CSC8208 GROUP PROJECT

A Secure Ticketing System for In-person Events

Team Number: 4

Contents

Table of Figures 2
Introduction 3
Project Scope 3
Design 4
Web Stack 4
Flask 4
React 4
MySQL 5
Nginx 5
QR Code & Cryptographic Primitives 5
QR Codes 5
Ed25519 Signature Scheme (Ticket Integrity) 5
Implementation 6
Authentication and Access Control 6
Ticket Issuance 7
QR Code Ticket Generation 8
Server-Side Process 8
QR Data Produced 9
Client-Side Process 9
Ticket Verification 10
Connection and Database Security 10
Testing and Evaluation 11
Performance Evaluation 11
System Testing 11
Conclusion 12
Limitations 12
Future Work 12
References 13
Appendix 14
Screenshots of React Application 14
Unit Tests 18
Add Ticket 18
Request Ticket QR Data 18
Ticket Validation 18

Events 19
Users 19
Performance Testing 20

Table of Figures

Figure 1: System diagram showing the overview of the system 4
Figure 2: Sequence diagram 7
Figure 3: Activity diagram showing the process of requesting QR data from the server 8
Figure 4: Activity diagram showing the verification of a ticket 10
Figure 5: Registration page 14
Figure 6: Login Page 14
Figure 7: Home Page with event list 14
Figure 8: Search Page with event option 15
Figure 9: Ticket details of specific event 15
Figure 10: Ability to buy tickets 15
Figure 11: User account page with tickets purchased 16
Figure 12: QR code of a valid ticket and animation of ‘firework’ when QR is clicked 16
Figure 13: Screenshots of QR code scanning 17
Figure 14: Figma design for the application 17
Figure 15: Screenshot showing the add ticket unit tests 18
Figure 16: Screenshot showing the request for QR data unit tests 18
Figure 17: Screenshot showing the validated ticket unit tests 18
Figure 18: Screenshot showing the event-related endpoint unit tests 19
Figure 19: Screenshot showing the user-related endpoint unit tests 19
Figure 20: Response time of '/ticket/add' endpoint 20
Figure 21: Response time of '/ticket/request_qr_data' endpoint 20
Figure 22: Response time of ‘/ticket/validate’ endpoint 20

Introduction

Digital ticketing has revolutionised event management by offering a convenient and flexible solution. However, the security aspect of digital ticketing requires careful consideration to provide a seamless and safe experience for users. This was the main motivation for us to choose digital ticketing security as our project. Our goal was to contribute to this field and gain a better understanding of the security measures required to ensure reliable digital ticketing systems. We decided to create a web app in React, with a Flask REST API for the backend, to enable users to 'purchase' tickets (giving them a secure QR code) for in-person events, with a focus on security.

To achieve our aim, we began by researching the topic thoroughly to understand the current solutions in the literature and analysing their advantages and disadvantages. Then, we regularly met, brainstormed, and discussed the different possible designs and technologies until we reached the current, robust design that we think has many strengths which will be further explained throughout this report. To implement our design, we formed two sub-teams, the frontend, and the backend. Whenever the sub-teams finish some features which are ready to be integrated, we sat together to combine them into one robust component. Once our implementation was realized, it was reviewed and tested to make sure it meets our expectations. We utilised git as a tool for each sub-team to develop the code concurrently.

In this report, we first explain our design and chosen technologies and cryptographic primitives. Following that, we discuss in detail the implementation including the authentication and access control system, event handling and secure issuance and validation of tickets. Then, we present the results of testing and evaluating our solution. Finally, we discuss the limitations and possible extensions.

Project Scope

The assumptions that have been made during the design:

- Tickets can be scanned for an event up to 12 hours after the event has started.
- No payment system will be implemented – This would require a secure payment system to be implemented and would go beyond the scope of the project.
- Users can buy a maximum of 4 tickets per purchase on the server.

Design

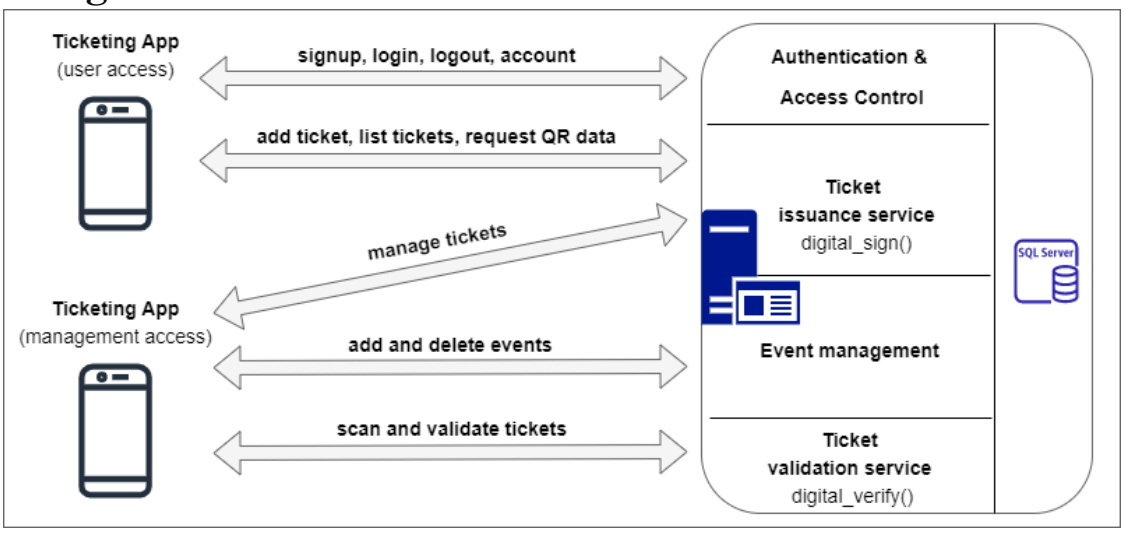

Figure 1: System diagram showing the overview of the system

Figure 1 illustrates the overall secure ticketing system which includes a frontend application and several backend services. The frontend application can be accessed by normal users to register, login, view information about events, buy tickets and view them for scanning and validation. Additionally, users with management roles can utilize the application to add events, scan, and request ticket validation. Low-fidelity Wireframes, as shown in *Figure 14* were implemented at the beginning of the project to get a better expectation of the application and to make sure user mental models were included. On the backend side, an authentication and access control component is used to register users in a SQL database and perform login and logout using JWT cookies. Another backend component is a ticket issuing service which allows the users to add tickets which are digitally signed. Also, there is an event management component which allows the users with management roles to add and delete events. Furthermore, the backend side contains a ticket validation service which receives ticket validation requests and does a few verifications on the tickets including a verification of the digital signature. The connection between the client and server sides is secured using HTTPS.

Web Stack

Flask

During the planning phase of this project, we decided to develop a cross-platform web application which can be utilized using different devices. We discovered, after evaluating our team's collective experience with various programming languages, that Python stood out as the language in which we possessed the most experience. Consequently, we concluded that Python was the most suitable choice and decided to employ the Flask web framework, considering several factors such as the previous experience our team members have with Flask, its scalability and flexible development. Flask also allows for communication between the client and server through a REST API. This allows frontend UI and backend data manipulation to be independent which allows for more flexibility and easier concurrent development.

React

We decided to use the React framework for our project. This meant we could develop a web app for our ticketing system, meaning we can deploy to desktop, Android, and iOS at the same time, without the need to deploy native applications. Using a service worker, the app can be installed from the website easily on

mobile devices, giving a native app experience. As React builds on JavaScript, it was also easy to integrate with Flask and get a sufficient app up and running in the limited time frame for the project.

MySQL

For storing data, we decided to use MySQL as it is one of the most popular open-source relational database management systems that is widely used for web applications and provides the advantages of easier maintenance and scalability to handle large amounts of data with high performance. As MySQL is supported by many programming languages, including python, which is used for the Flask web framework, this means that integrating MySQL with our Flask application proved to be more straightforward and efficient. Additionally, the vast community of developers that use MySQL means that there are plenty of resources available for troubleshooting, optimizing performance and implementing the best-recommended practices.

Nginx

Nginx adds an extra layer between the client and server, which serves as both an HTTPS reverse proxy and a load balancer. Nginx implements SSL, which can use self-signed certificates for this project, which means that the Flask and React services do not have to directly handle this as it acts as a reverse proxy. Additionally, the Flask server is not exposed directly to the outside world which means an extra layer of security. In addition, employing a load balancer ensures scalability and availability if the Flask server is replicated on a large scale.

QR Code & Cryptographic Primitives

QR Codes

Among the other technologies such as Bluetooth and NFC, we have chosen to employ QR codes in our ticketing application as they are easily and efficiently scanned by smartphones, and do not require uncommon hardware (e.g., NFC terminal). They carry huge amounts of information and are widely used in the industry, with users already being familiar with them. Furthermore, we were able to specify viable and effective techniques to secure them based on our literature review.

AES (Ticket Confidentiality)

The Advanced Encryption Standard (AES) is a symmetric key encryption cypher and is currently one of the best encryption protocols available. From research, we also found that AES-256-CBC would be compatible with the QR codes through which we planned to encrypt ticket information over HTTPS to maintain the confidentiality of tickets, while they are transmitted from the server to the client, to prevent an attacker from being able to observe or steal the tickets from packet sniffing.

Ed25519 Signature Scheme (Ticket Integrity)

For verifying and authenticating the integrity of tickets, the Ed25519 digital signature scheme has been used. Ed25519 produces a small signature of 64 bits which is relatively small compared to other signature schemes such as RSA. Given the limited space available on a QR code, utilizing a signature scheme that generates a compact signature would ensure that all the data fits within the code. This not only makes the ticket easier to scan but also requires less detailed information, as a smaller set of data is sufficient. A smaller set of data is sufficient.

Implementation

Authentication and Access Control

According to the literature, a ticketing system could be stateless or stateful. We have chosen to implement a stateful system to have a complete and scalable system which have a management and user side with relevant access control. In addition, user information is used in the ticket-issuing process. Also, by allowing users to have accounts, they can store and retrieve their tickets securely using our system. For all these reasons, we have preferred a stateful system over a system which is based on anonymity.

To signup, a new user fills out a form of details in *Figure 5,* including *first name, surname, date of birth, postcode, email address,* and *password* as they are used for login. When a user signs up to the application, they are given the role 'user' which allows them to buy tickets for events and see their previously bought tickets. There is also the role of 'management' which can only be given from the server as this project did not implement a full admin system.

Before a password is stored in the database, a random salt is added to it and the resulting string is hashed. Both the hash and the random salt are then stored in the database. This significantly enhances the difficulty of offline birthday attacks in the event of a database compromise. For the hashing and random salts, we used SHA-256 which is considered a secure hashing algorithm that produces a large hash. The registration system also includes validation checks; for instance, it checks email format, postcode length and that a phone number is unique and numeric.

For the login system, we decided to use JWTs (JSON Web Tokens) as they are the most used (Henry, 2019). JWTs are also lightweight and does not require storage in the server-side removing open session tracking overhead. A JWT consists of a header, a payload, and an integrity signature and it is regularly refreshed. They contain the user's ID, and they can be used to protect the endpoints for access control (such as buying a ticket without being logged in or verifying a ticket without the role of management). Protect the endpoints for access control (such as buying a ticket without being logged in or verifying a ticket without the role of management). Protect the endpoints for access control (such as buying a ticket without being logged in or verifying a ticket without the role of management).

To login, a user enters their email address and password *Figure 6,* and the credentials are sent to the Flask server which checks their validity. If valid, the server generates a JWT access token which contains the user's ID and signs it with a secret key. This JWT is included in the response to a successful login and is stored in a cookie. All the subsequent requests to the server must include this cookie to maintain the session with the server. Before responding to any request to a protected route, the server verifies the authenticity and integrity of the JWT. The server also generates additional tokens which are anti-CSRF tokens to mitigate CSRF attacks in which a legitimate logged-in user could be tricked (e.g., by clicking on a malicious link) to submit a request with hidden forms or perform unwanted actions.

One of the main aspects of this project is preventing a ticket from being shared between multiple users. A possible way of sharing a ticket could be to login with the same account on multiple devices and generate the QR codes to present to the security guards. To prevent this, a system has been implemented to allow the user to only have one valid session with the server as the most recent session will only get stored in the database. If the user logs into multiple devices, any previous sessions will be invalidated, and the user would have to login again to perform any actions on the original device.

Ticket Issuance

Upon logging in as a user, the user is initially presented with a carousel, listing all upcoming events on the home page, as shown in *Figure 7*. All of these are retrieved from the Flask server using an API route. Alternatively, they can type in a query for an event name without having to look through all the events on the homepage, as seen in *Figure 8*. The user can then view the specifics and details of the event by clicking the view event button under an event card, to which they will be presented with a page similar to the one seen in *Figure 9*. Through this ticket detail page, the user has the option to buy multiple tickets (up to 4) as well as the type of ticket that they want for that event, as seen in *Figure 10*.

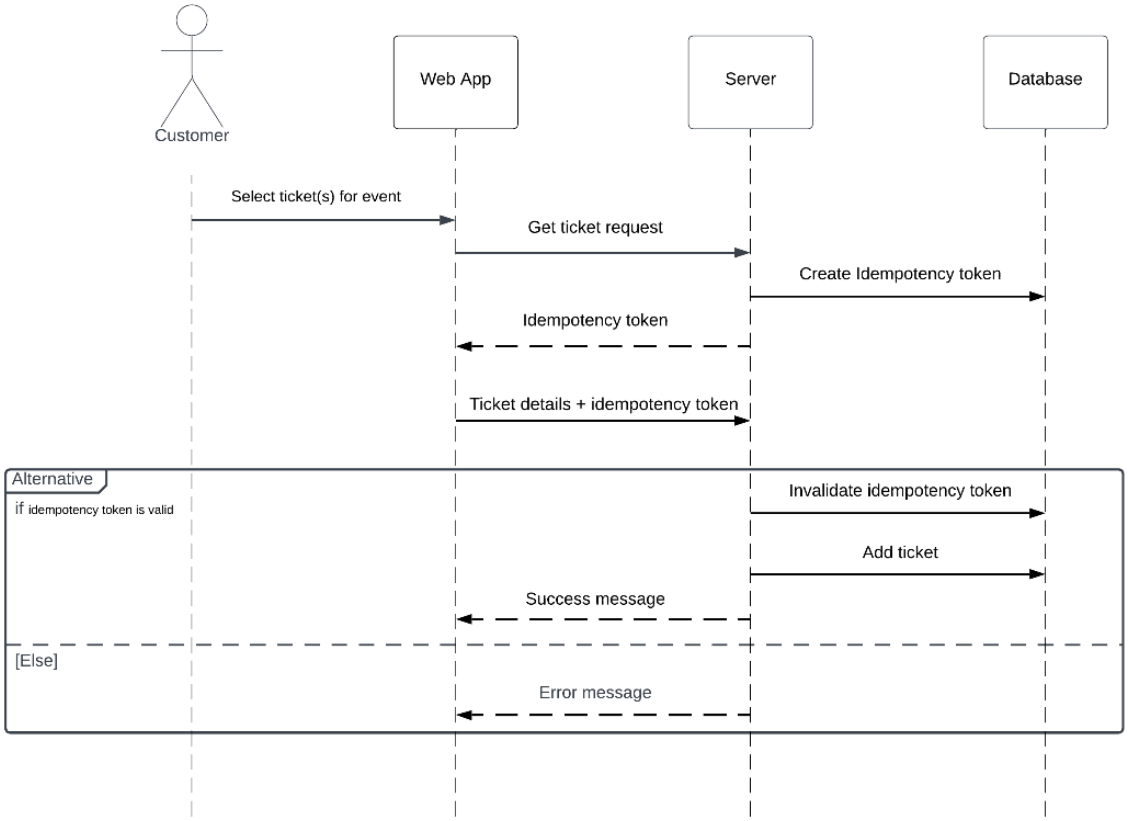

Figure 2: Sequence diagram

This information is encrypted and sent to the server to be authenticated, then added to the database. An attacker can intercept this data while it is in travel and, without decrypting, can replicate this request many times, this is called a replay attack. A user can also perform this attack accidentally by clicking a button many times, or a network error can cause TCP to resend this request many times. A fix to this is to use an idempotency token. Before a user sends the ticket-issuing request, an idempotency token is generated and affixed to the request. When an idempotency token is generated, it is added to a database and marked as valid. When a request is received the token is looked up in the database. If the token is valid then the request is processed, and the token is marked as invalid. If the token is invalid/doesn't exist in the database, then the ticket request is denied. This means if an attacker replayed a request, it would be invalid because the idempotency token is invalid.

Once a ticket is purchased, the user can view it, along with previously purchased tickets, on their account page, as seen in *Figure 11*. Consistent with the home and search pages, the purchased tickets are displayed

in a carousel format to maintain familiarity for the user. Each ticket card contains relevant details about the event, allowing the user to stay informed without having to search for and navigate back to the specific event page. Consequently, the QR code for each ticket can be accessed directly from this page, as seen in *Figure 12.*

For events, we limit the ability to add or remove events to the development side only, as we deemed it unnecessary to make these functions accessible to users. Such functions could also pose security risks, as granting access to these endpoints could potentially result in accidental or intentional modifications to critical stored data, causing significant damage. To facilitate testing, we generated example data by running a file that creates and saves test data, including example events and admin users.

QR Code Ticket Generation

Server-Side Process

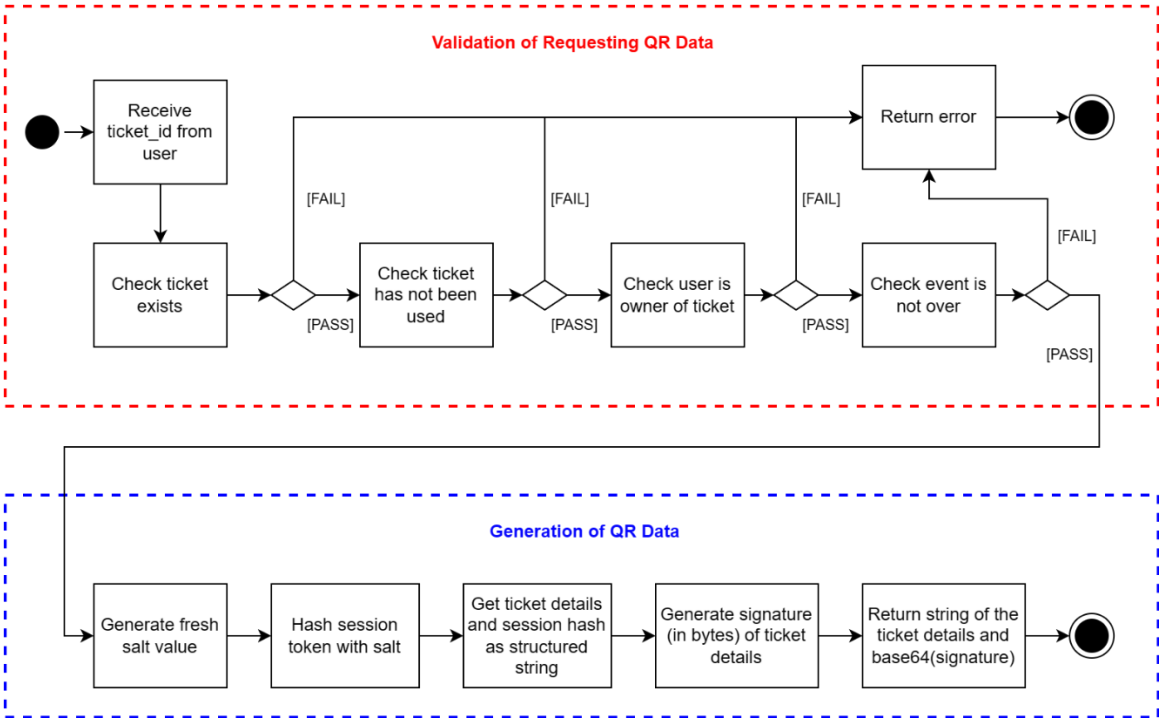

Figure 3: Activity diagram showing the process of requesting QR data from the server

The process of requesting the QR data for a ticket from the server is split into two parts:

1. **Validation of request for QR data:** The validation process involves several checks: whether the ticket exists in the database, whether it has not been used before, whether the user is authorized to request the ticket (by verifying ownership), and whether the event associated with the ticket is still ongoing.
2. **Generation of QR Data:** Firstly, a fresh salt value is generated for the ticket which will be used to invalidate any previously generated codes for the same ticket. Next, the user's session is hashed with the salt. A small hashing scheme is used, generating 8 bytes. This may have a higher collision rate than a larger hashing algorithm (such as SHA-256), however, the space is limited in a QR code and adding more data can make it harder to scan. The ticket details are then structured into a string, which is then signed using the server's Ed25519 private key. Finally, the structured ticket details and base64 signature string are returned to the user over an encrypted HTTPS connection.

QR Data Produced

The process, outlined on the server, will produce the following code to be included in the QR code on the frontend:

ticket_id, event_id, ticket_type, hash(session + salt), signature

The ticket ID, event ID and ticket type will be integers to keep the size of the code smaller. The hash value will be 8 bytes long and be refreshed each time the user requests the ticket from the server. The signature is encoded in base64 as when it is signed by the server it is in bytes and needs to be readable as a string for the QR data.

Client-Side Process

The QR code itself, as depicted in *Figure 12*, is automatically generated by the react application on the client side to prevent any performance or security issues in generating and storing the QR code image on the server. To enhance the level of security of the QR code, we employ two subtle animation techniques.

First, to combat the risk of ticket screenshots, an animated logo is overlaid on the ticket QR codes, like the approach that many existing identification systems make use of, such as online train rail cards. This image changes colour indefinitely so that when an employee goes to scan a customer's tickets, they can quickly identify whether the ticket is valid from the added animation. This image is not included in the QR code data and therefore does not affect scanning performance. However, one drawback of this approach is that if an employee were to scan a ticket from a screenshot, it would still be validated, potentially posing a security risk.

Another potential security threat we identified was a situation where a malicious user could record the QR code and its overlaying animated logo to replicate it. To address this risk, we drew inspiration from existing banking apps, such as the Swish payment app implemented an additional animation feature in the QR code that displays fireworks whenever a user taps the screen. This technique counters screen recording attempts of the ticket, as the animation is not synchronized with the random tapping of a security guard, for example. However, there is still a risk if a member of staff neglects to tap the screen to confirm the legitimacy of the ticket during dynamic animation features that provide additional layers of security to our system. However, there is still a risk if a member of staff neglects to tap the screen to confirm the legitimacy of the ticket during scanning and validation. In such cases, the ticket would still validate regardless, and this could pose a security risk. Nonetheless, these dynamic animation features provide additional layers of security to our system.

Ticket Verification

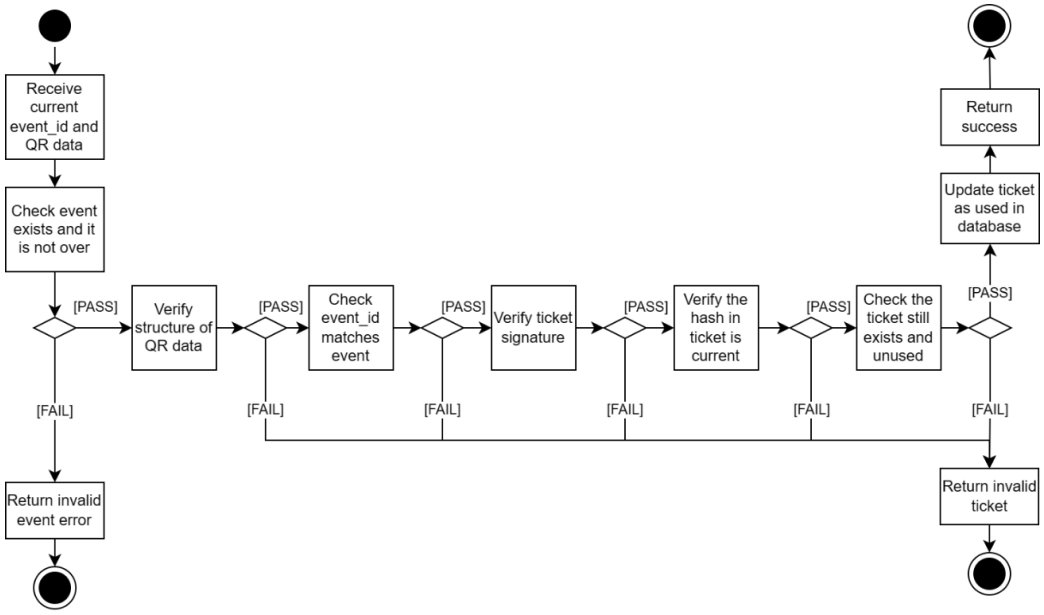

Figure 4: Activity diagram showing the verification of a ticket

The verification of a ticket is performed server-side. The management application will select an event to scan tickets for, and then scan the QR code tickets. The server will receive this information and first check if the event is valid and has not ended. The server will then begin to verify the data in the ticket by validating its structure against the general format generated by ticket issuance. The event ID is then checked to see if it matches the current event. This is before the signature is verified because if the event ID does not match the current event, without the signature, then the ticket is clearly for the wrong event. The ticket signature is verified using the server's public key to ensure the integrity of the ticket and prevent any unauthorised modifications. The hash contained within the ticket will be verified against the current hash for the ticket owner's session and ticket, which would be produced by the server, to see if it matches to ensure that it is fresh and could not have been generated in an invalid session or on multiple devices. The server then queries the ticket in the database to check that it has not been deleted or previously scanned. Finally, if the ticket is valid, the server will update the ticket in the database to mark it as used and then return the successful result to the management application.

The verification of the structure of the data contained in the QR code, checking some of the ticket details, and verifying the ticket signature (using the server's public key) could have been possible to do within the client-side React application. The other parts of the verification were required to be performed on the server as they required querying the database for the event, user, and ticket information. Due to the time constraints of this project, optimising the ticket verification by using React to partially verify the ticket before being sent to the server was not possible, however, it could be added in future.

Connection and Database Security

For the connection between the client side and the server side, we used HTTPS. This was accomplished using a self-signed TLS certificate (cert.pem & key.pem) generated using OpenSSL. This meant that the shared key exchange is done securely and that the integrity and confidentiality of the connection is guaranteed. This was implemented using Ngnix as a reverse proxy server to avoid exposing the Flask server directly to the clients which we think is a more secure approach. Regarding the database, we decided to use

the object-relational mapping (ORM) library 'SQLAlchemy' to help avoid writing raw SQL commands which could leave the application prone to SQL injection attacks.

Testing and Evaluation

Performance Evaluation

To evaluate the performance of our system, we considered the most critical endpoints which are ticket addition, QR request and ticket validation, which all must be reasonably responsive for the system to be useful. The metric that we considered is the *response time* which we specified to be the time taken from the moment a request leaves the client device to when the corresponding response is received. As both sides of our system run on localhost, we almost neglected the network delay. As depicted in the graphs below, the measurement scripts of endpoint response times were run 1000 times. *Figure 20* shows average the response time of 'ticket/add' endpoint which encompasses two requests (GET & POST) to add a digitally signed ticket, being approximately 6.17ms. *Figure 21* shows the response time of '/ticket/request_qr_data' which is the endpoint that generates QR data and sends it to the frontend to be used for the actual QR code generation, being 2.57ms on average. *Figure 22* shows the average response time of ticket validation including digital signature verification which is performed by the '/ticket/validate' endpoint, being 3.18ms, nearly. This time does not include the scanning time as we decided to neglect it because it is based on many factors that are out of our control including human factors and the camera or scanner used.

As the server performs all the verification of a ticket, using its public key to check the signature of the ticket. If the public key had been securely distributed to the management web application, it would have been possible for the management to partially verify a ticket without the server. In this case, the management application could have verified the signature of the tickets and checked that the ticket details matched the event before sending it to the server. The server could then have verified that the ticket had not been used before and the ticket was fresh and generated the user's most recent session. Including this modification, could slightly improve the performance of the system as server resources would not need to be used on verifying the signature and checking the basic ticket details if it were already detected as invalid by the management application. This could be helpful in a real-world scenario, where there could be thousands of tickets being scanned in a short period and having a shorter response time would reduce the server's resources.

System Testing

To test the main part of the application, the ticket issuance and verification services, unit tests have been created shown in *Figure 15*, *Figure 16*, *Figure 17*, *Figure 18*, and *Figure 19*. The main unit tests focused on the user adding a ticket for an event, generating the QR code for the ticket and verifying tickets.

The tests for a user adding a ticket for an event tested the functionality of buying a ticket. The first test looked at a user adding a ticket for an event in a valid situation. The other tests reviewed the validation if a user entered an unexpected input, such as buying a large quantity of tickets or selecting an invalid event. A key aspect of security that was tested was the validation of idempotency tokens and tests were written to check that an invalid token cannot be used to buy a ticket, and a used token cannot be replayed.

The tests for the ticket QR data tested the functionality of generating the data for the frontend to use to generate the QR code for the user. The first test checks that the QR data is successfully generated with the correct structure for a user with a valid ticket for an event. The other tests for generating the QR data, test the validation to ensure that a user cannot: generate a non-existent ticket, use the system to generate a new QR code for an old event, generate a ticket that another user owns, or generate QR data for a ticket that has already used.

The ticket validation tests reviewed the possible situations that the verification service should be able to process. Some of the tests reviewed the case when a QR code is scanned that does not follow the normal structure of a ticket generated by the issuance service. There were also tests to test if a ticket had been modified to prevent a user from getting access to an unauthorised event. Additionally, other tests looked at testing how fresh a generated QR code was to ensure that a ticket could not be used from a user's previous login session, or a ticket could not be used twice.

Other unit tests were also included to test other parts of the system such as logging in and signing up. These unit tests ensured that validation was correctly performed on any user inputs and checked if the routes returned the correct responses.

Conclusion

The primary focus of this project was the design and delivery of an implementation of a secure ticketing system for in-person events with a strong emphasis on the security aspects of ticket issuance and verification. QR codes were chosen as the realization technology for tickets with AES and digital signatures being the main employed security primitives. The unit testing shows that all the components are working as expected and the performance evaluation shows that the system is reasonably usable although optimizations are possible, such as performing some client-side ticket verification on the client-side management application.

Limitations

Completing this project would have been easier and more complete had we not faced some limitations. One limitation was the dependencies and compatibility issues such as libraries as the team did not agree on a Python version until later in the project. Another limitation is that our system requires an online part and cannot be used offline, although this can be extended in future work. Also, we had little access to more advanced architecture tools to make sure the application performs efficiently in a real-world scenario with thousands of tickets being scanned each minute, and this is one of the extensions we consider for future work. With regards to the HTTPS connection, a self-signed certificate fits only for a proof-of-concept solution and should be replaced with a real certificate issued by a well-known certificate authority. Finally, a real threat not only to our ticketing system but to every system which relies on cryptography is the quick advent of quantum computing.

Future Work

One possible enhancement in terms of the architecture is splitting our monolithic application into several microservices. For instance, the backend could be split into an authentication microservice, a ticket issuing microservice as well as a ticket validation microservice which can be achieved using containers or virtual machines.

In terms of digital signatures, a possible extension would be to add timestamps so that they can only be valid for a certain period. Currently, the user's session is used to verify that a ticket was generated using the correct session however expiry timestamps could limit the timescale of using the QR code on multiple devices through screenshotting or recording if the human verifier did not check the liveness features.

Another possible extension is having extra verification of a QR code's liveness as it currently depends on human factors, which could mean that it might not be noticed that the QR code presented by a user might is forged. Future work in this area could look at adding verification to the management application to detect whether the liveness feature is correct without requiring human verification.

References

Anon., (2022) *Ed25519 signing - Cryptography* Available at : https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ed25519/ (Accessed : 21 03 2023).

Henry, G. (2019) *Justin Richer on OAuth* s.l.: THE IEEE COMPUTER SOCIETY.

Johnson, M. (2016) *Swish: 3 Takeaways for Every Bank* Available at : https://content.11fs.com/article/swish-3-takeaways-every-bank (Accessed : 23 03 2023).

Appendix

Screenshots of React Application

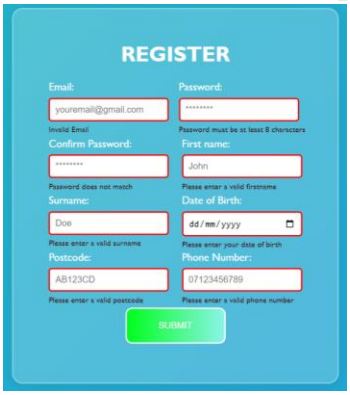

Figure 5: Registration page

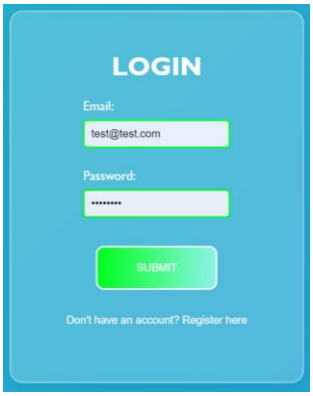

Figure 6: Login Page

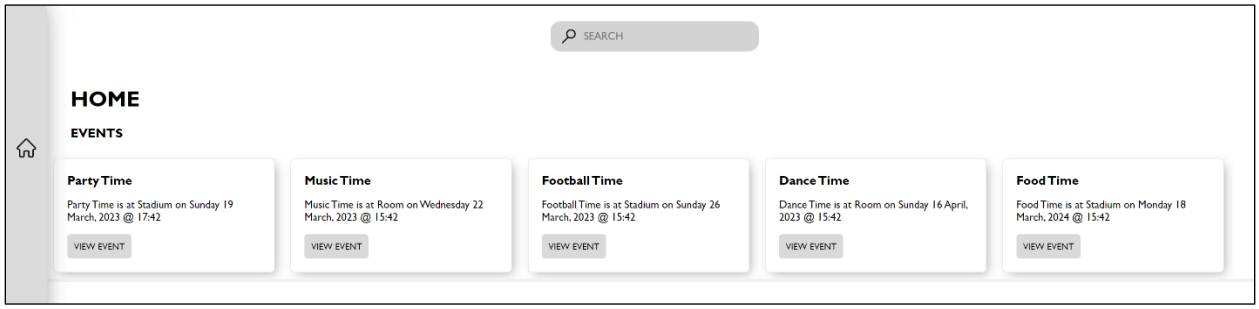

Figure 7: Home Page with event list

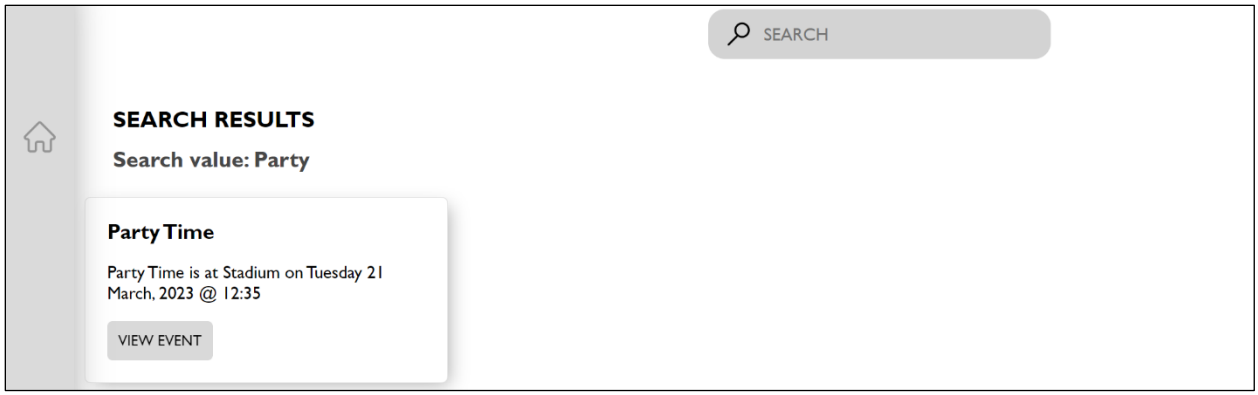

Figure 8: Search Page with event option

Figure 9: Ticket details of specific event

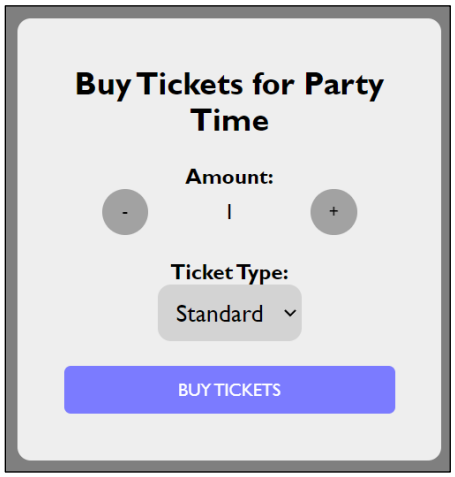

Figure 10: Ability to buy tickets

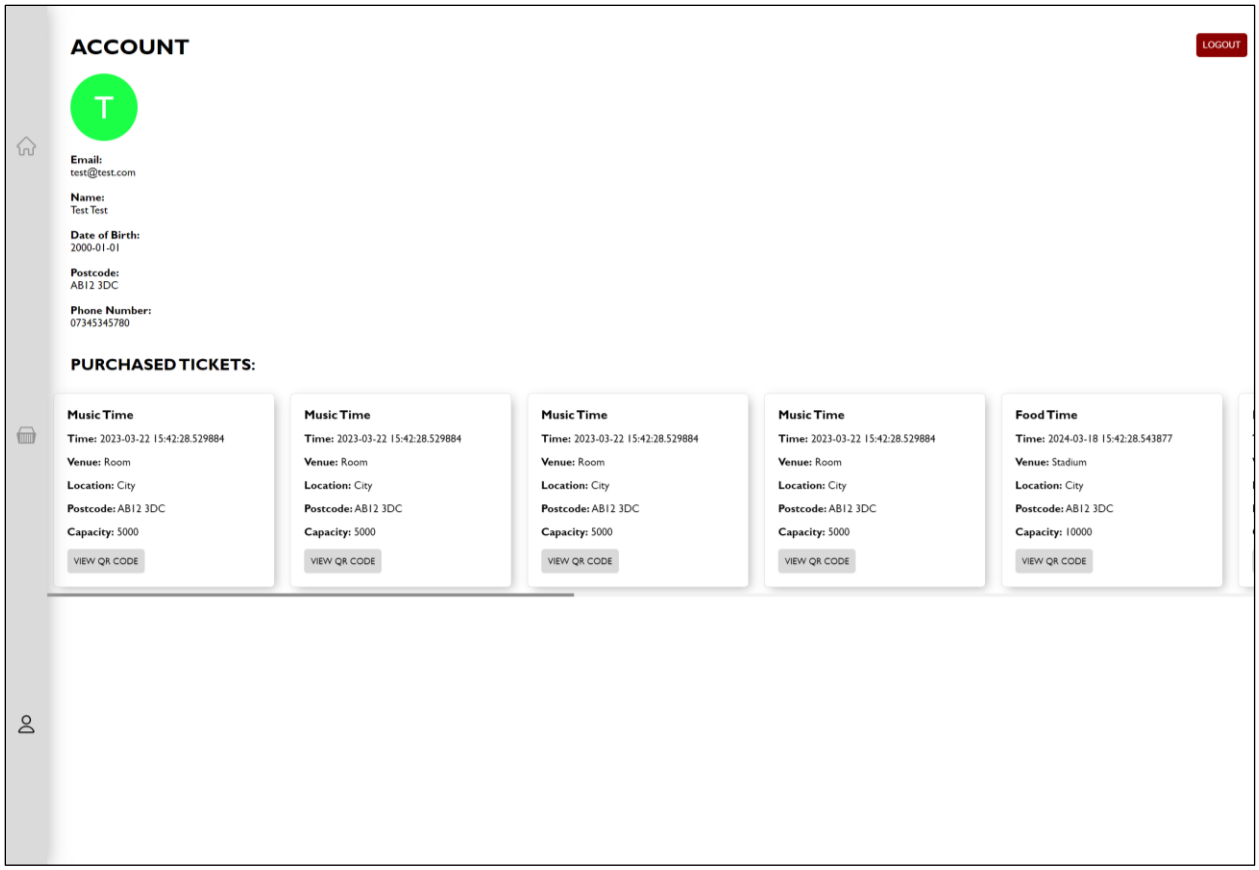

Figure 11: User account page with tickets purchased

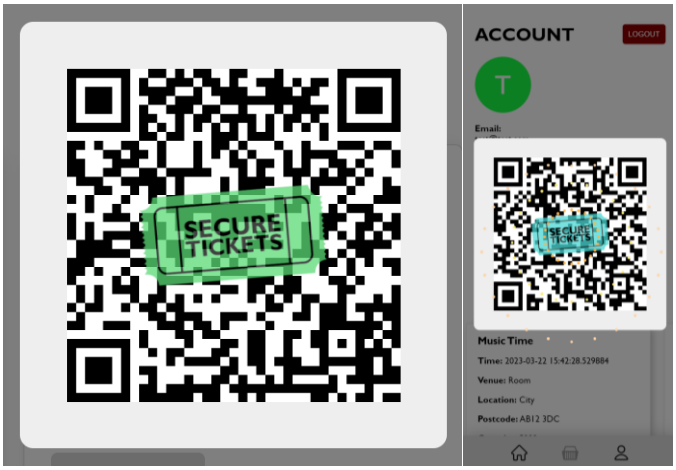

Figure 12: QR code of a valid ticket and animation of 'firework' when QR is clicked

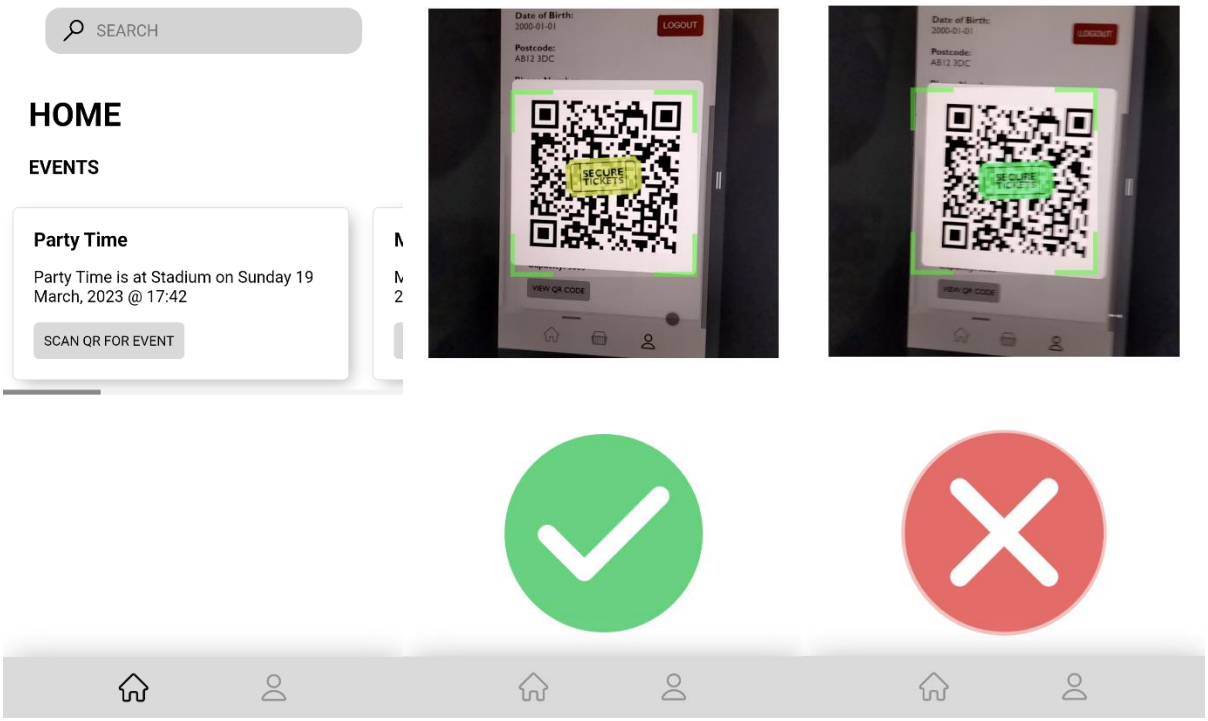

Figure 13: Screenshots of QR code scanning

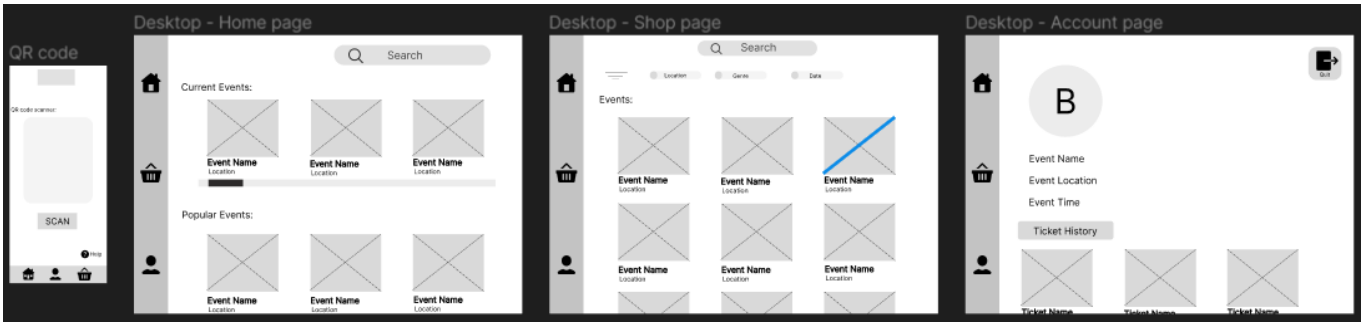

Figure 14: Figma design for the application

Unit Tests

Add Ticket

```
test_flask-server.py::Tests::test_ticket_add
test_flask-server.py::Tests::test_ticket_add_invalid_event
test_flask-server.py::Tests::test_ticket_add_invalid_quantity_large
test_flask-server.py::Tests::test_ticket_add_invalid_quantity_small
test_flask-server.py::Tests::test_ticket_add_invalid_ticket_type
test_flask-server.py::Tests::test_ticket_add_invalid_token
test_flask-server.py::Tests::test_ticket_add_used_token
```

Figure 15: Screenshot showing the add ticket unit tests

Request Ticket QR Data

```
test_flask-server.py::Tests::test_ticket_qr_data
test_flask-server.py::Tests::test_ticket_qr_data_non_existent_ticket
test_flask-server.py::Tests::test_ticket_qr_data_old_ticket
test_flask-server.py::Tests::test_ticket_qr_data_unauthorised_ticket
test_flask-server.py::Tests::test_ticket_qr_data_used_ticket
```

Figure 16: Screenshot showing the request for QR data unit tests

Ticket Validation

```
test_flask-server.py::Tests::test_ticket_validate
test_flask-server.py::Tests::test_ticket_validate_invalid_event
test_flask-server.py::Tests::test_ticket_validate_invalid_signature
test_flask-server.py::Tests::test_ticket_validate_invalid_structure
test_flask-server.py::Tests::test_ticket_validate_invalid_structure_ints
test_flask-server.py::Tests::test_ticket_validate_modified_event_id
test_flask-server.py::Tests::test_ticket_validate_modified_ticket_id
test_flask-server.py::Tests::test_ticket_validate_modified_ticket_type
test_flask-server.py::Tests::test_ticket_validate_non_management
test_flask-server.py::Tests::test_ticket_validate_old_event
test_flask-server.py::Tests::test_ticket_validate_old_session
test_flask-server.py::Tests::test_ticket_validate_previous_qr_code
test_flask-server.py::Tests::test_ticket_validate_used_ticket
```

Figure 17: Screenshot showing the validated ticket unit tests

Events

```
test_flask-server.py::Tests::test_event_details
test_flask-server.py::Tests::test_event_list
test_flask-server.py::Tests::test_event_search
test_flask-server.py::Tests::test_event_search_old
```

Figure 18: Screenshot showing the event-related endpoint unit tests

Users

```
test_flask-server.py::Tests::test_user_account
test_flask-server.py::Tests::test_user_login
test_flask-server.py::Tests::test_user_login_invalid
test_flask-server.py::Tests::test_user_logout
test_flask-server.py::Tests::test_user_signup
test_flask-server.py::Tests::test_user_signup_duplicate_email
test_flask-server.py::Tests::test_user_signup_duplicate_phone_number
test_flask-server.py::Tests::test_user_signup_invalid_dob
test_flask-server.py::Tests::test_user_signup_invalid_email
test_flask-server.py::Tests::test_user_signup_invalid_firstname
test_flask-server.py::Tests::test_user_signup_invalid_password
test_flask-server.py::Tests::test_user_signup_invalid_phone_number_long
test_flask-server.py::Tests::test_user_signup_invalid_phone_number_non_numeric
test_flask-server.py::Tests::test_user_signup_invalid_postcode
```

Figure 19: Screenshot showing the user-related endpoint unit tests

Performance Testing

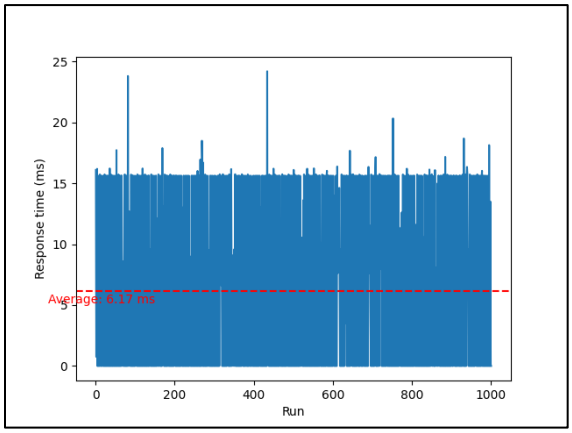

Figure 20: Response time of '/ticket/add' endpoint

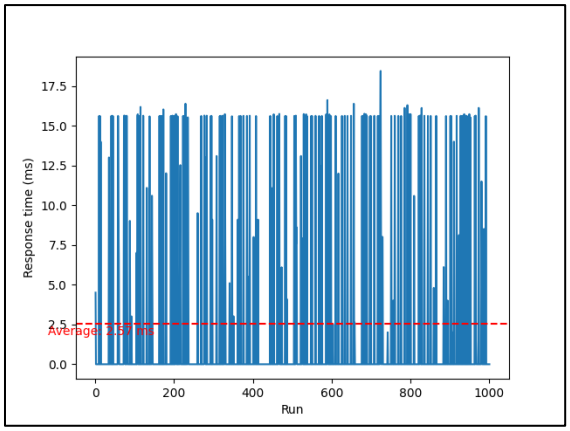

Figure 21: Response time of '/ticket/request_qr_data' endpoint

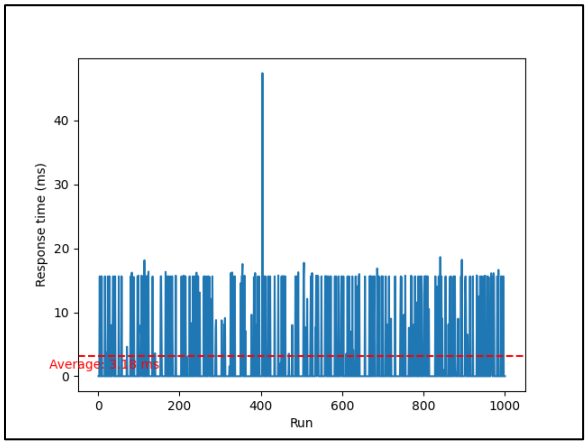

Figure 22: Response time of '/ticket/validate' endpoint