ReactJS

Table Of Content

Module 1 - Getting started

What is react?

- It is a UI library developed at Facebook
- Create interactive, stateful, and reusable components
- Example:
  - Instagram.com is written in React completely
- Uses virtual DOM
  - In short: React selectively renders subtree of DOM based on state changes
- Server side rendering is available
  - Because fake/virtual DOM can be rendered on server
- Makes use of Isomorphic JS
  - Same JS code can run on servers and clients...
  - Other egs which does this- `Rendr, Meteor & Derby`
- It is the `V in MVC`

Features

- Quick, responsive apps
- Uses virtual dom
- Does server side rendering
- One way data binding / Single-Way data flow
- Open source

Why React, Benefits, and Limitations

Why react / Benefits

- Simple, Easy to learn
- It is fast, scalable, and simple
- No need of separate template files JSX!
- It uses component based approach
    - Separation of concerns
- No need of direct DOM manipulation
- Increases app performance

Other Benefits

- Can be used on client and server side
- Better readability with JSX
- Easy to integrate with other frameworks
- Easy to write UI test cases

Limitations

- It is very rapidly evolving
    - Might get difficult to keep up

- It only covers `V of the MVP app`.
    - So you still need other tech/frameworks to complete the environment
    - Like Redux, GraphQL, Firebase, etc.
- Inline HTML and JSX.
    - Can be awkward and confusing for some developers
- Size of library is quite large

Module 2 - Basics of React

React JSX

- It is `JavascriptXML`
- It is used for templating
  - Basically to write HTML in React
- It lets you write HTML-ish tags in your javascript
- It's an extension to `ECMAScript`
  - Which looks like XML
- You can also use plain JS with React
  - You *don't HAVE* to use JSX
  - But JSX is recommended
  - JSX makes code more readable and maintainable
- Ultimately Reacts transforms JSX to JS
  - Performs optimization
- JXK is type safe
  - so errors are caught at compilation phase

```
// With JSX
const myelement = <h1>First JSX element!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));
```

```
// Without JSX

const myelement = React.createElement('h1', {}, 'no JSX!');

ReactDOM.render(myelement, document.getElementById('root'));
```

JSX - confusing parts

- JSX is not JS
  - So won't be handled by browsers directly
  - You need to include `React.createElement` so that React can understand it
  - We need babel to transpile it

```
// You get to write this JSX
const myDiv = <div>Hello World!</div>

// And Babel will rewrite it to be this:
const myDiv = React.createElement('div', null, 'Hello World')
```

- `whitespaces`
  - React removes spaces by default
  - You specifically give it using `{' '}`...
  - For adding margin padding

```
// JSX
const body = (
  <body>
    <span>Hello</span>
    <span>World</span>
  </body>
)

<!-- JSX - HTML Output -->
<body><span>Hello</span><span>World</span></body>
```

- Children props
  - They are special kind of props
    - You will learn about props more in the following sections
  - Whatever we put between tags is `children`
  - Received as `props.children`

```
<User age={56}>Brad</User>

// Same as
<User age={56} children="Brad" />
```

- There are some attribute name changes
  - NOTE: class becomes "className", for becomes "htmlFor"
- Cannot use `if-else` inside JSX
  - But you can use ternary!

Virtual DOM

- This is said to be one of the most important reasons why React app performances is very good
- You know that Document Object Model or DOM is the tree representation of the HTML page

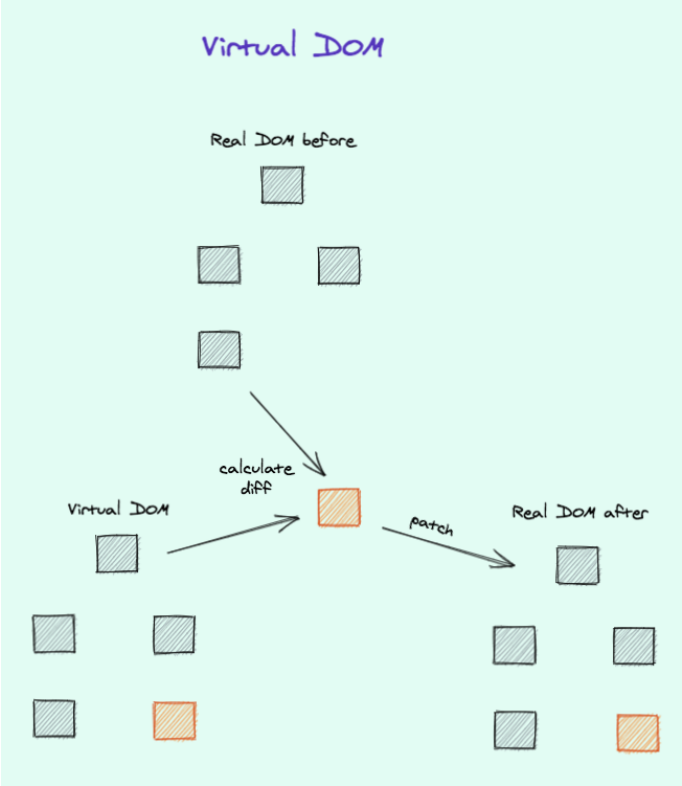
Cons of real DOM

- Updating DOM is a slow and expensive process
  - You have to traverse DOM to find a node and update it
- Updating in DOM is inefficient
  - Finding what needs to be updated is hard
- Updating DOM has cascading effects - things need to be recalculated

Enter -> Virtual DOM

- Virtual DOM is just a JavaScript object that represents the DOM nodes
- Updating JavaScript object is efficient and fast
- Virtual DOM is the blueprint of the DOM - the actual building
- React listens to the changes via observables to find out which components changed and need to be updated

Diffing



- Please check the illustration above
- When an update occurs in your React app - the entire Virtual DOM is recreated
- This happens super fast
- React then checks the difference between the previous virtual DOM and the new updated virtual DOM
- This process is called diffing
- It does not affect the react DOM yet
- React also calculates the minimum number of steps it would take to apply just the updates to the real DOM
- React then batch-updates all the changes and re-paints the DOM as the last step

Module 3 - Styling your components

Inline Styles

- Inline styling react component means using JavaScript object to style it

> Tip: Styles should live close to where they are used - near your component

```
class MyComponent extends React.Component {
  let styleObject = {
    color: "red",
    backgroundColor: "blue"
  }

  render() {
    return (
      <div>
        <h1 style={styleObject}>Hi</h1>
      </div>
    );
  }
}
```

```
// You can also skip defining objects to make it simpler
// But, I don't recommend it because it can quickly get out of hands and
```

```
difficult to maintain

render() {
  return (
    <div>
      <h1 style={{backgroundColor: "blue"}}>Hi</h1>
    </div>
  );
}
```

Module 4 - Advanced React

Conditional Rendering

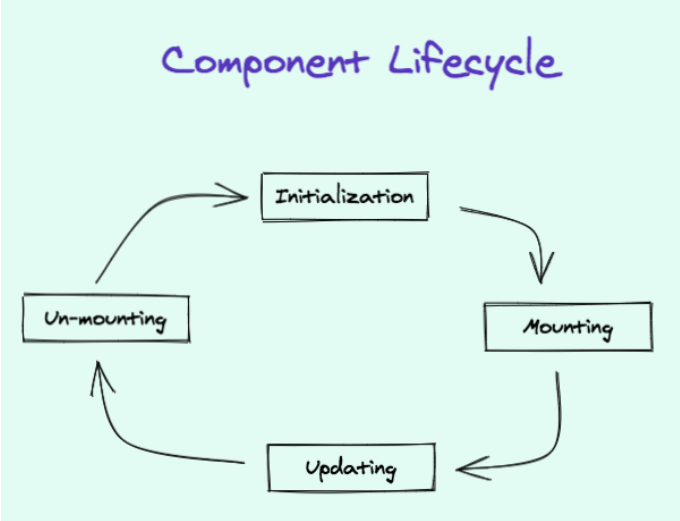
- React components lets you render conditionally using traditional conditional logic
- This is useful in situations for example: showing loading state if data is not yet retrieved else show the component details when data is retrieved
- You can use `if..else` or `ternary` or `short-circuit` operators to achieve this

```
// IF-ELSE

const Greeting = <div>Hello</div>;

// displayed conditionally
function SayGreeting() {
  if (loading) {
    return <div>Loading</div>;
  } else {
    return <Greeting />; // displays: Hello
  }
}
```

```
// Using ternary and short-circuit method

const Greeting = <div>Hello</div>;
const Loading = <div>Loading</div>;

function SayGreeting() {
  const isAuthenticated = checkAuth();

  return (
    <div>
      {/* if isAuth is true, show AuthLinks. If false, Login  */}
      {isAuthenticated ? <Greeting /> : <Loading />}

      {/* if isAuth is true, show Greeting. If false, nothing. */}
      {isAuthenticated && <Greeting />}
    </div>
  );
}
```

Component Lifecycle

- These lifecycle methods pertains to class-based React components
- 4 phases: `initialization, mounting, updating and unmounting` in that order



initialization

- This is where we define defaults and initial values for this.props and this.state
- Implementing `getDefaultProps()` and `getInitialState()`

mounting

- Occurs when component is being inserted into DOM
- NOTE: Child component is mounted before the parent component
- `componentWillMount()` and `componentDidMount()` methods are available in this phase
- Calling `this.setState()` within this method will not trigger a re-render
    - This notion can be used to your advantage
- This phase methods are called after `getInitialState()` and before `render()`

`componentWillMount()`

- This method is called before render
- Available on client and server side both
- Executed after constructor
- You can `setState` here based on the props
- This method runs only once
- Also, this is the only hook that runs on server rendering
- Parent component's `componentWillMount` runs before child's `componentWillMount`

`componentDidMount()`

- This method is executed *after* first render -> executed only on client side
- This is a great place to set up initial data
- Child component's `componentDidMount` runs before parent's `componentDidMount`
- It runs only once
- You can make **ajax calls** here
- You can also setup any subscriptions here
    - NOTE: You can unsubscribe in `componentWillUnmount`

`static getDerivedStateFromProps()`

- This method is called (or invoked) before the component is rendered to the DOM on initial mount
- It allows a component to update its internal state in response to a change in props
- **Remember:** this should be used sparingly as you can introduce subtle bugs into your application if you aren't sure of what you're doing.
- To update the state -> return object with new values
- Return null to make no updates

```
static getDerivedStateFromProps(props, state) {
  return {
    points: 200 // update state with this
  }
}
```

updating

- When component state and props are getting updated
- During this phase the component is already inserted into DOM

`componentWillReceiveProps()`

- This method runs before render
- You can `setState` in this method
- Remember: DON'T change props here

`shouldComponentUpdate()`

- Use this hook to decide whether or not to re-render component
  - `true` -> re-render
  - `false` -> do not re-render
- This hook is used for performance enhancements

```
shouldComponentUpdate(nextProps, nextState) {
  return this.state.value != nextState.value;
}
```

`getSnapshotBeforeUpdate()`

- This hook is executed right after the render method is called -
  - The `getSnapshotBeforeUpdate` lifecycle method is called next
- Handy when you want some DOM info or want to change DOM just after an update is made
  - Ex: Getting information about the scroll position

```
getSnapshotBeforeUpdate(prevProps, prevState) {

  // Capture the scroll position so we can adjust scroll later
  if (prevProps.list.length < this.props.list.length) {
    const list = this.listRef.current;
    return list.scrollHeight - list.scrollTop;
  }

  return null;
}
```

- Value queried from the DOM in `getSnapshotBeforeUpdate` will refer to the value just before the DOM is updated
  - Think of it as staged changes before actually pushing to the DOM
- Doesn't work on its own
  - It is meant to be used in conjunction with the `componentDidUpdate` lifecycle method.
- Example usage:
  - in chat application scroll down to the last chat

`componentWillUpdate()`

- It is similar to `componentWillMount`
- You can set variables based on state and props
- Remember: do not setState here -> you will go into an infinite loop

`componentDidUpdate()`

- This hook it has `prevProps` and `prevState` available
- This lifecycle method is invoked after the `getSnapshotBeforeUpdate` is invoked
    - Whatever value is returned from the `getSnapshotBeforeUpdate` lifecycle method is passed as the THIRD argument to the `componentDidUpdate` method.

```
componentDidUpdate(prevProps, prevState, snapshot) {
  if (condition) {
    this.setState({..})
  } else {
    // do something else or noop
  }
}
```

unmount

- This phase has only one method → `componentWillUnmount()`
- It is executed immediately BEFORE component is unmounted from DOM
- You can use to perform any cleanup needed
    - Ex: you can unsubscribe from any data subscriptions

```
componentWillUnmount(){
  this.unsubscribe();
}
```

React Hooks Basics

- Introduced in `React 16.8`
- It is a way to add `React.Component` features to functional components
    - Specifically you can add state and lifecycle hooks
- It offers a powerful and expressive new way to reuse functionality between components
- You can now use `state` in functional components
    - Not only the class components
- Real world examples:
    - Wrapper for firebase API
    - React based animation library
        - `react-spring`

Why React Hooks?

- Why do we want these?
    - JS `class` confuses humans and machines too!
- Hooks are like `mixins`
    - A way to share `stateful and side-effectful` functionality between components.
- It offers a great way to reuse stateful code across components
- It can now replace your render props or HOCs
- Developers can care LESS about the underline framework
    - Focus more on the business logic
    - No more nested and complex JSX (as needed in render props)

useState React Hook

- `useState` hook gives us local state in a function component
  - Or you can simply use `React.useState()`
- Just import `useState` from `react`
- It gives 2 values
  - 1st - value of the state
  - 2nd - function to update the state
- It takes in initial state data as a parameter in `useState()`

```
import React, { useState } from 'react';

function MyComponent() {

  // use array destructuring to declare state variable
  const [language] = useState('react');

  return <div>I love {language}</div>;
}
```

- Second value returned by `useState` hook is a setter function
- Setter function can be used to change the state of the component

```
function MyComponent() {

  // the setter function is always the second destructured value
  const [language, setLanguage] = useState('react');

  return (
    <div>
      <button onClick={() => setLanguage("javascript")}>
        I love JS
      </button>
      <p>I love {language}</p>
    </div>
  );
```

```
}
```

- You can create as many states as you'd want in your component

```
const [language, setLanguage] = React.useState('React');
const [job, setJob] = React.useState('Google');
```

- You can initiate your state variable with an object too

```
const [profile, setProfile] = React.useState({
  language: 'react',
  job: 'Google'
});
```

Module 6 - App performance optimization

Improve React app performance

- Measure performance using these tools
  - Chrome dev tools
    - Play with the `throttle` feature
    - Check out the performance timeline and flame charts
  - Chrome's Lighthouse tool
- Minimize unnecessary component re-renders
  - use `shouldComponentUpdate` where applicable
  - use `PureComponent`
  - use `Rect.memo` for functional components
    - along with the `useMemo()` hook
  - use `React.lazy` if you are not doing server-side rendering
  - use `service worker` to cache files that are worth caching
  - use libraries like `react-snap` to pre-render components

- Example of `shouldComponentUpdate`
  - NOTE: It is encouraged to use function components over class components
  - With function components you can use `useMemo()` and `Rect.memo` hooks

```
// example of using shouldComponentUpdate to decide whether to re-render
component or not
// Re-render if returned true. No re-render if returned false
function shouldComponentUpdate(nextProps, nextState) {
    return nextProps.id !== this.props.id;
}
```

- React devtools
  - Install the chrome extension
  - These are super power tools to profile your application
  - You can also check why the component was updated
  - For this - install `why-did-you-update` package - https://github.com/maicki/why-did-you-update

```
// example from https://github.com/maicki/why-did-you-update

import React from 'react';

if (process.env.NODE_ENV !== 'production') {

  const {whyDidYouUpdate} = require('why-did-you-update');

  whyDidYouUpdate(React);
}

// NOTE: Be sure to disable this feature in your final production build
```

- Lazy load the components
  - Webpack optimizes your bundles
  - Webpack creates separate bundles for lazy loaded components
  - Multiple small bundles are good for performance

```
// TODOComponent.js
class TODOComponent extends Component{
    render() {
        return <div>TODOComponent</div>
    }
}

// Lazy load your component
const TODOComponent = React.lazy(()=>{import('./TODOComponent.js')})
```

```
function AppComponent() {
    return (<div>
      <TODOComponent />
    </div>)
}
```

- Cache things worth caching
    - use `service worker`
        - It runs in the background
    - Include specific functionality that requires heavy computation on a separate thread
        - To improve UX
        - This will unblock the main thread
- Server-side rendering
- Pre-render if you cannot SSR
    - It's a middle-ground between SSR and CSR
    - This usually involves generating HTML pages for every route during build time
    - And serving that to the user while a JavaScript bundle finishes compiling.
- Check the app performance on mobile
- Lists cause most of the performance problems
    - Long list renders cause problems
    - To fix
        - Implement virtualized lists -> like infinite scrolling
        - Or pagination