**001. React**

**Introduction**

The biggest trend in Front-End user interface development in the recent years has been the **Component-Driven Development** and the React library is all about that, as it is heavily based on the idea about components. This way of building software is not new. **Modularity** is known from a long time. It is one of the key principles in **Software Engineering** and it represents the process of dividing a software system into multiple independent modules, where each module works independently. The aim of this idea, this development by gathering similar functionality into modules and reusable code into components aims to achieve low coupling and high cohesion in a software.

**Open the following resource:** <https://beta.reactjs.org/learn/start-a-new-react-project>

React is a library for building user interfaces and as such is nearly never used by itself. The recommended way to use the React library is as a part of industry recognized toolchains or frameworks, because these tools provide a comfortable battle tested development environment. However, I do believe that we, as developers, should know how our tools work under the hood and what problems they are solving, instead of simply using them, because if we do not attempt to understand the tools by reverse engineering them out of the complex toolchains they are incorporated into, we will never grasp the exact purpose that they are serving us and will always use them, because they are necessary. Therefore, the approach that we will follow in this course will be to gradually build a project with React, without using any of those tools at all and introduce the various tools one at a time, so that we understand the actual problem solved by the tool.

**Starting a new project with React**

Now because this course is a really deep dive into React, we are going to take it very slow and step by step when we explore those ideas like modularity and components. The first thing we are going to do is setup a React project, so let’s go and check the official documentation to get us started.

Alright, so according to the React documentation, there are several ways to start a new project with React.

- We can start with an HTML file and a script tag.

- Start with a minimal toolchain. (e.g., CRA (Create React App))

- Start with an opinionated framework. (e.g., Next.js)

Throughout this course we are going to see all of those options and more, like custom setup. We are going to explore the advantages and disadvantages of each approach, but the approach we will start with right now would be the first one. We will start by writing pure React, no JSX, no Babel, no Webpack. Just a plain old index.html page with some JavaScript.

**Step 1:**

Create a project directory, we can name it "react-fundamentals".

Then create a "src" directory inside and create an "index.html" file inside.

Then open the project with vscode and unfold the standard HTML 5 document template inside, by using emmet and typing "!" or "html:5".

Then set the "title" tag text content to "React Fundamentals"

You might have asked yourself why is this root html file always named **"index.html"**, why not **"main.html"** or something else? – And the answer to that is because when we deploy client code, like a webpage, like an html file with HTML, CSS and JS inside of it to a web server like Nginx, Apache or IIS, we would usually deploy this page in the public folder of this server and we can deploy many things, but all servers have server settings, a config file, and in this config file we can specify what file name is the default file the server looks for when in a directory. Every server has a list of configured default files and by convention the name **"index.html"** is with the biggest priority, and if it doesn't find this file it will look for the next file with second priority, which for example in Apache would be **"index.shtml"** then third priority **"index.php"**, fourth priority, fifth priority et cetera. This and the order of default files to look for can be customized in the server settings.

**Step 2:**

Add **div tag** with attribute **id="root"** inside the **body tag**. It doesn't have to be called root, but that's how it is by convention for SPA (Single Page Applications), because when you are using React, you are essentially building a tree of DOM elements and every tree has a root, hence the name.

|  |
| --- |
| <div id="root">App not rendered</div> |

**Step 3:**

Add 2 **script tags** inside the **head tag**, which are the libraries React and ReactDOM.

The first one lets you define components, use methods and is the interface of how to interact with React.

The second one takes care of things like rendering HTML elements made with React to the DOM and taking care of tracking changes in the virtual DOM and applying a diffing algorithm, before updating the real DOM, because DOM manipulations are expansive operation. Since we are in the browser, we are using ReactDOM, otherwise there are other rendering layer libraries like React Native for the mobile.

|  |
| --- |
| <script crossorigin="anonymous" src="https://unpkg.com/react@18.2.0/umd/react.development.js"></script>  <script crossorigin="anonymous" src="https://unpkg.com/react-dom@18.2.0/umd/react-dom.development.js"></script> |

**The "React and ReactDOM" libraries**

Now navigate to wherever you saved the "index.html" file and open it in the browser. You should see a line of text saying "App not rendered". Let's open up the "Inspector Tool -> Console", and confirm that our React and ReactDOM libraries are defined, as they should be, since we imported them with a script tag.

Type "React" and you should get a list of methods the library's API exposes, like "createElement" and my personal favorite "\_\_SECRET\_INTERNALS\_DO\_NOT\_USE\_OR\_YOU\_WILL\_BE\_FIRED", which contains React internal fields and methods, if we expand it we would see a field named "Scheduler", from where the joke about how ReactJS should have been called ScheduleJS.

Type ReactDOM and you should get a list of methods the library's API exposes, like "createRoot" and again you will see the "\_\_SECRET\_INTERNALS\_DO\_NOT\_USE\_OR\_YOU\_WILL\_BE\_FIRED", but with different fields and methods.

"\_\_SECRET\_INTERNALS\_DO\_NOT\_USE\_OR\_YOU\_WILL\_BE\_FIRED", is just an object which contains React internal fields and methods. Dan Abramov wrote about it here <https://github.com/reactjs/reactjs.org/issues/3896> , explaining that those internals are not safe to use and that's why it has a scary name. The React team offers no guarantees about what will happen if we access things there, as the behavior can change on any release.

Now let's explore the React library. Let's check the code behind the "createElement" and "createRoot" methods, as those are the ones, we will use next.

**Open the following links:**

[https://beta.reactjs.org/apis/react/createElement](https://beta.reactjs.org/apis/react/createElement#reference)

<https://unpkg.com/react@18.2.0/umd/react.development.js>

The "createElement" function is used to create a React element with the given type, props and children.

Read about the "createElement" function in the docs and find the "createElement" function by searching the code for "exports.createElement = createElement$1;" and navigate your way up to the actual function definition. Let's give it a bird's eye view. What parameters does it accept and does it return an element?

**Open the following links:**

<https://beta.reactjs.org/apis/react-dom/client/createRoot>

<https://unpkg.com/react-dom@18.2.0/umd/react-dom.development.js>

The "createRoot" function is used to create a root for specific domNode, and will take over managing the DOM inside it. After you have created a root, you need to call "root.render" in order to display a React component, or a whole tree of components inside of it. The function returns an object with 2 functions "render" and "unmount"

Read about the "createRoot" function in the docs and find the "createRoot" function by searching the code for " exports.createRoot = createRoot$1;" and navigate your way up to the actual function definition. Let's give it a bird's eye view. What parameters does it accept?

**The "crossorigin" HTML attribute**

Navigate to wherever you saved the "index.html" file and open it in the browser, and then open up the "Inspector Tool". Let's remove remove the crossorigin attribute from the 2 script tags, which we added earlier, what will happen? You should see 4 error issues, with the following message: "Indicate whether to send a cookie in a cross-site request by specifying its SameSite attribute".

So, what is this and why are you getting it? – You are getting this, because you are using a resource from external website and that server is attempting to set a "cookie", but it does not have the SameSite attribute, which is being reported in our browser.

Therefore, what we can do is add this crossorigin="anonymous" or the shorthand version crossorigin="", attribute, in order to say that when cross-origin request is performed, we are NOT going to send any credentials (e.g., a cookie, a certificate, a HTTP Basic authentication)

**Step 4**:

Create a simple **"Hello World"** component**.** Add new **script tag** inside the body and add the following lines.

|  |
| --- |
| <script>  const { React, ReactDOM } = window;  const rootTree = React.createElement("div", {}, React.createElement("h1", {}, "Hello World"));  const rootNode = document.getElementById("root");  const root = ReactDOM.createRoot(rootNode);  root.render(rootTree);  </script> |

The first thing we can do, is to use a **destructuring assignment** and destructure the global window variable, and assign React and ReactDOM as const variables, since we know where they came from. This step is optional for convenience, as window is the global variable, we can omit this step, meaning we can also use the libraries without referencing them with their full path, e.g. window.React; window.ReactDOM.

The next thing we create is a root component, with tag "div". Inside of it we create another component, with tag "h1". The "createElement" calls from the React library will turn this expression into a tree of DOM elements, also called a React object, which we can render to the DOM, using an appropriate rendering method from the ReactDOM library.

The third thing we do is get a **domNode**, which will serve as a container. We call it **rootNode** and pass it to **ReactDOM.createRoot**, which returns an object with 2 methods – render and unmount. In order to add our React object or tree of components to the page, we need to call the render method and provide which React object it needs to render. Once this happens, React will clear all the existing HTML content inside the provided container and will then render the component into it. It will also begin tracking and managing changes in the **vDOM** and updating the real **DOM** for this container only when necessary.

You should see **"Hello World"** on the screen.