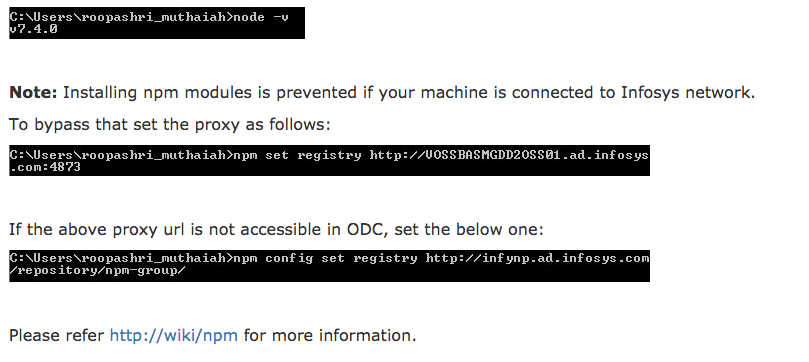
**System Configuration:**

To develop a React application on a local machine, we need to install the below packages:

1. Install node.js from <https://nodejs.org/en/> or from <http://sparshv2/portals/CCD/Downloads/Pages/Downloads.aspx>

To check whether node is installed or not in your machine, go to **node command prompt** and check the node version by typing the following



2. Enabling ES6 and JSX: React with Babel to let you use ES6 and JSX in JavaScript code.

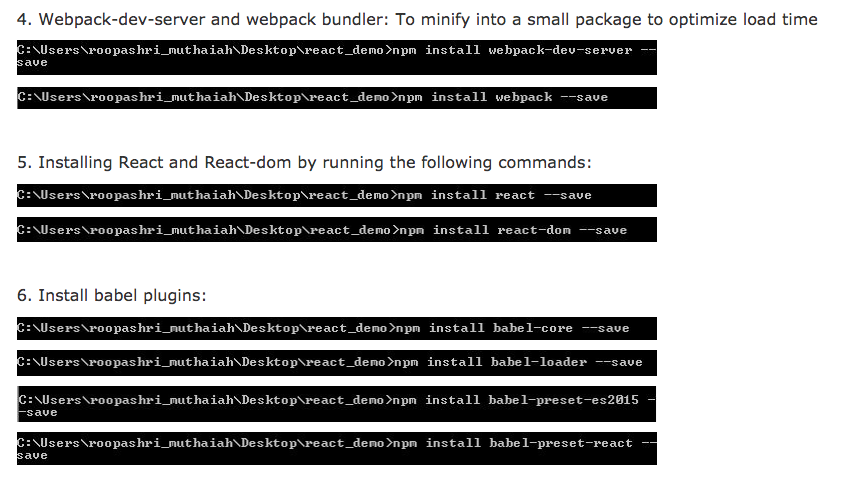


3. Create a folder to store your application and navigate to it through node.js command prompt. Configure package.json file by running the command npm init.  
Ignore the details asked to enter for the properties by pressing enter key.



Update the scripts property as follows by changing the test command:

"scripts": { "start": "webpack-dev-server --hot" }



Configure the webpack.config.js file as follows which tells how the execution should start.

a. Webpack entry point to be main.js

b. Output path is where the bundled app will be served

c. Bundled file is index.js

d. Set development server to 7777

var config = {

entry: './main.js',

output: {

path:'/',

filename: 'index.js',

},

devServer: {

inline: true,

port: 7777

},

module: {

loaders: [ {

test: /\.jsx?$/,

exclude: /node\_modules/,

loader: 'babel-loader',

query: {

presets: ['es2015', 'react']

}

}]

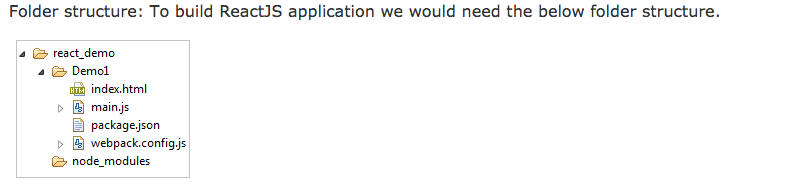
}

}

module.exports = config;

JavaScript

Copy



Create index.html page with the below code:

<!DOCTYPE html>

<html lang = "en">

<head>

<meta charset = "UTF-8">

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"></div>

<script src = "index.js"> </script>

</body>

</html>

HTML

Copy

Create main.js page with the below code:

import React from 'react';

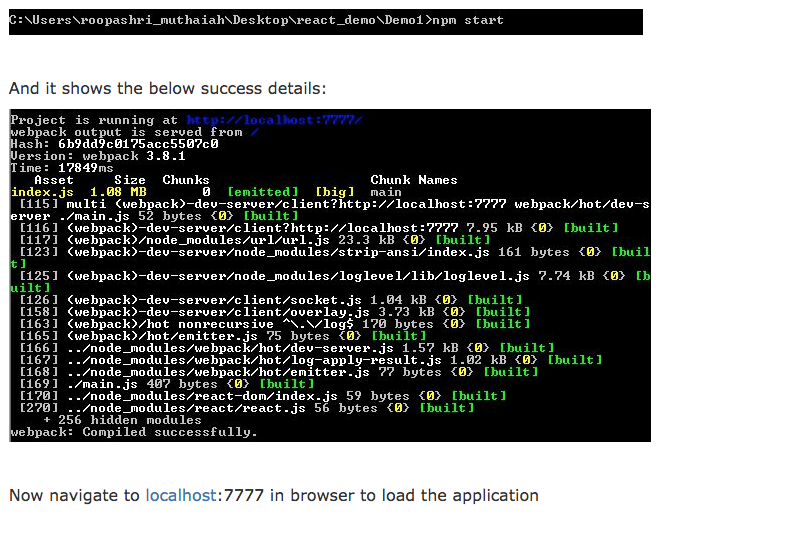
import ReactDOM from 'react-dom';

ReactDOM.render(<h1> Hello World </h1>, document.getElementById('app'));

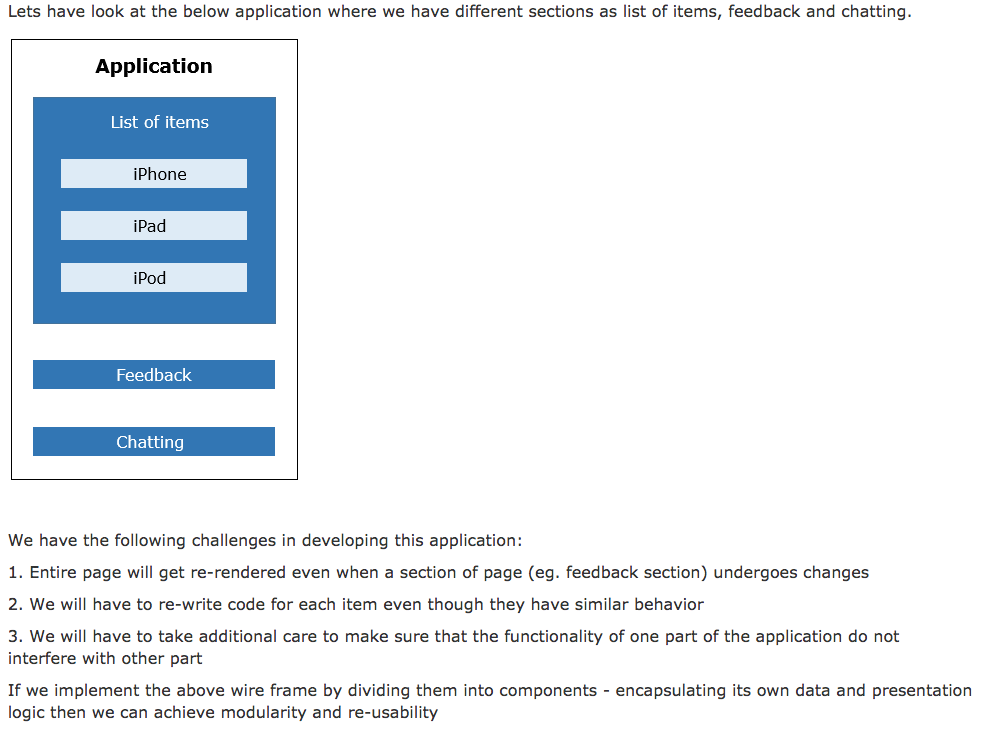
JavaScript

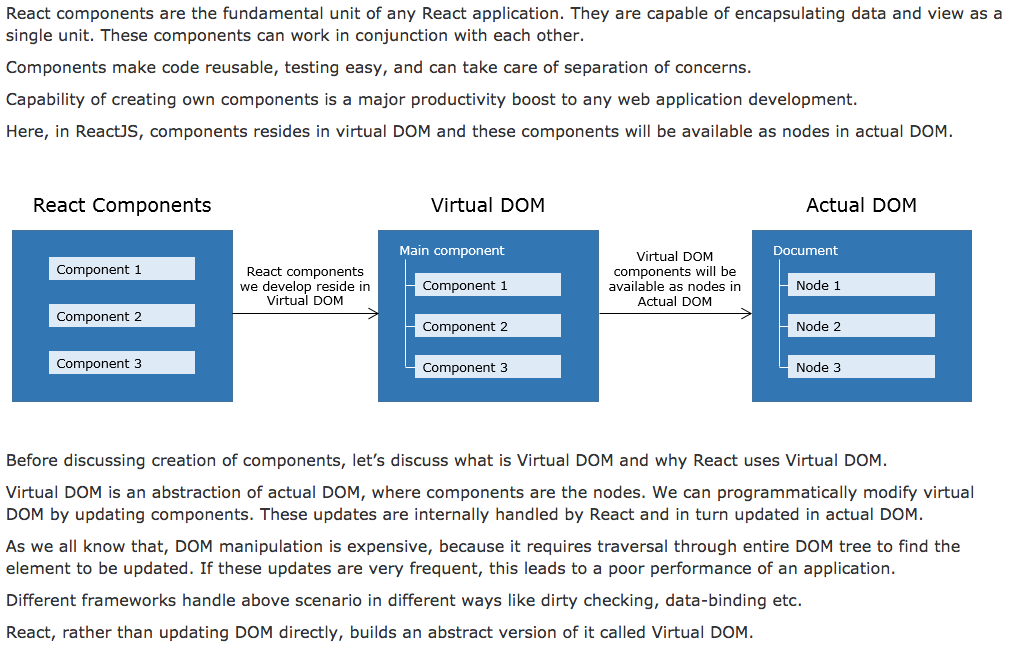
Copy

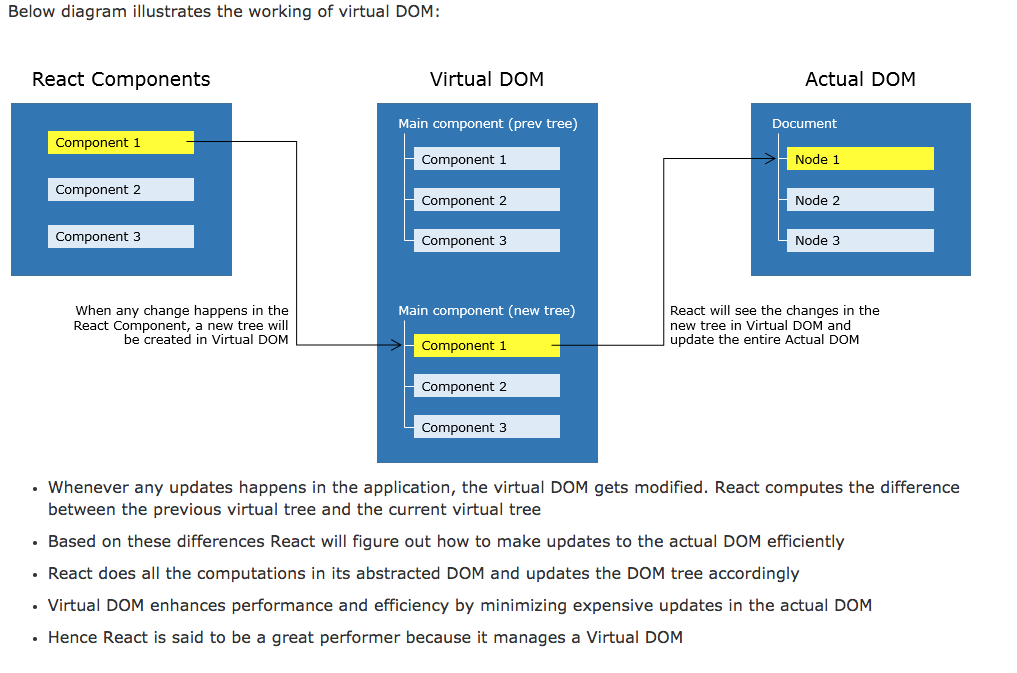
To run the application navigate to this folder and start the server by giving the following command:



Why Compononets?







Components can be created in 2 different ways:

* Using createReactClass() method
* By extending React component class
* We can create React component using **createReactClass** method which accepts an object as argument.
* Install create-react-class package by running the following command:
* npm install create-react-class --save
* Syntax:
* var AppComp = createReactClass({ });
* JavaScript
* Copy
* If we want to create an element within the component we can use **React.createElement** method.
* Example:
* React.createElement("h1", {}, "Hello World!!!");
* JavaScript
* Copy
* In the above code, we are creating h1 element with text Hello World!!! and {} is used to mention attributes of the element.
* In order to render the element within the component we need **render** method.
* Let's see how to create a component having one element using above concept.
* App.js:
* import React from 'react';
* import createReactClass from 'create-react-class';
* var AppComp = createReactClass({
* render: function() {
* return React.createElement("h1", {}, "Hello World!!!");
* }
* });
* export default AppComp;
* JavaScript
* Copy
* The AppComp component created in App.js file is exported so that it can be used in other files.

Creating Method 2:

Creating component using createReactClass method is as per the earlier version. The latest way of creating component is by extending React.Component class which is as per the ES6 specification.  
Let's see how to create AppComp component.

App.js:

import React from 'react';

class AppComp extends React.Component {

render() {

return React.createElement("h1", {}, "Hello World!!!");

}

}

export default AppComp;

JavaScript

Copy

* AppComp - component name should be in Pascal Casing
* render() method will render the component’s elements
* React.createElement - helps to create an element in plain JavaScript
* export default AppComp - AppComp component has to be exported so that it could be used in any other files

rendering Components:

To display elements of a component, component has to be rendered.

For rendering a component, ReactDOM.render method is used as follows:

Syntax:

ReactDOM.render(<parameter 1/>, parameter 2);

JavaScript

Copy

* ReactDOM.render method will take 2 parameters:
  + **'parameter 1'** is the name of the component to be rendered
  + **'parameter 2'** is the HTML node reference where the component to be rendered

So our final code would be as follows:

main.js:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from ‘/App.js;

ReactDOM.render(<AppComp/>, document.getElementById('app'));

JavaScript

Copy

'app' is the id referring to a node of index.html page.

index.html:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

DEMO1: CREATING COMPONENTS:

Highlights:

 Creation of component

 Rendering a component

Demo Steps:

1. Create AppComp in App.js and create an element to render "Hello World" in heading format as shown below:

import React from 'react';

class AppComp extends React.Component {

render(){

return React.createElement("h1", {}, "Hello World");

}

}

export default AppComp;

JavaScript

Copy

2. In main.js, render AppComp using ReactDOM.render method as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.js';

ReactDOM.render(<AppComp />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

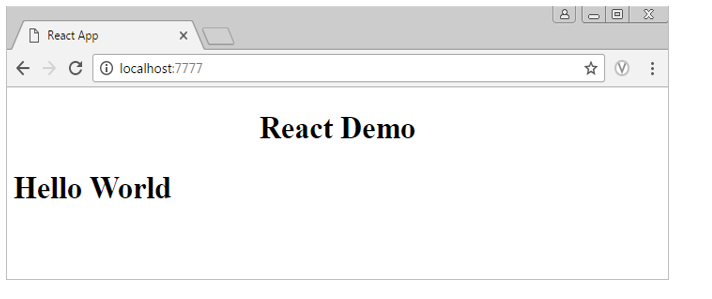
</body>

</html>

HTML

Copy

4.  Observe the below output:



WHY JSX?

Lets see the implementation of a login component:

Code using pure JavaScript:

class AppComp extends React.Component{

render(){

return (

React.createElement('form', {},

React.createElement("h1", {}, "Login"),

React.createElement('input', {type: 'text',placeholder:'Name', value: '',}),

React.createElement('br', {}),React.createElement('br', {}),

React.createElement('input', {type: 'password', placeholder: 'password',

value: '',}),

React.createElement('br', {}), React.createElement('br', {}),

React.createElement('button', {type: 'submit'}, "Login"))

)

}

};

export default AppComp;

JavaScript

Copy

We can observe from the above code, that we need to write more lines of JavaScript code to implement this simple component. Code looks difficult to understand and hence productivity goes down.

JSX has been introduced in React to create elements which is very easy to read and write, which makes component's code simple and understandable.

Code using JSX:

class AppComp extends React.Component{

render(){

return (<form><h2>Login</h2>

<input type="text" placeholder="Name" /><br/><br/>

<input type="password" placeholder="password" /> <br/><br/>

<input type="submit" nvalue="log" />

</form>);

}

};

export default AppComp;

React JSX

Copy

This technique is a replacement of writing pure JavaScript code and hence enhances the productivity.

WHAT IS JSX?

JSX is a special syntax introduced in ReactJS to write elements of components. It is syntactically identical to HTML and hence it can be easily read and written. Code written using JSX helps in visualizing DOM structure easily.

We can modify the demo-1 using JSX as follows:

class AppComp extends React.Component {

render() {

return <h1> Hello World </h1>

}

}

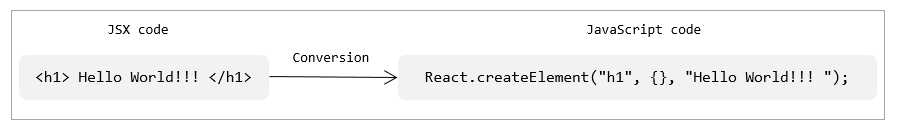
export default AppComp;

React JSX

Copy

As browser does not understand JSX code, this has to be converted to JavaScript and this conversion is done using browser.min.js library.

Conversion of JSX to JavaScript happens as shown below:



DEMO 2: JSX IN COMPONENTS:

Highlights:

 Creating elements in JSX

 How JSX code will be converted to JavaScript

Demo Steps:

1. Create a file App.jsx and write the following code using JSX:

import React from 'react';

class AppComp extends React.Component {

render(){

return <h1> Hello World </h1>

}

}

export default AppComp;

React JSX

Copy

2. Create main.js file as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html file as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

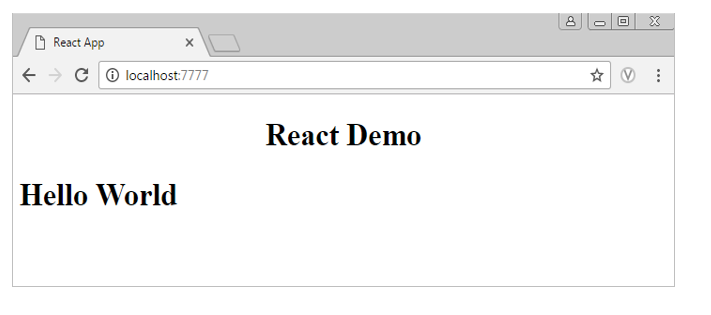
</body>

</html>

HTML

Copy

4. Observe the below output:



DEMO 3: JSX IN COMPONETS:

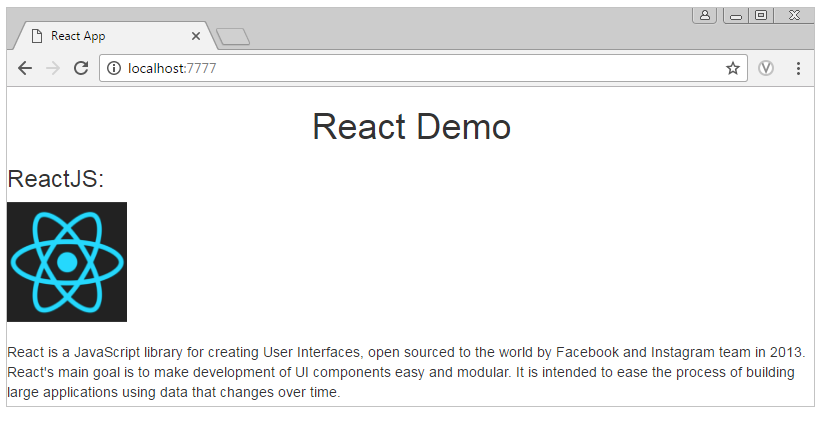
Highlights:

 How to render multiple elements

 Nesting JSX elements

Demo Steps:

Create a component to render an image with its description as follows:



1. Create App.jsx as shown below:

import React from 'react';

class AppComp extends React.Component {

render(){

return(<div>

<h3>ReactJS:</h3>

<img src="./image/react.PNG" width="120" height="120"/>

<p> React is a JavaScript library for creating User Interfaces,

open sourced to the world by Facebook and Instagram team in 2013.<br/>

React’s main goal is to make development of UI components easy and modular.

It is intended to ease the process of building<br/> large applications using

data that changes over time.</p>

</div>);

}

}

export default AppComp;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

HTML

Copy

4. Download the image react.PNG from the link - [react.PNG](http://ilpce/ce/Course/UploadedCourseFiles/REACT/REACT_4.zip)

JAVASCRIPT EXPRESSIONS IN JSX:

We discussed how to use JSX to create React elements, we may also require to use JavaScript expressions in React elements, so let's see how to write JavaScript expressions in JSX.

JavaScript expressions to be evaluated should be wrapped within curly braces as follows:

<h1> { Expression to be evaluated } </h1>

React JSX

Copy

Content which has to be displayed as is will be written in double quotes, wrapped within curly braces as follows:

<h1> {" Content to be displayed "} </h1>

DEMO 4:JSX IN COMPONENETS:

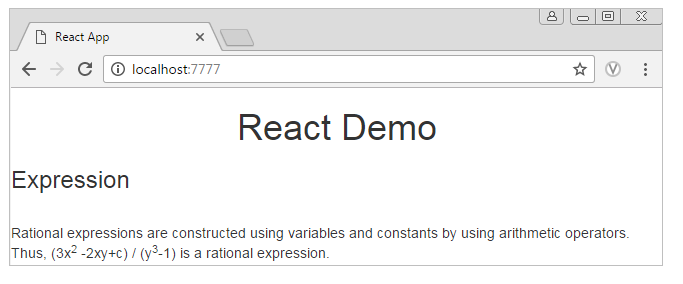
Highlights:

 Displaying content

 Writing an expression

Demo Steps:

Create a component to display an algebraic expression as follows:



1. Create App.jsx as shown below:

import React from 'react';

class AppComp extends React.Component {

render() {

return (<div>

<h3> Expression </h3><br/>

<p> Rational expressions are constructed using variables

and constants by using arithmetic

operators. Thus,

{"(3x"}<sup>2</sup> {"-2xy+c) / (y"}

<sup>3</sup>{"-1)"} is a rational

expression.

</p>

</div>)

}

}

export default AppComp;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

DEMO5: JSX IN COMPONENTS:

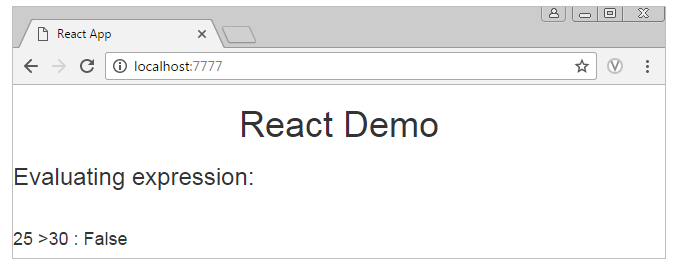
Highlights:

 Passing values to expression

 Evaluating the expression

Demo Steps:

Create a component to display an algebraic expression as follows:



1. Create App.jsx as shown below:

import React from 'react';

class AppComp extends React.Component {

render() {

var x=25, y=30

return (<div>

<h3> Evaluating expression </h3><br/>

<h4> {x} {">"}{y} {":"} {x>y ? 'True' : 'False'} </h4>

</div>)

}

}

export default AppComp;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

DEMO6 : CASE STUDY IMPLEMENTATION:

Highlights:

 Creating components required for the application

 Implementing star rating functionality

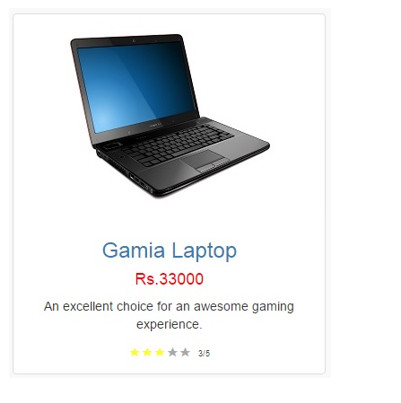
Demo Steps:

Below is the list of components to be created to build this application:

1. App: Acts as the container of all other components
2. LandingPage: Component which displays landingpage image and login link
3. Login: Form elements for login functionality
4. LoginForm: Contains the Login component and associated styles
5. Navigation: All the top navigation bar functionalities are listed in this
6. Rater: 5 Star rating functionality
7. Product: Includes product details and 5 star rating  component of a single product
8. FeedbackComp: Implements Feedback submission form, list of all feedbacks and 5 star rating functionalities
9. MessageComposer: Implements the message entering funcationality and associated methods
10. MessagelistItem: Component to display the sent and received messages with the usernames
11. MessageSection: Container of MessageComposer and MessagelistItem components
12. ThreadListItem: Display the details of individual conversation thread with username name and part of the last sent message with time
13. ThreadSection: Visible only for Admin. Container of all the active conversation ThreadListItem(s)
14. ChatApp: this is the container of chat components viz MessageSection and ThreadSection
15. PurchasedItems:  Visible only for customer role. Contains Product(s) and MessageSection Components
16. ProductDetails: Visible only for customer role. Contains FeedbackComp, Product and MessageSection Components
17. ProductDisc: Visible only for admin role. Implements product details and discontinue product functionalities
18. Dashboard: Visible only for admin role. Acts as the container of ProductDisc, ThreadSection and MessageSection Components

Out of the listed components let us create 2 components Product and Rater component. Screenshot and the code is as follows:

**Create Product component:**



import React from 'react';

import Rater from './Rater.jsx';

class Product extends React.Component{

render(){

return(<div className={"thumbnail"}>

<img src = ”Images/Laptop.jpg” class="img img-rounded img-responsive"/>

<div class = "caption" style="text-align:center">

<a href = ”/productDetails”}> <h3> Gamia Laptop </h3></a>

<h4><span style = "color:red">Rs. 33000</span></h4>

<p>An excellent choice for an awesome gaming experience.</p>}

<Rater />

<span style = "fontSize:9px">3/5</span>

</div>

</div>

);

}

}

export default Product;

React JSX

Copy

**Create Rater component:**



var React=require('react');

class Rater extends React.Component{

render(){

return(<ul className="rating">

<li class = ”filled”>{'\u2605'}</li>

<li class = ”filled”>{'\u2605'}</li>

<li class = ”filled”>{'\u2605'}</li>

<li> {'\u2605'} </li>

<li> {'\u2605'} </li>

</ul>)

}

}

export default Rater;

React JSX

Copy

Note: The star in the Rater component will be in yellow color if the class “filled” in added.

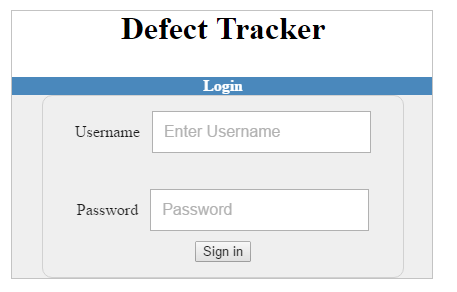
EXCERISE 1 :

**Time Limit:**40 Minutes

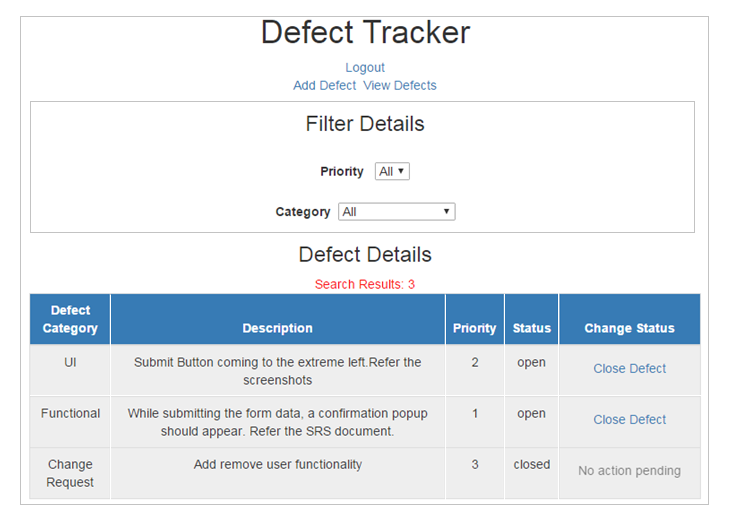
Problem Statement

We will be systematically developing an application to add, view and filter the defects in an application which is identified by the tester. Here we have two roles - tester and developer.  
Below are the screenshot of entire application with its functionality:

Login view: Users should be able to enter the credentials and submit the form

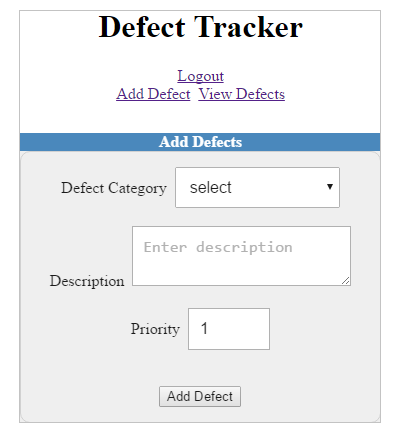


View Defects: In View Defects functionality, the user will be able to view the defects with corresponding Description, Category, Status, Priority and Status changing button.

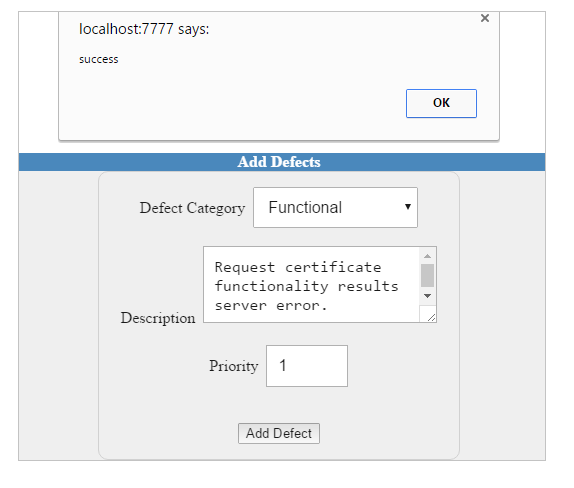


And develop other functionalities too by dividing it into different components.

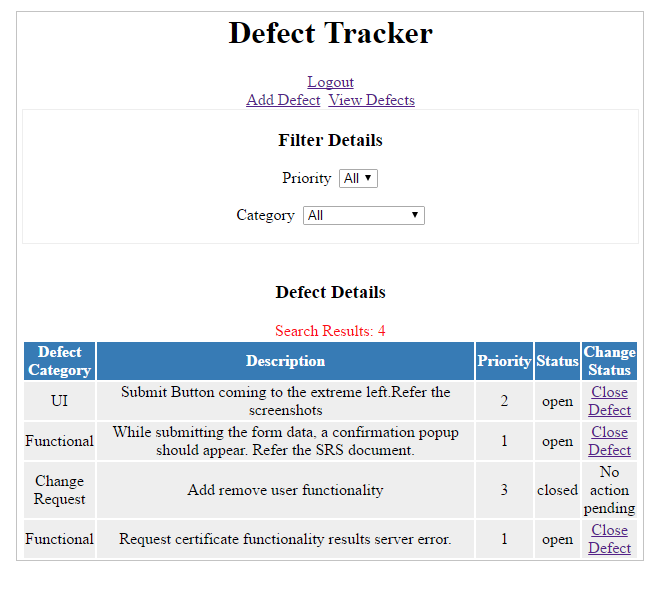
Add defects (available only for Tester role): Testers will be logging different defects after testing any application developed by the team. Tester can enter the below details and add the defects to existing list.



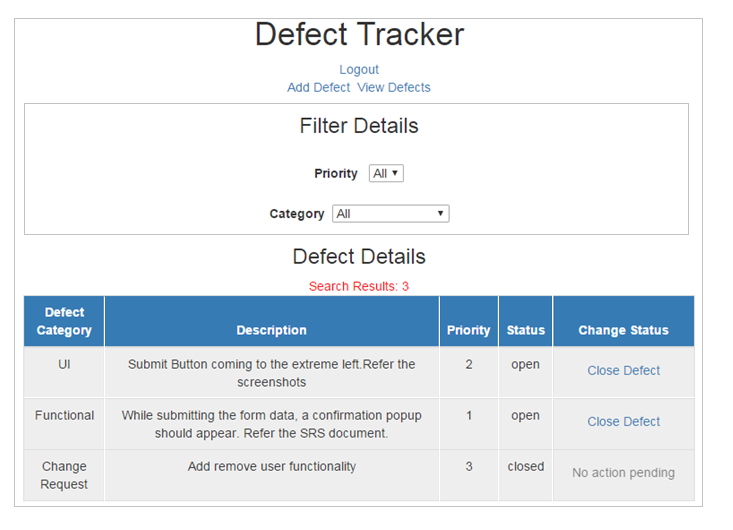
Success message should be displayed on submitting the form with the details as shown below:



And the updated details will be reflected in View defects as follows:



Now create a component as per the below screenshot for the View Defects page and hardcode the value wherever required:



TOPC : HANDLIND DATA USING PROP:

WHY PROP AND STATE:

Let's have a look at the below Product component. We can notice that the product details are hardcoded.

**Product component:**

class Product extends React.Component{

render(){

return(<div>

<div className={"thumbnail"}>

<img src = ”Images/Laptop.jpg” class="img img-rounded img-responsive"/>

<div class = "caption" style="text-align:center">

**/\* Hardcoded value for a product \*/**

**<a href = ”/productDetails”}> <h3> Gamia Laptop </h3></a>**

**<h4><span style = "color:red">Rs. 33000</span></h4>**

<p>An excellent choice for an awesome gaming experience.</p>}

<Rater />

<span style = "fontSize:9px">3/5</span>

</div>

</div>

</div>);

}

}

React JSX

Copy

In real scenario we will be retrieving data from the server and will be passed as an array to components.

In order to bind the retrieved data to the component, we need two JS objects i.e. props and state.

In this module, lets understand the concept of props and state to handle the data in our application.

# What is State?

State is an initial value set for a component, which is used for interactivity.

Let's see how to set state of a component.

We use constructor of the component for initializing state of a component, where parent component's constructor needs to be invoked using super(props), to pass properties. State of the component is set using **this.state**as shown below.

Syntax:

constructor(props) {

super(props);

this.state = { counter: 1 };

}

React JSX

Copy

And state 'counter' would be accessed as **this.state.counter**

UPDATING STATE:

Now let's create a Timer component where on clicking a button, timer starts.

Below is the implementation to start the timer:

When a button is clicked, by invoking handleClick() method - set the interval and pass it to start() method:

handleClick(e) {

this.interval = setInterval(this.start, 1000);

}

React JSX

Copy

In start() method, for every second, state will be updated using setState() method.

start() {

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

}

React JSX

Copy

All the methods in our component, should be bound to constructor as follows:

class Timer extends React.Component{

constructor(props){

super(props);

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

this.start = this.start.bind(this);

this.state = {

secondsElapsed: 0

};

}

}

DEMO7: UPADTING STATE:

Highlights:

 Handling data in a component

 Making component interactive

 Usage of state

Demo Steps:

Create a component to start the timer:

1. Create App.jsx as shown below:

import React from 'react';

class Timer extends React.Component{

constructor(props){

super(props);

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

this.start = this.start.bind(this);

this.state = { secondsElapsed: 0 };

}

start(){

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

}

handleClick(e){

this.interval = setInterval(this.start, 1000);

}

render(){

return (<div>

<button onClick = {this.handleClick}> Start timer </button>

<h2> Seconds Elapsed: {this.state.secondsElapsed} </h2>

</div>);

}

}

export default Timer;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Timer from './App.jsx';

ReactDOM.render(<Timer />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

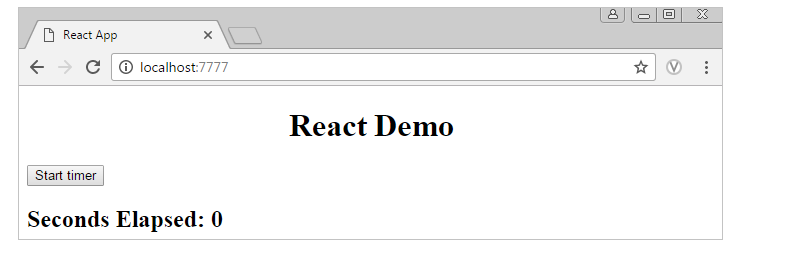
</body>

</html>

HTML

Copy

4. Observe the below output:



STATE JEY NOTES:

Key points to be remembered while handling data in a component:

* States are mutable and only used on top level component
* They are reserved only for interactivity and component's event handlers may change to trigger a UI update
* State will be set with a default value when component mounts and will mutate in time based on user events generated

PROPS:

Props allows you to pass data from parent component to the child component.

Props are immutable - a component cannot change its props however it is responsible for putting together.

Props can be accessed as **this.props**.

**How to use props?**

Let's now learn how to pass data to a component:

    <AppComp initial = {10} />

Here, we are passing a property 'initial' with a value '10', to a component AppComp.

In the component AppComp, this property would be accessed as **this.props.initial**.

DEMO8: PROPS:

Highlights:

 Passing data

 Accessing data

Demo Steps:

Create 2 components as mentioned below:

1. Timer - It holds state and handles button click

2. Resultant - It takes result from Timer component and displays it

1. Create App.jsx as shown below:

import React from 'react';

class Timer extends React.Component{

constructor(props){

super(props);

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

this.start = this.start.bind(this);

this.state = { secondsElapsed: 0 };

}

start(){

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

}

handleClick(e){

this.interval = setInterval(this.start, 1000);

}

render(){

return (<div>

<button onClick = {this.handleClick}> Start Timer </button>

<Resultant new = {this.state.secondsElapsed} />

</div>);

}

}

class Resultant extends React.Component{

render(){

return (<div>

<h3> Seconds Elapsed: {this.props.new} </h3>

</div>);

}

}

export default Timer;

React JSX

Copy

As we want to display result from Resultant component, we are passing state (which holds result) to Resultant component.

* When data is passed from parent component to child component, it becomes property of child component
* These properties are immutable and are accessible as this.props.new in Resultant component

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Timer from './App.jsx';

ReactDOM.render(<Timer />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

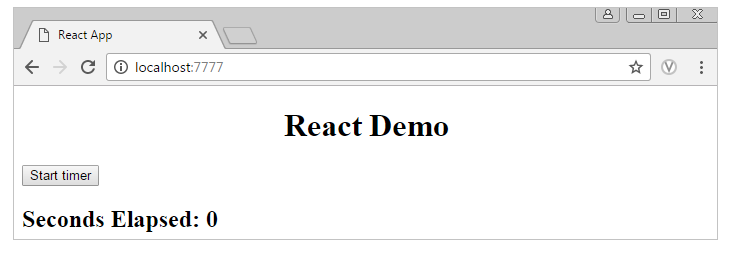
</body>

</html>

HTML

Copy

4. Observe the below output:



COMPONENT TYPES:

We can categorize components as below:

1. Stateless Component:

* It contains only props
* It is used to render data of a component

2. Stateful Component :

* It has both props and states
* It encapsulates interaction logic as well as renders data

In our demo, Resultant component is stateless and Timer component is stateful.

**Tip:** While developing an app, create several stateless components that renders data, and have a stateful component above them in the hierarchy that passes its state to its children via props.

ACCESSING CHILD NODE:

Let's see how to access child nodes of a component.

Consider an AppComp which has child nodes as shown below:

<AppComp>

<li> List element </li>

<h4> Heading element </h4>

<p> Paragraph element </p>

<span> Span element </span>

<Appcomp />

HTML

Copy

To access these child nodes of a component, React uses **this.props.children.**

React.Children provides utilities to deal with this.props.children.

React.Children.map can be used to iterate this.props.children as follows:

React.Children.map (this.props.children, function fn (arg) { })

React JSX

Copy

Function fn() will be invoked on every immediate child contained within children. If children is null or undefined, fn() returns null or undefined.

DEMO 9: ACCESSING CHILD NODE:

Highlights:

 Creating child nodes to a parent component

 Accessing child nodes from parent

Demo Steps:

Create a component to access child nodes of a component:

1. Create App.jsx as shown below:

import React from 'react';

var count=1;

class AppComp extends React.Component {

render() {

React.Children.map( this.props.children, function() {

console.log ('child', count++); } );

return (<ul>

{ this.props.children }

</ul>)

}

};

export default AppComp;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp>

<li> List element </li>

<h4> Heading element </h4>

<p> Paragraph element </p>

<span> Span element </span>

</AppComp>, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

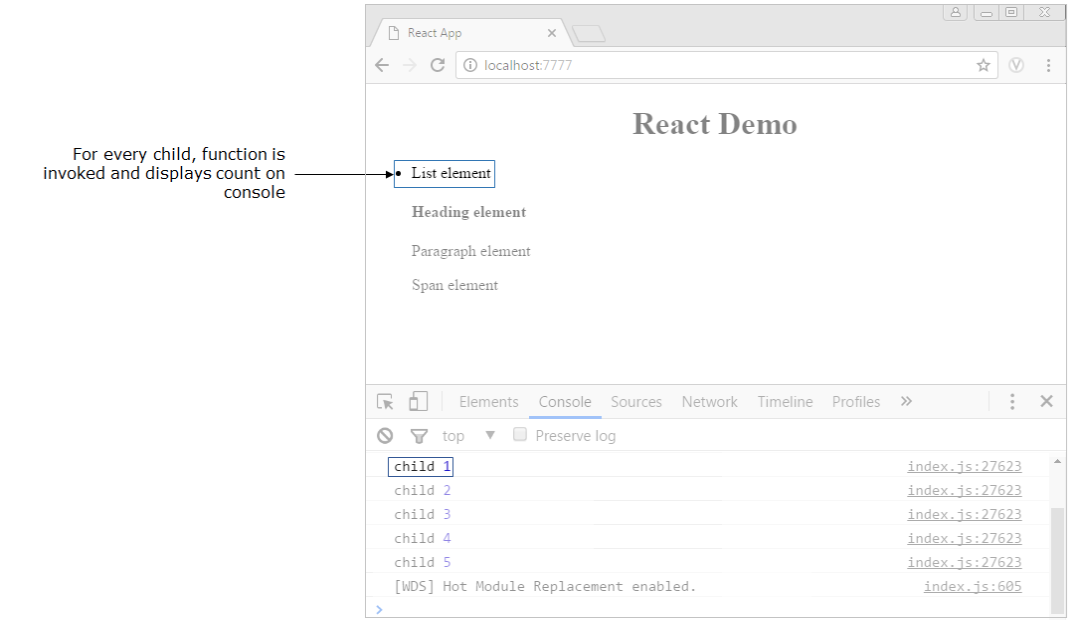
</body>

</html>

HTML

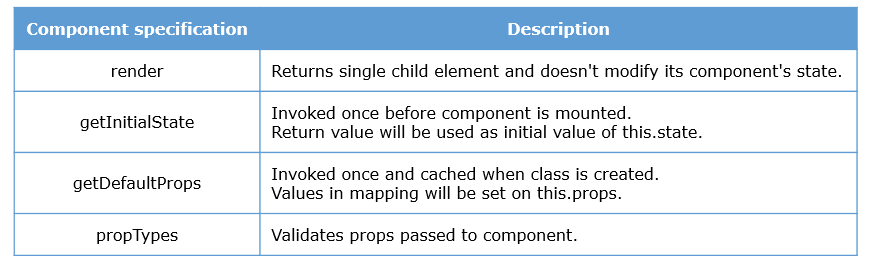
Copy

4. Observe the below output:



Component Specifications:

Till now, while working with component we have used constructor and render method. Now let's see complete listing of component's specification.



Let's understand each specification through demos.

### Component specification : getInitialState

It is used to set state of a component, when the component is created using createReactClass() method.

Syntax:

var Timer = createReactClass({

getInitialState: function() {

return { counter: 1 };

},

render: function() {

return <div> State of a component: {this.state.counter} </div>

}

});

export default Timer;

JavaScript

Copy

When component is created using createReactClass, all the methods would be bound automatically.

However, when a component is created using React.component, constructor is used to set it's state.

### Component specification : getDefaultProps

It sets default values for the properties of a component.

Syntax:

getDefaultProps : function () {

return {

element1 : "Hello",

element2 : "World",

};

}

React JSX

Copy

When the value for these 2 properties are not provided explicitly then the default value will be taken.

In the latest version, getDefaultProps has changed to **defaultProps**.

Syntax:

AppComp.defaultProps = {

element1 : "Hello",

element2: "default value",

}

React JSX

Copy

Here, we need to mention to which component's properties, default value has to be set.

Demo 10 : Component specification

Highlights:

 Setting default value for props

 Using getDefaultProps()

Demo Steps:

Create a component to set default value for props and observe how the default value will be considered:

1. Create App.jsx as shown below:

import React from 'react';

import createReactClass from 'create-react-class';

var AppComp = createReactClass({

getDefaultProps : function () {

return {

element1: "Hello",

element2: "React",

};

},

render: function() {

return(<div>

<h1> {this.props.element1} {this.props.element2}</h1>

</div>)

}

});

export default AppComp;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp element2 = {"World"} />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

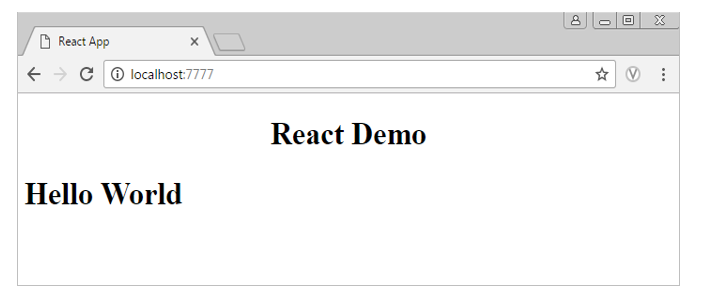
</html>

HTML

Copy

In this demo, default value will be taken for a prop element1, as we are not passing any value explicitly for it, and new value will be taken for element2 prop.

4. Observe the below output:



Demo 11 : Component specification

Highlights:

 Setting default value for props

 Using defaultProps()

Demo Steps:

* Create a component as per the latest version
* Set default values for properties as per the ES6 specification

1. Create App.jsx as shown below::

import React from 'react';

class AppComp extends React.Component {

render() {

return(<div>

<h1> {this.props.element1} </h1>

<h1> {this.props.element2} </h1>

</div>);

}

}

AppComp.defaultProps = {

element1 : "Hello",

element2 : "React",

}

export default AppComp;

React JSX

Copy

2. Create main.js as shown below::

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp element2 = {"World"} />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below::

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

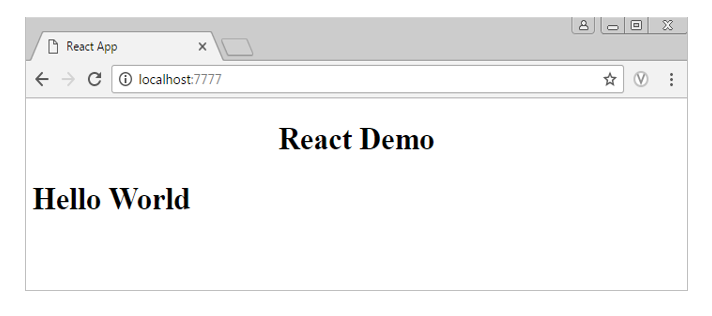
</html>

HTML

Copy

In this demo, default value will be taken for a prop element1, as we are not passing any value explicitly for it, and new value will be taken for element2 prop.

4. Observe the below output:



Component specification : PropTypes

Install PropTypes by running the following command:

npm install prop-types --save

It validate props using different validators. For invalid value, a warning is logged on console.

Consider below code snippet for setting value for properties and validating it using PropTypes:

**Setting different set of default values for props for testing:**

AppComp.defaultProps = {

array: [1,2,3,4,5],

boolean: false,

function: function(e){return e},

number: 23,

string: "React",

emp: {

empName:"Roopashri",

empId: 681592,

unit: "ETA - UIM"

}

}

React JSX

Copy

**Validating these properties using PropTypes:**

AppComp.propTypes = {

array: PropTypes.array,

boolean: PropTypes.bool,

function: PropTypes.func,

number: PropTypes.number,

string: PropTypes.string,

emp: PropTypes.object

}

React JSX

Copy

Demo 12 : Component specification

Highlights:

 Validating props using PropTypes

 Using different validators

Demo Steps:

Create a component with properties defined for it and validate these props using PropTypes

1. Create App.jsx as shown below:

import React from 'react';

import PropTypes from 'prop-types';

class AppComp extends React.Component {

render() {

return (

<div>

<h3>Array: </h3> <p>{this.props.array}</p>

<h3>Boolean: </h3><p>{this.props.boolean ? "True" : "False"}</p>

<h3>Function: </h3><p>{this.props.function(10)}</p>

<h3>Number: </h3><p>{this.props.number}</p>

<h3>String: </h3><p>{this.props.string}</p>

<h3>Object: </h3><p>{this.props.emp.empName} <br/>

{this.props.emp.empId}< br/>

{this.props.emp.unit}</p>

</div>

);

}

}

AppComp.propTypes = {

array: PropTypes.array,

boolean: PropTypes.bool,

function: PropTypes.func,

number: PropTypes.number,

string: PropTypes.string,

emp: PropTypes.object

}

AppComp.defaultProps = {

array: [1,2,3,4,5],

boolean: false,

function: function(e){return e},

number: 23,

string: "React",

emp: {

empName:"Roopashri",

empId: 681592,

unit: "ETA - UIM"

}

}

export default AppComp;

React JSX

Copy

2. Create main.js as shown below::

import React from 'react';

import ReactDOM from 'react-dom';

import AppComp from './App.jsx';

ReactDOM.render(<AppComp />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below::

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

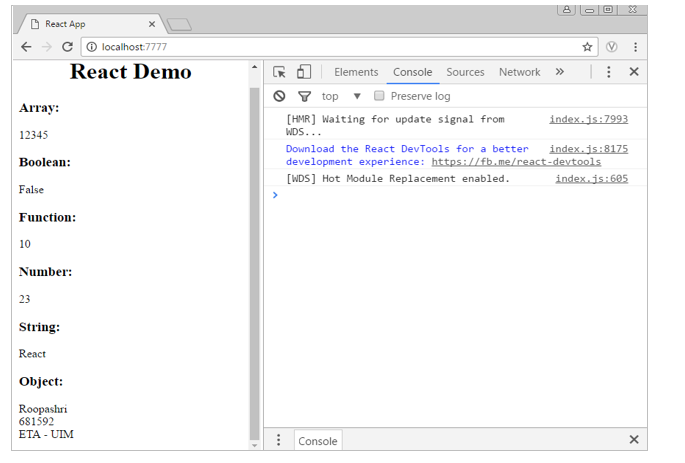
</body>

</html>

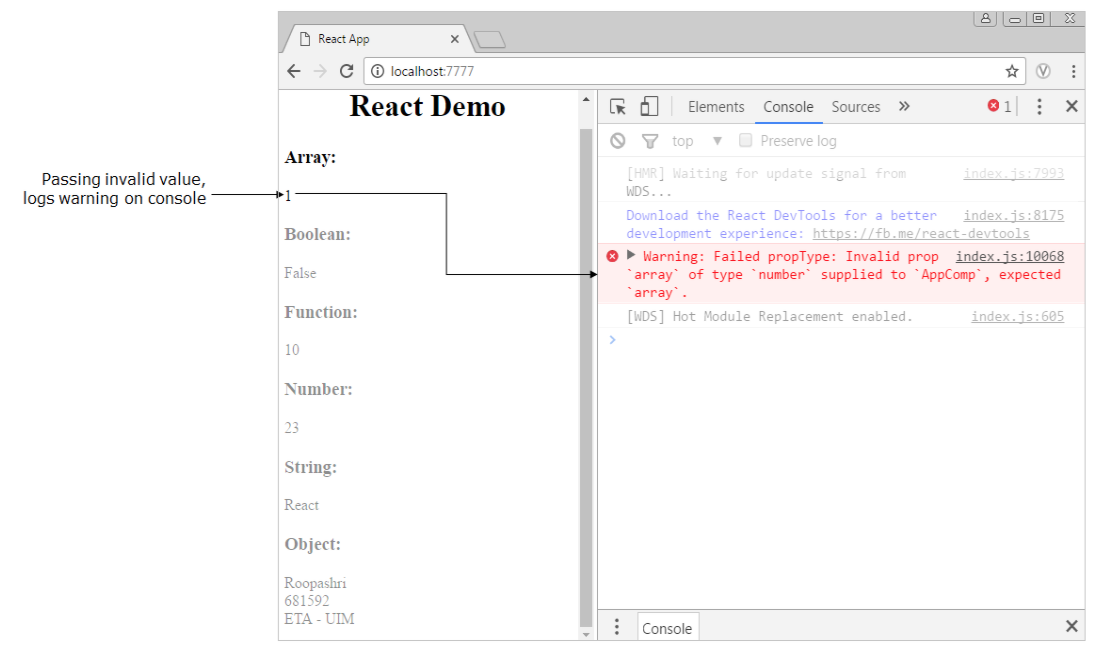
HTML

Copy

4. Observe the below output:



Observe the warning message logged when invalid value passed to array:



Demo 13 : Case study implementation milestone 2

Highlights:

 Passing data to a component

 Updating state

Demo Steps:

Now we shall add data to Product and Rater components using props and state as follows:

Instead of hardcoding the values in the component, lets move the data to a separate JSON array as follows:

var data= [{

\_id:123901239412,

pdtName: “Gamia Laptop”,

pdtPrice: “33000”,

pdtDescription: “An excellent choice for an awesome gaming experience

img: “Images/laptop.jpg”,

rating: “3”,

isDiscontinued:false

}];

JSON

Copy

Let us create a parent component named ProductDetailsand Product component. As we discussed props are used to pass the data from parent to child, here data will be passed to Product component from ProductDetails.

**ProductDetails Component:**

import React from ‘react’;

import Product from ‘./Product.jsx’;

class ProductDetails extends React.Component{

constructor(props){

super(props)

}

render(){

return <div>

<Product pid={item.\_id}

price={item.pdtPrice}

name={item.pdtName}

key={item.\_id}

desc={item.pdtDescription} img={item.pdtImg} rating={item.avgFeedback}

status={item.isDiscontinued}/>

</div>

}

}

export default ProductDetails;

React JSX

Copy

Let us see the changes to be made inside Product and Rater components after making the above changes:

**Product component:**

var React=require('react');

var Rater =require('./Rater.jsx').Rater;

class Product extends React.Component{

constructor(props){

super(props);

}

render(){

return(<div className={"thumbnail"}>

<img src={this.props.img} class="img img-rounded img-responsive"/>

<div class="caption" style="text-align:center">

<a href=”/productDetails”}><h3> {this.props.name} </h3></a>

<h4><span style="color:red"> Rs.{this.props.price} </span></h4>

{this.props.status ? <h5 style={{"color":"red","font-weight":"bold"}}>

The product is discontinued.</h5> :

<p className={"description"}>{this.props.desc}</p>}

<Rater value={this.props.rating} maxlength="6"/>&nbsp;&nbsp;

<span style={{"fontSize":"9px"}}>{this.props.rating}/5</span>

</div>

</div>)

}

}

export default Product;

React JSX

Copy

**Rater component:**

var React=require('react');

class Rater extends React.Component{

render(){

var items=[];

for(var i=1;i<this.props.maxlength;i++){

var clickHandler=this.props.onSelected && this.props.onSelected.bind(null,i);

items.push(<li className={i<=this.props.value && 'filled'}

onClick={clickHandler}>{'\u2605'}</li>)

}

return(

<ul className="rating">{items}</ul>

)

}

}

export default Rater;

React JSX

Copy

Now we have successfully added the props to the components. As we have static data, user cannot interact with it. In order to understand how to handle the dynamic data, let us create a new component as shown below:



In the above screenshot, component is just displaying the existing feedback rating and feedback form for providing feedback, which are not interactive.

But the Rater component, above the submit button is interactive.

Let us make it interactive by making the below changes:

Create a JSON array of feedback data:

var feedbackData = [{

user:”khalid”,

feedbackDetails:”Worst product. First of all received very late. Forget about it.”,

rating:”2”

}];

JSON

Copy

Add the FeedbackComp reference in ProductDetails component and modify render method of ProductDetailscomponent as follows:

render(){

return <div>

<Product pid={item.\_id}

price={item.pdtPrice}

name={item.pdtName}

key={item.\_id}

desc={item.pdtDescription} img={item.pdtImg} rating={item.avgFeedback}

status={item.isDiscontinued}/>

<FeedbackComp feedbacks={feedbackData} />

</div>

}

React JSX

Copy

Create a new jsx page named FeedbackComp.jsx and include the below code:

import React from ‘react’;

import Rater from ‘Rater.jsx’;

class FeedbackComponent extends React.Component{

constructor(props){

super(props);

this.state={

latestFeedback:"",

rating:'5'

};

}

render(){

return(<div>

<div key={fb.pdtCode}>

<a href="#"><h4>{fb.user}</h4></a>

<Rater value={fb.rating} maxlength="6" onSelected={this.handleClick}/>

<span>{fb.rating}/5</span><br/>

<div>{fb.body}</div>

<h6 style={{"fontStyle":"italic", "color":"lightgrey"}}>

-{fb.updatedAt.substr(0,10)}</h6>

</div>

<div className={"form-group"}>

<textarea className={"form-control"} rows="5" cols="12"

value={this.state.latestFeedback}

name="username"></textarea></div>

<div>

<Rater value={this.state.rating} maxlength="6"/>

<span style={{"fontSize":"9px"}}/>

<span>{this.state.rating}/5</span>

</div><br/>

<button class="btn btn-primary">Submit Feedback</button

</div>);

}

}

export default FeedbackComp;

React JSX

Copy

As per the above updates, now user will be able to provide feedback and feedback rating.

Exercise 2 : Defect tracker application

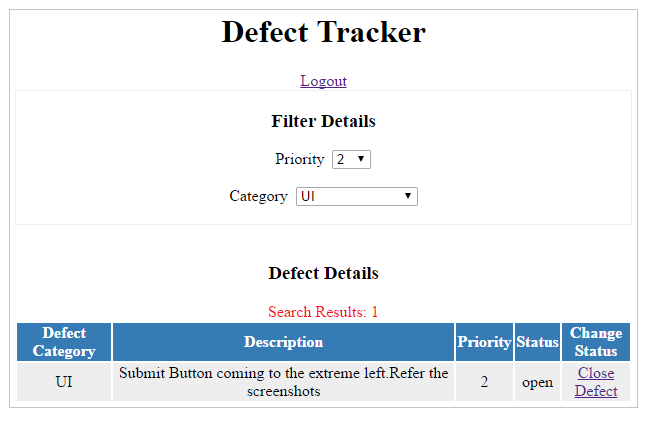
**Time Limit:**50 Minutes

Problem Statement

Continuing with the previous assignment, wherever we had the hard coded data, that should be used as state and props.

Note 1: hardcode the priority and category value.

Developer view of Defect Tracker is shown below:



### Why lifecycle methods?

Let's consider below Educator component which uses an array of JSON data assigned to a JavaScript variable course.

Code snippet:

var course = [{

"Name": "Angular",

"educator": "Khalid"

},

{

"Name": "ReactJS",

"educator": "Roopashri"

},

{

"Name": "Ajax",

"educator": "Krishna"

}

]

class Educator extends React.Component {

constructor(props){

super(props);

this.state = {crsData:course};

}

render(){

var users = this.state.crsData;

return(<div>

<h4>UI Courses list</h4>

<div> {users.map(function(userDetails){

return (<div>

<b>Course: </b>{userDetails.Name}, <b><br/>Educator: </b>

{userDetails.educator} <br/><br/>

</div>); })}

</div>

</div>);

}

}

export default Educator;

JSON

Copy

In the above example course data is hard coded in a component. But in real time examples, we may require to load data from server, for this we use Ajax.

Using Ajax, we can retrieve the data from server, but Ajax call cannot be made from render method of a component. And if we want to display this data on load of the page, we should make a call to server immediately after the component mounts to DOM.

But we will not be aware of, when the component mounts to DOM and when it is available for accessing.

In this scenario, we can use the component lifecycle methods of React that allows us execute actions at particular times.

Now let us discuss what are these lifecycle methods and how it helps us in different scenarios.

### What are lifecycle methods?

Every component will have following phases in its lifecycle:

* Mounting phase - when the component is mounted to DOM tree
* Updating phase - when component is being updated with new state, new props are being received
* Unmounting phase - destroying component from DOM tree

Every phase in the lifecycle of a component has few methods which will be invoked during that phase of a component's lifecycle. We can override these methods to provide the desired functionality.

These methods can be used in the following cases:

* For making an Ajax call, to set timer and to integrate with other frameworks
* To avoid unnecessary re-rendering of a component
* To update the component, based on the props changes
* For clearing the values when component is unmounted

### Lifecycle methods : Mounting phase

At mounting phase of component, following methods which will be invoked:

* componentWillMount()
* componentDidMount()

**componentWillMount():**

* Will be invoked both on client and server side
* Executed before the initial rendering of a component happens
* Executed only once in component's lifecycle

**componentDidMount():**

* Invoked once on client side
* Invoked immediately after initial render occurs
* Now will have access to all DOM nodes
* Best place for integrate with other JS frameworks, set timers and handling Ajax request

Demo 14 : Lifecycle methods

Highlights:

 Understanding componentWillMount()

 Understanding componentDidMount()

Demo Steps:

Modify the Timer component by introducing lifecycle methods in it.

Here in this demo, Timer will start immediately after the component is mounted.

1. Create App.jsx as shown below:

import React from 'react';

class Timer extends React.Component{

constructor(props){

super(props);

this.start = this.start.bind(this);

this.state = {secondsElapsed: 0};

}

start(){

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

}

componentWillMount(){

alert('Component mounted');

}

componentDidMount(){

this.interval = setInterval(this.start, 1000);

}

render(){

return (<div>

<h2>Seconds Elapsed: {this.state.secondsElapsed}</h2>

</div>);

}

}

export default Timer;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Timer from './App.jsx';

ReactDOM.render(<Timer />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

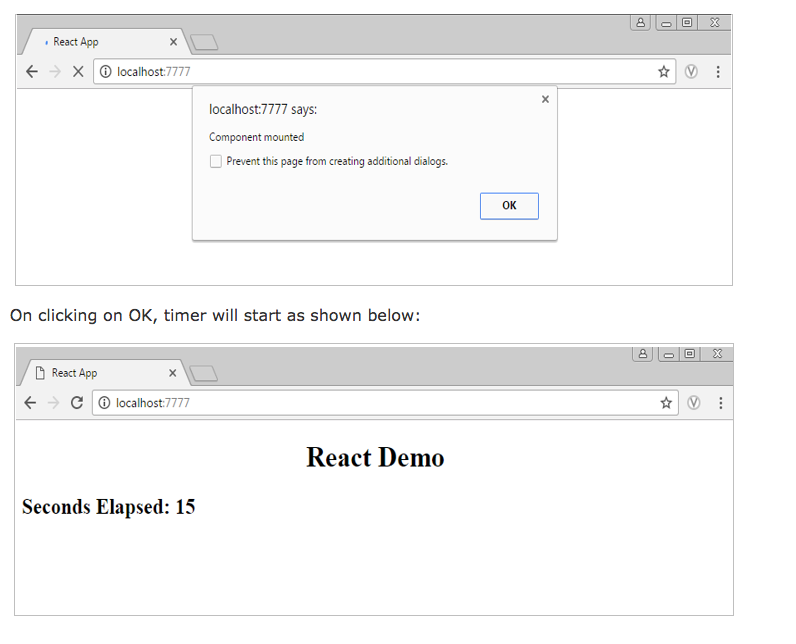
</body>

</html>

HTML

Copy

4. Observe the below output:



### Lifecycle methods : Updating phase

At updating phase of a component, below methods are executed based on props or state being received and updates changes in component:

* componentWillReceiveProps(newProps)
* shouldComponentUpdate(nextProps, nextState)
* componentWillUpdate(nextProps, nextState)
* componentDidUpdate()

**componentWillReceiveProps(newProps):**

* Invoked when component receives new props
* Can be used to perform any operation based on received props

**shouldComponentUpdate(nextProps, nextState):**

* Executed before rendering when new state or props are being received
* Changes in props or state will cause re rendering of a component
* By default it will return true so component will get re-rendered by invoking next lifecycle methods
* In case we want to avoid re rendering, this method could be used by returning false, so this would skips render(). Hence it is useful for performance

**componentWillUpdate():**

* Invoked before rendering new props or state
* This would be the place for performing preparation before the update happens to the DOM
* Once the updates flushed to the DOM, immediately next lifecycle method will be invoked

**componentDidUpdate():**

* Invoked immediately after the changes are updated to the DOM

Demo 15 : Lifecycle methods

Highlights:

 Lifecycle methods usage at updating phase

 Clearing the values when component unmounted

 Avoiding re-rendering of a component

Demo Steps:

Add other lifecycle methods to Timer component and observe how the component will be updated:

Component will be updated only if the value is <=4, and component will be unmounted from DOM after 15seconds.

1. Create App.jsx as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import $ from 'jquery';

class Timer extends React.Component{

constructor(props){

super(props);

this.start = this.start.bind(this);

this.state = {secondsElapsed: 0};

}

start(){

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

}

componentDidMount(){

this.interval = setInterval(this.start, 2000);

}

componentWillUnmount(){

console.log('Component WILL UNMOUNT!')

clearInterval(this.interval);

}

render(){

return (<div>

<Updates new={this.state.secondsElapsed} />

</div>);

}

}

class Updates extends React.Component{

componentWillReceiveProps(newProps){

console.log('Received new props:', newProps);

}

shouldComponentUpdate(newProps, newState){

if(this.props.new <=4){

console.log('shouldComponentUpdate:', newProps);

return true;}

else{ return false; }

}

componentWillUpdate(nextProps, nextState){

console.log('Component updated:', nextProps);

var node = $(ReactDOM.findDOMNode(this));

node.slideUp();

node.slideDown();

console.log('Updating');

}

componentDidUpdate(prevProps, prevState){

console.log('Previous value destroyed:', prevProps);

console.log('Updated');

}

render(){

return (<div>

<h2>Seconds Elapsed: {this.props.new}</h2>

</div>);

}

}

export default Timer;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Timer from './App.jsx';

ReactDOM.render(<Timer />, document.getElementById('app'));

setTimeout(() => {

ReactDOM.unmountComponentAtNode( document.getElementById('app') );

}, 15000);

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

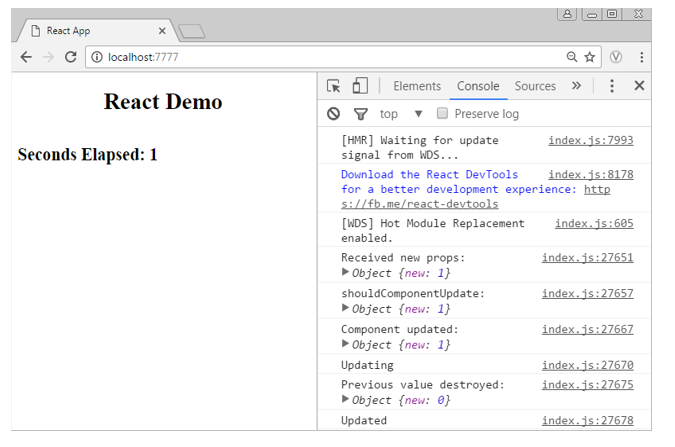
</body>

</html>

HTML

Copy

4. Observe the below output:



### Lifecycle methods : Unmounting phase

When the component is unmounted from DOM, componentWillUnmount() method will be invoked.

* This method is used at unmounting process of a component
* By unmounting the component, browser memory will be cleaned up
* A component can be unmounted from DOM using ReactDOM.unmountComponentAtNode()

### Loading data using AJAX

In React, to retrieve data from the server, Ajax will be used.

* React by default does not provide any helper method to manage Ajax requests
* Any other third party JavaScript library can be used to handle Ajax requests
* As we discussed already, in lifecycle methods, componentDidMount method is the best place to handle Ajax request
* Using Ajax fetch the data in the event handler of componentDidMount()
* Once we have the data, make it available to our component by setting state of a component to trigger re-render

### Loading data using AJAX

Below demo illustrates how to use Ajax to retrieve data from server and how data will be stored as component's state:

* In a componentDidMount() method, using $.ajax method retrieve data by mentioning path using url property
* Update the retrieved data as state of a component so that the component will be re rendered

componentDidMount() {

this.setState({crsData : $.parseJSON(

$.ajax({

url: '/course.json',

async: false,

dataType: 'json'

}).responseText

)});

}

Demo 16 : Loading data using Ajax

Highlights:

 Retrieving data

 Usage of componentDidMount() to handle Ajax request

Demo Steps:

Create Courses component and retrieve data from the course.json file.

1. Create course.json as shown below:

[{

"Name": "React",

"description": "This course is intended to introduce the concepts of React JS to develop web pages. It helps to understand features of React, JSX, components, how to modify the data in a very efficient manner and flux overview."

},

{

"Name": "Angular",

"description": "This course is intended to introduce the concepts of Angular JS to develop web pages. It helps to understand basic concepts of Single Page application (SPA), Module View Controller (MVC) concepts in angular application, Angular directives, filters, Dependency Injection, and Ajax Programming"

},

{

"Name": "Handlebars",

"description": "The course introduces the client side templating and when to use a javacsript templating engine. It then helps in understanding Handlebar framework and explains Expressions, Blocks, Paths, Partials and Helpers ."

},

{

"Name": "SASS",

"description": "This course is intended to introduce the need of CSS preprocessor and SASS. The course details about SASS compilation, installation, variables, mixins, operations, color functions, nested rules, directives, SASS IDEs and elements."

}]

JSON

Copy

2. Create App.jsx as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import $ from 'jquery';

class Courses extends React.Component {

constructor(props){

super(props);

this.state = {crsData: []};

}

componentDidMount(){

this.setState({ crsData : $.parseJSON($.ajax ({

url: '/course.json',

async: false,

dataType: 'json'}). responseText)

});

}

render() {

var users = this.state.crsData;

return(<div>

<table>

<tbody><tr><th> UI Courses list </th></tr>

{users.map(function(userDetails){

return(<tr>

<td><b>Course:{userDetails.Name}</b><br/><br/>

<b>Description:</b>{userDetails.description}

<br/><br/></td></tr> );})}

</tbody>

</table>

<br/><br/>

<div><br/>

<a href="http://digitaltutor/#/track/17">DigitalTutor</a> |

<a href="http://ilearn/"> iLearn </a> |

<a href="http://kshop/"> KShop </a><br/><br/>

<p> &copy; 2016 ETA | UI & Markup,Infosys Limited.

All rights reserved.</p>

<br/>

</div>

</div>);

}

}

export default Courses;

React JSX

Copy

3. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Courses from './App.jsx';

ReactDOM.render(<Courses />, document.getElementById('app'));

JavaScript

Copy

4. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> UI - ETA </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

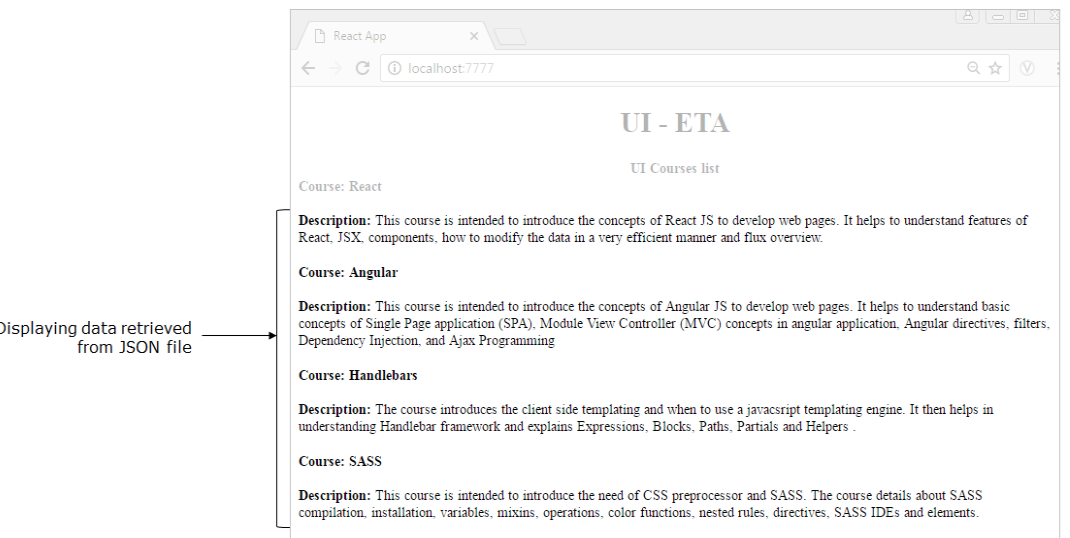
</body>

</html>

HTML

Copy

4. Observe the below output:



Demo 17 : Case study implementation milestone 3

Highlights:

 Using lifecycle method

 Retrieving data from server

Demo Steps:

In our last assignment we rendered the component with data from a hardcoded JSON array. Now let us see how to fetch the same data from the  server using jQuery $.ajax method.

**ProductDetails Component:**

import React from ‘react’;

import Product from ‘./Product.jsx’;

import FeedbackComp from ‘./FeedbackComp.jsx’;

import $ from 'jquery';

var productId;

//user defined function which makes the ajax call.

function getProductDetailsState(){

let params={"pdtId":’lt-023’};

$.ajax({

url:’ /productDetails’,

type:'POST',

data:data,

async:false,

success:function(retObj){

retData =retObj;

},

error:function(err){

retData =err;

}

});

return {

productDetails:retData.pdtDetails,

feedbackDetails:retData.fbDetails.feedbackDetails,

productId:retData.fbDetails.pdtCode

};

}

class ProductDetails extends React.Component{

constructor(props){

super(props);

this.state = {

productDetails:{},

feedbackDetails:[],

pdtId:""

};

this.\_onChange=this.\_.onChange.bind(this);

}

componentWillMount(){

this.loadStates();

}

componentDidMount(){

setInterval(this.loadStates,1000);

}

render(){

return <div>

<Product pid={item.\_id}

price={item.pdtPrice}

name={item.pdtName}

key={item.\_id}

desc={item.pdtDescription} img={item.pdtImg} rating={item.avgFeedback}

status={item.isDiscontinued}/>

<FeedbackComp feedbacks={feedbackData} />

</div>

}

function loadStates(){

let stateObj=getProductDetailsState();

this.setState({

productDetails:stateObj.productDetails,

feedbackDetails:stateObj.feedbackDetails,

pdtId:stateObj.productId

});

}

}

export default ProductDetails;

React JSX

Copy

ComponentWillMount will fetch the initial value from the server and componentDidMount periodically calls same method for every 1000ms for the updates.

Exercise 3 : Defect tracker application

**Time Limit:**30 Minutes

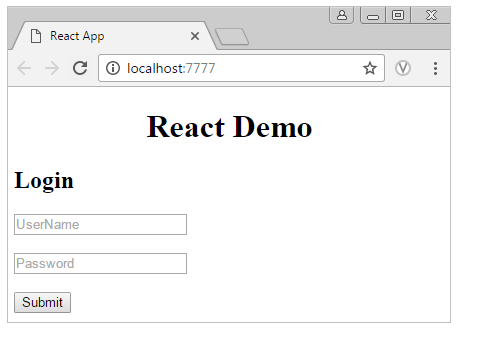
Problem Statement

In the previous assignments, we had hard coded the defect details values as props and states properties. Now fetch these defect details from any of the sources like localStorage, sessionStorage or a database. Component life cycle methods should be used to implement the above calls to any of the above sources.

### Why forms?

In most of the application we need forms for taking user input, for placing an order and for booking tickets etc.:

Example: Login component



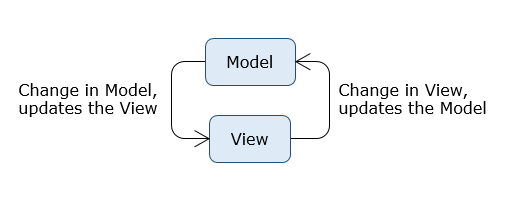
The form that we create in React for accepting user input are different from HTML forms.

User will not be able to input value in form input field because ReactJS uses unidirectional data flow.

Let us now learn unidirectional data flow and form handling in ReactJS.

Forms : Unidirectional data flow

Most of the frameworks follow two way data binding. This would allow for updates from both the end i.e. changes can be done from view to model and vice versa.



In this approach state of model could be mutated by both model and view, it would cause unpredictable data flow.

React uses unidirectional data flow pattern where in changes will be done only from component to view but not vice versa.

This would prevent unpredictable data flow and easy to debug.

So let's discuss how HTML form works in ReactJS through a demo.

Demo 18 : Forms

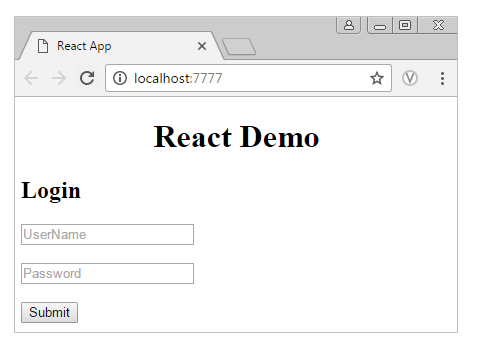
Highlights:

 Creating forms in a component

 Creating form elements

Demo Steps:

Create a Login component using form elements as shown below and observe how it works:



1. Create App.jsx as shown below:

import React from 'react';

class Login extends React.Component {

render(){

return <form>

<h2> Login </h2>

<input type = "text" value = "" placeholder="UserName" /> <br/><br/>

<input type = "password" value = "" placeholder="Password" /> <br/><br/>

<input type = "submit" />

</form>

}

}

export default Login;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Login from './App.jsx';

ReactDOM.render(<Login />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

HTML

Copy

How to work with React form elements?

In the previous demo, user is not able interact with an input element.

We can make form elements interactive by setting a callback to the onChange prop. Form components listen for changes and this fires when,

* The value of <input> or <textarea> changes
* Checked state of <input> changes
* Selected state of <input> changes

For updating the value in response to user interaction, onChange prop could be used as follows:

<input type = "name"

name = "username"

placeholder = "Enter Name"

onChange = {this.setEmpState}

value = {this.state.password} />

React JSX

Copy

Demo 19 : Forms

Highlights:

 Making form element interactive

 Using onChange props

Demo Steps:

Revisit previous Login component and make it interactive using onChange props:

1. Create App.jsx as shown below:

import React from 'react';

class Login extends React.Component{

constructor(props){

super(props);

this.state = { data: {name: '', password: ''} }

this.setEmpState = this.setEmpState.bind(this);

this.handleLogin = this.handleLogin.bind(this);

};

setEmpState(e){

var field = e.target.name;

var value = e.target.value;

this.state.data[field] = value;

this.setState({ data: this.state.data });

}

handleLogin(e){

e.preventDefault();

alert('Logged in successfully');

}

render(){

return <form><h2> Login </h2>

<input type = "text" name="name" placeholder = "UserName" value = {this.state.data.name} onChange = {this.setEmpState}/><br/><br/>

<input type = "password" name="password" value = {this.state.data.password} placeholder = "Password" onChange = {this.setEmpState}/><br/><br/>

<input type = "submit" onClick = {this.handleLogin} />

</form>

}

}

export default Login;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Login from './App.jsx';

ReactDOM.render(<Login />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

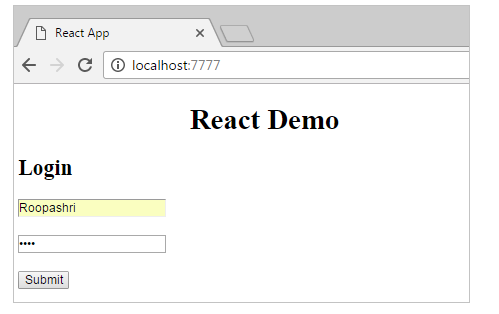
</body>

</html>

HTML

Copy

4. Observe the below output:



### Forms : React refs

In React we use **ref** for getting reference of a node. And React **refs** will return the reference of the DOM node.

Let's see how ref and refs will be used to get the reference of a node.

ref will be used to give the reference name to a node as shown below:

<input type = "text" ref = "userName" value = "" />

HTML

Copy

To refer the node, refs will be used as follows:

this.refs.userName.value

Demo 20 : React refs

Highlights:

 How to refer a node element

 How to access the value of a node

Demo Steps:

Add validations to the previous Login component and display values entered by user:

1. Create App.jsx as shown below:

import React from 'react';

class Login extends React.Component{

constructor(props){

super(props);

this.state = {

data: {name: '', password: ''}

}

this.setEmpState = this.setEmpState.bind(this);

this.handleLogin = this.handleLogin.bind(this);

};

setEmpState(e){

var field = e.target.name;

var value = e.target.value;

this.state.data[field] = value;

return this.setState({ data: this.state.data });

}

handleLogin(e){

e.preventDefault();

if((this.refs.name.value == "") || (this.refs.pwd.value == ""))

{ alert(" Please enter data ");}

else

{console.log(" Username: " + this.state.data.name );

console.log( "Password: " + this.state.data.password );

}

}

render(){

return <form>

<h2> Login </h2>

<input type = "text" name = "name"

ref = "name"

value = {this.state.data.name}

placeholder = "UserName" onChange = {this.setEmpState}/><br/><br/>

<input type = "password" name = "password"

ref = "pwd" value = {this.state.data.password} placeholder = "Password"

onChange = {this.setEmpState}/> <br/><br/>

<input type = "submit" onClick = {this.handleLogin} />

</form>

}

}

export default Login;

React JSX

Copy

2. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Login from './App.jsx';

ReactDOM.render(<Login />, document.getElementById('app'));

JavaScript

Copy

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

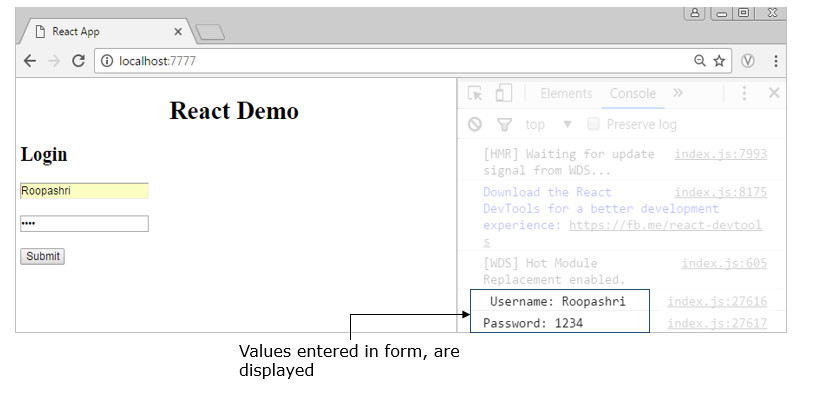
</body>

</html>

HTML

Copy

4. Observe the below output:



Demo 21 : Case study implementation milestone 4

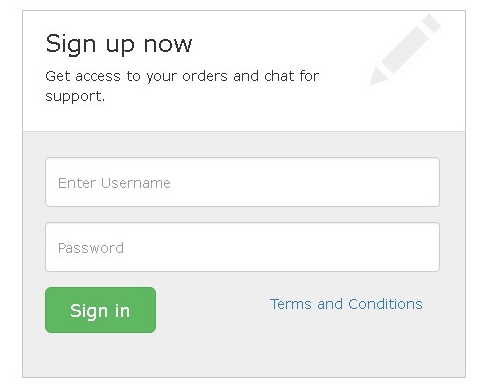
Highlights:

 Using forms

 Creating login form

Demo Steps:

Lets create a login form for the case study application as follows:



Login component:

import React from 'react';

class Login extends React.Component{

constructor(props,context){

super(props);

this.handleSubmit=this.handleSubmit.bind(this);

this.handleUsernameChange=this.handleUsernameChange.bind(this);

this.handlePasswordChange=this.handlePasswordChange.bind(this);

this.state = {username:"",password:""};

}

handleUsernameChange(e){

this.setState({username:e.target.value});

}

handlePasswordChange(e){

this.setState({password:e.target.value});

}

handleSubmit(event){

event.preventDefault();

//Login Logic goes here.

}

componentDidMount(){

document.body.style.background="url('Images/bg2.jpg') no-repeat center center fixed";

document.body.style.backgroundSize="cover";

}

render(){

return(<div className="form-layout">

<div className={"panel-heading"}>

<div className="panel-heading-left">

<h3>Sign up now</h3>

<p>Get access to your orders and chat for support.</p>

</div>

<div className="panel-heading-right">

<span className="glyphicon glyphicon-pencil"></span>

</div>

</div>

<div id="divLogin" className={"bgImage panel-body"}>

<form onSubmit={this.handleSubmit}>

Username:<input type=”text”

onChange={this.handleUsernameChange}/>

Password:<input type=”text”

onChange={this.handlePasswordChange}/>

<button type=”button”>Sign in</button>

<a href="#" block onClick={this.onFnf}>

Terms and Conditions

</a>

</form>

</div>

</div>);

}

}

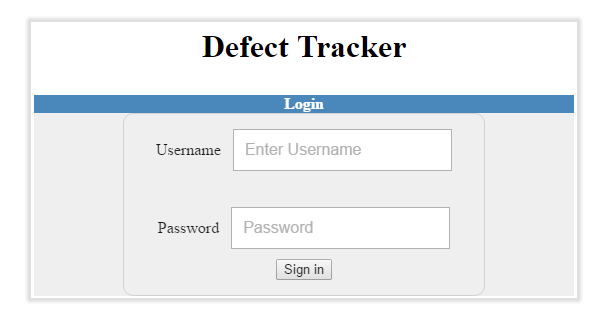
export default Login;

Exercise 4 : Defect tracker application

**Time Limit:**30 Minutes

Problem Statement

Let us create a login page which accepts the credentials and navigates to defect details page on validation of credentials. Screenshot is shown below.



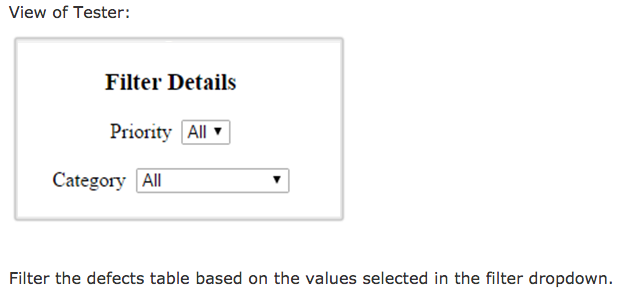
At the beginning of the assignment we had mentioned that the assignment application has two roles, tester and developer. Privileges given to the roles are listed below.

**Tester role:**

1. View Defects page: After logging in tester should be routed to View Details page where she/he can view the existing defects, filter those defects and close the same
2. Add Defects page: This right is granted only for tester role, where he can log new defects with priority, category and description. Added defects should be immediately listed in the View Defect tables

**Developer role:**  
View Defects page: Developers can only view the defects, filter the view and close the defects once it is fixed

Screenshots of the Add defect view is shown below:



### Why routers?

As we all know, these days we are switching from multi page application app to single page application (SPA) because of the below issues we had in multi page applications:

* Every request will be sent to server from client
* Server responds to client with new HTML content
* Everytime page reload will happen for every request
* This would increase the round trips to server and also delay in response

SPA overcomes limitations of multi-page application as described below:

* Rather than loading new page from server on every user interaction (such as clicking login button) instead, it loads entire web page containing all views from server when the application starts
* As a result, after initial page load, no server communication is required for further page updates upon user interaction

So here, we have to navigate from one view to another without hitting server. For this ReactJS provides react-router-dom library.

Next let's see how to configure routes in ReactJS application using react-router-dom library.

### Router configuration

For configuring routes in our application, follow below steps:

Step 1: Install react-router-dom library by running npm command in your application folder:

**npm install react-router-dom**

Example: If your folder "react\_demo" is on desktop then install it in that particular path as shown below:



Step 2: Import required router components from react-router-dom library as shown below:

import { BrowserRouter as Router, Route, Link} from 'react-router-dom';

React JSX

Copy

Let's discuss how to use these components and their props:

**<BrowserRouter>:** It is a primary component containing entire routing configuration.It is more popular one because it uses the HTML5 History API to keep track of your router history. All the routes should be wrapped within a container element such as div.

**<Route>:**It is a prop of Router. It map routes to our application’s component hierarchy using "path“ and "component" props and sets all navigational routes

**path:**Maps to path given in URL

**component:** Contains component name to be rendered when the route is mapped

**exact:**This property tells Route to match the exact path. In the below example Header component will render when browser's location path matches exactly /

Example:

<Router>

<!--Configures route for main component i.e. Header-->

<div>

<Route exact path = "/" component = {Header} />

<Route path = "/login" component = {Login} /> <!--Configures route for Login component-->

<Route path = "/home" component = {Home} /> <!--Configures route for Home component-->

</div>

</Router>

React JSX

Copy

**<Link>:** Create navigational links using "to" prop

Example:

<Link to="/home"> Home </Link>

<Link to="/login"> Login </Link>

HTML

Copy

Demo 22 : Router

Highlights:

 Configuring routes

 Navigation from a component to an other component

Demo Steps:

Create components to observe how navigation happens in SPA:

1. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter as Router, Route, Link} from 'react-router-dom';

class Header extends React.Component {

render() {

return (<Router>

<div>

<Link to="/">Home</Link> |

<Link to="/about"> About Us </Link> |

<Link to="/courses"> Courses </Link> |

<Link to="/contact"> Contact Us </Link>

<Route exact path = "/" component = {Home} />

<Route path = "/about" component = {About} />

<Route path = "/courses" component = {Courses} />

<Route path = "/contact" component = {ContactUs} />

</div>

</Router>)

}

}

export default Header;

export class Home extends React.Component {

render() {

return (

<div>

<p> Inside home</p>

</div>

)

}

}

export class About extends React.Component {

render() {

return (

<div>

<p>Inside AboutUs component...</p>

<p>Rendered as default child component to Home component</p>

</div>

)

}

}

export class Courses extends React.Component {

render() {

return (

<div>

<p>Inside Courses component...</p>

</div>

)

}

}

export class ContactUs extends React.Component {

render() {

return (

<div>

<p>Inside ContactUs component...</p>

</div>

)

}

}

ReactDOM.render(<Header />, document.getElementById('app'));

JavaScript

Copy

2. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

HTML

Copy

3. Observe the below output: For about route, About component is rendered.



Demo 23 : Router

Highlights:

 Configuring routes

 Configuring route programmatically

Demo Steps:

Configure routes programmatically:

**history**object is used to programmatically navigate to a new path using the method push as follows:

this.props.history.push('/home');

React JSX

Copy

1. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter as Router, Route, Link} from 'react-router-dom';

export class Home extends React.Component {

render() {

return (

<div><br/><br/>

<img src="/img/UI.PNG" width="600" height="190"/><br/><br/>

<h4> Just Launched...</h4><br/>

<li>Angular</li>

<li>React</li>

<li>Express</li>

<li>Vue</li>

</div>

)

}

}

export class Login extends React.Component {

constructor(props){

super(props);

this.state = { data: {name: '', password: ''}}

this.setEmpState = this.setEmpState.bind(this);

this.handleLogin = this.handleLogin.bind(this);

};

setEmpState(e){

var field = e.target.name;

var value = e.target.value;

this.state.data[field] = value;

return this.setState({data: this.state.data});

}

handleLogin(e){

e.preventDefault();

if((this.refs.name.value=="") || (this.refs.pwd.value==""))

{ alert("Please enter details"); }

else

{ this.props.history.push('/home'); }

}

render(){

return (<div>

<form onSubmit={this.handleLogin}>

<br/><h2>Login</h2>

<input type="text" name="name" ref="name"

placeholder="User Name" />

<br/><br/>

<input type="password" name="password" ref="pwd"

placeholder="Password" />

<br/><br/>

<button>Submit</button>

</form>

</div>);

}

}

ReactDOM.render((<Router>

<div>

<Route exact path = "/" component = {Login} />

<Route path = "/home" component = {Home} />

</div>

</Router>),document.getElementById('app'));

JavaScript

Copy

2. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

HTML

Copy

3. Download the image UI.PNG from the link - [UI.PNG](http://ilpce/ce/Course/UploadedCourseFiles/REACT/REACT_3.zip)

### Router configuration

**Route parameters:**

Parameters passed along with URL are called route parameters.

Example:

<Link to="/display/React"> React </Link>

HTML

Copy

In the above code snippet, on click of **React**link, url will be updated as localhost:7777/display/React.

To render a component, based on the url changes route can be configured as follows:

<Route path="/display/:topic" component={Display}/>

HTML

Copy

In the above code snippet, **Display** component gets rendered, only when a parameter is passed to the path.

Here, **topic** is the route param.

In Display component route param can be accessed as follows:

this.props.match.params.topic

React JSX

Copy

**match:** react-router-dom passes in a prop called match into every route that is rendered. Inside this match object there is an other object called params.

**params:** It is an object containing URL parameters

Demo 24 : Router

Highlights:

 Configuring routes

 Route parameters

Demo Steps:

Accessing route parameters:

1. Create main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter as Router, Route, Link} from 'react-router-dom';

export class Home extends React.Component {

render() {

return (

<Router>

<div>

<h4>Welcome&nbsp;{this.props.match.params.name} to UI courses</h4>

<img src="/img/UI.PNG" width="600" height="190" /><br/><br/>

<h4> Just Launched...</h4>

<Link to="/display/Angular"> <li>Angular</li> </Link> |

<Link to="/display/React"> <li>React</li></Link> |

<Link to="/display/Express"> <li>Express</li> </Link> |

<Link to="/display/Vue"> <li>Vue</li> </Link>

<br/>

<Route path="/display/:topic" component={Display}/>

</div>

</Router>

)

}

}

export class Display extends React.Component{

render(){

return <div>

<h2>Inside {this.props.match.params.topic} component</h2>

</div>

}

};

export class Login extends React.Component {

constructor(props){

super(props);

this.state = { data: {name: '', password: ''}}

this.setEmpState = this.setEmpState.bind(this);

this.handleLogin = this.handleLogin.bind(this);

};

setEmpState(e){

var field = e.target.name;

var value = e.target.value;

this.state.data[field] = value;

return this.setState({data: this.state.data});

}

handleLogin(e){

e.preventDefault();

if((this.refs.name.value=="") || (this.refs.pwd.value==""))

{ alert("Please enter details"); }

else

{ this.props.history.push('/home/'+this.refs.name.value); }

}

render(){

return (<div>

<form onSubmit={this.handleLogin}>

<br/><h2>Login</h2>

<input type="text" name="name" ref="name"

placeholder="User Name" />

<br/><br/>

<input type="password" name="password" ref="pwd"

placeholder="Password" />

<br/><br/>

<button>Submit</button>

</form>

</div>);

}

}

ReactDOM.render((<Router >

<div>

<Route exact path = "/" component = {Login} />

<Route path = "/home/:name" component = {Home} />

</div>

</Router>), document.getElementById('app'));

JavaScript

Copy

2. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

<h1 align="center"> React Demo </h1>

<div id = "app"> </div>

<script src = "index.js"></script>

</body>

</html>

HTML

Copy

Demo 25 : Case study implementation milestone 5

Highlights:

 Configuring routes

 Navigation of components

Demo Steps:

Let us configure the routes required for our Customer Service application:

**./src/shared/routes.js:**

import React from "react";

import LoginForm from "./components/LoginForm";

import App from "./components/App";

import PurchasedItems from "./components/PurchasedItems";

import ProductDetails from "./components/ProductDetails";

import Dashboard from "./components/Dashboard";

var routes = [

{

path:'/',

component:LoginForm,

exact:true

},

{

path:'/purchasedItems',

component:PurchasedItems

},

{

path:'/dashboard',

component:Dashboard

},

{

path:'/productDetails/:id',

component:ProductDetails

}

]

export default routes;

JavaScript

Copy

Here App component acts as the container of all other components in the case study.

**Rendering - src/client/routes.js:**

import React from "react";

import { BrowserRouter as Router,Route } from "react-router-dom";

import sharedRoutes from "../shared/routes";

export default (<Router>

<div>

{sharedRoutes.map(route => (

<Route {...route} />

)

)

}

</div>

</Router>

);

JavaScript

Copy

**src/client/main.js:**

import ReactDOM from 'react-dom'

import React from 'react'

import routes from './routes'

ReactDOM.render(routes, document.getElementById('react'))

Exercise 5 : Defect tracker application

**Time Limit:**30 Minutes

Problem Statement

Implement routing functionality for the application and render the routes instead of individual components in the entry js file. After implementing routing the urls of the views should be according to the routes added.

### Server side rendering

JavaScript frameworks are not usually search engine friendly as search engines have trouble reading full-featured JavaScript applications. Since single page application loads data dynamically and creates the markup, hence search engines cannot read and index such pages.

React resolves this problem as we can run the same React code on both client and the server side i.e. called Isomorphic rendering. And this gives the benefits of fast initial page load from the server and a great experience on the client.

Server side rendering is the process of handling the initial render when it is requested for the first time by a user.  When the request is received at the server, the required content will be rendered into an HTML string  and then sends it as a response to the client side.

When you render components on the server side and send HTML to the client side, React on the client side notices that the HTML already exists and it simply attaches event handlers to the existing elements and we can then enhance the experience or render more components at the client side.

This means that you can ship down only the HTML needed to render the page; then, any additional things can be pulled in and rendered on the client as needed. We get the benefit of fast page loading by server rendering, and can reuse the components.

Benefits of Server side rendering:  
1. Performance  
2. Search engine optimization

Let's see how to render React components on server side.

Here, we are using Express.js for server side rendering.

Express.js can be referred [here](http://ilpce/ce/LandingPage/Course/1842) for more details.

Demo 26 : Server side rendering

Highlights:

 How to render component on server side

 How to handle the rendered component at client side

Demo Steps:

Let's create a React component, render the same component on both side.

1. Create Component.jsx as shown below to create **Component**:

var React = require ('react');

var createReactClass =require ('create-react-class');

module.exports = createReactClass({

handleClick: function(){

alert("Clicked");

},

render: function (){

return (

<html>

<body>

<div>

<h1>React server rendering</h1>

<button onClick={this.\_handleClick}> Click </button>

</div>

<script src="/bundle.js" />

</body>

</html>

);

}

});

React JSX

Copy

2. Create main.js as shown below for rendering component on client side:

var React = require('react');

var ReactDOM = require('react-dom');

var createReactClass =require ('create-react-class');

var Component = require('./Component.jsx');

ReactDOM.render(React.createElement(Component), document);

JavaScript

Copy

3. Create a express server as shown below which will accept the request on port 3000 and returns the response which is a React component in HTML string.

server.js

require('babel-register')({

presets: ['react']

});

var express = require ('express'); // import express module

var app= express(); // The application object is an Express instance, usually denoted by the variable **app**

var React = require ('react'); // import react module

var Component = require('./Component.jsx'); // import Component

var ReactDOMServer = require('react-dom/server'); // import ReactDOMServer object

app.use(express.static('public')); // asset folder where bundle.js will be generated

app.get('/', function(request, response) { // function will be executed when the path is '/'

var html = ReactDOMServer.renderToString( // renders React element to its initial HTML and returns HTML string

React.createElement(Component)

);

response.send(html); // html string will be sent to client side

});

var PORT =3000; // Port number

app.listen(PORT, function() { // listen() method on application object to specify the port number on which the application listens for requests

console.log('http://localhost' +PORT);

});

JavaScript

Copy

Steps to run the demo:

a. Download this demo from the link - [server\_rendering](http://ilpce/ce/Course/UploadedCourseFiles/REACT/REACT_1.zip)

b. Navigate to the server\_rendering folder and install the required packages by running the command npm install as shown below:



c. Command to be used to run demo: npm start

d. Output can be seen in port 3000

Observe the below points in the given demo:

a. When JavaScript is disabled at browser side, we cannot handle click event, it just renders the content from the sever

b. To handle the click event, JavaScript has to be enabled at browser side

### Why flux architecture?

The ReactJS application which we have created so far is simple and can be maintained easily. But when the complexity of application grows, it will be difficult to manage. In this case, we can use an architectural pattern for developing and maintaining large scale applications to streamline the development process.

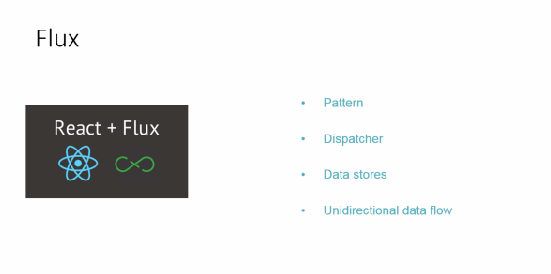
ReactJS uses Flux architectural pattern for developing an application where unidirectional dataflow is ensured. As the data flow is unidirectional it would be easy to track the event flow.

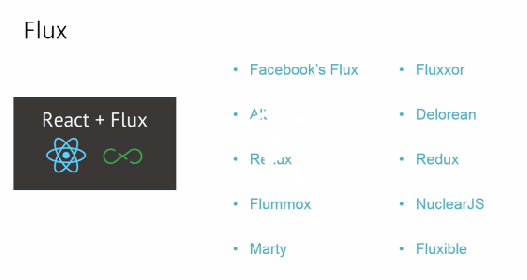
Let us understand and implement flux architecture in our case study.

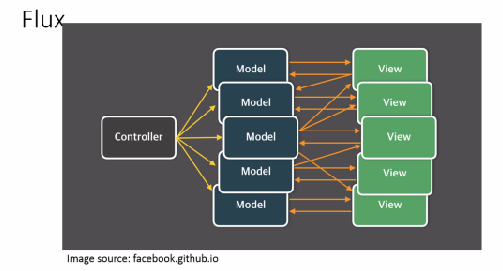
What is flux architecture?

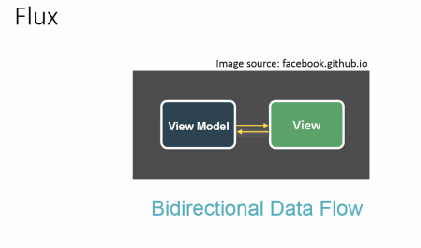
* Flux is an application architectural pattern which has an unidirectional data-flow
* To maintain unidirectional flow in an application, facebook has introduced flux architecture
* It's more of a pattern rather than a formal framework

Check react with flux

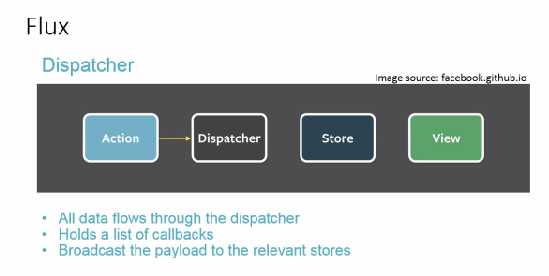


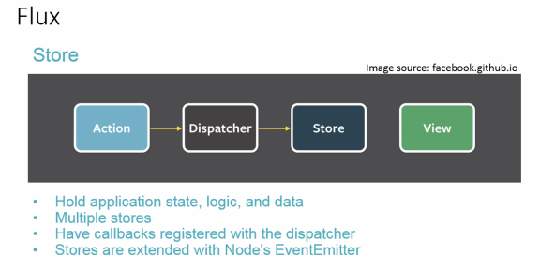


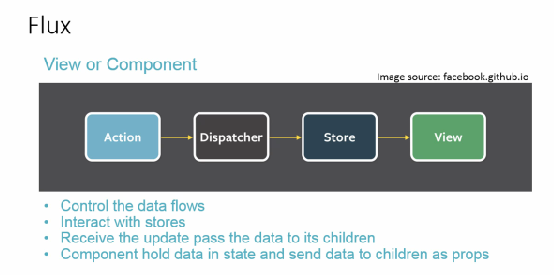








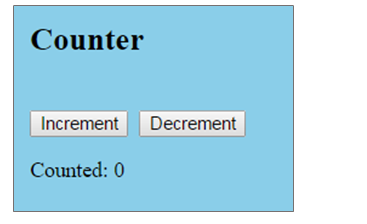


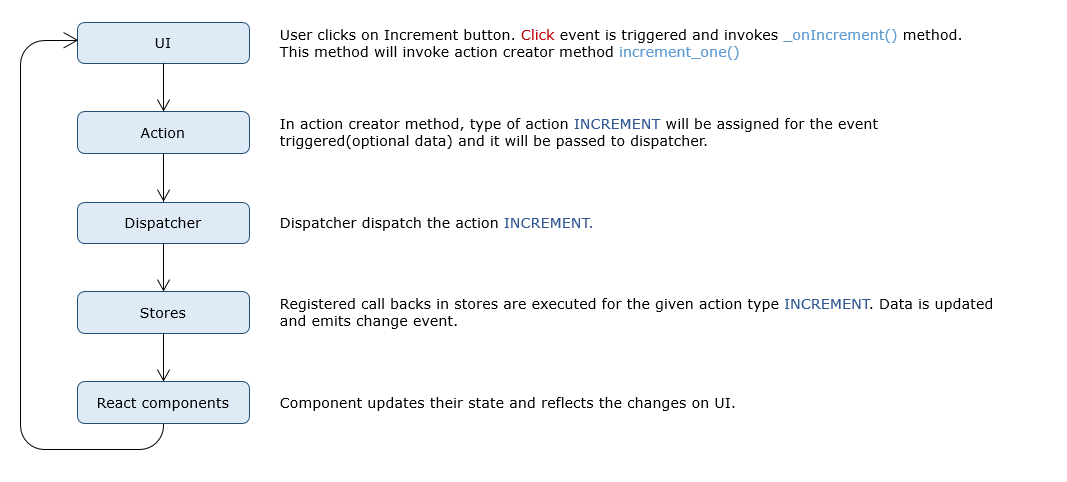


### Core concepts of flux

Data flow in flux architecture can be illustrated as follows by taking an example:

UI of the example:





### Core concepts of flux : Actions

Actions encapsulates specific events which occurs within the application. Actions which are getting triggered will be passed to dispatcher.

Dispatch helper methods that facilitate passing data to the Dispatcher.

Actions are triggered typically at 2 different places:

* When user interacts with the UI
* When the server returns an error code or when the server has updates to provide to the application

Example:

import AppDispatcher from '../dispatcher/AppDispatcher';

import CounterConstants from '../constants/CounterConstants';

var CounterActions = {

increment\_one: function(){

AppDispatcher.dispatch({

type: CounterConstants.INCREMENT

});

},

decrement\_one: function(){

AppDispatcher.dispatch({

type: CounterConstants.DECREMENT

});

},

};

module.exports = CounterActions;

### Core concepts of flux : Dispatcher

Dispatcher is the central hub that manages all data flow in a Flux application. It is a simple mechanism for distributing the actions to the stores.

When an action creator provides the dispatcher with a new action, all stores in the application receive the action via the callbacks in the registry.

Example:

var Dispatcher=require('flux').Dispatcher;

var Dispatcher= new Dispatcher();

Dispatcher.handleAction = function(action){

this.dispatch( actionObject );

}

module.exports=Dispatcher;

### Core concepts of flux : Stores

It is a container for application state & logic that have callbacks registered to the dispatcher. Stores registers itself with the dispatcher and provides it with a callback.

Within the store's registered callback, a switch statement based on the action's type is used which determines, for a given broadcast, is there any relevant actions to take.

State of Store will be updated and they broadcast a change event. Views that are listening (via addChangeListener() ) for this event, update their states accordingly.

Example:

var AppDispatcher = require('../dispatcher/AppDispatcher');

var CounterConstants = require('../constants/CounterConstants');

var BaseStore = require('./BaseStore');

var objectAssign = require('object-assign');

var \_count = 0;

var CounterStore = objectAssign({}, BaseStore, {

getCount: function () {

return \_count;

},

dispatchToken: AppDispatcher.register(function(action){

switch(action.type){

case CounterConstants.INCREMENT:

\_count = \_count + 1;

CounterStore.emitChange();

break;

case CounterConstants.DECREMENT:

\_count = \_count > 0 ? \_count - 1 : 0;

CounterStore.emitChange();

break;

}

})

});

module.exports = CounterStore;

Core concepts of flux : Controller views

Controller views are just React components that listen to change events (via addChangeListener()) and retrieve Application state from Stores. And they pass that data down to their child components via props.

Example:

import React from 'react';

import counterStore from '../stores/TickerStore';

import CounterActions from '../actions/CounterActions';

class Counter extends React.Component{

constructor(props) {

super(props);

this.\_onChange = this.\_onChange.bind(this);

this.\_getStateFromStores = this.\_getStateFromStores.bind(this);

this.\_onIncrement = this.\_onIncrement.bind(this);

this.\_onDecrement = this.\_onDecrement.bind(this);

this.state = this.\_getStateFromStores();

}

componentDidMount() {

counterStore.addChangeListener(this.\_onChange)

}

\_getStateFromStores() {

return {

count: counterStore.getCount()

};

}

\_onChange() {

this.setState(this.\_getStateFromStores());

}

componentWillUnmount() {

CounterStore.removeChangeListener(this.\_onChange)

}

\_onIncrement() {

CounterActions.increment\_one();

}

\_onDecrement() {

CounterActions.decrement\_one();

}

render() {

return <div style={{backgroundColor:"skyblue" }}><br/>

<h2>&nbsp;&nbsp;Counter</h2><br/>

<div >

&nbsp; &nbsp;<button onClick={this.\_onIncrement}>Increment</button> &nbsp;

<button onClick={this.\_onDecrement}>Decrement</button>

<p>&nbsp; &nbsp;Counted: {this.state.count} {this.state.data}</p><br/>

</div>

</div>

}

}

export default Counter;

Demo 27 : Flux architecture

Highlights:

 How data will flow in single direction

 Flux concepts

Demo Steps:

Below demo illustrates how the data flow happens in flux architecture:

1. Create modules/actions/CounterActions.js as shown below:

import AppDispatcher from '../dispatcher/AppDispatcher';

import CounterConstants from '../constants/CounterConstants';

var CounterActions = {

increment\_one: function(){

AppDispatcher.dispatch({

type: CounterConstants.INCREMENT

});

},

decrement\_one: function(){

AppDispatcher.dispatch({

type: CounterConstants.DECREMENT

});

},

};

module.exports = CounterActions;

JavaScript

Copy

2. Create modules/constants/CounterConstants.js as shown below:

module.exports = {

INCREMENT: 'INCREMENT',

DECREMENT: 'DECREMENT',

};

JavaScript

Copy

3. Create modules/dispatcher/AppDispatcher.js as shown below:

var Dispatcher=require('flux').Dispatcher;

var Dispatcher= new Dispatcher();

Dispatcher.handleAction = function(action){

this.dispatch( actionObject );

}

module.exports=Dispatcher;

JavaScript

Copy

4. Create modules/stores/BaseStore.js as shown below:

var AppDispatcher = require('../dispatcher/AppDispatcher');

var EventEmitter = require('events').EventEmitter;

var CounterConstants = require('../constants/CounterConstants');

var objectAssign = require('object-assign');

var CHANGE\_EVENT = 'change';

var BaseStore = objectAssign({}, EventEmitter.prototype, {

emitChange: function() {

this.emit(CHANGE\_EVENT);

},

addChangeListener: function(callback) {

this.on(CHANGE\_EVENT, callback);

},

removeChangeListener: function(callback) {

this.removeListener(CHANGE\_EVENT, callback);

}

});

module.exports = BaseStore;

JavaScript

Copy

5. Create modules/stores/TickerStore.js as shown below:

var AppDispatcher = require('../dispatcher/AppDispatcher');

var CounterConstants = require('../constants/CounterConstants');

var BaseStore = require('./BaseStore');

var objectAssign = require('object-assign');

var \_count = 0;

var CounterStore = objectAssign({}, BaseStore, {

getCount: function () {

return \_count;

},

dispatchToken: AppDispatcher.register(function(action){

switch(action.type){

case CounterConstants.INCREMENT:

\_count = \_count + 1;

CounterStore.emitChange();

break;

case CounterConstants.DECREMENT:

\_count = \_count > 0 ? \_count - 1 : 0;

CounterStore.emitChange();

break;

}

})

});

module.exports = CounterStore;

JavaScript

Copy

6. Create modules/main.js as shown below:

import React from 'react';

import ReactDOM from 'react-dom';

import Counter from './components/CounterComponent.jsx';

ReactDOM.render(<Counter />, document.getElementById('app'));

JavaScript

Copy

7. Create build/index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Counter</title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<style media="screen">

html,body{

margin:0;

height:100%;

}

#app{

height: 100%;

width: 100%;

}

</style>

</head>

<body>

<div id="app" ></div>

</body>

<script type="text/javascript" src="bundle.js"></script>

</html>

HTML

Copy

8. Create webpack.config.js as shown below:

var path = require("path");

module.exports = {

entry: [

'./modules/main.js',

'webpack-dev-server/client?http://localhost:8080'

],

output: {

path: path.resolve(\_\_dirname, "build"),

publicPath: "/",

filename: "bundle.js"

},

debug: true,

devtool: 'source-map',

module: {

loaders: [{

test: /\.jsx?$/,

exclude: /node\_modules/,

loader: "babel",

query:

{

presets:['es2015','react']

}

}]

},

devServer: {

contentBase: "./build"

}

};

JavaScript

Copy

**Note:**

a. Download this demo from here - [FluxDemo](http://ilpce/ce/Course/UploadedCourseFiles/REACT/REACT_2.zip)

b. Navigate to the FluxDemo folder and install the required packages by running the command npm install as shown below:



Demo 28 : Case study implementation milestone 6

Highlights:

 Flux concepts

 Implementing flux

Demo Steps:

Let us implement the Flux for Feedback component which we have created earlier:

**FeedbackComp.jsx:**

import React from 'react';

import ReactBootstrap from 'react-bootstrap';

var Rater = require('./Rater.jsx').Rater;

class FeedbackComp extends React.Component{

constructor(){

super();

this.state={

latestFeedback:"",

rating:'5'

};

this.feedbackSubmitHandler=this.feedbackSubmitHandler.bind(this);

this.handleFeedbackChange=this.handleFeedbackChange.bind(this);

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

}

handleFeedbackChange(e){

this.setState({latestFeedback:e.target.value})

}

handleClick(rating){

this.setState({rating:rating});

}

feedbackSubmitHandler(e){

this.props.onFeedback(this.state.rating,this.state.latestFeedback);

this.setState({latestFeedback:"",rating:"5"});

}

render(){

var items=[];//array of existing feedback

this.props.feedbacks.forEach(function(fb){

items.push(<div key={fb.pdtCode}><a href="#"><h4>{fb.user}</h4></a>

<Rater value={fb.rating} maxlength="6" />&nbsp;&nbsp;

<span style={{"fontSize":"9px"}}>{fb.rating}/5</span><br/>

<div>{fb.body}</div>

<h6 style={{"fontStyle":"italic","color":"lightgrey"}}>-{fb.updatedAt.substr(0,10)}</h6></div>)

});

return(

<div>

{items}

{this.props.status ? null :<Form horizontal className={""}>

<div className={"form-group"}>

<textarea className={"form-control"} rows="5"

cols="12" value={this.state.latestFeedback} name="username"

onChange={this.handleFeedbackChange} ref="username" ></textarea>

</div>

<div><Rater value={this.state.rating} maxlength="6" onSelected={this.handleClick}/>&nbsp;&nbsp;

<span style={{"fontSize":"9px"}}>{this.state.rating}/5</span></div><br/>

<ReactBootstrap.FormGroup>

<ReactBootstrap.Col sm={8}>

<ReactBootstrap.Button bsStyle="primary" onClick={this.feedbackSubmitHandler} >

Submit Feedback

</ReactBootstrap.Button>

</ReactBootstrap.Col>

</ReactBootstrap.FormGroup>

</ReactBootstrap.Form>}

</div>

);

}

}

export default FeedbackComp;

React JSX

Copy

This feedbackcomponent is used inside ProductDetails component, so when we call this.props on we call this.props.onFeedback(this.state.rating, this.state.latestFeedback), we are passing parameters to a function written inside the parent components i.e. to ProductDetails component

**ProductDetails.jsx:**

import React from 'react';

import Product from './Product.jsx';

import FeedbackComp from './FeedbackComp.jsx';

import AppStore from './../Stores/ApplicationStore';

import Action from './../Actions/Action';

var productId;

function getProductDetailsState(){

let params={"pdtId":productId};

var retData= AppStore.FetchProductDetails(params);

return {

productDetails:retData.pdtDetails,

feedbackDetails:retData.fbDetails.feedbackDetails,

productId:retData.fbDetails.pdtCode

};

}

class ProductDetails extends React.Component{

constructor(props){

super(props);

this.handleSubmitFeedback=this.handleSubmitFeedback.bind(this);

this.\_onChange=this.\_onChange.bind(this);

this.state={

productDetails:{},

feedbackDetails:[],

pdtId:""

};

}

componentWillMount(){

productId=this.props.params.id;

var stateObj=getProductDetailsState();

this.setState({productDetails:stateObj.productDetails, feedbackDetails:stateObj.feedbackDetails,

pdtId:stateObj.productId});

}

handleSubmitFeedback(rating,feedback){

let fbData= {'feedback':feedback,

'rating':rating,

'productId':this.state.pdtId};

Action.SubmitFeedback(fbData);

}

componentDidMount(){

AppStore.addChangeListener(this.\_onChange);

//setInterval(this.\_onChange,1000);

}

componentWillUnmount(){

AppStore.removeChangeListener(this.\_onChange);

}

render() {

return(

<div className={"container-fluid"}>

<div className={"row"} style={{"maxWidth":"100%"}}>

<div className={"col-sm-3"}>

<Product status={this.state.productDetails.isDiscontinued} pid={this.state.productDetails.\_id}

price={this.state.productDetails.pdtPrice} name={this.state.productDetails.pdtName}

desc={this.state.productDetails.pdtDescription} img={this.state.productDetails.pdtImg}

rating={this.state.productDetails.avgFeedback}/>

</div>

<div className={"col-sm-4"}>

{this.state.productDetails.isDiscontinued}

<FeedbackComp status={this.state.productDetails.isDiscontinued} feedbacks={this.state.feedbackDetails}

pdtCode={this.state.pdtId} onFeedback={this.handleSubmitFeedback}/>

</div>

</div>

</div>

)}

\_onChange(){

let stateObj=getProductDetailsState();

this.setState({productDetails:stateObj.productDetails, feedbackDetails:stateObj.feedbackDetails,

pdtId:stateObj.productId});

}

}

export default ProductDetails;

React JSX

Copy

**IssueContstants.js:**

module.exports={

SUBMIT\_FEEDBACK:"SUBMIT\_FEEDBACK"

};

JavaScript

Copy

**Action.jsx:**

var Dispatcher=require('./../Dispatcher/dispatcher.js').AppDispatcher;

var IssueConstants=require('./../Constants/IssueConstants');

module.exports={

SubmitFeedback:function(data){

Dispatcher.handleAction({

actionType:IssueConstants.SUBMIT\_FEEDBACK,

data:data

});

}

}

**Dispatcher.js:**

var Dispatcher=require('flux').Dispatcher;

var AppDispatcher= new Dispatcher();

AppDispatcher.handleAction=function(action){

this.dispatch({

source:'DISPATCH',

action:action

});

}

module.exports.AppDispatcher=AppDispatcher;

JavaScript

Copy

After the Dispatcher broadcasts the action, all stores in the application receive the action via the callbacks in the registry.

ApplicationStore.js(contains the code required for ProductDetails component and feedback component):

var Dispatcher=require('./../Dispatcher/dispatcher.js').AppDispatcher;

var \_=require('underscore');

var EventEmitter=require('events').EventEmitter;

var AjaxHelper=require('./../Helper/AjaxHelper.js');

var Router =require('react-router');

var IssueConstants=require('./../Constants/IssueConstants');

var CHANGE\_EVENT="change";

var \_feedbackDetails=[];

function AddFeedback(feedbackDetails){

AjaxHelper.submitFeedback('/submitFeedback',feedbackDetails);

}

var ApplicationStore=\_.extend({},EventEmitter.prototype,{

FetchProductDetails:function(pdtId){

\_productDetails=AjaxHelper.pdtDetails("/productDetails",pdtId);

return \_productDetails;

},

emitChange:function(){

this.emit(CHANGE\_EVENT);

},

addChangeListener: function(callback) {

this.on(CHANGE\_EVENT, callback);

},

removeChangeListener: function(callback) {

this.removeListener(CHANGE\_EVENT, callback);

}

});

Dispatcher.register(function(payload){

var action=payload.action;

switch(action.actionType){

case IssueConstants.SUBMIT\_FEEDBACK:

AddFeedback(payload.action.data);

break;

default:

return true;

}

ApplicationStore.emitChange();

return true;

});

module.exports=ApplicationStore;

JavaScript

Copy

From the Stores call to database is made form another file named AjaxHelper which handles all the asynchronous calls.

**AjaxHelper.js:**

var $=require('jquery');

function PostRequest(url,data){

var data;

$.ajax({

url:url,

type:'POST',

data:data,

async:false,

success:function(retObj){

data=retObj;

},

error:function(err){

data=err;

}

});

return data;

}

module.exports={

pdtDetails:PostRequest,

submitFeedback:PostRequest

}

Exercise 6 : Defect tracker application

**Time Limit:**30 Minutes

Problem Statement

Implement Flux architecture for all the functionalities in the application. Some functionalities are listed below:

1. Login
2. Filtering the details
3. Add defect
4. Closing the defects