IMPLEMENTATION AND ANALYSIS OF SOFTWARE DEVELOPMENT IN SPRING BOOT

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Xinyu Luan

2021

SIGNATURE PAGE

THESIS:	IMPLEMENTATION AND ANALYSIS OF SOFTWARE DEVELOPMENT IN SPRING BOOT
AUTHOR:	Xinyu Luan
DATE SUBMITTED:	Fall 2021
	Department of Computer Science

Gilbert Young
Thesis Committee Chair
Professor of Computer Science _____

Yu Sun
Professor of Computer Science _____

Dominick A. Atanasio
Professor of Computer Science _____

ABSTRACT

In today's cloud computing era, a full stack application – the development of both frontend (client side) and backend (server side) components of a web application [1] is widely used. In such applications, the backend represents the heart and essence of a program. Without a scalable solution, an application cannot be expanded to meet today's cloud computing environment needs. Therefore, a backend application's implementation and system design should be thoughtfully examined. My research uses a well know backend framework called Java Spring Boot, which has a widely accepted usage rate of about 62% within the computer software industry, thus, some of the most renowned web applications incorporate the Java Spring Boot framework for building a backend application. In this real-world case study, a client that seeks to use an online educational platform can benefit from daily operations between its administration, instructors, and students. My thesis explores the system design and implementation principles where clients will assume multiple roles and fulfil their purpose in an online educational curriculum.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Statement of Problem

As mentioned in the abstract abstracting statement of this research, in the cloud computing world; while the end user needs various fashions to interact and access to an application such as allowing users to interact with a designed application via mobile or an online browser, a full stack application was born to meet such demands. In this case, a mobile device or online browser will be considered as a form of front-end development. However, our research will focus on the backend application where behind the scenes, all business logic needs to be in place, and it processes all requests and responds to the front-end users. Moreover, a question raised, while an immediate solution can be developed in a short period of time, but when the user may need to expand its business and functionalities, how should a backend application be designed or what are the best practices to meet such a challenge? The answer relies on the choice of a powerful backend application framework that supports the paradigm of Object-Oriented Programming (OOP) and well-built support functions that enhances the twenty years' worth of the software industry's dynamic growth – Java Spring Boot Framework.

1.2 Targeting Case – HBU's academy

Our target case is the client – HBU academy. HBU academy is an online educational curriculum that aims at improving and modernizing a virtual classroom. A motivating factor behind the necessity for an online educational service is due to the effect of the global pandemic (Covid 19), in addition, HBU was expecting an influx of new admissions in the future. As a result, a scalable design was needed.

1.3 Challenges of HBU

During the discovery phase of gathering client requirements, the following was documented:

1. HBU needs a platform for students to learn such subject expertise online.
2. Students need to interact with instructors and administrators through its platform.
3. Upon competition, students need to get a certification for HBU issued degree.

4. Instructors can design and upload any course content, such as lecture videos.
5. Instructors need to be able grade assignment and tests.
6. Admin can manage registration, students, instructors, and courses by using such platform.
7. Any possible components that HBU desired to add for later to meet its expanding needs.

Based on those challenges, one needs to come out with a system designed to meet the flexibility of handling a growing amount of work by adding additional resources and functionality to this system.

1.4 Research Organization

In Section 2 System Design, this research will conduct a formal system analysis and design based on HBU's system requirements gathering and challenges. Section 3 will focus on the technology of Java Spring Boot Framework. Finally, Section 4 HBU Implementation will go into details of how the Java Spring Boot Framework was used in HUB's implementation. Finally, in Section 5, the discussion of possible improvement will be generalized in the form of future work and conclusion.

2. DESIGN

2.1 Identified Components

Based on Section 1.3 Challenges of HUB, eight components were identified as necessary: 1) Course, 2) Instructor, 3) Student, 4) Course module, 5) Chapter component, 6) Section, 7) Utility, and 8) Admin. The below section will pinpoint the responsibilities and functions of the aforementioned components:

- Course component: It allows administrator to update the information and description of a course that composed an advertisement on the official public website.
- Instructor component: It keeps all the profile information of an instructor, as a type of user, who is working in the institution. At the same time, it will save all the course modules where the instructor built for uploading course content.
- Student component: It saves the profile information of a student, as one type of user in this system, who enrolled the institution. This component also adds course modules in addition to related grades for the student.
- Course module component: this component saves all the contents of a course for live education with one instructor and students. They can interact on this page for everything about the course content.
- Chapter component: a chapter is a partition of a course module. The way to divide the course content is decided by instructors. It is easier for students to check the outline of a course.
- Section component: instructors can use this component to create or edit different kind of files or pages in a chapter. Students can view the instructor posts in section type.
- Utility component: it is for instructors and students to send some of inputs with different formats. That provides the skeleton of different type of editing windows for users.

- Admin component: it saves the information of an admin of the institution. An administrator can provide a role to manage the whole system including the responsibility of managing all the courses, students, and instructors.

2.2 Business logic and use case analysis

During this phase of research, the use case analysis was driven by the business requirements and logic. Further, the use case analysis identified three major users: administrator, students, and instructors. The use case is depicted in the following (*figure 1*):

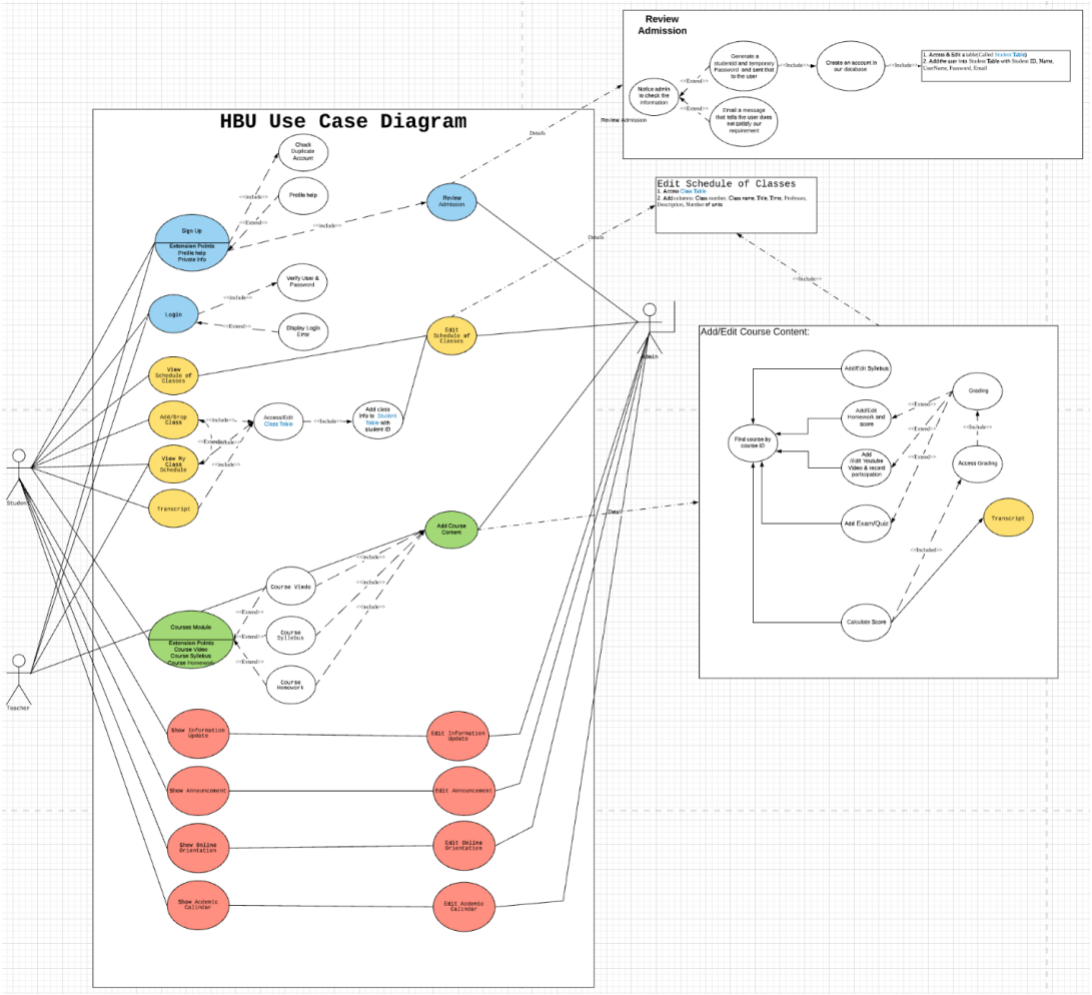


Figure 1. HBU Use Case Diagram

Then a thorough use case analysis report was generated and elaborated via each user's action and workflow. For example, the following is the use case of a user from the web can log into the HBU system (*figure 2*):

Use Case:	User Logs into HBU Website		
Created By:	Xinyu Luan	Last Updated By:	Xinyu Luan
Date Created:	8/26/20	Last Revision Date:	9/10/20
Description:	A user from the web can log into HBU.		
Actors:	General user, Student, Instructor, Administrator, New User (User that did not complete survey)		
Preconditions:	1. A user must have access to the internet 2. A user must have successfully registered an account within the HBU website prior to logging in.		
Postconditions:	1. A user successfully logs into the HBU given that they provided their username and password.		
Flow:	1. A web user visits the homepage of the HBU. 2. The web user clicks on 'login' within the homepage and the user will be directed to a login page. 3. The login page will display username and password text field entries. 4. The user will also have the option to enter their username and password in the corresponding text field entries. The user will click the 'submit' button. 4.1. If any of the entries are left blank, an error message will be prompted to the user, notifying that the fields that are left blank need to be filled out. Go back to step 4. 5. The login page will pass input validation to the user-login service: 5.1. If not valid, then the user login service prepares an error message for the login page to display. 5.2. the login page proceeds to warn the user that either the username or password is incorrect and needs to be reentered. 5.2.1. The user will have 3 attempts to login, otherwise, the administrator will lock them out of their account. An email will be sent to the user to notify them of their locked-out status and the resolution to rectify the problem. The user can click the active link contained in the email to reset the lockout. 5.3. The user will reenter the username and password and click on submit. 6. The user login service will verify the type of user logging in by confirming their ID. If the user logging in is a/an: 6.1. admin: they will be directed to the admin service. 6.1.1. the admin service will direct the user to the admin dashboard. 6.1.2. the admin dashboard will be displayed to the admin. 6.2. student: they will be directed to the student service. 6.2.1. the student service will direct the user to the student dashboard. 6.2.2. the student dashboard will be displayed to the student. 6.3. instructor: they will be directed to the instructor service. 6.3.1. the instructor service will direct the user to the instructor dashboard. 6.3.2. the instructor dashboard will be displayed to the instructor. 6.4. new user: they will be directed to the registration service. 6.4.1. the registration service will direct the user to the HBU survey questionnaire page. 6.4.2. The survey questionnaire page will be displayed to the new user.		
Requirements:	A user must have successfully registered as a user within the HBU website.		

Figure 2. User Log in User Case

The above will be used as an example how such analysis is conducted, but this research will not be going to detail of each use case; for purpose of this thesis, it is vital to demonstrate such

methodology instead of actual business logic and use case analysis implementation. They are out of scope for our discussion.

2.3 Apply use case analysis to develop User Interface (UI) Design

Through the use case analysis and information gathering, the UI design can concisely gather the information that are used by the users. It further solidifies the future database design. For example, in our UI design, we provided two versions of view for students and instructors respectively. The student and instructor view will directly affect the database design. As *figure 3* shown, an instructor can access to the courses that the instructor is responsible for. The list of modules on the left are the real courses that contains course content. The instructor can also get notification from students' messages for questions about the courses. The announcement part is for receiving messages from admin or institution. The view of students (*figure 4*) for this page is similar to what the instructor is viewing, but an instructor can modify course modules, which students cannot.

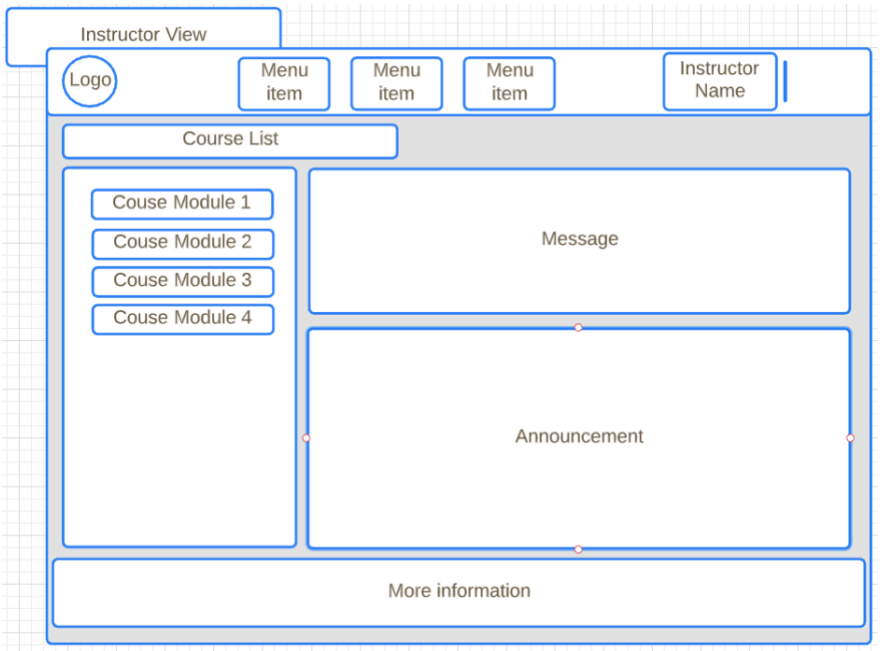


Figure 3. Instructor View 1

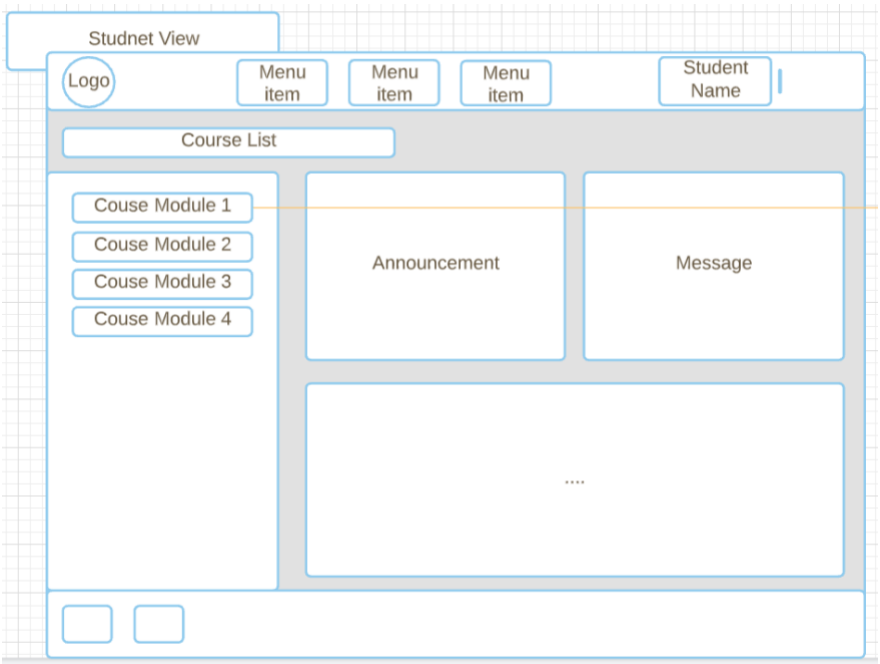


Figure 4. Student View 1

The exemplified workflow is as follows. When a user clicks on the course module button in *figure 3*, it will direct the user to the UI depicted in *figure 5* for instructor viewing, which shown the course details, section information, et cetera. However, if the user's role is student, it will have a different view as shown in *figure 6*.

Specifically, the concept is each course modules would include chapters and each of the chapter will contain different type of sections, such as lectures, announcement, assignments, and quizzes or exams. In reality, a user that has the role of the student will see a similar view with the user that has the role of instructor. The only difference between them are the access permissions. In other words, instructors can edit sections, students cannot.

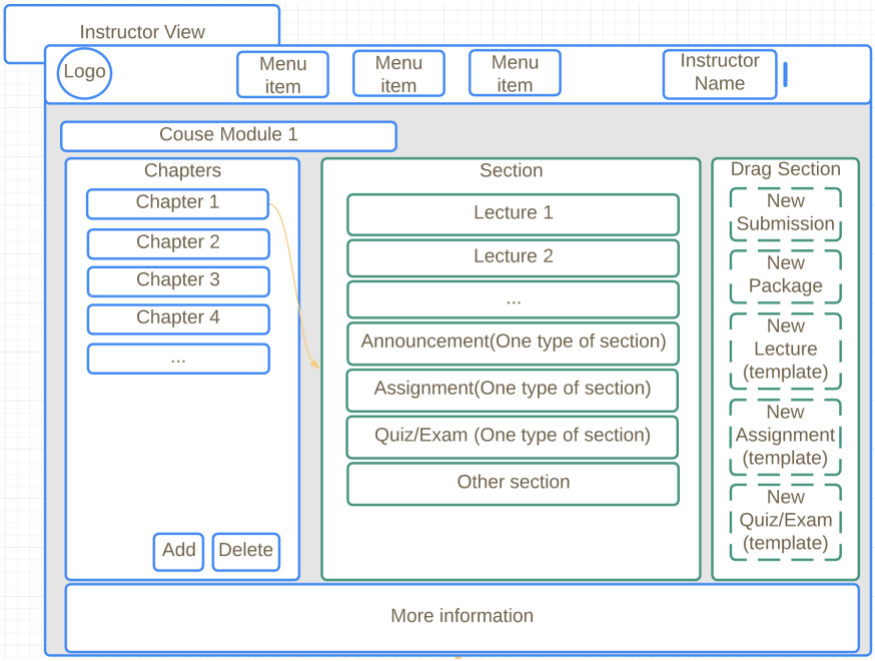


Figure 5. Instructor View 2

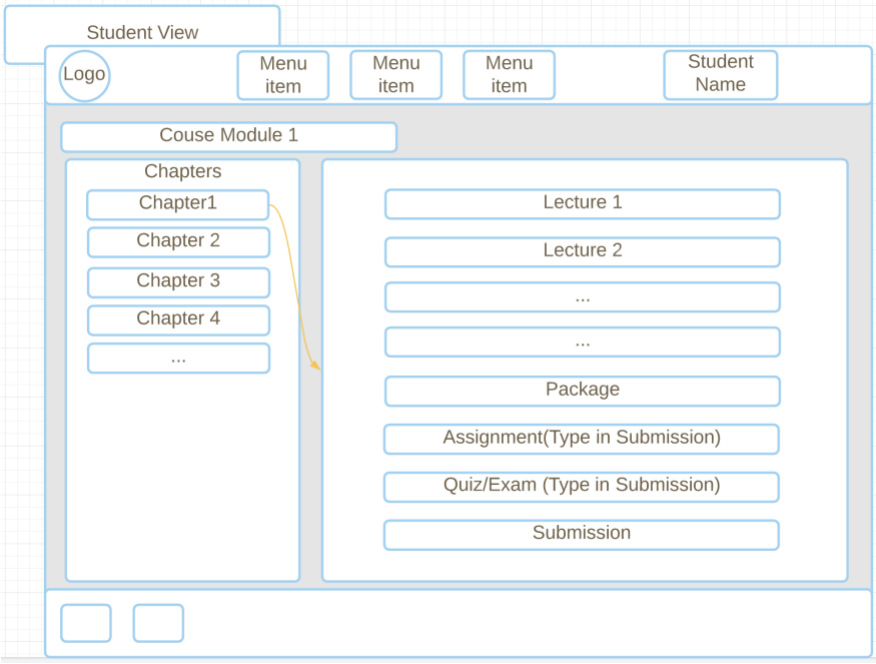


Figure 6. Student View 2

In the circumstance of when an instructor adds new contents such as new lectures, announcements, and such. He or she needs to add a new component. Those components can be categorized into three main types – 1) file component, 2) plaintext component and 3) multiple

choice component. Through the information gathering, we can use the extracted information into our database design.

2.4 Database Design

After finalizing the UI design, all user interactive information will be collected and then the database will be conceptualized and designed by utilizing the Entity Relationship (ER) Diagram, which depicted in *figure 7* below with Peter Chen's notation. It is the type of flowchart that illustrates how "entities" such as people, objects or concepts relate to each other within a system.

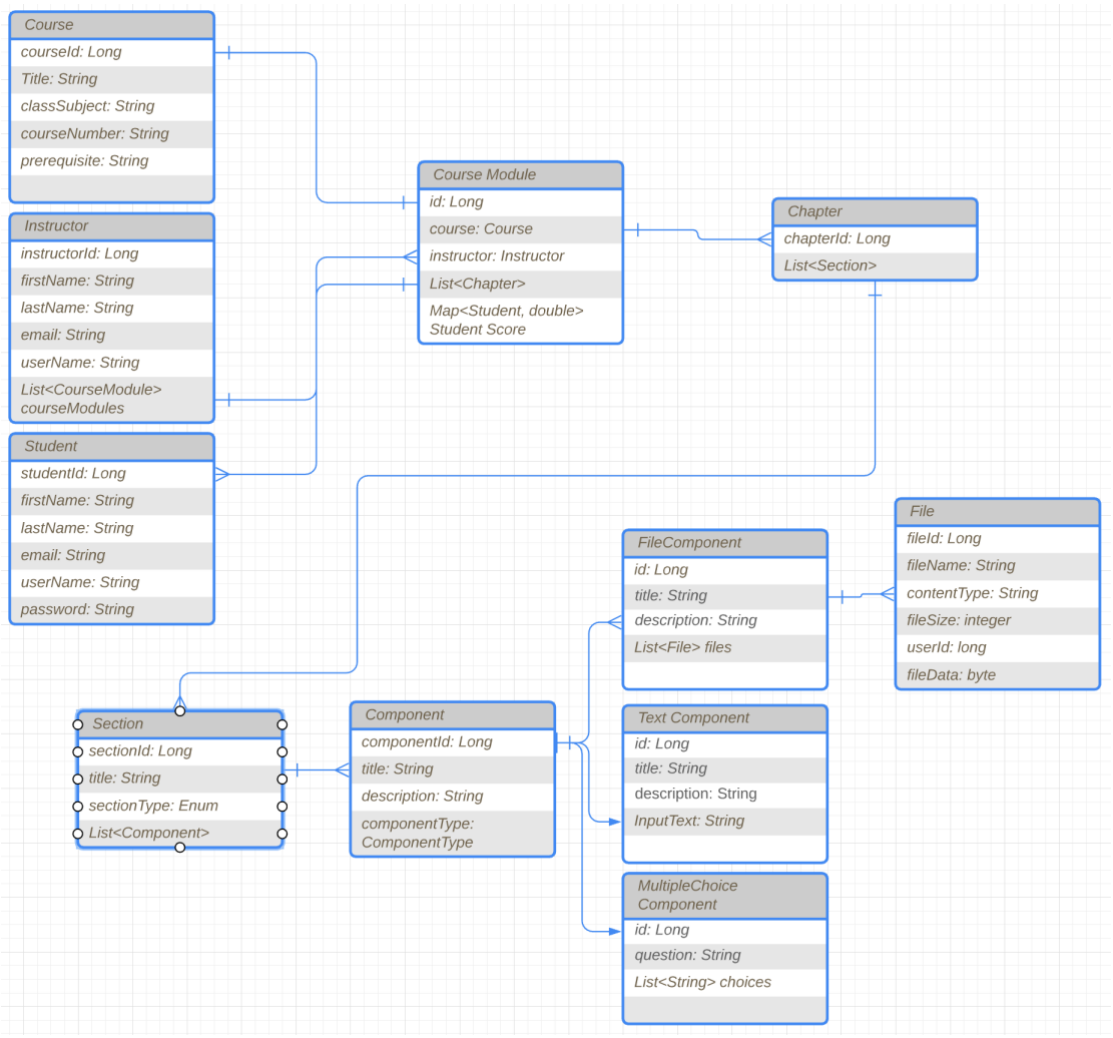


Figure 7. HBU Entity Relationship Diagram

As one can see that eight entities were created in database for the core system, including Course, Instructor, Student, Course Module, Chapter, Section, Components and Admin. However, Admin is the only entity doesn't have any relationship with others. The course module component is the central component that connects to course instructor, student, and chapter. If I have a recorder (row) in course module entity, it must indicate that one of the instructors is teaching one of the courses with a list of students and the course contents (a list of chapters). At the same time, the related IDs of course, instructor, chapters and students represent data in corresponding entities in the database. Every entity has various fields categorized by its properties in the project. For example, the course entity has course id, course title, course subject, course number and prerequisite of a course instance while the file component entity, Text component entity, and Multiple-choice component entity share partial fields from the component entity. The reason is justified from the design since the file, text, and multiple-choice components are subtypes of a component.

2.5 Objective-Oriented Design (OOD)

Since we are planning to use Java as programming language of the choice. Java is a programming language and computing platform released in 1995 [2]. Therefore, the approach of Objective-Oriented Design is vital for planning this project, which will come in handy for the Java Spring Boot Framework, such benefits will be mentioned in Section 3 Java Spring Boot. The following is our designed solution, which depicted in figure 8:

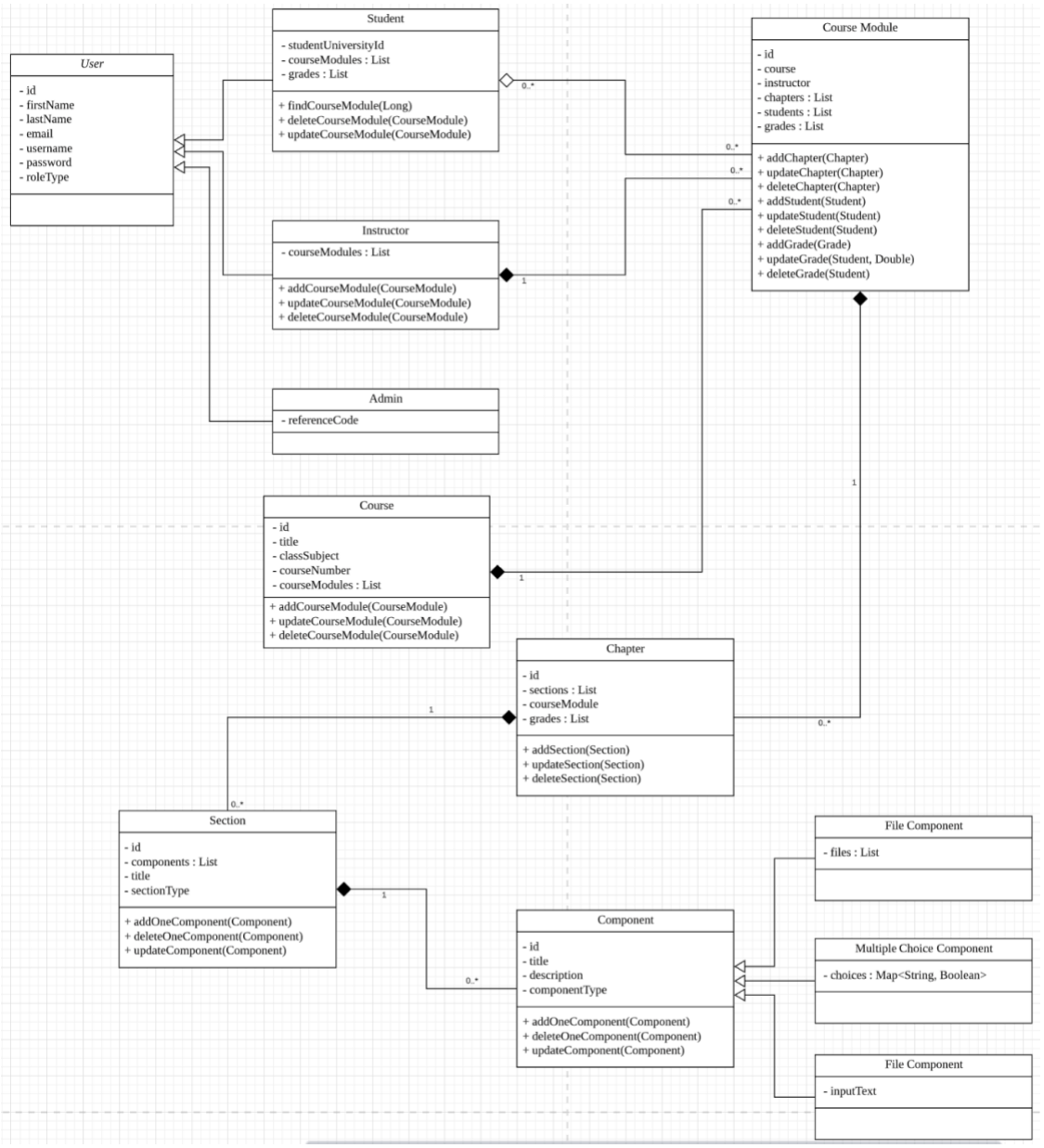


Figure 8 HBU Objective-Oriented Design

Our desired result of the online education system is an integrated system combining multiple classes in use. It includes the following nine major components:

2.5.1 Component, Section, Chapter, Course Module and Course Classes

One of the challenges of this HBU system is that it needs a container that is able to store all the information that simulates an entity of a course space, which has an instructor, students, course materials and students' grade. However, when the content of a course is breaking down into its

atomic form, it comes to realization that a course can possess many “Course Modules”; and each course module can have one/many “Chapter(s)”. Each “Chapter” can have one or many “Section(s).” Finally, we need a base unit to store content and give capabilities to add, delete and update, which embodies the purpose of the Component Class.

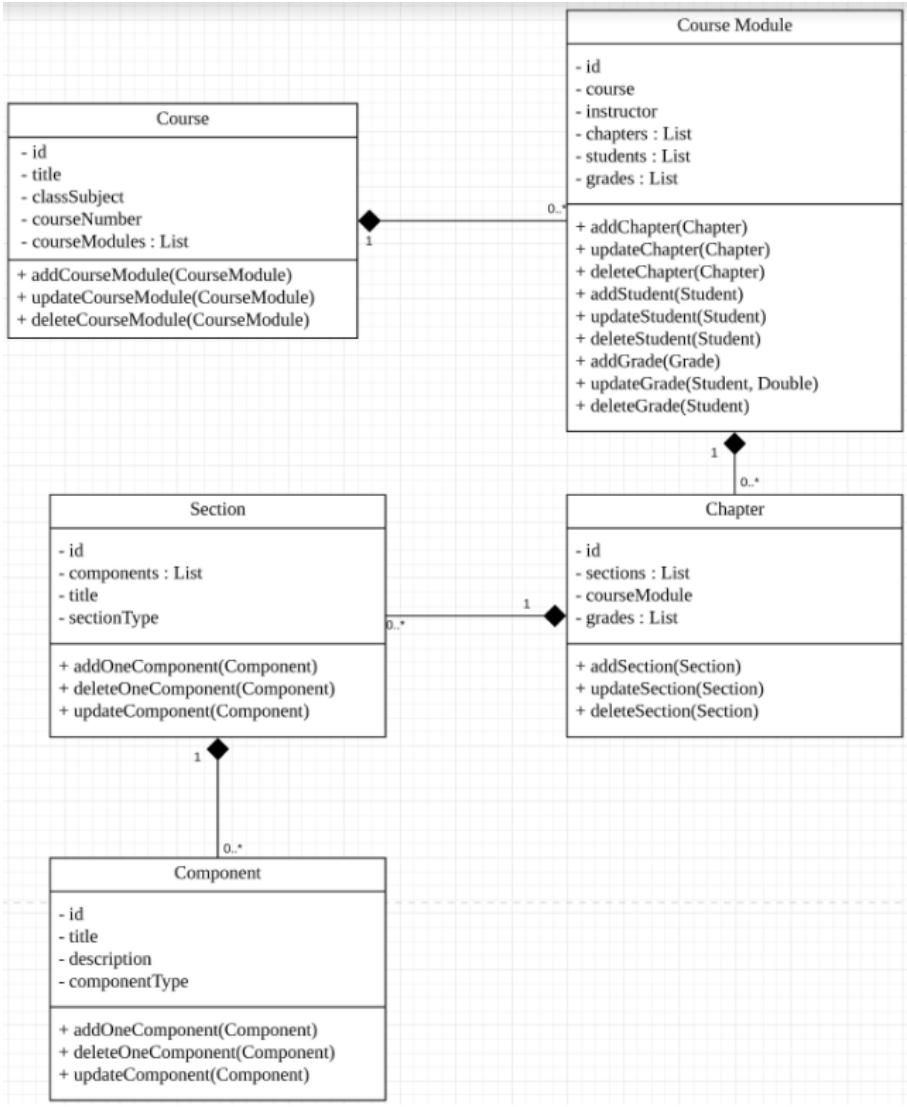


Figure 9. Objective-Oriented Design 1

a. Component Class:

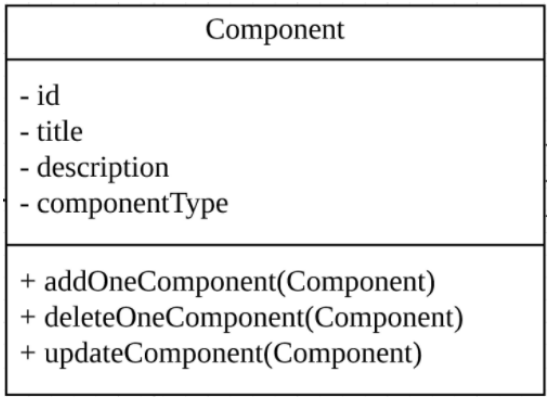


Figure 10. Component Class

i. A class holds unique id to refer a content, title of this content, description of this content, and *componentType* (an attribute to denote type of component).

ii. Perspective functions to add, delete, and update Component's attributes.

b. Section Class:

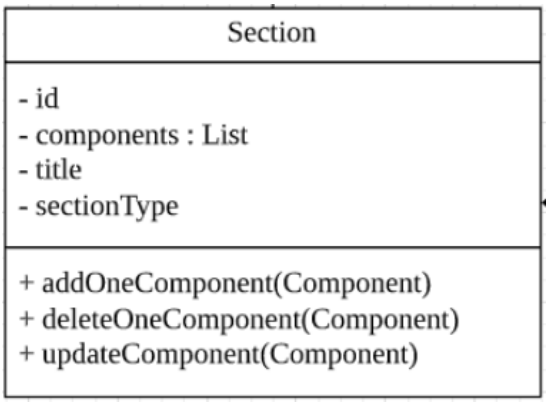


Figure 11. Component Class

i. A class holds unique id to refer a section, title of this section, a list of components, and *sectionType* (an attribute to denote type of section).

ii. Perspective functions to add, delete, and update the Components list.

c. Chapter Class:

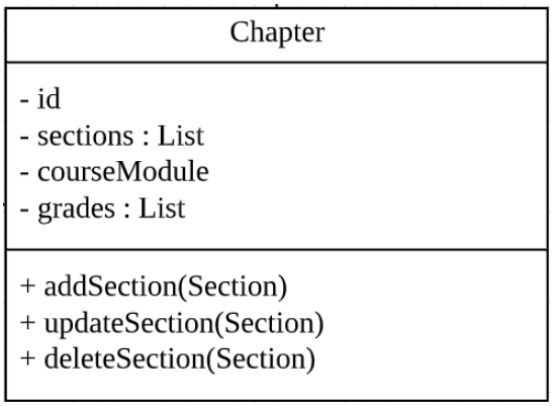


Figure 12. Chapter Class

i. A class holds unique id to refer a chapter, a list of Section class, a *courseModule* as which *courseModule* class that it is refer to, and a list of grade class.

ii. Perspective functions to add, delete, and update the Section list.

iii. Perspective functions to add, delete, and update the Grade list.

d. Course Module Class:

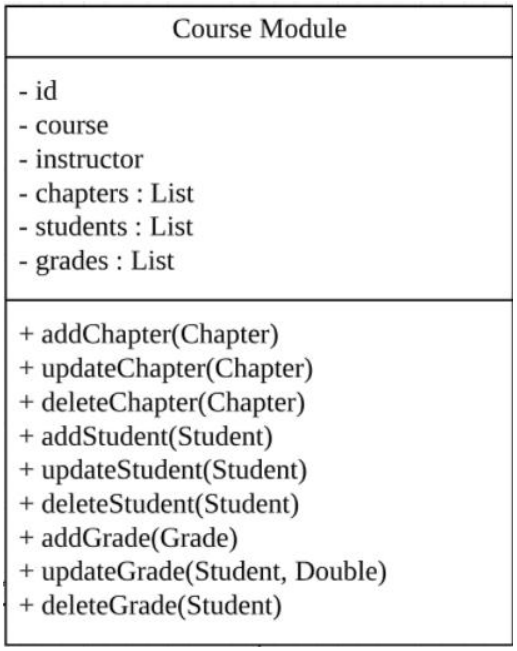


Figure 13. Course Module Class

A class holds unique id to refer a course module, a course as which course class that this class is referring to, instructor, a list of chapters, a list of students and a list of grades.

i. Perspective functions to add, delete, and update the Chapter list.

ii. Perspective functions to add, delete, and update the student list.

iii. Perspective functions to add, delete, and update the Grade list.

e. Course Class:

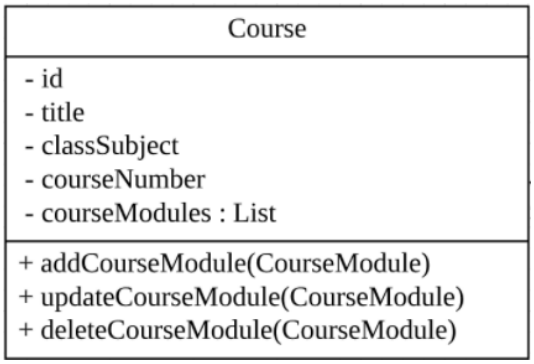


Figure 14. Course Class

i. Course class is designed to capture and simulate a real course information. With those requirements in mind, we need a class holds unique id to refer a class, its course title, course subject and number, and a list of Course Modules.

ii. Perspective functions to add, delete, and update the Course Modules list.

With this design, the HBU system can dynamically add, delete, and update any course related information.

2.5.2 User, student, instructor, and instructor classes

Based on prior business analysis, it is discovered that those three roles utilizes the same attributes such as first name, last name, email, username, password. Therefore, the OOP principles of inheritance – the ability of one object to acquire some/all properties of another object, and

polymorphism – a way to use a class exactly like its parent so there is no confusion with mixing types [3].

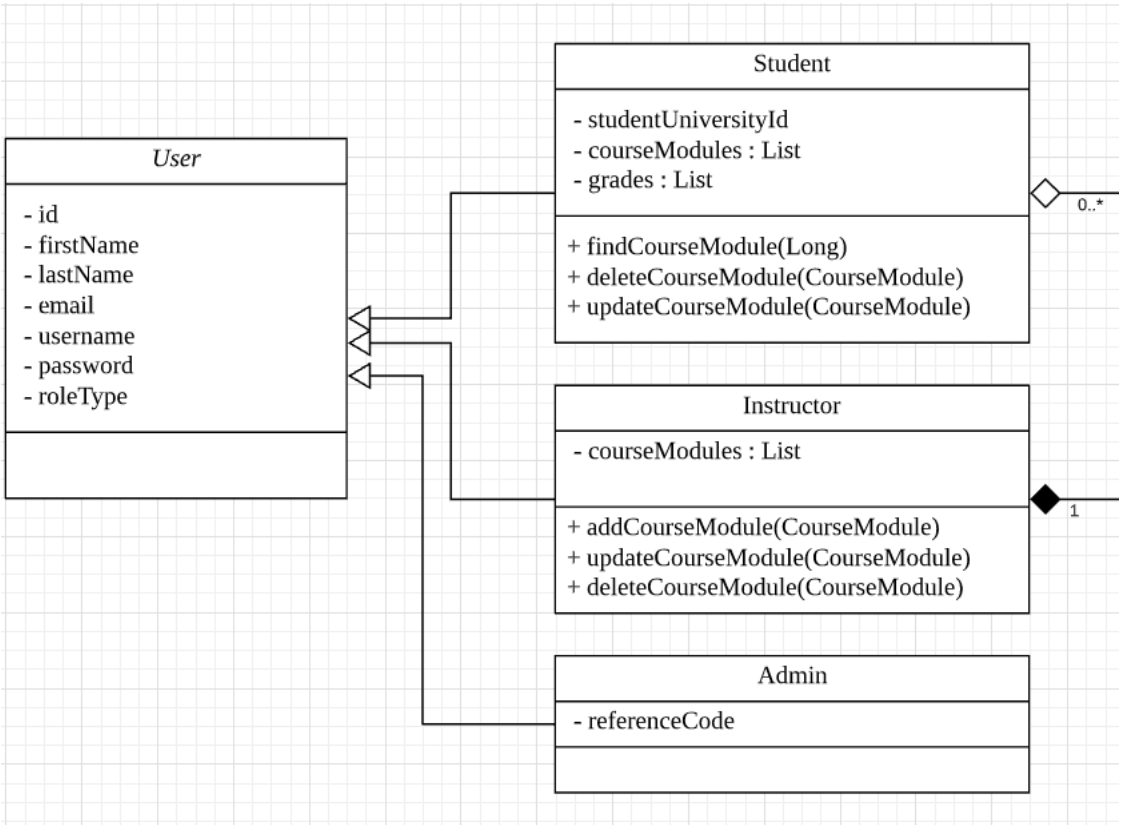


Figure 15. Objective-Oriented Design 2

a. User class: A super class with attributes of *firstName*, *lastName*, email, username, password and *roleType*. The attribute of *roleType* aids the possibility of identifying which roles it can cast into.

b. Student class: A child class of User class with attributes of *id*, *studentUniversityId*, a list of *coureModules* (it represents all the course that this student enrolled), and grades with perspective functions to find, delete and update the *courseModules* attribute.

c. Instructor class: A child class of User class with attributes of *studentUniversityId*, a list of *coureModules* (it represents all the course that this student enrolled), and grades with perspective functions to find, delete and update the *courseModules* attribute.

d. Admin class: A child class of User class only with attributes of reference code to refer this user as an Admin.

With appropriated system design, use case analysis, business analysis and UI design, database design and OOP design, one need to possess necessary knowledge of Java Spring Boot to fully implement HBU backend system. Therefore, the literature review of Spring Boot is introduced next.

3. SPRING BOOT FRAMEWORK

3.1 What is Spring Boot Framework and its history

Spring Boot is open-source web application framework written in Java. Spring Boot emphasizes the reusability and pluggability of components to achieve rapid development. As a result, using Spring Boot framework can help make the complex development faster and easier. Spring Boot Framework has a true and tested history in the software industry. It began in October 2002, when Rod Johnson wrote the book of "Expert One-one-One J2EE Design and Development". In Rod's book, he proposed POJO – plain old java objects and dependency injection. In February 2003, the name "Spring" for the new framework was established and made as an open-source project-based infrastructure code created by Rod. Spring is a framework that encompasses many useful java libraries for web development. It includes numerous essential web development like "components", such as database access, security, cloud deployment, web services and so on. Spring Framework rapidly grew since 2004 [4]. The main actions from Spring (core module) includes Inversion of Control (IoC), dependency injection, common annotations, and implements and provides the foundation for other frameworks, such as *Thymeleaf*, *MyBatis*, *Spring MVC*, etc.

Spring Boot, a part of the Spring framework, which is released in April 2014, helps in rapid application development. Compared to Spring, Spring Boot can get started with Spring's core libraries with less configuration setup, so that it helps with stand-alone and enterprise web application development. It can automatically configure Spring and third-party libraries and provides necessary dependencies to ease the build configurations. First, we will introduce Spring Boot's Configuration and Dependency management.

3.2 Spring Boot's Configuration and Dependency management

Spring uses a dependency management tool called Maven. Maven builds a project using its project object model (POM) and a set of plugins. It is an industry standard that allows development projects to declare dependencies on open-source libraries which are hosted on a

public central repository [5]. It is convenient for someone who needs to manage complex dependencies by generating a Maven configuration file with the dependency details. This saves time for navigating projects. More importantly, one should understand the fundamental concept and its basic for Spring Framework for web development.

3.3 Spring Framework Basic for Web Development

3.3.1 Issue

Most applications can be thought of as a collection of code components. Those components include business logic, which contains the high-level steps of application feature, and persistent objects that the project should be initialized and maintained with a database. It becomes harder and more error-prone with connecting the components to each other. Spring Boot solves dependency issues by Appling Inversion of Control (IoC), which will be covered in the following section.

3.3.2 Inversion of Control (IoC)

Inversion of Control is a design pattern that inverts the flow of control in application components in order to define a framework, and it defines a way for the framework to include dependencies between its components. IoC is also known as dependency injection [6]. In the process of running the framework, the IoC Framework automatically initializes the components and connects them, and procedurally determines a logical order and organization along with the dependencies of the components, which can solve the knowing issue mentioned in 3.3.1. When Spring Boot builds its components with their class files, the framework also configures their dependencies. When the application is running, Spring ensures that all components are created in a compatible order and injects the dependencies in the components that declared them.

3.3.3 How Spring configures its IoC?

When Spring's configuration process starts, it will scan configuration files and annotations. It then initializes and stores the components in the application context. A collection similar to a map or a set is used so that Spring can cache, retrieve, and manage components in an

application. In this process, application contexts will receive queries to find instances which are called beans by Spring according to the criteria of name, supertype, and scope of a bean [7]. Since Spring is a closed system, beans in the application context are only aware of other beans in the same context. Spring instantiates components only because other Spring components depend on them.

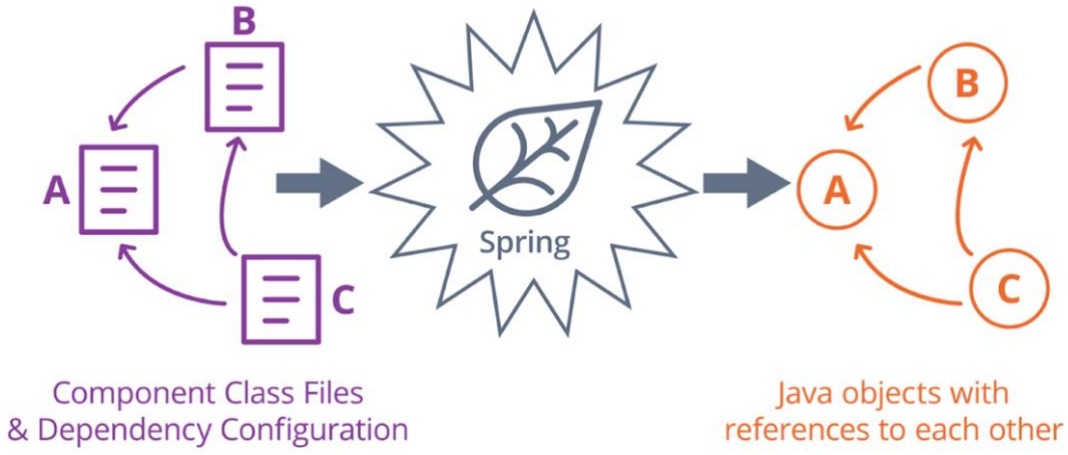


Figure 16. Spring IoC Configuration

3.3.4 Spring Framework's Advantage

With Spring and IoC, developers can use application-wide constants, such as service URLs and driver class named as dependencies. We can define entire layers of business logic methods as components that depend on resources like database drivers that can be changed with a simple configuration file edit, so that we can focus our development efforts on features and business logic. Ultimately, the use of an IoC framework like Spring focuses on dividing the application's requirement into single-purpose components.

3.4 Spring Framework's application control flow pattern: Model View Controller (MVC)

Model View Comptroller (MVC) design pattern consists of a data model, presentation information and control information. The design pattern requires that operations on data should transfer and manipulate in different layers. The definition of controller, model and view are listed as following.

3.4.1 Model

Model contains only pure data with our any business logic for how to present the data to users.

3.4.2 View

The View present the model's data to the user who sent the request. The view layer can access the data from model layer, but it doesn't know the meaning of the data and methods to make changes.

3.4.3 Controller

The controller is utilized to process user requests from the View to update the Model layer and forwarding those updates back to View. This reaction is to call a method on the model [8].

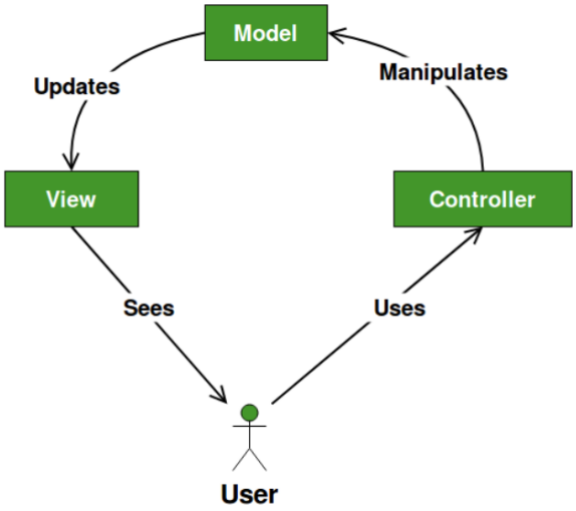


Figure 17. Model View Controller [8]

All the three layers constitutes a mechanism to reflect the online instructions and show it to the users. In the real-life, the project using MVC is triggered because a user sends a HTTP request for changing some data, as shown in *figure 17*. The request will be sent to controller layer, and it will reflect to manipulate the model layer for persisting or fetching in the database.

After that, the view updates the data that is provided from Model. The more details of MVC responsibility will be listed in next section.

3.5 How Spring Boot use MVC?

3.5.1 View

In Spring Boot, the View is responsible for displaying the UI to the user. In the Spring Boot framework, HTML templates are the views - each one represents a specific screen or screen components that the user is shown.

3.5.2 Model

Spring Boot's Model is responsible for maintaining the state of an application. Model objects are the models - every controller method can take an optional Model argument, and by reading and changing the data inside of it, the controller can read user-submitted data and populate the template with the changes. Think of the Model class as a simple data-transfer object: something that can store various bits of data with keys to look that data up, and that can be passed between the browser, the template engine, and the controller to facilitate the transfer of data between the user and the application.

3.5.3 Controller

In Spring Boot, the controller is responsible for processing user requests from the View to update the Model layer and forwarding those updates back to View. In the Spring Boot framework, controllers are specified by @Controller annotation, so we can use to register our beans as controllers. Think of Spring bean controllers as specialized application components that can define methods to handle specific user requests. Those methods are responsible for choosing the HTML template that is generated in response, as well as for populating the Model object for that template.

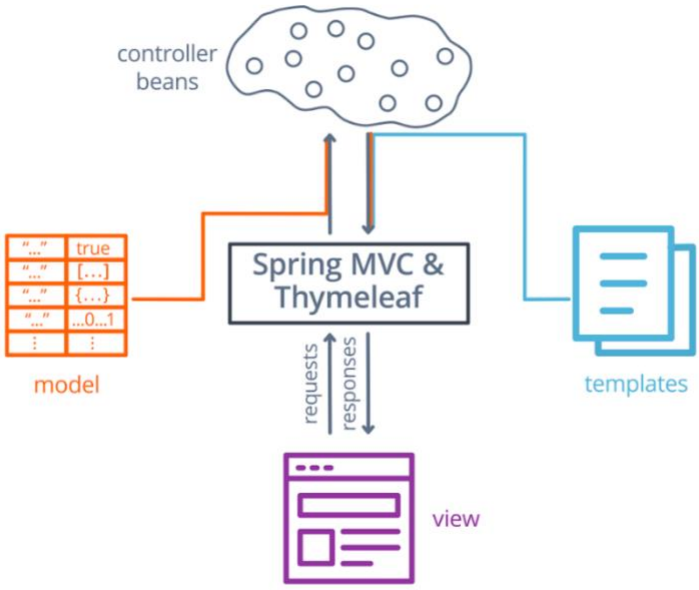


Figure 18. Spring MVC and Thymeleaf

Figure 18 above shown the Spring MVC architecture. The view in the graph represents the browser that the user interacts with. When the user sends a request from the browser, Spring MVC processes the request and creates a model object. The controller updates the model chooses a template to render the template in the response and finally sends the response to the browser.

3.6 Spring Boot's Support for Data Stores and Persistence

As we introduced in section 3.4, Spring Boot Framework uses MVC as a design pattern to send the data between frontend and backend. Spring Boot allows developer to utilized two design patterns to persistence data: Data Access Object pattern (DAO) and Repository pattern, which will be introduced in the following:

3.6.1 Data Access Object (DAO) Pattern

DAO is an abstract definition of data persistence as the format of table-centric and closer to underly storage. Without applying DAO, it is necessary to create database query, a request to access data from a database to manipulate data or retrieve data [9]. Sending or retrieving data from storage which can avoid using the database queries in Spring Boot [10].

3.6.2 Repository Pattern

Repository is a mechanism for encapsulating storage, retrieval, and search behavior, which emulates a collection of objects [11]. Once received such encapsulated information, the repository will try to persist the information to a database. It "mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects [12]. Repository is a layer that implements the DAO layers and interacts with data from the Spring Boot framework and the next layer is the database. It is the highest layer before business logic, which is also called service layer in Spring Boot. Consequently, a repository can use a DAO to fetch data from the database and populate a domain object. Or it can prepare the data from a domain object and send it to a storage system using a DAO for persistence [10]. Regardless which patterns that one developed, Spring Boot provides Java Persistence API (JPA) to accomplish one's goal.

3.6.3 Java Persistence API (JPA)

Besides using JDBC and database queries with it, we can use a Java Persistence API (JPA), a specification describing how to manage relational data, to manage backend databases. *JPARepositroty* has default CRUD (Create, Read, Update, Delete) operators:

- *Optional<T> findById(ID primaryKey);*
- *Iterable<T> findAll ().*
- *<S extends T> S save (S entity).*
- *<S extends T> List<S> saveAll (Iterable<S> entities).*
- *void delete (T entity).*
- *void delete All (); [11]*

3.6.4 Java Persistence API (JPA) With Repository Pattern implementation

In the case of applying repository pattern, if a repository extends the *JPARepository* with its entity and id type, the *JPARepository* will help us to generate queries, connect to databases,

and make changes or retrieve on the databases. The query is generated by Hibernate, which is an implementation of the JPA Specification. If an *EmployeeRepository* is needed to do operations, we can just call the CRUD methods in the *EmployeeService* and *EmployeeController*.

The below is a code example of such approach:

Repository:

```
16  public interface EmployeeRepository extends JpaRepository<Employee, Long> {
17      List<Employee> getAllByDaysAvailableContains(DayOfWeek dayOfWeek);
18  }
```

Service:

```
24      public Employee save(Employee employee){
25          return employeeRepository.save(employee);
26      }
```

Controller:

```
80      @PostMapping("/employee")
81      public EmployeeDTO saveEmployee(@RequestBody EmployeeDTO employeeDTO) {
82          Employee employee = DtoUtility.convertToEmployee(employeeDTO);
83          return DtoUtility.convertToEmployeeDTO(empService.saveEmployee(employee));
84      }
```

Figure 19. JPA Example

We don't need to implement any CRUD method in the *EmployeeRepository* in this example. The save method is called in save method to trigger the save or update SQL query. Hibernate in here will help us to generate database query (*figure 20*).

```
Hibernate:
    insert
    into
        employee
        (name)
    values
        (?)
Hibernate:
    insert
    into
        employee_skills
        (employee_id, skills)
    values
        (?, ?)
```

Figure 20. Hibernate Output

JPA also provides Derived Query Methods to create a query for other features. The *getAllByDaysAvailableContains* method in the *EmployeeRepository* is an example of derived query methods to get all the employees in the company that are available on the given date. The Hibernate query result is shown in *Figure 21*.

```
Hibernate:
    select
        employee0_.id as id1_2_,
        employee0_.name as name2_2_
    from
        employee employee0_
    where
        ? in (
            select
                daysavaila1_.days_available
            from
                employee_days_available daysavaila1_
            where
                employee0_.id=daysavaila1_.employee_id
        )
```

Figure 21. Hibernate Result with Special Methods

3.6.5 Java Persistence API (JPA) With DAO Pattern implementation

An alternative way for repository pattern is Data Access Object Objects (DAO), A programmatic interface to a table or collection of related tables. DAO can present a collection of entities that reflects databases. It can provide various functional methods, along with CRUD operations. To achieve updates or retrieve from the database, we need to use JDBC Template to provide connection, execute queries and manage transitions between repository and database. It has a format that we need to follow for creating a query. *Select * from employee where id =? and age >=?* for instance, can find all the employees who have age greater than a certain value, and the id equals a specific employee id. The question mark is waiting for a value that is passed in later when we apply it. We use it when we set the query to a string for example called *SELECT_EMPLOYEE_BY_ID* and use the string by the code shown in *figure 22*.

```
124       public EmployeeData getEmployeeById(Long Id){
125           jdbcTemplate.queryForObject(
126                   SELECT_EMPLOYEE_BY_ID,
127                   New MapSqlParameterSource()
128                   .addValue("id", Id),
129                   new BeanPropertyRowMapper<>(EmployeeData.class)
130            );
131       }
```

Figure 22. getEmployeeById method

The *MapSqlParameterSource* is the method for mapping parameter names to parameter values. The *BeanPropertyRowMapper* returns the instance of the object for which we're querying. Therefore, the object is created for database connection, updating, and retrieving in this process. While the Repository pattern is common with Hibernate because most modifications can be done simply by editing Entities, there is nothing that prevents us from using the DAO pattern in either a Hibernate or SQL-based persistence model. In the example of *Figure 20*, we can do both JPA and DAO, we create a DAO JDBC template and send it to *JPARepository*, such that the DAO object can use Hibernate as well.

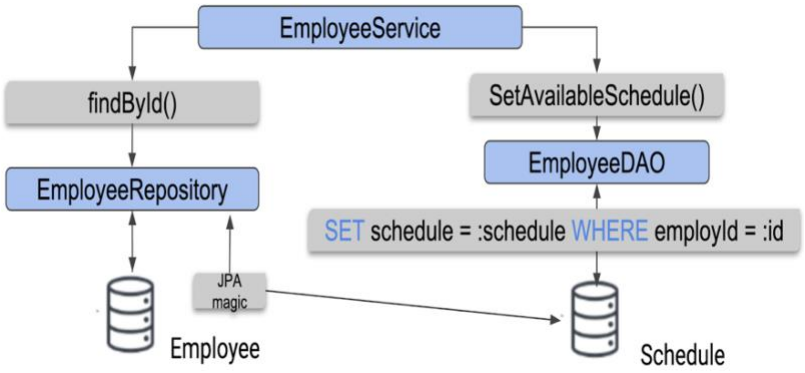


Figure 23. DAO vs. JPA

3.6.6 Spring Boot's JDBC vs. JPA approaches

Since JDBC makes a request to a database by a given query provided by a developer. The relationships between the entities are built in the database when we use JDBC. It is running faster than JPA because JPA needs to load the whole database when we run some of the queries from Hibernate. However, it is easy to make a mistake when we write a database query. With incorrect

query, it is hard to debug during the development. The developer must know the database query well with JDBC. It is convenient for programmers to access databases by using JPA without queries, such as MySQL, NoSQL, or MongoDB, since the database query is generated from default methods and other Java methods with Camel Case name convention. In addition, the table relationships between entities are built in Java. However, loading the data from the database by JPA is slower than JDBC, because the whole database will be loaded when we only retrieve a partial of data.

3.7 How HBU Backend system can utilize Spring Boot?

HBU backend system need a standard communication method that allows its various platform of fronted applications to request and receive data from the from the backend application, which Spring Boot framework provides support for building Application Programming Interface (API). What is this API in question? How does it going to help HBU system? It will be covered in the following:

3.7.1 What is Application Programming Interface (API)?

Application Programming Interface (API) communicate with other open-source application's data and functionality to extrinsic web services. Developers who want to use the extrinsic web service do not need to know how it works or how it is implemented. By using such interface to communicate with others, it reduced workload of developing. API was popular in the past decades, so that most of web applications would not be without APIs nowadays.

3.7.2 How does an API works

1) initial a request: a user can make an API call view application. The API will retrieve the information, which also called request, and break up the information to a request verb, headers, and a request body. Before API process the request, it is necessary to check if the request is valid.
2) the API will send the request to the external program or web service.
3) the web server makes a response to the API for the request.

4) The API transfer the data to the initial requesting end [13].

3.7.3 Representational State Transfer API (RESTful API)

REST (Representational State Transfer) or RESTful API is designed for sharing data between two or more systems has always been a fundamental requirement of software development [19]. Spring Boot is a very good tool to build a REST API. Restful API can help developers to transform information between servers and clients. By creating an API in our project, we can use the API to post and get data from the database of a project. It is not necessary to bind the view with the model and controller. For example, in *figure 24*, to develop the version of the desktop and mobile app for business requirements, we can just build one API to share the same data shared and change the data in the backend. We can only change the annotation @Controller to @RestController to make the controller utilize the Rest API, instead of a view of the web page. With all the fundamental knowledge of our weapon – Spring Boot Framework, the implementation plan is in place to be proceed in the coming section.

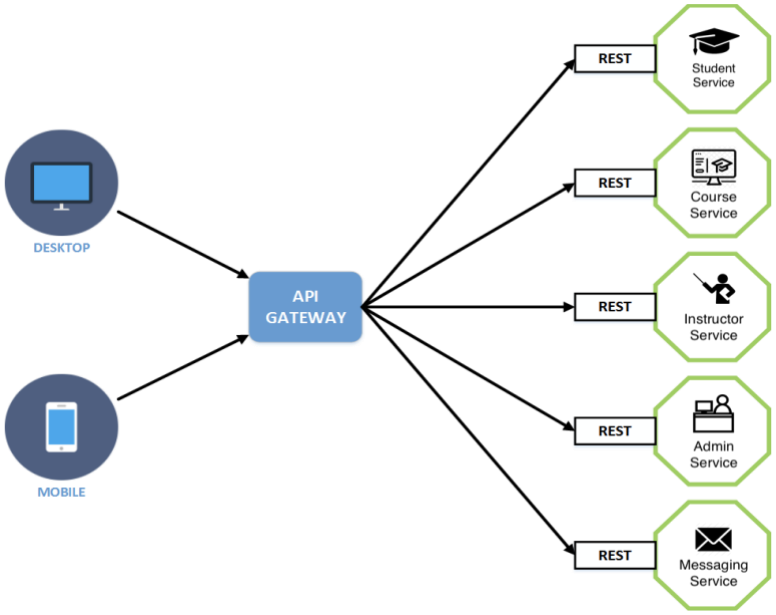


Figure 24. Online Education Management Design Pattern

4. HBU BACKEND APPLICATION IMPLMENTATION

4.1 An Overview of HBU Ecosystem

HBU is an online interactive educational system. The users are students, educators, and institutions. With the system, institutions can set up their courses, add instructors and register for students. Students can access their enrolled courses and the course materials. Moreover, the enrolled students can do their assignments, exams, and quizzes submission. The system also allows the students to chat with the instructor if they have questions. Instructors can create courses and add course materials including announcements, lecture videos, assignments, and exams. The project released 70% of administrative workloads, and reduced the error rate during course, students, and instructors' management.

4.2 HBU Spring Boot Implementation

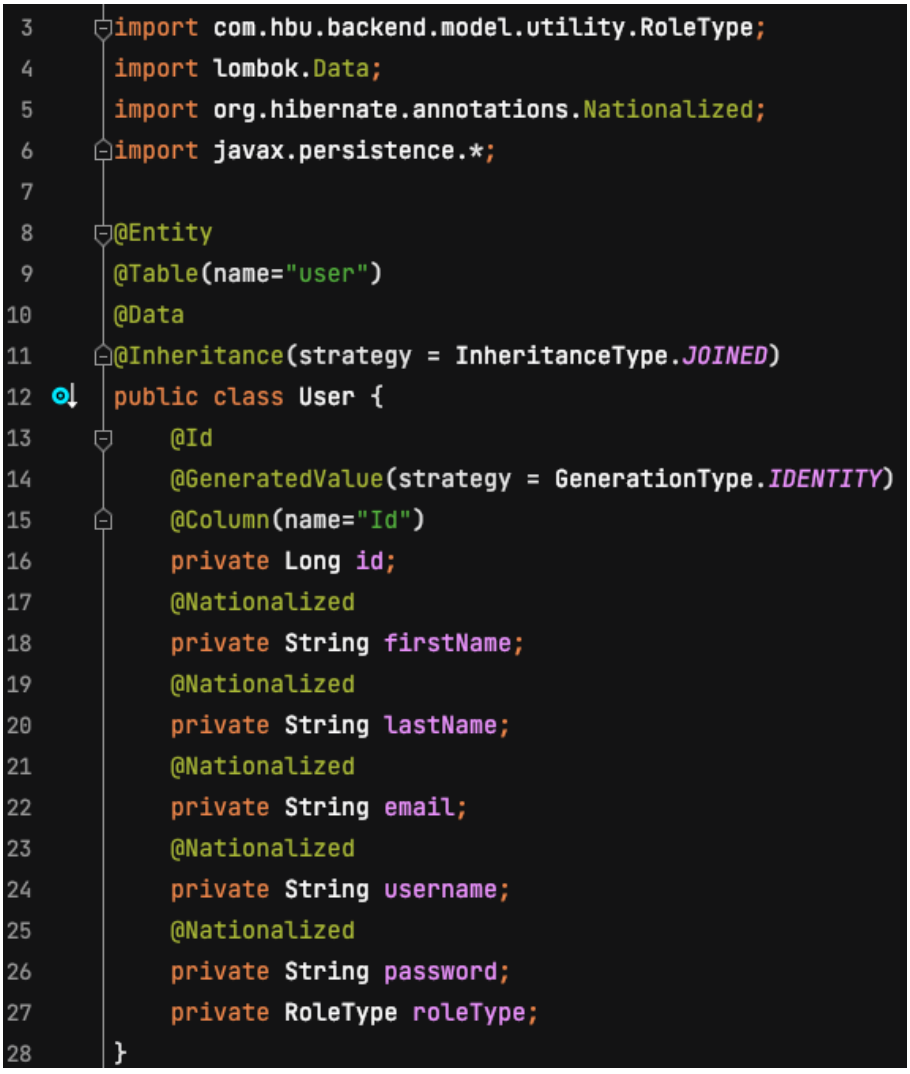
With the existing OOP and database design in completion, the next step for building this system in Spring Boot are creating the following components in sequence to deploy. First, implementing the Entities?

4.2.1 Entity

An entity, a lightweight persistence domain object, represents a row of table in a relational database. Entity is mainly applied for an object synchronously build a table in database [14]. To map and persist the table and the object in Spring Boot framework, we need to implement with the following steps as shown in *Figure 25*:

a. A class that is required to persist with a table, should be added with *javax.persistence.Entity* annotation like line 8 in *Figure 25*.
b. The class need to create public, protected, or private numbers, like line 13 to line 27.
c. If an entity possesses passed valued as a detached object, the class should implement as a serializable interface.

d. Some private, protected or package private attribute must be created if the variable is a persistent instance object. Other clients can only access the attributes through getter and setter methods.



```
3   import com.hbu.backend.model.utility.RoleType;
4   import lombok.Data;
5   import org.hibernate.annotations.Nationalized;
6   import javax.persistence.*;
7
8   @Entity
9   @Table(name="user")
10  @Data
11  @Inheritance(strategy = InheritanceType.JOINED)
12  public class User {
13      @Id
14      @GeneratedValue(strategy = GenerationType.IDENTITY)
15      @Column(name="Id")
16      private Long id;
17      @Nationalized
18      private String firstName;
19      @Nationalized
20      private String lastName;
21      @Nationalized
22      private String email;
23      @Nationalized
24      private String username;
25      @Nationalized
26      private String password;
27      private RoleType roleType;
28  }
```

Figure 25. User Entity

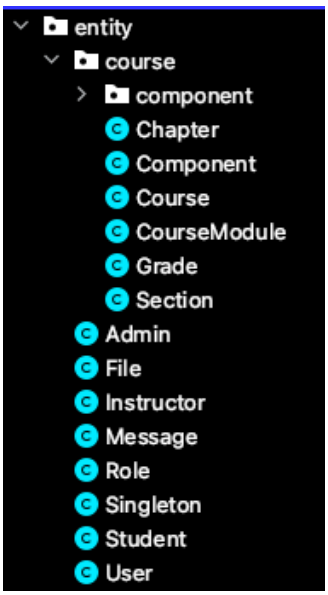


Figure 26. All Entities in HBU

All the entities in *Figure 26* are in the model layer for creating, persistence and fetching data. Following with database design, nine classes were created. Those classes are Course, Student, Instructor, Admin, Course Module, Chapter, Section and Component classes for corresponding to entities in database. Via business logic attributes and methods were created in the class. The attribute will the tables in the database and build relationship in the system in the next step. Using *@Entity* annotation can help the project to do persistent automatically, so that the table creation in the database using hibernate when the project is running. If the database does not exist, and update changes if anything is changed in the class. The user class has id, first name, last name, email address, username, password, and role type attributes shown in *figure 22*. The id is the primary key of each user as the identity or foreign key in other entities. We used *@Id* to indicate mark it is the key to this entity [15] and *@GeneratedValue (strategy = GenerationType.IDENTITY)* to auto generate to database based on the last id in this entity [16]. Every time when an entity is added, the id will be generated by increasing one from the last id of the entity. *@Nataionalized annotation* "marks a character data type (String, Character) as being a nationalized variant. As explained before, *@Entity* finally will persist or update all the attributes to the fields in user table in the database.

The student and instructor are subclasses of user class. The user class can share its private attributes to student class and instructor class because the principle of inheritance in java. The user class will build the common feature, and the student and instructor class still have some unique attributes that we should create in its own class.

```java
@Entity
@Table(name="student")
public class Student extends User {

    @Nationalized
    private String studentUniversityId;

    @ManyToMany(fetch = FetchType.LAZY,
        cascade = {
                CascadeType.PERSIST,
                CascadeType.MERGE
        })
    @JoinTable(name = "student_course_module",
            joinColumns = { @JoinColumn(name = "student_id") },
            inverseJoinColumns = { @JoinColumn(name = "courseModule_id") })
    private List<CourseModule> courseModules;

    @OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
    List<Grade> grades;
```

Figure 27. Student Entity

For *FetchType* [17] in *figure 27*, with the entity added to existing designed class, the HBU system is ready to connect to database, which lead to the implementation of repository. As mentioned before, there are two implementations of connecting to database via Spring Boot: DAO and Repository patterns. Repository pattern was preferred for this HUB system after careful consideration; it also means the need for implement the Service Class to wrap repository to make single responsible class.

4.2.2 Repository, Service and Controller

As mentioned in section 3.4, we have introduced the working principle of Spring MVC. However, there are more details between Model, View and Controller as shown in *figure 28*.

When users send request from view, the request will be broken down to the different layers – 1) View 2) Controller 3) Service 4) Repository 5) Database. In the next sections, we will elaborate on the functionality of those layers.

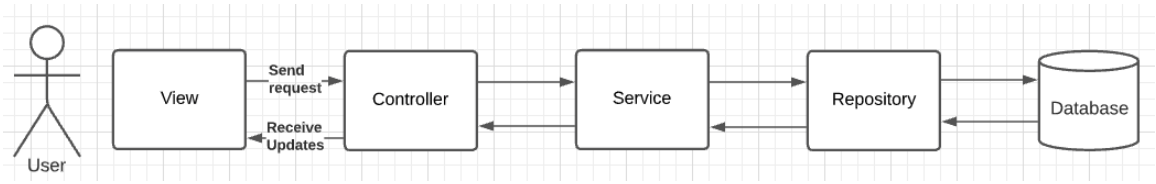


Figure 28. Repository, Service and Controller

4.2.3 Controller Class

As we mentioned in section 3.6.2, Controller is the layer for receiving and disposing incoming user requests. In our project, there are eleven controllers in *figure 29*, but only nine of them are the core controllers with the nine core classes. Each of the controllers are followed with their entities. After receiving a user's requirement, the controller will catch that by the annotation *@RequestMapping* and *@PostMapping* with it specific path. For example, in line 115 in *figure 30*, the *saveStrudent* method in line 116 will be called if the user request is [host URL]/admin/student, because, in our admin controller. We used *@RequestMapping("/admin")* and *@PostMapping("/student")*. As the line 122 and line133 displayed, the *studentService* is applied to transfer the request with a business logic, once we ensure that the data passed in with the user request is valid. In the next section, we will do deep dive with service layer to analysis the business logic.

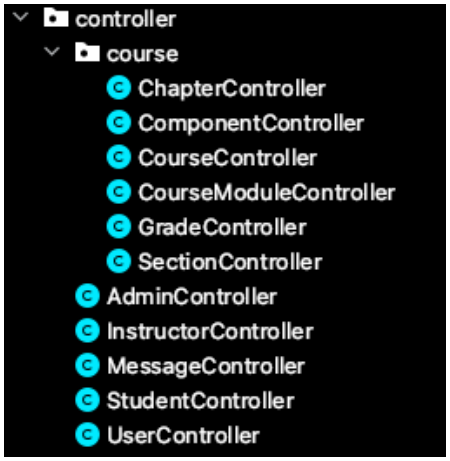


Figure 29. All Controllers

```
114     /** for students */
115     @PostMapping("/student")
116     public ResponseEntity<StudentDTO> saveStudent(@RequestBody StudentDTO studentDTO){
117         Student student = com.hbu.backend.model.utility.DtoUtility.toStudent(studentDTO);
118
119         List<CourseModule> courseModules = new ArrayList<>();
120         if(studentDTO.getCourseModuleIds() != null){
121             for(Long courseModuleId : studentDTO.getCourseModuleIds()){
122                 CourseModule courseModule = courseModuleService.findCourseModule(courseModuleId);
123
124                 if(courseModule == null){
125                     continue;
126                 }
127                 courseModules.add(courseModule);
128             }
129         }
130
131         student.setCourseModules(courseModules);
132
133         Student newStudent = studentService.addStudent(student);
134         for(CourseModule courseModule : courseModules){
135             courseModule.addStudent(newStudent);
136             courseModuleService.saveCourseModule(courseModule);
137         }
138         return new ResponseEntity<StudentDTO>(com.hbu.backend.model.utility.DtoUtility.toStudentDTO(newStudent), HttpStatus.OK);
139     }
```

Figure 30. saveStrudent Implementation for Admin

4.2.4 Service Class

In service layer, we usually implement the add, delete, find, find all, update that corresponds to CRUD operations default methods with JPA repository. Multiple repositories, sometimes, involve in a service for gathering the entities fields or updating related entities changes. For example, in the student service in *figure 28*, each of the method for a student, it needs to call student repository for reflecting the action in database. For instance, in line 18 in *figure 31*, the *addStudent* method can manipulate the database by sending the student object to the

save method in *studentRepository*. At the same time, it provides the business logic to manage entities to avail some of the non-existing cases. For example, the *deleteStudent* method shown line 22 (*figure 31*), we should check if the user we plan to delete is in the database.

```
11  @Service
12  @Transactional
13  public class StudentService {
14      @Autowired
15      StudentRepository studentRepository;
16
17      /** create a student */
18      public Student addStudent(Student student) { return studentRepository.save(student); }
21      /** delete a student */
22      public void deleteStudent(Student student) {
23          if(student != null) {
24              studentRepository.delete(student);
25          }
26      }
27      /** read one */
28      public Student findStudent(Long id) { return studentRepository.findById(id).orElse( other: null); }
31      /** read all */
32      public List<Student> findAllStudents() { return studentRepository.findAll(); }
35      /**
36       * update a student
37       */
38      public Student updateStudent(Student student, Long id) {
39          Student foundStudent = studentRepository.findById(id).orElse( other: null);
40
41          if (foundStudent == null) {
42              return null;
43          }
44          foundStudent.setStudentUniversityId(student.getStudentUniversityId());
45          foundStudent.setFirstName(student.getFirstName());
46          foundStudent.setLastName(student.getLastName());
47          foundStudent.setEmail(student.getEmail());
48          foundStudent.setUsername(student.getUsername());
49          foundStudent.setPassword(student.getPassword());
50          foundStudent.setRoleType(student.getRoleType());
51          foundStudent.setCourseModules(student.getCourseModules());
52          return studentRepository.save(foundStudent);
53      }
54  }
```

Figure 31. Implementation of Student Service

4.2.5 Repository Layer

As we discussed in section 3.6.2, the repository has responsibility to persist the data from Spring Boot framework to database. As the *figure 32* shown, the data will be persisted when the base methods are called in service which we will mention in previous section. Following with our

example, the *adminRepository* doesn't have any implemented function. However, JPA will help the admin repository because admin repository extends the *JPARepository*. JPA can recognize with its corresponding entity because of the admin type we mentioned at the end of line 6.

```
6   public interface AdminRepository extends JpaRepository<Admin, Long> {
7   }
```

Figure 32. Implementation of Admin Repository

All the repository classes are listed below in *figure 33*, those repository classes accurately matched with the entities in *figure 26*. When one type of data is updated in database base on a request, the repository will be an interface to send queries to database. Every entity is matched to it repository, as a result, it is updated by its own query.

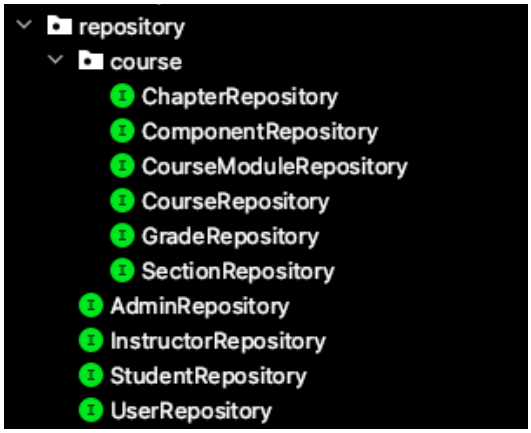


Figure 33. All Repository of HBU

4.2.6 Example of one request

We would use an example of adding a student by admin to simulate the flow when the action is triggered. The sequence diagram in *figure 34* displayed the flow for how an admin adds a student. First, when an admin checks the manage page and click on an add student button. The use will see a new student profile page. The admin can fill the new student's information and hit the submit button. This event will trigger out API and send the user post request to *saveStudent* method in *figure 30* and pass the model or entity to admin service and student service. After checking the business logic, the entity is sent to the admin and student repository layer for

creating database query. The query will act on the database, so the database is changed. Database will response a result from the query to indicate if the act works or not. At same time, the response will be shown in the student profile page to the user.

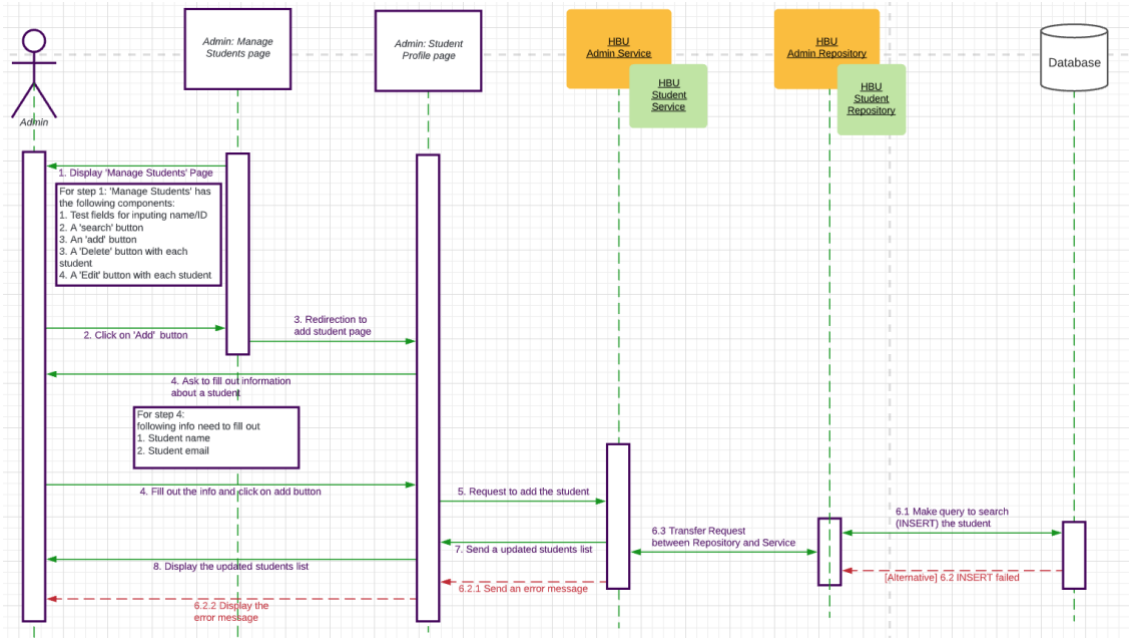


Figure 34. Sequence Diagram of Adding Student by Admin

What we have shown in the last part is the whole process for a user request. APIs receives requests and return response followed by the similar sequence processes like *figure 34*. Next, we would try to run the project and analysis the input and outputs we can get.

4.3 Endpoints

API endpoint is the end of communication channel. It interacted with other system or API and formed as URL of a server or service. For example, we created 73 endpoints and categorized them our entities (*figure 35*), such that we can check if the endpoints work correctly for the controllers. We put the endpoints by utilizing postman, an API platform for building and using APIs [18].

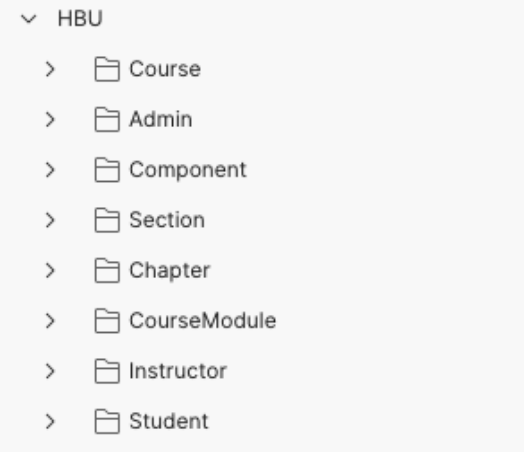


Figure 35. Categories of Endpoints

Figure 36 shows one of the instances of the post type request (*http://localhost:8082/admin/student*). The json format input, in the upper box in *figure 36* is the student object that an admin adds. If the API get the request, it will send back to postman and show result in the lower output box with the status 200 OK to indicate no-error during this progress. It is obvious that some new attributes are added in the object. The id is autogenerated by database since *@Id* and *@GeneratedValue (strategy = GenerationType.IDENTITY)* is applied. Our project is tested with all aspects in a high coverage. Consequentially, our API is fully designed, implemented, and tested. We will summarize and analysis the outputs and result in the next section.

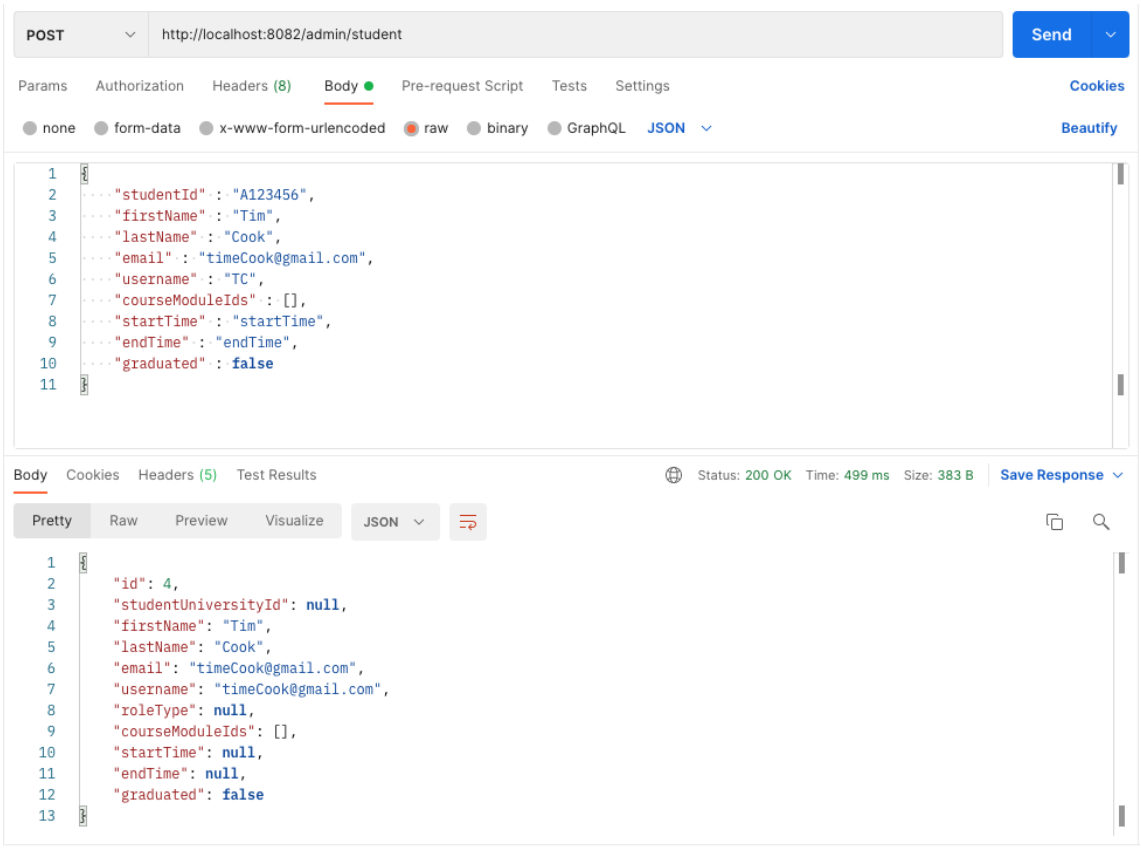


Figure 36. Test with Post Student on Postman

5. CONCLUSION

The Spring Boot framework gives us a fast and reliable way to create the online education system. It provides powerful functionalities and concise syntax to help programmers communicate with databases. Even though HBU is not deployed to the public, it is well-designed for an online education that facilitates a remote curriculum for students, educators, and institutions. Our project is easily adaptable for flexibility and reusability because of database design and object-oriented design (OOD). The entities do not couple with each other, and the relationships between them are manageable. The Spring Model View Controller pattern emphasizes a cohesive way for a user to interact with an endpoint, while separating the business logic interaction between users, databases, and web servers into different layers. The Spring Boot Framework offers reusability by providing an efficient pattern that connects to endpoints without needing developers to recreate boilerplate code. As a result, it is easy to add, delete, update and search data (CRUD operations) in the database compared to not using the Spring Boot framework at all. Endpoints using CRUD operations can manipulate persistent data in a database without having developers reinvent the wheel by having them implement fundamental methods. It will be helpful for developers to make more complex systems by adding more features later without modify the original business logic. In conclusion, the completion of our project provides appropriately designed infrastructures and is implemented for our users' remote online educational curriculum. This project represents an ongoing effort to provide a service for students and instructors online and it serves as an integrated platform that is subject to modification for future development.

6. REFERENCES

[1] "What is full stack development?" Geeksforgeeks.org, 26-Jun-2019. [Online]. Available: https://www.geeksforgeeks.org/what-is-full-stack-development/. [Accessed: 20-Nov-2021].

[2] Java.com. [Online]. Available: https://www.java.com/en/download/help/whatis_java.html. [Accessed: 29-Oct-2021].

[3] F. Banda, "The four pillars of object-oriented programming," Keylimeinteractive.com. [Online]. Available: https://info.keylimeinteractive.com/the-four-pillars-of-object-oriented-programming. [Accessed: 20-Nov-2021].

[4] "History of spring framework and spring boot," Quickprogrammingtips.com. [Online]. Available: https://www.quickprogrammingtips.com/spring-boot/history-of-spring-framework-and-spring-boot.html. [Accessed: 20-Nov-2021].

[5] "Maven – Introduction," Apache.org. [Online]. Available: https://maven.apache.org/what-is-maven.html. [Accessed: 20-Nov-2021].

[6] "5. The IoC container," Spring.io. [Online]. Available: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/beans.html. [Accessed: 20-Nov-2021].

[7] B. N. Nandan, "Component scanning with Spring Boot," Reflectoring.io, 23-Sep-2020. [Online]. Available: https://reflectoring.io/spring-component-scanning/. [Accessed: 21-Nov-2021].

[8] "MVC Design Pattern," Geeksforgeeks.org, 18-Aug-2017. [Online]. Available: https://www.geeksforgeeks.org/mvc-design-pattern/. [Accessed: 20-Nov-2021].

[9] "What is a database query? SQL and NoSQL queries explained," Educative.io. [Online]. Available: https://www.educative.io/blog/what-is-database-query-sql-nosql. [Accessed: 21-Nov-2021].

[10] Baeldung.com. [Online]. Available: https://www.baeldung.com/. [Accessed: 20-Nov-2021].

[11] “Domain-driven design: Tackling complexity in the heart of software,” Pearson.com. [Online]. Available: https://www.pearson.com/store/p/domain-driven-design-tackling-complexity-in-the-heart-of-software/P100000775942/9780321125217. [Accessed: 21-Nov-2021].

[12] M. Fowler, Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2012.

[13] IBM Cloud Education, “What is an Application Programming Interface (API),” Ibm.com. [Online]. Available: https://www.ibm.com/cloud/learn/api. [Accessed: 23-Nov-2021].

[14] “Entities - the java EE 6 tutorial,” Oracle.com, 01-Jan-2013. [Online]. Available: https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html. [Accessed: 23-Nov-2021]

[15] J. Bodnar, “Spring Boot findById,” Zetcode.com. [Online]. Available: https://zetcode.com/springboot/findbyid/. [Accessed: 20-Nov-2021].

[16] “Nationalized (Hibernate JavaDocs),” Jboss.org. [Online]. Available: https://docs.jboss.org/hibernate/orm/5.4/javadocs/org/hibernate/annotations/Nationalized.html. [Accessed: 20-Nov-2021].

[17] “FetchType (Java EE 6),” Oracle.com, 10-Feb-2011. [Online]. Available: https://docs.oracle.com/javaee/6/api/javax/persistence/FetchType.html. [Accessed: 20-Nov-2021].

[18] “Postman,” Postman.com. [Online]. Available: https://www.postman.com/. [Accessed: 28-Nov-2021].

[19] Redhat.com. [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. [Accessed: 30-Nov-2021].