

**National College of Ireland**  
**Secure Programming for Web (H9SPW)**  
**MSc/PGD Cybersecurity**  
**MSCCYB1\_JAN22I, PGDCYB\_SEP**  
**Project**  
**Technical Report**

**Name & Student number:**

[Simon Lowry – x21168938]

Lecturers: Dr Vanessa Ayala-Rivera, Liam McCabe

## Contents

Executive Summary.....	3
1.0 Link to your video.....	3
2.0 Introduction.....	3
3.0 Software Development Methodology Employed.....	4
4.0 Requirements.....	6
4.1. Functional Requirements.....	
4.1.1. Use Case Diagram.....	
4.1.1.1. Requirement 1 <Name of requirement in a few words>.....	
4.2. Non-functional Requirements.....	
5.0 Design & Architecture.....	9
5.1. Architecture of the System.....	
5.2. Threat Modelling.....	
5.2.1. Step 1: Decompose the Application.....	
5.2.2. Step 2: Determine Threats.....	
5.3. Security Design.....	
5.4. Graphical User Interface (GUI).....	
6.0 Implementation.....	21
7.0 Testing.....	28
7.1 OWASP Penetration Testing Checklist	
8.0 Conclusions.....	43
9.0 References.....	44

## Executive Summary

It's no longer sufficient to stand idly by and not protect software and systems with adequate protections. With revenue, customer trust, intellectual property and large potential fines at risk, incorporating security in the design process and throughout the software lifecycle an imperative for business big and small. The purpose of this project is to create an application that has security embedded throughout it's lifecycle. It looks to deliver a design process, methodology and implementation centered around secure practices and principles.

A vast array of security concerns were identified in the design phase. The use of security requirements, principles and practices, threat modeling, abuse cases, misuse cases aims to identify security vulnerabilities from the get-go and reduce the potential cost and impact from not identifying them later. Security considerations continued with the use of industry used secure frameworks like Spring Security, using a UI language that caters for security features and that is well maintained like ReactJS. It has a database selection which has had security features incorporated from the beginning like MySQL.

The application has manifested the security features from the design phase and the threats and mitigations highlighted there. These have all been well tested and scrutinized with SAST, DAST tools, component testing, integration testing and a Penetration Testing checklist provided by OWASP.

A POC has been created here that won't require too much extra to become production ready or serve as an example for other projects in how secure programming can be applied from design, testing, implementation and eventually, release. There was a slight overshoot in terms of producing a Jenkins pipeline and a password reset form. These were only partially complete. Otherwise all other main functionality and security goals were reached.

## 1.0 Link to your video & Code

- **Link to code:** <https://github.com/Simon-Lowry/Cybersecurity-Masters-NCI/tree/main/Secure%20Programming%20for%20Web/Project>
- **Link to video:**  
[https://drive.google.com/drive/folders/1-ohuaKKF3wwlys\\_ovRwjWEyVZA8RmQIm?usp=sharing](https://drive.google.com/drive/folders/1-ohuaKKF3wwlys_ovRwjWEyVZA8RmQIm?usp=sharing)

## 2.0 Introduction

The main aim of this project is to develop a secure application which leverage secure design principles, architecture, frameworks, selections of databases and encompassing infrastructure to try and protect against security vulnerabilities. In the 7-Eleven attack hackers were able to execute SQL injection attacks on the convenience chain store 7-Eleven obtaining 130 million credit card numbers and 2 million dollars. For the Ghost Shell APT Attack, An APT hacking group known as Ghost Shell made use of the SQLMap tool to exploit SQL injection vulnerabilities obtaining 30000 personal records, this included from 53 different universities, as well as banks, government agencies and consulting agencies. [17] [18]

Placing an emphasis on the very aspects of design, implementation and maintenance ultimately yields better returns. Producing software defects not only reduces the quality of the overall product but also ends up slowing down production schedules and costing the company according to Capers Jones who conducted research on over 4000 projects. [17] Projects with high amounts of errors also tend to be increasingly complex, poorly structured and very large. On top of this, about 60 percent of defects exist in the software design phase and this includes security defects. [16]

A study showed that of those who experience a breach, 38% endure a loss of 20% of their revenue or more! [23] 60% of attacks have been to smaller businesses since there's the belief that they won't be targeted. [22] No company is immune to being targeted. It's better to look at it as, it's not if you'll incur a security breach, but when.

It's no longer acceptable to not protect user's data in Europe in particular with GDPR fines becoming increasingly pronounced with up to 4% of global turnover (or 20 million euro) at risk for failing to adequately protect customer data. [21] Here we're creating a heavily security centered application that's included security from design to implementation and can serve as a basis for further applications of what's been learned and explored in this module. [20] The impact of applying security in a shift left approach here will help result in less security defects, decreased likelihood of a security breach resulting in revenue loss, customer trust.

### 3.0 Software Development Methodology Employed

According to a study done on Agile, 95% of organizations embrace and use some form of Agile development. [13] The preceding main software process has significantly decreased in its application being seen as less of a fit for modern software development. Instead of the requirements all done up front and a linear process of going from design to implementation, testing etc, Agile is deemed to be a more adaptive, flexible, customer centric approach with self-organizing teams. It's a more iterative approach to building applications. In reality, this can usually result in a "get it done" and we'll worry about the documentation, testing and other non-functional requirements when time allows. This loop of getting features released at all costs and don't care about non-functional requirements or code quality is the antithesis of producing secure code. According to a survey by Veracode, 79% of organizations knowingly push code to production with security vulnerabilities in their code (48% doing this regularly) while, ironically, still thinking their applications are secure [14]

Security proponents are increasingly looking to adopt a shift left mindset increasing security from inception of requirements and throughout the development lifecycle. As a result of this there will always be a tension between producing results and producing secure code though this doesn't need to be fatal. Teams in certain organizations are now resorting to having application security engineers that either work across one to a few different teams which can be responsible for the security activities in conjunction with developers. These application security engineers are tasked with incorporating security throughout the lifecycle and championing security considerations in conjunction with developers and architects alike. This takes some of the onus away from developers, many of which are not incentivized to care about security. And for those that do care many are short on secure development knowledge and under pressure to meet the short iteration demands of Agile. [15] In this application and report, we're playing the role of both the application security

engineer and a developer in developing the code in an agile way with security embedded throughout the software lifecycle.

The security activities need to be doable and workable in parallel to the design of features and incorporating that very design while still being capable of being done at some degree of speed and flexibility. As such we have activities like threat modelling, abuse cases, misuse cases incorporated in the design process which are relatively lightweight processes and can be adapted over time to meet every changing and evolving requirements and still ensure security requirements and concerns are being met and not overlooked. Automation is increasingly important in effecting security considerations throughout the lifecycle with pipeline tools like Jenkins playing a big role here. DevSecOps comes into the picture. Here the application security engineer can put in some upfront effort to setup a variety of SAST and DAST tools in place that can be configured to run for every push of new changes of code to the repos. This is done as part of the build process usually in a containerized setup with tools like Docker. Snyk brings the static analysis of code, infrastructure like Docker containers, secrets/credentials scans, and dynamic tools like ZAP attacking the application to each and every build getting rapid feedback to potential security issues at the commit level playing into the continuous integration and continuous deployment pipelines exercised in many software development environments.

Some of the building blocks of that will be put in place for this application environment with tools like Snyk & ZAP being installed and configured to run periodically in the case of Snyk. The next stage in this iterative approach of DevSecOps is instead to have them run in a Jenkins pipeline as part of the build process. This will be done with time permitting. A GitHub hook would be the next following step to trigger the Jenkins pipeline build. This would build and run the application in a container and runs the various security tools for every commit as mentioned. So the approach will be to apply threat models, abuse cases, misuses cases, security requirements on a feature by feature basis and iteratively complete those features. Then these features will be tested with component testing, integration testing (for both security and functional testing) and then followed up with SAST and DAST tools being run either through a Jenkins pipeline or separately otherwise. This will help to not only get features done but also get instant feedback on both security and functional concerns. After most of the features are in place, to complement and compound the security efforts thus far, there will also be the use of a Penetration Testing checklist provided by OWASP applied to the application. Here, like with threat modelling, abuse cases, misuses cases, we have a set of retrospective checks in the persona of an attacker that can help identify any security issues in the application that have managed to get through the variety methods and controls in place thus far.



**Gantt Chart - Project Plan**

## 4.0 Requirements

### 4.1. Functional Requirements

This section lists the functional requirements which describe the possible effects of a software system, in other words, *what* the system must accomplish:

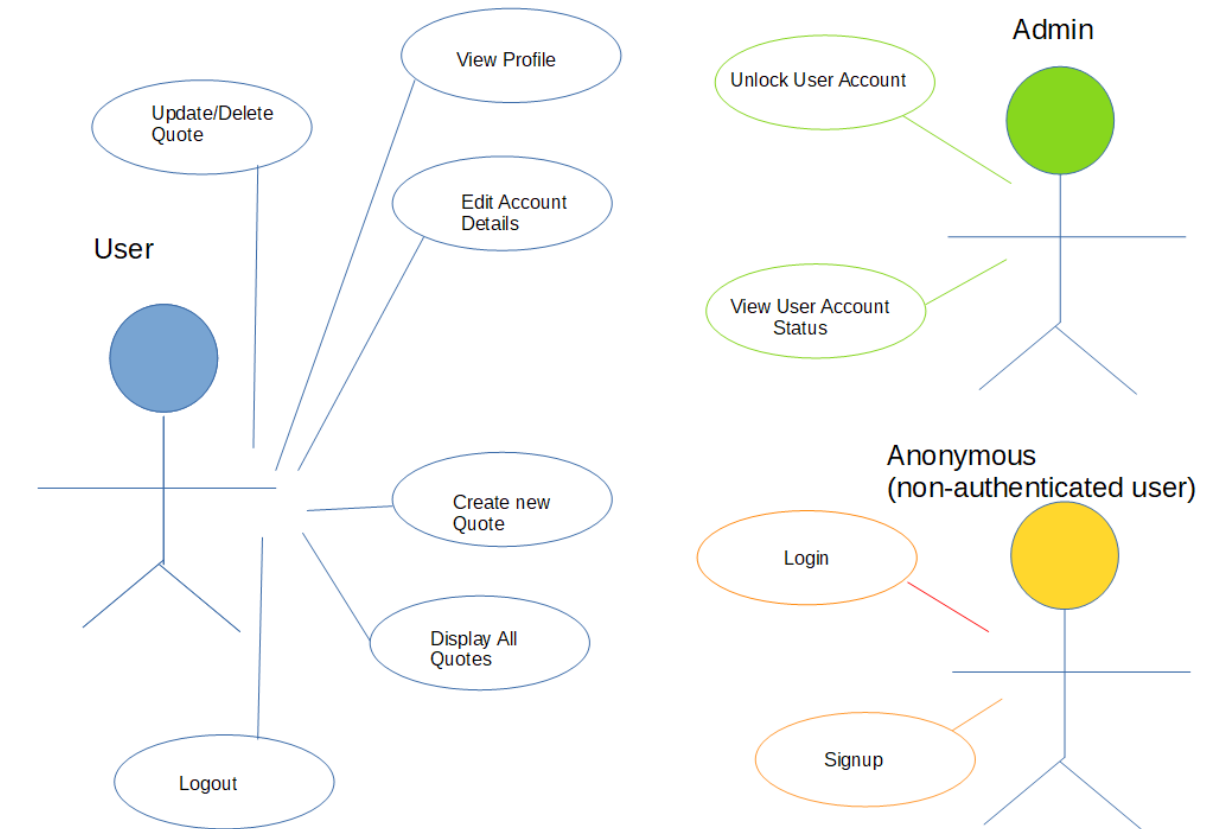
- Requirement #1: *The server must authenticate every request accessing the restricted Web pages with Json Web Tokens.*
- Requirement #2: The user must be able to perform create, delete, read and update operations on quotes.
- Requirement #3: Admins must be able to gain the current state of a given user's account and perform unlocking of their accounts where necessary.
- Requirement #4: Users should be able to update their account details when authenticated.
- Requirement #5: Users need to be able to signup and login to the application and receive a JWT token when authenticated.

#### **Functional Security Requirements**

- Requirement #1: Passwords must be stored in hashed format with a salt using a secure hashing algorithm.
- Requirement #2: Users are authorized to be able to act on entities for which they are the owner. This includes their own user data and their own quotes created.
- Requirement #3: Not authenticated users should only be able to access the signup and login pages.
- Requirement #4: Users that are authenticated will be able to access the user functionality and quote functionality only.
- Requirements #5: Changing a user's role with privilege escalation attempts should be protected against and prevented.
- Requirements #6: The application must protect data in transit with TLS.

- Requirements #7: The software must store passwords in a hashed format with a salt and utilize a high work factor to delay & frustrated attackers attempting to brute force or launch dictionary attacks.

#### 4.1.1. Use Case Diagram



This section shows the use case diagram of the system:

#### 4.2. Non-functional Requirements

This section lists the non-functional requirements which specify a quality or attribute that

**Note:** both users and admins can also login, that's encompassed here by all functionality that can be carried out by any non-authenticated user.

##### Non-Functional Security requirements

(1) Security property requirements which specify the properties that software must exhibit. E.g.,

*The software must ensure the integrity of the customer account information.*

*The software must log pertinent information of every request for security operations, security alerting and dev support.*

*The software must protect against brute force attacks.*

##### (2) Constraint/ Negative requirements

*The application must not accept invalid URLs.*

*The application must not allow concurrent sessions.*

*The application must reject any users without an invalid JWT token or an expired token.*

*The application must reject any user acting on an entity other than those for which it is the owner. This includes a given user's own quotes and user data. No other user should be able to modify or alter that data.*

*The application must not transport session data in plaintext.*

*The application must not allow for the cookie to be readable on the client side.*

*The application must not allow requests to come from origins other than the user interface with cross origin resource sharing.*

### **(3) Security Assurance Requirements**

*Default configurations of the database and application must be changed.*

*Leverage industry standard security frameworks and cryptographic algorithms.*

*The system must protect data and be GDPR compliant in it's handling of data only storing what's relevant for the given usage and no more.*

### **Non-functional Requirements**

*Maintainability: The system must be design to optimally allow changes in the future in a decoupled manner.*

*Portability: The system must be able to run on computers and android phones.*

*Failure Management: Errors should not reveal any sensitive information.*

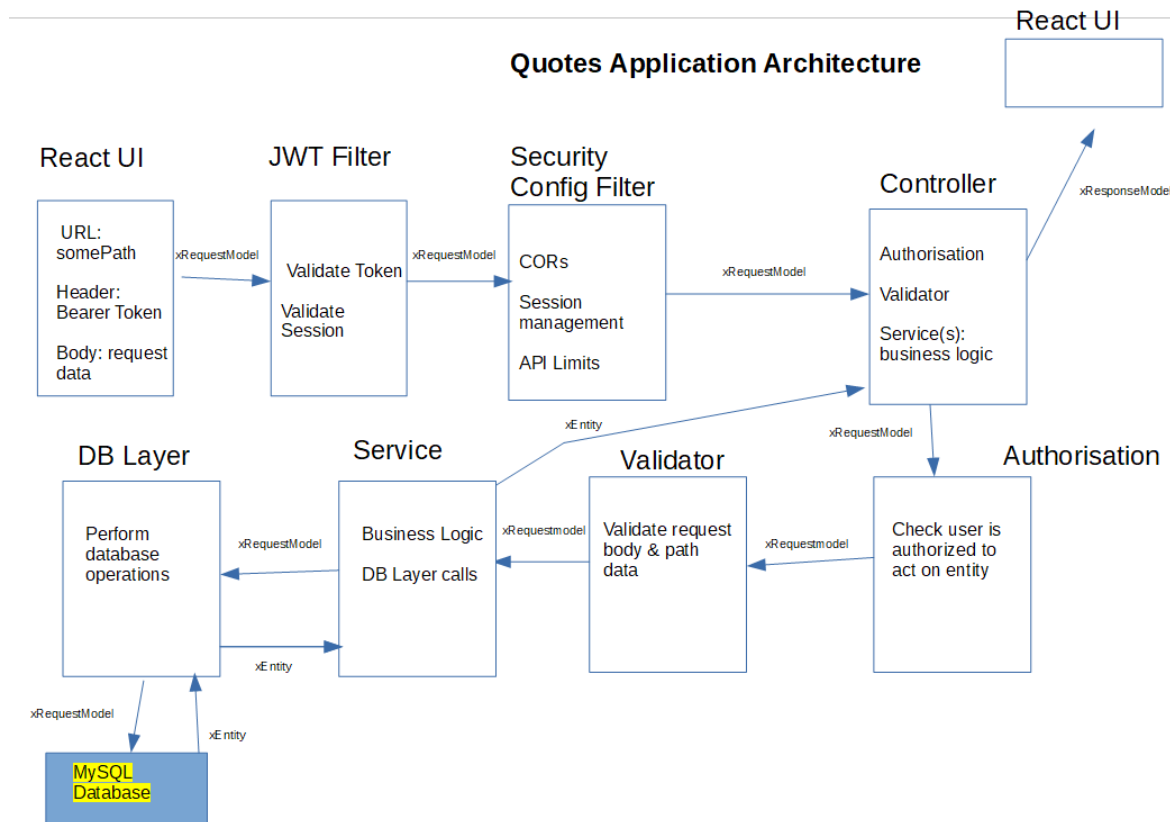
*Reliability: The system ought to perform without unexpected failure 95% of the time.*

*Availability: The system must be available to users 99% of the time.*



## 5.0 Design & Architecture

### 5.1. Architecture of the System



The architecture is constructed in such a way to leverage decoupling wherever possible. Dependency injection with Spring and autowiring the current implementations of various classes from underlying interfaces allows for this to be a seamless process. The uses of models with request models and response models, as well as entities all being distinct from one another all plays into the effect of having an architecture capable of evolution and iterative change while decreasing the levels of maintenance required with different layers maintaining independence from each other. They can grow or contract in their capabilities provided they meet the requirements of the next layer in sending through a model or entity. Services are based on interfaces to incorporate the base level expectations of functionality and allow for other implementations to be applied at a later stage where needed and can be easily swapped out with the `@Autowired` annotation provided by Spring Boot.

The database layer is based off of `JpaRepository` which provides a set of CRUD functionality for the entities through their interfaces as well as capabilities to expand beyond with prepared statements providing a level of one aspect of protection against SQL injection attacks executing the SQL queries separate to the introduction of the data provided to the queries. Beyond that, the DB access layer leverage the native methods and some custom queries where needed are created as well.

The models are converted into one another as they make their way through the various layers of the architecture sustaining only the data relevant for the execution on that part of the journey. This helps to protect against the unintended release of sensitive information for example as responses are curated to only maintain the necessary information needed and do not contain the same information as what's in the database for these entities. Models and entities only contain instance variables, setters and getters, toString and a conversion methods to setup the next level model or entity at the junctions of where their purpose ends and other starts.

### **Models:**

**SomeRequest ---> SomeEntity ---> SomeResponse**

*his benefit played itself out in the development process as well with initially using an in memory database in h2 for the sake of getting working functionality in place and then switching the long term database option and more production ready system in MySQL.*

The above architecture image depicts the flow of operations that takes place for almost all of the http requests. The exception being that the authorisation check does not take place for admins who have the permission to READ user's status and lock/unlock their account in the event of an account lockout. The path of each http request is checked in the JWT filter and for requests that are for unauthenticated users for the login page and the signup page these are allowed through without further investigation and the same applies at the security config filter level provided they have come from the UI interface with CORS still acting on them.

For other requests the JWT token is validated in the JWT Filter and checked to ensure it's both in the proper format and has not yet expired. The session is also checked as well for validity. The use of the JWT token provides additional protection against cross site request forgery attacks. The request is then passed onto the Security Config Filter which ensures that there is only one session in operation, generates a new session on login to protect against session fixation and disables url rewriting. It also invalidates sessions and deletes cookie data on logout. Extra protections are also in place for cookies and protect against XSS attacks and session hijacking with httponly and secure header flags set to true in the application.properties file. This ensures that the cookie can't be modified on the client side and encrypts the data in transit as well.

Attempts to access paths are checked at this point too and access is determined by roles. The application operates at a deny by default setup beyond these outlined paths that are accessible specifically by the correct role only.

### **Admin:**

admin/\*\*

### **sUser:**

quotes/\*\*

users/\*\*

auth/logout

### **Anonymous (non-authenticated users):**

auth/login

auth/signUp

Admins are allowed to access the admin controller only viewing user status with restricted information to that only and they are also able to lock or unlock user accounts at their discretion. Users are able to access the quotes API, users API, where they can perform CRUD operations on quotes and read, update account information and delete their account. Anonymous users can only access the login page or the signUp page.

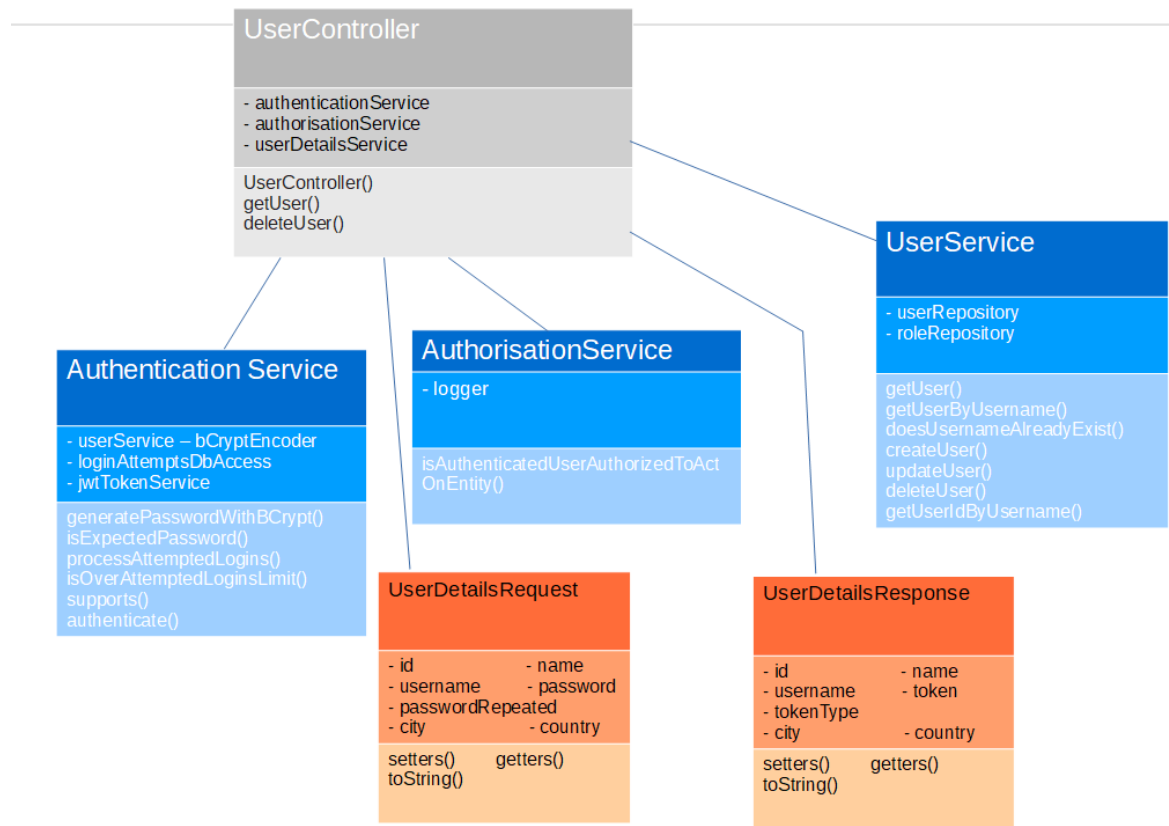
Beyond the filters the request reaches the controllers which orchestrate the operations that's to take place with the various services and validators. The controllers make use of one to many services like managers co-ordinating their many employees to carry out actions. On entrance to any of the API endpoints, the Authorisation service is called to ensure the user is authorised to act on the given entity. Then, the data of the request is validated with a corresponding validator that contains regexes tailored to the particular data in question. Finally then if these security checks are all passed, the data is deemed safe enough to pass onto the service layer where business logic operations take place. From there they then make their way to the db access layer to perform database operations.

The user interface also has protections in place. It makes use of `jsx` for all data that is rendered on the page. This helps protect against injection attacks since it escapes all of the data nullifying any injected malicious data. For example:

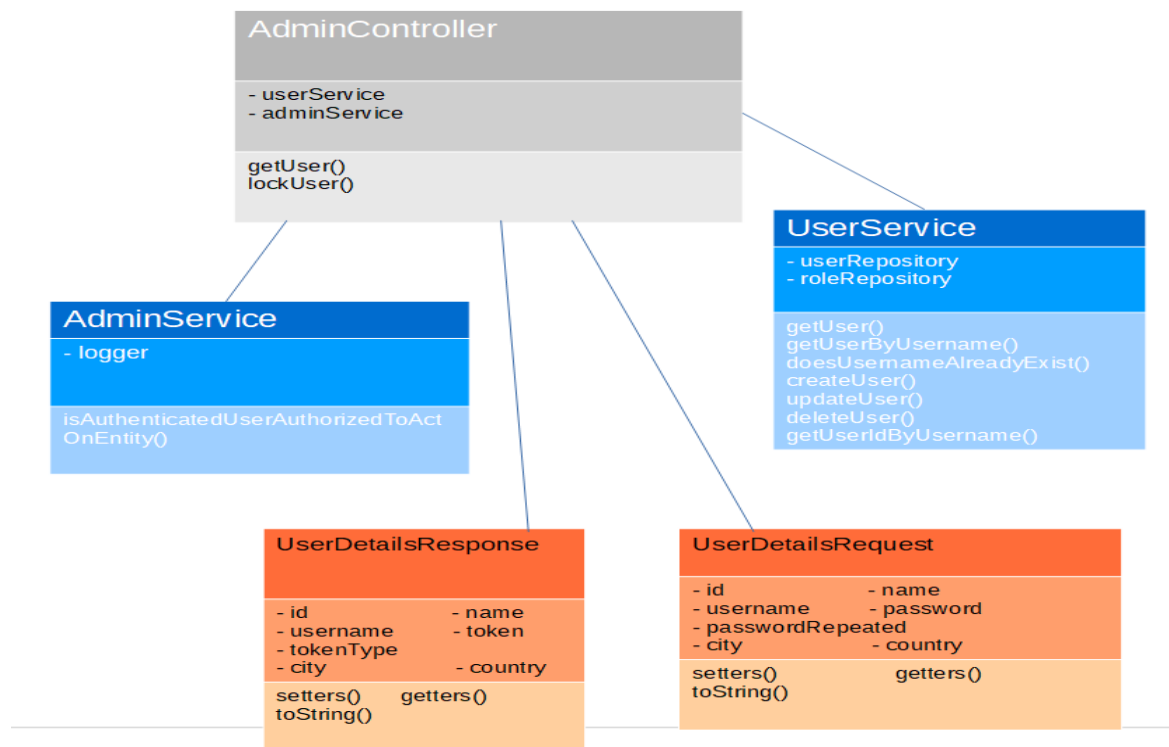
```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

Beyond that, the UI is also making use of `ProtectedRoutes` for any of the webpages that are accessed. They ensure to check that the `localStorage` contains `JWT` token for the given client before allowing any of the UI page to be rendered. When the token is not present, it instead redirects to login. The backend serves up resource when the business logic, validation, authentication and authorisation have taken place. It acts independently of the UI on the client side. Even in the event of a user being able to bypass this protection, their data would not be accessible unless they had managed to meet all of the aforementioned required on the backend that make its way through to the APIs and subsequently allow them access to the data. The protected routes are a defence in depth mechanism.

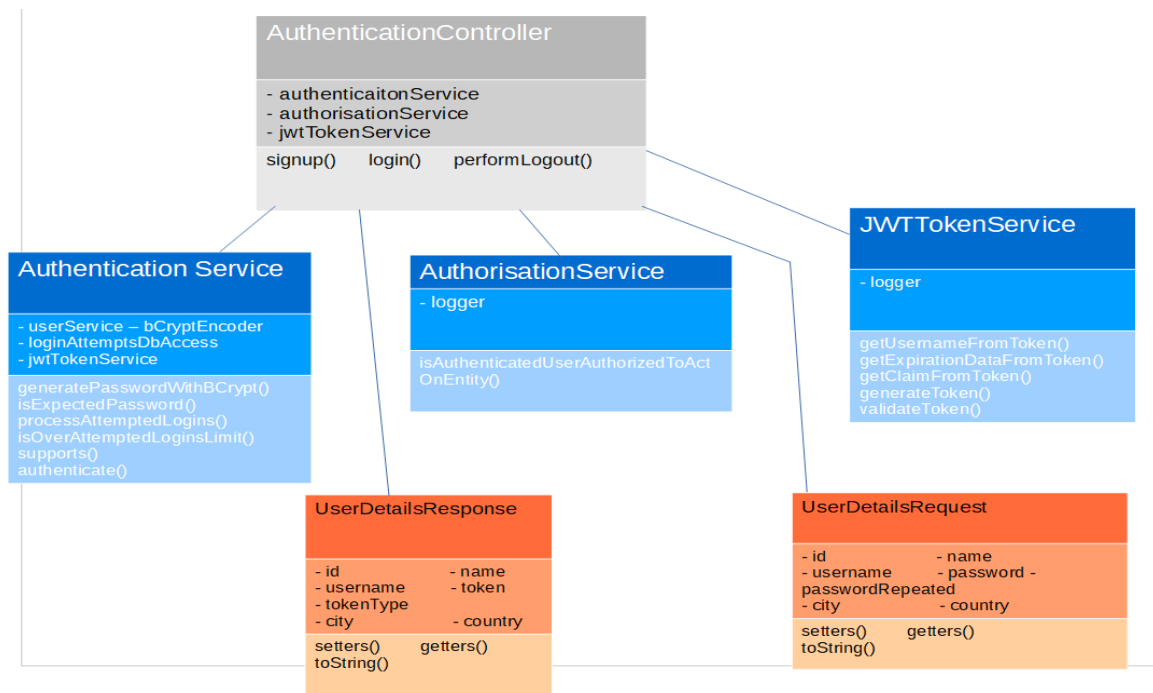
### 5.1.1 Class Diagrams



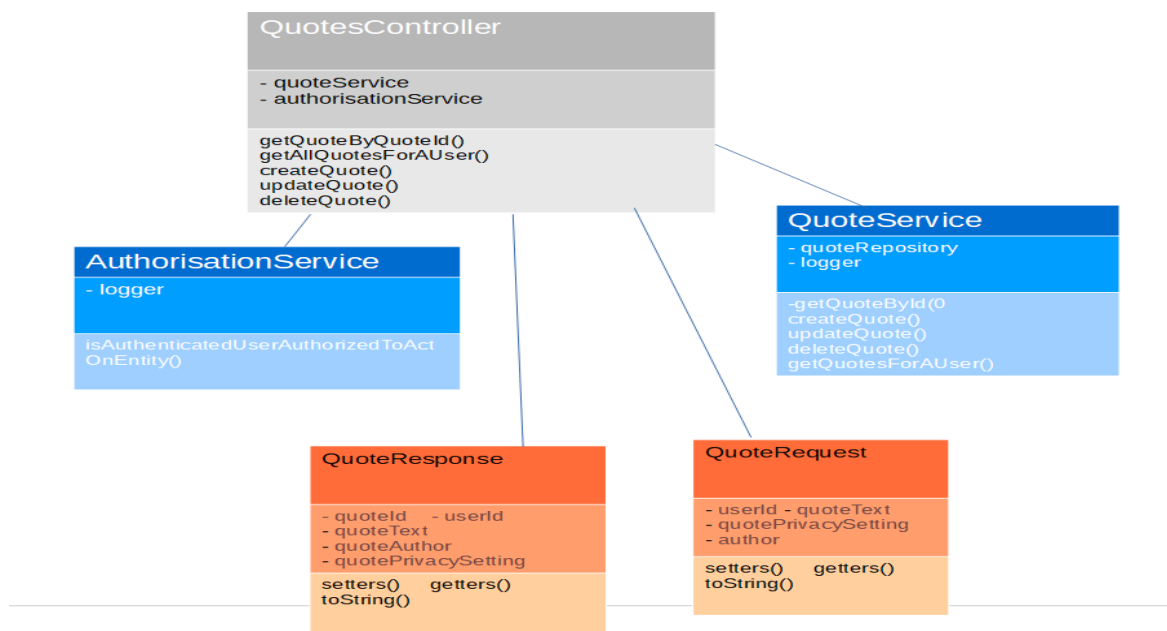
**User Controller Class Diagram**



**Admin Controller Class Diagram**



**Authentication Controller Class Diagram**



**Quote Controller Class Diagram**

## 5.2. Threat Modelling

Included in this section is how threat modelling was performed on the system as well as the appropriate diagrams such as Data Flow Diagrams, Abuse cases, and Misuse cases. Beyond that some mitigations and security controls to be put in place as well.

### 5.2.1. Step 1: Decompose the Application

**Backend Entry points to see where a potential attacker could interact with the application**

**Admin Role:**

admin/\*\*

**User Role:**

quotes/\*\*

users/\*\*

auth/logout

**Anonymous (non-authenticated users) Role:**

auth/login

auth/signUp

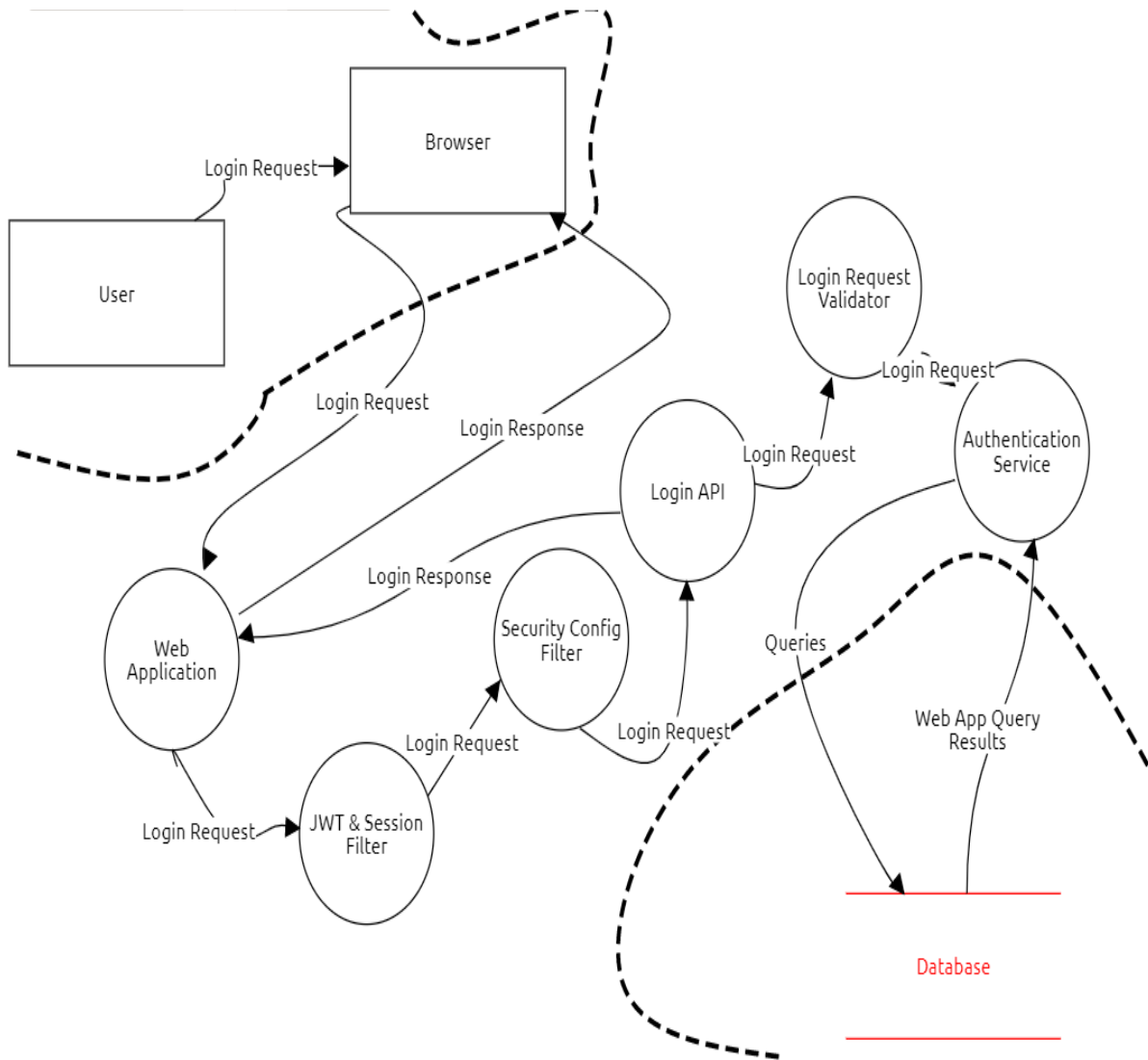
**Identify assets, i.e., items or areas that the attacker would be interested in.**

Cookie, session data, jwt tokens, user & admin credentials, user data, denial of service.

**Identifying trust levels that represent the access rights that the application will grant to external entities.**

By making use of role based access control (RBAC) it brings the advantage of a lot less overhead in operational maintenance. It's also a strictly defined set of privileges that a given entity can have. We're not going into a fine grained approach were it may end up inadvertently leading to some individuals operating at a higher level of privileges than they ought to be. It makes it easier to adhere to least privilege and ensure that no entity ends up falling through the cracks and gaining access/authorisation they should not have.

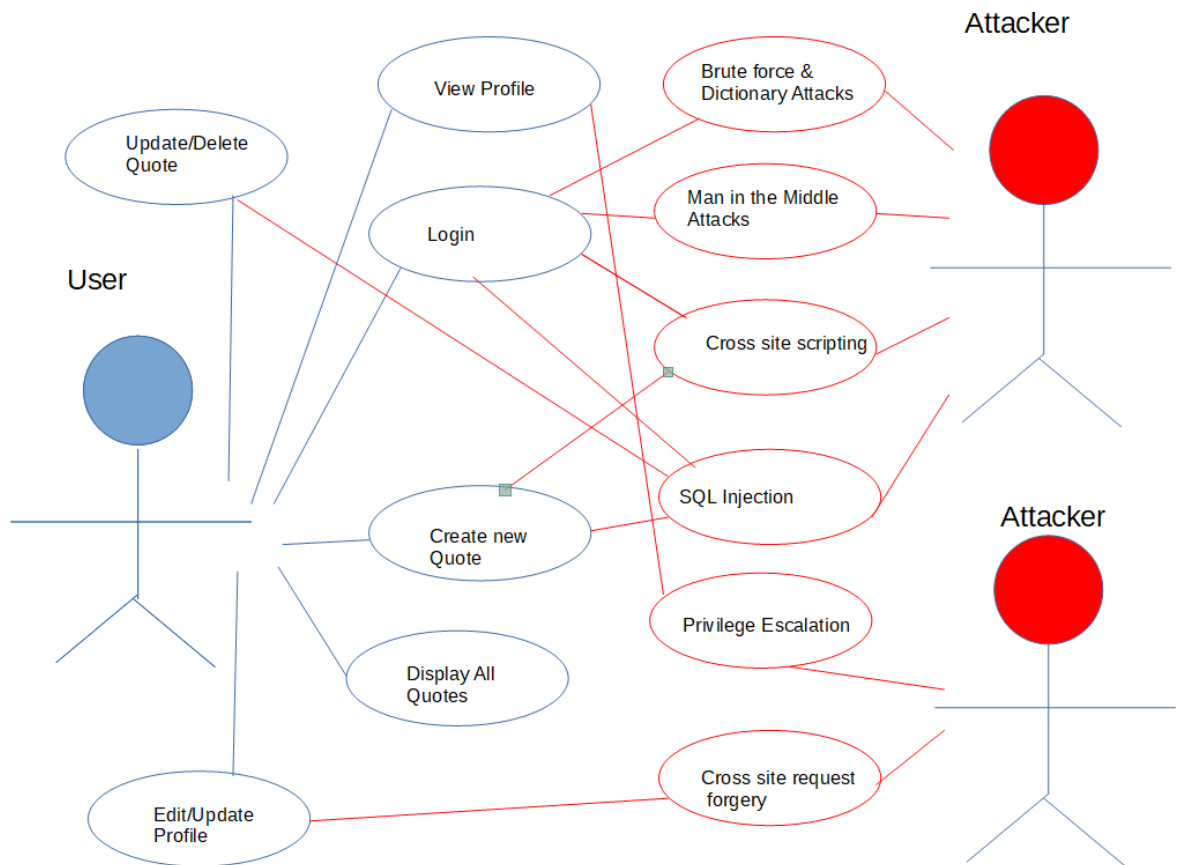
The authorisation would be handled at a number of places. The user interface will operate with protected routes which signify all of the routes that can be accessed for an authenticated user. The regular routes/urls like the signup page and login can be accessed without being authenticated. Beyond this the backend employs a sequence of filters from initially the JWT Filter asserting that the user/admin has a legitimate token, that the user has a legitimate session, that the correct role is in that token which matches what's in the db, that the user is the owner of the entity they are acting on etc.



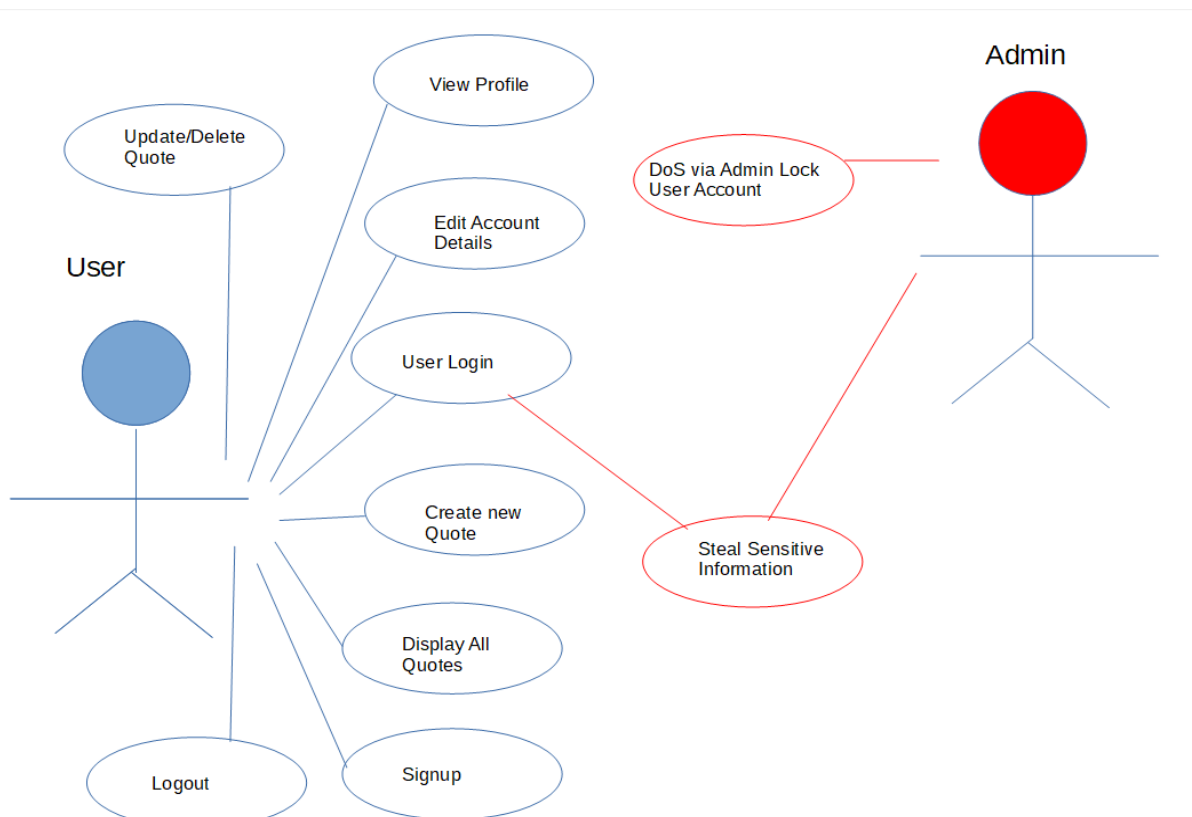
**Data Flow Diagram of a Login**

### 5.2.2. Step 2: Determine Threats

Here are some abuse cases and misuses cases below potential threat vectors for the application from both attackers and insider threats. This gives an opportunity to put our attacker hats on and start thinking like a hacker to identify potential threats and look to counteract those with effective security controls.

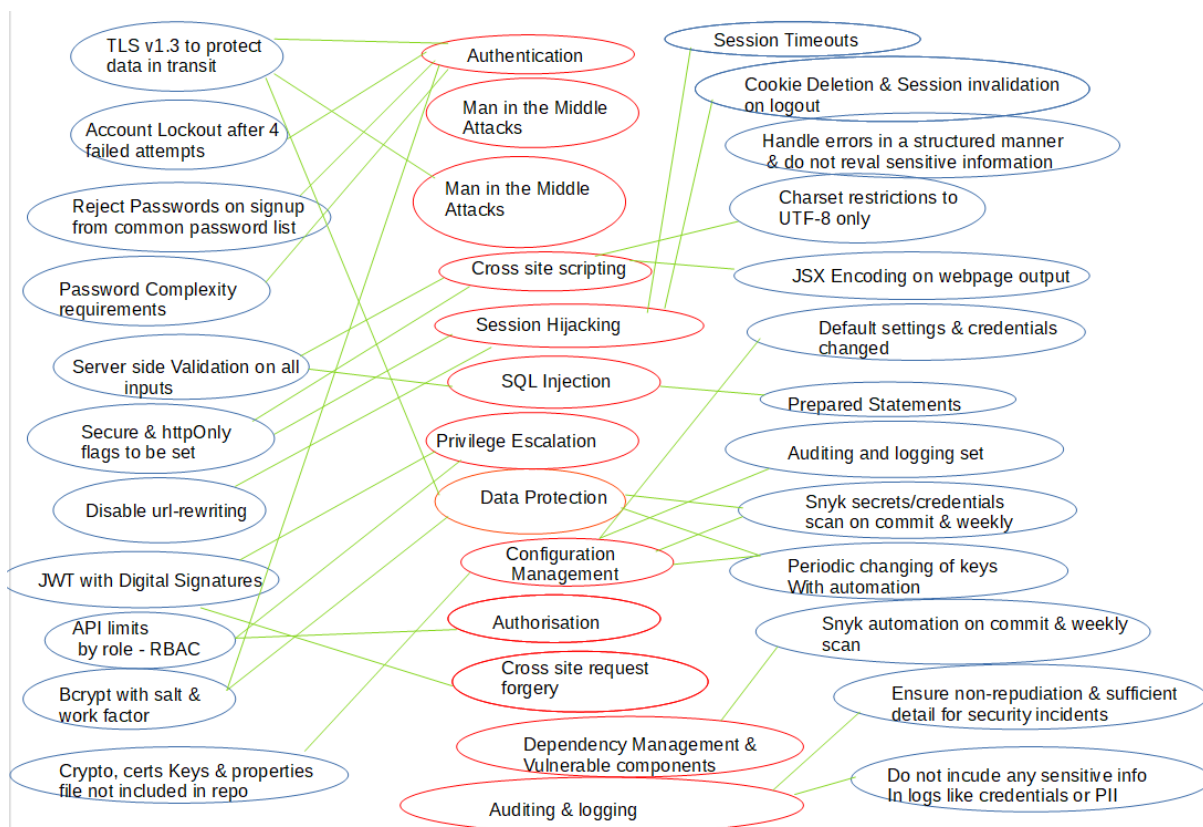


### Abuse case



### Misuse Case





## Mitigations for abuse and misuse cases

### Abuse cases:

- As an attacker, the following could be exploited with SQL injection payloads including the signUp, login, the create, read, update and delete operations on user's accounts. The same for quotes CRUD operations and also the operations admins are capable of performing. The database could be targeted for SQL injection as well as for stored XSS attacks.
- On top of this you have the cookie containing session data. As an attacker this could be targeted for session hijacking, session fixation attacks and reflected XSS attacks. The reflected XSS attacks could steal session data, deface the website or direct user's to other malicious sites and potentially phish for credentials. The user interface itself could be targeted for DOM based XSS attacks as well.
- As an attacker the data in transit could be snooped if not encrypted, and manipulated or stolen by an attacker. Attackers could obtain sensitive information in these cases as well.
- As an attacker they may also look to exploit any would be sensitive information enclosed in the JWT tokens as part of the authentication process.
- As an attacker they may look to perform privilege escalation and see if they could access functionality or endpoints for which they are not authorized.
- As an attacker the application also could be susceptible to brute force attacks on the login.

- As an attacker cross site request forgery as well for the various APIs in particular for the admin capabilities or any of the user's functionality.
- As an attacker they may look to exploit broken authentication mechanisms if any existed on the login & signup features.
- As an attacker if there is poor error handling /configuration in place and it reveals components with known vulnerabilities, they may look to leverage existing exploits.
- As an attacker they may look to exploit any misconfigurations if the database is for example directly accessible via some path allowed.
- As an attacker if weak cryptography algorithms are used they may look to reverse engineer what the data is & obtain sensitive information or system compromise through frequency analysis. They may also utilize rainbow tables for exploiting weak hashing algorithms.
- As an attacker if insufficient logging is not in place they may be able to continue their attacks without being recognized or compromise servers and systems without detection. They may also look to perform denial of service attacks and take the application offline.
- As an attacker if they are able to get access to the logging through system compromise or otherwise, if sensitive information is revealed in the logs, this could be utilized to further exploit the system and individuals on this application and other applications.
- As an attacker, denial of service could be caused by overwhelming the system with requests for which they can not manage the load and as result regular users can't use the application.

### **Misuse Case**

Disgruntled employees could look to cause a denial of service if they were to be allowed to both lock and unlock accounts and use that to lock out a whole host of accounts. Sufficient logging would need to be in place to identify whom is carrying out the actions and a certain low figure of accounts being locked within a short period of time could be used to set off an alert that could be addressed by a security analyst in a SOC. The proper details would need to be in place in the logs and accessible to only specific SIEMs. The legitimate use case for admins requiring this capability is for when there is an abusive user performing suspicious , so alternatively this could be addressed by some automation.

Disgruntled employees could also look to expose sensitive information online, this could be out of retribution, revenge or to obtain some money on a darkweb forum for example if there's a large enough database of credentials obtained. The loss of Personally Identifiable information could result in substantial fines by GDPR and the loss of credentials, could lead to a loss of customer trust and revenue.

The amount of information exposed to an admin needs to be restricted to that which is only required to their role when it comes to either unlocking an account (in the event of a lockout) or locking an account (in the event of breaching our site rules). So this should only be the status of the account, the user id and username, nothing beyond that. Again all

actions on any user account ought to be logged and tracked to obtain non-repudiation in terms of asserting which admin took any given action on a particular user account.

### 5.3. Security Design

The application has incorporated a whole security design principles and these are outlined in this section.

#### **Principle of Defense in Depth**

An example of where Defense in Depth is applied would be with session management. Poor management of sessions can lead to session fixation attacks, session hijacking, cross site scripting attacks to name a few concerns. Leveraging Spring Security's capabilities we're able to limit sessions to only one. Sessions are a long random value provided by JSESSIONID making them not guessable or brute forcible either. Sessions are changed for a new value after login to protect against session fixation attacks, sessions are invalidated on logout (along with all other cookie information). Sessions are also restricted to only 15 minutes (limiting the window of opportunity for any would be attacker if they were to somehow obtain them). The cookie flags of http only and secure are set to ensure sessions will only be sent over https, protecting the data in transit with TLS as done with this application as well. This provides confidentiality in transit and protects against man in the middle attacks. HttpOnly also ensures that cookies can't be modified protecting against XSS attacks and session hijacking. Disable url-writing is also set with Spring security to prevent the session id from being placed in the url. It will only be sent through the cookie, encrypted and not modifiable. These provide multiple layers of protection in an effort to protect against session based attacks and provide vast other protections as well.

*Files/classes that include most of the session management: SecurityConfig, AuthenticationController, JWTAuthenticationFilterImpl, application.properties*

#### **Principle of Least privilege**

Principle of least privilege ensures that only the minimum amount of privileges is given to an individual in order to perform their duties and no more. This has been applied in relation to authorisation. Role based access control is utilized in conjunction with Spring Security's capabilities. Users & admins are limited to the capabilities necessary to perform their roles and nothing more. Authenticated users are limited to the user API, quotes API and the subsequent functionality called from there. Users can only act on entities for which they are the owner. They can't act on other user's data or quotes. This authorisation check is performed after the user has been authenticated at the start of every API endpoint prior to validating data provided any performing any other actions. This includes CRUD actions on quotes and anything related to performing actions on their account whether that be reading data, updating or deleting data.

Admins are restricted to the Admin controller/API only. From there they can view user details as well as locking and unlocking user accounts in a support/security enforcement capacity. Unauthorized users are only able to reach the signup page and login page. All

three possible roles (USER, ADMIN, ANONYMOUS) are restricted to performing their own duties and capabilities and no more. Testing in

`PrivilegeEscalationAndRoutingComponentTest` asserts and ensures that this has been setup correctly and that each role only has their expected capabilities and no more adhering to this principle.

### **Principle of Fail Securely**

Failing securely through login (don't give info away), actions on entities. [25] Leveraging a whole host of testing from static analysis, component testing, integration testing and dynamic analysis testing we've looked to unearth how the system behaves under erroneous or irregular conditions as well as when being attacked. This includes firing non-ASCII characters at the application and much else with the dynamic testing. From this we're able to iterate over the current code implementation and change where insight has been given from these tests that there's security risk or vulnerability in the code. Results are analyzed, false positives discounted and legitimate vulnerabilities and improper failure conditions addressed.

### **Principle of Don't trust external input/services**

All APIs that take user input treat it as untrustworthy. As a result of this, validator classes have been put in place to reject any input that could look to compromise the security of the application or is deemed to be not of any acceptable nature. Prepared statements are acting with the JPA repositories and sql actions. On top of that we're using react on the frontend with jsx which provides additional protection in the event that any data were to make it's way through the validation checks, it would not output, as jsx escapes any data being output.

### **Principle of Secure Defaults**

Privilege escalation from user to admin are also not possible and protected against through logic on the database access layer preventing any change from a user's initial role. User's are required to meet stringent password complexity requirements. Usernames are only allowed to be in the format of an email address as one part of mitigating against attempts at injection. Quotes are set to private by default meaning only the creator of the quote can view/modify/delete the quote. Database defaults have been changed including login.

### **Principle of Avoid security by obscurity**

Security through obscurity is not included in the design and as a result the codebase is an openly accessible platform on github. It's not relying on hiding the code as a means of security. However, the cryptographic keys and application.properties file is not included in the repo. These have been entered into a gitignore file to ensure that they do not make their way onto the repo. This also avoids the common pitfall of including your passwords or keys within your product. There's also no reliance on any hidden urls through this open

design approach or relying on encoding. No sensitive information is included in JWT tokens as well.

#### **Additional Privilege Applied: Minimize Attack Surface & Deny by Default:**

Requests must have a legitimate JWT token that is not expired before it can make it through the first filter. Requests are denied by default leveraging Spring Security's in the second filter functionality to `denyAll()`. An allowlist based on specific roles and whether the user is authenticated or not will be utilized thus drastically reducing the attack surface and not falling into the common security pitfall of opting for a blacklist approach. On top of all of this, cross origin resource sharing is to limit requests to those coming from the UI (currently `localhost:3000`).

#### **Additional Privilege Applied: Avoid Rolling Your Own Cryptography**

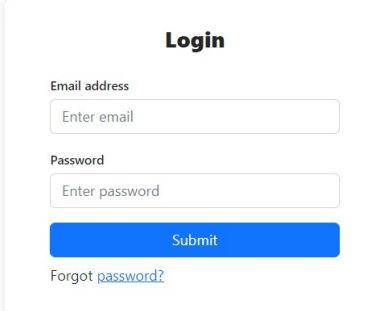
The system makes use of industry standard algorithms for securing data at rest and in transit. These include `bcrypt` for the storing of passwords and `TLS v1.3` for the secure transmission of data.

### **5.4. Graphical User Interface (GUI)**

Screenshots of a key screens and a brief explanation of what can be seen:

#### **Quotes**

*For all you witty quotaholics out there.*

A screenshot of a login form titled "Login". It features two input fields: "Email address" with a placeholder "Enter email" and "Password" with a placeholder "Enter password". Below the password field is a blue "Submit" button. At the bottom, there is a link that says "Forgot [password?](#)".

**Login UI Page** -> login form where credentials can be entered and also a Forgot Password link to another UI page.

## **6.0 Implementation**

Spring security provide the infrastructure to intercept requests and ensure that whomever made the request has the authorisation to do so and is properly authenticated. It helps in protecting data at rest with encryption and hashing options and also helps with setting up

protections for data in transit as well ensuring that data is only accessed by those with the necessary privileges to do so. It provides ways of producing security functionality related to session management, authentication, authorisation, cross site scripting protection, CSRF protection, amongst other aspects decreasing the amount of code required to perform these actions while giving a standardized framework that's widely deployed universally and has stood the test of time in helping to provide security functionality for applications.

Generally, at the application level, one of the most encountered use cases is when you're deciding whether someone is allowed to perform an action or use some piece of data. Based on configurations, you write Spring Security components that intercept the requests and that ensure whoever makes the requests has permission to access protected resources. [3]

http

```
        .httpBasic()

        .authenticationEntryPoint(new
NoPopupBasicAuthenticationEntryPoint())

        .and()

        .authorizeRequests()

        .mvcMatchers("/admin/**").hasRole("ADMIN")

        .mvcMatchers("/auth/logout").hasRole("USER")

        .mvcMatchers("/quotes/**").hasRole("USER")

        .mvcMatchers("/users/**").hasAnyRole("USER")

        .mvcMatchers("/auth/signUp").hasAnyRole("ANONYMOUS")

        .mvcMatchers("/auth/login").hasAnyRole("ANONYMOUS")

        .anyRequest().denyAll();
```

Above is some of the code from SecurityConfig, we can see the deny by default approach for APIs ensuring that no other paths can be accessible other than what's listed here. It also depicts the roles necessary to access certain APIs and thus functionality and as a result the privileges allowed for each role. Anonymous represents non-authenticated users and users and admin are both roles that are authenticated users.

```
http.csrf().disable()

        .sessionManagement()

        .sessionCreationPolicy(SessionCreationPolicy.ALWAYS)

        .maximumSessions(1).and().sessionFixation().changeSessionId()

        .and().logout().logoutRequestMatcher(new
AntPathRequestMatcher(LOGOUT_URI))

        .deleteCookies(JSESSIONID).invalidateHttpSession(true)

        .clearAuthentication(true);
```

Above we have some of the session management code of the SecurityConfig and also the actions to be taken on logout. Sessions are limited to a single session, session ids are changed on login to protect from session fixation and every request has a session id. Cookies are deleted on logout and sessions are invalidated, the security context which stores authentication information for Spring Security is also cleared.

Here is the additional session based settings from the application.properties. Here we can see the http-only and secure flags set for xss protection, url re-writing is disabled and sessions must be in the cookie only and timeout in 15 minutes.

```
# session management, xss protection & disable url re-writing to include the session id
```

```
server.servlet.session.cookie.http-only=true
```

```
server.servlet.session.cookie.secure=true
```

```
server.servlet.session.tracking-modes=cookie
```

```
server.servlet.session.timeout=15m
```

```
server.servlet.session.disable-url-rewriting=true
```

Note: application.properties is not included in the git repo as this would be bad security practice. The relevant parts of it are included where needed omitting credentials.

```
CorsConfiguration configuration = new CorsConfiguration();
```

```
configuration.setAllowedOrigins(Collections.singletonList("http://localhost:3000"));
```

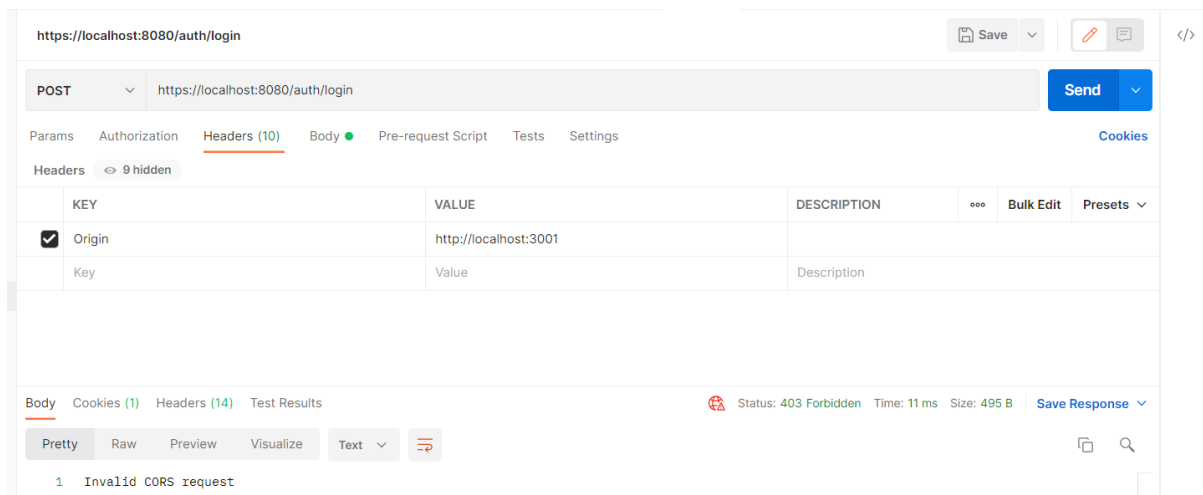
```
configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"));
```

```
configuration.setExposedHeaders(Arrays.asList("Authorization", "content-type"));
```

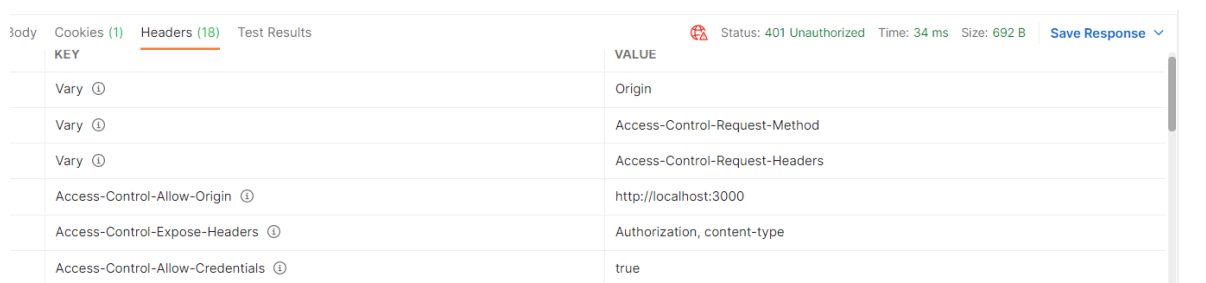
```
configuration.setAllowedHeaders(Arrays.asList("Authorization", "content-type"));
```

```
configuration.setAllowCredentials(true);
```

This is also in the Security config and it's the Cross origin Request policy, limiting requests to the UI only as well as limiting the http verbs and the allowed headers in both requests and responses to be only two. To be noted, CORS offers the client the headers and they can choose not to apply them. Browsers generally do apply them but tools like Postman and ZAP need to be configured to have the Origin header set. Here we can see http://localhost:3001 being rejected as the origin.



However when we change that to `http://localhost:3000` which is on the allowlist, the request is allowed through the CORs filter and we now get the CORs filter displayed in the response to the request:



Also from the browser, we can validate a cors error with using `127.0.0.1`



`127.0.0.1` is the equivalent of `localhost`, but the cors check looks specifically for the string. This validates that it's working and `127.0.0.1` will also be added to the list of allowed origins.

Spring boot offers the powerful capabilities of Spring native without the higher levels of maintenance required with managing every expanding configuration files. This done through simple annotations applied at the class level. It also provides the benefit of speed in being able to get up and running fast with a tomcat server running the application.



## JWT

**Json Web Tokens** provide a great additional token that's required for authenticated entities. The data stored within the tokens is stored in base 64 encoding and is made up of three parts, the header, the body and the digital signature.

```
eyJhbGciOiJIUzI1NiIsInR5cGU6IiwiZW50bGljaWVkaWVudC5jb20iLCJyb2xlcyl6IiwiaWF0IjE5MTY3MDU1ODY2NSwiZXhwIjoxNjcwNTU5NTY1fQ.IN0rhV0SvPb75C0HLZsWUiaY50-JjK0obHA17eAhT0g
```

Above we have the base 64 token and below we have that same token decoded:

```
{"alg":"HS256","sub":"simon@gmail.com","roles":["ROLE_USER"],"iat":1670558665,"exp":1670559565} M4U+oBr'e□it□^□□
```

As you can see the header shows the algorithm that was used, the body contains the username, role, created at date & time (iat) and expire date and time. Outside of the brackets we can see some non-ASCII characters and this is the digital signature that accompanies every JWT token.

The digital signature is checked as part of authenticating every request and if the contents of the JWT token were changed (for example changing the role of USER to ADMIN or changing the expiry date to an indefinite period), the expected digital signature would no longer match and as a result, the request would be rejected. The digital signature has been encrypted with a key and HMAC SHA-256 creating a result that would only be viewable with that key and as such, this ensures the authenticity of the digital signature and also the integrity of data within the JWT token. [1] The use of JWT makes the use of a CSRF token redundant since it serves the same purpose and protects and cross site request forgery.

The following classes: `JWTAuthenticationFilter` and `JWTTokenServiceImpl` contain the JWT related code.

Application.properties settings relevant to JWT setup:

```
# jwt tokens
jwt.token.validity=900
jwt.signing.key=
jwt.authorities.key=roles
jwt.token.prefix=Bearer
jwt.header.string=Authorization
```

### Password Implementation:

Utilizing bcrypt for password hashing is due to it's strength relative to other options. It's stronger than PBKDF2, as well as argon2 and scrypt for memory requirements that are less than 4MB which is the case here. [10]

Requiring passwords to be of length 11 or greater provides great protection against brute force attacks. With the complexity requirements required the total permutations of the 63

possible characters for a password of length 11 goes to 43,513,917,611,435,838,661 possible length 11 passwords. The jump from length 10 to length 11 passwords is exponential as shown here in this image:

PASSWORD LENGTH	POSSIBLE COMBINATIONS	TIME TO CRACK S = SECONDS H = HOURS M = MINUTES Y = YEARS
4	45697	<1 S
5	11881376	<1 S
6	308915776	<1 S
7	8031810176	~4 S
8	208827064576	~1.5 M
9	5429503678976	~45 M
10	1411677095653376	~19 H
11	3670344486987780	~.1 Y
*12	95428956661682200	~1.5 Y
13	248115287320374E4	~39.3 Y
14	645099747032972E5	~1,022.8 Y
15	167725934228573E7	~26,592.8 Y
16	436087428994289E8	~691,412.1 Y
17	113382731538515E10	~17,976,714 Y
18	2947951020001390E10	~467,394,568 Y

The password complexity requires 11 to 20 characters, one uppercase, lowercase, number, and a special character. These are criticized due to people changing specific letters to numbers. To compound this, the password validator also rejects commonly used passwords from a list of 1575 passwords online. These passwords were found to be used at least 250 times each providing a broad spectrum of obvious passwords a lot of people tend to use and filtering them out as a possibility. They also contain the very values that can be candidates like Pa55word to protect against their use as well while still maintaining the huge number of possible choices that having 61 possible characters for each letter of the 11 to 20 password length provides. [6]

**MySQL** is used as the database option. It offers multiple authentication methods including Old Password Authentication and Secure Password Authentication [9]. With Secure Password Authentication, it utilises SHA1 hashing to store passwords. MySQL supports SSL encryption (not in operation by default for performance reasons). It deploys firewall rules for blocking traffic. It has a high level of auditing whereby all actions end up logged. From authentication and sign out attempts, to efforts access a given table, also altering database schema. It offers both offline and online backup options. It also offers features such as MFA, account resource limits, privilege restrictions, account locking and password. They are however susceptible to SQL injection attacks, DDoS attacks if attackers overwhelm the database from multiple accounts, race conditions.

SQL databases also offer greater data reliability and integrity. They require vertical scaling which can be a drawback on one availability metric. On most security metrics a SQL database like MySQL is going to offer a much greater suite of capabilities than a NoSQL offering like MongoDB. They have incorporated security as part of their design, features and benefits and not as a retrospective compensation for not valuing security enough.

**Maven** is used as the dependency manager program. It helps to obtain the necessary dependencies required for the application and is maintained in the pom.xml of the application. It provides a centralized way of keeping all of the dependencies up to date, it's a big part of the Software Bill of Materials (SBOM) that's becoming increasingly more relevant as attackers look to target low hanging fruit of dependencies that have not been kept up to date and have known security vulnerabilities in them. [4] it's used in conjunction with the findings of Snyk which highlights any dependencies which have security vulnerabilities as well as the versions which have those vulnerabilities fixed in, allowing us a means of keeping our dependencies as secure as possible.

**TLS**, Transport Layer Security has been setup within the application using self-signed certificate. This would be swapped out for a certificate generated by a trusted certificate authority for moving towards a production system. TLS is set to version 1.3 which is the latest version and doesn't contain any known vulnerabilities unlike some of the earlier versions of both SSL and TLS with vulnerabilities such as the heartbleed vulnerability.

The most recent version of TLS features:

- o Enhanced security encryption and hashing
- o No backward compatibility
- o HMAC
- o More efficient handshake process
- o Eliminates compromised cipher suites

It provides our application with much needed protection for data in transit.

Here are the application.properties settings for this with credentials and keynames removed:

```
# TLS configuration
server.ssl.enabled=true
server.ssl.key-store-type=
server.ssl.key-store=
server.ssl.key-store-password=
server.ssl.key-password=
server.ssl.protocol=TLS
server.ssl.enabled-protocols=TLSv1.3
```

[11]

**Logging and auditing** is carried out by Spring Sleuth and slf4j. Spring sleuth provides output for each logging message that contains the application name, trace id and span id. The trace id is particularly important for both security concerns and development support. Every request that goes through will have one unique trace id and from that we're able to see all of the services and actions that occur on that given request. This could be particularly useful in the event of a security incident or alert. It will allow security personnel to be able to trace what happened for throughout a given request to the application. The logger also contains a timestamp with the time and date included which again can be important information in a security alert and to add to this, the IP address of the client making the request is also logged at the beginning of every request when they reach the JWT filter, which is the first filter all requests reach on arrival to the application. So if a given attacker is performing malicious actions over many different requests, we'll be able to combine these pieces of information from the IP, timestamp, trace id and accompanying actions to create a picture of what the attacker has been doing. It also provides non-repudiation and in the event of even a malicious insider performing actions through an admin role, this will also be tracked. Here below we have an output example of a request reaching our JWT filter with a timestamp included, an ip and a trace id:

```
2022-12-11 23:07:40.317 INFO [quotes,,] 17456 --- [main] c.s.q.i.QuotesControllerIntegrationTest : Entered test: deleteQuote_WithLegitData_ThenSucceed
2022-12-11 23:07:40.317 INFO [quotes,fb463f69bb8af7af,fb463f69bb8af7af] 17456 --- [main] c.s.quotes.JWTAuthenticationFilter : Entered JWT Filter for request.
2022-12-11 23:07:40.317 INFO [quotes,fb463f69bb8af7af,fb463f69bb8af7af] 17456 --- [main] c.s.quotes.JWTAuthenticationFilter : IP address of sender of request: 127.0.0.1
2022-12-11 23:07:40.317 INFO [quotes,fb463f69bb8af7af,fb463f69bb8af7af] 17456 --- [main] c.s.quotes.JWTAuthenticationFilter : Note: If the system is behind a proxy, this will be the IP of the proxy.
```

Since this is one of the integration tests running, the IP is just our loopback address of the my current system, this will be a legitimate public facing IP in the event of anyone else making requests on the application in a live setting. All input entered into the logs (like user ids for example) from the controller onwards is validated and thus non-malicious data.

**Git** is also used for version control, offering a means to sustain the availability of the codebase and ensure that a backup is always available online. **Postman** and **Curl** are also used for API calls and manual testing.

## 7.0 Testing

This application leveraged the use of both integration tests and component testing. Unit testing did not suit in this case as it's not an algorithm heavy application. Instead, we get full visibility of how the application performs it's most important duties from api endpoint to response and after all the subsequent layers have been passed through. Beyond that component testing has served place particular emphasis on certain high priority aspects of the applications, whether that be ensuring the path and authorisation access of different roles as well as non-authenticated users to brute force attacks on the login, to xss on the

various inputs and so on. The vast suite of testing including both integration and component testing has culminated in 20+ tests which run on each build.

On top of that there is also the SAST tool Snyk that's run on each commit and also set to run on a weekly basis via automation. It checks for code based vulnerabilities, dependency vulnerabilities in the various jars used and generated by maven, secrets/credentials in the code and container vulnerabilities as well.

There is also the use of DAST with OWASP which is outlined further below. To compound there, there are additional security test cases added in line with assertions for the OWASP Penetration Testing checklist utilized to compound the security effort of finding flaws with an attackers mentality and testing.

### Some security test cases

#### **1. Privilege escalation and authorisation on API paths and functionality**

These scenarios and tests are all in the class:

`PrivilegeEscalationAndRoutingComponentTest`

**- A user with the role of anonymous (not authenticated) should be able to access the signup api, login api and no other API.**

See the following test:

`whenNotAuthenticated_thenAllowSignupButNoAdminOrUserEndpointsAllowed`

authorisation -> routes,

**- A user with the role of USER should not be able to access admin functionality.**

See the following test:

`requestProtectedAdminUrlsWithRegularUser_ThrowException`

**- A user with the role of ADMIN should be able to access admin controller functionality but not the user controller or quote controller.**

See the following test:

`requestProtectedUrlWithAdmin`

**- A user with the role of USER should be able to access the user controller functionality and the quote controller functionality.**

`givenUserRole_WhenAccessingUserAndQuotesAPIs_AllowActions`

#### **2. A user should not be able to act on entities belonging to another user.**

Test Class:

`UserControllerIntegrationTest`

See test:

```
getUser_WithAuthenticatedButNotAuthorisedUser_ThenThrowException
```

### **3. A user who attempts to brute force login on an account should be locked out.**

TestClass:

```
AuthenticationControllerIntegrationTest
```

Test

```
testLogin_WithBadCredentialsFourTimes_ThenExpectAccountTobeLocked
```

### **4. A password generated should be in hashed format:**

Test class:

```
AuthenticationServiceComponentTest
```

Test:

```
testGeneratePasswordHashAndItsContentsAndIsPlaintextPasswordHashedEqualsToIt
```

### **5. Ensure session ids are changed on login to protect against session fixation attacks:**

Test Class:

```
SessionsComponentTest
```

Test:

```
sessionReuseAndSessionFixation_whenTheUserLogsInANewSessionIsGenerated
```

**Note: A whole host of other security testing and considerations are carried out in relation to the Penetration Testing Checklist later in the report.**

**Sample test output:**

```

12-16 14:33:26.861 INFO [quotes,,] 2884 --- [main] c.s.q.i.AdminControllerIntegrationTest : Entered test: testLockUser_WithAdminWithLockingAndUnlockingAUser_ExpectSuccess
12-16 14:33:26.863 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Entered JWT Filter for request.
12-16 14:33:26.863 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : IP address of sender of request: 127.0.0.1
12-16 14:33:26.863 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Note: If the system is behind a proxy, this will be the IP of the proxy
12-16 14:33:26.863 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Path requested for request: /admin/lockUser
12-16 14:33:26.864 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Pre-get user with username from token
12-16 14:33:26.865 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.DbAccessLayer.UserDBAccess : Entered method: loadUserByUsername with username: jesus@godmail.com
12-16 14:33:26.873 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.DbAccessLayer.UserDBAccess : Exiting method: loadUserByUsername
12-16 14:33:26.877 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Post validate token, user role: ADMIN
12-16 14:33:26.880 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.services.AdminServiceImpl : Entered updateUser
12-16 14:33:26.917 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.services.AdminServiceImpl : Exiting method updateUser
12-16 14:33:26.920 INFO [quotes,64dcfa5b38409741,64dcfa5b38409741] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Successfully validated jwt token filter. Leaving jwt token filter
12-16 14:33:26.923 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Entered JWT Filter for request.
12-16 14:33:26.923 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : IP address of sender of request: 127.0.0.1
12-16 14:33:26.923 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Note: If the system is behind a proxy, this will be the IP of the proxy
12-16 14:33:26.923 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Path requested for request: /admin/lockUser
12-16 14:33:26.923 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Pre-get user with username from token
12-16 14:33:26.925 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.DbAccessLayer.UserDBAccess : Entered method: loadUserByUsername with username: jesus@godmail.com
12-16 14:33:26.934 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.DbAccessLayer.UserDBAccess : Exiting method: loadUserByUsername
12-16 14:33:26.938 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Post validate token, user role: ADMIN
12-16 14:33:26.941 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.services.AdminServiceImpl : Entered updateUser
12-16 14:33:26.967 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.services.AdminServiceImpl : Exiting method updateUser
12-16 14:33:26.970 INFO [quotes,4767d811b865be4f,4767d811b865be4f] 2884 --- [main] c.s.quotes.JwtAuthenticationFilter : Successfully validated jwt token filter. Leaving jwt token filter
12-16 14:33:26.971 INFO [quotes,,] 2884 --- [main] c.s.q.i.AdminControllerIntegrationTest : Completed test testLockUser_WithAdminWithLockingAndUnlockingAUser_ExpectSuccess

```

*This is the test case: testLockUser\_WithAdminWithLockingAndUnlockingAUser\_ExpectSuccess*

**Overall results from running all 25 component test cases and integration test cases:**

```

[INFO] Results:
[INFO]
[INFO] Tests run: 26, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- maven-jar-plugin:3.2.2:jar (default-jar) @ quotes ---
[INFO] Building jar: C:\Users\simon\Documents\Programming Code\My Programs\Cybermasters-NCI\Cybersecurity-Masters-NCI\Secure Programming for Web\Project\quotes\target\quotes-0.0.1-SNAPSHOT.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:2.7.4:repackage (repackage) @ quotes ---
[INFO] Replacing main artifact with repackaged archive
[INFO]
[INFO] --- maven-install-plugin:2.5.2:install (default-install) @ quotes ---
[INFO] Installing C:\Users\simon\Documents\Programming Code\My Programs\Cybermasters-NCI\Cybersecurity-Masters-NCI\Secure Programming for Web\Project\quotes\target\quotes-0.0.1-SNAPSHOT.jar to C:\Users\simon\m2\repository\com\spfw-project\quotes\0.0.1-SNAPSHOT\quotes-0.0.1-SNAPSHOT.jar
[INFO] Installing C:\Users\simon\Documents\Programming Code\My Programs\Cybermasters-NCI\Cybersecurity-Masters-NCI\Secure Programming for Web\Project\quotes\pom.xml to C:\Users\simon\m2\repository\com\spfw-project\quotes\0.0.1-SNAPSHOT\quotes-0.0.1-SNAPSHOT.pom
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 49.155 s
[INFO] Finished at: 2022-12-18T14:07:04Z

```

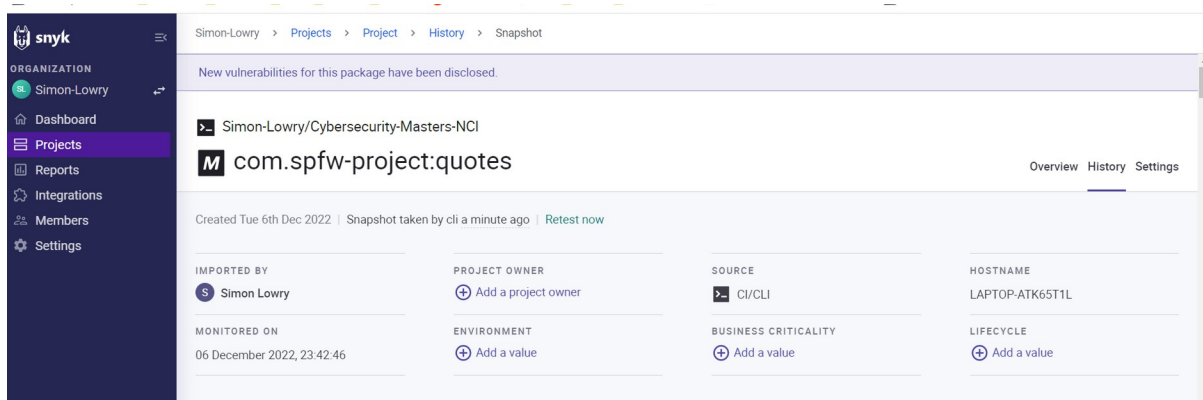
*25 test cases run, no errors or failures, all succeeded.*

## Static Application Security Testing

SAST testing was done with the static application security testing tool Snky. Snky helps to identify and fix security vulnerabilities in a given project during our development lifecycle. It helps to make security reviews smoother and reduce the cost of fixes further in the development phase. It provides free results on code, containers and configurations and leverages the snky vulnerability database to help find open source vulnerabilities. Snky has been integrated with my github repo and configured to run weekly automatically and then there will run on each git push to the repo. It will help identify both dependency related security issues and code related security issues. For the dependency changes of versions to address security concerns, these will need to be tested locally to ensure that they have not caused a breaking of functionality and then pushed to the repo subsequently when this is clear. The code related security issues raised must be addressed by rank first and processed to determine whether they are false positives or not. For those that are false positives, these are to be clicked as ignored in Snky with valid reasoning and the rest immediately addressed.

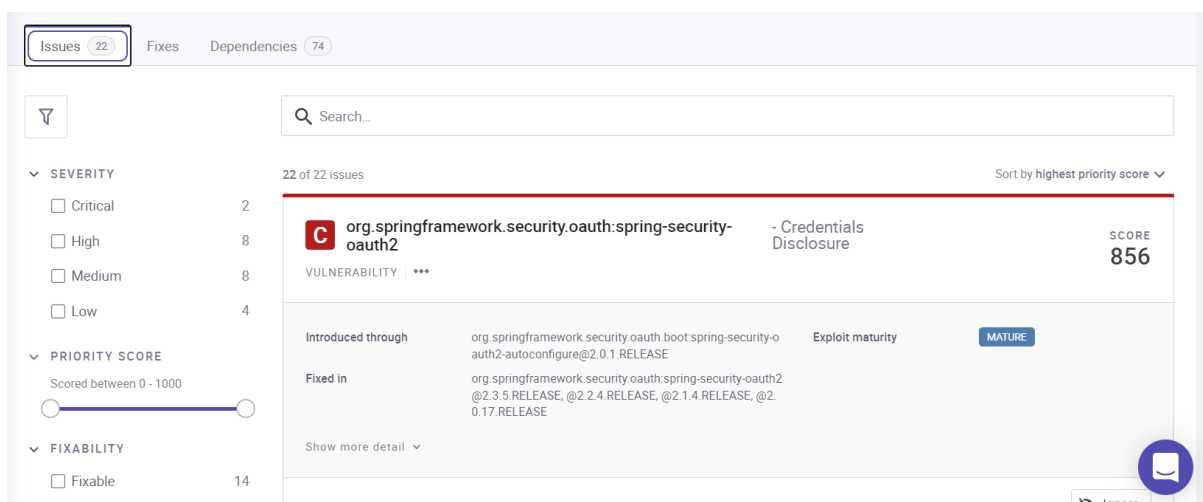
[ ] - <https://marketplace.eclipse.org/content/snyk-security-code%E2%80%8Bopen-source%E2%80%8Biac-configurations>





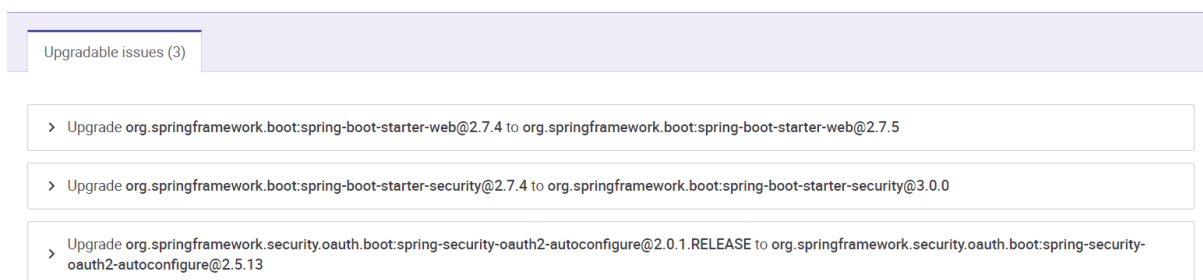
Snyk's Static analysis security testing User interface after running analysis:

Snyk identified 22 issues with dependencies:



Fix advice was to upgrade a number of the spring boot and spring security packages.

#### Fix Advice



Updating these dependencies and removing some others helped to resolve these security issues for this particular iteration of snyk being run. This was addressed in commit: 1062cc2596086d3085cc68362d88304799e9ff1d.

Snyk also performs code analysis:



Cybersecurity-Masters-NCI
main

# Cross-Site Request Forgery (CSRF)

Data flow
Fix analysis
X

SNYK CODE | CWE-352

CSRF protection is disabled by `disable`. This allows the attackers to execute requests on a user's behalf.

Data flow - 1 step in 1 file

...ain/java/com/spfwproject/quotes/SecurityConfig.java 1 step

55:15 | `disable` | SOURCE & SINK | 1

Find out how to remediate this issue through our Fix analysis »

Share Feedback

Ignore

```

47 }
48
49 // csrf disabled since we're using jwt, making that redundant.
50 // session management: create new session on login for session fixation & onl
51 // httponly, secure flags are set in application.properties and 15 minutes se
52 // on logout, invalidate session and delete cookies
53 @Bean
54 protected SecurityFilterChain securityFilterChain(HttpSecurity http) throws E
55     http.csrf().disable()
56         .sessionManagement().invalidSessionUrl(LOGIN_URI)
57         .maximumSessions(1).and().sessionFixation().newSession()
58         .and().logout().logoutUrl(LOGOUT_URI).deleteCookies(JSESSIONID).i
59
60 // specify paths allowed based on roles, anonymous represents unauthentica
61 http
62     .httpBasic()
63     .authenticationEntryPoint(new NoPopupBasicAuthenticationEntryPoint())

```

Snyk here has suggested 10 potential security issues: 1 high severity, 2 medium and 7 low.

As an example of addressing these, the tool raises a concern about having set csrf protections in spring security to disabled. This is a false positive as the tool is not able to realise that JWT is being used instead. This serves as CSRF protection preventing users from being tricked into taking an action on the website when authenticated. Attackers would not be able to craft a link which contains the JWT tokens as they are only accepted via the Authorization headers of http requests. Snyk offers the capability to set to ignore for a defined length of time or indefinitely while offering a reason as well to back this up which has been done here.

...ogramming for Web/Project/quotes/src/main/java/com/spfwproject/quotes/SecurityConfig.java

```

47 }
48
49 // csrf disabled since we're using jwt, making that redundant.
50 // session management: create new session on login for session fixation & onl
51 // http
52 // on l
53 @Bean
54 protecto
55 http
56
57
58
59
60 //
61 http
62 .ht
63 .au

```

Snyk Code uses source code features to identify the issues ⓘ

Not vulnerable | Ignore temporarily | Ignore permanently

This is a false positive as the tool is not able to realise that JWT is being used instead. This serves as CSRF protection preventing users from being tricked into taking an action on the website when

Cancel | Save

Ignore

Of the 10 there was 9 False positives (the tool suggested hardcoded passwords when a setPassword method call twice, claims of using comparison timing that could alert attackers when it wasn't a comparison operation taking place, hard coded credentials on tests with fake passwords entered on purpose for login failure tests for the remainder). The only legitimate concern was raised here:

**Unprotected Storage of Credentials**

[Data flow](#)
[Fix analysis](#)

SNYK CODE

CWE-256

An attacker might be able to detect the value of the password due to the exposure of comparison timing. When the functions `Arrays.equals()` or `String.equals()` are called, they will exit earlier if fewer bytes are matched. Use password encoder such as `BCrypt` for comparing passwords.

Data flow - 1 step in 1 file

...

m/spfwproject/quotes/services/UserServiceImpl.java

1 step

146:23

getPassword

SOURCE & SINK

1

Find out how to remediate this issue through our [Fix analysis](#) »

...

for Web/Project/quotes/src/main/java/com/spfwproject/quotes/services/UserServiceImpl.java

```

138         boolean isAccountLockedUpdated() {
139             // TODO: opt for hashmap of updated vars?
140             final String password = updaterDetails.getPassword();
141             final String city = updaterDetails.getCity();
142             final String country = updaterDetails.getCountry();
143             final boolean isAccountLocked = updaterDetails.isAccountLocked();
144
145             if (password != null && password.isBlank()) {
146                 if (!currentEntity.getPassword().equals(password)) {
147                     currentEntity.setPassword(password);
148                 }
149             }
150
151             if (city != null && city.isBlank()) {
152                 if (!currentEntity.getCity().equals(city)) {
153                     currentEntity.setCity(city);
154                 }
155             }

```

Share Feedback

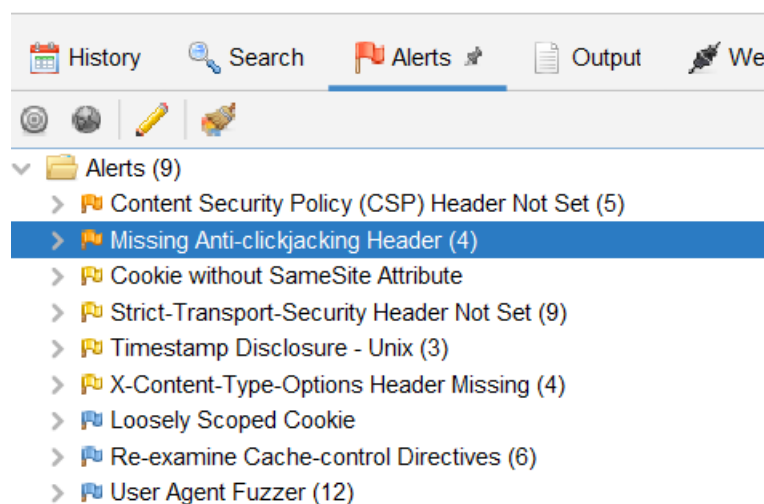
Ignore

A passive timing based attack whereby attackers may be able to discern passwords based on `.equals` exiting earlier. Instead the tool suggests using the `bcrypt` encoder `equals` method which is what this will be changed to.

## Dynamic Application Security Testing

The dynamic application security testing was conducted with OWASP ZAP tool. [8]

Here's a screenshot of it run on `https://localhost:8080/auth/login`, the login API on the backend.



### Missing Anti-clickjacking Header

The response does not include either `Content-Security-Policy` with `'frame-ancestors'` directive or `X-Frame-Options` to protect against `'ClickJacking'` attacks.

<b>Cookie without SameSite Attribute</b>	
URL:	https://localhost:8080/auth/login
Risk:	Low
Confidence:	Medium
Parameter:	JSESSIONID
Attack:	
Evidence:	Set-Cookie: JSESSIONID
CWE ID:	1275
WASC ID:	13
Source:	Passive (10054 - Cookie without SameSite Attribute)
Input Vector:	
Description:	A cookie has been set without the SameSite attribute, which means that the cookie can be sent as a result of a 'cross-site' request. The SameSite attribute is an effective counter measure to cross-site request forgery, cross-site script inclusion, and timing attacks.

A cookie has been set without the SameSite attribute, which means that the cookie can be sent as a result of a 'cross-site' request. The SameSite attribute is an effective counter measure to cross-site request forgery, cross-site script inclusion, and timing attacks.

These have been added to the list of features to consider adding next.

The active scan of my login API showed that I had not properly configured my CORS setup. While the bean with all the setup for CORS was in place:

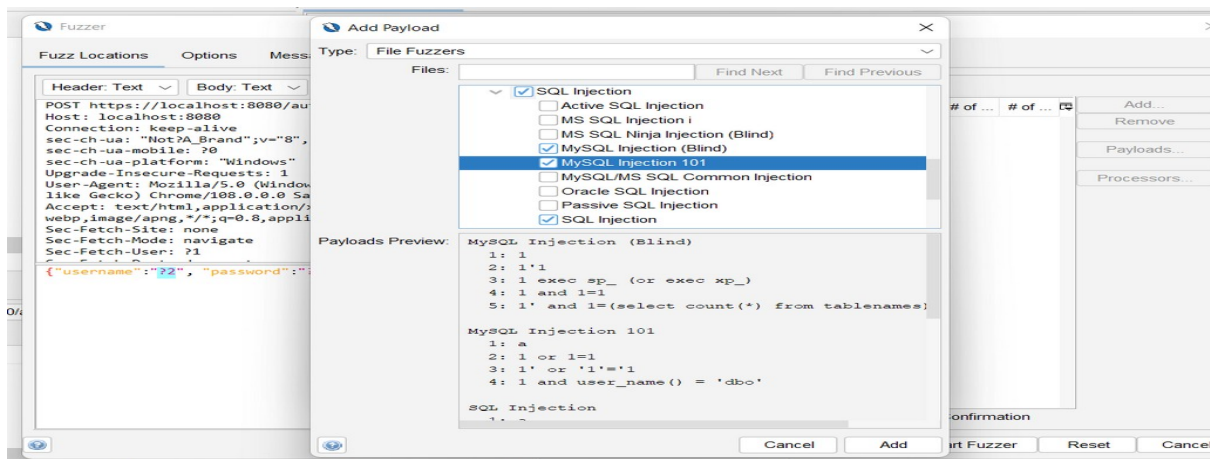
```
// Cross origin allowlist limited to the UI of the application and specific headers and http verbs relevant
// to the application
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    configuration.setAllowedOrigins(Collections.singletonList("http://localhost:3000"));
    configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"));
    configuration.setExposedHeaders(Arrays.asList("Authorization", "content-type"));
    configuration.setAllowedHeaders(Arrays.asList("Authorization", "content-type"));
    configuration.setAllowCredentials(true);

    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/*", configuration);
    return source;
}
```

I hadn't added `.and().cors()` to the `securityFilterChain`

```
protected SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    // specify paths allowed based on roles, anonymous represents unauthenticated users
    http
        .httpBasic()
        .authenticationEntryPoint(new NoPopupBasicAuthenticationEntryPoint())
        .and().cors().and()
        .authorizeRequests()
        .mvcMatchers("/admin/**").hasRole("ADMIN")
        .mvcMatchers("/auth/logout").hasRole("USER")
        .mvcMatchers("/quotes/**").hasRole("USER")
        .mvcMatchers("/users/**").hasAnyRole("USER")
        .mvcMatchers("/auth/signUp").hasAnyRole("ANONYMOUS")
        .mvcMatchers("/auth/login").hasAnyRole("ANONYMOUS")
        .anyRequest().denyAll();
}
```

Below we have SQL injection payloads bombarding the application's **login API**:



This amounted to **1156 payloads** applied across the username and password parameters:

Task ID	Message Type	Code	Reason	RTT	Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
1,145	Fuzzed	400		28 ms	543 bytes	31 bytes			'exec master..xp_cmdshell 'ping 172.10.1.255'-, @
1,146	Fuzzed	400		557 ms	543 bytes	31 bytes			'exec master..xp_cmdshell 'ping 172.10.1.255'-, ?

As you can see this did not return a single error which can give some confidence in the setup of protections against SQL injection on the login page.

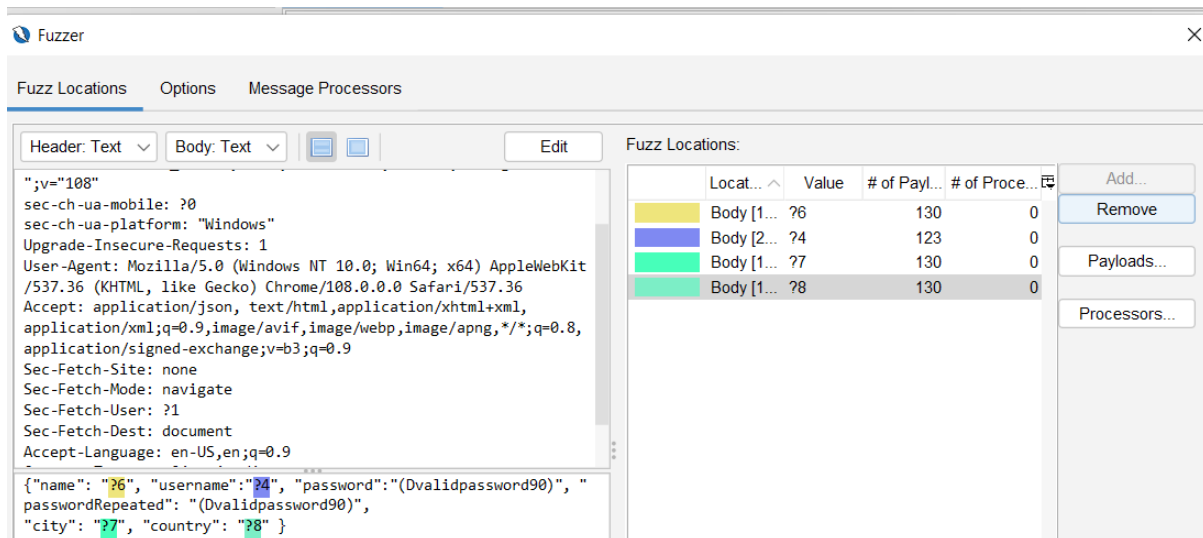
Next up was **XSS payloads**:

Task ID	Message Type	Code	Reason	RTT	Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
1,285	Fuzzed	400		34 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,286	Fuzzed	400		536 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,287	Fuzzed	400		30 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,288	Fuzzed	400		550 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,289	Fuzzed	400		43 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,290	Fuzzed	400		24 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,291	Fuzzed	400		35 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,292	Fuzzed	400		498 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></
1,293	Fuzzed	400		55 ms	576 bytes	277 bytes			<SCRIPT/XSS SRC="http://ha.ckers.org/xss.js"></

1296 payloads across the two login parameters of username and password again, helping provide some additional assurance that the protections put in place for protecting against XSS are having some effect.

## SignUp API

Here we've setup payloads for both SQL injection and XSS payloads on the parameters of the Signup form:



Again the signup form did not show up anything. There was a false positive on the passwordRepeated flagging for a possible sql injection. The payload was:

Vi(08mfu2)1ahj£\*' AND '1'='1' --

This is a false positive for a number of reasons, passwords are encrypted by bcrypt with no output of the input data on response. The password length max is 20 characters and this is 32 characters. Spaces are also not accepted in passwords. The password must match the repeated characters as well. So for all of these reasons this is false positive.

Task ID	Message Type	Code	Reason	RTT	Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
81	Fuzzed	400		79 ms	576 bytes	277 bytes		?	?
82	Fuzzed	400		100 ms	576 bytes	277 bytes			x' and user is NULL; --
83	Fuzzed	400		1.19 s	576 bytes	277 bytes			x' and email is NULL; --
84	Fuzzed	400		1.17 s	576 bytes	277 bytes			anything' or 'x'x
85	Fuzzed	400		105 ms	576 bytes	277 bytes			x' and 1=(select count(*) from tablename); --
86	Fuzzed	400		160 ms	576 bytes	277 bytes			x' and members email is NULL; --
87	Fuzzed	400		112 ms	576 bytes	277 bytes			x' or full_name like %sbo%b%
88	Fuzzed	400		214 ms	576 bytes	277 bytes			23 or 1=1; --
89	Fuzzed	400		216 ms	576 bytes	277 bytes			! exec master..xp_cmdshell 'ping 172.10.1.255'--

## 7.1 OWASP Penetration Testing Checklist

Since this is a one man project and therefore code review is not possible, I've used the OWASP Penetration testing checklist to compound the findings from the SAST & DAST tools and component and integration testing. [7]

### Access Control

#### Authorisation

**Ensure that resources that require authorization perform adequate authorization checks before being sent to a user.**

Done, every resources can only be accessed by the entity owners. See the following test examples cases for proof:

getUser\_WithAuthenticatedButNotAuthorisedUser\_ThenThrowException

```
getQuote_WithNotAuthorisedQuote_ThenThrowException
```

More of these tests in both of those test classes.

### **Ensure that once valid user has logged in it is not possible to change the session ID's parameter to reflect another user account**

The httpOnly flag is set so that cookies can't be changed on client side.

### **Check to see if its possible to access pages or functions which require logon but can be bypassed:**

Done, UI pages are set for all that require authentication are locked down as private routes. Any requests that look go directly to the backend are rejected with deny by default as the base and then non-authenticated users (ANONYMOUS role users) are only able to gain access to the login API and signup API.

```
// specify paths allowed based on roles, anonymous represents unauthenticated users
http
  .httpBasic()
  .authenticationEntryPoint(new NoPopupBasicAuthenticationEntryPoint())
  .and()
  .authorizeRequests()
  .mvcMatchers("/admin/**").hasRole("ADMIN")
  .mvcMatchers("/auth/logout").hasRole("USER")
  .mvcMatchers("/quotes/**").hasRole("USER")
  .mvcMatchers("/users/**").hasAnyRole("USER")
  .mvcMatchers("/auth/signup").hasAnyRole("ANONYMOUS")
  .mvcMatchers("/auth/login").hasAnyRole("ANONYMOUS")
  .anyRequest().denyAll();
```

See the following test class for proof:

PrivilegeEscalationAndRoutingComponentTest

## **Authentication**

### **Authentication endpoint request should be HTTPS: Ensure that users are only asked to submit authentication credentials on pages that are served with SSL.**

Done, see code from application.properties below, TLS is setup with version 1.3 using a self-signed certificate. This certificate would be swapped out for one signed by a certificate authority when moving from this POC to production.

```
# TLS configuration
server.ssl.enabled=true
server.ssl.key-store-type=PKCS12
server.ssl.key-store=classpath:keystore.p12
server.ssl.protocol=TLS
server.ssl.enabled-protocols=TLSv1.3
```

} package.json

JS Login.js M

JS Signup.js M X

TS AdminPa

onents > JS Signup.js > Signup > render

```
handleSubmit(event) {
  event.preventDefault();
  const form = event.target;

  fetch('https://localhost:8080/auth/signup', {
    method: 'POST'
```

Sample UI request to backend being served over https (with TLS).

**AuthenticationBypass: Ensure that the authentication process can not be bypassed.**

Done, see the following test class:

```
PrivilegeEscalationAndRoutingComponentTest
```

**Ensure that usernames and passwords are sent over an encrypted channel.**

Yes, all traffic is sent over TLS as denoted before.

**Default Accounts and Password Quality: Check for default account names and passwords in use. Ensure that the password complexity makes guessing passwords difficult.**

This is done with a commonly used password list to cover the first part making dictionary attacks more difficult and password complexity requirements that make brute forcing the password very difficult as well. See the UserDetailsValidator for more details.

**Ensure that user must respond to a secret answer / secret question or other predetermined information before passwords can be reset.**

Not completed functionality due to time constraints.

**Password Lockout: Ensure that the users account is locked out for a period of time when the incorrect password is entered more that a specific number of times (usually 5).**

Done, see the following test displaying that in the test class

```
AuthenticationControllerIntegrationTest:  
testLogin_WithBadCredentialsFourTimes_ThenExpectAccountTobeLocked
```

**Password Structure: Ensure that special meta characters cannot be used within the password**

Passwords only accept A-Z, upper and lowercase, special characters such as !£\$%^&\*(). Any characters beyond that are rejected including meta characters.

See the test class:

```
// QWERTYUIOPASDFGHJKLZXCVBNMQWERTY
// use non-ASCII characters, expect error
setPasswordAndRepeatPassword("ABCDEF̂GHĬ/KLMNŌPQRṢTUV̄WXYZ", signUpForm);
validator = authenticationService.validateSignupForm(signUpForm);
assertTrue(validator.containsErrors());
assertTrue(validator.getListOfErrors().contains(passwordErrorExpected));
```

```
// use more non-ASCII characters, expect error
setPasswordAndRepeatPassword("Lorēm ipsūm dōiōē sīt amēt cōnsectētur̄ ādīpīscīng̃ #iī, "
    + "Ŧēd dō ēiūmōō tēm̄pōr̄ mciq̄dūnr̄p̄t̄ īaōōrē #r̄dōlōrē Māgnā #iīquā.", signUpForm);
validator = authenticationService.validateSignupForm(signUpForm);
assertTrue(validator.containsErrors());
assertTrue(validator.getListOfErrors().contains(passwordErrorExpected));
```

AuthenticationServiceComponentTest

## Blank Passwords: Ensure that passwords are not blank

Done in test class: **AuthenticationControllerIntegrationTest**

## Session Length: Ensure that the session token is of adequate length to provide protection from guessing during an authenticated session

Yes it is long enough with spring security, 10+ Hex characters:

Name	Value	Domain	Path	Expires	HttpOnly	Secure
JSESSIONID	8C40A67E9E1 C5469E82B2E 93C4774B03	localhost	/	Session	true	true

## Session Timeout: Ensure that the session tokens are only valid for a predetermined period after the last request by the user.

Session timeout is restricted to 15 minutes in the application.properties file and the remainder if it's configuration is in SecurityConfig:

```
server.servlet.session.timeout=15m
```

## Session Reuse (and Session fixation): Ensure that session tokens are changed when the user moves from an SSL protected resource to a non-SSL protected resource.

Sessions are changed on login and signup going from the unprotected pages to protected resources and protecting against session fixation.

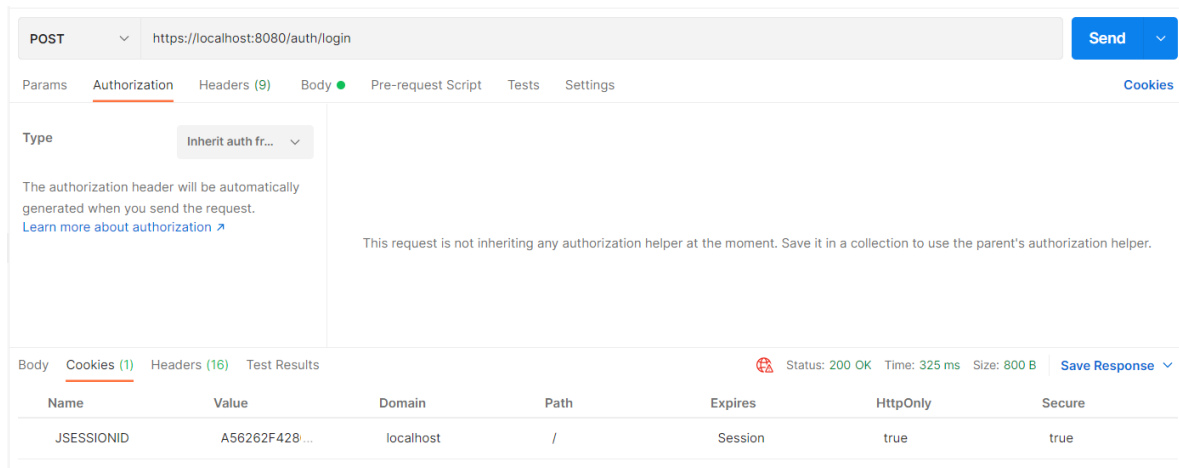
See SecurityConfig for more details, here's the image of the code performing this:

```
.and().sessionFixation().newSession()
```

The screenshot shows a REST client interface. At the top, a POST request is sent to `https://localhost:8080/auth/login`. Below the request, the 'Headers' tab is selected, showing 9 headers. The response is displayed at the bottom, showing a status of 401 Unauthorized. The 'Cookies' tab is selected, showing a new session cookie: `JSESSIONID=90D01C670F...` with domain `localhost` and path `/`.



Above we have session id change from prior to login (with a failed request) to after successfully logged in. This combats against session fixation attacks.



**Session Deletion: Ensure that the session token is invalidated when the user logs out.**

Done, this is in place in the SecurityConfig for that reason:

```
.and().logout().logoutUrl(LOGOUT_URI).deleteCookies(JSESSIONID).invalidateHttpSession(true);
```

Doubled up in AuthenticationController to be certain.

```
@PostMapping("logout")
```

```
public ResponseEntity<Object> performLogout(HttpServletRequest request) {  
    HttpSession session= request.getSession(false);  
    SecurityContextHolder.clearContext();  
    session= request.getSession(false);  
    if(session != null) {  
        session.invalidate();  
    }  
    if (request.getCookies() != null) {  
        for(Cookie cookie : request.getCookies()) {  
            cookie.setMaxAge(0);  
        }  
    }  
    String loggedOutMessage = "User is successfully logged out.";  
    logger.info(loggedOutMessage);  
}
```

```

        ResponseEntity<Object> reponse =
ResponseEntity.ok().body(loggedOutMessage);

        return reponse;
    }

```

The above code firstly checks to see if the session is present in the HttpRequest to logout, then it invalidates the session. It then looks to see if any cookies are present and sets them all to have a maxAge of 0, meaning they will expire immediately, thus completing the logout.

**Session Token Format: Ensure that the session token is non-persistent and is never written to the browsers history or cache.**

Caching is not used by the application and thus is completely set to disabled:

Cache-Control ⓘ	no-cache, no-store, max-age=0, must-revalidate
Pragma ⓘ	no-cache

### Configuration Management

**Ensure that the web server does not support the ability to manipulate resources from the Internet (e.g. PUT and DELETE)**

Cross-origin Resource sharing restricts access to resources is. Only authenticated users that are of the correct role and the owner of an entity can update or delete resources. Users must have a valid JWT token that's not expired and a session not expired as well.

**Known Vulnerabilities / Security Patches: Ensure that known vulnerabilities which vendors have patched are not present.**

All known vulnerabilities in dependencies have been highlighted by SAST tool Snyk and remediated and would continue to do so with each push and otherwise on a weekly basis with Snyk performing a weekly automated scan.

### Error Handling

**Application Error Messages: Ensure that the application does not present application error messages to an attacker that could be used in an attack.**

Errors customized to limit the amount of information revealed and not reveal anything useful to attackers.

**Ensure that the application does not present user error messages to an attacker that could be used in an attack.**

Errors for login are generic and do not reveal anything about whether the password or username was incorrect on failure.

E.g. the message for login failures is:

`Credentials have been entered incorrectly.`

This does not show whether it was the username or password that was incorrect, preventing username enumeration from this.

Same applies on the signup form. If the username already exists, no indication is given just a generic username failure message to protect from username enumeration:

`Invalid username.`

### **DataProtection. Transport**

#### **SSL Version: Ensure that SSL versions supported do not have cryptographic weaknesses**

The application only supports TLS v1.3 which is the latest version and doesn't have any known cryptographic weaknesses.

#### **Digital Certificate Validity: Ensure the application uses valid digital certificates.**

Since this is a POC, a self-signed certificate is used, this would be replaced for a production setting.

### **Input Validation**

#### **SQL Injection: Ensure the application will not process SQL commands from the user.**

#### **Cross Site Scripting: Ensure that the application will not store or reflect malicious script code.**

The UI of the application uses JSX with React which escapes any data rendered, thus protecting against XSS attacks. This is the only way data is output on the page in the application. Secure and httpOnly cookie flags are set to protect and prevent manipulation of the cookie. Cookies will only be sent over https due to the secure flag, ensuring that they are encrypted with TLS in transit preventing them from being snooped with any man in the middle attacks. HttpOnly prevents javascript from being able to act on the cookies which is one of the primary ways both reflected XSS and stored XSS can act on cookies and session data. UTF-8 is set to limit the character set and prevent any additional characters being entered. All input is validated on entry to each API. Cross origin resource sharing is also restricted to the user interface preventing the running of other scripts.

## **8.0 Conclusions**

Some of the main strengths of this project are a plethora of defences in place taking advantage of industry standard practices for creating an architecture which reduces the likelihood of successful attacks from attackers. Each of the core aspects of security in relation to applications have been carefully considered, researched and implemented with using a framework dedicated to application security in Spring Security and React on the UI. The attack surface is minimized through the number of conditions and filters that must be met to even touch the surface of any business logic or the database of data. Every request must make it's way through a sequence of steps that scrutinize the legitimacy of the request, the data contained within it and also the authorisation and authentication of that user. A whole host of attack vectors have been taken into account from the OWASP Top 10 and beyond.

A variety of different layers of protection come together to provide a defence in depth approach covering authentication, session management, authorisation, cross origin request sharing, jwt filters protecting against Cross site request forgery, protections of data in transit and at rest, protections against brute force attacks, SQL injection (validation on server side, prepared statements, react encoding), Cross site scripting (in it's various forms), a vast amount of logging to monitor and potentially be used to alert on security incidents increase the likelihood of non-repudiation, error handling that does not reveal sensitive information. There was also a number of tools, SAST, DAST, component testing, integration testing and Penetration Testing review of the code and addressing any findings where needed to keep improving the system.

Disadvantages were that there was a decent learning curve required to get a lot of the required aspects of spring security in place. This is not a framework that you can wing it with, it requires a lot of research and trial and error to get things how they should be and executing properly. Also, features like the password reset form and jenkins pipelines did not get fully implemented. It may have been an overreach to try and get those included as well.

#### **Future improvements:**

- jenkins pipeline which runs automatically when a given user pushes a commit to the repo. This would run a suite of security tools, including SAST tools like Snyk, DAST tools ZAP, secrets finder and OWASP Dependency Check to provide immediate feedback on possible security issues in the latest commit. It could also run all of the integration and component tests as well.
- rolling key changes: pipeline created to generate a new key periodically. These would also be pulled in with the application build and stored separately in a protected keystore.
- container orchestration & load balancing with kubernetes , DDOS protection (from providers like Akamai).
- application.properties and key used for certificate and jwt would be pulled in from a secrets storage. This was not done due to time constraints.
- password reset form

## **9.0 References**

Please include references throughout your document where appropriate. See [here](#) for a guide on referencing from the NCI library. Use Harvard referencing style.

[1] W. Johnson, Auth0 blog, May 04 2022, [Online] Available:

<https://auth0.com/blog/rs256-vs-hs256-whats-the-difference/>

[2] Snyk, [Online] Available: <https://snyk.io/learn/application-security/static-application-security-testing/>

[3] L. Spilicu, Spring Security in Action, 2020, [Online] Available:

<https://www.amazon.co.uk/Spring-Security-Action-Laurentiu-Spilca/dp/1617297739>

- [4] Cybersecurity & Infrastructure Security Agency, Software Bill of Materials, [Online] Available: <https://www.cisa.gov/sbom>
- [5] Apache Maven, [Online] Available: <https://maven.apache.org/>
- [6] berzerk0, Real Passwords, Dec 4 2019, [Online] Available: <https://github.com/berzerk0/Probable-Wordlists/tree/master/Real-Passwords>
- [7] OWASP, OWASP Web Application Penetration Testing Checklist, [Online] Available: [https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP\\_Web\\_Application\\_Penetration\\_Checklist\\_v1\\_1.pdf](https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP_Web_Application_Penetration_Checklist_v1_1.pdf)
- [8] ZAP, [Online] Available: <https://owasp.org/www-project-zap/>
- [9] K. Poojary, December 14 2011, [Online] Available: <https://www.computerweekly.com/tip/CSRF-attack-How-hackers-use-trusted-users-for-theirexploits>
- [10] NIST, Secure Hash Standard, August 1 2002, [Online] Available: <https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf>
- [11] HHS Cybersecurity Room, SSL/TLS Vulnerabilities, February 25th 2021, [Online] Available: <https://www.hhs.gov/sites/default/files/securing-ssl-tls-in-healthcare-tlpwhite.pdf>
- [12] Spring Sleuth, [Online] Available: <https://spring.io/projects/spring-cloud-sleuth>
- [13] - Digital.AI, 16th Annual State of Agile Report, 2022, [Online] Available: <https://info.digital.ai/rs/981-LQX-968/images/AR-SA-2022-16th-Annual-State-Of-Agile-Report.pdf>
- [14] D.Gruber, Modern Application Development Security, August 2020, [Online] Available: <https://www.veracode.com/sites/default/files/pdf/resources/surveyreports/esg-modern-application-development-security-veracode-survey-report.pdf>
- [15] - T.Nguyen, Integrating Security into Agile Development Methods, 2015, [Online] Available: <https://www.umsl.edu/~sauterv/analysis/F2015/Integrating%20Security%20into%20Agile%20methodologies.html.htm>
- [16] - T. Gilb, Principles of Software Engineering Management(Wokingham, England: Addison-Wesley), 1988
- [17] - J. Capers, Assessment and Control of Software Risks. Englewood Cliffs, N.J.: Yourdon Press, 1994

[22] - C. Gustke, New York Times, No Business is too Small to be Hacked, January 13 2016, [Online] Available: <https://www.nytimes.com/2016/01/14/business/smallbusiness/no-business-too-small-to-be-hacked.html>

[23] Online Trust Alliance, January 25 2018, [Online] Available: <https://www.internetsociety.org/wp-content/uploads/2019/04/2018-cyber-incident-report.pdf>

[24] CISA, [Online] Available: <https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/failing-securely#:~:text=This%20principle%20requires%20that%20the,granted%2C%20it%20should%20be%20denied.>