Lab 13: React + Bootstrap - II

Version 2: Portfolio SPA - Reactstrap, JSX, props, babel

Portfolio App:
React + Bootstrap

IMPORTANT NOTE:
Use HTTP protocol for this LAB, cannot use File protocol.

Table of Content for Implementing Profile SPA - Reactstrap:

Lab Introduction

Prerequisites

Must have completed Lab 12: React + JSX, as this lab is a direct continuation from it.

Motivation

Build Portfolio page using React, JSX, & styled using Reactstrap components.

Goal

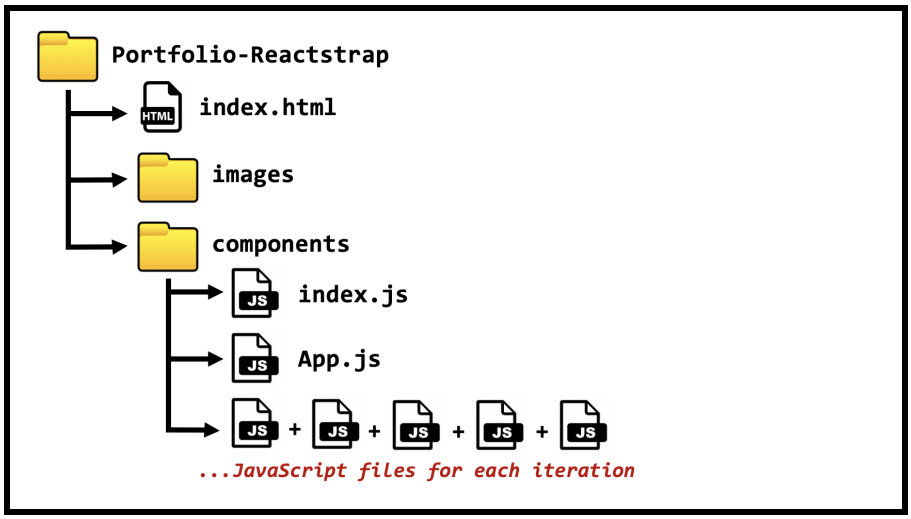
Use the React JS library to define an App's HTML view & use JS library to define the styles of the app.

Learning Objectives

- React, JSX, Components, Props, Virtual DOM
- Babel, Babel Standalone, Build tools
- Deployment: HTTP protocol vs file protocol
- Reactstrap, Bootstrap v4

Client-side Architecture:

Start this project by making a project folder where all your assets & scripts will be organized.
Create all necessary files and folders as illustrated below.



Concepts

Single Page Application (SPA)

An SPA is a web app implementation that loads a single web document, and updates the body content via JavaScript APIs (such as Fetch) when different content is to be shown.

React

React is a JavaScript library for building user interfaces. It manages your app's view. Learn more: https://reactjs.org/

- **React Components:** React is an object-oriented approach for designing modular, reusable views. Build encapsulated components that manage their own state, then compose them to make complex UIs.
- **React Props:** React components may be instantiated with specified properties (i.e. constructor arguments)
- **React State:** React components maintain their own state (i.e. instance variables) and will re-render the view whenever that state changes

---

Virtual DOM

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. Learn more: https://reactjs.org/docs/faq-internals.html#what-is-the-virtual-dom

---

JSX

JSX is a syntax extension to JavaScript. Use it to define the HTML elements. JSX may look like HTML, but it comes with the full power of JavaScript. https://reactjs.org/docs/introducing-jsx.html

---

Babel

Babel is a compiler that converts JSX code into JavaScript. Babel requires NodeJS installed. However, a standalone build of Babel for use in non-Node.js environments is available by using a backend service. Learn more at: https://babeljs.io/docs/en/index.html

---

Localhost

To serve web pages with HTTP protocol, you must launch a HTTP server application from the directory that your HTML files are in.

Iteration 1: Component: App

'Plan' Phase -- Approach

Goal #1: Define HTML & Setup Simple React App

Approach: HTML for React Project

React is a JavaScript library for the View. It allows a OOP-approach for developing reusable modular HTML components. React uses JSX syntax to define the presentation layout as markup. JSX code is not natively supported by the browser, so it must be compiled with a tool called babel. Note: Must use HTTP protocol to transpile code using babel standalone, thus file protocol will not work for this project.

'Do' Phase -- Apply

HTML Steps

`index.html` must import JavaScript dependencies for Bootstrap, React, React Dom, Babel, Reactstrap. The HTML file must also import all of your React components in the form of Javascript files.

Step 1 (HTML): Head element

The HTML head element imports most of the external JS dependencies & CSS.

index.html

```
<head>
  <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">
  <script src="https://unpkg.com/react/umd/react.development.js"></script>
  <script src="https://unpkg.com/react-dom/umd/react-dom.development.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/reactstrap/4.8.0/reactstrap.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.21.1/babel.min.js"></script>
</head>
```

Step 2 (HTML): Body element

The HTML body element defines an element to mount the app's view onto & import the React view.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

Step 1 (JS): Create & define: index.js

index.js: Mount App's view to a target HTML element & render to Browser from ReactDOM

index.js

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

Step 2 (JS): Create & define: App.js

Define an App class that is a React Component and overrides the render method that defines the HTML view with React JSX

App.js (class) App

```
class App extends React.Component {
   render() {
      return (
         <div>
            <h1> Hello World </h1>
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The Welcome section should render a Jumbotron to the Browser viewport.

Iteration 2: Component: Welcome

'Plan' Phase -- Approach

Goal #2: Add a Welcome section

Approach: Use React Component to define the Welcome section

'Do' Phase -- Apply

HTML Steps

1. The HTML file must import the React component (Javascript file).

Step 1 (HTML): Body element

The HTML must import the JavaScript script containing the Welcome component & label it for babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define a React component that renders the Welcome section's HTML
2. Declare the JSX Welcome tag in the App component

Step 1 (JSX): Create: Welcome.js

Destructure the Reactstrap variables: Jumbotron & Container. Define class Welcome that extends a React Component. Override the render method that defines the Welcome HTML in JSX. The JSX uses the Reactstrap Component tags named for the Bootstrap stylings.

Welcome.js

```
const {Jumbotron, Container} = Reactstrap;

class Welcome extends React.Component {
   render() {
      return (
         <Jumbotron>
            <Container id="home">
               <h1> My Title Here. </h1>
               <p> description goes here </p>
            </Container>
         </Jumbotron>
      );
   }
}
```

Step 2 (JSX): Refactor: App.js

Refactor the App class render method with the Welcome component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <Welcome />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The Welcome section should render a Jumbotron to the Browser viewport.

Iteration 3: Component: News

'Plan' Phase -- Approach

Goal #3: Add a News section

Approach: News as a React Component

'Do' Phase -- Apply

HTML Steps

1. HTML file must import the News React component (JSX file) as type Babel to transpile into JS.

Step 1 (HTML): Body

The HTML must import the JavaScript script containing the News component & label it for babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/News.js'></script>
  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define a React component that renders the News section's HTML
2. Declare the JSX Welcome tag in the App component

Step 1 (JSX): Create: News.js

Destructure the Reactstrap variable: Container. Define class Welcome that extends React Component. Override its render method to define the HTML for News in JSX syntax. This JSX uses Reactstrap tags named after the Bootstrap styles.

News.js

```
const {Container} = Reactstrap;

class News extends React.Component {
   render() {
      return (
         <Container className="bg-danger">
            <h1>Welcome</h1>
            <hr/>
            <p> This section summarizes what the page is about and its purpose. </p>
         </Container>
      );
   }
}
```

Step 2 (JSX): Refactor: App.js

Refactor the App class render method with the Welcome component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <Welcome />
            <News />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The News section should render into the Browser's viewport.

Iteration 4: Component: About

'Plan' Phase -- Approach

Goal #4: Add an About section

Approach: Define 'About' section as a React Component

'Do' Phase -- Apply

HTML Steps

1. HTML must import the 'About' component (JSX file) as Babel type to transpile into JS.

Step 1 (HTML): Body

HTML must import the 'About' component & label it for Babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/About.js'></script>
  <script type='text/babel' src='components/News.js'></script>
  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define a React component that renders the About section's HTML
2. Define an inner class component that renders Profile
3. Define an inner class component that renders Skill table
4. Declare the JSX 'About' tag in the App component

Step 1 (JSX): Create: About.js → *class About*

Destructure Reactstrap object for component: Container. Define class 'About' extends Component. Override render method to define HTML in JSX syntax. The JSX uses both Reactstrap tags & React tags.

About.js → (class) About

```
const {Container} = Reactstrap;

class About extends React.Component {
   render() {
      return (
         <Container id="about" className="bg-primary">
            <h1>About</h1>
            <hr/>
            <Profile />
            <SkillTable />
         </Container>
      );
   }
}
```

Step 2 (JSX): Refactor: About.js → *class Profile*

Destructure Reactstrap object for components: Row, Col. Define a class 'Profile' extends Component. Override render method to return HTML in JSX syntax. The JSX uses both Reactstrap tags & HTML tags.

About.js → (inner class) Profile

```
const {Row, Col} = Reactstrap;

class Profile extends React.Component {
   render(){
      return(
         <Row>
            <Col>
               <img src="images/profile.png" className="img-fluid p-1" />
            </Col>
            <Col>
               <h4> First Last </h4>
               <p> This is info about me. </p>
            </Col>
         </Row>
      );
   }
}
```

Step 3 (JSX): Refactor: About.js → *class SkillTable*

Destructure Reactstrap object for components: Table. Define a class 'SkillTable' extends Component. Override render method to return HTML in JSX syntax. The JSX uses both Reactstrap tags & HTML tags

About.js → (inner class) SkillTable

```
const {Table} = Reactstrap;

class SkillTable extends React.Component {
   render() {
      return (
         <Table>
            <thead>
               <tr>
                  <th colSpan='2' className='text-center'>Technical Skills</th>
               </tr>
            </thead>
            <tbody>
               <tr>
                  <th scope="row">Backend tools</th>
                  <td>JavaScript, Node, NPM, Express, Passport</td>
               </tr>
               <tr>
                  <th scope="row">Frontend tools</th>
                  <td>JavaScript, HTML, CSS, React, Bootstrap, Reactstrap</td>
               </tr>
               <tr>
                  <th scope="row">Database tools</th>
                  <td>JavaScript, MongoDB, Mongoose</td>
               </tr>
            </tbody>
         </Table>
      );
   }
}
```

Step 4 (JSX): Refactor: App.js

Refactor the App class render method with the Welcome component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <Welcome />
            <News />
            <About />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The About section should render into the Browser's viewport.

Iteration 5: Component: Blog

'Plan' Phase -- Approach

Goal #5: Add a Blog section, use React properties

Approach: Define 'Blog' section as a Component & Properties to pass data

The Blog component will contain BlogPost components. React properties passes data into a component

---

'Do' Phase -- Apply

HTML Steps

1. HTML must import the 'Blog.js' components (JSX file) as Babel type to transpile into JS.

Step 1 (HTML): Refactor Body

HTML must import the 'Blog' components & label it for Babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/Blog.js'></script>
  <script type='text/babel' src='components/About.js'></script>
  <script type='text/babel' src='components/News.js'></script>
  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define an 'Blog' component that renders the section's HTML, Blog requires BlogPost components
2. Define an inner component that models BlogPost that uses the React properties
3. Declare 'Blog' tag in the App view

Step 1 (JSX): Create: Blog.js → *class Blog*

Destructure Reactstrap object: Container. Define 'Blog' component. Override the render method. The JSX uses both Reactstrap & Blogpost tags. Each BlogPost tag defines its properties to pass to component.

Blog.js → (class) Blog

```
const {Container} = Reactstrap;

class Blog extends React.Component {
   render() {
      return (
         <Container id="blog" className="bg-warning">
            <h1>Blog</h1>
            <hr/>
            <Container>
               <BlogPost title='Post #1' date='Date-01, YYYY' summary="Summary of Post 1" />
               <BlogPost title='Post #2' date='Date-02, YYYY' summary="Summary of Post 2" />
               <BlogPost title='Post #3' date='Date-03, YYYY' summary="Summary of Post 3" />
            </Container>
         </Container>
      );
   }
}
```

Step 2 (JSX): Refactor: Blog.js → *class BlogPost*

Define a 'BlogPost' component. Override the render method. JSX uses both Reactstrap & HTML tags.

Blog.js → (inner class) BlogPost

```
class BlogPost extends React.Component {
   render(){
      return (
         <Container>
            <h3>{this.props.title}</h3>
            <small>{this.props.date}</small>
            <p>{this.props.summary} </p>
            <a className="btn btn-outline-dark" href="#">Read More</a>
            <hr />
         </Container>
      );
   }
}
```

Step 3 (JSX): Refactor: App.js

Refactor App render method to include the Blog component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <Welcome />
            <News />
            <About />
            <Blog />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The Blog section should render into the Browser's viewport.

Iteration 6: Component: Projects

'Plan' Phase -- Approach

Goal #6: Add Projects section, use React properties to pass in data

Approach: Define 'Projects' section as a Component & use Properties to pass data

'Projects' component contains 'ProjectCard' components. Use properties to pass data in to component

---

'Do' Phase -- Apply

HTML Steps

1. HTML imports the 'Projects.js' components (JSX file) as Babel type to transpile into JS.

Step 1 (HTML): Refactor Body

HTML imports the 'Projects' components & label it for Babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/Projects.js'></script>
  <script type='text/babel' src='components/Blog.js'></script>
  <script type='text/babel' src='components/About.js'></script>
  <script type='text/babel' src='components/News.js'></script>
  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define 'Projects' component to render the section's view, requires ProjectCard components
2. Define an inner component that models ProjectCard that uses React properties
3. Declare 'Projects' tag in the App view

Step 1 (JSX): Create: Projects.js → *class Projects*

Destructure Reactstrap object for: Container, CardDeck. Define a 'Projects' component. Override its render method. The JSX uses both Reactstrap & ProjectCard tags. Each ProjectCard tag defines its properties to pass into the component.

Projects.js → (class) Projects

```
const {Container, CardDeck} = Reactstrap;

class Projects extends React.Component {
   render() {
      return (
         <Container id="projects" className="bg-success">
            <h1>Projects</h1>
            <hr/>
            <Container>
               <CardDeck>
                  <ProjectCard type='Type 1' name='Project 1' author="me" />
                  <ProjectCard type='Type 2' name='Project 2' author="myself" />
                  <ProjectCard type='Type 3' name='Project 3' author="I" />
               </CardDeck>
            </Container>
         </Container>
      );
   }
}
```

Step 2 (JSX): Refactor: Projects.js → *class ProjectCard*

Destructure Reactstrap object for: Card, CardHeader, CardImg, CardTitle, CardFooter. Define a 'ProjectCard' component. Override the render method. JSX uses both Reactstrap & properties.

Projects.js → (inner class) ProjectCard

```
const {Card, CardHeader, CardImg, CardTitle, CardFooter} = Reactstrap;

class ProjectCard extends React.Component {
   render(){
      const textformat = "bg-dark text-light text-center";
      return (
         <Card color className={textformat}>
            <CardHeader>{this.props.type}</CardHeader>
            <CardImg src='images/logo.png' />
            <CardTitle> {this.props.name} </CardTitle>
            <CardFooter> {this.props.author} </CardFooter>
         </Card>
      );
   }
}
```

Step 3 (JSX): Refactor: App.js

Refactor App render method to include the 'Projects' component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <Welcome />
            <News />
            <About />
            <Blog />
            <Projects />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The Projects section should render into the Browser's viewport.

Iteration 7: Component: ContactForm

'Plan' Phase -- Approach

Goal #7: Add ContactForm section

Approach: Define 'ContactForm' section as a Component that uses Reactstrap

'ContactForm' component contains 'FormInput' components. Use properties to pass data in to component

---

'Do' Phase -- Apply

HTML Steps

1. HTML imports the 'ContactForm.js' components (JSX file) as Babel type to transpile into JS.

Step 1 (HTML): Refactor Body

HTML imports the 'ContactForm' file & labels it for Babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/ContactForm.js'></script>
  <script type='text/babel' src='components/Projects.js'></script>
  <script type='text/babel' src='components/Blog.js'></script>
  <script type='text/babel' src='components/About.js'></script>
  <script type='text/babel' src='components/News.js'></script>
  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define 'ContactForm' component to render view, requires FormInput components
2. Define an inner component that models FormInput that uses React properties
3. Declare the 'ContactForm' tag into the App view

Step 1 (JSX): Create: ContactForm.js → *class ContactForm*

Destructure Reactstrap object for components: Container, Form. Define a 'Projects' component. Override its render method. The JSX uses both Reactstrap & FormInput tags. Each FormInput tag sets the properties of the component.

ContactForm.js → (class) ContactForm

```
const {Container, Form} = Reactstrap;

class ContactForm extends React.Component {
   render() {
      return (
         <Container id="contact" className="bg-secondary">
            <h1>Contact</h1>
            <hr/>
            <Form>
               <FormInput type='text' placeholder='name' />
               <FormInput type='email' placeholder='email' />
               <FormInput type='textarea' placeholder='message' />
               <button className="btn btn-lg btn-primary">Submit</button>
            </Form>
         </Container>
      );
   }
}
```

Step 2 (JSX): Refactor: ContactForm.js → *class FormInput*

Destructure Reactstrap object for components: FormGroup, Input. Define a 'FormInput' component. Override its render method. JSX uses both Reactstrap & properties (props).

ContactForm.js → (inner class) FormInput

```
const {FormGroup, Input} = Reactstrap;

class FormInput extends React.Component {
   render(){
      return (
         <FormGroup>
            <Input type={this.props.type} placeholder={this.props.placeholder}></Input>
         </FormGroup>
      );
   }
}
```

Step 3 (JSX): Refactor: App.js

Refactor App's render method to include the 'ContactForm' component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <Welcome />
            <News />
            <About />
            <Blog />
            <Projects />
            <ContactForm />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The ContactForm section should render into the Browser's viewport.

Iteration 8: Component: TopNavbar

'Plan' Phase -- Approach

Goal #8: Add TopNavbar section

Approach: Define 'TopNavbar' section as a Component that uses Reactstrap

'TopNavbar' component contains 'NavOption' components. Use properties to pass data into component

---

'Do' Phase -- Apply

HTML Steps

1. HTML imports the 'TopNavbar.js' components (JSX file) & uses Babel to transpile into JS.

Step 1 (HTML): Refactor Body

HTML imports the 'TopNavbar' file with a type of Babel to transpile it from JSX to JavaScript.

index.html

```
<body>
  <div id="root"></div>

  <script type='text/babel' src='components/TopNavbar.js'></script>
  <script type='text/babel' src='components/ContactForm.js'></script>
  <script type='text/babel' src='components/Projects.js'></script>
  <script type='text/babel' src='components/Blog.js'></script>
  <script type='text/babel' src='components/About.js'></script>
  <script type='text/babel' src='components/News.js'></script>
  <script type='text/babel' src='components/Welcome.js'></script>
  <script type='text/babel' src='components/App.js'></script>
  <script type="text/babel" src='components/index.js'></script>
</body>
```

JavaScript Steps

1. Define 'TopNavbar' component to render view, requires NavOption components
2. Define an inner component that models NavOption that uses React properties
3. Declare the 'TopNavbar' tag into the App view

Step 1 (JSX): Create: TopNavbar.js → *class TopNavbar*

Destructure Reactstrap object for components: Navbar, NavbarBrand, Nav. Define a 'TopNavbar' component. Override its render method. The JSX uses both Reactstrap & NavOption tags. Each NavOption tag sets the properties of the component.

TopNavbar.js → (class) TopNavbar

```
const {Navbar, NavbarBrand, Nav} = Reactstrap;

class TopNavbar extends React.Component {
   render() {
       return (
           <Navbar color="dark" fixed="top">
               <NavbarBrand href="/" className="text-light"> Portfolio Name </NavbarBrand>
               <Nav>
                   <NavOption href='#home'     text='Home'     />
                   <NavOption href='#about'    text='About'    />
                   <NavOption href='#projects' text='Projects' />
                   <NavOption href='#blog'     text='Blog'     />
                   <NavOption href='#contact'  text='Contact'  />
               </Nav>
           </Navbar>
       );
   }
}
```

Step 2 (JSX): Refactor: TopNavbar.js → *class NavOption*

Destructure Reactstrap object for components: NavItem, NavLink. Define a 'NavOption' component. Override its render method. JSX uses both Reactstrap & properties (props).

TopNavbar.js → (inner class) NavOption

```
const {NavItem, NavLink} = Reactstrap;

class NavOption extends React.Component{
   render(){
      return(
         <NavItem>
            <NavLink href={this.props.href} className="text-light">
               {this.props.text}
            </NavLink>
         </NavItem>
      );
   }
}
```

Step 3 (JSX): Refactor: App.js

Refactor App's render method to include the 'TopNavbar' component.

App.js

```
class App extends React.Component {
   render() {
      return (
         <div>
            <TopNavbar />
            <Welcome />
            <News />
            <About />
            <Blog />
            <Projects />
            <ContactForm />
         </div>
      );
   }
}
```

'Test' Phase -- Assess

Launch a http server to serve your HTML file and open browser to localhost. The TopNavbar section should render into the Browser's viewport.

Concluding Notes

Final Comments

This is an introductory primer into React with a static site, however to take full advantage of its strengths we should request data from backend services and update the view . We'll use React in more complex ways in future labs where we'll establish web hooks and manage application data.

Future Improvements

- Use Babel locally to compile JSX into Javascript
- Use React state and re-render the view
- Use web hooks

Lab Submission

Compress your project folder into a zip file and submit on Moodle.