**React Js**

About ReactJS

ReactJS is a JavaScript library used for building User Interfaces. It is open sourced to the world by Facebook and Instagram .

Developers choose ReactJS to build large web applications which use data that can change over time without reloading page.

It aims primarily to provide speed, simplicity and scalability.

In this course, we'll discuss about creating interactive rich web application with reusable multiple components, handling huge data in an efficient manner and maintaining data flow in single direction which helps developers to create high performance web application.

**What is ReactJS?**

ReactJS is a JavaScript library for creating user interfaces, making development of UI components easy and modular.  
ReactJS was created by Jordan Walke, a software engineer at Facebook and open sourced to the world by Facebook and Instagram.

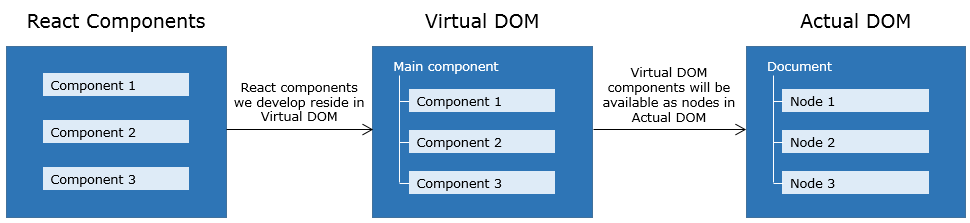
**What is React component?**

React components are the fundamental unit of any React application. They are capable of encapsulating data and view as a single unit. These components can work in conjunction with each other.

Components make code reusable, testing easy, and can take care of separation of concerns.

Capability of creating own components is a major productivity boost to any web application development.

Here, in ReactJS, components resides in virtual DOM and these components will be available as nodes in actual DOM.



Before discussing creation of components, let’s discuss what is Virtual DOM and why React uses Virtual DOM.

Virtual DOM is an abstraction of actual DOM, where components are the nodes. We can programmatically modify virtual DOM by updating components. These updates are internally handled by React and in turn updated in actual DOM.

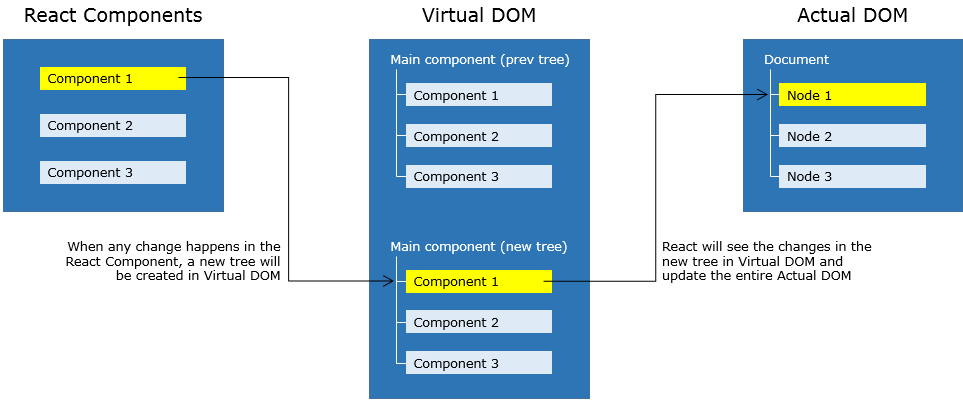
As we all know that, DOM manipulation is expensive, because it requires traversal through entire DOM tree to find the element to be updated. If these updates are very frequent, this leads to a poor performance of an application.

Different frameworks handle above scenario in different ways like dirty checking, data-binding etc.

React, rather than updating DOM directly, builds an abstract version of it called Virtual DOM.

**React component : Virtual DOM**

Below diagram illustrates the working of virtual DOM:



* Whenever any updates happens in the application, the virtual DOM gets modified. React computes the difference between the previous virtual tree and the current virtual tree
* Based on these differences React will figure out how to make updates to the actual DOM efficiently
* React does all the computations in its abstracted DOM and updates the DOM tree accordingly
* Virtual DOM enhances performance and efficiency by minimizing expensive updates in the actual DOM
* Hence React is said to be a great performer because it manages a Virtual DOM

**How to create a React component?**

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({ });

If we want to create an element within the component we can use **React.createElement** method.

Example:

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

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;

The AppComp component created in App.js file is exported so that it can be used in other files.

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;

* 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

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);

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'));

'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>

**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>

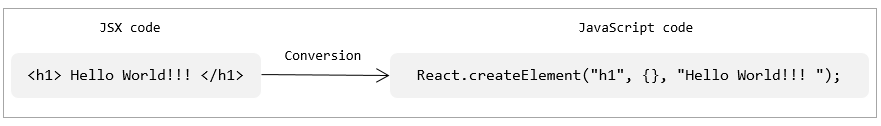
}

}

export default AppComp;

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:



` **JSX in component**

Highlights:

 Creating elements in JSX

 How JSX code will be converted to JavaScript

Demosteps:

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;

**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>

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>

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 };

}

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

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);

}

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

  start() {

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

}

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

};

}

}

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;

State : Key points

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**.

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;

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

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 nodes

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 />

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) { })

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

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;

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>

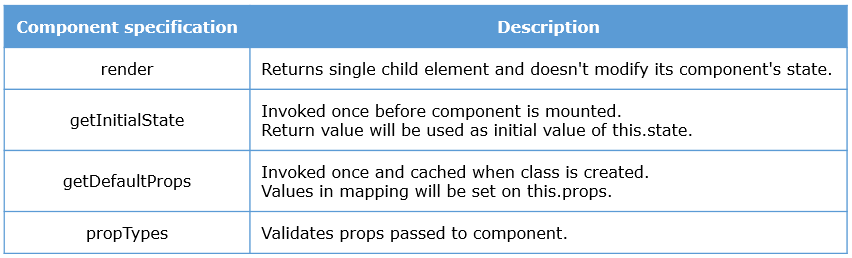
                    <p> Paragraph element </p>

                    <span> Span element </span>

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

Component specification

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



Component specification

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;

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.

It sets default values for the properties of a component.

**Syntax:**

getDefaultProps : function () {

return {

element1 : "Hello",

element2 : "World",

};

}

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",

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

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;

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'));

* 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;

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'));

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"

}

}

**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

}

Component specification

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;

## 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;

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

**Lifecycle methods**

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>

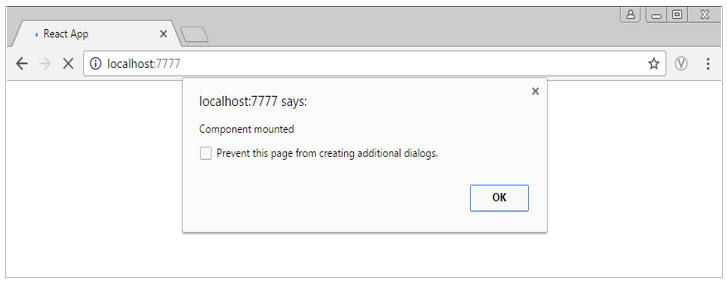
<h2>Seconds Elapsed: {this.state.secondsElapsed}</h2>

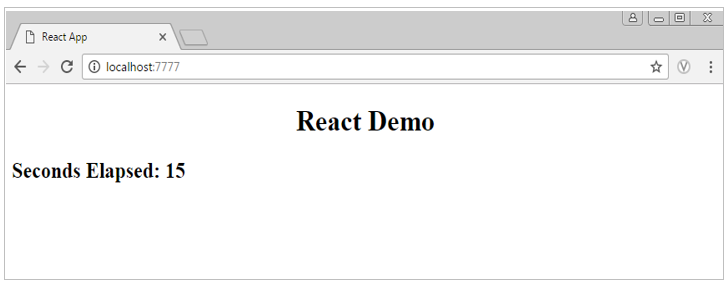
</div>);

}

}

export default Timer;





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

Lifecycle methods

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;

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);

3. Create index.html as shown below:

<!DOCTYPE html>

<html>

<head>

<title>React App</title>

</head>

<body>

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

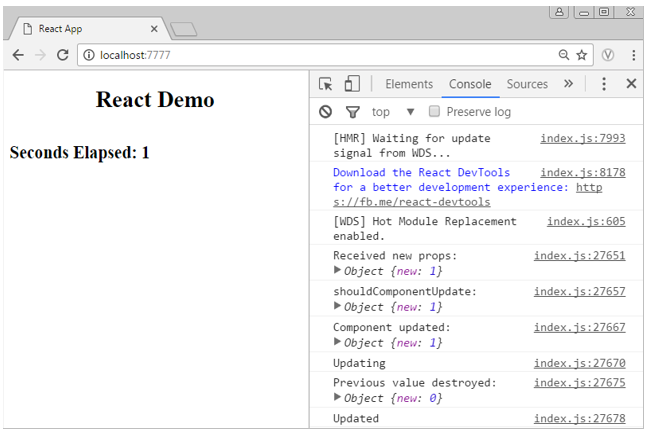
<div id = "app"> </div>

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

</body>

</html>

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

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

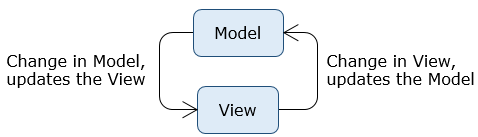
)});

}

## Forms

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.

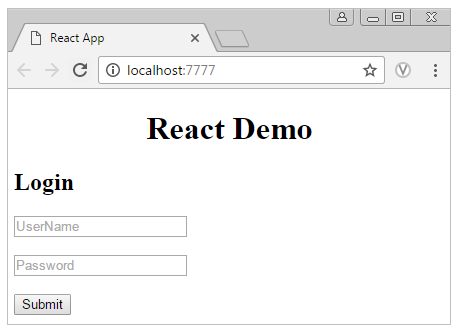
React uses unidirectional data flow pattern where in changes will be done only from component to view but not vice versa.

This would prevent unpredictable data flow and easy to debug.

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

Forms

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



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;

**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} />

Forms

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/>

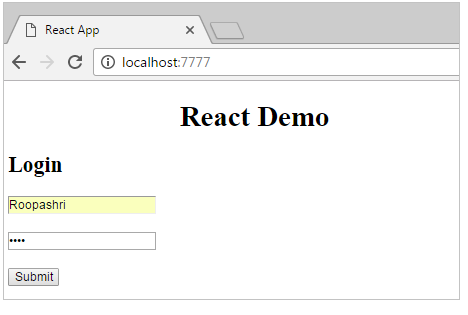
<input type = "submit" onClick = {this.handleLogin} />

</form>

}

}

export default Login;



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 = "" />

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

this.refs.userName.value

React refs

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/>

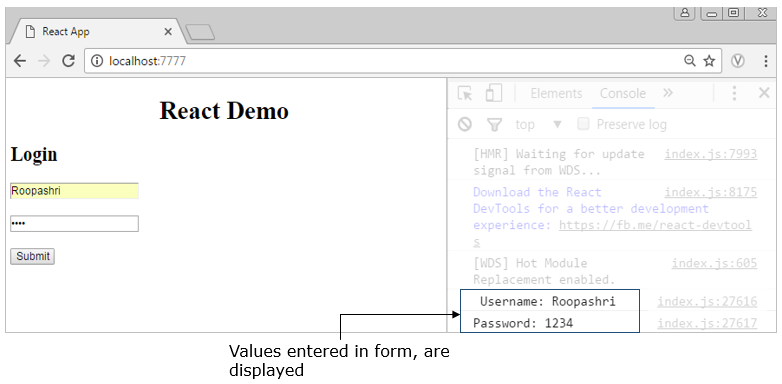
<input type = "submit" onClick = {this.handleLogin} />

</form>

}

}

export default Login;



## Router

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';

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>

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

Example:

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

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

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'));

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');

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'));

**Route parameters:**

Parameters passed along with URL are called route parameters.

Example:

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

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}/>

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

**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

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'));

## 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.

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>

);

}

});

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);

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);

});