1 Introducing React and React Hooks

Exercise 1: Initializing a project with create-react-app

Step 1: Creating a new project

In order to set up a new project, we run the following command, which creates a new directory named `<app-name>`:

```
> npx create-react-app <app-name>
```

> If you prefer using the `yarn` package manager, you can run `yarn create react-app <app-name>` instead.

We are now going to create a new project using `create-react-app`. Run the following command to create a new React project for the first example of the first chapter:

```
> npx create-react-app chapter1_1
```

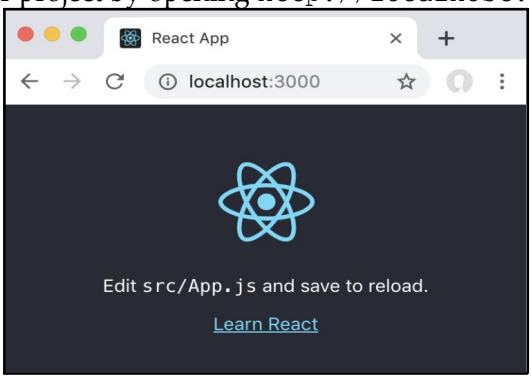
Now that we have initialized our project, let's move on to starting the project.

Step 2: Starting a project

In order to start a project in development mode, we have to run the `npm start` command. Run the following command:

```
> npm start
```

Now, we can access our project by opening `http://localhost:3000` in our browser:



Our first React app!

As we can see, with `create-react-app`, it is quite easy to set up a new React project!

Step 3: Deploying a project

To build a project for production deployments, we simply run the `build` script:

1. Run the following command to build the project for production deployment:

   ```
   > npm run-script build
   ```

   Using `yarn`, we can simply run `yarn build`. Actually, we can run any package script that does not conflict with the name of an internal `yarn` command in this way: `yarn <script-name>`, instead of `npm runscript <script-name>`.

2. We can then serve our static build folder with a web server, or by using the `serve` tool. First, we have to install it:

   ```
   > npm install -g serve
   ```

3. Then, we can run the `serve` command, as follows:

   ```
   > serve -s build
   ```

   The `-s` flag of the `serve` command rewrites all not-found requests to `index.html`, allowing for client-side routing.

Now, we can access the same app by opening `http://localhost:5000` in our browser. Please note that the `serve` tool does not automatically open the page in your browser.

After learning about `create-react-app`, we are now going to write our first component with React.

Exercise 2: Starting with a class component

First, we start out with a traditional React class component, which lets us enter a name, which we then display in our app.

Step 1: Setting up the project

As mentioned before, we are going to use `create-react-app` to initialize our project. If you have not done so already, run the following command now:

```
> npx create-react-app chapter1_1
```

Next we are going to define our app as a class component.

Step 2: Defining the class component

We first write our app as a traditional class component, as follows:

1. First, we remove all code from the `src/App.js` file.
2. Next, in `src/App.js`, we import `React`:

   ```
   import React from 'react'
   ```

3. We then start defining our own class component—`MyName`:

   ```
   class MyName extends React.Component {
   ```

4. Next, we have to define a `constructor` method, where we set the initial `state` object, which will be an empty string. Here, we also need to make sure to call `super(props)`, in order to let the `React.Component` constructor know about the `props` object:

   ```
   constructor (props) {
       super(props)
       this.state = { name: '' }
   }
   ```

5. Now, we define a method to set the `name` variable, by using `this.setState`. As we will be using this method to handle input from a text field, we need to use `evt.target.value` to get the value from the input field:

```
handleChange (evt) {
    this.setState({ name: evt.target.value })
}
```

6. Then, we define the `render` method, where we are going to display an input field and the name:

```
render () {
```

7. To get the `name` variable from the `this.state` object, we are going to use destructuring:

```
const { name } = this.state
```

The previous statement is the equivalent of doing the following:

```
const name = this.state.name
```

8. Then, we display the currently entered `name` state variable:

```
return (
    <div>
        <h1>My name is: {name}</h1>
```

9. We display an `input` field, passing the handler method to it:

```
            <input type="text" value={name}
                onChange={this.handleChange} />
        </div>
    )
}
}
```

10. Finally, we export our class component:

```
export default MyName
```

If we were to run this code now, we would get the following error when entering text, because passing the handler method to `onChange` changes the `this` context:

Uncaught TypeError: Cannot read property 'setState' of undefined

11. So, now we need to adjust the `constructor` method and rebind the `this` context of our handler method to the class:

```
constructor (props) {
    super(props)
    this.state = { name: '' }
    this.handleChange = this.handleChange.bind(this)
}
```

Finally, our component works! As you can see, there is a lot of code required to get state handling to work properly with class components. We also had to rebind the `this` context, because otherwise our handler method would not work. This is not very intuitive, and is easy to miss while developing, resulting in an annoying developer experience.

Exercise 3: Using Hooks instead(Chapter1_2)

After using a traditional class component to write our app, we are going to write the same app using Hooks. As before, our app is going to let us enter a name, which we then display in our app.ct class component!

We now start by setting up the project.

Step 1: Setting up the project

Again, we use `create-react-app` to set up our project:

```
> npx create-react-app chapter1_2
```

Let's get started with defining a function component using Hooks now.

Step 2: Defining the function component

Now, we define the same component as a function component:

1. First, we remove all code from the `src/App.js` file.
2. Next, in `src/App.js`, we import React, and the `useState` Hook:

   ```
   import React, { useState } from 'react'
   ```
3. We start with the function definition. In our case, we do not pass any arguments, because our component does not have any props:

   ```
   function MyName () {
   ```

   The next step would be to get the `name` variable from the component state. However, we cannot use `this.state` in function components. We have already learned that Hooks are just JavaScript functions, but what does that really mean? It means that we can simply use Hooks from function components, just like any other JavaScript function!

   To use state via Hooks, we call `useState()` with our initial state as the argument. This function returns an array with two elements:

   - The current state
   - A setter function to set the state

4. We can use destructuring to store these two elements in separate variables, as follows:

```
const [ name, setName ] = useState('')
```

The previous code is equivalent to the following:

```
const nameHook = useState('')
const name = nameHook[0]
const setName = nameHook[1]
```

5. Now, we define the input handler function, where we make use of the `setName` setter function:

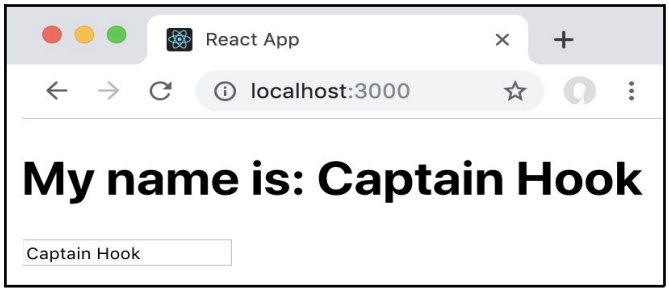
```
function handleChange (evt) {
    setName(evt.target.value)
}
```

6. Finally, we render our user interface by returning it from the function. Then, we export the function component:

```
    return (
        <div>
            <h1>My name is: {name}</h1>
            <input type="text" value={name}
                   onChange={handleChange} />
        </div>
    )
}
export default MyName
```

And that's it—we have successfully used Hooks for the first time! As you can see, the `useState` Hook is a drop-in replacement for `this.state` and `this.setState`.

Let's run our app by executing **`npm start`**, and opening `http://localhost:3000` in our browser:



Our first React app with Hooks

After implementing the same app with a class component and a function component, let's compare the solutions.

Exercise 4: Comparing the solutions

Let's compare our two solutions, in order to see the differences between class components, and function components using Hooks.

Step 1: Class component

The class component makes use of the `constructor` method in order to define state, and needs to rebind `this` in order to pass the handler method to the `input` field. The full class component code looks as follows:

```
import React from 'react'

    class MyName extends React.Component {
        constructor (props) {
            super(props)
            this.state = { name: '' }
            this.handleChange = this.handleChange.bind(this)
        }
        handleChange (evt) {
            this.setState({ name: evt.target.value })

        }

        render () {
            const { name } = this.state
            return (
                <div>
                    <h1>My name is: {name}</h1>
                    <input type="text" value={name}
                           onChange={this.handleChange} />
                </div>
            )
        }
    }

    export default MyName
```

As we can see, the class component needs a lot of boilerplate code to initialize the `state` object and handler functions.

Now, let's take a look at the function component.

Step 2: Function component with Hook

The function component makes use of the `useState` Hook instead, so we do not need to deal with `this` or a `constructor` method. The full function component code looks as follows:

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

function MyName () {
    const [ name, setName ] = useState('')

    function handleChange (evt) {
         setName(evt.target.value)
    }

    return (
        <div>
            <h1>My name is: {name}</h1>
            <input type="text" value={name}
                 onChange={handleChange} />
        </div>
    )
}

export default MyName
```

As we can see, Hooks make our code much more concise and easier to reason about. We do not need to worry about how things work internally anymore; we can simply use state, by accessing the `useState` function!