What patterns were used

The design patterns that were used by our group were Singleton, Dependency Injection, Repository and Decorator.

Where and how they were implemented

Singleton: Two areas of the program that implemented the Singleton pattern were in the SQLRepository class and the OnlineConsoleLogger class. The Singleton pattern requires that only one instance of a class exists throughout an application. Specifically, SQLRepository overrides the __new__() function to see if the class variable _instance already exists, where if it does, it returns the already existing instance, otherwise it will create and store a new instance. Similarly, OnlineConsoleLogger uses the class variable _db_connection which is shared for all logger instances, which ensures that there is only one database connection for all logging purposes.

Dependency Injection: Two areas of the program that implemented the Dependency Injection pattern were in the register_endpoints() function and the MasterPi class. The Dependency Injection pattern requires that dependencies, such as objects and configurations, are passed into a component as a parameter, and not having them created internally. Specifically, register_endpoints(app, config) has config passed in as a parameter, rather than reading the configuration from a variable inside of the function. Similarly, MasterPi's create_app() function also has config passed in through its parameters, rather than having the config read by a variable in the function.

Repository: Three areas of the program that implemented the Repository pattern were in the DatabaseRepository class, SQLRepository class and the classes that use the handle(self, data, config=None) function, such as handle_login_user or handle_register. The Repository pattern works by defining an abstract interface with data access methods, having concrete classes that handle database operations, and having business logic call repository methods rather than writing SQL directly in these classes. This can be seen in the DatabaseRepository class, as this is the abstract class that defines the data access methods that should be used. The concrete class that handles the database operations is the SQLRepository class, which implements the DatabaseRepository class. The business logic is seen in the handle() function, which calls repo.fetch_one() on the repository, which queries the database even though this method does not know where the data is coming from.

Decorator: Two areas of the program that implemented the Decorator pattern were the logger_decorator() function and the middleware.py files. The Decorator pattern works by wrapping a function with additional behaviour without modifying the functions code directly. This is seen in the logger_decorator() function, as this function allows other functions to be wrapped when @logger_decorator() is written above the function. This will cause calls to the fetch_all() function to be wrapped by the decorator. Similarly, decorators are used in the middleware.py files, where decorators such as @app.before_request and @app.after_request cause all requests to be automatically wrapped with authentication checking and MQTT management logic.

<u>Why each pattern was suitable</u>

Singleton: The Singleton pattern was suitable here primarily for resource management, as Raspberry Pi's have a limited amount of memory and CPU power, and consistency, to ensure that all database calls use the same connection instance, which could avoid potential connection issues.

Dependency Injection: The Dependency Injection pattern was suitable here primarily to allow for the different roles to receive each of their individual configurations without having to hardcode them into each role.

Repository: The Repository pattern was suitable as it allowed for centralized data access through the repository, which made it easy to log interactions with the database. By separating these layers, it also made it easier to modify and bugfix the code, as specific issues could only occur in certain places. On top of this, using the Repository pattern caused unit testing to be easier, as we were able to mock the repository, rather than requiring actual database connections.

Decorators: Decorators are suitable as features such as logging and authentication are required for many functions throughout the program, and this provides a way for these features to be implemented without needing to modify any code directly.

<u>Any challenges or alternatives considered</u>

Singleton: One challenge relating to the Singleton pattern was to do with unit testing, as mocking singleton instances was somewhat complex, requiring sys.modules manipulation to prevent actual database connections. This caused us to consider the alternative of not using Singleton and instead creating a new connection per request in this scenario, however we decided not to as this could have potentially resulted in even more issues, or at the least a very large increase in connection time and resource usage.

Dependency Injection: One challenge relating to the Dependency Injection pattern was to do with python's dynamic typing, which caused configuration bugs to not be obvious until runtime.

Repository: One challenge relating to the Repository pattern was to do with users registering and logging into the system, as initially the system would communicate directly with the database rather than going through these layers of abstraction. This was resolved by refactoring the registration and login endpoints to use JsonSocketClient to communicate with the Master Pi.

Decorators: An alternative to decorators was to fully commit to using the Middleware pattern, however, we opted to only keep aspects of the middleware and not fully adopt it, as this would’ve made it a lot more difficult to apply specific middleware logic to certain database features, where instead using decorators selectively made this process a lot more simple.