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:

```
<!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:

```
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).

Using State and Side Effects in React

Using State in React

- To **use state** in a **functional component**, you can use the **useState React hook**.
- The following example demonstrates how to use state:

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

function MyComponent () {

    // useState accepts an inital state value as an argument,
    // and returns an array containing the current state, and a
    // function to update the state.
    //
    // The function to set the state will receive the previous
    // state as the first parameter, this can be used to update
    // the state if necessary.
    const [state, setState] = useState(initialStateValue);

    // ...
}

```

Side Effects in React

- When performing **actions that have side effects** (or actions that are not involved in the rendering process) in **functional components**, the **useEffect hook should be used**.
- The following example demonstrates how to use an effect:

```
import {useEffect} from 'react';

function MyComponent() {

    // The code that performs the action with side effects or the
    // action that is not involved in the rendering process is the
    // first arugment provided in useEffect (a function must be passed).
    // The second argument is a dependency array, anytime the dependency
    // array values are modified, the useEffect hook will execute the
    // function again.
    //
    // If the dependency array provided is empty, the effect will only
    // be called once.
    useEffect(() => {
        // Perform side effects here.
    }, dependencyArray);

    // ...
}

```

React Reducer Hook

- Similar to the **useState hook** the **useReducer hook** allows us to create state, but provides the ability to automatically update the state with a **predetermined function**.
- The following example demonstrates how to use a reducer:

```
import {useReducer} from 'react';

function MyComponent() {

    // The reducer returns an array containing the current state, and
    // a function to update the state based on the predetermined function.
    // The first argument in useReducer is the function to update the state,
    // the second argument is the inital state.
    const [count, updateCount] = useReducer((count => count + 1), 0);

    // We can then do the following:
    updateCount();
```

```
14
15          // ...
16      }
17
```

Handling Forms in React

Uncontrolled and Controlled Components

- An **uncontrolled component** is a **component** that **renders form elements** such that the **element's data** is **managed by the DOM** (the default DOM behavior).
- A **controlled component** is a **component** that **renders form elements** such that the **elements data** is stored in the **form component's state**.
- The following is an example of a **controlled component**:

```
1
2       import {useState} from 'react';
3
4       function MyComponent() {
5
6           const [currentValue, setValue] = useState("");
7
8           return (
9               <input
10                  type="text"
11                  value={currentValue}
12                  onChange={event => setValue(event.target.value)}
13              />
14          )
15
16      }
17
```

Form Libraries

- There are many **existing form libraries** that exist to make **form development easier**. You can consider using them.
- Some libraries can be found at:
  1. `https://formik.org/`
  2. `https://react-hook-form.com/`
  3. `https://usehooks.com/`

Custom Hooks in React

Custom Hooks in React

- A **custom hook** is a **function** (that starts with use in its identifier by convention).
- You can then define the hook inside the function.
- Inside the custom hook, you are able to use other hooks.

React Router

Introduction to React Router

- **React Router** is a **standard library for routing in React**. It enables the navigation among views of various components in **React applications**.
- To install the **React Router** you run the following command:

```
npm install react-router-dom
```

Configuring the React Router

- The following is an example of how to configure the router:

```
import React from 'react';
import ReactDOM from 'react-dom';
import {BrowserRouter, Routes, Route} from 'react-router-dom';
import {Page1, Page2, Page3} from './Pages';

ReactDom.render(
    <BrowserRouter>
        <Routes>
            <Route path="/page1" element={<Page1 />}/>
            <Route path="/page2" element={<Page2 />}/>
            <Route path="/page3" element={<Page3 />}/>
        </Routes>
    </BrowserRouter>,
    document.getElementById('root')
);
```

- Note that the routes do not need to be at the top of the component tree, they can appear anywhere.

Linking React Router Pages

- To **link pages together** you can use the **Link** component.
- The following is an example of how to use the Link component:

```
        import {Link} from 'react-router-dom';

        // ...

        <Link to="path">Name</Link>
```

React Testing and Deployment

Testing Small Functions with Jest

- When you **install React** with **create-react-app**, a **test script is created** (this script uses the testing library Jest). This script will **run the test cases you create**.
- To **create a test file**, you make a file in the application with **the file extension ".test.js"**.
- The **naming convention** for testing files is to **give the file the same name as the one you are testing**, the only difference is the **file extension**.

- The following example demonstrates how to create test cases inside of the test files:

```
test("Descriptive Case Name", () => {
    // Create test environment.

    // You can use expect() to perform assertions, if the assertions
    // are true, the test passes, if false the test fails.
    // There is no limit to the amount of assertions you can have.
    //
    // The toBe function that is returned by the expect function is
    // one of many jest matchers that can be used for assertions.
    expect(thingThatIsBeingTest(args...)).toBe(result);
})

```

The React Testing Library

- When you **install React** with **create-react-app**, the **React Testing Library** is also installed.
- When writing **tests** that **involve rendering components**, you can use the **React testing library** to verify they were **rendered correctly**.
- The following example demonstrates how to test the rendering of a component:

```
import {render} from '@testing-library/react'
import MyComponent from './MyComponent';

test("render h1", () => {
    // Creates a react testing library query.
    const {getByText} = render(<MyComponent />);
    const h1 = getByText(/my text/);

    // You can then perform normal jest assertions.
    expect(h1).toHaveTextContent("my text");
})

```

Testing Events

- You can also **test events** with the **React Testing Library**.
- The following example demonstrates how to test an even:

  The component (Checkbox.js):

```
import {useReducer}

export function Checkbox() {
    const [checked, setChecked] = useReducer((checked => !checked), false);

    return (
        <>
            <label htmlFor="myCheckBox">
                {checked ? "checked" : "not checked"}
            </label>
            <input
                id="myCheckBox"
                type="checkbox"
                value={checked}
                onChange={setChecked}
            >
        </>
    )
}

```

The test (Checkbox.test.js):

```
import {render, fireEvent} from '@testing-library/react';
import {Checkbox} from './Checkbox';

test("Checkbox component check change event", () => {
    const {getByLabelText} = render(<Checkbox />);
    const checkbox = getByLabelText(/not checked/i);

    // Fire a click event.
    fireEvent.click(checkbox);

    // Assertion
    expect(checkbox.checked).toEqual(true);
});

```

Building React Projects for Production

- To **build the project** for **production**, you can use the **build script**:

```
npm run build

```

- This build can then be deployed to a server.