# Cyber Security

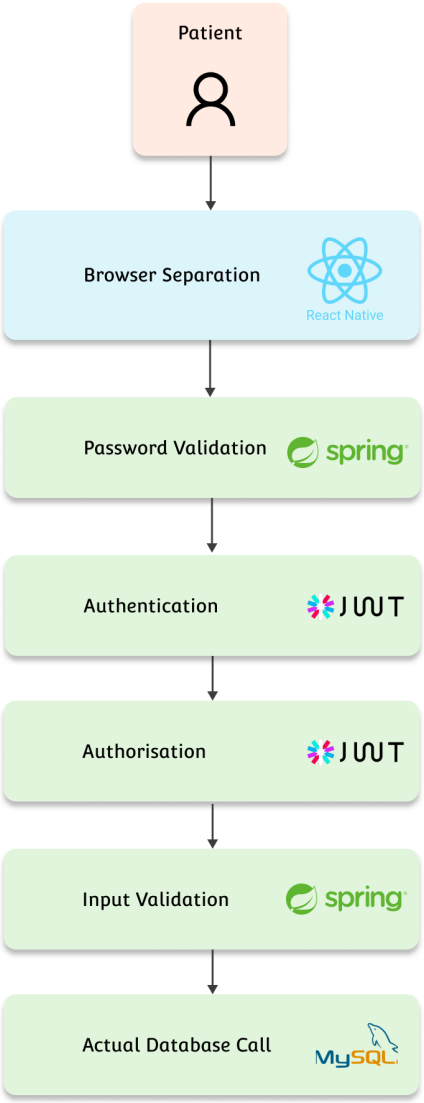


## Overview

Given that the application is related to real patients, data security is incredibly important. Patients should be able to trust that the data they trust to the clinic - and by extension, the MMS app - will be kept completely private and protected, and that no one other than themselves have access to it. Clinics and doctors should also be secure in knowing their internal data (from Genie) is untouchable and secure. In guiding the security decisions, the team has made a risk analysis (available in a subpage of this one), which looks at how damaging certain risks would be, and how important it is to prevent these.

## Security Layers

This diagram depicts the layers of security required to be passed through before accessing/updating any data in the database. Each section is discussed below.




Security Layers

### Browser Separation

Admin accounts (clinic secretaries) require access to more data than patient accounts, and have much more power in the system. As such, the first layer of security is to not use any of the admin endpoints in the mobile application. For example, endpoints such as getAllPatients are not used and filtered on the front end - only the most restrictive endpoints are used by the mobile application. This is beneficial, as if all other security on the mobile app became compromised somehow, bad actors would still not be aware of the required calls to make to access the admin-level privileged data. Of course, this somewhat constitutes 'security by obscurity' so is not at all an entire security suite for the application.

### Password Validation

Providing passwords for users is a fairly complex task in this specific architecture: since patient accounts are generated from data pulled from Genie, they are not involved in the process of creating their accounts, so they are not able to create a password to authenticate themselves. Simultaneously, it is preferred if passwords are not stored in plaintext, as this could cause a security vulnerability if the database ever becomes leaker or somehow exposed. The team's solution for this problem is to generate a fairly complex password automatically and store it only temporarily in plaintext. Once the user decides they want to login to the app, they can download it and enter their email, where they can then receive their password to login; once they first login using the password, the temporary passwords are cleared in the database and only hashes are used. (In a later sprint, the team is also considering moving to a one-time-only link for initial login).

Essentially, for a new patient, the flow for beginning to use the app will be as follows:

- Their accounts are initialised automatically by upload from Genie
  - A secure password is generated on the backend
  - This password is hashed and stored securely
  - This password is also stored in plaintext as a temporary password
    - (Each future Genie upload will regenerate passwords so no temp password is stored long-term)
- When the patient decides they want to, they download the app from the app store / play store
- The app prompts them for an email
- If the email given matches an email in the database, it sends an email with the login details
  - The temporary password is wiped in the system
- User can login securely

### Authentication & Authorisation

All authentication and authorisation works using JSON Web Tokens (JWTs), which once aquired, allow a user to access certain endpoints. The JWTs are cryptographic keys which are tied to a specific user (usernames are the payload), guaranteeing them as the owner when making a specific request, which is performed by including the JWT in the request header of a call to a REST endpoint. JWT tokens expire after some time period (currently set to 1 hour, but this is flexible, and can be changed for a different balance of UX and security). If a JWT expires whilst the user is still in the application, a simple refresh is done in the background, fetching a new token to be used. This allows them to enjoy extended sessions without needing to re-login. Each time the user opens the app, their phone system will remember their login details, so JWT tokens are never stored on-device outside of a logged in session.

JWTs first authenticate users because without the secret key stored on the server, it is impossible to get a jwt of the form required. Hence, all JWTs that will be accepted as valid must originate from the server through a login process..

Secondly, JWTs also authorise users at different levels, such as being an ‘admin’ or a ‘patient’. This means that certain endpoints, such as ‘get all patients’ are accessible only to JWTs linked to an admin account. Similarly, endpoints such as ‘get by patient ID’ require that the caller is actually the patient with that ID, which is verified using the JWT process.

### Input Validation

While all of these security layers essentially guarantee that users are who they say they are and can only access what they are permitted to, there are still some edge-case risks. If bona fide patients / admins decide to begin doing bad things, or their emails get hacked, there still needs to be prevention of attacks such as SQL injection and other similar checks, which we do through multiple layers of input validation.

All POST object requests, such as ‘create admin’ use Kotlin objects as the request bodies, rather than JSON of any arbitrary size. Given this, the types must match exactly, and are cast into these types. For example, if a request object requires a key, “x” (Int) and a key “y” (String), then any request not providing exactly and only these two keys in JSON will fail to be cast to a Kotlin object and the entire request will fail. This prevents against bad faith actors who have compromised an account using cURL or similar to make dangerous requests through injection.

Furthermore, we use JPA as the only entry point to the database; this means that no lines of SQL are actually being performed by software on the backend, meaning there is no point for someone to add in SQL lines to call, even if they wanted to. The range of actions permissible on the database is strictly defined by the JPA object and commands that exist at compile time.

## Encryption

To ensure all data is protected against eavesdropping attacks or other data leaks, all information passed between the mobile app and the backend, as well as the web app and the backend are encrypted using HTTPS. The SSL certificate for this is stored only in the server environment. (As of writing, this is not fully developed as a feature yet)

## Data storage

The application is built with the principle of ‘store only if needed’ in mind. Hence, we do not store particularly sensitive data in any case, especially where it is not completely necessary to be storing it. For example, appointments do not store the actual illness details that those appointments relate to, as this is highly sensitive if leaked. Whilst we take the protection of the data we do store very seriously, we generally prefer to still only store the data that is absolutely necessary for functionality.

This limited storage of data is made possible by the integration with Genie software, which remains the only source of truth in the application. Given this, we do not allow the update of most data directly in our application backend (All the ‘create X’ endpoints are test only). For example, appointments and user tables can only be updated by uploading Genie reports, not by adding them in the web app. This limits any data loss risk, as everything stored in our service is simply a mirror of Genie, which is the only source of truth.

A summary of key, potentially sensitive data that we do store:

- Demographic data
  - Names
  - Addresses
  - Emails
  - Dates of Birth
- Appointment data
  - Time/date details
  - Location/provider details
  - Doctor notes*

*Doctor notes is particularly challenging; we don’t actually intend on storing this data, but we have noticed that occasionally doctors will use the ‘name’ column in the table to store notes about the appointment, including illness details. This means that we may ‘accidentally’ store illness details which are - in our view - more sensitive than the other demographic data discussed above. Given that in our view, patients have not necessarily consented to this data being stored by the app (it is pitched as being appointment details only), we have made it a point to raise this with the client. She has suggested that in future, her team will be more strict on this issue, and ensure that doctors do not make notes in the name column. However, given that this may appear in the data we store (if doctors do not follow this advice), we are taking the safe storage of all patient data very seriously.