**React**

1. **What is React?**

React is a front-end JavaScript library developed by Facebook in 2011.

* It follows the component based approach which helps in building reusable UI components.
* It is used for developing complex and interactive web and mobile UI.
* Even though it was open-sourced only in 2015, it has one of the largest communities supporting it.

2. **What are the features of React?**

* It uses the virtual DOM instead of the real DOM.
* It uses server-side rendering.
* It follows uni-directional data flow or data binding

3. **How is React different of other JS Frameworks?**

The interesting fact to note here is ReactJS is only a frontend library and not a whole framework, which deals with the View component of MVC (Model – View – Controller).

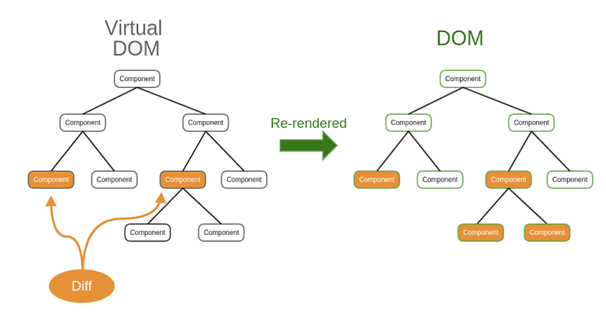
Also, in React, everything is a component. Consider one lego house as an entire application. Then compare each of the lego blocks to a component which acts as a building block. These blocks/ components are integrated together to build one bigger and dynamic application.

4. **What do you understand by Virtual DOM? Explain how its working.**

A virtual DOM is a lightweight JavaScript object which originally is just the copy of the real DOM. It is a node tree that lists the elements, their attributes and content as Objects and their properties. React’s render function creates a node tree out of the React components. It then updates this tree in response to the mutations in the data model which is caused by various actions done by the user or by the system.

This Virtual DOM works in three simple steps:

1. Whenever any underlying data changes, the entire UI is re-rendered in Virtual DOM representation.
2. Then the difference between the previous DOM representation and the new one is calculated.
3. Once the calculations are done, the real DOM will be updated with only the things that have actually changed.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--rorQuVGd--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/thepracticaldev.s3.amazonaws.com/i/d5amy5j4ly0ruq1inyet.png)

5. **What is an event in React?**

In React, events are the triggered reactions to specific actions like mouse hover, mouse click, key press, etc.

Handling these events are similar to handling events in DOM elements. But there are some syntactical differences like:

* Events are named using camel case instead of just using the lowercase.
* Events are passed as functions instead of strings.

The event argument contains a set of properties, which are specific to an event. Each event type contains its own properties and behavior which can be accessed via its event handler only.

6. **What is JSX?**

JSX is an XML/HTML-like syntax used by React that extends ECMAScript so that XML/HTML-like text can co-exist with JavaScript/React code. The syntax is intended to be used by preprocessors (i.e., transpilers like Babel) to transform HTML-like text found in JavaScript files into standard JavaScript objects that a JavaScript engine will parse.

7. **Why can’t browsers read JSX?**

Browsers can only read JavaScript objects but JSX in not a regular JavaScript object. Thus to enable a browser to read JSX, first, we need to transform JSX file into a JavaScript object using JSX transformers like Babel and then pass it to the browser.

8. **How different is React’s ES6 syntax when compared to ES5?**

Syntax has changed from ES5 to ES6 in following aspects:

* **require** vs **import**

// ES5

var React = require('react');

// ES6

import React from 'react';

export vs exports

* **export** vs **exports**

// ES5

module.exports = Component;

// ES6

export default Component;

* **component** and **function**

// ES5

var MyComponent = React.createClass({

render: function() {

return <h3>Hello Edureka!</h3>;

}

});

// ES6

class MyComponent extends React.Component {

render() {

return <h3>Hello Edureka!</h3>;

}

}

* **props**

// ES5

var App = React.createClass({

propTypes: { name: React.PropTypes.string },

render: function() {

return <h3>Hello, {this.props.name}!</h3>;

}

});

// ES6

class App extends React.Component {

render() {

return <h3>Hello, {this.props.name}!</h3>;

}

}

* **state**

// ES5

var App = React.createClass({

getInitialState: function() {

return { name: 'world' };

},

render: function() {

return <h3>Hello, {this.state.name}!</h3>;

}

});

// ES6

class App extends React.Component {

constructor() {

super();

this.state = { name: 'world' };

}

render() {

return <h3>Hello, {this.state.name}!</h3>;

}

}

9. **What are the different phases of React component’s lifecycle?**

There are three different phases of React component’s lifecycle:

* **Initial Rendering Phase:** This is the phase when the component is about to start its life journey and make its way to the DOM.
* **Updating Phase:** Once the component gets added to the DOM, it can potentially update and re-render only when a prop or state change occurs. That happens only in this phase.
* **Unmounting Phase:** This is the final phase of a component’s life cycle in which the component is destroyed and removed from the DOM.

10. **Explain the lifecycle methods of React components in detail**

Some of the most important lifecycle methods are:

* **componentWillMount()** – Executed just before rendering takes place both on the client as well as server-side.
* **componentDidMount()** – Executed on the client side only after the first render.
* **componentWillReceiveProps()** – Invoked as soon as the props are received from the parent class and before another render is called.
* **shouldComponentUpdate()** – Returns true or false value based on certain conditions. If you want your component to update, return true else return false. By default, it returns false.
* **componentWillUpdate()** – Called just before rendering takes place in the DOM.
* **componentDidUpdate()** – Called immediately after rendering takes place.
* **componentWillUnmount()** – Called after the component is unmounted from the DOM. It is used to clear up the memory spaces.

**------------------------------------------------------------------------------------------------------------------------**

in React 16.3.0, some lifecycle methods have been deprecated:

* **componentWillMount()**
* **componentWillReceiveProps()**
* **componentWillUpdate()**

They still can be used for now, but you would need to prefix it with UNSAFE\_, like UNSAFE\_componentWillMount, UNSAFE\_componentWillReceiveProps, and UNSAFE\_componentWillUpdate.

These are expected to be removed on React 17.

We got then some new methods to compensate for that:

* **getDerivedStateFromProps(props, state)** - Called after a component is instantiated as well as before it is re-rendered. It can return an object to update state, or null to indicate that the new props do not require any state updates.
* **getSnapshotBeforeUpdate(prevProps, prevState)** - Called right before mutations are made (e.g. before the DOM is updated). The return value for this lifecycle will be passed as the third parameter to componentDidUpdate. (This lifecycle isn’t often needed, but can be useful in cases like manually preserving scroll position during rerenders.)
* **What is server-side rendering (SSR)?**
* **Server-side rendering (SSR)**is when content on your web page is rendered on the server and not on your browser using JavaScript. For example, when you have a typical PHP or WordPress site, the page is loaded from content that is coming via HTTP which was rendered on the server and comes as fully rendered HTML. This is in contrast to a React app built with CRA, which just sends a .js file to the client and the clients’ browser JavaScript engine creates the markup after the .js file is loaded.
* Examples of traditional SSR languages/frameworks are PHP, Java, ASP .NET and Node.js.
* To be clear, this is how content was [rendered on early websites](https://medium.freecodecamp.org/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d), until the influx of client-side libraries. However, now, server-side rendered React apps use Node for the server which is a key difference to traditional server-rendered apps (we’ll see how later on in this post).
* **Reasons you should move to the server side**
* As I said before, server-side rendering initially means every page is rendered and loaded from the server. With the introduction of server-side (universal) React, however, things are slightly different.
* The initial page is rendered from the server, meaning the subsequent pages load directly from the client. So, you have the best of both worlds — the power of the initial server-side content plus the speedy subsequent loads which requests just the content that is needed for future requests.
* In addition to the above benefit, here are some other advantages you get from going SSR:
* **Performance**
* [Arunoda Susiripala](https://twitter.com/arunoda), an Engineer from Zeit, [talks about performance](https://youtu.be/ms2aOV06_qk) being the main reason for moving to server-side rendering. SSR means there is no need for loaders or spinners for the initial load.
* Faster load times lead to a better experience for the end user. This is one of the reasons [many large companies](https://medium.com/walmartlabs/using-electrode-to-improve-react-server-side-render-performance-by-up-to-70-e43f9494eb8b) are taking the SSR approach for their sites.
* **SEO**
* By now, you have probably heard that Google now crawls web apps built with JavaScript, you are better off having server-side rendered content ready for Google and other search engines to crawl your site.
* *Note that as of now, Google and Bing can index synchronous JavaScript applications — synchronous being the key word. If your app starts with a loading spinner, then fetches content via Ajax, the crawler will only wait a few seconds for loading to complete. This means if you have content fetched asynchronously on pages where SEO is important, SSR might be necessary.*
* *(*[*https://10up.github.io/Engineering-Best-Practices/react/*](https://10up.github.io/Engineering-Best-Practices/react/)*)*
* The advantage with SSR is that you get the benefits of a traditional website’s SEO since the entire page can now be crawled by bots.
* **Social sharing**
* The other benefit with SSR is that you get an elaborate snippet and featured image when sharing your web page’s content via social media. This will not be possible when you have just client-side rendered apps. For example, here is what a server-side rendered React app looks like when shared on LinkedIn:

### How to get started with an SSR app

Getting started without frameworks is possible, but I wouldn’t recommend this approach since there are many considerations and moving parts in a React SSR app. For example, you have to handle bundling, minification, hot reload (and more), all on your own.

However, if you want to go this route, I’d recommend reading [this tutorial](https://css-tricks.com/server-side-react-rendering/) by Roger Jin on CSS Tricks.

### React SSR frameworks

I would recommend picking up a framework if you want to render React on the server side. Here are some frameworks you can consider:

### Next.js

[Next.js](https://nextjs.org/) is a great framework with a great community around it. With Next.js, you don’t have to worry about bundling, minification or hot reloading, you get a lot of features out of the box. You are able to create pages as React components within files. You may be used to this if you worked with PHP. In addition to the community and support, there are many large companies using Next.js in production including [npm, Netflix and Auth0](https://hyperion.alpha.spectrum.chat/next-js/general/companies-sites-using-next-js~e425a8b6-c9cb-4cd1-90bb-740fb3bd7541?m=MTU1NDg5NDgzMDYzNg%3D%3D&msgsafter=MTU0NTc1MzAxNTU2Mg%3D%3D).

### Razzle

[Razzle](https://github.com/jaredpalmer/razzle) (a project by Jared Palmer) has been gaining a lot of traction lately.

“Razzle is a tool that abstracts all complex configuration needed for SSR into a single dependency — giving you the awesome developer experience of [create-react-app](https://github.com/facebookincubator/create-react-app), but then leaving the rest of your app’s architectural decisions about frameworks, routing, and data fetching up to you.” (<https://github.com/jaredpalmer/razzle>)

It’s easy to get started with Razzle and it uses React Router 4 by default, unlike Next.js which does not have a router out of the box.

### Alternatives

React is not a silver bullet. Perhaps your team is more familiar with Vue or another JavaScript framework. Maybe a static site will best suit your use case. If you don’t want to use React or if you would like to use a Static Site Generator, here are some alternatives.

### Nuxt.js

[Nuxt.js](https://nuxtjs.org/) is a server-side rendering framework for Vue.js and is popular in the Vue.js community. If you are looking for alternatives Next.js or Razzle in the Vue.js world, do give this a try.

### Gatsby

You would have seen almost all popular JavaScript developers talk about [Gatsby](https://gatsbyjs.org/). It is a React-based Static Site Generator that has won the hearts of many with its exceptional UX (User Experience) and DX (Developer Experience). To be precise, it doesn’t do SSR at run time. Rather, Gatsby does server-side rendering with Node.js at build time, where it creates static HTML, CSS, and JS when deploying the site. This leads to blazing fast load times and has further optimizations such as route-based code splitting and prefetching.

### An example app

I explored server rendered React apps a few months back and created an app with Next.js and hosted it on Now — a serverless platform. Both Next and Now are from a company called [Zeit](https://zeit.co/), who are doing a great job at educating developers about React and serverless technologies along with offering other fantastic products.

My app fetches data from a WooCommerce (a WordPress eCommerce plugin) REST API endpoint and displays it in a Next.js app. You can check out my app on [GitHub](https://github.com/m-muhsin/woocommerce-next) and take a look at the [demo here](https://woocommerce-next.now.sh/).

### Do you always need SSR?

The short answer would be **no**. Not all apps need server-side rendering, especially apps with a dashboard and authentication that will not need SEO or sharing via social media. Plus, the expertise for building a server-rendered React app is higher than an app initialized using create-react-app.

Most importantly, SSR React apps cost a lot more in terms of resources since you need to keep a Node server up and running. There are times you may be better off going the serverless route when you want to choose server-side rendering for your React applications.

### Conclusion

Client-side rendered React apps are great but having apps rendered on the server have noticeable benefits.

As we covered in this post, the benefits include:

1. Performance
2. Search engine visibility
3. Social sharing

I would highly encourage you to explore server-side rendering for your React apps and use it for your next product to see these benefits in action.