**UNIT V:**

**Introduction and History**

React is a JavaScript framework. React was originally created by engineers at Facebook to solve the challenges involved when developing complex user interfaces with datasets that change over time. Applications based on traditional client-side MVC approach consist of two-way data binding along with rendering template which is slow and not scalable. React has a rich set of features that make composing single-page applications or user interfaces easy and convenient. Features such as one-way binding, Virtual DOM, JSX and so on enabled this.

React is a different concept when it comes to web development in general. It is a shift from generally accepted workflows and best practices. React was a created to fill a specific need for a specific set of technological challenges faced by Facebook, which is of managing very complex user interfaces. React was built to deal with displaying data in a large-scale user interface with data that changes over time rapidly.

In web app development, much of the responsibility of UI can be offloaded to the browser and HTML, CSS and JS. These types of applications are commonly referred to as *single page applications*, where the common request/response to the server is limited. But problems may arise: while binding the data properly, changing requirements, interconnected user interfaces and interactions and so on. In the MVC architecture, the view (presentation, what is seen in the browser) has to be updated whenever either the model (data) changes or because of the user interaction (a request). This becomes complicated when the amount of data that must be dealt with is huge and changes rapidly. New models and views add more complexity and makes tracking of the components difficult. One solution is to run the startup code again when the data or the state changes, but likely to lead to performance problems. These are the problems React attempts to solve.

The model handles the state of the application and sends state-changing events to the view. The view is the user-facing look and interaction interface to the end user. The view can send events to the controller, and in some cases to the model. The controller is the main dispatcher of events, which can be sent to the model, to update state, and the view to update the presentation. React is not a full-scale JS framework (unlike Backbone, AngularJS, Ember, Dojo, and other MVC frameworks). React is (in its simple form) just the view of MVC. React is a way to describe the user interface of an application and a mechanism to change that over time as data changes. React is made with declarative components that describe an interface. React uses no observable data binding when building an application. React is also easy to manipulate, because we can take the components we create and combine them to make custom components that work as we expect every time because it can scale. React can scale better than other frameworks because of the principles that drove it from its creation. When creating React interfaces, we structure them in such a way that they are built out of multiple components.

**React components and terminology**

**Components:**

Components are the core of React and the view to the application.

Components are typically created by utilizing a call to React.createClass() as:

var MyClass = React.createClass({

render: function() {

return (

<div>hello world</div>

);

}

});

Or by using ES6 classes as:

class MyClass extends React.Component {

render() {

return <div>hello world</div>;

}

}

**Virtual DOM:**

React creates a Virtual DOM in memory. Instead of manipulating the browsers DOM directly, it creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM. React finds out what changes have been made and changes only what needs to be changed.

Facebook changed the framework from that of a set of DOM mutations each time the data changed, to what it called reconciliation. Facebook did this by creating a virtual DOM that they use each time they encounter an update to calculate the minimum set of changes needed to update the application’s actual DOM.

**JSX (Javascript XML)**

JSX is the transform layer that transforms XML syntax for writing React components into the syntax that React uses to render elements in JavaScript. This is not a required element of React, but it is most definitely highly regarded and can make building applications much smoother. The syntax can accept not only custom React classes, but also plain HTML tags. It translates the tags into the appropriate React elements.

// JSX version

React.render(

<div>

<h1>Header</h1>

</div>

);

// This would translate to

React.render(

React.createElement('div', null, React.createElement('h1', null, 'Header'));

);

JSX allows to write HTML in React. It makes easier to write and add HTML to React. JSX allows to write HTML elements in JavaScript and place them in the DOM without any createElement() or appendChild() methods.

**Properties**

Properties are commonly referenced in React as this.props, because that is the most frequent way that properties are accessed. Properties are the set of options that a component holds. Props are arguments passed into React components. Props are passed to components through HTML attributes.

this.props is a plain JavaScript object in React.

These properties will not change throughout the lifecycle of the component, so we should not treat them as if they were not immutable. If we want to alter something on the component, we will be altering its state and we should utilize the state object.

React props are like function arguments in JS and attributes in HTML. To send props into a component, use the same syntax as HTML attributes:

const myelement = <Car brand="Ford"> (adding a brand attribute to the Car element)

the component receives the argument as a props object:

class Car extends React.Component{

render() {

return <h2>I am a {this.props.brand}</h2>

(use the brand attribute in the component)

Props also are the means to pass data from one component to another as parameters.

**State**

State is set on each component as it is initialized and is also altered throughout

the lifecycle of a component. The state should not be accessed from outside of the component, unless a parent component is adding or setting the initial state of the component. In general though, we should try to author the components with as little state objects as possible. This is because as we add state, the complexity of components increases because the React component will not change over time depending on the state. If it can be avoided, it is acceptable to not have any state in a component at all.

React components has a built-in state object. The state object is where we store property values that belongs to the component. When the state object changes, the component re-renders.

class Car extends React.Component{

constructor(props){

super(props);

this.state = {brand: "Ford"};

}

render() { return (<div> <h1> my car </h1> </div>);

}

}

**Flux**

Flux is a project that is closely related to React. Flux is Facebook's application architecture for how to get data to interact with React components in an organized and meaningful way. Flux is not a Model-View-Controller architecture because those utilize a bi-directional data flow. Flux is essential to React because it helps to promote the use of React components in the way they are intended. Flux does this by creating a one-directional data flow. Data flows through three main portions of the Flux architecture: the dispatcher, the stores, and finally the React views.

**Mixins**

A mixin in the component specification is an array. A mixin can share the lifecycle events of your component and you can be assured that the functionality will execute during the proper time during the component's lifecycle. An example mixin is a timer control that merges the lifecycle events of a SetIntervalMixin with the main component called TickTock.

var SetIntervalMixin = {

componentWillMount: function() {

this.intervals = [];

},

setInterval: function() {

this.intervals.push(setInterval.apply(null, arguments));

},

componentWillUnmount: function() {

this.intervals.map(clearInterval);

}

};

var TickTock = React.createClass({

mixins: [SetIntervalMixin], // Use the mixin

getInitialState: function() {

return {seconds: 0};

},

componentDidMount: function() {

this.setInterval(this.tick, 1000); // Call a method on the mixin

},

tick: function() {

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

},

render: function() {

return (<p>React has been running for {this.state.seconds} seconds.</p>);

}

});

**Lifecycle of components**

Each component in React has a lifecycle which we can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

**Mounting**

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

constructor()

getDerivedStateFromProps()

render()

componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if we define them.

The constructor() method is called before anything else, when the component is initiated, and is the place to set up the initial state and other initial values. This method is called with props as the argument, and we should always start by calling the super(props) (this will initiate the parent's constructor). This method is called by React every time a component is made.

The getDerivedStateFromProps() method is called right before rendering the elements in the DOM. This is the place to set the state object based on the initial props. It takes state as an argument, and returns an object with changes to the state.

The render() method is required and is the method that actually outputs HTML to the DOM.

The componentDidMount() method is called after the component is rendered. Here we can run the statements that requires that the component is already placed in the DOM.

**Updating**

The next phase in the lifecycle is when a component is updated. A component is updated whenever there is a change in the component's state or props.

React has five built-in methods that gets called, in this order, when a component is updated:

getDerivedStateFromProps()

shouldComponentUpdate()

render()

getSnapshotBeforeUpdate()

componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if they are defined.

The getDerivedStateFromProps() method is first called when a component gets updated. This is the place to set the state object based on the initial props.

In the shouldComponentUpdate() method we can return a Boolean value that specifies whether React should continue with the rendering or not. The default value is true.

The render() method is called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.

In the getSnapshotBeforeUpdate() method we have access to the props and state before the update, meaning that even after the update, we can check what the values were before the update. If this method is present, we should also include the componentDidUpdate() method, otherwise it results in an error.

The componentDidUpdate() method is called after the component is updated in the DOM.

**Unmounting**

The next phase in the lifecycle is when a component is removed from the DOM.

React has only one built-in method that gets called when a component is unmounted:

componentWillUnmount()

This method is called when the component is about to be removed from the DOM.

**React**

The React object contains several **methods** and **properties**. A few of them are explained below.

**1**.

The **createClass** method will create a new component class in React. It can be created with an object, which must have a render() function.

The basic implementation of createClass is as follows, where specification is the object that will contain the render() method.

React.createClass( specification );

var MyComponent = React.createClass({

render: function() {

return (<div>{this.props.name}</div>);

}

});

React.render(<MyComponent name="aname" />, document.getElementById('container'));

Here, a simple component is created using createClass. This component creates a div element and passes a name property to that div to be rendered.

It is also possible to create components ES6 classes by inheriting from **React**.**Component**.

class MyComponent extends React.Component {

render() {

return (<div>{this.props.name}</div>);

}

};

React.render(<MyComponent name="aname" />, document.getElementById('container'));

**2.**

**React.Children.map** is a function within **React.Children**. It's an object that holds

several helper functions that allow to easily work with the components properties

this.props.children, which will perform a function on each of the immediate children contained and will return an object. The usage for React.Children.map is as follows

React.Children.map( children, myFunction [, context])

Here, the *children* argument is an object containing the children we want to target. The function, myFunction, is then applied to each of the children. The final argument, context, is optional and will set this on the mapping function.

var MyComponent = React.createClass({

render: function() {

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

console.log(child)

});

return (<div>{this.props.name}</div>);

}

});

React.render(<MyComponent name="aname" ><p key="first">a child</p><p key="2">another</p>

</MyComponent>, document.getElementById('container')

);

**3**.

**forEach** is another utility that can be used on this.props.children in React. It is similar to the React.Children.map function except that it does not return an object.

React.Children.forEach( children, myFunction [, context])

**4**.

The **count** method will return the number of components that are contained in

this.props.children. The function is executed as follows and accepts a single argument, an object.

React.Children.count( children );

**5**.

The **only** method will return the only child that is in this.props.children. It accepts children as a single object argument, just as the count function.

React.Children.only( children );

**6**.

The **createElement** method will generate a new ReactElement. It is created using at least one, and optionally up to three, arguments to the function — a string type, optionally an object props, and optionally children.

React.createElement( type, [props[, [children ...] );

**7**.

The **React.DOM** object provides utility functions that help to create DOM elements if you are not utilizing JSX. Instead of just writing <div>my div</div> in JSX, we could create the element by writing something like the following.

React.DOM.div(null, "my div");

**8**.

React.**createFactory** is a function that will call createElement on a given ReactElement type.

React.createFactory( type );

**9**.

**React.render** will take a ReactElement and render it to the DOM. React only knows

where to place the element by you providing it with a container, which is a DOM

element. Optionally, you can provide a callback function that is executed once the ReactElement has been rendered to the DOM node.

React.render( element, container [, callback ] );

**Discovering React components**

React components are the main building blocks when constructing a React application.

React components are created when we extend from the base React.Component class using ES6. Or, more traditionally, we can use the React.createClass method.

React components have their own API that contains several methods and helpers.

The base React.Component class is the friendly version of the component API. This means that it only implements the ES6 features, setState and forceUpdate. To use setState, we can either pass a function or a plain object to the setState method.

Optionally, we can add a callback function that will be executed once the state has been set.

When setState is called, we are really queuing the new object into the React update queue, which is the mechanism React uses to control when things are changed. Once the state is ready to alter, the new state object, or partial state, will be merged with the remainder of the components state. The actual update process is a handle in a batch update.

The other core API method present in the React.Component class’s prototype is a

function called forceUpdate. What forceUpdate does is it forces the component to update. It does so by once again utilizing React's queue system and then forcing the component to update. It does this by bypassing one portion of a component's

lifecycle, componentShouldUpdate().

**Building a React web application**

There are several ways that we can outline your application’s basic functionality

that will be transferred into a React application. One way is to wireframe a design.

First have an idea for the application. Identify the main areas of functionality that will represent the entire picture of this application. Say, for example, a way for the users to authenticate to the application. Specify a broad outline for the application.

Building on the outline, there are two ways to build the application.

One way is to create wireframes that follow the outline of the application. This gives a fresh start to identify where we can create components that fit into the new React application.

The other method is to base the structure on an existing application and its source in order to break down the functionality into components.

**Wireframes**

When creating wireframes, choose any tool that helps to express ideas in rough sketches that describe the ideas. What follows are the sections of the application that we decided to break apart into React components. At the root of all components is the app, and it will be a parent component to all of the following nested components. Then decompose each section into sub-sections as needed and thereby creating a hierarchy of components.

**Rewriting an existing application**

The first step is to identify where we can create components, or child components within the application. This requires examining the existing code. We can have each of the atomic or subatomic snippets of code that represents a singular code path to producing a user interface component. This is what is needed to split sections of functionality into their own components. New components are inserted where needed and integrate them into existing components to get the desired functionality.

**React Forms**

React uses forms to allow users to interact with the web pages (like in HTML).

Add a form with React like any other element.

class MyForm extends React.Component{

render(){

return (

<form>

<p>Enter a name</p>

<input type="text" />

</form>

);

}

}

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

Handling forms is about how to handle the data when it changes value or gets submitted. In React form data is usually handled by the components. When the data is handled by the components, all the data is stored in the component state. We can control changes by adding event handlers in the onChange attribute. We can control the submit action by adding an event handler in the onSubmit attribute. Various other elements can be added in React.

We can validate form input when the user is typing (using events) or can wait until the form gets submitted. Error messages can be displayed when the user types anything that is invalid.