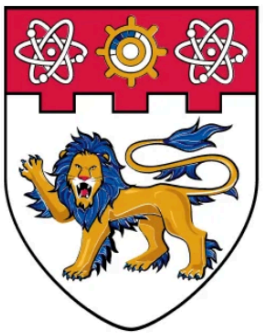


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

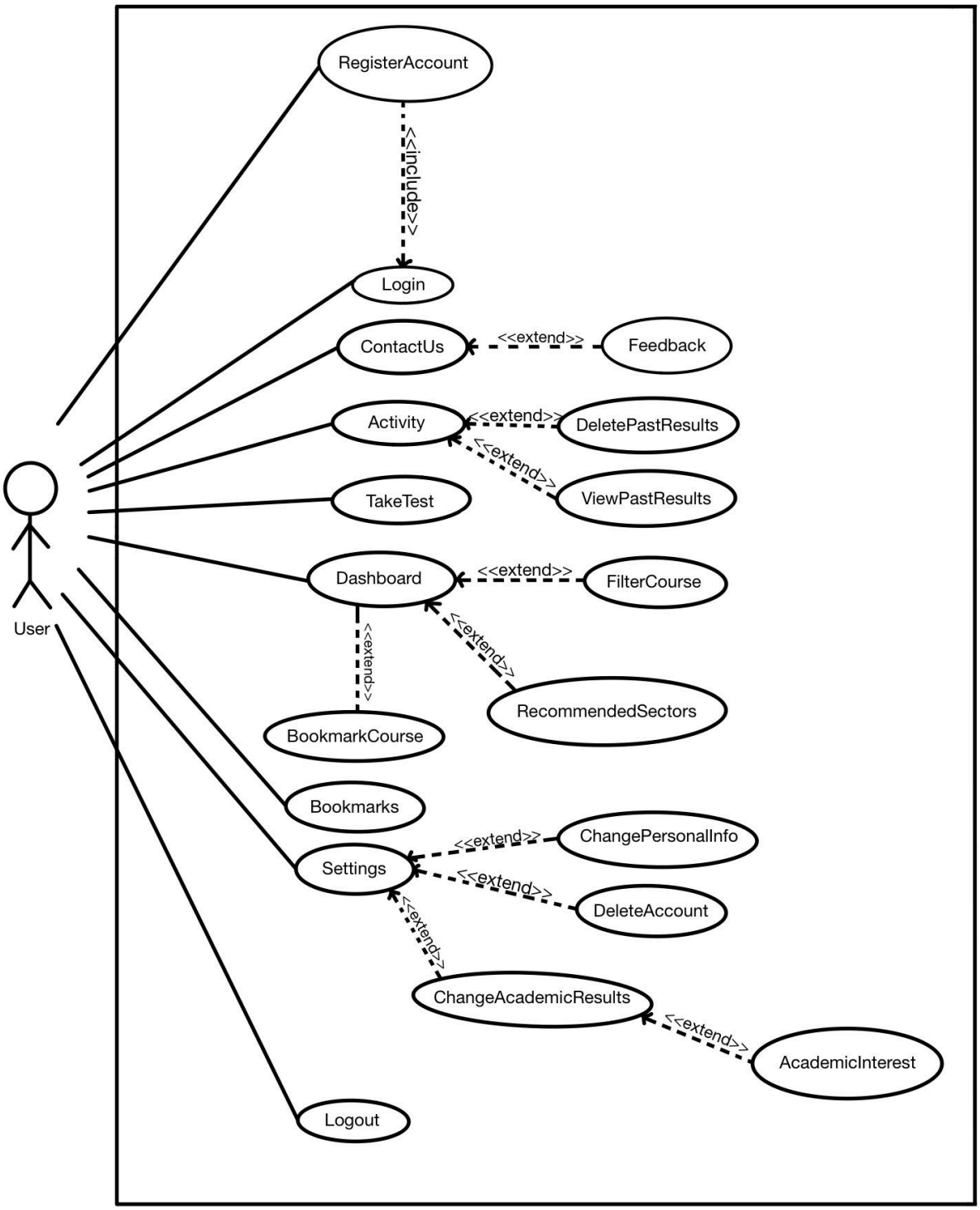
SC2006 - Software Engineering
Lab 3 Deliverables

Lab Group	TDDA
Team Name :	Project PolyQuest
Members	Lam Yan Yee U2323642L
	Dallas Ng Zhi Hao U2323084C
	Malcolm Fong Cheng Hong U2321081B
	Yeo, Ruizhi Carwyn U2322527C
	Khoo Ze Kang U2321266K
	Jerwin Lee Chu Hao U2321487B

Table of Contents

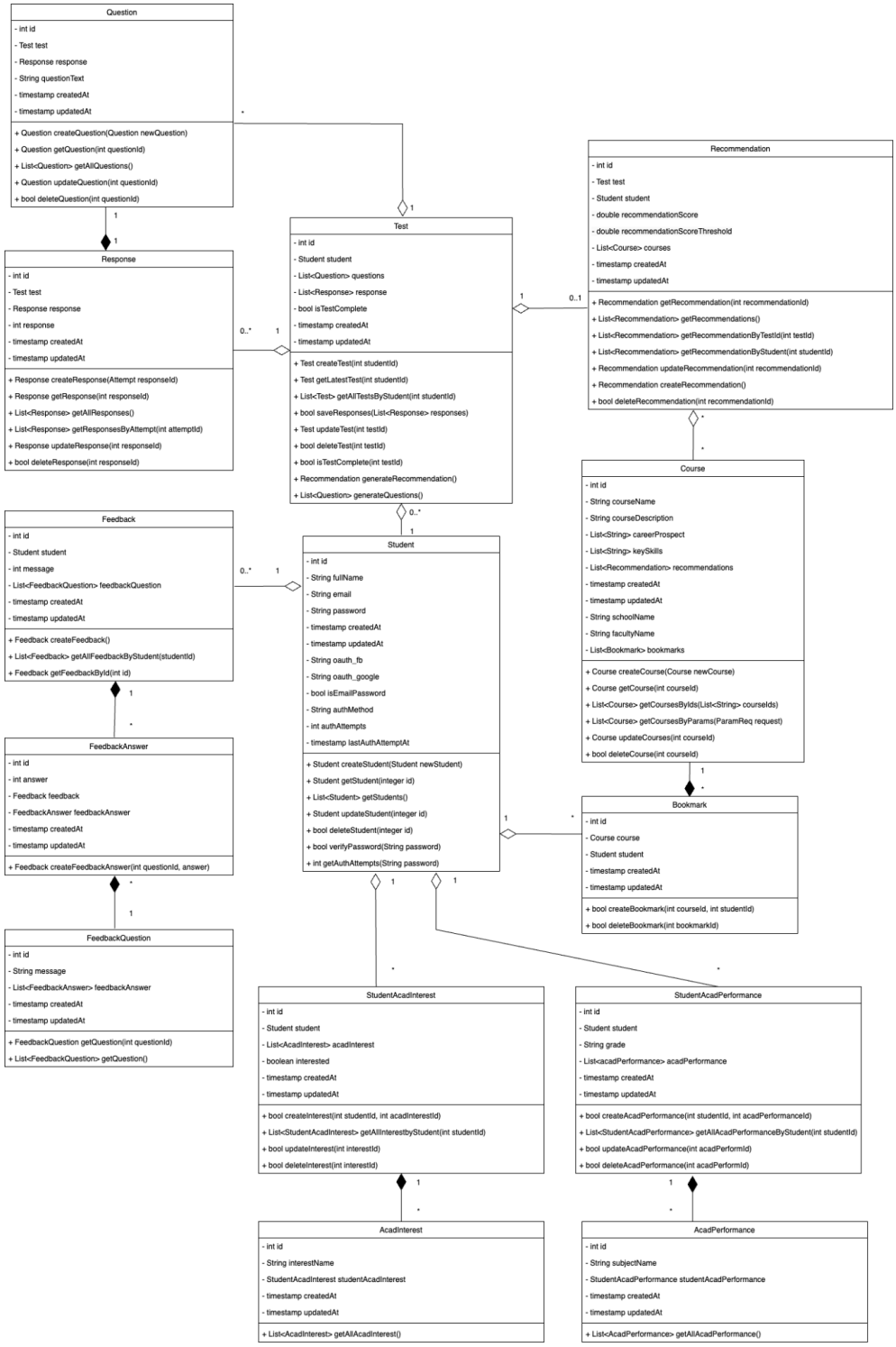
If the images are blurred, please refer to the raw PNG files uploaded with this document.

1. Complete Use Case Model



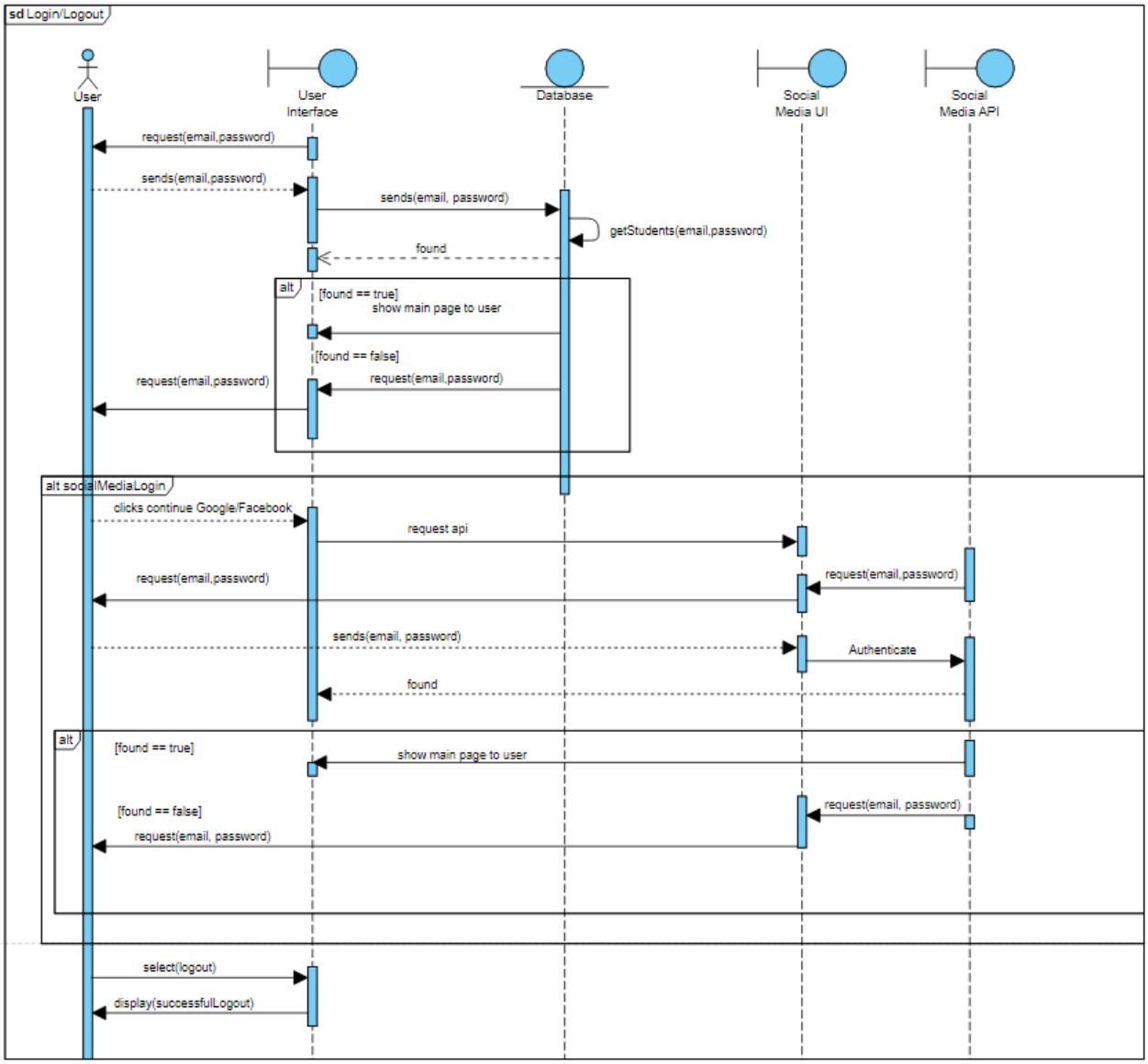
2. Design Model

A. Class Diagram

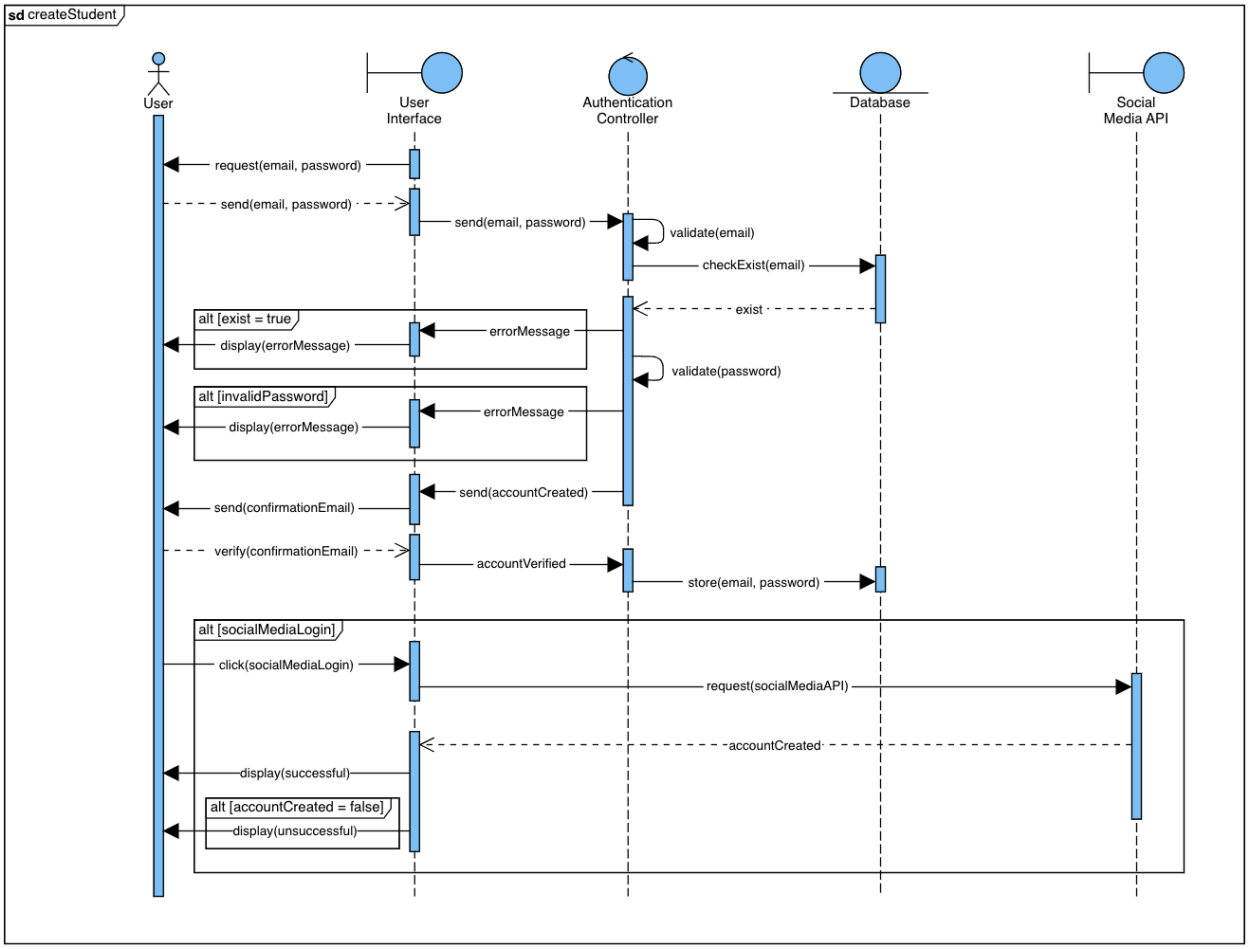


B. Sequence Diagrams

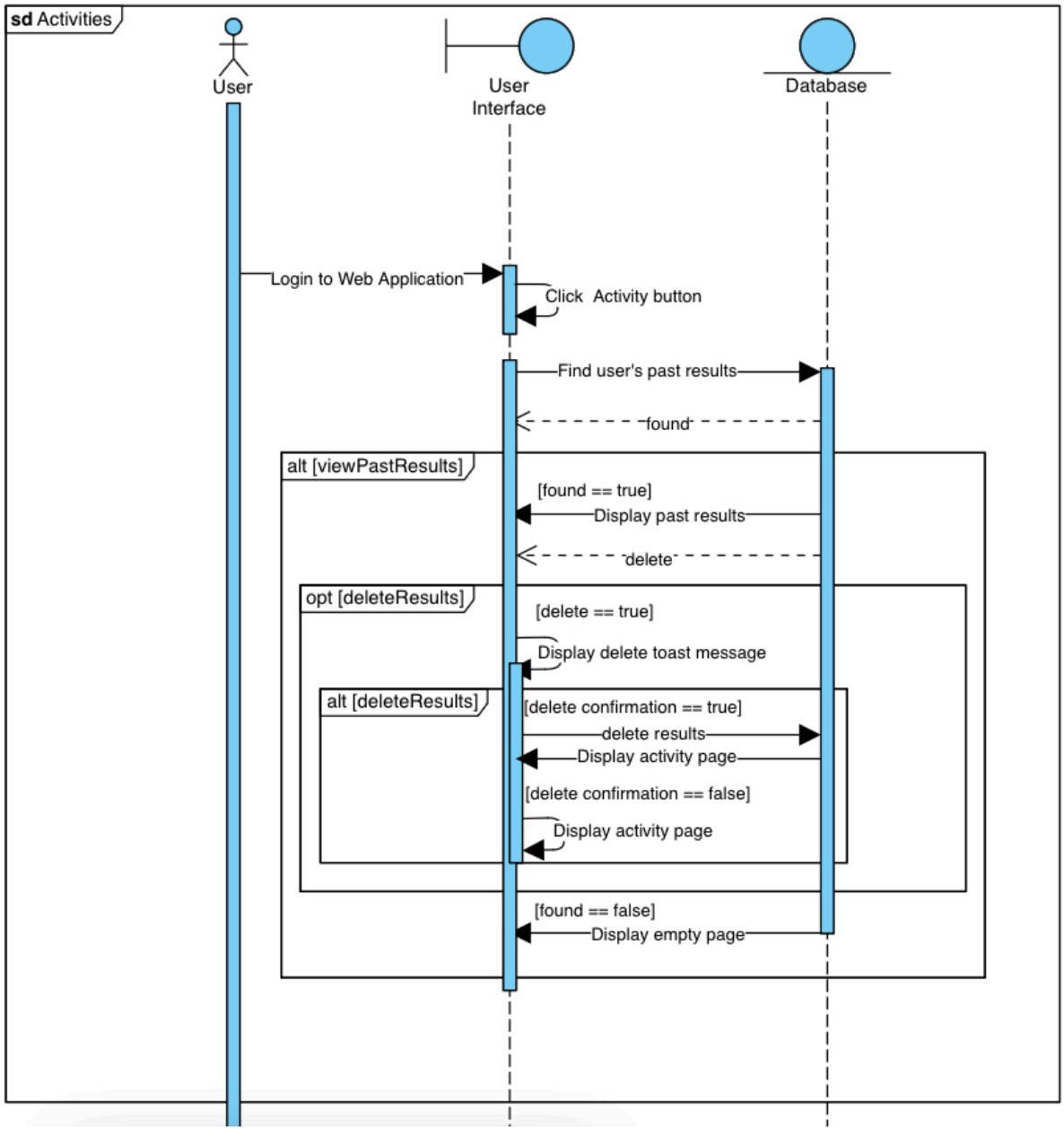
a. Login/Logout



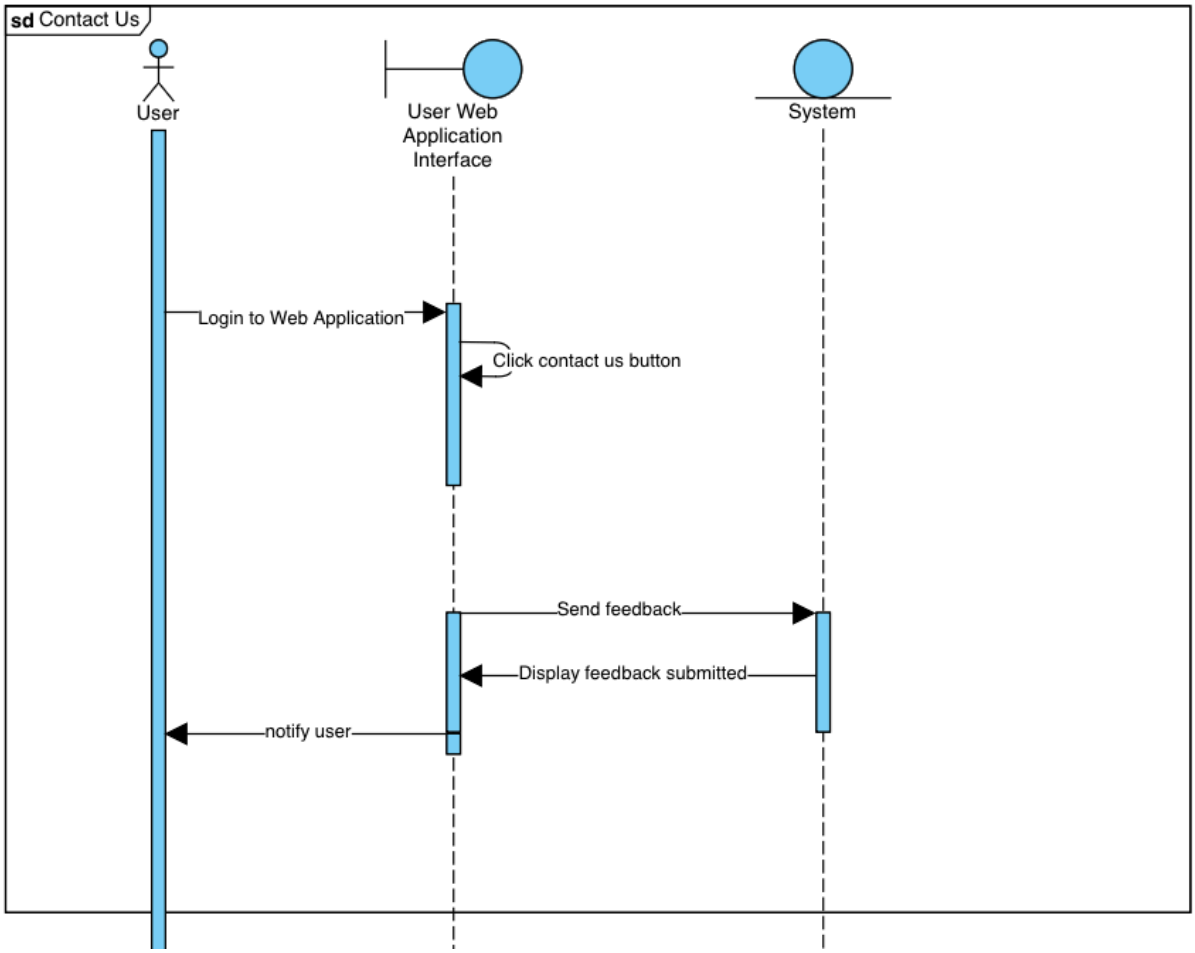
b. Account Registration



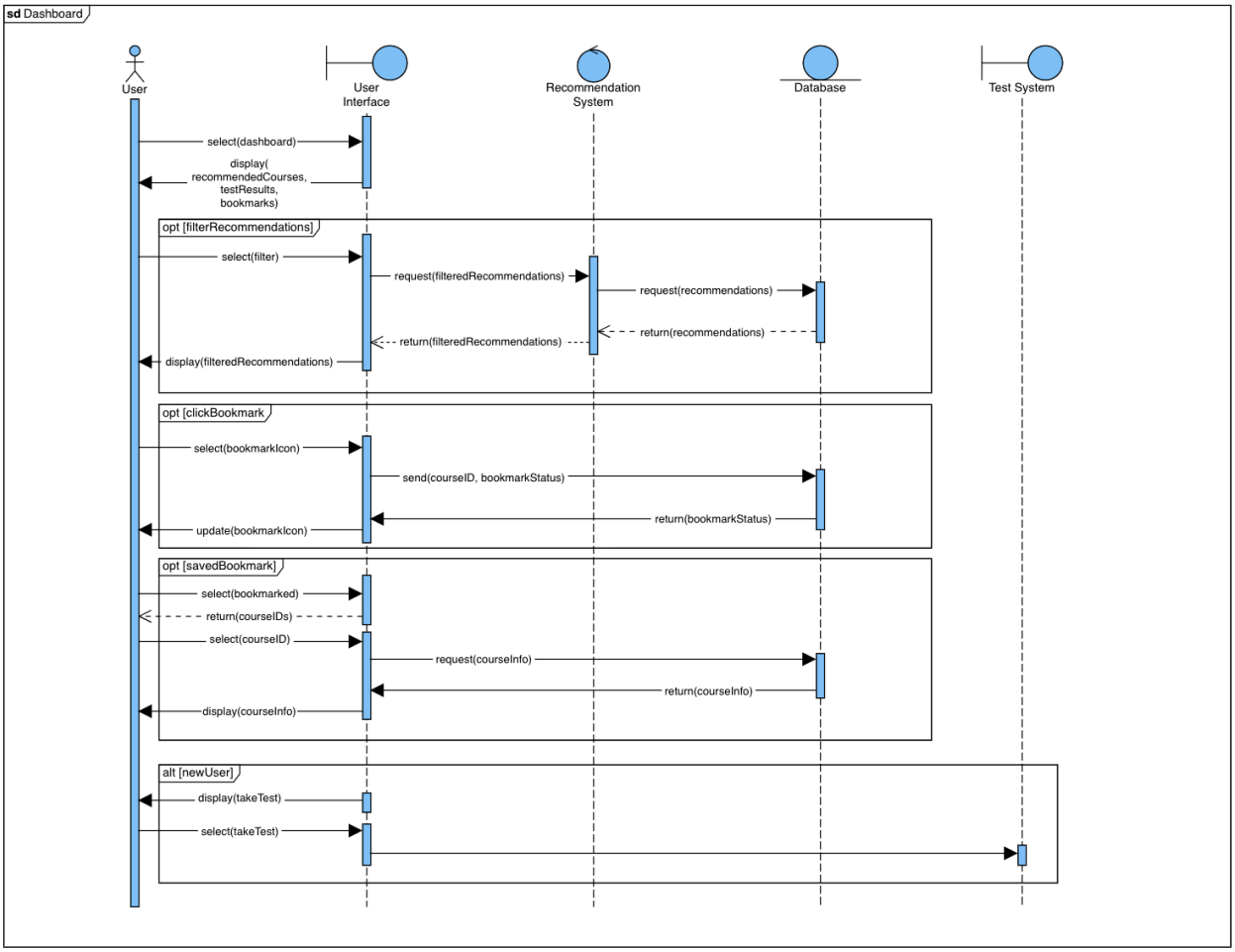
c. Activities



d. Contact Us

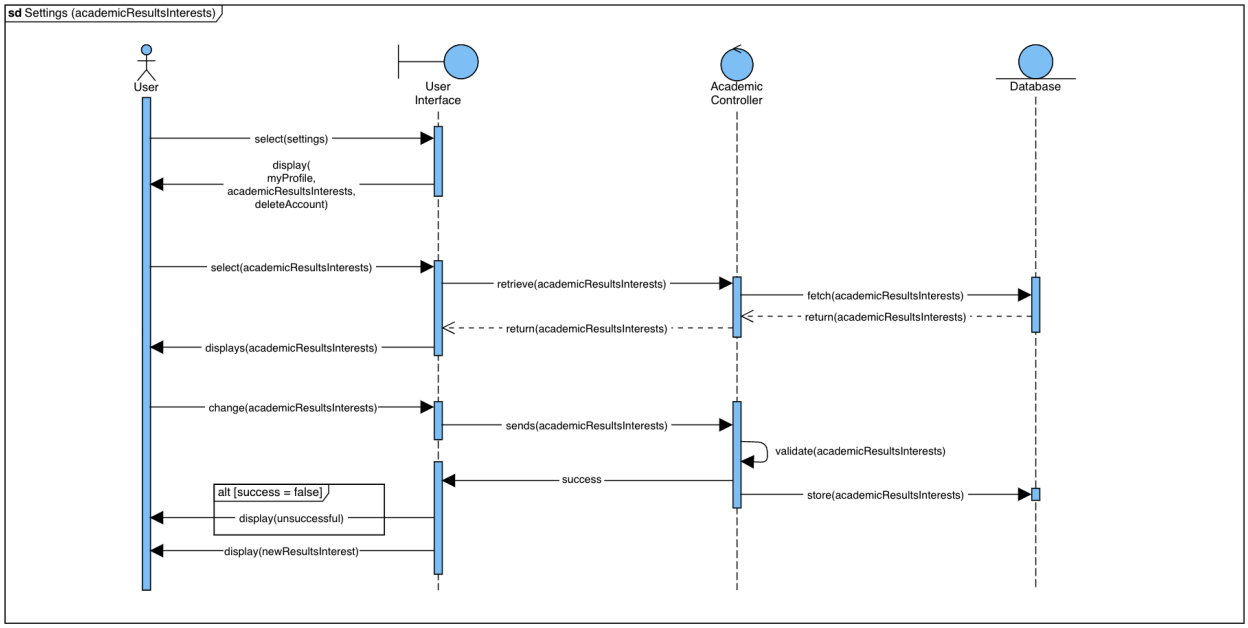

sd Contact Us
User
User Web Application Interface
System
Login to Web Application
Click contact us button
Send feedback
Display feedback submitted
notify user

e. Dashboard

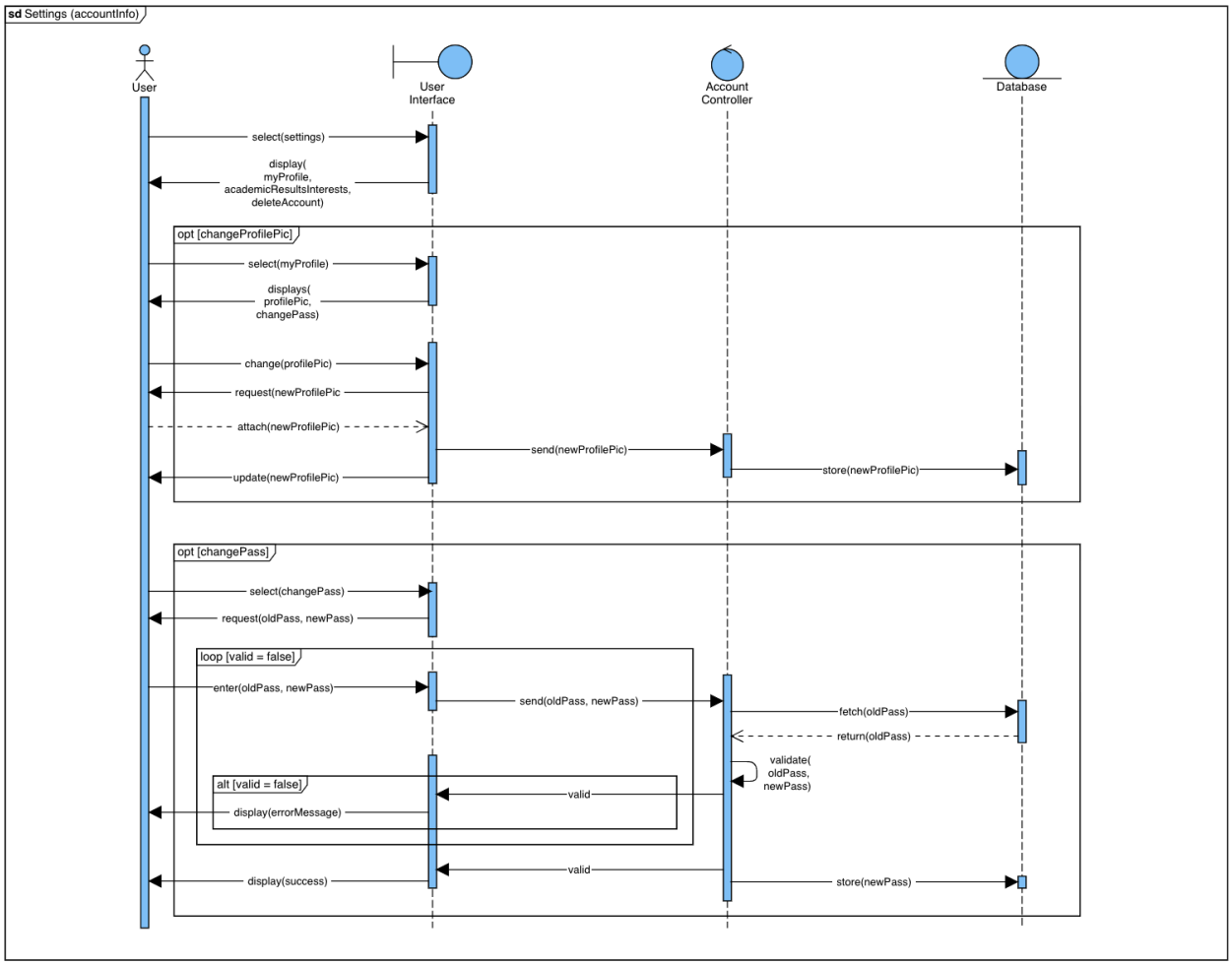

sd Dashboard
User
User Interface
Recommendation System
Database
Test System
select(dashboard)
display(recommendedCourses, testResults, bookmarks)
opt [filterRecommendations]
select(filter)
request(filteredRecommendations)
request(recommendations)
return(recommendations)
return(filteredRecommendations)
display(filteredRecommendations)
opt [clickBookmark
select(bookmarkIcon)
send(courseID, bookmarkStatus)
return(bookmarkStatus)
update(bookmarkIcon)
opt [savedBookmark]
select(bookmarked)
return(courseIDs)
select(courseID)
request(courseInfo)
return(courseInfo)
display(courseInfo)
alt [newUser]
display(takeTest)
select(takeTest)

f. Settings

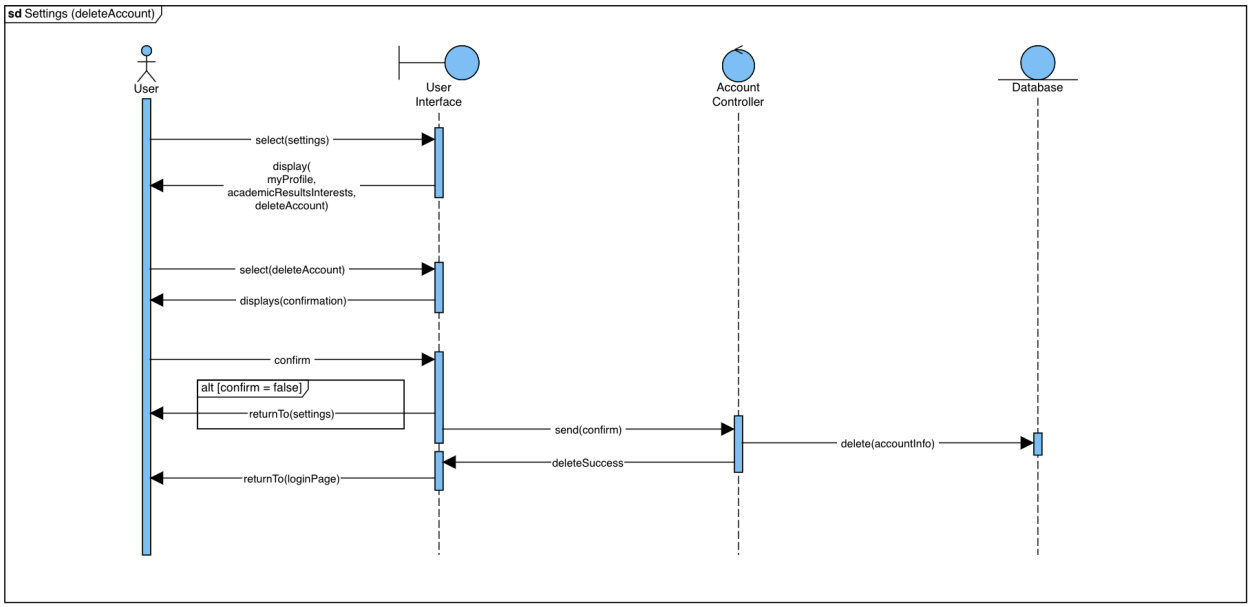
i. Academic Results & Interests



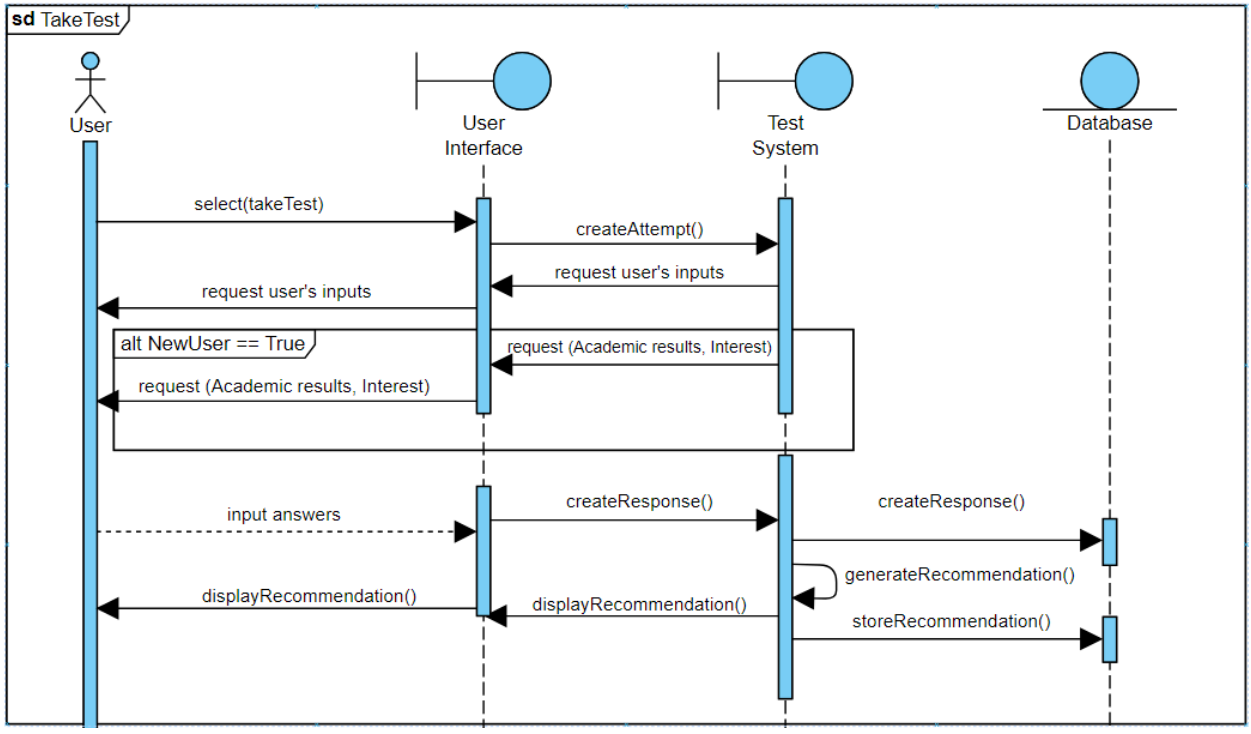
ii. Account Information



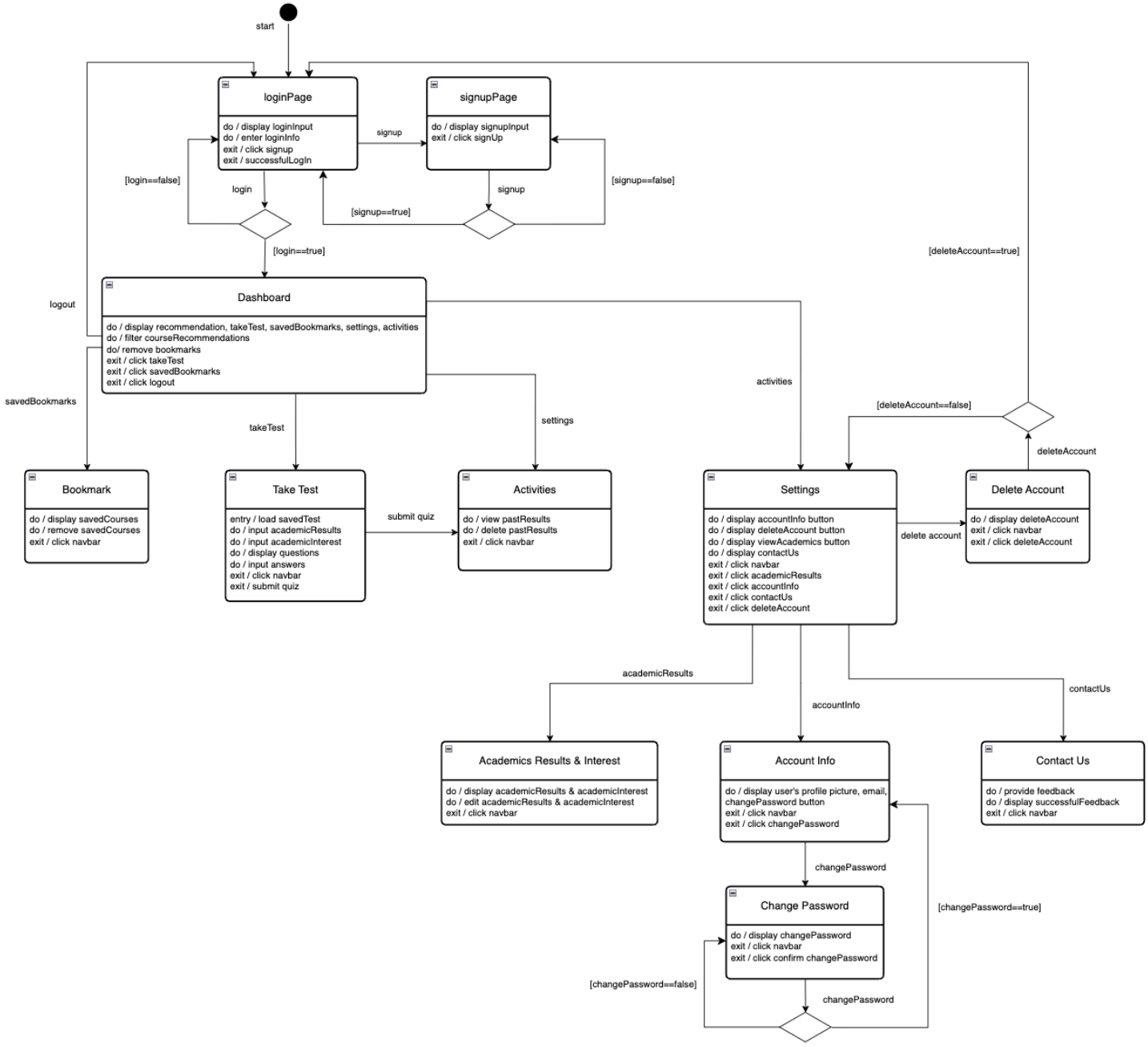
iii. Delete Account



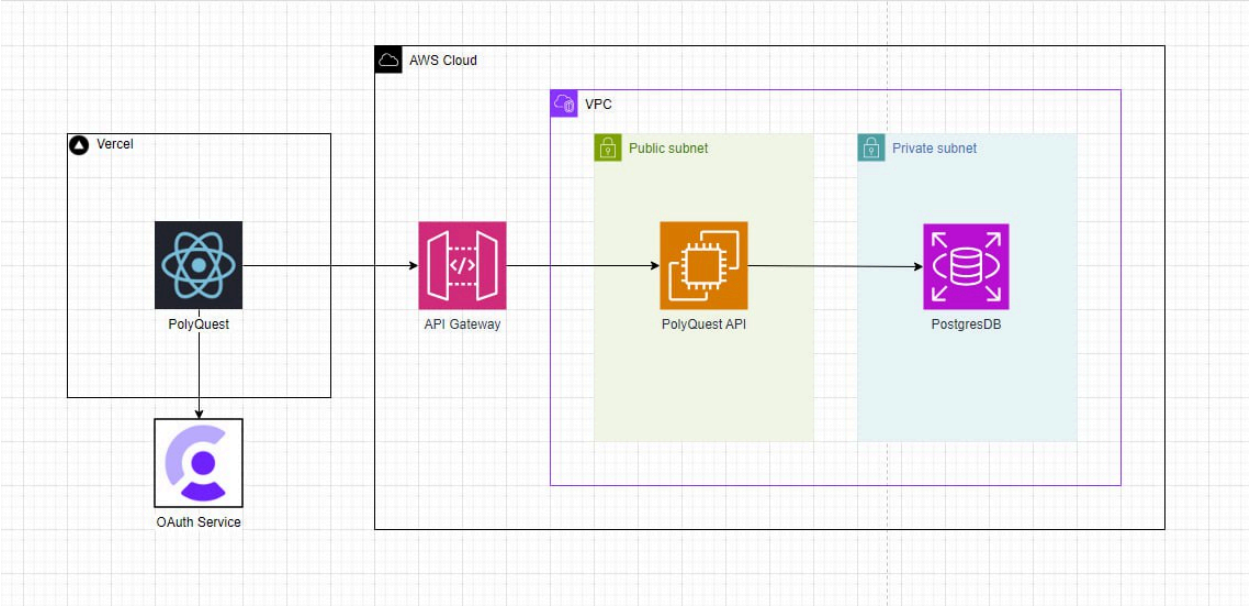
g. Take Test



C. Dialog Map Diagram



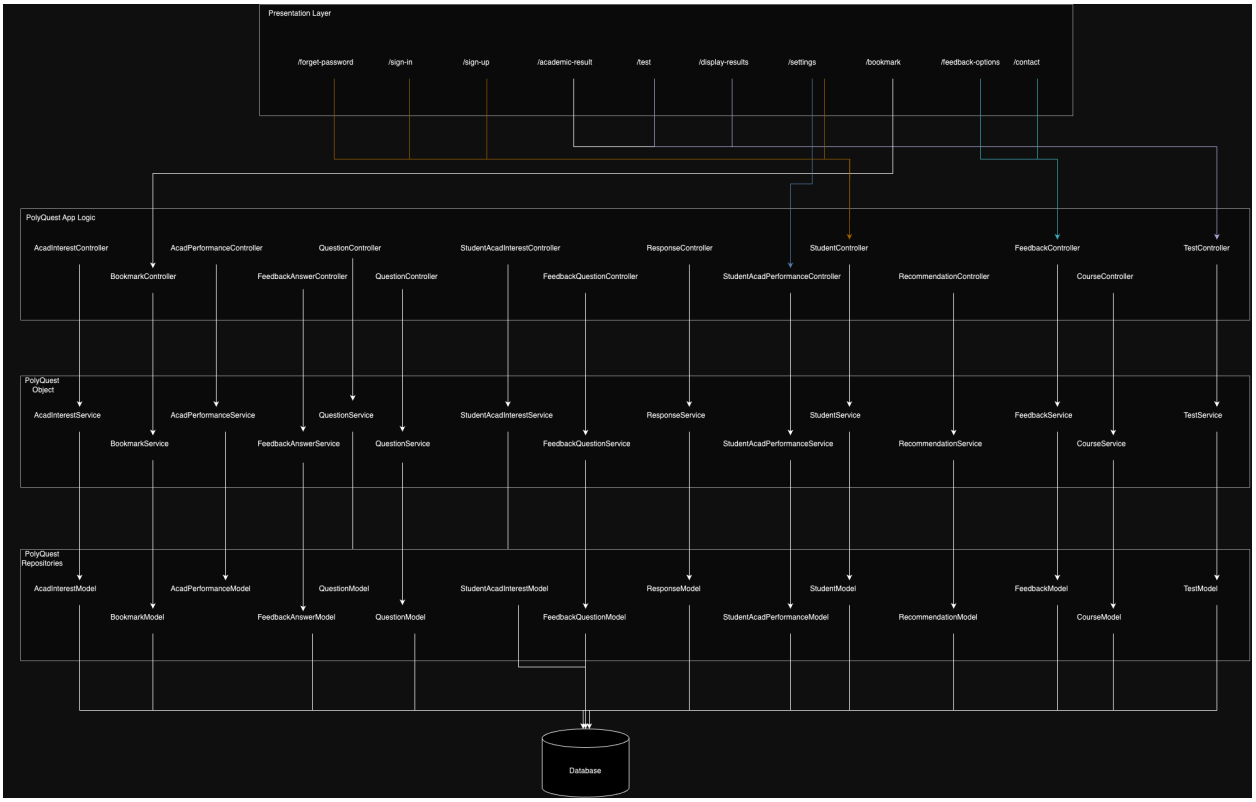
3. System Architecture



The system architecture diagram provided shows the deployment and operational structure of the PolyQuest application. Here's a detailed breakdown, with each component mapped to its respective layer and function in the architecture. The architecture follows a multi-layered approach with distinct layers for presentation, application logic, and data (or app object) layers.

Data Flow Across Layers

1. **Presentation Layer**: User inputs (grades, interests, quiz responses) are entered in the UI (React app).
2. **App Logic Layer**: API Gateway routes these inputs to the relevant controllers in the PolyQuest API, which processes and validates data.
3. **App Object Layer**: The PolyQuest API interacts with the data models to store or retrieve data from PostgreSQL.
4. **Response Generation**: After processing the data, the API generates a recommendation or feedback summary, which flows back through the API Gateway to the UI.
5. **Recommendation Display**: The UI displays recommended courses or feedback summaries based on user interaction and backend processing.



1. Presentation Layer

This is the user-facing layer of the PolyQuest application, where users interact with the frontend interface hosted on **Vercel**.

- **Frontend (PolyQuest UI on Vercel)**:
    - Built with **React**
    - This layer is responsible for rendering the user interface, handling user input, and sending HTTP requests to the backend.
    - **Authentication**: Utilizes an **OAuth Service** to handle user authentication and authorisation. The frontend uses OAuth for secure login, which allows the app to verify user identity without storing passwords directly.
- **Communication with API Gateway**:
    - The frontend communicates with the backend through **API Gateway**, which acts as a secure entry point for all HTTP requests going into the AWS cloud environment.
    - API Gateway routes requests from the frontend to the PolyQuest API services, ensuring a clear separation between the client and server layers.

UIs in PolyQuest App: The different UIs will then call for the respective controllers to run the App Logic.

Forget Password: Allows users to reset their password in case they forget it.

Sign In: Users log in to their accounts using email and password (or OAuth).

Sign Up: New users can create an account by providing the necessary details.

Academic Result: Users can view their academic results and performance metrics.

Test: Users can take assessments to evaluate their interests and academic suitability.

Display Results: After completing a test, users receive personalised results and recommendations.

Settings: Users can update their profile information and preferences.

Bookmark: Users can bookmark courses or resources for easy access later.

Feedback Options: Users can provide feedback on the application, courses, or tests.

Contact: Users can reach out for support or inquiries.

---

2. Application Logic Layer

This layer, running within AWS Cloud's VPC (Virtual Private Cloud), handles the core business logic of the application, orchestrating data flow and application processes. The components in this layer are primarily the **API Gateway** and **PolyQuest API**.

- **API Gateway**:
  - Acts as an intermediary that securely routes requests from the frontend to the backend services.
  - Helps manage and scale requests, provides caching, and ensures that requests are authenticated and authorised before reaching the PolyQuest API.
- **PolyQuest API**:
  - Deployed on an **EC2 instance** or similar compute service within a **public subnet**.
  - The PolyQuest API contains controllers that handle requests from the frontend, process them according to business logic, and interact with the models in the data layer.

Controllers in PolyQuest API: Each controller represents a domain entity and is responsible for handling operations related to that entity.

- **AcadinterestController**: Handles requests and actions related to academic interests.
- **AcadperformanceController**: Manages academic performance data for each student.
- **BookmarkController**: Manages bookmarks that users create for specific courses.
- **CourseController**: Handles requests related to course information and recommendations.
- **FeedbackController**: Manages feedback from students and links to other feedback-related entities.
- **FeedbackAnswerController**: Handles student responses to feedback questions.
- **FeedbackQuestionController**: Manages questions that are used in feedback forms.
- **QuestionController**: Handles actions related to the question pool for tests.
- **RecommendationController**: Interfaces with the Machine Learning model to generate and manage recommendations based on student responses and performance.
- **ResponseController**: Manages student responses to questions in tests.
- **StudentController**: Handles user information and profile actions.
- **StudentAcadInterestController**: Manages the relationship between students and their academic interests.
- **StudentAcadPerformanceController**: Manages academic performance records specific to each student.
- **TestController**: Manages the test-taking process and its components, including associated questions and responses.

Machine Learning Model for Recommendations:

- A Machine Learning model, integrated within this layer, generates personalised course recommendations for students based on test results and other student data. This model leverages data from entities such as `StudentAcadInterest`, `StudentAcadPerformance`, `Response`, `Test`, and `Question` to compute recommendations.
- The **RecommendationController** interacts with this model, passing test results and student information to generate and retrieve recommendations.

---

3. App Object (Data) Layer

The app object layer consists of the database and data models, which define the structure of the data stored and used by the application. This layer resides in the **private subnet** of the VPC for security.

- **Database (PostgresDB)**:
    - A PostgreSQL database instance that stores all data related to the PolyQuest application.
    - Accessible only by the PolyQuest API within the VPC for security, ensuring that the database is isolated from direct internet access.

Models in the Database: Each entity represents a table in the PostgresDB and is mapped to a specific data model in the application. Here's how the models are structured:

- **Acadinterest**: Stores academic interests that students can choose.
- **Acadperformance**: Tracks overall academic performance metrics.
- **Bookmark**: Contains information on courses that students have bookmarked.
- **Course**: Stores details of available polytechnic courses.
- **Feedback**: Represents a feedback session or form that students complete.
- **FeedbackAnswer**: Stores answers given by students in feedback forms.
- **FeedbackQuestion**: Stores questions that appear in feedback forms.
- **Question**: Represents individual questions used in tests.
- **Recommendation**: Stores generated course recommendations based on student data.
- **Response**: Tracks individual responses to questions in tests.
- **Student**: Represents user-profiles and basic student information.
- **StudentAcadInterest**: Maps students to their selected academic interests.
- **StudentAcadPerformance**: Represents detailed academic performance records for students.
- **Test**: Represents tests taken by students, containing associated questions and responses.

Linking the Layers

- **Frontend (React) ➔ API Gateway**: The presentation layer (frontend) makes HTTP requests to the API Gateway, which routes requests to appropriate endpoints within the PolyQuest API.
- **API Gateway ➔ PolyQuest API**: API Gateway forwards these requests to the controllers in the PolyQuest API. Each controller processes requests, applies business logic, and interacts with the appropriate model.
- **PolyQuest API ➔ Database (PostgresDB)**: The controllers in the PolyQuest API interact with the data models, which represent tables in the PostgresDB. This interaction involves creating, reading, updating, and deleting records in response to user requests.

Each model is linked to a specific controller that manages Create, Read, Update and Delete (CRUD) operations for that entity. For example:

- **StudentController** interacts with the **Student** model to manage user data.

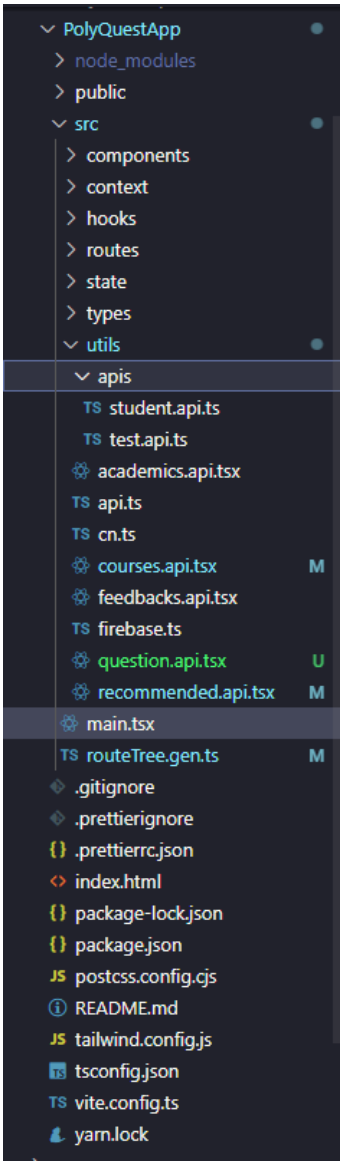
- **TestController** interacts with the **Test** and **Question** models to manage the test and question data.
- **RecommendationController** uses **StudentAcadPerformance**, **StudentAcadInterest**, and other models to generate personalised recommendations for students.

This layered architecture ensures:

1. **Separation of Concerns**: Each layer is responsible for specific aspects of the application, improving modularity and maintainability.
2. **Security**: By placing the database in a private subnet and securing access through the API Gateway, unauthorised access to sensitive data is prevented.
3. **Scalability**: The API Gateway and the layered design allow for scaling different components of the application independently, which is essential for handling increased traffic and load.

4. Application Skeleton

a. Frontend



The frontend project structure, built with React, is organised to promote modularity and maintainability. Here's an overview based on the provided structure:

components: This folder houses reusable UI components used across different application parts, making it easier to maintain a consistent look and feel.

context: Contains context providers for managing and sharing global states between components using React Context API. This allows for centralising state management for specific features.

hooks: Custom hooks are stored here to encapsulate reusable logic, enhancing the modularity of the codebase.

routes: This folder manages route definitions for the application, defining paths and page components.

state: Likely used for managing global or local state data, possibly with libraries like Zustand, to ensure predictable and organised state flow.

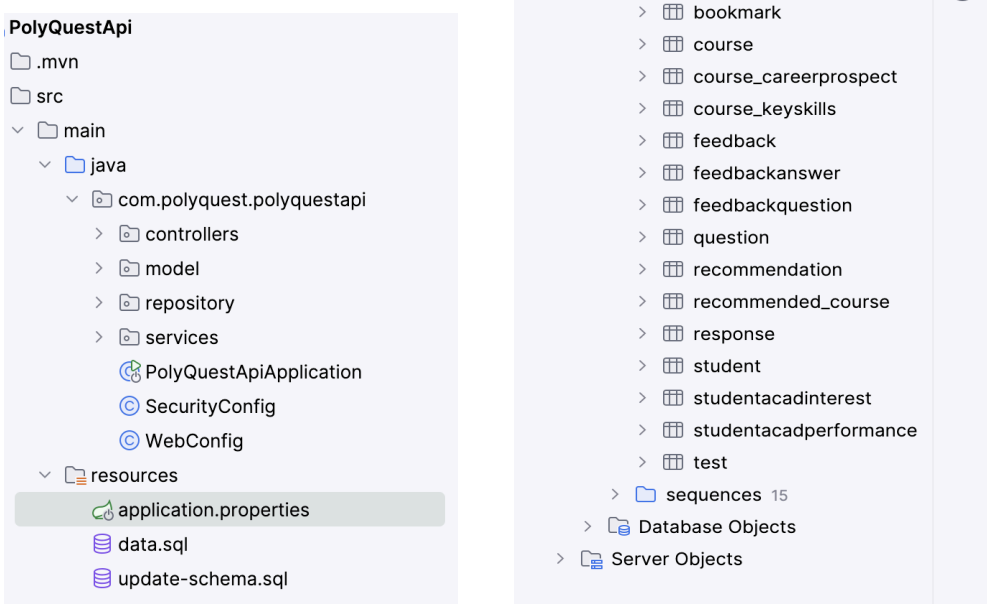
types: Holds TypeScript type definitions, ensuring consistency in data structures and interfaces across the application.

utils: A utility folder containing helper functions and configurations, such as API handlers and general-purpose functions, to simplify tasks across the app.

Inside **utils**, there's an **apis** subfolder where API modules for different parts of the application (e.g., courses.api.tsx, student.api.ts, test.api.ts) are organised. Each file in this folder defines specific API requests for a particular resource, making data fetching and manipulation modular and easy to manage.
main.tsx: The entry point of the application, where the main application component is rendered and the core setup (e.g., providers, routing) is initialised.

This organised structure facilitates collaboration by providing a clear separation between UI components, state management, and utility functions. Each module is isolated and categorised based on functionality, making the codebase easier to understand and extend.

b. Backend



The backend for PolyQuest is built with **Java Spring Boot**, following a structured design to separate concerns and streamline development. Here's a breakdown of each key component:

- **Controllers**: This layer defines REST API endpoints that handle HTTP requests (e.g., POST, GET, PATCH). Each endpoint has a unique URL path, specifying the route for client interactions. Controllers translate incoming requests from the frontend into service calls and return responses, ensuring communication between the frontend and backend.
- **Models**: These represent the core business entities of the application, defining the structure and properties of data objects, such as `Student`, `Course`, and `Test`. Models map directly to database tables, providing a clear representation of data entities within the system.
- **Repositories**: The repository layer handles data persistence, syncing business entities with the database. Spring Data JPA manages these interactions, providing CRUD operations and complex queries to retrieve and modify data without extensive SQL.
- **Services**: Services encapsulate the business logic, serving as an intermediary between controllers and repositories. They offer methods to access and modify business entities,

allowing for reusable, organized, and testable logic across various parts of the application.
- **application.properties**: This configuration file initializes and updates settings for the application, including database connection parameters and Swagger.io integration for API documentation. It simplifies adjustments to environment-specific properties, such as database URLs and credentials.
- **Database**: The relational database (e.g., PostgreSQL) stores the data in structured tables, reflecting each model entity. This data layer is essential for persisting user profiles, test results, recommendations, and other vital records, ensuring data consistency and integrity.

This layered approach in Spring Boot enables efficient data flow, modularity, and maintainability, ensuring that each component serves a distinct purpose in the overall architecture.

5. Appendix

a. Tech Stack

- Frontend
    - React.js
    - TypeScript
    - TailwindCSS
    - Mantine
- Backend
    - Java SpringBoot
- Database
    - PostgreSQL