
Bilkent University

Department of Computer Engineering

# CS319 Term Project

*ERMAN*

## Design Report

**Section 1**

**Group 1-D**

**Group Members**

- Ali Emre Ahmetoğlu
- Alperen Utku Yalçın
- Mehmet Onur Uysal
- Utku Boran Torun
- Yarkın Sakıncı

Instructor: Eray Tüzün

Teaching Assistant(s): Muhammad Umair Ahmed, Emre Sülün, Mert Kara, İdil Hanhan

## Contents

# 1. Introduction

## 1.1. Purpose of the System

ERMAN is a web application that has the purpose of making the Erasmus and Exchange processes easier for anyone involved; including the students, coordinators, and instructors. It enables students to manage their Erasmus or Exchange program application by making processes such as course selection, uploading necessary files, contacting the coordinators of the Erasmus program, and contacting peers; simpler. The coordinators are able to manage the applications of the students, contact students/other coordinators and approve/reject courses proposed by the students through the app. Moreover, ERMAN makes it possible for instructors to deal with the approve/reject course processes within the app.

## 1.2. Design Goals

The design goals of the project are revised from the non-functional requirements in the analysis report. The Erman System must be easy to use by all users with a simple interface while containing the functionalities that will be useful for its users. Also, it must be designed to be maintainable and secure for possible future use.

### 1.2.1.

### 1.2.2. Usability

The user interface of Erman is designed to help all kinds of users understand the functionalities. Buttons are self-explanatory on each page and all the icons that are used are highly relevant to their context. Tooltips and short explanations are used for parts that might be confusing. Considering that the users may not be familiar with the application, all features must be clear. Also, the user interface is designed to be responsive for all screen sizes and ratios because the user may want to use the application from different devices.

### 1.2.3. Functionality

The Erman system offers functionalities that will drastically improve the exchange process for coordinators, students, and instructors. Students will be able to see previously accepted courses and propose a new course. All forms will be automatically generated and course approvals will not require any mail traffic. The program is built to make life easier for all participants in the exchange programs, therefore one of the design goals is functionality.

### 1.2.4. Maintainability

The application is built with the aim to be used in the future for Bilkent Erasmus/ Exchange programs for a long time. Therefore, it is important that adding new features and modifying existing ones according to the requests of the clients are as easy as possible. With the object-oriented programming principles and the design patterns that are used in the program, the maintainability of the program is ensured.

### 1.2.5. Security

The system will be storing sensitive information of students and coordinators. Also, it is possible for someone to cancel a student's exchange application or imitate a coordinator if the password of the account is known. In order to prevent data breaches and fake accounts, all accounts are created by the system and the passwords are salted and hashed. Therefore, the project is designed to have a safe and secure environment.

# 2. High-Level Software Architecture

## 2.1. Subsystem Decomposition

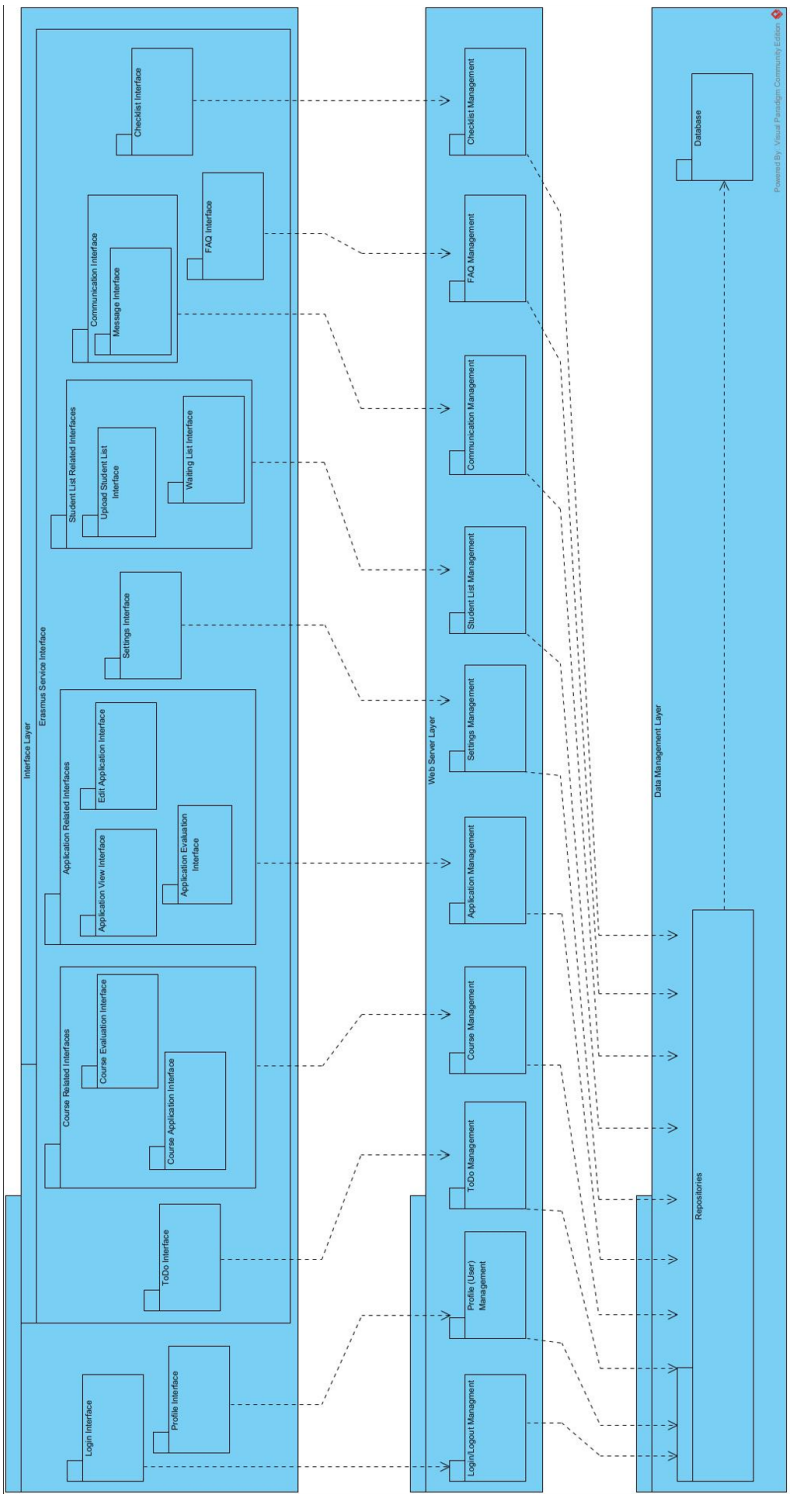



***Figure 2.1**: Subsystem Decomposition of the System*

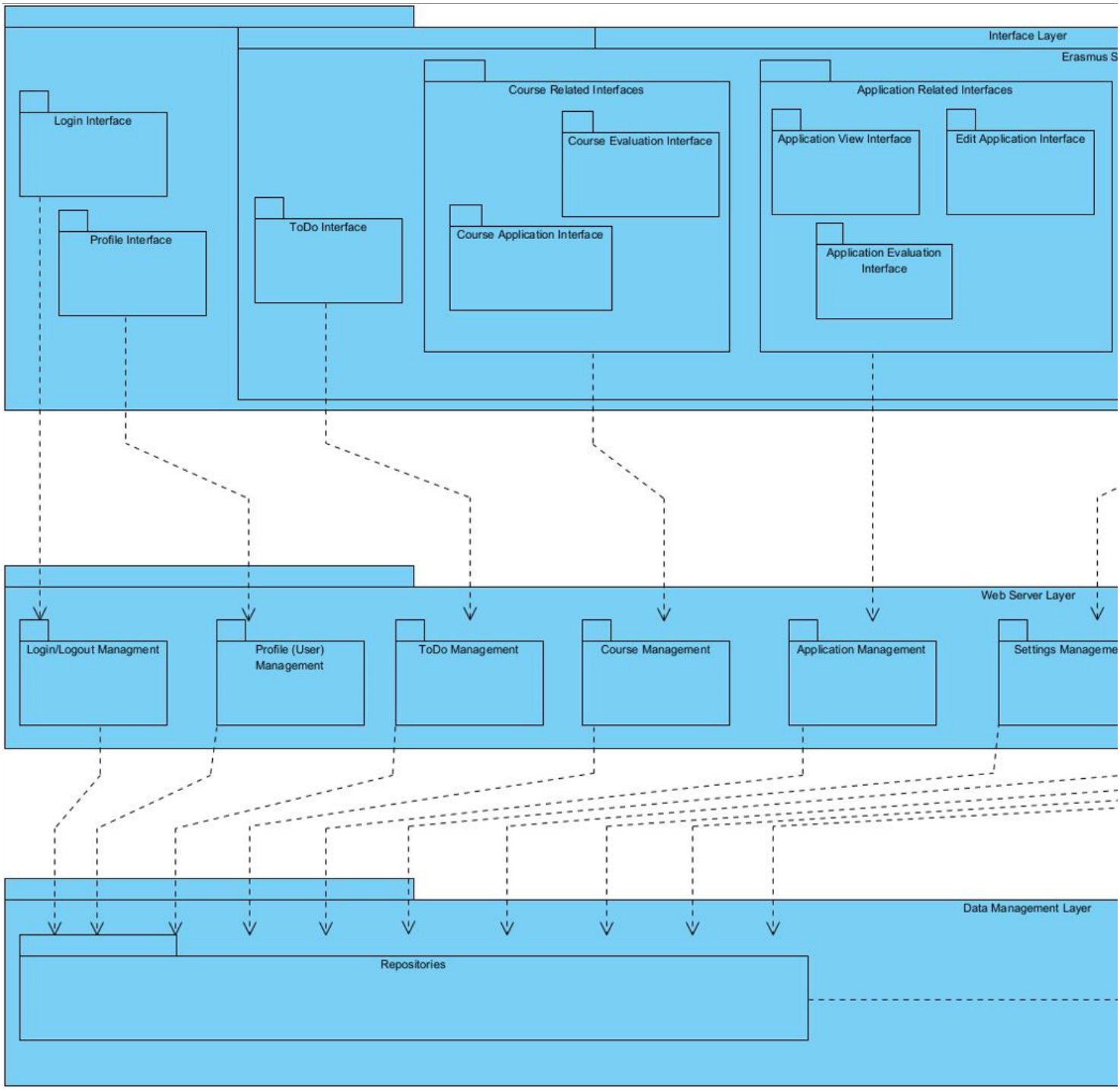



***Figure 2.1.1**: Left side of Figure 2.1*

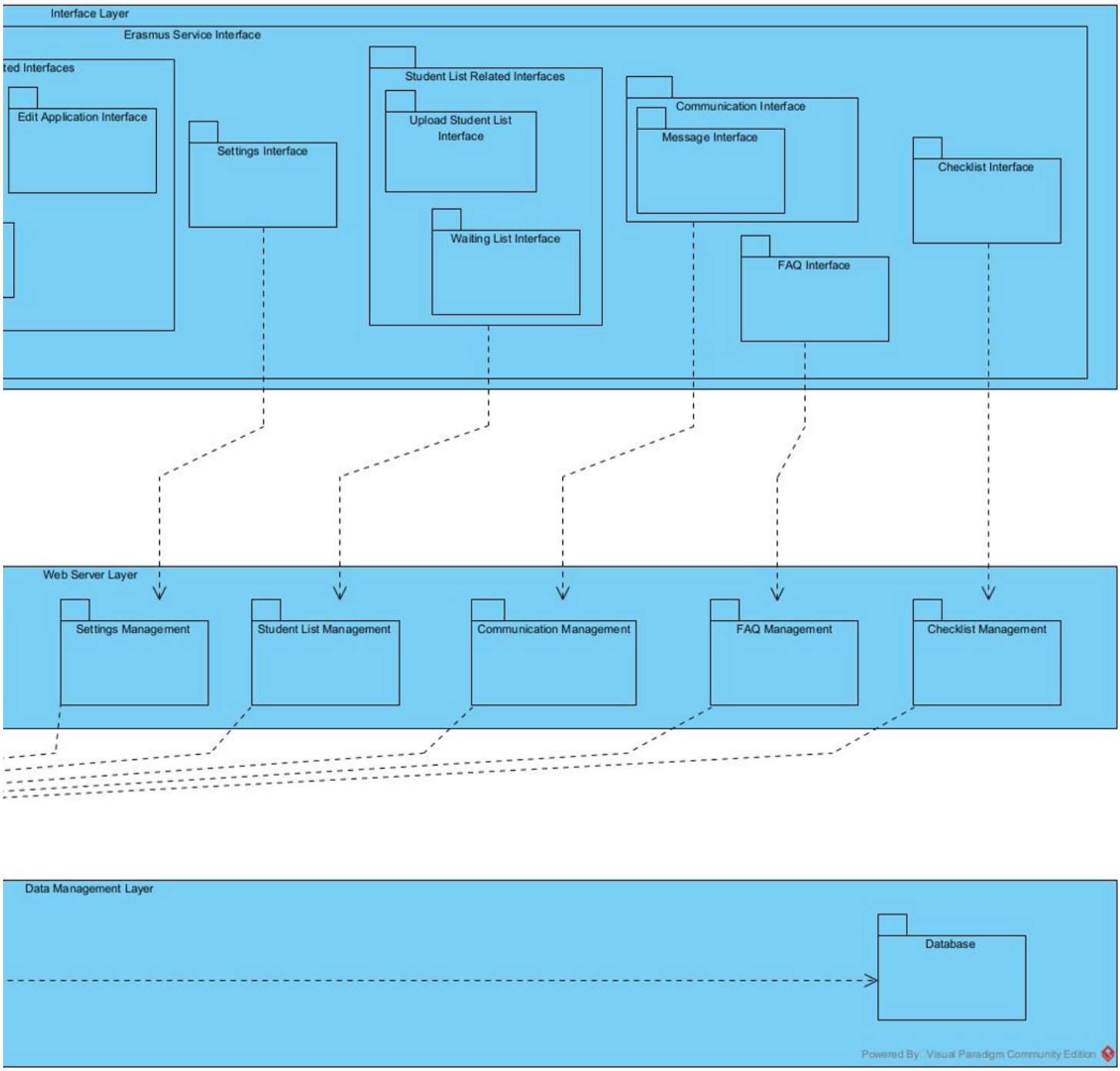



***Figure 2.1.2**: Right Side of the Figure 2.1*

In the Subsystem decomposition part, we have decomposed our entire system to three main layers. These layers are similar to how an MVC design would operate for the subsystem decomposition level of complexity, so these layers' counterparts in MVC's are also going to be

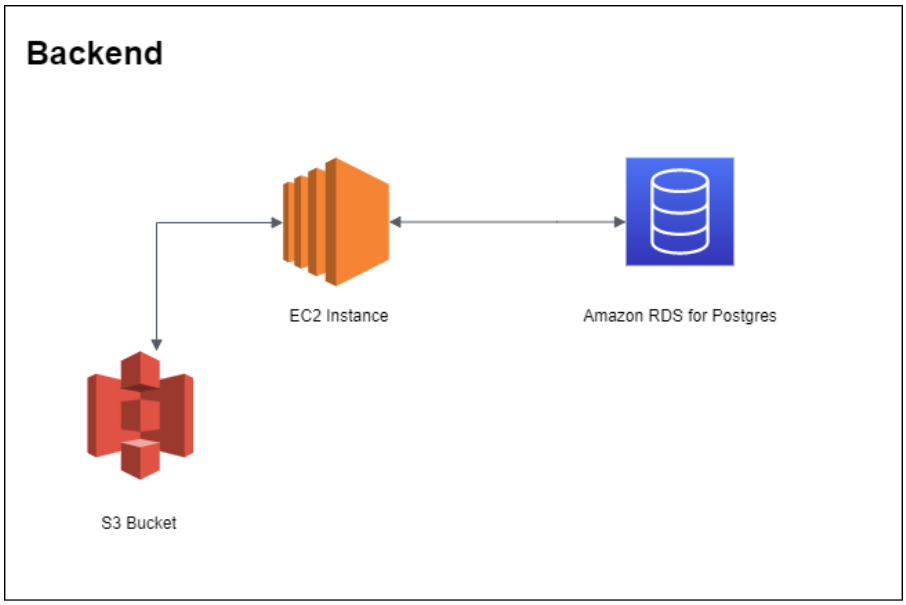
given and explained for the reader to better understand. To list all the layers in our system; we have the Interface, Web server, and Data management layers.

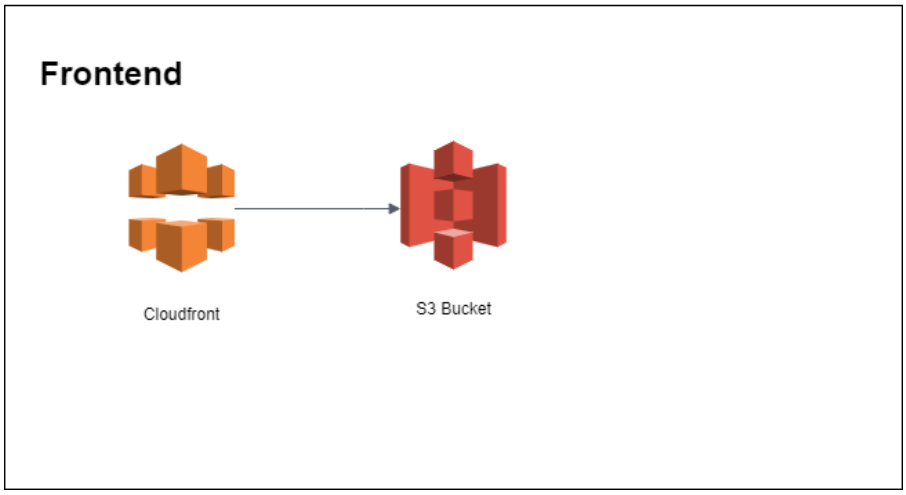
In the Interface layer we have the main accessing pages in the website, all categorized into small packages, as shown in Figure 2.1. These interfaces include what any user (Students, Coordinators, Instructors) have in their ERMAN workspace as a UI page, similar to how a View would function in an MVC design. Certain interface subsystems contain multiple interfaces for different users, such as the “Course Related Interfaces” part contains two different interfaces that operate for different users (the functionalities Course Application and Course Evaluation are available for student and instructors respectively), and can be represented within this same package. Note that in the actual Final Object Design of this report, the UI interfaces and other layers are given in more detail and thoroughly.

In the Web server layer we have the backend server functionalities, given as separate packages that serve different sub purposes of web management for the Interface layer. The managers listed in this subsystem contain operations and core functionalities that process the data provided, and serve as if they are Controller classes. Each manager has a singular interface as an input and connects them to the data management layer.

In the Data management layer, we have the database package that holds all repositories within the system. Each manager is connected to these repositories and gives the information to be stored, and gets what must be retrieved. These repositories communicate with the database and serve as the Model counterpart of our system in an MVC design. The repositories retrieve, replace and add to the database by using Data Transfer Objects (DTOs) and Models. These repositories serve as a means of communication between both the UI interfaces of the web API and also the Web server layer classes and operations.

## 2.2. Hardware/Software Mapping




End users of ERMAN only require a web browser and an internet connection to use ERMAN. Any modern web browser will be sufficient. Both PC and mobile users can use ERMAN. PC users will require a monitor and a mouse. Currently, ERMAN does not have screen reader support for visually-impaired people. The servers of ERMAN can run on any commodity hardware with sufficient specs. The server requires at least 4 GB of RAM and 100 GB of disk space. For the backend we will use Amazon's managed Relational Database Service offering for hosting the Postgres database. The PDF files users upload or generate will be stored in S3 buckets. Our frontend consists of static html, css and js files. These static files will be stored in a S3 bucket and AWS Cloudfront will be used to serve them.

## 2.3. Persistent Data Management

PostgreSQL will be our main database. We chose PostgreSQL because we all have knowledge about PostgreSQL. PostgreSQL was created in 1986 so it had many years to mature and get better. Since it is older software, there are many guides on the internet about solving problems if they arise. We will have a table for every entity, such as Student, Coordinator, Instructor, University, Course, Message,Checklist, FAQ, Todo. Our database will be hosted on AWS.

## 2.4. Access Control and Security

Access Control Matrix:

| | **Accepted Student** | **Rejected Student** | **Coordinator** | **Instructor** | **Exchange Office** |
|---|---|---|---|---|---|
| **Login** | **X** | **X** | **X** | **X** | **X** |
| **Sign up** | **X** | **X** | | | |
| **View Notification** | **X** | **X** | **X** | **X** | **X** |
| **View Forms** | **X** | | **X** | | |
| **Upload Files** | **X** | | **X** | | **X** |
| **View Files** | **X** | **X** | **X** | **X** | |
| **Upload Student Excel** | | | | | **X** |
| **View Student Excel** | | | | | **X** |
| **Remove Student Excel** | | | | | **X** |
| **Send Message** | **X** | **X** | **X** | **X** | **X** |
| **View Messages** | **X** | **X** | **X** | **X** | **X** |
| **View FAQ** | **X** | **X** | **X** | | |

| | | | | | |
|---|---|---|---|---|---|
| **Add new FAQ** | | | **X** | | |
| **Edit FAQ** | | | **X** | | |
| **View Student List** | | | **X** | | |
| **View Student Details** | | | **X** | | |
| **Approve Course Proposal** | | | **X** | **X** | |
| **Reject Course Proposal** | | | **X** | **X** | |
| **View Course Proposals** | | | **X** | **X** | |
| **Propose new Course** | **X** | | | | |
| **Cancel Course Proposal** | **X** | | | | |
| **Add course to selection** | **X** | | | | |
| **Remove course from selection** | **X** | | | | |
| **Cancel application** | **X** | | | | |
| **View Checklist** | **X** | **X** | | | |
| **Check checklist item** | **X** | **X** | | | |
| **Leave waiting list** | | **X** | | | |
| **Approve program** | | **X** | | | |
| **Reject program** | | **X** | | | |
| **View new opportunity** | | **X** | | | |
| **View Todo list** | **X** | **X** | **X** | **X** | **X** |
| **Complete task** | **X** | **X** | **X** | **X** | **X** |
| **Uncomplete task** | **X** | **X** | **X** | **X** | **X** |
| **Star task** | **X** | **X** | **X** | **X** | **X** |

In order to maintain security, ERMAN gets the email, password and Bilkent ID information of the registering user in the registration step. Following this, the registration is enabled if the specified email and Bilkent ID information are matched with the information gathered from the ERMAN database. Through this way, only the allowed users will be using the app and the users will be frequently checked by the admins to ensure that there won't be any outsiders. Besides this, the passwords of all the users will be salted and hashed. Since we won't be saving user passwords directly, even if an attacker gets access to our databases they won't be able to get user's passwords. The authentication of the logged in user is done through cookies. If the name and password information supplied by the user are correctly matched with the user information mapped in the user database, then the user is authenticated and a cookie is generated. This cookie is responsible for storing user specific information where the user can send requests to the API layer. Since the cookie generated following authentication is user specific, the access control is sustained due to the fact that the requests sent will be only linked with the user that sends them. Therefore, the cookie use takes part in both sustaining authentication and authorization.

## 2.5. Boundary Conditions

### 2.5.1 Initialization

The project ERMAN is initialized within its first launch, and remains active as long as the web server functions expectedly, since ERMAN application will be running on a web server 24/7. There is an initialization need of storing and configuring the aws for the application. When deploying the application, the aws credentials will be stored in the .env file. The application will retrieve the aws from here and process the information after initialization.

In any shutdown situations, upon fixing the emergency issue, the web server and the other subsystems will be initialized once again for all user accesses to the system. This initialization process will be in line with the first ever initialization of the system. The existing data will be accessed from the database when initialization is completed.

Note that the users will be able to connect to the application via any device with proper internet connection -though the usage of computers is advised. The users may log in from

different devices or tabs to the system more than once at a time and the system will still function as expected.

### 2.5.2 Termination

The application Erman works on a web server, so when the project is terminated, the web server will be shut down and will be initialized again when the bugs and errors are fixed by our developers. For emergency shutdown situations, the user data will be saved to the database protection system- that contains multiple databases, one being the primary leading database, the others being the secondary idle databases- before shutdown and the data will be available upon the re-initialisation postfix of the problem. This will enable there to be zero data loss.

If the issue is related to the web server, either secondary options of web servers will be used that were present as backup, or simply the web server issues will be awaited to be fixed by the web server related professional teams of developers. These kinds of issues are expected to be solved in minimal time durations, so waiting for web servers to go back online could be as efficient as using other web servers for backup.

### 2.5.3 Failure

Upon any kinds of error that may arise within the project due to unknown circumstances, the web server will be temporarily disabled with a shutdown after saving all data. If an error due to the web server arises, the web server will again be shut down for investigation.

In the cases of failure of the system in any hardware related issues, the system of data protection will enable the data to be maintained even under extreme malfunction circumstances. Our way of data protection is by using multiple separate idle databases (about 4 should be enough) that are constantly updated every few hours/ minutes depending on the amount of changes in the leader database (such as when 10 items were changed in the database). The leader database will be constantly updated with every change to the

system at all times. In a situation where the leading database system is harmed, the leading database role will be given another standby database, and the system will be in a functioning state, instantaneously post-malfunction. The data from the malfunctioning database will be extracted first to the new leading database. In the cases that the hardware issue does not affect this process, no data will be lost. If the leading database hardware is damaged beyond the circumstances where it is unable to do this operation, there may be a minor data loss, yet the data lost is rather insignificant compared to the size of the database, and do note that this case is an insignificant possibility. The harmed hardware will then be fixed and adjusted to the current database configuration as one of the many standby databases, meanwhile the system will operate as expected.

For the development process after failure in software issues, when failure occurs, a page that shows an explanatory error message will be displayed to the users and the problem will be reported to the developers. Some of the system failures will be related to small bugs that only happen very rarely (perhaps only a few of users have come across this issue within the entire user group), where the error will be reported to the developers and will be fixed in upcoming system updates in future iterations of the system, since these failures have less importance.

High priority failures that are seen in a great percentage of users will be instantaneously focused upon by the developers and the system will be updated shortly after the bug has first shown itself, thus reducing the damage caused by the bug. If the bug is too major and complex, the system will be shut down and upon the fixation of the said bug, the system will restart with the updated version. To solve the issues within the backend/ frontend development of the software, the developers will tackle these issues by opening new branches within the GitHub repository of the system with proper error names. Upon the fix of the issues related to the failure, the system will be tested for any other issues that may arise with the changes that solve the previous bug. If the system works as expected, the GitHub error branch will be committed to the main branch, independently of the future planned updates that the developers may be working upon in other branches. This will only update the current system with a single update regarding the bug, thus causing minimal amount of wait time and possible bugs related to complexity of the system.

# 3. Low-Level Design

## 3.1. Object Design Trade-offs

- **Performance versus Functionality**

Providing additional functionalities such as uploading a pdf, and creating the pdf based on the selected courses of the student and related information of the student decreases the performance of the ERMAN application.

- **Usability versus Functionality**

Since the ERMAN application has various features that enable students and coordinators to handle the majority of their work, the functionality of the app increases while the usability declines due to the fact that adding more features indicates adding more components which increase the complexity of the design.

- **Performance versus Maintainability**

ERMAN application has a design that has layers that communicate with each other and together make up the MVC architectural pattern. This is useful for the purpose of making changes or fixing bugs in the feature. However, it decreases the performance as more classes interact in order to provide the functionality requested.

## 3.2. Final Object Design

The figure that is in the next page is our final object design. Since it is long and wide, we separated this figure into two and put it in different pages.

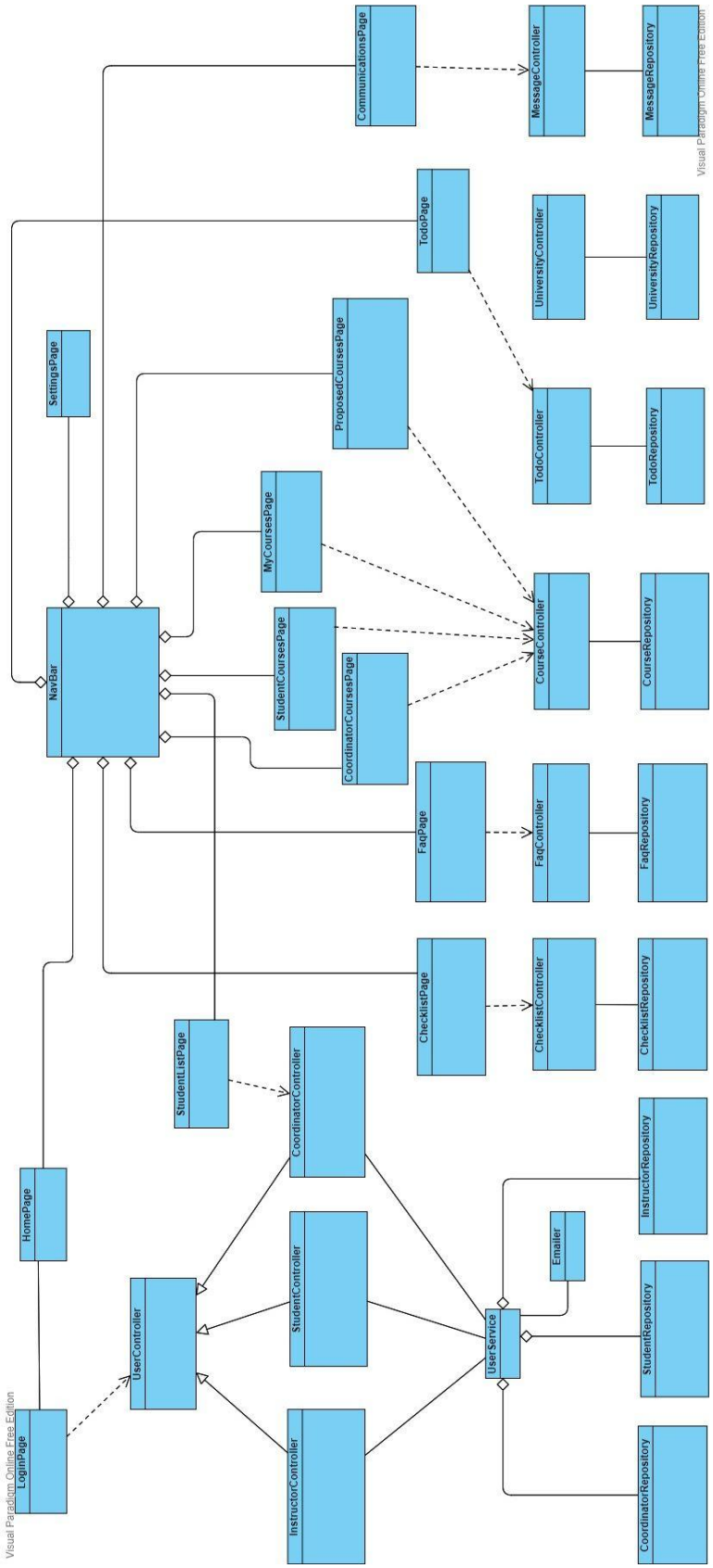



***Figure 3.2**: Final Object Design*

Left side of the diagram:

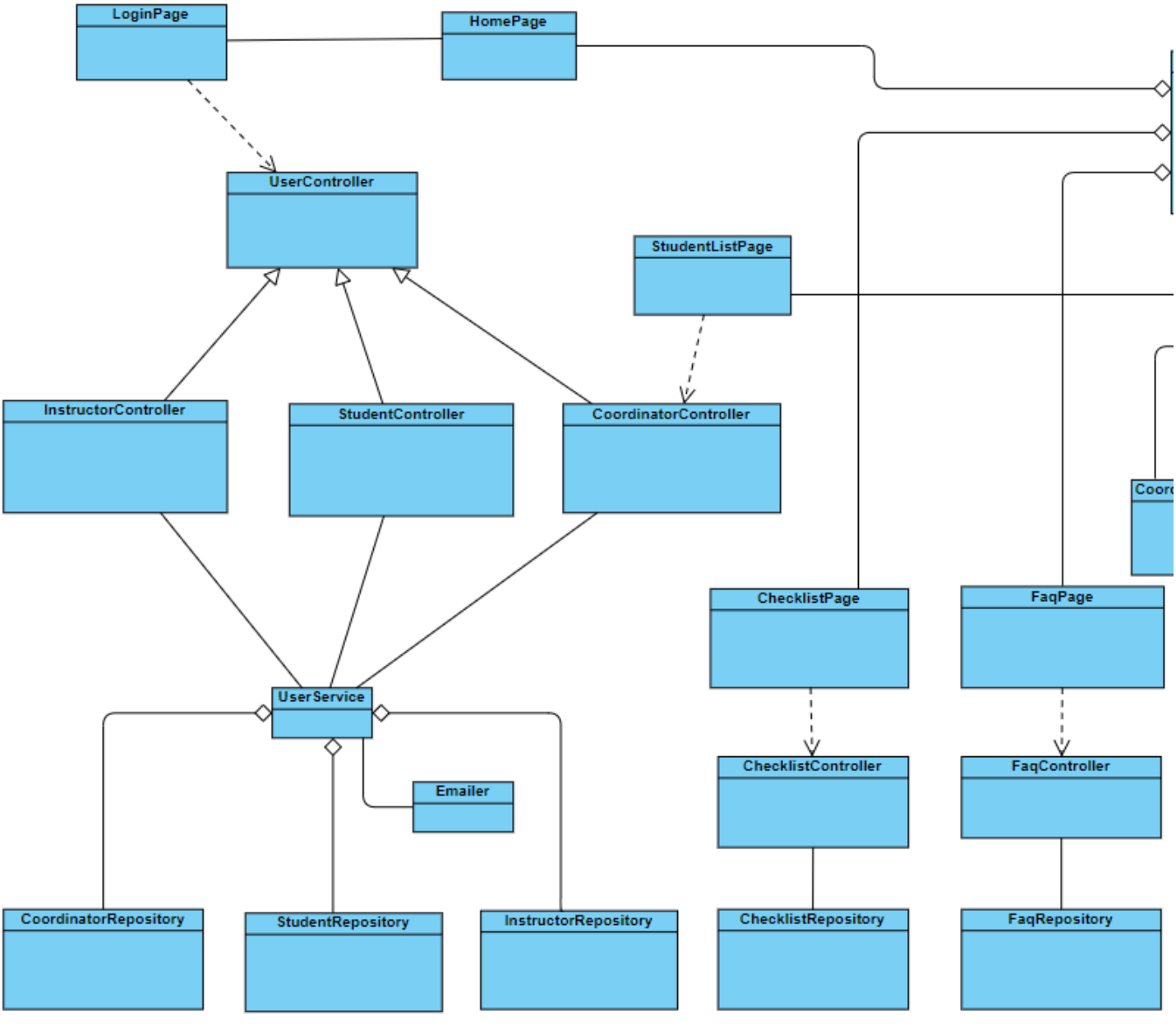



***Figure 3.2.1**: Left Side Of the Final Object Design*

Right side of the diagram:

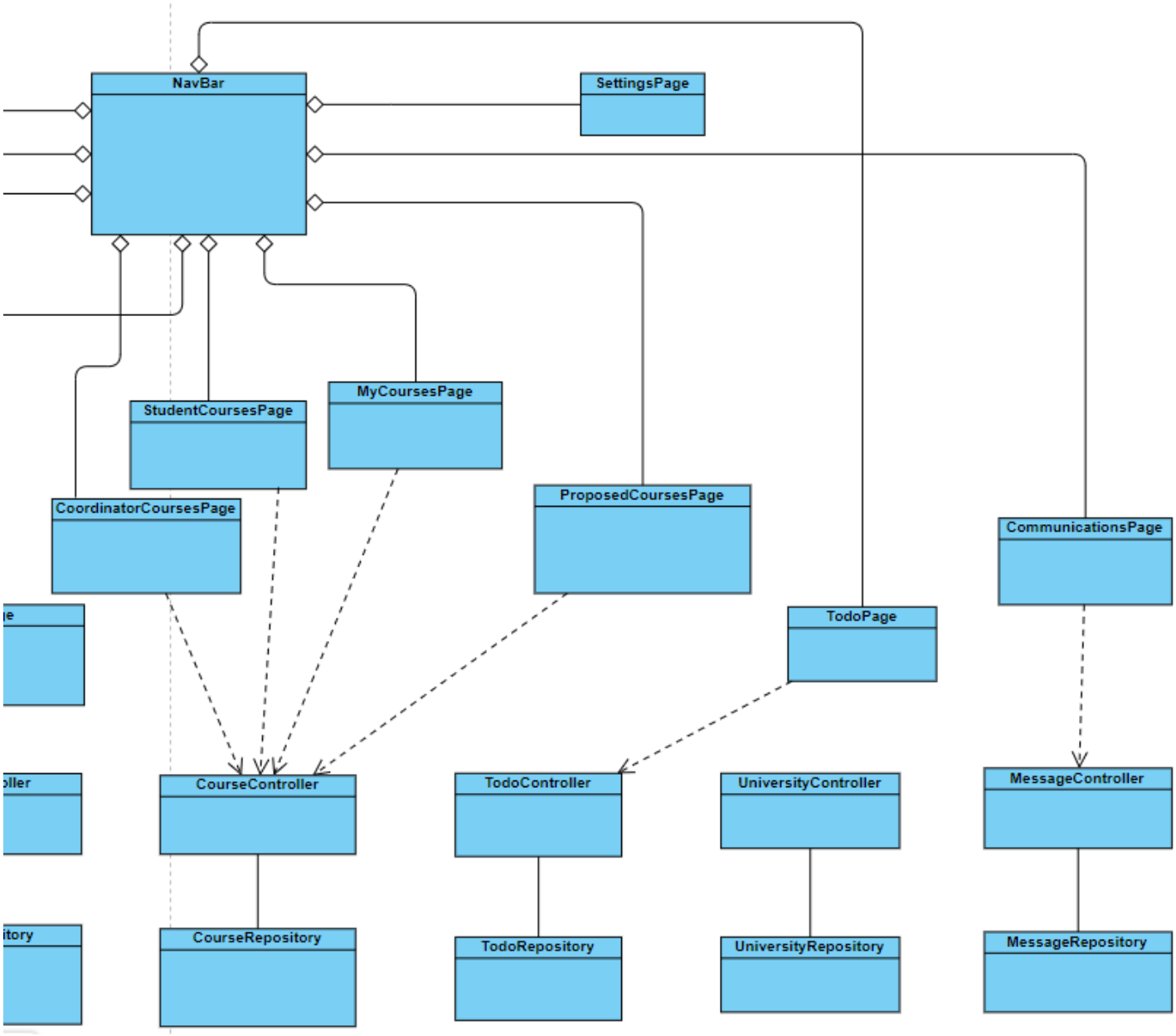



***Figure 3.2.2**: Right Side Of the Final Object Design*

## 3.3. Layers

### 3.3.1. User Interface Management Layer

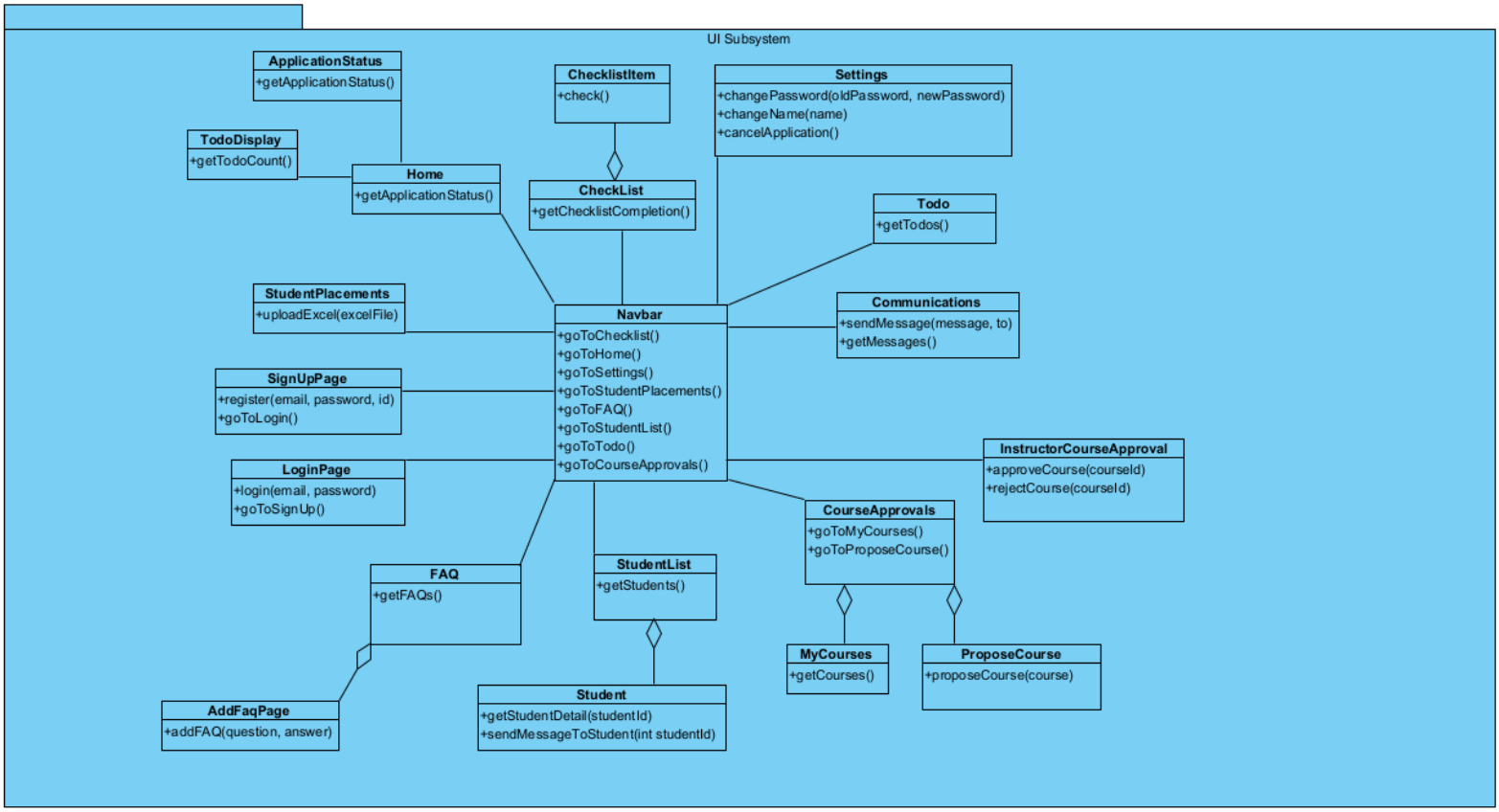



***Figure 3.3.1**: User Interface Management Layer*

This diagram represents the user interface layer of the project. Classes shown on the diagram above are React components. These React components communicate with the Web Server Layer. The state of the application is handled using Redux.

### 3.3.2. Web Server Layer

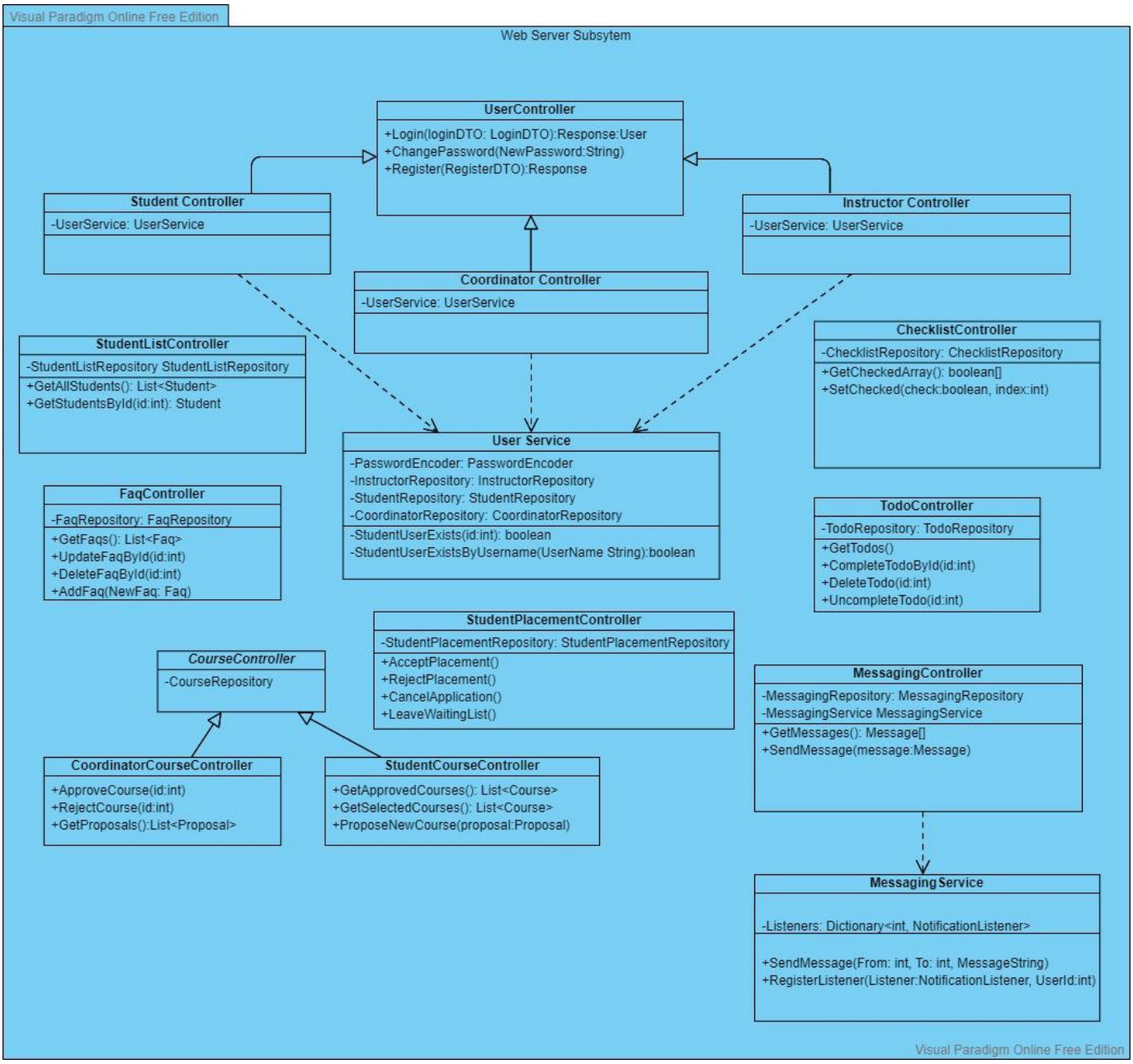



***Figure 3.3.2**: Web Server Layer*

This diagram represents the Web Server Layer of the project. This layer is responsible for linking the user interaction to the data management layer. It handles the requests of the user interface management layer and interacts with the data management layer accordingly.

### 3.3.3. Data Management Layer

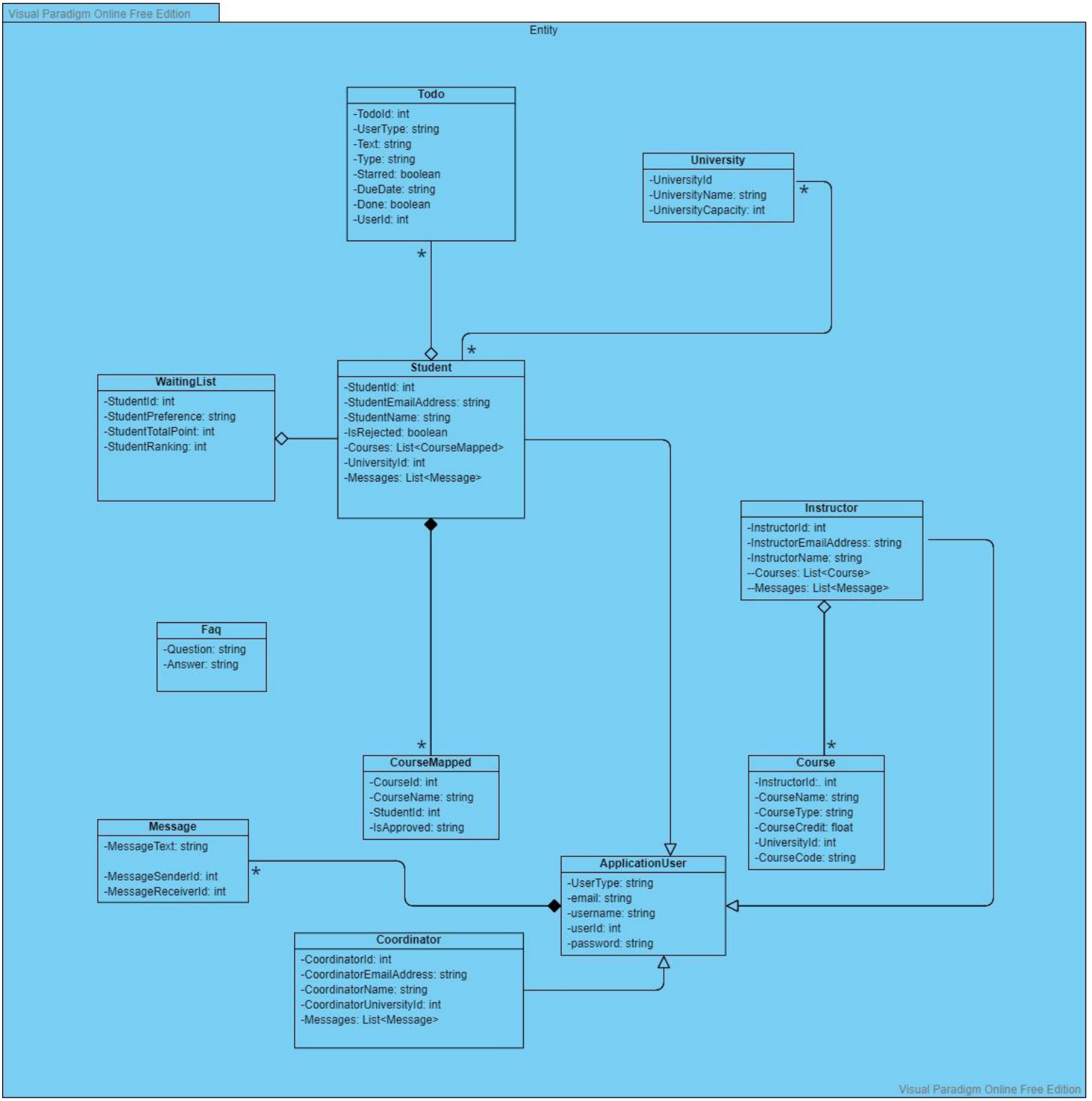



***Figure 3.3.3.1:*** *Entity Subsystem*

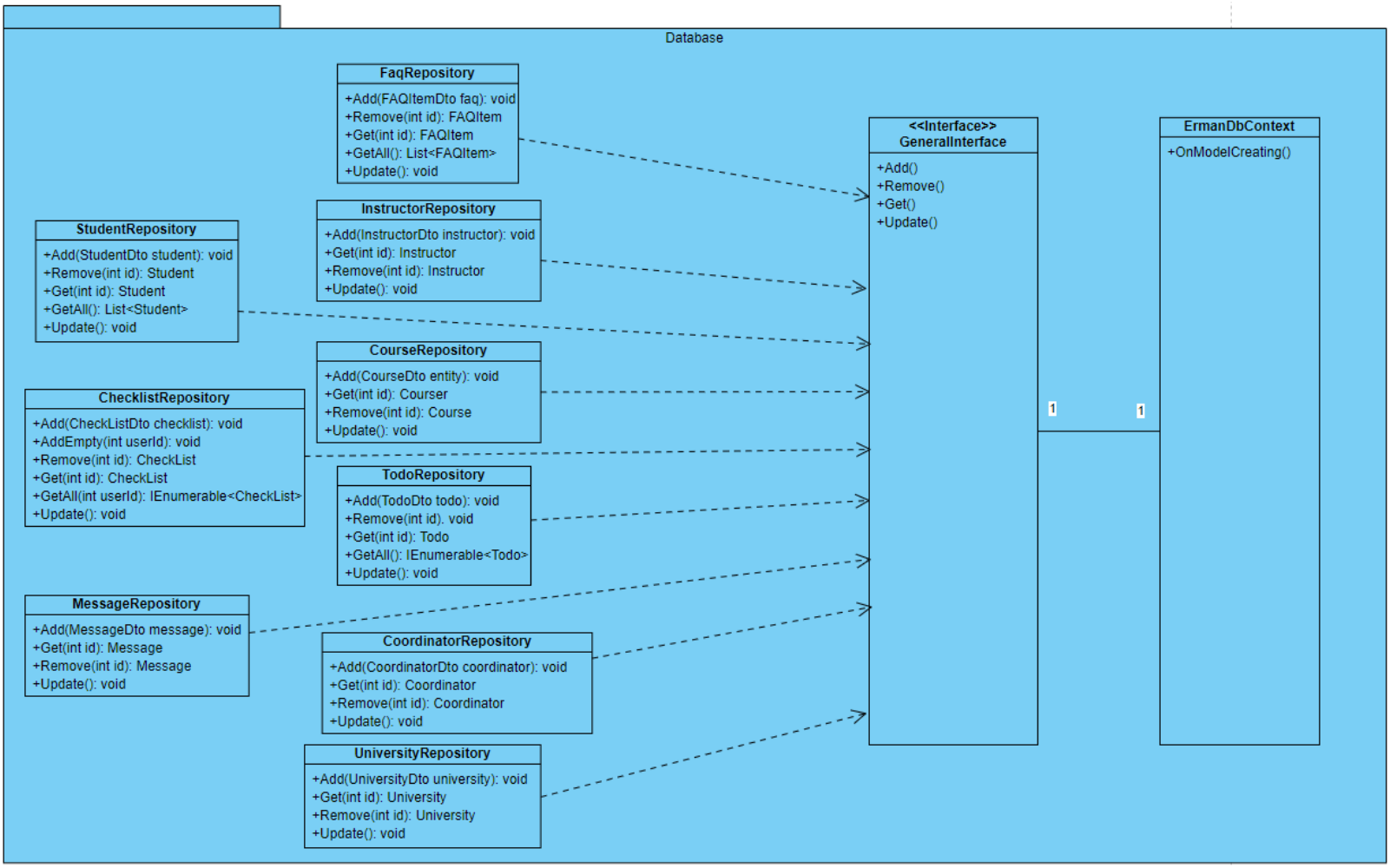



***Figure 3.3.3.2:*** *Database Subsystem*

This layer consists of two separate subsystems which are the Entity subsystem and Database subsystem and is responsible for the data transfer to the databases. The Entity subsystem is responsible for the mapping of the classes to the database and the repository subsystem is responsible for the communication with the database. The communication with the database is done through the ErmanDbContext class.

## 3.4.Design Patterns

### 3.4.1. Singleton Pattern

The MessagingService class uses the singleton pattern. The MessagingService class holds some state regarding the currently online users to deliver them messages so multiple instances of this class does not make sense. There is a single instance of the MessagingService class and every controller that needs to use that service uses the same MessagingService class instance to send messages.

### 3.4.2. Observer Pattern

The Observer pattern is used in the MessagingService. This pattern allows listeners to subscribe to MessagingService's events. Listeners register themselves with the MessagingService and the MessagingService notifies them of new messages. This allows us to add new listeners whenever we want without having to change the MessagingService's code.

## 3.5.Packages

### 3.5.1. Packages Introduced by Developers

**Models**

This package includes classes which are models for our API. These classes is used in every part of the application and saved to the databases.

**Dtos**

This package includes Data Transfer Objects for Models that we have. We used these data transfer objects for several reasons. These are, we need to pass around data between classes and it is safer to do with the DTO and we need to separate domain objects from presentation layer.

**Controllers**

This package includes controllers for our API. These classes are responsible for controlling the way that user interacts with the application.

**Repositories**

This package includes classes which manage repositories for every entity. These repositories saves the data to database via using ErmanDbContext class.

**Services**

This package includes classes which manages services in our system such as Email Service, User Service, Messaging Service and Notification Service which is in Messaging Service.

### 3.5.2. External Library Packages

**Microsoft.EntityFrameworkCore**

This package is developed by Microsoft. This package is an object-database mapper for .NET.

**Microsoft.EntityFrameworkCore.Design**

This package is developed by Microsoft. This package is for design-time components for Entity Framework Core tools.

**Microsoft.EntityFrameworkCore.Tools**

This package is developed by Microsoft. This package enables us to do some functionalities such as, add migration, drop table, update database, etc.

**Npgsql.EntityFrameworkCore.PostgreSQL**

This package is developed by Shay Rojansky, Austin Drenski, and Yoh Deadfall. It is basically a PostgreSQL provider for Entity Framework Core.

**Npgsql.EntityFrameworkCore.PostgreSQL.Design**

This package provides design tools for Npsql Entity Framework Core.

**React.js**

This JavaScript library is used for building the user interface.

**Redux**

This package is used as the state container. It is a centralized and predictable container built for JavaScript apps.

**Pdf-me**

This library is used for generating and manipulating the necessary forms and files.

**BluePrint**

This library is used as a UI toolkit in the front end of the application.

## 3.6.Class Interfaces

### 3.6.1. UI Layer Class Interfaces

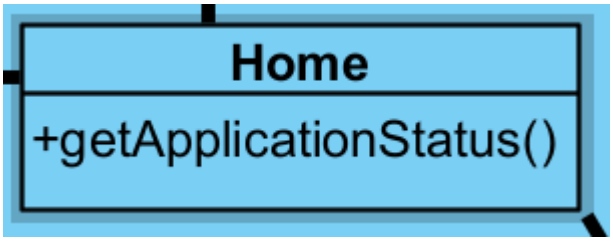
**Checklist**


This page shows a checklist of things the students should do before going abroad.

**Operations:**

**public getChecklistCompletion():** This operation runs on page load. Fetches the checklist completion from the backend.

**Home**




This page shows an overview of the student's application. It also shows how many todo items the student has on her todo list.

**Operations:**

**public getApplicationStatus():** This operation runs on page load. Fetches the current application status from the backend.

**TodoDisplay**


This component shows how many todos a user has.

**Operations:**

**public getTodoCount():** This operation runs on page load. Fetches the number of todo items a user has from the backend.

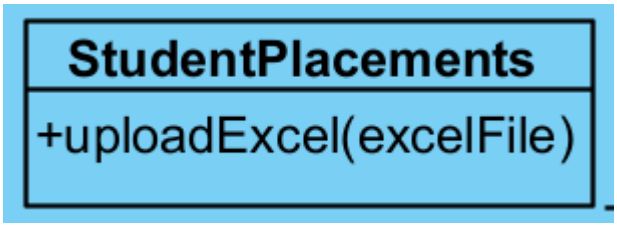
**ApplicationStatus**


This component shows the status of the student's application

**Operations:**

**public getApplicationStatus():** This operation runs on page load. Fetches the student's application's status from the backend.

**StudentPlacements**

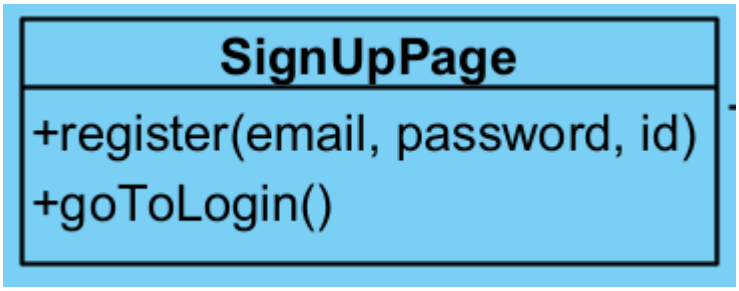



The exchange office will use this page to upload the excel file containing the student's university preferences and their total points.

**Operations:**

**public uploadExcel(excelFile):** Uploads the excel file to the backend for further processing. The backend will sort and place the students into universities using the data in the excel file.

**SignUpPage**




Signup page for students. Accounts for coordinators, instructors and exchange office will be created by the admins.

**Operations:**

**public register(email, password, id):** sends a request to the backend to register with the given data.

**public goToLogin():** navigates the user to the Login page.

**LoginPage**

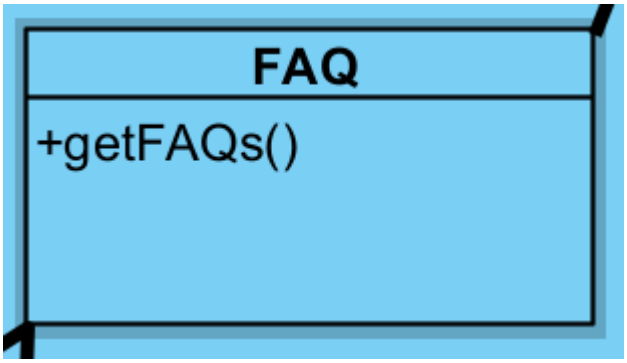

Login page for all users (students, coordinators, instructors and the exchange office).

**Operations:**

**public login(email, password):** sends a login request to the backend to log in.

**public goToSignUp():** navigates the user to the SignUp page.

**FAQ**




This page shows some frequently asked questions about the erasmus application process.

**Operations:**

**public getFAQs():** This operation runs on page load. Fetches the FAQs from the backend

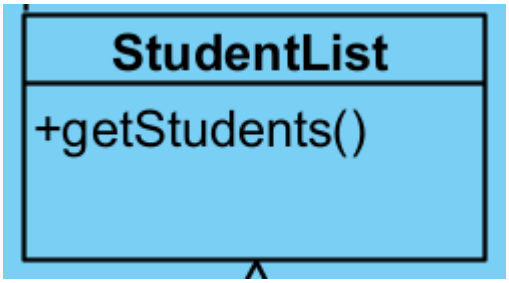
**AddFaqPage**


The coordinators can use this page to add more FAQs to the FAQ page.

**Operations:**

**public addFAQ(question, answer):** Sends a request to the backend to add a new FAQ to the FAQ page.

**StudentList**

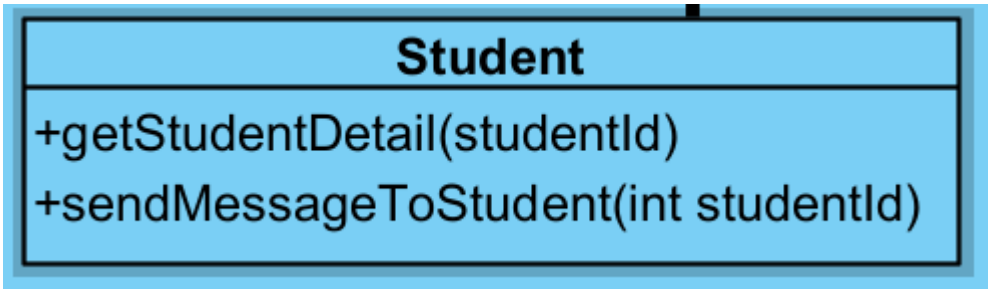



This page shows a list of all the students in the system.

**Operations:**

**public getStudents():** This operation runs on page load. Fetches the list of students from the backend.

**Student**



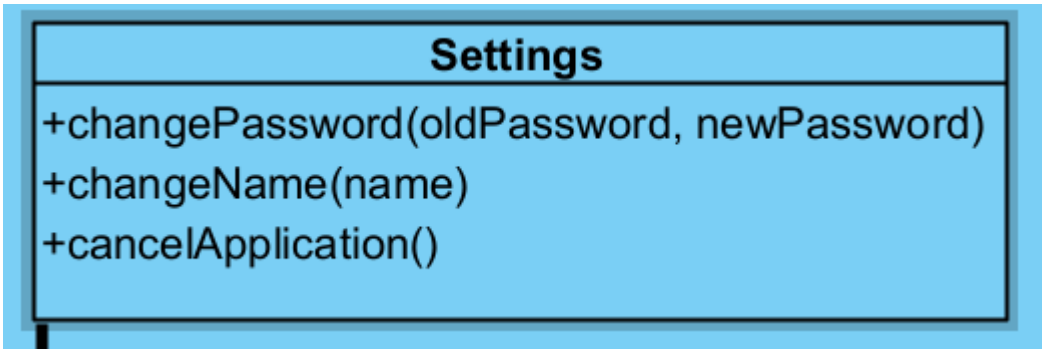

This component shows a single student.

**Operations:**

**public getStudentDetail(studentId):** Fetches the details of the selected student from the backend

**public sendMessageToStudent(studentId):** Sends a request to the backend to send a message to the given student

**Settings**




Users can use this page to change their password and other settings.

**Operations:**

**public changePassword(oldPassword, newPassword):** Sends a request to the backend to change the password of the account.

**public changeName(name):** Sends a request to the backend to change the name of the account owner.

**public cancelApplication():** Sends a request to the backend to cancel the erasmus/exchange application of the user.

**Todo**


This page shows a list of todos that need to be done.

**Operations:**

**public getTodos():** This operation runs on page load. Fetches the todos for the user from the backend.

**Communications**

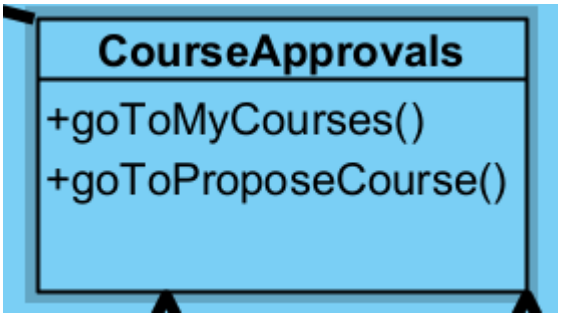

Users can use this page to send messages to each other.

**Operations:**

**public sendMessage(message, to):** Sends a request to the backend to send a message to another user.

**public getMessages():** Sends a request to the backend to fetch the user's messages.

**CourseApprovals**




Students can use this page to propose new courses and see their approved courses.

**Operations:**

**public goToMyCourses():** Navigates the user to the MyCourses page.

**public goToProposeCourse():** Navigates the user to the ProposeCourse page.

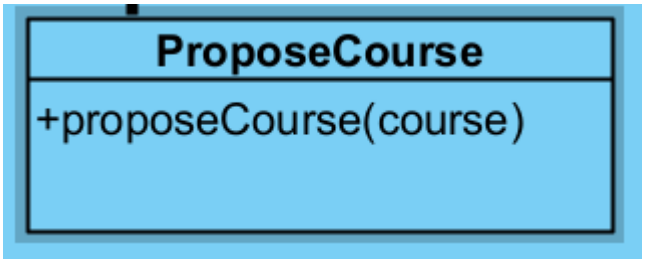
**MyCourses**


This page shows a student's approved and approval pending courses.

**Operations:**

**public getCourses():** This operation runs on page load. Fetches the student's approved and approval pending courses.

**ProposeCourse**

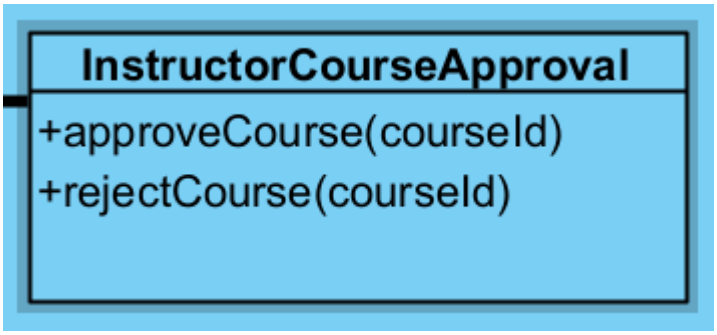



Students can propose new courses for approval from this page..

**Operations:**

**public proposeCourse(course):** Sends a request to the backend to propose a new course for approval.

**InstructorCourseApproval**



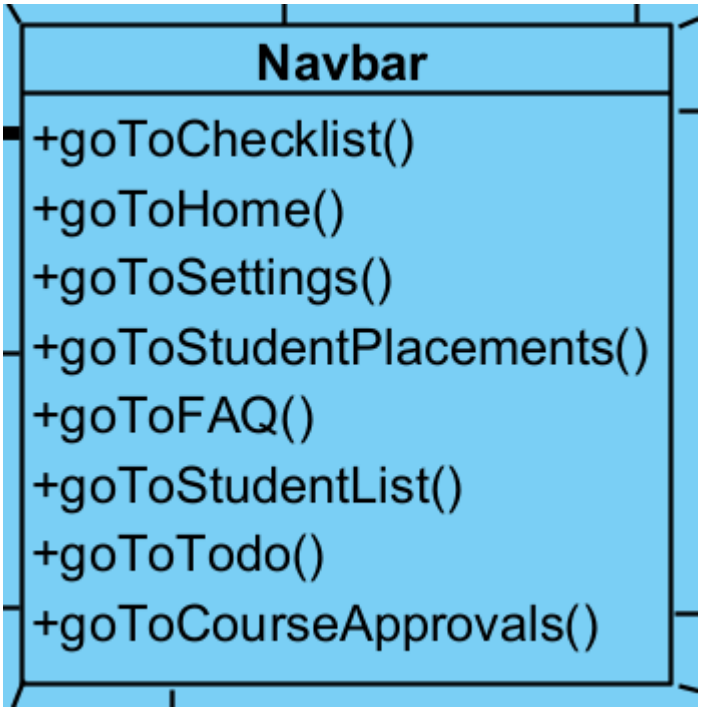

Instructors can use this page to approve or reject courses.

**Operations:**

**public approveCourse(courseId):** Sends a request to the backend to approve a course

**public rejectCourse(courseId):** Sends a request to the backend to reject a course

**Navbar**




Navbar allows users to navigate between pages

**Operations:**

**public goToChecklist():** Navigates the user to the Checklist page.

**public goToHome():** Navigates the user to the Home page.

**public goToSettings():** Navigates the user to the Settings page.

**public goToStudentPlacements():** Navigates the user to the StudentPlacements page.

**public goToFAQ():** Navigates the user to the FAQ page.

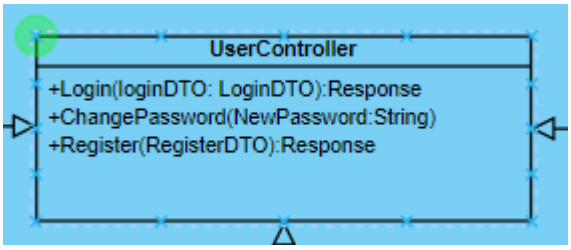
**public goToStudentList():** Navigates the user to the StudentList page.

**public goToTodo():** Navigates the user to the Todo page.

**public goToChecklist():** Navigates the user to the Checklist page.

### 3.6.2. Web Server Layer Class Interfaces

**UserController**




.A class that controls the actions of the user.

**Operations:**

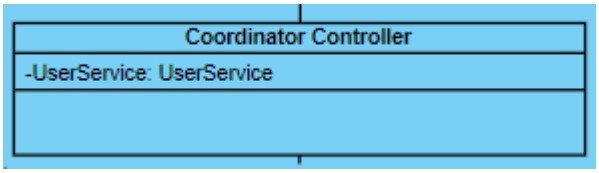
**public void Login(LoginDTO: LoginDTO):** This operation takes a LoginDTO of email and password checks the database and logs the user in and sends user info if the email and password matches.

**public void Register(RegisterDTO: RegisterDTO):** This operation takes a RegisterDTO

and saves the user to the database.

**public void ChangePassword():** This operation takes the new password hashes it with PasswordEncoder and sends to UserRepository

### CoordinatorController

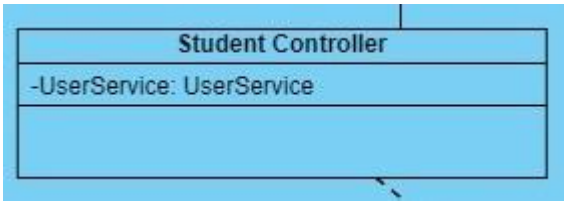



.A class that controls the actions of the coordinator and sends them to the user service.

**Attributes:**

**private UserService UserService:** This attribute handles the user actions initiated by the coordinator.

### StudentController

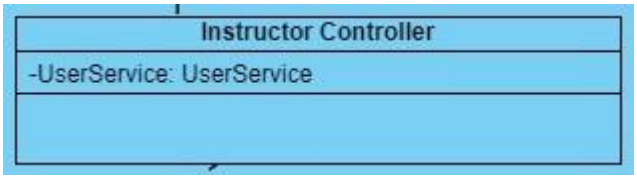



.A class that controls the actions of the student and sends them to the user service.

**Attributes:**

**private UserService UserService:** This attribute handles the user actions initiated by the student.

## InstructorController

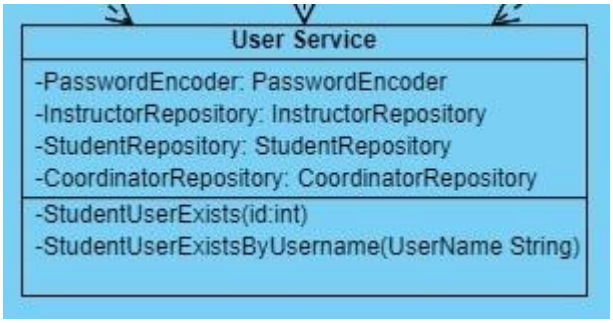



.A class that controls the actions of the instructor and sends them to the user service.

**Attributes:**

**private UserService UserService:** This attribute handles the user actions initiated by the instructor.

## UserService




.A class that handles the user actions

**Attributes:**

**private PasswordEncoder PasswordEncoder:** This attribute encodes the passwords and sends them to the repository as salted and hashed

**private InstructorRepository InstructorRepository:** This is the database repository for instructor users.

**private StudentRepository StudentRepository:** This is the database repository for

student users.

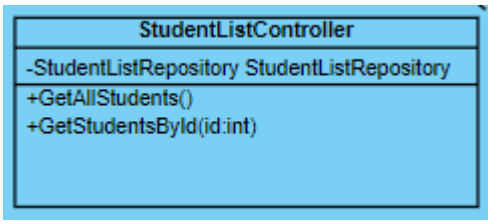
**private CoordinatorRepository CoordinatorRepository:** This is the database repository for coordinator users.

**Operations:**

**private boolean UserExists(id: int):** This functions checks if a user with given id already exists

**private boolean UserExistsByUserName(UserName: string):** This functions checks if a user with given user name already exists

### StudentListController




A class that controls the actions regarding the student list

**Attributes:**

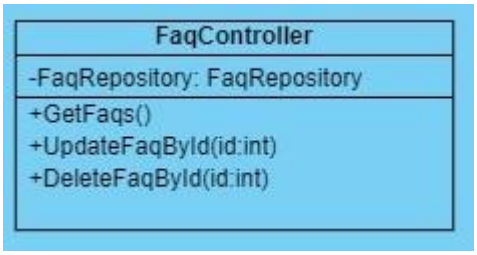
**private StudentListRepository StudentListRepository:** This is the database repository for the student list.

**Operations:**

**public List<Student> GetAllStudents():** This function returns all students stored in the database

**public Student GetStudentById(id: int):** This function returns the student with the given id.

## FaqController




A class that controls the actions regarding the faq

**Attributes:**

**private FaqRepository FaqRepository:** This is the database repository for the faq.

**Operations:**

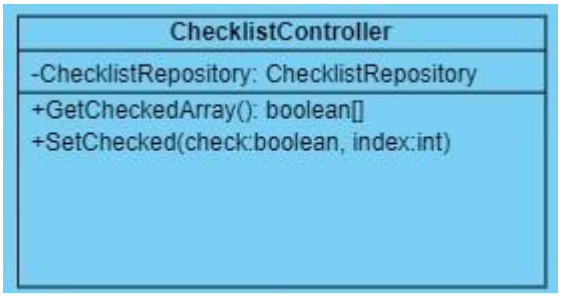
**public List<Faq> GetFaqs():** This function returns all faqs stored in the database

**public void UpdateFaqByID(id: int, Faq:Faq):** This function updates the faq with the given id.

**public void DeleteFaqById(id: int):** This function deletes the faq that has the given id.

**public void AddFaq(NewFaq:Faq):** This function adds the given faq to the faq list.

## ChecklistController




A class that controls the actions regarding thechecklist

**Attributes:**

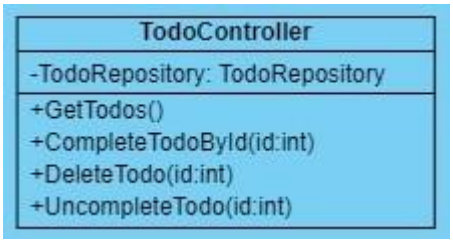
**private ChecklistRepository ChecklistRepository:** This is the database repository for the checklist.

**Operations:**

**public List<boolean> GetCheckedArray():** This function fetches the checked array from the database

**public void SetChecked(Check: boolean, index: int):** This function sets the checked property of the checklist item to the given boolean

## TodoController




A class that controls the actions regarding the todo list.

**Attributes:**

**private TodoRepository TodoRepository:** This is the database repository for the todo.

**Operations:**

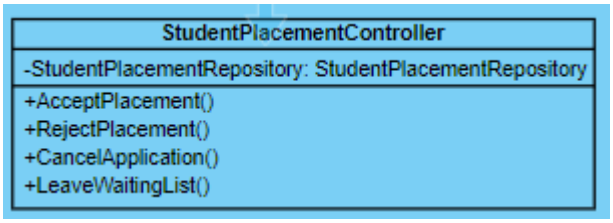
**public List<Todo> GetTodos():** This function fetches the todos from the database and returns it

**public void CompleteTodoById(Id: int):** This function sets completed property of the todo to true

**public void UncompleteTodoById(Id: int):** This function sets completed property of the todo to false

**public void DeleteTodoById(Id: int):** This function removes the todo from the todo list

### StudentPlacementController




A class that controls the actions regarding the student placements

**Attributes:**

**private StudentPlacementRepository StudentPlacementRepository:** This is the database repository for the student placements.

**Operations:**

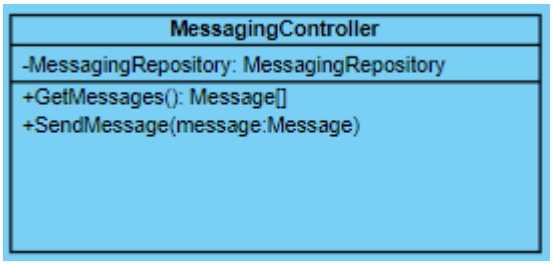
**public void AcceptPlacement():** This function is used when a rejected student accepts placement offer.

**public void RejectPlacement():** This function is used when a rejected student rejects placement offer.

**public void CancelApplication():** This function is used when a student cancels their application.

**public void LeaveWaitingList():** This function is used when a student in the waiting list wants to leave the waiting list

## MessagingController




A class that controls the actions regarding the messaging and notifications

**Attributes:**

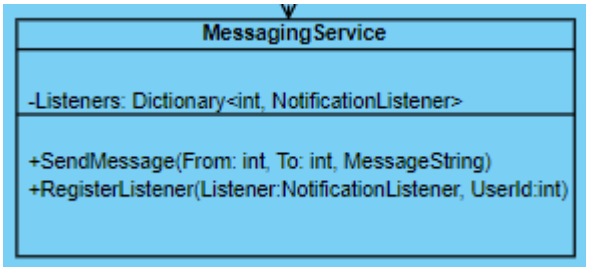
**private MessagingRepository MessagingRepository:** This is the database repository for the messaging.

**Operations:**

**public List<Message> GetMessages():** This function fetches and returns the messages of the user

**public void SendMessage(Message:Message):** This function sends the message in the arguments to another user.

## MessagingService




A service that controls the actions regarding the messaging and notifications

**Attributes:**

**private Dictionary<int, NotificationListener> Listeners:** listeners are added to this dictionary.

**Operations:**

**public void sendMessage(int from, int to, String message):** this function notifies the receiver listener if they are currently listening

**public void registerListener(NotificationListener listener, int userId):** This function registers a listener with the MessagingService

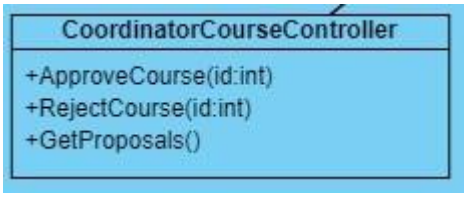
**CourseController**


.A class that controls the actions regarding the courses

**Attributes:**

**CourseRepository:** This is the database repository for the courses

**CoordinatorCourseController**



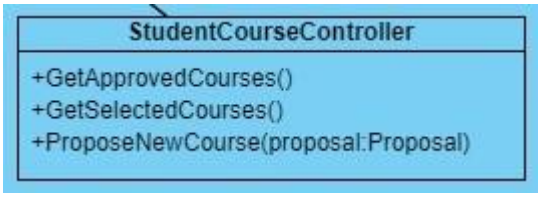

**Operations:**

**public void ApproveCourse(Id:int):** This function sets the approved status of the course to approved.

**public void RejectCourse(Id:int):** This function sets the approved status of the course to rejected.

**public List<Course> GetProposals():** This function fetches the course proposals and returns them

**StudentCourseController**




**Operations:**

**public void GetApprovedCourses():** This function fetches the approved courses and returns them

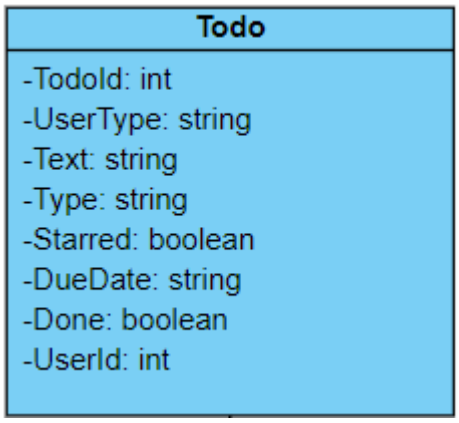
**public void GetSelectedCourses():** This function fetches the selected courses and returns them

**public void ProposeNewCourse(Proposal:Proposal):** This function sends the proposal to the coordinator and instructors.

## 3.6.3. Data Management Layer Class Interfaces

### 3.6.3.1. Entity Layer Class Interfaces

**ToDo**




This class is for modeling the to do list items to be stored in the database.

**Attributes**

**public Int ToDoId**: Id of the ToDo

**public string UserType**: User type for the ToDo

**public string Text**: Text of the ToDo

**public int UserId**: User Id of the ToDo

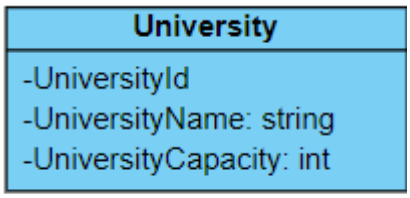
**public string Type**: Type of the ToDo

**public bool Starred**: Boolean value of the starred property of the ToDo

**public bool Done**: Boolean value of the done property of the ToDo

**public string DueDate**: Due date of the ToDo

**University**




This class is for modeling the universities linked in the Erasmus/Exchange Program.

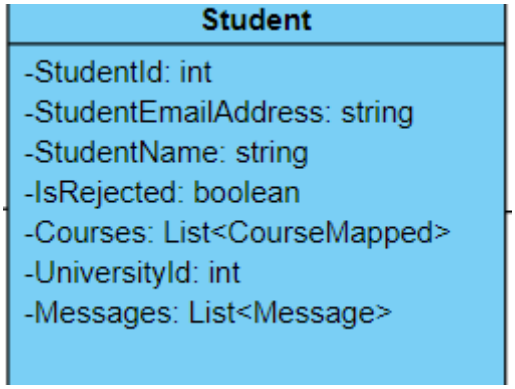
**Attributes**

**public int Id:** Id of the university

**public string UniversityName:** Name of the university

**public int UniversityCapacity:** Capacity of the university

**Student**




This class is for modeling the students who have applied for the Erasmus/Exchange Program.

**Attributes**

**public int StudentId:** Id of the student

**public string StudentEmailAddress:** Email of the student

**public string StudentName:** Name of the student

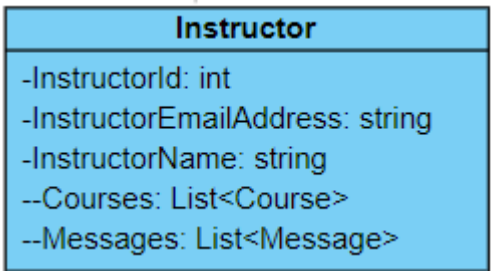
**public int UniversityId:** University Id of the student

**public bool IsRejected:** Boolean property of the rejection status of student

**public List<Course> Courses:** Courses of the student

**public List<Message> Messages:** Messages of the student

**Instructor**




This class is for modeling the instructors who teach the courses that are mapped.

**Attributes**

**public int InstructorId:** Id of the instructor

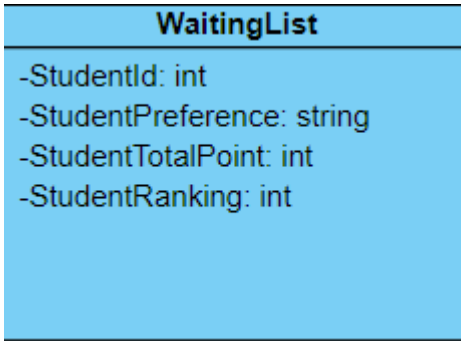
**public string InstructorEmailAddress:** Email of the instructor

**public string InstructorName:** Name of the instructor

**public List<Course> Courses:** Courses of the instructor

**public List<Message> Messages:** Messages of the instructor

**Waiting List**




This class is for modeling the students who are in the waiting list of the Program.

**Attributes**

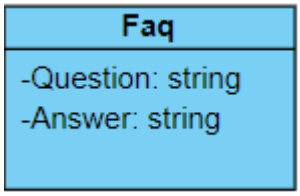
**public int StudentId:** Id of the student

**public string StudentPreference:** University preference of the student

**public int StudentTotalPoint:** Total points of the student

**public int StudentTotalRanking:** Ranking of the student

**Faq**



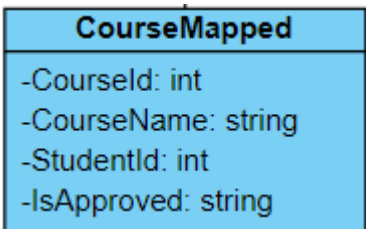

This class is for modeling the frequently asked questions and their answers.

**Attributes**

**public string Question:** Question of the faq

**public string Answer:** Answer of the faq

**CourseMapped**




This class is for modeling the courses that are mapped with the other universities' courses.

**Attributes**

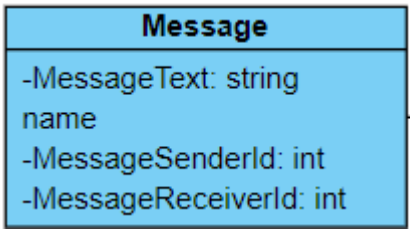
**public int CourseId:** Id of the course

**public string CourseName:** Name of the course

**public int StudentId:** Id of the student

**public bool IsApproved:** Boolean value of the approved status of course

**Message**




This class is for modeling the messages that are sent between users.

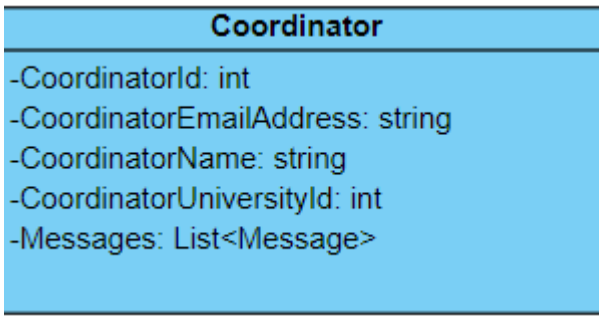
**Attributes**

**public string MessageText:** Text of the message

**public int MessageSenderId:** Id of the sender of the message

**public int MessageReceiverId:** Id of the receiver of the message

**Coordinator**




This class is for modeling the coordinators of the Erasmus/Exchange Program.

**Attributes**

**public int CoordinatorId:** Id of the coordinator

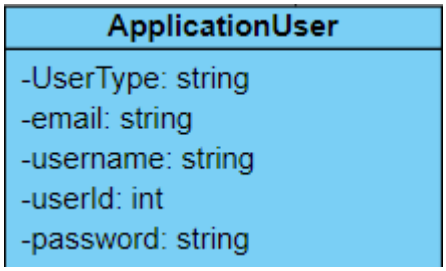
**public string CoordinatorEmailAddress:** Email of the coordinator

**public string CoordinatorName:** Name of the coordinator

**public int CoordinatorUniversityId:** Id of the university of the coordinator

**public List<Message> Messages:** Messages of the coordinator

**ApplicationUser**




This class is for modeling the general user of the application.

**Attributes**

**public string UserType:** Type of the user

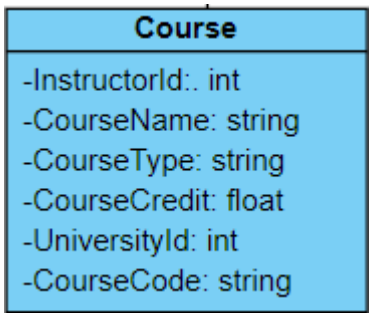
**public string Email:** Email of the user

**public string Username:** Username of the student

**public int UserId:** User Id of the user

**public String Password:** Password of the user

**Course**




This class is for modeling the course of the university.

**Attributes**

**public int InstructorId:** Id of the instructor of the course

**public string CourseName:** Name of the course

**public string CourseType:** Type of the course

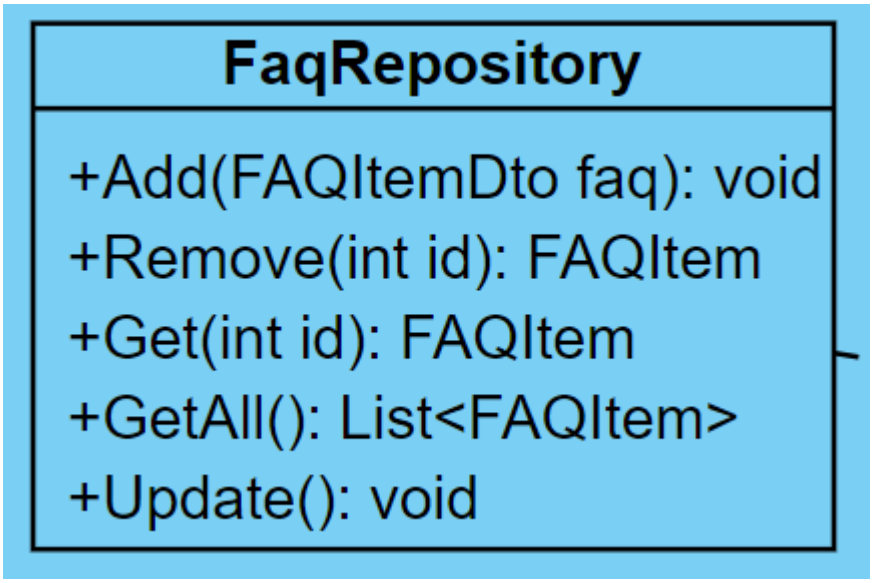
**public float CourseCredit:** Credits of the course

**public int UniversityId:** Id of the university

**public string CourseCode:** Code of the course

#### 3.6.3.2. Database Subsystem Class Interface

**FaqRepository**




This class is repository for FAQ (Frequently Asked Questions).

**Operations:**

**public void Add(FAQItemDto faq)**: This function is for adding FAQ item to the database.

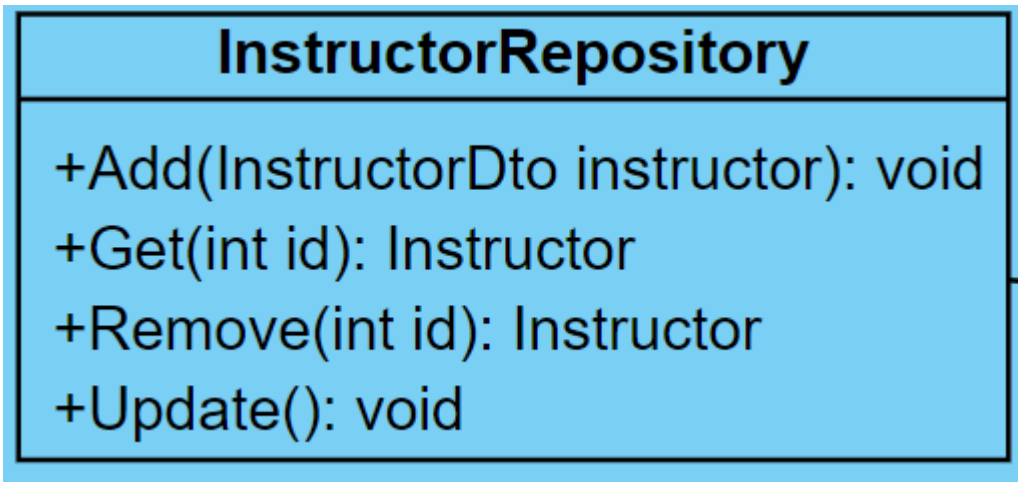
**public FAQItem Remove(int id)**: This function is for removing a FAQ item with given id from database.

**public FAQItem Get(int id)**: This function returns a FAQ item with the given id from database.

**public List<FAQItem> GetAll():** This function returns all FAQ item in the database.

**public void Update()**: This function is for saving the changes to the database.

**InstructorRepository**




This class is repository for Instructor object.

**Operations:**

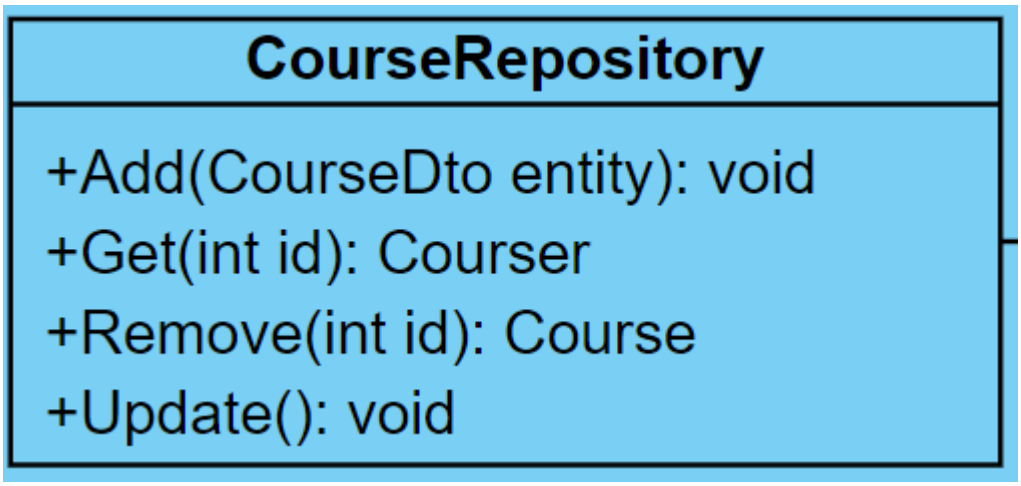
**public void Add(InstructorDto instructor)**: This function is for adding new instructor to the database.

**public Instructor Get(int id)**: This function is for getting the Instructor object with given id from database.

**public Instructor Remove(int id):** This function is for removing Instructor object with given id and returning the removed object.

**public void Update()**: This function is for saving the changes to the database.

**Course Repository**




This class is repository for Course object.

**Operations:**

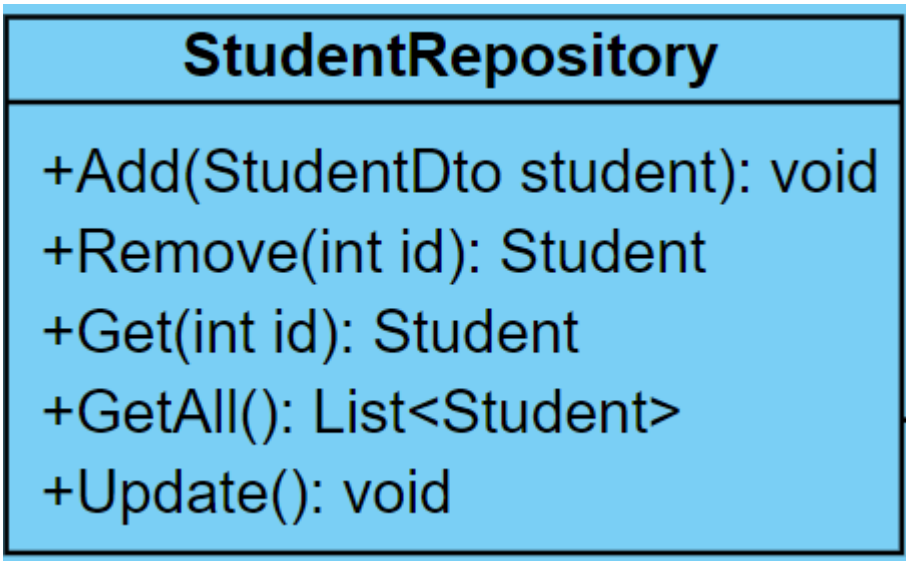
**public void Add(CourseDto course)**: This function is for adding new course to the database.

**public Course Get(int id)**: This function is for getting the Course object with given id from database.

**public Course Remove(int id):** This function is for removing Course object with given id and returning the removed object.

**public void Update()**: This function is for saving the changes to the database.

**Student Repository**




This class is repository for Student object.

**Operations:**

**public void Add(StudentDto student)**: This function is for adding new student to the database.

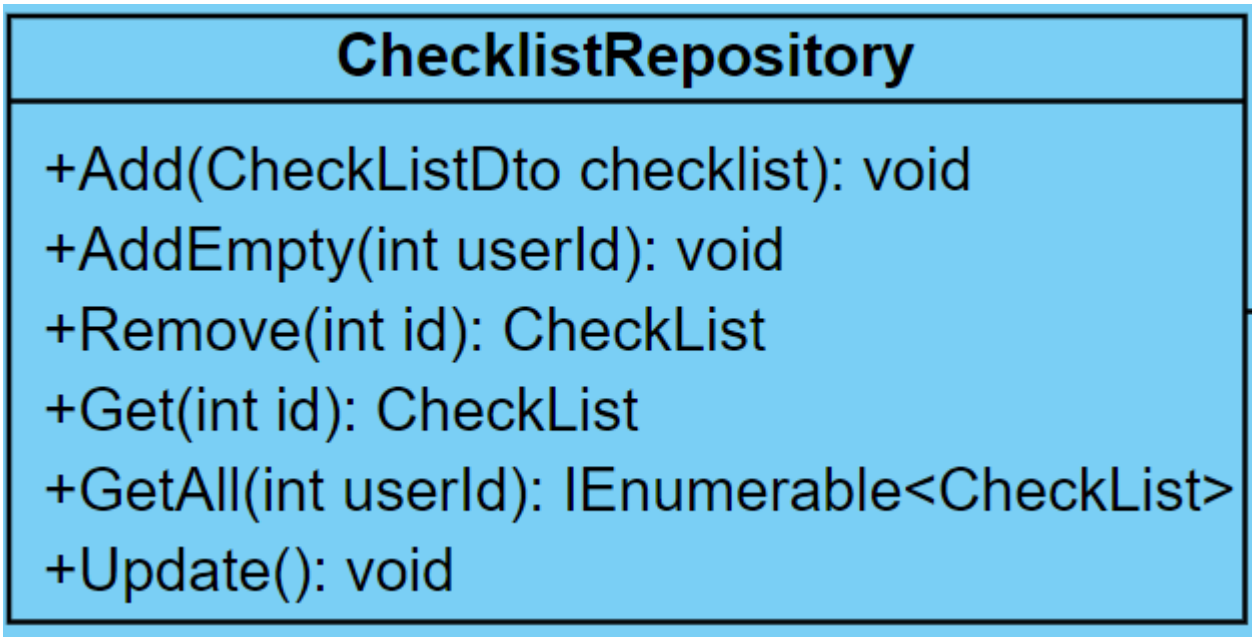
**public Student Get(int id)**: This function is for getting the Student object with given id from database.

**public Student Remove(int id):** This function is for removing Student object with given id and returning the removed object.

**public List<Student> GetAll()**: This function returns all students in the database.

**public void Update()**: This function is for saving the changes to the database.

**Checklist Repository**




This class is repository for Checklist.

**Operations:**

**public void Add(CheckListDto checklist)**: This function is for adding new checklist item to the database.

**public void AddEmpty(int userId)**: This function adds a fully unchecked checklist to the system.

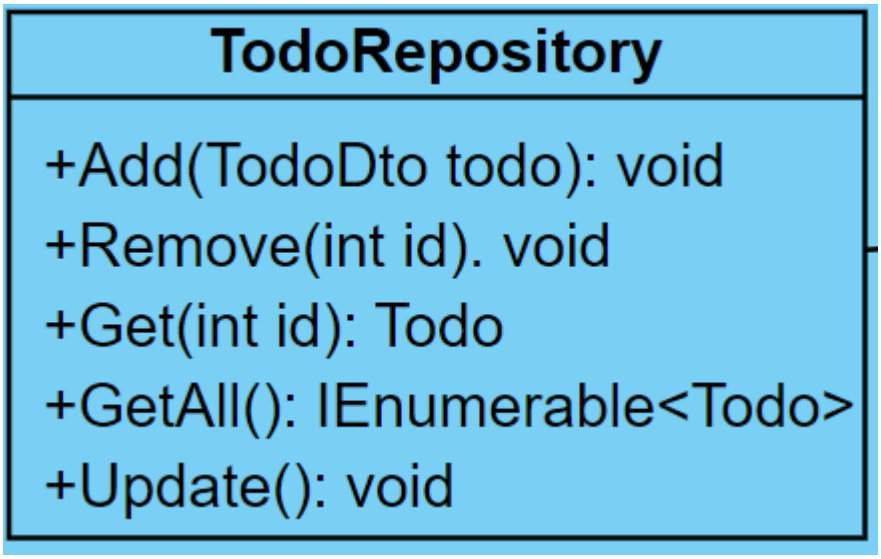
**public CheckList Remove(int id)**: This function is for removing Checklist item with given id and returning the removed object.

**public CheckList Get(int id)**: This function is for getting the Checklist item with given id from database.

**public IEnumerable<CheckList> GetAll(int userId)**: This function is for getting all Checklist items for given user by its id.

**public void Update()**: This function is for saving the changes to the database.

**Todo Repository**




This class is repository for To-Do items in the system.

**Operations:**

**public void Add(TodoDto todo)**: This function is for adding new todo item to the database.

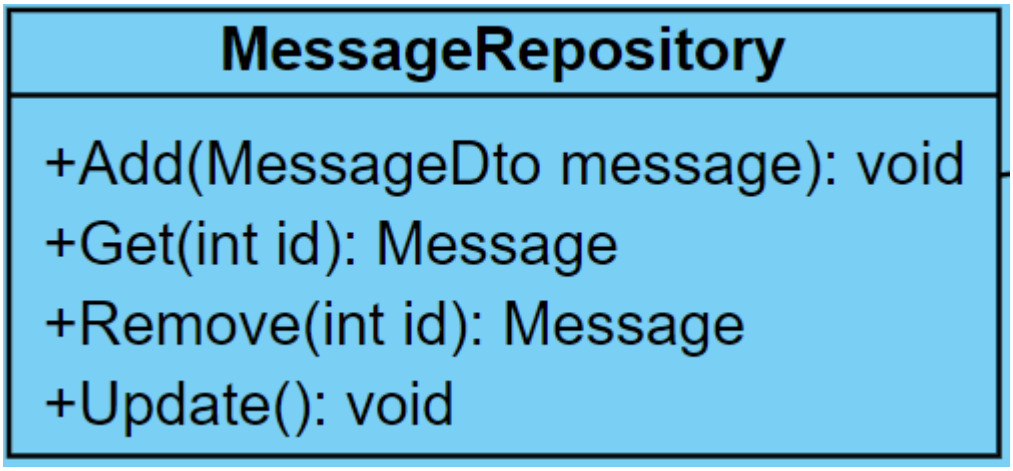
**public Todo Get(int id)**: This function is for getting the Todo item object with given id from database.

**public Todo Remove(int id):** This function is for removing Todo item object with given id and returning the removed object.

**public IEnumerable<Todo> GetAll()**: This function returns all todo items in the database.

**public void Update()**: This function is for saving the changes to the database.

**MessageRepository**




This class is repository for Message objects in the system.

**Operations:**

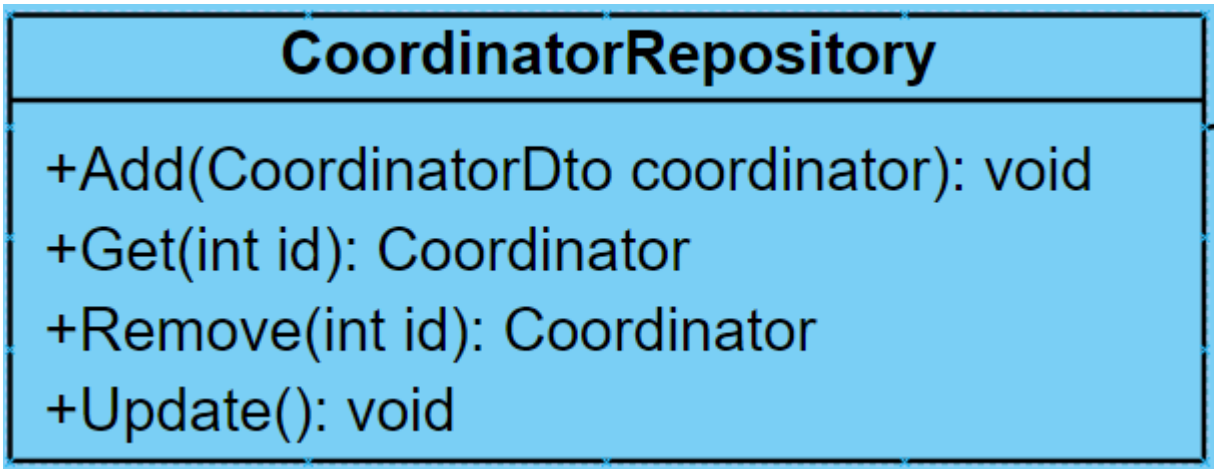
**public void Add(MessageDto message)**: This function is for adding new message object to the database.

**public Message Get(int id)**: This function is for getting the Message object with given id fom database.

**public Message Remove(int id):** This function is for removing Message object with given id and returning the removed object.

**public void Update()**: This function is for saving the changes to the database.

**Coordinator Repository**




This class is repository for Coordinator object.

**Operations:**

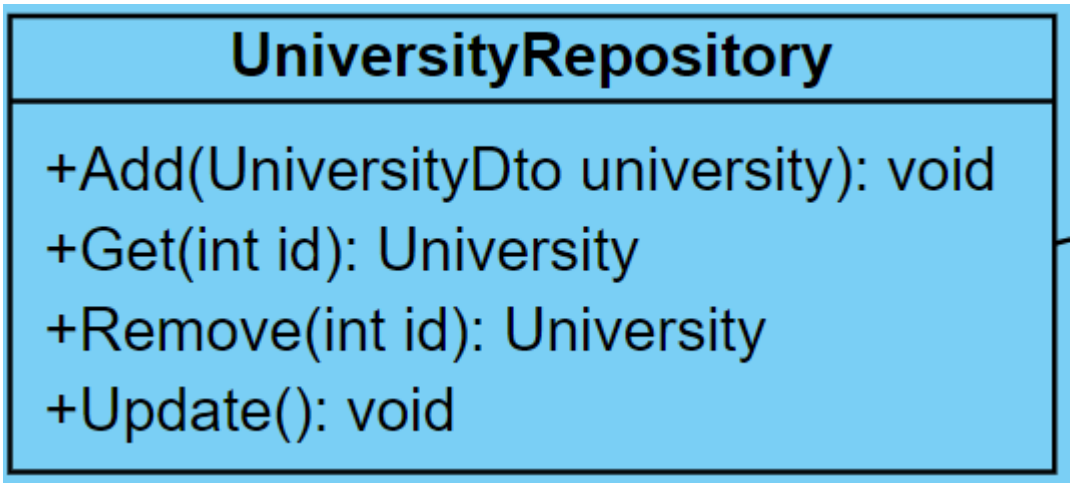
**public void Add(CoordinatorDto coordinator)**: This function is for adding new coordinator to the database.

**public Coordinator Get(int id)**: This function is for getting the Coordinator object with given id from database.

**public Coordinator Remove(int id):** This function is for removing Coordinator object with given id and returning the removed object.

**public void Update()**: This function is for saving the changes to the database.

**University Repository**




This class is repository for Universit objects in the system.

**Operations:**

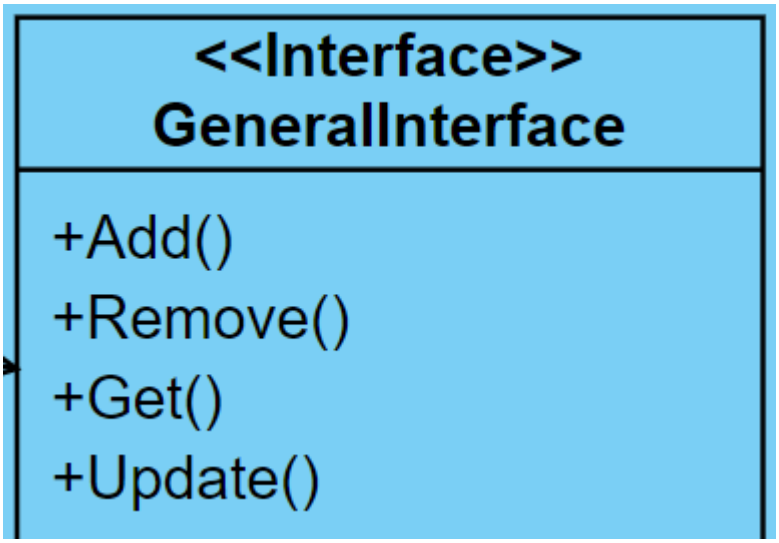
**public void Add(UniversityDto coordinator)**: This function is for adding new university to the database.

**public University Get(int id)**: This function is for getting the University object with given id from database.

**public University Remove(int id):** This function is for removing University object with given id and returning the removed object.

**public void Update()**: This function is for saving the changes to the database.

**General Interface**




This interface is for general functionalities of repositories.

**public void Add()**: This function is for adding new data to the database.

**public T Remove()**: This function is for removing data from database and returns the removed object (T = Entity).

**public T Get()**: This function is for getting specified data from database.

**public void Update()**: This function is for saving changes in database.

# 4.Improvement Summary

## 4.1. Subsystem Decomposition

- Added more abstraction with broader subsystem designs.

## 4.2. Boundary Conditions

- Solution for hardware related malfunctions added.

## 4.3. Persistent Data Management

- Which entities will be held, added in detail.
- Why PostgreSQL is chosen is added in detail.
- Added AWS deployment diagram.

## 4.4. Low-Level Design

- Final Object Design fixed according to aggregation of the objects.

## 4.5. Access Control and Security

- Added an access matrix to specify user actions for different kinds of users

## 4.6. Data Management

- All classes and interfaces' operations and attributes are explained

## 4.7.Web Server Layer Diagram

- Added missing classes to Web Server Layer Diagram

## 4.8. Packages Introduced By Developers

- Packages explained in detail and why they are chosen is explained in detail.