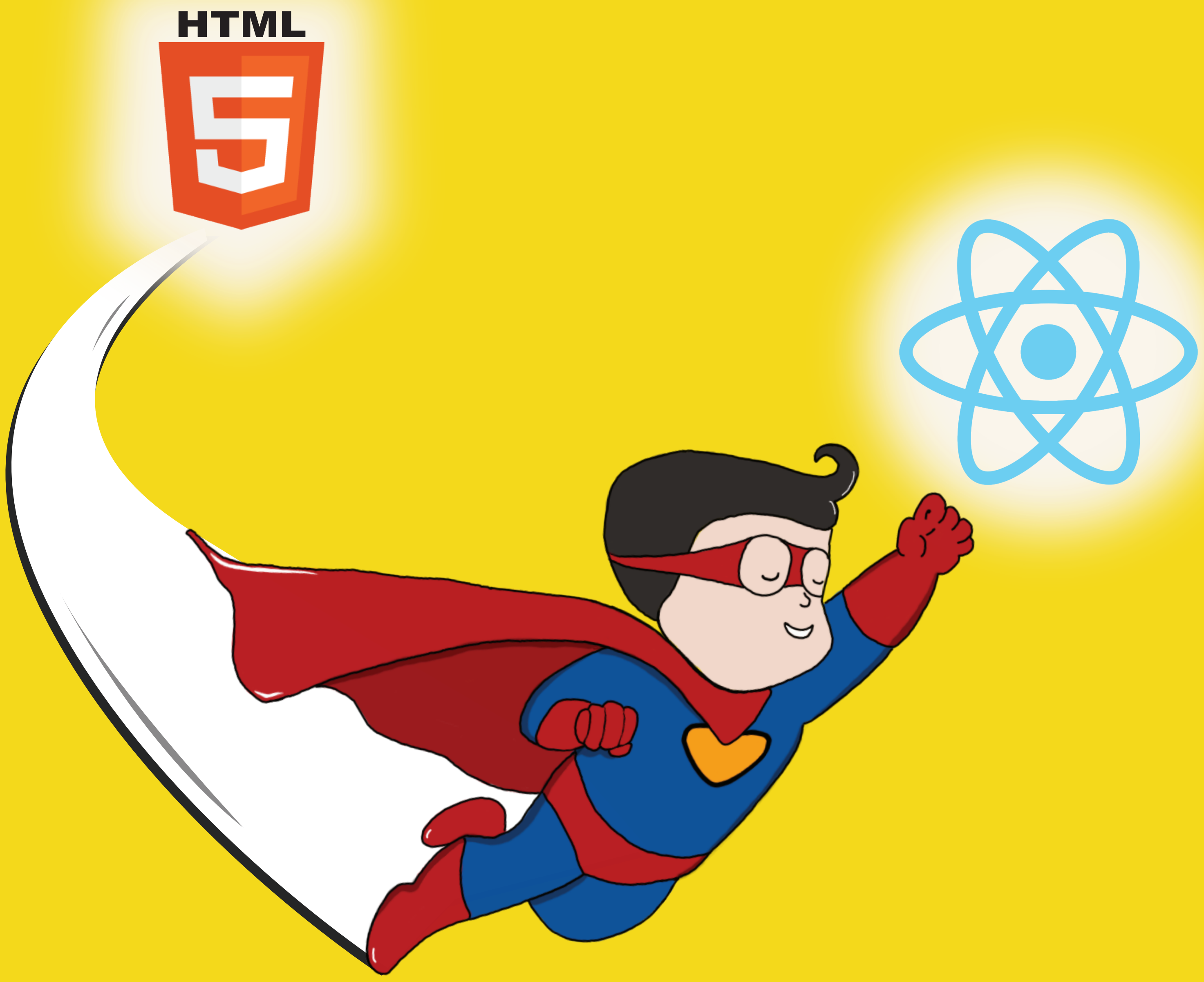


HTML TO REACT

The Ultimate Guide

NGNINJA ACADEMY

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

What Are SPAs

SPA

- Single-page application
- You load the app code JUST once
- The JavaScript framework (like React, AngularJS) intercepts the browser events
    - Instead of making a new request to the server that then returns a new document
    - Behaves more like a desktop application
- You don't need to refresh the page for updates
    - Your framework handles reload internally
- Some examples:
    - Gmail
    - Google Maps
    - Google Drive

Pros

- Feels much faster to the user
    - Instant feedback
    - No client-server communication for getting pages
- Server will consume less resources
    - Just use server for creating data APIs
- Easy to transform into Progressive Web Apps
- Better support for local caching and offline experiences
- Best to use when there is limited or no need for SEO

Cons

- Single Page Apps rely heavily on JavaScript
- Some of your visitors might just have JavaScript disabled
- Scroll positions have to be handled and remembered programmatically
- Cancelling API calls have to be done programmatically when user navigates to another page
- Handling unsaved changes have to be done programmatically too
- Search engine optimization (SEO) with SPAs is difficult
- Increases the chance of memory leaks
    - As SPAs don't load pages, the initial page will stay open for a long time

Installing React

Prerequisites

- Node.js and NPM or Yarn
- Node.js is a Javascript run-time environment that allow us to execute Javascript code like if we were working on a server.
- NPM amd Yarn are package manager for Javascript.
  - Allows us to easily install Javascript libraries
- You’ll need to have Node >= 8.10 and npm >= 5.6 on your machine

Install Node

- Download Node here - https://nodejs.org/en/
- Double click the downloaded file
- Go through the installation widget
- After completing the installation open up Terminal or Command Prompt window and run

```
node -v

// output - v8.9.4
```

- If a version was output, then you're all set.

Install Yarn

- Follow the step by step guide here - https://yarnpkg.com/lang/en/docs/install

Install ReactJS using create-react-app

- Create React App is a comfortable environment for learning React.
- Best way to start building a new single-page application in React.
- Sets up your development environment
- Provides a nice developer experience, and optimizes your app for production

```
npx create-react-app ninja-academy
cd ninja-academy
yarn start
```

Online Playgrounds

- If you just want to play around with React to test it out follow these links
- Codepen - https://codepen.io/pen?&editable=true&editors=0010
- Sandbox - https://codesandbox.io/s/new

Deploying React App To Internet

- Developing locally is good for testing but you may want to deploy your amazing app to the internet for the world to see
- You can do it for free by using services like Netlify, Firebase, and many others
- We will look at how to deploy your app to the internet using Netlify
- Netlify is a great service that lets you deploy static sites for free
- It also includes tooling around managing your deployment process

Deploy to Netlify

- First step, register your Netlify account - https://app.netlify.com/signup
- Select on `New site from Git` button
- Authorize Github to give access to Netlify
- Then select your project repo
- Select `master` branch to deploy from
- Enter `npm run build` in the build command
- Enter `build/` folder as your publish directory
- Click on "Deploy Site"
- Once the deployment is complete the status will change to "Published"
- Then go to your site
    - URL will be on the top left on the Netlify project page
    - URL will look something like this `https://happy-ramanujan-9ca090.netlify.com/`
- Your React app is now deployed

Easy setup deploy

- If you want an easy way to deploy your simple HTML and CSS page follow these steps
- Go to Netlify Drop
- Drop the folder that contains your HTML and CSS file on that page where it says `Drag and drop your site folder here`
- And Voila! It should create a unique URL for your project
- URL will look something like this `https://happy-ramanujan-9ca090.netlify.com/`

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

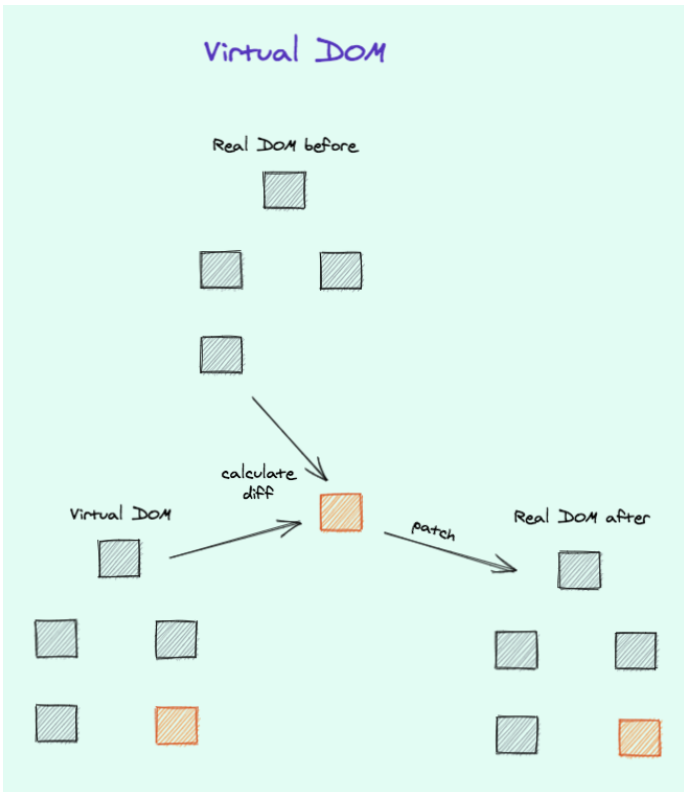
Cons of real DOM

- 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

React Components

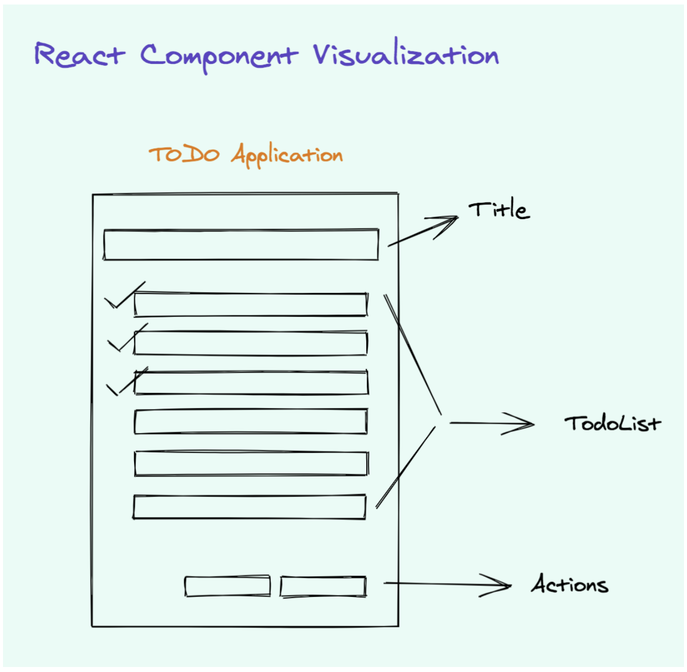
- It is a building block that describes what to render
- To create component
    - Create local variable with Uppercase letter
    - React uses this to distinguish between components and HTML
- It is the heart and soul of react
- Every component must implements "render" method
- Every component has `state` obj and `prop` obj

```
// Create a component named FirstComponent
class FirstComponent extends React.Component {
  render() {
    return <h2>Hi, I am first react component!</h2>;
  }
}
```

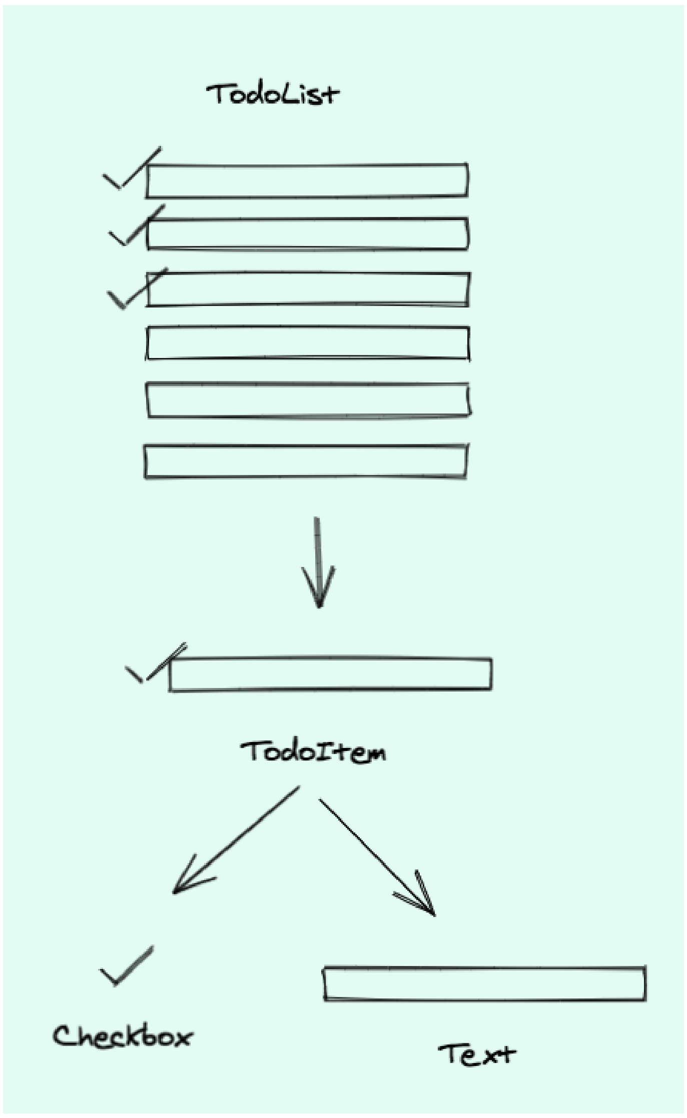
- Your first component is called `FirstComponent`
- Your component returns `h2` element with string `Hi, I am first react component!`
- Using your first component

```
// As a root component

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

// As a child component

<div>
  <FirstComponent />
</div>
```

Thinking in components



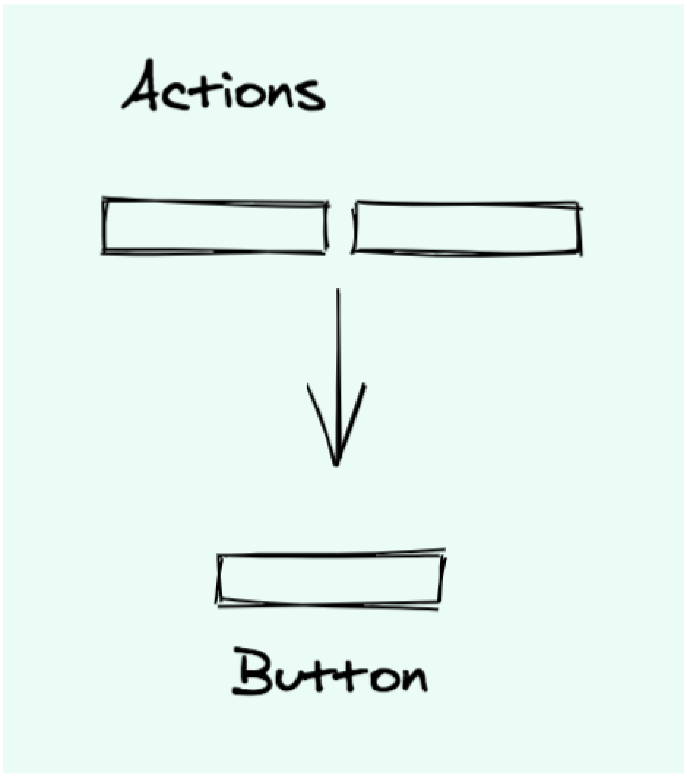
- Let's see what components are and how to use them
- Check out the illustration above
- It is a page from a TODO application
- It has a title
- It has list of TODO items
- Then at the bottom of the page - there are actions you can take
  - Like adding new item, editing an item, etc.
- Imaging you wanted to implement this using React framework
- Best thing about React is you don't have to implement this entire page in a single file
- You can break this TODO page into logical parts and code them separately
- And then join them together linking them with line of communication between them
- See below



- Look at the TODO list
- Do you see any similarity between the list items?
- Yes - they all have a title and a checkbox to mark them complete
- So, that should instantly spark an idea in you that this part can be refactored out into it's own component
- Like below

```
// Pseudo code for the component

class TODOItem extends React.Component {
  render() {
    return (
      <div>
        <input type="checkbox" isDone={isDone}/>
        <label title={title} />
        </div>
      )
    }
  }
```

- We called our component `TODOItem`
- It has a checkbox and text-label baked into it
- You can go even further and refactor out the `checkbox` and `label` into its own component
- I'd highly suggest you to do that because it will keep your code cleaner and give you an opportunity to customize it however you like



- Similar to our `TODOItem` component we can refactor out the actions too
- See below

```
// Pseudo code for the component

class Actions extends React.Component {
  render() {
    return (
      <div>
        <button title="Clear"/>
        <button title="Add"/>
      </div>
    )
  }
}
```

- We have declared `Actions` component above
- It has two buttons - one for clearing the list and another for adding an item
- Even here you can go ahead and refactor out a `Button` component
- Doing this you can encapsulate the similar looks and behavior of the button and give the consumer ability to customize it as they would like

```
// Pseudo code for the component

class Button extends React.Component {
  render() {
    return (
      <button title={title} onClick={onClickHandler} />
    )
  }
}
```

Component Render

- Every component must have render function
- It should return single React object
    - Which is a DOM component
- It should be a pure function..
    - It should not change the state
    - Changing state in render function is not allowed

    - So, do not call `setState` here
        - Because it will call render again
- Component renders when `props` or `state` changes

```
// Root level usage

ReactDOM.render(<p>Hi!</p>, document.getElementById('root'));


// Using render inside component

class MyComponent extends React.Component {
  render() {
    return <h2>I am a Component!</h2>;
  }
}
```

Function Components

- They are stateless components by default
  - You can use React hooks to add states though
  - More about this in React hooks section
- It receive `props` as arguments
- The is no internal state
- There are no react lifecycle methods available
  - You can you react hooks to achieve this
  - React hook like `useEffect` should give you most of what you'd need
- Use it whenever possible
  - Whenever you don't care about the state of the component
  - Greatly reduce the baggage of class components
  - Also offer performance advantages
  - Improves readability

```
// First function component

function FirstComponent() {
  return <h2>Hi, I am also a Car!</h2>;
}
```

Class Components

- This is considered as the "old way" of defining components in React
- Define components using Classes
- They are stateful
  - They store component state change in memory
- You can access props via `this.props`
- You can access state using `this.state`

- They can use component lifecycle methods
- Whenever you care state of the component
  - Use class component

```
// Class component

class ClassComponent extends React.Component {
  constructor() {
    super();
    this.state = {age: 26};
  }

  render() {
    return <h2>I am a class component!</h2>;
  }
}
```

NOTE: Please prefer using Function components with React hooks whenever you need to create component that need to track it's internal state.

Pure components

- They are simplest and the fastest components
- You can replace any component that only has `render()` function with pure components
- It enhances simplicity and performance of app
- When `setState` method is called:
    - Re-rendering happens - it's blind
    - It does not know if you really changed the value
    - It just re-renders - this is where pure component comes in handy
- Pure components does a shallow compares the objects
    - So, do not blindly use pure components!
    - Know how your component state works before you use it

```
// App is a PureComponent
class App extends PureComponent {
  state = {
    val: 1
  }

  componentDidMount(){
    setInterval(()=> {
      this.setState(()=>{
        return { val : 1}
      });
    }, 2000)
  }

  render() {
    console.log('render App');
    return (
      <div className="App">
      <Temp val={this.state.val}/>
      </div>
    );
  }
}
```

Reusing Components

- We can refer to components inside other components
- This is great for writing a clean code
- It is recommended to refactor duplicate elements to their own reusable components
- Good rule of thumb - if you are using something more than 2 times refactor it out to be able to reuse it

```
class ChildComponent extends React.Component {
  render() {
    return <h2>I am child!</h2>;
  }
}

class FirstComponent extends React.Component {
  render() {
    return (
      <div>
        <h1>I am parent</h1>
        <ChildComponent />
      </div>
    );
  }
}

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

States And Props

States

- It represents data internal to the component
- It represents the STATE of the component!
- It is objects which supplies data to component
- Variables which will modify appearance -
    - Make them state variables
- DON'T include something in state if you DON'T use it for re-rendering your component
    - Include it in prop instead
- Changing state re-renders component
- State object is immutable

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      array: [3,4],
      userName: 'ninja'
    };

  }

  changeUserName = () => {
    // NOTE: this will NOT change this.state.array
    this.setState({
      userName: 'rjNinja'
    });
  }

  render() {
    return (
      <div>
        <p>
          It is a {this.state.userName}
        </p>
        <button
          type="button"
          onClick={this.changeColor}
        >Change UserName</button>
      </div>
    );
  }
}
```

Props

- Props are used to communicate between components
- Child components should be passed `state` data using `props`
- Props are immutable
- They are readonly
    - You will get an error if you try to change their value.
- Always passed down from parent to child
- Data Flow:
    - `state c1 -> prop c2 -> state c2 -> render`
    - State data of c1 component are passed at props to the c2 component
    - c2 component use the props as initialize its own state data
    - And then the state data is used to render the c2 component
- Keep props in control
    - Don't go crazy about defining them

```
// In Parent component
<Child name="Dan" id="101"></Child>

// Child component definition
<h1>{props.name}</h1>
<p>{props.id}</p>
```

Event Handling

- Very similar to handling events on DOM elements
- Events are named using camelCase, rather than lowercase
- With JSX you pass a function as the event handler
- React has the same events as HTML: click, change, mouseover etc

```
// handleClick is an event handler attached to the click event
<button onClick={handleClick}>
  Activate Lasers
</button>
```

- Call `preventDefault` explicitly to prevent default behavior

```
function handleClick(e) {
    e.preventDefault();
}
```

Bind `this`

- Needed only in case of Class components
- `this` represents the component that owns the method

> Tip: Use arrow functions to avoid confusion about `this`

```
class Car extends React.Component {
  drive = () => {
    // The 'this' keyword refers to the component object
    alert(this);
  }

  render() {
    return (
      <button onClick={this.drive}>DRIVE</button>
    );
  }
}
```

Passing Arguments

```
// event handler
drive = (a) => {
  alert(a);
}

// inside render function
<button onClick={() => this.drive("Far")}>Drive!</button>
```

Two Way Binding

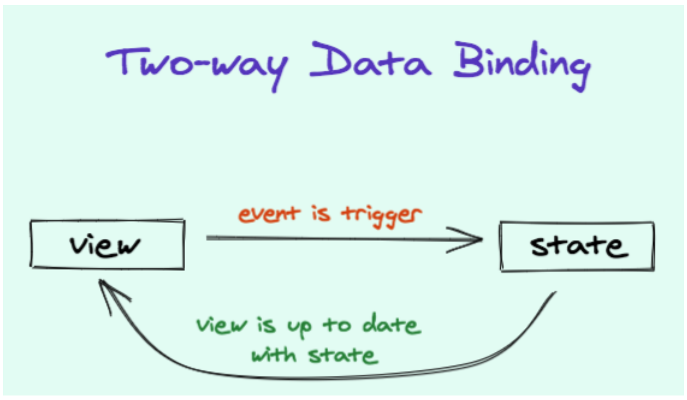
- Idea of two way data binding is that the state of the component and the view that user sees remains in sync all the time

One way data binding

- React supports one-way data binding by default
    - Unlike `AngularJS`
- So, a parent component has to manage states and pass the result down the chain to its children

Two Way - 2 way binding

- Two Way data binding is not available by default - out of the box like `AngularJS`
- You need event handling to do that



- Check the above illustration

- Users types something on the input control on the view - the UI

- This triggers an `on-change` event

- We can listen on this event and update the state of the component with the value that user typed
- And the state of component becomes the source of the truth - meaning its value is then fed to the view that user sees
- Below code example is how you update state with the value entered on the input

```
onHandleChange(event) {
  this.setState({
    homeLink: event.target.value
  });
}

<input type="text" value={this.state.homeLink} onChange={(event) =>
this.onHandleChange(event)} />
```

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>
  );
}
```

CSS Stylesheets

- For bigger applications it is recommended to break out styles into separate files
- This way you can also reuse styles in multiple components by just importing the CSS file

```
// style.css file
.myStyle {
  color: "red";
  backgroundColor: "blue";
}

// component.js file
import './style.css';

class MyComponent extends React.Component {
  render() {
    return (
      <div>
        <h1 className="myStyle">Hello Style!</h1>
      </div>
    );
  }
}
```

Dynamic Styles

- You can dynamically change your component styles
- Real world example: when your input is invalid you'd want to show red color text inside the input

```
onHandleChange(event) {
  if(error) {
    this.setState({
      color: "red"
    });
  }
  else {
    this.setState({
      homeLink: event.target.value
    });
  }

}

<input type="text" value={this.state.homeLink} onChange={(event) =>
this.onHandleChange(event)} style={{color: this.state.color}} />
```

- You can also dynamically change the class names instead of individual styles

```
// Traditional way
<input type="text" className={'valid ' + this.state.newClass} />

// Using string template
<input type="text" className={`valid ${this.state.something}`} />
```

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>
  );
}
```

Outputting Lists

- React uses `Array.map` method to output items in an array
- This is needed in situations for example outputting list of fruits, or list of users in your database
- You can render multiple component using `Array.map` just as you'd do with normal collection of numbers or string

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <div>{number}</div>
);
```

- A little advanced example where you have a component that accepts list of numbers through props

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <div>{number}</div>
  );
  return (
    <div>{listItems}</div>
  );
}

const numbers = [1, 2, 3, 4, 5];
<NumberList numbers={numbers} />
```

Keys

- When you run the above code it will give you a warning that a key should be provided to each of the item
- Also, when you provide that key it should be unique for each item
- You can modify the code to below

```
const listItems = numbers.map((number) =>
  <div key={number.toString()}>{number}</div>
);
```

- Keys are useful construct in React
- It helps React identify and keep track updates on each item in the collection
- That is why it is important for keys to be unique
- Avoid using `index` of array item as keys as it may negatively impact performance and cause unexpected bugs

Higher Order Components

- It is a design pattern
    - Not specifically for React - but widely used in React
- It is a technique for reusing component logic
- Higher-order component is a function that takes a component and returns a new component

```
const ModifiedComponent = higherOrderComponent(WrappedComponent);
```

- It is used to move out the shared same functionality across multiple components
- Example: manage the state of currently logged in users in your application
    - create a higher-order component to separate the logged in user state into a container component
    - then pass that state to the components that will make use of it

```
// commonStyles.js
// this is used by HOC
const styles = {
  default : {
    backgroundColor: '#737373',
    color: '#eae8e8',
  },
  disable : {
    backgroundColor: '#9c9c9c',
    color: '#c7c6c6',
  }
}

export default styles;


// HOC

const translateProps = (props) => {
  let _styles = {...commonStyles.default}

  if(props.disable){
    // existing plus disable styles!!!
    _styles = {..._styles, ...commonStyles.disable};
  }

  const newProps = {...props, styles:_styles }
  return newProps;
}

// this function is the HOC function
// it takes in button... and passes the necessary props to that component
export const MyHOC = (WrappedComponent) => {
  return function wrappedRender(args) {
    return WrappedComponent(translateProps(args));
  }
}

// USAGE: button component
const MyButton = (props) => {
  return (
    <button style={props.styles}>I am MyButton</button>
  )
}

// USAGE of HOC
export default MyHOC(ButtonOne);
```

Cons of HOC

- HOC can be difficult to understand initially
- Every view that will be used with the HOC has to understand the shape of the `props` passed
- Sometimes we have to have a component whose only purpose is transforming `props` into the intended ones, and that feels inefficient
- Some HOCs will always lead to branched views...
  - similar views but not exactly same looking
- Multiple/nested HOCs are bug prone
  - hard to debug bugs
- Typing is not easy

Render Props

- Render prop is a pattern in which you are passing a function as a prop
- Also known as `Children as Function` pattern
- `render props` and `HOC` patterns are interchangeable
  - You can use one or the other
- Both patterns are used to improve reusability and code clarity
- Simply put, when you are using `this.props.children` you are using render prop
  - We are using children as the render prop

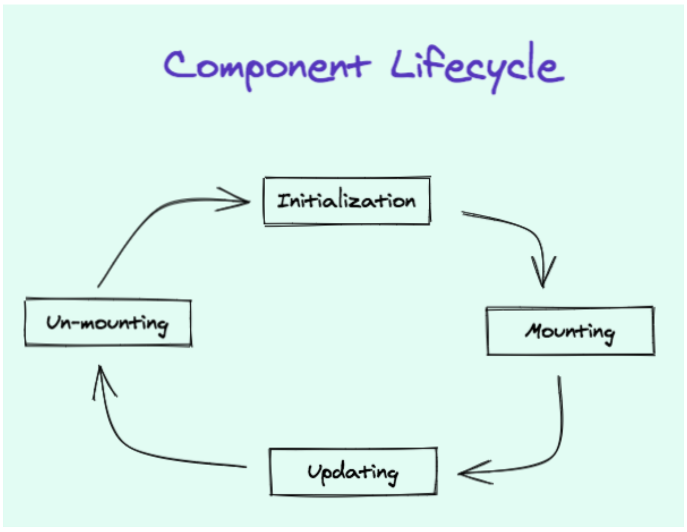
Con

- It can lead to a nasty callback hell
  - This is solved by react hooks

```
// Section
const Section = ({children}) => {
  const number = 1000;

  return children(number);
};


// App
<Section>
  {({number}) => (
    <div>
      <p>You are at {number}</p>
    </div>
  )}
</Section>
```

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();
}
```

Error handling

- `static getDerivedStateFromError()`
- Whenever an error is thrown in a descendant component, this method is called first

```
static getDerivedStateFromError(error) {
  console.log(`Error log from getDerivedStateFromError: ${error}`);
  return { hasError: true };
}
```

`componentDidCatch()`

- Also called after an error in a descendant component is thrown
- It is passed one more argument which represents more information about the error

```
componentDidCatch(error, info) {

}
```

Module 5 - React hooks

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'
});
```

useEffect React Hook

- It lets you perform side effects in function components
- It is used to manage the side effects
- Examples of side effects:
    - Data fetching
    - Setting up subscription
    - Manually changing DOM in react components
- Think of `useEffect` Hook as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` combined together into one function
    - So previously we would have to duplicate the code in all those 3 lifecycle hooks
    - This is solved with `useEffect`

More About `useEffect`

- Effects scheduled with `useEffect` don't block the browser from updating the screen
    - Unlike the class based lifecycle
- By using this Hook, you tell React that your component needs to do something after rendering
- Does `useEffect` run after every render? Yes!
- You can add multiple `useEffect` hook functions to do different things in a single component
    - That is the beauty of `useEffect`

Cleanup

- `useEffect` also handles cleanup
- Just return a function which does the cleanup
    - `useEffect` will run it when it is time to clean up
- React performs the cleanup when the component unmounts

- React also cleans up effects from the previous render before running the effects next time

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

function MyComponent() {
  const [count, setCount] = useState(0);

  // NOTE: when component is mounted pass empty array
  // this hook only runs 1 time after component is mounted
  useEffect(() => {
    console.log("I am born. Runs 1 time.")
  }, []);

  // NOTE: define another useeffects for componentDidMount and
componentDidUpdate
  useEffect(() => {
    console.log("Runs on initial mount, and also after each update")

    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;

    // NOTE: return function that does the cleanup
    // Unsubscribe any subscriptions here
    return function cleanup() {
      console.log("This function takes care of cleanup")
      // ex: window.removeEventListener()
    }
  });

  return <button onClick={() => setCount(count + 1)}>Update
Count</button>;
}
```

useRef React Hook

- `useRef` returns a mutable ref object.
- The object's `.current` property is initialized to the passed argument - default initial value.
- The returned object will persist for the full lifetime of the component.

```
const refObject = useRef(initialValue);
```

Two Use Cases

1. Accessing DOM nodes or React elements

- You can use it to grab a reference of the text input element and focus it on button click

```
import React, { useRef } from "react";

const MyComponent = () => {
  const myInput = useRef();

  focusMyInput = () => myInput.current.focus();

  return (
    <>
      <input type="text" ref={myInput} />
      <button onClick={focusMyInput}>Focus input</button>
    </>
  );
}
```

- NOTE: you can use `createRef` if you are using Class components

2. Keeping a mutable variable

- Equivalent to instance variables in class components
- Mutating the `.current` property won't cause re-renders

```
const MyComponent = (prop) => {
  const myCounter = useRef(0);

  const updateState = () => {
    // Now we can update the current property of Referenced object
    myCounter.current++;
  }

  return (
    <div>
      <div>
        <div>myCounter : {myCounter.current}</div>
        <input type="button" onClick = {() => updateState()} value="Update
myCounter"></input>
      </div>
    </div>
  );
}
```

- Mutating the `.current` property doesn't cause a re-render
- React recommends three occasions where you can use it because you have no other choice.
  - Managing focus, text selection, or media playback.
  - Integrating with third-party DOM libraries.
  - Triggering imperative animations.

Context

- Provides a way to pass data through the component tree
    - without having to pass props down manually at every level
- Passing down props can be cumbersome when they are required by many components in the hierarchy
- Use it when you want to share data that can be considered “global” for a tree of React components
    - Ex: User information, Theme, Browser history

React.createContext

- Use `React.createContext` to create a new context object

```
const MyThemeContext = React.createContext("light");
```

Context provider

- We need a context provider to make the context available to all our React components
- This provider lets consuming components to subscribe to context changes

```
function MyComponent() {
  const theme = "light";

  return (
    <MyThemeContext.Provider value={theme}>
      <div>
        my component
      </div>
    </MyThemeContext.Provider>
  );
}
```

Consuming context

- We assign a `contextType` property to read the current theme context
- In this example, the current theme is "light"
- After that, we will be able to access the context value using `this.context`

```
class ThemedButton extends React.Component {
  static contextType = MyThemeContext;

  render() {
    return <Button theme={this.context} />;
  }
}
```

useContext

- In the previous chapter we learned about what is `Context` and why it is useful in React
- `Context` can also be used in Function components

- Consuming context with functional components is easier and less verbose
- We just have to use a hook called `useContext`
- It accepts a context object
- It returns the current context value for that context

```
const Main = () => {
  const currentTheme = useContext(MyThemeContext);

  return(
    <Button theme={currentTheme} />
  );
}
```

- `useContext(MyThemeContext)` only lets you read the context and subscribe to its changes
- You still need a `<MyThemeContext.Provider>` above in the tree to provide the value for this context

```
function MyComponent() {
  const theme = "light";

  return (
    <MyThemeContext.Provider value={theme}>
      <div>
        my component
      </div>
    </MyThemeContext.Provider>
  );
}
```

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

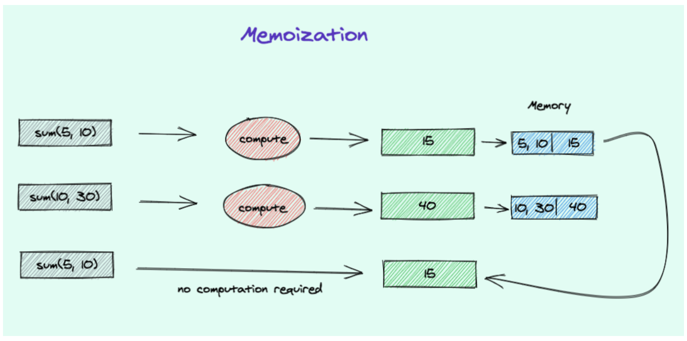
```
// TODOComponent.js
class TODOComponent extends Component{
    render() {
        return <div>TODOComponent</div>
    }
}

// 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

Memoization

- This feature is available from React > 16.6 version
- Memoization is a technique to use previously stored results and fasten computation

- Memoization means caching the output based on the input
    - In the case of functions, it means caching the return value based on the arguments
- It is similar to `shouldComponentUpdate` or using pure components
    - But you don't need class components
    - You can use `React.memo` on function components



- Let's see how memoization works using the above illustration
- You can see that `sum()` method is being called multiple times
- Without memoization the `sum()` method would be called and executed multiple times even if the parameters passed to the methods are the same
- From the above illustration `sum(5, 10)` would be computed twice WITHOUT memoization
- This becomes expensive as your application starts to grow
- The solution for this is - `memoization`
- If the same method is called multiple times and the parameters passed to it are the same - it logically means the answer would be the same
- So, in memoization instead of computing the answer it is stored in a memory
- And when the same method is called with the same parameters the stored answer is returned without wasting the computation cycle on it
- Similar idea is applied on memoized react components too
- Because under the hood react components are nothing but JavaScript functions
- And parameters passed to it are the `props`
- So, if the `props` are the same memoized components are not re-rendered unnecessarily

When to use it?

- It is used to not re-render your components unnecessarily
- Suppose your `state` object is updating
  - But the value is not really changing
  - You can use memo to NOT re-render your functional component
- If your component just takes primitive values as props,
  - Just wrap it in `memo()` to prevent an unwanted re-render.

```
export default React.memo((props) => {
  return (<div>{props.val}</div>)
})
```

- `React.memo()` by default just compares the top-level props
- It is not that simple for nested objects
- For nested objects, you can pass custom comparer function to check if prop values are same

```
const MemoedElemenet = React.memo(Element, areEqual)

export function areEqual(prevProps: Props, nextProps: Props) {
  const cardId = nextProps.id
  const newActiveCardId = nextProps.activeCardId
  const isActive = newActiveCardId === cardId

  return !some([
    isActive,
  ])
}
```

> TIP: General advice is to avoid memoization until the profiler tells you to optimize

`useMemo`

- `React.memo` is used to memoize components
- You can use `useMemo` to memoize inner variables
- If there's CPU intensive operation going on to calculate those variables
- And the variables does not really change that often - use `useMemo`

```
const allItems = getItems()

// CPU intensive logic
const itemCategories = useMemo(() => getUniqueCategories(allItems),
[allItems])
```

Lazy Loading

- Lazy loading is another technique to improve your app's performance

- You can split your JavaScript bundles and dynamically import the modules
- Example: only import lodash `sortby` function dynamically in the code where it is actually needed

```
import('lodash.sortby')
  .then(module => module.default)
  .then(module => doSomethingCool(module))
```

- Another way is to load components only when they are in the viewport
- You can use this awesome library react-loadable-visibility for this purpose

```
// example from https://github.com/stratiformltd/react-loadable-visibility

import LoadableVisibility from "react-loadable-visibility/react-loadable";
import MyLoader from "./my-loader-component";

const LoadableComponent = LoadableVisibility({
  loader: () => import("./my-component"),
  loading: MyLoader
});

export default function App() {
  return <LoadableComponent />;
}
```

Suspense

> NOTE: Suspense is an experimental feature at this time. Experimental features may change significantly and without a warning before they become a part of React.

- Suspense is another technique used for lazy loading
- It lets you "wait" for some code to load and lets you declaratively specify a loading state (like a spinner) while waiting

```
import React, { Component, lazy, Suspense } from 'react'

const MyComp = lazy(() => import('../myComp'))

<Suspense fallback={<div>Loading...</div>}>

<div>
  <MyComp></MyComp>
</div>
```

- Suspense is most popularly used for waiting for the data
- But it can also be used to wait for images, script, or any asynchronous code
- It helps you avoid race conditions
  - Race conditions are bugs that happen due to incorrect assumptions about the order in which our code may run.
  - Suspense feels more like reading data synchronously — as if it was already loaded.

Suspense - Data Fetching

- It is used to wait for rendering component until the required data is fetched

    - That means - no need to add conditional code anymore!
- You need to enable `concurrent mode`
    - So the rendering is not blocked
    - This gives better user experience

```
//  install the experimental version

npm install react@experimental react-dom@experimental
```

```
// enable concurrent mode

const rootEl = document.getElementById('root')

// ReactDOM.render(<App />, rootEl)
const root = ReactDOM.createRoot(rootEl) // You'll use a new createRoot
API

root.render(<App />)
```

- Using `Suspense` for data fetching

```
const CartPage = React.lazy(() => import('./CartPage')); // Lazy-loaded

// Show a spinner while the cart is loading
<Suspense fallback={<Spinner />}>
  <CartPage />
</Suspense>
```

This approach is called `Render-as-You-Fetch`

1. Start fetching
2. Start rendering
3. Finish fetching

- It means we don't wait for the response to come back before we start rendering
- We start rendering pretty much immediately after kicking off the network request

```
const resource = fetchCartData();

function CartPage() {
  return (
    <Suspense fallback={<h1>Loading cart...</h1>}>
      <CartDetails />
    </Suspense>
  );
}

function CartDetails() {
  // Try to read product info, although it might not have loaded yet
  const product = resource.product.read();
  return <h1>{product.name}</h1>;
}
```

Sequence of action in the above example

1. `CartPage` is loaded
2. It tries to load `CartDetails`
3. But, `CartDetails` makes call to `resource.product.read()` - so this component `"suspends"`
4. React shows the fallback loader and keep fetching the data in the background
5. When all the data is retrieved the fallback loader is replaced by `CartDetails` children