Episode - 1 | Inception

How can you add React in your project?

cdn react

→ cdn stands for Content Delivery Networks

→ These are websites where React has been hosted, and we are just pulling React from there into our project

→ React's official documentation has these CDN links

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

→ We can paste these script links inside our HTML

What happens though?

→ Now, our project has React in it!

→ Now, whatever code we write in React, our browser will be able to understand

→ In other words, we have injected React into our project

→ The src links inside these script tags are JS files with JS code

→ At the end of the day, React is a Javascript library containing JavaScript files

→ We have support for all this JS code / React into our project now

Check out what Happens Next!

```
Elements  Console  Sources  Application  Network  Performance  Memory  Security  Lighthouse  Record
top ▾  Filter
Download the React DevTools for a better development experience: https://reactjs.org/link/react-devtools
[Five Server] connecting...
[Five Server] connected.
> React
<  ▸ {Children: {…}, Fragment: Symbol(react.fragment), Profiler: Symbol(react.profiler), Component: f, PureComponent: f, …}
```

↳ our project now has React support!

→ We can even access the 'React' object in the DOM now!!

→ This 'React' has been made available using the CDN

But there were two script files right?

What does the other one do?

→ The first file (... development.js) is the core of React, the framework in it's entirety!

→ The second file (react-dom ...) is the React library, useful for DOM operations.

→ Or, it's that file we need to modify the DOM

Why does React have two separate files for this?

Couldn't they have just clubbed it in one file?

→ No.

→ React doesn't only work on browsers alone. It also works on mobile phones (React Native)

→ The second file is like a bridge between React and browsers, it's like a bridge to connect to the DOM

```
> ReactDOM
  ▼ {__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED: {…}, createPortal: f, createRoot: f, findDOMNode: f, flushSync: f,
    ▶ createPortal: ƒ createPortal$1(children, container)
    ▶ createRoot: ƒ createRoot$1(container, options)
    ▶ findDOMNode: ƒ findDOMNode(componentOrElement)
    ▶ flushSync: ƒ flushSync$1(fn)
    ▶ hydrate: ƒ hydrate(element, container, callback)
    ▶ hydrateRoot: ƒ hydrateRoot$1(container, initialChildren, options)
    ▶ render: ƒ render(element, container, callback)
    ▶ unmountComponentAtNode: ƒ unmountComponentAtNode(container)
    ▶ unstable_batchedUpdates: ƒ batchedUpdates$1(fn, a)
    ▶ unstable_renderSubtreeIntoContainer: ƒ renderSubtreeIntoContainer(parentComponent, element, containerNode, callback)
      version: "18.3.1"
    ▶ __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED: {usingClientEntryPoint: false, Events: Array(6)}
    ▶ [[Prototype]]: Object
```

→ This is what we get from the second file!

Creating Elements with React

→ Previously we have used "document.createElement()" Web API

→ In React, we use "React.createElement()" API

→ It takes three arguments

- **first** — the element
- **second** — an object (let's make do with an empty object {} for now)
- **third** — content (what we want to put inside our element)

```
<script>
    const heading = React.createElement("h1", {}, "Hello React!")
</script>
```

first — second — third

→ Nice, but this still job not done as we need to display, or append this element into the actual DOM!

→ This is where the React DOM comes into power

→ Although, while performing DOM operations, React has a different way of going about things

→ We first need a root, where we can render this heading

→ Our root in this case will be the parent element of our element, where we want to insert it.

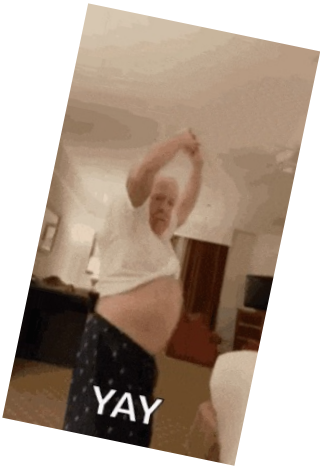
```
<script>
  const heading = React.createElement("h1", {}, "Hello React!");
                  > different roles
  const root = ReactDOM.createRoot(document.getElementById("root"));
                                   where we want to render

  root.render(heading);
</script>
```

→ Finally Rendering

Hello React!

It's a success!



What is the most costly operation in a browser / web page?

→ The most important hit that the browser takes, is when the DOM nodes need to be manipulated.

→ Like something showing up, when you close the web page (i.e the DOM tree is being changed) → shit is costly af

→ So, all these frameworks and libraries, are trying to optimize it

→ React also comes with the philosophy that whenever you need to do something on a webpage, do it using JavaScript

What was the empty object we passed as second argument while creating an element?

→ In that object, we can provide attributes to our tags!

What does React.createElement() return us?

→ It does not return us a DOM element, if that's what you were guessing!

→ It returns us a React Element which is nothing but a JavaScript Object!

```
▼ {$$typeof: Symbol(react.element), type: 'h1', key: null, ref: null, props: {…}, …}
    $$typeof: Symbol(react.element)
    key: null
  ▼ props:
      children: "Hello React!"
      id: "heading"
    ▶ [[Prototype]]: Object
    ref: null
    type: "h1"
    _owner: null
  ▶ _store: {validated: false}
    _self: null
    _source: null
  ▶ [[Prototype]]: Object
```

This is what we passed while doing React.createElement()

What does root.render() do?

→ It's job is to take the React Element / JS object, and create an element which the browser understands, and put it inside the root element that we initialized

→ This root is a ReactDOM element and a JavaScript object

→ It's the responsibility of the render() method to take the React element and convert it into the actual HTML element and manipulate the DOM

Creating Nested Elements

→ The third argument is basically the child

→ If there is no child, it renders as the actual content of the element

```
const parent = React.createElement(
  "div",
  { id: "parent" },
  React.createElement(
    "div",
    { id: "child" },
    React.createElement("h1", {}, "I am a nested H1 tag!")
  )
);
```

third argument

third argument

```
▼<div id="root">  flex
  ▼<div id="parent">
    ▼<div id="child">
       <h1>I am a nested H1 tag!</h1>
     </div>
   </div>
 </div>
```

I am a nested H1 tag!

→ But, the more you nest, the uglier it can become

```
const parent = React.createElement("div", { id: "parent" }, [
  React.createElement("div", { id: "child1" }, [
    React.createElement("h1", {}, "I am a nested H1 tag!"),
    React.createElement("h2", {}, "I am nested sibling"),
  ]),
  React.createElement("div", { id: "child2" }, [
    React.createElement("h1", {}, "I am a nested H1 tag!"),
    React.createElement("h2", {}, "I am nested sibling"),
  ]),
]);
```

I am a nested H1 tag!

I am nested sibling

I am a nested H1 tag!

I am nested sibling

Interesting Observation

const root = ReactDOM.createElement($\underbrace{\text{document ....}}_{\text{actual DOM element which will help in creating a ReactDOM element}}$)

→ When render() is invoked on the root, whatever ReactElement is passed as argument, will replace any existing children of the DOM element from which the root ReactDOM element was created!

What is the difference between library and a framework?

→ React is a library as it can be applied on a small portion of the page itself, individually

↳ React will only work in places where you define the root?

→ It can work independently on any portion of the code, and hence it's a library and not a full-fledged framework!

→ Our root element, on which we work on can easily be a small HTML element, ranging to the HTML tag itself

So, can we create large scale applications from React?

→ Yo