MINISTRY OF EDUCATION AND SCIENTIFIC RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

React App

Hands on REACT Framework

Ioan Salomie
Cristina Pop

Tudor Cioara
Marcel Antal

Ionut Anghel
Claudia Antal

Contents

1. Introduction

The objective of this laboratory work is to teach you the basic steps for building a React application. React is a JavaScript library used for building single page applications by means of reusable UI components.

To get started with React you need to know the basics of the Hyper Text Markup Language (HTML), JavaScript, HTML Document Object Model (DOM) and Cascading Style Sheets (CSS) which will be introduced in Sections 2-5.

Section 6 will detail the steps required to build a basic React application that communicates with the Spring 3-tier REST Service application (i.e. *spring-demo*) introduced in the Hands on Spring Rest Framework document.

2. Hyper Text Markup Language

HTML is a standard markup language which is used to describe the structure of a Web page by means of HTML elements. An HTML element is represented by specific start and end tags, can have (i) attributes whose purpose is to provide more information to the HTML element, and (ii) a content specified between the start and end tags. Figure 1 illustrates an example of HTML document describing the content of a Web page that will display a list of students and a link to navigate to a faculty department's Web site.

```
<!DOCTYPE html>
<html>
    <head>
        <title>My first HTML</title>
        <style>
            td, th {
                border: 1px solid #dddddd;
                text-align: left;
                padding: 8px;
            }
        </style>
    </head>
    <body>
        <h1>List of students</h1>
        <table id="tableStudents" style="border: 1px solid #dddddd;">
            <tr>
                <th>Name</th>
                <th>Group</th>
            </tr>
            <tr>
                <td>John Doe</td>
                <td>30431</td>
            </tr>
            <tr>
                <td>Annie Smith</td>
                <td>30431</td>
            </tr>
        </table>
        <a href="...">Navigate to the department Web site</a>
    </body>
</html>
```

List of students

Name	Group
John Doe	30431
Annie Smith	30431

Navigate to the department Web site

Figure 1: Example of HTML document and the corresponding displayed Web page

Table 1 describes the HTML elements part of the HTML document in Figure 1.

Table 1: Description of the HTML elements from Figure 1

HTML element	Description
<!DOCTYPE html>	Document type declaration - defines the document to be HTML5. Compulsory!
<html>	Root element of the HTML document. Compulsory!
<head>	Container for the metadata (e.g. title, styles, scripts) of an HTML document.
<title>	The title of the document which will be displayed in the browser's title bar or in the page's tab.
<style>	Defines the style information for the current HTML document. For example, in this case all table columns will have a solid border 1px thick and colored in light grey, the text will be left aligned and the space between the content of a cell and its border will be of 8px.

	Note: the style of an HTML element can also be defined inline with the HTML element meaning that the style will be applicable only to that element – see how the style of the HTML table is defined in Figure 1.
<body>	Encloses the part of the HTML document that will be displayed in the browser.
<h1>	Defines a large heading with the text "List of students".
<table>	Defines a table with a header, rows and columns with content. Note: • The "id" attribute can be used in Web page manipulations operations to refer to this table. All HTML elements can have an ID. • The "style" attribute is used to add styles to this table.
<tr>	Defines a row of a table.
<th>	Defines the header of a table.
<td>	Defines the column of a table.
<a>	Defines an HTML link. Notice the "href" attribute used to indicate the URL of the destination.

HTML defines other elements too, such as paragraphs (<p>), images (<img>), buttons (<button>), lists (<ul>), etc. For a complete list of HTML elements check this link.

Inside an HTML document you can also define a form, by using the <form> element, for collecting user input that will be sent to a server for further processing. The <form> element can contain elements such as text input fields (<input type="text">), radio buttons (<input type="radio">), checkboxes (<input type="checkbox">), submit buttons (<input type="submit">), or clickable buttons (<input type="button">), labels (<label>), etc. For a form element you can specify the resource that will process the user input collected in the form by using the **action** attribute, and the HTTP method to be used when sending the form data within the method attribute. Figure 2 illustrates an example of using an HTML form to collect the information about a student:

```
<!DOCTYPE html>
<html>
<body>

    <h2>Add student</h2>

    <form action="/student.php" method="POST">
        <label for="txtName">Name</label>
        <input type="text" id="txtName"></input>
        <label for="txtGroup">Group</label>
        <input type="text" id="txtGroup"></input>

        <button type="submit">Add student</button>
    </form>

</body>
</html>
```

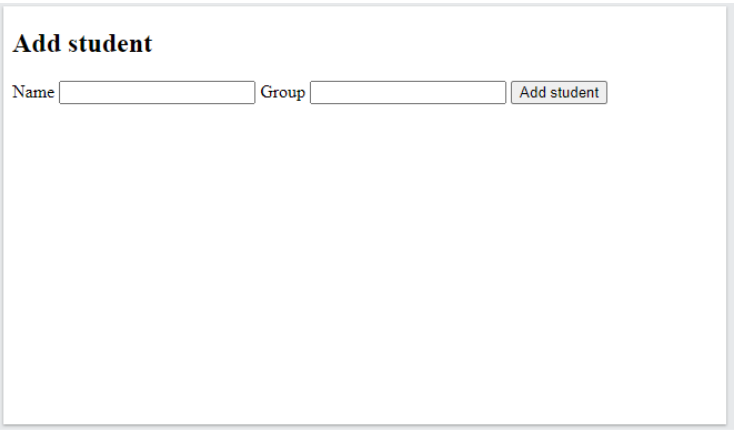


Figure 2: Example of defining an HTML form

3. JavaScript

JavaScript is the programming language of the Web which allows a full integration with HTML and CSS and is supported by all popular browsers [11-12]. Using JavaScript, you can write programs called **scripts** that can be inserted in the HTML of a Web page and executed once the page loads in the browser.

With the emergence of **JavaScript engines** (i.e., computer programs that execute JavaScript), JavaScript can be executed on any device that has such an engine installed. The browsers have JavaScript engines embedded; for example, V8 is found in Chrome and Opera, while SpiderMonkey in Firefox. The basic processing steps of a JavaScript script in a JavaScript engine are illustrated in Figure 3.

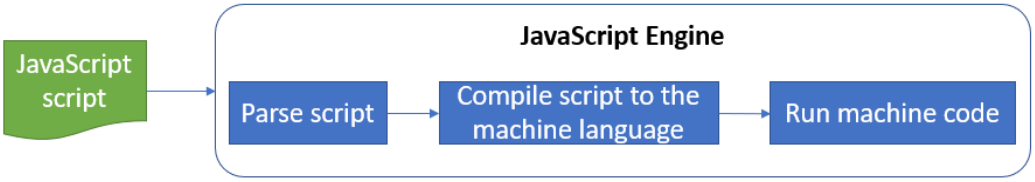


Figure 3: JavaScript Processing Steps in a JavaScript Engine

JavaScript's capabilities depend on the environment it's running in. For example, **Node.js**, which is a JavaScript runtime built on V8 engine, supports functions that allow JavaScript to read/write arbitrary files, perform network requests, etc. Generally, with JavaScript you can **manipulate Web pages by adding new HTML**, **changing the content or the styles**, **reacting to user actions**, **sending requests over the network** to remote servers, **getting and setting cookies**, remembering the data on the client side in the "**local storage**".

Recently, many languages providing several features appeared that are converted to JavaScript, such as TypeScript, CoffeeScript, Brython, etc.

JavaScript can be inserted almost anywhere into an HTML document using the <script> tag.

Table 2 overviews the main concepts of the JavaScript language.

Table 2: Main concepts of the JavaScript language (adapted from [11])

Concept	Description	
Statements	Definition	Syntax constructs and commands that perform actions.
	Example	`alert('Please insert the name of the student');`
Comments	Definition	Informative text that can be placed anywhere within the script without affecting the functionality of the script.

	Example	`// One line comment`
		`alert('Please insert the name of the student');`
		
		`/* Multiline comment.`
		`Multiline comment.*/`
		`alert('Please insert the name of the student);`

Variables	Definition	A variable is created with the "let" keyword and is used to store information. The assignment operator can be used to put data into a variable. You can put any type in a variable – a variable can be a string, then a number, and so on. !!! *JavaScript is case sensitive*!!!

	Example	`let product;`
		`product = 'Apple';`
		OR
		`let product = 'Apple';`

Constants	Definition	Variables declared using "const" which cannot be reassigned.
	Example	`const aConstant = 'I am a constant';`
Data types	Definition	A value in JavaScript is always of a certain type. JavaScript defines 8 basic data types in JavaScript: Number, BigInt, String, Boolean, the "null" value, the "undefined" value, object type, symbol type. ***JavaScript is called "dynamically typed", because even if it has data types, the variables are not bound to any of them.*** The **typeof operator** can be used to return the type of an argument.

	Example	`// The assignments bellow do not generate any errors`
		`let message = "John";`
		`message = 123;`

Interaction: alert, prompt, confirm	Definition	**alert** - shows a message and waits for the user to press "OK". **prompt** - shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel. **confirm** - shows a modal window with a question and two buttons: OK and Cancel. The result is true if OK is pressed and false otherwise.

	Example	`alert("Hello");`
		`result = prompt(title, [default]);`
		`let isStudent = confirm("Are you a student?");`
		`alert( isStudent ); // true if OK is pressed`

Basic operators	Definition	Addition +, Subtraction -, Multiplication *, Division /, Remainder %, Exponentiation ** String concatenation with binary +, Assignment =, Increment ++, Decrement --
Bitwise operators	Definition	AND(&), OR (|), etc.
Comparisons	Definition	Greater/less than: a > b, a < b. Greater/less than or equals: a >= b, a <= b. Equals: a == b. Not equals: a != b
Conditional branching: if, '?'	Example if(…) statement	`let year = prompt('When were you born?', '');` `if (year == 2015) alert( 'You are young!' );`
	Conditional operator '?'	`let result = condition ? value1 : value2;`
Logical operators	Definition	|| (OR), && (AND), ! (NOT).

Loops: while and for	Definition - while	`while (condition) {`
		`  // loop body`
		`}`
		
		`do {`
		`  // loop body`
		`} while (condition);`

	Definition - for	`for (begin; condition; step) {`
		`  // ... loop body ...`
		`}`
		
		`for (let i = 0; i < 3; i++) {`
		`  alert(i);`
		`}`

The "switch" statement	Definition	`switch(x) {`
		`  case 'value1':  // if (x === 'value1')`
		`  ...`
		`  [break]`
		
		`  case 'value2':  // if (x === 'value2')`
		
		`  [break]`
		
		`  default:`
		`  ...`
		`  [break]`
		`}`

Functions	**Definition**	They are created as separate statements in the main code flow and they can be invoked anytime regardless their place in the script. **`function name(parameters) {`** **`  ...body...`** **`}`**

	Example	`function addition(number1, number2){`
		`  return number1+number2;`
		`}`

Function expressions	Definition	Represent another means for creating functions. The function is created when the execution reaches it and is available only from then on.

	Example	`let addition = function(number1, number2) {`
		`  return number1 + number2;`
		`};`

Callback functions	Definition	Function passed as arguments to other functions that will be executed in certain conditions.

	Example	`/* if the condition evaluates to true then the function action1 will be executed, otherwise function action2 will be executed*/`
		
		`function execute(condition, action1, action2){`
		`  if(condition) action1().`
		`  else action2();`
		`}`
		
		`function action1(){`
		`…`
		`}`
		
		`function action2(){`
		`…`
		`}`

Arrow functions	Definition	Represent another means for creating functions in a more simple and concise way.

	Example	Syntax for creating an arrow function with N arguments which evaluates an expression and returns the result of the evaluation:
		`let func = (arg1, arg2, …, argN) => expression`
		Equivalent to:
		`let func = function(arg1, arg2, …, argN){`
		`  return expression;`
		`}`
		!!!*In case of multiline arrow functions, enclose the expressions or statements in curly braces and include an explicit return keyword*!!!

Objects	Definition	Objects are used to store keyed collections of various data and more complex entities. An object can be created with figure brackets {…} with an optional list of properties. A property is a "key: value" pair, where key is a string (also called a "property name"), and value can be anything. Use t**he "for…in" loop** to go through the keys of an object: for(key in object) {…}

	Example	`let student = {`
		`  name: "John Doe",`
		`  group: "30431"`
		`}`
		// Property values are accessible using the dot notation: student.name.

Arrays	Definition	An array can store elements of any type and can be declared as:
		`let array = new Array();`
		OR
		`let array = [];`
		
		Loop an array:
		`for (let i=0; i<array.length; i++){alert(array[i]);}`
		OR
		`for (let elem of array){alert(elem);}`

4. HTML Document Object Model

When a Web page is loaded, the browser creates a Document Object Model (DOM) of the page which is represented as a tree of nodes (i.e. objects), each node corresponding to an HTML element in that page. Additionally, HTML DOM defines properties, methods and events for all HTML elements. HTML DOM is also an API for JavaScript allowing it to manipulate (i.e. add, change, remove) the HTML elements, their attributes, CSS styles and HTML events on one hand, and react to HTML events on the other hand. Figure 4 illustrates the graphical representation of the Document Object Model for the HTML document in Figure 1.

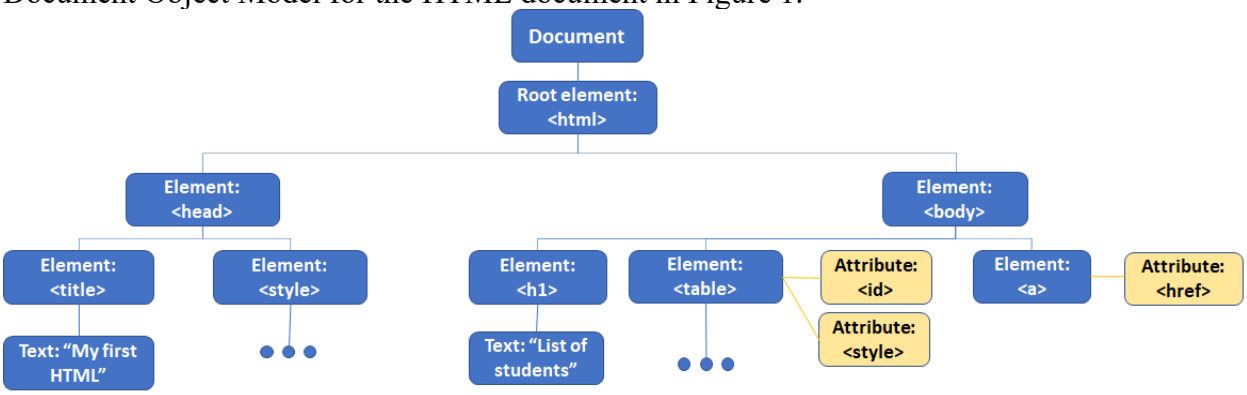


Figure 4: Document Object Model representation for the HTML document in Figure 1

All operations on the DOM start with the *document* object. Then, in order to manipulate HTML elements with JavaScript we first have to find the elements – HTML DOM provides several means to find elements such as by:

- **ID** – the example below finds the HTML element with the ID equal to "tableStudents"

```
var tableElement = document.getElementByID("tableStudents");
```

- **Tag name** – the example below finds all "td" elements

```
var tableElement = document.getElementsByTagName("td");
```

and others.

The DOM can also be used to change the value of HTML elements' attributes. The syntax for changing an HTML element by one of its attributes is:

```
element.attribute = new value
```

The syntax for using DOM to change the style of an HTML element is the following:

```
element.style.property = new style
```

Using DOM you can add event handlers:

```
element.onClick = function(){code}
```

All DOM nodes can generate events (e.g. mouse events, keyboard events, form element events, document events, CSS events). In order to handle events, an **event handler** must be defined which runs some JavaScript code in case of user actions. Figure 5 shows an example of how the onclick event for a button can trigger the execution of a JavaScript function that displays an alert with the name that has been inserted.

```
<!DOCTYPE html>
<html>
<head>
    <script>
        function showName() {
            alert(Hellodocument.getElementById('txtName').value);
        }
    </script>
</head>
<body>

    <h2>Add student</h2>

    <div>
        <label for="txtName">What is your name?</label>
        <input type="text" id="txtName"></input>
        <button type="submit" onclick="showName()">Add name</button>
    </div>

</body>
</html>
```

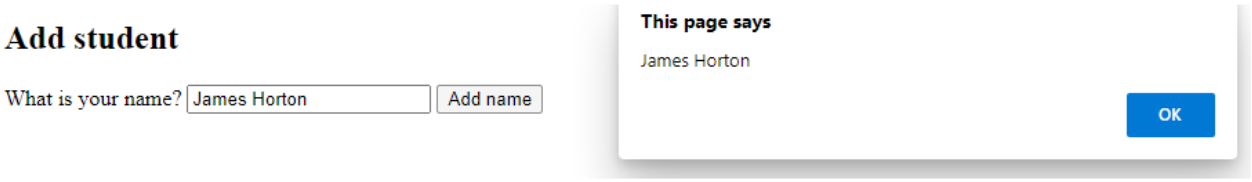


Figure 5: Example of handling the onclick event for a button

The most common HTML events that can occur are the following [12]:

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

5. Cascading Style Sheets

Cascading Style Sheets (CSS) is a rule-based language that allows us to specify how the HTML elements will look like (in terms of layout, color, size etc.) when they are displayed in the browser.

A CSS rule consists of the following elements (see an example in Figure 6) [Ref]:

- **Selector** – used to select the HTML element that will be styled. The selectors in Figure 6 are:
  - **"h1"** – in this case the selector applies the color red and sets the font size as 5 times greater than the current font size.
  - **".special"** – in this case the selector applies the color blue to all elements with the class *special.*
  - **"#customized"** – in this case the selector sets the base font of the element with the id *customized* to 16-pixel tall Lucida Grande or one of a few fallback fonts.
- **Declarations** – one declaration is composed of a property and a value and specifies what value should be given to the property of an element.

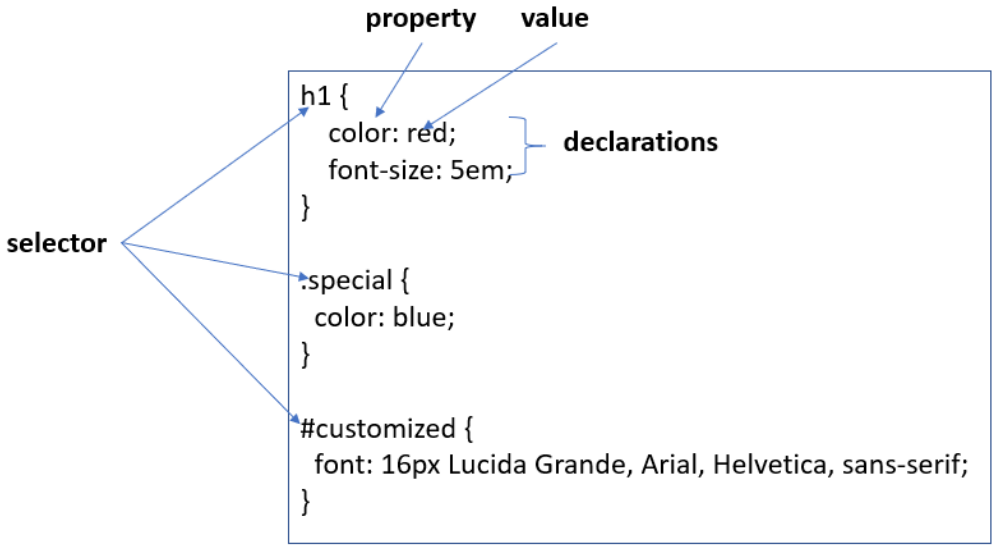


Figure 6: CSS file fragment illustrating the types of selectors that can be defined (adapted from Ref)

For a complete list of CSS selector please check this link.

The styling of HTML elements can be also done **based on their location** in the HTML document (e.g. *li em* means a selector that will select any <*em*> element that is located inside an <*li*> element), or **based on state** (e.g. *a:visited* means a selector that selects an <*a*> anchor element that is visited).

Beware of CSS rules collision which occurs when multiple types of selectors are used in order to control the appearance of HTML elements. The order of priority is the following:

1) #id selectors
2) .class selectors
3) tag selectors

CSS rules can be applied to an HTML document using one of the following methods:

1) Link the CSS file from the head of the HTML document – ***the recommended method***!

Figure 7: Example of linking the CSS file from the head of the HTML document

2) Inline, using the style *attribute* inside an HTML element:

```
<table id="tableStudents" style="border: 1px solid #dddddd;">
```

Figure 8: Example of applying a CSS rule inline

3) Internally – by using a *style* element in the <head> section:

```
<style>
    td, th {
        border: 1px solid #dddddd;
        text-align: left;
        padding: 8px;
    }
</style>
```

Figure 9: Example of applying a CSS rule internally

6. Hands-on React

React is a JavaScript library developed by Facebook that is used to build complex user interfaces out of independent and reusable components [5]. The name *React* comes from the fact that this technology keeps a lightweight representation of the Document Object Model (DOM) in memory, called the *virtual DOM* - virtual representation of the user interface [7], and once React notices that a component changes it updates the component in the *real DOM* – this is how React reacts to the components' state changes [1].

As React's only objectives are to render the view and synchronize it with the state, in order to build complete applications, the developer must use other external libraries (e.g. libraries for routing, for calling HTTP services, etc.) [1]. This feature may be considered an advantage as it allows the developer to choose the preferred libraries thus not being forced to use one bundled in a framework such as Angular [1].

The project example from this laboratory work is a skeleton for a React application that can be used to communicate with the 3-tier REST Service application developed in Spring and introduced in the Hands on Spring Rest Framework document.

6.1. Prerequisites: Install and configure the following resources

Resource	Link	Description
Node.js and Node Package Manager (NPM)	Link	Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. NPM is a package manager for Node.js and is installed with Node.js – can be used to install 3rd party libraries. **Note**: After installing the last version of Node.js, check that Node and NPM are installed using the commands: - node –v - npm –v
IntellijIdea	Link	IDE that allows the development of JavaScript projects as well. Note: WebStorm [Link] , or Visual Studio Code [Link] can be considered an alternative IDEs
NGINX	Link	NGINX is a lightweight and high performance Web server

6.2. Get the Project

To download and configure the example follow the steps described below:

1) Setup GIT and download the project from
   https://gitlab.com/ds_20201/react-demo
   - Create an empty local folder in the workspace on your computer
   - Right-click on the folder and select Git Bash
   - Write the command:
     - *git clone https://gitlab.com/ds_20201/react-demo.git*

2) In IntelliJIdea check if the *JavaScript and TypeScript* bundled plugin is enabled by going to the Plugins page as follows: File->Settings->Plugins (see Figure 10):

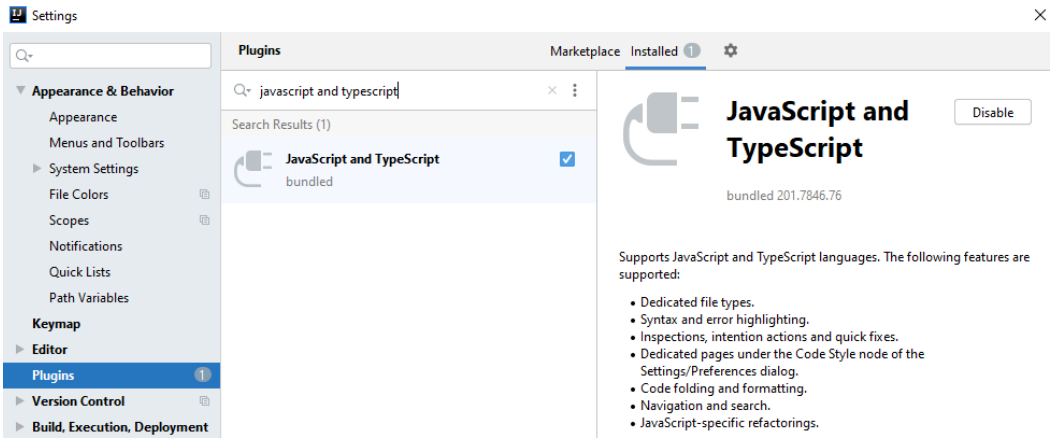
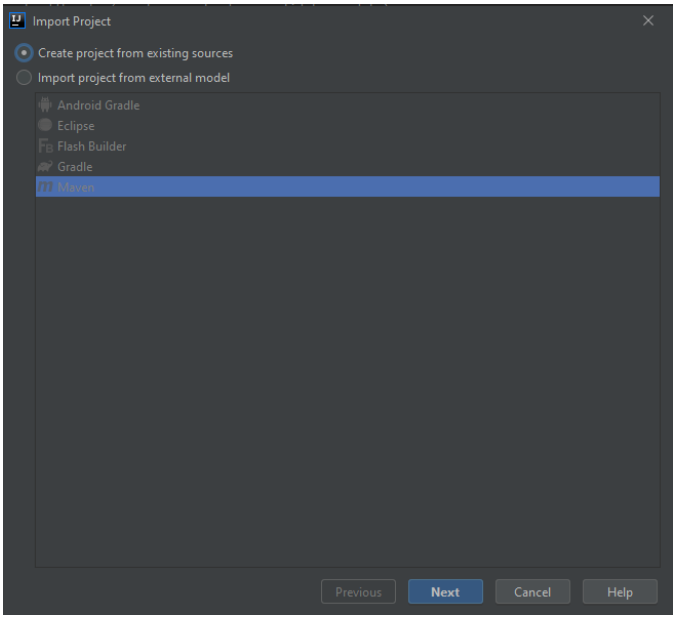


Figure 10: IntelliJIdea Plugins administration page

If the *JavaScript and TypeScript* bundled plugin is not installed, search for it in the JetBrains Plugin Repository (check the Marketplace tab in Figure 10) and install it.

3) Search for the location of the project on the disk, enter the name of the project react-demo and click OK.



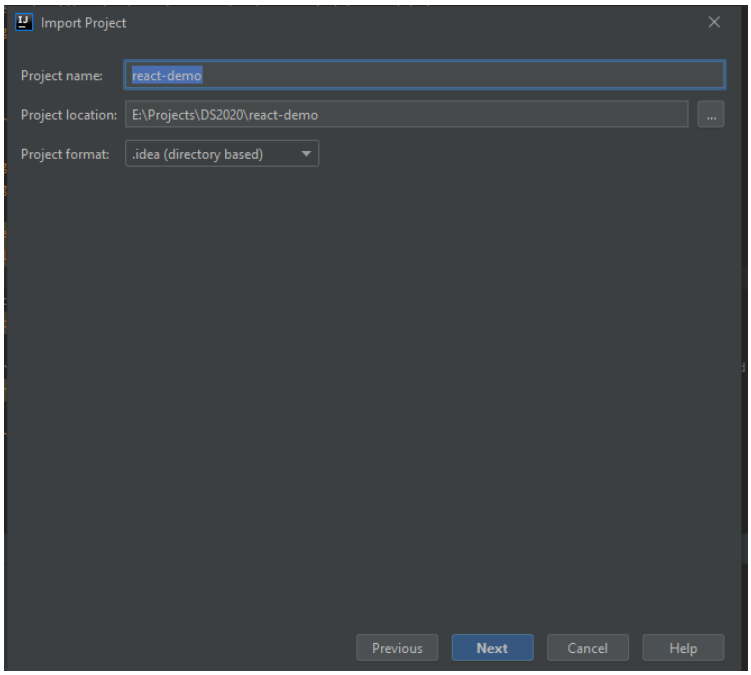


Figure 11. Steps to import project

When opened in IntelliJIdea, the project has the structure from Figure 12. All the components are detailed in Section 6.2.

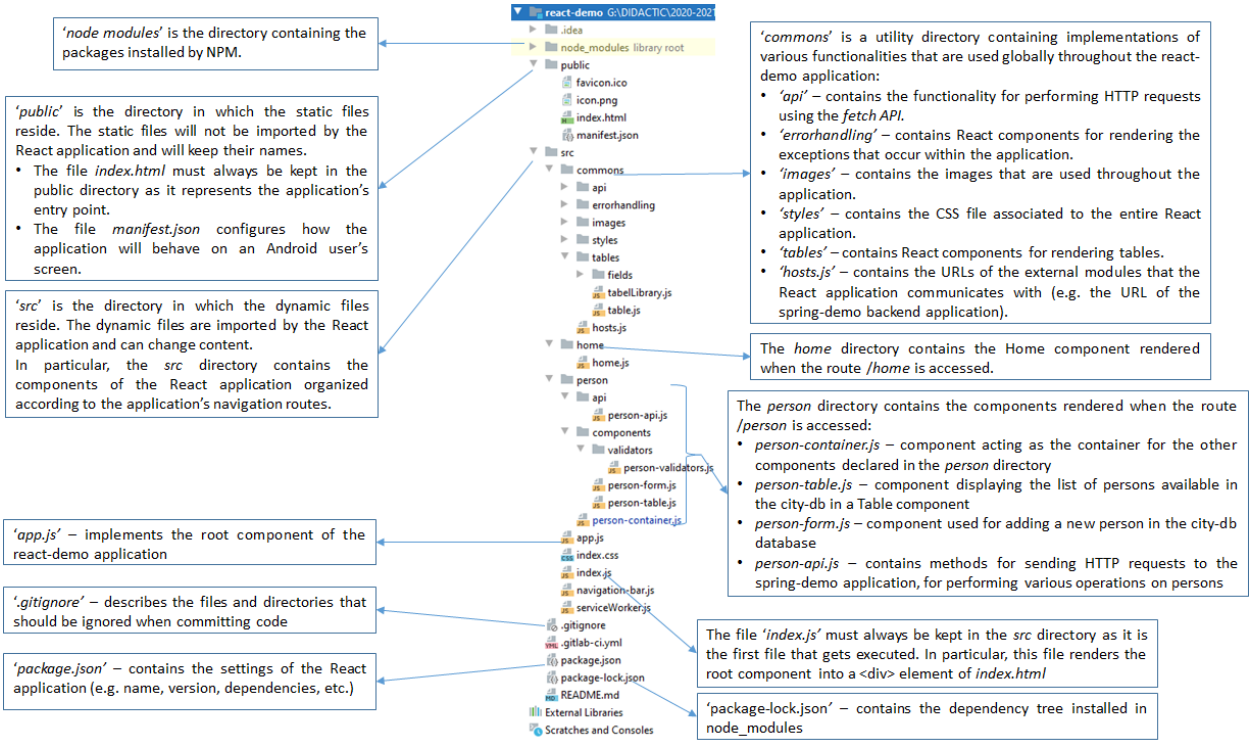
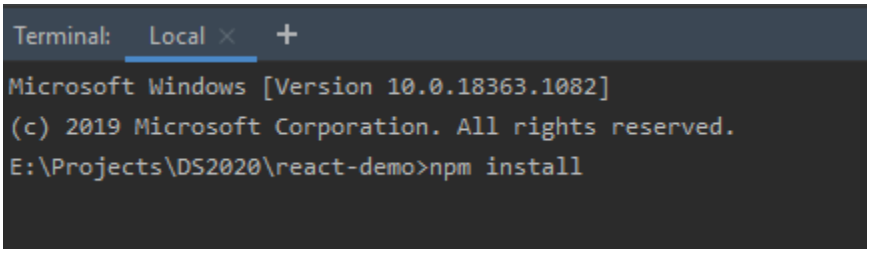


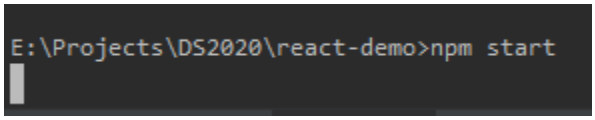
Figure 12: React project structure

To properly run the project, follow the next steps:

1) From IntelliJIdea open the terminal from View->Tool Windows-> Terminal.
2) Download the packages writing the following command in the terminal:
   npm install



3) Run the project: in the IntelliJIdea terminal write the command: *npm start*



   The command will run the development server on port 3000.
   Then open *http://localhost:3000/* to see the application.
4) Upon successful execution, you should access the react-demo web application from the browser (see Figure 13).

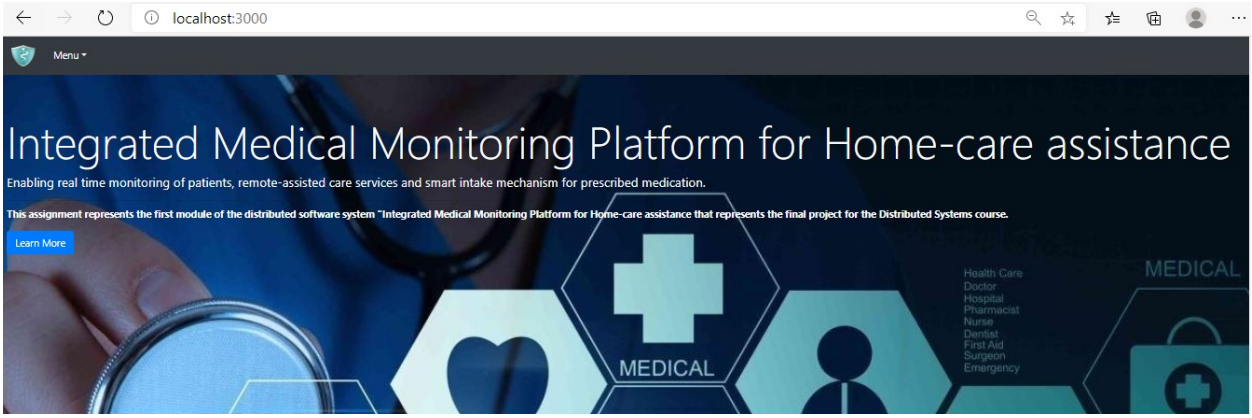


Figure 13: Welcome page of the react-demo Web application

6.3. Testing the application

1. Insert several persons in the table persons from the city-db database;
2. Run the spring-demo application as presented in Laboratory 3-Tier Rest Services;
3. Run the react-demo application as explained section 6.2 of this Hands-On;
4. Access the following link in the browser: http://localhost:3000/
5. Click on the Persons option from the menu. You should be able to visualize the persons that populate the person table from the city-db database.

6.4. Project Conceptual Architecture

The conceptual architecture of the react-demo application is presented in Figure 14. The application consists of a *COMMONS* module and a set of modules, one for each route defined in the application. The *COMMONS* module includes functionality that is used by all the other components such as functionality for (i) performing HTTP requests, (ii) implementing error handling components, etc. Each module defined for each route consists of a component container, a set of child components and an API used to send specific HTTP requests to the spring-demo application. In particular, the react-demo application communicates with the spring-demo application through JSON (JavaScript Object Notation) objects. JSON is a syntax which can be used to store and exchange data.

NOTE: *when designing a React application, the first step is to decompose the design of the user interfaces in a set of independent components that can be further assembled to form the final user interfaces [2].*

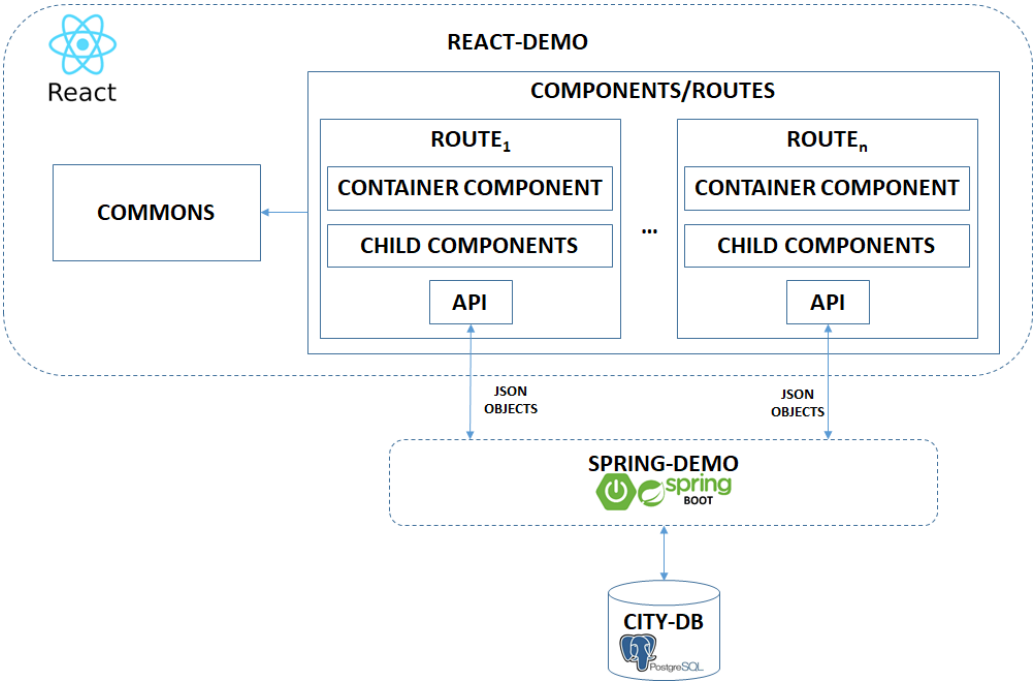


Figure 14: Conceptual architecture of the react-demo application and its integration with 4the spring-demo application

6.5. Project Functionality and Implementation Details

6.5.1. Main Project Components

The **index.html** file (see a fragment in Figure 15) located in the *public* directory of the React application is the first file that the browser will load when we write the URL (i.e. http://localhost:3000/) of the React application in the browser. **Note**: *The project will build only if the **index.html** file exists and is located in the public directory*.

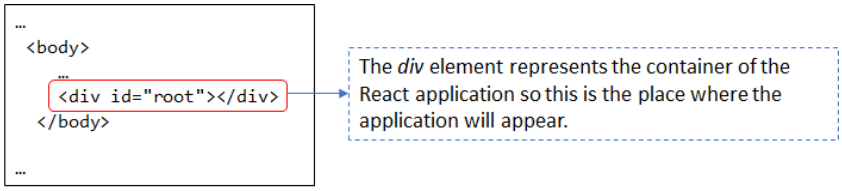


Figure 15: Fragment of the index.html file from the *public* directory

The file **index.js** (see a fragment in Figure 16) located in the *src* directory is the first file that gets executed – it can be considered as the entry point of the application. Like the **index.html** file, the file **index.js** must exist in the *src* directory to allow the project to build. If we analyze the code fragment from Figure 16 we can notice that the main task of the **index.js** file is to render the App component in the 'root' div in the DOM.

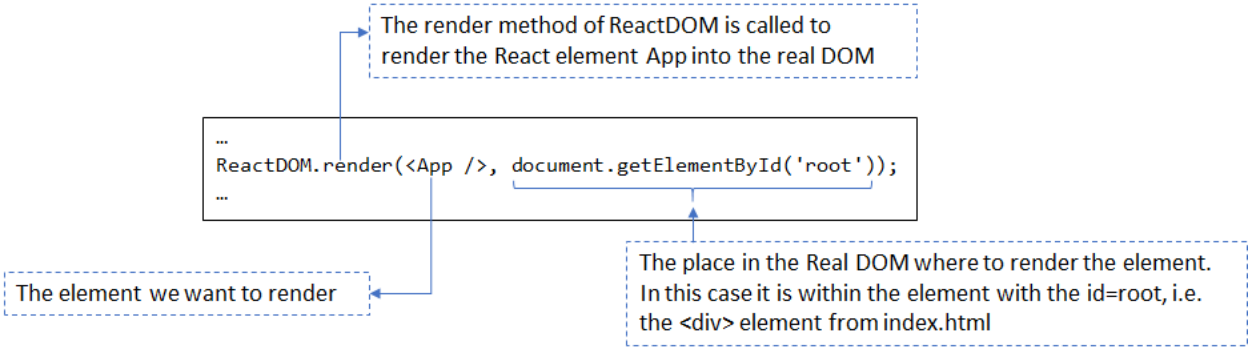


Figure 16: Fragment of the index.js file

The App component is implemented in the file **app.js** (see Figure 17). This component represents the root component that contains the other child components of the application.

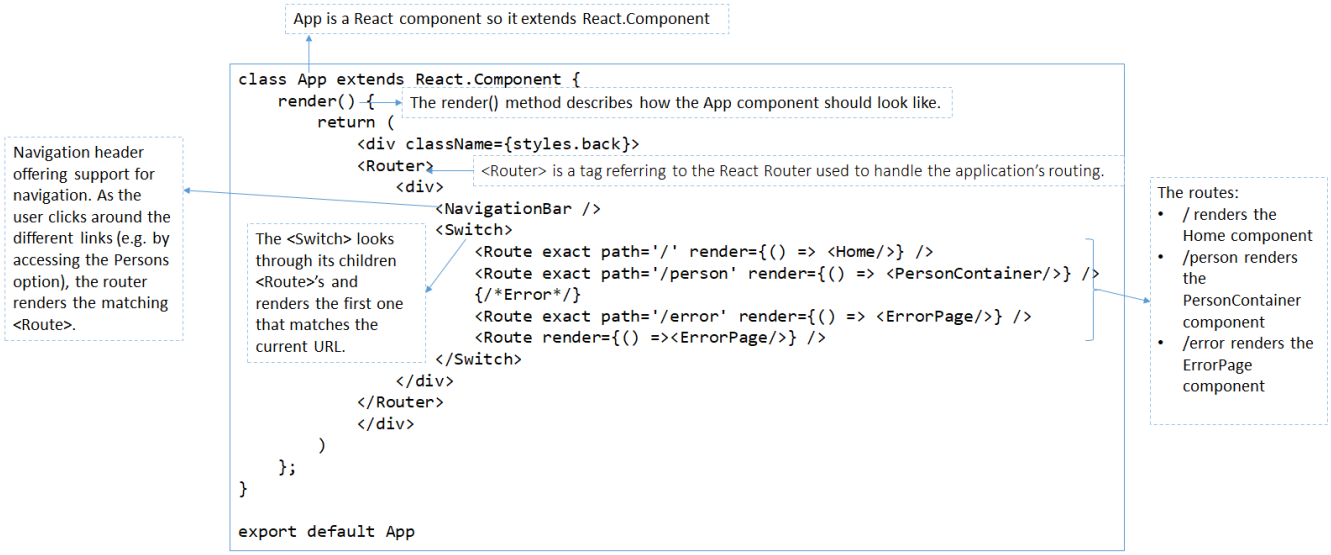


Figure 17: Fragment of the App.js file

The App component's render method returns a JSX fragment describing how the user interface will look like (see Figure 18). The App component uses a React Router to implement the routing of the application. In this demo, the React Router is already installed, however if you want to include it in another project you can install it from the public npm registry using npm. In the App component, the router handles 3 "pages" namely the *home* "page", the *persons* "page" and the *error* "page".

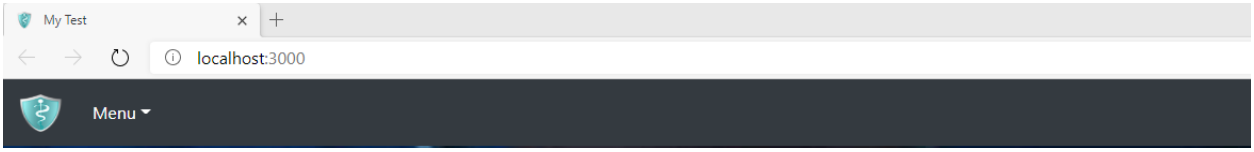


Figure 18: The part of the UI displayed by the App component

GOOD TO KNOW

1) What is a React component?

A React component is a JavaScript class extending **React.Component** having:

- a **state property -** represents the data we want to display when the component is rendered; it may change over time, and it should be a plain JavaScript object. !!! Never modify *this.state* directly – use the *setState*() method!!!
- a **render method** (this is the only compulsory method). The render method describes how the component should look like. More specifically, the render method outputs a **React Element** which is a simple JavaScript object that represents a DOM element in memory.
- **props** – the parameters that a React component can take in**.**

Note:

- A React component can be also defined as a JavaScript function – in this case the component is called a **function component.**

```
function FunctionComponent(props){
    return <div>
        <h1>Hello {props.name}! I am a function component!</h1>
    </div>
}
```

- The values that should not be rendered can be defined as fields of the component

2) What is JSX?

JavaScript XML (JSX) is an XML-based extension of JavaScript that is used for describing how the user interface will look like. JSX produces React elements that are further translated into objects used to build and update the DOM [8]. JSX expressions can be returned from functions, can be passed to functions or can be used as the value for a constant or variable.

3) What is the difference between .js and .jsx file extensions?

The file implementing a React component can have either the extension js or jsx – the difference between the two extensions is the fact that jsx facilitates better code completion.

Figure 19 illustrates how the App React component is rendered in the *root* from the index.html file.

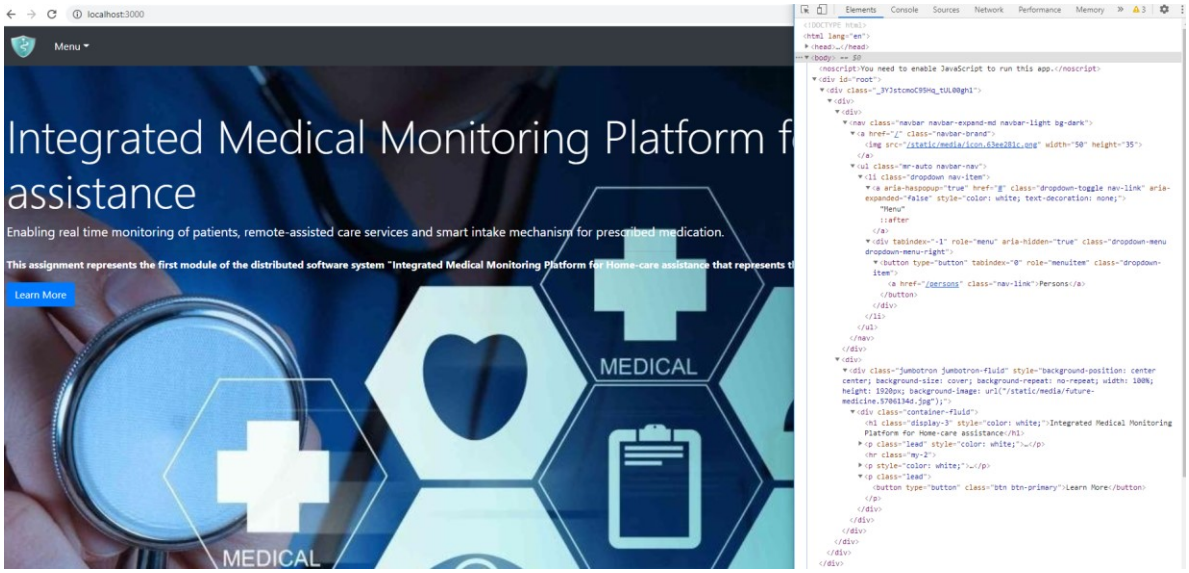


Figure 19: Example showing how the App component is rendered in the 'root' div from index.html

It can be noticed in Figure 19, that besides the App component, the user interface also includes the Home component. This component (see Figure 20) is rendered each time the react-demo application is launched, or when no other path is specified after /. The component uses other components (i.e. Jumbotron; Container; Button) imported from the *reactstrap* library. In this demo, *reactstrap* library is already installed, however if you want to include it in another project you can install it from the public npm registry using *npm install --save reactstrap react react-dom*.

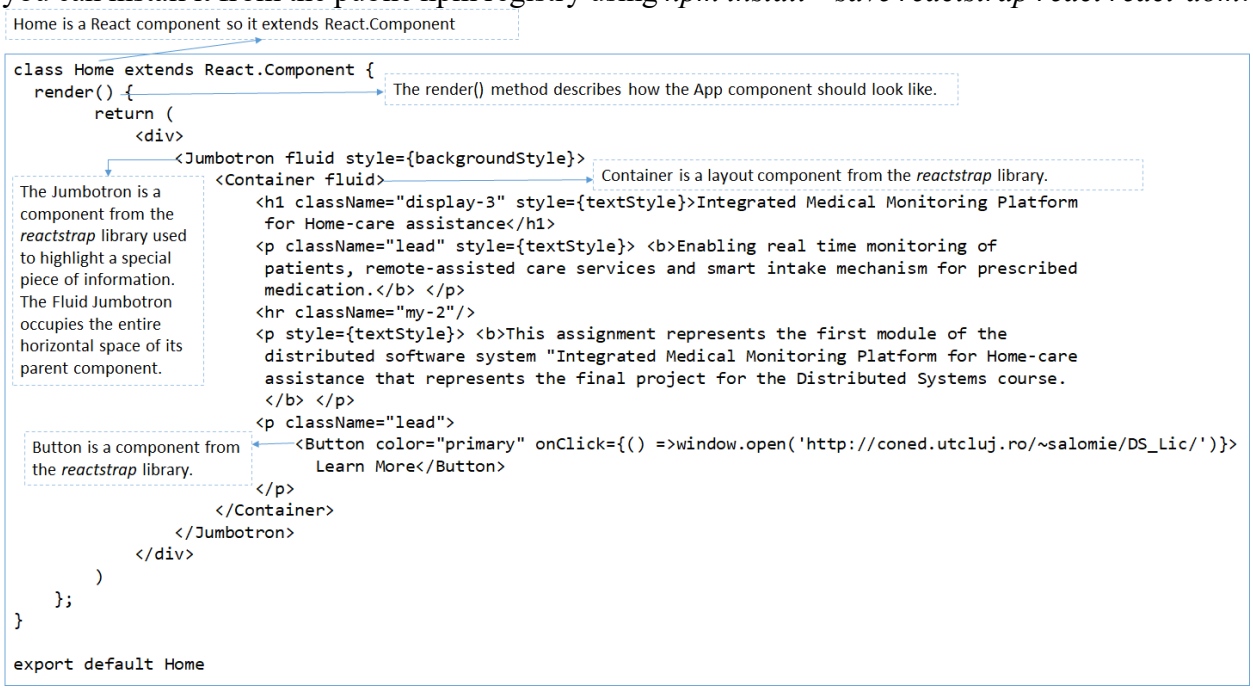


Figure 20: Fragment of the Home.js file

6.5.2. Project Functionality – Retrieve and Display Scenario

This sub-section will detail the implementation of the functionalities for retrieving and displaying the list of persons stored in the city-db database. To have access to the data persisted in the city-db database, the front-demo application will interact with the spring-demo application.

An overview of the flow of operations is presented in Figure 21 and detailed below:

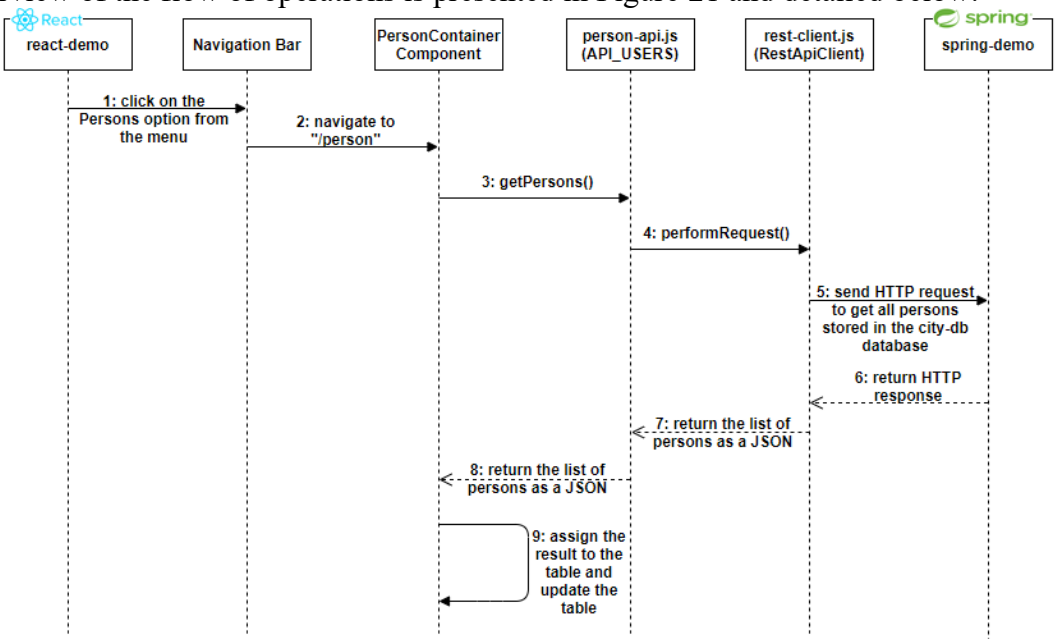


Figure 21: Overview of the flow of operations performed to retrieve and display the list of persons stored in the city-db database

Steps 1-2: The PersonContainer component (see Figure 22) is rendered each time the option Persons is chosen from the Menu (see Figure 19). This component is responsible for showing the list of persons stored in the city-db database.

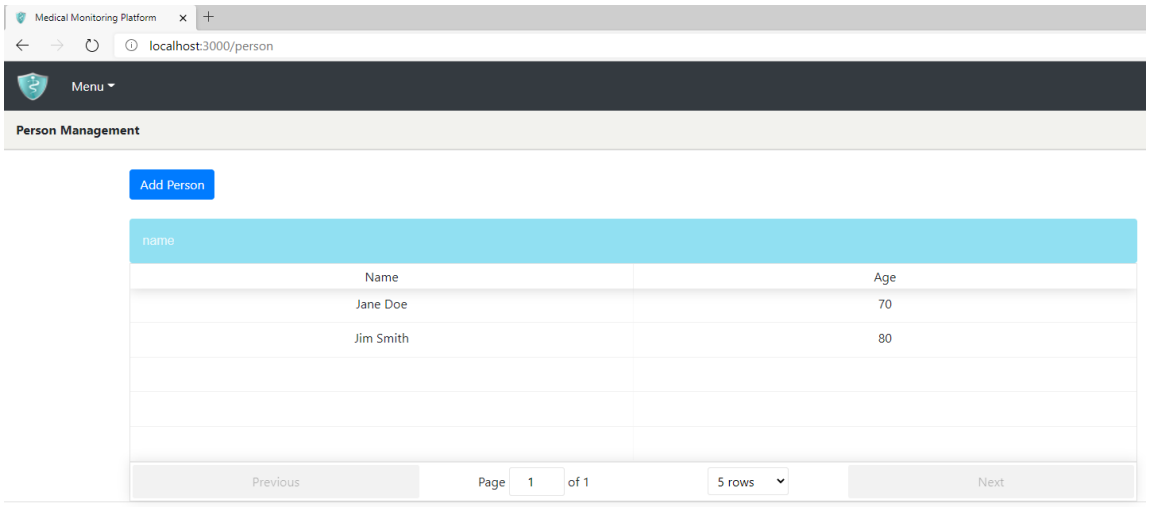


Figure 22: The Person Management page displayed when the PersonContainer component is rendered

Once an instance of the PersonContainer component is created, React calls the component's constructor (see Figure 23). Generally, the role of a constructor is to (1) initialize the local state of a component, by assigning an object to *this.state* and to (2) bind event handler methods to an instance [3]. For more information regarding techniques for handling events check [4].

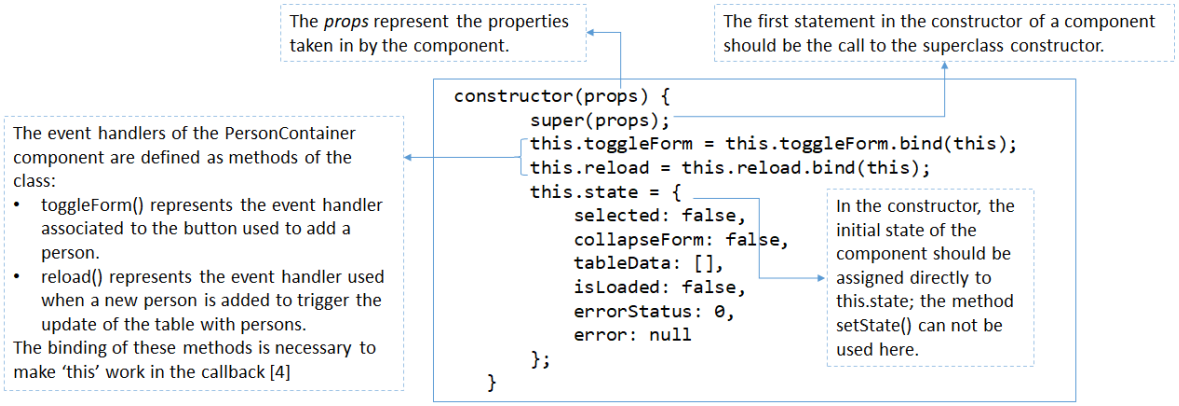


Figure 23: implementation of the constructor in the PersonContainer component

React then calls the render() method (see Figure 24) which returns the elements (i.e. , <CardHeader>, <Card>, <Row>, <Col>, <Table> etc.) that will instruct React to render the corresponding DOM nodes and user-defined components. React then updates the DOM to match the component's render output.

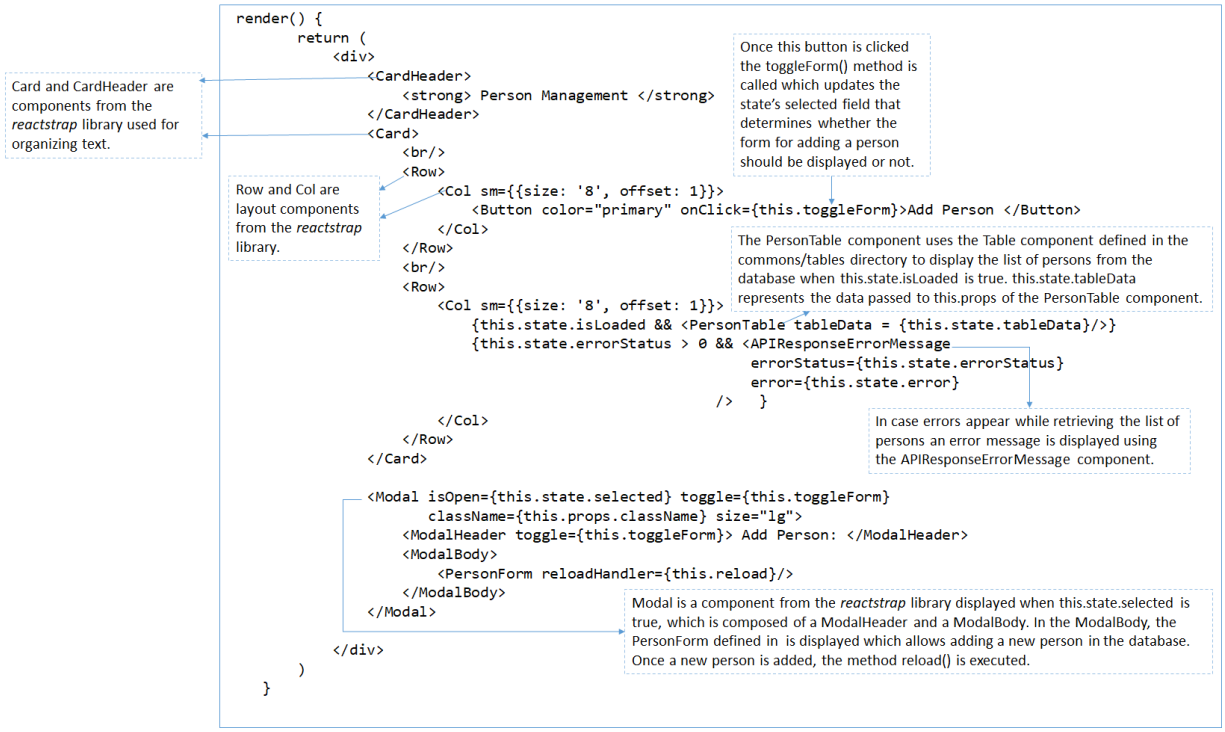


Figure 24: Implementation of the render() method in the PersonContainer component

GOOD TO KNOW [3]

What is the role of the *render()* method?

The render() method is the only method required to be implemented in a React component. The aim of this method is to analyze *this.props* and *this.state* and return one of the following: a React element that will eventually be rendered to a DOM, a user-defined component, arrays, strings, numbers, Booleans, null, etc.

Note: the render() method cannot modify the state of the component.

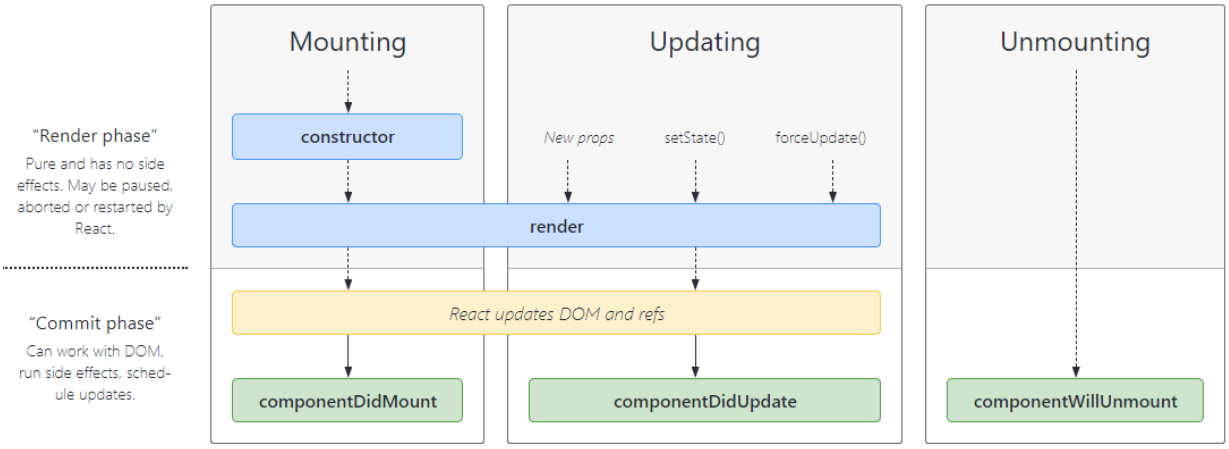
Component mounting versus unmounting

Component mounting occurs when a React component is rendered to the DOM for the first time**.**

Component unmounting occurs when the DOM produced by a React component is removed.

Other lifecycle methods

A React component has several lifecycle methods which can be overridden to implement specific functionalities desired to be executed at particular stages. The figure below shows the lifecycle diagram [3] corresponding to a React component.



- *componentDidMount*() – is invoked immediately after a React component is mounted; in this method you should add any initialization code such as performing remote requests to the back-end application in order to get some data required for initialization.
  - The method *setState*() can be called after obtaining the data to trigger an extra rendering before the browser updates the screen.
- *componentDidUpdate*() – is invoked immediately after updating occurs. Network requests can be performed here only if the current *props* are different than the previous *props*.
  - The method *setState*() can be invoked but it must be wrapped in a condition so as to avoid an infinite loop.
- *componentWillUnmount*() – invoked immediately after a component is unmounted and destroyed. This is the place to perform any necessary cleanup such as canceling network requests. The method *setState()* cannot be invoked here.

The *setState()* method – enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state. This is the primary method you use to update the user interface in response to event handlers and server responses. It does not always immediately update the component.

When the PersonContainer component's output is inserted in the DOM, React will call the *componentDidMount*() lifecycle method which in turn invokes the method *fetchPersons*() to get the list of persons to be displayed – see Figure 25.

The fetchPersons() method is responsible for sending an HTTP GET request to the spring-demo back-end application in order to obtain the list of persons stored in the city-db database.

In case of success, the data returned by the back-end will be set as the content for the table displaying the list of persons.

In case an error occurs, the received error will be assigned to this.state.errorStatus and will be further displayed by the APIResponseErrorMessage component.

```
componentDidMount() {
        this.fetchPersons();
    }

fetchPersons() {
        return API_USERS.getPersons((result, status, err) => {

            if (result !== null && status === 200) {
                this.setState({
                    tableData: result,
                    isLoaded: true
                });
            } else {
                this.setState({
                    errorStatus: status,
                    error: err
                });
            }
        });
    }
```

Figure 25: Implementation of the *componentDidMount*() and *fetchPersons*() methods in the *PersonContainer* component

Note at this step how a **callback** is passed as an argument to the *getPersons* method. The callback is represented by an arrow function, which takes as arguments the *result* returned in the HTTP response message, the *status* code of the HTTP response message and the *err* representing the error returned if the request has not succeeded. When the callback is executed, the following steps are performed:

- If the result is not null and the HTTP status code is 200 (meaning that the request has succeeded) then the state of the component *Person Container* is updated by calling the *setState*() method. In particular, the *tableData* is assigned to the result returned by the back-end application, and *isLoaded* is set to true meaning that the table displaying the list of persons can be rendered.
    - Note that for rendering the table with the list of persons, **conditional rendering** is used (see Figure 24): only if the value of *this.state.isLoaded* is true then the table is rendered!
- Otherwise, the state of the component *Person Container* is updated by calling the *setState()* method. In particular, the *errorStatus* is assigned the value of the HTTP status code, while *error* is assigned the error details.
    - Note that for rendering the *APIResponseErrorMessage* component, **conditional rendering** is used (see Figure 24): only if the *errorStatus* is greater than 0 then the *APIResponseErrorMessage* is rendered. In case the *APIResponseErrorMessage* is rendered, the *errorStatus* and *error* will pe passed as props to the component.

Step 3: In order to get the list of persons stored in the city-db database, the Persons component uses the *getPersons()* JavaScript function implemented in person-api.js to perform an HTTP GET request to the spring-demo back-end (see Figure 26).

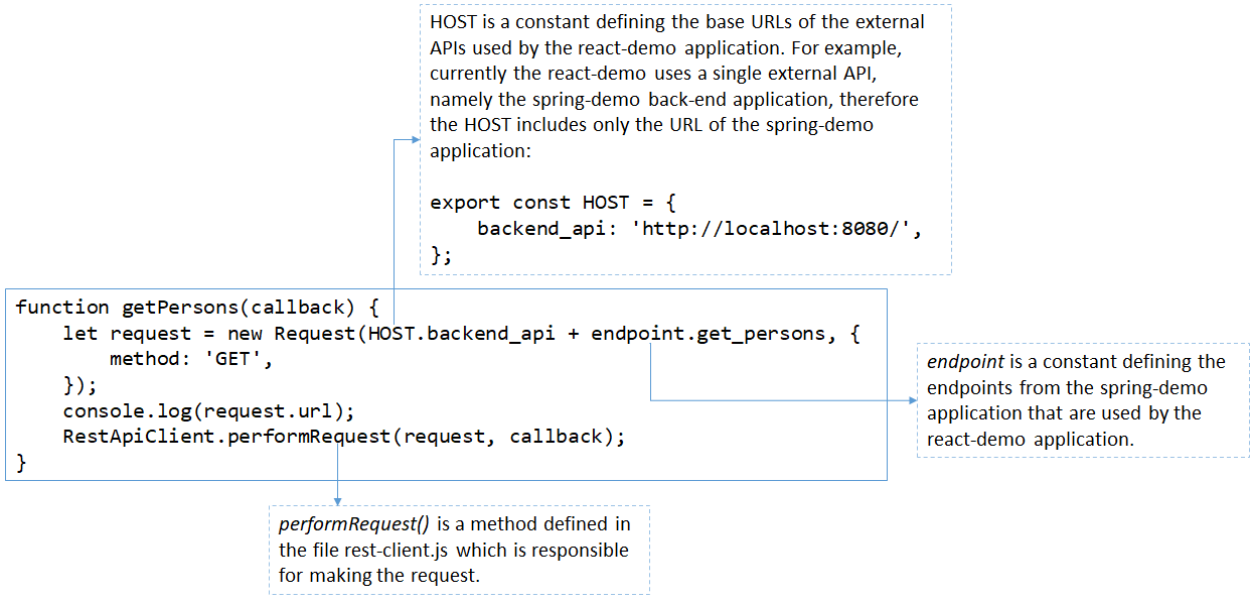


Figure 26: Implementation of the *getPersons()* method in person-api.js

Step 4: In order to send HTTP requests, react-demo uses the *fetch API* (see Figure 27) in the file rest-client.js. Other APIs can be used for this such as XMLHttpRequest [9] and axios [10].

```
function performRequest(request, callback){
    fetch(request)
        .then(
            function(response) {
                if (response.ok) {
                    response.json().then(json => callback(json, response.status,null));
                }
                else {
                    response.json().then(err => callback(null, response.status,  err));
                }
            })
        .catch(function (err) {
            //catch any other unexpected error, and set custom code for error = 1
            callback(null, 1, err)
        });
}
```

Figure 27: Implementation of the function performRequest in the rest-client.js

Step 5: an HTTP GET request for retrieving the list of persons is sent to the spring-demo application.

Step 6: the spring-demo application returns a list of PersonDTO objects converted to a JSON object together with the status of the response (see Figure 28).

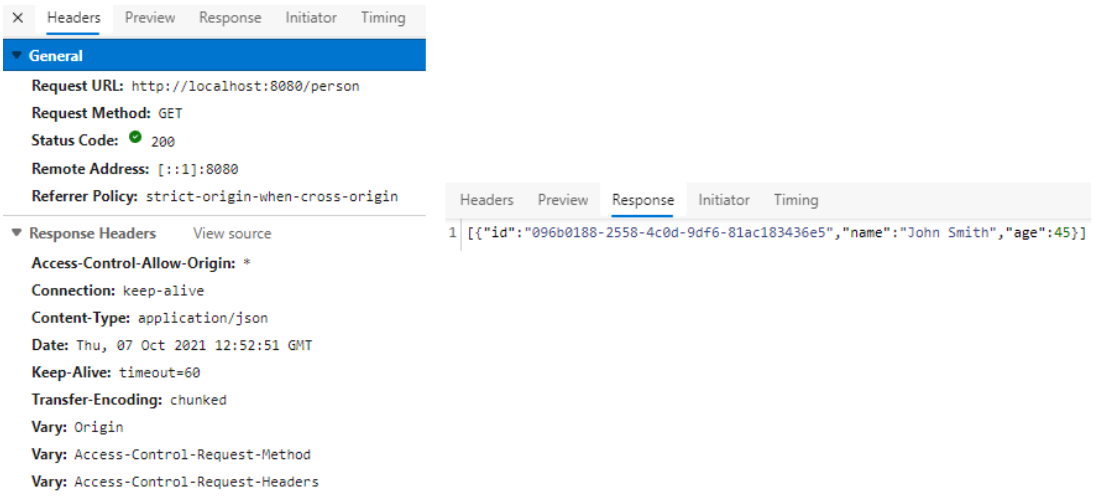


Figure 28: Response returned to the react-demo application in case the request succedeed (screenshot from the Browser Developer Tools)

Steps 7-8: In case the request succeeded, the callback is executed, in this case the arrow function presented in Steps 1-2. The arguments passed to the arrow function are the JSON structure containing the list of persons, and the response status code which is 200 in this case. If the request failed, then callback is executed with the following arguments passed to it: the response status code and the received error.

7. Reinforcement Learning

Answer the following questions:

- What is a React Component?
- What is a route?
- How is data serialized and de-serialized?
- Which component is rendered first in a React App?

8. References

This section presents a selective bibliography recommended to the students and used by the instructors in the preparation of the Hands On React document.

[1] https://www.youtube.com/watch?v=Ke90Tje7VS0
[2] https://ibaslogic.com/react-tutorial-for-beginners/
[3] https://reactjs.org/docs/react-component.html
[4] https://reactjs.org/docs/handling-events.html
[5] https://reactjs.org/tutorial/tutorial.html
[6] https://www.w3schools.com/js/js_htmldom.asp

[7] https://reactjs.org/docs/faq-internals.html
[8] https://reactjs.org/docs/introducing-jsx.html
[9] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[10] https://github.com/axios/axios
[11] https://javascript.info/
[12] https://www.w3schools.com/js/
[13] https://developer.mozilla.org/en-US/docs/Web/HTML/Element
[14] https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors