**Architecture and Frameworks Assignment 1 Technical Justification**

Frontend (client-side)

I chose to use React as my framework for the frontend of this ticketing system project. I felt that this project was a good opportunity for me to improve my understanding and confidence using React. React was also a good choice because of its templating abilities which makes systems easier to write and maintain. React’s JSX components also have useful functions which are called at different stages of the component’s lifecycle which are very useful to be able to hook onto as a developer. React is also a very popular, well known, framework which has a lot of modules available to speed up development by removing tedious tasks or making common functionality easier to implement. React’s ability to be reactive, and render dynamic content without having to reload the page also played a large part in my choosing React as my framework. An alternative frontend framework I considered was VueJS - I am familiar with VueJS and understand how to use it. After researching, I concluded that React would be the most appropriate framework for this project. One source I read on React was *JavaScript frameworks: Angular vs React vs Vue* by Elar Saks which states ‘react is designed to enhance interactive UI development by making it easier to update the view when the data changes. It is done through dividing the view into smaller components that can be composed to create complex UIs. Components are built in JavaScript instead of templates, enabling easy flow of data.’

I imported a number of module dependencies for the frontend. I installed axios to make asynchronous requests to my backend easy. Axios allows me to make asynchronous calls to the backend, define the HTTP request method for the call and send a payload with the request. Axios’ calls return promises which is also very useful when doing frontend error handling and executing functionality within a callback once a response from the request has been received. I included react-router-dom as a dependency because it made defining routes easier on the frontend. It allowed me to define custom route middleware, handle 404 errors and easily create a multi-page web app with React. I used the *npm audit* command to check for vulnerabilities within imported dependencies which helps assure the system is secure.

I utilized the react-router-dom to create middleware and apply these to different routes. This meant I could add requirements for users to be able to access certain routes on the frontend - this resulted in a better UX and a more secure and stable system. I also used react-router-dom to help me handle 404 requests. The authentication middlewares for accessing routes were largely based around a local storage value set when the user logs in and logs out.

I chose to create a local login session on the client-side when a user logs in and logs out. This means that I can reference this within frontend middleware to restrict / grant access to pages. Although this local storage session can be easily manipulated directly in the browser’s console by malicious users, this local storage session is only referenced to in frontend middleware. If a user was to modify the local storage session to gain access to routes then requests to the backend on this route would be blocked because more reliable security checks are done on the backend for every request. This means even if a user gains access to a frontend route, they would be no closer to accessing unauthorised routes on the backend which includes reading / writing of any data. The local storage authentication login session’s main purpose is to provide a better UX.

To make the code easier to navigate and maintain, I used a folder structure which helped to keep the files organised. Within the src folder of the client-side system I created components, middleware and pages folders. Components were React JSX components which are going to be nested inside other components / pages. Pages are React JS pages which routes were linked to. Middleware are the custom routes used to grant / restrict access to routes. The CSS styling which is specific for pages / components are kept within separate CSS files and are stored in the same directory as the relevant JSX file. Any CSS styling which is used globally across the system is stored in App.css. Using a folder structure like this made it easier for me to work on the system as it grew in size.

I set up a proxy on the frontend system to point to the backend system so any relative requests on the frontend are forwarded onto the backend. This made writing requests to the backend easier.

I achieved error handling on the frontend in a few different ways. One way was by taking advantage of promises - this allowed me to use the *catch* method on promises to identify and handle errors. Once I have the errors I can display them to the user. I use this to catch system errors as well as error messages which I defined on the backend. For example if the user’s data input is not valid then the backend checks the data and returns relevant validation error messages which are then displayed to the user through the UI. Error handling within a system is important because it allows you to make a system fail gracefully when errors occur which creates a better user experience. Users also often expect some feedback from the system when there is an error which is why I display the errors on the UI.

I used React’s testing library and Jest to write frontend unit tests. Writing these tests allowed me to quickly and easily know if units were failing. The tests would watch for changes and re-run so I could make a change and watch the test suite to make sure that change hasn’t broken any units. Using test scripts like this are much more efficient than manually checking units. I also used BrowserStack to test my web app on different real-world mobile devices to make sure the UI was responsive and looked good cross-platform.

Backend (server-side)

Express

When choosing the server-side framework for this project, the options were Express with Mongoose (NoSQL database) or Laravel with PhpMyAdmin (SQL database). I am already confident in using Laravel to create restFUL APIs because I have used this on multiple projects before. I have never worked with Express, Mongoose / MongoDB / NoSQL databases before so there would be a learning curve whilst I gained experience and knowledge. Express would allow me to write the backend in JavaScript which would make the ticketing system as a whole easier to maintain for juniors who do not know PHP, which is what Laravel uses. Express JS provides solid implementation of commonly-required server-side functionality. Express can handle request parsing, routing, handling of request parameters and embedded payloads which would be useful in the development of the ticketing system. Express also has a good developer community behind it which makes it stable and efficient, and has easy integration of helpful modules which makes writing code easier. By handling common parts of developing a system, Express increases productivity and reduces development timescales. For these reasons, I chose Express as my server-side framework

A collection of modules were installed to the backend system. Here are the modules I installed, and their purposes:

* Bcryptjs - provide an encryption library for salting and hashing user passwords before they’re stored in the database. This makes the data more secure because passwords are not stored as plain text in a human-readable format.
* Body-parser - middleware that extracts the body of an incoming request and exposes it on req.body.
* Cookie-parser - middleware that extracts cookies of an incoming request and exposes them to req.cookies.
* Cors - used to allow the frontend system and the backend system communicate on from different ports
* Dotenv - creates and utilises a .env file in the system which allows environment variables to be declared and referenced to from anywhere in the system.
* Express-mongo-sanitize - middleware that sanitises user-provided data to mitigate unauthorised, malicious injection into MongoDB.
* Express-rate-limit - middleware that prevents repeated requests to the backend. This helps mitigate DDOS attacks and brute force attacks.
* Express-validator - middleware that allows data to be validated against rules before being passed to the controller. User-friendly error messages are generated and passed back to the client if validation is not met.
* Helmet - helps secure Express systems by setting various headers. Helmet can help fight against XSS attacks.
* Jsonwebtoken - generate secure authentication tokens for authentication sessions
* Messagebird - used for SMS 2 factor authentication
* Mongoose - makes interactions with MongoDB databases easier
* Xss-clean - middleware for sanitising user inputs in requests to the backend
* Nodemon (developer dependency) - watches for changes to the backend code and restarts the express server automatically

I create a connection to the MongoDB database before the backend starts listening for requests. This ensures that there is a successful connection to the database before any requests are accepted - this is a clean way of preventing database interactions from being executed when there isn’t a successful connection to the database. I find that creating the database connection right at the beginning of the server’s script keeps the code clean, rather than creating a connection within the controller / controller methods.

I split the project into a folder structure which made the system easier to develop and maintain. The folder structure I used created a clear split between routes, controllers, middleware and models. Each model has its own model file, controller file and routing file which I find useful because I can easily navigate the project and know where everything can be found without having to search large files. I also think using a folder structure, like I have described, makes it easier for other developers to work on a system because they can also find code easier even if they didn’t write it themselves.

I achieved authentication by using the jwtwebtoken module which allowed me to easily create secure authentication tokens. Once the token is generated, I can pass it back to the frontend as a HTTP-only cookie. Using a HTTP-only cookie helps mitigate cross-site scripting attacks. Setting the secure property on the cookie means that the cookie will only be sent over HTTPS which helps add more security. When registering, I use the messagebird module to add 2FA via SMS to verify new users. I also only allow a user to get their password incorrect 5 times, at which stage the account is locked - the account has to be unlocked by a system admin. This mitigates brute force attacks. A user’s locked / verified status is checked as a part of the authentication middleware. The user’s username must be unique within the database when registering - this is checked within the registration process, with validation messages being shown to the user where necessary. The user’s password must also meet criteria to ensure the password is a reasonable strength to mitigate password being guessed - the password is validated within middleware validation.

I wrote middleware for input data validation and authentication restrictions to endpoints. Using middleware for input data validation is a good way to validate data server-side before it hits the controller’s method. The module I used to create validation middleware is easy to use and generates user-friendly validation errors which are passed back to the frontend so the user can be notified. I wrote middleware for user authentication which adds an important layer of security to an API - writing authentication checks as middleware is a cleaner and more efficient way than checking authentication inside each controller method, then middleware can be easily applied to routes. The authentication middleware uses the cookie-parser module to read the jwt token cookie from the request, verify the token, retrieve the user which the token belongs to and adds the user to the request so it can be easily accessed within controller methods. If a token is not present, is present but cannot be verified or the user does not have sufficient permissions, appropriate HTTP error responses are returned. Authenticating requests on the server-side is more secure than only validating on the frontend because the server-side is reading a HTTP-only cookie which is harder to be manipulated than a client-side login session.

When ‘deleting’ a model, the model is marked as deleted in the database with the timestamp at which it was marked as deleted instead of the record being deleted from the database. This creates a better user experience because users can restore the deleted models. The intention is for an automated script like a cron job to automatically delete records from the database if they have been marked as deleted for 30 days.

I wrote a seeding script which allowed me to easily clean a database with freshly seeded data to help me whilst developing. Seeding scripts are far more efficient than manually recreating a dataset each time you want a clean database. The seeding script quickly provides me with a base dataset that allows me to test my entire system with. I opted to use Postman to help me execute unit tests on my backend. Postman makes it very easy to write custom tests - it lets you define endpoints, the HTTP verb / method of the request and payloads. This allowed me to develop the API completely independently from the frontend because I did not need to develop a UI to interact with the API. I have included a collection of my tests that I used against my backend with my project. Here is the specification for each Postman test:

| HTTP Verb / Method | Postman Test Name | Description | Controller Method | Request Data |
| --- | --- | --- | --- | --- |
| POST | New verified user | Create a new user that is verified upon creation | createVerified | Request payload:  *username*: string,  *email*: string,  *password*: string |
| POST | New user | Create a new user using 2FA SMS for user verification | create | Request payload:  *username*: string,  *email*: string,  *password*: string |
| POST | Login | Validates and verifies the user’s login credentials to create a login session | login | Request payload:  *email*: string,  *password*: string |
| GET | Logout | Ends the login session by clearing jwttoken in cookie | logout | n/a |
| POST | Send verification code | Sends a 6 digits verification code to the user’s SMS for 2FA | sendVerification | Request payload:  *phoneNumber*: string |
| POST | Verify user code | The verification request ID is used to check the user provided 6 digit verification token for user verification. | verify | Request payload:  *requestId*: string,  *token*: integer |
| GET | Read all users | Read all users from the database | getAll | N/A |
| GET | Get authenticated user | Get the current authenticated user model | getAuth | N/A |
| DELETE | Soft delete by ID | Mark a user as deleted | softDelete | Endpoint url users/{userId} |
| PUT | Update user | Update a user model | updateById | Endpoint url users/{userId}  Request payload:  A json object where the properties and values are the properties and values you wish to be updated within the model |
| GET | Read all tickets | Get all of the tickets the authenticated user has access to. This depends on user type (permissions) | getAll | N/A |
| GET | Read single ticket | Get a ticket model from ID | getById | Endpoint url tickets/{ticketId} |
| POST | Create ticket | Create a new ticket and assign it to the authenticated user | create | Request payload:  JSON object with *title* and *description* values |
| PUT | Update ticket | Update a ticket model by ID | updateById | Endpoint url tickets/{ticketId}  Request payload:  JSON object where properties and values represent properties and values that wish to be updated within the model |
| DELETE | Soft delete ticket | Mark a ticket as deleted | softDelete | Endpoint url tickets/{ticketId} |
| POST | Create comment | Create a new comment within a ticket | create | Request payload:  *content*: string,  *ticket*: string |
| PUT | Update comment | Update a comment model | updateById | Endpoint url comments/{commentId}  Request payload:  JSON object where properties and values represent properties and values that wish to be updated within the model |
| DELETE | Soft delete comments | Mark a comment as deleted | softDelete | Endpoint url comments/{commentId} |

General Choices used Throughout

I used ES6 JavaScript syntax on the frontend and backend systems. This provides shorthand syntax which makes code easier to write, read and maintain.

I make use of promises, async and await functionality to execute code asynchronously on a different thread and execute code within callback of the request. If I made an asynchronous request, but the next part of the script depending on a response from the request, I could use the await keyword to stop the script executing further until a response has been received.

I used comments where I felt it was necessary. I have not commented on every line of code because I assume that any developers working on this system can read basic JavaScript as they are familiar with express / node and react.

References

* *JavaScript frameworks: Angular vs React vs Vue,* Elar Saks, page 2 - https://www.theseus.fi/bitstream/handle/10024/261970/Thesis-Elar-Saks.pdf?sequence=2