
    return (
        <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <h1 className="App-title">Welcome to React</h1>
            </header>
            <p className="App-intro">
                To get started, edit <code>src/App.js</code> and save to reload.
            </p>
        /div>
    );
}
```

- JSX is so much cleaner for creating the React elements
- It still works exactly the same way, JSX is compiled to *React.createElement* as seen before

Attributes vs Props

```
<div className="App">
```

- The HTML attribute is *class*, but the DOM property is *className*
- React elements always use browser DOM property names, so *className*

React Components

React Elements
React Components

React Components

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App
```

```
import React from 'react'
import ReactDOM from 'react-dom'

import App from './App'

ReactDOM.render(<App />, document.getElementById('root'))
```

- Components are complex objects and render other components and elements
- A component can be used as any other element
- This is example is the App component generated with *create-react-app*

Classes

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App
```

- JavaScript 2015 classes are the simple way to create a new component type
- Override the render method and return the tree that the component renders
- The tree returned must have a single root element

Composition and Delegation

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

import Footer from './Footer'
import Header from './Header'
import Landing from './Landing'

class App extends Component {
  render() {
    return (
      <div className="App">
        <Header />
        <Landing />
        <Footer />
      </div>
    );
  }
}

export default App
```

- Divide the work into modules: for reuse or simply organization

What Happened Before JavaScript 2015?

```
var App = React.createClass({
  render: function () {
    return (
      <div className="App">
        <Header />
        <Landing />
        <Footer />
      </div>
    );
  }
});
```

- React provided the createClass method to simulate building a class
- You should not use this, it is just described in case you see it
- createClass creates a constructor function-object with the class name, and merges the given object with the

constructor prototype

Functional Components

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

import Footer from './Footer'
import Header from './Header'
import Landing from './Landing'

function App(props) {
    return (
      <div className="App">
        <Header />
        <Landing />
        <Footer />
      </div>
    );
  }
}

export default App
```

- Components could be a function as well, the function receives an object of props
- Limited: what the function returns is the rendered tree, no persistent state
- Simpler: easier when there does not have to be any state (more about that later)

Checkpoint

- Are JSX elements rendered into the browser DOM?
- What are three ways you expect to see elements created?
- Why build React components?
- How is a React component different from a React element?
- How is a React element different from a Browser DOM element?
- What are four ways to create a new React component?

Lab 2 - Components

Goals

- Build and use a hierarchy of custom components
- Separation of concerns and single responsibility
- Use logic to define what is output

Estimated Completion Time

60 minutes

Synopsis

The focus is to start organizing the cafe application for the Caribbean Coffee Company. The project needs a header, footer, and for the moment a landing view for the content. What the user sees for the content of the landing page is going to change based on the time of day.

Project Requirements

1. Organize the application; start by merging the the Resources/O2_Components/public and src folders into the project. If there are any conflicting files, choose the new ones.
2. Move the App and App.test modules to the App folder.
3. Remove the registration worker module and the logo image.
4. Create new React components for the application header and footer in the Boilerplate folder. The header will have a background and the logo for the company, and the footer will have a copyright notice.
5. Add a Landing component that welcomes the visitor in the Landing folder. If the hour is in the morning, it should say "Good morning, welcome to the Caribbean Coffee Company." If it is afternoon, then "Good afternoon, welcome..." If it is after five, then "Welcome... Sorry we are closed."
6. Fix the App component to render the header, footer, and the landing page in between them. The landing view should be wrapped in a div that uses the app-content class. Note that in the next lab the landing view will set aside for a short period of time.
7. Fix the index module to use the new CSS file, and remove the registration worker.

Detailed Steps

1. Merge the Resources/02_Components/public and src folders with the project. If there are any conflicting files, choose the new ones.

2. Peak at the contents of the Assets folder; take a look at the application.css file.

   - Note the difference from the App.css file: while App.css was tied to the App component, application.css will be shared by many components in the application and uses more-generic, lower-case names.

  - While App.css was strictly for the App component, application.css has common features shared across components. If a component needs a dedicated, un-shared feature, then a CSS file just for that component will added.
  - These styles will be used throughout the application, so take some time to familiarize yourself with them. CSS is not part of the course, so you are not responsible for developing these styles.
3. Move the App and App.test modules to the App folder.
4. Remove the App.css, index.css, logo.svg, and registerServiceWorker.js files, they will no longer be used.
5. Add Header and Footer component modules to the Boilerplate folder.
   - Both modules need to import "Assets/styles/application.css."
   - The Header should render an header element, 150 pixels high with the class "app-header."
   - Import "Assets/images/tccc-logo.png" and render it in the header with a class of "app-logo."
   - The Footer should render a footer element with class "app-footer," and a generic copyright notice: "Copyright © The Caribbean Coffee Company. All rights reserved".
6. In the Landing folder create a Landing component.
   - Import the css file.
   - The component should render a welcome message in a span with a class of "welcome-message."
   - The component should check the hour and adjust what is rendered (note the two levels of checks): if the hour is past 5 o'clock the message should say that the cafe is closed, otherwise welcome the user with a "Good Morning" or "Good Afternoon" message. Hint: just get the current date and time with the Date class and check the hour with the getHours() method, remember that hours past 12:00PM will be 13, 14, etc.
7. Update the App component to render the new boilerplate:
   - Change the App component to include the "application.css" file instead of App.css.
   - Make it render to render a div with the class "app."
   - In the div render a Header and a Footer component.
   - Between the Header and Footer add a div with the class "app-content."
   - Inside the content div, render a Landing component.
8. Update the index.js:
   - Fix the index.js file to include "application.css" instead of "index.css" (remember that the file location is different).
   - Remove the import or registerServiceWorker and the call to the it.

1. Save and check your work.
2. Open the browser's developer tools:
   - Compare the browser DOM contents with the React using the React Developer Tools plugin. The plugin shows the React DOM with the composite components, but they are not rendered into the browser DOM. Only the React elements that are leaf nodes appear in the browser DOM.

Recap

This lab accomplished setting up a better organization and explored a hierarchy of components:

1. The components are organized by *features* or *facets* of the application. Currently there are three: the app, the boilerplate for all views, and the landing view.

2. The shared assets have been grouped together.
3. All of the core CSS has been moved into one file for the application, instead of starting down a path of individual CSS files for each module (index and App). If module specific CSS is necessary then add a file for that module, but only if the styles need not be shared.
4. Conditional statements are used to control what is rendered.

Congratulations, you have completed this lab!

Chapter 3 - Props

Objectives

- Push data down the React DOM tree using props

Overview

In order to control the flow of information, React adopts a model where data for child components is delivered from the parent component via props.

Props

Component Props
Advanced and Custom Props
Component Rendering

Props (Properties)

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

- Props are the equivalent of attributes in HTML
- React HTML elements use browser DOM property names

Custom Component Props

```
class Menu extends Component {

    public render() {

        return <div className={ this.props.className }></div>
    }
}
```

- Props arrive as members of *this.props*
- Parents are expected to provide the props a child expects

Constructors and Props

```
<Menu className="App-menu" />
```

```
class Menu extends Component {

    constructor(props) {

        super(props)
    }
}
```

- *this.props* does not exist when the constructor is called
- It is created by the Component super-class constructor
- Call the super-class constructor and pass the props argument

Props & JavaScript

```
class App extends Component {

    render() {

        const colonists = [
            { name: "John Smith" },
            { name: "John Rolfe" },
            { name: "Thomas Wotton" }
        ]

        return <ColonistList people={ colonists } />
    }
}
```

```
class ColonistList extends Component {

    render() {

        const formattedPeople = this.props.people.map( name => <li>{ formatName(name) }</li> )

        return (
            <ul>
                { formattedPeople }
            </ul>
        )
    }

    formatName(name) {

        return name.replace(/(\w*) (\w*)/, "$2, $1")
    }
}
```

- Props may be strings, or any other JavaScript element
- JavaScript elements are passed enclosed in JSX expressions

Props are immutable

```
render() {

    this.props.people = [] // will not execute
    this.props.people.push( { name: " " } ) // very bad: do not modify the parent's data
}
```

- Props are considered to be immutable
- The value of a prop cannot be changed, JavaScript blocks it
- But if a prop is a reference then the referenced object *can* be changed, so avoid doing that!

Container Components

```
class App extends Component {

    render() {

        const colonists = [
            { name: "John Smith" },
            { name: "John Rolfe" },
            { name: "Thomas Wotton" }
        ]

        return <ColonistList people={ colonists } />
    }
}
```

```
class ColonistList extends Component {

    render() {

        const formattedPeople = this.props.people.map( name => <li>{ formatName(name) }</li> )

        return (
            <ul>
                { formattedPeople }
            </ul>
        )
    }

    formatName(name) {

        return name.replace(/(\w*) (\w*)/, "$2, $1")
    }
}
```

- Two types of components: presentation and container
- Presentational components render output; ColonistList is a presentational component
- Container components manage data and presentational components; App is a container component

Declaring Props

```
class ColonistList extends Component {

    render() {

        const formattedPeople = this.props.people.map( name => <li>{ formatName(name) }</li> )

        return (
            <ul>
                { formattedPeople }
            </ul>
        )
    }

    formatName(name) {

        return name.replace(/(\w*) (\w*)/, "$2, $1")
    }
}

ColonistList.propTypes = {

    people: PropTypes.array
}
```

- Expected props may be declared in a component
- The declaration is an object where each property is a prop name
- Each property specifies what type the prop should be

Prop Types

name	description
PropTypes.bool	A boolean value
PropTypes.number	A numeric value
PropTypes.object	An object reference
PropTypes.string	A string object reference
PropTypes.array	An array object reference
PropTypes.func	A function object reference
PropTypes.node	Anything that may be rendered (string, element, an array or fragment containing these types)
PropTypes.element	A react element reference

Required

```
ColonistList.propTypes = {

    people: PropTypes.array.isRequired
    date: PropTypes.any.isRequired
}
```

- Two possible warning messages: wrong type or not present
- The prop type *any* exists to define a prop that is required, of any type

- The prop must reference an object matching the specified shape

Declaring Props in JavaScript 2015

```
class ColonistList extends Component {

    render() {

        const formattedPeople = this.props.people.map( name => <li>{ formatName(name) }</li> )
        return (
            <ul>
                { formattedPeople }
            </ul>
        )
    }

    formatName(name) {

        return name.replace(/(\w*) (\w*)/, "$2, $1")
    }

    static get propTypes() {

        return {

            people: PropTypes.array.required
        }
    }
}
```

- Keeping prop declarations as a class member is desirable
- So, build a static *get accessor* to return the props

Prop Default Values

```
class ColonistList extends Component {

    render() {

        const formattedPeople = this.props.people.map( name => <li>{ formatName(name) }</li> )

        return (
            <ul>
                { formattedPeople }
            </ul>
        )
    }

    formatName(name) {

        return name.replace(/(\w*) (\w*)/, "$2, $1")
    }

    static get propTypes() {

        return {

            people: PropTypes.array
        }
    }

    static get defaultProps() {

        return {

            people: []
        }
    }
}
```

- Props may have default values
- The defaultProps property may be added to the constructor function-object, or returned from a static get accessor
- *required* property type setting is mutually exclusive with a default value

Advanced and Custom Props

Advanced Prop Types

name	description
PropTypes.oneOf(["John Smith", "John Rolfe"])	A prop matching one of the listed values
PropTypes.instanceOf(class)	Use JavaScript instanceof to match a class type
PropTypes.arrayOf(PropTypes.bool)	An array of a particular type, in this case bool
PropTypes.oneOfType([type, type])	An array containing the specified types
PropTypes.objectOf(type)	An object who's properties are of the specified type

Objects Matching a Shape

```
PropTypes.shape( {
    name: PropTypes.string,
    age: PropTypes.number
})
```

Custom Validation

```
    static get propTypes() {

        return {

            people: function (props, propName, componentName) {

                if (typeof props[propName] !== "string") {

                    return new Error(`Invalid prop \`${propName}\` supplied ` +
                        `to \'${componentName}\`. Validation failed.`)
                }
            }
        }
    }
```

- A function may be provided as a callback to validate the prop
- The function gets all of the props, the name of the prop to validate, and the component name
- The function returns an Error object if validation fails

Arrays and Custom Validation

```
static get propTypes() {

    return {

        people: PropTypes.arrayOf( (props, propName, componentName) => {

            if (typeof props[propName] !=== "string") {

                return new Error(`Invalid prop \`${propName}\` supplied ` +
                    `to \'${componentName}\`. Validation failed.`)
            }
        })
    }
}
```

- Every element of the array must pass validation

Component Rendering

Component Rendering

```
class ColonistList extends Component {

    render() {

        const formattedPeople = this.props.people.map( name => <li>{ formatName(name) }</li> )

        return (
            <ul>
                { formattedPeople }
            </ul>
        )
    }
```

- React calls the *render* method of a component
- When *render* creates other components (JSX or React.createComponent)
- React calls the render method on the new components

React DOM

- React components are first rendered into a React DOM
- Only after the React DOM is created is the browser DOM updated

Adjacent Components

- React can easily follow the tree and keep track of parent and child components
- Adjacent children of the same type are a problem to identify

Keys

- React depends on a key prop in adjacent components to identify and track them
- The key value must be unique for each component in a group
- Often a primary key from the data source is the best choice

Checkpoint

- Why do props need to be immutable?
- What does "rendering" a component mean?
- Why should adjacent components of the same type have keys?

Lab 3 - Props

Goals

- Continue to build out the hierarchy of application components
- Explore container component and presentation components
- Let data flow down to child components

Estimated Completion Time

60 minutes

Synopsis

Continue to build the hierarchy of components: explore container components and pass data down to child components.

Project Requirements

1. Merge the contents of *Resources/03_Props/src* with the project.
2. Build and use a Menu component in place of the Landing to show the beverages and pastries for the cafe.
3. The Menu component will be a container that holds the data and passes it to the child components to render.
4. Menu will render two lists: a Beverage list and Pastry list using a ProductList component. ProductList renders ProductItem components. The lists render a enclosing div of className="list", a title div inside it of className="list-title", and following the title div a table of className="list."
5. The ProductList will render a div that displays a title, which should be passed in as a prop (Beverages, Pastries, etc.) Following the title will be a table where each row will be an item. The first row is column titles: (blank) and price. The CSS class names for the columns (th and td) are "list-name" and "list-price."
6. Each ProductItem renders a new table row in the enclosing table.

Steps

1. Merge the contents of *Resources/03_Props/src* with the src folder in the new project.

2. Review the CSS classes for rendering the lists and items.

3. Review the Product entity class that provided in Data-Access.

4. Add a ProductItem component to the Menu facet (folder):

    i. A ProductItem takes one prop: a "product" which is a Product instance.
    ii. Make sure the ProductItem defines the property and data type it should have.
    iii. It renders a table row with two table cells, of class "list-name" and "list-price."
    iv. The content of the cells are the Product name and price properties.

1. Add the ProductList component in the Menu facet:

- The component should receive two props: "title" and "products", the second being a list of Product items.
- Set the prop requirements for type.
- In the render method use the array map function to produce an array of ProductItems components from the array of products.
- Render a div with class "list."
- Inside that div render another div with CSS class "list-title" and the value of the title prop.
- Directly following the title div (and inside the "list" div), render a table of CSS class "list."
- Add a row and two header columns: (empty) and "price." The CSS classes for the header elements are "list-name" and "list-price."
- Render the list of ProductItems.

1. A template for the Menu component has been provided.

   - Examine the two data members initialized: *beverages* and *pastries*.
   - Render an h1 title "Menu," followed by two instances of ProductList, one for beverages and the other for pastries.
   - Pass the *titles*, *beverages*, and *pastries* as props to the lists.

1. In the App component replace the Landing component with the Menu component (temporarily).

2. Save and check your work. You should see a view with the beverage and pastry lists.

Results

This lab accomplished building a container component (Menu) and pushing data down through props to child component instances (the two ProductList instances):

Congratulations, you have completed this lab!

Chapter 4 - State

Objectives

- Leverage component events
- Manage a components state
- Learn how state forces rendering

Overview

The advantage of managed state changes in React is that the component with the state is automatically re-rendered. When information is maintained in state or pushed through props, it helps eliminate the need to coordinate among different components that share data.

Components and Events

Components and Events
Component State

Pure vs Stateful Components

- Pure components have no state, and only use immutable data provided to them
- Stateful components manage data, and have a state

Input Components are Pure

```
export default class CheckName extends Component {
    constructor(props) {
        super(props)
        this.name = ''
        this.checked = false
    }
    render() {
        return  (
            <form>
                <label>name:
                    <input type="text" defaultValue={ this.name } />&nbsp;
                    <input type="checkbox" defaultChecked={ this.checked } />
                </label><br />
            </form>
        )
    }
}
```

- React merely passes input components through
- React knows nothing about how they work, and they are pure components
- Note: the label here wraps the fields, to eliminate *for* and superfluous id props

Synthetic Events

Touch and Mouse Events: onClick, onContextMenu, onDoubleClick, onDrag, onDragEnd, onDragEnter, onDragExit, onDragLeave, onDragOver, onDragStart, onDrop, onMouseDOwn, onMouseEnter, onMouseLeave, onMouseOut, onMouseOver, onMouseUp, onTouchCancel, onTouchEnd, onTouchMove, onTouchStart

Keyboard Events: onKeyDown, onKeyPress, onKeyUp

Focus and Form Events: onBlur, onChange, onFocus, onInput, onSubmit

Other Events: onCopy, onCut, onPaste, onScroll, onWheel

- React "normalizes" the browser events for consistent behavior across browsers
- The *synthetic events* have consistent names, and consistent event objects
- Note the camel-case names

Parent Components and Values

```
export default class CheckName extends Component {
    constructor(props) {
        super(props)
        this.name = ''
        this.checked = false
    }
    render() {
        return  (
            <form>
                <label>name:
                    <input type="text"
                        defaultValue={ this.name }
                        onChange={ (event) => this.name = event.target.value } />&nbsp;
                    <input type="checkbox"
                        defaultChecked={ this.checked }
                        onChange={ (event) => this.checked = event.target.checked } />
                </label><br />
            </form>
        )
    }
}
```

- It is the parent component that needs the input field value
- JSX links a function directly to the event
- The value is accessible through the event object

forceUpdate

```
    <input type="text"
        defaultValue={ this.name }
        onChange={ (event) => { this.name = event.target.value;
            this.checked = this.name ? this.checked : false;
            this.forceUpdate() } }/>&nbsp;
    <input type="checkbox" defaultChecked={ this.checked }
        disabled={ this.name }}
        onChange={ (event) => this.checked = event.target.checked } />
```

- What if changing a value affects other fields?
- this.forceUpdate() forces the component to be re-rendered
- Note: *forceUpdate* skips the *shouldComponentUpdate* event

Uncontrolled Components

```
    <input type="text"
        defaultValue={ this.name }
        onChange={ (event) => { this.name = event.target.value;
            this.checked = this.name ? this.checked : false;
            this.forceUpdate() } }/>&nbsp;
    <input type="checkbox" defaultChecked={ this.checked }
        disabled={ this.name }}
        onChange={ (event) => this.checked = event.target.checked } />
```

- React controls *value* and *checked*, and synchronizes them
- *defaultValue* and *defaultChecked* set the initial state and then leave it alone

Controlled Components

```
    <input type="text"
        value={ this.name }
        onChange={ (event) => { this.name = event.target.value;
            this.checked = this.name ? this.checked : false;
            this.forceUpdate() } }/>&nbsp;
    <input type="checkbox" checked={ this.checked }
        disabled={ this.name }}
        onChange={ (event) => this.checked = event.target.checked } />
```

- This example will not work, because the component is *controlled*
- React will always set the value back to the *initial* value of this.checked
- It will work if *state* is used

Component State

Components and Events
Component State

State and setState

```
export default class CheckName extends Component {
    constructor(props) {
        super(props)
        this.state = { name: '', checked: false }
    }
    render() {
        return  (
            <form>
                <label>name:
                    <input type="text"
                        value={ this.state.name }
                        onChange={ (event) => setState( { name: event.target.value,
                            checked: event.target.value ? this.state.checked : false } ) } />&nbsp;
                    <input type="checkbox" checked={ this.state.checked }
                        onChange={ (event) => setState( { checked: event.target.checked } ) } />
                </label><br />
            </form>
        )
    }
}
```

- State is a formal mechanism for managed local data
- *setState* merges an object with the current state
- *setState* triggers a re-rendering of the component

Initializing State

```
constructor(props) {
    super(props)
    this.state = { name: '', checked: false }
}
```

- State may be initialized in the constructor by assigning an object to *state*
- It does not have to be initialized
- *setState* CANNOT BE CALLED IN THE CONSTRUCTOR

State is Immutable

- Treat *state* as immutable
- *setState* changes are merged together to make a new state
- The *state* object is always replaced

setState Sets Fields

- *setState* sets the properties of the state object, *this.state*
- *setState* cannot set any deeper properties, except by assigning a new object to a property of *this.state*
- *setState* is simple, it changes the state and renders even if there are no changes

Callbacks

```
export default class CheckName extends Component {
    constructor(props) {
        super(props)
        this.state = { name: '', checked: false }
        this.nameChanged = this.nameChanged.bind(this)
        this.nameChecked = this.nameChecked.bind(this)
    }
    render() {
        return  (
            <form>
                <label>name:
                    <input type="text"
                        onChange={ this.nameChanged } />&nbsp;
                    <input type="checkbox" checked={ this.state.checked }
                        onChange={ this.nameChecked) } />
                </label><br />
            </form>
        )
    }
    nameChanged(event) {
        setState( { name: event.target.value,
                    checked: event.target.value ? true : false } )
    }
    nameChecked(event) {
        setState( { checked: event.target.checked } )
    }
}
```

- *setState* does not eliminate the need for callbacks
- Callbacks to not need to reference other components, or force rendering
- To use a method that uses *this* internally as a callback it must be bound to *this*

setState is Asynchronous

```
<input type="text"
    value={ this.state.name }
    onChange={ (event) => setState( { name: event.target.value,
        checked: event.target.value ? this.state.checked : false } ) } />&nbsp;
```

- This code fails - multiple calls to *setState* in a method are asynchronous
- This is intentional to keep the current *state* in sync with current *props*

New State

```
<input type="text"
    value={ this.state.name }
    onChange={ (event) => setState( (newState) => { return { name: event.target.value,
        checked: event.target.value ? newState.checked : false } } ) } />&nbsp;
```

- Pass a callback form of *setState* to get the new, unset state

Asynchronous Loading

```
constructor(props) {
    super(props)
    this.state = { name: '', checked: false }

    const p = fetch('http://randcomsystem/nameservice/randomname')

    p.then( (response) => response.json() )
        .then( (data) => this.setState( { name: data.name } ))
}
```

- *state* supports asynchronous data loading
- When the data is loaded a callback calls *setState*, and triggers a re-render

Asynchronous Loading with Async/Await

```
constructor(props) {
    super(props)
    this.state = { name: '', checked: false }

    fetchData()
}

async fetchData() {

    const jsonData = await fetch('http://randcomsystem/nameservice/randomname')
    const data = await jsonData.json()

    this.setState( { name: data.name } ))
}
```

- A constructor may not be asynchronous
- Move the await syntax into an async method the constructor can call

TextArea

```
<TextArea value={ this.state.description } />
```

- HTML TextArea wraps the content

- React normalizes it with a *value* prop

Checkpoint

- When is direct access to the browser DOM necessary?
- Why not just use an id prop which becomes an id attribute?
- What makes an uncontrolled component?
- What makes a controlled component?
- Instance data does not work with a controlled component, why?
- What makes setState special?

Lab 4 - State

Goals

- Explore container components
- Manage and render data from state
- Use state for data from a source that can change

Estimated Completion Time

60 minutes

Synopsis

State is critical because components automatically re-render themselves when their state changes. Data received from an AJAX request is delayed, so if a callback function changes the component state, then the state will re-render with the new data!

Project Requirements

1. Merge the contents *Resources/04_State/src* with the project, and review the *dataContext* object to see how to call the getBeverages() and getPastries() methods.
2. Change the Menu component to be a container that loads the beverage and pastry lists. Using the constructor to load the data is fine.
3. The data query returns a promise, so the data arrives as the resolution of the promise. Hint: if the component state is changed, then the Menu will be re-rendered.
4. If the promise is rejected, then an render should display a message the list (beverages or pastries). Hint: change the ProductList component to include an error prop (true or false).

Steps

1. Merge the contents of *Resources/04_State/src* with the src folder in the new project.

2. In another window, move to the Resources/WebService folder and use "npm install" followed by "npm start" to launch the web service. Check the web service by browsing to http://localhost:3001/data/beverages and getting the list of beverages.

3. Review the DataContext class to see how to communicate with the web service:

    - There are actually three contexts: BeverageContext, PastryContext, and DataContext which is the entry point for everything.
    - The DataContext has properties: beverageContext and pastryContext.
    - Look specifically at the getBeverages and getPastries methods of the two contexts.
4. Change the Menu class:

   - Import that singleton copy of the DataContext. The reason the module is labeled with a lower-case name is because the module returns a singleton, not a class definition:

     ```
     import dataContext from '../Data-Access/dataContext'
     ```

   - Use the dataContext instance to load the lists of beverages and pastries in the the constructor.
   - When the promises are resolved (you can handle them separately) add the data to the Menu component state as beverages and pastries, and push them through as props to the correct list component.
   - If a promise is rejected, put that into state as beverageError or pastryError, and push it through to the correct list component as the error prop.
5. Change the ProductList.
   - Add the error prop to the component, making sure the boolean data type is specified.
   - If the error prop is set, display the a generic error message as the list content: "The list is unable to be retrieved at this time."
   - If error is not set, then process the products as before.
6. Save and check your work, the beverages and pastries should be loaded from the web service (the list is more extensive).
7. Check the application with the web service disabled and make sure the errors are displayed.

Results

This is a simple use of state, but also the most typical use of state. An AJAX request is fired off, and the component renders without any data (empty). When the asynchronous request is complete, the state is updated and the components re-render with the actual data! If the AJAX request is rejected (incorrect or simply the service is not available), an error message is displayed. Note that the Menu is a container handling the data, and the two lists are simply handed the data as props.

Congratulations, you have completed this lab!

Chapter 5 - Composition

State, Render, and Inheritance
Composition over Inheritance

Objectives

- Understand that state and render are in opposition to inheritance
- Learn to reverse the structure; wrap instead of extending

Overview

State and render impede using Inheritance to extend React components. This chapter explores why that happens, and solves the problem by favoring composition over inheritance.

State and Inheritance

State, Render, and Inheritance
Composition over Inheritance

Defining State

```
constructor(props) {
    super(props)
    this.state = { name: '', checked: false }
}
```

- State is defined by assigning an object to *this.state*
- Technically a call to *setState* could initialize state, but that would be rare

Overriding State and Render

```
class AccordianList extends Component {

    constructor(props) {
        super(props)
        this.state = {
            expanded: false,
            listData: []
        }
    }

    render() {

        let content = null

        if (this.state.expanded) {

            content = <div class="accoridan-content">{ this.state.listData }</div>
        }

        return {
            <div class="accordian">
                <div class="accoridan-title" onClick={ () => setState({ expanded: !this.state.expanded } )
}>
                    <h1>{ this.props.title }</h1>
                </div>
                { content }
            </div>
        }
    }
}

class ColonistList extends AccordianList {

    constructor(props) {
        super(props)
        // Can only inherit and add to state, and AccordianList depends on the subclass getting it right

        this.state.listData = this.props.colonists.map( (colonist) => <li>{ colonist.name }</li> )
    }

    render() {

        // There is no easy way to override what is rendered; only wrap it. The super-class render relies
        // on the list being added to the state, which is really yucky.

        return <div class="colonists">{ super.render() }</div>
    }
}
```

- A subclass overriding state would destroy the super-class state
- The subclass has to follow rules set by the super-class, so the programmer must know them

Strategy Pattern Alternative

```
class AccordianList extends Component {

    constructor(props) {
        super(props)
        this.state = {
            expanded: false
        }
    }

    render() {

        let content = null

        if (this.state.expanded) {

            content = <div class="accoridan-content">{ this.getData() }</div>
        }

        return {
            <div class="accordian">
                <div class="accoridan-title" onClick={ () => setState({ expanded: !this.state.expanded } )
>
                    <h1>{ this.props.title }</h1>
                </div>
                { content }
            </div>
        }
    }
}

class ColonistList extends AccordianList {

    getData() {

        return this.props.colonists.map( (colonist) => <li>{ colonist.name }</li> )
    }

    render() {

        return <div class="colonists">{ super.render() }</div>
    }
}
```

- An alternative is to use a method instead of state, a.k.a Strategy Pattern from Eric Gamma's design patterns book.
- The method is expected in the super-class, and must be defined in the sub-class
- Of course, state is still shared...

Questionable OOP

- Sharing state clearly violates the principles of good object-oriented programming
- Using the *strategy pattern* alternative is reduces the risk, but coupling is high and state is still shared!
- And, extending *render* is complicated

Composition over Inheritance

State, Render, and Inheritance
Composition over Inheritance

Composition

```
        return {
            <div class="accordian">
                <div class="accoridan-title" onClick={ () => setState({ expanded: !this.state.expanded } )
}>
                    <h1>{ this.props.title }</h1>
                </div>
                { content }
            </div>
        }
    }
}
```

- Composition is one class containing another
- We see this all the time; the Accordion class wraps several div elements and a header

Composition over Inheritance

```
class AccordionList extends Component {

    constructor(props) {
        super(props)
        this.state = {
            expanded: false
        }
    }

    render() {

        let content = null

        if (this.state.expanded) {

            content = <div class="accoridan-content">{ this.props.children }</div>
        }

        return {
            <div class="accordian">
                <div class="accoridan-title" onClick={ () => setState({ expanded: !this.state.expanded } )
}>
                    <h1>{ this.props.title }</h1>
                </div>
                { content }
            </div>
        }
    }
}
```

```
class ColonistList extends Cmponent {

    render() {

        let colonists = this.props.colonists.map( (colonist) => <li>{ colonist.name }</li> )

        // There is no easy way to override what is rendered; only wrap it. The super-class render relies
        // on the list being added to the state, which is really yucky.

        return <Accordion>{ colonists }</div>
    }
}
```

- This is counter-intuitive: the ColonistList class is composed of the Accordion class
- ColonistList absorbs the functionality of Accordion by wrapping it
- ColonistList does not depend on the encapsulated functionality of the Accordion

Checkpoint

- Do super and subclasses manage separate state?
- Why not let super and subclasses share a state object?
- How does the render method get in the way of inheritance?

Lab 5 - Composition

Goals

- Favor composition over inheritance - always!

Estimated Completion Time

60 minutes

Synopsis

Component inheritance doesn't work for several reasons: state conflicts between the super-class and subclasses, generally the super-class render needs to wrap the sub-class render, not the other way around, and propTypes and defaultProps need to be static members of the class. The second reason could perhaps be solved by using the strategy pattern. But the first reason is uglier: the sub-class needs to make sure that it never steps on the super-class state. Object-oriented paradigms already recognize that composition should be favored over inheritance. Our problem of creating an accordion list for the menu products drives this point home.

Project Requirements

1. Change the menu lists so that they expand and collapse to show this list of items or just a title.
2. An *AccordionList* component should be reusable for any content, the content could be anything to hide or reveal, it does not even have to be a list (so it will work with the error message too!).
3. By default the content of an AccordionList should be shown as open.
4. In project though, the menu lists should initially be shown as closed. That means the initial open/closed state needs to be passed to the list as a prop *open*.
5. When the visible title area is clicked, the list expands or collapses.
6. CSS will be used to show an arrow in front of the title: right-facing when the list is closed and pointing down when the list is open. Look at the ::before aspect of the list-title and list-title-closed classes.

Steps

1. Create a Common folder and a shareable AccordionList.js module.

2. The AccordionList class will:

    - Have a state property *display* that indicated if the list is expanded (true)
    - Render an outermost div of class "list"
    - Render a div of class list-title with the title prop for the component.
    - If the list is closed, add the class list-title-closed to the title div.
    - Render the children prop after the title div (but still in the list div).
    - The component should expect props *open* and *title. title* should be required, check the definition.

   - *open* should have a default value of *false*.

3. In the ProductList component wrap the table with an AccordionList. This is how the AccordionList is leveraged: ProductList renders an AccordionList to re-use its drop-down functionality, and injects the contents as the child of the AccordionList.
4. Save and check your work.
5. Check the application with the web service disabled and make sure the errors still work.

Results

This lab uses composition and delegation to create reusable drop-down lists of items.

Congratulations, you have completed this lab!

Chapter 6 - React-Router

Objectives

- Set up routing for application features to initiate context-switches
- Leverage static and dynamic switch mechanisms
- Pick up the router context to make switches in child components

Overview

The function of a router is to centralize the control of context changes in the application. Different facets or features of the application are still required to decide when a context change is necessary, but instead of making the switch themselves they instruct the router to make the switch. What view the router switches to is not the concern of the view that initiated the switch, and can easily be replaced with another view in the future.

Routers, Facets, and Switches

Routers, Features, and Switches
React-Router Configuration

Context Switching

- Context switching takes place when one feature wants to display another
- React creates a problem, because the second feature is rendered by the first?
- If a landing page wants to show a menu, does it wrap them menu component in its own render? How many other components would have to wrap the menu? How does the menu go back to the landing page, by wrapping it?

Routing

- Routing is simple moving the functionality into a component that decides what to show
- Changing the view from one feature to another is a "context switch"
- A feature tells the router what it wants the user to see next, the router makes it happen!

Routers, Facets, and Switches

Routers, Features, and Switches
React-Router Configuration

The Router Component

```
import { BrowserRouter as Router } from 'react-router-dom'
import Main from './Main'

class App extends Component {

    render() {

        return (
            <Router>
                <Main />
            </Router>
        )
    }
}
```

- To switch views, the router component needs to be at the top of the component tree
- Or, at least the part of the tree it will change! Routers may be nested!

Types of Routers

Component	Description
BrowserRouter	A router that uses the address line with a '#' to specify a route
HashRouter	A router that uses the address line, but in the HTML5 form without '#'
MemoryRouter	A router that keeps the route invisible, in memory

- React-router provides three types of Router components

Selecting a Router

```
import { BrowserRouter as Router } from 'react-router-dom'
```

- Use the alias feature of modules to name whatever router "Router"

Defining Routes

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

~~~
class Main extends Component {
    render() {
        return (
            <div>
                <Switch>
                    <Route path="/" exact={ true } component={ Landing } />
                    <Route path="/menu" component={ Menu } /> } />
                    <Route path="/checkout" component={ Checkout } /> } />
                </Switch>
            </div>
        )
    }
}

* In this example in Main, to keep the responsibility separate from App
* The Router is very intelligent, but "/" is special and needs an "exact" match
* The component chosen is displayed in the place occupied by the Routes

### Changing Routes with Link

```javascript
<button>
    <Link to="/menu">Menu</Link>
</button>
~~~

- Components farther down in the tree decide on a new route
- The Link component renders as a tag that will set the route when clicked
- Hint: wrap a link with a button to make it appear as a button

Changing Routes Dynamically

```javascript
import { withRouter } from 'react-router'

class Navigation extends Component {

    render() {

        let NavWithRouter = withRouter((props) => (

            <div className="navigation">
                <div className={ `${ props.location.pathname === '/' ? 'navbutton-selected' : 'navbutton'
}` }
                    onClick={ () => props.history.push('/') }>Home</div>
                <div className={ `${ props.location.pathname === '/menu' ? 'navbutton-selected' : 'navbutt
on' }` }
                    onClick={ () => props.history.push('/menu') }>Menu</div>
                <div className={ `${ props.location.pathname === '/checkout' ? 'navbutton-selected' : 'nav
button' }` }
                    onClick={ () => props.history.push('/checkout') }>Checkout</div>
            </div>
        ))

        return <NavWithRouter />
    }
}
```

- The *history* object is required to push a new route address

- The best way to get it is with the *withRouter* function
- Here in a *Navigation* component *withRouter* calls a function and passes it the history as a prop

Using Dynamic Routes Efficiently

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

class Navigation extends Component {

    constructor(props) {
        super(props)

        this.NavWithRouter = withRouter((props) => (

            <div className="navigation">
                <div className={ `${ props.location.pathname === '/' ? 'navbutton-selected' : 'navbutton'
}` }
                    onClick={ () => props.history.push('/') }>Home</div>
                <div className={ `${ props.location.pathname === '/menu' ? 'navbutton-selected' : 'navbutt
on' }` }
                    onClick={ () => props.history.push('/menu') }>Menu</div>
                <div className={ `${ props.location.pathname === '/checkout' ? 'navbutton-selected' : 'nav
button' }` }
                    onClick={ () => props.history.push('/checkout') }>Checkout</div>
            </div>
        ))
    }

    render() {

        let NavWithRouter = this.NaveWithRouter

        return <NavWithRouter />
    }
}
```

- Call *withRouter* once, and use the results in every render!
- Note the minor problem: withRouter returns a *component*, that must be in a local variable to render in JSX

Make the Callbacks Methods?

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

class Navigation extends Component {

    constructor(props) {
        super(props)

        this.pushHome = this.pushHome.bind(this)
        this.pushMenu = this.pushMenu.bind(this)
        this.pushCheckout = this.pushCheckout.bind(this)

        this.NavWithRouter = withRouter((props) => {

            this.history = props.history
            return (

                <div className="navigation">
                    <div className={ `${ props.location.pathname === '/' ? 'navbutton-selected' : 'navbutt
on' }` }
                        onClick={ this.pushHome }>Home</div>
                    <div className={ `${ props.location.pathname === '/menu' ? 'navbutton-selected' : 'nav
button' }` }
                        onClick={ this.pushMenu }>Menu</div>
                    <div className={ `${ props.location.pathname === '/checkout' ? 'navbutton-selected' : '
navbutton' }` }
                        onClick={ this.pushCheckout }>Checkout</div>
                </div>
            ))
        }
    }

    pushHome() {

        this.history.push('/')
    }

    pushMenu() {

        this.history.push('/menu')
    }

    pushCheckout() {

        this.history.push('/checkout')
    }

    render() {

        let NavWithRouter = this.NaveWithRouter

        return <NavWithRouter />
    }
}
```

- A little more complicated; the history object has to be saved from within *withRouter* for the methods to use
- The methods must be bound to this before they are referenced in *withRouter*

Checkpoint

- Why not let one component decide what other component will be rendered?
- What is the difference between BrowserRouter, HashRouter, and MemoryRouter?
- Why is the Router placed at the top of the App?
- To what end are the route definitions separated from the Router component?
- What does the Link component do?
- How is a routed changed dynamically?

Lab 6 - React-Router

Goals

- Allow selection of a view using react-router
- Implement static links and dynamic route changes
- Replace the landing view and add the checkout view

Estimated Completion Time

60 minutes

Synopsis

Controlling the choice of a view through state or props can be messy. The React-Router simplifies this by putting the management of the view in one location, but allowing any child component to change the view by changing the router path for the application.

Project Requirements

1. Add an Checkout component in a Checkout "facet" of the application. The component will simply render an h1 header "Checkout."
2. Move the rendering of the application view into a Main component; App should now render Main wrapped in a Router.
3. Have the Main component use a Switch to determine which view to show: the Landing, the Menu, or the Checkout.
4. Render a new Navigation component below the header that creates menu bar with "Home," "Menu,C and "Checkout." The Navigation component renders a div with class "navigation," and each button is a div with either "navbutton" or "navbutton-selected" as the class. Determine the class by matching the current route to the button route. Each button will change the route when clicked.

Steps

1. Create a Checkout folder in the src.
2. Add a Checkout component in the new folder, simply render an h1 header "Checkout."
3. Copy the App component to a new Main component, in the App folder.
4. Change the App component so that it imports a new *BroswerRouter* from react-router, alias it to *Router*.
5. Change the rendering of the App component to render the Router component, and a single child, the Main component.
6. Modify the Main component to import Route and Switch from react-router.
7. Import the new Navigation component (it has not been created yet, just go with it).

8. Replace the rendering of the Menu component with a Switch.
9. The first Route in the switch should indicate a path of "/" as an exact route, and render the Landing component.
10. The second Route should indicate a path of "/menu" and render the Menu component.
11. The third route should indicate a path of "/checkout" and render the Checkout component.
12. Create the missing Navigation component in the App folder.
13. In Navigation, import *withRouter* from the react-router module.
14. In the constructor use withRouter to wrap a functional component (that is a fancy name for an arrow function that returns a component definition). Remember, the function gets *props*. Save the result of calling *withRouter* to *this.NavWithRouter.*
15. In the arrow function set *this.history* to *props.history*.
16. Return a hierarchy of a div containing three other divs. The container div will have a class of *navigation.*
17. In each of the three divs created a menu buttons (HTML <button>). Each button will have a class of *navbutton* or *navbutton-selected.* Use the *location* prop to compare the current pathname to the path for the specific button: "/," "/menu," and "/checkout." Set the class of the button accordingly.
18. When each button is clicked, it needs to call a method to set the history and change the route. Create three methods *pushHome*, *pushMenu*, and *pushCheckout*, and set up the calls.
19. In each method, call *this.history.push* and pass it the route string: /, /menu, or /checkout.
20. Before calling *withRouter* in the constructor, bind each of the methods *pushHome*, *pushMenu*, and *pushCheckout* to *this*.
21. In the *render* method copy *this.NavWithRouter* to a local variable *NavWIthRouter*.
22. From *render* return an instance of *NavWithRouter*.
23. Save and check your work. As you click the menu buttons the view should change.

Results

When the user first opens the application, the landing page is visible. They can now move to the menu page to make selections, and then to the checkout page. But, what happens when you move off of the menu and back to it? What could be done to remember the state of the menus, open or closed?

Congratulations, you have completed this lab!

Chapter 7 - Forms

React Form Handling
Form Field Validation

Objectives

- Explore the nuances of using forms and fields in React
- Avoid form submission to the server
- Validate form fields and display error messages

Overview

Handling forms and fields is straight-forward, React provides an element for the form and all of the HTML 5 form fields. The tricky part is that React does not provide a standard form of validation for fields. While there are several available from npm, this chapter explores a generic mechanism that follows the spirit of React.

React Form Handling

React Form Handling
Form Field Validation

Form

```
render() {
    return (
        <form>
        </form>
    )
}
```

- Most forms in React are used by the application
- Simply using a form tag is sufficient
- HTML5 insists that fields be in a form, most browsers do not care, but the form is important if you want to link submit buttons to a group of fields for the Enter/Return key to work properly

submit and onSubmit

```
render() {
    return (
        <form onSubmit={ (event) => {

            // Do something on submission

            // Prevent form submission to the server

            event.preventDefault()
        }}>
            <input type="submit" />
        </form>
    )
}
```

- A submit button will cause a form submission (so maybe do not include one!)
- Without an action attribute, the form goes back to its source
- Returning *false* will not prevent form submission, *preventDefault* must be used (because the code calling the handler would need to return false)

Uncontrolled Components

```
<input type="text"
    defaultValue={ this.name }
    onChange={ (event) => { this.name = event.target.value;
        this.checked = this.name ? this.checked : false;
        this.forceUpdate() } }/>
```

- React controls *value* and *checked*, and synchronizes them
- *defaultValue* and *defaultChecked* set the initial state and then leave it alone

Controlled Components

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

- A callback is still required to set state
- *setState* changes the state and forces a re-render

TextArea

```
<TextArea value={ this.state.text } />
```

- React modifies how TextArea works, normalizing it with a value prop

Form Field Validation

React Form Handling
Form Field Validation

Wrap Form Fields

- A common solution is to wrap form fields with a validator
- This follows the preference for composition, but requires wrapping each field
- *react-form-validator-core* is an module that follows this pattern

Parallel Validation

- Another way is to invoke logic to validate on every change
- The logic can be shared in another module
- Confusing: what if the validation is missed in a callback?

Adjacent Validation

```
<Validator className="validation"
    value={ this.state.password }
    isRequired={ true }
    constraint={ /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])[0-9a-zA-Z]{8,}$/ }
    notify={ () => this.valid = false }>
    Password does not meet constraints
</Validator>
```

- Another solution is a validation component that validates on each render
- *react-data-validator* is a module that follows this pattern
- The validator is a component checks values, not fields

Using the Validator

Prop	Description
value	The value to validate
isRequired	If true, the value must be defined and not null
constraint	A
notify	If define, a function to call when validation fails
children	The message to display if validation fails

Consolidated Error Messages

```
import Validator from 'react-data-validator'

...

render() {
    return (
        <div>
            <Validator value={ this.state.name }>The name is required</Validator>
            <Validator value={ this.state.password }>The password must be a
                minimum of eight characters with an uppercase letter,
                lowercase letter, and a digit</Validator>
        </div>
    )
}
```

- Multiple validators may validate the same value
- Group some validators and have others next to the fields

Rendering on Validation

```
render() {
    return (
        <div>
            <Validator value={ this.state.name }
                notify={ () => this.valid = false }>The name is required</Validator>
            <button disabled={ !this.state.valid }>Submit</button>
        </div>
    )
}

componentDidUpdate() {

    if (this.valid !== this.state.valid) {

        this.setState({ valid: this.valid })
    }
}
```

- The notify method may not call *setState*, because it is called from render
- *setState* may be called from *componentDidMount* or *componentDidUpdate*
- Call *setState* ONLY IF THE STATE CHANGES, or it will recurse infinitely

Checkpoint

- What makes forms in React different from forms in HTML?
- Do form fields need to be inside of a form element?
- How do you stop a form from being submitted in React?
- Is form field validation and prop validation the same thing?
- How do you validate the data in form fields?
- Can we disable other fields if validation fails?

Lab 7 - Forms

Goals

- Add shopping cart functionality to purchase products.
- Integrate the cart with the Menu and the Checkout components.

Estimated Completion Time

90 minutes

Synopsis

The user needs to be able to select items from the beverages and pastries and add them to the cart. They should be able to check out from the shopping cart on the Checkout page, which resets the application back to an empty cart.

Project Requirements

1. Add a third column to the products with a button, which will be used to add an item to the cart.
2. The method to add to the cart should mask the window and display a div show the item being purchased and asking for special instructions. The user will be allowed to cancel the add item on this window. A submit button will add the item to the cart.
3. Display the cart with a total at the bottom of the Menu page. Each entry in the cart should have a remove button to clear it.
4. Add a checkout button at the bottom of the cart, which switches to the Checkout view.
5. The Checkout component should show the cart, also with a remove button to clear individual items. Hint: use the same component to display the cart in both views!
6. Below the cart should be two fields, one for the customer name and another for the credit card.
7. A cancel button that goes back to the menu should always be visible. The submit button to checkout should only be visible (or enabled) if the cart is not empty, and the two fields are not empty.
8. Click the submit button will clear the cart and take the user back to the menu (yeah, it is kind of simple but what the heck)

Steps

1. Merge the Resources/08_Forms/src module with the project. This will add a Cart folder.
2. Explore the cart.js and CartEntry.js files. The cart is a singleton that the Menu and Checkout pages share when it is imported.
3. Add a new column to the BeverageItem and PastryItem components. The new column will be of class *list-add-button.*
4. The button needs to invoke a method that adds an entry to the cart. The method should be defined in Menu, and passed as a prop to the ProductList, and from there as a prop into the ProductItem.

5. The add to cart method in the Menu component will take an instance of a product item as its argument. Hold the item in the state. Set a showSpecialInstructions property in the state to true.
6. Add two divs at the top of the menu display. The first one is empty, but has a class of *special-instructions-visible* when state.showSpecialInstructions is true, and *special-instructions-hidden* when state.showSpecialInstructions is false.
7. The second div will have a class of *special-instructions* when state.showSpecialInstructions is true, and *special-instructions-hidden* when state.showSpecialInstructions is false. This div should show the name of the item being purchased (state.item), an input field with the placeholder *Special Instructions,* and a cancel and submit button. The submit button should add the item with the special instructions to the cart, the cancel button should abort the transaction. Both buttons should set state.showSpecialInstructions to false and value of specialInstructions in state to an empty string. The input field should have a class of *special-instructions*, and the buttons should be wrapped in a div of class *special-instructions-buttons* to right-align them.
8. The method to add an entry in Menu should add the entry to the cart. In order to get the cart to refresh on the screen, Menu will need to trick React by setting its state to cause a re-render.
9. Create a new component CartList in the Checkout folder. This should display the list of items selected with a total below it. Check the CSS file, there are classes for this shopping cart list.
10. Each item in the cart will have a remove button that calls a method in the CartList component that finds and removes the current item from the cart. In order to get the cart to refresh, this method also needs to trick React by setting state after the cart is modified.
11. The total should not be displayed if the cart is empty.
12. Modify the checkout page so that it displays the CartList. Underneath it add two fields, one for the customer name and another for the card number.
13. The name and number fields should be each in their own div. CSS classes are available to place the titles and the fields next to each other: *cc-form-row*, *cc-form-label*, *cc-form-field*. After the two fields in the row div, place an empty div with the class *clear-all* to make sure vertical space is consumed for the floating label and field.
14. Add another row to the form that holds a cancel and submit button. The buttons should go in the second column to line up under the fields.
15. The submit button should not be available if there is nothing in the cart, or the two fields are not valid. Build a method to validate the fields, and then in the render method decide if the user can submit the order.
16. When the user submits the order, it should call a method that clears the cart and returns to the Menu view. Because the return is programatic, you will have to wrap the submit button in a *withRouter* function call and pass the props with the router history through to the callback method. Another choice would be to wrap the Checkout page in the Main component so it gets the history in its props, but that will increase the coupling between the two components.
17. Since the requirements only state that the form fields are required, that is all you need to consider. CSS should be used to change the state of the name and card number fields; when they are empty they should have the class *cc-form-field-required*, and if they contain something then they should have the class *cc-form-field-requirement-met*.
18. Save and check your work. As you click the menu buttons the view should change.

Results

When the user adds a product to the cart, it updates the order on the Menu screen. THe user can remove items from the order in that list. Clicking the checkout button or the menu button for Checkout moves to a view where the cart is displayed, and form fields are given for the customer name and card number. The user can remove items from the cart on this page as well. When the user is ready, they can cancel continuing the transaction and go back to the Menu page, or they can complete the transaction in which case the cart is cleared and they are back at the menu page.

The cart is a singleton, a global instance shared by both the Menu and CartList components. This is a bit of a problem, because both components are dependent on a shared structure. Also, React was tricked to show that the cart was updated. There must be a better way...

Congratulations, you have completed this lab!

Chapter 8 - React DOM

Objectives

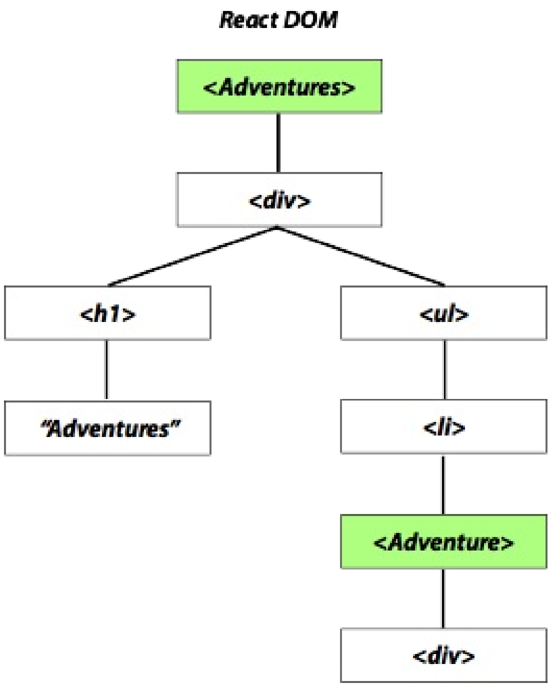
- Understand the React component life cycle
- Capture and respond to life cycle events
- Map React components to browser DOM elements

Overview

This chapter explores the advanced life cycle of a React component; how it is created, when it is updated, and when it goes away. And, occasionally it is necessary to obtain a reference to the browser element backing a React component.

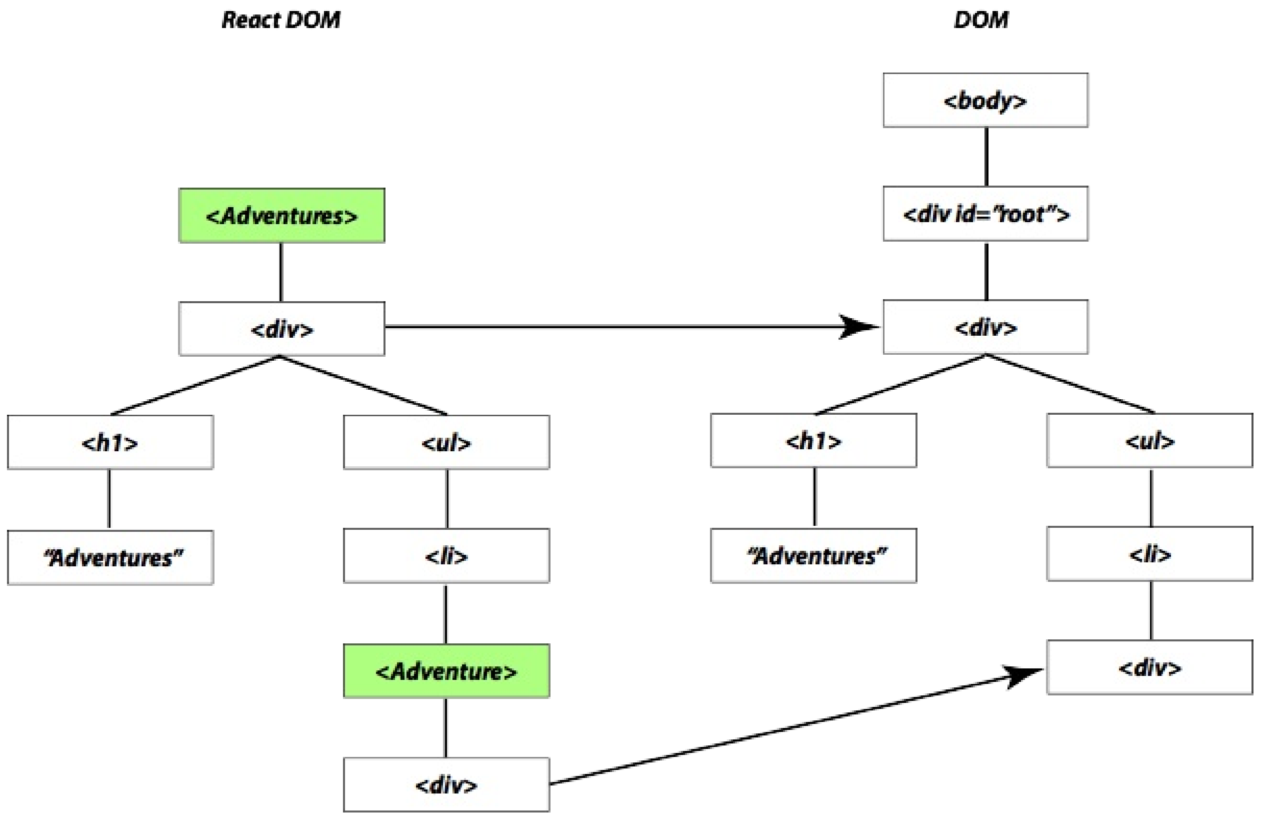
React DOM and Life Cycle

React DOM



- The browser DOM is not JavaScript, heavyweight, and slow
- React maintains a light-weight DOM away from the browser DOM
- The React DOM is faster to manage

Rendering



- React creates components in the React DOM
- The browser DOM is synchronized to the React DOM
- React compound components are not present in the browser DOM

Updating

- A new component is made only when it does not already exist in the DOM
- If it replaces an existing component, the old component and children are dismounted
- If an existing component is updated, only the props are changed

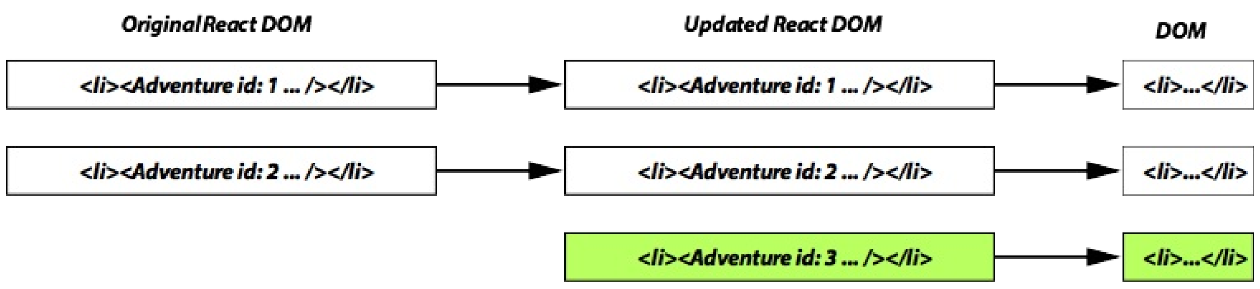
React DOM Rendering

- Updating creates a second (possibly partial) DOM
- Only new or changed components need to be rendered to the browser
- This speeds up browser rendering

React DOM Rendering

- React is efficient
- It does not replace browser DOM elements if it can simply update them

Appending Adjacent Components

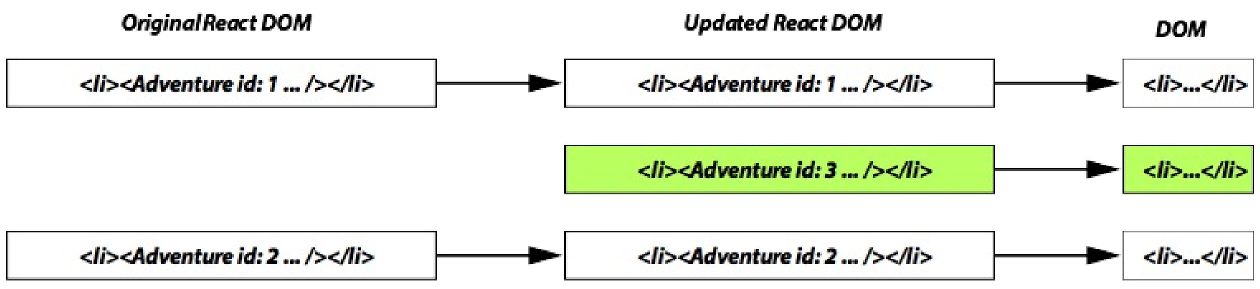

Original React DOM
Updated React DOM
DOM
<li><Adventure id: 1 ... /></li>
<li><Adventure id: 2 ... /></li>
<li><Adventure id: 3 ... /></li>
<li>...</li>

- Adding a new component simply adds it at then end
- Since there is a new component, a new DOM element is appended

Inserting a Component

- Adjacent components of the same type may create a problem
- When a new component is *inserted*, or an existing component *deleted*
- React will update another existing component with the wrong props, and state is transfered

Updating with Keys


Original React DOM
Updated React DOM
DOM
<li><Adventure id: 1 ... /></li>
<li><Adventure id: 3 ... /></li>
<li><Adventure id: 2 ... /></li>
<li>...</li>

- Keys identify adjacent components uniquely
- React will not match the wrong props with existing component

Component Events

React DOM and Life Cycle
Component Events
References

Component Methods

- React provides a series of events during a component's life cycle
- Component events are delivered as method calls
- The events execute in a well-defined order

Mounting

```
constructor(props)
getDerivedStateFromProps(props, state)
render()
componentDidMount(prevProps, prevState, snapshot)
```

- A new component instance is created and mounted
- The methods are called in this order

getDerivedStateFromProps

```
static getDerivedPropsFromState(props, state)
```

- Infrequently used in the case state must be updated from new props
- Return an object to update the state (as setState expects)
- Return null to not do anything

componentWillMount

```
constructor(props)
getDerivedStateFromProps(props, state)
componentWillMount()
render()
componentDidMount(prevProps, prevState, snapshot)
```

- componentWillMount is deprecated

Update

```
getDerivedStateFromProps(props, state)
shouldComponentUpdate(nextProps, nextState)
render()
getSnapshotBeforeUpdate(prevProps, prevState)
componentDidUpdate(prevProps, prevState, snapshot)
```

- The state or props are modified on an existing component
- *shouldComponentUpdate* may return false to abort an update
- *getSnapshotBeforeUpdate* - any value returned from this becomes the snapshot for *componentDidUpdate*

componentWillReceiveProps and *componentWillUpdate*

```
getDerivedStateFromProps(props, state)
componentWillReceiveProps(nextProps)
shouldComponentUpdate(nextProps, nextState)
componentWillUpdate(nextProps, nextState)
render()
getSnapshotBeforeUpdate(prevProps, prevState)
componentDidUpdate(prevProps, prevState, snapshot)
```

- *componentWillReciveProps* and *componentWillUpdate* are both deprecated

Dismount

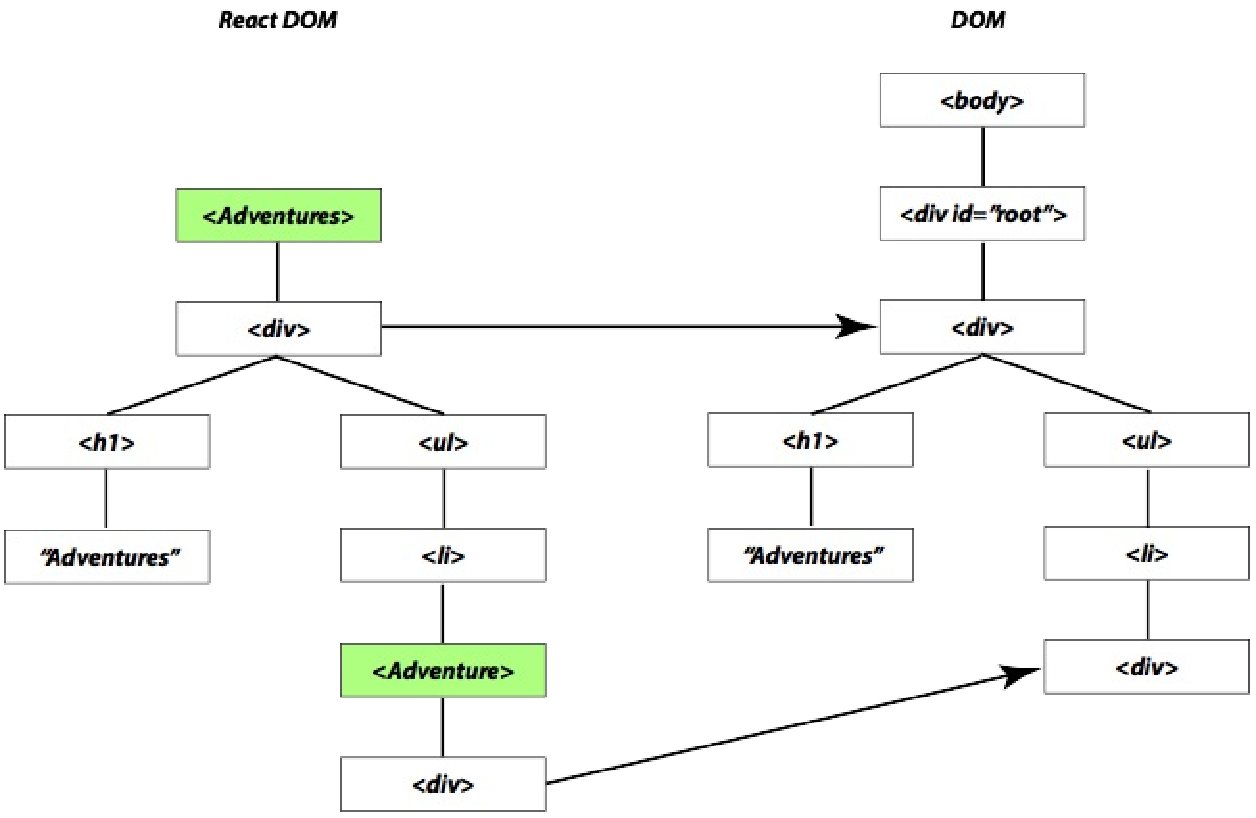
```
componentWillUnmount()
```

- The component instance is released (garbage collected) after a dismount

References

React DOM and Life Cycle
Component Events
References

References



- References point to a browser element
- Or a React component that does not map to a browser element

Why a Reference?

- Manage element focus, text selection, media playback
- Integrate with libraries like jQuery
- Trigger DOM animations

ID Prop

```
render() {
    return (
        <div id="someid">
        </div>
    )
}
```

- A simple way to find a browser element is with an *id*
- The problem is the application does not know when it is added to the browser DOM!

ref Prop

```
render() {
    return (
        <input type="text" ref="nameField" />
    )
}
onSomething() {
    this.refs.nameField.focus()
}
```

- A *ref* prop may be assigned a string name
- A property with that name and the reference will be created in *this.refs*
- This form of using *ref* is deprecated

ref to Named Property

```
constructor(props) {
    super(props)

    this.nameField = React.createDef()
}
render() {
    return (
        <input type="text" ref={ this.nameFIeld } />
    )
}
onSomething() {
    this.nameField.focus()
}
```

ref Prop

```
render() {
    return (
        <input type="text" ref={ (backingRef) => { } } />
    )
}
```

- The *ref* prop may be assigned a callback

The Reference

- The backing browser DOM element if it exists, or a component reference
- Null if the React component is being dismounted
- Null for an inline/arrow function being replaced on an update

The Browser Element

```
render() {
    return (
        <input type="text" ref={ (nameField) => {
            if (nameField) {
                nameField.focus()
            }
        } } />
    )
}
```

- Only use direct browser DOM manipulation when there is no other way
- One example is to set the focus after rendering
- Of course, the prop autofocus={ true } would also accomplish this

Checkpoint

- What happens during the component life cycle?
- When can a browser element reference be retrieved?
- Why does React have problems with adjacent elements?
- What is the first event in the component life cycle?
- Why is the life cycle important for asynchronous programming?
- What events are deprecated?

Lab 8 - React-DOM

The lab for this chapter was to work through the examples and verify the results using the browser developer tools.

Estimated Completion Time

0 minutes

Congratulations, you have completed this lab!

Chapter 8 - React-Redux

Redux
React-Redux

Objectives

- Manage a components state
- Leverage component events to manage state

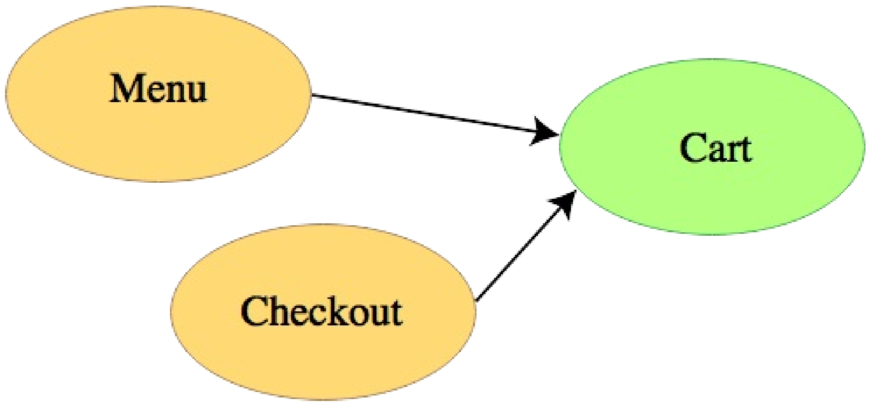
Overview

Redux provides an application-wide data store that components can share. While managing state and pushing immutable data down to child components helps significantly with avoiding cross-component communication, pushing data through a tree of components that do not really use it increases dependency between the parents and children. Managing a data store that multiple components can see and share at any level in the tree eliminates this problem.

Redux

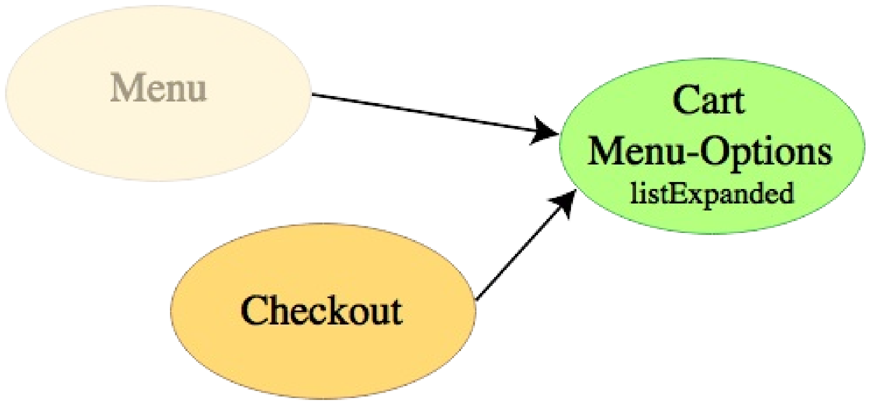
Redux
React-Redux

Global State



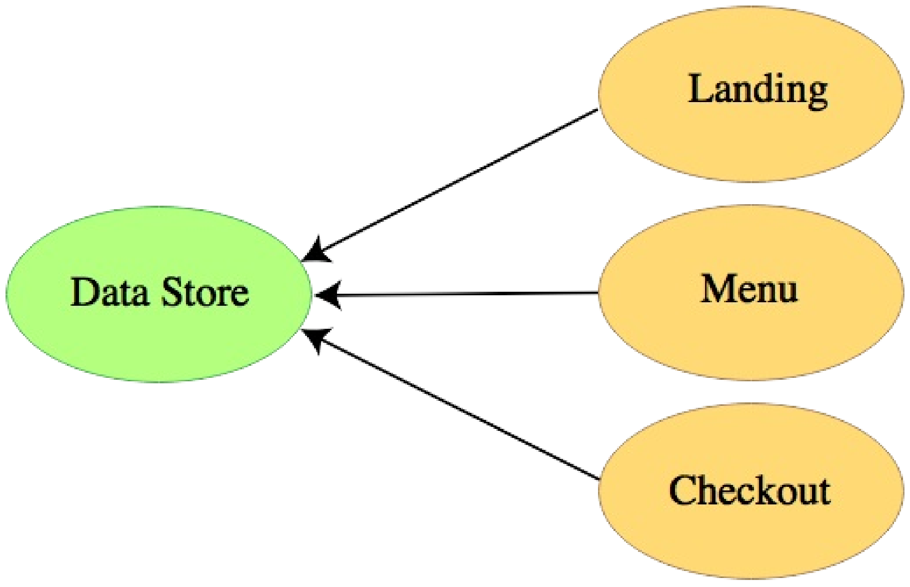
- Programmers struggle with global state
- One solution would be a singleton object shared by the whole application
- React is focused on presentation, so this is not a React issue

What Goes In Global State?



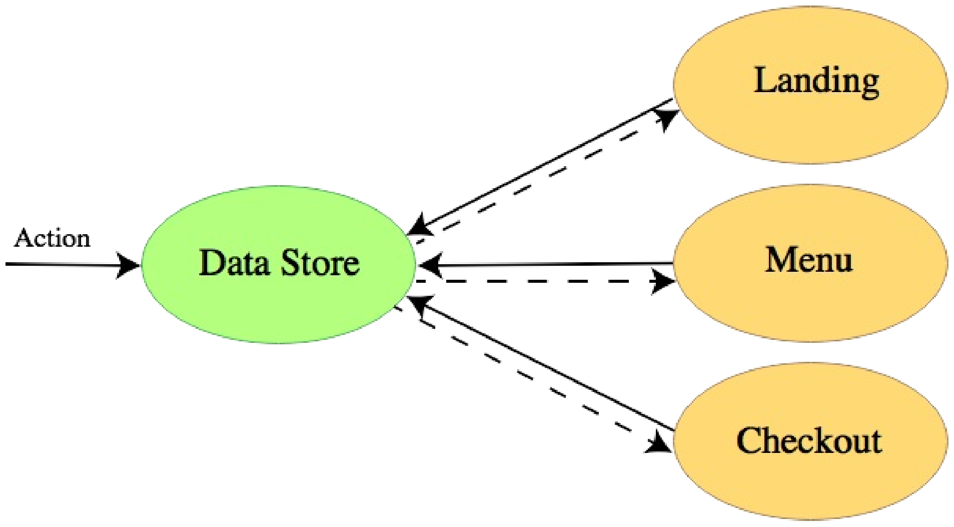
- Mixture of data and presentation state
- Data arriving from an external sources, and updated periodically
- Presentation state maintained over the life of the application when components are rendered or not

Flux



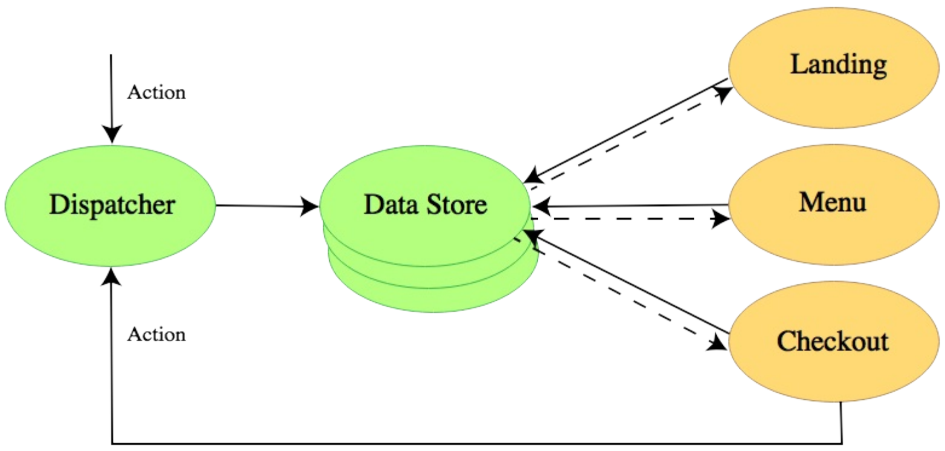
- Flux addresses global state with immutable stores
- Flux mimics component state, but for the whole application
- Components listen for events changing a store

Flux Action



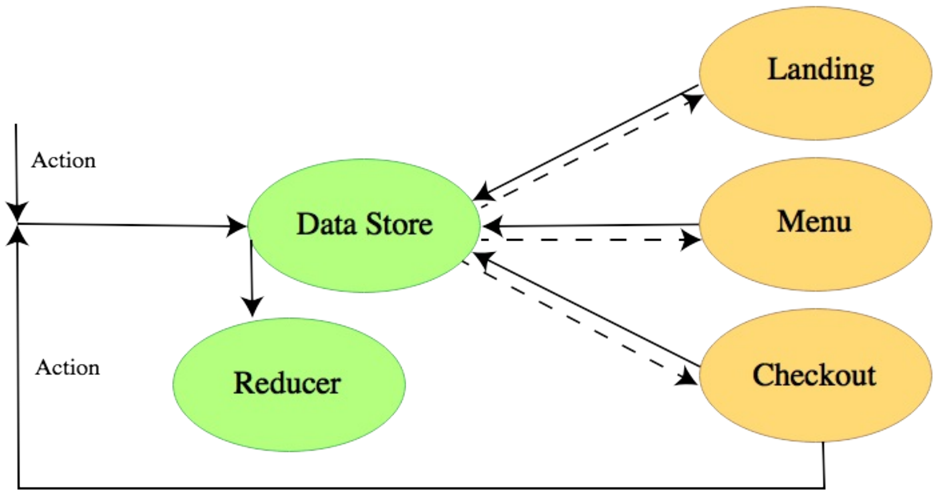
- Actions arriving at the store trigger notifications to observers

Flux Dispatcher



- All actions are sent to a dispatcher
- The dispatcher sends all actions to every store
- Flux allows for multiple stores

Redux



- Redux simplifies Flux: single store, reducer called from the store
- The dispatcher is part of the store
- The reducer's job is to merge a copy of the store with new data, replacing the store

React-Redux

Redux
React-Redux

React-Redux

```
import Redux from 'redux'
import ReactRedux from 'react-redux'
```

- React-Redux integrates Redux and React
- Both Redux and React-Redux must be imported

Defining Actions

```
export const GET_BEVERAGES_ACTION = 'get_beverages_action'
export const GET_PASTRIES_ACTION = 'get_pastries_action'
```

- Start with defining what actions may take place
- Needed in both the store and the reducer, so put them in a separate module *modelActions*
- Actions can be any value, but using a string helps debugging

Defining a Reducer

```
import { GET_BEVERAGES_ACTION, GET_PASTRIES_ACTION } from './modelActions'

class ProductReducer {

    constructor() {

        this.reduce = this.reduce.bind(this)
    }

    reduce(state, action) {
    }
}

export default (new Reducer()).reduce
```

- The reducer must be defined before the store
- The reducer *function* takes an existing store and merges the changes dictated by the action
- export just the reducer function

Actions

```
let action = {
    type: GET_BEVERAGES_ACTION
    beverageList: []
}
```

- Actions are objects, each may have unique properties: one property should be the action type
- Any other properties are the data the action needs
- Action names are funny: GET_BEVERAGES_ACTION *sets* the beverages into the store (more later)

Reducing

~~~
reduce(state, action) {

    switch (action.type) {
        case GET_BEVERAGES_ACTION:
            return getBeveragesAction(state, action)
            break
    }
}

getBeveragesAction(state, action) {

    let newState = { state }

    newState.beverageList = action.beverageList

    return newState
}

* Replicate any data in the store being changed
* If an object, change the property value and change the object reference
* If an array, create a new array but keep any values except what is changed

### Unknown Actions

```javascript
switch(action.type) {
    case GET_BEVERAGES_ACTION:
        return getBeveragesAction(state, action)
        break

    default:
        break
}
~~~

- Ignore unknown actions; error messages will become frequent
- All actions are sent to all reducers (more later)
- There are some actions in the Redux space, not the application space

Immutability Helpers

```javascript
import update from 'immutability-helper'
```

```
let newData = update(oldData, { x: { y: { z: { $push: [ 99 ] }}}})
```

- Do not waste time duplicating the portions of an object tree for reduction
- Leverage an existing immutability helper: https://github.com/kolodny/immutability-helper
- This merges new branches with the original tree and returns the results

Defining a Store

```
import { combineReducers, createStore } from 'redux'
import productReducer from 'product-reducer'

class ReduxModel {

    constructor() {

        this._store = createStore(productReducer, this.initialState)
    }

    get store() {

        return this._store
    }

    get initialState() {

        return {

            beverages: [],
            pastries: []
        }
    }
}

let model = new ReduxModel()
export default model
```

- Defining a store is simple, but the reducer is required
- Maybe define the store as a class, but it does not have to be

Combining Reducers

```
import { combineReducers, createStore } from 'redux'

import cart from '../Cart/cart';
import productReducer from './productReducer'
import orderReducer from './orderReducer'

class ReduxModel {

    constructor() {

        let reducer = combineReducers( { product: productReducer, order: orderReducer } )
        this._store = createStore(reducer, this.initialState)
    }
    get store() {

        return this._store
    }

    get initialState() {

        return {

            product: {
                beverages: [],
                pastries: []
            },

            order: new cart()
        }
    }
}

let model = new ReduxModel()
export default model
```

- If the store has multiple features, use multiple reducers
- Note: the reducers only get and should return *their part* of the store
- The division of the state is based on name: *orderReducer* gets *order*, *productReducer* gets *product*

Defining Action-Creators

```
import { GET_BEVERAGES_ACTION, GET_PASTRIES_ACTION } from './modelActions'

class ProductActionCreator {

    constructor(dispatch) {

        this.dispatch = dispatch;
    }

    getBeverages() {

        // Where did beverages come from?

        this.dispatch({ type: GET_BEVERAGES_ACTION, beverageList: beverages }) )
    }
```

- The same actions are created all over the place
- DRY: create an object with methods to create action objects, and delegate to it
- Dispatch to the store; the dispatcher is injected when the class is instantiated

Action-Creator Responsibilities

```
import dataContext from '../Data-Access/dataContext'
import { GET_BEVERAGES_ACTION, GET_PASTRIES_ACTION } from './modelActions'
```

```
class ProductActionCreator {

    constructor(dispatch) {

        this.dispatch = dispatch;
    }

    getBeverages() {

        dataContext.beverageContext.getBeverages()
            .then( (beverages) => this.dispatch({ type: GET_BEVERAGES_ACTION, data: beverages }) )
            .catch( (error) => console.log(error) )
    }
```

- Let the action-creator dispatch the action to the store
- Let the action-creator have the responsibility of the business logic; the promise is handled in one place (DRY)
- This explains "getBeverages" instead of "updateBeverages"

Prop Injection

```
import { connect } from "react-redux";
import OrderActionCreator from "../Model/OrderActionCreator";
import ProductActionCreator from "../Model/ProductActionCreator";

class Menu {
}

export default connect((state, ownProps) => {

    return {

        beverages: state.product.beverages,
        pastries: state.product.pastries,
        order: state.order
    }

}, (dispatch, ownProps) => {

    return {

        orderActionCreator: new OrderActionCreator(dispatch),
        productActionCreator: new ProductActionCreator(dispatch)
    }

})(Menu)
```

- *connect* maps the upstream store state and dispatcher into props named as the properties of the returned objects, building a new component class
- *mapStateToProps* to inject desired store data as props, *mapDispatchToProps* to inject the dispatcher as a prop
- Notice that the dispatcher is injected into new action-creators, which are then injected as props

Prop Injection

```
import { connect } from "react-redux";
import OrderActionCreator from "../Model/OrderActionCreator";
import ProductActionCreator from "../Model/ProductActionCreator";

class Menu {

    static mapStateToProps(state, ownProps) {

        return {

            beverages: state.product.beverages,
            pastries: state.product.pastries,
            order: state.order
        }
    }

    static mapDispatchToProps(dispatch, ownProps) {

        return {

            orderActionCreator: new OrderActionCreator(dispatch),
            productActionCreator: new ProductActionCreator(dispatch)
        }
    }
}

export default connect(Menu.mapStateToProps, Menu.mapDispatchToProps)(Menu)
```

- A cleaner way to handle it may be to use static methods in the class, since the mapping is cohesive with the class

External Prop Injection

```
import { connect } from "react-redux";
import Menu from './Menu'
import OrderActionCreator from "../Model/OrderActionCreator";
import ProductActionCreator from "../Model/ProductActionCreator";

class MenuContainer {

    static mapStateToProps(state, ownProps) {

        return {

            beverages: state.product.beverages,
            pastries: state.product.pastries,
            order: state.order
        }
    }

    sttaic mapDispatchToProps(dispatch, ownProps) {

        return {

            orderActionCreator: new OrderActionCreator(dispatch),
            productActionCreator: new ProductActionCreator(dispatch)
        }
    }
}
```

```
export default connect(MenuContainer.mapStateToProps, MenuContainer.mapDispatchToProps)(Menu)
```

- Or, move the responsibility into a container component that wraps the expected component

Checkpoint

- Why is managing global state important?
- What did Flux offer as a big advantage?
- How does Redux enhance the Flux paradigm?
- Should action-creators also be responsible for business logic?
- How is data from the store injected into observing components?
- What is the purpose of the dispatcher?