React JS

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. It lets you compose complex UIs from small and isolated pieces of code called “components”.

React.js is an open-source JavaScript library that is used for building user interfaces specifically for single-page applications. It’s used for handling the view layer for web and mobile apps. React also allows us to create reusable UI components. React was first created by Jordan Walke, a software engineer working for Facebook. React first deployed on Facebook’s newsfeed in 2011 and on Instagram.com in 2012.

React allows developers to create large web applications that can change data, without reloading the page. The main purpose of React is to be fast, scalable, and simple. It works only on user interfaces in the application. This corresponds to the view in the MVC template. It can be used with a combination of other JavaScript libraries or frameworks, such as Angular JS in MVC.

React.js properties includes the following

- React.js is declarative
- React.js is simple
- React.js is component based
- React.js supports server side
- React.js is extensive
- React.js is fast
- React.js is easy to learn

Now, the main question arises in front of us is why one should use React. There are so many open-source platforms for making the front-end web application development easier, like Angular. Let us take a quick look on the benefits of React over other competitive technologies or frameworks. With the front-end world-changing on a daily basis, it’s hard to devote time to learning a new framework – especially when that framework could ultimately become a dead end. So, if you're looking for the next best thing but you're feeling a little bit lost in the framework jungle, I suggest checking out React.

1. Simplicity

ReactJS is just simpler to grasp right away. The component-based approach, well-defined lifecycle, and use of just plain JavaScript make React very simple to learn, build a professional web (and mobile applications), and support it. React uses a special syntax called JSX which allows you to mix HTML with JavaScript. This is not a requirement; Developer can still write in plain JavaScript but JSX is much easier to use.

2. Easy to learn

Anyone with a basic previous knowledge in programming can easily understand React while Angular and Ember are referred to as ‘Domain-specific Language’, implying that it is difficult to learn them. To react, you just need basic knowledge of CSS and HTML.

3. Native Approach

React can be used to create mobile applications (React Native). And React is a diehard fan of reusability, meaning extensive code reusability is supported. So at the same time, we can make IOS, Android and Web applications.

4. Data Binding

React uses one-way data binding and an application architecture called Flux controls the flow of data to components through one control point – the dispatcher. It's easier to debug self-contained components of large ReactJS apps.

5. Performance

React does not offer any concept of a built-in container for dependency. You can use Browserify, Require JS, EcmaScript 6 modules which we can use via Babel, ReactJS-di to inject dependencies automatically.

6. Testability

ReactJS applications are super easy to test. React views can be treated as functions of the state, so we can manipulate with the state we pass to the ReactJS view and take a look at the output and triggered actions, events, functions, etc.

What is Babel?

Babel is a JavaScript compiler

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments.

Babel is a JavaScript compiler that includes the ability to compile JSX into regular JavaScript.

Here are the main things Babel can do for you:

- Transform syntax
- Polyfill features that are missing in your target environment (through a third-party polyfill such as core-js)
- Source code transformations (codemods)

ES2015 and beyond

Babel has support for the latest version of JavaScript through syntax transformers.

These plugins allow you to use new syntax, **right now** without waiting for browser support.

What is a web pack

At its core, **webpack** is a *static module bundler* for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph from one or more *entry points* and then combines every module your project needs into one or more *bundles*, which are static assets to serve your content from.

JSX expression

JSX is a JavaScript Extension Syntax used in React to easily write HTML and JavaScript together.

```
import logo from './logo.svg';
import './App.css';
var mystyle={
  backgroundColor:"green",
  width:100,
  height:100
}
const element=<div style={mystyle}>
<p>
  Hello World!!
</p>

</div>
function App() {
  return (
    element
  );
}

export default App;
```

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:
By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

Creating React without JSX

```
import logo from './logo.svg';
import './App.css';
import  React from 'react'

function App() {
  return (
    React.createElement("P",null,"Hello World")
  );
}

export default App;
```

Props

Props are arguments passed into React components .They are passed to components via HTML attributes.

React Props are like function arguments in JavaScript *and* attributes in HTML.

```
import logo from './logo.svg';
import './App.css';
import  React from 'react'
  var user={
    name:"peter",
    email:"peter@gmail.com"
  }
function Welcome(props)
{
  return(<div>
    <h3>Hello , {props.name}</h3>
   <h3>Mail at {props.email}</h3>
  </div>)
}

function App() {
  return (
   <Welcome name={user.name} email={user.email}/>
  );
}

export default App;
```

React props are readonly ,They can be however changed using state of an object

Writing Events

```
function App() {
  return (
     <div>
       <input type='button'value="click" onClick={()=>{
         console.log("hello world")
       }}/>
     </div>
  );
}
```

Creating Components

```
import logo from './logo.svg';
import './App.css';
import  React, { Component } from 'react'

export default class App extends Component
{
  render()
  {
    return(<div>Hello World</div>)
  }
}
```

Creating a form using a state variable

React components has a built-in `state` object.

The `state` object is where you store property values that belongs to the component.

When the `state` object changes, the component re-renders.

```
import logo from './logo.svg';
import './App.css';
import  React, { Component } from 'react'

export default class App extends Component
{
  constructor()

{
  super()
 this.state={
   username:"",
   password:"",
   email:""
 }
this.setUserName=this.setUserName.bind(this)
this.renderForm=this.renderForm.bind(this)
}
setUserName(e)
{
  this.setState({username:e.target.value})
  console.log(this.state.username)
}
  renderForm()
  {
      const frm=(<form>
            Enter User Name
              <input type='text'  onChange={
                this.setUserName
              }/>
              Enter Password

              <input type='password' name='password'/>
              Enter Email
              <input type='button' value='Login' onClick={()=>{
                console.log(this.state.username)
              }}/>

      </form>);
      return frm;
  }
  render()
  {
    return(<this.renderForm/>)
  }
}
```

Working with an Array and creating a search box

```
import { Component } from "react";
```

```
export default class SearchPage extends Component
{
    constructor(props)
    {
      super(props)
      this.state={
        fruitname:[
          'Apple',
          'Appricott',
          'Avacado',
          'Banana',
          'BlueBery',
          'Mango',
          'Strawberry',
          'PineApple',
          'Pappaya',
          'Peach',
          'Orange',
          'DragonFruit',
          'Cherry'
        ],
        searchterm:''

      }
      this.editsearchterm=this.editsearchterm.bind(this);

    }
  editsearchterm(e)
  {
    this.setState({searchterm:e.target.value})
  }

    render()
  {
    return (<div className="App">
    <h1>Search Box</h1>
     <div>
       <input type='text' value={this.state.searchterm}  onChange={this.editse
archterm} placeholder='Enter Fruit Name' />
     </div>
     <div>

     {

       this.state.fruitname.filter((x)=>{
                    return   x.search(this.state.searchterm)>=0
       }
       ).map((e)=><li>{e}</li>)
     }

     </div>
  </div>
    );
  }

}
```

Working with Components

- Nested Components
- Getting Value from a nested component
- Styling components

Nested Components

```
import MyBox from "./MessageBox";
import MyList from "./ListInfo";
import { Component } from "react";
export default class Mailer extends Component
{
    constructor()
    {
super()
    }
    render()
    {
        return (<div>
   <MyBox />
   <MyList myarr={["peter","sam","edgar","mathew"]}/>
        </div>)
    }
}
```

Getting Value from Child Components

```
class Parent extends React.Component {
  state = { message: "" }
  callbackFunction = (childData) => {
        this.setState({message: childData})
  },
  render() {
          return (
              <div>
                   <Child1 parentCallback = {this.callbackFunction}/>
                   <p> {this.state.message} </p>
              </div>
          );
  }
}

class Child1 extends React.Component{
  sendData = () => {
           this.props.parentCallback("Hey Popsie, How’s it going?");
      },
  render() {
  <div><input type=’button’ onClick={this.sendData}</div>
      }
};
```

Default Property Value

Component Life Cycle

Fetch API

Default Property Value

```
import { Component } from "react/cjs/react.production.min";

export default class Employee extends Component{
    constructor(props){
        super(props)
    }
    render()
    {
        return(<div>
            <p>Employee Name :{this.props.name}</p>
            <p>Employee Department:{this.props.department}</p>
            <p>Employee Branch:{this.props.branch}</p>
        </div>)
    }
}

Employee.defaultProps={
   department:"Administration",
   branch:"Head Office"
}
```

React Component Life Cycle

render()

The `render()` method is the only required method in a class component.

When called, it should examine `this.props` and `this.state` and render elements on page

`componentDidMount()`

`componentDidMount()` is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

`componentDidUpdate()`

`componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

`componentWillUnmount()`

`componentWillUnmount()` is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests,

REDUX

Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience

You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.

Redux is a pattern and library for managing and updating application state, using events called "actions". It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

Redux helps you manage "global" state - state that is needed across many parts of your application.

The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur. Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected.

Redux helps you deal with shared state management, but like any tool, it has tradeoffs. There are more concepts to learn, and more code to write. It also adds some indirection to your code, and asks you to follow certain restrictions. It's a trade-off between short term and long term productivity.

Redux is more useful when:

You have large amounts of application state that are needed in many places in the app

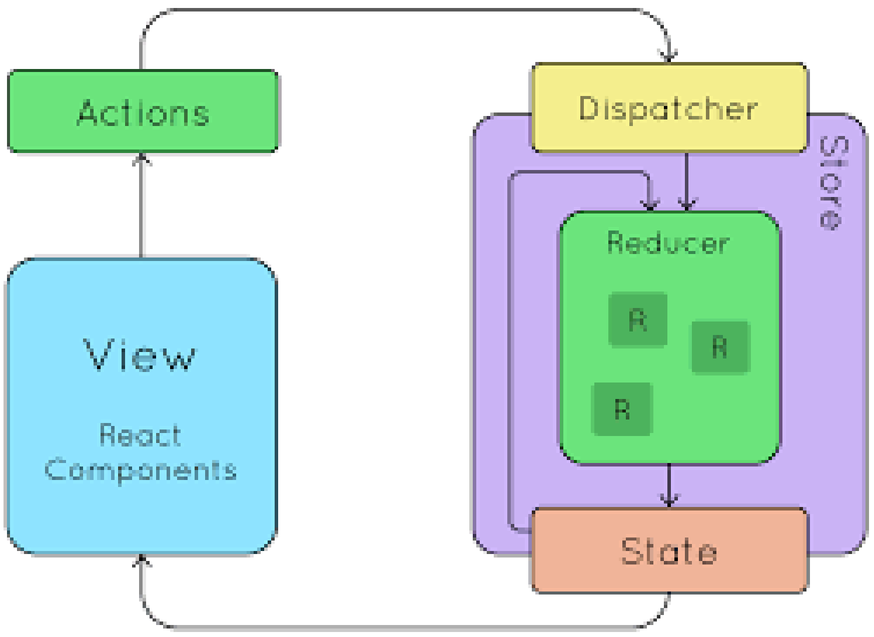
The app state is updated frequently over time

The logic to update that state may be complex

The app has a medium or large-sized codebase, and might be worked on by many people

To install redux libraries use

```
npm install react-redux
npm install redux
```


Actions
Dispatcher
Store
Reducer
R
R
R
View
React Components
State