Day 2

Create a new project with *create-react-app*, as the start project for the following exercises

- Remove the icon from the project (logo.svg) and all references to the icon (in App.js)
- Remove the style from the header in App.js, and use this code as the start code for the following exercises.

Functional versus Class Components

Read/skim this article (the section Functional and Class Components)

Whenever you don't use state or life cycle methods (we will come back to both later) you can write your Components as either a Functional or a Class Component. Actually functional components are always preferred when possible.

a) Rewrite the App.js into a functional component, instead of the Class Component generated by create-react-app.

Note the exercise below demonstrates a simple way to have "many" small code examples in a single create-react-app generated project. Use this strategy for all the "smaller" exercises in this set

Composing Components

a) In the src folder, add a new file App2.js, and add an import for React in the first line (as in App.js)

b) Copy the `Welcome` and the App component (and only those) from this example into this file:

Make sure you understand why the parenthesis is needed after the return statement in the App component, and not in the Welcome Component.

c) Open index.js and comment out this line, and the corresponding import statement for App.

```
//ReactDOM.render(<App />, document.getElementById('root'));
```

Add this line instead, and import App2:

```
ReactDOM.render(<App2 />, document.getElementById('root'));
```

Verify that you can see the three Welcome Components rendered.

Obviously there are ways to navigate between pages (views) in react. For now however, use this simple strategy to hold your different examples, in a single project.

d) Answer the following questions (if you skip this part, you have wasted your time)

- What is a functional component?
  - It is like JavaScript functions. It can take an object “props” as an input and returns React element. Functional components are stateless. They are more about presentation.
- What is a Class Component?
  - It is close to OOP, what we know from Java. A new syntax coming with ES6. With Class Component (CC), we can benefit from React’s Life Cycle methods and can designate a state for this CC.

- What is the idea with props?
  - Props is an Object with JSX attributes. It is more like properties that we are sending from the parent to the children. A rule coming from React is that props needs to stay read-only, immutable. (We can use props to send/pass data between components, like uni-directional communication channel, always moving from parent to child)
- Provide a simple example in how you write a Component that accepts props:

```
function HelloToName(props){ return <div>Hello, {props.name}</div>; }
```

- Provide a simple example (could be a line from the exercise above) that demonstrates how you pass props into a Component:

  - Because this example (for me) is the same as the previous, here will show the same using of props, but this time using Class Component instead of Functional Component:

```
class HelloToName extends React.Component {
    render() {
      return <h1>Hello, {this.props.name}</h1>;
    }
  }
```

Adding Local State and Lifecycle Methods to a class

☒ **a**) Add a new file Clock.js to the *src* folder and add the code from this example (only the Clock class, remember to export it, and remember to include React)

☒ **b**) Test the code, using the simple strategy given in the previous exercise

☒ **c**) Right now the Example sleeps a hardcoded second between each update. Change this, so you can provide a sleep-time as a props value to the Clock Component.

☒ **d**) We also don't like the hardcoded message "Hello World". Add the necessary changes so you can pass in any text, via props to the Clock Component (Like: "Check our cool React driven timer ;-)"

☒ **f**) Answer the following questions (if you skip this part, you have wasted your time)

- Would it be possible to rewrite the Clock component into a functional component (provide arguments for your answer)?
  - With Functional Components we cannot benefit from the setState and Life Cycle methods, so we cannot rewrite the Clock component. We want to update(change) the UI over time, so we need to re-render the component, but because React wants to manage rendering on its own, it provides us with the setState() method. Every time setState() calls the render() method of the component, this is how the element is updated every second, two or whatever we have wrote.
- How do you set new values for state: In the constructor, and all other places?
  - We are initializing the state in the constructor of the component. Then we are calling the setState() method outside of the constructor, where we are providing the new data that we want to redraw the component with .
- How is it possible to "tell" React that you want the UI to be updated (re-rendered)?
  - Change the state of the component we want to change, using the setState() interface method. (There is another method ,the forceUpdate() , but it is not recommended to use.)

- What is the difference(s) between state and props?

- We are using state to store/save the necessary data for our component, while we are using props to pass (that) data from a parent to a child component.

- How do you pass in prop values to a Component?

- We are using JSX syntax to do it, which is very similar to the HTML/XML syntax that we are familiar with, like to set an attribute: `<Component name="something" />`

- What is the purpose of React Components Life Cycle Methods?

- If we want to change something during the program runs, we can override this Life Cycle methods. For every component we are about to use, we can control what happens when each tiny part of our UI renders, re-renders and then disappears eventually if we need to.

Lifecycle method Continued

This exercise continues with the previous example, however before you start, you should skim this article about lifecycle methods.

The article introduces the lifecycle methods for Mounting, Updating and Unmounting components.

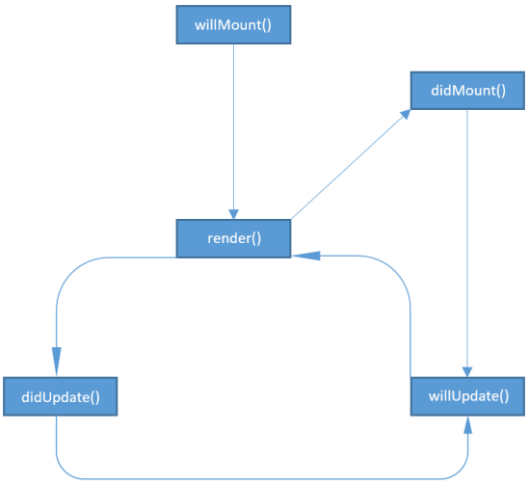
a) The Clock Component already includes four lifecycle methods (which?). In each of those methods ad a console.log statement like: `console.log("I am the componentDidMount");`

b) (yellow) Add an override for each of the missing lifecycle methods (like componentWillMount), with a message as sketched above.

Note: while you do a and b make sure to have your browsers Developer Console open so you can correct any mistakes you might make.

c) Clear the log in your Developer Console, and observe the messages printed (change the sleep time to a larger value to make it easier to read)

d) Provide a small explanation for each of the outputs, the purpose of the override, and when you observed it to print



- My observation of the life cycle methods can be described with the picture on the left. First method executed, when the program started and the constructor have been invoked already, was componentWillMount(), after that the render() method of the class Clock was called. Next printed text in the console came from componentDidMount(). From here now on, I have observed a closed cycle starts from the next method componentWillUpdate(), then render() have been called again, followed by componentDidUpdate(), after that back in the componentWillUpdate() and so on and so forth, inside infinite loop(cycle). /even now continues to print ☺ /

The purpose of the override:

- **componentWillMount()** – most of the developers suggest that this method is not very useful, as we can use the constructor for most of the setup, and mostly of the time will use this method at the very high level component, where we need to setup API for example(e.g. for database). Here we cannot change nothing in the DOM.
- **render()** – this method is a must have in our classes/components. Depends on the props and state, this method will return some type, but never changes component state, means this function needs to be "pure".
- **componentDidMount()** - now we have our component alive and loaded, and here we can load data from remote endpoint, like AJAX calls. We can provide all the setup that we need, because right before this method we have a DOM and can interact with it. Can also call **setState()** here, but it will trigger a re-rendering of the component.
- **shouldComponentUpdate()** – checks if the component's output is affected from the current change of props and state. It is not included and used in this example. Ract developers planning in the future to change it and make this function "informative" for React, and even after returning false, the next two methods in this bulleted list will be called, respectively result in re-rendering of the component.(for now, they will be skipped if the function returns false)
- **componentWillUpdate()** – if the preceding method returns true, for which we know that the props is changed or the state differs, here we can do something like a preparation before updating. Here we cannot call **setState()**.
- **componentDidUpdate()** – here we can implement logic close to what we did in the **componentDidMount()** function, like network requests, manipulating the DOM.

Events and ES6 arrow Functions

Before you start you should skim this article about Reacts way of handling events.

a) Add a new file `Toggle.js` into the src folder, and paste in the Toogle class from this example (remember to: import React, only include the class, and also export the Class)

b) Test the example, similar to how you did with the previous examples

c) In the constructor comment out this line, and explain the result:

```
this.handleClick = this.handleClick.bind(this);
```

Info:

JavaScript includes the "this" reference, spelled like in Java, but there are serious differences between the two. This can be tricky in several situations, especially if you expect JavaScript-this to behave like in Java.

ES6 (es2015) has tried to clean up the this-behavior inside classes, where you can get a behavior (almost) similar to Java if you use ES6 arrow functions.

d) Still with the line commented out as you did in c), rewrite the *handleClick* function into an arrow function as sketched below.

```
handleClick = () => {
    ...
}
```

e) Verify that the example works again

Info:

*These are the two ways you can (and must) use to deal with this for event handlers (**bind** and **arrow** functions). You can use whatever you prefer, but you need to know both strategies, since they are both used "out there"*

f) Answer the following questions (if you skip this part, you have wasted your time)

- What is the purpose of this line in the constructor in the example: `this.handleClick = this.handleClick.bind(this);`

- The bind method is often used to attach event listeners. By writing the preceding line, we want to express our intention to bound the object (passed with the keyword "this") to the handleClick function, so we can use this object inside the latter to change its state.

- How can we disable the default behavior of an event handler (for example prevent a submit handler to submit?)

- We can use the **preventDefault()** method, but not all events are cancelable.

- What is the benefit(s) you get from using arrow-functions in a ES6 class?

- Shortened format, therefore easier for the eyes.
- Solve problems with the keywords this and that , e.g. automatically binds this ,
- Implicit return, where we can use expression or statement body.

Using what we have learned up until now

a) Create a new project with create-react-app, as the start project for this exercise. Clean up the generated start code, similar to what we have done up until now.

b) Create a new JavaScript file and inside this file, create a class that extends React.Component (Remember to import React and to export the class similar to the previous examples)

c) Add some content into the `render()` method and make the application load the component and render the content

d) Create an image folder in the 'public' folder and put some images inside

e) In your new class component add an array of references to the images
Hint: ['image/img1', 'image/img2', ...]

f) Let your component loop through the array and display all the images in its render() method

g) (yellow) Style your component to give each image a visible border and a padding of 2px.

h) (yellow) Change the Component so that it will display only a single random image from the collection.

i) (yellow) Add a button + event handling, so that clicking it will change to a new random image on display

j) (yellow/red) Inspired by the way you used the setInterval method in the Clock Component Exercise, change the code to automatically display a new random image after a given amount of time (say 6 seconds)

k) (red) Create an ImageHolder component as a functional stateless component, that can have both an image and some text related to the image
Hint: use props to send information about the image source and the text to the sub component.

l) (red) Use ImageHolder component inside the image gallery component to show the image plus the text of each image in the gallery

Was that it, I want more ;-)

Great to hear. Continue with the egghead-video tutorial or just Google to find additional material.

Or even better, start to play around with what you have learned, and invent your own small cases. This is how you become a true React Expert :-)