



IKT222 - SOFTWARESECURITY

---

**Assignment 3: User Authentication**

---

Autumn 2023

Github repository:  
<https://github.com/Siverteh/UiA-message-board>

Sivert Espeland Husebø

December 31, 2023

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Database Integration</b>	<b>2</b>
2.1	Technical background . . . . .	2
2.1.1	SQLite . . . . .	2
2.1.2	SQLAlchemy . . . . .	2
2.2	Database Implementation . . . . .	2
2.2.1	Schema Design . . . . .	2
2.2.2	Data Relationships . . . . .	4
2.3	Database Migrations . . . . .	4
2.4	Vulnerabilities & Mitigations . . . . .	5
2.4.1	SQL injections . . . . .	5
2.4.2	Data Relationships and Integrity . . . . .	5
<b>3</b>	<b>Basic User Authentication</b>	<b>6</b>
3.1	Technical background . . . . .	6
3.1.1	bcrypt . . . . .	6
3.2	User Registration Process . . . . .	6
3.2.1	Form Design and User Input . . . . .	6
3.2.2	Password hashing and Storage . . . . .	6
3.3	User Login Process . . . . .	7
3.3.1	Form Design and User Input . . . . .	7
3.3.2	Credential Verification and User Authentication . . . . .	7
3.4	Vulnerabilities & Mitigations . . . . .	8
3.4.1	Form Input Validation . . . . .	8
3.4.2	Session management . . . . .	8
3.4.3	Cross-Site Request Forgery (CSRF) . . . . .	8
<b>4</b>	<b>Protection Against Brute Force Attacks</b>	<b>9</b>
4.1	Brute Force Attacks Theory . . . . .	9
4.1.1	Methodology . . . . .	9
4.1.2	Countermeasures . . . . .	9
4.2	Implementation of protective measures . . . . .	10
4.2.1	Password Policy . . . . .	10

4.2.2	Rate-limiting mechanism . . . . .	11
4.2.3	Time-out mechanism . . . . .	11
<b>5</b>	<b>Two-Factor Authentication</b>	<b>13</b>
5.1	Theory and Technical Background . . . . .	13
5.1.1	Two-Factor Authentication . . . . .	13
5.1.2	pyopt . . . . .	13
5.2	Implementation . . . . .	13
5.2.1	User Model . . . . .	13
5.2.2	2FA Setup Process . . . . .	14
5.2.3	2FA Verification Process . . . . .	16
5.3	Vulnerabilities & Mitigations . . . . .	16
5.3.1	2FA Token Security . . . . .	16
5.3.2	QR Code Generation . . . . .	17
5.3.3	Session Fixation in 2FA Setup and Verification . . . . .	17
5.3.4	User Convenience Issues . . . . .	17
<b>6</b>	<b>OAuth2</b>	<b>18</b>
6.1	Oauth2 theory . . . . .	18
6.2	Implementation . . . . .	18
6.2.1	Integration of Google OAuth2 Authentication . . . . .	18
6.2.2	Integration of GitHub OAuth2 Authentication . . . . .	19
6.2.3	BackEnd implementation of both Google and GitHub login . . . . .	19
6.3	Vulnerabilities & Mitigations . . . . .	20
6.3.1	Redirect URI manipulation . . . . .	20
6.3.2	Access Token Exposure . . . . .	20
6.3.3	Client Impersonation . . . . .	21
<b>7</b>	<b>Additional security measures</b>	<b>22</b>
7.1	HTTPS . . . . .	22
7.1.1	Theory . . . . .	22
7.1.2	Implementation . . . . .	22
7.2	Email verification . . . . .	24
7.2.1	Theory . . . . .	24
7.2.2	Implementation . . . . .	25

7.2.3	Email verification functionality limitation . . . . .	26
7.3	CAPTCHA . . . . .	26
7.3.1	Theory . . . . .	26
7.3.2	Implementation . . . . .	26
<b>8</b>	<b>Architectural Choices</b>	<b>29</b>
8.1	Main directory structure . . . . .	29
8.2	Route management . . . . .	30
8.3	Front-End Assets . . . . .	30
8.3.1	Static . . . . .	30
8.3.2	Templates . . . . .	31
8.4	Utility . . . . .	31
<b>9</b>	<b>Conclusion</b>	<b>32</b>
	<b>References</b>	<b>33</b>

# 1 Introduction

In this assignment, I will be continuing the development of the University of Agder message board that I started in assignment 2. I will be focusing on enhancing the security of my website by employing user authentication methods.

I'm set to learn about implementing an SQLite database with a security-first approach. I will be implementing a basic username-password authentication system with hashing to protect credentials. Additionally, I will be working on measures to counteract brute force attacks through the use of rate-limiting.

Another key area of focus will be on implementing Two-Factor Authentication (2FA). This will involve creating a system that supports time-based one-time passwords (TOTP) to enhance security. TOTP lets users use apps like Google Authenticator to add a new layer of security to their accounts.

Lastly, I will be learning the basic concepts of OAuth2 by developing an OAuth2 client. This feature will allow users to register and log in using a third-party provider like Google or Github.

## **2 Database Integration**

### **2.1 Technical background**

#### **2.1.1 SQLite**

SQLite is a lightweight, file-based database management system that is known for its simplicity and reliability [1]. It is serverless and requires no separate server process which simplifies its deployment and maintenance. Its simple and lightweight nature makes it a good choice for applications with modest database requirements like small to medium web applications. SQLite stores the entire database as a single cross-platform file which further simplifies its data portability. In the context of projects developed with Flask, SQLite provides sufficient functionality to handle data storage and retrieval efficiently. Its compatibility with SQL lets developers utilize familiar database operations which makes it a robust choice of backend data management in web applications [1]. I chose to use SQLite because of its simplicity and reliability. This was an ideal choice for my simple lightweight Flask application.

#### **2.1.2 SQLAlchemy**

SQLAlchemy is a SQL toolkit and Object-Relational Mapping (ORM) library for Python [2]. It is designed to simplify the interaction between Python applications and databases. The ORM part of this library lets developers create, query, and manipulate databases without writing raw SQL code. Instead, developers can work with database entities through Python classes and objects. This ability makes it easier to manage complex data models and relationships. SQLAlchemy's integration with Flask simplifies database operations and enhances maintainability which makes it versatile and scalable for web development [2]. I chose to use SQLAlchemy due to its ORM capabilities. I always use ORM for my projects and SQLAlchemy was the best and most secure library for this.

## **2.2 Database Implementation**

My database is implemented using SQLite. As my web application is designed to be a small, security-focused message board I intentionally limited the complexity of my database to ensure streamlined operations and maintainability. This approach results in a straightforward and efficient database schema that supports the core functionalities of a message board without being overly complex. This enhanced both performance and ease of management.

#### **2.2.1 Schema Design**

For my web application, the database is designed to efficiently manage user data, messages, and comments while ensuring robustness and security. My database schema revolves around three primary models: 'User', 'Message', and 'Comment'

## User model:

- **Fields:**

- **id:** A unique integer that serves as the primary key for each user.
- **username:** A string field, indexed and unique, and is for the user's identification.
- **email:** A unique string field that holds the users email address.
- **password\_hash:** Stores the user's password securely as a hash.
- **failed\_attempts:** Tracks the number of consecutive failed login attempts for the account. Gets reset to zero after a set time.
- **lock\_until:** Holds the amount of time the account is locked after a set amount of failed login attempts.
- **totp\_secret:** Stores the secret key for Two-Factor Authentication.
- **is\_2fa\_setup:** A boolean that indicates if the user has configured 2FA.
- **messages:** A relationship linking to the user's messages.
- **comments:** A relationship linking to the user's comments.

- **Methods:**

- **hash\_password:** A setter method to hash and set the user's password.
- **verify\_password:** Checks if the provided password matches the stored hash.
- **increment\_failed\_attempts:** Increases the failed attempts count, and locks the account if it reaches more than three attempts.
- **is\_account\_locked:** Checks if the account is currently locked.
- **reset\_failed\_attempts:** Resets the count of failed login attempts. **set\_totp\_secret:** Generates and sets the TOTP secret for 2FA.

## Message model:

- **Fields:**

- **id:** The primary key for each message.
- **title:** The title of the message.
- **content:** The body of the message.
- **date\_posted:** Date and time when the message was posted.
- **updated\_at:** Date and time when the message was last updated.
- **author\_id:** A foreign key linking to the message author's user record.
- **comments:** A relationship to the comments associated with the message.

## Comment model:

- **Fields:**

- **id:** Primary key for each comment.
- **content:** The content of the comment.
- **date\_posted** Date and time when the comment was posted.
- **message\_id:** A foreign key linking the comment to a message.
- **author\_id:** Links the comment to the user who created it.

### 2.2.2 Data Relationships

As my web application is only designed as a small message board with a focus on security the backend database is somewhat small. This also means that there aren't that many complex relationships within my application. There is basically the bare minimum needed database-wise to create a well-functioning message board. Still, there are multiple relationships between the three models.

The model with the most relationships is the user model. This model has a one-to-many relationship with both messages and comments. A user can post as many messages or comments as they want, but a specific message or comment can only belong to one user. Furthermore, messages and comments also have one-to-many relationships. Messages can have multiple comments, but a specific comment can only belong to one message. This means messages have the user id as a foreign key, and a comment has both the user id as well as the comment id as its foreign keys. Lastly, there are cascading deletes in place. This means that if you delete a message, all of its comments will also be deleted, and if you delete a user all of their messages and comments will be deleted.

## 2.3 Database Migrations

For my web application, I decided to integrate Flask-Migrate as it is considered best practice for the security of a database. Flask-Migrate is a tool that works together with SQLAlchemy to manage database migrations effectively. It allows for controlled and tested schema changes which reduces the risk of accidental data loss or corruption. The integration of Flask-Migrate is set up in my main 'app.py' file and facilitates the automatic generation, application, and reversion of migration scripts in response to changes in the database models. This entire process works similarly to version control processes like GIT but is used purely for databases. Each migration script is akin to a GIT commit and represents the state of the database schema at that point in time. Database migrations create a new migrations directory in my project that stores the current as well as previous database versions. This process allows for efficient management of database schema changes and ensures the application's stability and data integrity.



## 2.4 Vulnerabilities & Mitigations

### 2.4.1 SQL injections

- **Vulnerability:** Injection of malicious SQL queries through a user input.
- **Mitigation:** The use of SQLAlchemy's ORM prevents SQL injections as it makes the use of raw SQL queries impossible. This ensures all user inputs are handled securely.

### 2.4.2 Data Relationships and Integrity

- **Vulnerability:** Inconsistencies in data relationships can lead to poor data integrity.
- **Mitigation:** I have carefully designed the relationships between the different models with appropriate foreign key constraints. Moreover, I have cascade rules in place so all deletions happen correctly. This ensures data integrity.

## 3 Basic User Authentication

### 3.1 Technical background

#### 3.1.1 bcrypt

'bcrypt' is a widely used cryptographic algorithm designed for hashing passwords, and it is widely renowned for its security and adaptability [3]. One of the key features of 'bcrypt' is its built-in salt mechanism. In cryptography, a 'salt' is a random sequence of characters added to a password before it is hashed. This process ensures that even identical passwords will have different hashes. This mitigates the risk of rainbow attacks that employ precomputed hash tables to crack passwords. Another significant aspect of 'bcrypt' is its 'work factor'. This is an adjustable parameter that determines the complexity of the hashing process which makes it more robust against brute force attacks [3]. In my application 'bcrypt' hashes and salts a user's password before it is stored in the database.

### 3.2 User Registration Process

The user registration process in my web application is an important step in creating a secure and personalized user experience. This process involves collecting and verifying user details, securely storing the credentials, and setting up the user's account.

#### 3.2.1 Form Design and User Input

The registration process for my application starts with a user-friendly interface facilitated by the 'RegistrationForm' class. This utilizes the 'Flask-WTF' for form handling and the form captures the following essential user information:

- **username:** Ensures each user has a unique identifier. This field is validated for uniqueness and presence.
- **email:** An email field that requires a user to have an email. This field is also validated for uniqueness and presence.
- **password and confirm:** Validates that the password meets the security requirements and that the confirmation password matches the original password. These fields also ensure user data security and prevent typographical errors.

#### 3.2.2 Password hashing and Storage

Whenever a user registers an account by filling out the registration form the user's password undergoes a critical security process. The password is hashed using bcrypt's advanced hashing

algorithm. This securely transforms the password into a hash before it is stored in the database. Furthermore, the password is automatically salted by bcrypt. This process adds a unique string to each password and ensures that each hash is distinct, even for identical passwords. After a user has submitted their registration form their password is never shown or stored anywhere in the application unhashed.

### 3.3 User Login Process

The user login process is a crucial component for the security of my web application. It ensures that only authenticated users gain access to their accounts. This process is as streamlined and secure as possible, and it verifies user credentials against the stored data.

#### 3.3.1 Form Design and User Input

The login process is also initialized with a simple and intuitive form, implemented using the 'LoginForm' class. This class also uses the 'Flask-WTF' library which aids in efficient form handling. The login form is designed to be straightforward as it only requests essential information from the user. The form captures this information:

- **username:** Users are required to enter their unique username. This is the same username the user registered with and serves as their identifier during the login process.
- **password:** Users must enter their password. This field is important for authentication and it is handled as securely as possible.

#### 3.3.2 Credential Verification and User Authentication

When a user submits the login form the application performs a series of security checks:

1. **Username verification:** The application starts by checking if the entered username exists in the database. This step ensures that the user attempting to log in is registered.
2. **Password hash comparison:** If a username is found the application retrieves the hashed password associated with that username from the database. Using bcrypt the entered password is hashed and then compared against the stored hashed password using the 'verify\_password' function. This comparison is crucial as it verifies the authenticity of the user without ever storing or transmitting the actual unhashed password.
3. **Authentication and session management:** If the entered password matches the user is authenticated. The application then initiates a new session for the user. This session allows the user to interact with the website securely until they log out or the session expires.

## 3.4 Vulnerabilities & Mitigations

### 3.4.1 Form Input Validation

- **Vulnerability:** Insecure user input can lead to SQL injections or other similar attacks.
- **Mitigation:** I am using Flask-WTF which has secure form validation. The library ensures all user input is sanitized and validated before processing it.

### 3.4.2 Session management

- **Vulnerability:** Poor session management can lead to hijacking or fixation attacks.
- **Mitigation:** On each login I make sure to regenerate the session IDs. Moreover, I ensure sessions are encrypted and timed out appropriately.

### 3.4.3 Cross-Site Request Forgery (CSRF)

- **Vulnerability:** CSRF is a type of attack where an attacker tricks a user into submitting a request to a web application where they are authenticated without their knowledge or intention.
- **Mitigation:** I use `{{ form.hidden_tag() }}` in my forms. This is a hidden form that contains a CSRF token, which is a unique token for each user session and form. This token ensures all form submission is authenticated and originates from the users legitimate session.

## 4 Protection Against Brute Force Attacks

### 4.1 Brute Force Attacks Theory

Brute force attacks are a method used by attackers to gain unauthorized access to user accounts. This type of attack is characterized by an attacker attempting a large number of different password combinations with the aim of eventually guessing the right password [4].

#### 4.1.1 Methodology

When an attacker performs a brute force attack they usually use automated software that generates a large number of guesses for a user's password. These software tools can try thousands or even millions of password combinations in a very short amount of time. Moreover, these tools usually utilize dictionaries of common passwords, alphanumerical combinations, and password patterns [4]. The most common brute force attack techniques are:

- **Simple brute force:** This technique is to simply try every possible combination of characters until the correct password is found.
- **Dictionary attacks:** This attack uses a predefined dictionary of commonly used passwords, phrases, and words.
- **Hybrid attacks:** This attack combines dictionary attacks with the use of brute force. It often modifies dictionary words with numbers or symbols.

While brute force attacks are a relatively unsophisticated attack they continue to be effective. This is mainly because of poor password practices among users. The simplicity of setting up a brute force attack, coupled with the availability of computing resources makes brute force attacks a persistent threat [4].

#### 4.1.2 Countermeasures

There are many efficient countermeasures to brute force attacks [4]. The most simple one is to implement a strong password policy as most successful brute force attacks depend on poor password practices among users. Another efficient countermeasure is to implement a rate limiter. A rate limiter limits the number of login attempts that are allowed in a certain time span. If anyone goes over this limit they will get redirected to the HTTP error 429 for too many requests. Furthermore, you can expand upon the rate limiter and implement an account lockout policy. For example, after five login attempts on the same account, the account will be locked for 30 minutes. Lastly, the use of third-party services like CAPTCHA and Two-Factor Authentication (2FA) mechanisms are effective mitigation strategies [4].

## 4.2 Implementation of protective measures

### 4.2.1 Password Policy

As stated earlier the biggest reason behind successful brute force attacks is poor password practices among users. This means that one of the easiest and most efficient ways to defend against these types of attacks is by forcing users to have strong passwords by enforcing a password policy. When a password policy is in place a user is unable to create an account unless they use a strong enough password.

To create a password policy for my application I started by incorporating the policy into my 'RegistrationForm'. I used the modules DataRequired, EqualTo, Length, and Regexp from the 'wtforms' library. The DataRequired model lets me set a policy that a password must be filled out, EqualTo lets me make sure that the password and confirm fields match, and length lets me specify the length of the password. The Regexp module lets me ensure that every password has at least one lowercase letter, one uppercase letter, one number, and one symbol.

Lastly, using HTML and JavaScript I placed the password policy in a user-friendly way into the register page. This lets new users easily see the requirements for the password. If a requirement is missing it will be shown as a red X and the text will be clear. On the other hand, if a requirement is fulfilled it will be marked by a green checkmark, and the text will be semitransparent. This is how my register page looks like with the password policy in place:

The figure displays two versions of a registration form titled "Register at University of Agder".

**(a) Password policy empty:** This version shows the form with empty input fields. The "Username" field contains "Sivert". The "Password" and "Confirm Password" fields are empty. Below the fields, a list of requirements is shown, each preceded by a red "X", indicating they are not met:

- X Be at least 8 characters long
- X Contain at least 1 lowercase character
- X Contain at least 1 uppercase character
- X Contain at least 1 number
- X Contain at least 1 special character
- X Match

**(b) Password policy filled in:** This version shows the form with the "Password" and "Confirm Password" fields filled with dots, indicating they contain text. The list of requirements is now shown with green checkmarks, indicating all requirements are met:

- ✓ Be at least 8 characters long
- ✓ Contain at least 1 lowercase character
- ✓ Contain at least 1 uppercase character
- ✓ Contain at least 1 number
- ✓ Contain at least 1 special character
- ✓ Match

Both versions of the form include a "Register" button and a "Return to homepage" link at the bottom.

(a) Password policy empty

(b) Password policy filled in

Figure 1: Register page with password policy

### 4.2.2 Rate-limiting mechanism

In my application, the implementation of a rate-limiting mechanism plays an important role in mitigating brute-force attacks. This mechanism is implemented using the Flask-Limiter library, and it lets me restrict the number of POST attempts a user can make. In my application, I have limited the number of allowed POST attempts to five attempts per minute. This limitation is applied to login, register, and both 2FA setup and verification.

When a user attempts to send a POST request to my application, the system checks the number of POST requests made within the last minute. If this number exceeds five attempts the server will respond by redirecting the user to the HTTP 429 error page, indicating too many requests. This rate limiter effectively makes it impossible to perform brute force attacks that rely on making numerous login or register attempts in a short period of time.

The rate-limiting strategy is applied to both the normal login and 2FA verification pages as well as the normal register and 2FA setup pages. This is the result of performing too many POST requests:

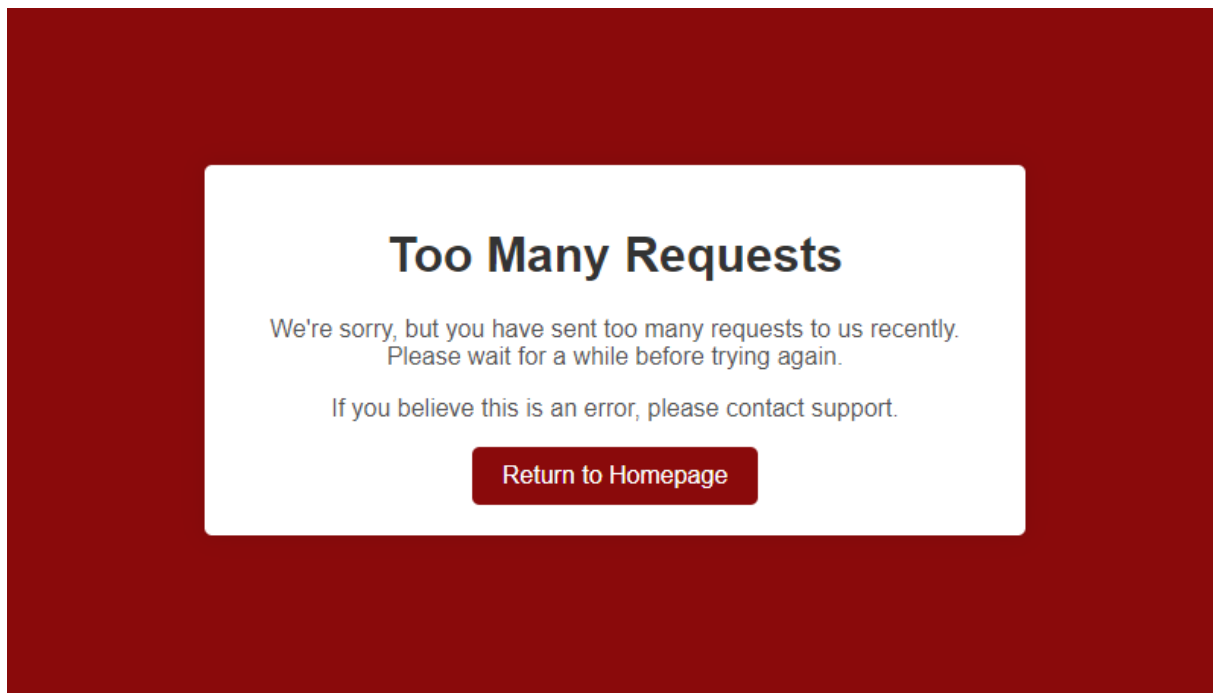


Figure 2: Being sent to HTTP 429 too many requests

### 4.2.3 Time-out mechanism

Alongside the rate-limiting mechanism, my application also incorporates a lock-out mechanism as part of its defense against brute force attacks. A lot of this implementation was shown in the database section of my report as most of the lock-out mechanism is implemented within the 'User' model of my applications database schema. Specifically, it involves fields and methods for tracking the number of consecutive failed login attempts, and locking the account if the number of attempts exceeds a certain threshold.

For every failed login attempt the 'increment\_failed\_attempts' method runs which increments the 'failed\_attempts' count. If this count reaches three the account is locked for 30 minutes. The 'lock\_until' field holds the datetime until the account is unlocked, and during this time no login attempts are permitted for the account. If a user attempts to log into a locked account they will be notified that their account is locked and informed of the time when the account will be unlocked.

The time-out mechanism is very efficient in preventing brute force attacks as it provides the attacker with a significant delay after just a few failed login attempts. This makes it impractical for attackers to continue trying passwords. Here is what happens after three failed login attempts:

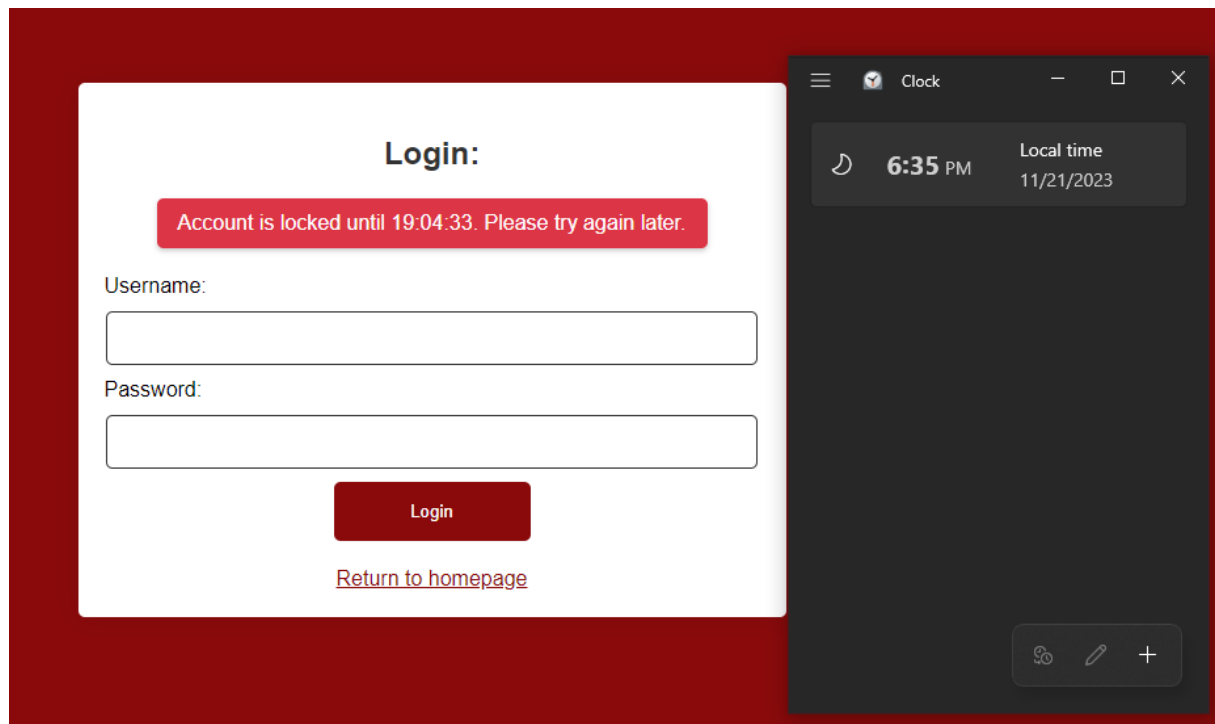


Figure 3: An account having a time-out



## 5 Two-Factor Authentication

### 5.1 Theory and Technical Background

#### 5.1.1 Two-Factor Authentication

Two-factor Authentication (2FA) is a security process where a user has to provide two different kinds of identification to access their accounts [5]. This method significantly enhances security and is looked at as one of the most secure ways of protecting an account. The method combines something the user knows like a password with something the user has such as a mobile device. Having this combination is very effective in preventing unauthorized access as even if a password or a personal device is compromised the account cannot be accessed without both. The most common types of 2FA include SMS codes, authenticator app tokens, and physical security tokens. Implementing 2FA into systems is regarded as one of the most critical steps in securing user data and access to digital applications [5].

#### 5.1.2 pyotp

'pyotp' is a Python library used for integrating Two-Factor Authentication in applications [5]. It supports the generation of One-Time Passwords (OTPs) for either event-based codes (HOTPs) or time-based codes (TOTPs). The 'pyotp' library gives applications the feature to generate temporary codes that are used as a second factor in authentication processes which enhances security beyond traditional password-only applications. Moreover, the library supports QR code generation. This feature simplifies the process of linking a user's account with a 2FA application like Google Authenticator [5]. I chose to use the 'pyotp' library as it is a robust choice for implementing 2FA functionality into my application. I especially like the QR code generation feature.

### 5.2 Implementation

#### 5.2.1 User Model

To make the 2FA specific for each account in my application a part of the implementation had to be made within the User Model. This entailed the fields 'totp\_secret', and 'is\_2fa\_setup', as well as the method 'set\_totp\_secret'. The field 'totp\_secret' holds the TOTP secret number that connects the user's account to a 2FA authenticator app, and 'is\_2fa\_setup' is a boolean field that holds whether or not the user has setup 2FA. The method 'set\_totp\_secret' is simply a setter for the TOTP secret number and is used when the user first sets up their 2FA with an authenticator app. All the implementation within the User Model is to keep track of the individual 2FA information for each user.

### 5.2.2 2FA Setup Process

After a user has gone through the registration process successfully the registration route will redirect them to the setup 2FA page. On this page, the user will be able to set up 2FA for their account. The page contains instructions on how to set up 2FA, a QR code the user can scan to connect their account with an authenticator app, and a text input to link their account with said authenticator app. There is also a clickable link to cancel registration and return to the homepage. If a user chooses to cancel their registration and return to the homepage they will be redirected back to the 2FA setup screen the next time they try to log in.

Most of the logic behind my 2FA implementation happens within the `'setup_2fa'` route. This route starts with a nested `'generate_qr_code'` function that generates a QR code the user can scan with their authenticator app. This QR code is embedded with a TOTP URI that contains the user's TOTP secret number.

After this nested function, a TOTP form is used to capture the TOTP code entered by the user during the setup process. The TOTP code entered by the user is then verified using the `'pyotp.TOTP'` object. This basically validates the user's TOTP secret number embedded within the QR code with the TOTP code provided by the user's authenticator app. If this is valid the 2FA setup is marked as complete for the user. Lastly for enhanced security, after a successful 2FA setup, the session is re-initialized with the user's ID to prevent session fixation attacks.

This is how the 2FA setup page looks like for the user:

## 2-Factor Authentication Setup

Please scan the QR code below with your authenticator app.



After scanning the QR code, enter the 2FA code from your app below to verify setup.

Enter authenticator code:

Verify

[Cancel registration](#)

Figure 4: Two-Factor Authentication setup page

### 5.2.3 2FA Verification Process

After a user has successfully registered and set up their 2FA they can finally log into their account. When they log in they will be redirected to the 2FA verification route which is a crucial step during user login. This route ensures that the necessary session parameters are present which indicates that a user is in the process of logging in. Moreover, this session information is used to verify a user's identity. The 2FA verification page simply consists of a text input field where the user can enter the TOTP code from their authenticator app. If this code is valid the user will be logged in. Here there are multiple previously explained security measures in place like rate limiting, account lockout, and session re-initialization. The 2FA verification page looks like this:

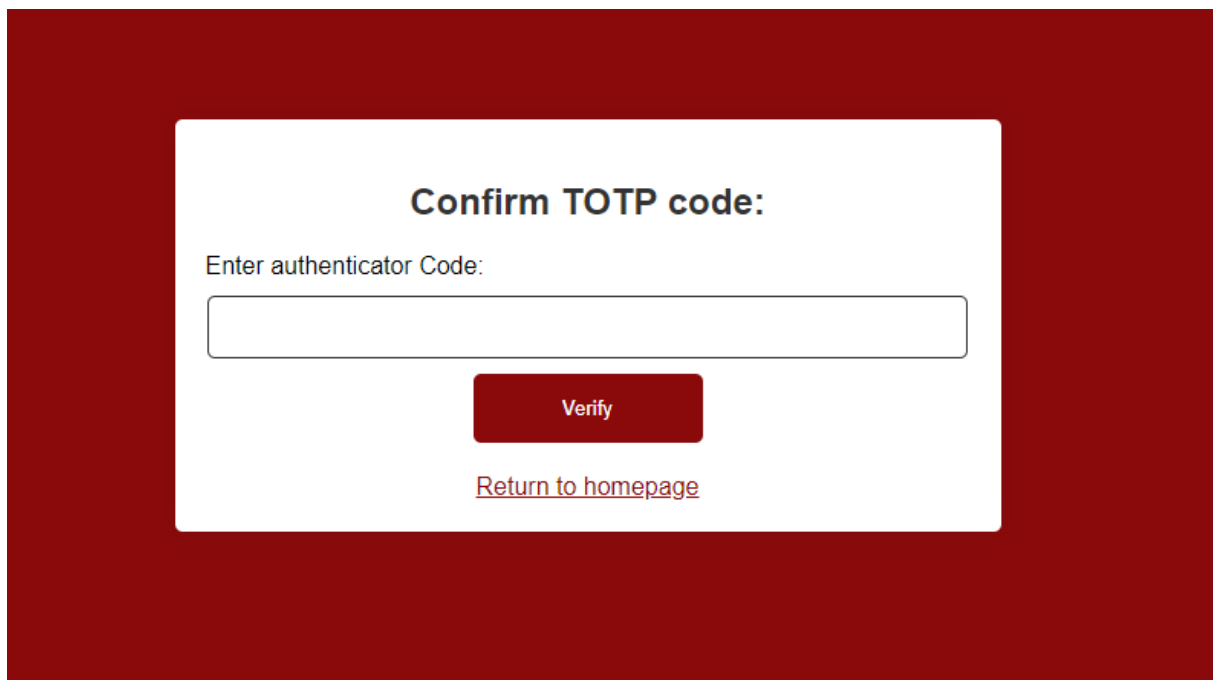


Figure 5: Two-Factor Authentication verification page

## 5.3 Vulnerabilities & Mitigations

### 5.3.1 2FA Token Security

- **Vulnerability:** If a 2FA token is transmitted or stored insecurely it could be intercepted by an attacker.
- **Mitigation:** The use of `pyotp` for generating and verifying 2FA tokens ensure that tokens are handled securely. Furthermore, I store the TOTP secret number securely in the User model.

### 5.3.2 QR Code Generation

- **Vulnerability:** Insecure generation and transmissions of the QR codes for 2FA setup could lead to interception and unauthorized access.
- **Mitigation:** All of the QR code generation happens on the server side in a secure environment. It is also encoded in base64 for secure transmission.

### 5.3.3 Session Fixation in 2FA Setup and Verification

- **Vulnerability:** The application could be susceptible to session fixation attacks during the 2FA setup or verification process.
- **Mitigation:** After every 2FA setup or verification I regenerate the session. This includes clearing the existing session and generating a new session for the specific user. This ensures a new session is always established after authentication.

### 5.3.4 User Convenience Issues

- **Vulnerability:** Users may attempt to bypass or misuse the 2FA process due to inconvenience.
- **Mitigation:** I have made the 2FA setup and verification as user-friendly as I can while still having it as a must. Furthermore, there is no way for a user to bypass or misuse the 2FA process in any way. At least any way I have found as I have tested multiple edge cases.

## 6 OAuth2

### 6.1 Oauth2 theory

OAuth2 is an authorization framework that lets applications access user accounts on external services like Google, GitHub, or Facebook [6]. It works by giving user authentication to the service that has the external user account and authorizing third-party applications to access the user information. OAuth2 consists of four main components:

- **Client:** The application that requests access to a user's account. (Example: my web application)
- **Resource owner:** The user who authorized an application to access their account. (Example: any random user with a third-party account)
- **Resource server:** The server hosting the protected resource. (Example: Google)
- **Authorization server:** The server that authenticates the resource owner and sends out an access token to the client. (Example: Google)

The OAuth2 flow starts with the Client (application) requesting access to a user's third-party account. This Client then redirects the Resource Owner (user) to the Authorization Server (i.e. Google) to get permission. The Resource Owner then authenticates their account with the Authorization Server and confirms that they consent to grant the Client access to their user data.

Once the Client has received the resource owner's consent the Authorization Server sends an authorization code to the Client. The Client then sends this authorization code back to the Authorization Server to request an Access token. This Access Token exchange ensures that the user's credentials are never exposed to the Client. Once the Client has obtained an Access Token it can use it to request the Resource Server (i.e. Google) to get things like user data. This user data can then be used to create an account in the application for the user and log them in. The use of Access Tokens lets a third-party application get access to user data in a secure and controlled manner [6].

### 6.2 Implementation

#### 6.2.1 Integration of Google OAuth2 Authentication

To integrate Google OAuth2 authentication into my application I needed to use Google's own OAuth2 client. I started with setting up a project in the Google Developer Console. This involved creating a new project and configuring the OAuth consent screen by providing it with information like the application name and contact information. Then I had to create OAuth credentials by selecting the "Web application" type and specifying the authorized redirect URIs. This is critical to have a secure flow of authentication data and makes it so only the URIs I specified are allowed to access the Google OAuth client. After successfully setting up a Google

OAuth account for my application I received a client ID and client secret key which I could use to link my application with the service.

### **6.2.2 Integration of GitHub OAuth2 Authentication**

The integration of GitHub OAuth2 Authentication was very similar to the integration of Google OAuth2 Authentication. I had to first register my application on GitHub. This involved creating a new OAuth application in the developer settings. In this setup, I had to provide details of my application like application name, homepage URL, and a callback URL. The callback URL is the most important detail as it is vital for a secure redirection after authentication. After registering my application with GitHub I received a client ID and a client secret. In my application, I could then use these credentials to link it with GitHub's OAuth2 service.

### **6.2.3 BackEnd implementation of both Google and GitHub login**

As the implementation of both Google and GitHub login is identical with the only difference being the OAuth provider used I will explain them both generally. I implemented OAuth2 authentication for them both using the 'Authlib' library. This involved configuring OAuth2 clients for each service and defining routes for login and callback processes. Each OAuth2 client is configured using the client ID and client secret key obtained from the developer accounts, as well as other standard configurations needed to register an OAuth2 client.

Next, I have two functions `{provider}_login` and `{provider}_authorize`. The login function is the simplest of the two and only handles the login through an external service. This function simply checks if the current user is already authenticated. If they are they are redirected back to the homepage. Otherwise, they get redirected to the provider's callback function.

The provider's callback function is where a user gets authenticated through a third-party OAuth provider. At the beginning of the function, a user is given an access token from the provider. This token is then validated and checked that it actually exists. Then a JSON file holding all the user information is created. I use this JSON file to check if the current third-party provider user exists in my database. If they do not I create a new entry for them in the User table containing their username, email, and provider ID.

For third-party users, I created two new fields in my User table, `google_id`, and `github_id`, as well as making the password field nullable. Furthermore, I added a function `generate_unique_username` that adds a number at the end of a third-party provider username to ensure no duplicate usernames in my database. Lastly, GitHub does not always give out the email of a user in their JSON file. Therefore, I check if I got an email in the JSON file, and if not I add their mail as `"{username}@noemail.github.com"`. This email should be changeable in a future works account settings page.

After ensuring a third-party user has an entry in my User table everything is set up correctly for the new user. Now I just check if the current third-party user has set up 2FA. If they haven't they are redirected to the 2FA setup page. Otherwise, they are redirected to the 2FA verification page. This is what the options for OAuth2 logins look like on my login page:

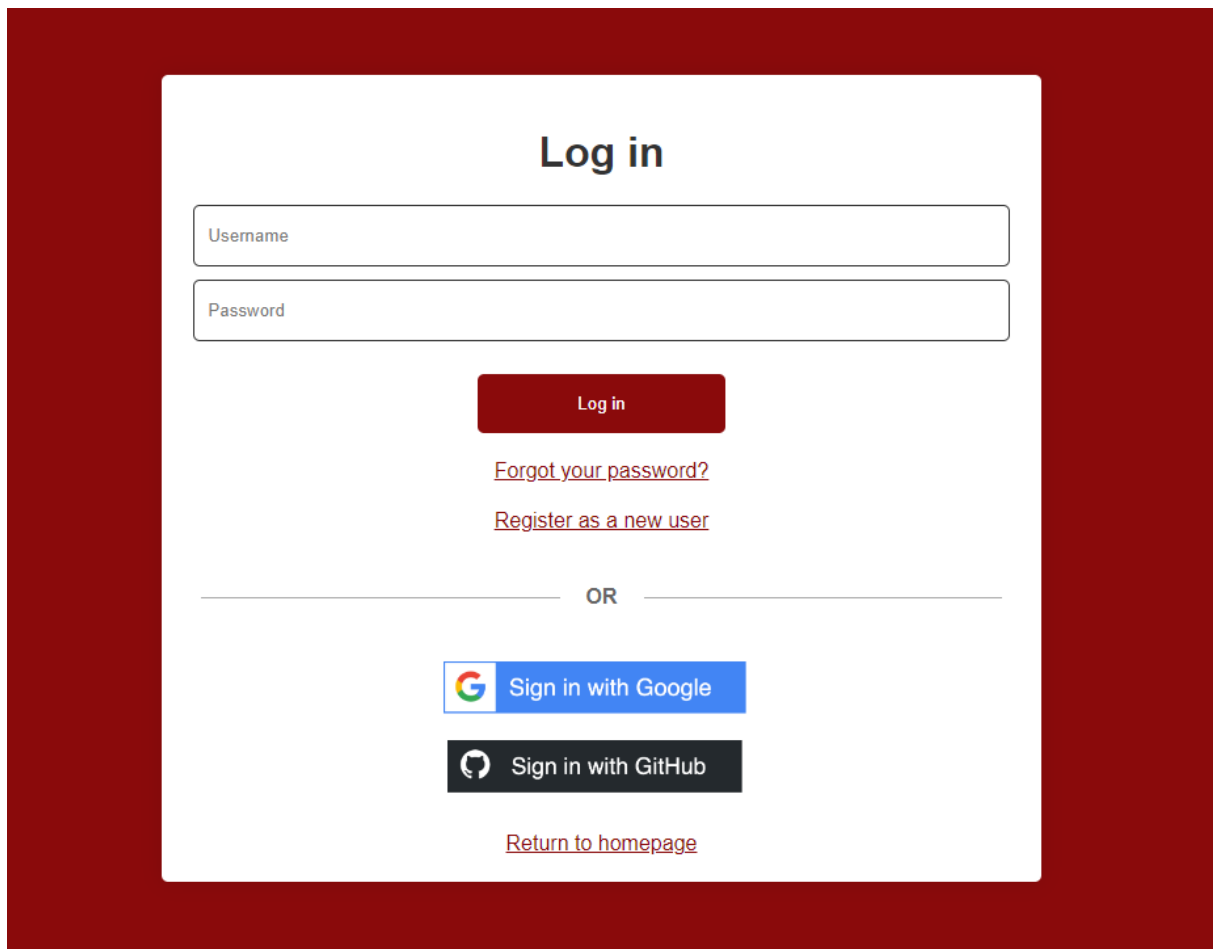


Figure 6: Oauth2 login choices

## 6.3 Vulnerabilities & Mitigations

### 6.3.1 Redirect URI manipulation

- **Vulnerability:** An attacker can alter the `redirect_uri` in the authorization request to try to intercept the authorization code.
- **Mitigation:** I registered only trusted URIs in the developer accounts of the application. This also makes sure the authorization server strictly matches the registered URI.

### 6.3.2 Access Token Exposure

- **Vulnerability:** Tokens transmitted over non-secure HTTP channels, or tokens that are stored insecurely can be intercepted.
- **Mitigation:** I use HTTPS for all the communications in my application including tokens. Moreover, I don't store the authorization tokens so they can't be intercepted that way either.



### 6.3.3 Client Impersonation

- **Vulnerability:** Attackers can create fake authorization requests and pretend to be legitimate users.
- **Mitigation:** The two third-party OAuth2 services I am using have robust client authentication systems making fake authorization requests impossible.

## 7 Additional security measures

### 7.1 HTTPS

#### 7.1.1 Theory

Hypertext Transfer Protocol Secure (HTTPS) is the secure version of HTTP [7]. Both of these are protocols for sending data between a user's browser and the website they are connected to. The difference is that HTTPS encrypts all the communication between the browser and the website using Transport Layer Security (TLS). This encryption is mainly used for stopping wiretapping and man-in-the-middle attacks. It is highly recommended for all websites to run over HTTPS, especially websites that handle sensitive data [7].

For a website to process an HTTPS connection, it needs to have an SSL/TLS digital certificate [7]. This certificate ensures two vital things. One is that the server your browser is connected to is actually the correct server. Two, it makes sure the data exchanged is encrypted to secure it from eavesdroppers. For a website to have an actual legit certificate for production, it needs to be given one from a recognized Certificate Authority (CA). Web applications that are in the development or testing phase can use self-signed certificates instead. This lets them test HTTPS-specific functionalities without having to get a certificate from a CA [7].

#### 7.1.2 Implementation

Implementing HTTPS for my Flask application which is only used for development purposes involved creating a self-signed SSL certificate rather than getting an official certificate from a CA. After creating a self-signed SSL certificate I configured my Flask application to use the certificate for its HTTPS connections. Here is an overview of this process.

##### 1. Creating a self-signed SSL certificate:

- I used OpenSSL which is a toolkit for SSL and TLS to create both a private key and a self-signed certificate for my application.
- The commands I used to generate these files were:

```
1 openssl genrsa -out localhost.key 2048
2 openssl req -new -key localhost.key -out localhost.csr
3 openssl x509 -req -days 365 -in localhost.csr -signkey
  ↪ localhost.key -out localhost.crt
```

- These commands generates me two files: 'localhost.key' (the private key) and 'localhost.crt' (the self signed certificate).

##### 2. Configuring Flask to run over HTTPS:

- In my Flask application, within my 'app.py' file I specified the SSL context to include the paths to my self-signed certificate and my private key.
- This just involves adding them to my 'app.run' like this:

```
1 app.run(ssl_context=('self-signed_ssl_certificate/localhost.crt',  
    ↪ 'self-signed_ssl_certificate/localhost.key'))
```

### 3. Running and testing the application:

- My Flask application will now run as normal and be accessible from the same URL's as previously.
- Whenever I access my application now my browser will display a warning about the self-signed nature of my certificate. This is proof that my web application is now running over a HTTPS connection with a self-signed certificate.

Through this implementation, I can not test different HTTPS-specific functionalities for my Flask application. Still, if I ever were to employ my application for production I would be using a trusted CA-issued certificate. The self-signed certificate can be seen within my application like this:

General		Details
Issued To		
Common Name (CN)	Sivert Espeland Husebo	
Organization (O)	University of Agder	
Organizational Unit (OU)	Faculty of Technology	
Issued By		
Common Name (CN)	Sivert Espeland Husebo	
Organization (O)	University of Agder	
Organizational Unit (OU)	Faculty of Technology	
Validity Period		
Issued On	Wednesday, November 22, 2023 at 7:39:44 PM	
Expires On	Thursday, November 21, 2024 at 7:39:44 PM	
SHA-256 Fingerprints		
Certificate	7eb539984f9b4c4377a8f68be1afd754db7b36e4e066596888f2a5bbcffc2be0	
Public Key	6dc8ffd05647c083de209fdd0509f3d36317efe3f37491942ccd36c436317b98	

Figure 7: Self-signed SSL certificate

## 7.2 Email verification

### 7.2.1 Theory

Email verification is a security practice in software applications that validates the authenticity of an email address provided by a user [8]. In other words, it lets us confirm that the email a user registers with really belongs to that user. This measure ensures that the email is both

valid and accessible to the user, and helps to ensure user authentication and data integrity. The process typically involves sending a unique, time-sensitive token in the form of a link to the user's email which the user must click to confirm their email. This verification is important to prevent unauthorized account access, maintain accurate user data, as well as enhancing the overall system security. Despite its importance, email verification can impose challenges like balancing user convenience and addressing privacy concerns [8].

## 7.2.2 Implementation

### SendGrid:

SendGrid is a cloud-based email service that can connect to an application and send emails on the application's behalf. The service is recognized for its robust deliverance, detailed analytics, and comprehensive API system. For my application, I utilized SendGrid to send email verification emails. To use SendGrid I had to set up an account, obtain an API secret key, configure my application to use SendGrids API, and incorporate my SendGrid secret key into my application's email-sending functions. Furthermore, I set up a new personal Gmail "uiamessageboard@gmail.com" and connected it to my SendGrid account to be the email sender for my application.

### Backend implementation:

I started my backend implementation by creating a `email_utils.py` file that contains the email utility functions `send_email`, `generate_confirmation_token`, and `confirm_token`. These functions as their name implies are used to send a confirmation email containing a generated confirmation token and then confirm this token when the link is pressed.

In my register route after a user has registered for my application, I generate a confirmation token and send them a confirmation email containing their token as well as a custom-made email verification HTML form. This email contains a link that will redirect them to a `confirm_email` route that confirms their token and redirects them to the 2FA setup. I added a `email_confirmed` field in my user table that is defaulted to `False` and gets set to `True` with successful email verification. This is how the email verification will look like for the user:

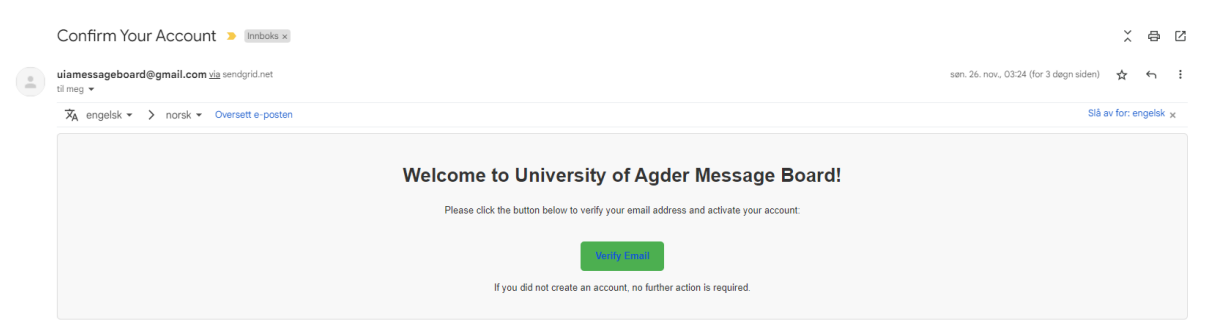


Figure 8: Email verification example

### **7.2.3 Email verification functionality limitation**

As stated above the email verification feature in my application relies on SendGrid's API which requires a secret API key to work. For security reasons, SendGrid won't allow me to post this secret API key to a public repository as it would compromise the integrity of my email service. Because of this, the email verification functionality will not be operational in the version of my application submitted for evaluation. As I prioritize software security there is no way for me to share this API key in a secure manner and email verification will therefore not be functional in my final deliverance. However, the implementation details will remain in my code and be available for code review.

## **7.3 CAPTCHA**

### **7.3.1 Theory**

CAPTCHA, which stands for "Completely Automated Public Turing test to tell Computers and Humans Apart" (great name), is as the name implies a challenge-response test used to determine if a user is a human or a computer [9]. The purpose of CAPTCHA is to prevent bots from performing automated actions on a website that degrades security or service quality. Examples of these actions can be creating numerous fake accounts or making automated attempts at guessing a user's password (brute force attacks). There are mainly two different types of CAPTCHA, checkbox and invisible. Checkbox CAPTCHA has a checkbox the user has to press. If the user seems like a human the checkbox will go green and the user can continue, but if the user seems like a box they will have to perform some sort of challenge before proceeding. Invisible CAPTCHA operates in the background and evaluates the user's mouse movement, typing browser history, etc. This CAPTCHA then scores the user from 0.1 (very likely a robot) to 1 (very likely a human). A developer can use this score with different thresholds to choose what actions should be taken based on the score [9].

### **7.3.2 Implementation**

#### **Google's reCAPTCHA**

To integrate CAPTCHA into my web application I used Google's reCAPTCHA service. This involved registering my application with a reCAPTCHA account, choosing a reCAPTCHA type, and registering my domain 'localhost'. I chose to use reCAPTCHA v2 instead of reCAPTCHA v3 as v2 is a more secure visible checkbox challenge style CAPTCHA instead of v3 which is a hidden scoring system. Then I chose to have my security preference set to max as this is a Software Security course. In a real in-production website this might not be the best choice as it makes the CAPTCHA process less user-friendly. After registering my application with reCAPTCHA I got an API site key and a secret key which I could use to implement reCAPTCHA into my application.

#### **Backend implementation**

The backend implementation of my reCAPTCHA is very simple and straightforward. In my `register.html` file I include a Google reCAPTCHA API script in my header, and then I implement a `g-recaptcha` element above my register button. All of the logic behind the reCAPTCHA checkbox button is done through the Google API script. Then in my register route, I get the response from the checkbox element and verify it using my reCAPTCHA secret key. Here all of the CAPTCHA logic is also done through a post request to a Google API. I chose to hardcode my secret key into my register route so my CAPTCHA functionality works in my delivered code repository. This is very bad practice and should be changed to an environment variable if the web application were ever deployed for production.

## New user

Your password must:

- ✗ Be at least 8 characters long
- ✗ Contain at least 1 lowercase letter
- ✗ Contain at least 1 uppercase letter
- ✗ Contain at least 1 number
- ✗ Contain at least 1 special character
- ✗ Match

Select all images with  
**motorcycles**



VERIFY

Figure 9: Working CAPTCHA on register



## 8 Architectural Choices

In developing my Flask application I adopted a structured and modular approach. This ensures maintainability and scalability for the future and it also provides ease of navigation. Furthermore, my architectural choices has helped with ease of development, simpler debugging, and future-proofing. This section will be an overview of the key architectural components and their roles within my application.

### 8.1 Main directory structure

The main directory is the heart of my application. It contains all the primary files as well as the essential directories for the functions of my applications.

#### Files:

- **app.py:** The core application file where the Flask app is initialized.
- **forms.py:** Contains the Flask-WTForms classes for form handling.
- **models.py:** Defines the SQLAlchemy database models for my model.
- **Dockerfile:** Used for containerizing my application so it works across different development environments.
- **requirements.txt:** Lists all python dependencies needed for the application.

#### Directories:

- **routes/:** Contains the various routing files of my application, logically separating different aspects of my application.
- **self-signed\_ssl\_certificate/:** Contains self-signed SSL certificates so i can use HTTPS for my development.
- **static/:** Stores static files like CSS, JavaScript, and images. Got subdirectories categorizing the static files into auth, message, and error static files.
- **templates/:** Contains all the HTML files for my application. Structured the same as static with directories for auth, messages, and error HTML files.
- **utility/** Contains utility scripts for configuration, extensions, email, etc.

## 8.2 Route management

To have my routes organized and structured I split them up into various files and used blueprints to tie them all back together in `app.py`. This makes it so I don't have a gigantic unstructured file but instead a bunch of smaller in-category files. The route structure I have is:

- **Auth Routes:** Handles all the routing for the normal authentication processes. This includes login, log out, registration, 2FA setup and verification, and email verification.
- **OAuth2 Routes:** Manages the OAuth2 authentication flow. These are the routes that allow users to log in through external providers like Google and GitHub.
- **Message Routes:** Manages routes specific to user messaging functionality in the application. This includes posting messages, editing messages, deleting messages, seeing a specific message, and commenting messages.
- **Error Routes:** Used for handling application errors and presenting users with appropriate nice-looking error messages.
- **Main Routes:** Per-now only contains the route for the main index page of the application. Is in its own blueprint so I can have the main page without a URL prefix.

## 8.3 Front-End Assets

My front-end assets are stored in the directories `static` and `templates`. These are all the files correlating to what a user will see on the screen.

### 8.3.1 Static

My static files are structured like this:

- **auth:** Contains a Javascript file corresponding to all my authentication template files. Also contains a CSS file for each authentication file. This includes a CSS file for login, registration, 2FA setup, and 2FA verification.
- **messages:** Similarly, contains one Javascript file for all my message's front-end logic. Also contains a CSS file for each message template file. This contains a CSS file for create-message, edit-message, and an individual message.
- **errors:** Contains a singular CSS file as all my error pages are styled the same way.
- **Images:** Contains all the images used for my application. Per-now this only includes images for the login with Google and Login with GitHub buttons.

There is also a singular CSS file for the front page directly in the `static` directory.

### 8.3.2 Templates

My template files are structured like this:

- **auth:** Contains HTML files for email verification, login, register, 2FA setup, and 2FA verification.
- **messages:** Contains HTML files for create-message, edit-message, and individual-message.
- **errors:** Contains HTML files for the HTTP errors 403, 404, 429, and 500.

This also contains a singular HTML file for the homepage directly in the templates directory.

## 8.4 Utility

My utility directory contains as the name implies all the utility files used for my application. These utility files include:

- **config.py:** Contains all the configurations for my application like a secret key, database URI, session cookie HTTPONLY, session lifetime, etc.
- **Extensions.py:** Initializes all the flask extensions for my applications. I mainly implemented this file as I had circular import problems with having the initializations in app.py. Contains initializations for db, bcrypt, login manager, limiter, etc.
- **populate\_database.py:** Contains a script for populating my database with some users, messages, and comments. This is just so the application will look good, and show of the different implementations without having to create a bunch of user accounts and messages by hand.
- **email\_utils.py:** Contains the functions for sending an email, generating confirmation tokens, and confirming tokens. Used for email verification.

## 9 Conclusion

In this project, I have successfully demonstrated the implementation of several comprehensive user authentication systems for a Flask web application. The achievement of the objectives I have shown in my report shows my now deep understanding of fundamental concepts within Software Security, especially within user authentication. The key accomplishments of this project have been the implementation of an SQLite database, basic user authentication, brute force attack prevention, Two-Factor authentication, and OAuth2. Furthermore, I have gone out of my way to learn the basic principles of integrating HTTPS, email verification, and CAPTCHA into a web application. Throughout this project, I have had a security-first mindset, and I have made sure my code has been well structured by using a modular approach to my code structure.

By doing this assignment alone I have had a huge learning outcome for various aspects of Software Security. I believe that this is the assignment I have had the highest learning outcome of ever. The main learning outcomes for this assignment have been:

- Gained practical experience in managing a database with a security-first approach.
- Gotten a good understanding of basic user-password authentication, especially with the use of hashing and salting methods.
- Learned about brute-force attacks and their mitigations.
- Got hands-on experience with 2FA and implemented its functionality for the first time. Also learned how to connect authenticator apps to personal applications.
- Learned how OAuth2 frameworks work, especially how the flow of OAuth2 works. Also learned for the first time how to implement login to a personal website using third-party login providers.
- I also chose to extend the objectives of the projects where I learned how to implement HTTPS using a self-signed certificate, setting up a confirmation email provider using SendGrid, and setting up CAPTCHA using Google's reCAPTCHA API.

The skills and understanding I have developed through this assignment will aid me in all future code endeavors by helping me perform more secure application development.

## References

- [1] A. Dyouri, “How To Use an SQLite Database in a Flask Application,” *DigitalOcean*, Nov. 2021. [Online]. Available: <https://www.digitalocean.com/community/tutorials/how-to-use-an-sqlite-database-in-a-flask-application>.
- [2] A. Dyouri, “How to Use Flask-SQLAlchemy to Interact with Databases in a Flask Application,” *DigitalOcean*, Mar. 2022. [Online]. Available: <https://www.digitalocean.com/community/tutorials/how-to-use-flask-sqlalchemy-to-interact-with-databases-in-a-flask-application>.
- [3] *Password Hashing with Bcrypt in Flask*, [Online; accessed 30. Nov. 2023], Mar. 2023. [Online]. Available: <https://www.geeksforgeeks.org/password-hashing-with-bcrypt-in-flask>.
- [4] *What is a Brute Force Attack? | Definition, Types & How It Works*, [Online; accessed 30. Nov. 2023], Nov. 2023. [Online]. Available: <https://www.fortinet.com/resources/cyberglossary/brute-force-attack>.
- [5] *Implementing TOTP 2FA in Python and Flask*, [Online; accessed 30. Nov. 2023], Jul. 2023. [Online]. Available: <https://www.section.io/engineering-education/implementing-totp-2fa-using-flask>.
- [6] M. Grinberg, *OAuth Authentication with Flask in 2023*, [Online; accessed 30. Nov. 2023], Nov. 2023. [Online]. Available: <https://blog.miguelgrinberg.com/post/oauth-authentication-with-flask-in-2023>.
- [7] M. Grinberg, *Running Your Flask Application Over HTTPS*, [Online; accessed 30. Nov. 2023], Nov. 2023. [Online]. Available: <https://blog.miguelgrinberg.com/post/running-your-flask-application-over-https>.
- [8] A. Krishna, “How to Set Up Email Verification in a Flask App,” *FreeCodeCamp*, Jan. 2023. [Online]. Available: <https://www.freecodecamp.org/news/setup-email-verification-in-flask-app>.
- [9] S. Dillikar, “How To Use Google reCAPTCHA With Flask - Python in Plain English,” *Medium*, Jan. 2022. [Online]. Available: <https://python.plainenglish.io/how-to-use-google-recaptcha-with-flask-dbd79d5ea193>.