Smart Employee Management System [SEMS]

Concrete Architecture

15.11.2020

Muhammad Arifur Rahman
Samuel On
Jaiveer Singh
Dong Jae Lee
Karmit Patel

Website: http://pentad-x5.unaux.com/
Github: https://github.com/arifrahmanca/Smart_Employee_Management_System
SEMS Application: https://employee-managment-f5252.firebaseapp.com/
Source Code: https://github.com/KarmitP98/Employee-Managment.git

Overview

The primary goal of this document is to provide the concrete service architecture for the Smart Employee Management System or SEMS for short. SEMS is a cloud-based software application designed to reproduce the traditional employee management system in a virtual environment. In this phase of architectural design, the focus is to implement two of the top-level sub-systems of SEMS namely the User Management subsystem and Leave Service subsystem. Through implementation, the system is better understood and as such that updated understanding of the system is detailed within this document. First, a broad overview of the system is given by explaining the overall architecture of the system and providing an overview of the major subsystems of SEMS. Then detailed class diagrams and component diagrams are used to better understand the system. Further, use case diagrams and sequence diagrams are used to illustrate how users interact with the system and how data flows within the system from such interactions. Finally, the deployment strategy is discussed, and comparisons are made between the conceptual architecture and concrete architecture.

Description of High-Level Architecture

The three-tier architecture is divided up into three parts: presentation layer (client tier), Application layer (business tier), and the database layer (data tier). Within our system, the client tier acts as an external web interface the end-users will interface with and have access to different system features of the webpage. The main function of this layer is to communicate with the lower application layer as it will pass on the user actions that will need to be executed. Within the application layer of our system which will act as the main core, it will be responsible for handling the user actions that were passed on from the client tier. This layer will be the intermediary layer that will process the data received from the client tier as well as the data tier. The data tier is the layer that will store all the data necessary for the system to run and will comprise the database system. In the case of our system, the database will be from a firebase. And data access will occur via API calls from the business tier when required.

There are a few benefits we gain from using the three-tier architecture compared to using the traditional client-server architecture in our system. By segregating the database by putting in an intermediate layer, the application layer in between the client tier and the database, the system is more secure as the clients are not allowed to directly communicate with the database. Due to the separation of layers, each tier can scale horizontally and load balancing in each tier can help reduce the network utilization in other layers. Our system also implements the model-view-controller

(MVC) architecture which greatly differs from the three-tier architecture described above.

Although one may find similarities between the MVC and the three-tier architecture described above, the fundamental concept behind the two architectures shows why both of these were necessary for our system. While the three-tier architecture presents itself to have a linear structure where the data flows from the presentation layer to the database layer and back, the MVC resembles a triangular form. The view in the MVC is part of the application that is responsible for the presentation of the data and will be the main interface that the end-users will observe. The controller is part of the system that handles the user interactions and updates the model acting as the data when needed. Afterwards, the model will alert the view that it has been changed and the view will be updated accordingly. All three components, model, view, and the controller will communicate with each other, unlike the three-tier architecture which only interacts with the layer that is adjacent to it.

Another architecture our system utilizes is the service-oriented architecture or SOA for short. The system will be reliant on ReCaptcha service from Google and two-factor authentication from Google Authenticator. By implementing the reCAPTCHA at login, the system will be able to validate real users from automated access trying to get into the system, enhancing the security. Likewise, the two-factor authentication will put another layer to the end-user login process making it much difficult for a threat actor to gain unauthorized access into the system. The main purpose of using these services from the service provider, which is Google in the case of our system, would be the capability to allow possible substitution of these services when necessary. Since these services are not managed by our team, it allows the system to be able to substitute any of these two services at any point. Depending on the change in the requirements of the system, these services can be linked and unlinked at any time which brings a huge advantage in terms of system management with one less component to manage.

Third-party Services and COTS

Our team has proceeded with using service-oriented architecture or SOA within our system and incorporates third-party services from Google. As proposed in previous documents our team has proceeded to use a CAPTCHA system from Google and the system is reliant on Google reCAPTCHA API to be up and running alongside our system. For storage of data, unlike the initially proposed Amazon Web Services for the system's cloud database, the system will entrust its data management responsibility to Firebase. Both the reCAPTCHA and Firebase database is a commercial off-the-shelf or COTS product where our system is delegating

responsibilities such as user verification and managing data available on the cloud. Dissimilar to a physical database, our system requires an always-on cloud database that can be accessed from any endpoints reachable by a network connection.

Software Quality Attributes & Design Tactics:

Availability:

SEMS shall be available for use 24x7x365 and shall achieve 99.5% uptime and SEMS shall inform users of any planned unavailability such as system maintenance. These were both achieved through being hosted by the firebase server which in their Service Commitment agree says at least 99.95% uptime percentage [1].

Reliability:

SEMS shall not corrupt or delete user's data and any update of user records shall be saved effectively to the database. This is achieved by using a firebase database that hosts our data on multiple servers.

Correctness:

SEMS shall not allow the ability to run any defective source codes that can cause storing wrong data, performing wrong calculations, or initiating infinite loops causing a crash of the system. The design tactic used to achieve this is the MVC model that allows the separation of computing to the controller and maintaining the single responsibility design principle for our services. SEMS shall provide real-time data to the end-users. This is achieved by our choice in architecture which is MVC.

Modifiability:

SEMS shall allow users to add new features or delete existing features without discontinuing the services. For example, SEMS shall allow Employers to add new employees in the system or remove an existing employee without affecting the functionality of other users.

Usability:

SEMS shall support operations in different operating systems or web browsers as well as in multiple platforms including Computer or Mobile devices and shall be accessible remotely. The design tactic chosen was to create a website that is supported by the internet. SEMS shall be easy to use for the users. The design tactic used was to focus on the functionality and button visibility such as timesheet and leave applications.

Overview of the major subsystems:

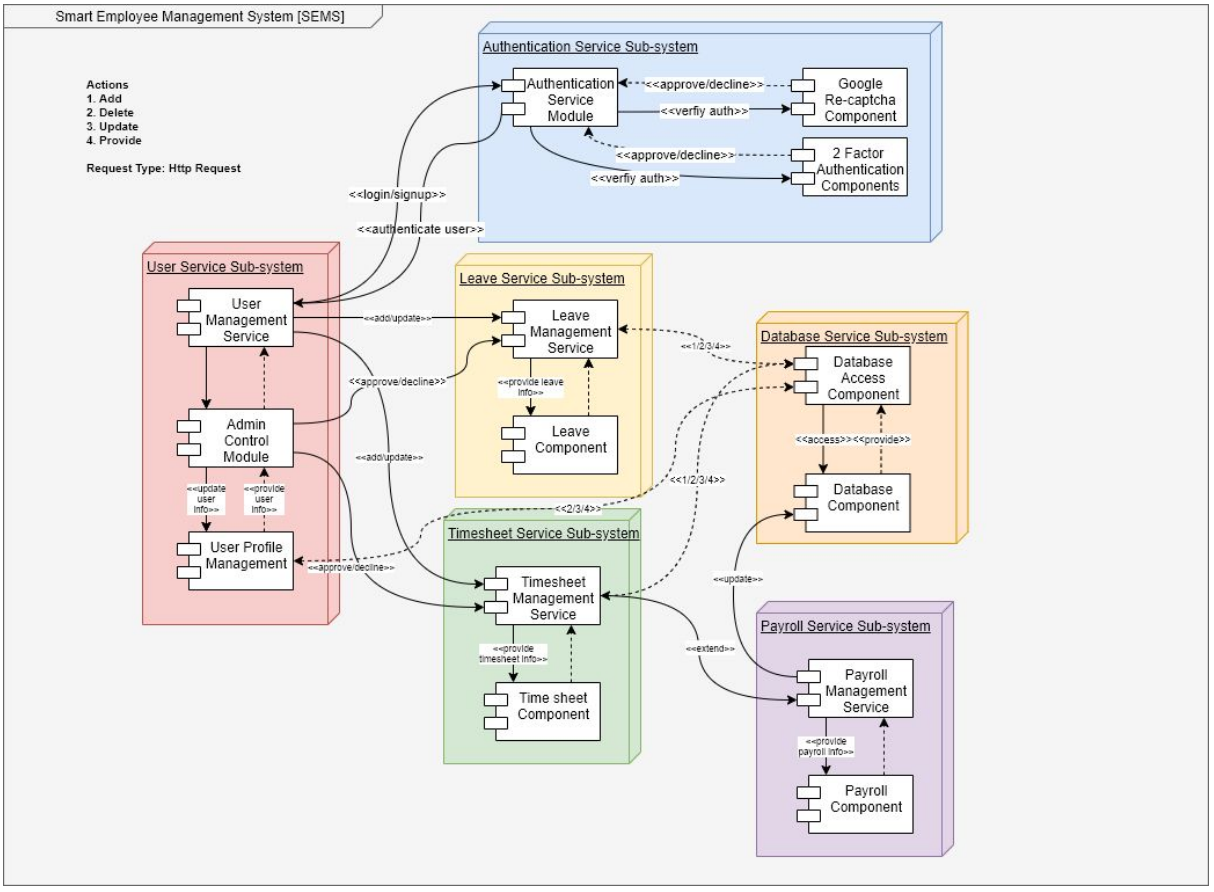

Figure - 1: Shows the top-level Sub-system diagram

Our system has divided into the following major subsystems:

- **User Service Sub-system:** This subsystem manages to create, updating, and deleting user information. It also acts as a verification layer for different levels of users in the system.
- **Timesheet Service Sub-system:** This subsystem allows users to create, update or delete work logs. It also manages to access and update work logs with the database subsystem.
- **Leave Service Sub-system:** This subsystem provides the functionality to request leave by employees and approve or decline leave requests by HR employees. This also manages to access and to update leaves from the database subsystem.
- **Payroll Service Sub-system:** This sub-system consists of components that allow the employees to view their invoices and HR employees to approve or disapprove payroll. It also manages to access timesheets from the Timesheet Service subsystem and provides payroll data to the database subsystem.

- **Authentication Service Sub-system:** This subsystem manages authentication of users during login and sign-up. Its major function is to maintain the uniqueness of each user and data integrity during the login process.
- **Database Service Sub-system:** This subsystem handles the storage of the entire software. It consists of components and 3rd party cloud services provided by Firebase to handle simple HTTP requests like get, put, update, deletes. The database component consists of a SQL database that allows the system to perform specific queries and real-time data updates.

The following component diagram represents a better understanding of how the subsystems are organized and interconnected with each other:

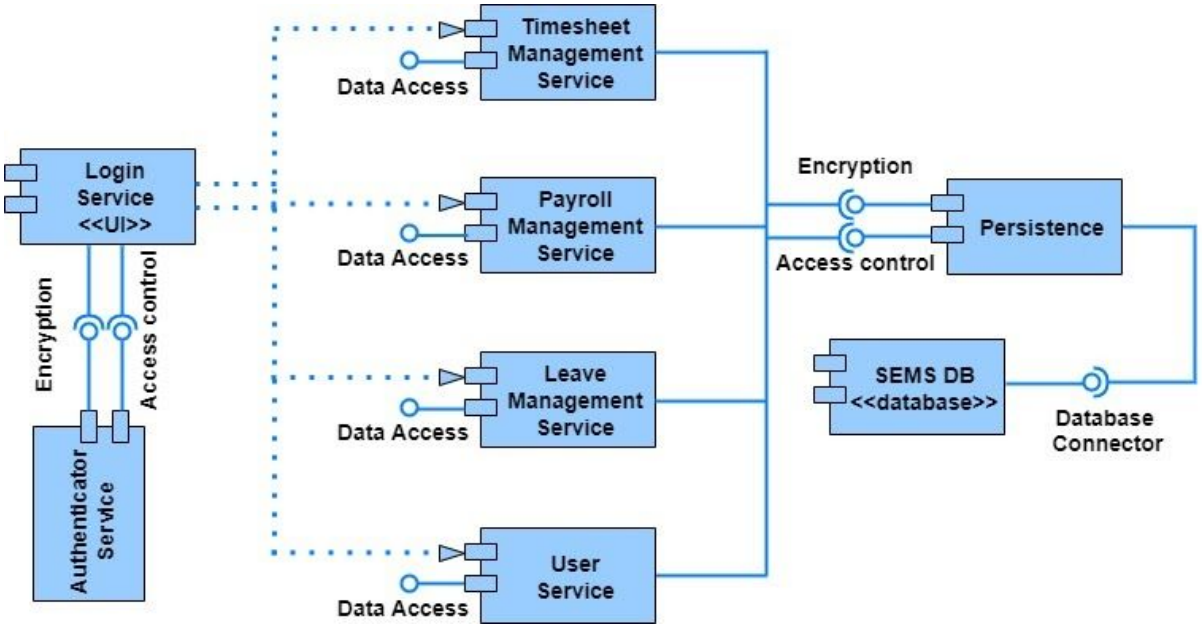

Figure – 2: Component Diagram of Smart Employee Management System.

Class Diagram:

Our system implements the MVC (model-view-controller) architecture to facilitate building the system that separates different logical aspects. The class SEMSModel implements the business logic and can access the database via a Data Access Object (DAO) layer (EmployeeDAO, TimesheetDAO, PayrollDAO, and LeaveDAO). Only SEMSModel has access to the Database and SEMSController can only access data that is provided by SEMSModel. This single class access of the database has been used to reduce coupling of subsystems. The class diagram below represents both the MVC and 3-tier architectures with the only exception that the view part in MVC architecture and the presentation layer in 3-tier architecture is absent.

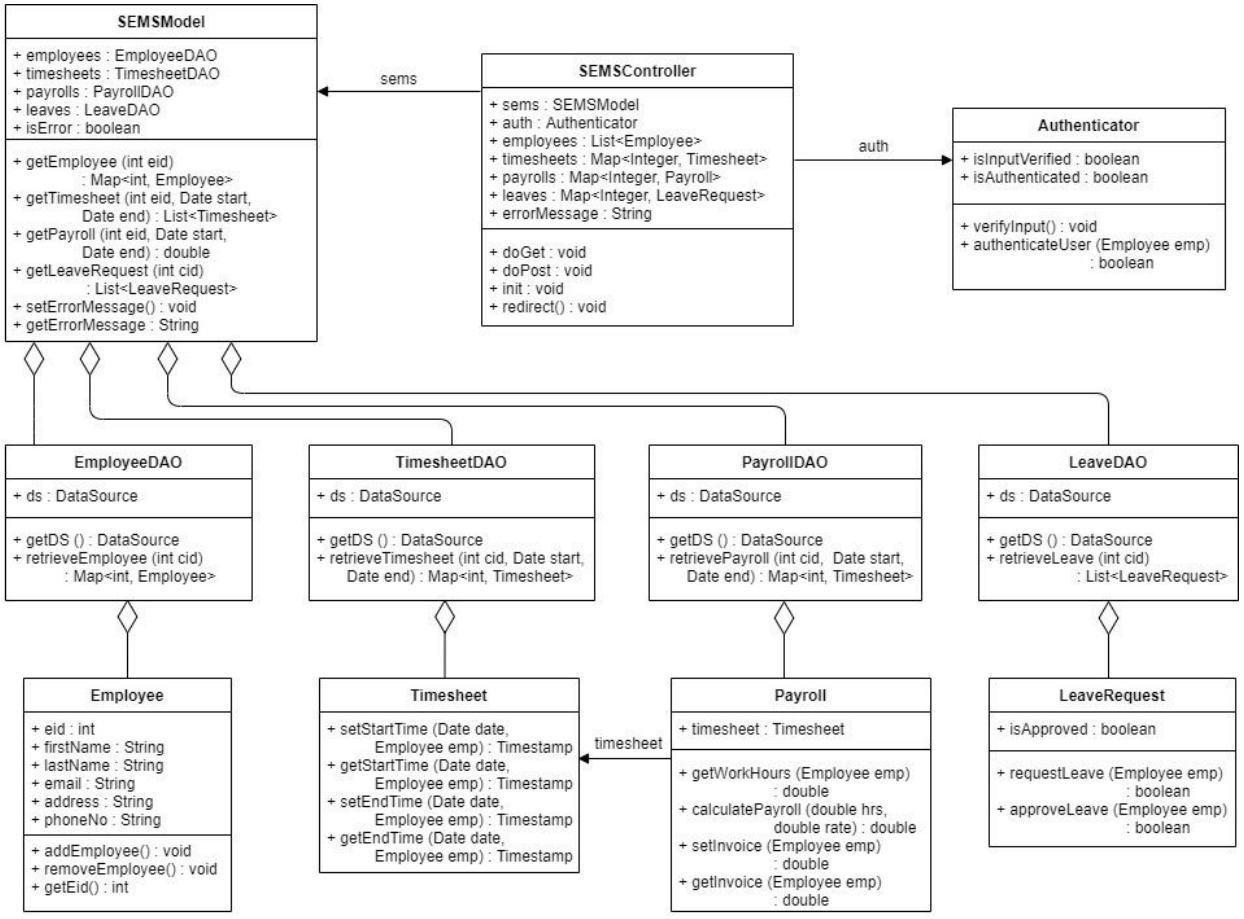

Figure – 3: Class Diagram exhibiting MVC and 3-tier Architectures.

Description of use cases utilizing major Subsystems:

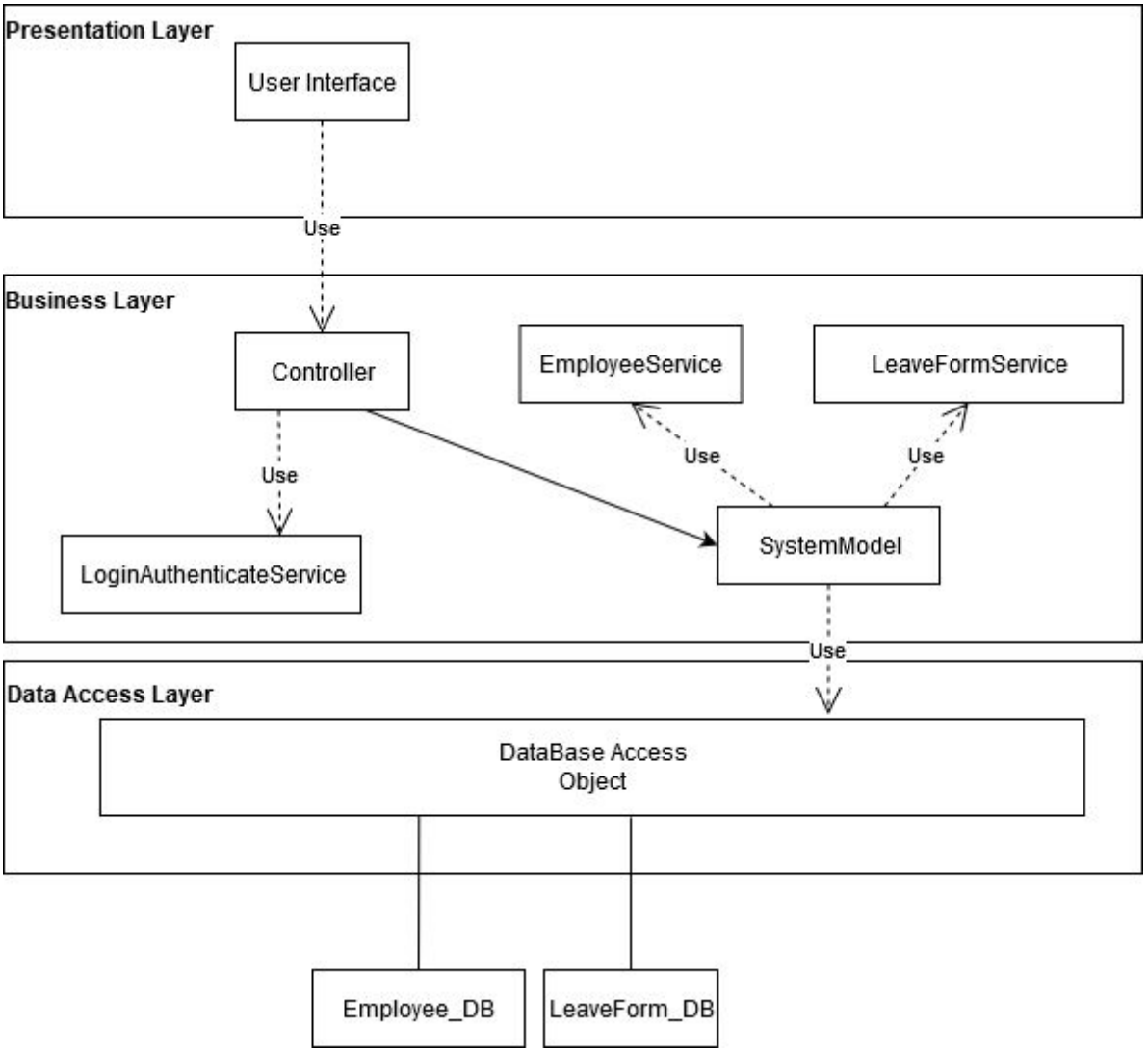

Figure – 4: Subsystem of use case scenarios

In this subsystem, we are modelling our 2 use cases which are (1) update employee information as administrator and (2) Logging in and inputting leave applications as employees. We are now using 3-tier throughout our whole system and MVC to control our user interface. MVC allows you to separate each of the functions required to drive the UI while N-Tier ensures layers will access only in a linear order, so only necessary components can communicate. The model has access to the services and databases that allows it to maintain statefulness, the single responsibility design. The controller's sole job is to manipulate the data in the model and use the AuthenticateService for the job.

Use Case (1):

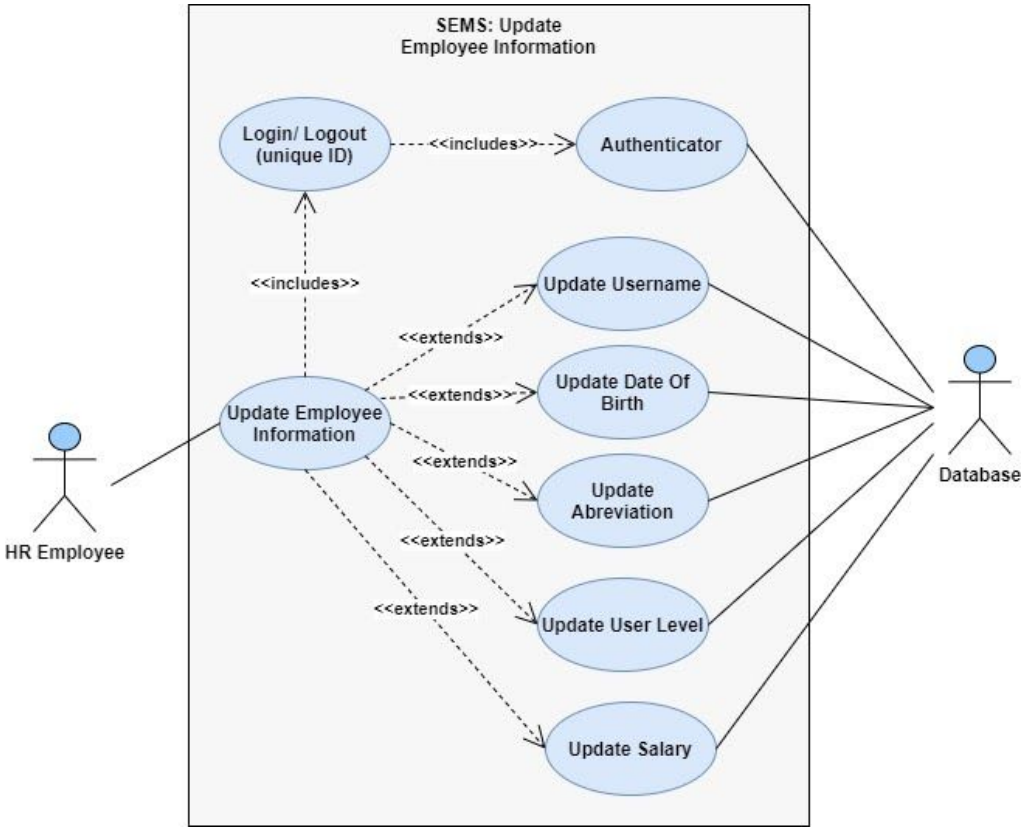

Figure – 5: Use case - Update Employee Information

Use Case Name: Update Employee Information
System Name: SEMS
Level: User
Primary Actor: HR Employee
Stakeholders: HR Employee, System Database
Precondition: The HR Employee has logged into the system using a valid username and password.
Success Postcondition:

- An updated employee profile.

Main Success Scenario:

1. The HR Employee selects “Update Employee Information” from the main menu.

Alternative Flow:

1a. The HR Employee updates Username.

1b. The HR Employee updates Date of Birth.

1c. The HR Employee updates Abbreviation.

1d. The HR Employee updates User Level.

1e. The HR Employee updates Salary.

Use Case (2):

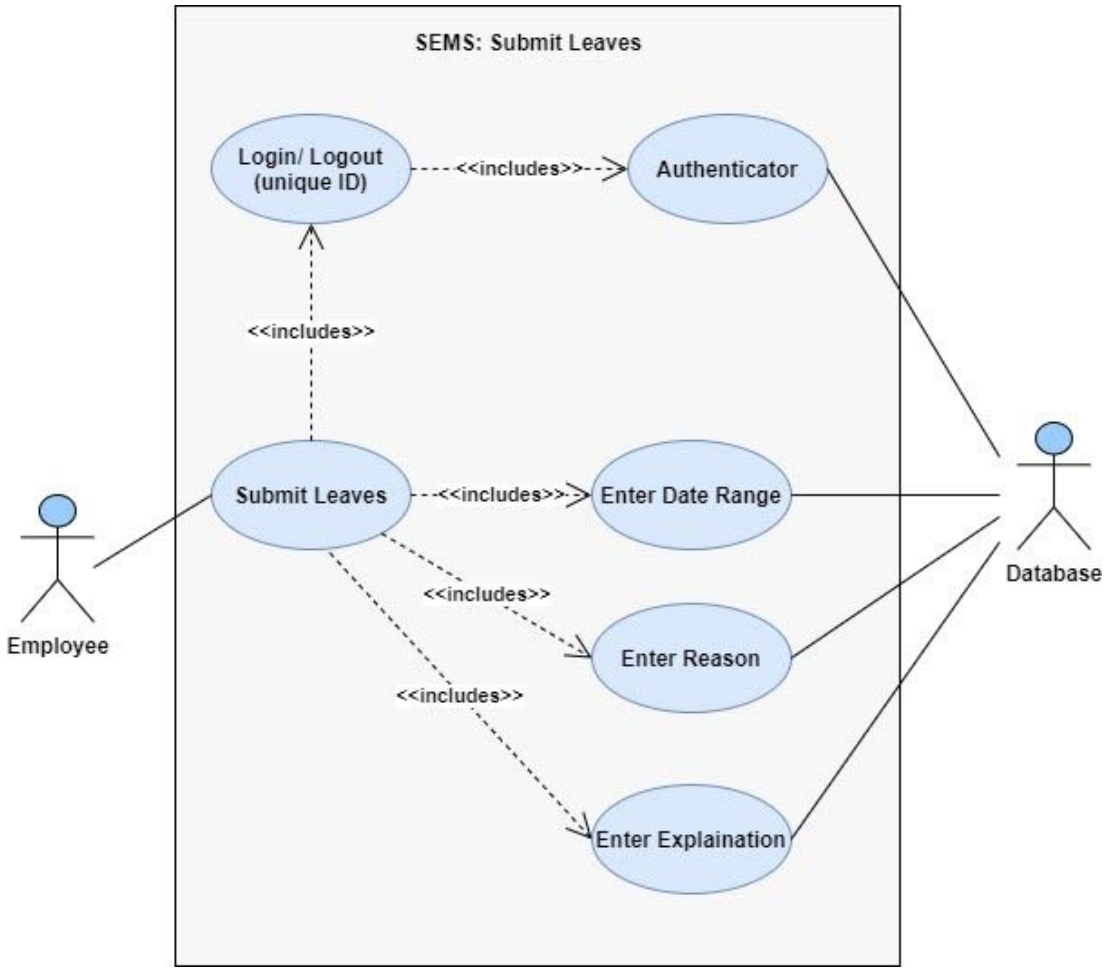

Figure – 6: Use Case - Submit Leaves

Use Case Name: Submit Leaves
System Name: SEMS
Level: User
Primary Actor: Employee
Stakeholders: Employee, System Database
Precondition:

The employee has logged into the system using a valid username and password.

Success Postcondition:

- A successfully submitted leave application.

Main Success Scenario:

1. The Employee selects “Submits Leave” from the main menu.
2. The Employee enters the data range.
3. The Employee enters the reason.
4. The Employee enters the explanation.

Subsystems interactions and data flow:

The following sequence diagram illustrates the interactions between different sub-systems and data flow while an HR employee approves the payroll of an employee. After providing accurate login credentials and reCAPTCHA, the user is verified and authenticated to log into the system. The Login Service interacts with the Authenticator Service to authenticate and the database to verify the user's login information. Then the user can access the Payroll management service. Before approving the payroll, the HR employee can access the Timesheet management service to verify the work logs of employees. Finally, the HR employee can approve the payroll and save it into the database.

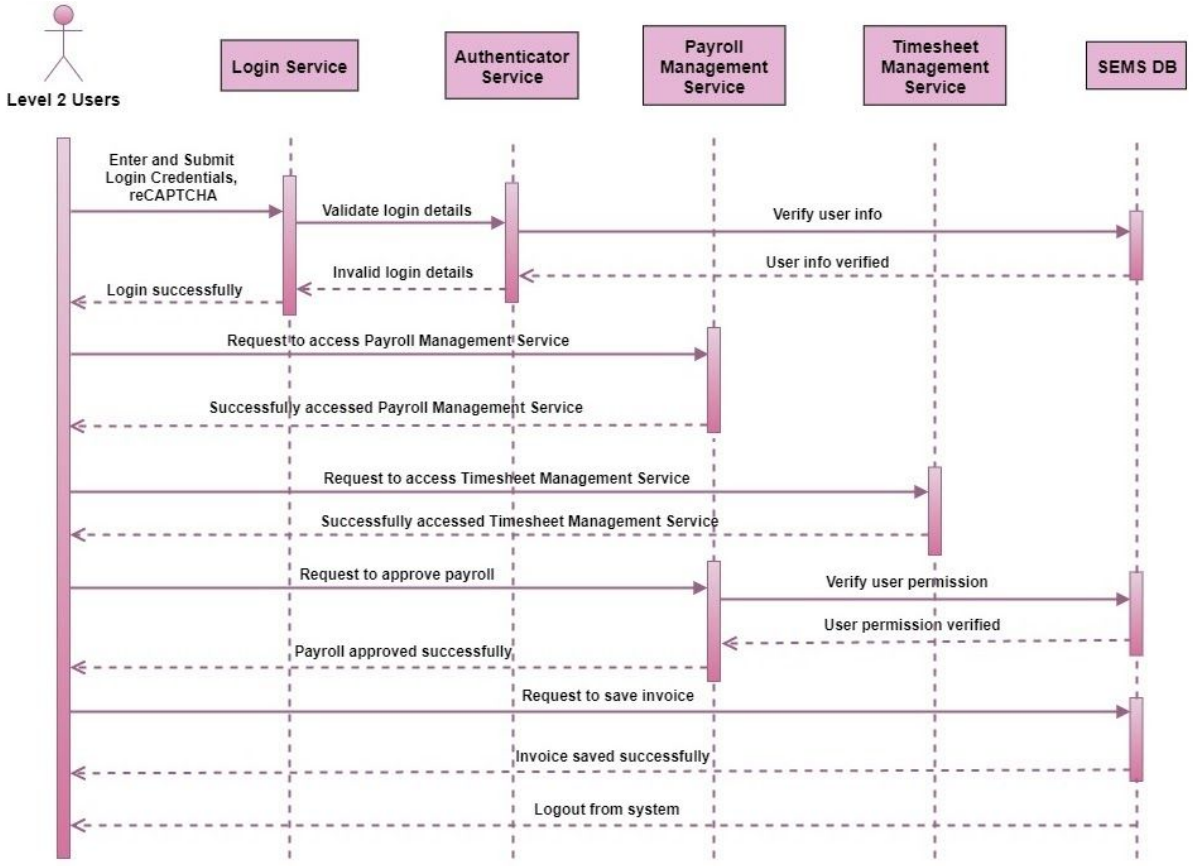

Figure – 7: Sequence Diagram showing the interaction between subsystems and data flow while a Level 2 User (HR Employee) is approving payroll of an employee.

Deployment strategies:

Our software has deployed in the Firebase cloud which provides the platform, Backend-as-a-Service (BaaS) for our web application. We used the Firebase Admin SDK to deploy our SEMS.war application to Firebase. The software uses the Firebase database which is a cloud-hosted NoSQL database embedded in the Firebase cloud. This NoSQL database provides real-time updates of data and ensures the performance of our system. The database server is backed up automatically which ensures the safety of user's data. The cloud server also provides the interface for the clients to use the software through web browsers using HTTP internet protocols that ensures the availability of the system over the web.

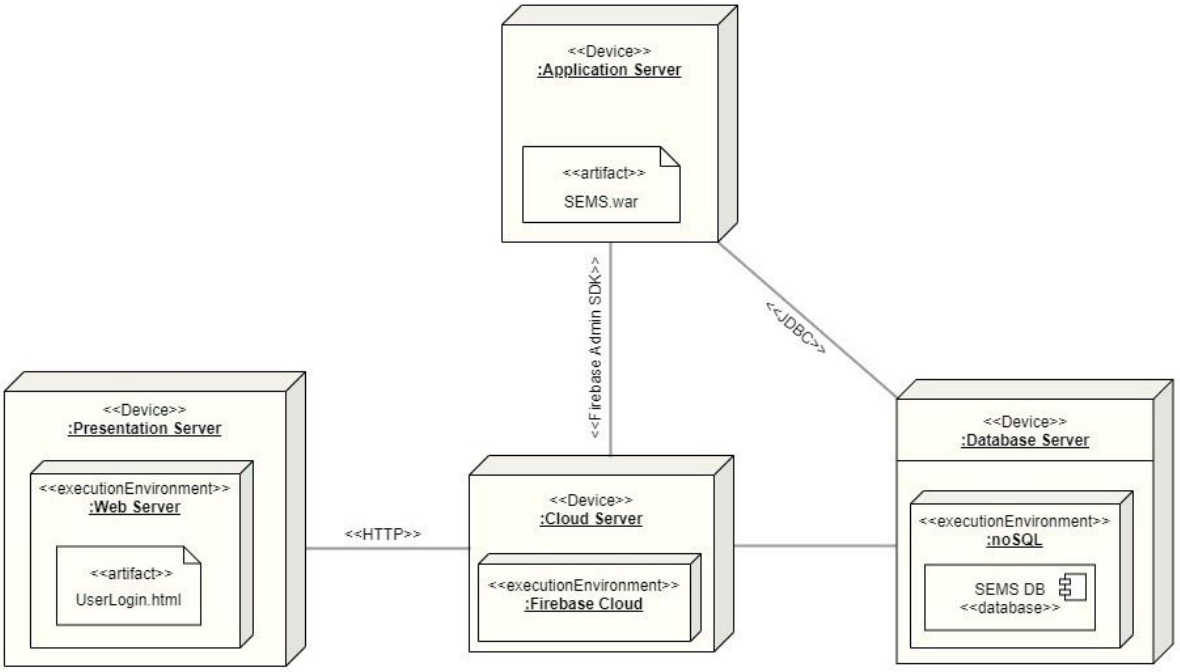

Figure – 8: Deployment Diagram showing different nodes of the system and their communication protocols.

Concrete Architecture VS Conceptual Architecture:

When developing the conceptual architecture of the SEMS system we mainly focused on the functionalities of the system, the different components of the system, what these components are responsible for and how they interact with each other to provide the functionalities of the system. However, this view of the system was accomplished through abstracting actual implementation details and as such is not a true depiction of how the system actually functions. In the concrete

architecture subsystems of the SEMS system were deployed, implementation details were realized, and the true nature of the system was seen. As such through the concrete architecture the real relationships between components and how the high-level architecture affects the organization of the different components of the system were seen.

Further, when comparing the two architectures we realized that our organization of how subsystems will communicate with each other was misconstrued and as such fixed our subsystems diagram to better reflect the true nature of how they work. In the concrete architecture, the realization was made that the way in which the high-level architecture was interpreted was also a little flawed as such the architecture style of using a 2-tier client-server was changed to a 3-tier client-server architecture to include a data access layer. This makes the codebase much easier to organize as it creates an extra layer of logical separation. This extra layer can increase the maintainability of the system as the functionalities of the system grow. The assumed relationships within the system were clarified to depict the true nature of their relationships as such the diagrams have been improved to depict such relationships. These relationships include how end-users interact with the system in terms of use case diagrams, and the diagrams that better help show the flow of data within the system.

Lessons Learned:

Throughout the development of concrete architecture, we better learned the importance of our conceptual architecture as it helped guide the direction of the concrete architecture. By abstracting certain details in the conceptual architecture we better understood what was required of the system then through developing the concrete architecture we learned to better understand how the architectural styles and patterns can influence how the actual system is implemented. Through implementation, we learned that the concrete architecture could cause violations in the conceptual architecture and that the concrete architecture can drift from the conceptual architecture as well. Ultimately the two work together to help better understand the overall architecture of a given system.

References:

1. https://firebase.google.com/terms/service-level-agreement/