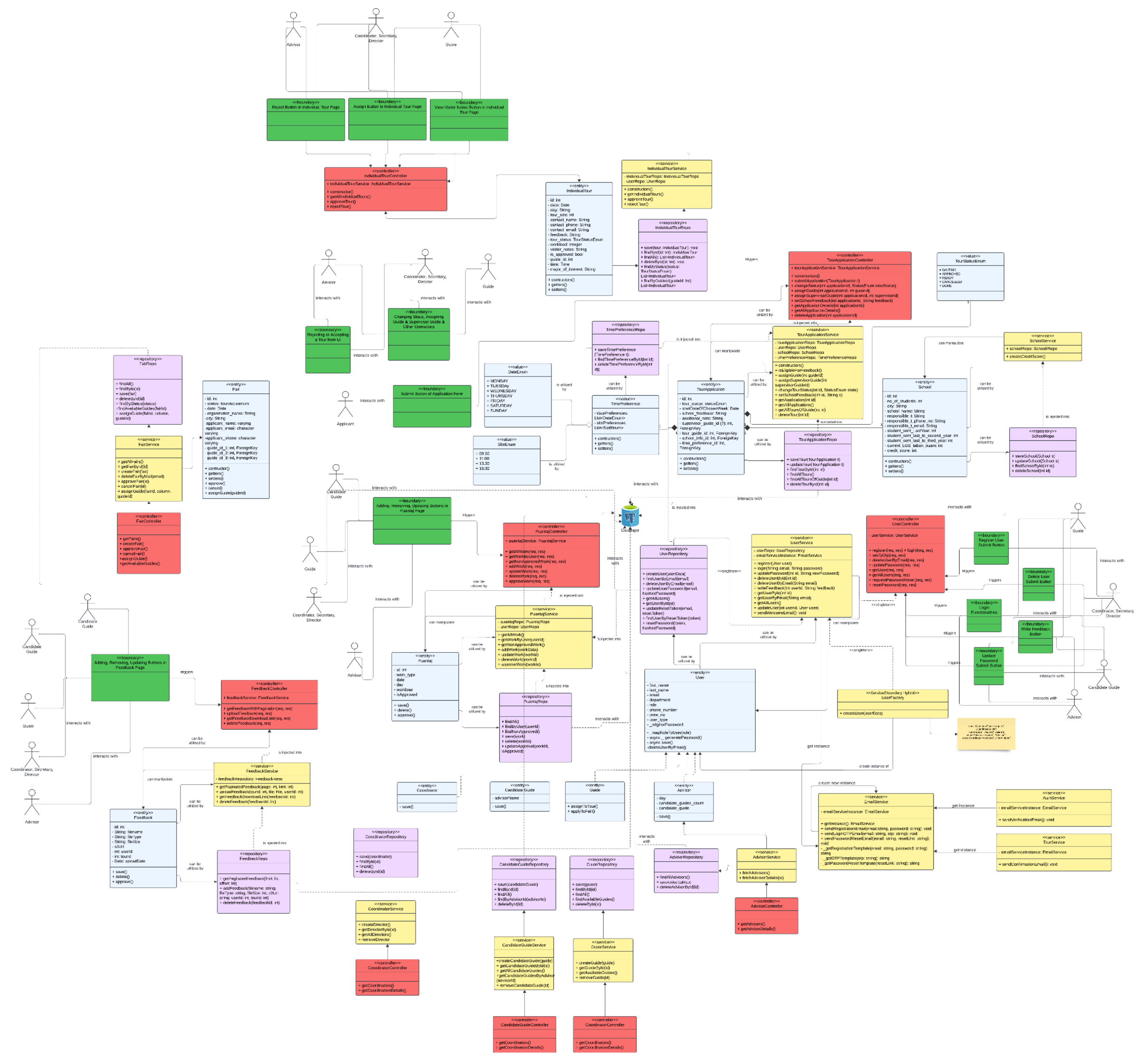
CS 319 Object-Oriented Software Engineering
Project Deliverable 5 (D5) - S2T7 - agora

Team Members:

- Bertan Uran - 22102541
- Egehan Yıldız - 22203014
- Ekin Köylü - 22103867
- Emre Yazıcıoğlu - 22201668
- İlke Latifoğlu - 22203818
- Merve Güleç - 22103231

CONTENTS

Final Class Diagram



https://lucid.app/lucidchart/cb5672a1-217e-49f6-8119-09ecca92768d/edit?viewport_loc=-4133%2C-410%2C7166%2C3345%2CHWEp-vi-RSFO&invitationId=inv_1228cd1c-e43e-4505-be03-9722e1ad60f6

Design Patterns

1. Singleton Pattern

The Singleton pattern ensures that a class has only one instance throughout the application's lifecycle and provides a global point of access to that instance. This design pattern is particularly useful when managing shared resources like configuration managers, logging services, or service instances, ensuring they remain consistent across the entire application. In the code, EmailService, OTPService and UserService are Singleton patterns because they provide reusable methods that are accessed by multiple parts of the application and these services do not need to maintain separate instances for each request but rather share a single instance to reduce redundancy and ensure consistency.

1.1 EmailService

The EmailService class is responsible for sending emails with specific templates for registration, OTP verification, and password reset.

Consistency: Email templates and sending logic are centralized and reused across the application.

Resource Efficiency: No need to initialize a new instance for every email-related operation.

Simplified Maintenance: Changes to the email-sending logic only affect the singleton instance, making updates straightforward.

```
class EmailService {
  // Private methods with service logic
  async sendRegistrationEmail() {...}
  async sendLoginOTPEmail() {...}
  async sendPasswordResetEmail() {...}
  _getRegistrationTemplate() {...}
  _getOTPTemplate() {...}
  _getPasswordResetTemplate() {...}
}

// Single instance exported
module.exports = new EmailService();
```

1.2 OTPService

The OTPService class is responsible for generating and verifying One-Time Passwords (OTPs).

Shared Responsibility: OTP generation and verification logic should remain consistent across requests.

Thread Safety: Centralized management ensures the OTP's lifecycle is predictable.

Global Access: Other parts of the application can easily reuse OTP-related methods without creating multiple instances.

```
class OTPService {
  async generateAndSaveOTP() {...}
  async verifyOTP() {...}
  async clearOTP() {...}
}

// Single instance exported
module.exports = new OTPService();
```

1.3 UserService

The UserService class handles all operations related to users, such as creating, finding, updating, and deleting users. It is implemented as a Singleton to ensure that these operations are consistent and efficient across the application.

Why Singleton for UserService?

Consistency: Ensures all user-related logic (e.g., password updates, user creation) is centralized and behaves the same everywhere.

Efficiency: Prevents creating multiple instances, saving resources, and improving performance.

Simpler Maintenance: Updates to user logic only need to be made in one place.

Global Access: Any part of the app can use UserService without creating duplicate instances.

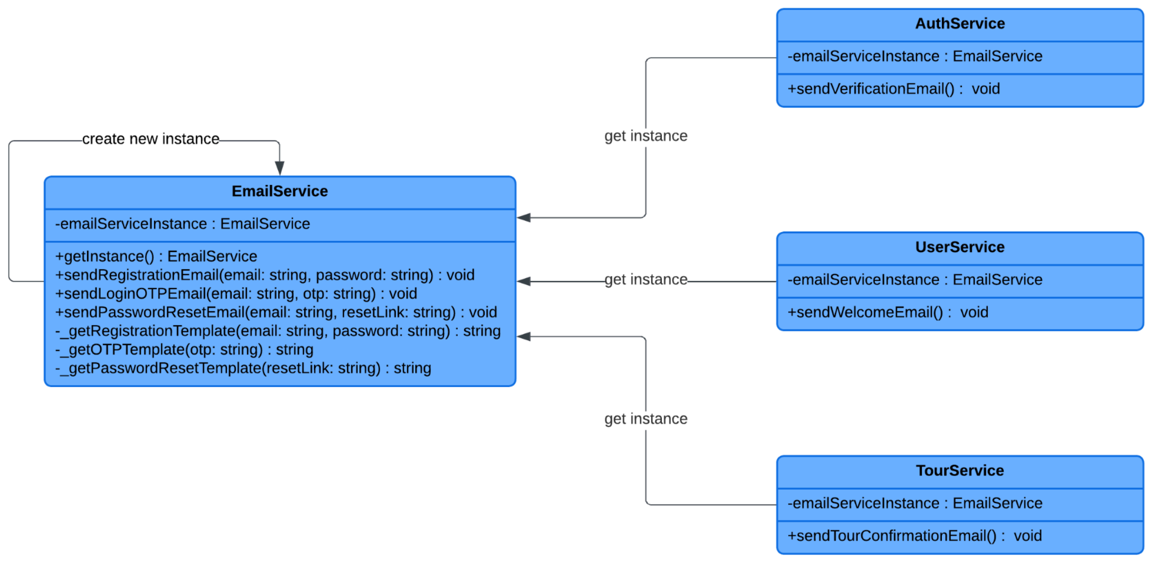
```
class UserService {
  async createUser() {...}
  async findUserByEmail() {...}
  async deleteUserByEmail() {...}
  async updateUserPassword() {...}
  async getAllusers() {...}
  async getUserById() {...}
  async requestPasswordReset() {...}
  async resetPassword() {...}
  async handleRoleSpecificLogic() {...}
  async createAdvisor() {...}
  async createCandidateGuide() {...}
  async verifyPassword() {...}
}

// Single instance exported
module.exports = new UserService();
```

Key aspects of the Singleton implementation
Each service class is instantiated once at export. The same instance is reused across imports. Maintains a consistent state across the application. Provides global access point to service functionality.

Benefits of using Singleton here
Single source of truth for email sending, OTP management, and user operations. Prevents multiple instances from interfering with each other. Efficient resource usage since only one instance exists. Consistent handling of operations across the application

1.4 Structural Diagram of Singleton Design


AuthService
-emailServiceInstance : EmailService
+sendVerificationEmail() : void
create new instance
get instance
EmailService
-emailServiceInstance : EmailService
+getInstance() : EmailService
+sendRegistrationEmail(email: string, password: string) : void
+sendLoginOTPEmail(email: string, otp: string) : void
+sendPasswordResetEmail(email: string, resetLink: string) : void
-_getRegistrationTemplate(email: string, password: string) : string
-_getOTPTemplate(otp: string) : string
-_getPasswordResetTemplate(resetLink: string) : string
get instance
UserService
-emailServiceInstance : EmailService
+sendWelcomeEmail() : void
get instance
TourService
-emailServiceInstance : EmailService
+sendTourConfirmationEmail() : void

2. Factory Pattern

The Factory pattern provides a way to create objects without specifying their concrete classes. Instead, the creation logic is centralized in a Factory class, which determines the type of object to create based on input parameters or conditions.

Encapsulation: Encapsulates the logic for creating objects, making the code more modular and easier to maintain.

Flexibility: Simplifies the addition of new object types by modifying the factory rather than the client code.

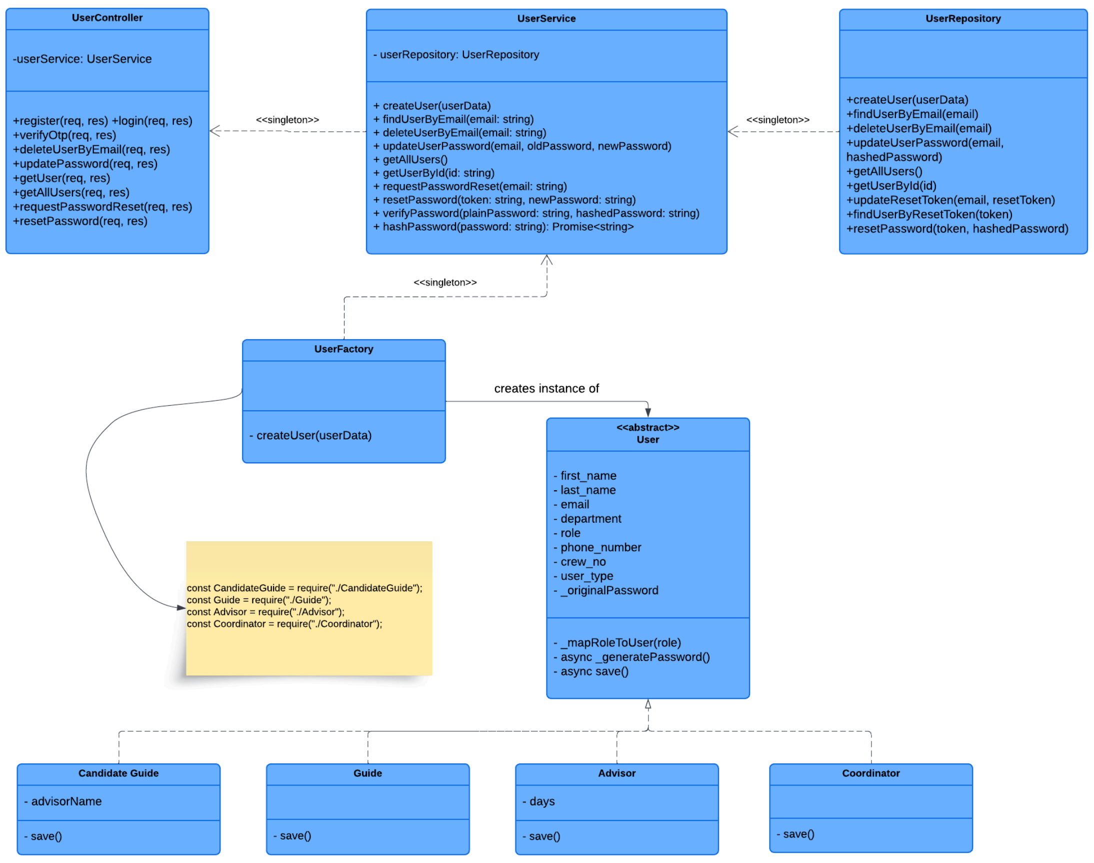
Decoupling: The client code doesn't need to know the details of object creation, which promotes loose coupling.

Centralized Logic: The UserFactory class encapsulates the logic for creating different user roles.

Reusability: The factory can be reused wherever user creation logic is needed.

Extendability: Adding a new user role only requires updating the factory and creating a new class.

2.1 Structure Diagram of Factory Pattern


UserController
-userService: UserService
+register(req, res) +login(req, res)
+verifyOtp(req, res)
+deleteUserByEmail(req, res)
+updatePassword(req, res)
+getUser(req, res)
+getAllUsers(req, res)
+requestPasswordReset(req, res)
+resetPassword(req, res)
<<singleton>>
UserService
- userRepository: UserRepository
+ createUser(userData)
+ findUserByEmail(email: string)
+ deleteUserByEmail(email: string)
+ updateUserPassword(email, oldPassword, newPassword)
+ getAllUsers()
+ getUserById(id: string)
+ requestPasswordReset(email: string)
+ resetPassword(token: string, newPassword: string)
+ verifyPassword(plainPassword: string, hashedPassword: string)
+ hashPassword(password: string): Promise<string>
<<singleton>>
UserRepository
+createUser(userData)
+findUserByEmail(email)
+deleteUserByEmail(email)
+updateUserPassword(email, hashedPassword)
+getAllUsers()
+getUserById(id)
+updateResetToken(email, resetToken)
+findUserByResetToken(token)
+resetPassword(token, hashedPassword)
<<singleton>>
UserFactory
- createUser(userData)
creates instance of
<<abstract>>
User
- first_name
- last_name
- email
- department
- role
- phone_number
- crew_no
- user_type
- _originalPassword
- _mapRoleToUser(role)
- async _generatePassword()
- async save()
Candidate Guide
- advisorName
- save()
Guide
- save()
Advisor
- days
- save()
Coordinator
- save()

```
const CandidateGuide = require("./CandidateGuide");
const Guide = require("./Guide");
const Advisor = require("./Advisor");
const Coordinator = require("./Coordinator");
```

2.2 Implementation of Factory Design

```
const CandidateGuide = require("../models/CandidateGuide");
const Guide = require("../models/Guide");
const Advisor = require("../models/Advisor");
const Coordinator = require("../models/Coordinator");

class UserFactory {
 createUser(userData) {
   switch (userData.role.toLowerCase()) {
     case "candidate guide":
       return new CandidateGuide(userData);
     case "guide":
       return new Guide(userData);
     case "advisor":
       return new Advisor(userData);
     case "coordinator":
       return new Coordinator(userData);
     default:
       throw new Error(`Invalid user role: ${userData.role}`);
   }
 }
}

module.exports = new UserFactory();
```