Tampere University of Applied Sciences

Authentication Mechanisms for a Web Developer

Quang Tri Cao

BACHELOR'S THESIS
April 2023

Bachelor's Degree Programme in Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Bachelor's Degree Programme in Software Engineering

CAO, QUANG TRI
Authentication and Authorization for a Web Developer

Bachelor's thesis 34 pages, appendices 0 pages
April 2023

Advancement in web development technologies has brought forth the appearance of complete standalone web applications. They are known as single page applications (SPA) which are executed completely in the browser and only rely on the server for data. Because code running on a web browser can be inspected by anyone, incorporating authentication and authorization in SPA-style web applications becomes challenging. In other words, storing user credentials on the browser code seems not secure at all.

The thesis explores and documents the token-based solution as well as some other authentication and authorization mechanisms. In addition, a web application utilizing the token-based method was implemented, through which the outcome of the author’s learning process is also demonstrated.

The project ran well, and users can login. Many new terms had to be studied thoroughly and problems arose while building the project. However, all these obstacles provided a great learning opportunity for the author. A better understanding of authentication, authorization and different authentication approaches was gained through in the thesis.

Key words: single page application, authentication, authorization, token-based authentication.

CONTENTS

ABBREVIATIONS AND TERMS

API	Application Programming Interface
CSS	Cascading Style Sheet
DOM	Document Object Model
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
REST	Representational State Transfer

1 INTRODUCTION

The 21st century has witnessed widespread usage of the Internet since its introduction in the early 1980s. The Internet is a worldwide system of computer networks connecting innumerable computers to each other. A simple internet connection consists of a client and a server. For example, when a user visits a website, his/her browser (e.g., Safari, Google Chrome) makes a request to a web server. The server then responds with the data that is used by the browser to render the webpage.

In early times, webpages were very minimalistic as most of their business logic stayed on servers. Since then, the web development technologies have grown tremendously. One significant advancement is the emerge of the single-page application (SPA). In a traditional web application, any changes on a webpage cause the browser to send a new request to the web server and render a new page. An SPA-style application, however, can comprise only one webpage whose contents are manipulated with JavaScript that executes in the browser so that redundant server calls and webpage refreshing are reduced.

In web applications, the front end commonly communicates with the back end using hypertext transfer protocol to operate data on the server. Operations are typically based on CRUD philosophy, which is an acronym for: read, create, update, delete. But before the back end can perform the operation, the process of authorization and authentication steps in to protect the data from unauthorized users.

Authentication and authorization operations have always been a part of the backend code because it is not available to the public. But with the introduction of token-based authentication, moving those operations to the web browser (or front end) has become possible. Web browsers now provide better security to store sensitive information.

The focus of the thesis is to study and show how to apply the token-based authentication and authorization in an single-page web application. The thesis also

documents some basic authentication and authorization methods that any frontend developers should be aware of.

2 BACKGROUND

2.1 A web application

A web application uses Hypertext Markup Language (HTML) to present the content, Cascading style sheets (CSS) to style the content and JavaScript (JS) to make HTML elements interactive. Web applications are normally hosted by a web server.

The term “web server” can refer to both software and hardware. In terms of hardware, a web server is a computer system that hosts websites. It runs web server software (e.g., Apache or Microsoft IIS) which provides access to hosted webpages over the Internet, and it stores a website's component files (HTML documents, CSS stylesheets, and JS files). Regarding the software side, a web server controls how users access hosted files. (Tamara J, 2023)

A web server typically listens to requests sent by the browser and delivers back the appropriate resources for each request. Upon receiving the requested content, the browser parses the data and renders it to users. Considering this flow, web applications can be divided into two categories: traditional web applications and single-page applications.

2.1.1 Traditional Web Applications

A traditional web application is indicated by the fact that the whole webpage is refreshed every time data displayed on the browser get changed or updated. Traditionally, web applications contained minimal use of JavaScript and only rendered the HTML data as instructed while most of the application’s logic stayed on the server.

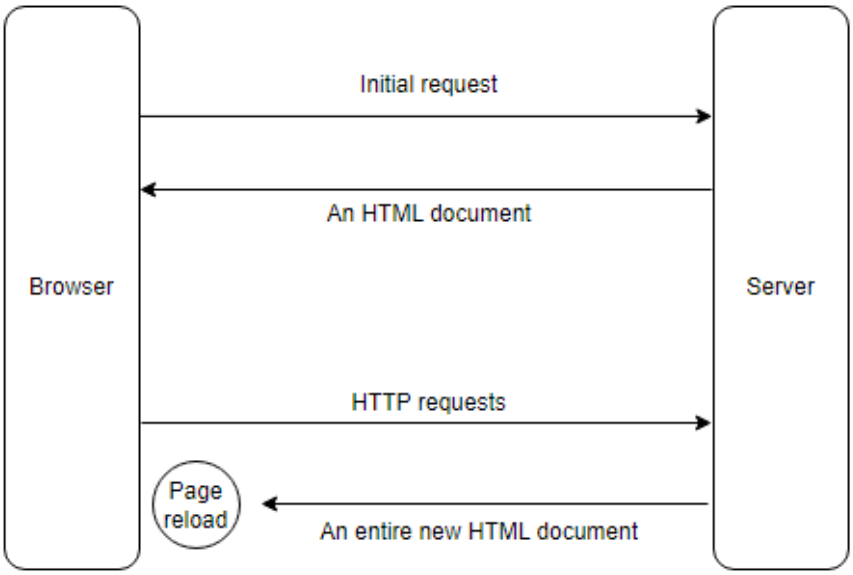

Figure 1. Traditional web applications lifecycle.

In Figure 1, the server responds to every requests with a new HTML document. For example, when users click a button to change data, the browser would receive a new HTML document that has the new updated data and construct a new webpage from scratch. This is also known as the multi-page application since it consists of a large number of separate pages. The approach is best suited for enormous websites of which Amazon and eBay are great examples.

2.1.2 Single-Page Applications (SPA)

In contrast to traditional web applications, single-page applications (SPAs) do not fetch multiple webpages but load the whole web application into one. On the webpage, only necessary parts are dynamically updated, usually in response to user actions, without page refreshing.

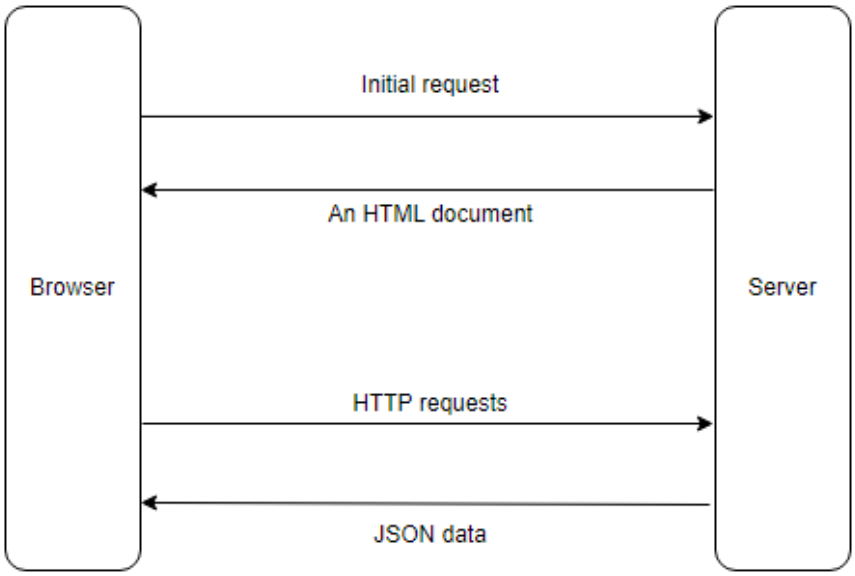

Figure 2. Single-page applications lifecycle.

In Figure 2, after the first page finishes loading, requests are typically responded with JavaScript Object Notation (JSON) files. JSON is a standard and lightweight format file for representing, storing, and transferring data (Edwin, H. 2019). The browser uses the data in the JSON file to rewrite (or update) directly on the webpage using JavaScript. For example, when a user clicks a button to like a post, only the element that displays the number of likes gets updated while other elements stay intact, and the page is not reloaded.

Because redundant re-renderings and server calls are reduced, SPAs are more fluid, responsive and require less network bandwidth. They also provide offline capabilities as the browser can cache all the necessary data to work offline (e.g., Google Docs offline mode). Gmail, Airbnb, and Facebook are all great examples of SPAs. (Shah, 2022)

2.2 REST

An Application Programming Interface (API) is a mechanism that enables data transmission between two software products. An API architecture usually consists of a client that dispatches requests, and a server that sends back responses. For example, the weather application (the client) retrieves weather data in the response from the weather database (the server) via API and dis-

plays the information to users. Among four different API architectures, Representational State Transfer (REST) is the most popular one. (Amazon, No Date)

In REST, individual data objects are resources, and each resource has an associated unique address – a URL (Uniform resource locator). Having a consistent and robust REST resource naming strategy will leverage the REST architecture to its full potential (Lokesh, G. 2022). One convention is to create the unique address for a resource by combining the name of resource type with the resource's unique identifier. For example, assuming the root URL is /, the resource type of a user is users, then the unique address for a user resource with the identifier 3 is /users/3 while the URL for the entire collection of all user-type resources is /users.

Table 1. Different operations on resources in REST.

URL	HTTP verb	Functionality
users	GET	Retrieve all resources
users/3	GET	Retrieve the identified resource
users	POST	Create a new resource
users/3	DELETE	Delete the identified resource
users/3	PUT	Replace entirely the identified resource with the data attached in the request
users/3	PATCH	Replace partly the identified resource with the data attached in the request

```
POST https://domain.com/user/ HTTP/1.1

Content-Type: application/json
Accept: application/json
Authorization: bearer abcxyz

{
  "name": "Quang Tri Cao",
  "age": 23
}
```

Figure 3. An example of a POST request.

A request often has four parts: an HTTP verb (e.g., a GET request in Table 1), a header, a path to the resource and an optional message body. In Figure 3, the first line basically tells that the client wants to create a new user (HTTP POST method) and the HTTP version of the request is 1.1. The response from the server typically contains the HTTP version that it supports so the requests must state HTTP version accordingly (IBM, 2021). Three next lines construct the header of the request. The Content-type header indicates the type of the content attached in the message body is JSON, the Accept header tells that the client wants data in JSON format, and the Authorization header tells the sever that the client has the right to make the request. Lastly is the body of the request that contains the payload of data wrapped inside curly braces.

One important feature of the REST API architecture is statelessness, which means that neither servers nor clients save information of each other in subsequent requests. In another words, the server does not save the data sent from the client whether the process is successful or not. Therefore, each request-response cycle is totally independent which results in a well-behaved, predictable, and reliable web application.

HTTP and REST

As mentioned before, the front end communicates with the back end via Hypertext Transfer Protocol (HTTP). However, SPAs commonly utilize the REST architecture to make the communication easier. Although the terms HTTP and REST are often used interchangeably, they are different. HTTP is the standard internet communication protocol that powers most of interactions happening on the internet. Meanwhile, REST refers to a set of rules enforcing developers to use HTTP methods (e.g., GET, PUT, POST) correctly (Michael, 2022). For instance, an HTTP GET request can do a delete operation as follows:

GET http://abc.com?method=delete&item=xxx

But in REST, the request would be done in a much more obvious way:

DELETE http://abc.com?item=xxx

So eventually REST is just a recommended architecture style that controls and manipulates data via HTTP commands.

2.3 Authentication and authorization

Generally, the authentication process identifies the user who is requesting access to a restricted system or resources. Authorization, meanwhile, verifies requester's permission to perform certain tasks on the system or resources. To put it simply, authentication answers the question "Who are you?" and authorization answers the question "Are you allowed to do that?". Authentication and authorization typically come together, and authentication should come before authorization. Once authenticated, admins (or moderators) grant users different access level depending on their authorization level in the system. (Andra Andrioaie, 2022)

User authentication is critical because it would protect confidential information from being accessed by unauthorized users. In web applications, the role of authentication comes into play when the front end wants to access APIs endpoints exposed by the back end (or a web server). An API endpoint is basically a specific digital location that directs requests to the correct address in the back end (TechTarget, 2021). Next, some of the common authentication mechanisms used in web applications are outlined before the token-based authentication and authorization is discussed in detail.

2.4 HTTP Basic Authentication

HTTP basic authentication, which is built into the HTTP protocol, is the most basic method. With the authentication method, login credentials (username and password) are sent in each request in the Authorization header as follows:

Authorization: Basic arandomstring

Username and password of a user are concatenated together separated by a colon, *username:password*, before being encoded with Base64 and results in the string "arandomstring". A logout functionality is not supported but developers can implement it by overwriting the valid credential with an invalid one. (Julian F. Reschke, 2015)

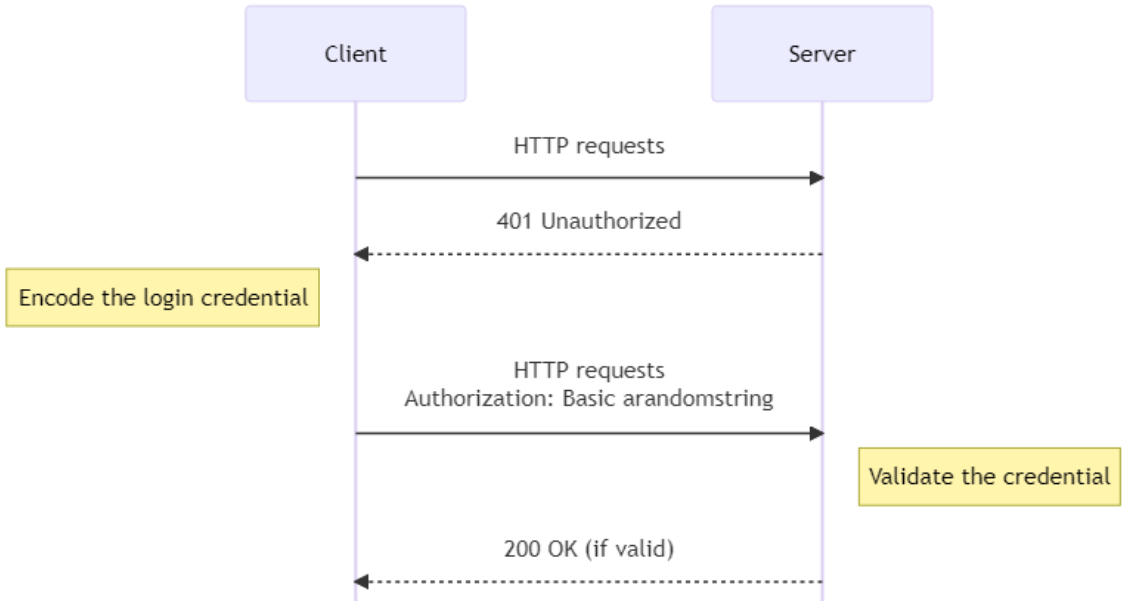

Figure 4. The principle of HTTP Basic Authentication.

In Figure 4, HTTP requests to a restricted resource without a proper Authorization header would receive 401-status code responses. With encoded credentials that are attached in the Authorization header, the server validates them, if valid, an HTTP response with a 200-status code is sent back which means the request is conducted successfully.

Overall, the method is quite straightforward to understand and is supported by almost all browsers. However, it does have a serious flaw, the Base64 encoded string can be easily decoded to obtain the attached credential. The reason is that the Base64 is not an encryption mechanism, but just a way of representing the data in another format. Therefore, the authentication mechanism itself does not inherently provide any security and is vulnerable. To address the problem, the use of HTTPS (Hypertext transfer protocol secure), which is the secure version of HTTP, is essential when applying the method.

2.5 Session-based Authentication

Session-based authentication is a stateful protocol as it stores user's state on the server. Due to its statefulness, the approach does not work with stateless APIs (e.g., REST). With the method, each current logged-in user has a unique

session file that is stored in the database and when a user logs out, the server would destroy the corresponding session file.

Initially users submit their login credentials. If the credential is verified, the user logins successfully and the server generates a session file. A session is a file, mostly in JSON format, that stores information about the user (e.g., login time, expiration date). The session file is stored in a database while only some necessary information of the file, especially its ID, are delivered back to the client as a cookie. The browser stores the received cookie and attaches it in requests, so username and password are only required in the login step. Upon processing requests with attached cookies, the server validates the session ID within the cookie with the one stored in the database and perform the request if the ID is valid. Due to working with cookies, the technique is also known as the cookie-based authentication. The process is depicted in the following Figure 5.

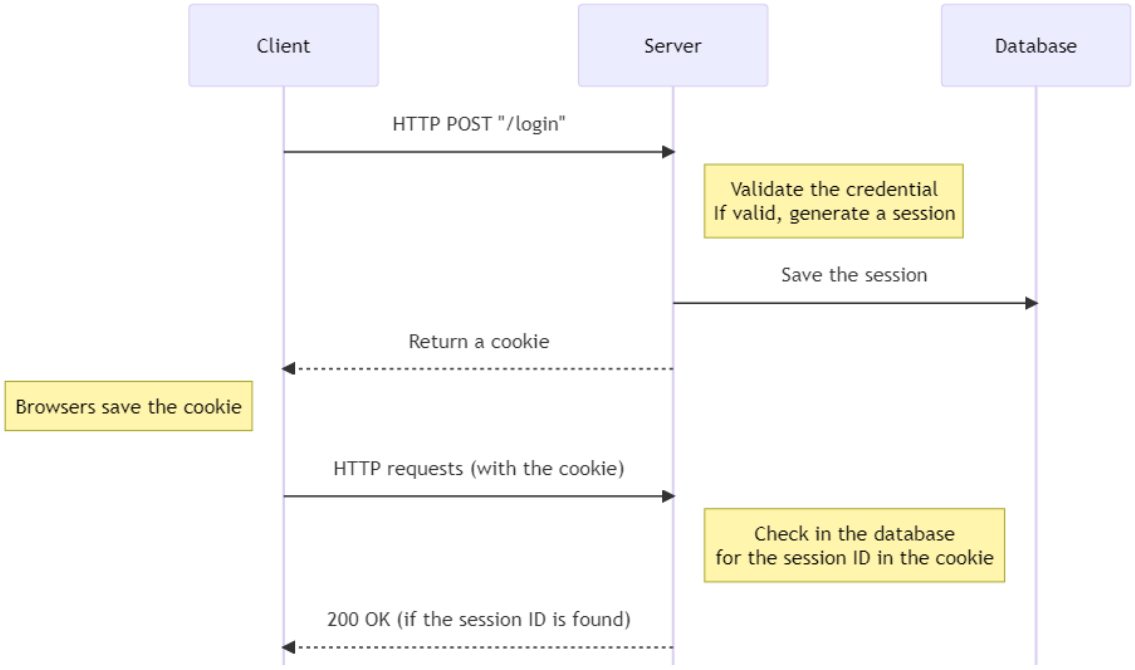

Figure 5. The principle of session-based authentication.

One remarkable advantage of the session-based authentication is that administrators are in full power over users management. Because sessions are stored in the server side, administrators can easily neutralize any suspicious users. On the other hand, as the server is fully responsible for managing sessions, scalability problems can easily arise when expanding the server.

2.6 Token-based Authentication

Token-based authentication is a stateless protocol. At first, a user submits his/her credential as usual. If the credential is valid, a token is generated, digitally signed using a secret key and sent back. The browser saves the token and attach it in requests as shown in Figure 6, which is like sending cookies in the session-based authentication method.

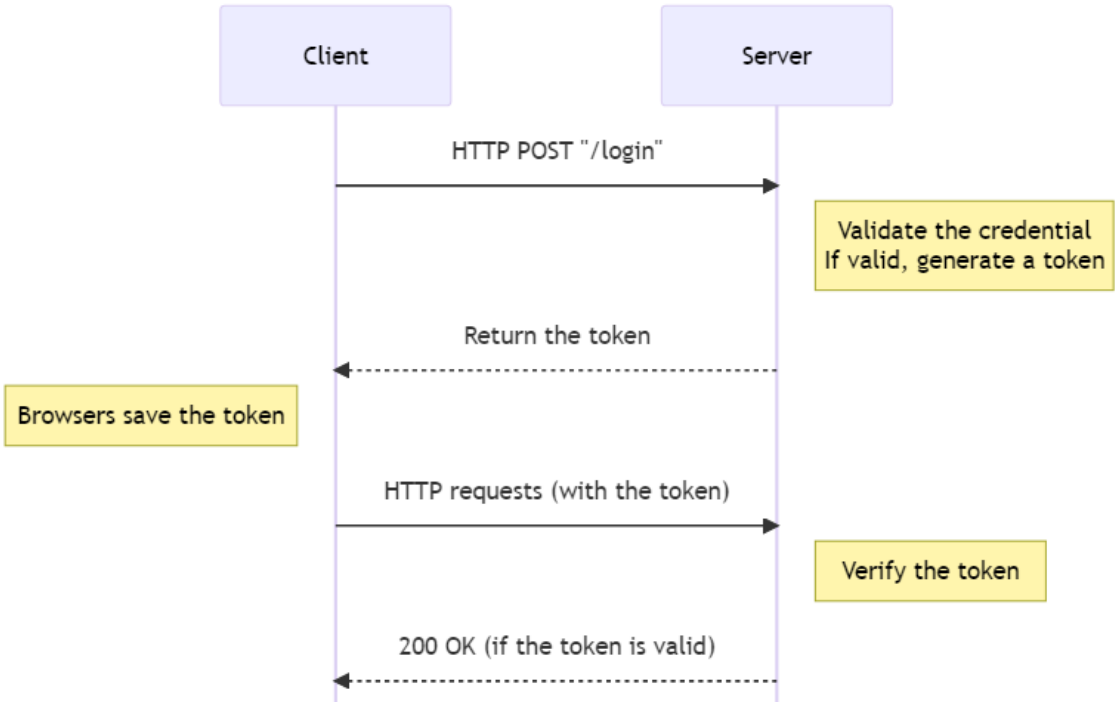

Figure 6. The principle of the token-based authentication

A token is capable of storing user credentials and related information securely and proving that the data inside it is authentic. Tokens are typically not stored in the server but by users which makes the mechanism stateless. In other authentication and authorization mechanisms, the server normally needs an authentication process to identify users before authorizing. With the token-based approach, the server can directly authorize the request relying on the validity of the token which would greatly improve the performance. The validity process of a token often includes decrypting the token and verify the signature inside it.

Anatomy of a JSON Web Token

Among various token types (e.g., hardware tokens, one-time password tokens), JSON Web Token (JWT) is the most common one. A JWT consists of three key components: header, payload, and signature, that are concatenated by a dot. The header defines the token type, which in this case is JWT, and the encryption algorithm. The payload contains user credentials and other related information (e.g., the role of the user). The signature includes a secret key that is used to verify the authenticity of data in the payload, or that the data stayed intact while being transferred. A JWT looks as follows:

xxxxx.yyyyy.zzzzz

“xxxxx” and “yyyyy” are the base64url encoded forms of the header and payload representatively. To develop a signature, the encoded forms of header and payload are chained together then hashed with a secret key. The hashing algorithm is defined in header. Because only the server knows the secret key, only it can create an authentic signature for a new token (Aviad, 2021). Tokens are not encrypted; they can be decoded revealing header and payload parts but not the secret key.

Pros and cons of token-based authentication

Token-based authentication recently has gained popularity because of the following benefits. First, the mechanism is stateless. The server does not need to store tokens since tokens are normally validated directly using the signature. Therefore, requests are responded to faster as a database lookup is no longer required. The second advantage is its efficiency. The server can easily handle and verify a large number of tokens, which makes applications easier to scale to the number of users accessing. Thirdly, the flexibility of tokens enables multiple applications to use the authentication method simultaneously. Lastly, considering the security aspect, tokens can only be verified when the private key is recognized by the server that uses the key to generate tokens, hence tokens can be considered as a secure method of authentication.

On the other hand, the method has many drawbacks. Firstly, servers are not able to perform security operations because tokens are held by users. Hence,

the method is vulnerable to cross-site scripting or cross-site request forgery attacks. Moreover, tokens cannot be deleted but only expire. So hypothetically if tokens get leaked, the attacker can have unlimited access to the server resource. Thus, it is important to set a short expiration time to a token. However, the solution makes the token-based method less ideal for systems that allow users to remain logged-in for prolonged periods. These short-time tokens might cause users to re-login multiple times. (Geeksforgeek, 2022)

3 TECHNOLOGIES

3.1 React

The front end of the demo web application was built using React (or React.js). React is a free and open-source JavaScript library maintained by Facebook, which has gained popularity. The main purpose of the library is to ease frontend development.

The document object model (DOM) is a programming interface that represents a web page. By working with DOM, developers can directly change the document structure, style, or content of a web page. React, however, creates a virtual DOM in memory and any manipulations made by developers are changed on that virtual DOM first. React then compares the two version (the actual DOM and the updated one in memory), finds what has been changed and only updates that part accordingly. Meanwhile directly manipulating the DOM would typically cause the whole page refresh.

In React, HTML content can be written in a JavaScript file using JavaScript XML (JSX) syntax, which is a syntax extension for JavaScript and is preferred by most developers. A React application is constructed of components which are independent, reusable user interface elements. Technically React components are JavaScript functions, and a single component can contain HTML, CSS, and JavaScript simultaneously. A React component returns JSX which under the hood would be compiled into JavaScript by Babel.

```
const App = () => {
  const name = "Quang Tri Cao";

  return <div>Say hello to {name}</div>;
};

ReactDOM.createRoot(document.getElementById("root")).render(<App />);
```

Figure 7. A React application.

The code presented in Figure 7 has a component named App and renders to the browser the line: Say hello to Quang Tri Cao. The content returned by the

App component is an example of a JSX syntax. The last line of the file simply renders the App component into a div element with ID “root” in the HTML file, which in turn causes the screen to display the line.

Creating a React application from scratch was notorious for requiring many tools that were difficult to configure. Nowadays thanks to the create-react-app framework, getting started working with React applications has been easier. One benefit of the framework is that it creates React applications that are configured to compile automatically. The frontend of the demo application was initialized using the framework.

3.2 Express

The back end of the demo web application was built on top of NodeJS. NodeJS is an environment for executing JavaScript outside browser, so it is neither a programming language nor a framework. To ease server-side development with NodeJS, the Express library, which is based on NodeJS, was utilized. Benefits of using Express include providing middleware and routing functionalities.

```
const http = require("http");

const server = http.createServer((request, response) => {
  if (request.url === "/about") {
    response.end("About Page");
  } else {
    response.end("Hello World");
  }
});

const PORT = 3001;
server.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Figure 8. A NodeJS server.

Figure 8 demonstrates how the server works with only NodeJS. The “http” variable represents the http module that contains the “createServer” function to create a server. The server is bound to listen to HTTP requests sent to the

PORT 3001. On the address localhost:3001, “Hello World” is rendered. (DigitalOcean, 2020)

Since routing is not supported in NodeJS, navigating to other addresses requires defining all possible paths in the “url” field of the object “request” in a single if-else block. This navigation approach will soon become cumbersome and hard for reading when the application grows. Moreover, besides rendering “About Page” on the address localhost:3001/about as expected, the server does not throw an error while accessing an undefined URL. In another word, the browser would render “Hello World” regardless of the path (e.g., localhost:3001/an-undefined-path). This server creation approach and its limits can be improved when using the Express library as show in Figure 9. An Express server..

```
const express = require("express");
const app = express();

const middlewareExample = () => {
  // code
};

app.use(middlewareExample);

app.get("/", (request, response) => {
  response.send("Hello World");
});

app.get("/users", (request, response) => {
  // code
});

app.post("/users", (request, response) => {
  // code
});

const unknownEndpoint = (request, response) => {
  response.status(404).send({ error: "Unknown endpoint" });
};

app.use(unknownEndpoint);

const PORT = 3001;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Figure 9. An Express server.

The first two lines in Figure 9 create and store an Express application in the variable “app”. With Express, the route for each path is clearly defined and more readable and the REST architecture can be easily applied. The first route handles GET requests made to the path “/” and send back “Hello world”. The second and the last one handles GET and POST requests to “/users” representatively.

The Express library also provides middleware feature. Middleware is a function that executes after the server receives the request and before the response is sent to the client, and several middlewares are executed in the order they are taken into use (Pratik, 2022). It also handles “request” and “response” objects the same way a route does. For instance, in the Figure 9. An Express server., the “middlewareExample” middleware always executes first. And the “unknownEndpoint” middleware should be placed at the end, otherwise every address would throw an error, even the valid one. Accessing invalid paths is handled gracefully thanks to the “unknownEndpoint” middleware.

3.3 MongoDB

When sketching an application in the project initiation phase, deciding which database to use is essential. The application used MongoDB, which is a document database, to store data indefinitely. In general, MongoDB stores a data record as a document, and a database store at least one collection of documents. (MongoDB, No Date)

Figure 10. A document in MongoDB

Figure 10 describes an example of a document format. A document has a field-value format like an object in JavaScript. A field can contain many types varying from string, number, array to even ObjectId of other documents. As its name implies, ObjectId type in the “_id” field makes the document distinctive and can be referred to by other documents. For example, the “user” field in the figure refers to another document while the “_id” is the identifier of the document itself.

3.4 Mongoose

Mongoose is a library that relieves working with MongoDB since it offers a high-level API. The library manages is used to translate between JavaScript objects in code and documents in MongoDB so that, for example, saving JavaScript objects to MongoDB as documents would be straightforward.

There are two important terms in Mongoose which are schema and model. A schema defines the data structure of a MongoDB document, default values, validators, etc. Meanwhile a model creates an instance based on a schema, provides an interface for database to perform actions (e.g., create, query, update or delete) (Kim, 2019)

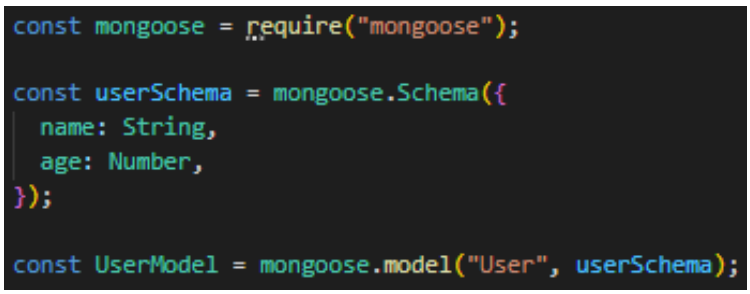

Figure 11. A Mongoose schema and model.

The first line in Figure 11 enables using the Mongoose library. A schema of a user is created and stored in the “userSchema” variable which notifies how user objects are going to be stored in MongoDB. In the model definition, the "User" parameter is the singular name of the model. The name of the collection in MongoDB is the lowercase plural of the first parameter which in this case would be “users” due to the Mongoose convention. The convention automatically

names a collection in a plural form while a schema is singular (Mongoose, No Date). Since document databases like Mongo are schemaless, it is possible to store documents with completely different fields in the same collection. With Mongoose, data is given a schema at the level of the application that defines the shape of the documents stored in a collection in MongoDB.

4 IMPLEMENTATION

4.1 Project overview

The demo project is a web application where users can login and add, delete, and view blogs. The chapter focuses on explaining how to implement the authentication method and how the server responds to a request that is attached with a token.

The front end of the project is a single-page web application, and its back end follows the REST API architecture. Beside the root path "/" that normally renders the whole application, the project also has an additional path "/sign-in" that renders the login page. To avoid conflicting with paths in the front end, all API endpoints were added the string "/api" in the beginning. Table 2 shows some example endpoints.

Table 2. Operations on different resource types in the back end

Resource	URL	HTTP Verb	Functionality
Login	/api/login	POST	Send username and password of a user and receive a token if valid
Blog	/api/blogs	GET	Retrieve all blogs

4.2 Database

The application's database has two collections named "users" and "blogs" as shown in Figure 14 and two corresponding Mongoose schemas defined in Figure 12 and Figure 13. For example, Figure 12 describes the schema of a user in a JavaScript object before being stored as a document in the "users" collection. There are two required and string type fields, and their value must contain at least three characters. Both schemas and their models are defined in separate files so that they can be imported and easily used all over the project code.

```
const mongoose = require("mongoose");

const userSchema = mongoose.Schema({
  username: {
    type: String,
    required: true,
    minlength: 3,
  },
  password: {
    type: String,
    required: true,
    minlength: 3,
  },
});

module.exports = mongoose.model("User", userSchema);
```

Figure 12. A user schema

```
const mongoose = require("mongoose");

const blogSchema = mongoose.Schema({
  title: {
    type: String,
    required: true,
  },
  url: {
    type: String,
    required: true,
  },
  author: {
    type: String,
    required: true,
  },
  likes: {
    type: Number,
    default: 0,
  },
});

module.exports = mongoose.model("Blog", blogSchema);
```

Figure 13. A blog schema

Figure 14. "users" and "blogs" collections in MongoDB

4.3 Login route in back end

At first, the jsonwebtoken library, that was used to generate JSON web tokens, was installed via the command line “npm install jsonwebtoken”. To implement login functionality, a new route was created to handle login requests sent to the path “/api/login”.

```
const express = require("express");
const app = express();
const jwt = require("jsonwebtoken");
require("dotenv").config();

app.post("/api/login", async (request, response) => {
  const { username, password } = request.body;

  const user = await User.findOne({ username });

  if (!user || !password === user.password) {
    return response.status(401).json({
      error: "wrong username or password",
    });
  }

  const userForToken = {
    username: user.username,
    id: user._id,
  };

  const token = jwt.sign(userForToken, process.env.SECRET);

  response.status(200).send({ token, username: user.username });
});
```

Figure 15. A route for login functionality

In Figure 15, the first and second lines create an Express application and store it inside the variable “app”. The third line takes into use the jsonwebtoken library and the fourth line enables the use of environment variables. At first, the event handler function destructures the body of the request into an object with two fields “username” and “password”. Next, the function searches the “users” collection for a user that matches the “username” field’s value. If the user is not found or the password does not match, the request is responded with the status code 401 and an error message. Otherwise, a token is created by being digitally signed using the “sign” method from the jsonwebtoken library.

The sign method receives two parameters: the first one is an object containing username and ObjectId of the user document, and the other is the environment variable “SECRET” defined in the “.env” file as in Figure 16. The file is not visible to users which enables the environment variable to act as a secret key and only the server and parties who know it can generate a valid token. A successful request is responded with the status code 200 and an object containing the

token and the username. The route can be tested by using VS Code REST-client. Figure 17 shows an example of a successful login request.

```
SECRET = aRandomStringThatActsAsTheSecretKey
```

Figure 16. The SECRET environment variable in the ".env" file.

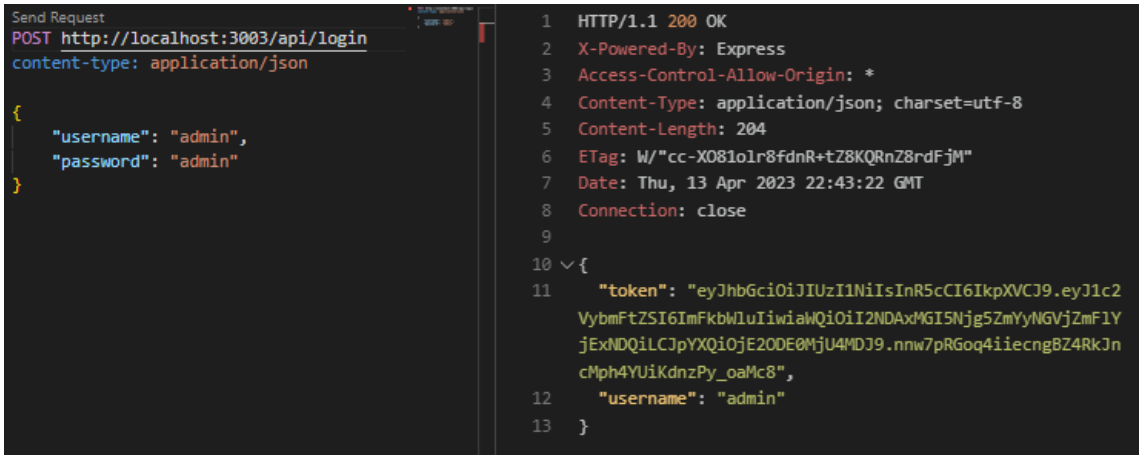

Figure 17. Login route test

4.4 Login page in front end

In the front end, the project used the Axios library to send requests. Login is done by sending an HTTP POST request to server address “/api/login” as shown in Figure 18. The code in the figure was extracted to a separate file so that different places in the project could easily import and use it. After a successful login, the browser will receive an object containing the token and the username in the field “data” of the “response” object.

```
import axios from "axios";
const baseUrl = "/api/login";

const login = async (credentials) => {
  const response = await axios.post(baseUrl, credentials);
  return response.data;
};

export default { login };
```

Figure 18. Login requests.

```
import { useState } from "react";
import { useNavigate } from "react-router-dom";
import loginService from "../services/login";

const SignIn = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    loginService
      .login({
        username: username,
        password: password,
      })
      .then((response) => {
        window.localStorage.setItem("loggedBlogAppUser", JSON.stringify(response));
        useNavigate("/");
      });
  };

  // ...
```

Figure 19. Login functionality.

Figure 19 partly describes the “SignIn” component that is responsible for rendering the login page. Once users click the submit button to login, the “handleSubmit” function sends an objects that contains username and password to the API endpoint “/api/login” using the imported “login” function. If the user logs in successfully, the component saves the received object “response” in the browser’s local storage and the user is navigated to the home page.

The local storage is typically a key-value database, and the value has to be parsed to JSON (MDN. No Date). The JSON data in Figure 19. Login functionality. is saved with a key named “loggedBlogAppUser” so that the data can be retrieved as follows:

window.localStorage.getItem(“loggedBlogAppUser”)

Data in the local storage does not disappear when the page is re-rendered. The local storage can be inspected manually by typing “localStorage” in the Console tab on the browser as in Figure 20.

```
> localStorage
<-  Storage {loggedBlogAppUser: '{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI
  ▶ 6IkpXVCJ9.eyJ…bwjcUa5tqx1mqJ_QpmXgo9Gov0-Y","username":"admin"}',
    length: 1}
```

Figure 20. The local storage on the browser.

4.5 A GET request attached with a token

In the project, all requests had to be attached with a valid token. The project used the Bearer Authorization mechanism to send a token in the Authorization header in each request (ReqBin, 2022). After login successfully, the client would send a GET request that looks roughly like Figure 21.

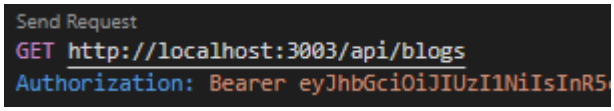

Figure 21. A GET request attached with a token.

```
app.get("/api/blogs", async (request, response) => {
  const getTokenFrom = (request) => {
    const authorization = request.get("authorization");
    if (authorization && authorization.startsWith("Bearer ")) {
      return authorization.replace("Bearer ", "");
    }
    return null;
  };

  const token = getTokenFrom(request);

  const decodedToken = jwt.verify(token, process.env.SECRET);

  if (!decodedToken) {
    return response.status(401).json({ error: "token invalid" });
  }

  const blogs = await Blog.find({});
  response.json(blogs);
});
```

Figure 22. The handler function of the GET request.

Upon receiving requests made to the path “/api/blogs”, the server detaches the token from the header to verify it as shown in Figure 22. If there is a value in the Authorization header and it starts with “Bearer “, then the function “getTokenFrom” return exactly the token and store it in the “token” variable. If not, the “token” variable is undefined and the server declines to perform the request. Next, the code verifies the token using the “verify” method from the jsonwebtoken library. The method receives two parameters, the token sent from client and the

secret key in the ".env" file. If the token is valid, the variable "decodedToken", which is the result of the verification process, must contain a value. Otherwise, the variable is undefined, and the server send back a response with status code 401 and an error message. After the token is verified, the server finds all blogs in the "blogs" collection and return them in JSON format.

4.6 Neutralize a token

As discussed earlier, the token-based authentication has one major problem which is that once the user gets a token; the API has a blind trust to the token holder. So how would the server revoke the access rights of the token holder? There are two solutions to the problem. One is to limit the validity period of a token when creating one.

```
const token = jwt.sign(userForToken, process.env.SECRET, {
    expiresIn: 60 * 60,
});
```

Figure 23. A token with an expiration time.

The token in Figure 23 expires in 60*60 seconds, that is, in one hour. Once the token expires, the client app needs to get a new token, usually by re-logging in. The shorter the expiration time, the safer the solution is, but it creates a pain to a user as one must login to the system more frequently.

The other solution is to save tokens on the server and to check if the token attached in requests corresponds to the still-valid-token. With this scheme, the access rights can be revoked at any time. This solution is known as a server-side session, which is similar to the session-based authentication.

5 DISCUSSION

The thesis has explained and documented some authentication methods, especially in detail the token-based authentication. In addition, a simple React application with SPA-style was created to apply the authentication method. In the end, the main goal was achieved as all required features were implemented successfully. Throughout the project, the author has also learned how to generate and verify a JSON Web Token using the jsonwebtoken library and how to implement the token-based approach.

However, there is still room for improvement in the application. The backend structure was put in a single file, which apparently will soon become overload when the number of API endpoints increases. The folder structure needs some great modifications such as extracting Mongoose schemas and routes definition. Moreover, there were no testing files in the application. Some end-to-end tests or unit tests would be great as they could help find bugs before deploying the application into production.

REFERENCES

Tamara J. 2023. What is a web server? How it works and more. Read on 01.11.2022.
https://www.hostinger.com/tutorials/what-is-a-web-server

Shah, A. 2022. Single page application (SPA) vs. multi-page application (MPA) – which one is the best in 2022. Read on 10.11.2022.
https://www.tekrevol.com/blogs/spa-vs-mpa/

Edwin, H. 2019. JSON Objects Explained. Read on 01.02.2023
https://www.shapediver.com/blog/json-objects-explained

IBM. 2021. The HTTP Protocol. Read on 02.03.2023.
https://www.ibm.com/docs/en/cics-ts/5.3?topic=concepts-http-protocol

Amazon. No Date. What is an API (Application Programming Interface). Read on 20.01.2023.
https://aws.amazon.com/what-is/api/

TechTarget. 2021. What is an API Endpoint? Read on 10.04.2023.
https://www.techtarget.com/searchapparchitecture/definition/API-endpoint

Michael, P. 2022. Difference Between REST and HTTP. Read on 07.04.2023.
https://www.baeldung.com/cs/rest-vs-http

Lokesh, G. 2022. REST Resource Naming Guide. Read on 07.04.2023.
https://restfulapi.net/resource-naming/

Andra, A. 2022. Authentication vs. Authorization: The Difference Explained. Read on 20.11.2022.
https://heimdalsecurity.com/blog/authentication-vs-authorization/

Julian, R. 2015. The 'Basic' HTTP Authentication Scheme. Read on 08.04.2023.
https://www.rfc-editor.org/rfc/rfc7617.txt

Aviad, M. 2021. All you need to know about JWT Authentication. Read on 05.12.2023.
https://frontegg.com/blog/jwt-authentication

Geeksforgeek. 2022. Session vs Token Based Authentication
https://www.geeksforgeeks.org/session-vs-token-based-authentication/

DigitalOcean. 2020. How to create a web server in Node.js with the HTTP Module. Read on 10.04.2023.
https://www.digitalocean.com/community/tutorials/how-to-create-a-web-server-in-node-js-with-the-http-module

Pratik, D. 2022. Complete Guide to Express Middleware. Read on 12.04.2023.
https://reflectoring.io/express-middleware/

MongoDB. No Date. Databases and Collections. Read on 12.04.2023.
https://www.mongodb.com/docs/manual/core/databases-and-collections/

Kim, M. 2019. Mongoose Schemas in React. Read on 13.04.2023.
https://medium.com/@KLMcGrath2/mongoose-schemas-in-react-f5c0afa5a47d

Mongoose. No Date. Models. Read on 13.04.2023.
https://mongoosejs.com/docs/models.html

MDN. No Date. Web Storage API. Read on 14.04.2023.
https://developer.mozilla.org/en-US/docs/Web/API/Storage

ReqBin. 2022. Sending Authorization Bearer Token Header. Read on 11.02.2023.
https://reqbin.com/req/adf8b77i/authorization-bearer-header