Lab 01 - Introduction

Estimated time: 60 minutes.

Goals

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

Synopsis

This is an instructor-lead, group-lab that first introduces the roles of NodeJS, npm, and WebPack in a React project. Then the group explores using JSX to render basic React elements that display the date, time, and a welcome message. The welcome message is conditional on the time: "Good morining," "Good afternoon," or "Sorry, we are closed for the rest of the day."

Requirements

- Build a landing page that displays the cafe header, the date and time, and a welcome message.
- If the time is before noon, display "Good morning!" followed by "Welcome to the Carribean Coffee Company!"
- If the time is before 5 pm, display "Good afternoon!" followed by "Welcome to the Carribean Coffee Company!"
- If the time is after 5 pm, the message is "Welcome to the Carribean Coffee Company! Sorry, we're closed for the rest of the day."

Project Steps

1. Work in the Labs/01_Introduction folder. Explore how the project uses Node as the environment for building the react application, and npm to install production and development dependencies. We need to compile ES6 and ES7 code to ES5, and bundle it for delivery to the client.
2. Look at package.json. The project uses Node for build with the scripts ***build*** and ***dev***. Build will compile the source and build a bundle of ES5 code. Dev launches the WebPack development web server. The server creates the bundle in memory, not disk. It monitors the source folders and recompiles, re-bundles, and pushes the changes to the browser each time a source file is saved.
3. Install the npm packages.
4. Explore webpack.config.js. We are only interested in a quick look at what files it parses and the bundle it builds.
5. Examine the index.html file; it links to the CSS file and includes the bundle at the end of the body.
6. Create the app.jsx that renders Hello, World! in an h1 React element. Assign the JSX element to the variable ***content*** and use ReactDOM.render to inject the variable into the div with the id ***react-content*** in index.html.
7. Run npm run dev and Test the page. Remember to look on the console for errors. Use the browser developer tools to explore the bundle and the source. Set breakpoints in the source and watch what happens.
8. Change the app.jsx file to delay running until the load event fires, and move the script tag to the head in index.html.

9. Check the page. Changing a static file will not cause the changes to be pushed to the browser or the page to be reloaded, so refresh it and check the source.
10. Insert a div in front of "Hello, World!" with className "page" to wrap the rendered contents. Highlight using the DOM className property instead of class. This fixes the margins; check the page.
11. Nest a div with className "header" to wrap the header.
12. Nest an img with src "assets/images/tccc-logo.png"
13. Check the page.
14. Use npm to install the ***moment*** package.
15. In app.jsx import the moment package.
16. In front of the return statement use an if statement and the getHours method of the current date to check if the current hour is before 5 pm. When it is, set a ***message*** variable to a span element containing the.message described in the requitements. Use a conditional (ternary) operator to pick the morning or afternoon value.
17. If the hour is after 5 pm, set the ***message*** variable to the closed message as in the requirements.
18. In the content assignment replace the ***h1*** element containing "Hello, World!" with a ***span*** element containing a jsx expression to format the date and time using ***moment***.
19. Follow it with a jsx expression to inject the value of ***message***.
20. Check the page.

Lab 2 - Components

Goals

- Build and use a hierarchy of custom components
- Seperation of concerns and single responsibility
- Promote reusability through components

Synopsis

The focus is to start organizing the Cafe application. The project needs a header, footer, and for the moment a landing view for the content. Just for fun, some of the content on the landing page is going to change based on the time of day.

Project Requirements

1. Organize the application; start by merging the Resources/O2_Components/src folder with the project.
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 Components for the application header and footer.
5. Add a Landing component that welcomes the visitor. If the hour is in the morning, it should say "Good morning, welcome..." 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. The landing view should be wrapped in a div that uses the app-content class. Eventually, the landing view will be replaced.
7. Fix the index module to use the new CSS file and remove the registration worker.

Steps

1. Copy the 01_React lab to 02_Components.
    - If the application fails to start after moving it, try removing node_modules and run "npm install" to rebuild it.
2. Merge the contents of Resources/02_Components/src with the src folder in the new project.
3. Peak at the contents of the Assets folder; take a look at the application.css file.
    - Note the changes that were made; 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.
    - These styles will be used throughout the application, so take some time to familiarize yourself with them.
4. Move the App and App.test modules to the App folder.
5. Remove the App.css, index.css, logo.svg, and registerServiceWorker.js files.
6. 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 copyright notice.

7. 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.

8. Update the App component to render the new boilerplate:

    - Change the App component to include the "application.css" file instead of App.css.
    - Have 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."
    - In the content div, render a Landing component.

9. 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 and call to registerServiceWorker.

10. Save and check your work.

11. Open 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.

Results

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

1. The components are organized by facets.
2. The assets have been moved 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 as there was (index and App). If module specific CSS is necessary then add a file for the module, but only if it is not shared.
4. Conditional statements are used to control what is rendered.

Congratulations, you have completed this lab!

Lab 3 - Props

Goals

- Continue to build out the hierarchy of application components
- Create a container component
- Let data flow down to child components

Synopsis

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

Project Requirements

1. Build and use a Menu component in place of the Landing to show the beverages and pastries for the cafe.
2. Add Beverage and Pastry classes to the application.
3. The Menu component will be a container that holds the data and passes it child components to render it.
4. Menu will render two lists: BeverageList and PastryList. Each list renders items: BeverageItem and PastryItem. Each is a div className="list", title div className="list-title", and table className="list."
5. The lists will have a div that displays a title (Beverages, Pastries, etc.) Under the heading will be a table where each row will be an item. THe first row is column titles: (blank) and price. The class names for the columns (th and td) are "list-name" and "list-price."
6. Each Beverage item renders a new table row.
7. The column titles for the Pastries are: (blank), price. The column class names are "list-name" and "list-price."
8. Use the appropriate CSS classes to render the lists and items.

Steps

1. Copy the 02_Components lab to 03_Props.
    - If the application fails to start after moving it, try removing node_modules and run "npm install" to rebuild it.
2. Merge the contents of Resources/03_Props/src with the src folder in the new project.
3. Review the CSS classes for rendering the lists and items.
4. Review the Beverage and Pastry entity classes that are provided in Data-Access.
5. Add the BeverageList component in the Menu facet (folder):
    - Use the array map function to produce an array of BeverageItems from the array of Beverages. A BeverageItem takes a single prop "beverage" which is a Beverage instance.
    - Render a div with class "list."
    - Inside that render another div with class "list-title" and the content "Beverages."
    - After the title div, render a table of class "list."
    - Add a row and the header columns: (empty) and "price." The CSS classes are "list-name" and "list-price."

    - Render the list of BeverageItems.

6. Add the PastryList component following the pattern of the BeverageList. There are two columns, the name and the price.

7. 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 the BeverageList and the PastryList.

8. Replace the Landing component with the Menu component (temporarily) in the Asp component.

9. Save and check your work.

Results

This lab accomplished building a container component and pushing data down through props:

Congratulations, you have completed this lab!

Lab 4 - State

Goals

- Manage data in state

Synopsis

State is critical because components automaticall 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. Change the props in Menu to pass a data source instead of the data.
2. Each list should query the data source for it's own data.
3. The data query returns a promise, so the list should update its own state when the promise is fufilled.

Steps

1. Copy the 03_Props lab to 04_State.
    - If the application fails to start after moving it, try removing node_modules and run "npm install" to rebuild it.
2. Merge the contents of Resources/04_State/src with the src folder in the new project.
3. 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/beverages and getting the list of beverages.
4. 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.
5. 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'
```

    - Change the rendering of the BeverageList and PastryList to push the dataContext as the only prop (as 'dataContext'); The lists do not load the data context themselves, because then which data context is used can be changed in one place!

6. Change the two lists to query the data context:

    - Initialize and empty array in state in the constructor for each list: *this.state.beverages* or *this.state.pastries*.
    - Add another state property ***error*** set to null.
    - Query the appropriate data source. This can be done in the constructor, hint: props is the first argument to the constructor. Note: this web service is configured using CORS to allow the application to communicate with it directly, so it is OK that the application loads on a different URL than the data source. The service is a JavaScript NodeJS application, check it out.
    - Set the state property (beverages or pastries) when the promise resolves, and make sure the error property is set to false.
    - If the promise is rejected, log the error to the console and set the error property in state to true.
    - Change the rendering so that if the error property of the state is true, it prints a span of class ***error*** with a message stating the list could not be loaded.
    - Otherwise, use the array (beverages or pastries) in the state to render the list of items.

7. Save and check your work.

8. Check the application with the web service disabled and make sure the errors work.

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 component re-renders with the actual data! If the AJAX request is rejected (incorrect or simply the service is not available), an error message is displayed.

Congratulations, you have completed this lab!

Lab 5 - Composition

Goals

- Favor composition over inheritance - always!

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 be solved by a strategy pattern, perhaps. Object-oriented paradigms already recognize that composition should be favored over inheritance. Our problem of creating an accordian 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 ***AccordianList*** should be reusable for any content, and by default a list should be shown as open.
3. In this case the menu lists should initially be shown as closed, which means the initial open/closed state needs to be passed to the list as a prop ***open***.
4. A ***title*** prop should be passed to the list, which it will display as the title.
5. When the title 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.

Steps

1. Copy the 04_State lab to 05_Composition.
    - If the application fails to start after moving it, try removing node_modules and run "npm install" to rebuild it.
2. Create a Common folder and an AccordianList.js module.
3. The AccordianList 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.
    - The component should expect props ***open*** and ***title. title*** is required.
    - ***open*** should have a default value of ***false***.
4. In the BeverageList component wrap the table with an AccordianList. This is how the AccordianList is leveraged: BeverageList renders an AccordianList to re-use its drop-down functionality, and injects the contents as the child of the AccordianList.
5. Save and check your work.

6. Check the application with the web service disabled and make sure the errors work.

Results

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

Congratulations, you have completed this lab!

Lab 6 - DOM

Goals

- Fix the re-rendering of the lists.

Synopsis

Add keys to the list items so changes in the list are reflected properly in the DOM.

Project Requirements

1. Add keys to the list items, both for the BeverageList and the PastryList.

Steps

1. Copy the 05_Composition to 06_React-DOM.
    - If the application fails to start after moving it, try removing node_modules and run "npm install" to rebuild it.
2. Use the primary key in the entity classes, *id*, as the key for the items generated in BeverageList and PastryList.
3. Save and check your work. Make sure there aren't any messages about the keys on the JavaScript console.

Results

Adding keys to the list items fixes the re-rendering vs state issues when items are added to or deleted from the list. It also removes the warning messages generated while the application is running.

Congratulations, you have completed this lab!