Project PolyQuest

SC2006

TDDA Group 53

PROBLEM

Young Secondary Students who have no idea what the future holds for them

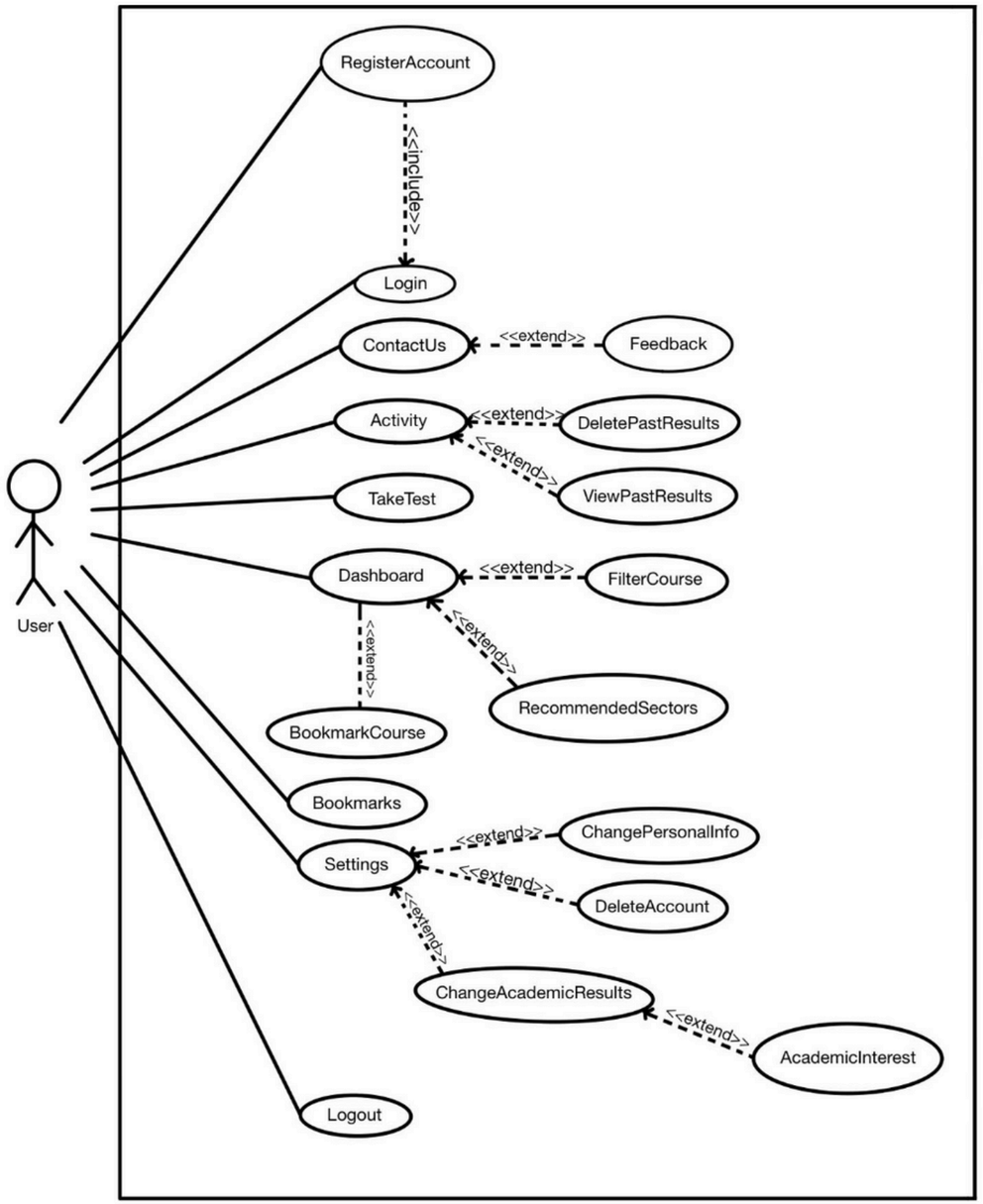
Solution - PolyQuest

A web app guiding students in polytechnic course selection based on academic performance, interests, and personality traits.

Key Features

- Authentication
- Interest Quiz
- Dashboard
- Profile / Settings
- Contact Us

USE CASE DIAGRAM



Feature List

Authentication

- Login / Logout
- Register Account
  - Google
  - Manual

Profile / Settings

- Change Personal Info
  - Password (Only for Manual)
- Delete Account
- Academic Grades and Interest
  - Save (First time use)
  - Change (subsequently)

Interest Test

- It gives you recommended courses based on your response

Contact Us

- Feedback
- Contact Details (Email)

Dashboard

- Search Courses
  - All available courses across all the Polytechnics
  - Able to search for different courses (School name, Course Name)
- Recommended Courses
  - Will be updated after a test has been taken
- Bookmark
  - Students can favorite a course that attracts them, to view them easily later on

Tech Stack

FrontEnd

- React.js
- TypeScript
- TailwindCSS
- Mantine

BackEnd

- Java SpringBoot
- Database: PostgreSQL

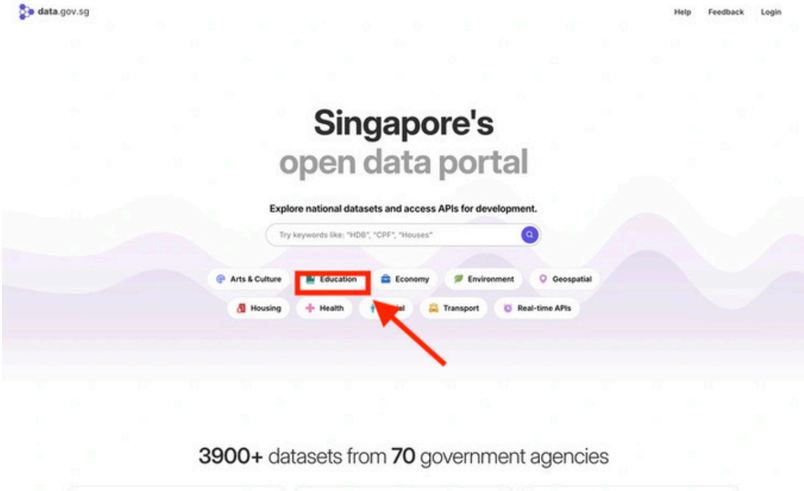
External APIs and Datasets

External APIs

Google OAuth 2.0 API

Datasets

data.gov.sg



Software Engineering Practices & Design Patterns

Good SWE Practices

1. Documentation
2. Good Code practices
3. Reusability and Refactoring
4. SCRUM

Documentation

README

Table of Contents

App Design

Overview

The PolyQuest system is structured to deliver a user-centric experience, leveraging both backend and frontend to support students' educational planning.

Frontend

The frontend is built with React and is organized by user role (Student, Teacher, Admin) to streamline user navigation. Components, routes, and utilities are housed in `/src/` folders:

- **/routes**: Contains role-specific routes.
- **/components**: Reusable components.
- **/utils**: Helper functions to enhance frontend code readability.

The entry point is `/src/routes/_root.tsx`.

Backend

The backend structure, built with Java SpringBoot, is designed for modularity and scalability:

- **Controller**: Manages incoming HTTP requests, forwarding them to appropriate service methods and returning the responses to clients.
- **Service**: Contains the core business logic, interacting with repositories to process data and execute application-specific functions.
- **Repository**: Acts as a data access layer, handling database operations like CRUD (Create, Read, Update, Delete) for entities, abstracted via JPA (Java Persistence API).
- **Model**: Defines the structure and attributes of database entities, representing tables and their relationships in Java objects.
- **DTO**: Defines the data transfer objects used for some APIs.

Documentation

Code Comments

```java
package com.polyquest.polyquestapi.repository;

import ...

@Repository  6 usages
public interface CourseRepository  extends JpaRepository<Course, Integer> {
    // Fetch courses by their IDs
    List<Course> findByIdIn(List<Integer> ids);  1 usage

    // Fetch courses by school name and/or course name
    List<Course> findBySchoolNameAndCourseName(String schoolName, String courseName);  1 usage

    // Fetch courses by only school name
    List<Course> findBySchoolName(String schoolName);  1 usage

    // Fetch courses by only course name
    List<Course> findByCourseName(String courseName);  1 usage

}
```

Student.java

```java
/**
 * The Student class represents a student entity in the system with attributes
 * like name, email, authentication details, and associated lists for tests,
 * recommendations, bookmarks, interests, performances, and feedbacks.
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity

// Ignore specific properties during serialization to avoid LazyInitializationException issues with Hibernate
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler", "tests", "recommendations", "bookmarks", "interes
public class Student {

    // Primary key, auto-generated for each student record
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    // Student's name
    private String name;

    // Student's email address
    private String email;

    // Encrypted password for authentication
    private String password;

    // Flag indicating whether the user logs in with email and password
    private Boolean isEmailPassword;

    // Timestamp marking the creation time of the student record
```

Transfer Knowledge

SwaggerUI API Docs

Swagger
Supported by SMARTBEAR

/v3/api-docs

Explore

OpenAPI definition v0 OAS 3.0

/v3/api-docs

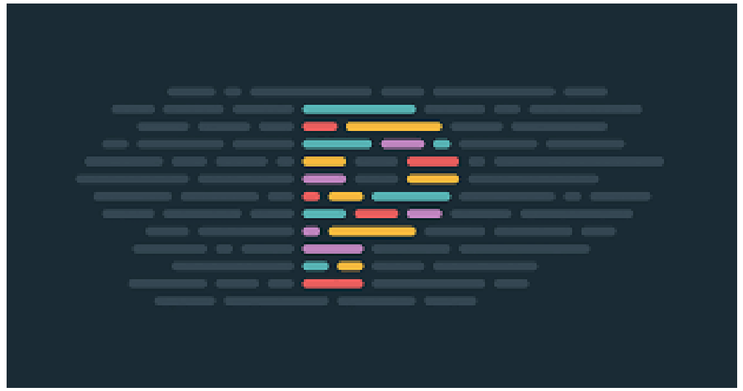
Servers

http://localhost:8080 - Generated server url

test-controller

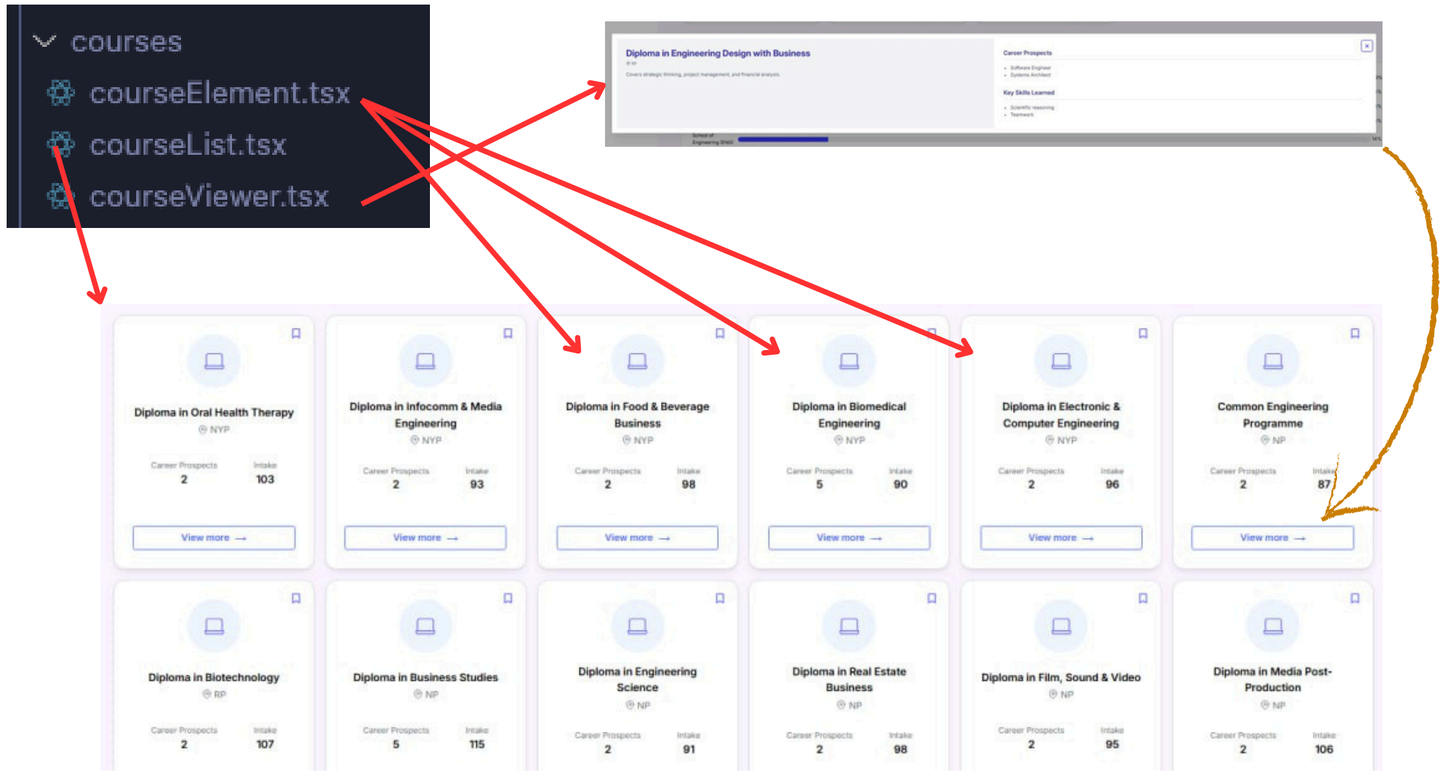
PUT	/test/{testId}
DELETE	/test/{testId}
PUT	/test/{testId}/complete
POST	/test/{testId}/responses
GET	/test/student/{studentId}
POST	/test/student/{studentId}
GET	/test/{testId}/recommendation
GET	/test/{testId}/isComplete
GET	/test/student/{studentId}/latest

Consistent Code Style and Naming Convention



Reusability & Refactoring

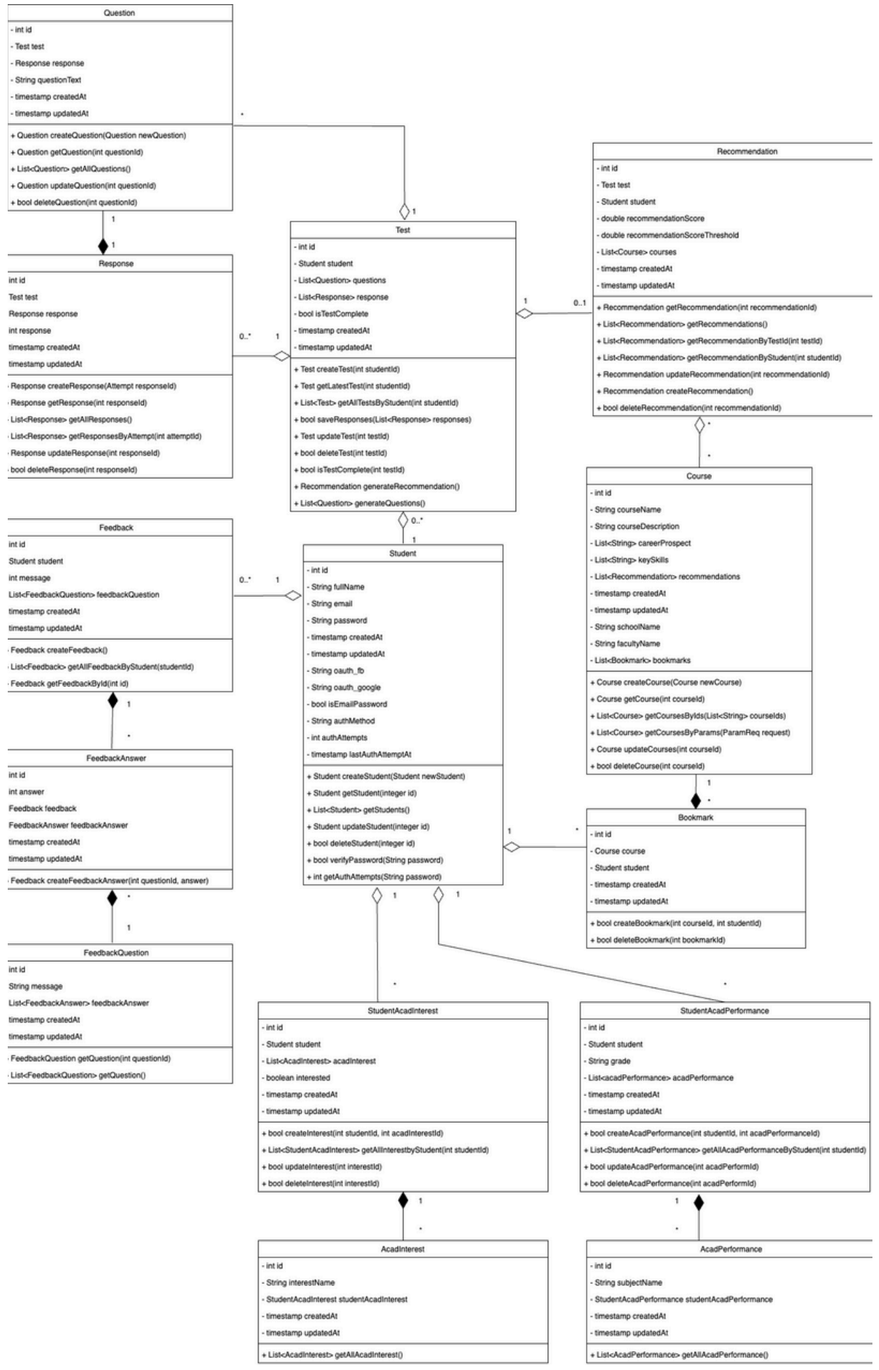
- Remove Duplication
- Remove Redundancy
- Consistent Naming Convention
- Improve Reusability
- Composability



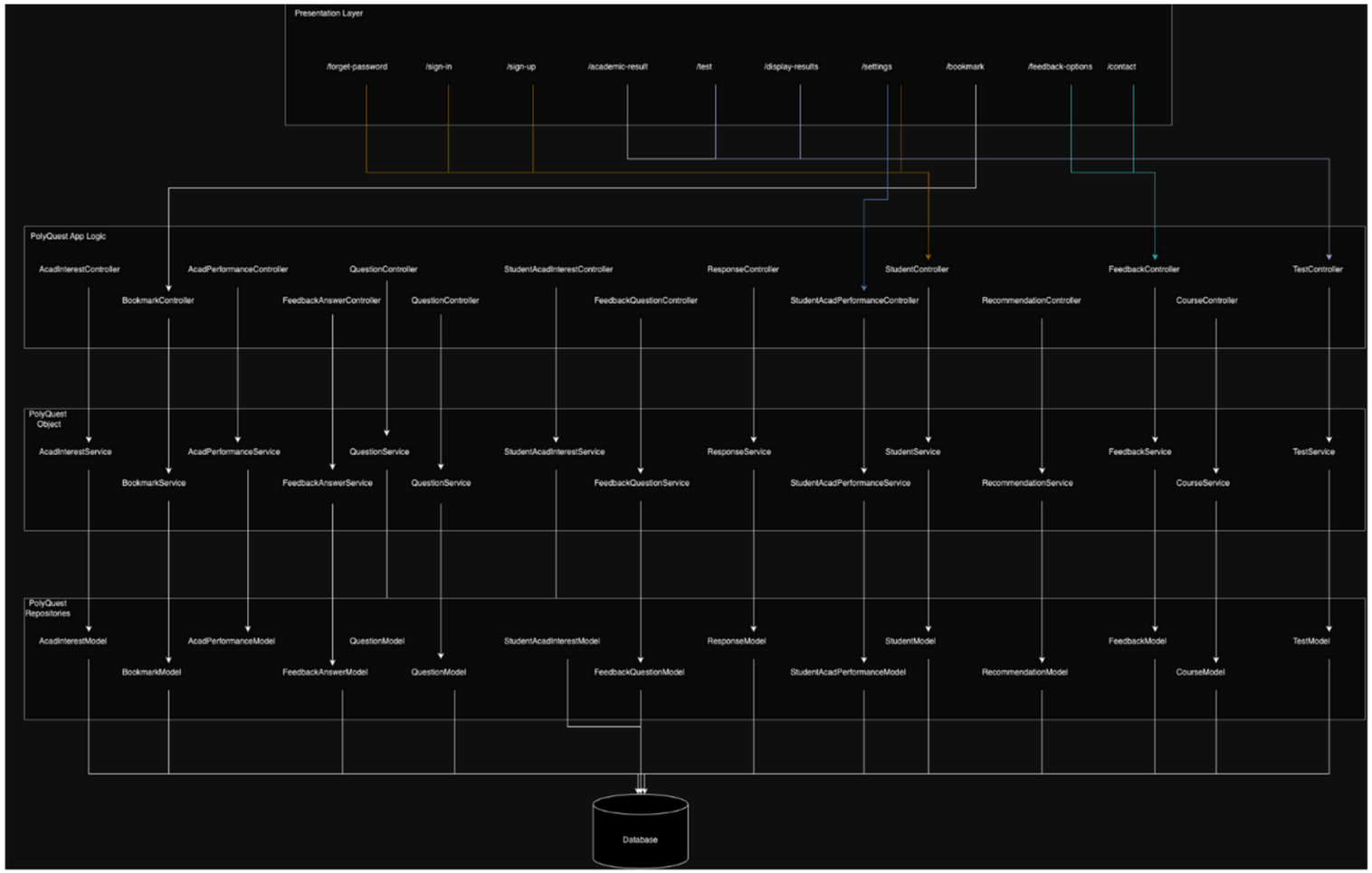
SCRUM Process

System Design

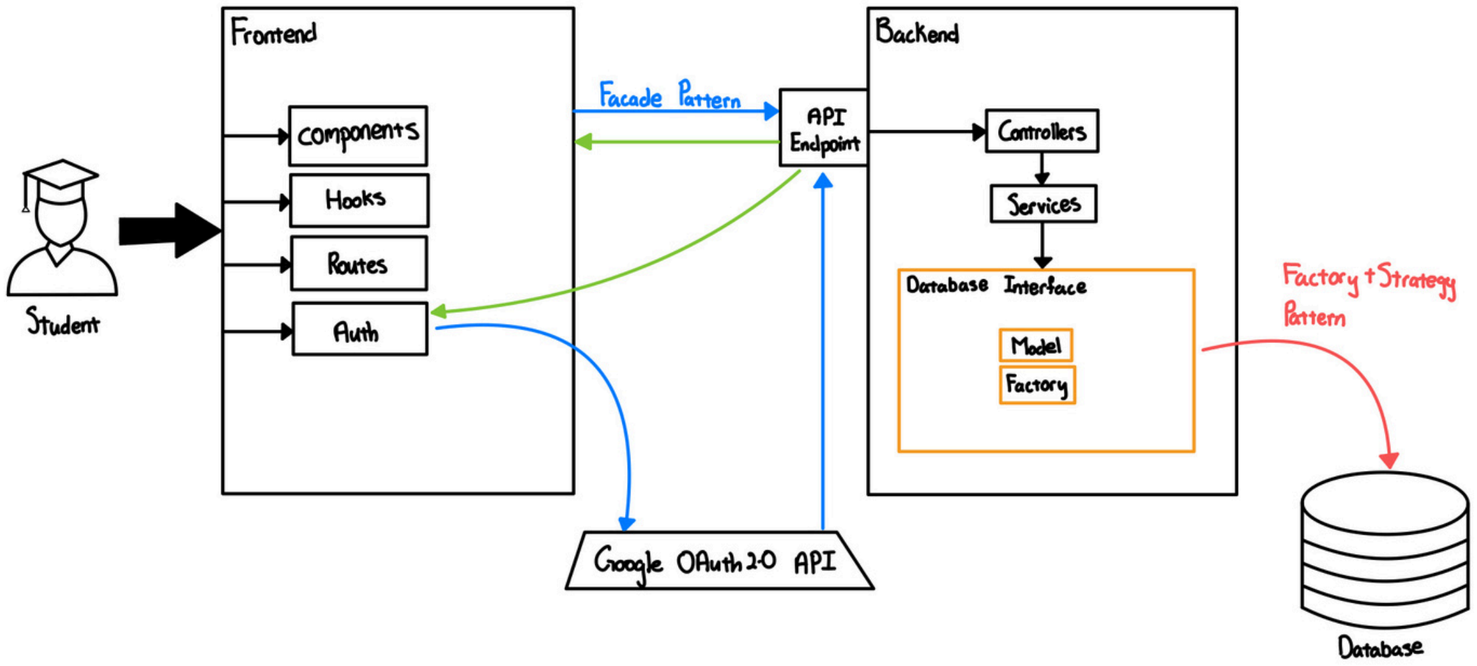
Backend Class Diagram



Architecture Diagram (Layered Architecture)



Overview Diagram



Design Patterns

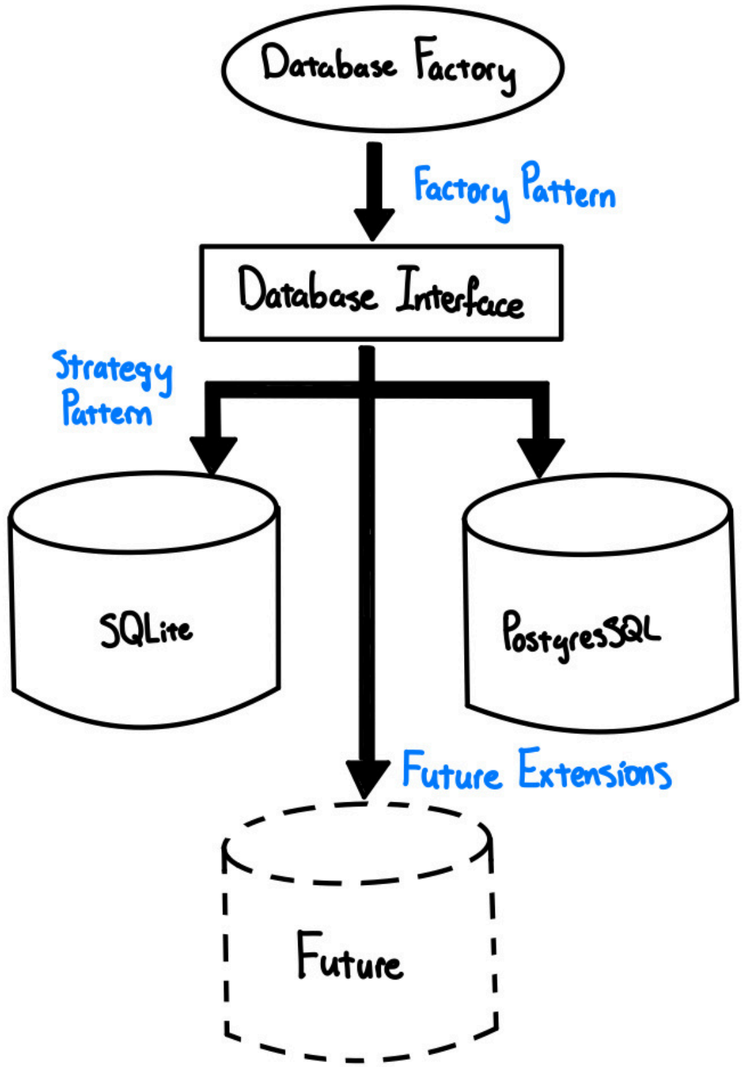
Strategy & Factory Pattern

Flexible Behaviour & Modularity

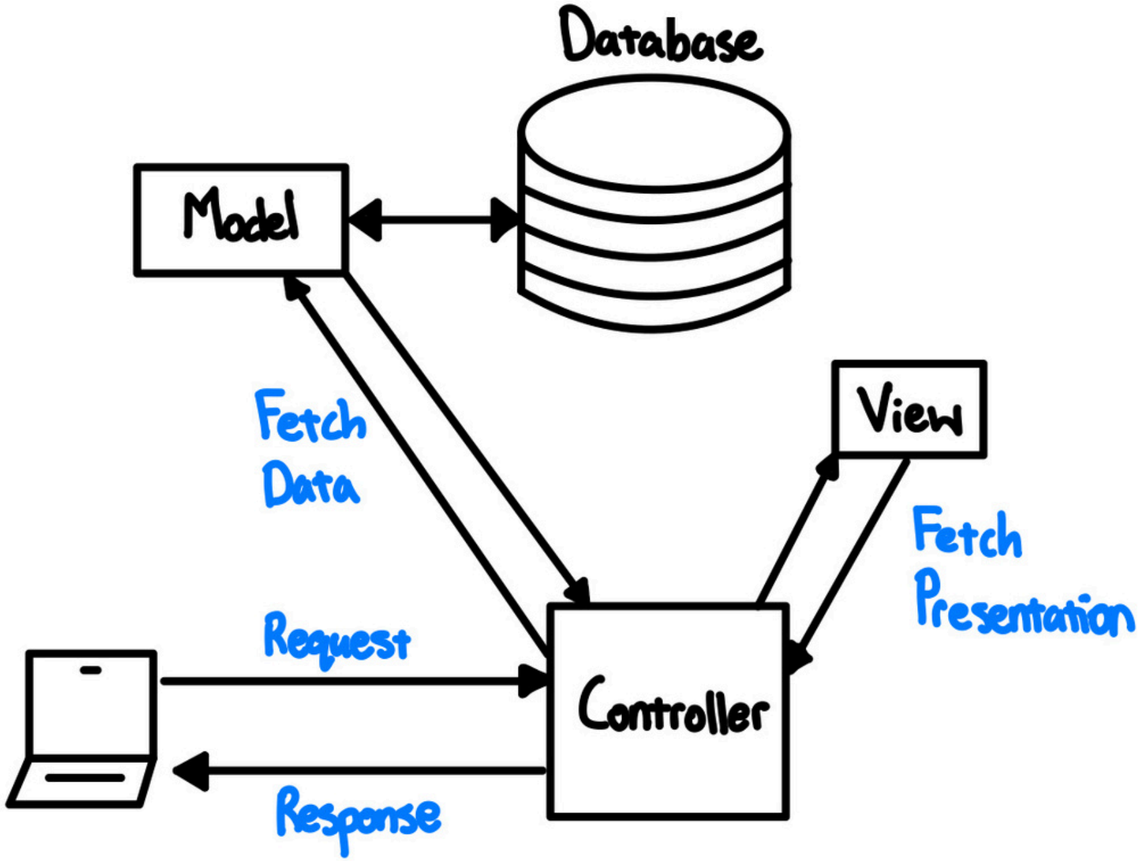
MVC

Model-View-Controller

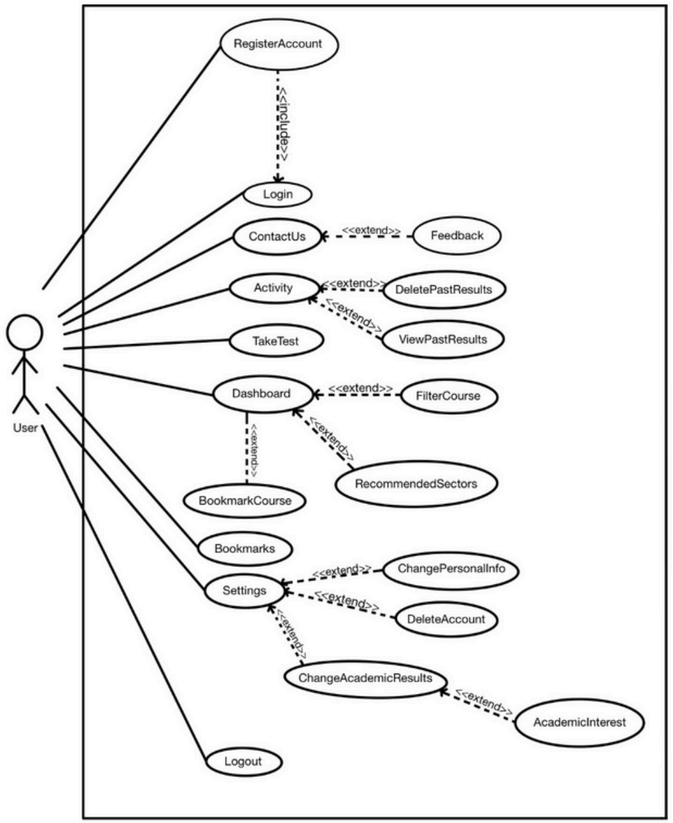
Strategy & Factory Patterns

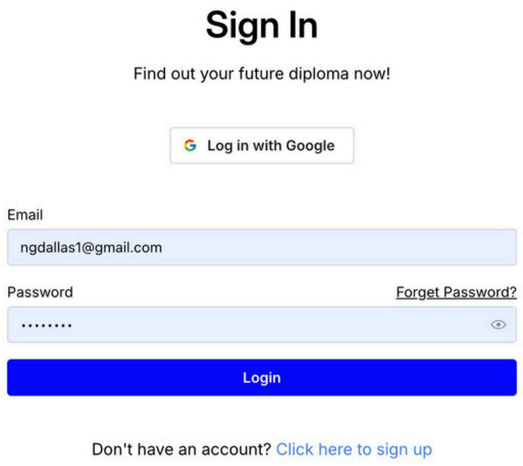


Model-View Controller



Traceability in Project Deliverables





1.1.1 Login (Email & Password)

No.	Email Password	Expected Output	Actual Output	Pass / Fail ?
1.	abc@gmail.com "Empty password"	"Please fill in all fields"	"Please fill in all fields"	Pass
2.	"Empty email" abc123	"Please fill in all fields"	"Please fill in all fields"	Pass

- Using the use case diagram to create our features (E.g Sign Up, etc..)

LIVE DEMO

FUTURE DEVELOPMENTS

INPUT

PDF Transcript Upload

Sharing

Easy Sharing on Social Media

Details

More detailed information

THANKS!