CS615
Internet Solutions Engineering
Lab Week 03 (1%)

React JS

Tip: Consider the following sources for catching up, trying out and looking up details about JavaScript:

- https://javascript.info/ : A sequenced, integrated and detailed overview that also works well for looking up snippets;
- https://www.w3schools.com/js/ : The classic tutorial with try-it-yourself examples;
- https://codepen.io/ : A useful tool to test front-end code in general; and javascript in particular (instantaneous results for quick prototyping).

Part 1: Self Study (~1 h)

Visit https://www.w3schools.com/REACT/default.asp and work through the first few lessons of getting started, ES6, the render HTML, and components.

Note: If you do not understand a specific step in the later tutorials, review the respective section of W3Schools. Use W3Schools as a reference source in your work.

Part 2: Playing a bit with React JS (~30+ min)

In this part, you will install React JS and explore a few concepts of React JS: *(see next page of this pdf)*

Part 3: React JS Tutorial (~1+ h)

This additional tutorial shows you how to build a game as an example of a front-end application. Follow the tutorial on: https://beta.reactjs.org/learn/tutorial-tic-tac-toe

Make sure you personalise the application to show your name / student number.

Important: Upload a screenshot of your webpage to Moodle to receive the mark.

Playing a bit with ReactJS

In this tutorial, you will install React JS and explore some of its concepts. The tutorial is very initial and meant to get you started thinking about React components.

Step 0: Installing React and creating a generic app

Open the command line interface (CLI) and run:

```
npm install -g create-react-app
```

> If you get any error messages, update and upgrade your `npm` installation. If you use Homebrew (for Mac users), update that too).

If you have not already done so, create a code or project folder where you keep all your projects (e.g. ./projects). Using the CLI, create a code folder for your application, and run the following command:

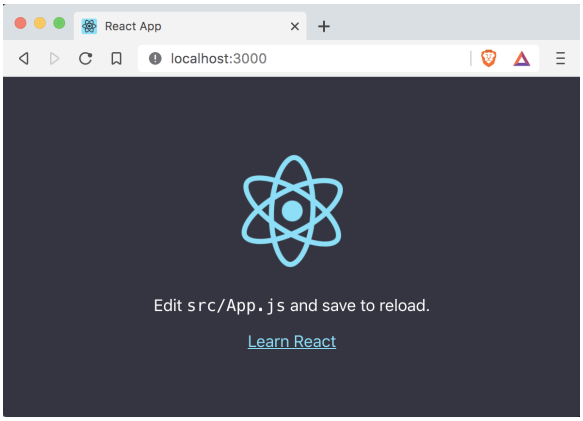
```
npx create-react-app note-app
```

An automated process now creates a new React app folder for the project (similar when creating a Node.js project) and installs packages and dependencies.

Run what you just created by changing into the folder and running the app:

```
cd reactive-notes
npm start
```

Per default, React applications are running at http://localhost:3000. Check it!



In the next step, we will be looking at all the folders and files a bit and build a minimal example that shows how these parts work together.

Step 1: Playing a bit with React

> **Tip:** It helps if you develop code by having your editor on one side of the screen and your browser on the other. Changes will then be shown immediately in your browser. Open your editor and link the react folder as a project folder so that you can see all

folders and files and you have a way to create files and folder, move them and rename them if its needed. Open a console (or a console plugin in your editor to make things easier).

Overview and cleanup

Your generic application from earlier has the following file structure:

first-app/
first-app/node_modules
first-app/public
first-app/src
package-lock.json
package.json
README.md

In the public folder, you find an index.html, plus logos and manifest files. In the src folder, you find javascript and css (re)sources. The main folder contains the configuration file package.json and a readme file.

We learn best by breaking things and by starting minimal to see how things sticks together. Remove all files in public and src. You should now see your application break in the browser because index.js has gone.

Start simple

Now, create a new index.html in /public and a new index.js in /src. You should now see a blank page.

The way we would write a HTML page is by writing HTML in a index.html. Let us write a bit of HTML into index.html and run the react.

```
<div><h1>MyComponent</h1></div>
```

You should now see MyComponent on the screen. React finds the HTML file and runs it, nothing spectacular. We don't really use anything about React yet.

Start injecting data

What we would like to do is to inject content into the HTML code. So remove the content from the div and give it a unique identifier.

```
<div id="root"></div>
```

You should now get an empty page again, because there is nothing to display. No errors though.

Also, open index.js file, and add some basic React code:

```
ReactDOM.render(
  <h1>MyComponent</h1>,
  document.getElementById('root')
);
```

This basically wants to inject the header into the div in the HTML file. It uses the DOM tree to find the node with the id that you assigned (root).

You should now get an error that complains about ReactDOM not being defined. Note that you also get the error displayed in your browser.

Resolve this by adding these two import statements:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

Now you should see MyComponent being displayed. React JS successfully injected the HTML into the div tag in the HTML file.

Well, this is nice but maybe not as valuable. Normally, you have data coming from databases or elsewhere. Let us pull this apart a bit.

Add a variable after the imports of your .js file:

```
const name = "MyComponent";
```

and inside the renderer, substitute MyComponent with {name}. This all should now look like this:

```
import React from 'react';
import ReactDOM from 'react-dom';

const name = "MyComponent";

ReactDOM.render(
  <div>{name}</div>,
  document.getElementById('root')
);
```

The variable is now entered at the {name} where we previously used our data directly. This is JSX, React's own template engine / tool.

Components

Having HTML inside of the render function is not very flexible. What you normally do it separating parts of useful HTML into so-called components. Let turn our component here into a React component.

Remove the `<div>` from the renderer and instead write a new function:

function MyComponent(){}

In the renderer, you just enter <MyComponent /> where you first had your .

The function is used as a component and consists of javascript code while returning the template HTML code that we used earlier. We put out old HTML template code there so that it looks like this:

```
function MyComponent(){
  // js
  // template
  return (
    <h1>MyComponent</h1>
```

```
  );
}
```

You can see the return value containing the template HTML code. We now use the component like HTML in our renderer instead.

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

This means that our renderer now uses the function like a component.

The interesting thing about components that that they are modular and can be reused. Try this

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

and you can see how easily components can be reused. We can of course also bring in variables as we did before:

```
const name = "MyComponent 2.0";

function MyComponent(){
  // js
  // template
  return (
    <h1>{name}</h1>
  );
}
```

You can also use javascript variables inside the component. Try this:

```
const name = "MyComponent 2.0";

function MyComponent(){
  const newName = "Totally" + name;
  // template
  return (
    <h1>{newName}</h1>
  );
}
```

By now you have an idea about how to create a project, and dissect some of its structures.