COMP3334 Computer Systems Security

Group Project Report

Group 4

|  |  |
| --- | --- |
| LI Chak Yiu | 17040377D |
| CHAN Ho Wing | 17054281D |
| SIN Wing Sze | 17024086D |
| SZE Ming Tai | 18022533D |

**Abstract**

This report will document the implementation of security mechanisms in order to protect a Digital Identity application.

Our aim is to develop an application that handles users digital identity and applies various security mechanisms to protect the system.

Security analysis was conducted and the security requirements were proposed. Protection mechanisms were then proposed and implemented with considerations.

**Proposed Application**

The proposed digital identity application is a web application that takes users email and password to create an account and users can access to the website with their accounts. We expect ourselves to focus on handling access control, user authentication, password storage.

To prevent the project from being too large in scale, we decided to make the below assumptions and scope where it helps scaling down the project and making the security mechanism we wanted to implement feasible.

Assumptions

1. Our server accepts HTTPS connections and it comes with authorized digital certificate from major authorities. Implies that connection is encrypted under SSL protocol. The implementation cost is unaffordable for us.
2. The server is protected under proxy services like Cloudflare, the cost of implementation elements like DDoS mitigation and DNS is tremendously high.
3. Only the admin has access to the hardware and right to edit or view the source code and database.

Scope

1. Frontend pages where you can register, login, be verified and logout.
2. Backend server that can create users from valid input data, verify login input and access control.
3. Passwords are hashed before storage, countermeasure against online dictionary attack, two-factor authentication.

In the next section, a thorough analysis on security requirements will be examined.

**Security Requirement Analysis**

In this section, the security requirements for the proposed system will be analyzed. It will be divided into two parts : application requirements and vulnerability centric requirements.

**Application Requirements**

The general security requirements on application level will be explored.

1. Authentication

Authentication is the process of identifying an individual which is the basic part in our application. It usually uses username and password to do the authentication. For instance, when users want to login their account, they should input some credentials such as password, account number. After that, the system will check whether the credentials match the information in the database and permit the user to access its resources.

Since the authentication is the most important part in the application, it is really dangerous if we only do the authentication depending on userID and password. If attackers are able to access the database, they could use brute force attacks to extract the password. Thus, it is necessary to develop the process in authentication.

Using two-factor authentication, one-time password or multifactor authentication could improve the security in authentication and we use two-factor authentication which will add an extra layer of protection to the process of authentication.

1. Session Management

Common vulnerability of web application is caused by lack of protection on session tokens and account credentials. In an attack, unauthorized attackers can hijack the identity of a user and the session. Since HTTP is a stateless protocol, it will answer any request. Cookie and session ID are used to maintain state in HTTP sessions and keeping all the requests together.

Since cookies are plain text, the content inside the cookie must not contain any sensitive information to prevent attackers stealing information from it.

Session ID should be randomly generated and unique for every user after successful authentication. To prevent hijacking, the id should be protected with SSL. There should also be a timeout timer for a session. When the session becomes inactive, the user will have to authenticate again .

1. Auditing and logging

To detect unauthorized attempts, there should be auditing and logging for suspicious events. Besides detecting suspicious attempts, it can also provide a trail for reconstruction when system failure. Some vital operations should be recorded, such as administrator activity, modification of data and authorization events. To make the log meaningful, the record should contain the user context such as process and date to allow tracing.

1. Access Control

Web application requires access control to restrict users to certain pages. One common vulnerability is modifying the URL to bypass access control checks. Users should only be able to view pages that they have the privilege to view.

1. Cryptography

Encryption should be applied to confidential data such as password. For password, the password should be stored with a strong hashing function and a salt. The salt should also be unique and randomly generated for each user. Besides the strength of the hash function, it should also be a slow hash function to mitigate attacks such as rainbow tables.

To avoid attackers replaying cookies and hijacking the session by monitoring network traffic, data in transit should be protected with secure protocols such as Transport Layer Security.

**Vulnerability-Centric Requirements**

This section investigates critical security risks for web application. For our application, common website security risk will be considered to avoid such vulnerabilities exploited by attackers. Countermeasures should be provided to avoid the following attacks.

1. Query Selector Injection

Regardless of the type of database you are using, there are threats on query selector injection attacks. These injections serve for the same purposes, to exploit the input field and attain information that they are not authorized to.

For a NoSQL database when an object field is stored as a string and there is no format limitation, the attacker can easily input a string of query selector and extract information from the database. What we need to do is to make templates for the parameters and detect the query selector and remove them before passing it.

1. Cross Site Scripting

If the website creates output without validating the data submitted by the user, the output might contain malicious code. To avoid such attacks,the input from form data, cookies, headers should be checked against acceptable values.

1. Cross Site Request Forgery

It is an attack that tricks the end users to execute an unwanted action on a web application in which they are authenticated. As the user is authenticated, the application cannot identify whether the request is legitimate or forged. It can be prevented by having a randomly generated anti CSRF token, which is stored in a session and only known to the user's browser and the web application.

1. Buffer overflow

Buffer overflow is caused by copying an input buffer without validating the size of the buffer. When the application receives an amount of data that exceeds what it expected, the application will start to abandon its normal behavior. To prevent this attack, all input fields that accept user input must be reviewed to ensure it can handle large input. These data input fields should have a specific data type and maximum field length to limit user input.

1. Brute Force Attacks

Out of all different types of brute force attacks, the most possible threat is considered to be dictionary attack. The attacker contains a list of strings that are commonly used as passwords, there are open sources of password dictionaries on the Internet and hence it might be the most practical brute force attack.

From Lab 2, we asked about the solutions to online dictionary attacks, so we take the chance to implement the counter-measure in our project, the approach is to keep track of login failure attempts and refrain user logging in after 3 failed attempts.

Due to limited resources, some of the requirements cannot be implemented in our system.

**System design specification**

We will discuss the design specification in-depth on application and database level on how the system works under certain policies , how the data are being protected and how the design can resist major attacks.

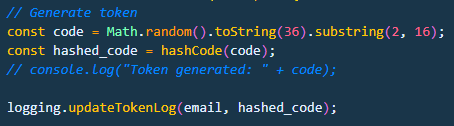
**Security on Authentication**

While the password of an user supposed to be only known by the user, there are always risks that the user’s password is known by unrightful individuals or parties. Applying such assumptions, using one factor to authenticate the identity of a user is dangerous. Therefore, two-factor authentication is applied to provide a stronger promise on user authentication.

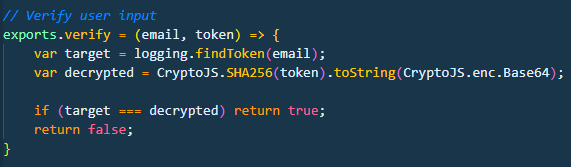
In this case, we decided to implement the email verification method as the second factor for its simplicity.

When the user submitted his or her email, our server side will generate a one-time token and it will be sent to the user’s email, the user is required to input the token in order to verify his or her identity. Every time a user attempts to login, a new token will be generated and the user will be prompted to be verified.

Here is the implementation of email verification. The token otherwise named as verification code is generated and before it is logged, it is encrypted in SHA-256.



The verification code will then be sent to the user email and server will verify when it receives the input from the user.

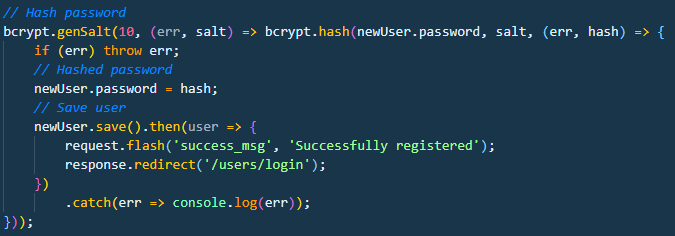


**Security on Password Storage**

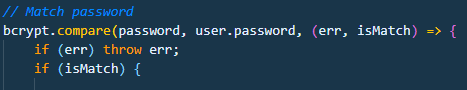
As taught in the lectures, it was never a good idea to store passwords in plaintext and our application took this into consideration of course. We decided to hash the passwords with a function that is designed for password which is bcrypt.

bcrypt is a well known hash function that is developed from Blowfish cipher and it takes cost, salt and the password and hashes through a slow key schedule [1]. The heavy cost of keys hashing could slow down the attacker hash calculation. And the cost could be scaled following the computational power of modern processors, time needed for hash computation increases exponentially and from a study, it needs 513.55 days to compute for cost equals to 30 [2]. It can effectively mitigate brute force attacks and the use of salt makes it resistant against precomputation attacks.

Below is the implementation of the encryption stage of bcrypt with the help of the library. The cost of setup is set to default which is 10 then it takes the newUser.password and generated salt to encrypt.



Below is the implementation of the decryption stage of bcrypt. bcrypt.compare() decrypts the stored hash of passwords and compares it with the user input. The function only returns a boolean value, the decrypted result is never stored, isMatch would either be true if the passwords matched or false if they do not match.

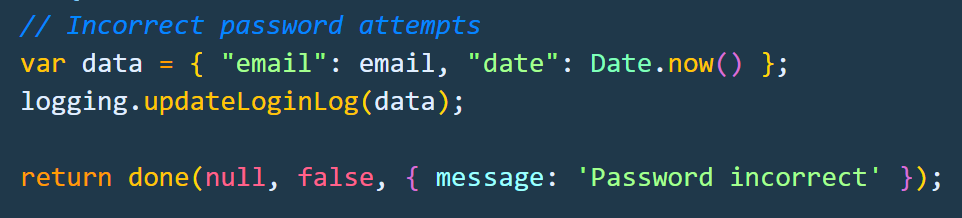
****

**Security on Dictionary attack**

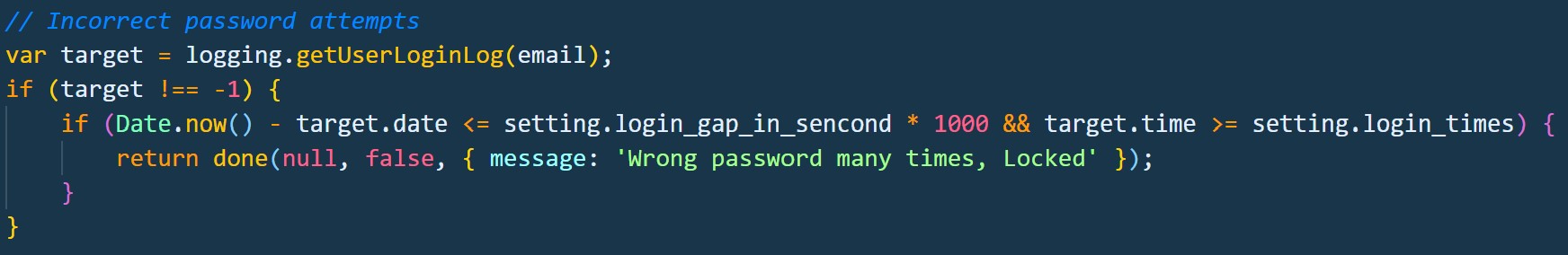
As mentioned above, passwords are hashed in bcrypt which can effectively mitigate offline dictionary attacks. For each trial, the calculation could take weeks or even years, which makes offline dictionary attack inefficient and impractical regardless of its advantage on unlimited attempts.

We also tried to tackle online dictionary attacks, a simple implementation of login fail attempts would do the work for our web application. Our default configuration would be if the user fails to login 3 times in 10 seconds, the account will be locked so no more attempts could be made.

Below is the implementation of this security mechanism. We increment the log record whenever the password does not match and reset the log once a successful login is made.



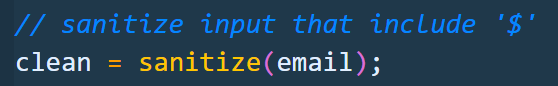
Then it will be checked whenever the email of the login input is found in the database. If too many fail attempts, the account will be locked.



**Security on Query selector injection**

As mentioned in the analysis section, attackers may post requests from query selector syntax, client credential information will be easily leaked if there is no safe guard on this problem. The solution we implemented is to “sanitize” the input value before passing it as a parameter. For MongoDB, queries are denoted by ‘$’ the dollar sign, for example ‘{ $ne: 1 }’ refers to returning all matching results where the field does not equal to 1. If this security issue is left unhandled, there is a severe risk of leaking all records in our database.

Therefore, we use the package ‘mongo-sanitize’ which detects query operators like ‘$ne’ and removes it before passing it as parameters. It only requires one line of code to keep the parameters injection free. We can pass clean as parameters instead of the using the input string directly.



**Security on Transmission**

To ensure the data security between the application server and the client web browser, only access to the application server with HTTPS under SSL will be allowed. The data transfer between the client and server would be encrypted by RSA.

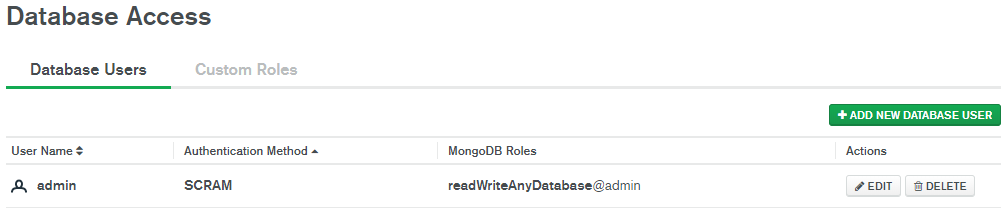
**Application installation guide**

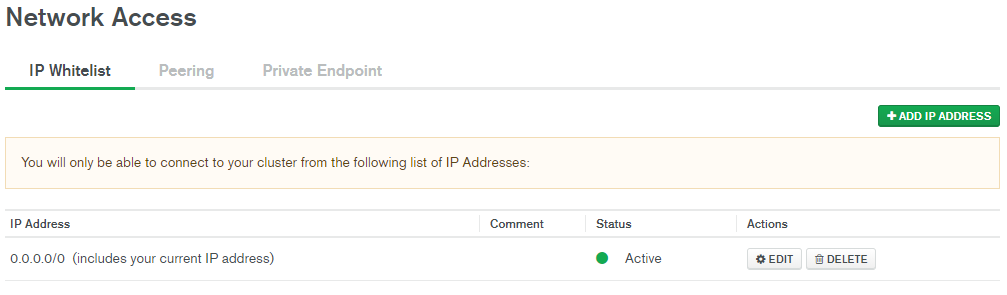
The application is developed using Node.js, necessary packages needed are listed as belows:

|  |  |
| --- | --- |
| Package | Version |
| bcryptjs | 2.4.3 |
| connect-flash | 0.1.1 |
| crypto-js | 4.0.0 |
| ejs | 3.1.2 |
| express | 4.17.1 |
| express-ejs-layouts | 2.5.0 |
| express-session | 1.17.1 |
| mongo-sanitize | 1.1.0 |
| mongoose | 5.9.14 |
| nodemailer | 6.4.6 |
| passport | 0.4.1 |
| passport-local | 1.0.0 |
| nodemon | 2.0.3 |

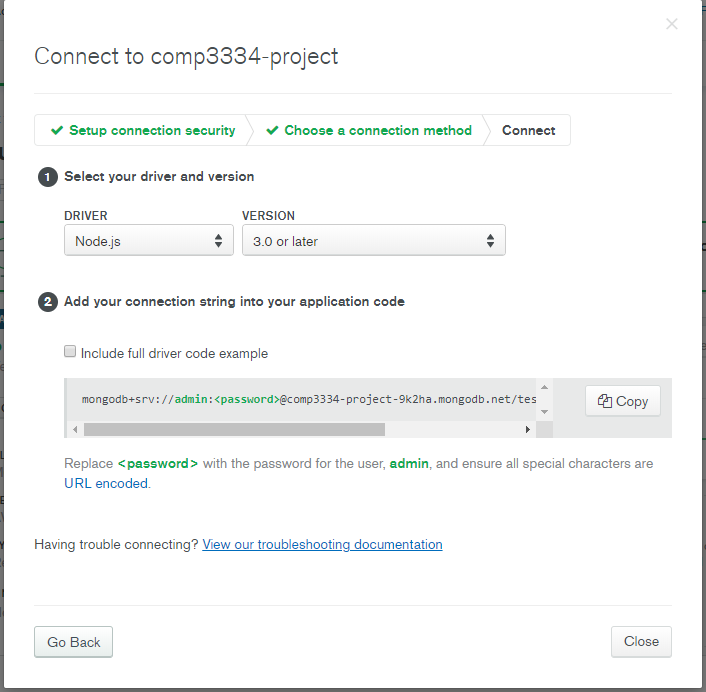
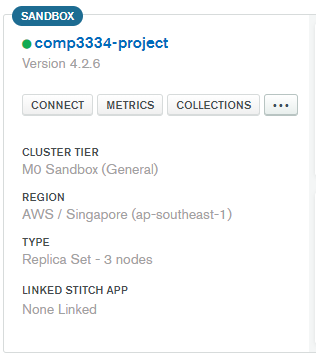
Step 1. Install Node.js from <https://nodejs.org/en/>. The version we used is v12.16.1 and npm with 6.13.4.

Step 2. Register a free account in <https://www.mongodb.com/>. And create a cluster with the region which is closest to you. Create an account under database access with **readWriteAnyDatabase permission** and create a IP address as **0.0.0.0/0** under network access.

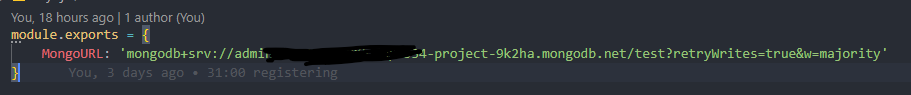




After doing so, click connect in the sandbox of the cluster, and choose **“Connect your application”** to get the connection string.



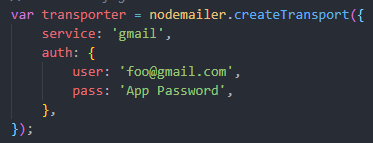
Replace the password with what you have set before and copy the whole string in to ‘./config/keys.js’.MongoURL.



Step 3. Open Command Prompt and navigate to the project directory (where you can see **README.MD**) Run **“cd LIChakYiu\_17040377d”**

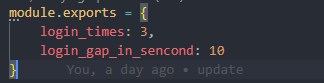
Step 4. Install the needed packages. Run **“npm install”** to install all the packages in package.json.

Step 5. Edit “./config/verify” to your gmail, which **IMAP** is enabled. Change the **user** and **pass** under Email configuration, remember using app passwords to authorize your account instead of your personal password.



Follow the guide provided below to generate your own app password. And remember to do the two-authentication setting of your account.<https://support.google.com/accounts/answer/185833?hl=en>

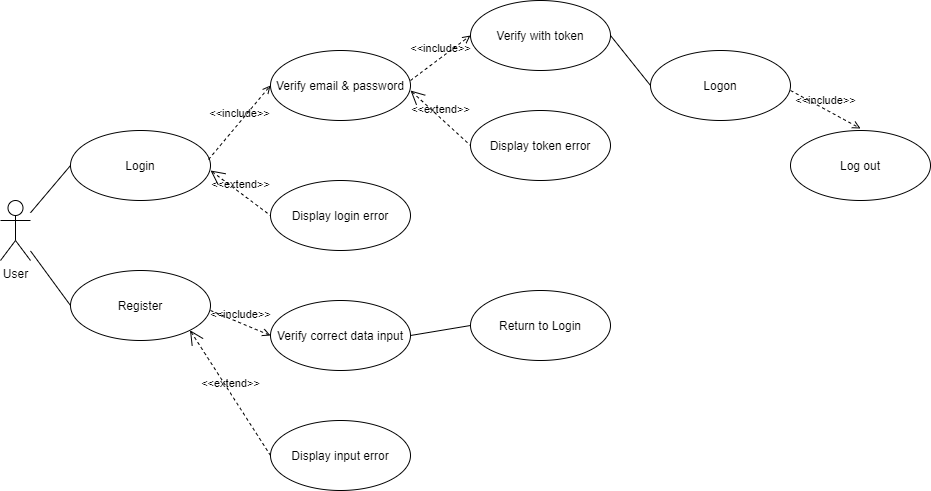
Step 6. Under the “./conig/setting.js”, you can edit the login fail times and lock account time. The default will be 3 and 10 second respectively for demonstration.



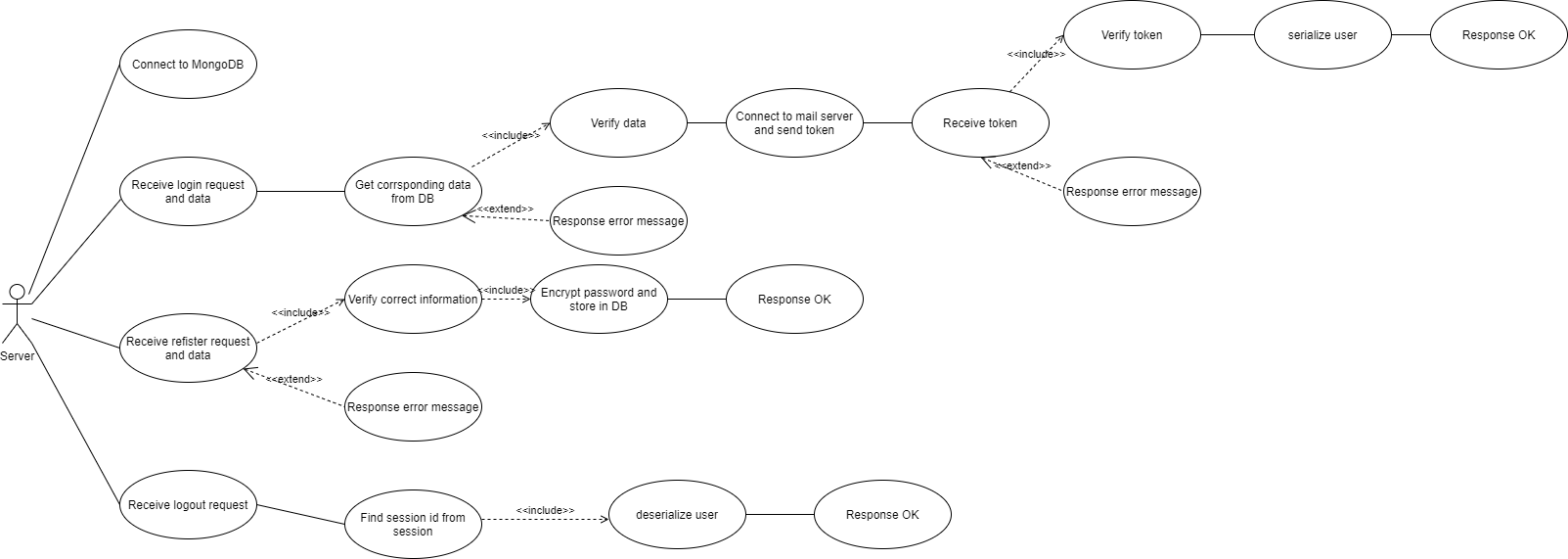
Step 7. Start the server. Run **“npm run start”**

**Use cases**

User-side:

****

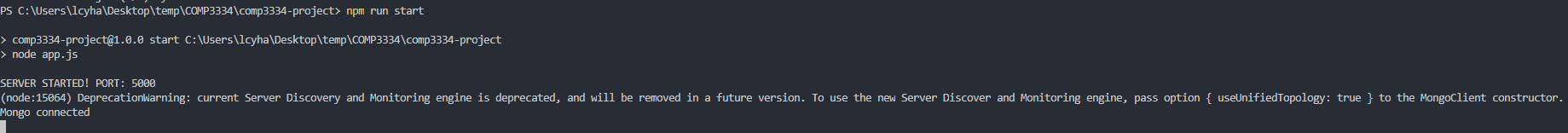
Server-side:

****

**Flow of the system**

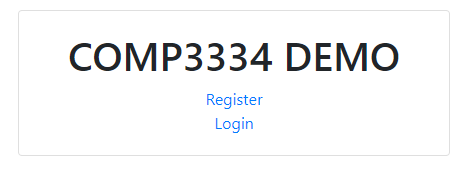
**client-side:**

After following the installation guide, open a terminal and direct to the folder and run “npm run start”.

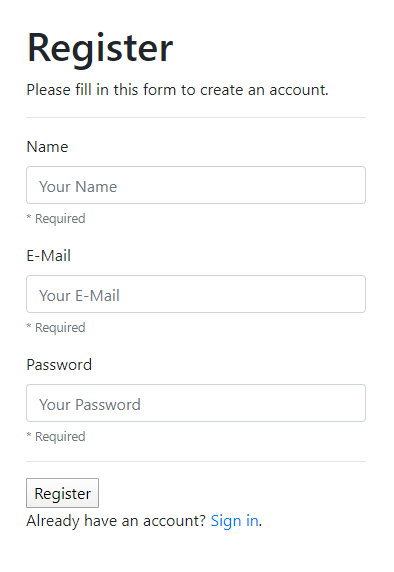


Terminal will show the port used and the database connected message.

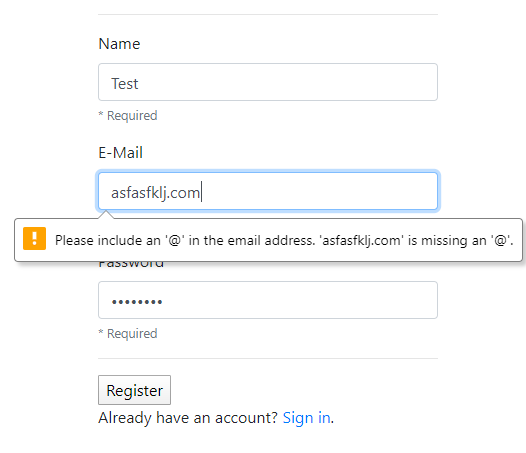
Open a browser and access to <http://localhost:5000/>. A very simple container will be shown

.

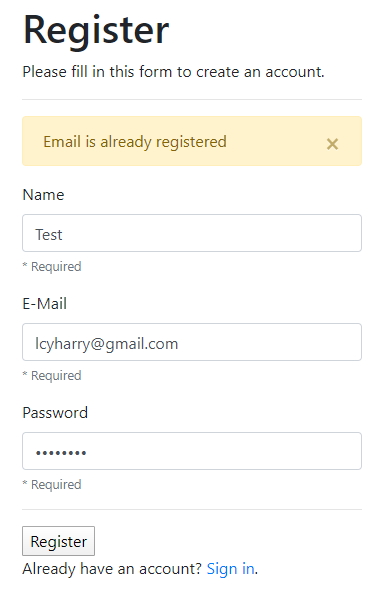
First we demonstrate the register process first.



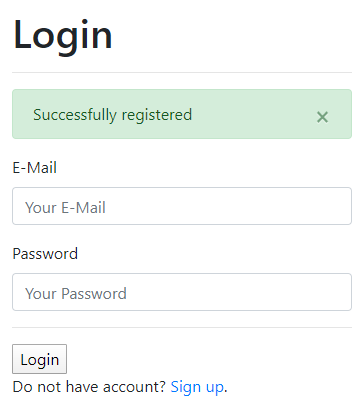
We try to type some error information first. For example, an wrong format email.



And if we input an email which already registered before. Error messages will be shown.



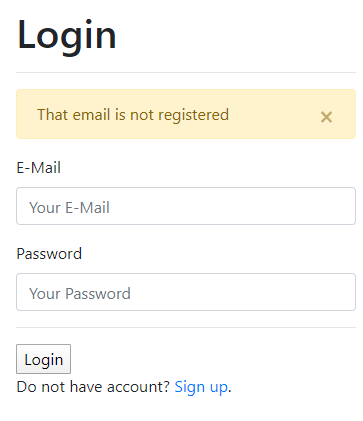
And we input correct information to register an account. It will redirect us to the login page and a success message will be shown.



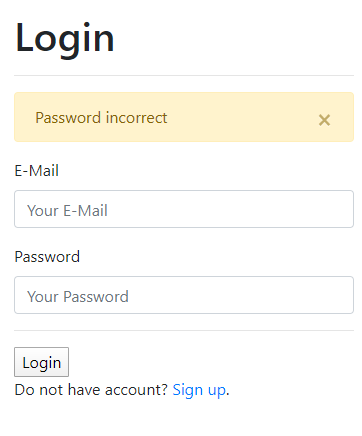
In addition, password will be encrypted and stored in database.



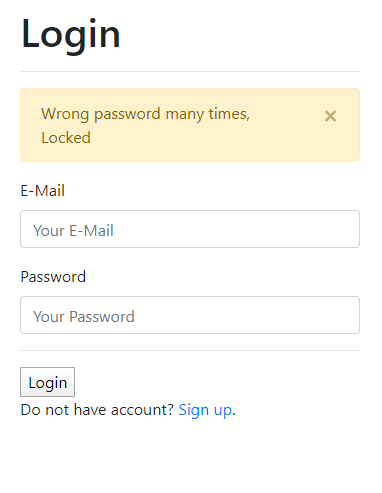
If we type an email which does not exist.



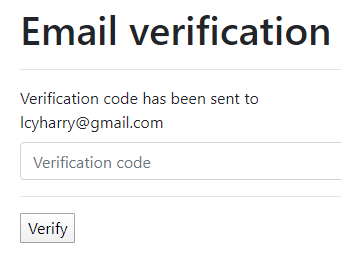
And if we type an incorrect password.

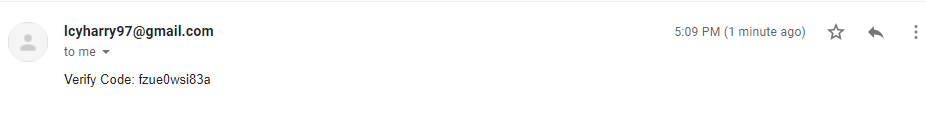


Moreover, if we try the wrong password more than 3 times within 10 second(for demonstration only, settings can be changed), the account will be locked.

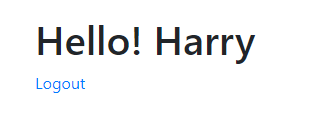


After that if we input a correct email and password. It will direct us to email verification page. And it will send the token to your email by smtp server.

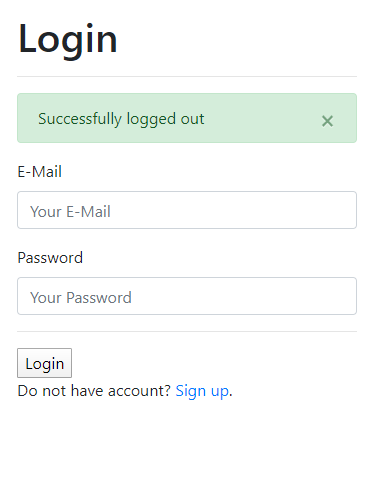




After typing the correct token, we successfully logon.



And if we logout, it will return us to login page and message will be shown.



**Flow of the system**

**Server-side:**

When the server starts, the server will first connect to the database. The database we used is MongoDB.

**const** db = "Your MongoDB connection string";

mongoose.connect(db, { useNewUrlParser: **true** }).then(() => console.log("Mongo connected")).**catch**(e => console.log(e));

After that the server is started and awaiting request from the user through the port. The port used is 5000.

**const** PORT = process.env.PORT || 5000;

app.listen(PORT, console.log(`SERVER STARTED! PORT: ${PORT}`));

When the server receives a request for registration, data will also be received at the same time. The data will be checked on the server side and response to client side about success or not. For correct information, the server is able to encrypt the passport and store it on the server by applying bcrypt. Knowing that bcrypt uses blowfish encryption which is improved DES encryption and has not been proven insecure yet.

bcrypt.genSalt(10, (err, salt) => bcrypt.hash(newUser.password, salt, (err, hash) => {

**if** (err) **throw** err;

newUser.password = hash;

newUser.save().then(user => {

request.flash('success\_msg', 'Successfully registered');

response.redirect('/users/login');

})

.**catch**(err => console.log(err));

}));

For the login request, server-side uses passport authentication to store the user information in session and process authentication for client-side.

router.post('/login', (request, response, next) => {

passport.authenticate('local', {

successRedirect: '/users/verify',

failureRedirect: '/users/login',

failureFlash: **true**

})(request, response, next);

});

Inside the passport function, we used local strategy as it uses username and password to access the server but not via facebook, google or other passport services providers. When the server side receive the user’s email and password, it will get the corresponding information of that account from the database. And it will process the comparison of the password of input and password from the database.

bcrypt.compare(password, user.password, (err, isMatch) => {

**if** (err) **throw** err;

**if** (isMatch) {

**return** done(**null**, user);

} **else** {

**return** done(**null**, **false**, { message: 'Password incorrect' });

}

});

After that the server will send the verification code to user email via smtp service provider. The smtp server used in this project is Gmail. And the code is randomly generated and encrypted by using SHA-256 by CyptoJS in the server-side and save in log file in server.

**const** code = Math.random().toString(36).substring(2, 16);

**const** hashed\_code = CryptoJS.SHA256(token).toString(CryptoJS.enc.Base64);

logging.updateTokenLog(email, hashed\_code);

When the server receives the verification code from the user, it will compare the code.

**var** target = logging.findToken(email);

**var** decrypted = CryptoJS.SHA256(token).toString(CryptoJS.enc.Base64);

**if** (target === decrypted) **return** **true**;

**return** **false**;

Finally, when the server receives a logout request, the server will then deserialize the user.

passport.deserializeUser(**function** (id, done) {

User.findById(id, **function** (err, user) {

done(err, user);

});

});

# **Conclusion**

In conclusion, we design a web application which could use email and password to get the permission on a website. The implementation of security is really important in such digital identity applications because the system will store many private data such as account number, email...

As a result, we focus our application on access control, user authentication, password storage and imply security mechanisms to protect it. For instance, we use two-factor authentication to provide more safety authentication that uses email verification to confirm the authenticity of the user. Moreover, we hash the password by using bcrypt that is replaced with plaintext to make it more safe.

We have learnt how to pull the security mechanisms in practical application during the project. It is a necessary requirement in all of the network applications that involve personal information and we believe that this project could help us to design other applications in the future.

# **Bibliography**

Akamai, “Q2 2017 State of the Internet Security Report “ Akamai, 2017. [Online]. Available: https://www.akamai.com/de/de/multimedia/documents/state-of-the-internet/q2-2017-state-of-the-internet-security-report.pdf [Accessed: 16- May- 2020]

Microsoft, “SQL Injection”, Microsoft Docs, 2012. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms161953(v=sql.105)?redirectedfrom=MSDN [Accessed: 16- May- 2020]

H. Elkamchouchi, M. Nasr and R. Ismail, "Secure and Fast Hashing Algorithm with Multiple Security Levels", *Journal of Software*, vol. 4, no. 9, 2009. Available: 10.4304/jsw.4.9.935-942.

S. Jajodia, P. Samarati, M. Sapino and V. Subrahmanian, "Flexible support for multiple access control policies", *ACM Transactions on Database Systems*, vol. 26, no. 2, pp. 214-260, 2001. Available: 10.1145/383891.383894.

"OWASP Top Ten Web Application Security Risks | OWASP", *Owasp.org*, 2020. [Online]. Available: [ht](https://owasp.org/www-project-top-ten/)tps://owasp.org/www-project-top-ten[/](https://owasp.org/www-project-top-ten/). [Accessed: 16- May- 2020].

D. Arias, 2018. *Hashing In Action: Understanding Bcrypt*. [online] Auth0 - Blog. Available at: <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/> [Accessed 15 May 2020].

N. Provos, D. Mazières, 1999. *A Future-Adaptable Password Scheme.* [online] USENIX. Available at: <https://www.usenix.org/legacy/events/usenix99/provos/provos\_html/node1.html> [Accessed 15 May 2020].