Exercise 2-3

Using React and Office UI Fabric React Components

In this exercise you will create a SharePoint Framework client-side web part that leverages the React web framework.

NOTE: The instructions below assume you are using v1.11.0 of the SharePoint Framework Yeoman generator.

1. Open a command prompt and change to the folder where you want to create the project.
2. Run the SharePoint Yeoman generator by executing the following command:

yo @microsoft/sharepoint

Use the following to complete the prompt that is displayed:

* + **What is your solution name?**: ReactDemo
  + **Which baseline packages do you want to target for your component(s)?**: SharePoint Online only (latest)
  + **Where do you want to place the files?**: Use the current folder
  + **Do you want to allow the tenant admin the choice of being able to deploy the solution to all sites immediately without running any feature deployment or adding apps in sites?**: No
  + **Will the components in the solution require permissions to access web APIs that are unique and not shared with other components in the tenant?**: No
  + **Which type of client-side component to create?**: WebPart
  + **What is your Web part name?**: ReactDemo
  + **What is your Web part description?**: React description
  + **Which framework would you like to use?**: React

After provisioning the folders required for the project, the generator will install all the dependency packages using NPM.

1. When NPM completes downloading all dependencies, run the project by executing the following command:

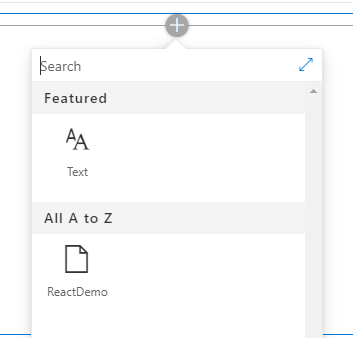
gulp serve

1. The SharePoint Framework's gulp **serve** task will build the project, start a local web server, and launch a browser open to the SharePoint Workbench:

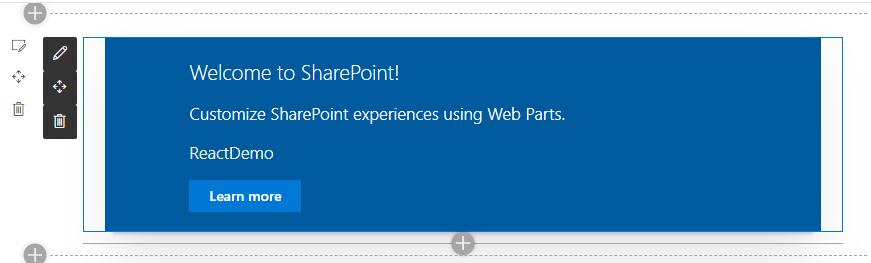
A screenshot of a cell phone

Description automatically generated

1. Select the web part icon button to open the list of available web parts:



1. Select the **React Demo** web part:



1. Close the browser and stop the local web server by pressing CTRL+C in the command prompt.

**Examine the Web Part code**

Before modifying the web part code, take a minute to see how this project differs from a SPFx project that does not utilize React.

1. Open the **package.json** file in the root of the project

Notice the presence of a few extra packages in the dependencies section. The react & react-dom packages contain the core React web framework and the bridge between React amd the DOM, while @types/react & @types/react-dom contain the TypeScript type declarations for the libraries.

1. Open the web part in the   
   **./src/webparts/react/ReactWebPart.ts** file.

The main difference between this web part and one that does not leverage the React Framework is in the render() method. Instead of writing HTML out, it first creates a new React component and then uses the ReactDom.render() method to render the component into the div for the web part.

The React component that is created is **React**.

1. Open the React component:  
    **./src/webparts/React/components/React.tsx**.

This file contains a class declaration that extents the base React.Component abstract class. It contains a single method, render(), that serves the same purpose as the render() method in the web part: it renders the control out.

Because we are using the TypeScript Extended (\***.tsx**) language and syntax, well-formed HTML can be directly returned to the caller. The caller is the React web framework.

This component has a single public property: description. This is defined as an interface in the **./src/webparts/React/components/IReactProps.ts** file.

Now that you understand how a React project is structured, add some data and a child component to the web part.

**Update the web part code**

Update the web part to show a list of colors using a child React component.

1. Create a new file in the **./src/webparts/ReactDemo** folder named **IColor.ts**.

Add the following code to the file. This will act as an interface to our new object type:

export interface IColor {

id: number;

title: string;

}

1. Create a new React component that will show a list of colors provided to it as a public property:
   1. Create a new file **ColorList.tsx** in the **./src/webparts/ReactDemo/components**.
   2. Add the following code to the **ColorList.tsx** file. This creates an public interface for the component's public signature and creates the core React component:

import \* as React from 'react';

import { IColor } from './IColor';

export interface IColorListProps {

colors: IColor[];

}

export class ColorList extends React.Component<IColorListProps, {}> {

public render(): React.ReactElement<IColorListProps> {

return (

<ul>

{

this.props.colors.map(colorItem => (

<li>{ colorItem.title }</li>

))

}

</ul>

);

}

}

1. With a React component created that will display a list of colors, the next step is to use it. Do this by updating the default React component created by the SPFx Yeoman generator:
   1. Open the **./src/webparts/ReactDemo/components/ReactDemo.tsx** file.
   2. Add the following import statements after the existing import statements. These will add references to the new files and objects you previously created:

import { IColor } from "../IColor";

import { ColorList, IColorListProps } from "./ColorList";

* 1. Add a new private member to the React class that contains a static collection of colors:

private \_colors: IColor[] = [

{ id: 1, title: 'red' },

{ id: 2, title: 'blue' },

{ id: 3, title: 'green' }

];

* 1. Next, update the React's render() method to use the new React component you previously created. The only important line here is the <ColorList> component reference.

Notice the colors public property on the component is bound to the private array of colors you created above:

public render(): React.ReactElement<IReactProps> {

return (

<div className={ styles.React }>

<div className={ styles.container }>

<div className={ styles.row }>

<div className={ styles.column }>

<span className={ styles.title }>Welcome to SharePoint + React!</span>

<ColorList colors={this.\_colors} />

</div>

</div>

</div>

</div>

);

}

1. Test the project:
   1. Start the local web server using the provided gulp **serve** task:

gulp serve

* 1. The SharePoint Framework's gulp **serve** task will build the project, start a local web server and launch a browser open to the local SharePoint Workbench.
  2. Add the web part to the workbench. Notice our list of three colors is rendered up exactly as we would expect.

A screenshot of a cell phone

Description automatically generated

* 1. Close the browser and stop the local web server by pressing CTRL+C in the command prompt.

**Part 2: Leveraging Fabric React**

In this exercise, you will update the existing React-based SPFx web part to leverage a few controls from the [Fabric React](https://developer.microsoft.com/fabric) controls.

1. Update the existing **ColorList** React component to leverage Fabric React controls.
2. Open the **./src/webparts/ReactDemo/components/ColorList.tsx** file.
3. Add the following import statements to the top of the file. These will enable adding a DefaultButton and List control to the component:

import { List } from 'office-ui-fabric-react/lib/List';

import { DefaultButton } from 'office-ui-fabric-react/lib/Button';

1. Update the render() method within the ColorList class to the following code:

public render(): React.ReactElement<IColorListProps> {

return (

<div>

<List items={ this.props.colors }

onRenderCell={ this.\_onRenderListCell }

/>

</div>

);

}

1. Handle the rendering of each item in the list by adding the following method to the ReactDemo class:

private \_onRenderListCell = (color: IColor, index: number | undefined): JSX.Element => {

return (

<div>

{ color.title }<br />

<DefaultButton text="delete"

data={ color.id }

onClick={ () => this.\_onButtonClick(color) }

/>

</div>

);

}

1. Next, add an event handler for when the button is selected:

private \_onButtonClick(color:IColor): void {

console.log('clicked delete for color', color);

}

1. Test the project:
   1. Start the local web server using the provided gulp **serve** task:

gulp serve

* 1. The SharePoint Framework's gulp **serve** task will build the project, start a local web server and launch a browser open to the local SharePoint Workbench.
  2. Add the web part to the workbench. Notice our list of three colors is rendered up exactly as we would expect.

A screenshot of a cell phone

Description automatically generated

* 1. Select the **delete** button for one of the colors & examine the browser's JavaScript console, usually located in the browser's developer tools. You should see a log message displayed each time a button is selected:

A screenshot of a cell phone

Description automatically generated

* 1. Close the browser and stop the local web server by pressing CTRL+C in the command prompt.

**Part 3: Dynamic React Components**

In this exercise, you will update the existing React-based SPFx web part to make it more dynamic by introducing state and data will eventually come from a SharePoint list.

1. Add a new interface to represent the component's state.
2. Create a new file **IReactDemoState.ts** in the folder **./src/webparts/ReactDemo/components**.
3. Add the following code to the file:

import { IColor } from '../IColor';

export interface IReactState {

colors: IColor[];

}

1. Update the **ReactDemo** react component with new state:
   1. Locate and open the file **./src/webparts/ReactDemo/components/ReactDemo.tsx**.
   2. Add the following import statements to the top of the file:

import { IReactDemoState } from './IReactDemoState';

* 1. Update the ReactDemo component to define its state interface.
     1. Locate the class definition for the **ReactDemo**. Notice the end of the line contains a {} as the second parameter for the ReactDemo.Component base class:

export default class React extends ReactDemo.Component<IReactDemoProps, {}> {

* + 1. This second parameter is how you define the interface for the state of the component. Update this to use the new interface you previously created: IReactState.

export default class React extends React.Component<IReactDemoProps, IReactDemoState> {

* 1. Initialize the state of the component by defining a constructor that sets the default component state:

constructor(props: IReactDemoProps) {

super(props);

this.state = { colors: [] };

}

* 1. Update the React component's lifecycle by retrieving data and setting the component's state when the component is loaded on the page. Setting the state will trigger React to re-render the component, displaying the retrieved data.
     1. Add the following method to the **ReactDemo** class:

public componentDidMount(): void {

this.getSpColors()

.then((spColors: IColor[]) => {

this.setState({ colors: spColors });

});

}

* 1. Add the following method to the **ReactDemo** class. This will retrieve an array of items from the previously created SharePoint list using the SharePoint REST API:

private getSpColors(): Promise<IColor[]> {

return new Promise<IColor[]>((resolve, reject) => {

resolve(this.\_colors);

}

}

* 1. Update the render() method in the **ReactDemo** class to do two things: first, change the binding to instead of using the static collection of colors, use the colors from the new state object that is updated by the lifecycle event componentDidMount() and then, attach to an event on the custom component created previously.

1. Update the **ReactDemo** React component to handle remove callback.
2. Locate and open the file **./src/webparts/ReactDemo/components/ReactDemo.tsx**.
3. Add the following type to the file, after the existing import statements. This will define a new callback type that will represent a new event exposed by this component:

export type RemoveColorCallback = (color: IColor) => void;

1. Add a new event as a public property to the component by updating the existing **IColorListProps** interface. Add the following code to the interface:

onRemoveColor: RemoveColorCallback;

1. Locate the existing \_onButtonClick method. Currently this method just writes to the browser's JavaScript console. Change the contents to raise the new public event, passing in the color that the button represented:

private \_onButtonClick(color:IColor): void {

this.props.onRemoveColor(color);

}

1. When this event occurs (something you will implement later in this exercise), it will run our handler. Do this by updating the <ColorList> control reference in the render() method:

<ColorList colors={ this.state.colors } onRemoveColor={ this.\_removeColor }/>

1. Add the following method to the **ReactDemoWebpart** class to implement the event handler. This will create a new collection of colors that include all the original colors, except the one included in the onRemoveColor callback:

private \_removeColor = (colorToRemove: IColor): void => {

const newColors = this.state.colors.filter(color => color != colorToRemove);

this.setState({ colors: newColors });

}

1. Test the project:
   1. Start the local web server using the provided gulp **serve** task:

gulp serve --nobrowser

* 1. Add the web part to the hosted workbench. Notice our list of three colors is rendered up exactly as we would expect.

A screenshot of a cell phone

Description automatically generated

* 1. Select the **delete** button for one of the colors. Notice the color is removed from the set of colors displayed in the web part.

That is because selecting the button raised an event on the **ReactDemo** component. The **ReactDemo** component handles this event by removing the color from the existing collection of colors in the current state and then sets this new collection on the state.

That action triggers React to re-render the component which results in a list of colors missing the one that was removed.

* 1. Close the browser and stop the local web server by pressing CTRL+C in the command prompt.