SC2006 Project: Edufinder

Lim Kiat Yang Ryan, Peng Sizhe, Tarun Ilangovan, Yong Chee Seng, Yu Wenhao

Table of contents

01

Application Overview

What is Edufinder and its Use Cases?

What is Edufinder?

Problem Statement:

Though websites to find school details using filters already exist, they do not allow users to:

Have a more **comprehensive** outlook on schools (No public comments)

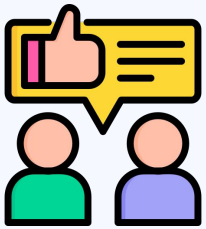
Compare between schools **interactively**

Easily find **travel times** (car, public transport, cycling) between home & school

What is Edufinder?

Aim:

To make school exploration **interactive** and smarter, we want to use **NLP**, **user feedback** and **Google Maps API** to help parents and students make better and more **suitable** choices



What can it do?

Filtering

Users can use a **variety** of filters to look at **relevant** schools for themselves or their children

Comments

Authenticated users can post comments & replies under school pages, giving others a more complete impression of the school

Comparison

Users can choose multiple schools and enter a **NLP prompt** to let the system choose the most **suitable** school for them

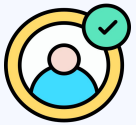
Travel Times

System shows the users travel times (using **various** modes of transport) from **home** to **school**

Who uses Edufinder?

Anonymous Users

These users could be **casual** users, like parents/children that are just viewing school details of individual school pages

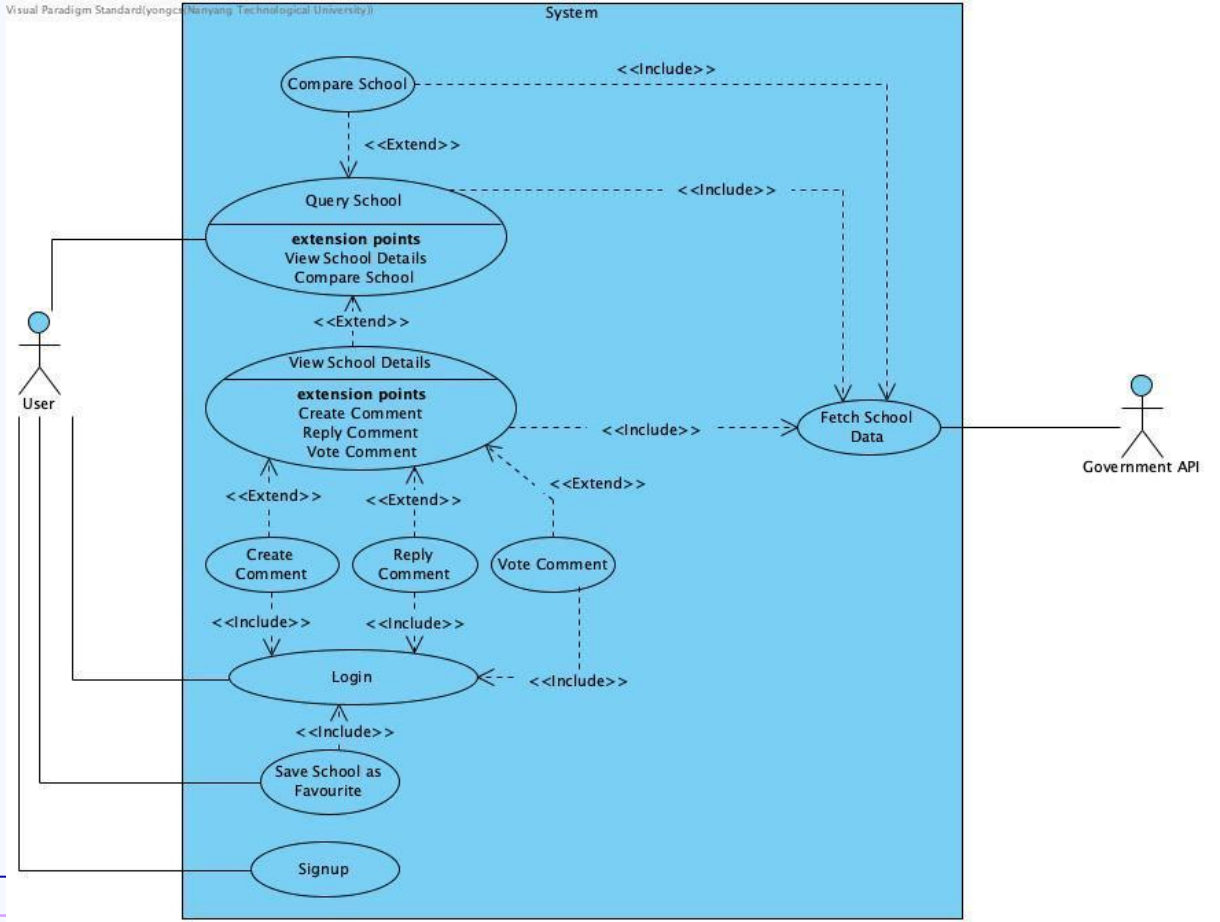


Authenticated Users

These users are able to access more **features**, such as

- **Commenting/Replying**
- **Saving** Schools
- **Comparing** schools

Use Case Diagram



02

Live Demonstration

See Edufinder in Action!

03

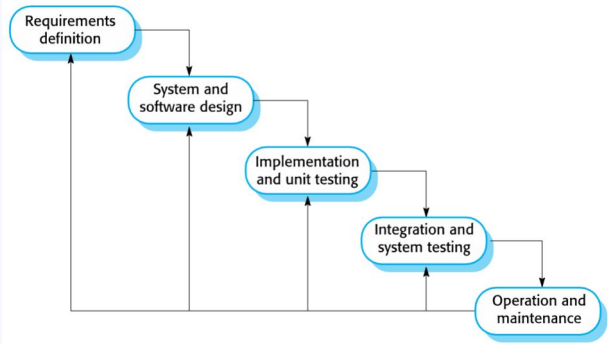
Software Development Life Cycle

How we developed Edufinder

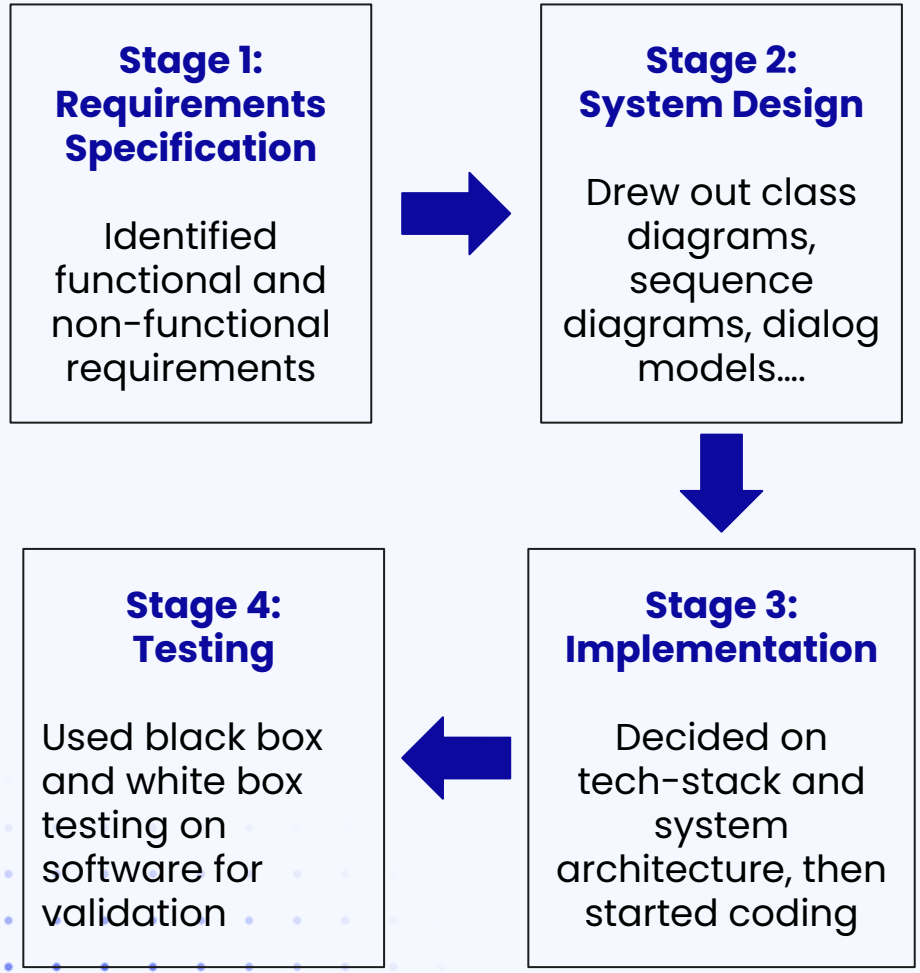
Waterfall Method

We adopted the **waterfall** method when developing our software as we wanted a **plan-driven** approach

- Relatively **simple** with **well-defined** requirements
- We **did not anticipate** much changes to our requirements
- Waterfall stages coincided well with **SWE lab deliverables**



SDLC


Stage 1: Requirements Specification
Identified functional and non-functional requirements
Stage 2: System Design
Drew out class diagrams, sequence diagrams, dialog models....
Stage 3: Implementation
Decided on tech-stack and system architecture, then started coding
Stage 4: Testing
Used black box and white box testing on software for validation

Functional Requirements

1. The user shall be able to query the system for schools using zero, one, or multiple **search requirements**.
2. The user shall be able to view **school details** when selecting a school from the query results.
3. The user shall be able to compare 2 schools side by side from the query results, including an option to input a **NLP prompt.**
4. The system shall enforce **authentication** for all functions that require user identity.
5. The system shall have a **comment section** under the school detail page.
6. The user shall be able to **save** a school as **favourite**.

Non-Functional Requirements

Performance

System shall respond to school query results in **<2 seconds** **95%** of the time

Usability

80% of **first-time users** shall make a simple school search request within **2 minutes** of opening the website

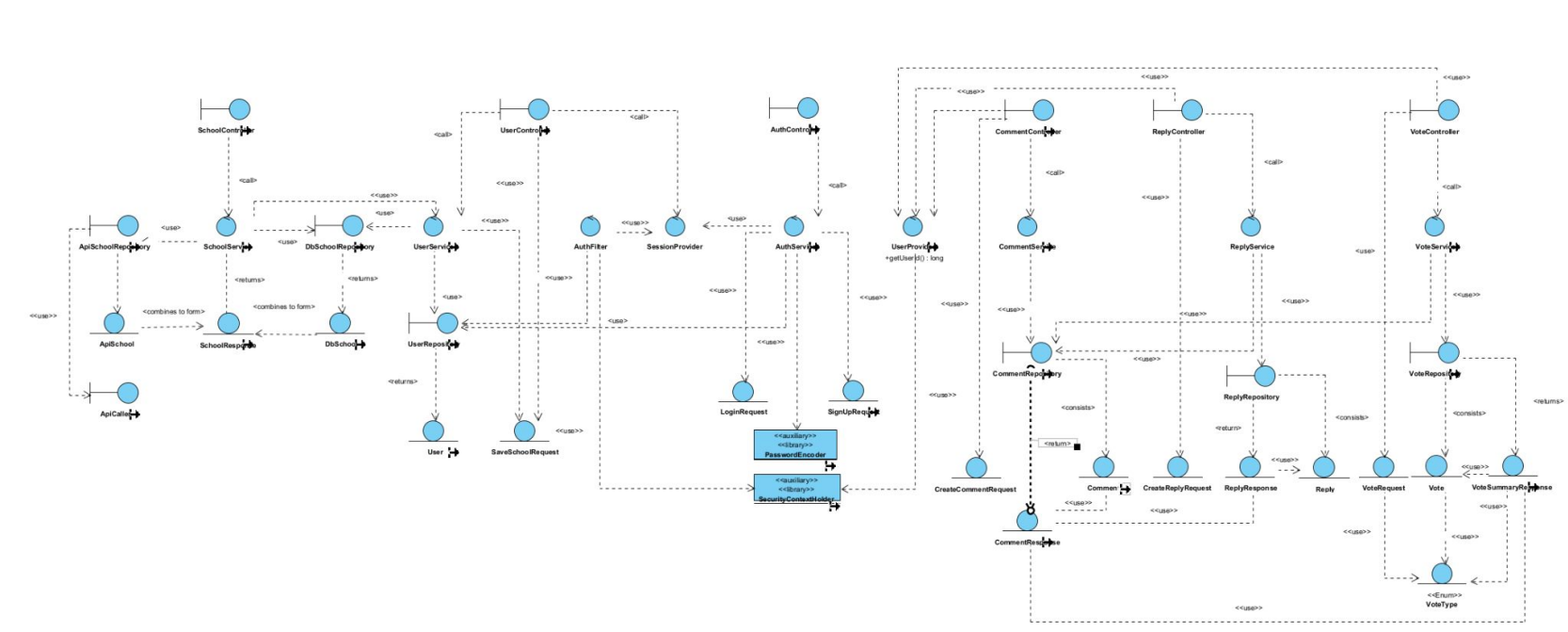
Reliability

System should be up **99.5%** of the time, excluding planned maintenance periods

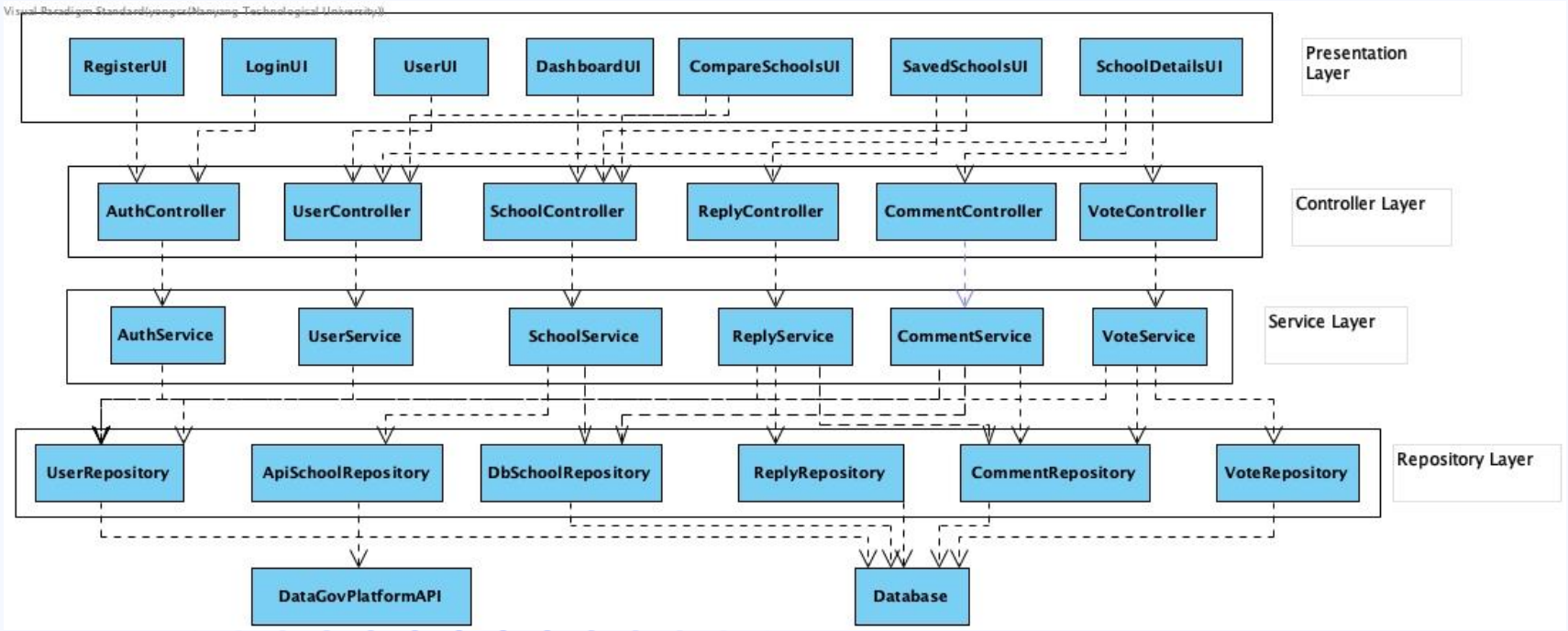
Authentication

System shall use **JWT authentication** to **maintain sessions** for authenticated users

Class Diagram

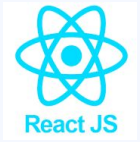


System Architecture



Techstack

Front-end



Back-end

Database + APIs

04

System Design

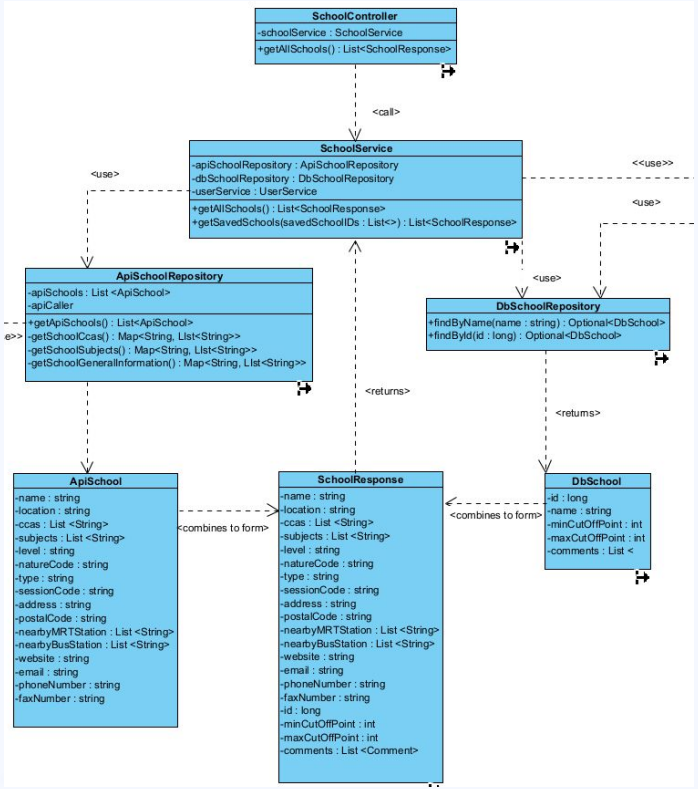
SE Practices & Design Principles We Used

MVC Architecture

What is MVC?

Model-View-Controller is an architecture that contains

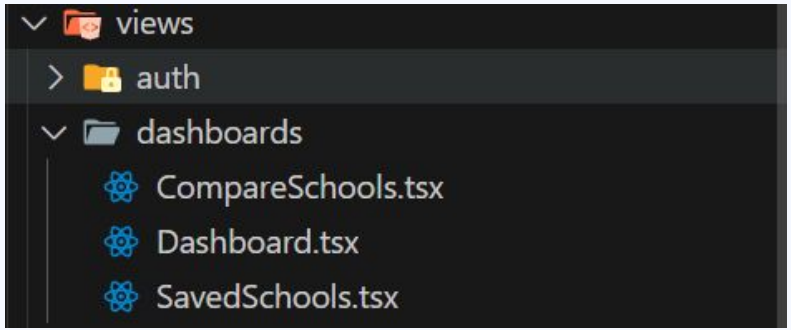
- View: Frontend UI that user interacts with
- Controller: Handles the business logic
- Model: Manipulates data



MVC Architecture

How is this MVC?

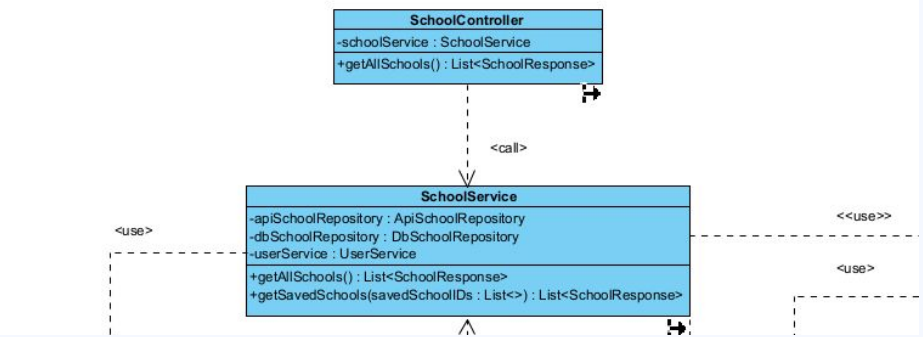
View: Represented by a React Page that calls an API which activates SchoolController, in this case Dashboard.tsx



MVC Architecture

How is this MVC?

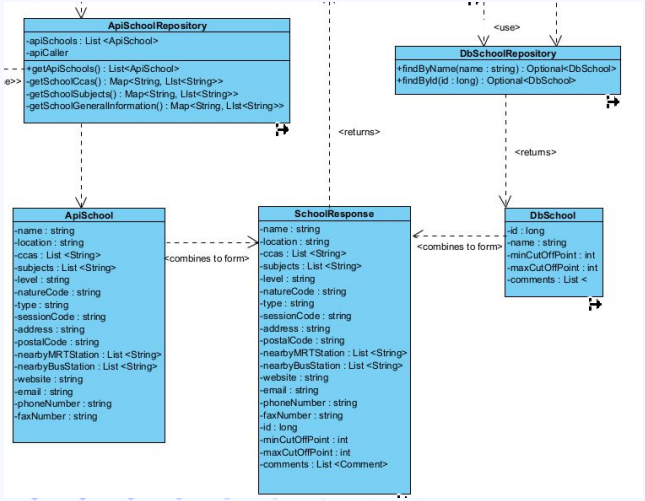
Controller: SchoolController and SchoolService combine to act as the controller of logic



MVC Architecture

How is this MVC?

Model: ApiSchoolRepository & DbSchoolRepository act as the model, managing data access and manipulation



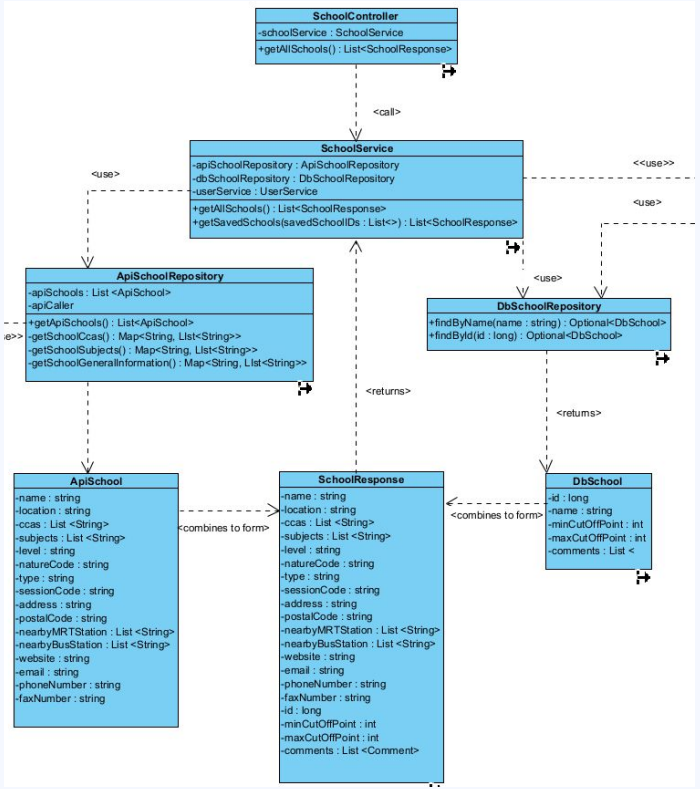
MVC Architecture

Why MVC?

Low Coupling: Each layer is not dependent on each other and can be interchangeable

High Cohesion: Related functionalities are grouped together

Independent Testing: Frontend and backend can be tested in isolation



Singleton Pattern

All Spring-managed beans are singletons by default, which ensures:

- **Controlled Access**: Crucial for managing shared resources such as database connections.
- **Consistency**: All services interact with the same instance, preventing **conflicting** versions.
- **Efficiency**: Only one instance of the bean is kept in **memory**

Observer Pattern

React Hooks (useState & useEffect) demonstrates the observer pattern

- Observer: Page UI
- Subject: useState variables *e.g. schools*
- Subscription Method: useEffect

```
const [savedSchoolIds, setSavedSchoolIds] = useState<number[]>([]);
const [allSchools, setAllSchools] = useState<School[]>([]);
```

```
useEffect(() : void  => {
  setAllSchools(schools);
}, [schools]);
```

Step 1: Filtering school changes *schools* **variable**

Step 2: Observer **observes** this change via **useEffect**

Step 3: Observer **changes** the Page UI view based on changes

OOP Solid Design Principles

Single Responsibility Principle

```
@Service  yongcs106
public class AuthServiceImpl implements AuthService {

    private final UserRepository userRepository;  4 usages
    private final PasswordEncoder passwordEncoder;  3 usages
    private final SessionProvider sessionProvider;  3 usages
```

- **AuthServiceImpl** -> authentication logic
- **UserRepository** -> persistence
- **PasswordEncoder** -> encryption
- **SessionProvider** -> session management

Open-Closed Principle

```
public interface SessionProvider {  1 implementation  yongcs106

    /** Encodes {@code userId} into authentication token. ...*/
    String generateToken(Long userId);  11 usages  1 implementation  yongcs106

    /** Decodes authentication token into {@code userId}. ...*/
    Long validateAndGetUserId(String token);  4 usages  1 implementation  yongcs106

}
```

- **AuthServiceImpl** depends on the **SessionProvider** interface
- A different token type can be implemented without **modifying existing code**

Liskov Substitution Principle

```
@Service  yongcs106
public class AuthServiceImpl implements AuthService {

    private final UserRepository userRepository;  4 usages
    private final PasswordEncoder passwordEncoder;  3 usages
    private final SessionProvider sessionProvider;  3 usages
```

- Small, well-defined interfaces such as **AuthService**, **UserRepository**, and **SessionProvider**
- No class is forced to implement unused methods

Interface Segregation Principle

```
@Override  9 usages  yongcs106
public String login(LoginRequest loginRequest) {...}

@Override  11 usages  yongcs106
public String signup(SignupRequest signupRequest) {...}
```

- **AuthServiceImpl** implements **AuthService** interface, which defines the **login** and **signup** operations.
- Can be substituted anywhere an AuthService is required without causing errors
- Ensures consistency & promotes interface-based design

Dependency Inversion Principle

```
@Autowired  yongcs106
public AuthServiceImpl(UserRepository userRepository, PasswordEnco
    this.userRepository = userRepository;
    this.passwordEncoder = passwordEncoder;
    this.sessionProvider = sessionProvider;
}
```

- Dependencies such as **UserRepository**, **PasswordEncoder**, and **SessionProvider** are defined as interfaces
- Their concrete implementations are injected by Spring using **@Autowired**.

Other Good Practices

Don’t Repeat Yourself

- Common logic extracted into helper functions
- Avoids duplication
- Promotes code reuse

Readability

- Meaningful class & method names
- Minimised deeply-nested logic

Documentation

- Comprehensive JavaDoc comments
- Improves maintainability
- Easy onboarding

05

Traceability

Deep Dive into Our SignUp Use Case

Requirements

REQ-1: The user shall be able to create an account.

REQ-1.1: The user shall provide a username.

REQ-1.1.1: The username shall be a string of 6-14 characters.

REQ-1.1.2: The username shall be unique across the user database.

REQ-1.2: The user shall provide a password.

REQ-1.2.1: The password shall be a string of at least 8 characters.

REQ-1.2.2: The password shall meet minimum complexity requirements:

REQ-1.2.2.1: The password shall have at least 1 lowercase letter.

REQ-1.2.2.2: The password shall have at least 1 uppercase alphabet.

REQ-1.2.2.3: The password shall have at least 1 number.

REQ-1.2.2.4: The password shall have at least 1 non-alphanumeric character.

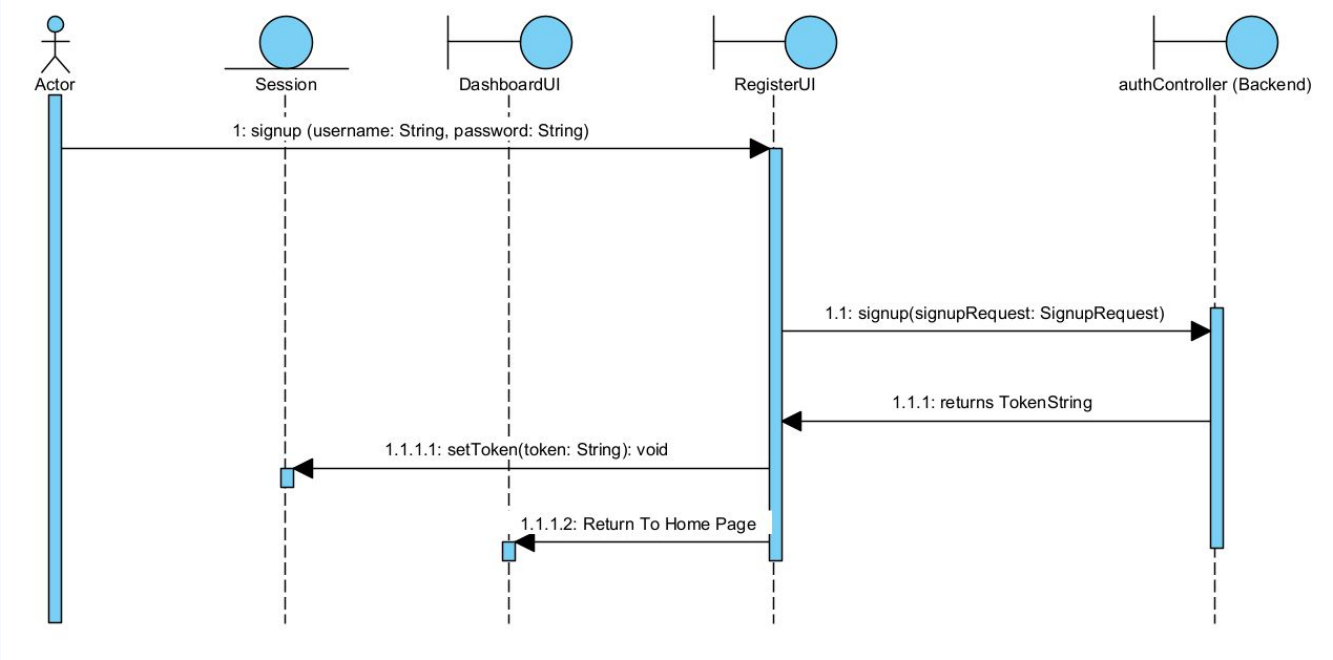
Use Case

4.6.2 Stimulus/Response Sequences

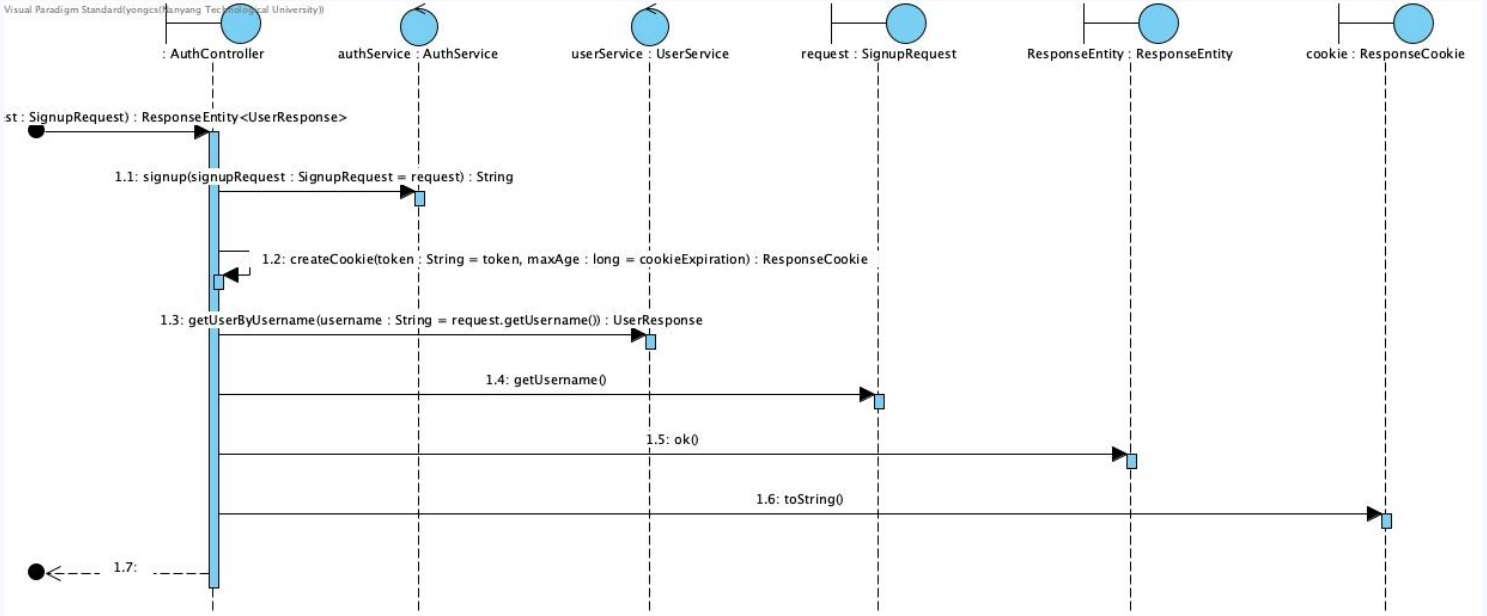
Use Case ID:	UCA-2		
Use Case Name:	Signup		
Created By:	Yong Chee Seng	Last Updated By:	Lim Kiat Yang Ryan
Date Created:	4 September 2025	Date Last Updated:	4 November 2025

Actor:	User (Initiating)
Description:	User shall be able to sign up an account by providing valid credentials.
Preconditions:	NA
Postconditions:	1. System stores user credentials securely. 2. System allow user to login with the provided credentials
Priority:	Medium-High
Frequency of Use:	Estimated 3-5 per day across all users
Flow of Events:	1. User navigates to "Signup" page. 2. User inputs username, password, and confirm password. 3. System checks whether the username is already taken. 4. If the username is available, the system validates that the password meets the security policy (e.g., minimum length of 8 characters, includes uppercase, lowercase, number, and special symbol). 5. If password is secure, system verifies that the password and confirm password match. 6. If the confirm password is same as password, system stores user credentials. 7. The system displays message "Account created. You can login now". 8. System redirects the user to the "Login" page.
Alternative Flows:	AFS4a: **Username Taken** 1. If the entered username does not exist, system displays error "Username already exists. Please choose another".
	AFS4b: **User Unavailable** 1. If system cannot access user data, system displays error "Signup unavailable. Please try again later". AFS5: **Weak Password** 1. If the password does not meet the security policy, system displays error: "Password does not meet security requirements". AFS6: **Password Mismatch** 1. If the password and confirm password do not match, system displays the error: "Passwords do not match". AFS7: **System Error** 1. If credentials cannot be stored due to a system error, system displays error: "Signup unavailable. Please try again later".
Exceptions:	NA
Includes:	NA
Special Requirements:	1. System shall enforce a strong password policy (e.g., minimum length of 8 characters, including uppercase, lowercase, numbers, and special symbols). 2. System shall complete the signup process within 5 seconds under normal load conditions. 3. System shall protect against automated/bot signups (e.g., reCAPTCHA or an equivalent). 4. System shall store all user passwords using industry-standard encryption methods (e.g., bcrypt). 5. The system shall ensure that users explicitly agree to the **Terms and Conditions** and **Privacy Policy** in a clear and visible manner before account creation.
Assumptions:	NA
Notes and Issues:	NA

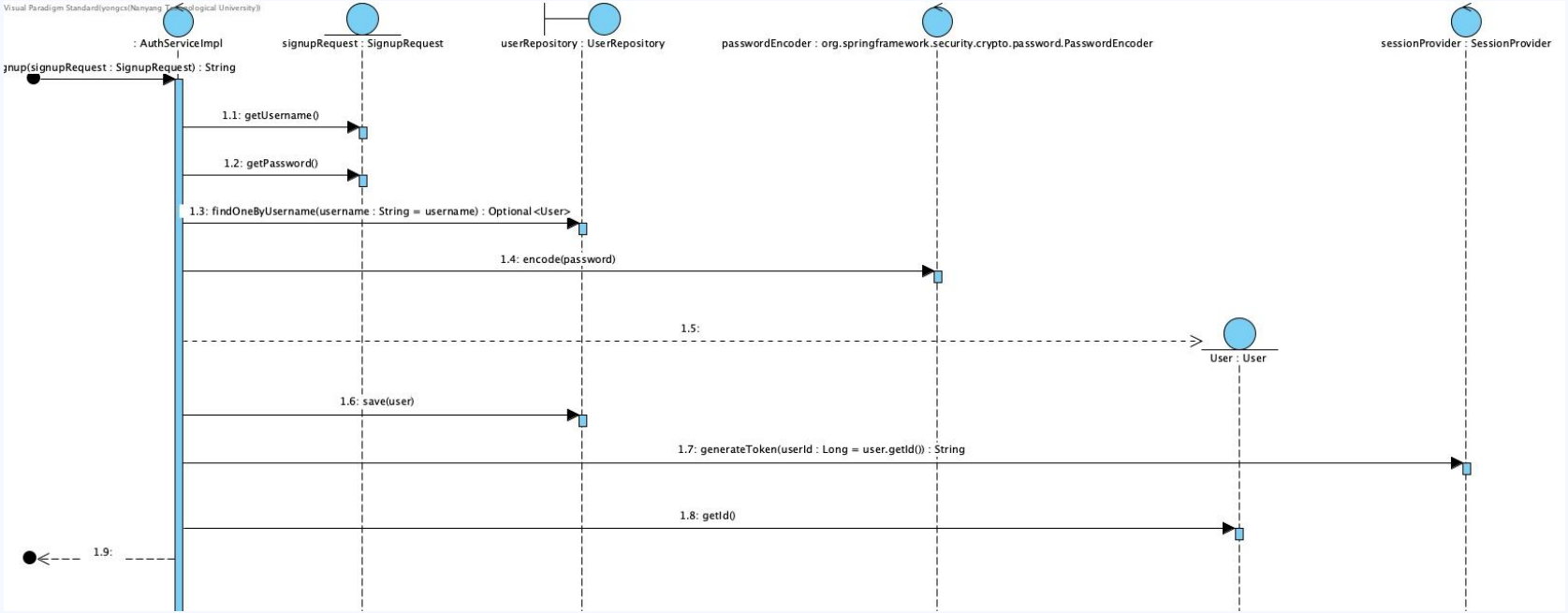
Sequence Diagram (Frontend)



Sequence Diagram (AuthController)



Sequence Diagram (AuthService)



Implementation (AuthController)

```java
@PostMapping("/signup")  yongcs106
public ResponseEntity<UserResponse> signup(@Valid @RequestBody SignupRequest request) {
    String token = authService.signup(request);

    ResponseCookie cookie = createCookie(token, cookieExpiration);
    UserResponse user = userService.getUserByUsername(request.getUsername());

    return ResponseEntity.ok()
        .header(HttpHeaders.SET_COOKIE, cookie.toString())
        .body(user);
}
```

Implementation (AuthService)

```java
@Override  11 usages  yongcs106
public String signup(SignupRequest signupRequest) {
    String username = signupRequest.getUsername();
    String password = signupRequest.getPassword();

    User user = userRepository.findOneByUsername(username).orElse( other: null);
    if(user != null){
        throw new DuplicateUsernameException();
    }

    password = passwordEncoder.encode(password);

    user = User.builder()
        .username(username)
        .password(password)
        .build();

    userRepository.save(user);

    return sessionProvider.generateToken(user.getId());
}
```

Implementation (UserService)

```
@Override  12 usages  yongcs106
public UserResponse getUserByUsername(String username) {
    User user = userRepository.findOneByUsername(username)
        .orElseThrow(() -> new UserNotFoundException(username));

    return userMapper.toUserResponse(user);
}
```

Implementation (SignupRequest)

```java
public class SignupRequest {

    @NotNull
    @Size(min = 6, max = 14)
    private String username;

    @NotNull
    @Size(min = 8, message = "Password must be longer than 8 characters")
    @Pattern(
        regexp = "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$!%*?&]{8,}$",
        message = "Password must contains at 1 uppercase, 1 lowercase, 1 number, and 1 special symbol."
    )
    private String password;

}
```

Separation of Concerns

Each class has a **distinct responsibility**, reducing **coupling** and improving **maintainability**.

AuthController

Handles incoming HTTP requests & delegates business logic to service layer

AuthService

Contains the core business logic for user signup

UserService

Provides user-related operations, such as retrieving user information.

UserRepository

Manages data persistence by interacting directly with the database.

SignUpRequest

Acts as a Data Transfer Object (DTO) that encapsulates user input

DTOs and Mapper Pattern

Data Transfer Object (DTO) and Mapper patterns are applied to improve **modularity** and **maintainability**.

Decoupling

The service and controller layers remain independent of how data is represented externally.

Security

Sensitive entity fields (password hashes) are excluded from UserResponse

Maintainability

Modifying fields requires minimal changes to business logic

Reusability

Mapper can be reused in different use case.

Exceptions and Status Code

Custom exceptions are used to provide standardized HTTP status codes and meaningful error messages and that helps both developers and users understand the issue.

UserNotFoundException

Message: User not found with username: {username}

Code: 404 (Not Found)

DuplicateUsernameException

Message: Username Existed

Code: 409 (Conflict)

Exceptions and Status Code

These exceptions are thrown from the service layer, where business validation occurs.

A global exception handler catches them and constructs uniform error responses, ensuring

- **Consistency**: All API errors follow a common structure and status convention.
- **Clarity**: Clients receive clear, human-readable messages suitable for display.
- **Maintainability**: New error types can be added easily without changing controller logic.

Testing

We used JUnit write unit test on **AuthController** & **AuthService** and these tests can be found in our git repository.

AuthController: Equivalence class testing
AuthService: Basic Path Testing

Equivalence Class Testing

Test ID	Input	Equivalence Class	Expected Output
T1	username: valid_username password: AbCd123@	Valid	Status: 200 Token: Issued and Set Body: User Information
T2	{}	Malformed input	Status: 400
T3	password: AbCd123@	Missing Username	Status: 400
T4	username: valid_username	Missing Password	Status: 400

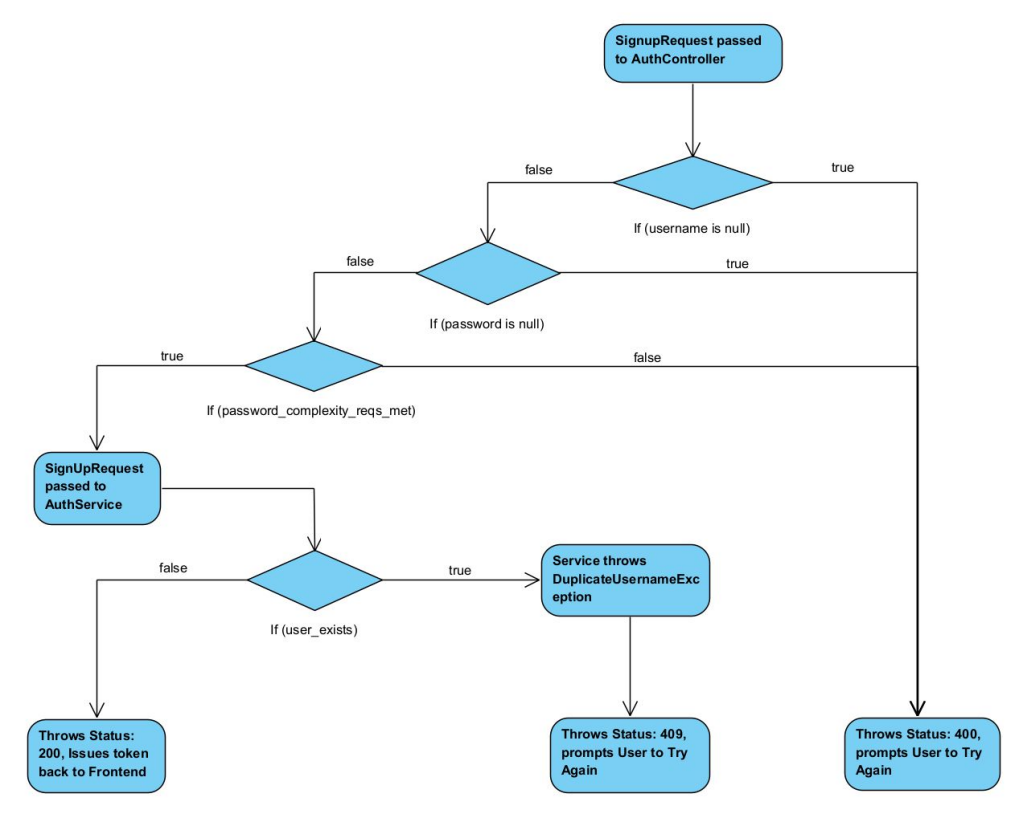
Equivalence Class Testing

Test ID	Input	Equivalence Class	Expected Output
T5	username: user1 password: AbCd123@	Short Username	Status: 400
T6	username: user12341234123 password: AbCd123@	Long Username	Status: 400
T7	username: user12 password: AbCd123@	Valid Shortest Username	Status: 200
T8	username: user1234123412 password: AbCd123@	Valid Longest Username	Status: 200

Equivalence Class Testing

Test ID	Input	Equivalence Class	Expected Output
T9	username: valid_username password: ABCD123@	No Lower Case Password	Status: 400
T10	username: valid_username password: abcd123@	No Upper Case Password	Status: 400
T11	username: valid_username password: Abcdefg@	No Digit Password	Status: 400
T12	username: valid_username password: Abcd1234	No Special Symbol Password	Status: 400
T13	username: valid_username password: Abab12@	Short Password	Status: 400

Basic Path Testing



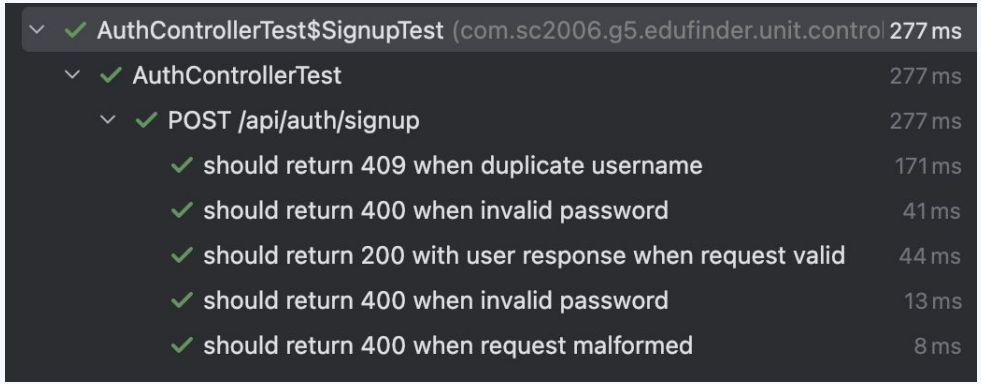
Basic Path Testing

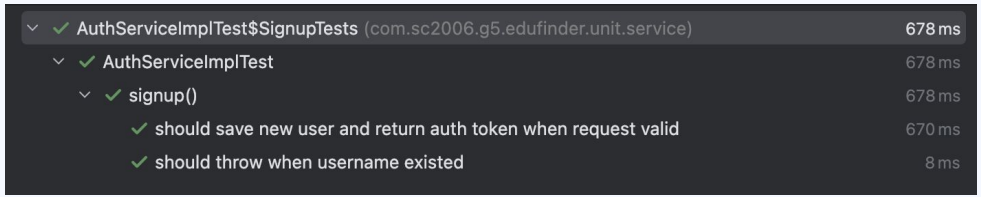
```java
@Override  11 usages  yongcs106
public String signup(SignupRequest signupRequest) {
    String username = signupRequest.getUsername();
    String password = signupRequest.getPassword();

    User user = userRepository.findOneByUsername(username).orElse( other: null);
    if(user != null){
        throw new DuplicateUsernameException();
    }

    password = passwordEncoder.encode(password);

    user = User.builder()
        .username(username)
        .password(password)
        .build();

    userRepository.save(user);

    return sessionProvider.generateToken(user.getId());
}
```

Basic Path Testing

Path ID	Predicate Coverage	Input Condition	Expected Output
P1	DuplicateUsername = false	Username does not exist	Password is encoded; user is saved; token with user ID is returned
P2	DuplicateUsername = true	Username already exists	DuplicatedUsernameException is thrown

Testing Result





06

Future Improvements

How can Edufinder be further developed?

Future Improvements

Administrative Page

To **moderate** comments & replies to **prevent misuse**

Comment Sentiment Analysis

Instead of manually browsing all the comments, add a **comment analyser** that captures the **overall sentiments** of a school

Datacrawler for COP

Instead of manual input, implement a **datacrawler** to find most recent COPs after exam season

Thank You!