CSTS

Concrete Architecture Report

Esra Kastrati - 215507205

Mostafa Mohamed - 215830292

Yassin Mohamed - 213867064

November 16th, 2020

Abstract

This report details the concrete architecture of the Common Spaces Tracking System as it will be built. It starts off outlining the design process of the system from the start until the current phase, then describes in detail the architecture styles present in the system as it was implemented. The report contrasts between the previous architecture choices detailed in the conceptual report, and the actual choices the team went with. It details the system's implementation of the Layered and Microservice software architectures, and the connections and interactions between the different classes and subsystems present in the software.

Introduction

In the early stages of working on our project, our group was concerned first with the real-world domain. We started by examining the events and circumstances of people in our community, and what sort of software solutions we could implement to aid with the current difficulties everyone is facing. Through this examination we decided to build a Common Spaces Tracking System to allow establishments to make use of the resources that have become underutilized due to the pandemic. To that end, we wanted our system to accomplish certain goals like providing clients with the ability to monitor who goes in and out of their spaces, historical data about who has been in a certain location, and overall provide an easy to use interface for our clients. We then laid out an essential user story that characterized the overall behavior of our system. That research and preparation was the starting point for our design phase.

During the design phase, we started by working out the requirements of our software. This step involved moving down from the abstract visualisation of the purpose of our project to a more concrete vision of the specific things the system needs to accomplish. We established a set of

functional requirements that provided specific details as to the functionalities offered by the software and the conditions they need to meet, and a set of non-functional requirements that dealt with the requirements of the aspects of our system that aren't directly related to the functionality of the software; for instance the performance, security and availability requirements. With our goal for the software system well defined, and a robust set of requirements that lay out its functionalities, our team started examining the different architecture styles and patterns that would best compliment our needs.

Deciding our software system's conceptual architecture was one of the most crucial points in the design process. Despite the fact that we were still looking at different architectures from an abstract standpoint, and weren't tied to decisions we made, choosing an abstract architecture was, in some ways, the culmination of the work we had done up to that point. Our goal was to find architecture styles and patterns that corresponded the closest to the goals and requirements of our project. This was a challenging task, as by design different architecture styles and patterns have pros and cons both, and our team set out to find different styles that we could incorporate into our system design to meet our requirements as closely as possible and with the least compromise.

This report details our chosen architecture patterns for the implementation of our software system. It discusses the choices that we had considered in the conceptual architecture report, and how we made the choice between different alternatives. In addition, it outlines the different patterns that we've chosen to incorporate into our implementation. Finally, it examines in greater detail the main subsystem of our application; responsible for adding and removing users to existing common spaces.

Architecture

In the conceptual architecture report, we discussed the client-server architecture, and how it might suit the needs of our software system. Upon further research, we found that the client-server architecture was more suited for software systems that rely more heavily on a centralized computing system, which any number of clients can connect to. Our system is a web application, and requires very little resources to run, so a client-server architecture would impose unnecessary restrictions on our design for no benefits. After that point we started examining our other options in the conceptual design and other previously unthought of design patterns that would fit our project.

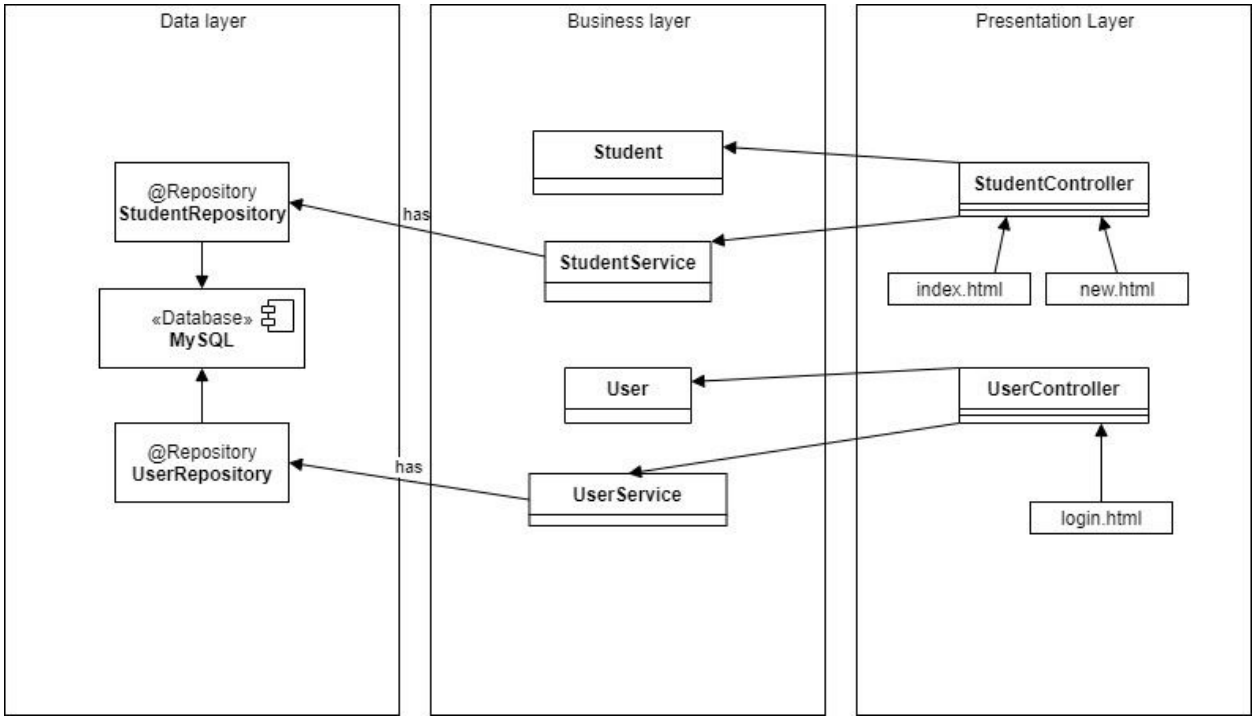
Since we had decided in the conceptual architecture report that the internal design pattern of our application would follow the Model-View-Controller pattern, we started looking at it as our starting design pattern, around which we will adapt the system. We started looking through

different MVC frameworks online that fit our purposes. During our research we reexamined the different design patterns that were considered previously, and realized that a Layered architecture as the backbone of our software would provide all the benefits of the different patterns that we considered, while also not constraining our design.

As such, our system implements the Layered software architecture. This architecture and design pattern is primarily characterized by two main things: the different "layers" of the system are well defined and share a functionality (Presentation, Business etc.), and each layer can only communicate with the layer above and below it. This decision allowed us to start working on different parts of the system without having all the details of implementation pre-determined. It also allowed us freedom to integrate third-party services and components into our system.

For our purposes, we chose to combine the persistence layer, responsible for the entity objects and for translating objects into SQL, with the database layer, containing only our database. This meant we ended up with a 3-layer architecture, seen in the diagram below. This is different from the 3-tier architecture style we had chosen, as the 3-tier design pattern is concerned mostly with separation of tasks and not as much about the interactions between the different layers.

Layered software architecture



From the diagram above we can see that the application is divided into three distinct layers, each of which only interacts with the layer(s) around it. The three layers present in our system are:

Presentation layer - handles the http requests, contains the controller classes and the views (html), handles mapping from client requests to business elements.

Business layer - handles all the business elements of our system, contains the StudentService and UserService classes, which handle the services offered by the software

Data layer - contains the database itself and all storage logic, translates classes into database objects and passes them to the database.

This design was a natural fit for our project, and we adopted it as the main structure of the system.

The other main architecture pattern that guided our design was the Microservices based architecture. Our system offers a specific set of services that require a small number of steps to accomplish, so to that end, we decided to have specific classes for each service, these classes are only concerned with one task and only communicate with the classes they need to accomplish their tasks. The microservice classes are Studentservice and Userservice.

Finally, the repository design pattern was a necessary inclusion as our system needs to have persistent storage that can be accessed by anyone using the web application. The repository is implemented by the Spring boot framework which we built our project on. As previously discussed, one of the main advantages of the layered architecture pattern is it allows us more freedom in terms of the inner workings of the layers, which gave us more space to choose the implementation of the repository design pattern that would work best with our system. As such, we decided to use the Spring boot framework instead of a standard MVC framework, as Spring boot internalizes a lot of the necessary set up and steps that go into database operations.

To ensure concurrency in accessing and modifying the database, we used ACID transaction-compliant databases. ACID transactions ensure that database operations are performed in a timely manner, without interruption from other access attempts. The database locking functionalities are provided by the Java Persistence API (JPA).

Software Quality Attributes

In our design we defined several quality attributes through which we can measure the quality of our system and guide our design choices.

Adaptability - The system will function on various web browsers and hardware devices

Part of our decision to go with Spring boot for the framework of our project is its compatibility with any HTML and Javascript compatible web browser. This lets us guarantee our system would run on all expected platforms.

Availability – The system should be accessible on clients' browsers most of the time

We chose to design our system as a web application since that ensured the highest possible uptime, as it depends on whether or not the domain host is online.

Correctness – All rooms will have the correct information.

Form validation is used when users enter information to ensure databases have consistent and correct formats.

Flexibility - The system can run automatically for the user without developer interaction.

Being a web application, the system can run for any user as long as the domain host is online.

Interoperability - Standard interface implementation provides potential for future implementation of different interfaces to add or change functions.

The layered structure of the Spring boot framework allows us to change implementation details in one layer without affecting the rest of the code, as long as we maintain the same interface between the layers.

Maintainability - The system will be developed in a way to make adding functionalities easier, this would also benefit developers when fixing possible bugs.

Layered software architecture simplifies the process of adding functionalities and fixing bugs.

Portability - The system should run on various operating systems.

The system will run on any operating system that can run a modern web-browser.

Robustness - Errors in querying the database will be displayed to the user in real-time. Unprecedented system failures will be logged for debugging.

Logging component ensures developers follow up on bugs.

Testability - updates to the system can be tested locally to see functionality before implementation.

Spring boot allows us to run the application locally to preview changes before deploying it on the server.

Usability – User's experience with the system is simple and intuitive to use.

Web page design follows best practice guidelines to ensure useability and accessibility.

Third-party technologies/COTS

Java 8 - Class-based object-oriented programming language with which the program is written.

Maven 3.2 - Software management tool for managing software structure and dependency.

Spring Boot 2.3 - The Java-based framework on which we based our application. It allows us to easily separate our application into 4 layers and simplifies the integration of the microservice architecture.

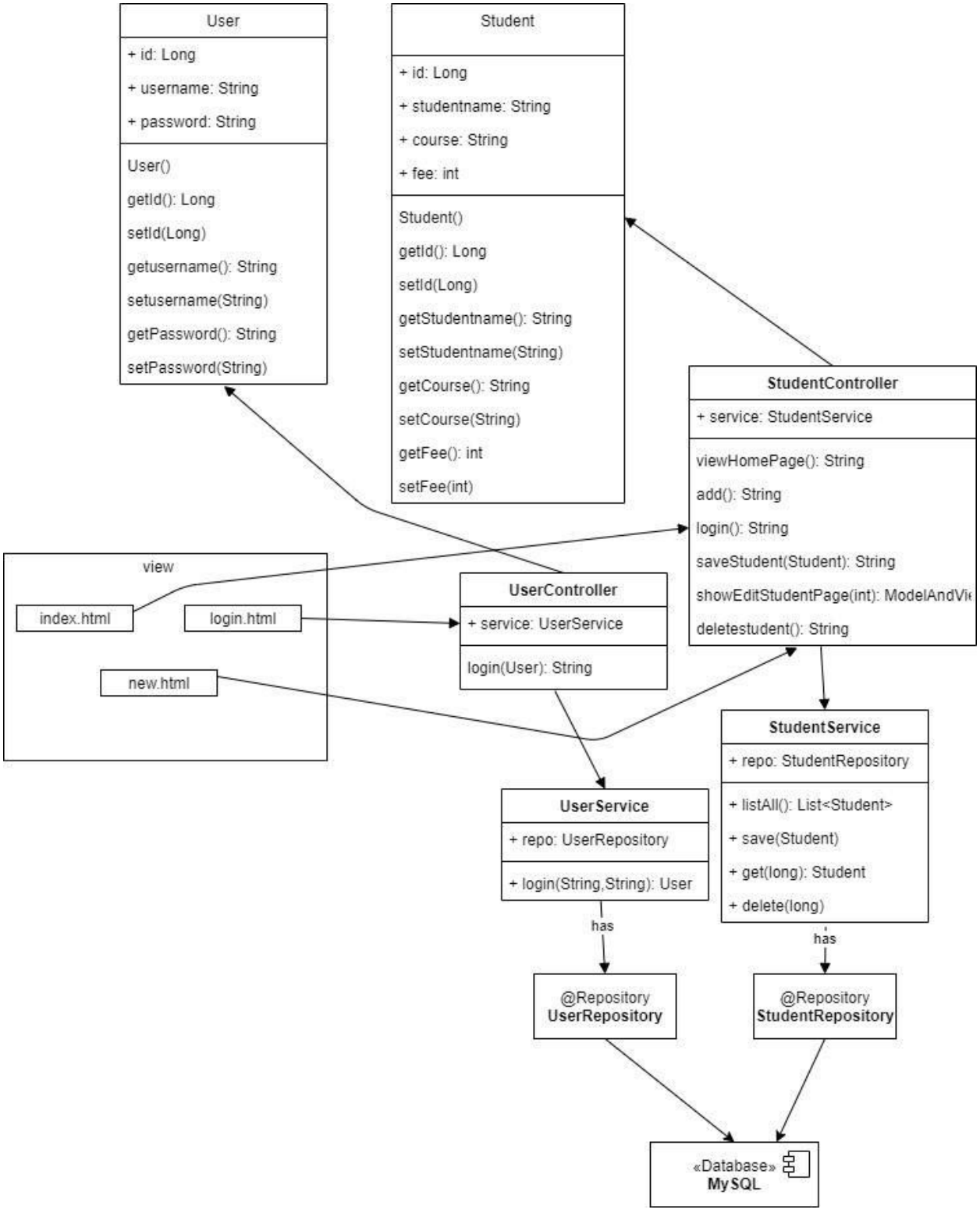
Spring Data JPA - Provides the functionalities of the Java Persistence API (JPA) repository in the Data layer

Thymeleaf - Provides premade templates and complex functionalities to HTML pages.

MySQL - Relational database management system used by our software to store data about rooms.

Primary subsystem

Diagram of subsystem responsible for adding/removing students from common spaces



In the Spring Boot framework, the duties of the front-controller are provided by the framework itself, and coordinated through mappings to ensure the correct control-flow. Control is essentially passed to the internal front-controller every time, which then redirects control to the appropriate resources depending on the mappings. For the sake of detailing the structure of our system, we will skip mentioning the in-between step of passing through the hidden controller, as it simply facilitates the connections that we established when building our system. The default directory of our application directs the user to the index.html view. Depending on the user's input, the view then redirects control to the appropriate controller, which calls one of the services in the business layer. That service, through dependency injection, calls the appropriate data persistence objects in the data layer, which translates the action into an SQL statement and passes it into the database.

Use cases, sequence diagrams, component and deployment diagrams

Use Case Diagram with relevant cases

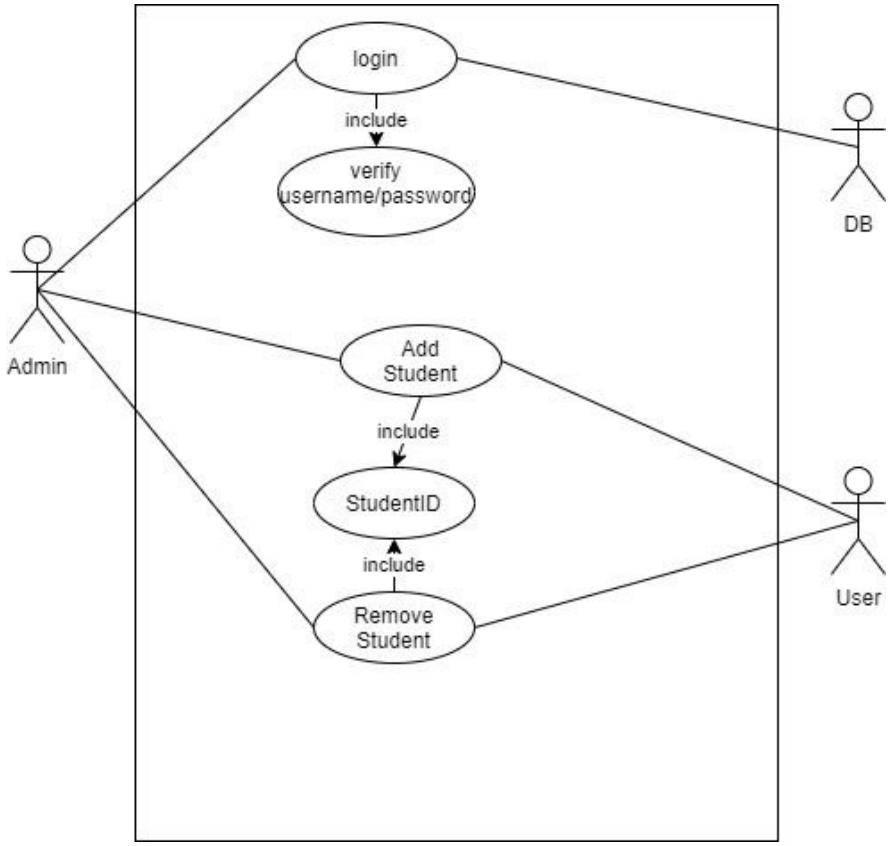
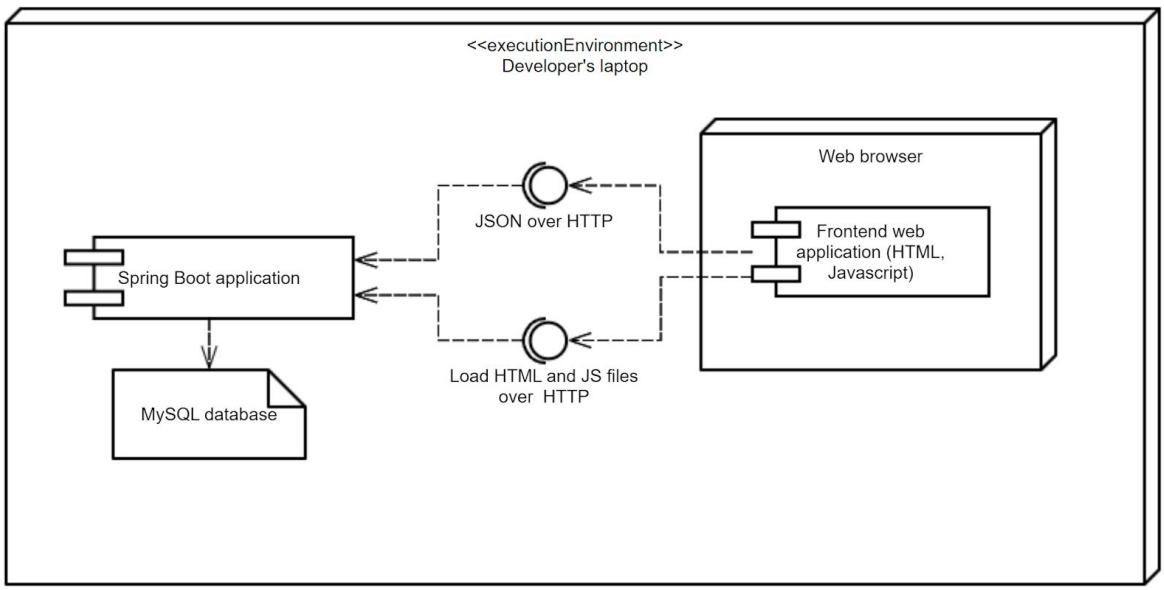
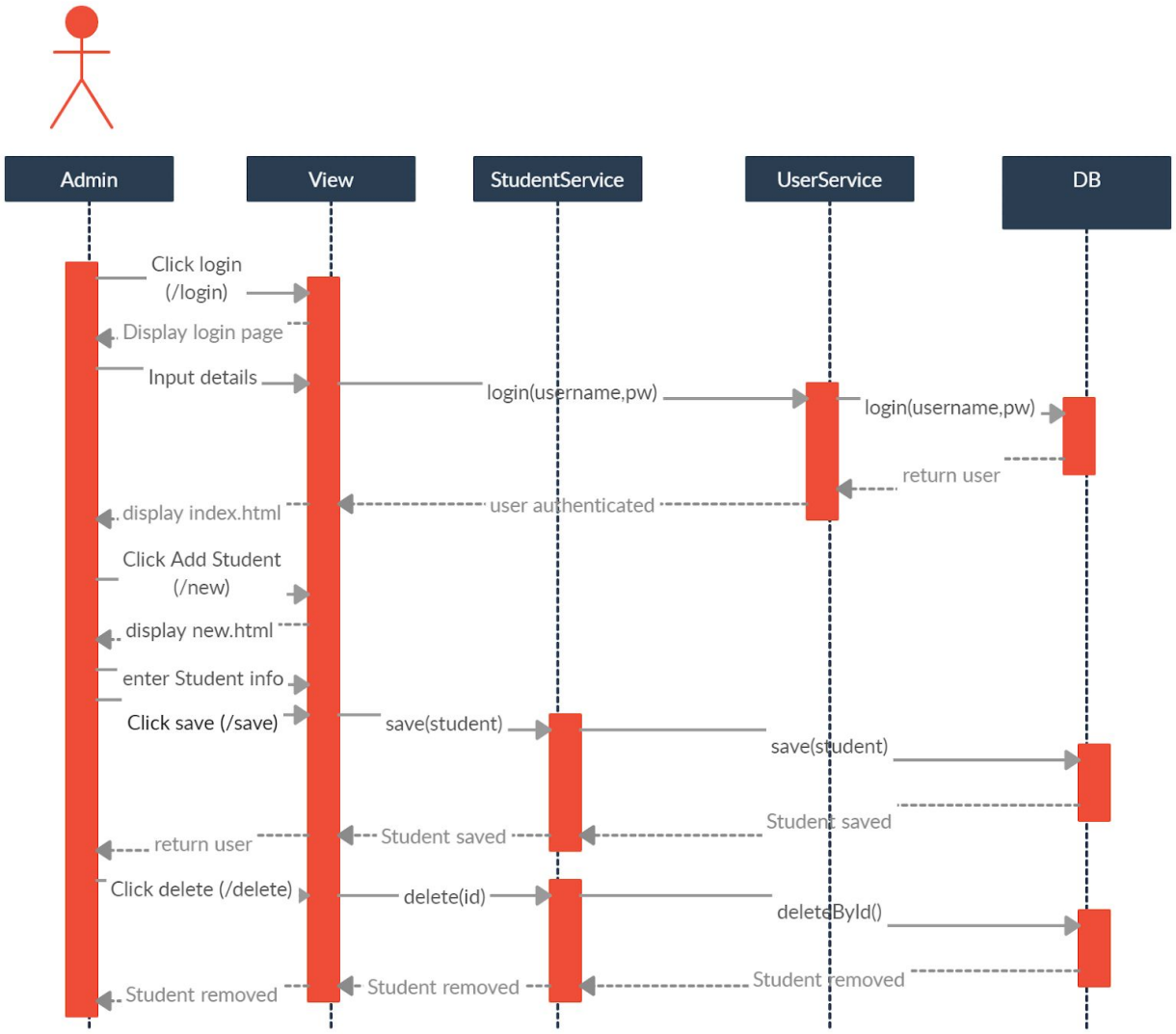


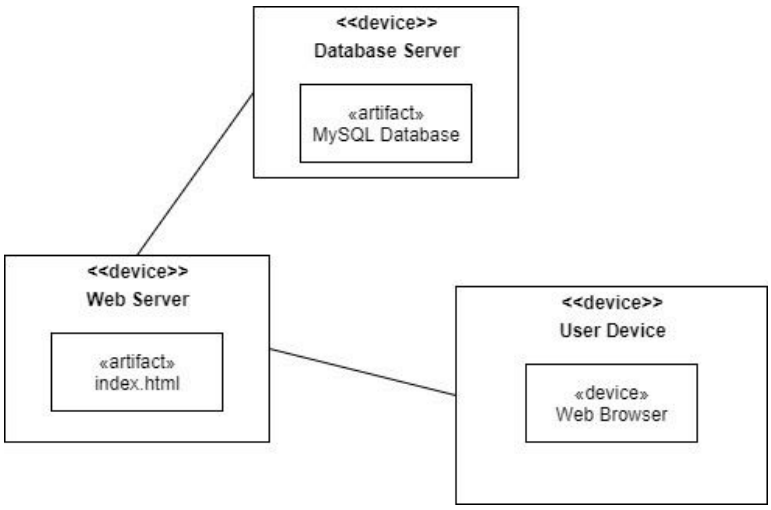
*Diagram of components responsible for maintaining applicat*ion



Sequence diagram for an Administrator of the system



Deployment diagram



Conclusion

The process for designing a complete software system is a complicated one, necessitating constant revisions and reworking of past assumptions. When working on our project we first started by examining the real-world domain and defining the requirements of our software based on real-world problems. We then researched and examined different software architecture patterns to potentially use in our system. Then, after weighing the different options we could take, we settled on our architecture pattern of choice and set to implementing the system. As detailed in this report, the overarching architecture of our system is the Layered software architecture, characterized by the separation of the code into distinct layers that share a functionality and only interact with adjacent layers. The system also implements the Microservices software architecture, ensuring services take up as little resources as possible and relegating each service to performing one task, or a set of strongly related features. We then detailed the chosen Quality Attributes of our system and how they relate to our design choices. Finally, the report detailed one main subsystem and its component, class and deployment diagrams. Overall, the chosen architecture patterns suit the purposes of our system very well, and provide the needed benefits with minimal compromises.