Bilkent University

Department of Computer Engineering

CS319 Term Project

Section 2
Group 2G
Erasmus App

Project Design Report

Group Members:

- İlker Özgen - 21902719
- Onurcan Ataç - 22002194
- Kutay Şenyiğit - 21902377
- Utku Kurtulmuş - 21903025
- Mert Ünlü - 22003747
- Ömer Oktay Gültekin - 21901413

Instructor:
Eray Tüzün

TA's:

- Muhammad Umair Ahmed
- Emre Sülün
- İdil Hanhan
- Mert Kara
- Metehan Saçakçı

Table of Contents

1. Introduction

1.1 Purpose of the System

Erasmus procedure in Bilkent has always been considerably disorganized and not connected due to the lack of a comprehensive system. Emailing is the main communication method during most of the Erasmus procedure, which leads to difficulties in keeping track of and accessing relevant information.

Therefore, a web-based Erasmus Application that enables all users (Students, Coordinators, etc.) to interact with each other and push the procedure forward easily without errors was needed. With this Erasmus manager, the aim is to minimize the use of emailing and other inefficient methods of communication and enable all users to be on the same page at all times. Viewing application and approval status, uploading forms, asking relevant questions, and viewing course information are all part of the system. Due to the nature of the Erasmus procedure, different kinds of users are involved in the system. The abilities, allowed features, and views depend on the user type. That implies while a student cannot approve a given form, the corresponding coordinator can. Users can also generate and send all the needed forms for every stage of the procedure. Notifications, task lists for students and coordinators, and direct messaging features are all components of the system to prevent any possible delay and difficulty.

1.2 Design Goals

1.2.1 Usability

The application will be used for many actors such as students, Erasmus coordinator, system admin, international student office, and department secretary. Different kinds of users should be using Erasmus App easily to complete their tasks. That is why Erasmus App will have a user-friendly user interface for handling complex tasks in a short time. Thus, it will be intuitively understandable for different actors.

1.2.2 Scalability

For the erasmus application, scalability is necessary for longevity. It's because over time user accounts and their information will accumulate on the database. The application should be able to meet the stress caused by large numbers of users on the application. Erasmus App will prevent this issue by increasing the server's response time when it's needed. Also, the past applicants' information will be deleted from the database. Therefore, not keeping unnecessary information on the database will increase the scalability.

1.2.3 Security

The Erasmus application will have the personal data of the students such as their mail addresses, university IDs, GPAs, and passwords. These personal pieces of information should be protected from malicious attacks. The protection of their personal data is a priority for Erasmus App. For that reason, we use a very secure and efficient hashing algorithm provided by the bcrypt npm package, and hash user passwords several times before saving them to the database.

1.2.4 Maintainability

The Erasmus application will be used for long generations. The requirements for the application process may be changed during this time. That is why Erasmus App should easily be updated when any change is needed. Because of this requirement, React is used when the system is built. Therefore, the project has a component-based project structure. This feature allows developers to add and remove components of the project easily. Therefore, it helps Erasmus App to be maintainable.

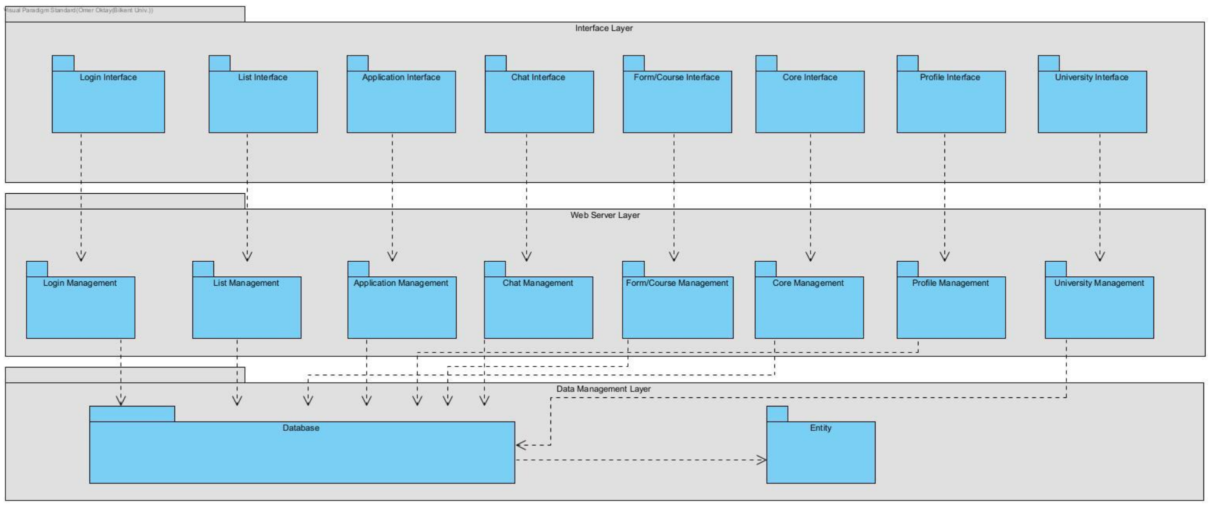
Backend Architecture will use NodeJS and ExpressJS, which are an open-source frameworks of JS and maintained by millions of developers around the globe. The backend will use the routers and controllers, that is, in case of adding new pages to the app, the only thing that needs to be done is adding appropriate routers that deal with the new HTTP requests from the new page and corresponding controllers. There will be no need to change anything in the existing system.

1.2.5 Performance

The Erasmus application should be efficient because of the user experience. Users should not get a long response time from the server. As a result of this, the application should be efficient. That is why the Erasmus App will be built with consideration of efficiency. For that reason, we have chosen node.js to implement the backend of Erasmus App. Node.js uses a non-blocking I/O model, therefore our program can continue to do other operations while waiting for one operation to be completed.

2. High-Level Software Architecture

2.1 Subsystem Decomposition



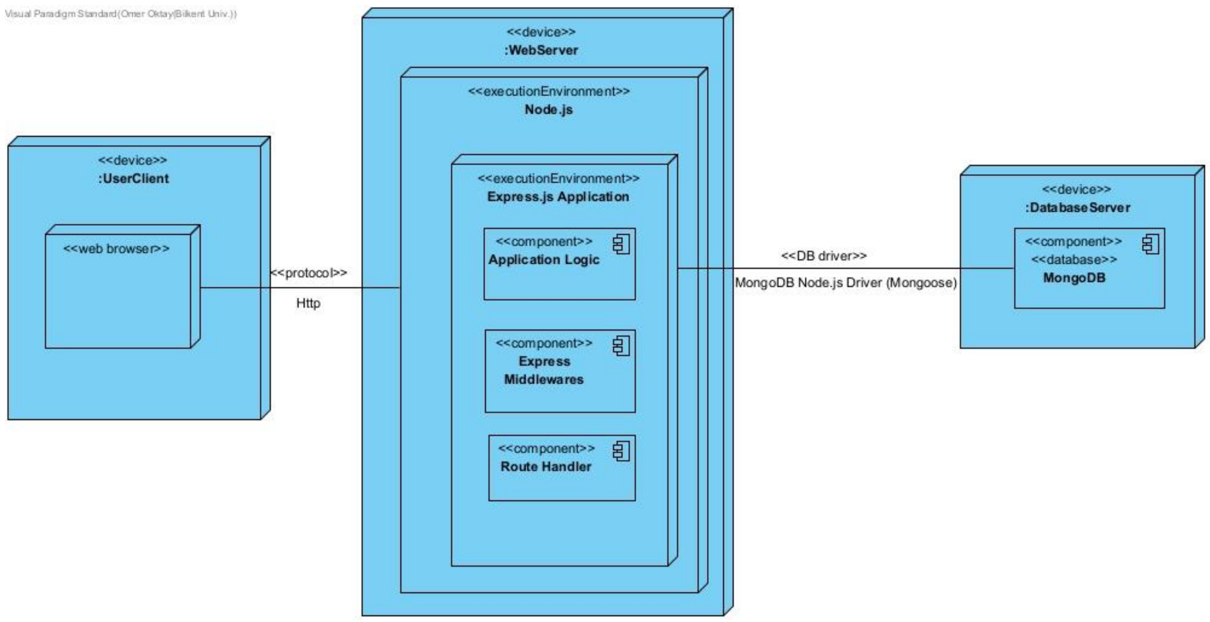
The full figure can be accessed through:
https://github.com/trelans/319-Project/blob/main/Diagrams/ClassDiagram.jpg

To provide ease for maintainability, a 3 layer architecture is used for the Erasmus App. The Data Management Layer contains Entity and Database subsystems. The Web Server Layer contains router and controller classes. The Interface Layer contains different pages of the website. The pages are connected with their designated controller and router classes which are found in the web server layer.

One may not understand the meaning of the Core Interface and Core Management. Core Interface includes main page, navigation and search bar, and notifications popup of the application and Core Management is used for manipulating the Core Interface.

2.2 Hardware/software mapping

2.2.1 Deployment Diagram



Erasmus App is a web application designed for desktops and laptops, but it can also be used by mobile phones or tablets. To use the application, a web browser will be needed. The user should have a machine powerful enough to run a web browser.

Erasmus App uses MERN stack. Therefore, React is used for its frontend, and NodeJS, ExpressJS and MongoDB are used for its backend architecture. Database will be deployed to the official cloud service of the MongoDB, MongoDB Atlas as it is the industry standard for the MongoDB applications. The source code will be deployed to Heroku cloud application platform which offers some advantages such as automatic scalability, always on website, and app metrics.

MERN stack is chosen purposely since the only additional requirement to running html and css files expected from the client is having a browser with a capability of running javascript files.

2.3 Persistent data management

Erasmus App is used MongoDB. The reasons of using MongoDB are its high compatibility with Node.js and React. Unlike traditional SQL databases that store data in fixed rows and columns, MongoDb allows documents that store data in JSON format with a flexible format. Therefore, all communication between layers, Ui, Web Server, and Database, can be done only with Json format. Suppose, Erasmus Coordinator clicks a button to create new host university for his department which triggers client code to send a json file that includes all properties user provides. Then server code can directly use the Json file as follows by using the fact that JSON and Javascript Objects are interchangeable in JS code:

```
const university = new University(req.body);
await university.save()
```

Then, the new university is created and saved in database. Also, flexible data types allows creating dynamic datas which is highly important in Erasmus app since it enables us to store datas that have mostly common properties in the same document like all users regardless of their types are stored in Users table. If SQL databases are used, then so much empty values should be in tables. As an example, that feature allows us to check for one document in case of login; otherwise, we would have to do the following process: if loginned user is a Erasmus Candidate, then check for ErasmusCandidate document etc.. The login example is a typical scenario that is occurred in many places in the Erasmus App, MongoDB prevents us from the trade-off between using same table with so many empty values (SQL Databases) and having multiple tables with no empty values. Json file will not have directly the property rather than having a keyword with a null value. As you may understand that the MongoDB documents are the equivalent of tables in SQL databases and our entity objects will be stored as documents in the database by using the mongoose library. The data again will be fetched from the database using the mongoose library. Moreover, some additional information for the authentication process will be provided from page to page by using cookies.

Additionally, client is able to manipulate objects(applications, forms etc.) in all kinds of CRUD (Create, Read, Update, Delete) operations by using http requests such as POST, DELETE, and PUT. Note that GET will not be used in Erasmus app since it needs to show every parameters it sends to database in URL that is not secure. Instead, POST with simple check for GET in server will be used.

2.4 Access control and security

Erasmus App has various types of users, and those users have different permissions in the app. Therefore, our app provides access control mechanisms to manage those permissions. Each user has an usertype, and this usertype information is checked when the user logins to the system. Then, this information will be transferred to other pages using cookies. Also, tokens will be created that will be stored both in local storage and database for a limited time when user log in to the Erasmus App. Then, every page will use this token along with usertype information to get only the required amount of data (like Application information) which is the most secure and industry standard way of getting data from the server.

Furthermore, Erasmus App prioritizes the security of user data. For that purpose, user passwords are hashed before saved to the database.

Access control matrix:

	Erasmus Coordinator	**Course Coordinator**	**Erasmus Candidate**	**Incoming Student**	**International Student Office**	**Faculty Administration Committee Member**	**System Admin**
Log in	**x**	**x**	**x**	**x**	**x**	**x**	**x**
Direct Messages	**x**	**x**	**x**	**x**	**x**	**x**	**x**
View Notifications	**x**	**x**	**x**	**x**	**x**	**x**	**x**
View Formerly	**x**	**x**	**x**	**x**	**x**	**x**	**x**

Accepted Courses							
Report Problem	**x**	**x**	**x**	**x**	**x**	**x**	**x**
Upload Course Transfer Form		**x**					
Activate Application Period	**x**						
Upload Erasmus Placement Results	**x**						
Notify Placement	**x**						
Nominate Students	**x**						
Confirm New Courses	**x**						
Reject New Courses	**x**						
Upload Signature	**x**		**x**				
Accept Form	**x**						
Reject Form	**x**						
Display Logs	**x**						**x**
Add University	**x**						**x**
Remove University	**x**						**x**
Edit University Information	**x**						**x**
Approve Course Transfer						**x**	

Form							
Reject Course Transfer Form						**x**	
See Confirmed Courses			**x**				
Approve Pre Approval Form						**x**	
Reject Pre Approval Form						**x**	
Submit Pre Approval Form			**x**				
Submit Learning Agreement Form			**x**				
View Application Status			**x**				
Review University			**x**	**x**			
Discard Placement			**x**				
Remove From Pre Approval Form			**x**				
Request New Course			**x**				
Add to Pre Approval Form			**x**				
Cancel Application					**x**		

2.5 Boundary conditions

2.5.1 Initialization

Erasmus App has 4 kinds of initialization: first, year-based, client, and subsystem initialization. For the first initialization, the admin user will add all departments in Bilkent to the database and create users except for the students. For the year-based initialization, Erasmus coordinators will upload an excel file that contains applicant info which will be processed by the Erasmus app to create user accounts for the applied students. Also erasmus coordinator will update University infos as necessary before students can use the application. For the client initialization, there will be no specific initialization steps other than login into the app since once the Erasmus App deployed into cloud platforms like Heroku, the application will be run on a web server 24/7. Thus, the application does not require any sort of downloading / installing. Therefore, an individual who has internet connection can access the website which is an advantage in terms of practicality. In order for a user to sign in into their account, they need to enter their email and password which is then checked from the database. Before being logged in, a user can only see a few pages such as the login page, sign up page and forgot password page. After a successful login process, the user meets other pages which are filled with actual data that is fetched from the database. For the subsystem initialization, the connected subsystems will be initialized along with the initialized subsystem such as when a student type of user is created, app will automatically create an application for that student and creation of the application will trigger the creation of the necessary forms.

2.5.2 Termination

Termination can be divided into 4 kinds in the application: termination of an account, termination of the system, termination of the year, and termination of subsystem. Termination of an account is a process which lets go of the unused and not needed data which are selected with various attributes given to the accounts. Termination of the system is decided by an admin and all the actual data which are stored in the database is saved before the termination to obstruct any kind of data loss. There will be yearly termination to delete the students' info from the database for the ones that successfully complete the Erasmus process. Erasmus Coordinators will decide how many years should have passed to delete applicant info from the system. For the termination of the subsystem, all connected subsystems will be terminated. For example, when an erasmus candidate decides to discard his application, all forms connected to the application will be deleted before deleting the application. Then erasmus candidates will be placed into the waiting list.

2.5.3 Failure

Erasmus App is run on MongoDB which uses the advantages of sharding, high availability and scalability. Sharding allows MongoDB to partition the database into many small databases while every one of the databases can be stored in different machines thanks to scalability. Also MongoDB has features such as replication that creates many virtual read-only tables and one writable table that makes the app less fail-prone thanks to MongoDB's high availability. These reasons make Erasmus App less likely to fail due to database errors but in case of the failure, since we will deploy the app in MongoDB Atlas, the cloud system of the MongoDB, our app will be quickly recovered by using another machine that will be initialized by MongoDB Atlas.

In case of server failures, the same principle of recovering the app will hold since application source codes will be deployed into cloud platforms like Heroku which promise an always up and running server.

In case of wrong HTTP responses, the correct HTTP Error codes will be sent to the client side and for debugging purposes, terminal will be used.

3. Low-Level Design

3.1 Object design trade-offs

Functionality versus Maintainability:

Erasmus App uses various npm packages to increase the functionality. However, those npm packages change frequently, therefore Erasmus App can require frequent maintenance.

Security versus Performance:

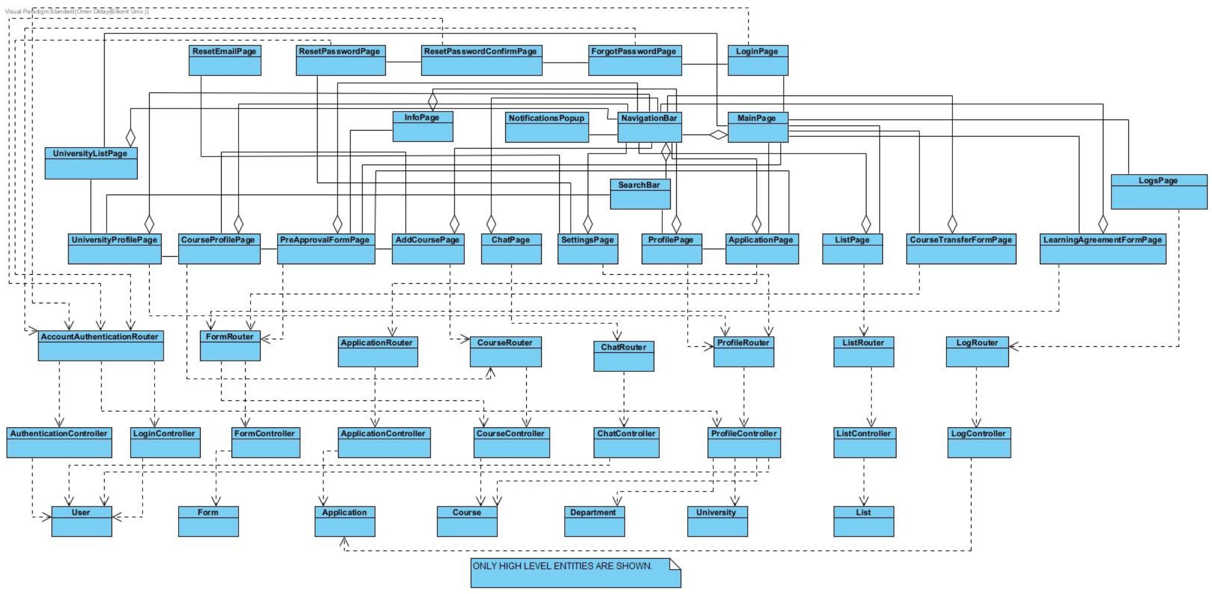
Erasmus App uses local storage for storing the token of the current logged in user and its user type. Therefore, when the pages are loaded, the required amount of info came from the server every time instead of storing them in the browser. While it increases security of the application, it makes a bit of a decrease in performance.

Functionality versus Usability:

Erasmus App is used by many types of users, and this increases the functionality of our app. Therefore, as the functionality increases, complexity of the Erasmus App increases as well. Hence, this might decrease the usability of Erasmus App.

3.2 Final Object Design

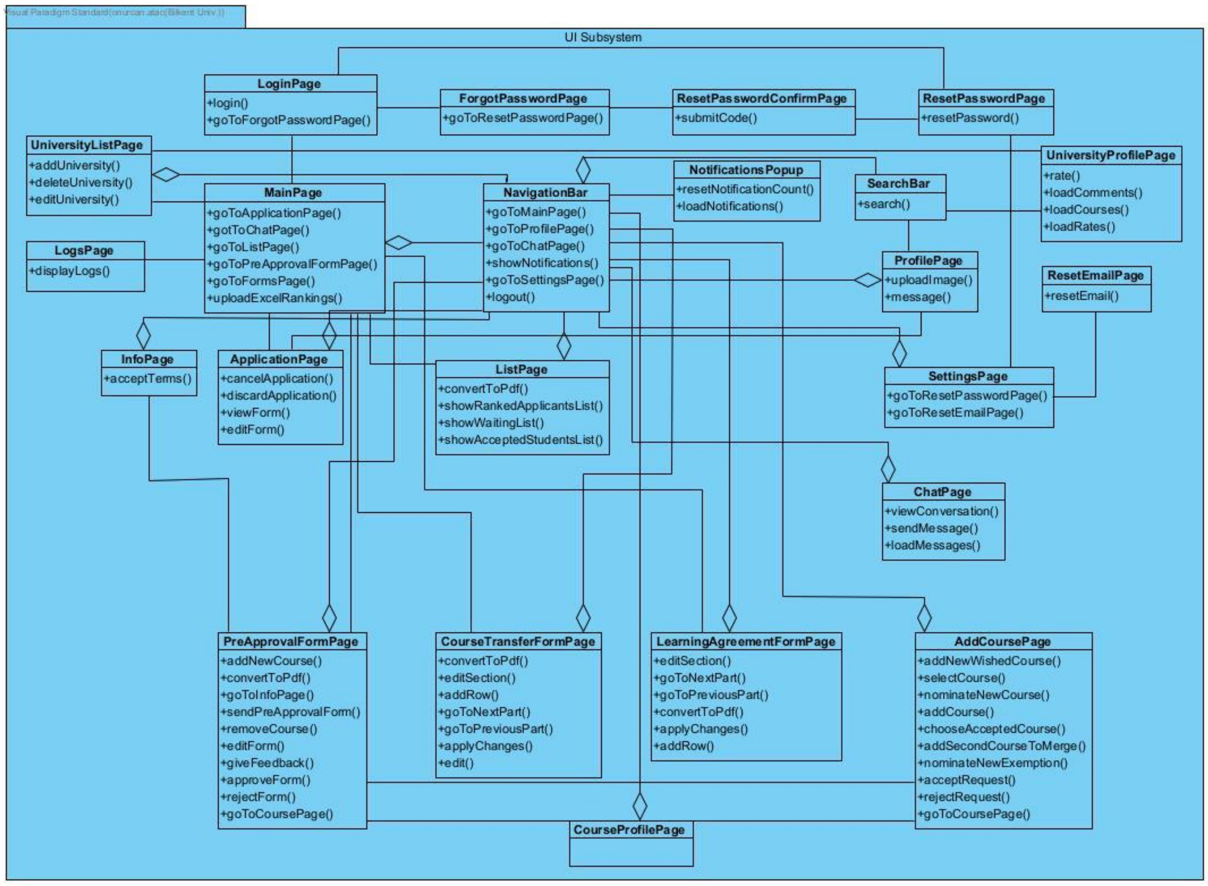
The following diagram is Erasmus Apps' final object design diagram.



3.3 Layers

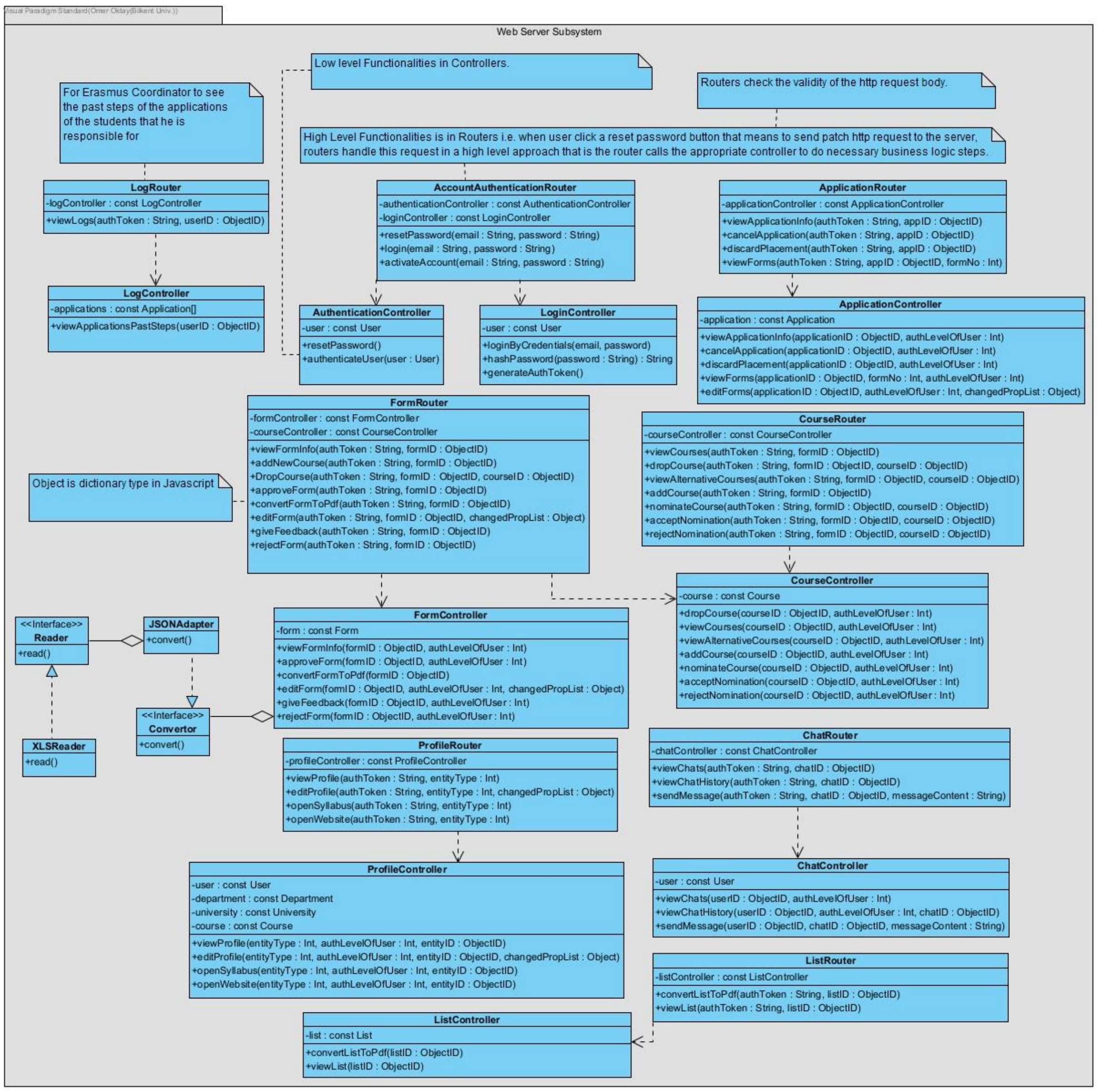
3.3.1 User Interface Management Layer

This layer represents a boundary object, and it provides a connection between the user and the web server layer. All classes are implemented using the React library of Javascript. The source code is compiled into HTML files, which then makes it possible for the user to interact with the application.



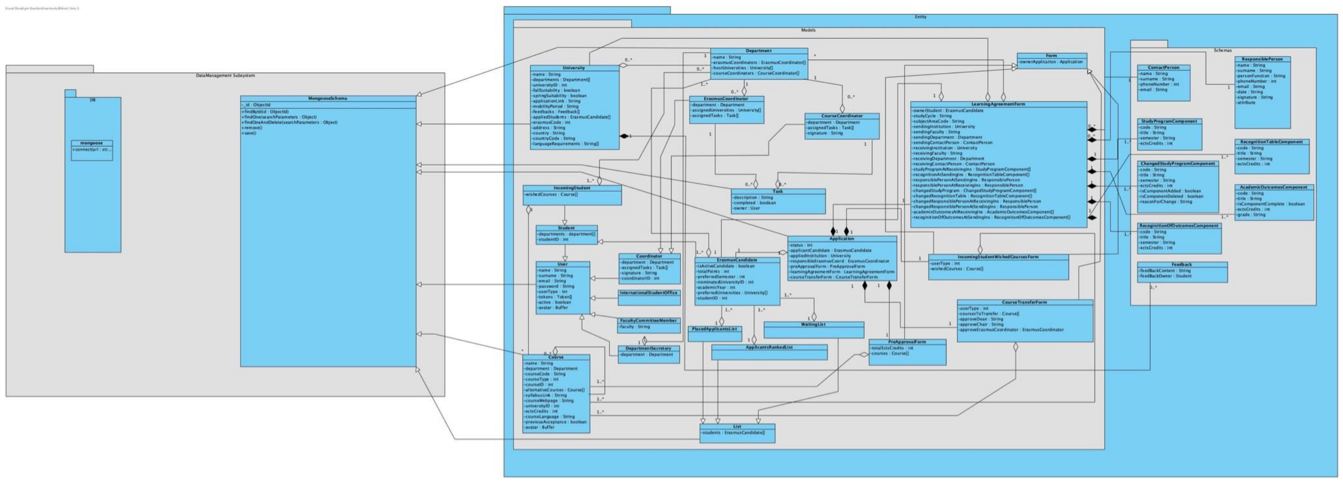
3.3.2 Web Server Management Layer

Web Server Management Layer is mainly responsible for handling the incoming requests from the client side. It further divides into two subparts: Routers and Controllers. Routers deal with the high level functionality of the backend functions that use the controllers to use the business logic operations to send an appropriate response to the client side. Controllers do their job by using the Data Management Layer entities.


Visual Paradigm Standard(Omer Oktay(Bilkent Univ.))
Web Server Subsystem
Low level Functionalities in Controllers.
Routers check the validity of the http request body.
For Erasmus Coordinator to see the past steps of the applications of the students that he is responsible for
High Level Functionalities is in Routers i.e. when user click a reset password button that means to send patch http request to the server, routers handle this request in a high level approach that is the router calls the appropriate controller to do necessary business logic steps.
LogRouter
-logController : const LogController
+viewLogs(authToken : String, userID : ObjectID)
AccountAuthenticationRouter
-authenticationController : const AuthenticationController
-loginController : const LoginController
+resetPassword(email : String, password : String)
+login(email : String, password : String)
+activateAccount(email : String, password : String)
ApplicationRouter
-applicationController : const ApplicationController
+viewApplicationInfo(authToken : String, appID : ObjectID)
+cancelApplication(authToken : String, appID : ObjectID)
+discardPlacement(authToken : String, appID : ObjectID)
+viewForms(authToken : String, appID : ObjectID, formNo : Int)
LogController
-applications : const Application[]
+viewApplicationsPastSteps(userID : ObjectID)
AuthenticationController
-user : const User
+resetPassword()
+authenticateUser(user : User)
LoginController
-user : const User
+loginByCredentials(email, password)
+hashPassword(password : String) : String
+generateAuthToken()
ApplicationController
-application : const Application
+viewApplicationInfo(applicationID : ObjectID, authLevelOfUser : Int)
+cancelApplication(applicationID : ObjectID, authLevelOfUser : Int)
+discardPlacement(applicationID : ObjectID, authLevelOfUser : Int)
+viewForms(applicationID : ObjectID, formNo : Int, authLevelOfUser : Int)
+editForms(applicationID : ObjectID, authLevelOfUser : Int, changedPropList : Object)
FormRouter
-formController : const FormController
-courseController : const CourseController
+viewFormInfo(authToken : String, formID : ObjectID)
+addNewCourse(authToken : String, formID : ObjectID)
+DropCourse(authToken : String, formID : ObjectID, courseID : ObjectID)
+approveForm(authToken : String, formID : ObjectID)
+convertFormToPdf(authToken : String, formID : ObjectID)
+editForm(authToken : String, formID : ObjectID, changedPropList : Object)
+giveFeedback(authToken : String, formID : ObjectID)
+rejectForm(authToken : String, formID : ObjectID)
Object is dictionary type in Javascript
CourseRouter
-courseController : const CourseController
+viewCourses(authToken : String, formID : ObjectID)
+dropCourse(authToken : String, formID : ObjectID, courseID : ObjectID)
+viewAlternativeCourses(authToken : String, formID : ObjectID, courseID : ObjectID)
+addCourse(authToken : String, formID : ObjectID)
+nominateCourse(authToken : String, formID : ObjectID, courseID : ObjectID)
+acceptNomination(authToken : String, formID : ObjectID, courseID : ObjectID)
+rejectNomination(authToken : String, formID : ObjectID, courseID : ObjectID)
CourseController
-course : const Course
+dropCourse(courseID : ObjectID, authLevelOfUser : Int)
+viewCourses(courseID : ObjectID, authLevelOfUser : Int)
+viewAlternativeCourses(courseID : ObjectID, authLevelOfUser : Int)
+addCourse(courseID : ObjectID, authLevelOfUser : Int)
+nominateCourse(courseID : ObjectID, authLevelOfUser : Int)
+acceptNomination(courseID : ObjectID, authLevelOfUser : Int)
+rejectNomination(courseID : ObjectID, authLevelOfUser : Int)
<<Interface>>
Reader
+read()
JSONAdapter
+convert()
FormController
-form : const Form
+viewFormInfo(formID : ObjectID, authLevelOfUser : Int)
+approveForm(formID : ObjectID, authLevelOfUser : Int)
+convertFormToPdf(formID : ObjectID)
+editForm(formID : ObjectID, authLevelOfUser : Int, changedPropList : Object)
+giveFeedback(formID : ObjectID, authLevelOfUser : Int)
+rejectForm(formID : ObjectID, authLevelOfUser : Int)
<<Interface>>
Convertor
+convert()
XLSReader
+read()
ProfileRouter
-profileController : const ProfileController
+viewProfile(authToken : String, entityType : Int)
+editProfile(authToken : String, entityType : Int, changedPropList : Object)
+openSyllabus(authToken : String, entityType : Int)
+openWebsite(authToken : String, entityType : Int)
ChatRouter
-chatController : const ChatController
+viewChats(authToken : String, chatID : ObjectID)
+viewChatHistory(authToken : String, chatID : ObjectID)
+sendMessage(authToken : String, chatID : ObjectID, messageContent : String)
ProfileController
-user : const User
-department : const Department
-university : const University
-course : const Course
+viewProfile(entityType : Int, authLevelOfUser : Int, entityID : ObjectID)
+editProfile(entityType : Int, authLevelOfUser : Int, entityID : ObjectID, changedPropList : Object)
+openSyllabus(entityType : Int, authLevelOfUser : Int, entityID : ObjectID)
+openWebsite(entityType : Int, authLevelOfUser : Int, entityID : ObjectID)
ChatController
-user : const User
+viewChats(userID : ObjectID, authLevelOfUser : Int)
+viewChatHistory(userID : ObjectID, authLevelOfUser : Int, chatID : ObjectID)
+sendMessage(userID : ObjectID, chatID : ObjectID, messageContent : String)
ListRouter
-listController : const ListController
+convertListToPdf(authToken : String, listID : ObjectID)
+viewList(authToken : String, listID : ObjectID)
ListController
-list : const List
+convertListToPdf(listID : ObjectID)
+viewList(listID : ObjectID)

3.3.3 Data Management Layer

Data management layer is handled with the help of the Mongoose library. Our entities are inherited using the Mongoose Scheme structure, therefore they have built in functions to manage the database operations.



The full figure can be accessed through:
https://github.com/trelans/319-Project/blob/main/Diagrams/modelDiagram.jpg

3.4 Packages

3.4.1 Packages Introduced by Developers

3.4.1.1 Models

This package contains the models that are used to ease the use of the CRUD operations on Database.

3.4.1.2 Middleware

This package contains the middle processes that need to be done before further processing the request from client-side.The following algorithm is an example of the middleware usage. First, the backend receives the get request for user data. Then the authentication middleware function checks whether the token that is sent by client-size is valid. If authentication fails, http status 401 (Authentication Error) will be sent to the client side and the processing of the request will not be started. Otherwise, the request will be processed.

3.4.1.3 Routers

This package contains the routers that handle the http traffic of the app, that is, they are responsible for getting http requests from the client side and giving appropriate http responses.

3.4.1.4 Controllers

This package contains the controllers that handle the business logic of the app, that is, they are responsible for making appropriate operations according to received http requests from the client side. Routers use controllers to get the appropriate responses and routers send the prepared responses.

3.4.2 External Library Packages

3.4.2.1 Jsonwebtoken

This npm package is used to create authentication tokens for the users of the application.

3.4.2.2 Bcrypt

This npm package is used to hash the passwords of the user before saving them to the database.

3.4.2.3 Mongoose

This npm package is used to manage the database operations like connecting to the database or fetching data from the database etc. This package also helps to model our objects for the database, and save them to the database.

3.4.2.4 Express

This npm package helps to create a simple http server.

3.4.2.5 Validator

This npm package is used to validate the strings, mainly to check if the given email is valid or not.

3.4.2.6 react-router-dom

This package is used in the frontend development, which is done by React. Constant Link is imported in order to link pages, which enables the transition between pages.

3.4.2.7 react-router

This package is used in the frontend development, which is done by React. Route and Routes functions are used in order to assign certain paths to pages.

3.4.2.8 convert-excel-to-json

This npm package helps to convert the datas in excel to json data format.

3.5 Design Patterns

3.5.1 Strategy Pattern

As the features of our application increases, our software gets more and more complex and harder to maintain. Thus, we have chosen to use strategy design pattern to deal with the increasing complexity. With the help of the strategy design pattern, we can decide the behaviour of our algorithms at run time.

3.5.2 Adapter Pattern

In our application, we consistently use JSON data format to fetch data, and send data. Hence, it is important that other data formats adapted to JSON data format to be easily processed. Therefore, we use adapter design pattern to convert other data formats to JSON data format. As a concrete example, when the applicants list is uploaded into the application as an excel file, excel file is converted into JSON file by using adapter pattern.

3.5.3 HOC Pattern

Higher-order-component (HOC) design pattern is used when necessary while implementing components that are going to be used with minor changes during the implementation of multiple pages. HOC pattern is very suitable for React, since React encourages component behavior similar to HTML format. Using components also makes the code more maintainable and readable. As examples, Card and NavigationBar are higher order components that are used in the user interface layer. HOC is also used in external libraries which are included in the user interface, such as Grid, Table and PhoneInput.

4. Glossary and References

Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110-0.