Introduction to React

React

- **React** is a **JavaScript library** for creating **user interfaces**. React was created by **Facebook**.
- **React** supports **web applications** via React, and **native applications** via React Native.
- **React documentation** can be found at `https://reactjs.org/`.

Project Structure

- **React projects** consist of **two main directories**:
  1. **public** — The **public directory** contains **static context** (html, images, etc) that **webpack will not process**.
  2. **src** — The **src directory** contains the **JavaScript code** that **will be processes by webpack**.
- Inside the **public directory**, there is a file called **index.html**, this file is the **entry point** of the **webpage**. A minimalistic example of this document is:

```html

<!DOCTYPE html>
<html lang="en">

    <head>
        <title>React App</title>
    </head>

    <body>
        <noscript>
            You need to enable JavaScript to run this app.
        </noscript>
        <div id="root">
        </div>
    </body>

</html>

```

  – Inside this HTML document, there will be a **div (typically with id="root")** that you will use to **inject elements** with **react**.

- Inside the **src directory** there will be a **JavaScript file (typically named "index.js")** that will serve as the **entry point** for the code bundled by **webpack**. A minimalistic example of this document is:

```javascript

import React from "react";
import ReactDom from "react-dom/client";

const rootElement = document.getElementById("root");
const root = ReactDom.createRoot(rootElement);

root.render(
);

```

  – Inside the **ReactDom root element** is where **elements will be rendered from**.

- Another important file is the **package.json file** that is in the same directory as **public and src**. This file is not specific to React, rather NodeJS. This file **defines metadata about the project**.

Adding Elements to the Page

- One way you can **add an element to the page** is with the **createElement** function:

```

// Arguments are: Element Tag Name, Properties, Inner HTML
React.createElement("h1", null, "Hello, World!");

```

  - This way of **creating elements** can become very confusing when other elements are **nested**.

- A more popular way to create elements is with the **JavaScript XML (JSX) syntax**:

```

<h1>Hello, World!</h1>

```

  - Behind the scenes **babel (a JavaScript "compiler") will convert JSX** to a **createElement function call**.
  - To use **JavaScript code inside JSX elements**, you have to **wrap it** in a **pair of curly braces**.

React Components

Creating Custom Components

- A **component** is a **JavaScript function or class** that returns **JSX**.
  - Only **one element / component can be returned**, however they **can contain nested elements / components**.
- **Components** are reusable.
- The **naming convention for components** is **pascal case**.
- An example component is:

```

// Defining the component.
function MyComponent() {
    return (
        <h1>This is my component</h1>
    );
}

```

- There are **two ways** to **use components**:

```

// The first way is with self closing tags.
<MyComponent />

// The second way is with opening and closing tags.
<MyComponent></MyComponent>

```

  - **Opening and closing tags** are typically used **if the component has nested elements / components**. Other than that, they do the same thing.

React Fragments

- It is possible to **render several elements** from a **"single component"** using **fragments**.
- **Fragments** are an **empty component** that **only renders its children**.
- There are two ways to do this:

```

// The first way is with the React.Fragment component
function MyComponent() {
    return (
        <React.Fragment>
            // Elements and components
        </React.Fragment>
    )
}

// The second way is with the empty component
function MyComponent() {
    return (
        <>
            // Element / component list.
        </>
    )
}

```

Component Properties

- To make **components more dynamic and reusable** we can pass **properties to components** to change the **content rendered**.
- Using the **JSX syntax**, you can use **key-value pairs** the same way you would with **regular HTML** to pass properties.
- To **receive the properties** in the **component's definition**, you add a **props parameter** which will receive the key-value pairs as an object.
- For example:

```

// Component definition.
function MyNumber(props) {
    return (
        <p>My number is {props.number}!</p>
    )
}

// Rending the component.
<MyNumber number={3} />

```

- When **dynamically rendering a list** you **MUST ALWAYS** give **each element in the list a "key" property that is unique** (the index of each element in the list is not a good key, it should be some type of unique immutable id).