TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Eric Tamm 206766IABB

Implementation and comparison of two-factor authentication methods

Bachelor's thesis

Supervisor: Viljam Puusep
MSc

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Eric Tamm 206766IABB

Kahefaktoriliste autentimismeetodite implementeerimine ja võrdlemine

Bakalaureusetöö

Juhendaja: Viljam Puusep
MSc

Tallinn 2024

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Eric Tamm

03.01.2024

Abstract

Currently, for business owners/developers, in order to choose which two-factor authentication method(s) to implement, each method needs to be researched separately. This costs precious time, which could be spent actually developing the product.

The aim of this bachelor thesis is to compare different methods of two-factor authentication (2FA). This consists of implementing multiple 2FA methods and then analysing and comparing the difficulty of their implementation, measured in how easy it was to find suitable API service provider (may not apply to some methods), dependency on outside services, tutorials, and lines of code.

In the framework of the work a website with multiple different 2FA methods is developed, then comparison between all the implemented methods is carried out.

This thesis is written in english and is 32 pages long, including 6 chapters, 34 figures and 1 table.

Annotatsioon

Kahefaktoriliste autentimismeetodite implementeerimine ja võrdlemine

Praegu tuleb ettevõtete omanikel/arendajatel valida, millist 2FA meetodit rakendada, iga meetodit eraldi uurida. See maksab väärtuslikku aega, mille võiks kulutada toote arendamisele.

Käesoleva bakalaureusetöö eesmärk on võrrelda erinevaid kahefaktorilise autentimise (2FA) meetodeid. See seisneb mitme 2FA meetodi rakendamises ning seejärel nende rakendamise keerukuse analüüsimises ja võrdlemises, mõõdetuna selles, kui lihtne oli leida sobivat API-teenuse pakkujat (ei pruugi kehtida mõne meetodi puhul), sõltuvusest välisteenustest, õpetustest ja koodiridadest.

Töö raames töötatakse välja mitme erineva 2FA meetodiga veebileht, seejärel viiakse läbi kõigi rakendatud meetodite võrdlus.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 32 leheküljel, 6 peatükki, 34 joonist, 1 tabelit.

List of abbreviations and terms

1FA/SFA	Single-Factor Authentication
2FA	Two-Factor Authentication
MFA	Multi-Factor Authentication
API	Application Programming Interface
JWT	Json Web Token
SMTP	Simple Mail Transfer Protocol
OTP	One Time Password
TOTP	Time-based One Time Password
CLI	Command Line Interface

Table of contents

List of figures

List of tables

1 Introduction

1.1 Introduction

Today, more than two thirds of businesses have a website [1], but almost half of them do not implement multi-factor authentication [2]. For example, Google Authenticator or a code sent via SMS can be used to implement 2FA. At this point, the way forward is that at some point, everyone who cares about protecting their customers' data will have to adopt at least two-factor authentication.

1.2 Problem

Currently, there is no known analysis known to Author from the software developer's point of view, which two-factor authentication is easier and faster to implement. Therefore, it would be necessary to implement different two-factor authentication options and compare the ease and speed of their implementation from the developer's point of view, as a result of which an analysis would be completed.

1.3 Objectives and expected result

The aim of the work is to realize a website with several two-factor authentication methods. Then, all realized 2FA development processes are analyzed. Analysis of the development process means how easily the developer can implement 2FA, which includes comparing how easy it is to get started with each authentication method, and the amount of lines of code for each method.

The expected results of the work are:

- A website with multiple two-factor authentication methods implemented.
- Comparison of implementation of realized two-factor authentication methods.
- The analysis resulting from the comparison, which can be taken into account when implementing new projects, where it is required to use at least one 2FA method.

From the point of view of the Author of the work, in addition, the expected result is new knowledge about the application and implementation of different two-factor authentication methods.

1.4 Work structure

This thesis consists of three parts:

- Methodology

Which discusses the definition and background of two-factor authentication, defines the primary objective of the thesis, discusses the work tools, and describes the work process.

- Results

Which describes what was accomplished: back-end and front-end of the web application.

- Analysis and conclusions

Which analyses the two-factor authentication methods, and compares them both before and after implementation.

2 Overview of similar works

This chapter provides a short overview of multiple works related to 2FA.

1. A Usability Study of Five Two-Factor Authentication Methods by Ken Reese, Trevor Smith, Jonathan Dutson, Jonathan Armknecht, Jacob Cameron, and Kent Seamons from Brigham Young University [3]

This work compares 5 common 2FA methods: SMS, TOTP, pre-generated codes (sometimes also known as backup codes), push notifications and U2F security keys. The big difference with the aforementioned work is that it studies authentication speed and convenience for the users of said 2FA methods, but it does not delve into their implementation.

2. Selecting and implementing a two-factor authentication method for a digital assessment platform by Niklas Tellini and Fredrik Vargas [4]

This work first deals with finding analysing multiple 2FA methods, mainly SMS, TOTP and a hardware key known as YubiKey, and then develops the prototype using chosen 2FA methods, which were SMS code and TOTP using authenticator app. The differences between this and current work lie not only in fact that in this case only two 2FA methods were chosen for implementation, but also the fact that they were implemented using different technologies, mainly PHP and mySQL.

3. Implementation of Two-Factor Authentication (2FA) to Enhance theSecurity of Academic Information System by Gigih Forda Nama and Kumia Muludi [5]

In this work, only one 2FA method is implemented: SMS code, but the work does provide very in-depth analysis of usage statistics, providing a use case diagram, activity diagram, statistics on how fast and how many SMS codes are delivered and so on. Although it also does describe the implementation and usage process, it does not provide any examples of the actual code that was developed, maybe because it was supposed to be used for real academic information system.

3 Methodology

3.1 Background

This chapter provides relevant background information about different authentication factors and how they are used to implement different authentication methods.

Currently authentication factors are usually separated into 3 categories:

1. Knowledge factors - something you know (traditionally a password or PIN)
2. Possession factors - something you have (smartphone, ID card etc)
3. Inherence factors - something you are (biometrics, such as fingerprints or face)

3.1.1 Single factor authentication

For single factor authentication (SFA, also known as 1FA), only one of these factors needs to be authenticated. A password is a traditional method of single factor authentication which has been in use for a very long time. 1FA is very vulnerable to automated attacks, considering that people tend to reuse the same password in multiple places, or even use default passwords.

3.1.2 Two factor authentication

Two factor authentication (2FA) needs confirmation using 2 factors. Multi-factor authentication requires 2 or more methods of identity confirmation. Thus all 2FA is MFA, but not all MFA is 2FA.

According to Microsoft [6], MFA can block 99.9 percent of account compromise attacks. But at the same time according to DCMS Cyber Security Breaches Survey 2022 [7], for example, only about 37% of UK businesses require their users to use 2FA. Although according to Statista [8], this percentage is higher, at more than 50%, that still leaves about half of the companies without an effective cyber attack prevention method.

Since implementing two factor authentication using inherence factors, such as fingerprint scanner, is unfeasible for this project, and is unlikely to be used by most companies, 4 of the more common 2FA methods were chosen: email, SMS, authenticator app and backup codes.

3.2 Primary objective

Objective of this thesis is to research and implement different methods of two-factor authentication, both to increase the author's knowledge on the subject and to provide readers with actual implementation comparison, which is rarely done.

The primary aim of this thesis extends beyond a mere exploration of two-factor authentication (2FA) methods; it is a comprehensive endeavor to delve into the intricacies of various 2FA techniques, not only to augment the author's understanding of this crucial security domain but also to present readers with a nuanced and comparative analysis of real-world implementations. The significance of this research lies not only in its academic pursuit but also in its practical implications for enhancing cybersecurity practices.

The multifaceted objective of this thesis encompasses a twofold mission. Firstly, it seeks to expand the author's knowledge base by delving into the diverse landscape of 2FA methodologies. This involves a thorough investigation into the underlying principles, technologies, and mechanisms that constitute effective two-factor authentication systems. Through this scholarly exploration, the author aspires to gain a profound understanding of the theoretical foundations and practical intricacies of 2FA, contributing to the academic discourse surrounding cybersecurity.

Secondly, and perhaps more uniquely, the thesis aims to bridge the gap between theoretical knowledge and practical implementation by providing readers with a comparative analysis of actual 2FA implementations. This aspect of the research distinguishes it from many other studies in the field, as it endeavors to offer a tangible and hands-on perspective on the effectiveness and usability of different 2FA methods. This comparative analysis is designed to be a valuable resource for practitioners, researchers, and cybersecurity enthusiasts seeking insights into the real-world applicability and performance of diverse 2FA solutions.

The scarcity of comprehensive implementation comparisons in existing literature highlights the pioneering nature of this thesis. By filling this gap, it not only contributes to the academic body of knowledge but also addresses the practical needs of individuals and organizations navigating the complex landscape of cybersecurity. Through meticulous research and practical implementation, this thesis aspires to offer a holistic and insightful examination of 2FA methods, fostering a deeper understanding of their strengths, limitations, and real-world implications.

3.3 Work tools

The strategic selection of ASP.NET Core with C# for the backend and Vue.js for the frontend in this project was not only influenced by Author's familiarity with these technologies from their time at TalTech but also driven by the recognition of their popularity and effectiveness in the wider development community. This decision is underscored by the fact that both C# and Vue.js, the core components of the Author's tech stack, hold significant positions in industry rankings.

3.3.1 C#

C# stands out as the 5th most popular programming language according to the TIOBE index, highlighting its widespread adoption and the extensive developer community supporting it [9]. This popularity is a testament to the language's versatility, performance, and the ecosystem it offers for building robust backend solutions.

3.3.2 PostgreSQL

According to StackOverflow, PostgreSQL is currently the most popular database development environment, recently overtaking MySQL [10], making it an obvious choice for a database system for this project. Npgsql data provider was used to interact with the database.

3.3.3 Vue.js

On the frontend side, the decision to use Vue.js is reinforced by its standing as the second most popular JavaScript framework, according to AppsDevPro [11]. This not only speaks to the framework's popularity but also underscores its capabilities in enabling efficient and performant web development.

3.3.4 Visual Studio Code

In terms of the development environment, Visual Studio Code was the natural choice as an integrated development environment (IDE). Its feature-rich interface, extensive language support, and active community contribute to a seamless development experience, allowing the Author to focus on coding logic and functionality.

3.3.5 GitLab

To ensure a streamlined and collaborative development process, GitLab was employed as the version control system. This platform facilitated efficient progress tracking, collaborative code review, and a centralized repository for the project, promoting a well-organized and transparent workflow.

3.3.6 Information research

In the pursuit of comprehensive and up-to-date information for this project, I turned to Google, Google Scholar, and ChatGPT 3.5 for relevant research (no code generation functionality was used). The combination of these tools allowed the Author to explore a diverse range of resources, from general information on programming practices to more specialized insights. Google and Google Scholar served as valuable search engines, while ChatGPT 3.5 added a dynamic layer of interaction, enabling the Author to refine ideas, explore concepts, and gain contextual insights for a more informed decision-making process.

3.4 Process

1. Analysis and initial research

First, a search for similar works on Google scholar was done. While there were some that compared security or speed of actual usage of the two-factor authentication methods, none were found that compared the difference in implementation of the 2FA methods. Different 2FA methods were researched.

2. Development

Then the development process started, which involved using a lot of Google search. Parts of the project were based on the project that Author has done during the university studies, specifically login and registration methods and API. These parts were implemented first, and then Author moved on to implementing the two-factor authentication methods.

3. Analysis and comparison

Lastly, all implemented 2FA methods were analysed and compared, and then a conclusion was drawn based on the results.

4 Results

Result of this work is a website with registration and login/logout functions, single-factor authentication, and 4 implemented two-factor authentication methods: e-mail, SMS, authenticator app and backup codes. This chapter discusses which methods or services were chosen for each 2FA method and why they were chosen.

4.1 Backend

This chapter delves into the backend code of the project.

4.1.1 Registration and login

The phased approach to implementing two-factor authentication necessitates the initial establishment of single-factor authentication as a foundational step. Consequently, the preliminary stages of this project were dedicated to the implementation of simple registration and login operations, each comprising a singular method. These operations, being fundamental to user interaction, were designed to require only a username and a password for both registration and login activities.

In adherence to best security practices, the passwords entered by users are subjected to a robust hashing mechanism before being stored in the database. This precautionary measure ensures that sensitive user information remains secure, adding an extra layer of defence against unauthorised access or data breaches. (Figure 12)

Upon the successful execution of a login attempt, the backend system generates and issues a Json Web Token. This token serves as a digital authentication credential, signalling that the user has been successfully logged into the system. The use of JWTs not only enhances the security of the authentication process but also streamlines subsequent user interactions by providing a secure and efficient means of managing session information. (Figure 10, Figure 11, Figure 13)

The implementation of these foundational methods drew heavily upon the knowledge and skills acquired during the author's academic tenure at TalTech. Leveraging the insights gained from the educational process, the author navigated the intricacies of designing and implementing registration and login functionalities. The proficiency acquired during this initial phase laid a solid groundwork for the subsequent integration of two-factor authentication, which stands as the focal point of this project.

It's worth noting that while the specifics of single-factor authentication are essential for a comprehensive understanding of the authentication process, the emphasis of this work leans more towards the intricacies of the second factor. As such, the nuances of SFA, while crucial in their own right, are considered within the broader context of paving the way for the more intricate and sophisticated two-factor authentication mechanisms that will be explored in subsequent stages of this project. This deliberate approach ensures a systematic and thorough exploration of authentication processes, with each phase building upon the knowledge and foundations laid in the preceding stages.

4.1.2 Email

Email code two factor authentication method implementation consists of 6 methods, which can be divided into three pairs:

1. Enabling authentication

   1. First method takes the input email, saves it to the “temporary” (just a naming scheme, technically it is not temporary) column in the database, generates the authentication code, stores it in a database and sends it to the input email. (Figure 14)

   2. Second method checks the input authentication code and compares it to the code in a database. If codes are the same, boolean, referring to email 2FA being enabled, is set to true, and email is saved to a “permanent” column in the database. (Figure 15)

2. Using authentication

   1. First method generates the 2FA code, saves it to the database and sends it to the email in the “permanent” column. (Figure 16)

   2. Second method compares the input 2FA code to the one saved in the database, and if they are the same, returns a Json token, which signifies a successful login. (Figure 17)

3. Disabling authentication

1. First method generates the code, saves it to the database and sends it to the registered email. It has the same functionality as the first method in the previous pair, so this method just returns the method from the last pair, but is defined separately for visual clarity. (Figure 18)

2. Second method validates the input code, and if the code is correct, sets the user’s email field to “”, and the boolean corresponding to email 2FA being enabled to false. (Figure 19)

Method for sending an email, for clean code purposes, since it is used multiple times, was written separately. (Figure 20)

4.1.3 SMS

Methods used for SMS two-factor authentication are very similar in structure to email 2FA, with the only big difference being the code delivery method (Figure 27), and the fact that it takes a phone number as an input, not email. Thus SMS 2FA also consists of 6 methods with similar patterns, screenshots of which can also be viewed in Appendix 2.

Of course, there may be ways to combine the authentication enabling, validating and disabling methods into two methods total, but that type of code would be way less clean and not necessarily have less lines of code.

4.1.4 Authenticator app

Methods for adding/using authenticator app 2FA are significantly different from the previous two methods, since this one does not require the user to input any of their information, such as email or phone number. A library named GoogleAuthenticator (not officially affiliated with Google) [12] was used, which allowed to skip the process of having to implement the logic, which is used by most authenticators, by the author. This allowed to greatly cut down on the code length. Thus implementation of this authentication option consists of 4 methods total:

1. Enabling authentication

   1. Method that generates user account secret key, and the setup code and QR code for input/scan in the chosen mobile authenticator app (since most apps use the same algorithms for code generation and validation, most popular apps should work). (Figure 28)

   2. Method that validates the code that user inputs after scanning the QR code/typing in the setup code. If successful, sets the boolean for app 2FA being enabled to true. (Figure 29)

2. Using authentication

   1. Method that validates the code during the second factor of authentication during the login process, and returns JWT as a sign of success. (Figure 30)

3. Disabling authentication

   1. Method that validates the input code, and on success, sets the boolean for authenticator app 2FA being enabled to false. User’s account secret key does not need to be deleted. (Figure 31)

4.1.5 Backup codes

Backup code 2FA does not need to be enabled separately. It is automatically enabled as an option, when any other authentication method is enabled. All that is needed is to generate the backup 2FA codes and write them down somewhere.

Authentication using backup codes consists of 2 methods total:

1. Enabling authentication

   1. Method that generates the 2FA codes using a for loop and a cryptographically secure pseudo random number generation method available in ASP.NET Core called RandomNumberGenerator, and then hashes and stores this list of numbers to the database. (Figure 32)

2. Using authentication

   1. Method that validates the input code during the authentication process, and on success returns the JWT. (Figure 33)

3. Disabling authentication

Since in this project the availability of the backup code 2FA method changes whether the user has any other 2FA method enabled, it does not need a separate disabling function.

4.1.6 2FA code generation

Two-factor authentication code, also known as one-time password (OTP), is a code, which can be generated by a multitude of various methods, and is used as one of the authentication factors. Most important aspect of OTP is that it can only be used once, and becomes invalid after it is used, or when a new code is generated. In this project, this type of code is generated for email, sms, and backup code authentication methods.

For OTP generation, the first choice was a pseudo random number generator class, present natively in .Net, was used. The problem with that number generator is in the fact that it requires a seed. That seed has to be different each time, otherwise the generated code will be the same. If a seed is not provided, a timestamp is used by default. That means that if a hacker knows that the number generator is time-based, they can brute force the authentication code. Thus a decision was made to switch to a class that generates cryptographically secure random values, called RandomNumberGenerator, which is a lot harder to predict.

A cryptographically secure pseudo-random number generator (CSPRNG) is a type of random number generator that is designed to be suitable for use in cryptography and other applications where it is important to generate unpredictable and unbiased random numbers. Cryptographically secure random number generators are essential for various security-sensitive tasks, such as generating cryptographic keys, initialization vectors, and nonces.

Key characteristics of a cryptographically secure number generator:

1. Unpredictability**:** The output should be statistically unpredictable, even if an adversary has partial knowledge of the generator's internal state or previous outputs. This property is crucial for resisting various attacks.

2. Resistance to backtracking**:** Even if a hacker learns the current or past outputs, it should be computationally infeasible to reconstruct the internal state of the generator or predict future outputs.

3. While non cryptographically secure number generators tend to use simple system info for seeding the generator, such as system time, cryptographically secure generators use a combination of different information sources to constantly reseed the number generator, such as user input, system interruptions, disk I/O and so on, sometimes even together with information from some other device using an API [13].

Generated codes are additionally hashed for security and then stored in string format. There are standards for implementing one-time password algorithms [14] [15], so it can be implemented by an independent developer, but manually implementing something of that sort in this work is unfeasible.

On the other hand, authenticator apps, such as Google Authenticator (currently known simply as Authenticator), Microsoft Authenticator and so on use a different version of the one-time password, called time based one-time password (TOTP). In those apps, a new code is generated every 30 seconds, and the old code becomes invalid no matter whether it was used or not. But in this case the code can be used multiple times during the 30 second timer. Since this type of code generation generally is done on the app side, it does not need to be implemented on the backend of the website. But, QR code and an alternative app setup key do need to be generated and stored on the backend, using the generated user secret key. In this project the author used the Google.Authenticator package.

4.1.7 2FA code authentication

For email, sms and backup code authentication methods, the input code that comes from the front end is hashed and compared to the authentication code, stored in the backend of the web app. In case of a 2FA authenticator app authentication, the code is validated using input code together with a user account secret key.

In all methods (and also when no additional authentication methods are enabled yet) after successful authentication a Json Web Token (JWT) is generated and returned as a confirmation.

4.1.8 Hashing

Hashing is a process of converting input data (or 'message') of any length into a fixed-length string of characters, which is typically a hash value. The output, known as the hash code or hash digest, should be unique to the input data and appear random, making it difficult to reverse the process or recreate the original input from the hash value. Hash functions are commonly used in computer science, cryptography, and various applications where data integrity and security are important.

Here are some of the key characteristics of hash functions:

- Deterministic: For the same input, the hash function should always produce the same output.
- Efficient: It should be computationally efficient to compute the hash value for any given input.
- Fixed Output Length: The hash function generates a fixed-size output, regardless of the size of the input.
- Sensitivity to input changes: A small change in the input should result in a significantly different hash output. This property ensures that similar inputs don't produce similar hash values.
- Irreversibility: It should be computationally infeasible to reverse the process and obtain the original input from the hash value. Hash functions are not meant to be encryption functions.

Hash functions have various applications, including:

- Data Integrity: Hashing is used to verify the integrity of data. If the hash value of the original data matches the hash value calculated from the received data, it indicates that the data has not been altered.

- Password Storage: Instead of storing actual passwords, systems often store the hash of the passwords. During authentication, the system hashes the entered password and compares it with the stored hash. This is implemented in this project.
- Digital Signatures: Hash functions are a fundamental component of digital signatures, providing a compact representation of the signed data.
- Blockchain: Hash functions are popular in blockchain technology. Each block in a blockchain contains a hash value of the previous block, ensuring the entire blockchain's integrity. [16]

Commonly used hash functions include MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), SHA-256, and SHA-3. It's important to note that MD5 and SHA-1 are considered insecure for cryptographic purposes due to vulnerabilities, and SHA-256 or stronger hashes are recommended for security-sensitive applications. For this project SHA-256 hashing function was chosen.

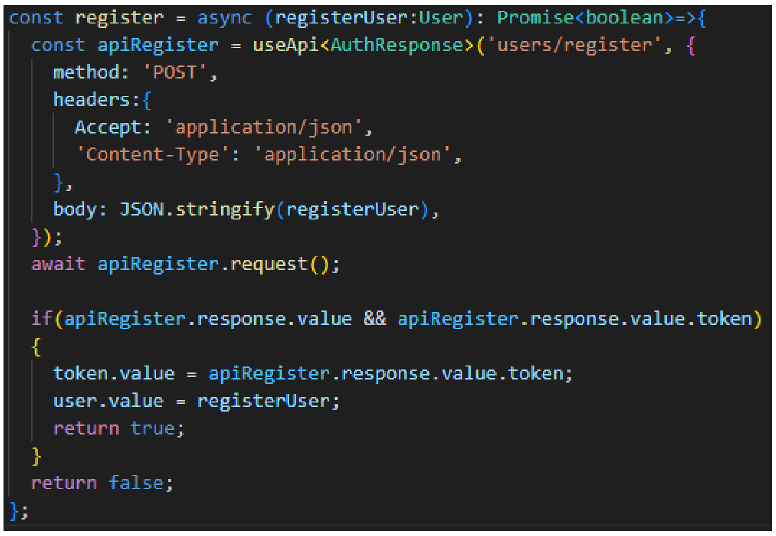
4.2 Frontend

Since most of the work, relevant for the research, is done in the backend of the web app, the frontend was kept quite simple. Vue.js framework was used for the frontend, together with the API for backend communication.

4.2.1 Application Programming Interface (API)

For the majority of operations an API was used to communicate with the backend via HTTP requests. Backend was written in C# in ASP.NET Core. Here is an example of an API request code used for the registration function of this project:

Figure 1



```
const register = async (registerUser:User): Promise<boolean>=>{
  const apiRegister = useApi<AuthResponse>('users/register', {
    method: 'POST',
    headers:{
      Accept: 'application/json',
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(registerUser),
  });
  await apiRegister.request();

  if(apiRegister.response.value && apiRegister.response.value.token)
  {
    token.value = apiRegister.response.value.token;
    user.value = registerUser;
    return true;
  }
  return false;
};
```

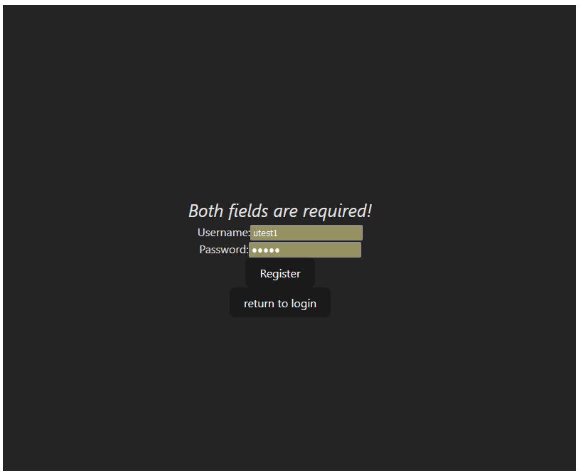
Code for useApi function is provided in Figure 34.

4.2.2 Vue pages

Vue pages consist of the usual HTML, with some additional vue specific functions, which help somewhat simplify the development process, and TypeScript scripts, which in this project were used for backend calls through the API. Following are examples of how the project looks visually:

- Registration page

Figure 2



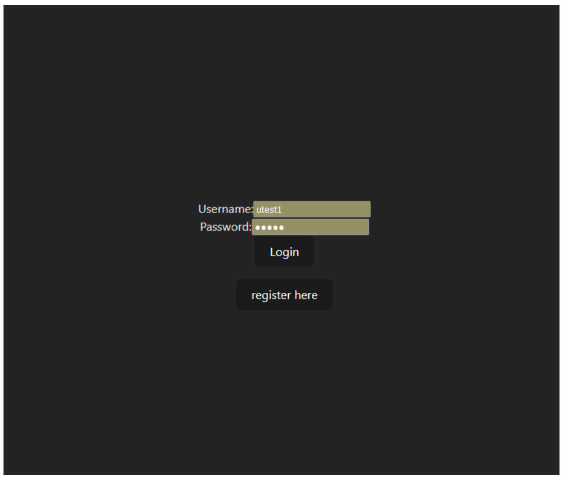
Both fields are required!

Username: utest1

Password: •••••

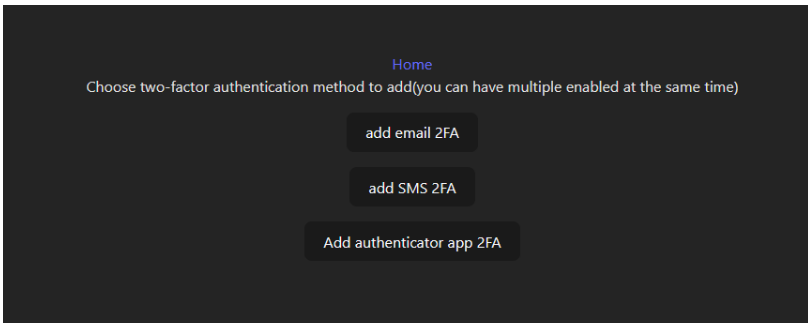
- Login page

Figure 3



- Choice of 2FA to enable

Figure 4



Choose two-factor authentication method to add(you can have multiple enabled at the same time)

- In this example, email 2FA will be enabled

Figure 5

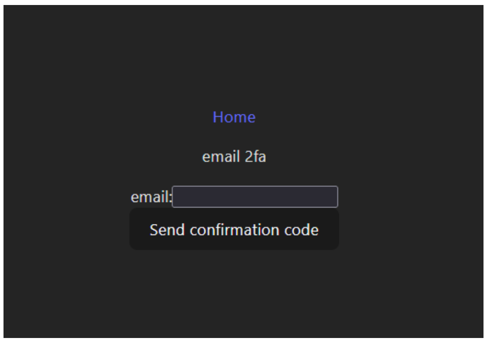
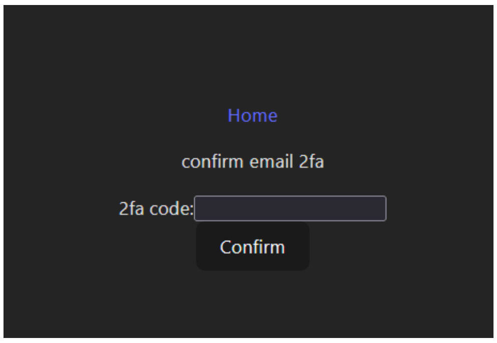
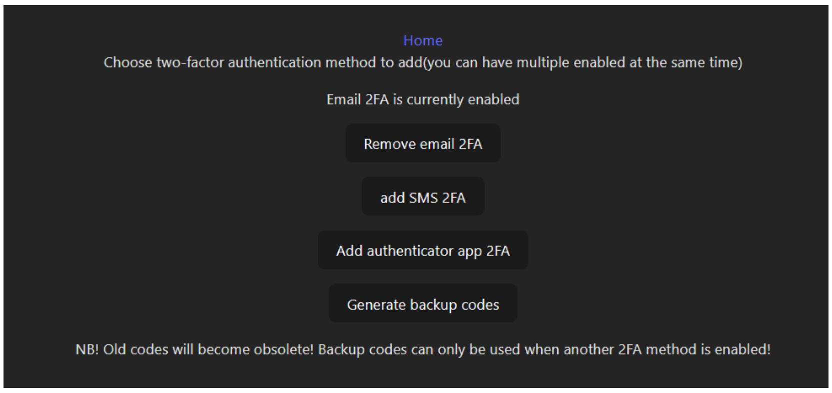


Figure 6



- Email 2FA is now enabled, and now backup codes can be generated if needed

Figure 7



Choose two-factor authentication method to add(you can have multiple enabled at the same time)

Email 2FA is currently enabled

NB! Old codes will become obsolete! Backup codes can only be used when another 2FA method is enabled!

- And here is the second factor of authentication during login

Figure 8

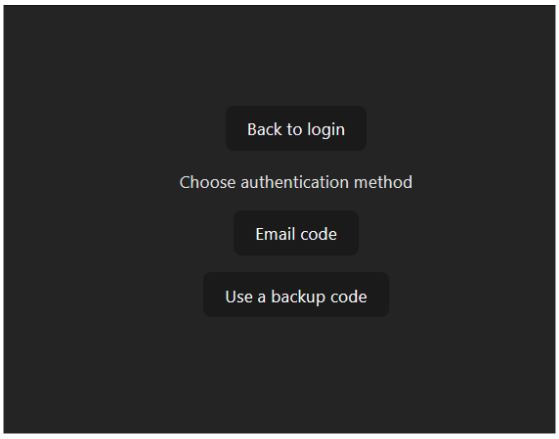
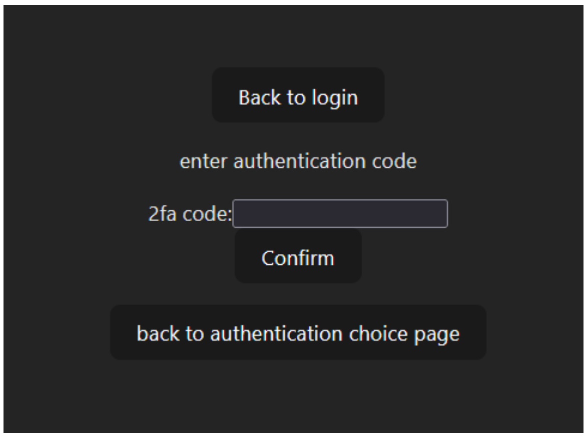


Figure 9


Back to login
enter authentication code
2fa code:
Confirm
back to authentication choice page

5 Analysis and conclusions

This chapter delves into specifics of the implementation of each two-factor authentication method, and draws conclusions based on the analysis.

5.1 Initial analysis

This chapter provides initial analysis/overview and comparison between the two-factor authentication methods.

5.1.1 Email through API service

While this method was not implemented due to the author being unable to register/get the service to work properly, an initial analysis can still be conducted.

1. What is required to get started:
   An account on the API service website.

2. Free or paid:
   Most give at least a 30 day free trial period, and then switch to pay-as-you-go. Some provide a fixed amount of free emails each month.

3. Dependency (on a service/services):
   If the service is down, your auth service is down too, and switching to another service quickly is not easy.

4. Tutorials:
   Usually provided by the service itself, although not always for all platforms.

5.1.2 Email through SMTP

Some of the information is specifically about using Google SMTP server, since that is what was used in this work.

1. What is required to get started:

   Google account, and an app password for the account.

2. Free or paid:

For Google SMTP users, the free tier with a standard Google account is 500 emails per day. To expand this limit, a Google Workspace account can be purchased, which starts at 6 dollars per day (price may vary), and allows to send up to 2000 emails per day. Even the free tier limit is a lot higher than what email API services and even other SMTP servers allow in their free tiers, which is usually around 1 to 3 thousand emails per month [17].

3. Dependency:

   In this specific case this method is dependent on the Google SMTP server. Of course, it is extremely unlikely for Google to experience an outage, but it is still something to not forget about. Compared to email API services, if an SMTP server experiences an outage, less code is needed to be altered if it is needed to urgently switch to another SMTP server, although both API services and SMTP servers would require registration.

4. Tutorials:

   Since this technology has been around for a long time, finding a tutorial online is quite easy for most platforms/frameworks.

5.1.3 SMS

Some of the information is specifically about using ASPSMS service, since that is what was used in this work.

1. What is required to get started:

   An account on the API service website (ASPSMS in this project's case).

2. Free or paid:

   ASPSMS gives a limited amount of test credits for the developers with no time limit (while similar services may offer a trial period), but after that the service and the majority of its competitors, such as Twilio and Vonage, switch to pay-as-you-go model, with prices usually ranging from 0.01 to 0.1 eur per SMS, depending on the service and location, with some exceptional locations, such as Afghanistan, reaching prices of up to 0.4 eur per SMS. No service was found that offered permanent free SMS service.

3. Dependency:

   As with email API services, availability of the SMS service depends on the uptime of the API service that is chosen for usage.

4. Tutorials:

   API services usually offer documentation on their API implementation. In the case of ASPSMS, aside from their own guides, it is also included in official Microsoft documentation on implementing SMS two-factor authentication [18].

5.1.4 Authenticator app

1. What is required to get started:

   Registration anywhere is not required for implementation, but may be needed to use some of the mobile authenticator apps.

2. Free or paid:

   In this project's case, a very popular (according to the fact that most ASP.NET Core tutorials found in Google related to this topic use it) free library called GoogleAuthenticator was used, which allowed generating setup codes and validating 2FA codes without using any external service and thus, totally free without any limits. And since the majority of libraries and authenticator apps use the same standard algorithm for authenticator setup, 2FA code generation and validation, they (libraries/apps) are very interchangeable, which offers a lot of flexibility.

3. Dependency:

   Since all of the work happens locally, this method is not dependent on any external API service, unless using one of those is a specific requirement.

4. Tutorials:

GoogleAuthenticator library has an abundance of implementation guides online.

5.1.5 Backup codes

This chapter is about local implementation of backup codes only, without using any external service.

1. What is required to get started:

   No registration required.

2. Free or paid:

   Free.

3. Dependency:

   Not dependent on any external service.

4. Tutorials:

   In author's case, no tutorial was required, just some basic C# knowledge. Simplest method to implement.

5.2 Backend code analysis and comparison

5.2.1 Email

When implementing both email and sms based two-factor authentication, the most crucial aspects are:

1. Generating the 2FA code
2. Sending it
3. Authenticating the code that comes from frontend

Since code generation and authentication is relevant for other methods too, it will be elaborated on in later chapters. As for sending the email, the first thought would be to use an external api. A research of different email api providers was conducted.

First email api where the author tried to register was Twilio SendGrid, which is considered one of the most popular email api providers [19]. But the author's account was shut down immediately after registration, no matter which email was used, taltech email included. And questions on why the accounts were closed received no substantial answer, only automated emails.

Second choice was Postmark, also one of the more popular email api providers. Author managed to register there using TalTech email, but then there were problems with sending emails from the TalTech server to gmail, so this method was also discarded. All other examined email api providers, such as mailgun and duo, offered only free trials for a month, which was unsuitable for the development.

Choice was made to use the oldest method of sending emails which is still in active use: Simple Mail Transfer Protocol (SMTP). Finding relevant information on SMTP implementation in ASP.NET Core was extremely easy [20] [21], with provided sources serving as an example, which can be attributed to its age. For actually sending the email, Google SMTP server was used. Aside from code, all that is needed to send an email this way is a gmail account and an app password, which can be generated separately on the google account management page.

5.2.2 SMS

For sms api providers, twilio is once again one of the most popular ones, but since the author did not manage to register there, using it was impossible. The choice fell on ASPSMS, which is suggested by a microsoft tutorial article [18]. Registration process was smooth, and the author quickly managed to receive test credits with no time limit. Microsoft tutorial was used as a base for implementing sms sender method.

5.2.3 Authenticator app

With authenticator apps, the code is generated on the said app side, so the code generation step does not apply here. But, something else has to be generated: QR Code that has to be scanned in the mobile authenticator app of choice, and its alternative, setup key. Since all authenticator API services/libraries use the same code generation algorithms, which comply with the Internet Engineering Task Force (IETF) standard RFC 6238 [15], any option can be chosen. In this case the GoogleAuthenticator library was chosen [12].

5.2.4 Backup codes

Since a for loop is used for backup code generation, which technically does not increase or decrease the code line length depending on the amount of loops, actual executed code length may vary significantly, depending on how many codes are needed to be generated. Thus, code length for three scenarios is provided, in which a for loop will be counted as writing the code from inside of the loop multiple times:

1. Best case, when only 1 code is needed to be generated
2. Average case, 6 codes
3. Worst case, 10 codes

5.2.5 Comparison table

This chapter provides a table, which lists all 2FA methods' code length in lines of code

Every time a new code is generated and saved to the database it is hashed beforehand.

For cleaner code purposes code generation, hashing and code delivery methods are written into separate methods, but for the purposes of counting lines of code they will be counted together with the methods that use them.

All of the methods that validate the 2FA code, aside from the methods that use the GoogleAuthenticator library, generate a new code after successful validation as a safety mechanism, to prevent the usage of the same code twice.

Also, all of the authentication options on a successful authentication return a newly generated Json web token. While this is done in a separate method, this method is also counted as a part of the login validation method.

Important! Since the first method in the "disabling authentication" sequence for email and SMS authentication methods simply calls the first method from "using authentication" sequence, only having a different name for clarity purposes, they can be said to be the same, and thus when the total code length is counted, this method will not be counted towards total code length.

Length is in lines of code

Table 1

	Enabling authentication		Using authentication		Disabling authentication		
Authentication option	M1L	M2L	M1L	M2L	M1L	M2L	TL
email	15+g+h+s1=49	14+g+h=28	11+g+h+s1=45	10+g+h+json=35	45	13+g+h=27	172
sms	15+g+h+s2=46	14+g+h=28	11+g+h+s2=42	10+g+h+json=35	42	13+g+h=27	168
Authenticator app	22	14	12+json=23	-	13	-	72
Backup codes best case	21+g+h=35	18+h+json=39	-	-	-	-	73
Backup codes average case	61+6*g+6*h=151	39	-	-	-	-	190
Backup codes worst case	93+10*g+10*h= 233	39	-	-	-	-	272

Legend:

M1L: method 1 length

M2L: method 2 length

TL: total length

g: separate 2FA code generation method, length = 4

h: hashing function, length = 10

json: function that generates Json Web Token, length = 11

s1: method for sending an email, length = 20

s2: method for sending an SMS, length = 17

“-“: no method

This table shows that even though backup codes take the least amount of methods to implement, implementing authenticator app takes the least lines of code, being 72 lines total. This was largely influenced by the GoogleAuthenticator library (not officially affiliated with Google), without which the implementation of authenticator app 2FA could have been the biggest in terms of lines of code. Backup code generation with 1 code only is in second place at 73 lines, but if more codes are needed then the amount of code lines increases significantly, with the worst case scenario being at 272 lines. Email and SMS authentication options stand in the middle, with 172 and 168 lines of code respectively. Average case of the backup code 2FA method is slightly bigger, with 190 code lines. Sizes of all of these 2FA methods more than 2 times the code line amount needed for authenticator app 2FA implementation.

With this data a conclusion can be reached, that authenticator app 2FA, using GoogleAuthenticator library, together with the best case scenario for backup code 2FA (which consists of generating only 1 backup code) are the methods that can be implemented the fastest, and also potentially the easiest, since implementation of these 2FA options does not require the developer to register in any service, which may prove difficult for some developers. Of course, technically, other backup code 2FA scenarios have the same code length purely from the perspective of writing code, but execution speed of a for loop 1 time and 10 times will definitely vary, and can be compared to writing the same code multiple times.

5.3 Frontend analysis

This chapter provides analysis and comparison of different 2FA methods' vue pages. Since the frontend part of the application was kept very simple, there is not that much significant information to be said.

5.3.1 API

Api requests for all authentication methods were POST requests ,which contained the body with user information, relevant to specific operation, for example the username and SMS code to validate. No GET requests are performed, since it is not recommended for GET requests to have a body [22], and it is inadvisable and unsafe to put user information in a url [23].

5.3.2 Email and SMS pages

Enabling email and SMS 2FA is a 2 step process:

1. Entering an email/phone number
2. Entering the code that was sent to the said email/phone number.

It is possible to combine the 2 steps into one page in Vue.js, for example with conditional rendering using v-if directive, but for simplicity sake it was done with two pages. Email page has input validation, built into HTML. Validating phone numbers is not that easy. HTML does have a built in data type for phone numbers, but for it to validate, a pattern needs to be specified. Meaning, HTML phone number validation may only allow phone numbers of one specific length at the time. Thus, it is better to validate the phone number on the backend of the application. In this project's case, validation was done in the HTML, allowing only for 7-digit numbers with Estonian country code.

5.3.3 Authenticator app

Enabling authenticator app 2FA is also a 2 step process:

1. Scanning the QR code/entering the setup code into the mobile app
2. Entering the code from the app into the input field on the website.

Since only one of these steps is done on the website, only 1 page is needed, which contains both the QR code/setup code and the input field for the OTP. One thing to keep in mind is that the QR code, generated by the GoogleAuthenticator library, is in plain text, not a .jpg or .png format.

5.3.4 Backup codes

Backup code generation does not require the user to input any of their information, just the press of a button, which brings the user to the page with the newly generated codes, thus it also consists of only 1 page.

5.3.5 Authentication process

Visually, the process of two-factor authentication during login is the same for all authentication methods, with the major differences being in the backend.

5.4 Storage of sensitive information

This chapter explains a problem that is present for most authentication methods, which require input of sensitive information to use them, so in this case it is relevant for all methods, aside from backup codes.

5.4.1 SMTP and SMS

Sending a code to an email through SMTP using google server requires the developer to present a google email address, and an app password for said email account. Sending a SMS through ASPSMS requires the developer to input the user key and API key they got from their account on the ASPSMS website. The problem with this is the fact that this information cannot be hashed, since it has to be presented in plain text. There are ways to mitigate the risk, for example by using Secret Manager, provided by ASP.NET Core. Secret manager allows you to store valuable information in a secret file, which is hidden on the developer’s computer. A better practice for bigger companies would be to use a separate encrypted database with valuable information.

5.4.2 Authenticator app

Authenticator app, on the other hand, has a slightly different problem. In this case, sensitive information is the user account secret key, which unlike the developer information, is generated separately for every user. This key also cannot be hashed, since it is later used to validate the code user inputs during two-factor authentication, and in this case using Secret Manager is impractical, because Secrets can only be set using .Net Command Line Interface (CLI) or manually. Which creates a problem, since if the hacker gains access to the secret key, they can generate the same 2FA code as the user. But luckily passwords can be hashed, which can still keep the account safe even if the user account secret key is compromised.

5.4.3 Code generation seed

Also, depending on the code generation method used, generation seed may also need to be stored safely, since using the same seed means generating the same codes, which is a big vulnerability. Storing the OTP code itself on the database may also be not the best option, since it puts additional strain on the database, and may be unsafe in case of unauthorised access to the database.

5.4.4 Password

All in all, the biggest security risk is not something like an account secret key leak, or hacker gaining access to the database that stores OTP codes, but a password leak. If a hacker gains access to a secret key, code generation seed, or even OTP code itself, the worst that will happen is that it may help the hacker log into an account on only one specific website. But if the passwords are leaked, the problem gets a lot worse, since a lot of people tend to reuse the same password in more than one place. As an example, according to Google, two-thirds of Americans use the same password on multiple websites [24]. Thus, keeping the passwords secure is a much higher concern than 2FA related data, although it does not mean that secure storage of 2FA related information can be forgotten about.

6 Summary

Choosing a suitable 2FA method to implement can be a difficult process, since there are so many options. At the same time it is hard to find actual implementation comparisons of different 2FA methods. This work was aimed at partially solving that problem, by providing in-depth comparison of multiple 2FA options.Not all programming languages may offer cryptographically secure code generation

For this project, ASP.NET Core was chosen for the back-end, and Vue.js for the front-end, since using these frameworks allowed the author to utilise knowledge acquired during the study in the university

The result is a website, which allows users to register, login, enable their choice of two-factor authentication (multiple can be enabled at the same time), and then use said 2FA method for authentication during the login process. The 2FA options that were implemented are: email code, SMS code, authenticator app (such as Google Authenticator) and backup codes, which can only be used when another 2FA method is enabled.

As for the results, the authenticator app 2FA has the least amount of code lines, which can definitely be attributed to the usage of a third-party library, together with the backup code 2FA. These methods are also more accessible than, for example, using an API service provider for sending the code through SMS or email, since they do not require registration. And finally, the reason why authenticator app can be considered a better option than all other methods that were implemented, is the fact that the code generation does not need to be performed in the back-end, and it is always easier to leave the difficult part to the party that has already done it according to the industry standards, than trying to implement it independently.

References

[1] N. Tambe and A. Jain, “Top Website Statistics For 2023,” Forbes, 8 Nov 2023. [Online]. Available: https://www.forbes.com/advisor/in/business/software/website-statistics/.

[2] R. Authors, “Almost Half of Companies Do Not Use MFA, 2022 Report Finds,” 10 May 2022. [Online]. Available: https://rublon.com/blog/half-companies-do-not-use-mfa-2022/.

[3] T. S. J. D. J. A. J. C. a. K. S. B. Y. U. Ken Reese, “A Usability Study of Five Two-Factor Authentication Methods,” 12-13 Aug 2019. [Online]. Available: https://www.usenix.org/conference/soups2019/presentation/reese.

[4] F. V. NIKLAS TELLINI, “Selecting and implementing a two-,” 2017. [Online]. Available: https://people.kth.se/~maguire/DEGREE-PROJECT-REPORTS/170531-Niklas_Tellini_and_Fredrik_Vargas.pdf.

[5] K. M. Gigih Forda Nama, “Implementation of two-factor authentication (2FA) to enhance the security of academic information system,” 2018. [Online]. Available: https://www.researchgate.net/publication/326057520_Implementation_of_two-factor_authentication_2FA_to_enhance_the_security_of_academic_information_system.

[6] M. Maynes, “One simple action you can take to prevent 99.9 percent of attacks on your accounts,” Microsoft, 20 Aug 2019. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2019/08/20/one-simple-action-you-can-take-to-prevent-99-9-percent-of-account-attacks/.

[7] M. Ell and R. Gallucci, “Cyber Security Breaches Survey 2022,” 11 July 2022. [Online]. Available: https://www.gov.uk/government/statistics/cyber-security-breaches-survey-2022/cyber-security-breaches-survey-2022.

[8] Statista Research Department, “Companies' cybersecurity measures adopted since COVID-19 in the United States, United Kingdom, and Japan in 2021,” Oct 2021. [Online]. Available: https://www.statista.com/statistics/1277272/employers-cybersecurity-measures-adopted-us-uk-japan/.

[9] TIOBE, “TIOBE Index for December 2023,” Dec 2023. [Online]. Available: https://www.tiobe.com/tiobe-index/.

[10] StackOverflow, “2023 Developer Survey,” May 2023. [Online]. Available: https://survey.stackoverflow.co/2023/.

[11] AppsDevPro , “Vue.js Vs React: Who Will Be the Winner in 2024?,” AppsDevPro, 18 Dec 2023. [Online]. Available: https://www.appsdevpro.com/blog/vue-vs-react/.

[12] B. Potter, “GoogleAuthenticator,” [Online]. Available: https://github.com/BrandonPotter/GoogleAuthenticator.

[13] cryptobook.nakov.com, “Secure Random Generators,” 2019. [Online]. Available: https://cryptobook.nakov.com/secure-random-generators.

[14] Network Working Group, “HOTP: An HMAC-Based One-Time Password Algorithm,” Dec 2005. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4226.

[15] Internet Engineering Task Force, “TOTP: Time-Based One-Time Password Algorithm,” May 2011. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6238.

[16] K. Rahul, “Hash Function,” [Online]. Available: https://www.wallstreetmojo.com/hash-function/.

[17] R. Khan, “10 Best Free SMTP Servers for Transactional Emails in 2024,” 10 Dec 2023. [Online]. Available: https://www.emailvendorselection.com/free-smtp-servers/.

[18] Microsoft, “Two-factor authentication with SMS in ASP.NET Core,” 11 Jan 2022. [Online]. Available: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/2fa?view=aspnetcore-1.1.

[19] L. Shipton, “Best Email APIs for Developers,” 1 Aug 2023. [Online]. Available: https://www.abstractapi.com/guides/best-email-apis.

[20] P. Malek, “Send and Receive Emails in ASP.NET C#,” 18 Oct 2022. [Online]. Available: https://mailtrap.io/blog/send-email-in-asp-net-c-sharp/.

[21] StackOverflow, “How to send email in ASP.NET C#,” [Online]. Available: https://stackoverflow.com/questions/18326738/how-to-send-email-in-asp-net-c-sharp.

[22] baeldung, “Why an HTTP Get Request Shouldn’t Have a Body,” 8 June 2023. [Online]. Available: https://www.baeldung.com/http-get-with-body.

[23] HttpWatch, “How Secure Are Query Strings Over HTTPS?,” 20 Feb 2009. [Online]. Available: https://blog.httpwatch.com/2009/02/20/how-secure-are-query-strings-over-https/.

[24] Google / Harris Poll, “The United States of P@ssw0rd$,” Oct 2019. [Online]. Available: https://storage.googleapis.com/gweb-uniblog-publish-prod/documents/PasswordCheckup-HarrisPoll-InfographicFINAL.pdf.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I, Eric Tamm

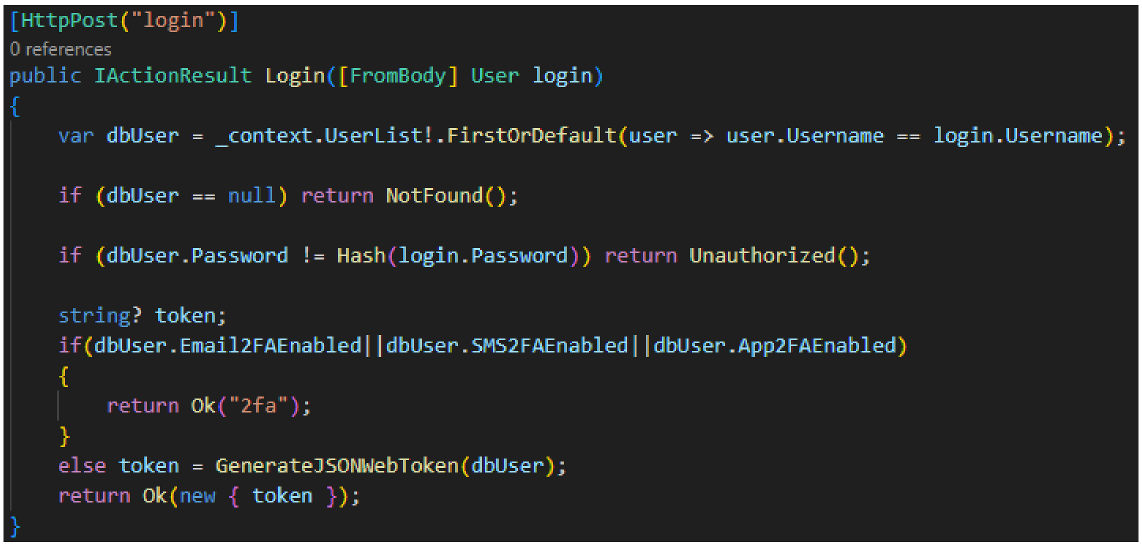
1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Implementation and comparison of two-factor authentication methods”, supervised by Viljam Puusep
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

03.01.2024

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

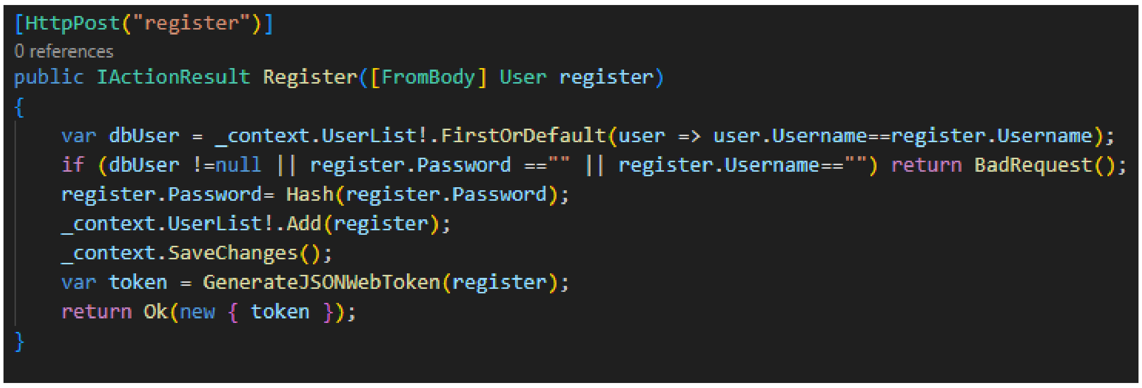
Appendix 2 – Code and website screenshots

Figure 10



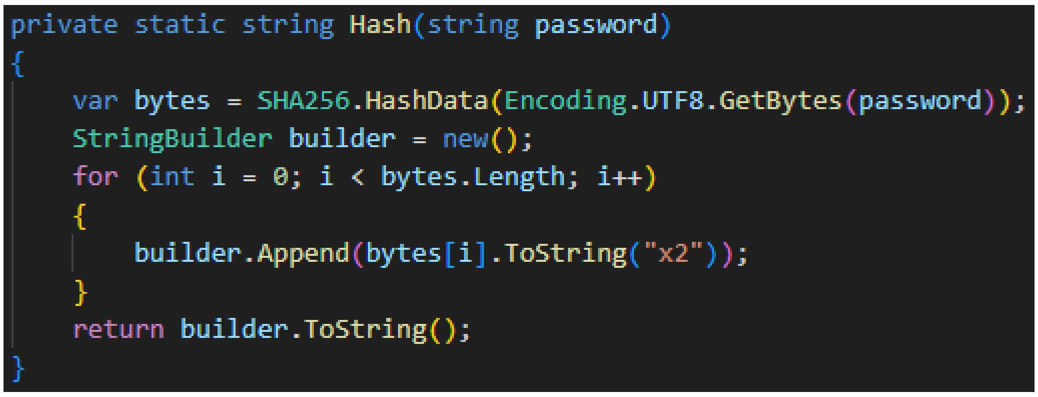
```
[HttpPost("login")]
0 references
public IActionResult Login([FromBody] User login)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == login.Username);

    if (dbUser == null) return NotFound();

    if (dbUser.Password != Hash(login.Password)) return Unauthorized();

    string? token;
    if(dbUser.Email2FAEnabled||dbUser.SMS2FAEnabled||dbUser.App2FAEnabled)
    {
        return Ok("2fa");
    }
    else token = GenerateJSONWebToken(dbUser);
    return Ok(new { token });
}
```

Figure 11



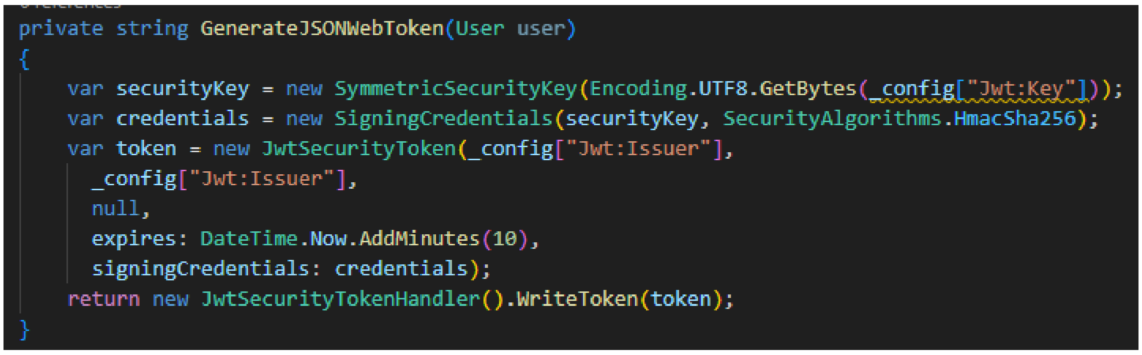
```
[HttpPost("register")]
0 references
public IActionResult Register([FromBody] User register)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username==register.Username);
    if (dbUser !=null || register.Password =="" || register.Username=="") return BadRequest();
    register.Password= Hash(register.Password);
    _context.UserList!.Add(register);
    _context.SaveChanges();
    var token = GenerateJSONWebToken(register);
    return Ok(new { token });
}
```

Figure 12



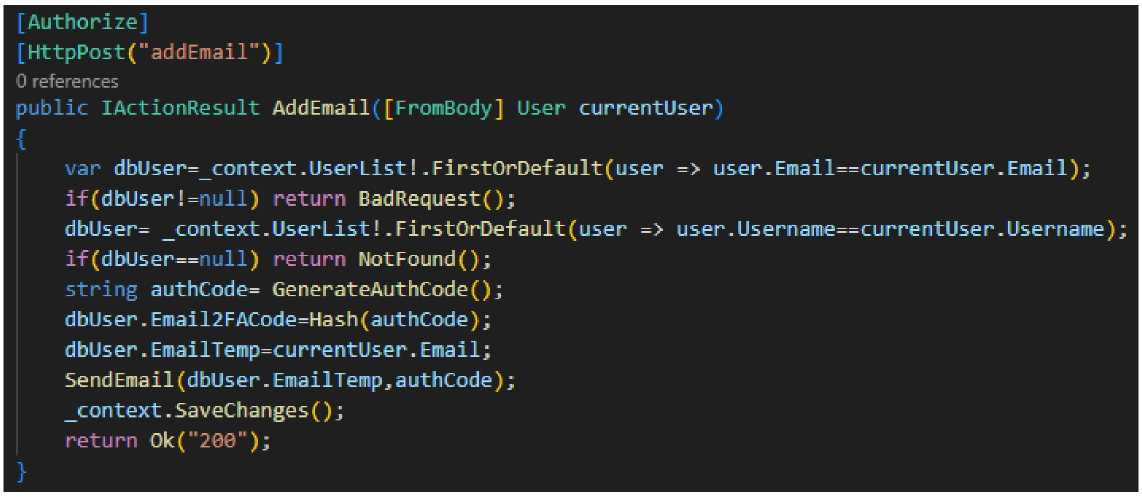
```
private static string Hash(string password)
{
    var bytes = SHA256.HashData(Encoding.UTF8.GetBytes(password));
    StringBuilder builder = new();
    for (int i = 0; i < bytes.Length; i++)
    {
        builder.Append(bytes[i].ToString("x2"));
    }
    return builder.ToString();
}
```

Figure 13



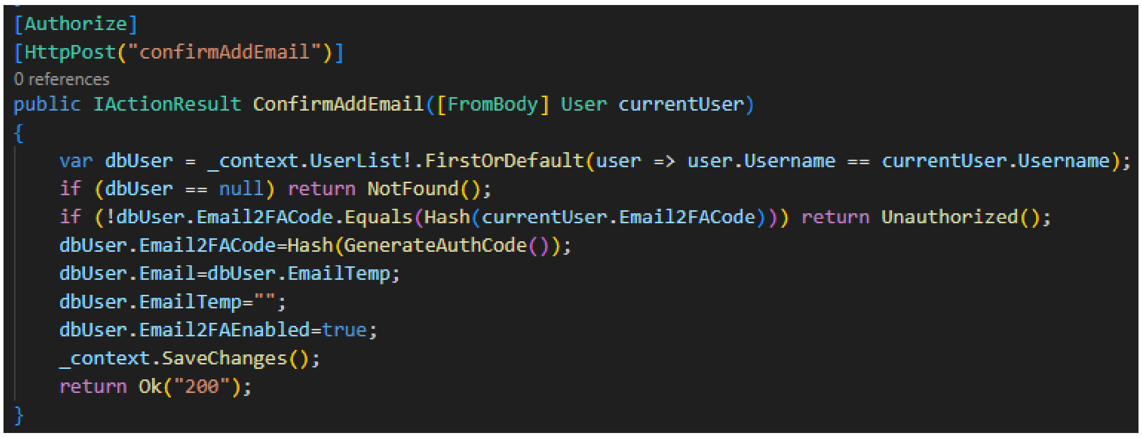
```
private string GenerateJSONWebToken(User user)
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken(_config["Jwt:Issuer"],
      _config["Jwt:Issuer"],
      null,
      expires: DateTime.Now.AddMinutes(10),
      signingCredentials: credentials);
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Figure 14



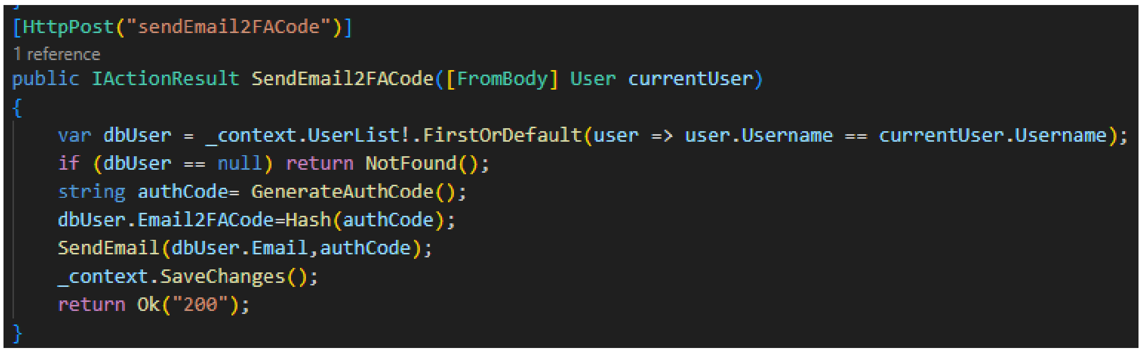
```
[Authorize]
[HttpPost("addEmail")]
0 references
public IActionResult AddEmail([FromBody] User currentUser)
{
    var dbUser=_context.UserList!.FirstOrDefault(user => user.Email==currentUser.Email);
    if(dbUser!=null) return BadRequest();
    dbUser= _context.UserList!.FirstOrDefault(user => user.Username==currentUser.Username);
    if(dbUser==null) return NotFound();
    string authCode= GenerateAuthCode();
    dbUser.Email2FACode=Hash(authCode);
    dbUser.EmailTemp=currentUser.Email;
    SendEmail(dbUser.EmailTemp,authCode);
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 15



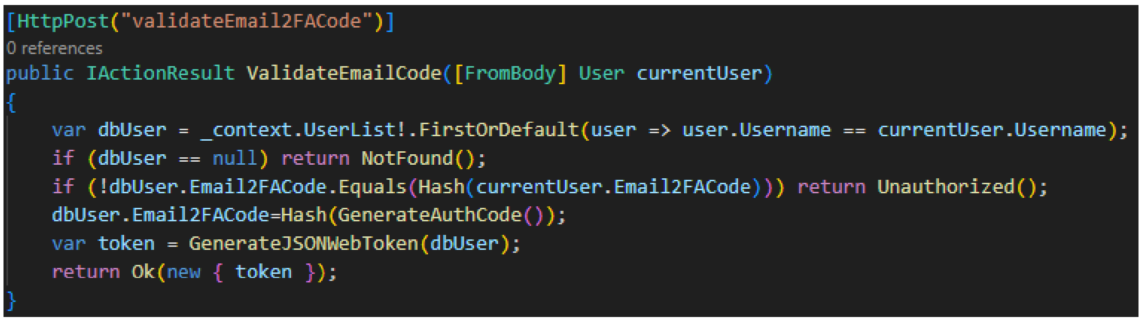
```
[Authorize]
[HttpPost("confirmAddEmail")]
0 references
public IActionResult ConfirmAddEmail([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if (!dbUser.Email2FACode.Equals(Hash(currentUser.Email2FACode))) return Unauthorized();
    dbUser.Email2FACode=Hash(GenerateAuthCode());
    dbUser.Email=dbUser.EmailTemp;
    dbUser.EmailTemp="";
    dbUser.Email2FAEnabled=true;
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 16



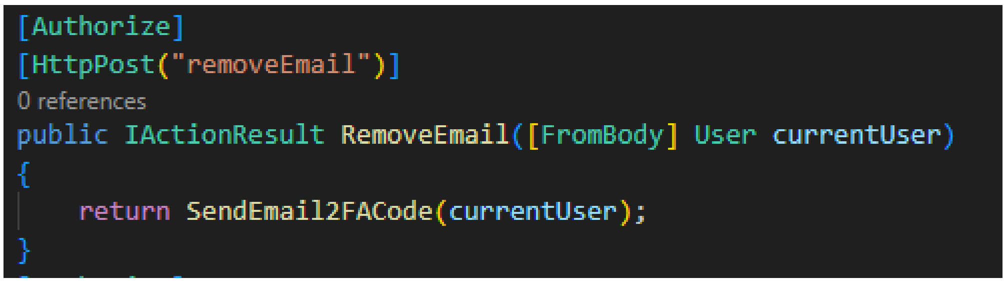
```
[HttpPost("sendEmail2FACode")]
1 reference
public IActionResult SendEmail2FACode([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    string authCode= GenerateAuthCode();
    dbUser.Email2FACode=Hash(authCode);
    SendEmail(dbUser.Email,authCode);
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 17



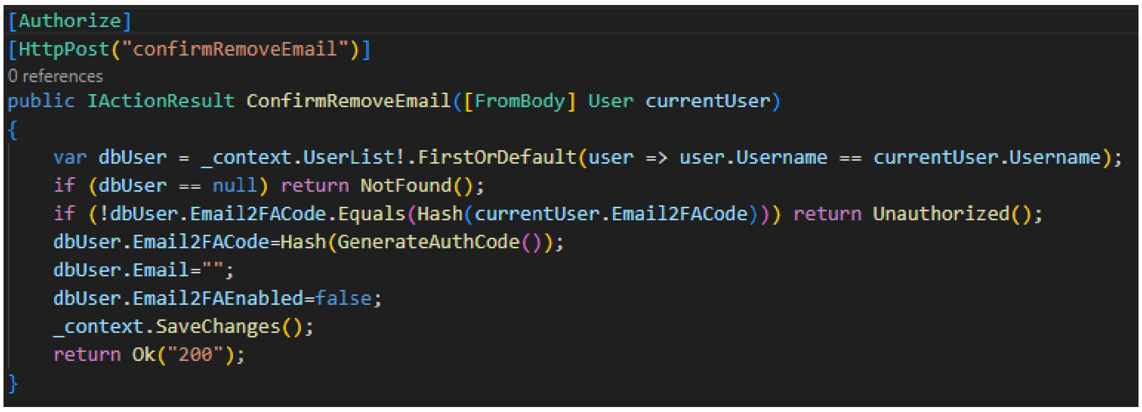
```
[HttpPost("validateEmail2FACode")]
0 references
public IActionResult ValidateEmailCode([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if (!dbUser.Email2FACode.Equals(Hash(currentUser.Email2FACode))) return Unauthorized();
    dbUser.Email2FACode=Hash(GenerateAuthCode());
    var token = GenerateJSONWebToken(dbUser);
    return Ok(new { token });
}
```

Figure 18



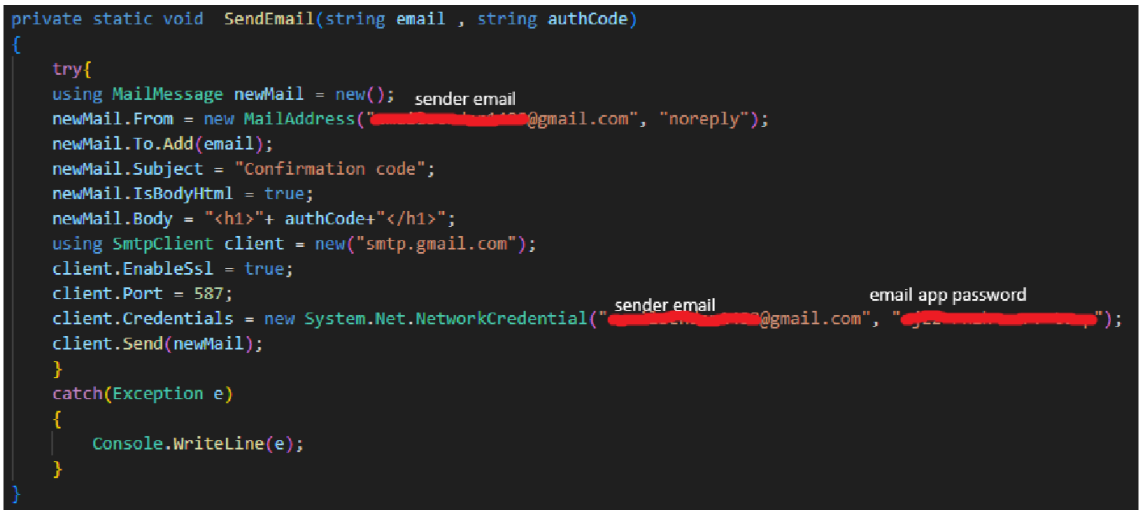
```
[Authorize]
[HttpPost("removeEmail")]
0 references
public IActionResult RemoveEmail([FromBody] User currentUser)
{
    return SendEmail2FACode(currentUser);
}
```

Figure 19



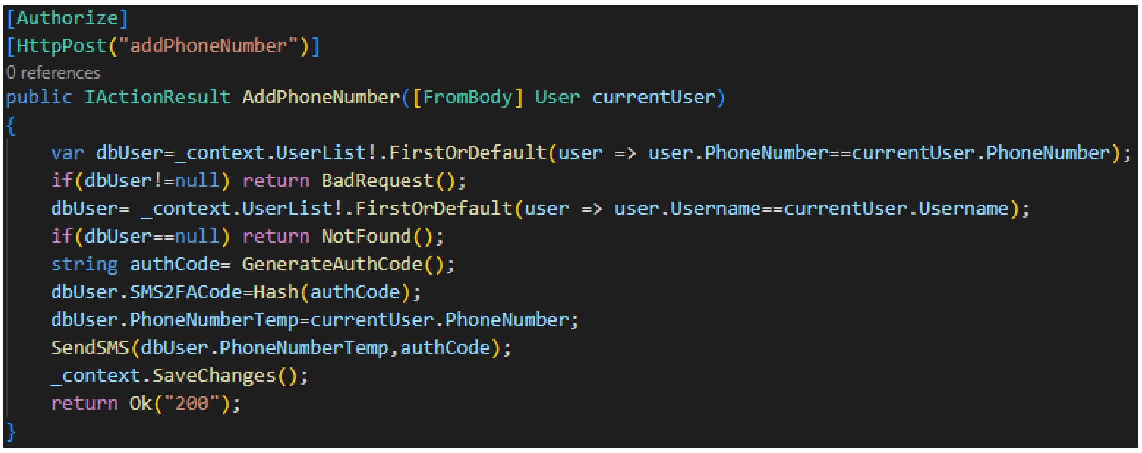
```
[Authorize]
[HttpPost("confirmRemoveEmail")]
0 references
public IActionResult ConfirmRemoveEmail([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if (!dbUser.Email2FACode.Equals(Hash(currentUser.Email2FACode))) return Unauthorized();
    dbUser.Email2FACode=Hash(GenerateAuthCode());
    dbUser.Email="";
    dbUser.Email2FAEnabled=false;
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 20



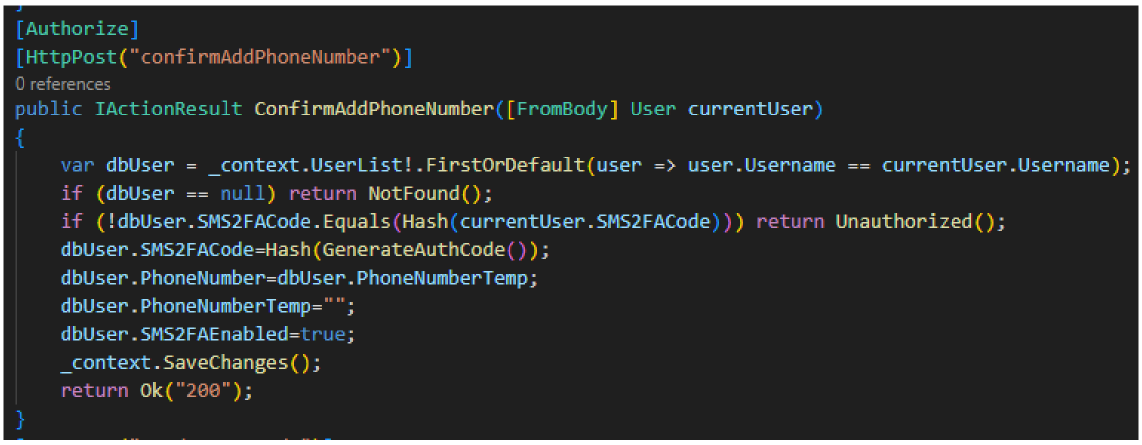
```
private static void  SendEmail(string email , string authCode)
{
    try{
    using MailMessage newMail = new();
    newMail.From = new MailAddress("[redacted]@gmail.com", "noreply");
    newMail.To.Add(email);
    newMail.Subject = "Confirmation code";
    newMail.IsBodyHtml = true;
    newMail.Body = "<h1>"+ authCode+"</h1>";
    using SmtpClient client = new("smtp.gmail.com");
    client.EnableSsl = true;
    client.Port = 587;
    client.Credentials = new System.Net.NetworkCredential("[redacted]@gmail.com", "[redacted]");
    client.Send(newMail);
    }
    catch(Exception e)
    {
        Console.WriteLine(e);
    }
}
```

Figure 21



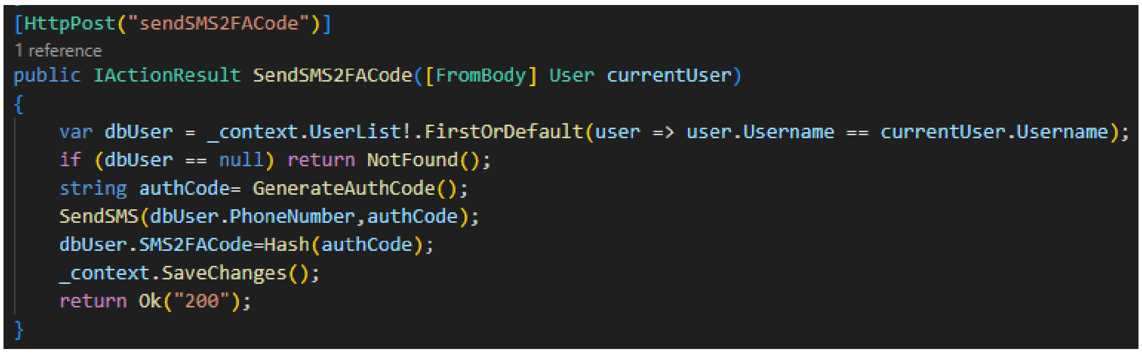
```
[Authorize]
[HttpPost("addPhoneNumber")]
0 references
public IActionResult AddPhoneNumber([FromBody] User currentUser)
{
    var dbUser=_context.UserList!.FirstOrDefault(user => user.PhoneNumber==currentUser.PhoneNumber);
    if(dbUser!=null) return BadRequest();
    dbUser= _context.UserList!.FirstOrDefault(user => user.Username==currentUser.Username);
    if(dbUser==null) return NotFound();
    string authCode= GenerateAuthCode();
    dbUser.SMS2FACode=Hash(authCode);
    dbUser.PhoneNumberTemp=currentUser.PhoneNumber;
    SendSMS(dbUser.PhoneNumberTemp,authCode);
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 22



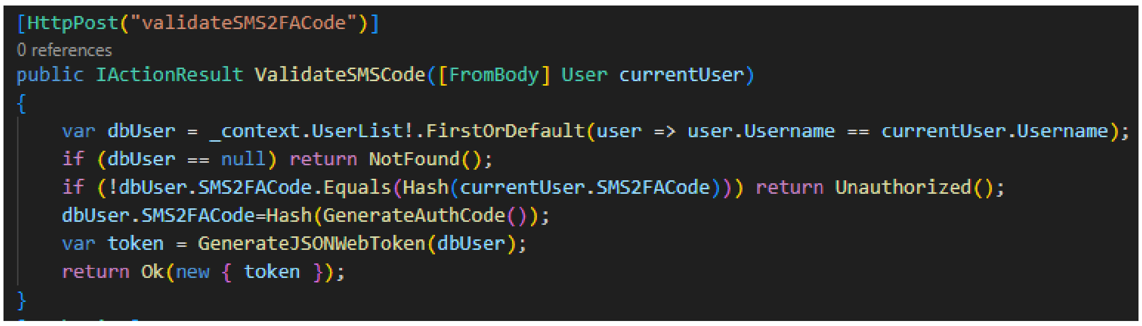
```
[Authorize]
[HttpPost("confirmAddPhoneNumber")]
0 references
public IActionResult ConfirmAddPhoneNumber([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if (!dbUser.SMS2FACode.Equals(Hash(currentUser.SMS2FACode))) return Unauthorized();
    dbUser.SMS2FACode=Hash(GenerateAuthCode());
    dbUser.PhoneNumber=dbUser.PhoneNumberTemp;
    dbUser.PhoneNumberTemp="";
    dbUser.SMS2FAEnabled=true;
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 23



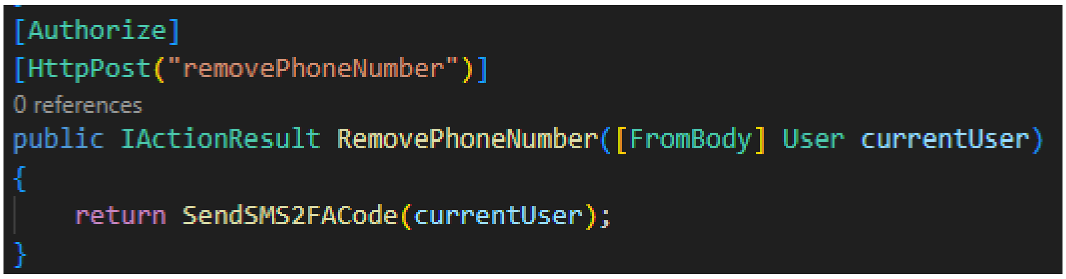
```
[HttpPost("sendSMS2FACode")]
1 reference
public IActionResult SendSMS2FACode([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    string authCode= GenerateAuthCode();
    SendSMS(dbUser.PhoneNumber,authCode);
    dbUser.SMS2FACode=Hash(authCode);
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 24



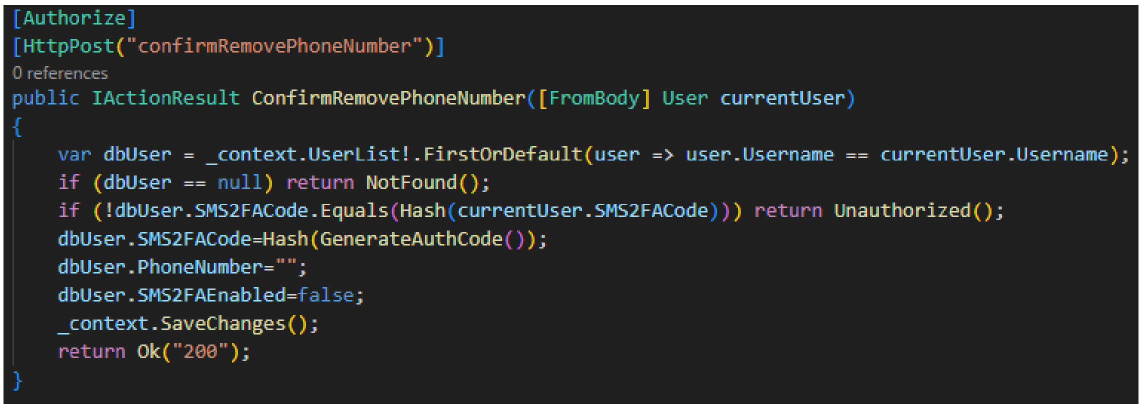
```
[HttpPost("validateSMS2FACode")]
0 references
public IActionResult ValidateSMSCode([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if (!dbUser.SMS2FACode.Equals(Hash(currentUser.SMS2FACode))) return Unauthorized();
    dbUser.SMS2FACode=Hash(GenerateAuthCode());
    var token = GenerateJSONWebToken(dbUser);
    return Ok(new { token });
}
```

Figure 25



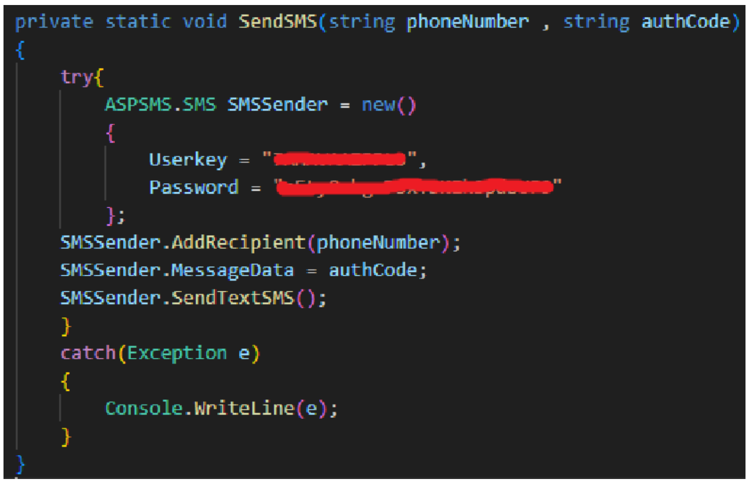
```
[Authorize]
[HttpPost("removePhoneNumber")]
0 references
public IActionResult RemovePhoneNumber([FromBody] User currentUser)
{
     return SendSMS2FACode(currentUser);
}
```

Figure 26



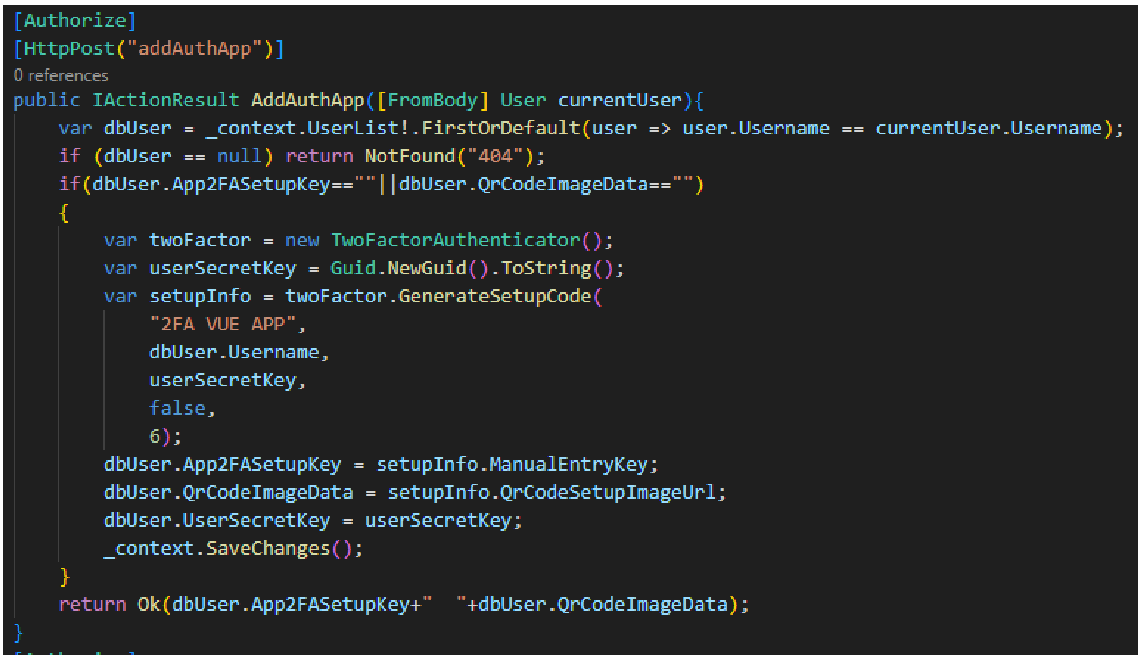
```
[Authorize]
[HttpPost("confirmRemovePhoneNumber")]
0 references
public IActionResult ConfirmRemovePhoneNumber([FromBody] User currentUser)
{
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if (!dbUser.SMS2FACode.Equals(Hash(currentUser.SMS2FACode))) return Unauthorized();
    dbUser.SMS2FACode=Hash(GenerateAuthCode());
    dbUser.PhoneNumber="";
    dbUser.SMS2FAEnabled=false;
    _context.SaveChanges();
    return Ok("200");
}
```

Figure 27



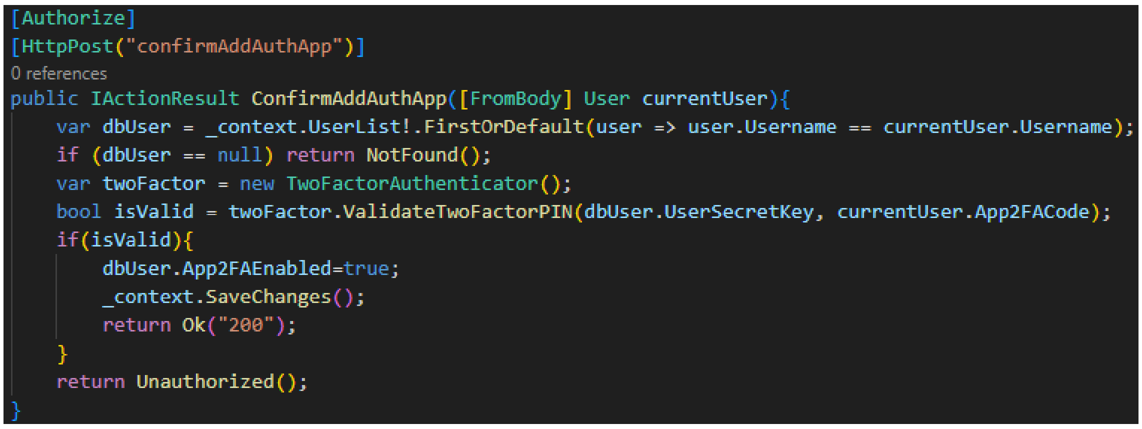
```
private static void SendSMS(string phoneNumber , string authCode)
{
    try{
        ASPSMS.SMS SMSSender = new()
        {
            Userkey = "[redacted]",
            Password = "[redacted]"
        };
    SMSSender.AddRecipient(phoneNumber);
    SMSSender.MessageData = authCode;
    SMSSender.SendTextSMS();
    }
    catch(Exception e)
    {
        Console.WriteLine(e);
    }
}
```

Figure 28



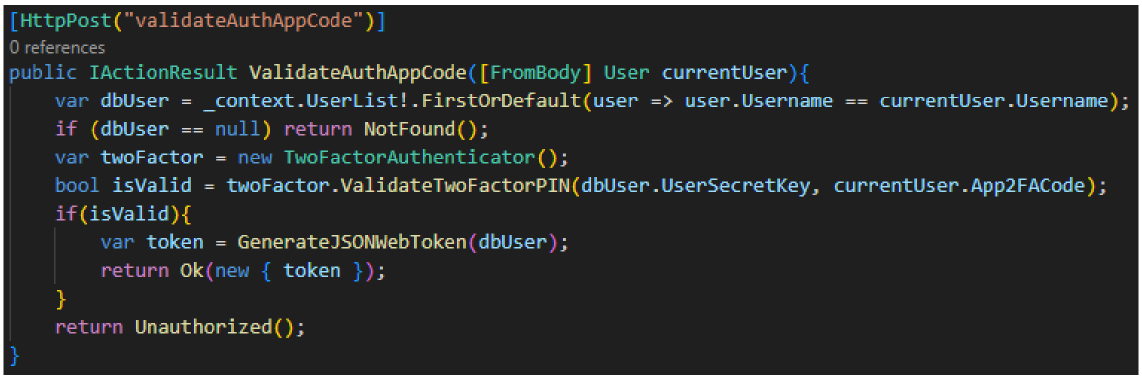
```
[Authorize]
[HttpPost("addAuthApp")]
0 references
public IActionResult AddAuthApp([FromBody] User currentUser){
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound("404");
    if(dbUser.App2FASetupKey==""||dbUser.QrCodeImageData=="")
    {
        var twoFactor = new TwoFactorAuthenticator();
        var userSecretKey = Guid.NewGuid().ToString();
        var setupInfo = twoFactor.GenerateSetupCode(
            "2FA VUE APP",
            dbUser.Username,
            userSecretKey,
            false,
            6);
        dbUser.App2FASetupKey = setupInfo.ManualEntryKey;
        dbUser.QrCodeImageData = setupInfo.QrCodeSetupImageUrl;
        dbUser.UserSecretKey = userSecretKey;
        _context.SaveChanges();
    }
    return Ok(dbUser.App2FASetupKey+"  "+dbUser.QrCodeImageData);
}
```

Figure 29



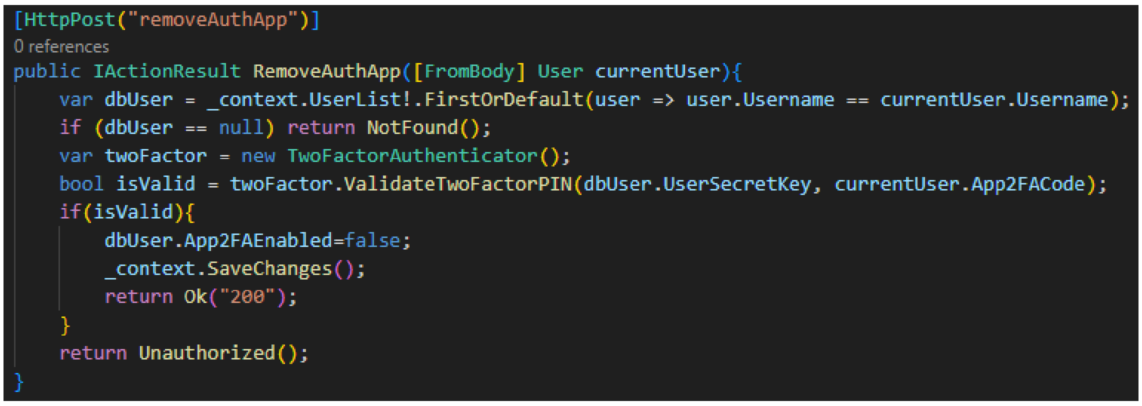
```
[Authorize]
[HttpPost("confirmAddAuthApp")]
0 references
public IActionResult ConfirmAddAuthApp([FromBody] User currentUser){
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    var twoFactor = new TwoFactorAuthenticator();
    bool isValid = twoFactor.ValidateTwoFactorPIN(dbUser.UserSecretKey, currentUser.App2FACode);
    if(isValid){
        dbUser.App2FAEnabled=true;
        _context.SaveChanges();
        return Ok("200");
    }
    return Unauthorized();
}
```

Figure 30



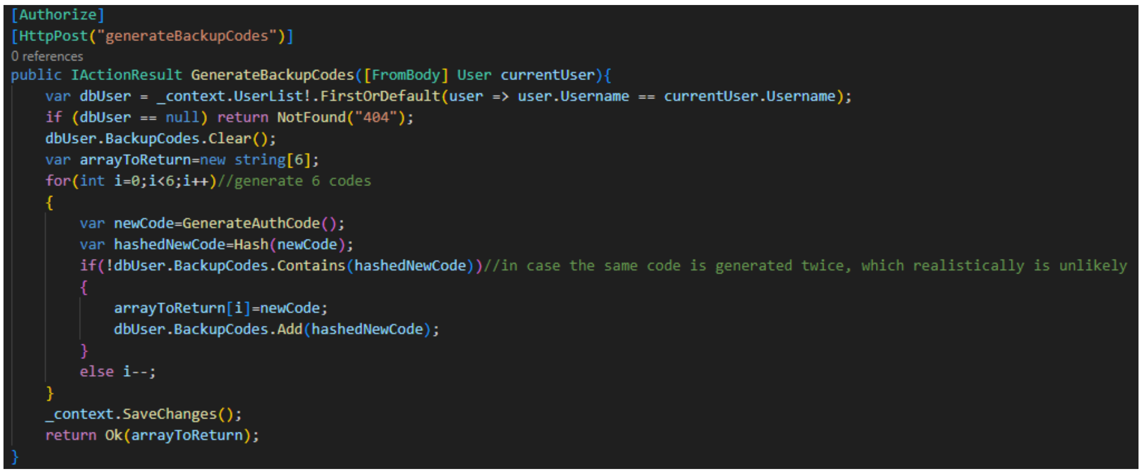
```
[HttpPost("validateAuthAppCode")]
0 references
public IActionResult ValidateAuthAppCode([FromBody] User currentUser){
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    var twoFactor = new TwoFactorAuthenticator();
    bool isValid = twoFactor.ValidateTwoFactorPIN(dbUser.UserSecretKey, currentUser.App2FACode);
    if(isValid){
        var token = GenerateJSONWebToken(dbUser);
        return Ok(new { token });
    }
    return Unauthorized();
}
```

Figure 31



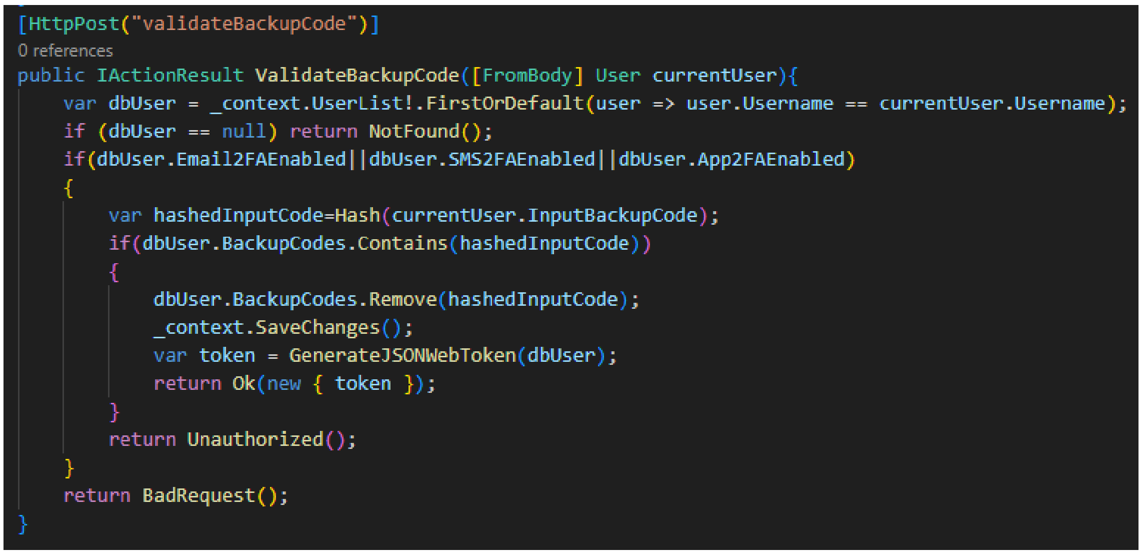
```
[HttpPost("removeAuthApp")]
0 references
public IActionResult RemoveAuthApp([FromBody] User currentUser){
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    var twoFactor = new TwoFactorAuthenticator();
    bool isValid = twoFactor.ValidateTwoFactorPIN(dbUser.UserSecretKey, currentUser.App2FACode);
    if(isValid){
        dbUser.App2FAEnabled=false;
        _context.SaveChanges();
        return Ok("200");
    }
    return Unauthorized();
}
```

Figure 32



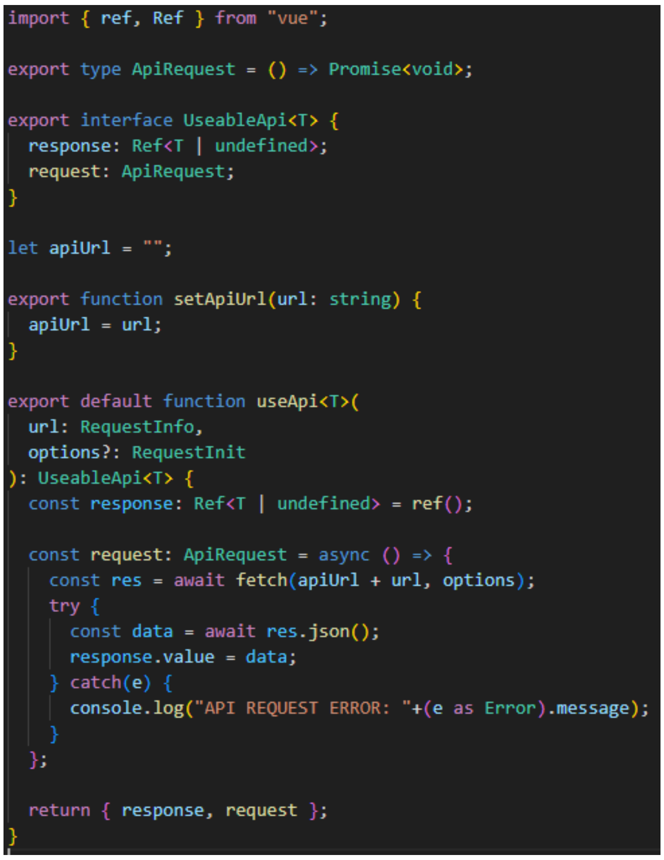
```
[Authorize]
[HttpPost("generateBackupCodes")]
0 references
public IActionResult GenerateBackupCodes([FromBody] User currentUser){
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound("404");
    dbUser.BackupCodes.Clear();
    var arrayToReturn=new string[6];
    for(int i=0;i<6;i++)//generate 6 codes
    {
        var newCode=GenerateAuthCode();
        var hashedNewCode=Hash(newCode);
        if(!dbUser.BackupCodes.Contains(hashedNewCode))//in case the same code is generated twice, which realistically is unlikely
        {
            arrayToReturn[i]=newCode;
            dbUser.BackupCodes.Add(hashedNewCode);
        }
        else i--;
    }
    _context.SaveChanges();
    return Ok(arrayToReturn);
}
```

Figure 33



```
[HttpPost("validateBackupCode")]
0 references
public IActionResult ValidateBackupCode([FromBody] User currentUser){
    var dbUser = _context.UserList!.FirstOrDefault(user => user.Username == currentUser.Username);
    if (dbUser == null) return NotFound();
    if(dbUser.Email2FAEnabled||dbUser.SMS2FAEnabled||dbUser.App2FAEnabled)
    {
        var hashedInputCode=Hash(currentUser.InputBackupCode);
        if(dbUser.BackupCodes.Contains(hashedInputCode))
        {
            dbUser.BackupCodes.Remove(hashedInputCode);
            _context.SaveChanges();
            var token = GenerateJSONWebToken(dbUser);
            return Ok(new { token });
        }
        return Unauthorized();
    }
    return BadRequest();
}
```

Figure 34



```
import { ref, Ref } from "vue";

export type ApiRequest = () => Promise<void>;

export interface UseableApi<T> {
  response: Ref<T | undefined>;
  request: ApiRequest;
}

let apiUrl = "";

export function setApiUrl(url: string) {
  apiUrl = url;
}

export default function useApi<T>(
  url: RequestInfo,
  options?: RequestInit
): UseableApi<T> {
  const response: Ref<T | undefined> = ref();

  const request: ApiRequest = async () => {
    const res = await fetch(apiUrl + url, options);
    try {
      const data = await res.json();
      response.value = data;
    } catch(e) {
      console.log("API REQUEST ERROR: "+(e as Error).message);
    }
  };

  return { response, request };
}
```