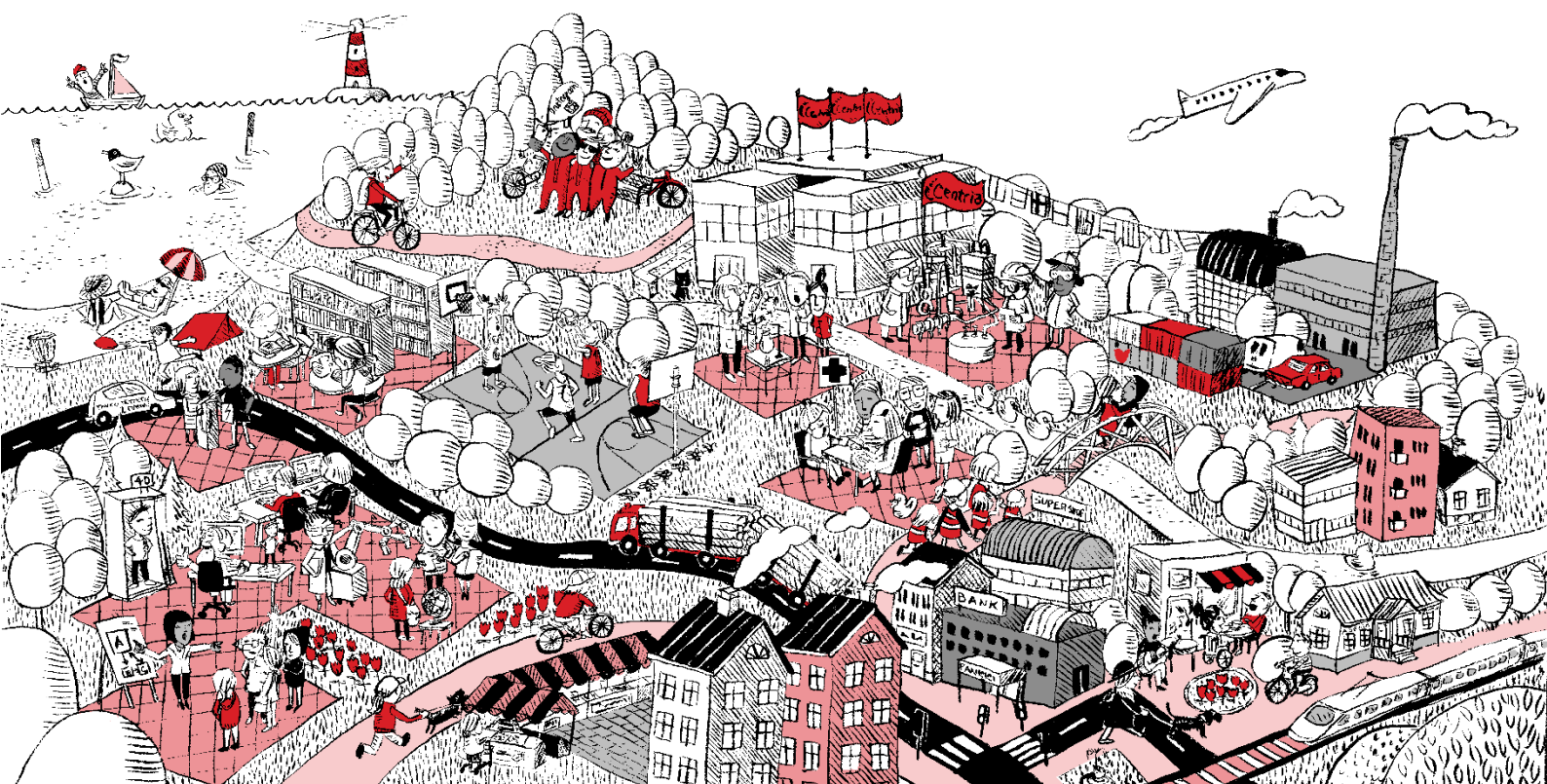


Kim Nguyen

BUILDING A TOURISM APPLICATION WITH REACT AND NODE.JS

Thesis
CENTRIA UNIVERSITY OF APPLIED SCIENCES
Information Technology
October 2021



ABSTRACT

Centria University of Applied Sciences	Date October 2021	Author Kim Nguyen
Degree programme Bachelor of Engineering, Information Technology		
Name of thesis BUILDING A TOURISM WEBSITE WITH REACT AND NODE.JS		
Centria supervisor Kauko Kolehmainen	Pages 60	
<p>The primary purpose of this thesis was to study and experiment on a method of building a single-page application website with the help of React, Node.js, Express.js, and MongoDB. The thesis consists of four major contents. The first content went through the tools and requirements to work with React and Node.js in a local machine. The second content presented the React library through generic experiments on the React fundamental building blocks, and the integrated Babel compiler. During the second content, the thesis explained the syntax structure, behaviors, anatomy, and core mechanism of React library, which made it a popular JavaScript library for front-end development. The third content was the study of the back-end RESTful APIs with Node.js, Express.js. The third content thoroughly described JavaScript's implementation in back-end development with the examples of building a complete RESTful API with Node.js and Express.js according to the MVC design pattern. The fourth content was about the process of step-by-step from planning and sketching to creating and testing a dynamic website. The thesis also stated the importance of authentication and security of web development through the implementation of hashing algorithms and access tokens.</p> <p>The result of the thesis work was a tourism website, which was a final product of the study and experience of MERN stack web development. The result showed specifically the efficiency, reliability, security, and reusability of the core features and principles of the previously mentioned technologies. The structure and functionality of the website were explained and demonstrated in detail.</p>		

Key words

React, ReactJS, HTML, CSS, JavaScript, Virtual DOM, Components, JSX, Node.js, Express.js, REST, RESTful, MongoDB, NoSQL

CONCEPT DEFINITIONS

List of Abbreviations

SGA	Single-page Application
ODM	Object Data Model
MERN	MongoDB, Express.js, React, and Node.js
I/O	Input/Output
RAM	Radom Access Memory
XHP	PHP extension
NPM	Node Package Manager
API	Application Programming Interface
MVC	Model-View-Controller
UCC	User-created content
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
CSS	Cascading Style Sheet
DOM	Document Object Model
CDN	Content Delivery Network
CORS	Cross-origin Resource Sharing
JSX	JavaScript XML
XML	Extensible Markup Language
UI-UX	User Interface and User Experience
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
JWT	JSON Web Tokens

ABSTRACT
CONCEPT DEFINITIONS
CONTENTS

1 INTRODUCTION.....	1
2 REQUIREMENTS AND TOOLS FOR DEVELOPMENT	2
2.1 The introduction to Node.js runtime	2
2.2 The Node Package Manager	3
2.3 The project manifest	3
3 REACT LIBRARY	5
3.1 The history of React library	5
3.2 React library's anatomy	6
3.3 Fetching React library from React CDN	7
3.4 Create a raw React Application.....	7
3.4.1 Optimizing performance with React Virtual DOM.....	10
3.4.2 Creating a React Component.....	12
3.4.3 Accessing data with React Props	14
3.4.4 Manipulating website behaviors with React State.....	14
3.4.5 Adding styling with React	16
3.5 The introduction to JSX	17
3.5.1 The Babel transpiler	18
3.5.2 Compiling JSX with Babel	21
3.6 Component Lifecycle	24
4 NODE.JS AND EXPRESSJS	27
4.1 Introduction to API and RESTful API	27
4.2 Node.js in the server-side.....	28
4.2.1 Starting a Node.js server from scratch	28
4.2.2 URIs structure and routing.....	29
4.2.3 Data transmission with streams and buffer	32
4.2.4 Node.js and asynchronous programming	34
4.3 ExpressJS, a node.js framework.....	36
4.3.1 Starting a Node.js server with ExpressJS.....	37
4.3.2 Routing in ExpressJS.....	37
4.3.3 The MVC Pattern	39
5 FINAL PROJECT	41
5.1 The basic ideas and plans	42
5.2 Sketching the UI	43
5.3 The front-end with React library.....	43
5.3.1 The Home page.....	44
5.3.2 The Authentication page	46
5.4 The back-end with ExpressJS and Mongoose	48
5.4.1 Planning and building a REST API with ExpressJS.....	49
5.4.2 Establishing connection with MongoDB Atlas and Mongoose	51
5.4.3 Adding password hashing function and web token	53
5.5 Adding Google Map API and Google Geocoding API.....	55

5.6 Testing the API with Postman	56
5.7 Final Result	58

6 CONCLUSION	60
---------------------------	-----------

REFERENCES

FIGURES

FIGURE 1. A version of Node.js on a local machine	3
FIGURE 2. A package.json file inside a Node.js project	4
FIGURE 3. The CDN links of React and ReactDOM (Facebook 2021).	7
FIGURE 4. An example raw React Application	8
FIGURE 5. The example raw React Application runs on a browser	9
FIGURE 6. The HTML DOM Tree of Objects (W3Schools 2021).	10
FIGURE 7. Five areas of the pixel pipeline (Lewis 2020).	10
FIGURE 8. The z-index concept (Soueidan 2015).	11
FIGURE 9. The calculations in the Virtual DOM reduce rendering (Eisenman 2021).	12
FIGURE 10. An example Bookmark class Component	13
FIGURE 11. Using state to changes the title of a component	15
FIGURE 12. Before and after the buttons change the title properties	16
FIGURE 13. Adding CSS styling in the definition of the Bookmark component.....	17
FIGURE 14. A JSX element	18
FIGURE 15. An example ES 6 arrow function inside a Node project	19
FIGURE 16. An example new file is created by Babel	19
FIGURE 17. The env preset is defined and configured in a .babelrc file	20
FIGURE 18. The result after transforming the arrow function.....	20
FIGURE 19. An example button with props written in JSX	21
FIGURE 20. A result after Babel compiling a JSX code.....	22
FIGURE 21. An example HTML page contains a mainContainer	22
FIGURE 22. An example JSX Bookmark class component.....	23
FIGURE 23. The result after transforming the JSX component.....	24
FIGURE 24. The lifecycle event of a React component (Zammetti, 2020).	25
FIGURE 25. Creating a Node.js server.....	28
FIGURE 26. The response from the Node.js server in port 3000.....	29
FIGURE 27. The syntax of a URI or HTTP URL (IBM 2020).	30
FIGURE 28. An example of hierarchical association between path segments (Allamaraju 2010).	30
FIGURE 29. Routing with url properties in Node.js	31
FIGURE 30. The browser redirects after the POST request is successfully sent	31
FIGURE 31. Manipulating data stream with Buffer class	32
FIGURE 32. The console logs the hexadecimal chunks in buffer	33
FIGURE 33. The text is converted from the data chunks in buffer	33
FIGURE 34. An example setTimeout() function.....	34
FIGURE 35. An example of a promise in JavaScript	35

FIGURE 36. An example of an async function in JavaScript	36
FIGURE 37. Starting a Node.js server with ExpressJS	37
FIGURE 38. Basic routing with ExpressJS	38
FIGURE 39. Import a URL encoder with express after version 4.16.0.....	38
FIGURE 40. ExpressJS parses the request body's data	39
FIGURE 41. A graphic example of the MVC pattern (Bakshi 2019).....	40
FIGURE 42. The JavaScript frameworks' popularity on Stack Overflow Trends (Robinson 2017).....	41
FIGURE 43. The dependencies chart of the website	42
FIGURE 44. The wireframe design of the project.....	43
FIGURE 45. The pages and routes setup of the application.....	44
FIGURE 46. Home page without a back-end connection on browser.....	45
FIGURE 47. The Users component of the application	46
FIGURE 48. The login form inputs are invalid	47
FIGURE 49. A part of the validator hook.....	47
FIGURE 50. The function handles sending login and signup requests to the server	48
FIGURE 51. The API's file structure according to the MVC pattern.....	50
FIGURE 52. The user-route.js and places-routes files	51
FIGURE 53. A MongoDB collection is stored in two different shards (MongoDB 2021).....	52
FIGURE 54. Establishing a database connection with Mongoose	52
FIGURE 55. The database schemas.....	53
FIGURE 56. Hashing and salting user passwords with bcrypt.....	54
FIGURE 57. Creating an access token with JWT.....	54
FIGURE 58. Adding and rendering Google Map in React.....	55
FIGURE 59. Implementing Google Geocoding API with ExpressJS	56
FIGURE 60. Sending a POST request to signup an account	57
FIGURE 61. A user account is stored in MongoDB Atlas Database	58
FIGURE 62. The home page and newsfeed.....	59
FIGURE 63. The form and the map.....	59

TABLES

TABLE 1. The URL paths of the Travelogue project.....	43
TABLE 2. The endpoints of the API	49

1 INTRODUCTION

The digital age has significantly broadened the availability to access documents and information since the rapid growth of internet connectivity and the invention of worldwide webs. Over the years, internet bandwidth and connectivity improvements have uncountably increased the amount of data stored and processed by worldwide webs. The rapid improvement of hardware technologies made mobile devices, computers, and laptops more affordable and accessible every year. Since data, information, and connectivity are daily increasing, websites are always challenged with enhancing performance, security, maintainability, and scalability. Thus, many technologies, tools, new web application architectures have been created to support these purposes specifically. The single-page application or SPA was born as the solution for large-scale with complex and heavy database type of websites. SPA is an emerging concept of web application architecture, which has been widely adopted after the successes of various giant companies, such as Facebook, Netflix, and Google.

The tools combination of MongoDB or Mongoose, Express.js with Node.js, and React, are commonly known as the MERN stack web development. They are widely used to develop single-page web applications. (Carnes 2021.) React library has built-in features for front-end development, which reduce the complexity of pure JavaScript syntax and enhance user interface development, giving websites a more modernistic style and user experience. In addition, React also gives developers numerous choices of utilizing third-party frameworks, for instance, Gatsby and Next.js, to boost up the developing process. Express.js is a widely-used Node.js framework, giving the Node.js project a lean and more maintainable structure but still keeps the logic behind the scene safe and guaranteed. Mongoose is an Object Data Modeling framework, which is built to work inside Node.js with various core features that are implemented to manipulate MongoDB, a non-relational database within a Node.js project. Even though there are other alternative options, this thesis chose the MERN stack as a development tool since its growing popularity and support from the community. The primary purpose of this thesis was to study, experiment, and build a project with the MERN stack. The project was a user-created content or UCC website that allows users to create accounts, store, view their images, and interact with the contents of other users on the website.

2 REQUIREMENTS AND TOOLS FOR DEVELOPMENT

Programming has brought many concepts based on real-life to make them easy to comprehend. The most popular concept is the environment concept—every application lives inside an environment built by many tools and technologies. Developing a single-page application also requires a development environment to maintain the application's life cycle. The first step of developing a website is to create a development environment, and with a MERN stack project, Node.js is a fundamental tool to support this purpose. (Zammetti 2020, 8 -9.)

2.1 The introduction to Node.js runtime

Since JavaScript is a web-based programming language, it was initially designed to only run on a web browser. However, when websites became a crucial part of the internet, the popularity of JavaScript started increasing. Many tools, frameworks, and environments that were built to interpret and execute JavaScript codes were born. The increasing number of web-based technologies have greatly expanded the territory of JavaScript to other platforms and actively changed its initial purpose. This was when Node.js comes to play an important role. Ryan Dahl created Node.js at Joyent in 2009 as an alternative solution for Master Control Program API, a scripting tool to manipulate website interaction (Preul 2017). Two of the essential dependencies of built-in Node.js are the V8 engine and the libuv library. The V8 engine, written in C++, is a core JavaScript engine implemented in Google Chrome or any other Chromium web browsers to parse and execute JavaScript programs. The libuv library is the vital part that makes the famous asynchronous behavior of Node.js or non-blocking I/O operations (Salim 2020). To install Node.js runtime, developers need to access <https://nodejs.org/>, and there are always two options for downloading Node.js to a local machine, the recommended version and the latest version. After choosing the suitable version, developers can click on the download button that displays the version to download a Node.js installer, wait until the download finishes and activate the installer file. The Setup Wizard then walks developers through the setting, step by step. Figure 1 demonstrates a local machine successfully installed Node.js, developers can check the version by typing `node -v` or `node --version` on the command prompt. With Node.js installed inside, any local machine now can be a JavaScript development environment, or in other words, they are capable of executing JavaScript programs. (Yushkevych 2021.)

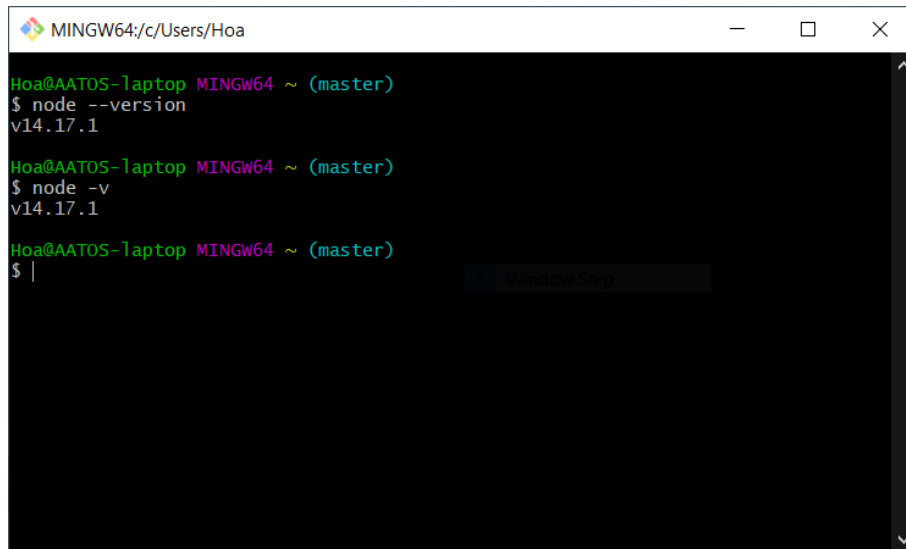
A screenshot of a terminal window titled 'MINGW64:/c/Users/Hoa'. The prompt is 'Hoa@AATOS-laptop MINGW64 ~ (master)'. The user enters '\$ node --version' and the output is 'v14.17.1'. The user then enters '\$ node -v' and the output is 'v14.17.1'. The prompt is now '\$ |'.

FIGURE 1. A version of Node.js on a local machine

2.2 The Node Package Manager

Node Package Manager or NPM is a free-to-use application that installs alongside Node.js, even though it was developed separately and may have a different update schedule. The general idea of NPM was a central package registry containing reusable and downloadable JavaScript modules or node packages. Each package has specific functionalities, for instance, authentication or authorization, that can be implemented to any Node.js project (Zammetti 2020, 8 -11.). NPM operates with two main parts, a command-line interface or CLI tool for publishing and downloading node packages and an online central repository or registry. The central repository is available for visiting at <https://www.npmjs.com/>. The NPM command-line tool supports various basic features that manipulate any node package inside a JavaScript project. For example, the command `npm install` allows the Node.js program to download a node package or `npm init` to create a `package.json` file in a Node.js project. (MDN 2021.)

2.3 The project manifest

Most of the NPM/Node projects contain a `package.json` file in the root directory as the project manifest file. The `package.json` file provides metadata information of the project to NPM that it needs to perform certain tasks, such as downloading available modules from its central package repository, which are required inside the project. It also contains the name, version, description, author, and information of integrated tools inside the project. (Zammetti 2020, 12 -13.)

```

1  {
2    "name": "back-end",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1",
8      "start": "node app.js"
9    },
10   "author": "Kim Nguyen",
11   "license": "ISC",
12   "dependencies": {
13     "axios": "^0.21.1",
14     "bcryptjs": "^2.4.3",
15     "dotenv": "^10.0.0",
16     "express": "^4.17.1",
17     "express-validator": "^6.12.0",
18     "jsonwebtoken": "^8.5.1",
19     "mongoose": "^5.12.14",
20     "mongoose-unique-validator": "^2.0.3",
21     "multer": "^1.4.2",
22     "uuid": "^8.3.2"
23   },
24   "devDependencies": {
25     "nodemon": "^2.0.7"
26   }
27 }
28

```

FIGURE 2. A package.json file inside a Node.js project

Figure 2 illustrates the dependencies and devDependencies parts inside a package.json file. The term dependency in any JavaScript project usually means a third-party software that solves a single problem. These third-party software pieces are preferred as node modules or node packages when NPM stores them on its central repository. Whenever NPM installs a module or package, it declares the module's name in the dependencies part of the package.json file. Inside every Node.js project contains a node_modules folder, the largest folder in the project. All the declared modules in the package.json file live in this folder, and it is detachable to reduce the project's capacity, which is essential for storing and transporting files. Later, these modules are able to reattach to the project with the npm install command by NPM. NPM looks into the package.json file and downloads all the declared modules from the central package repository. (Zammetti 2020, 13 -14.)

3 REACT LIBRARY

Over a decade, many vibrant JavaScript frameworks/libraries/toolkits have been made for building web-based client applications, and there were a few popular options that had reached the top, and React was one of them. The following sections explain the popularity, structures, and core features of React, which made it a dominant front-end library for Single-page Applications. (Zammetti 2020, 44 -45.)

3.1 The history of React library

Back in 2010, when Facebook was still using PHP, a popular general-purpose scripting language, as their primary tool, it later brought them into many issues with code maintenance. Consequently, these issues began to affect their development velocity and delivered quality significantly. Due to this problem, the engineers introduced XHP into their PHP stack to improve the situation, improve syntax, and make PHP code more readable. The initial concept of XHP was to provide the notion of composite components, which allows developers to break down an interface into mostly independent but easily integrated units of functionality. Then, in 2011, FaxJS was created by Jordan Walke from a part of XHP as a prototype, which contains several critical characteristics of React, including updating views with states and interface components. (Zammetti 2020, 44 -46.)

In 2012, Facebook again ran into more security problems with their advertisements management on the site. Since other servers usually serve the advertisements, Facebook does not control what they are, which creates a security loophole for the anonymous to break into their website. XHP was expected to be the primary tool to solve this problem. However, XHP principally focuses on minimizing Cross SiteScripting or XSS attacks, which inject malicious browser-side scripts into benign and trusted websites since some browsers at that time could not detect whether the scripts are trustworthy or not. The malicious script can access any cookies, session tokens to gain access into end-user accounts of any website or any sensitive information retained by the browser or even rewrite the content of the HTML page. But XHP still has a critical disadvantage, a dynamic web application requires clients to send many roundtrips to the server, and XHP does not help reduce them. The engineers at Facebook began to search for the solution, and FaxJs answered, which later would become React library. (Dawson 2014.)

In May of 2013, Pete Hunt and Jordan Walke published React as an open-source project, but the library did not impact the market until 2014 when several factors favored React. For instance, Google released React Developer Tools as an extension of Chrome Developer Tools, giving developers a robust set of

equipment to develop and debug React applications. Many integrated development environments or IDE began to introduce native support for React. The peak started in 2015 when Flipboard, Netflix, and Airbnb all began using React to help their workload of the client-side. Since 2017, React has grown increasingly and has become the most popular library for building client-side applications. (Zammetti 2020, 44 -46.)

3.2 React library's anatomy

As mentioned above, after version 0.14, React library has been separated into two packages, React and ReactDOM, because they support different purposes and have other mechanisms. Since version 0.14, Facebook has developed more packages explicitly for React libraries to expand their functionalities and create more convenient tools for React developers. There is a folder named packages, which lies inside of React library and can be found in React's Github repository. The folder stores all features of the library, from testing tools, debug tools, and core features to manipulate the user interface. However, they all still have to work around two and core packages: React and ReactDOM. React is responsible for creating views, and ReactDOM is used for optimizing the render of the UI in the browser under the hood. The list of dependencies of React library is available in the package.json file on their Github repository, which shows the technologies being used to create its behaviors and features. The list contains several key names start with @babel. They are plugins from Babel, a JavaScript compiler. These plugins are built specifically for React library, which plays a vital role in translating the syntax of React library so any browser can function correctly. This thesis also covers the information of Babel in section 3.5 JSX introduction. (Bank & Porcello 2017, 84 -87.)

3.3 Fetching React library from React CDN

To see through the mechanism of React and ReactDOM, the React library also provides their scripts from Facebook CDN or Content Delivery Network or sometimes is also preferred as Content Distribution Network. A CDN is a group of servers allocate around the globe and store duplicate data to facilitate the distribution of information generated by Web publishers quickly and efficiently. (Taylor & Francis 2010, 32 -33.)

```
<script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
```

FIGURE 3. The CDN links of React and ReactDOM (Facebook 2021).

Figure 3 shows the crossorigin attributes, which are recommended to set up with the CDN Links of React because of security purposes. Cross-Origin Resource Sharing or CORS is an HTTP-header base mechanism used to inform that the HTTP/HTTPS request comes from the same domain, scheme, or port of the current requested server. For instance, the front-end code from <https://domain-a.com> demands access to a resource of <https://domain-b.com/data.json>. For security purposes, the browser restricts this request because they are technically from two different domains. The request has to follow the CORS policy to access this resource, which means the script must inform the server that it is a cross-origin request by adding the crossorigin attribute. (Kosaka 2018.)

3.4 Create a raw React Application

A raw React application can be built without installing any module or JavaScript runtime. All it needs is a text editor and a browser. For example, Figure 4 is a React application with raw React code injected directly into HTML code. In this example, the React and ReactDOM package were distributed from the Facebook CDN, and when this application started, it downloaded the react.development.js and react-dom.development.js code (Zammetti 2020, 48 -50.). Here in Figure 5 is the result when the pure React code ran on a browser.

```

index.html > html > head > script
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <title>Introduction to React</title>
6      <script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>
7      <script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
8      <script>
9        //Pure React and JavaScript code
10       function start() {
11         const rootElement = React.createElement("div", { },
12           React.createElement("h1", { }, "Bookmarks"),
13           React.createElement("ul", { },
14             React.createElement("li", { },
15               React.createElement("h2", { }, "React"),
16               React.createElement("a",
17                 { href : "https://reactjs.org/" },
18                 "The home page of React library"
19               )
20             ),
21             React.createElement("li", { },
22               React.createElement("h2", { },
23                 "React's documentation"
24               ),
25               React.createElement("a",
26                 { href : "https://reactjs.org/docs/getting-started.html" },
27                 "The website contains documentation of React"
28               )
29             )
30           )
31       );
32       ReactDOM.render(rootElement,
33         document.getElementById("mainContainer")
34       );
35     }
36   </script>
37 </head>
38 <body onload="start();">
39   <div id="mainContainer"></div>
40 </body>
41 </html>

```

FIGURE 4. An example raw React Application

The mechanism behind the scene of the example React application can be explained concisely. Every HTML and JavaScript code is executed from top to bottom. In Figure 4, the browser firstly scanned the code from line 1 to line 7, and it received a command to download the raw React code from the CDN. From line 8 to line 36, it received a declaration of a function named `start()`. The function used other methods from the React and ReactDOM packages that were downloaded previously to create elements and rendered them inside an element with the id of `mainContainer`. The React keyword was treated as a class inside the example project, and this React class provided various methods to manage the logic of the DOM API. The term DOM API refers to a collection of objects that JavaScript can apply to interact with the browser. Some examples of manipulating DOM API are the `document.createElement()` and

document.appendChild() methods, which are often found in many pure JavaScript static websites. Nonetheless, among all the methods provided by React, the React.createElement() method is an essential topic that needs to be focused on since it relates to creating the most basic fundamental factor of React library. (Champion 1997.)

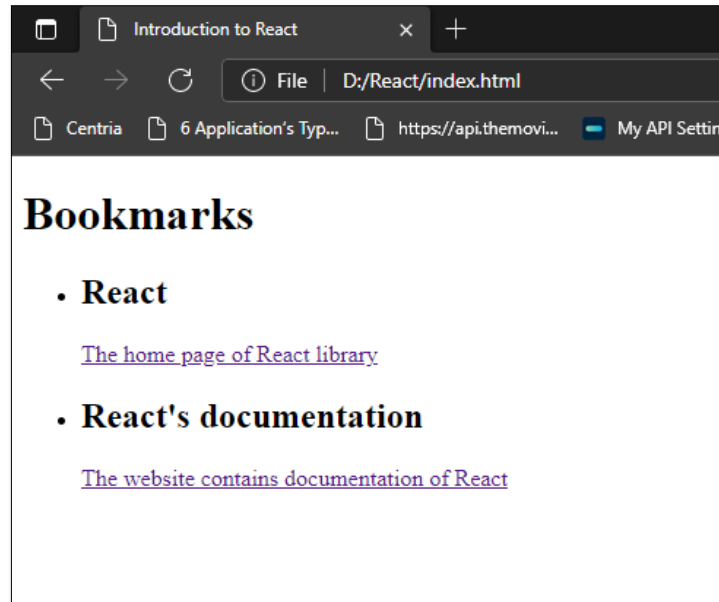


FIGURE 5. The example raw React Application runs on a browser

The createElement() method accepts three arguments, and the first argument is the type of element. This type technically means any HTML tag of choice, which is written in string type, for example, 'div' or 'span'. The second argument is props, which means the properties of a DOM element, or in other words, it also the attributes of an HTML tag. The props argument's default input is defined as an object containing all the predeclared attributes of an HTML tag. However, some of the HTML attributes are not required to be predefined and can be expressed by an empty object or null. For instance, on line 17 of Figure 4, there was an object containing a href attribute with a link <https://reactjs.org/>. The last argument is the children's argument, which is used to store the child text or nest many React elements into each other to create a more complicated element or a tree of elements, also called a DOM tree. The React.createElement() method constructs a new React element, which is the most crucial and smallest building block of the visual interface of a React application. However, the DOM tree was only constructed when the browser scanned to line 38 when it received an onload event and starts executing the start() function. After finishing constructing the DOM tree from many nested React elements in Figure 4, the render method in the ReactDOM package did its job, rendering the DOM tree to a visible website in Figure 5. (Chavan 2021.)

3.4.1 Optimizing performance with React Virtual DOM

The question came where React library separates ReactDOM into a different package when seemingly the only job of ReactDOM was to render a React element into a real DOM on a website. However, rendering a DOM element correctly without heavily affecting a website's performance was incredibly demanding. To solve this problem, React library introduced their Virtual DOM, which lives inside the ReactDOM package. The DOM tree technically represents the UI of a particular website. Whenever a change appears on the website, the DOM tree updates itself according to the change. The difficulty of updating the DOM tree comes when many changes occur, which significantly slows down a website. (Facebook 2021.)

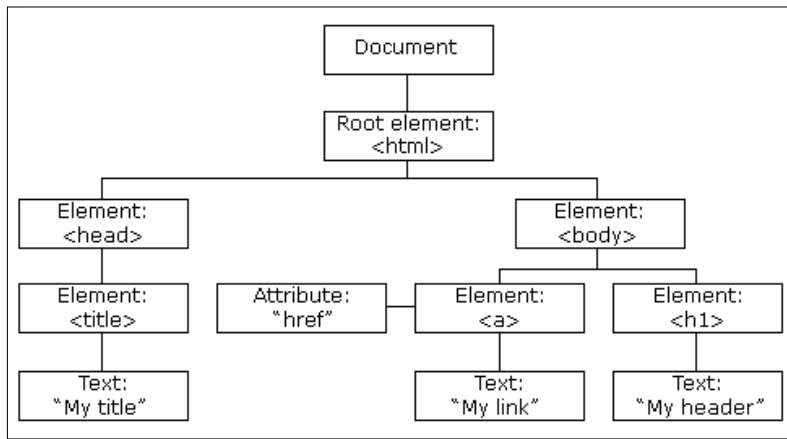


FIGURE 6. The HTML DOM Tree of Objects (W3Schools 2021).

The DOM tree displays in Figure 6 as a tree of data structure, making it easy to detect the changes inside the DOM tree by searching for each level of the tree. Nonetheless, the difficulty appears clearly when any change happens with any element on the upper levels of the DOM tree, which is nested by many complex children and descendent elements. Then the DOM tree has to update the entire complex structure of that branch to reflex the changes. (Ravichandran 2018.)

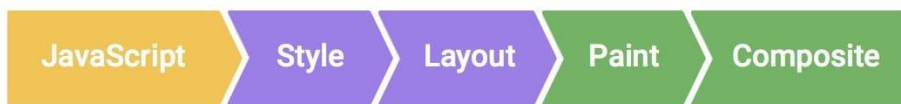


FIGURE 7. Five areas of the pixel pipeline (Lewis 2020).

Nowadays, every web browser applies the pixel pipeline to render a website, which usually is divided into five major areas. The first area in Figure 7 happens when a JavaScript code triggers a visual change,

and the browser detects and changes the DOM tree. The next area is style calculations, which find CSS style rules for each element based on the CSS selectors. When all the CSS rules of every element are found, they are applied to create the final rule. After every element has its final CSS rule, the layout area kicks in, where the browser calculates the volume and position of each element on the screen. Since the style of the descendent elements can affect the style of ancestor elements, the process sometimes needs to repeat to recalculate from the upper levels where these affected ancestor elements live. (Lewis 2020.)

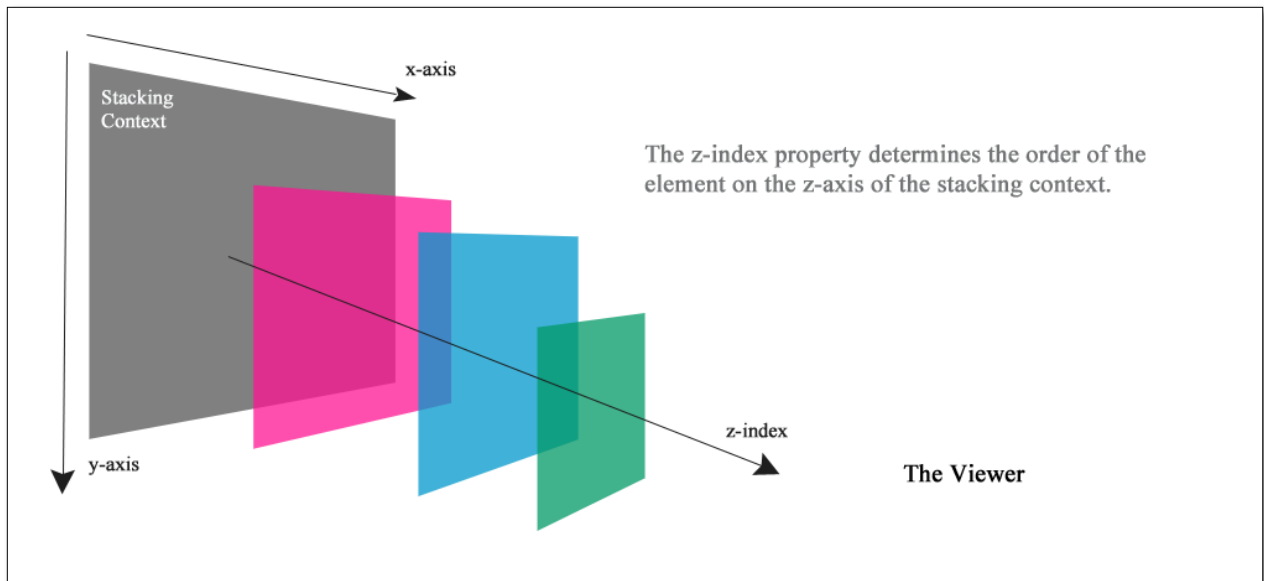


FIGURE 8. The z-index concept (Soueidan 2015).

The next area is the paint area, where the pixels start filling and drawing out text, colors, images, borders, and shadows, every visible part of the elements. Some elements sometimes stack on each other, creating many layers, as in Figure 8. Because of these layers, the final area of rendering a website is compositing. In this area, the browser keeps the layer stack order because a single mistake can result in one element being displayed on top of another. The pixel pipeline mechanism will slow down a website's performance if it has too many complex UI elements. A minor change triggers the update of an entire branch of the DOM tree, which forces the browser to handle more evaluating, detecting, calculating, painting, and compositing. In other words, the more extensive scale the website is, the more expensive resource every update consumes. Thus, React library presents an alternative update strategy to solve the problem, which is the virtual DOM. (Lewis 2020.)

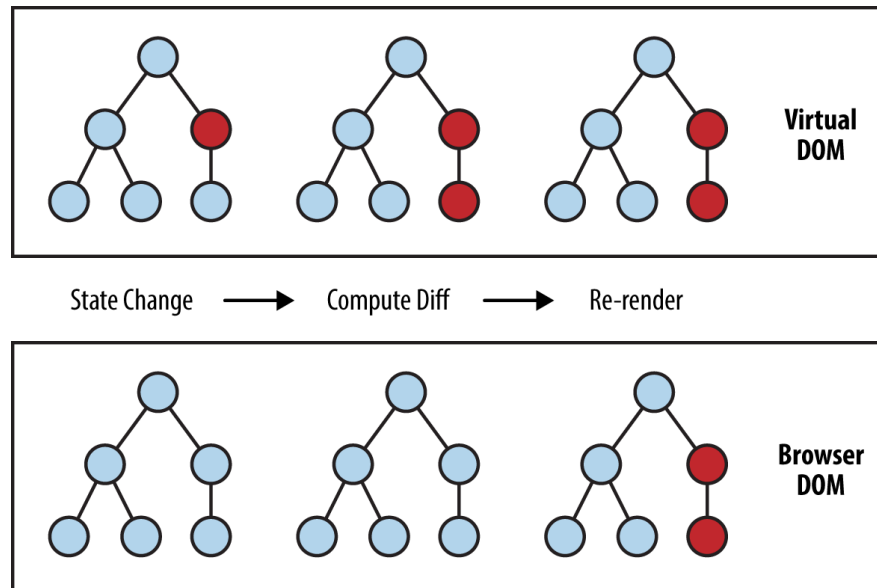


FIGURE 9. The calculations in the Virtual DOM reduce rendering (Eisenman 2021).

Virtual DOM is the virtual representation of the actual DOM tree. The difference between the `document.createElement()` and `React.createElement()` is that the `document.createElement()` method creates an element directly in the actual DOM, and the `React.createElement()`, on the other hand, creates an element in its own virtual DOM tree. The same mechanism is applied to update or delete any React element, which only changes the virtual DOM tree. After all the changes, ReactDOM has a special feature called diffing, where it compares the differences between the virtual DOM tree and the actual DOM tree, then it calculates the most efficient strategy to make these changes to the real DOM tree. Hence, ReactDOM reduces a remarkable cost for every update to the actual DOM tree. (Ravichandran 2018.).

3.4.2 Creating a React Component

Strong reusability is one of React library's primary concepts. To achieve reusability, React delivers a feature called component. A component is also a fundamental building block of React library, containing one or many React elements. In many websites, some segments have some identical functionalities, for instance, logos and icons. They may require different sizes or displays different shapes, but their core functionality is displaying a symbol. Based on these characteristics, they are categorized as the same type of component. So whenever a website needs an icon or a logo, they can reuse a component that displays a symbol and customize them slightly to fit the purposes. (Sathanathan 2021.)

```

index.html X
index.html > html > head > script > start
10 function start() {
11   //Bookmark Component
12   class Bookmark extends React.Component {
13     render() {
14       return (
15         React.createElement("li", { },
16           React.createElement("h2", { },
17             this.props.title),
18           React.createElement("a",
19             { href : this.props.href },
20             this.props.description
21           )
22         )
23       );
24     }
25   }
26
27   const rootElement = React.createElement("div", { },
28     React.createElement("h1", { }, "Bookmarks"),
29     React.createElement("ul", { },
30       React.createElement(Bookmark, {
31         title: "React library",
32         href: "https://reactjs.org/",
33         description: "The home page of React"
34       }),
35       React.createElement(Bookmark, {
36         title: "React's documentation",
37         href: "https://reactjs.org/docs/getting-started.html",
38         description: "The website contains documentation of React"
39       })
40     )
41   );
42   ReactDOM.render(rootElement,
43     document.getElementById("mainContainer")
44   );
45 }
46 </script>
47 </head>
48 <body onload="start();" >
49   <div id="mainContainer"></div>
50 </body>
51 </html>

```

FIGURE 10. An example Bookmark class Component

Figure 10 presents a class named `Bookmark`, which is one common way to create a React component; this type of component is called the class component. Every class component is created by the inheritance of the `Component` class inside React library. On line 12, the class `Bookmark` inherited all the properties and methods from the `React.Component` class by using the keyword `extends` in JavaScript. In 2019, React released version 16.8.0, which introduced a new way to create a React component called the functional components or React Hooks. Functional components are more straightforward to catch up with than class components since they do not follow the rules of Object-oriented Programming. Although they are created in different ways, they serve the same purpose. A class component or a functional component only needs to create once and use everywhere. (Surasani 2019.)

3.4.3 Accessing data with React Props

React components need data, and the props object is responsible for passing data into components. Inside the Bookmark class component or functional component, there is a props member that stands for properties. As mentioned before, every component contains many React elements; each element has its attributes, for instance, the href attribute for the <a> tag link. Every attribute receives data the same way a variable receives a value; for example, the href attribute accepts a URL as a value so the <a> tag can display and redirect a webpage to the given URL. The React.Component class from React provides an object named props, allows external data to be passed into a component as properties of the object. On lines 30 and 35 of Figure 10, when a new Bookmark component was created, new values of href attributes were passed into the component through the props.href, which was a property of the props object. Furthermore, data is commonly passed from a parent or the topmost component to children components, and the props object does not limit to delivering a specific type of data. Since React components are customizable, properties of the props object can also be user-defined attributes to serve different components' purposes. For example, the props.title, and props.description were user-defined attributes, which received user-defined data in Figure 10. Moreover, the props object plays a crucial role in creating React components' reusability since it can store and pass data from one component to another. The mechanism of the props object can be seen more clearly in functional components, where a props object is passed to a component in the same way as an argument is given to a function. In addition, there are two more important notes whenever the props object is used. Firstly, the props object is only passed into a component when the component is created. Secondly, the properties of the props object are immutable and read-only; the rendering mechanism of ReactDOM can explain this. Any time a change occurs inside a component, the virtual DOM starts detecting and diffing the change from the virtual DOM tree. Then the virtual DOM technically destroys the old component and replaces it with a new one from scratch, and a new props object is passed again into the component, giving a different result. For this reason, the props objects cannot be changed once they are passed to the children components, so React introduces state as a better way to update data inside a component. (Zammetti 2020.)

3.4.4 Manipulating website behaviors with React State

The previous section explained how props object works in a component as well as the effectiveness when it is mutated. To give a more effective way to handle mutable data, React library provides state as a solution. Every component has one or many default states. Any change that happens to these states

does not trigger ReactDOM to destroy the entire component and create a new one. Instead, it only informs ReactDOM changes a portion from the component that needs to change and updates that piece to the virtual DOM to minimize the real DOM change. In Figure 11, the Bookmark class had a constructor which printed a dialog when a Bookmark component was created. Inside the render method, a button changed the title property of the props when a click event occurred. The plan was to inspect whether the change triggered the ReactDOM to destroy the old component and recreate a new one. (Zammetti 2020, 57 -59.)

```

index.html x
index.html > html > head > script > start > Bookmark > render
3 <head>
4   <meta charset="utf-8">
5   <title>Introduction to React</title>
6   <script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>
7   <script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
8 </script>
9   //Pure React and JavaScript code
10  function start() {
11    //Bookmark Component
12    class Bookmark extends React.Component {
13      constructor(props) {
14        super(props);
15        console.log("Bookmark component created");
16        this.title = this.props.title;
17      }
18      render() {
19        return (
20          React.createElement("li", { },
21            React.createElement("h2", { }, this.title),
22            React.createElement("a",
23              { href: this.props.href }, this.props.description),
24            React.createElement("button", {
25              onClick: () => {
26                this.title = this.title + "-CHANGED";
27                this.setState({});
28              }
29            }, "Click me")
30          );
31        );
32      }
33    }
  }

```

FIGURE 11. Using state to changes the title of a component

Figure 12 demonstrates when the first time the application was executed, two Bookmark components were created. Hence, the console printed out the dialog "Bookmark component created" twice. Also in Figure 12, when the buttons were clicked and changed the title of two Bookmark components, the console did not print out any new dialog, proving that the constructor method did not create any new Bookmark component. The result of the setState() method on line 27 in Figure 11 informed React that the components needed to be adjusted partly instead of being recreated entirely. The setState() receives two

arguments, an updater and a callback function. An updater can be either a function or an object. However, if the updater is a function, it needs two arguments to work properly. The updater function returns a new state for the component. (Zammetti 2020, 57 -59.)

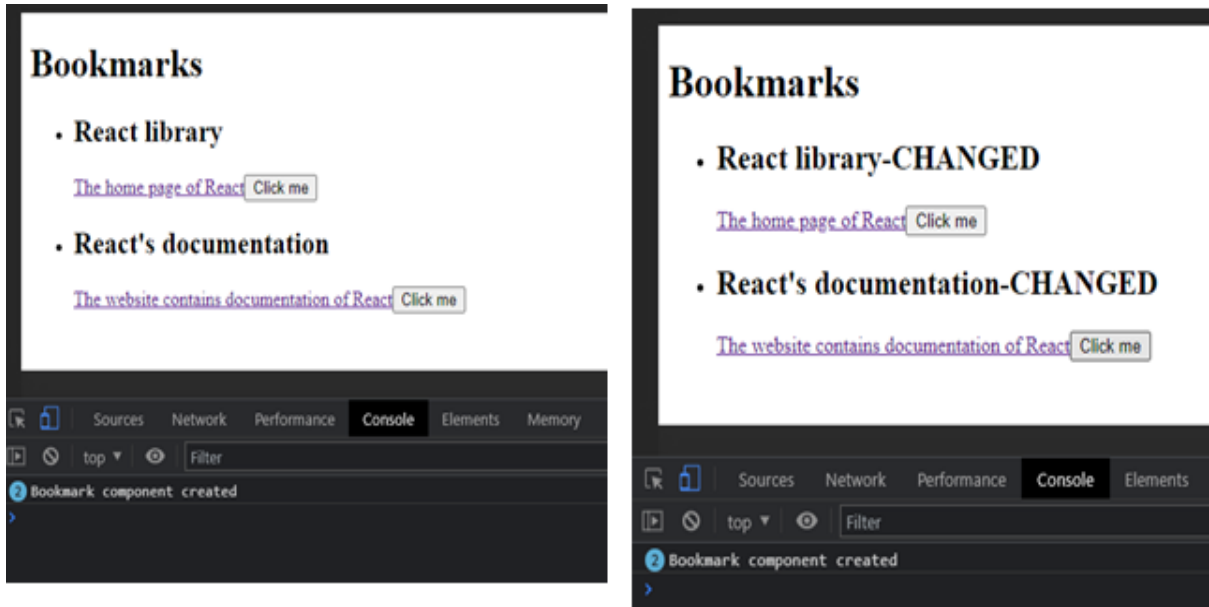


FIGURE 12. Before and after the buttons change the title properties

One important note is React does not recommend making direct changes by assigning new values to the `this.state` property, which is the default state of any component. To update a new state of a component, React always requires using the `setState()` method. The `setState()` method is asynchronous and does not immediately work when it is evoked. The `setState()` method has a procedure for updating new states since a component can have many states, and many components may need to update their data. React combines many update requests from the `setState()` methods into one batch and schedules them. The update is performed only when the scheduled time comes. Any direct change inside a component before the scheduled time is overwritten when React performs the updates in one batch. (Zammetti 2020.)

3.4.5 Adding styling with React

Behind the scenes of React library, everything is built inside the application by HTML, CSS, and JavaScript. The components are rendered based on the combination of those. However, React library supports styling the UI with CSS in three traditional ways. The first method is using the `<style>` tag to inject styling into the application. It is the same way as adding internal CSS styling to an HTML page. The second method is creating an external `.css` file for styling and importing it to the application. This method

is the best practice since it separates styling away from other files and reduces later maintenance complexity. The third method of adding styling is inline style, which adds CSS styling directly to an element using style prop. Figure 13 shows a CSS styling was added into a <h2> tag as an inline CSS by using the style prop. The value of the style prop must be an object, which contains valid CSS properties and values, for example, color and font-size properties. Usually, CSS styling is added directly to the definition of any React component to achieve the encapsulation of Object-oriented Programming. (Zammetti 2020, 60 -62.)

```

10 //Pure React and JavaScript code
11 function start() {
12     //Bookmark Component
13     class Bookmark extends React.Component {
14         constructor(props) {
15             super(props);
16             console.log("Bookmark component created");
17             this.title = this.props.title;
18             titleStyle = { color : "red" };
19         }
20         render() {
21             return (
22                 React.createElement("li", { },
23                     React.createElement("h2", { style : this.titleStyle }, this.title),
24                     React.createElement("a",
25                         { href: this.props.href }, this.props.description),
26                     React.createElement("button", {
27                         onClick: () => {
28                             this.title = this.title + "-CHANGED";
29                             this.setState({});
30                         }
31                     }, "Click me")
32                 )
33             );
34         }
35     }

```

FIGURE 13. Adding CSS styling in the definition of the Bookmark component

3.5 The introduction to JSX

The previous sections provided roughly the core ideas, basic structures of React library using the raw syntax, which is cumbersome as using DOM API. In section 3.2, a mentioned technology called Babel played a crucial role in simplifying the syntax of React and improving the developing process. Using Babel, React gives developers another option to work with a special syntax called JSX, a syntax extension to JavaScript. Figure 14 shows JSX is a mixture of HTML and JavaScript, which gives a more straightforward and declarative structure to any React project. (Facebook 2021.)

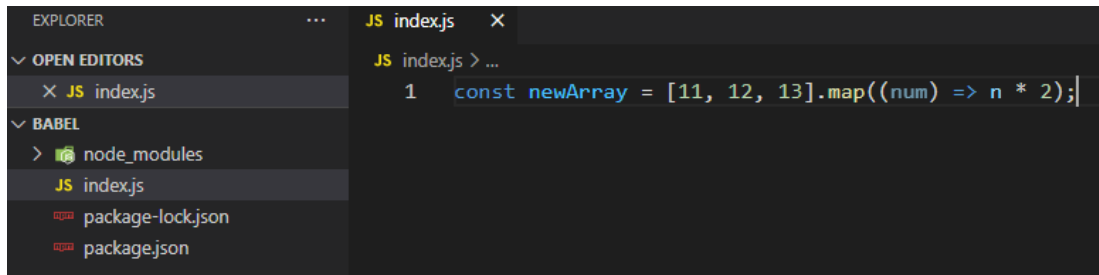
```
const element = <h1>Hello, world!</h1>;
```

FIGURE 14. A JSX element

3.5.1 The Babel transpiler

Before version 0.14, React used an in-house tool called JSXTransformer to translate their JSX syntax into a pure JavaScript syntax so that the web browsers can understand and run the code (O'Shannessy 2015). Since 2015, after version 0.14, React library has gone through a major change when developers split it into two separate packages, React package and React DOM package. This separation serves the clearness and essence of React, which does not have any functionality related to the browser or the DOM. Another major update from React library was the compiler optimization when React library started to take advantage of the JavaScript syntactic sugar, the ECMAScript 6 from Babel. (Alpert 2015.)

Babel is technically a transpiler, which means it transforms and compiles code at the same time. Babel compiles JSX and transforms the new ECMAScript 6 syntax into the syntax that allows every browser to work without throwing errors. There are two ways to work individually with Babel. Babel compiler is available online at <https://babeljs.io/>, where it allows developers to write JSX or TypeScript and see the compiled results. The other way is to install Babel to a local machine by using NPM or any other package manager tools. The first step of installing Babel is initializing a Node project by the command `npm init`. After creating a node project, Babel can be installed by `npm` through the command line. As mentioned above, Babel is capable of transforming the new syntax of JavaScript to the version that every browser supports. The arrow function in JavaScript is an example. The arrow function works in the latest versions of Chrome, Firefox, and Opera but does not work in Internet Explorer 11. So to make sure every new JavaScript-based project does not break in Internet Explorer 11, Babel changes them into the older syntax. Figure 15 shows a node project contains an `index.js` file, and inside the `index.js` file uses an arrow function. (Zammetti 2020, 68 -70.)



```

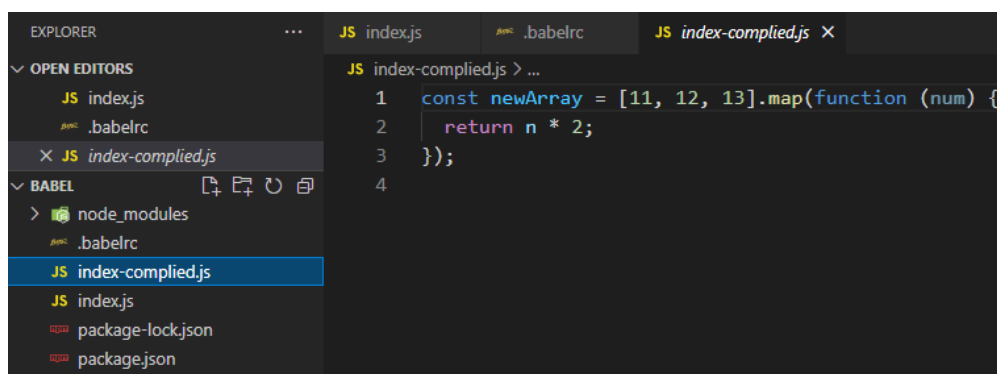
EXPLORER
  OPEN EDITORS
    JS index.js
  BABEL
    node_modules
    JS index.js
    package-lock.json
    package.json

JS index.js
1  const newArray = [11, 12, 13].map((num) => n * 2);

```

FIGURE 15. An example ES 6 arrow function inside a Node project

Babel has several plugins, and each serves different purposes. In this case, to transform the arrow function, Babel has a specific plugin to perform this feature. This plugin can be downloaded by npm through the command line. After the plugin is downloaded, it still needs to be defined to work correctly inside a Node project. Every plugin is defined inside a `.babelrc` file. When the `.babelrc` file defines the plugin, Babel could transform the code inside the project according to the plugin's feature by issuing the `npx babel` command. The transformation of the code in Figure 16 was made by Babel. The arrow function was converted into a standard JavaScript function, and the code now works in even Internet Explorer 11 or the latest versions of Chrome, Firefox. In addition, Babel also supports outputting the result into a new file instead of displaying it on the console. The command `"npm babel index.js --out-file index-compiled.js"` was used to create a new file, which contained all the code that Babel transformed. Figure 16 also shows the result after the command was issued, the `index-compiled.js` file was created inside the project where all the transformed code was stored. (Zammetti 2020, 70 -71.)



```

EXPLORER
  OPEN EDITORS
    JS index.js
    .babelrc
    JS index-compiled.js
  BABEL
    node_modules
    .babelrc
    JS index-compiled.js
    JS index.js
    package-lock.json
    package.json

JS index-compiled.js
1  const newArray = [11, 12, 13].map(function (num) {
2    return n * 2;
3  });
4

```

FIGURE 16. An example new file is created by Babel

However, it is inconvenient to download a plugin and configure it to perform a single JavaScript feature. Babel provides the presets, where it logically groups many plugins and allows developers to enable all in one batch. There are many presets, but the two most popular presets are `env` and `react`, which were developed specifically for React library. Babel presets also need to be downloaded and defined to work inside the `.babelrc` file. After downloading the preset, it needs to be configured to work as the developers

desire in the `.babelrc` file. Figure 17 shows the `env` preset in a `.babelrc` file, and this preset informed Babel to tweak the transformation of the code to be compatible with the versions defined inside the file. (Zammetti 2020, 70 -71.)



The screenshot shows a VS Code editor with three tabs: `index.js`, `.babelrc`, and `index-complied.js`. The `.babelrc` file is open and contains the following JSON configuration:

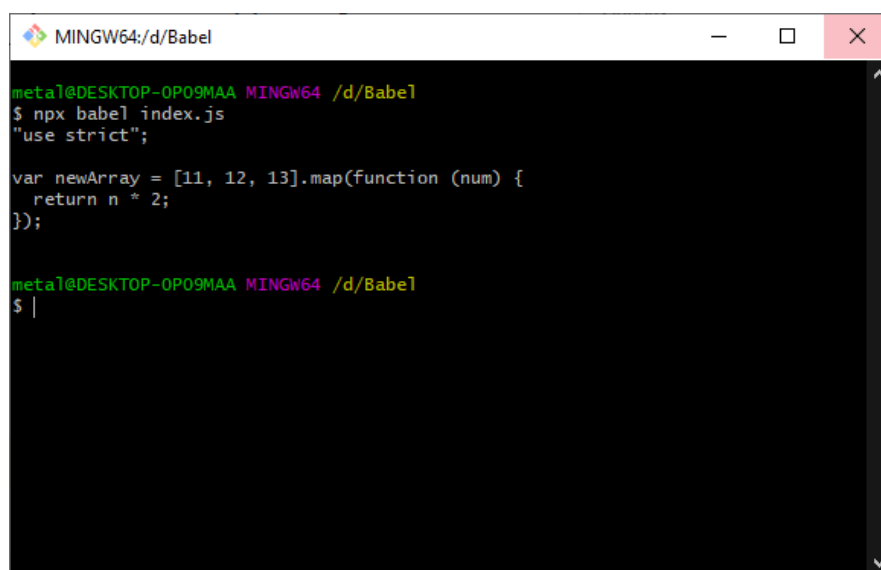
```

1 {
2   "presets": [
3     [ "@babel/preset-env", {
4       "targets": {
5         "browsers": [ "last 3 versions", "safari >= 6" ]
6       }
7     } ]
8   ]
9 }

```

FIGURE 17. The `env` preset is defined and configured in a `.babelrc` file

As in Figure 17, Babel had to transform the code to be compatible with the last three versions of all browsers and version 6 or above for Safari. When the configuration is finished, Babel could transform the code by issuing the `npx babel` command, and the result is different depending on how the preset is configured. Figure 18 illustrates when the command `"npx babel index.js"` was issued, the `env` preset was used to transform the ES arrow function. The only difference from the result in Figure 16 was the `"use strict";` mode added on top. Since the `env` preset was built on many plugins, Babel can use this preset to perform many JavaScript conversions without importing numerous plugins for every compilation in one project. (Zammetti 2020, 70 -71.)



The screenshot shows a terminal window titled `MINGW64:/d/Babel`. The user has run the command `npx babel index.js` and the output is as follows:

```

meta@DESKTOP-0P09MAA MINGW64 /d/Babe1
$ npx babel index.js
"use strict";

var newArray = [11, 12, 13].map(function (num) {
  return n * 2;
});

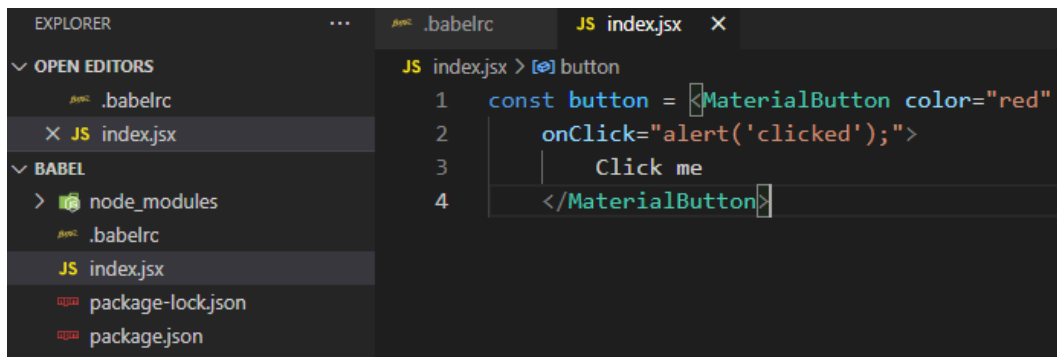
meta@DESKTOP-0P09MAA MINGW64 /d/Babe1
$ |

```

FIGURE 18. The result after transforming the arrow function

3.5.2 Compiling JSX with Babel

The way Babel compiles JSX syntax to standard JavaScript syntax is the same as it transforms ES6 syntax to standard JavaScript. Babel has a specific preset built only for compiling JSX; it is the react preset. And to work with this preset, it also needs to be downloaded inside a Node project using a download command from either NPM or any other node package manager tools. The next step was defining and configuring the preset inside the `.babelrc` file, the same as the env preset. The react preset compiles exclusively the JSX syntax code. Hence, the Node project needs to contain JSX based-code to test the react preset. In Figure 19, an `index.jsx` file was created, which is a button with a color property in it. (Zammetti 2020, 71 -72.)



The screenshot shows a code editor with two tabs: `.babelrc` and `JS index.jsx`. The `index.jsx` file contains the following code:

```

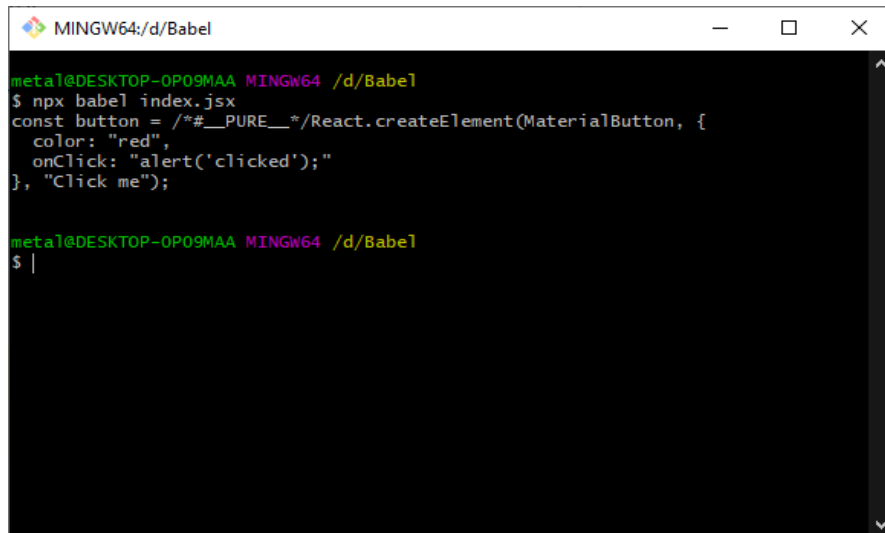
1  const button = <MaterialButton color="red"
2    onClick="alert('clicked');">
3    Click me
4  </MaterialButton>

```

The Explorer sidebar on the left shows the project structure with files like `.babelrc`, `index.jsx`, `node_modules`, `package-lock.json`, and `package.json`.

FIGURE 19. An example button with props written in JSX

The `npx babel` command was issued to instruct Babel to compile the JSX code through the react preset. After the `npx babel` command was executed, Figure 20 demonstrates Babel's standard JavaScript code compiled from the JSX button element in Figure 19. The syntax in figure 20 is also identical to the syntax that was used to create the Bookmark class components in previous sections. In section 3.4, the Bookmark components were created by using the raw React syntax, and it was recreated in this section. However, the component in this section was recreated in the JSX syntax to see how convenient JSX syntax is, and the project used the CDN from Facebook instead of downloading and importing React library to the local files. (Zammetti 2020, 71 -72.)



```

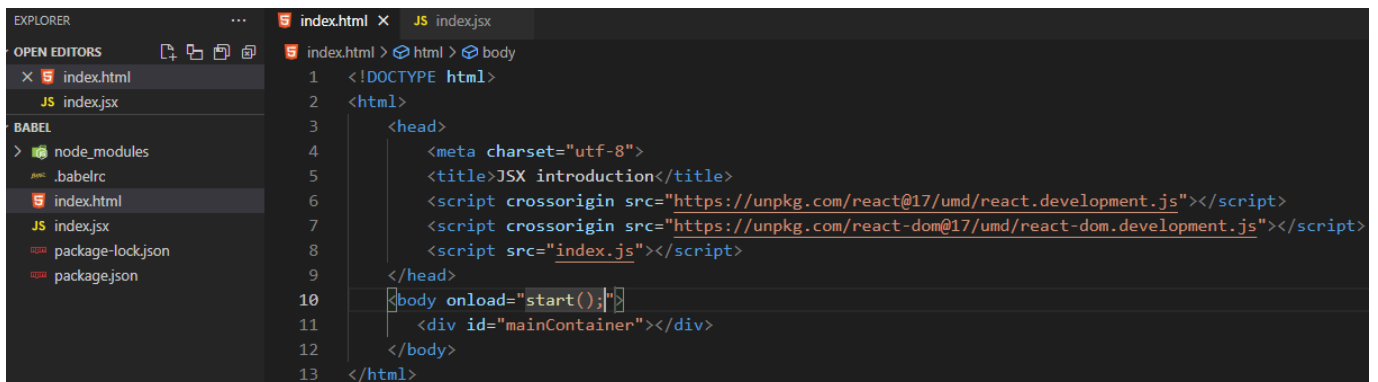
MINGW64:/d/Babel
meta1@DESKTOP-0P09MAA MINGW64 /d/Babel
$ npx babel index.jsx
const button = /*#__PURE__*/React.createElement(MaterialButton, {
  color: "red",
  onClick: "alert('clicked');"
}, "Click me");

meta1@DESKTOP-0P09MAA MINGW64 /d/Babel
$ |

```

FIGURE 20. A result after Babel compiling a JSX code

Figure 21 shows an HTML page where a `<div>` tag with an id of `mainContainer` lived. The term container is also referred to a `<div>` tag where all the components are rendered. In other words, this `mainContainer` `<div>` tag contains all the components of the project. In Line 8, a script tag was used to import an `index.js` file, although at the moment the `index.js` file is not created yet. The plan was to use Babel to compile all the JSX code inside the `index.jsx` into the standard JavaScript code and store them in an `index.js` file. The process was similar to the above examples. However, this example introduced some new ways to use the props object to store and pass data to components. (Zammetti 2020, 72 -73.)



```

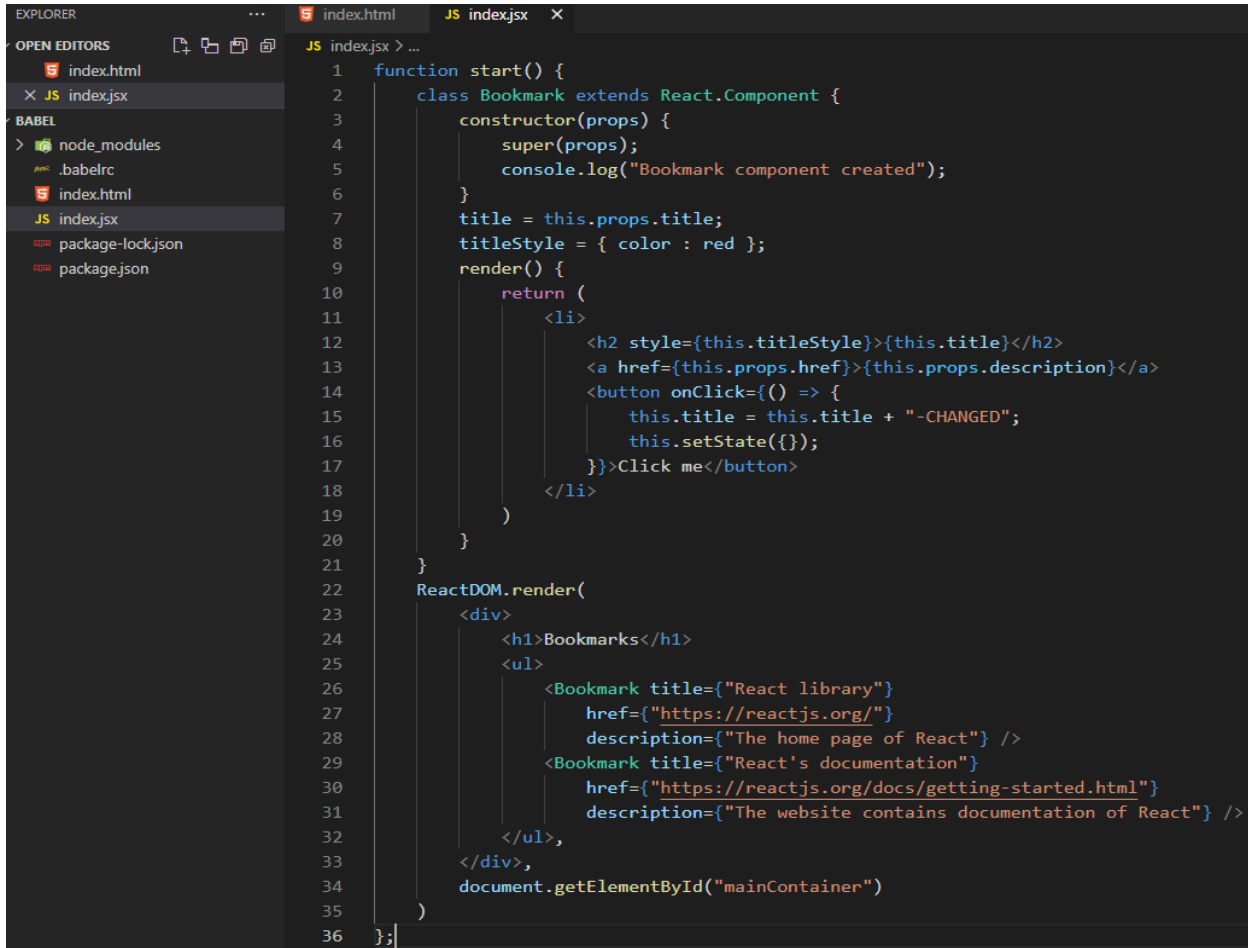
EXPLORER
  OPEN EDITORS
    index.html
    JS index.jsx
  BABEL
    node_modules
    .babelrc
    index.html
    JS index.jsx
    package-lock.json
    package.json
  index.html
    1 <!DOCTYPE html>
    2 <html>
    3   <head>
    4     <meta charset="utf-8">
    5     <title>JSX introduction</title>
    6     <script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>
    7     <script crossorigin src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
    8     <script src="index.js"></script>
    9   </head>
    10  <body onload="start();" >
    11    <div id="mainContainer"></div>
    12  </body>
    13 </html>

```

FIGURE 21. An example HTML page contains a `mainContainer`

Figure 22 shows a new `Bookmark` class component, which was defined inside the `index.jsx` file. Later, `React DOM` rendered the `Bookmark` component with the `ReactDOM.render()` method. Thanks to JSX syntax, the `Bookmark` components inside the `ReactDOM.render()` method were now more readable. All the previous `Bookmarks` components inside the `render()` method were wrapped inside the `React.createElement()`, which was more complicated when the project expanded to a larger scale and contained

hundreds of components; it can cause significant difficulty for maintenance. In figure 22, the JSX syntax helped omit all the `React.createElement()` methods and replaced them with a friendly syntax, which was identical to a mixture of HTML tags and JavaScript objects. (Zammetti 2020, 73 -74.)



```

1  function start() {
2      class Bookmark extends React.Component {
3          constructor(props) {
4              super(props);
5              console.log("Bookmark component created");
6          }
7          title = this.props.title;
8          titleStyle = { color : red };
9          render() {
10             return (
11                 <li>
12                     <h2 style={this.titleStyle}>{this.title}</h2>
13                     <a href={this.props.href}>{this.props.description}</a>
14                     <button onClick={() => {
15                         this.title = this.title + "-CHANGED";
16                         this.setState({});
17                     }}>Click me</button>
18                 </li>
19             )
20         }
21     }
22     ReactDOM.render(
23         <div>
24             <h1>Bookmarks</h1>
25             <ul>
26                 <Bookmark title={"React library"}
27                     href={"https://reactjs.org/"}
28                     description={"The home page of React"} />
29                 <Bookmark title={"React's documentation"}
30                     href={"https://reactjs.org/docs/getting-started.html"}
31                     description={"The website contains documentation of React"} />
32             </ul>,
33         </div>,
34         document.getElementById("mainContainer")
35     )
36 };
```

FIGURE 22. An example JSX Bookmark class component

The JSX syntax also helped the chaining of components become straightforward since every component had a structure of an HTML tag with props as element attributes. During the experiment of this thesis, the class was a new feature of JavaScript that was not supported in the react preset, Babel was required to download another plugin to compile this class feature into a compatible code for the react preset. After installing, the plugin needed to be defined inside the `.babelrc` file before using it. The next step was to compile the `index.jsx` file, store the compiled code to an `index.js` file, import the code from the `index.js` to the `index.html`, and test the result in a browser. Once the command `"npm babel index.jsx --out-file index.js"` was executed, Babel started compiling the JSX inside the `index.jsx` file and creating an `index.js` file to store the result in it. After the JSX was compiled, it was imported into the `index.html` file by the `<script>` on line 8 in Figure 21. And Figure 23 shows the result of the JSX code after finishing compiled by Babel. The `index.js` file was created by Babel, containing all the code that was compiled from JSX.

The compiled code was partly identical to the raw React code in the previous example, and both function in the same way, giving the same result. (Zammetti 2020, 75 -76.)

```

3   { Object.defineProperty(obj, key, { value: value, enumerable: true, configurable: true, writable: true }); }
4   else { obj[key] = value; } return obj; }
5
6   function start() {
7     class Bookmark extends React.Component {
8       constructor(props) {
9         super(props);
10
11        _defineProperty(this, "title", this.props.title);
12
13        _defineProperty(this, "titleStyle", {
14          color: red
15        });
16
17        console.log("Bookmark component created");
18      }
19      render() {
20        return /*#__PURE__*/React.createElement("li", null, /*#__PURE__*/React.createElement("h2", {
21          style: this.titleStyle
22        }, this.title), /*#__PURE__*/React.createElement("a", {
23          href: this.props.href
24        }, this.props.description), /*#__PURE__*/React.createElement("button", {
25          onClick: () => {
26            this.title = this.title + "-CHANGED";
27            this.setState({});
28          }
29        }, "Click me"));
30      }
31    }
32
33    ReactDOM.render( /*#__PURE__*/React.createElement("div", null, /*#__PURE__*/React.createElement("h1", null, "Bookmarks"),
34      /*#__PURE__*/React.createElement("ul", null, /*#__PURE__*/React.createElement(Bookmark, {
35        title: "React library",
36        href: "https://reactjs.org/",
37        description: "The home page of React"
38      })), /*#__PURE__*/React.createElement(Bookmark, {
39        title: "React's documentation",
40        href: "https://reactjs.org/docs/getting-started.html",
41        description: "The website contains documentation of React"
42      })), "", document.getElementById("mainContainer"));
43  };
44

```

FIGURE 23. The result after transforming the JSX component

3.6 Component Lifecycle

There are two ways to create a React component, and both were introduced in section 3.4. Before version 16.8, React was still heavily dependent on class components and Object-oriented Programming. With class components, React gives developers the possibility to create the application with the Object-oriented style, which stimulates their components abstractly close to the real-life objects. And it also gives developers more methods to control each component, known as the component lifecycle. However, it brings more complexity to the project when too many projects and relationships exist between them.

When React version 16.8 was released, the functional component became popular. The functional components are sometimes known as hooks. They are written in functional programming style and are easier to handle because of their friendlier syntax than the class components. Despite the difference in syntax between hooks and class components, when Babel compiles them into the standard code, they are all proved to have the same functionality (Facebook 2021).

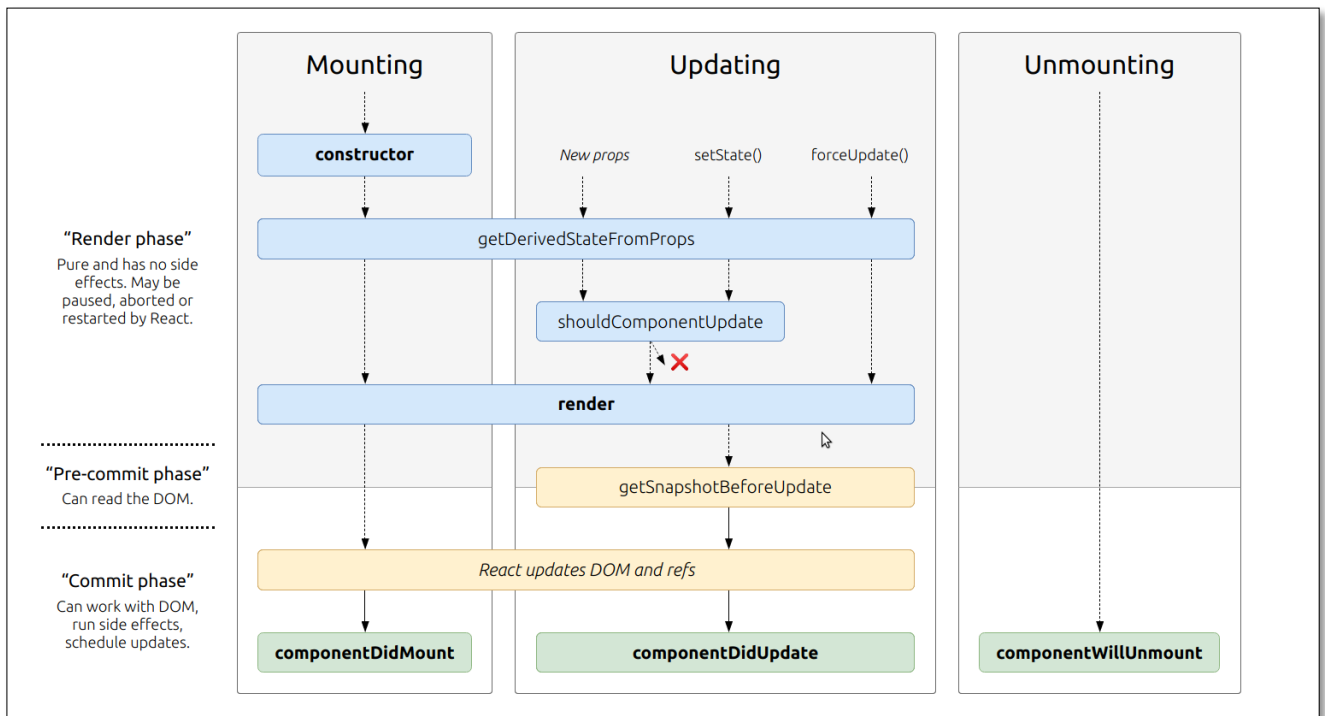


FIGURE 24. The lifecycle event of a React component (Zammetti, 2020).

The lifecycle event exists in both functional components and class components. However, it has many specific methods that can only be used in class components. Figure 24 is the lifecycle event model of a React component when it is created, updated, and destroyed. When a component is mounting or being created, the constructor is the first method evoked. It is similar to any other Object-oriented Programming language when creating a new object. The `getDerivedStateFromProps()` is the next method evoked after the constructor, which returns an object to update the state or null if nothing needs to update. The `getDerivedStateFromProps()` method depends on two arguments: the new props and state that need to be updated. The `getDerivedStateFromProps()` method is evoked during three specific processes. It is evoked when a component is being constructed (mounting), added to the virtual DOM (mounted), or being updated (updating); however, it has rare use-cases. The alternative of this method in a functional component is the `useState()` hook, which also returns a new object containing information to update the state or null if there is no update. After the component is added to the virtual DOM, React immediately calls the `componentDidMount()` method. Any process related to using network requests or initializing

the DOM, for example, loading images, videos, and maps, can be called inside this method. (Facebook 2021.)

When a component is updating, the sequence of methods is different. The `getDerivedStateFromProps()` method is evoked first when a component is updating. Based on the result of the `shouldComponentUpdate()` method, React then executes the `render()` method during the updating process. React evokes the `shouldComponentUpdate()` method automatically before rerendering a new component. The `shouldComponentUpdate()` method can also be used to manually exit the lifecycle and prevent the waste of resources on unnecessary rerendering. After the `render()` method is evoked during the updating process, the `getSnapshotBeforeUpdate()` method is called. This method captures information before the component commits the changes to the real DOM. The information is the states of components before they are updated. However, React does not recommend interfering with the structure of this method since it can cause errors when it commits the changes to the real DOM tree. Finally, the `componentDidUpdate()` method is evoked, which is also suitable for handling tasks related to network requests. The alternative way to replace the `componentDidMount()` and `componentDidUpdate()` methods is to use the `useEffect()` hook. The `useEffect()` hook can be used to work with network requests or any side effects in functional components. The last process of a lifecycle event is when a component is being destroyed. When the component is being destroyed, the `componentWillMount()` method is evoked. The component technically performs the cleanup tasks to prevent the memory leak, for instance, deleting a component from the DOM, canceling network requests that have been created by the `componentDidMount()`, or `componentDidUpdate()` methods. For functional components, the `useEffect()` hook also returns a function that helps handle the cleanup tasks similarly with the `componentWillMount()` method in class components. (Zammetti 2020, 84 -85.)

4 NODE.JS AND EXPRESSJS

Nowadays, there are numerous options of tools, platforms, and frameworks for back-end development, such as the famous PHP, Django, ASP.net. And when JavaScript became widely used, developers started to expand its territory. As the result, Node.js was born to give developers an option to use JavaScript over the fence. With Node.js, JavaScript is not limited to the client-side like it used to be. In the previous chapter 2, Node.js is presented as a downloadable runtime, allowing JavaScript to run on any local machine. Furthermore, Node.js is also capable of turning any machine into a server to host a website (Bojinov 2018, 27 -29). This chapter introduces the back-end development with Node.js and Express.js, a framework that is built specifically for Node.js development. (Ijas 2020.)

4.1 Introduction to API and RESTful API

The API term is popular in Software Engineering, Web Development, or any Information Technology field. API stands for Application Program Interface, which is a communication method between applications. API started to emerge as the beginning of internet commercialization. Three web APIs that marked their successes and changed web development are Salesforce, eBay, and Amazon. Salesforce published their API on 7th February 2000, at the IDG Demo conference. Salesforce introduced the API and defined it as a product of the Internet as a Service. Follow Salesforce was eBay when they launched their API on 20th November 2000. The API of eBay was created to help sellers manage their eBay model businesses at scale. And on 16th July 2002, Amazon published Amazon Web Services, which later became a giant in the web development field. (Lane 2019.)

There are many standards to build an API, and REST is one of the most famous architectural styles. REST stands for Representational State Transfer, which consists of a set of principles as a guideline. The first rule is the uniform interface, which requires communication between the front-end and back-end through relative Uniform Resource Identifiers (URIs). Communication can be one of four tasks: retrieving a resource, creating a resource, updating a resource, or removing a resource. The second rule of REST API is the separation between client and server to improve the portability of the front-end and scalability for the back-end of a website. The third rule is statelessness; every request that comes to the server must provide enough information since there is no session establishment between the front-end and back-end. The fourth rule is cacheable, which allows a response to be cacheable by the client-side. The sixth rule is the layered system, which allows developers to deploy their API on one server, but the

API can work with multiple servers. For instance, a REST API is deployed on server A, manipulates data in server B, and authenticates requests in server C. (Bojinov 2018, 12 -13.)

4.2 Node.js in the server-side

Every REST API lives in the server machine, so the first step in building a REST API is to build a server machine. By using Node.js, it allows a machine to perform and execute the server-side task. As mentioned above, Node.js is capable of turning a local machine into a server machine for the development process. (Bojinov 2018, 27 -29.)

4.2.1 Starting a Node.js server from scratch

Figure 25 is a testing file named `app.js`, which was created to check whether the local machine could perform as a developing server or not. In the first line, there is a constant named `http`, which requires something else, also called `http`. The `http` constant in Figure 25 is a built-in module, which is available from Node.js. This module is designed to support the basic features of the HTTP protocol. And the `createServer()` method is a part of the `http` module, which does exactly like the name, establishing a Node.js server. Node.js runtime consists of many modules; each has a set of features and is always available for direct integration in a Node.js project with the `import` keyword. For example, the `http` module is one of many core modules of Node.js. In Figure 25, the `createServer()` method received a function as an argument; this function is called the request listener function. The request listener receives two arguments, `res` and `req`, which stand for request and response. Request and response are two objects, which present for the HTTP request and response. (OpenJS 2021.)

```
JS app.js > ...
1  const http = require('http');
2
3  const server = http.createServer((req, res) => {
4    res.setHeader('Content-Type', 'text/html');
5    res.write('<h1>Hello from the Node.js Server</h1>');
6    res.end();
7  });
8  console.log("Server started in port 3000");
9  server.listen(3000);
```

FIGURE 25. Creating a Node.js server

An HTTP request consists of an HTTP method, the URL of the resource, the HTTP protocol version, optionally a header, or a body. On the other hand, the HTTP response consists of the HTTP protocol, a status code, a status message, HTTP headers, and optionally a body for the requested resource. The

HTTP methods are usually presented as verbs, such as GET for a get request and POST for a post request. The headers of a request and response provide additional information to the browser and the server. The additional information can be the requested or responded resource's data, the request to access cross-origin sites. In Figure 25, the header was added in the response object to inform the client that the responded data is HTML/text type. The application could be executed by announcing Node.js runtime with the command "node app.js" in the command line. (MDN 2021.)

Figure 26 illustrates the result after the command was executed, the server was live in port 3000 of the localhost. The default HTTP method of a URI is the GET method, which means anytime a URI is run by the browser, the browser sends the request to access the resource by the URI as default. The ultimate goal of a REST API is to manipulate the resources. The resources might have different data types, such as JPEG images, videos, text documents, or binary data. And these resources should be accessible with REST API via URIs, and the URIs should be identified uniquely. Node.js allows developers to create unique identifiers by routing. (Bojinov 2018, 51 -53.)

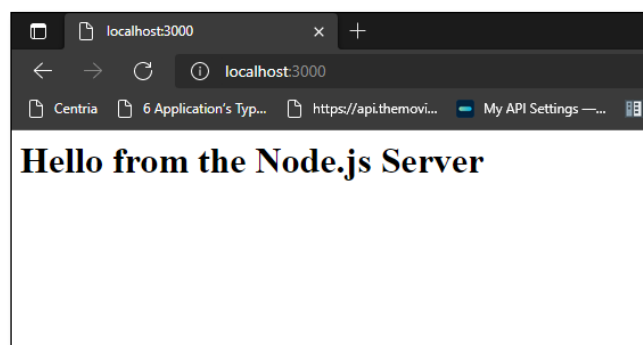


FIGURE 26. The response from the Node.js server in port 3000

4.2.2 URIs structure and routing

The forward slash character is conventionally used to convey the hierarchy of the resources. Sometimes, the resources can be stacked and stored in multiple levels of structure. The forward slashes are often used in a URI to simulate the structure levels, storing location, or to access a specific level to get a piece of data (Allamaraju 2010, 76 -77.) Some segments of a URI can be random strings, but in most cases, a URI is designed to show the purpose or to illustrate the context behind it. Figure 27 demonstrates the convention of a standard URI. The purpose of human-readable URIs is to simplify structure and logic to developers and keep logical errors to a minimum. (Bojinov 2018, 8 -9.)

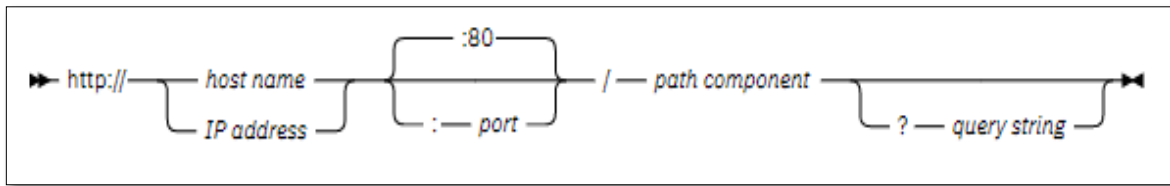


FIGURE 27. The syntax of a URI or HTTP URL (IBM 2020).

Routing in Node.js technically is designing the path segments for accessing resources. For instance, in Figure 28, the segments behind <http://www.example.org> are called paths. A URI also consists of another part called a query string. A query string is a regular string of name and value pairs, and an ampersand separates each pair. For example, `term=bluebird&source=browser-search` (IBM 2020.) In Node.js, every request and response has a `url` property, allowing developers to navigate the request and response. By using the URL properties, any URI can be designed to fit the logic of the API. (Bojinov 2018, 8 -9.)

```
http://www.example.org/messages/msg123
http://www.example.org/customer/orders/order1
http://www.example.org/earth/north-america/canada/manitoba
```

FIGURE 28. An example of hierarchical association between path segments (Allamaraju 2010).

In Figure 29, the `if` statements in lines 6 and 12 used the `url` property of the response to set up the paths to access resources; when the path parameter of the URI matched these paths, the URI accessed the code inside the `if` blocks. The first path in line 6 was the `'/'`, which was the default route. In the default route, the browser sent a GET request to it. The response from the default route was a form with a button named Send. When the button was clicked, the browser sent a POST request to the `'/message'` route, which activated the code in line 12. When the POST request was sent, the `writeFileSync()` method from the `fs` module created a `message.txt` file and stores the DUMMY text. When the file was successfully created, the response's status code was set as 302, and the page was redirected to the `'/'` route. (OpenJS 2011.)

```

JS app.js > [0] server > http.createServer() callback
1  const http = require('http');
2  const fs = require('fs');
3
4  const server = http.createServer((req, res) => {
5      const url = req.url;
6      const method = req.method;
7      if(url === '/') {
8          res.write('<html>');
9          res.write('<form action="/message" method="POST"><input type="text" name="message"><button>Send</button></form>');
10         res.write('</html>');
11         return res.end();
12     }
13     if(url === '/message' && method === 'POST'){
14         fs.writeFileSync('message.txt', 'DUMMY');
15         res.statusCode = 302;
16         res.setHeader('Location', '/');
17         return res.end();
18     }
19     res.setHeader('Content-Type', 'text/html');
20     res.write('<p>Hello from the Node.js Server</p>');
21     res.end();
22 });
23 console.log("Server started in port 3000");
24 server.listen(3000);

```

FIGURE 29. Routing with url properties in Node.js

The information of the request was checked inside the Network tab of the browser's developer tools. The developer tools display the name, sizes, time, and waterfall charts of the requests and responses. When the browser redirected, a message.txt was created and was viewed inside a text editor. In summary, Figure 30 shows a result of the client-server interaction, where a client sent signals to the server to tell the server to generate data. (Allamaraju 2010.)

The screenshot shows a web browser window with a form containing a text input field and a 'Send' button. Below the browser window, the Network tab of the developer tools is open, displaying a waterfall chart and a list of requests. The requests are as follows:

Name	Status	Type
message	302	document / Redirect
localhost	200	document
favicon.ico	200	text/html

FIGURE 30. The browser redirects after the POST request is successfully sent

4.2.3 Data transmission with streams and buffer

The majority of HTTP methods use their bodies as a place for data transmission. On any website, the data is not delivered or received in its original form, but it is converted into sequences of primitive data for efficient transmission. For example, Figure 31 shows the write method of the HTTP response object, which works with data by sending a data chunk of the response body. Noticeably, the write method was called multiple times to provide successive parts of the response body. Node.js automatically converted data into chunks of hexadecimal values. A hexadecimal number is able to represent a larger value than a binary number in a short form, which allows these chunks to hold more data but remain low-cost in transportation (OpenJS 2021.)

```

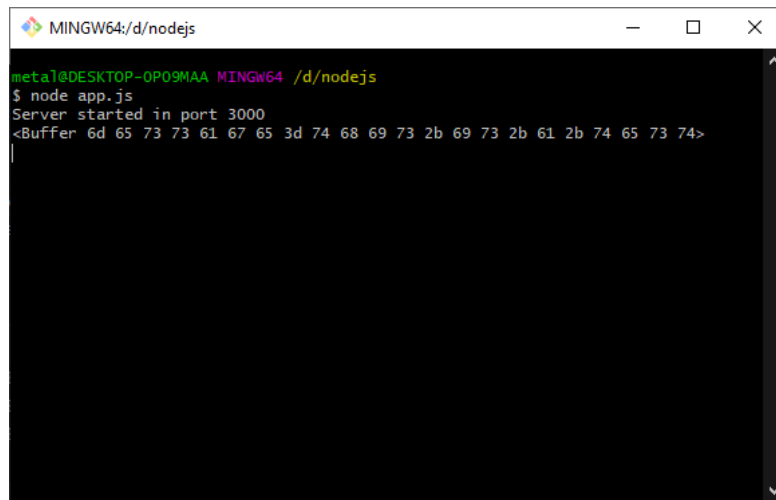
JS app.js > [Ⓜ] server > [Ⓜ] http.createServer() callback > [Ⓜ] req.on('end') callback
1  const http = require('http');
2  const fs = require('fs');
3
4  const server = http.createServer((req, res) => {
5    const url = req.url;
6    const method = req.method;
7    if(url === '/') {
8      res.write('<html>');
9      res.write('<form action="/message" method="POST"><input type="text" name="message"><button>Send</button></form>');
10     res.write('</html>');
11     return res.end();
12   }
13   if(url === '/message' && method === 'POST'){
14     const body = [];
15     req.on('data', (chunk) => {
16       console.log(chunk);
17       body.push(chunk);
18     });
19     req.on('end', () => {
20       const parsedBody = Buffer.concat(body).toString();
21       const message = parsedBody.split('=')[1].replace(/\+/g, ' ');
22       fs.writeFileSync('message.txt', message);
23     });
24     res.statusCode = 302;
25     res.setHeader('Location', '/');
26     return res.end();
27   }
28   res.setHeader('Content-Type', 'text/html');
29   res.write('<p>Hello from the Node.js Server</p>');
30   res.end();
31 });
32 console.log("Server started in port 3000");
33 server.listen(3000);

```

FIGURE 31. Manipulating data stream with Buffer class

Streams are a special interface that Node.js uses for data transmission. A stream is a sequence of data broken down from a large piece of data into small chunks. These chunks of data are transmitted to another place one by one, and it becomes a stream. For example, a client downloads a paragraph; instead of sending the whole paragraph at once, the server splits the paragraph into small chunks and sends them one by one. And Node.js added another feature for developers to interact with the data chunks, called the buffer. Buffer is a class added in Node.js API to manipulate or interact with data streams. (OpenJS

2011.) Buffer in the computer is a small physical location in the RAM, where data is temporally stored and wait for processing during streaming. In some programs, because some processes consume data faster than it arrives, some data chunks that arrive early need to wait until a certain amount of data arrives fully before the computer sends them all out for processing. In other words, a buffer is a place that stored the data chunks that arrive early before processing. (Mba 2017.)



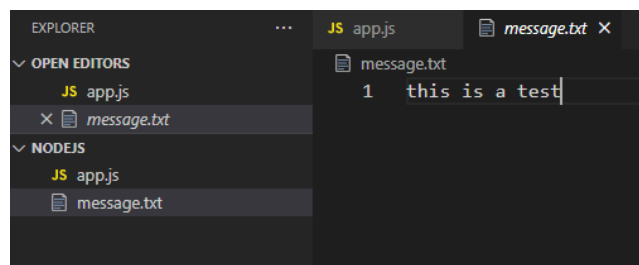
```

MINGW64/d/nodejs
meta1@DESKTOP-0P09MAA MINGW64 /d/nodejs
$ node app.js
Server started in port 3000
<Buffer 6d 65 73 73 61 67 65 3d 74 68 69 73 2b 69 73 2b 61 2b 74 65 73 74>

```

FIGURE 32. The console logs the hexadecimal chunks in buffer

On line 15 of Figure 31, an `on()` method listened to certain events. The data event fired when a new chunk of data was ready to be processed. These chunks of data were stored in an array named `body`, concatenated together by the buffer class, and converted into a string on line 20. In Figure 32, the data chunks in the buffer were displayed by the hexadecimal numeral system. Each hexadecimal number was a chunk of data, and this is how a stream is split and transferred. Figure 33 shows the result of the data chunks that were converted back to the human-readable text and stored in a `message.txt` file, it was also the form input in the `/` route and it was not the predefined DUMMY text as the previous example. (OpenJS 2021.)



```

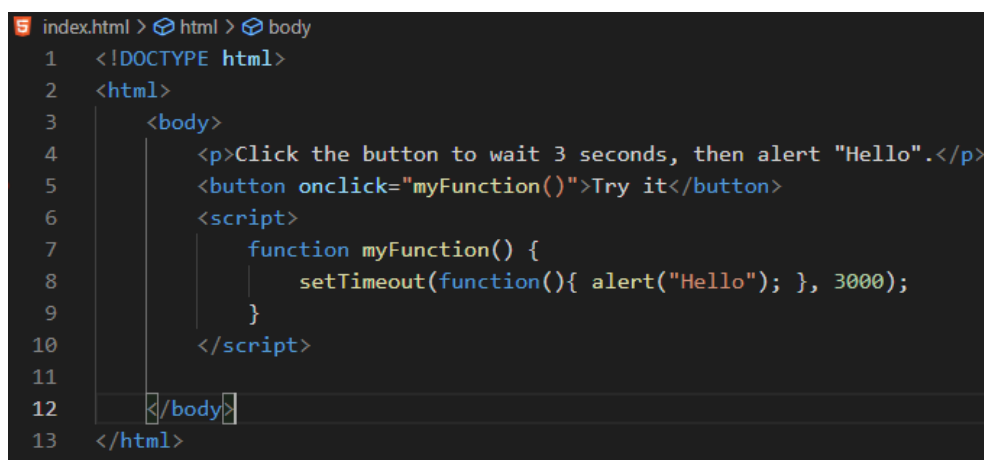
EXPLORER
OPEN EDITORS
  JS app.js
  message.txt
  X message.txt
NODEJS
  JS app.js
  message.txt
JS app.js
message.txt
1 this is a test

```

FIGURE 33. The text is converted from the data chunks in buffer

4.2.4 Node.js and asynchronous programming

Since every code inside a Node.js project works in a single thread, some code might block the other code and break the program's execution flow. However, thanks to the asynchrony of Node.js, it keeps the flow run smoothly just inside one single thread. An asynchronous program contains code that runs straight along, and everything inside the program runs one at a time. If a function relies on the result of another function, it has to wait until the other function is executed and returns the result to keep on doing its work. JavaScript has the callback function as an approach to asynchronous programming. A callback function is a function that is called after an action finishes with a result. (MDN 2021.)

A screenshot of a code editor showing the following code:

```
index.html > html > body
1  <!DOCTYPE html>
2  <html>
3    <body>
4      <p>Click the button to wait 3 seconds, then alert "Hello".</p>
5      <button onclick="myFunction()">Try it</button>
6      <script>
7        function myFunction() {
8          setTimeout(function(){ alert("Hello"); }, 3000);
9        }
10     </script>
11
12   </body>
13 </html>
```

FIGURE 34. An example setTimeout() function

The setTimeout() function in Figure 34 had an anonymous function inside it. The anonymous function gave a "Hello" alert and only executes after the setTimeout() function finished counting down its timer. This anonymous function is also called a callback function because of this asynchronous behavior. Asynchronous programming is a very important programming concept since data transmission cannot be finished at once. There is a standard class called Promise to handle the data fetching or any other tasks related to asynchrony in JavaScript. The most common usage of the Promise class is in fetching data from an API, connecting to a database, or manipulating a database. Promises are a better way to enhance the asynchrony in JavaScript. In the past, when JavaScript relied heavily on callback functions for asynchronous tasks, the nesting of callback functions is the way to arrange and schedule the actions. When the action is too complicated, the callback nest also becomes longer and forms the triangle of callback hell. Promises solve the nesting of callback functions by using the chaining method. Promises can be chained until they are resolved, which is a more efficient way for developing and maintenance. (Liew 2019.)


```

index.html > html > body > script > myPromise > <function>
1  <!DOCTYPE html>
2  <html>
3    <body>
4      <h2>JavaScript Promise</h2>
5      <p id="demo"></p>
6      <script>
7        function myDisplayer(some) {
8          document.getElementById("demo").innerHTML = some;
9        }
10
11       let myPromise = new Promise(function(myResolve, myReject) {
12         let x = 0;
13         // The producing code (this may take some time)
14         if (x == 0) {
15           myResolve("OK");
16         } else {
17           myReject("Error");
18         }
19       });
20
21       myPromise.then(
22         function(value) {myDisplayer(value);},
23         function(error) {myDisplayer(error);}
24       );
25     </script>
26   </body>
27 </html>

```

FIGURE 35. An example of a promise in JavaScript

A promise is an object that is created by the Promise class in JavaScript. A promise may be fulfilled at some point, but it also may fail as a promise in real life. In line 11, when a promise object was first created, the promise was pending or had not delivered a result during this period. Inside the Promise class, there are two methods, resolve and reject. When a promise is fulfilled, the resolve method is executed with the given value. In Figure 35, the promise was completed because the variable `x` had the value of 0. And then, the `myResolve()` method was executed with the "OK" string as an argument in line 22. In contrast, when the promise fails, the reject method is executed with a given reason. For example, when the value of the `x` variable was not 0, the `myReject()` method was executed with the "Error" string as a given reason in line 23. With the new ES6 version of JavaScript, Promises were replaced with async functions, which are shorter and have a straightforward syntax to the asynchronous programming. However, they both share the same mechanism under the hood. An async function and the Promise class both create a new promise when they are executed. (MDN 2021.)

```

index.html > html
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <h2>JavaScript Async</h2>
5          <p id="demo"></p>
6          <script>
7              function myDisplayer(some) {
8                  document.getElementById("demo").innerHTML = some;
9              }
10             async function myFunction() {
11                 let x = 0;
12                 // The producing code (this may take some time)
13                 if (x == 0) {
14                     return "OK";
15                 }
16                 return "Error";
17             };
18             myFunction().then(
19                 function(value) {myDisplayer(value);},
20                 function(error) {myDisplayer(error);}
21             );
22         </script>
23     </body>
24 </html>

```

FIGURE 36. An example of an async function in JavaScript

4.3 ExpressJS, a node.js framework

The back-end of a website is the place that contains, executes, and servers complicated logic. The more complicated logic, the more work has to be done. For instance, in the previous section, when a text was transferred from a client-side to a back-end, it had to be converted into small chunks, delivered one by one, and converted back to regular text. These data converting steps are sometimes too cumbersome and might make the business logic challenging to build. Although Node.js has many modules for various purposes, it still requires many processes to focus on a particular field. ExpressJS was created to solve the problem when Node.js comes to business websites. There are other frameworks of Node.js built for many usages; for example, Adonis.js, Koa, Sails.js. However, they all serve the same purpose of reducing the heavy works and letting the developers focus on building the core logic of any application. (TutorialsTeacher 2020.)

4.3.1 Starting a Node.js server with ExpressJS

ExpressJS can be installed by using NPM or other package manager tools. After ExpressJS's download is finished, it can be imported inside a Node.js project in the same way the other node modules are imported. ExpressJS allows developers to perform tasks directly to the HTTP requests and responses by using middlewares. The HTTP request and response have a lifecycle, where it is finished by sending a response to a client using the `end()` method. During the request-response cycle, the HTTP request or response can be accessed by using middleware functions. These middlewares work as validators or authenticators to authorize and validate resources access or sharing. In Figure 37, the content inside the `app.use()` function is a middleware. An HTTP request or response can travel from one to another middleware by using the `next()` function to keep the request-response cycle alive. The cycle ended in figure 37 when the response sends a `<h1>` tag to the client. (StrongLoop & IBM 2017.)

```
JS app.js > ...
1  const http = require('http');
2  const express = require('express');
3
4  const app = express();
5  app.use((req, res, next) => {
6    console.log('In the middleware!');
7    next();//Allows the request to continue to
8  }); //travel to the next middleware
9  app.use((req, res, next) => {
10   console.log('In another middleware!');
11   res.send('<h1>Hello from ExpressJS!</h1>');
12 });
13
14 const server = http.createServer(app);
15 server.listen(3000);
```

FIGURE 37. Starting a Node.js server with ExpressJS

4.3.2 Routing in ExpressJS

Routing in ExpressJS can be performed by using middleware functions. Instead of using the url property of each HTTP request and response, the `app.use()` middleware receives a path at the first argument and multiple callback functions to perform the logic with the request-response cycle. In Figure 38, there are three paths: `'/'` path, `'/add-product'`, and `'/product'` path. In the `'/'`, any incoming request was allowed to travel to the next middleware; however, the cycle ended in the `'/add-product'` path because of the `res.send()` function. In the `'/add-product'` was a form with the input type text and a button. When the form

was submitted, it sent a POST request that consists of the input data of the form to the '/product' middleware. The '/product' middleware printed the request's body, which contained the data from the form to the console, and redirected the page back to the '/' middleware. (StrongLoop & IBM 2017.)

```

JS app.js > app.use('/product') callback
1  const http = require('http');
2  const express = require('express');
3
4  const app = express();
5
6  app.use('/', (req, res, next) => {
7    console.log('This always runs!');
8    next();
9  });
10 app.use('/add-product', (req, res, next) => {
11   console.log('In another middleware!');
12   res.send('<form action="/product" method="POST"><input type="text" name="title"><button type="submit">Add Product</button></form>');
13 });
14
15 app.use('/product', (req, res, next) => {
16   console.log(req.body);
17   res.redirect('/');
18 });
19
20 const server = http.createServer(app);
21 server.listen(5000);

```

FIGURE 38. Basic routing with ExpressJS

In Figure 38, when the request reached the '/product' middleware, it printed out the undefined in the console instead of printing out the text from the input form. When an HTTP request transfers data, it converts data to different types, which are more convenient and resourceful than using the raw text. It needs a body parser for the request body to convert the data back to an interactable type. Before version 4.16.0, whenever a request body required a body-parser, it had to download it from NPM and import it to the file. Sometimes, the installation was cumbersome, so express added a body-parser as a part of their URL encoder for an HTTP request or response in the express package. Figure 39 shows that within one line of code, the body-parser was ready to use in a Node.js file. (StrongLoop & IBM 2019.)

```

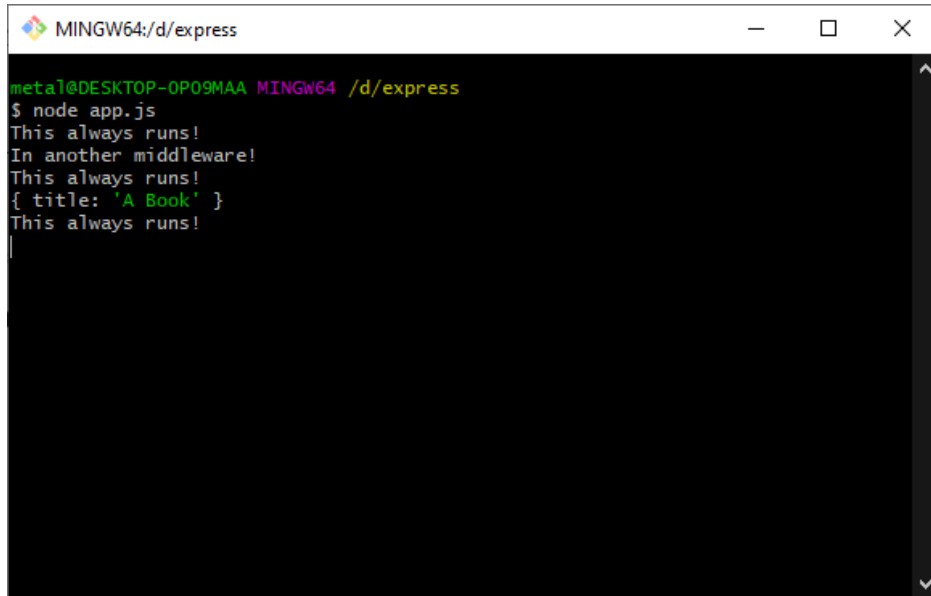
1  const http = require('http');
2  const express = require('express');
3
4  const app = express();
5
6  app.use(express.urlencoded({ extended: true }));

```

FIGURE 39. Import a URL encoder with express after version 4.16.0

After the body-parser is installed and imported, every data in the HTTP request body of the Node.js program can be converted back to an interactable data type. For example, a text was inputted from the

form in the '/add-product' middleware, and then the request contained the input traveled to the '/product' middleware. In the '/product' middleware, the input was converted to a JavaScript, logged object out in the console as in Figure 40. (StrongLoop & IBM 2017.)

A screenshot of a terminal window titled 'MINGW64:/d/express'. The terminal shows the following output:

```
meta1@DESKTOP-0P09MAA MINGW64 /d/express
$ node app.js
This always runs!
In another middleware!
This always runs!
{ title: 'A Book' }
This always runs!
```

FIGURE 40. ExpressJS parses the request body's data

4.3.3 The MVC Pattern

When a project grows to a larger scale, the logic inside it becomes more complicated. So at the beginning of any project, the project has to be planned to follow a significant design pattern. The design pattern is a way of organizing the files and modules of a project into a logical system. The design pattern benefits developers by reducing time searching for a file, a module, or a function inside a big project. The design pattern also helps developers create a maintainable system that can be convenient for anyone who has experienced it and later maintains the project. One of the most popular design patterns for back-end systems is the MVC design pattern. (Bakshi 2019.)

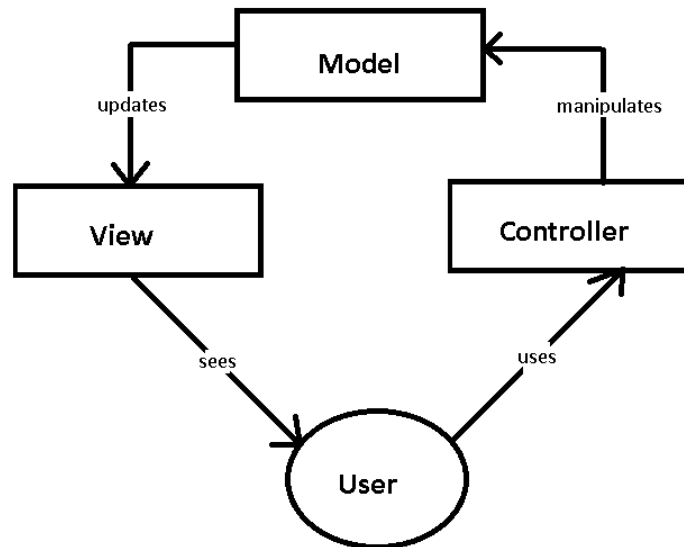


FIGURE 41. A graphic example of the MVC pattern (Bakshi 2019).

The MVC stands for Model, View, and Controller. The idea of the MVC design pattern is to separate a web server into three segments. The Controller segment is the place that stores all the business logic, interacts with user requests, and manipulates the data by using the Model segment. The Model segment is the database interface, this segment contains all the interactions between the back-end and the database API, for example, the format of each object in the database schemas or the relationships in the database. In other words, the Model segment works with the Controller to provide or manipulate the data that is requested from the user. The View segment is where all the information is compiled, rendered, and sent back to users as a complete website. (Bakshi 2019.)

5 FINAL PROJECT

During the academic period at Centria University of Applied Sciences, the author has been studying and working on many projects which use web-techs as building tools. Moreover, the research of the demand on job markets and the statistic of the stack overflow website has shown the remarkable rise of React over a few years, which could lead to many opportunities for web developers that have experience with React library. The Node.js framework is also the standard tool that the author has used in developing a webserver because it allows using JavaScript on both sides of a website and reduces a developer's learning time. Since Node.js became popular nowadays, many tools and frameworks have been built specifically for Node.js developers. In this final project, the ExpressJS plays an essential role in API design and database procession. A website cannot live without a database, and there are many options for a database. But they usually are defined by two types, relational databases and non-relational databases. The non-relational database has been introduced recently and has shown its benefit in many social media websites by managing their immersive users' data, for instance, Facebook, Amazon, and Twitter. (Forbes 2014.)

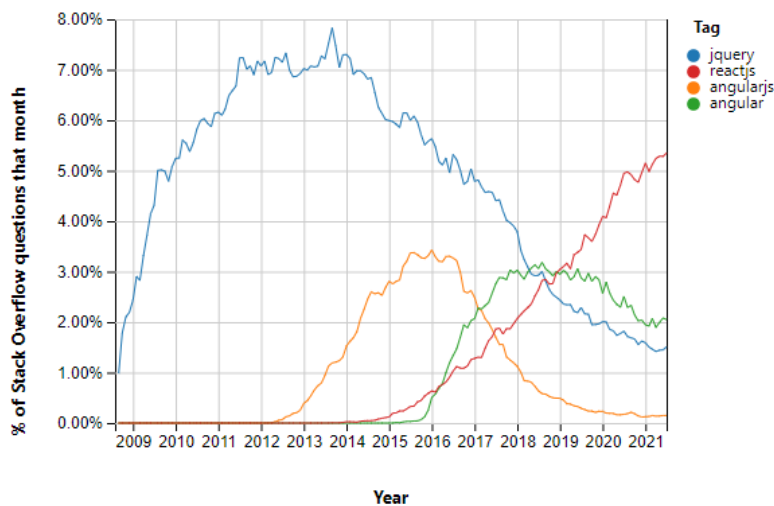


FIGURE 42. The JavaScript frameworks' popularity on Stack Overflow Trends (Robinson 2017).

5.1 The basic ideas and plans

The final project was a single-page application called Travelogue, a social media website that allows users to create an account, share places, images, and locations. The React library combined with the Google Map API for the front-end is suitable for this project. The ExpressJS was the chosen framework to design API for the back-end of this project. And for the database, the project used the MongoDB atlas to store users' information and locations. The main purpose of this final project was to experiment with the new MERN method of building an application. And the Travelogue website was the final result of this experiment. According to the plan, the Travelogue website had authentication to allow users to create an account. The website also had a feature that allowed users to create a new post, edit their posts, and visit other posts of other users. (Iriarte 2018, 72 -73.)

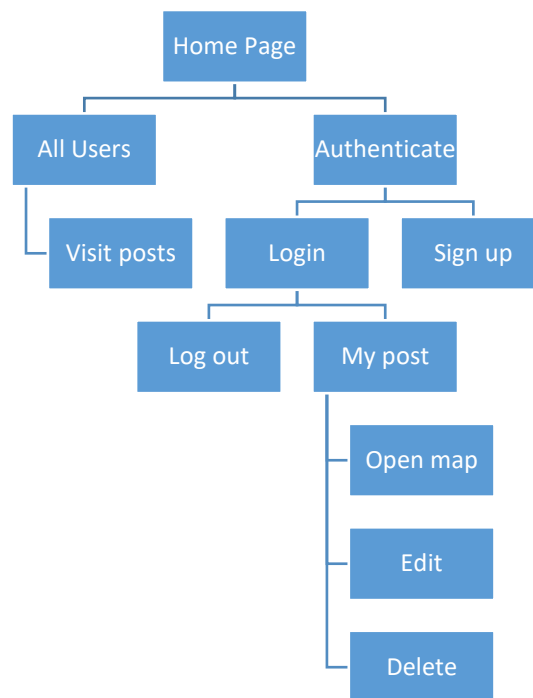


FIGURE 43. The dependencies chart of the website

Table 1 shows all the paths that lead to different pages of the website. Each page had many React components that were combined and loaded together depends on which path was chosen. The structure of the website can be seen in the dependencies chart above. The chart shows the hierarchy of every page of the Travelogue and its functionalities. (Wysocki 2021.)

TABLE 1. The URL paths of the Travelogue project

Path	Functionality	Restriction
/	Showing list of users	No restriction
/:uid/places	Showing list of places for a selected user	No restriction
/authenticate	Signing up and logging in	Only unauthenticated
/places/new	Adding a new place	Only authenticated
/places/:pid	Updating a place	Only authenticated

5.2 Sketching the UI

Figure 44 demonstrates the sketching phase was done by using the wireframe method of the draw.io website. The purpose of the sketching design was to clarify the project by going through all the interactions and layouts. The wireframe pointed out the primary focus of the project. The wireframe design delivered a clear picture of the elements and the content hierarchy in the sketching phase. When a team handles the project, the wireframe benefits more since it is simple, focuses on the main ideas, and helps the communication to express the ideas becomes more straightforward. (The Seque Creative 2016.)

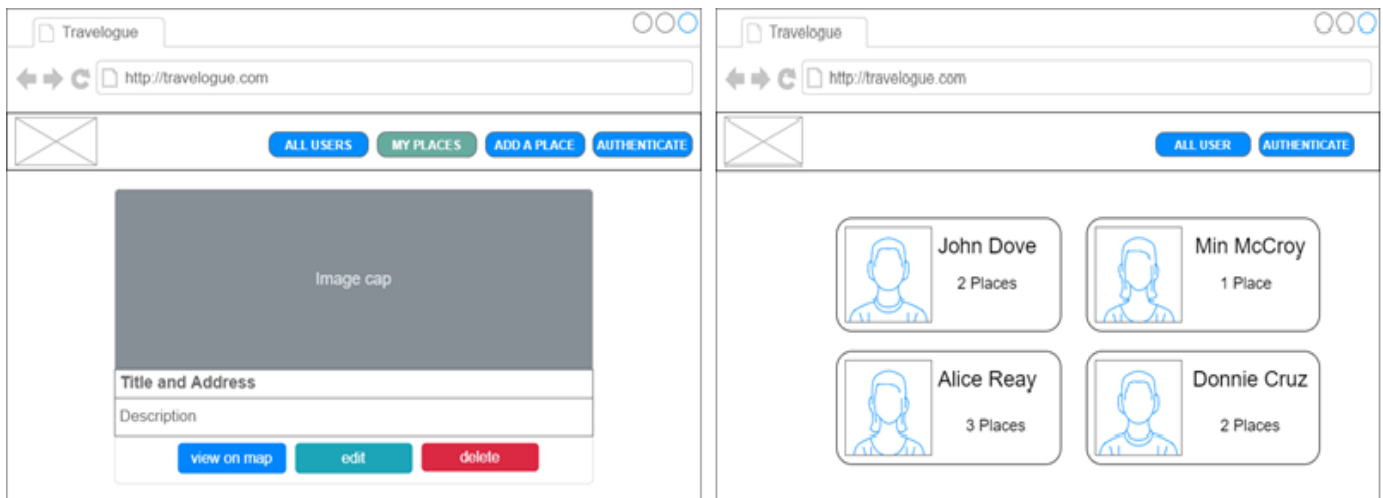


FIGURE 44. The wireframe design of the project

5.3 The front-end with React library

Because the application allowed users to create their accounts, some pages had login restrictions. Table 1 in the planning section provided a basics picture of the restrictions. Some routes could be displayed either with or without the authentication of the user. Figure 45 shows that the default route was the '/', which displays all users in the system. The ':userId/places' route displayed all posts of a chosen user. The '/auth' route was the place that users could create a new account or log in to their accounts, and these

routes could be displayed without a login token. However, the '/places/new' and '/places/:placeId' routes were expected to receive a user access token to access since they had the feature that manipulates the users' data. In the '/places/new' route, there was a form that allows a user to create a new post. The '/places/:placeId' was the route that allows a user to edit or delete a post. (Auth0 2021.)

```

36   if (token) {
37     routes = (
38       <Switch>
39         <Route path="/" exact>
40           <Users />
41         </Route>
42         <Route path="/:userId/places" exact>
43           <UserPlaces />
44         </Route>
45         <Route path="/places/new" exact>
46           <NewPlaces />
47         </Route>
48         <Route path="/places/:placeId/">
49           <UpdatePlace />
50         </Route>
51         <Redirect to="/" />
52       </Switch>);
53   } else {
54     routes = (
55       <Switch>
56         <Route path="/" exact>
57           <Users />
58         </Route>
59         <Route path="/:userId/places" exact>
60           <UserPlaces />
61         </Route>
62         <Route path="/auth">
63           <Auth />
64         </Route>
65         <Redirect to="/auth" />
66       </Switch>
67     );
68   }

```

FIGURE 45. The pages and routes setup of the application

5.3.1 The Home page

The home page's first task was to fetch all users' data from a back-end. When the data was fetched successfully, the home page rendered these users' data into a list of users. However, Figure 46 shows when the data fetching process was failed, an error modal was played to deliver a message. During this

section, because the connection between the front-end and back-end was not been established, the home page showed an error message to inform the failure in fetching users' data. (Rogozhny 2020.)

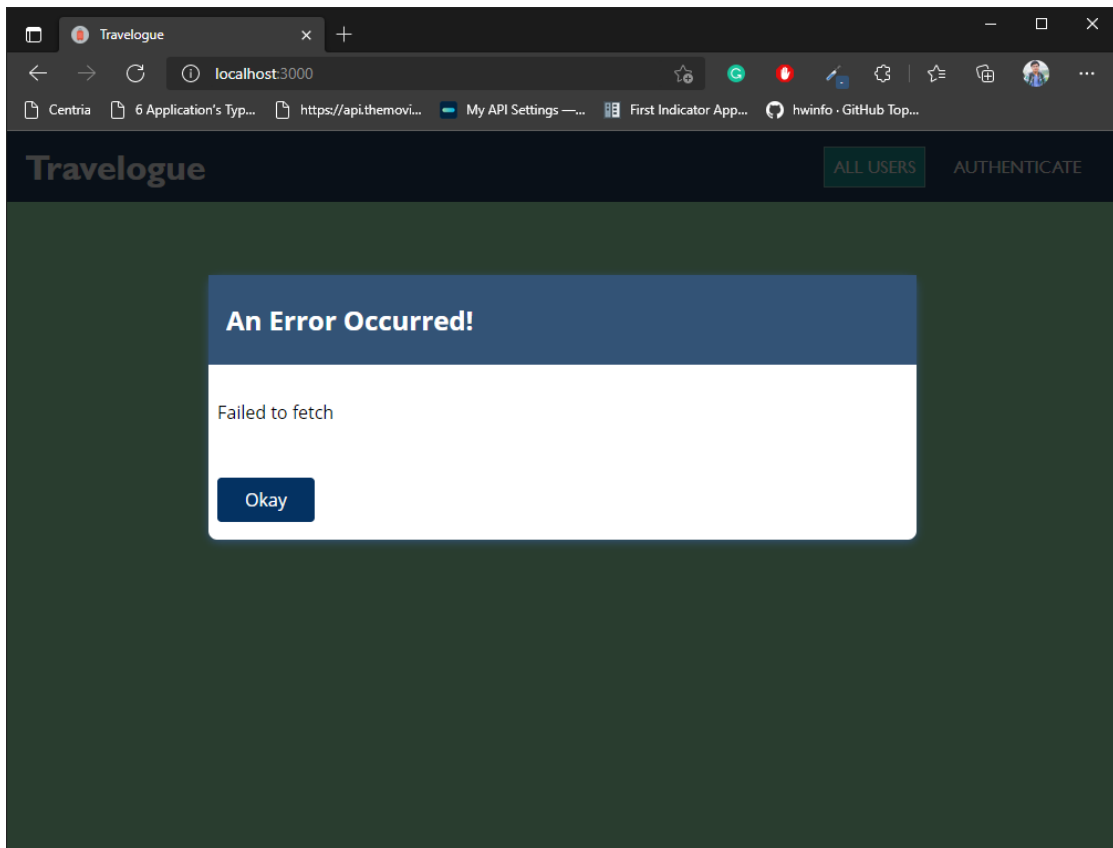


FIGURE 46. Home page without a back-end connection on browser

Figure 47 shows the Users component, which was a customized functional component. The Users component used the `useState()` hook, which is a default hook given by React library. However, the `useHttpClient()` was a custom hook, which was defined to fit the purposes of this application. The `useHttpClient()` hook had three main functions. The first function was the `isLoading`, which checked whether data was being loading or not. The second function was the `error`, which threw a message when there was an error occurred. The third function was the `sendRequest`, which sent an HTTP request to a server. And the last function was the `clearError`, which cleared the error message. (Facebook 2021.)

```

1  import React, { Fragment, useEffect, useState } from 'react';
2
3  import UserList from '../components/UserList';
4  import ErrorModal from '../../shared/components/UIElements/ErrorModal';
5  import LoadingSpinner from '../../shared/components/UIElements/LoadingSpinner';
6  import { useHttpClient } from '../../shared/hooks/http-hook';
7
8  const Users = () => {
9      const { isLoading, error, sendRequest, clearError } = useHttpClient();
10     const [loadedUsers, setLoadedUsers] = useState();
11
12     useEffect(() => {
13         const fetchUsers = async () => {
14             try {
15                 const responseData = await sendRequest('http://localhost:5000/api/users');
16                 setLoadedUsers(responseData.users);
17             } catch (err) {
18                 console.log(err.message);
19             }
20         }
21     }, [sendRequest]);
22     fetchUsers();
23
24     return <Fragment>
25         <ErrorModal error={error} onClear={clearError} />
26         {isLoading && (
27             <div className='center'>
28                 <LoadingSpinner />
29             </div>
30         )}
31         {!isLoading && loadedUsers && <UserList items={loadedUsers} />}
32     </Fragment>
33 };
34
35 export default Users;

```

FIGURE 47. The Users component of the application

5.3.2 The Authentication page

The authentication had two different input forms. Figure 48 illustrates the login form, which was always first on the authentication page, and the second form was the signup form. The form glowed red at the input and displayed a message to remind users when a form input was missing or had invalid data. The login button was also disabled until the form was filled in correctly. This feature was also included in the signup form. The main idea of this feature was to restrict users from entering invalid data into the form. However, the data validation on the client-side only is not secure enough since this feature can be broken through easily by disabling JavaScript from the browser. (Truth 2011.)

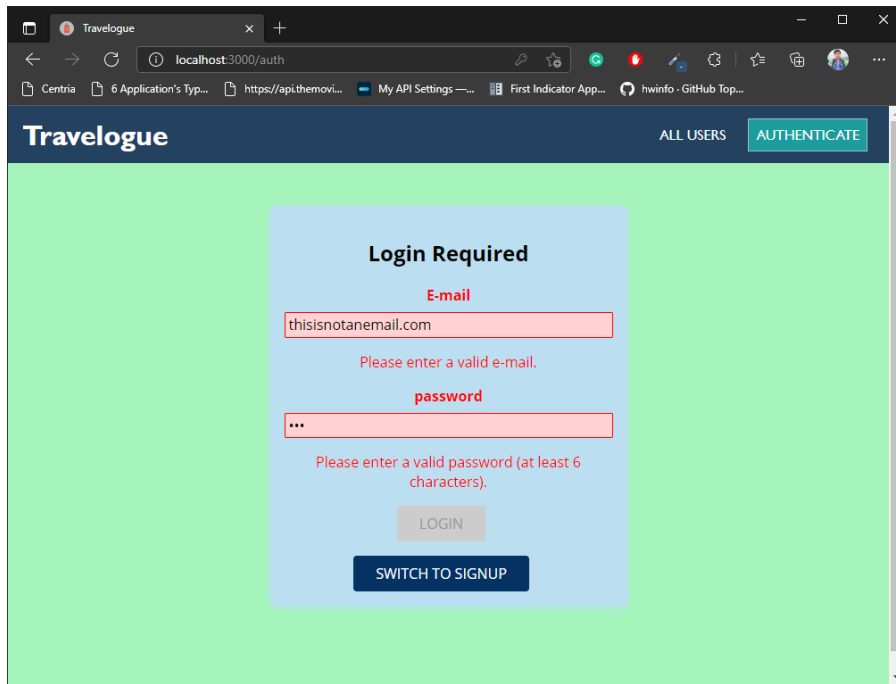


FIGURE 48. The login form inputs are invalid

Figure 49 shows the data validator custom hook, which was used to validate form inputs. The form validator had many options, which were pre-calculated from the possibility of the users' input and situations. For example, the validate function received two arguments, the value and validators. The validators argument was expected as an array, which consisted of the types of the form inputs. The login form had two inputs, the email type and password type. The email type was checked by the last if condition, and the validator returned true when the if condition was satisfied. The password input could be checked by combining the type of input and the length of the input, which was provided by the value argument. (MDN 2021.)

```

23 export const validate = (value, validators) => {
24   let isValid = true;
25   for (const validator of validators) {
26     if (validator.type === VALIDATOR_TYPE_REQUIRE) {
27       isValid = isValid && value.trim().length > 0;
28     }
29     if (validator.type === VALIDATOR_TYPE_MINLENGTH) {
30       isValid = isValid && value.trim().length >= validator.val;
31     }
32     if (validator.type === VALIDATOR_TYPE_MAXLENGTH) {
33       isValid = isValid && value.trim().length <= validator.val;
34     }
35     if (validator.type === VALIDATOR_TYPE_MIN) {
36       isValid = isValid && +value >= validator.val;
37     }
38     if (validator.type === VALIDATOR_TYPE_MAX) {
39       isValid = isValid && +value <= validator.val;
40     }
41     if (validator.type === VALIDATOR_TYPE_EMAIL) {
42       isValid = isValid && /^S+@S+\.S+$/ .test(value);
43     }
44   }
45   return isValid;
46 };

```

FIGURE 49. A part of the validator hook

When all the login or signup form inputs were correct, the data were sent to the back-end to authorize the access. The client expected to receive the user's ID and an access token. And the token was used as proof of authorized access. The token was also used to maintain the login state, which allowed a user to access their account without performing the login process again when they closed the page. However, the token had a limited lifetime because of security reasons. (Auth0 2021.)

```

56  const authSubmitHandler = async event => {
57  |   event.preventDefault();
58  |   if (isLoginMode) {
59  |     try {
60  |       const responseData = await sendRequest('http://localhost:5000/api/users/login', 'POST',
61  |       {
62  |         'Content-Type': 'application/json'
63  |       },
64  |       JSON.stringify({
65  |         email: formState.inputs.email.value,
66  |         password: formState.inputs.password.value
67  |       })
68  |     );
69  |     auth.login(responseData.userId, responseData.token);
70  |   } catch (err) {
71  |     console.log(err);
72  |   }
73  | } else {
74  |   try {
75  |     const formData = new FormData();
76  |     formData.append('name', formState.inputs.name.value);
77  |     formData.append('email', formState.inputs.email.value);
78  |     formData.append('password', formState.inputs.password.value);
79  |     formData.append('image', formState.inputs.image.value)
80  |     const responseData = await sendRequest('http://localhost:5000/api/users/signup', 'POST', {}, formData
81  |     );
82  |     auth.login(responseData.userId, responseData.token);
83  |   } catch (err) {
84  |     console.log(err);
85  |   }
86  | }
87  }

```

FIGURE 50. The function handles sending login and signup requests to the server

5.4 The back-end with ExpressJS and Mongoose

The application needed a server to authorize access and process data from the database. The server handled the logic of checking the login and signup requests, for example, the login and signup input data. The server is the safest place to perform any authenticating or authorizing process since it is close and difficult to break in. (Boston University 2020.)

5.4.1 Planning and building a REST API with ExpressJS

Every API has endpoints, which help to depict the exact location of the resource to access. An API endpoint is an entry in a communication between two or more applications. An API endpoint can be operated through HTTP requests and responses. As in the design in Table 2, this API had two main routes and seven endpoints. Some endpoints needed to meet certain conditions in order to respond. These conditions were set up for security reasons since keeping the database safe from unauthorized access is important. (Allamaraju 2010, 78 -79.)

TABLE 2. The endpoints of the API

/api/users/...	/api/places/...	
GET .../ Fetch a list of all users.	GET .../user/:uid Fetch a list of all places from a given user ID.	PATCH .../:pid Update a place by a given place ID.
POST .../signup Create a new user + log a user in.	GET .../:pid Fetch a specific place from a given place ID.	DELETE .../:pid Delete a place by a given place ID.
POST .../login Log a user in.	POST .../ Create a new place.	

In section 4.3.3, the MVC pattern was mentioned as a logical method to organize files and modules of a project based on three main segments: model, view, and controller. Figure 51 shows the MVC design pattern implemented in building this API. The two main routes can be seen in the routes folder, where they were exported to be used in the app.js file. The app.js played the role of the root file, which was always compiled first whenever the server was started. The endpoints can be found in each route file in the routes folder. (RapidAPI 2021.)

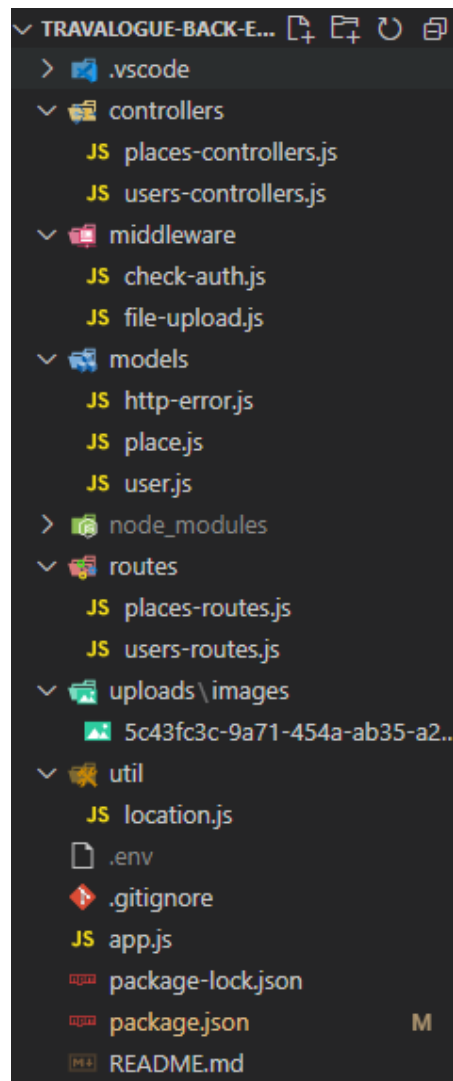


FIGURE 51. The API's file structure according to the MVC pattern

In Figure 51, the logic of each endpoint was separated and stored in the controllers folder. The places-controllers.js held all the endpoints' logic of the places-routes.js file. And the same for the users-controllers.js file, where all the endpoints' logic of the users-routes.js file was held. Figure 52 shows all the logic of the endpoints was imported by the "require " keyword. Besides the endpoints' logic, other modules that handle different tasks were also imported. For instance, line 5 of the places-routes.js file was the middleware that handled the file uploading task, and also line 6 was the middleware that handled validating the access token from the front-end. (Kalubowila 2020.)

The image shows two side-by-side screenshots of code editors. The left editor shows the code for 'users-routes.js' and the right editor shows the code for 'places-routes.js'. Both editors have a dark background with light-colored text.

```

JS users-routes.js M X
routes > JS users-routes.js > ...
1 const express = require('express');
2 const { check } = require('express-validator');
3 const usersController = require('../controllers/users-controllers');
4 const fileUpload = require('../middleware/file-upload');
5
6 const router = express.Router(); //gives an object to register middleware
7 router.get('/', usersController.getUsers);
8
9 router.post(
10   '/signup',
11   fileUpload.single('image'),
12   [
13     check('name').not().isEmpty(),
14     check('email').normalizeEmail().isEmail(),
15     check('password').isLength({ min: 6 })
16   ],
17   usersController.signup);
18
19 router.post('/login', usersController.login);
20
21 module.exports = router;

```

```

JS places-routes.js M X
routes > JS places-routes.js > ...
1 const express = require('express');
2 const { check } = require('express-validator');
3
4 const placesController = require('../controllers/places-controllers');
5 const fileUpload = require('../middleware/file-upload');
6 const checkAuth = require('../middleware/check-auth');
7
8 const router = express.Router(); //gives an object to register middleware
9 router.get('/:pid', placesController.getPlaceById);
10 router.get('/user/:uid', placesController.getPlacesByUserId);
11 router.use(checkAuth);
12
13 router.post(
14   '/',
15   fileUpload.single('image'),
16   [
17     check('title').not().isEmpty(),
18     check('description').isLength({ min: 5 }),
19     check('address').not().isEmpty()
20   ],
21   placesController.createPlace);
22
23 router.patch(
24  ('/:pid',
25   [
26     check('title').not().isEmpty(),
27     check('description').isLength({ min: 5 })
28   ],
29   placesController.updatePlace);
30
31 router.delete('/:pid', placesController.deletePlace);
32
33 module.exports = router;

```

FIGURE 52. The user-route.js and places-routes files

5.4.2 Establishing connection with MongoDB Atlas and Mongoose

MongoDB is a non-relational database or NoSQL. In SQL, the data is stored as records in tables. However, data is store as documents in collections. The first step of establishing the connection with MongoDB Atlas was to access their webpage at <https://www.mongodb.com/>. After signing in to MongoDB Atlas, the next step was to create a cluster. A cluster is a place that stores the databases. A MongoDB cluster can be preferred as a NoSQL Database-as-a-Service, which is offered in the public cloud. Clusters are basically a group of MongoDB servers that store documents of databases. MongoDB has a method for stabilizing its scalability, which is MongoDB sharding. MongoDB stores databases in many distributed shards to create horizontal scalability. (MongoDB 2021.)

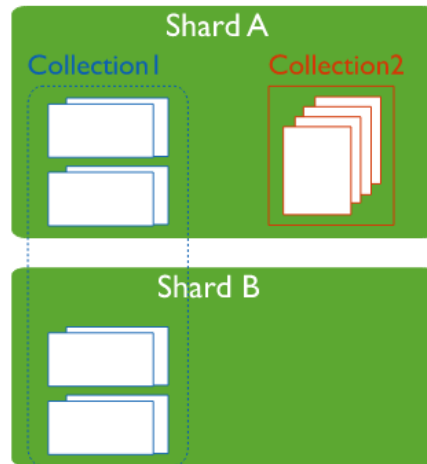


FIGURE 53. A MongoDB collection is stored in two different shards (MongoDB 2021).

Because the databases are stored in documents and do not have any strong relationship, they can be cut into different pieces and stored in different servers. This sharding method helps the server reduce the significant amount of work that needs to perform. For example, Figure 53 shows a part of the Collection 1 was kept in server A, and another part was kept in server B to optimize the server's capacity. In this section, Mongoose was used as the main tool to establish a connection and manipulate the database from the MongoDB Atlas server. Mongoose is an Object Data Modeling library, which was built specifically for MongoDB and Node.js. Mongoose can be installed by NPM and import directly to any Node.js project as a Node module. (MongoDB 2021.)

```

52  mongoose
53    .connect(url, {useNewUrlParser: true, useUnifiedTopology: true, useCreateIndex: true})
54    .then(() => {
55      app.listen(process.env.PORT || 5000);
56    })
57    .catch(err => {
58      console.log(err);
59    });

```

FIGURE 54. Establishing a database connection with Mongoose

The connection started on line 53 of figure 54; the connection required an URL string that describes the connecting person, the password, the cluster's name, and the collection's name. The URL string can be found after creating the cluster in the MongoDB Atlas server through their website. Mongoose allows developers to define the document properties by the built-in Schema class. According to the MVC pattern, this modeling task with the Schema class is classified to the Model segment. Figure 51 shows a models folder contained place.js and user.js files, which were two schemas of the places and users.

Figure 55 illustrates the users and places schemas, which defined how the object data structure was stored in the database and object data properties were required to exist. (Mongoose 2021.)

```

JS users.js X
models > JS users.js > @userSchema > places
1  const mongoose = require('mongoose');
2  const uniqueValidator = require('mongoose-unique-validator');
3
4  const Schema = mongoose.Schema;
5
6  const userSchema = new Schema({
7    name: { type: String, required: true },
8    email: { type: String, required: true, unique: true },
9    password: { type: String, required: true, minlength: 6 },
10   image: { type: String, required: true },
11   places: [{ type: mongoose.Types.ObjectId, required: true, ref: 'Place' }]
12 });
13
14 userSchema.plugin(uniqueValidator);
15
16 module.exports = mongoose.model('User', userSchema);

```

```

JS places.js X
models > JS places.js > ...
1  const mongoose = require('mongoose');
2
3  const Schema = mongoose.Schema;
4
5  const placeSchema = new Schema({
6    title: { type: String, required: true },
7    description: { type: String, required: true },
8    image: { type: String, required: true },
9    address: { type: String, required: true },
10   location: {
11     lat: { type: Number, required: true },
12     lng: { type: Number, required: true }
13   },
14   creator: { type: mongoose.Types.ObjectId, required: true, ref: 'User' }
15 })
16
17 module.exports = mongoose.model('Place', placeSchema);

```

FIGURE 55. The database schemas

5.4.3 Adding password hashing function and web token

Security is an essential factor of every server, which can cause severe issues if external threats breach it. This application stored user information in the database and the most critical data were the user passwords which should never be stored as plain text. Some systems implement data encryption to prevent data stolen by hackers. The encryption algorithm is often used to protect passwords, but this mechanism is not fully secured since data can be decrypted with the decryption key. Hashing is a better method for data protection since the hashing algorithm does not have a decryption key to decrypt data. A hashed password is undecryptable data or might take a significantly long period and a powerful machine to encrypt. The hashing algorithm also has a feature that appends a set of characters to the hashed data, whose purpose is to make it nearly impossible to encrypt. Figure 56 demonstrates the bcrypt in this application, an available Node.js package that consists of a hashing algorithm and or that function to customize the algorithm. (Malviya 2015.)

```

47     let hashedPassword;
48     try {
49         hashedPassword = await bcrypt.hash(password, 12);
50     } catch (err) {
51         const error = new HttpError('Could not create user, please try again.', 500);
52         return next(error);
53     }
54
55     const createdUser = new User({
56         name,
57         email,
58         image: req.file.path,
59         password: hashedPassword,
60         places: []
61     });

```

FIGURE 56. Hashing and salting user passwords with bcrypt

This application required users to log in to their accounts to use the basic features, and the login status could be destroyed whenever the tab or the browser was closed. However, the login status could be maintained by creating an access token in the server and sending it to the client every time a user logs into the system successfully. In the client, the token was stored inside either the local storage or cookie storage of the browser so that when the browser accessed the website, it could use the token as proof of authorized access. Node.js has a package named `jwt` or JSON Web Tokens, which allows developers to create an access token to maintain the login status of a website. However, this method can be risky if the token is stolen since it can be used to log in to an account without the need for authorization. The token also has a feature to set its lifetime to reduce the risk of unauthorized access. This might not prevent the hacker from accessing the user accounts through the stolen token, but it might help mitigate further damage since the token is valid for a limited period. (Auth0 2021.)

```

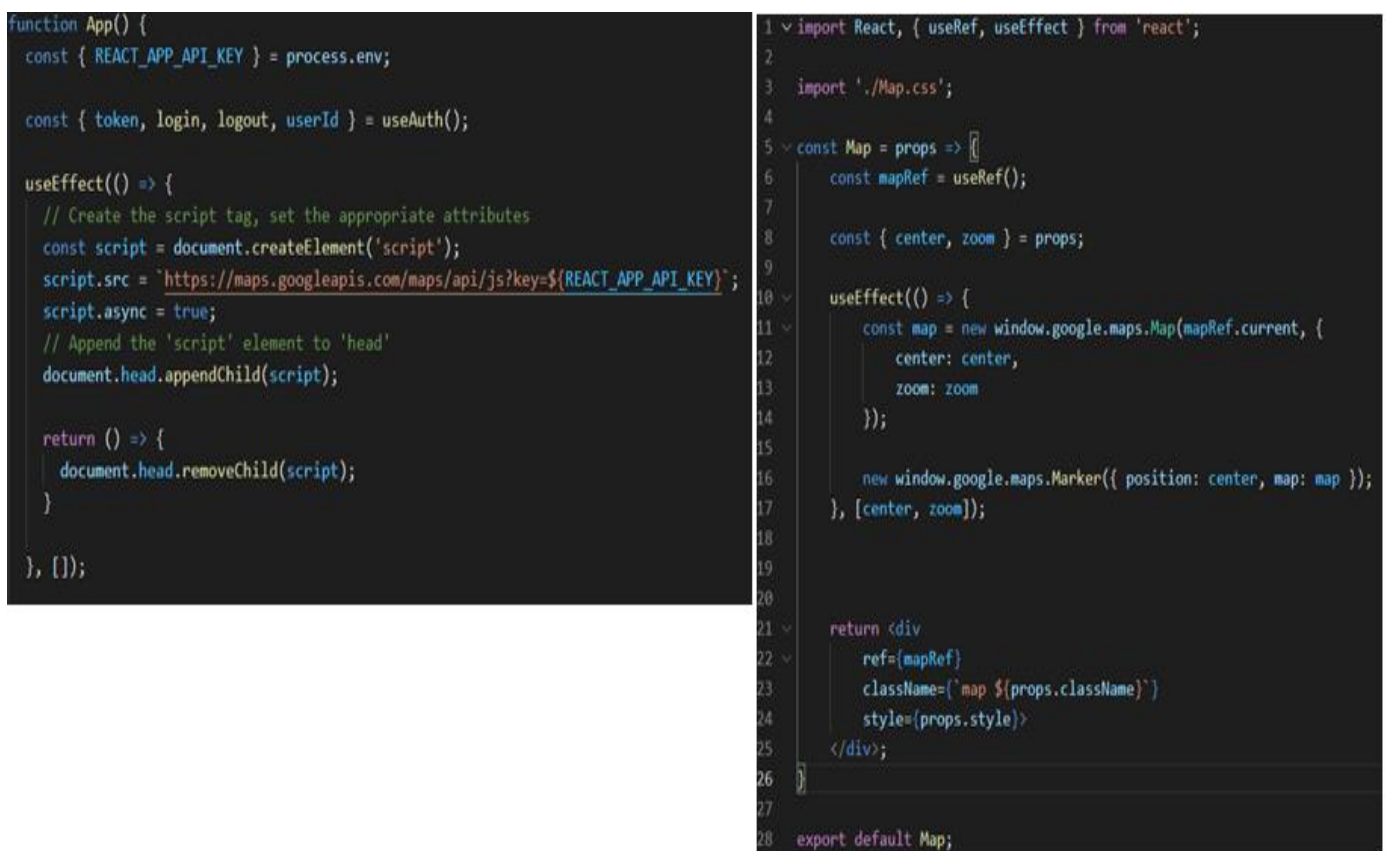
72     let token;
73     try {
74         token = jwt.sign({ userId: createdUser.id, email: createdUser.email }, process.env.JWT_KEY, { expiresIn: '1h' });
75     } catch (err) {
76         const error = new HttpError('Failed to create token.', 500);
77         return next(error);
78     }
79
80
81     res.status(201).json({ userId: createdUser.id, email: createdUser.email, token: token });
82 }

```

FIGURE 57. Creating an access token with JWT

5.5 Adding Google Map API and Google Geocoding API

The Google Map is a well-known tool that has been using on many devices. Since the Google Map becomes useful and popular, Google creates an API that allows other developers to integrate their Google Map into their websites or application. The Google Map API can be found on their website at <https://developers.google.com/maps>, where Google provides other APIs related to their Google Map. As the plan of this application, the Google Map was rendered by the front-end with React library. However, to display the location on the Map, the Google Map API requires the coordinate, which combines the longitude and the latitude. (Google Maps Platform 2021.)



```
function App() {
  const { REACT_APP_API_KEY } = process.env;

  const { token, login, logout, userId } = useAuth();

  useEffect(() => {
    // Create the script tag, set the appropriate attributes
    const script = document.createElement('script');
    script.src = `https://maps.googleapis.com/maps/api/js?key=${REACT_APP_API_KEY}`;
    script.async = true;
    // Append the 'script' element to 'head'
    document.head.appendChild(script);

    return () => {
      document.head.removeChild(script);
    }
  }, []);
}

import React, { useRef, useEffect } from 'react';
import './Map.css';

const Map = props => {
  const mapRef = useRef();

  const { center, zoom } = props;

  useEffect(() => {
    const map = new window.google.maps.Map(mapRef.current, {
      center: center,
      zoom: zoom
    });

    new window.google.maps.Marker({ position: center, map: map });
  }, [center, zoom]);

  return <div
    ref={mapRef}
    className={`map ${props.className}`}
    style={props.style}>
    </div>;
}

export default Map;
```

FIGURE 58. Adding and rendering Google Map in React

Google provides another API, allowing developers to retrieve a specific location's longitude and latitude through the location's name, the Google Geocoding API. In this application, the Google Geocoding API was implemented in the back-end since users sent the location names through a form in the front-end. The Google Geocoding API delivered the coordinates according to the location's name and sent the coordinates back to the front-end to create a marker on the Google Map. (Google Maps Platform 2021.)

```

1  const axios = require('axios');
2  const HttpError = require('../models/http-error');
3  require('dotenv').config();
4  const API_KEY = process.env.API_KEY;
5
6  async function getCoordsForAddress(address) {
7    const response = await axios.get(`https://maps.googleapis.com/maps/api/geocode/json?address=${encodeURIComponent(address)}&key=${API_KEY}`);
8
9    const data = response.data;
10
11    if(!data || data.status === 'ZERO_RESULTS') {
12      const error = new HttpError('Could not find location for the specified address.', 422);
13      throw error;
14    }
15
16    const coordinates = data.results[0].geometry.location;
17    return coordinates;
18  }
19
20  module.exports = getCoordsForAddress;

```

FIGURE 59. Implementing Google Geocoding API with ExpressJS

5.6 Testing the API with Postman

There are various API testing tools, such as SoapUI, Apigee, API Fortress, and Postman. They all have the same mechanism, providing developers with a testing environment for the API developing process. Postman is simple to use and provides all the necessary features to keep track of the API without interfering with the front-end code. Figure 60 shows the result after an account was created successfully with Postman, the application automatically logged the user into the system and responded to the client-side with an access token, user ID, and an e-mail address to maintain the login status for the next access. The API also responded with an access token, user ID, and an e-mail address for a successful login. (Postman 2021.)

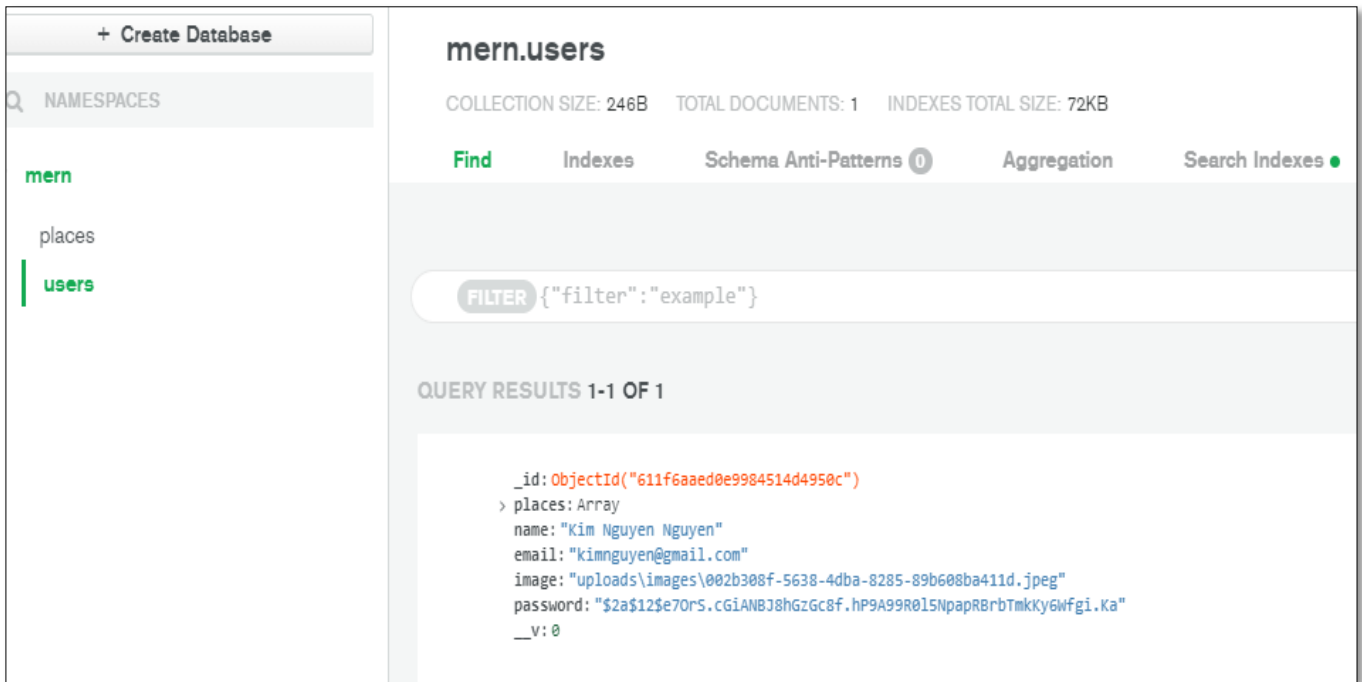


FIGURE 61. A user account is stored in MongoDB Atlas Database

5.7 Final Result

After the testing phase, when the API gave the correct result, it was ready to be implemented to fetch and send data from the front-end. In the front-end, the API was already integrated with the Fetch API. The back-end and front-end can be activated by the command "npm start", and the project was live on port 3000 of localhost. The Authenticate button on the navigation bar switched the front-end to the login form of the '/auth' route, which was set up at the beginning of the project. Figure 62 shows when a user successfully logged in to an account, the news feed showed the data of the places that the user added to the account by clicking the My Places button. The application also allowed users to delete or edit the content of any post just by clicking on the edit and delete button. However, since these features can manipulate the database, they required authorized access to proceed. These buttons appeared when users logged in to their account and users can only edit or delete their posts from their accounts. (Facebook 2021.)

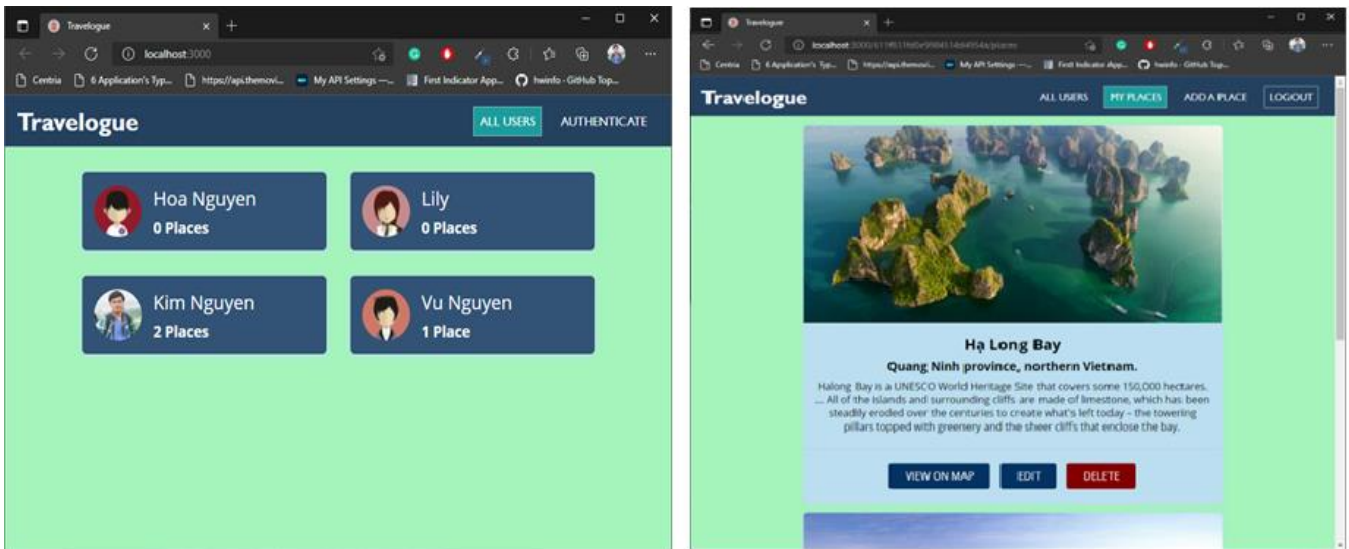


FIGURE 62. The home page and newsfeed

Figure 63 shows a form and a map where a user can add a new post to their account or view a specific location of a place on the map. By clicking the Add A Place button on the navigation bar, the application switched the page to the '/places/new' route. The '/places/new' route had a form where a user needs to fill in a title, a description, an address, and an image for their post. Every post on the website was integrated with Google Map API and Google Geocoding, so their locations can be seen on the map by clicking on the View On Map button below every post. (Google Maps Platform 2021.) Users can log out to their account by click on the Logout button on the top left corner of the website. When the account was logged out, the website did not allow users to edit or delete their posts, but they could still visit other posts of other users and view the location of these places.

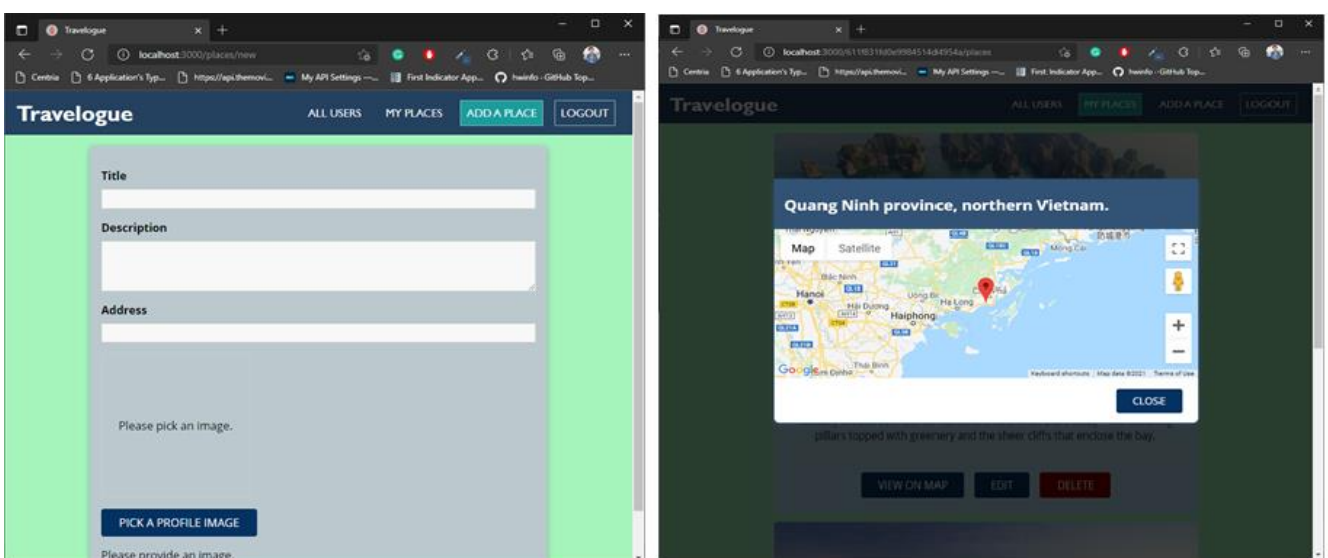


FIGURE 63. The form and the map

6 CONCLUSION

Throughout the thesis, the primary goal was to study from general surface to deeper levels of core architectures of the React library, Node.js, and MongoDB. The study was conducted through basic experiments and various documents of publishers. The final product of the thesis was a media website, which was designed and developed as a single-page application by combining the mentioned technologies. Overall, the React library proved its advantages from the powerful syntax, high maintainability, and optimization of the virtual DOM. The Node.js and ExpressJS framework proved to be convenient tools by allowing developers to use JavaScript everywhere to create a fully working website with all necessary features provided as importable modules and packages. Mongoose and MongoDB helped manage the database efficiently with high scalability, a suitable choice for storing blogs or any media website data.

The website had a standard security system to protect sensitive users' data from external threats using the bcrypt hashing algorithm and a customized authentication back-end. The website also implemented tokens and cookies from JWT as a popular web technology. The website also introduced the popular Google API, with two main examples of Google Map API and Google Geocoding API.

The website was created during the Covid-19 pandemic when many countries performed lockdown plans to prevent the outbreak of the virus. The event damaged the tourism and travel services deeply, and for this reason, the author wanted to create a website as an accessible platform where it could help various businesses, who depend on tourists to recover. The website had a core feature where a user can create an interactable post. The post can be a business promotion or an artistic picture of a location where people can find it on the map. The website is incomplete and needs further improvement with extra features as for many other social media websites nowadays, such as running videos, creating comments, sending private messages. However, the website had demonstrated as an informative example of full-stack web development with MERN stack. The experiment of the mentioned technologies and the development and testing processes of the final project was thoroughly documented in this thesis to explain the benefits and core features of the website clearly.

REFERENCES

Bank & Porcello, 2017. *Learning React - Functional Web Development With React And Redux*. First edition. Sebastopol, CA: O'Reilly Media.

Allamaraju, S., 2010. *RESTful Web Service Cookbook*. 1st edition. Sebastopol, CA: O'Reilly.

Alpert, S., 2015. *React v0.14*.

Available at: <https://reactjs.org/blog/2015/10/07/react-v0.14.html>.

Accessed 1st July 2021.

Auth0, 2021. *Auth0 Official Documents*.

Available at: <https://auth0.com/docs/security/tokens/access-tokens>.

Accessed 27th September 2021.

Babel, 2021. *Babel Official Documents*.

Available at: <https://babeljs.io/docs/en/>.

Accessed 19th July 2021.

Bakshi, A., 2019. *How To Setup MVC Design Pattern In Express*.

Available at: <https://www.c-sharpcorner.com/blogs/how-to-setup-mvc-design-pattern-in-express>.

Accessed 3rd August 2021.

Bojinov, V., 2018. *RESTful Web API Design with Node.js 10*. 3rd edition. Birmingham, UK: Packt.

Boston University, 2020. *Understanding Authentication, Authorization, and Encryption*.

Available at: <https://www.bu.edu/tech/about/security-resources/bestpractice/auth/>.

Accessed 27th September 2021.

Carnes, B., 2021. *Create a MERN Stack App With A Serverless Backend*.

Available at: <https://www.freecodecamp.org/news/create-a-mern-stack-app-with-a-serverless-backend/>. Accessed 26th June 2021.

Champion, M., 1997. *Document Object Model (Core) Level 1*.

Available at: <https://www.w3.org/TR/REC-DOM-Level-1/level-one-core.html>.

Accessed 23rd July 2021.

Chavan, Y., 2021. *JSX In React Introduction*.

Available at: <https://www.freecodecamp.org/news/jsx-in-react-introduction/>.

Accessed 12th July 2021.

Creative, T. S., 2016. *The Importance Of Wireframing For A Responsive Website*.

Available at: <https://www.seguetech.com/the-importance-of-wireframing-for-a-responsive-website/>.

Accessed 23rd August 2021.

Dawson, C., 2014. *JavaScript History And How It Led To Reactjs*.

Available at: <https://thenewstack.io/javascripts-history-and-how-it-led-to-reactjs/>.

Accessed 1st July 2021.

Eisenman, B., 2021. *Chapter 2. Working With React Native*.

Available at: <https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch02.html>.

Accessed 13th July 2021.

Facebook, 2021. *React Official Documents*.

Available at: <https://reactjs.org/docs/>.

Accessed 16th July 2021.

Forbes, 2014. *NoSQL Databases: Oracle's Big Opportunity*.

Available at: <https://www.forbes.com/sites/greatspeculations/2014/04/04/nosql-databases-oracles-big-opportunity/?sh=2857cb456aa6>.

Accessed 27th September 2021.

Google Maps Platform, 2021. *Google Maps Platform Official Documents*.

Available at: <https://developers.google.com/maps/documentation>.

Accessed 27th September 2021.

IBM, 2020. *The Component of A URL*.

Available at: <https://www.ibm.com/docs/en/cics-ts/5.2?topic=concepts-components-url>.

Accessed 24th July 2021.

Ijas, M., 2020. *Host websites using Node.js*.

Available at: <https://medium.com/@mohammedijas/host-websites-using-node-js-5b3a0832c94c>.

Accessed 27th September 2021.

Iriarte, K. E. W., 2018. *MERN Quick Start Guild*. First edition. Birmingham, UK: Packt Publishing Ltd.

Kalubowila, D., 2020. *JWT Bearer Token Authentication for ExpressJS via Middlewares and Request-Response pipeline*.

Available at: <https://medium.com/ms-club-of-sliit/jwt-bearer-token-authentication-for-express-js-5e95bf4dead0>.

Accessed 27th September 2021.

Kosaka, M., 2018. *Cross-Origin Resource Sharing (CORS)*.

Available at: <https://web.dev/cross-origin-resource-sharing/>.

Accessed 5th July 2020.

Lane, K., 2019. *Intro to APIs: History of APIs*.

Available at: <https://blog.postman.com/intro-to-apis-history-of-apis/>.

Accessed 22nd July 2021.

Lewis, P., 2020. *Rendering Performance*.

Available at: <https://developers.google.com/web/fundamentals/performance/rendering>.

Accessed 12th July 2021.

Liew, Z., 2019. *How To Deal with Nested Callbacks and Avoid "Callback Hell"*.

Available at: <https://www.freecodecamp.org/news/how-to-deal-with-nested-callbacks-and-avoid-callback-hell-1bc8dc4a2012/>.

Accessed 27th September 2021.

Malviya, G., 2015. *Password Security*.

Available at: <https://www.loginradius.com/blog/async/password-secure/>.

Accessed 20th August 2021.

Mba, J., 2017. *Do You Want a Better Understanding of Buffer in Node.js? Check This Out*.

Available at: <https://www.freecodecamp.org/news/do-you-want-a-better-understanding-of-buffer-in-node-js-check-this-out-2e29de2968e8/>.

Accessed 26th July 2021.

MDN, 2021. *MDN Web Docs*.

Available at: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/development_environment.

Accessed 29th June 2021.

MongoDB, 2021. *MongoDB Official Documents*.

Available at: <https://docs.mongodb.com/>.

Accessed 11th August 2021.

Mongoose, 2021. *Mongoose Official Documents*.

Available at: <https://mongoosejs.com/docs/guide.html>.

Accessed 27th September 2021.

OpenJS, 2011. *How to Use Buffer in Node.js*.

Available at: <https://nodejs.org/en/knowledge/advanced/buffers/how-to-use-buffers/>.

Accessed 26th July 2021.

OpenJS, 2021. *Node.js v15.14.0 documentation*.

Available at: <https://nodejs.org/docs/latest-v15.x/api/>

Accessed 22nd July 2021.

O'Shannessy, P., 2015. *Deprecating JSTransform and react-tools*.

Available at: <https://reactjs.org/blog/2015/06/12/deprecating-jstransform-and-react-tools.html>.

Accessed 1st July 2021.

Postman, 2021. *Postman, product tools*.

Available at: <https://www.postman.com/product/tools/>.

Accessed 27th September 2021.

Preul, W. L., 2017. *Node.js at Joyent*.

Available at: <https://www.joyent.com/blog/nodejs-at-joyent>.

Accessed 26th July 2021.

RapidAPI, 2021. *Endpoint - What is an API Endpoint?*.

Available at: <https://rapidapi.com/blog/api-glossary/endpoint/>.

Accessed 17th August 2021.

Ravichandran, A., 2018. *React Virtual DOM Explained in Simple English*.

Available at: <https://programmingwithmosh.com/react/react-virtual-dom-explained/>.

Accessed 27th September 2021.

Robinson, D., 2017. *Introducing Stack Overflow Trends*.

Available at: https://stackoverflow.blog/2017/05/09/introducing-stack-overflow-trends/?_ga=2.43445346.611692959.1628357118-168000583.1620139087.

Accessed 7th August 2021.

Rogozhny, D., 2020. *How to Display Modal Dialog in React with react-modal*.

Available at: <https://www.newline.co/@dmitryrogozhny/how-to-display-modal-dialog-in-react-with-react-modal--dbf46cda>.

Accessed 28th September 2021.

Salim, A. R., 2020. *Node.js Dependencies*.

Available at: <https://nodejs.org/en/docs/meta/topics/dependencies/>.

Accessed 28th June 2021.

Sathananthan, S., 2021. *How To Reuse React Components*.

Available at: <https://medium.com/codezillas/how-to-reuse-react-components-851ffcc68a9c>.

Accessed 27th September 2021.

Soueidan, S., 2015. *Understanding Stack Contexts*.

Available at: https://tympanus.net/codrops/css_reference/z-index/.

Accessed 27th September 2021.

SpyCloud, 2019. *How long would it take to crack your password?*.

Available at: <https://spycloud.com/how-long-would-it-take-to-crack-your-password/>.

Accessed 27th September 2021.

StrongLoop & IBM, 2019. *Express Release Change Log*.

Available at: <https://expressjs.com/en/changelog/4x.html>.

Accessed 27th September 2021.

StrongLoop, IBM, 2017. *Express Routing*.

Available at: <https://expressjs.com/en/guide/routing.html>

Accessed 22nd July 2021.

Surasani, L., 2019. *Let's get hooked: a quick introduction to React Hooks*.

Available at: <https://www.freecodecamp.org/news/lets-get-hooked-a-quick-introduction-to-react-hooks-9e8bc3fbaeac/>.

Accessed 14th July 2021.

Taylor & Francis Group, 2010. *A Practical Guild to Content Delivery Networks*. Second edition. Boca Raton, FL: CRC Press.

Truth, S., 2011. *Do Not Rely on Client-Side Validation*.

Available at: <https://blog.securityinnovation.com/blog/2011/07/do-not-rely-on-client-side-validation.html>.

Accessed 27th September 2021.

TutorialsTeacher, 2020. *Advantages of Express.js*.

Available at: <https://www.tutorialsteacher.com/nodejs/expressjs>

Accessed 27th September 2021.

W3Schools, 2021. *JavaScript HTML DOM*.

Available at: https://www.w3schools.com/js/js_htmlDOM.asp

Accessed 12th July 2021.

Wysocki, R. K., 2021. *Effectiveness Software Project Management*.

Available at: https://www.oreilly.com/library/view/effective-software-project/9780764596360/9780764596360_ch05lev1sec2.html.

Accessed 27th September 2021.

Yushkevych, A., 2021. *How to check node.js version?*.

Available at: <https://monovm.com/blog/how-to-check-nodejs-version/>.

Accessed 27 September 2021.

Zammetti, F., 2020. *Modern Full-Stack Development*. First edition. Pottstown PA: Apress.