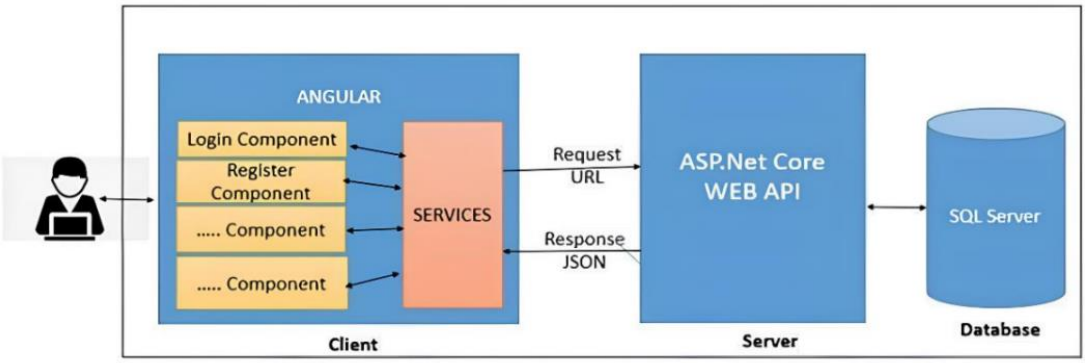
Introduction:

EduHub is designed to empower learners by providing access to comprehensive educational resources and tools. Students can explore course materials, engage in self-paced learning, and seek clarification through inquiries. Educators, on the other hand, utilize EduHub to create and manage courses efficiently, curating content tailored to student requirements. Additionally, EduHub facilitates seamless communication between students and educators, allowing for inquiries to be submitted and addressed promptly. With its focus on personalized learning experiences and effective communication channels, EduHub enables students to thrive academically while offering educators the means to facilitate impactful teaching.

System Architecture Diagram:

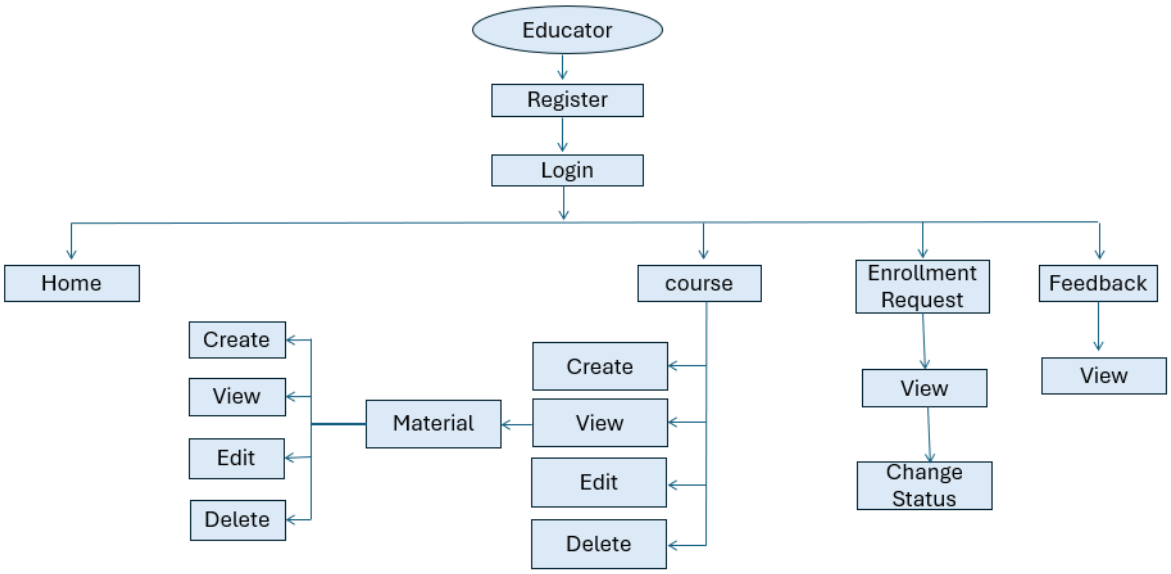


Users of the System:

1. Educator
2. Student

<u>Educator Actions</u>

Flow diagram:



Create Course: Educators can create new courses by providing the title, description, start date, and end date. They are automatically assigned as the educator of the course.

Manage Course: Educators can edit course details such as title, description, start date, and end date. They can also add or remove materials to the course.

View Enrollments: Educators can view the list of students enrolled in the courses. They can see enrollment statuses (e.g., Accepted, Rejected).

Upload Materials: Educators can upload URL learning materials/resources to their courses.

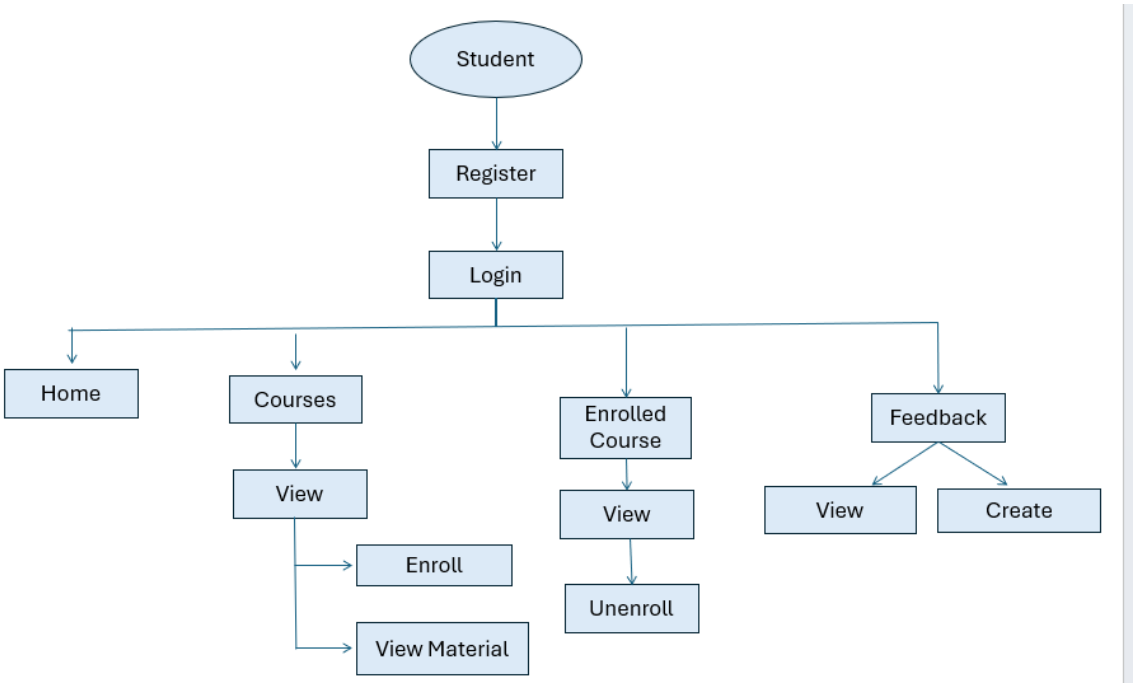
Respond to Enquiries: Educators can respond to enquiries submitted by students regarding the course. They can provide answers, solutions, or additional information to address the students' queries.

View All Students: Educators can view a list of all students.

View All Courses: Educators can view a list of all courses.

Feedback: Educators should have the option to view student's feedback.

<u>Student Actions</u>
Flow diagram:



Enroll in Course: Students can browse available courses and enroll in the ones they areas interested in. They must wait for the enrollment request to be accepted by the educator.

View Course Materials: Once enrolled, students can access course materials uploaded by the educator. They can view resources provided for the course.
View His Enrolled Courses: Students can view the list of courses they have enrolled in. They can see details such as course title, description, start date, and end date.

Submit Enquiries: Students can submit enquiries regarding the course to educators.

View Enquiries: Students can view enquiries they have submitted for a particular course. They can see the status of their enquiries and any responses provided by educators.

Feedback: Students should have the option to view and post their feedback.

Modules of the Application:

Educator:

1. Register
2. Login
3. Home
4. Course
5. Enrollment Request

6. Feedbacks

Student:

1. Register
2. Login
3. Home
4. Courses
5. Enrolled Course
6. Feedback

Technology Stack

Front End

Angular 10+, HTML, CSS

Back End

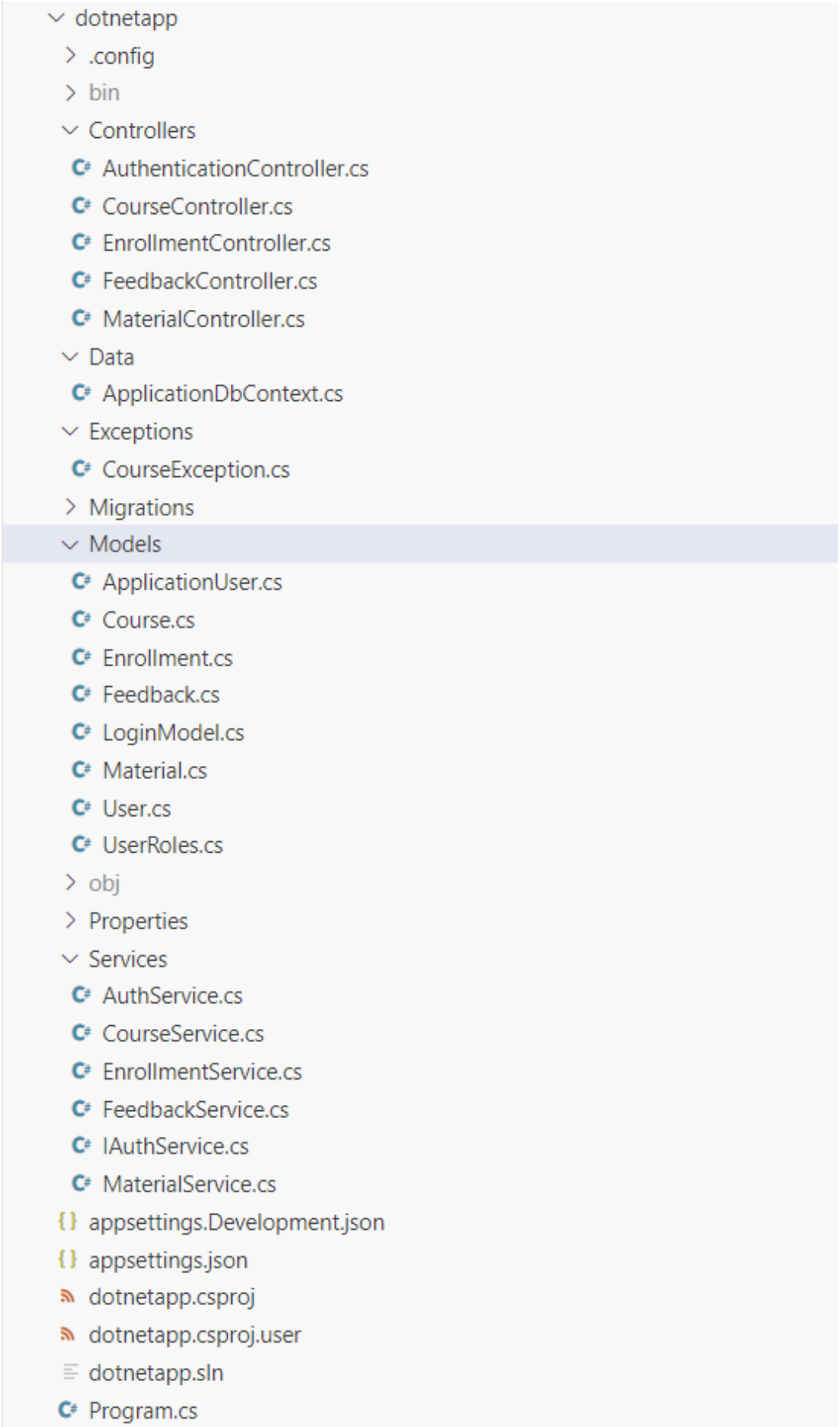
.NET Web API, EF Core, Microsoft SQL Server Database.

Application assumptions:

1. The login page should be the first page rendered when the application loads.
2. Manual routing should be restricted by implementing Auth Guard, utilizing the canActivate interface. For example, if the user enters as http://localhost:8080/dashboard or http://localhost:8080/user the page should not navigate to the corresponding page instead it should redirect to the login page.
3. Unless logged into the system, the user cannot navigate to any other pages.
4. Logging out must again redirect to the login page.

Backend Requirements:

Create folders named as **Models, Controllers, Services, Data and Exceptions** inside **dotnetapp** as mentioned in the below screenshot.


dotnetapp
.config
bin
Controllers
AuthenticationController.cs
CourseController.cs
EnrollmentController.cs
FeedbackController.cs
MaterialController.cs
Data
ApplicationDbContext.cs
Exceptions
CourseException.cs
Migrations
Models
ApplicationUser.cs
Course.cs
Enrollment.cs
Feedback.cs
LoginModel.cs
Material.cs
User.cs
UserRoles.cs
obj
Properties
Services
AuthService.cs
CourseService.cs
EnrollmentService.cs
FeedbackService.cs
IAuthService.cs
MaterialService.cs
appsettings.Development.json
appsettings.json
dotnetapp.csproj
dotnetapp.csproj.user
dotnetapp.sln
Program.cs

ApplicationDbContext: (**/Data/ApplicationDbContext.cs**)

Inside **Data** folder create **ApplicationDbContext** file with the following **DbSet** mentioned below.

```
public DbSet<Course> Courses{ get; set; }
public DbSet<Enrollment> Enrollments{ get; set; }
public DbSet<Material> Materials{ get; set; }
public DbSet<Feedback> Feedbacks { get; set; }
public DbSet< User> Users{ get; set; }
```

Model Classes:

Inside **Models** folder create all the model classes mentioned below.

Namespace: All the model classes should located within the **dotnetapp.Models** namespace.

User (Models / User.cs):
This class stores the user role (**Educator** or **Student**) and all user information.
Properties:

- UserId: int
- Email: string
- Password: string
- Username: string
- MobileNumber: string
- UserRole: string (**Educator/ Student**)

Feedback (Models / Feedback.cs):
This class represents feedback submitted by users.
Properties:

- FeedbackId: int
- UserId: int
- User?: User
- FeedbackText: string
- Date: DateTime

Course (Models / Course.cs):

This class represents adding the course by educator.

Properties:

- CourseId: int
- Title: string
- Description: string
- CourseStartDate: DateTime
- CourseEndDate: DateTime
- Category: string
- Level: string

Enrollment (Models / Enrollment.cs):

This class represents enrolling the course by student.

Properties:

- EnrollmentId: int
- UserId: int
- User?: User
- CourseId: int
- Course? Course
- EnrollmentDate: DateTime
- Status: string

Material (Models / Material.cs):

This class represents adding the material by educator.

Properties:

- MaterialId: int
- CourseId: int
- Course? Course
- Title: string
- Description: string
- URL: string
- UploadDate: DateTime
- ContentType: string

LoginModel (Models / LoginModel.cs):

This class stores the email and password to authenticate the user during login.

Properties:

- Email: string
- Password: string

UserRoles (Models / UserRoles.cs):

This class defines constants for user roles.

Constants:

1. Educator: string
2. Student: string

ApplicationUser (Models / ApplicationUser.cs):

This class represents a user in the application, inheriting from **IdentityUser** class.

Property:

- Name: string (Max length 30)

Exceptions:

CourseException (Exceptions CourseException.cs)

1. Inside **Exceptions** folder create the exception file named **CourseException (CourseException.cs).**
2. **Purpose**: The **CourseException** class provides a mechanism for handling exceptions related to adding the course operations within the application.
3. **Namespace**: It should located within the **dotnetapp.Exceptions** namespace.
4. **Inheritance**: Inherits from the base **Exception** class, enabling it to leverage existing exception handling mechanisms.
5. **Constructor**: Contains a constructor that accepts a message parameter, allowing to specify custom error messages when throwing exceptions.

Example Usage:

You might throw a **CourseException** in the following scenarios:

When attempting to add a course that overlaps with an existing one for the same course name throws the exceptions as **"`A course with the same name already exists`".**

Important note:

Implement database logic only in the **service file functions without using try-catch**. Use **try-catch only in the controller files** and call the service file functions inside it.

Services:

Inside "**Services"** folder create all the services file mentioned below.

Namespace: All the services file should located within the **dotnetapp.Services** namespace.

CourseService (Services / CourseService.cs):

The CourseService class provides CRUD operations (Create, Read, Update, Delete) for managing Course entities in the application, ensuring proper handling and validation of course data within the context of an ApplicationDbContext. It simplifies and abstracts the interaction with the database, offering a centralized service for course management.

Constructor:

```
public CourseService (ApplicationDbContext context)
    {
        _context = context;
    }
```

Functions:

1. **public async Task<IEnumerable< Course>> GetAllCourses ():**
    - Retrieves a course from the database with the specified courseId.
2. **public async Task< Course> GetCourseById (int courseId):**
    - Retrieves a course from the database with the specified courseId.
3. **public async Task<bool> AddCourse (Coursecourse):**
    - Adds a new course to the database.
    - Checks if a course with the same title already exists.
    - If a course with the same title exists, throws a CourseException with the message "A course with the same name already exists".
    - If no such course exists, adds the new course to the database.
    - Saves changes asynchronously to the database.
    - Returns true for a successful insertion.
4. **public async Task<bool> UpdateCourse (int courseId, Course course):**
    - Updates an existing course in the database with the values from the provided course object.
    - If no course with the specified courseId is found, returns false.
    - Checks if a course with overlapping dates already exists for the same user (commented out logic).
    - If an overlapping course exists, throws a CourseException with the message "There is already a course scheduled for these dates" (commented out logic).

- If no overlapping course exists, updates the existing course with the provided values.
- Saves changes asynchronously to the database.
- Returns true for a successful update.

5. **public async Task<bool> DeleteCourse (int courseId):**
    - Deletes a course from the database with the specified courseId.
    - Retrieves the course based on the provided courseId.
    - If no course with the specified courseId is found, returns false.
    - If the course is found, deletes it from the database.
    - Saves changes asynchronously to the database.
    - Returns true for a successful deletion.

6. **public async Task<IEnumerable< Course>> GetCourseById (int courseId):**
    - Retrieves a course from the database with the specified courseId.

EnrollmentService (Services / EnrollmentService.cs):

The EnrollmentService class provides CRUD operations for managing Enrollment entities, facilitating interactions with the database to handle enrollments, and ensuring that related User and Course data is included. It offers methods to retrieve enrollments by user and course, add, update, and delete enrollments.

Constructor:

```
public EnrollmentService (ApplicationDbContext context)
{
    _context = context;
}
```

<u>Functions:</u>

1. **public async Task<IEnumerable< Enrollment>> GetAllEnrollments ():**
    - Retrieves and returns all enrollments from the database.
    - Includes related User and Course data.
2. **public async Task< Enrollment > GetEnrollmentById (int enrollmentId):**
    - Retrieves an enrollment from the database with the specified enrollmentId.
    - Includes related User and Course data.

3. **public async Task<bool> AddEnrollment (Enrollment enrollment):**

    - Adds a new enrollment to the database.
    - Saves changes asynchronously to the database.
    - Returns true for a successful insertion.

4. **public async Task<bool> UpdateEnrollment (int enrollmentId, Enrollment enrollment):**

    - Updates an existing enrollment in the database with the values from the provided enrollment object.
    - If no enrollment with the specified enrollmentId is found, returns false.
    - Saves changes asynchronously to the database.
    - Returns true for a successful update.

5. **public async Task<bool> DeleteEnrollment (int enrollmentId):**

    - Deletes an enrollment from the database with the specified enrollmentId.
    - Retrieves the enrollment based on the provided enrollmentId.
    - If no enrollment with the specified enrollmentId is found, returns false.
    - Saves changes asynchronously to the database.
    - Returns true for a successful deletion.

6. **public async Task<IEnumerable< Enrollment >> GetEnrollmentsByUserId (int userId):**

    - Retrieves and returns all enrollments from the database for the specified userId.
    - Includes related User and Course data.

7. **<u>FeedbackService</u> (Services / FeedbackService.cs):**

    - This service class provides methods to interact with feedback data stored in the database.

<u>Constructor:</u>

```
public FeedbackService(ApplicationDbContext context)
{
_context = context;
}
```

Functions:

1. **public async Task<IEnumerable<Feedback>> GetAllFeedbacks():**
    - Retrieves all feedbacks from the database.

2. **public async Task<IEnumerable<Feedback>> GetFeedbacksByUserId(int userId):**

- Retrieves all feedbacks associated with a specific **userId** from the database.

3. **public async Task<bool> AddFeedback(Feedback feedback):**
    - Adds new feedback to the database.
    - Return **true** for the successful insertion.

4. **public async Task<bool> DeleteFeedback(int feedbackId):**
    - Retrieve the existing feedback from the database with the specified **feedbackId**.
    - If no feedback with the specified feedbackId is found, return **false**.
    - If found, delete the feedback with the provided feedbackId.
    - Save changes asynchronously to the database.
    - Return **true** for the successful delete.

AuthService (Services / AuthService.cs):

The **AuthService** class is responsible for user authentication and authorization.

Constructor:

public **AuthService**(UserManager<ApplicationUser> userManager, RoleManager<IdentityRole> roleManager, IConfiguration configuration, ApplicationDbContext context)

{

this.userManager = userManager;

this.roleManager = roleManager;

_configuration = configuration;

_context = context;

}

Functions:

1. public async Task<(int, string)> Registration (User model, string role):

- Check if the email already exists in the database. If so return "**User already exists**".
- Registers a new user with the provided details and assigns a role.

- If any error occurs return "**User creation failed! Please check user details and try again**".
- Return "**User created successfully!"** for the successful register.

2. public async Task<(int, string)> Login (LoginModel model):

- Find user by email in the database.
- Check if user exists, if not return "**Invalid email**".
- If the user exists, check the password is correct, if not return "**Invalid password**".
- Logs in a user with the provided credentials and generates a **JWT** token for authentication.

3. private string GenerateToken(IEnumerable<Claim> claims):

- Generates a JWT token based on the provided claims.

IAuthService (Services / IAuthService.cs):

The **IAuthService** is an interface that defines methods for user registration and login.

Methods:

1. Task< (int, string)> Registration (User model, string role);
2. Task< (int, string)> Login (LoginModel model);

Controllers:

Inside "**Controllers"** folder create all the controllers file mentioned below.

Namespace: All the controllers file should located within the **dotnetapp.Controllers** namespace.

AuthenticationController (Controllers / AuthenticationController.cs):

This controller handles user authentication and registration requests.

Functions:

1. public async Task<IActionResult> Login(LoginModel model)

a. Accepts login requests, validates the payload, and calls the authentication service to perform user login.

b. It utilizes **_authService.Login(model)** method.

c. Returns a **200 OK response** with a JWT token upon successful login.

d. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

2. public async Task<IActionResult> Register(User model):

a. Accepts registration requests, validates the payload. If fails, then returns error.

b. Calls the authentication service to register a new user(**_authService.Registration(model, model.UserRole)**). Returns a **200 OK response with** success message upon successful registration.

c. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

CourseController (Controllers / CourseController.cs):

This controller manages courses, interacting with the **CourseService** to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<Course>>> GetAllCourses ():

a) The **GetAllCourses** method is a controller action responsible for retrieving all courses.
b) Requires authorization to access.
c) Calls the **_courseService.GetAllCourses()** method to fetch all courses from the service layer.
d) Returns a 200 OK response with the retrieved courses upon successful retrieval.

2. public async Task<ActionResult<Course>> GetCourseById (int courseId):

a) The **GetCourseById** method is a controller action responsible for retrieving a course by its ID.

b) Calls **the _courseService.GetCourseById(courseId)** method to retrieve the course from the service layer.
c) If the course is not found, returns a 404 Not Found response with a message “Cannot find any course”.
d) If the course is found, returns a 200 OK response with the course data.

3. public async Task<ActionResult> AddCourse ([FromBody] Course course):

a) The **AddCourse** method is a controller action responsible for adding a new course.
b) Implements the logic inside a try-catch block.
c) Receives the course data in the request body.
d) Tries to add the course using the **_courseService.AddCourse(course)** method.
e) If adding the course is successful, returns a 200 OK response with a success message “Course added successfully”.
f) If adding the course fails, returns a 500 Internal Server Error response with a failure message “Failed to add course”.
g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> UpdateCourse (int courseId, [FromBody] Course course):

a) The **UpdateCourse** method is a controller action responsible for updating an existing course.
b) Implements the logic inside a try-catch block.
c) Receives the course ID and updated course data in the request body.
d) Tries to update the course using the **_courseService.UpdateCourse(courseId, course)** method.
e) If the update is successful, returns a 200 OK response with a success message “Course updated successfully”.
f) If the course is not found, returns a 404 Not Found response with a message “Cannot find any course”.
g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult> DeleteCourse (int courseId):

a) The **DeleteCourse** method is a controller action responsible for deleting a course.
b) Implements the logic inside a try-catch block.
c) Receives the course ID to be deleted.
d) Tries to delete the course using **the _courseService.DeleteCourse(courseId)** method.
e) If the deletion is successful, returns a 200 OK response with a success message “Course deleted successfully”.

f) If the course is not found, returns a 404 Not Found response with a message “Cannot find any course”.
g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

EnrollmentController (Controllers / EnrollmentController.cs):

This controller enrolls the course, interacting with the **EnrollmentService**to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<Enrollment>>> GetAllEnrollments ():

a) The **GetAllEnrollments** method is a controller action responsible for retrieving all enrollments.
b) Calls **the _enrollmentService.GetAllEnrollments()** method to fetch all enrollments from the service layer.
c) Returns a 200 OK response with the retrieved enrollments upon successful retrieval.

2. public async Task<ActionResult<Enrollment>> GetEnrollmentById (int enrollmentId):

a) The **GetEnrollmentById** method is a controller action responsible for retrieving an enrollment by its ID.
b) Calls the **_enrollmentService.GetEnrollmentById(enrollmentId)** method to retrieve the enrollment from the service layer.
c) If the enrollment is not found, returns a 404 Not Found response with a message “Cannot find any enrollment”.

3. public async Task<ActionResult> AddEnrollment ([FromBody] Enrollment enrollment):

a) The AddEnrollment method is a controller action responsible for adding a new enrollment.
b) Implements the logic inside a try-catch block.
c) Receives the enrollment data in the request body.
d) Tries to add the enrollment using the _enrollmentService.AddEnrollment(enrollment) method.
e) If adding the enrollment is successful, returns a 200 OK response with a success message “Enrollment added successfully”.
f) If adding the enrollment fails, returns a 500 Internal Server Error response with a failure message “Failed to add enrollment”.

g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

4. public async Task<ActionResult> UpdateEnrollment (int enrollmentId, [FromBody] Enrollment enrollment):

a) The UpdateEnrollment method is a controller action responsible for updating an existing enrollment.
b) Implements the logic inside a try-catch block.
c) Receives the enrollment ID and updated enrollment data in the request body.
d) Tries to update the enrollment using the _enrollmentService.UpdateEnrollment(enrollmentId, enrollment) method.
e) If the update is successful, returns a 200 OK response with a success message "Enrollment updated successfully".
f) If the enrollment is not found, returns a 404 Not Found response with a message "Cannot find any enrollment".
g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

5. public async Task<ActionResult> DeleteEnrollment (int enrollmentId):

a) The DeleteEnrollment method is a controller action responsible for deleting an enrollment.
b) Implements the logic inside a try-catch block.
c) Receives the enrollment ID to be deleted.
d) Tries to delete the enrollment using the _enrollmentService.DeleteEnrollment(enrollmentId) method.
e) If the deletion is successful, returns a 200 OK response with a success message "Enrollment deleted successfully".
f) If the enrollment is not found, returns a 404 Not Found response with a message "Cannot find any enrollment".
g) If an exception occurs during the process, returns a 500 Internal Server Error response with the exception message.

FeedbackController (Controllers / FeedbackController.cs):

This controller manages feedbacks, interacting with the **FeedbackService** to perform CRUD operations.

Functions:

1. public async Task<ActionResult<IEnumerable<Feedback>>> GetAllFeedbacks():

a. Implement the logic inside **try-catch block.**

b. The **GetAllFeedbacks** method is a controller action responsible for retrieving all feedbacks.

c. It tries to get all feedbacks using the **_feedbackService.GetAllFeedbacks()** method.

d. If the operation is successful, it returns a **200 OK response** with the retrieved feedbacks.

e. If an exception occurs during the process, it returns a **500 Internal Server Error response** with the exception message.

2. public async Task<ActionResult<IEnumerable<Feedback>>> GetFeedbacksByUserId(int userId):

a. Implement the logic inside **try-catch block**.

b. The **GetFeedbacksByUserId** method is a controller action responsible for retrieving feedbacks by **userId**.

c. It tries to get feedbacks by userId using the **_feedbackService.GetFeedbacksByUserId(userId)** method.

d. If feedbacks are found, it returns a **200 OK response** with the retrieved feedbacks.

e. If an exception occurs during the process, it returns a **500 Internal Server Error response** with the exception message.

3. public async Task<ActionResult> AddFeedback([FromBody] Feedback feedback):

a. Implement the logic inside **try-catch block**.

b. The **AddFeedback** method is a controller action responsible for adding a new feedback.

c. It receives the feedback data in the request body.

d. It tries to add the feedback using the **_feedbackService.AddFeedback(feedback)** method.

e. If adding the feedback is successful, it returns a **200 OK response** with a success message "**Feedback added successfully**".

f. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

4. public async Task<ActionResult> DeleteFeedback(int feedbackId):

a. Implement the logic inside **try-catch block**.

b. The **DeleteFeedback** method is a controller action responsible for deleting a feedback.

c. It receives the **feedbackId** to be deleted.

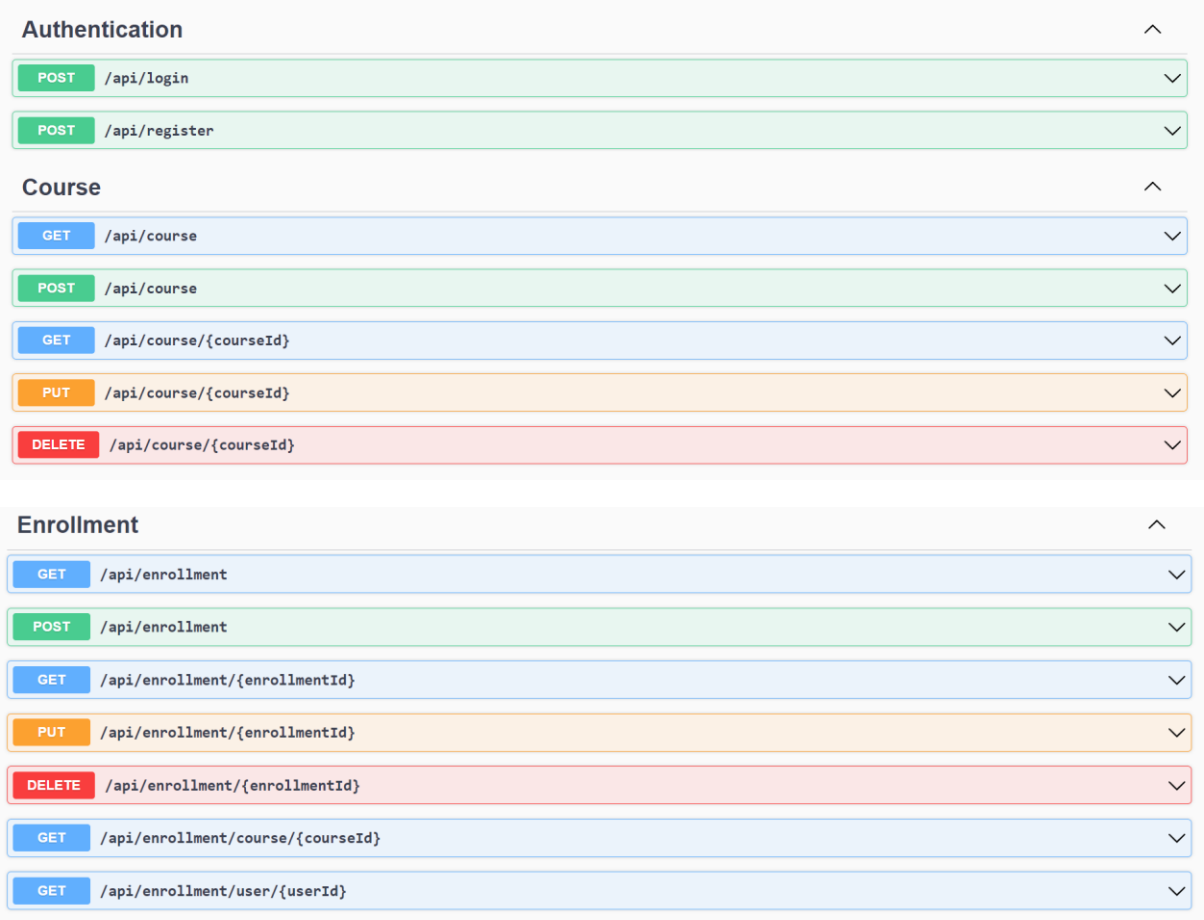
d. It tries to delete the feedback using the **_feedbackService.DeleteFeedback(feedbackId)** method.

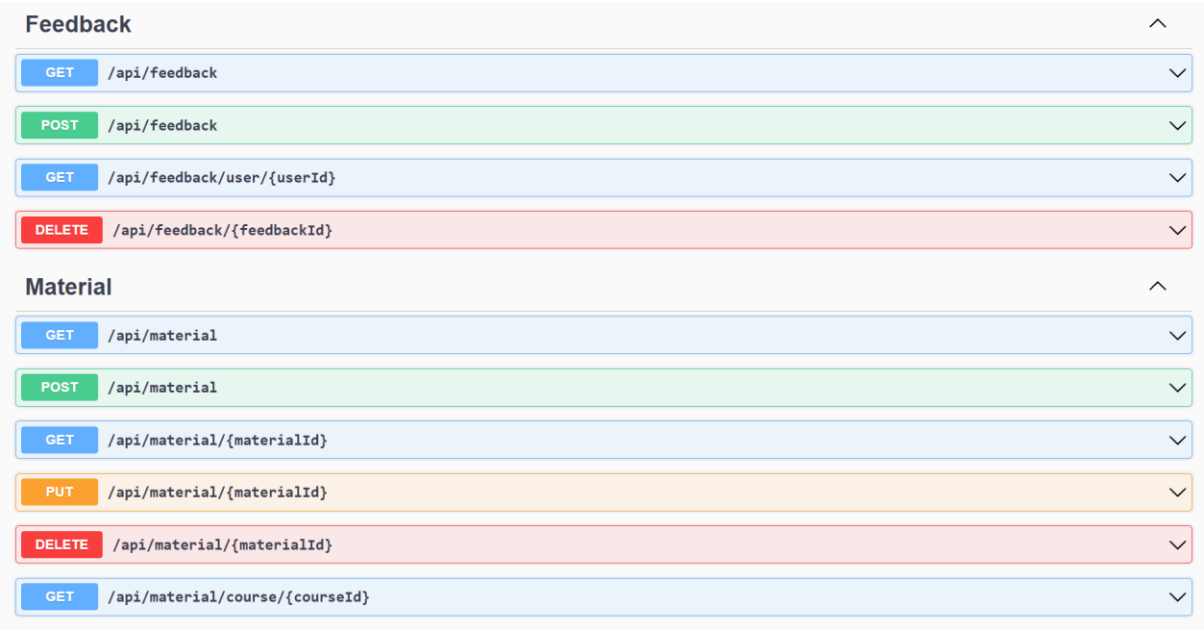
e. If the deletion is successful, it returns a **200 OK response** with a success message "**Feedback deleted successfully**".

f. If the feedback is not found, it returns a **404 Not Found** response with a message "**Cannot find any feedback**".

g. If an exception occurs during the process, it returns a **500 Internal Server Error** response with the exception message.

Endpoints:





Note:

Ensure that JWT authentication is enabled for all endpoints, with authorization based on user roles. For example, the endpoint for adding a course should only be accessible with a JWT present in the header. If not, return **Unauthorized**. Additionally, it should be restricted to users with the "**Student"** role. If a user **without the "Student" role (Educator)** attempts to access it, it should return a **Forbidden** error.

1. Login: [Access for both Student and Educator]

Endpoint name: "/api/login"
Method: POST
Request body: {
"Email": "string",
"Password": "string"
}
Response:

Status Code	Response body

200 (HttpStatusCode OK)	JSON object containing a JWT token.
	Example: {
	"Status": "Success",
	"token":
	"eyJhbGciOiJIUzI1NiiQWRtaW4iLCJqdGkiOiJmZTUyYzAxZS1"
	}

400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

2. Register: [Access for both Student and Educator]

Endpoint name: “/api/register”

Method: POST

Request body:

```
{
  "Username": "string",
  "Email": "user@example.com",
  "MobileNumber": "9876541221",
  "Password": "Pass@2425",
  "UserRole": "string"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

3. Get all courses: [Access for both **Educator, Student**]

Endpoint name: “/api/course”

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all the courses.
500	JSON object containing Error message.

4. Get courses by id: [Access for both **Student, Educator**]

Endpoint name: “/api/course/{coursed}”

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing course by id.

500	JSON object containing Error message.

5. Add course: [Access for only Educator]

Endpoint name: "/api/course"

Method: POST

Request body:


```
{
  "CourseId": 1,
  "Title": "string",
  "Description": "string",
  "CourseStartDate": "2024-05-21T07:55:12.075Z",
  "CourseEndDate": "2024-05-21T07:55:12.075Z",
  "Category": "string",
  "Level": "string"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

6. Update course: [Access only for Educator]

Endpoint name: "/api/course /{courseId}"

Method: PUT

Parameter: courseId

Request body:


```
{
  "CourseId": 1,
  "Title": "string",
```

```
  "Description": "string",
  "CourseStartDate": "2024-05-21T07:56:13.882Z",
  "CourseEndDate": "2024-05-21T07:56:13.882Z",
  "Category": "string",
  "Level": "string"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing course details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

7. Delete course: [Access for only Educator]

Endpoint name: "/api/course/{courseId}"

Method: DELETE

Parameter: courseId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

8. Get enrollment specific to user: [Access for only Educator]

Endpoint name: "/api/enrollment /user/{userId}"

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing enrollment details.

404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

9. <u>Get all enrollment request:</u> [Access for both **Educator and Student**]

Endpoint name: "/api/enrollment"

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing all the enrollments.
500	JSON object containing Error message.

10. <u>Add enrollment request:</u> [Access for only Student]

Endpoint name: "/api/enrollment"

Method: POST

Request body:


```
{
  "EnrollmentId": 0,
  "UserId": 0,
  "User": {
    "UserId": 0,
    "Email": "string",
    "Password": "string",
    "Username": "string",
    "MobileNumber": "string",
    "UserRole": "string"
  },
  "CourseId": 0,
  "Course": {
```

```
  "CourseId": 0,
  "Title": "string",
  "Description": "string",
  "CourseStartDate": "2024-05-21T08:06:27.894Z",
  "CourseEndDate": "2024-05-21T08:06:27.894Z",
  "Category": "string",
  "Level": "string"
 },
 "EnrollmentDate": "2024-05-21T08:06:27.894Z",
 "Status": "string"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

11. Get specific enrollment: [Access for only Student]

Endpoint name: "/api/enrollment /{enrollmentId}"

Method: GET

Parameter: enrollmentId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing enrollment details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

12. Update Enrollment : [Access only for Student]

Endpoint name: "/api/enrollment /{ enrollmentId}"

Method: PUT

Parameter: enrollmentId

Request body:


```
{
  "EnrollmentId": 0,
  "UserId": 0,
  "User": {
    "UserId": 0,
    "Email": "string",
    "Password": "string",
    "Username": "string",
    "MobileNumber": "string",
    "UserRole": "string"
  },
  "CourseId": 0,
  "Course": {
    "CourseId": 0,
    "Title": "string",
    "Description": "string",
    "CourseStartDate": "2024-05-21T08:12:52.080Z",
    "CourseEndDate": "2024-05-21T08:12:52.080Z",
    "Category": "string",
    "Level": "string"
  },
  "EnrollmentDate": "2024-05-21T08:12:52.080Z",
  "Status": "string"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing enrollment details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

13. Delete enrollment: [Access for only Student]

Endpoint name: "/api/enrollment/{enrollmentId }"

Method: DELETE

Parameter: enrollmentId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

14. Get enrollment specific to course: [Access for only Educator]

Endpoint name: "/api/enrollment /course/{courseId}"

Method: GET

Parameter: courseId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing course details.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

15. Get all feedbacks: [Access for only Educator]

Endpoint name: "/api/feedback"

Method: GET

Response:

Status Code	Response body

200 (HttpStatusCode OK)	JSON object containing all feedback.
500	JSON object containing Error message.

16. <u>Add feedback</u>: [Access for only Student]

Endpoint name: “/api/feedback”

Method: POST

Request body:

```
{
  "UserId": 0,
  "FeedbackText": "string",
  "Date": "2024-07-07T12:28:56.927Z"
}
```

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
400 BadRequest	JSON object containing Error message.
500	JSON object containing Error message.

17. <u>Get feedback specific to user</u>: [Access for only Student]

Endpoint name: “/api/feedback/{userId }”

Method: GET

Parameter: userId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing feedback details.

404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

18. <u>Delete feedback</u>: [Access for only Student]

Endpoint name: "/api/feedback /{feedbackId}"

Method: DELETE

Parameter: feedbackId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

19. <u>Material feedback</u>: [Access for only Educator]

Endpoint name: "/api/material"

Method: POST

Request body:


```
{
  "MaterialId": 0,
  "CourseId": 0,
  "Course": {
    "CourseId": 0,
    "Title": "string",
    "Description": "string",
    "CourseStartDate": "2024-05-21T09:34:18.185Z",
    "CourseEndDate": "2024-05-21T09:34:18.185Z",
    "Category": "string",
    "Level": "string"
  },
```

```
  "Title": "string",
  "Description": "string",
  "URL": "string",
  "UploadDate": "2024-05-21T09:34:18.185Z",
  "ContentType": "string"
}
```


Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

20. Get Material: [Access for only Student]

Endpoint name: "/api/material"

Method: GET

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

21. Get All Material by id: [Access for both Educator, Student]

Endpoint name: "/api/material/{materialId}"

Method: GET

Parameter: materialId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

22. Update Material: [Access for only Educator]

Endpoint name: "/api/materail/{ materialId }"

Method: PUT

Parameter: materialId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

23. Delete Material: [Access for only Educator]

Endpoint name: "/api/material /{materialId}"

Method: DELETE

Parameter: materialId

Response:

Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

24. GET Material by course: [Access for both Educator,Student]

Endpoint name: "/api/course/{ courseId }"

Method: GET

Parameter: courseId

Response:

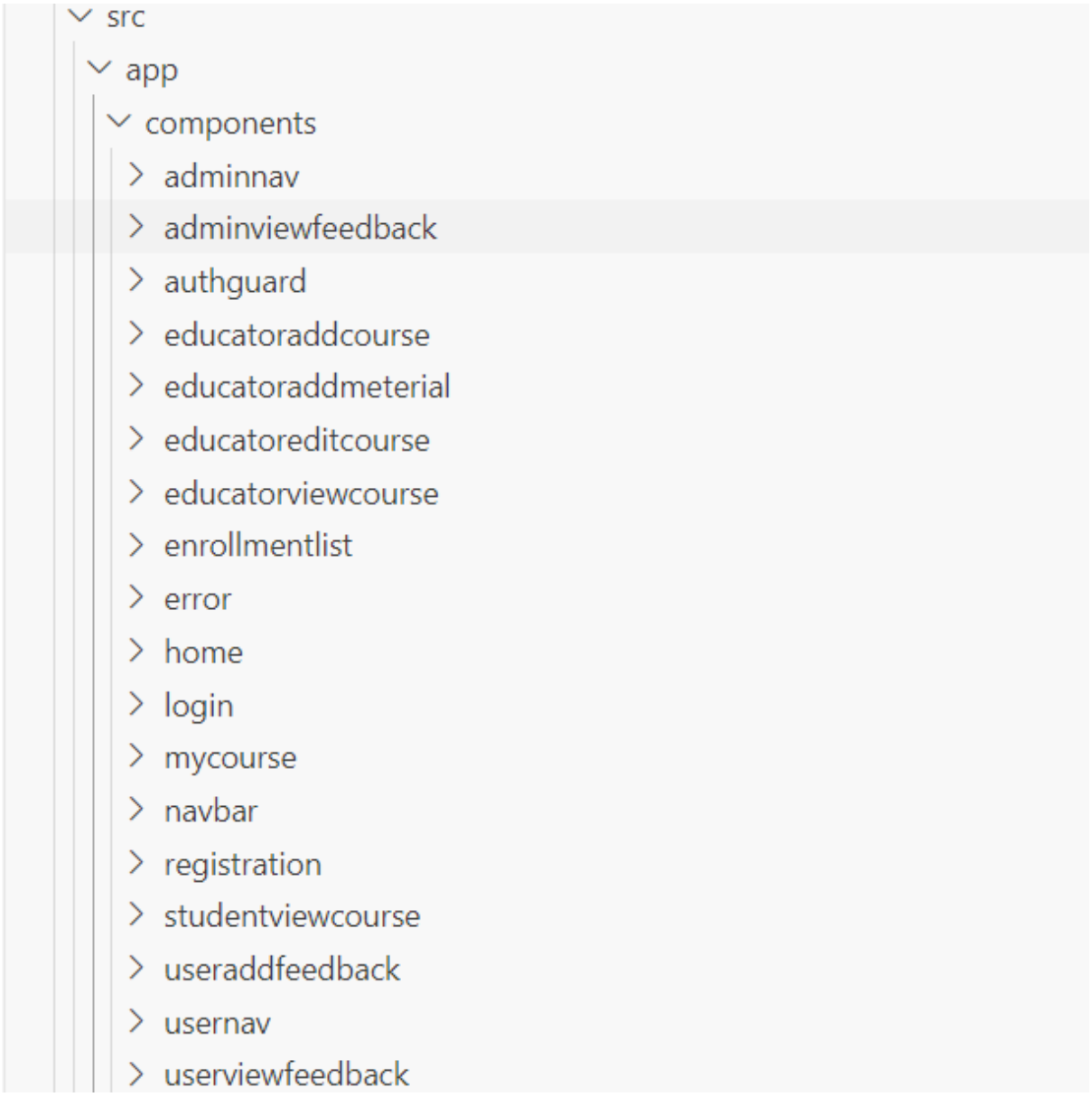
Status Code	Response body
200 (HttpStatusCode OK)	JSON object containing success message.
404 NotFound	JSON object containing Error message.
500	JSON object containing Error message.

Frontend Requirements:

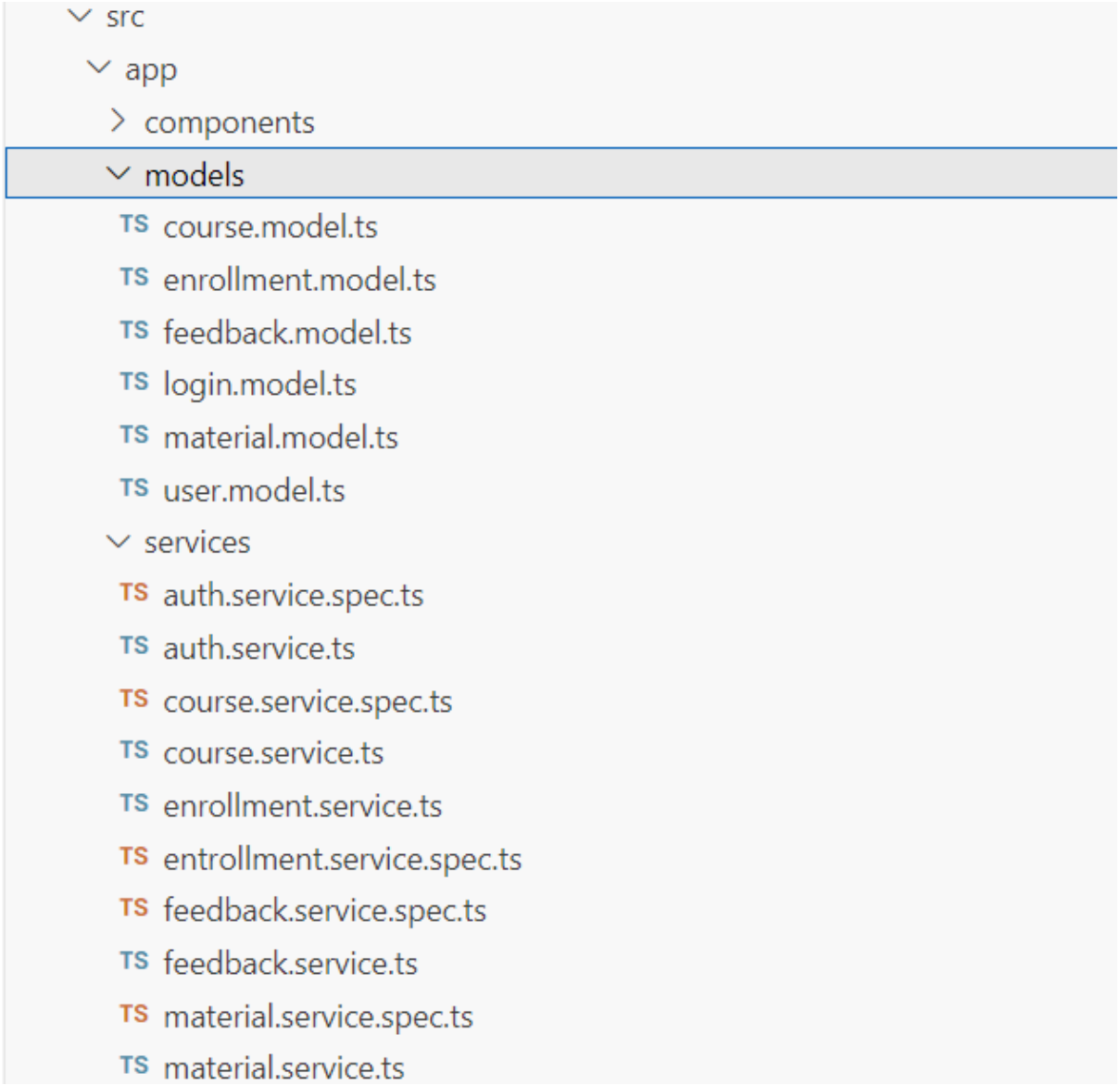
- Create a folder named components inside the app to store all the components. (Refer project structure screenshots).
- Create a folder named models inside app to store all the model interface.
- Create a folder named as services inside app to implement all the services.
- Create model interface referring the backend entities (User, Student, Course, Enrollment, Feedback) mentioned in the backend requirements accordingly.
- You can create your own components based on the application requirements.
- Import model files, services and components as required.

Project Folder Screenshot:

Components:



Services and Models:



Frontend Models:

User Model:

class User {

UserId?: number;

Email: string;

Password: string;

Username: string;

MobileNumber: string;

UserRole: string;

}

Login Model:

class Login {

Email: string;

Password: string;

}

Course Model:

interface Course {

CourseId Id?: number;

Title: string;

Description: string;

CourseStartDate: string;

CourseEndDate: string;

Category: string;

Level: string;

}

Enrollment Model:

interface Enrollment {

EnrollmentId?: number;

UserId: number;

User: User;

CourseId: number;

Course: Course;

```
  EnrollmentDate: string;
  Status: string;
}
```

Material Model:

```
interface Material{
MaterialId?: number;
CourseId?: number;
Title: string;
Description: string;
URL: string;
UploadDate: string;
ContentType: string;
}
```

Feedback Model:

```
class Feedback {
FeedbackId?: number;
UserId: number;
FeedbackText: string;
Date: Date;
}
```

Frontend services:

- Declare a public property apiUrl to store the backend URL in all the services.

- For example, public apiUrl = 'http://localhost:8080'. Instead of 'localhost', replace it with the URL of your workspace port 8080 URL.
- For the API's to be used please refer the API Table.
- Authorized token to be passed in headers for all end points.
- **Important note:** Authorization token should be prefixed with 'Bearer'.

CourseService (services /course.service.ts):

Create a service named CourseService inside the app/services folder to implement the following functions.

Methods Overview:

1. **getAllCourses (): Observable< Course []>:**
    a. Use this method to retrieve all courses. It sends a GET request to the '/api/course endpoint with the JWT token in the authorization header.

2. **getCourseById (id: number): Observable< Course>:**
    a. This method is used to retrieve a specific course by its ID. It sends a GET request to the '/api/course/{courseId}' endpoint with the JWT token in the authorization header.
3. **addCourse (requestObject: Course): Observable< Course>:**
    a. Use this method to add a new course. It sends a POST request to the '/api/course endpoint with the course object provided as the body and the JWT token in the authorization header.
4. **updateCourse (id: number, requestObject: Course): Observable< Course>:**
    a. This method is used to update an existing course. It sends a PUT request to the '/api/course/{courseId}' endpoint with the updated course object provided as the body and the JWT token in the authorization header.
5. **deleteCourse (courseId: number): Observable<void>:**
    a. Use this method to delete a specific course by its ID. It sends a DELETE request to the '/api/course/{courseId}' endpoint with the JWT token in the authorization header.

EnrollmentService (services / enrollment.service.ts):

Create a service named EnrollmentService inside the app/services folder to implement the following functions.

Methods Overview:

1. **getAllEnrollments (): Observable< Enrollment []>:**
    a. Use this method to retrieve all enrollments. It sends a GET request to the '/api/enrollment endpoint with the JWT token in the authorization header.

2. **enrollCourse (courseId: string, userId: string): Observable< Enrollment []>:**
    a. This method sends a POST request to enroll a user in a course, including authorization and necessary enrollment details, and returns an Observable of the created Enrollment object.

3. **updateEnrollmentStatus (id: string, enrollment: Enrollment): Observable< Enrollment>:**
    a. Use this method to update an existing enrollment. It sends a PUT request to the '/api/enrollment/{enrollmentId}' endpoint with the updated enrollment object provided as the body and the JWT token in the authorization header.

4. **unenrollCourse (enrollmentId: string): Observable< Enrollment>:**
    a. This method sends a DELETE request to unenroll a user from a course using the provided enrollment ID, including authorization, and returns an Observable of the deleted Enrollment object. It sends a DELETE request to the '/api/enrollment/{enrollmentId}' endpoint with the JWT token in the authorization header.

MaterialService (services / material.service.ts):

Create a service named **MaterialService**inside the **app/services** folder to implement the following function.

Methods Overview:

1. **getMaterialsByCourseId (coursed: string): Observable< Material []>:** This method sends a GET request to retrieve all materials for a specified course, including authorization, and returns an Observable of an array of Material objects. It sends a

GET request to the '/api/material endpoint with the JWT token in the authorization header.

2. **addMaterial(material: Material): Observable<Material>:** This method sends a POST request to the endpoint /api/material to add a new material, including authorization and content type headers, and returns an Observable of the created Material object.
3. **deleteMaterial(materialId: number): Observable<any>:** This method sends a DELETE request to the endpoint /api/material/${materialId} to delete a material by its ID, including authorization and content type headers, and returns an Observable of the response.

AuthService(services/auth.service.ts):

Create a service name as **auth** inside app/services folder to implement the following functions.

Methods Overview:

1. **register(user: User): Observable<any>:**
    a. Use this method to register a new user. It sends a POST request to the '/api/register' endpoint with the user data provided as the body.
2. **login(login : Login): Observable<any>:**
    a. This method is used to authenticate a user by logging them in. It sends a POST request to the '/api/login' endpoint with the user's email and password. Upon successful login, it stores the JWT token in localStorage and updates the user's role and ID using BehaviorSubjects.

FeedbackService(feedback.service.ts):

Create a service name as feedback inside app/services and implement the following functions in it.

Methods Overview:

1. **sendFeedback(feedback: Feedback): Observable<Feedback>:**
    a. Use this method to send feedback to the server. It sends a POST request to the '/api/feedback' endpoint with the feedback data provided and the authorization token prefixed with 'Bearer' stored in localStorage.
2. **getAllFeedbacksByUserId(userId: string): Observable<Feedback[]>:**

a. This method retrieves all feedbacks submitted by a specific user. It sends a GET request to the '/api/feedback/user/:userId' endpoint with the user ID and the authorization token prefixed with 'Bearer' stored in localStorage.

3. **deleteFeedback(feedbackId: string): Observable<void>:**

    a. Call this method to delete a feedback with the specified ID. It sends a DELETE request to the '/api/feedback/:feedbackId' endpoint with the feedback ID and the authorization token prefixed with 'Bearer' stored in localStorage.

4. **getFeedbacks(): Observable<Feedback[]>:**

    a. This method fetches all feedbacks from the server. It sends a GET request to the '/api/feedback' endpoint with the authorization token prefixed with 'Bearer' stored in localStorage.

Validations:

Client-Side Validation:

- Implement client-side validation using HTML5 attributes and JavaScript to validate user input before making API requests.
- Provide immediate feedback to users for invalid input, such as displaying error messages near the input fields.

Server-Side Validation:

- Implement server-side validation in the controllers to ensure data integrity.
- Validate user input and API responses to prevent unexpected or malicious data from affecting the application.
- Return appropriate validation error messages to the user interface for any validation failures.

Exception Handling:

- Implement exception handling mechanisms in the controllers to gracefully handle errors and exceptions. Define custom exception classes for different error scenarios, such as API communication errors or database errors.
- Log exceptions for debugging purposes while presenting user-friendly error messages to users. Record all the exceptions and errors handled store in separate table “ErrorLogs”.

Error Pages:

Create custom error pages for different HTTP status codes (e.g., 404 Not Found, 500 Internal Server Error) to provide a consistent and user-friendly error experience. Ensure that error pages contain helpful information and guidance for users.

Thus, create a reliable and user-friendly web application that not only meets user expectations but also provides a robust and secure experience, even when faced with unexpected situations. Error page has to be displayed if something goes wrong.

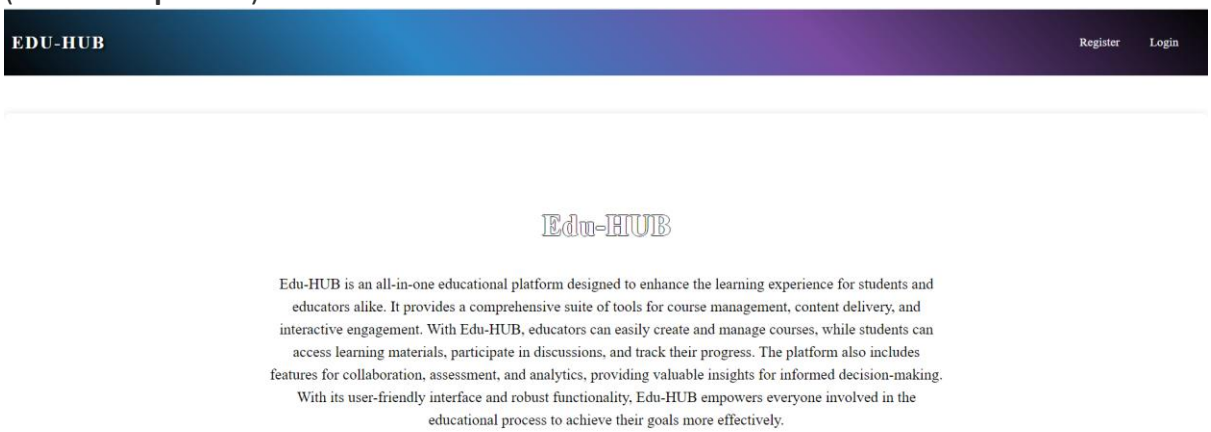
Frontend Screenshots:

• All the asterisk (*) marked fields are mandatory in the form. Make sure to mark all the field names with * symbol followed by the validations.

Navigation Bar:
(**navbar component**)

Component displays a title along with router links to "Register" and "Login".

Landing Page:
(**home component**)

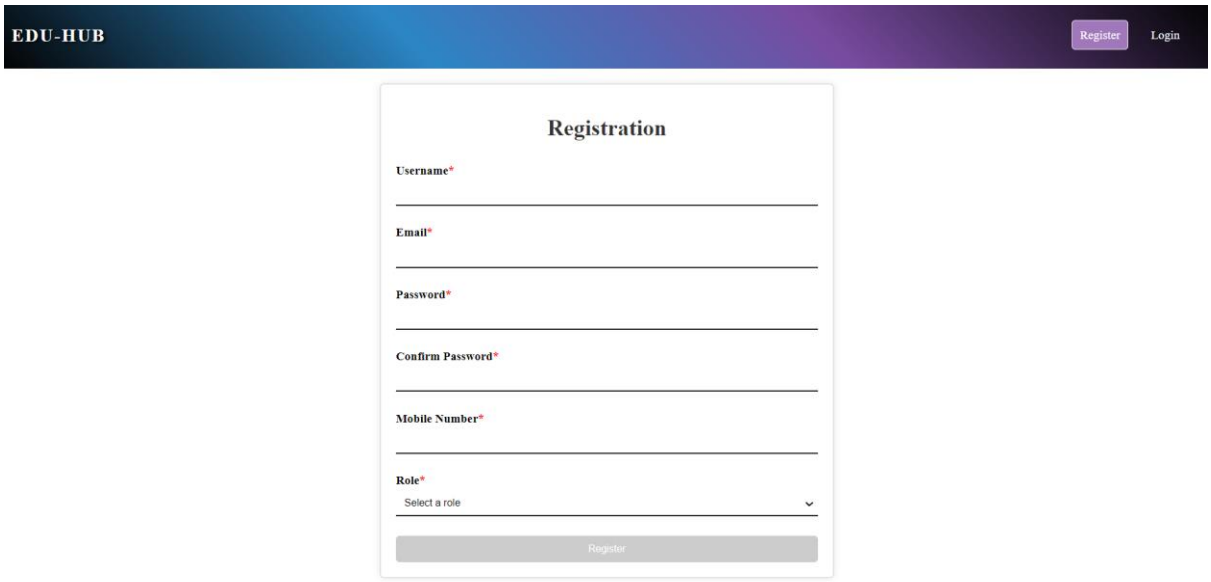


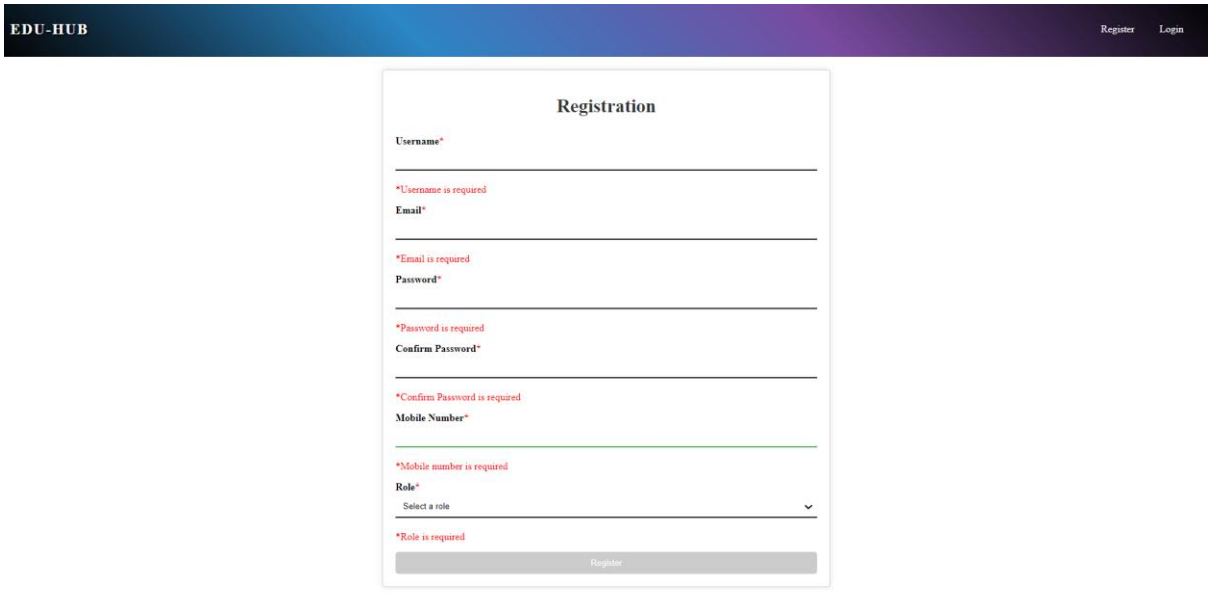
Component features a heading "**EDU-HUB**" accompanied by an introductory message that provides an overview of the application**.**

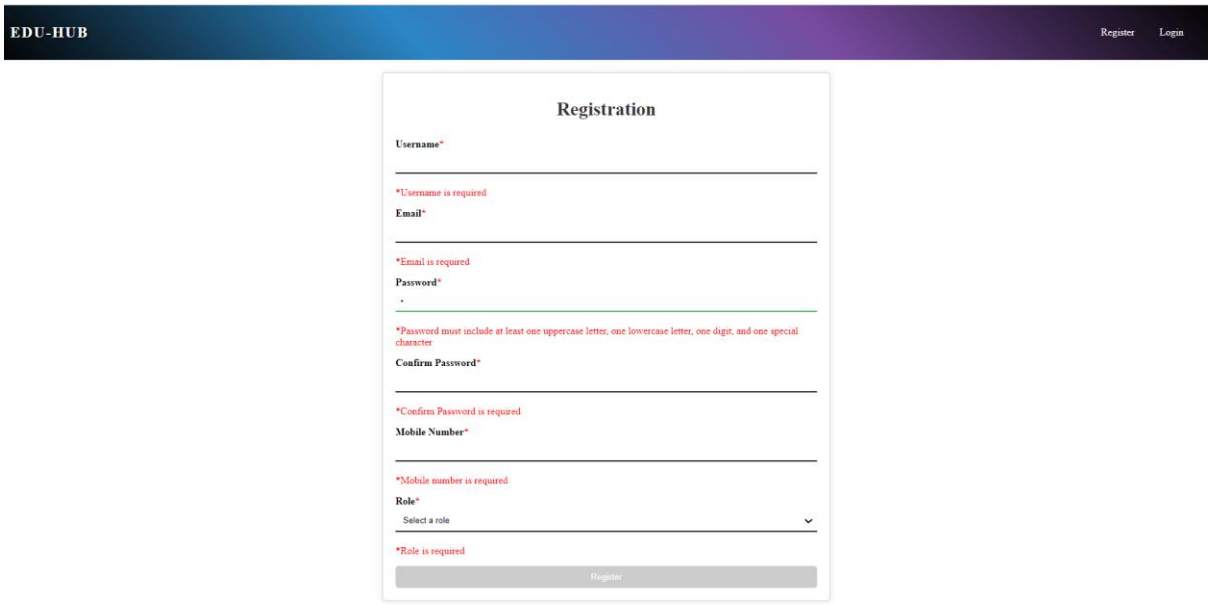
Common Screens (Educator and Student)

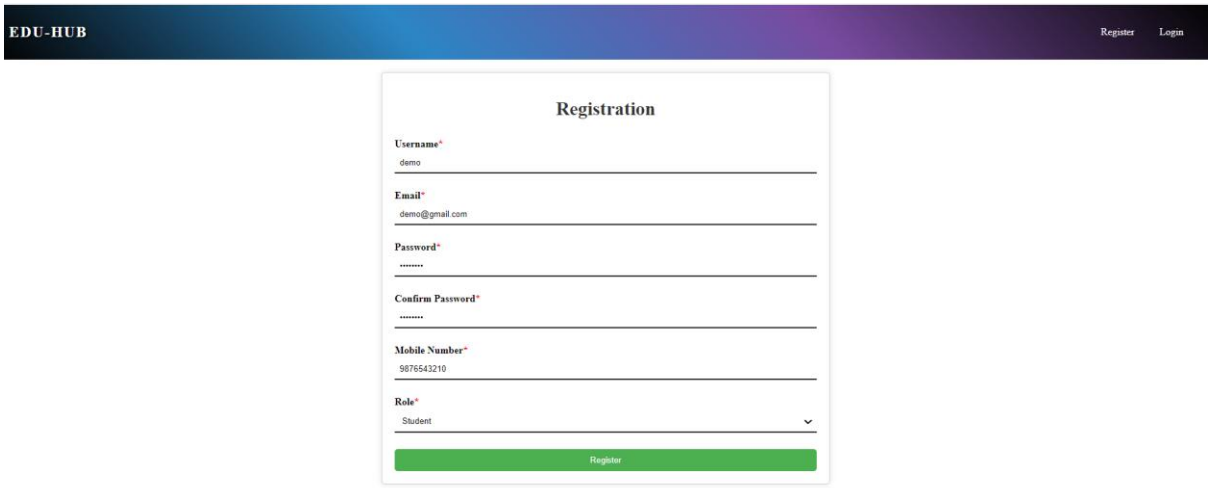
Registration Page:
(**registration component**)

Clicking "Register" in the navbar displays the registration page for both roles. Please refer to the screenshots below for validations.

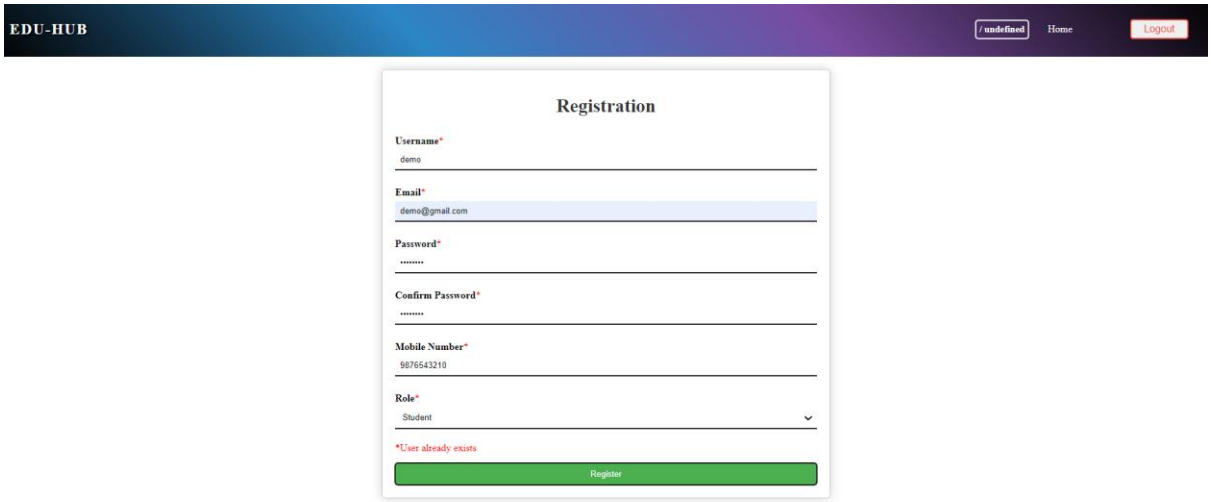








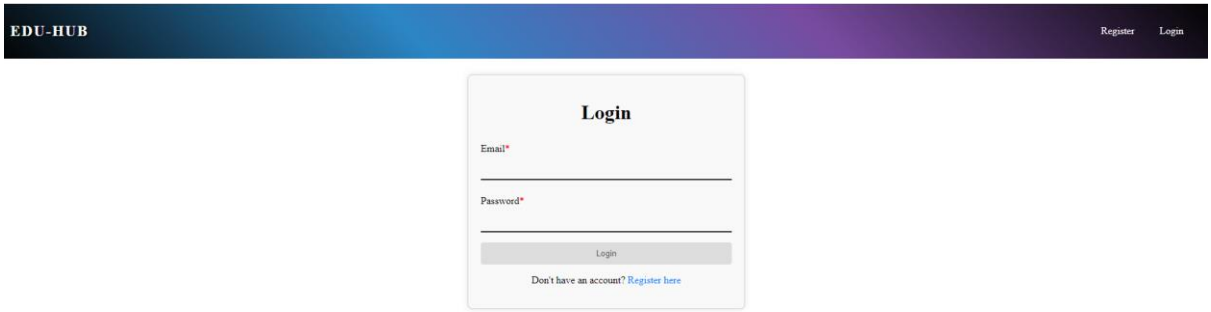
When the "Register" button is clicked, upon successful submission, the user must be navigated to the login page.



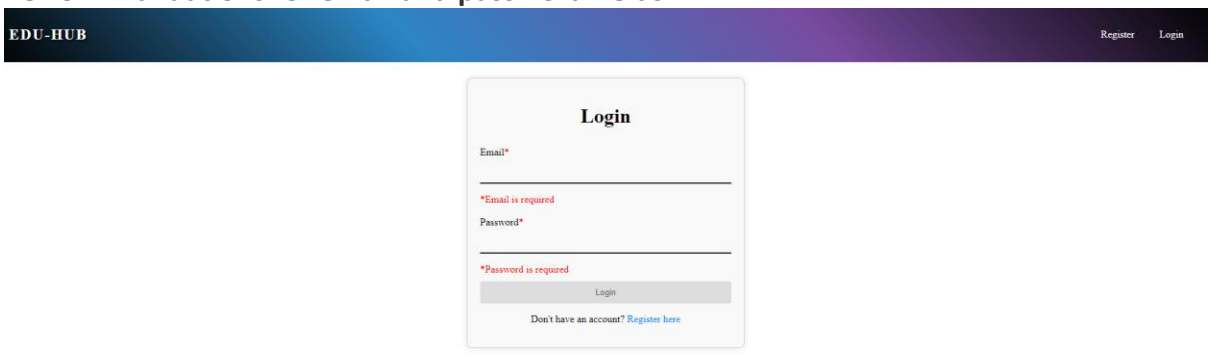
If a user attempts to register with an existing email, a message stating "User already exists" will be displayed.

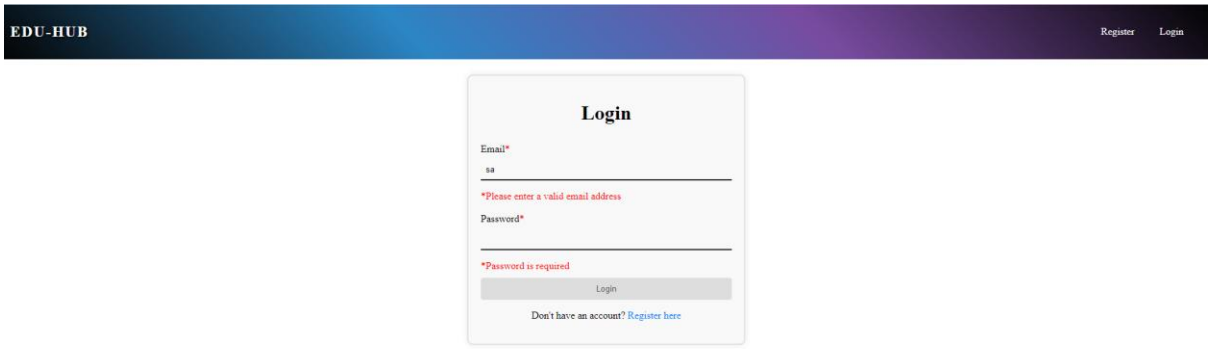
Login Page
(**login component**)

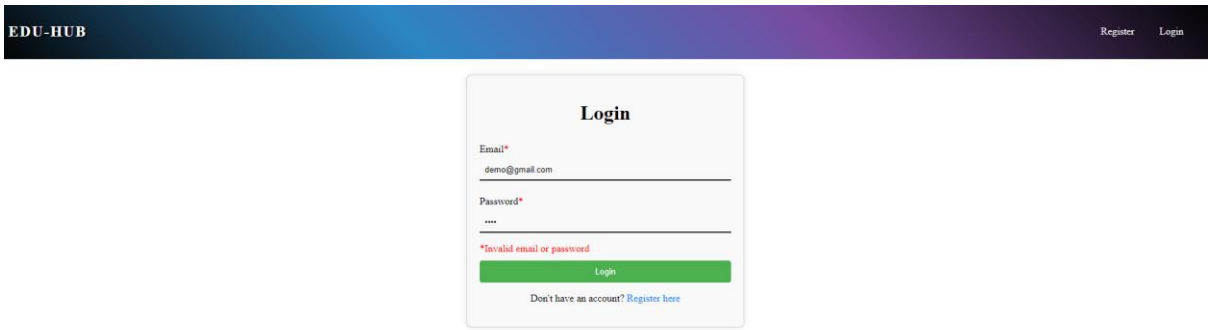
This page is used for logging in to the application. On providing the valid email and password, the user will be logged in.

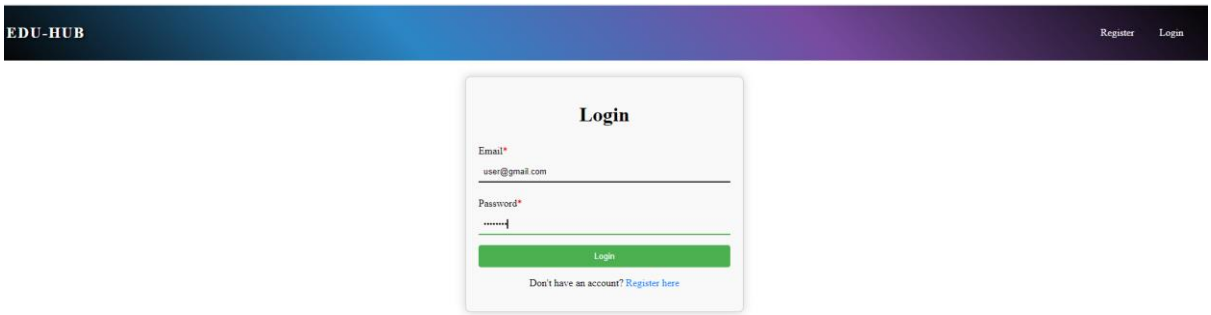


Perform validations for email and password fields.





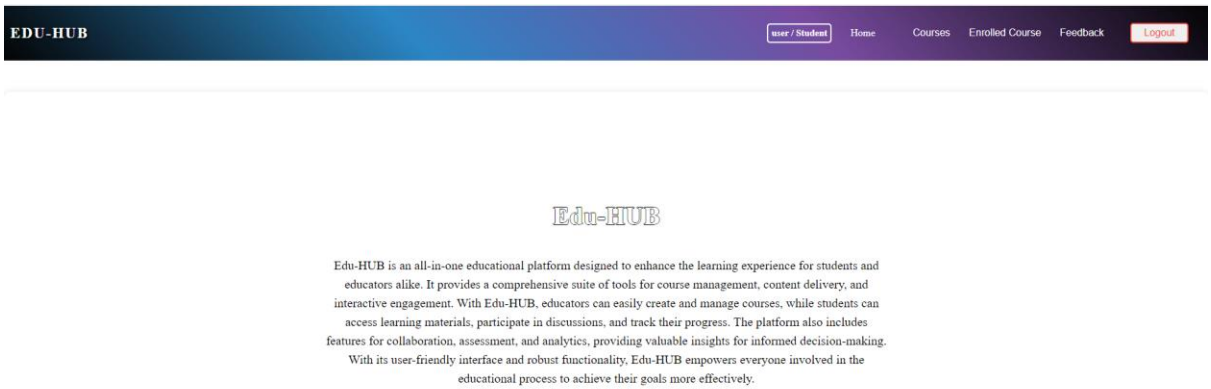




On Clicking the 'Login' button, user will be navigated to the (adminnav) based on their roles.

Student side:

Home Component: This page is used to display the information about the work buddy. On clicking the 'Home' tab, user can view the information about the application.



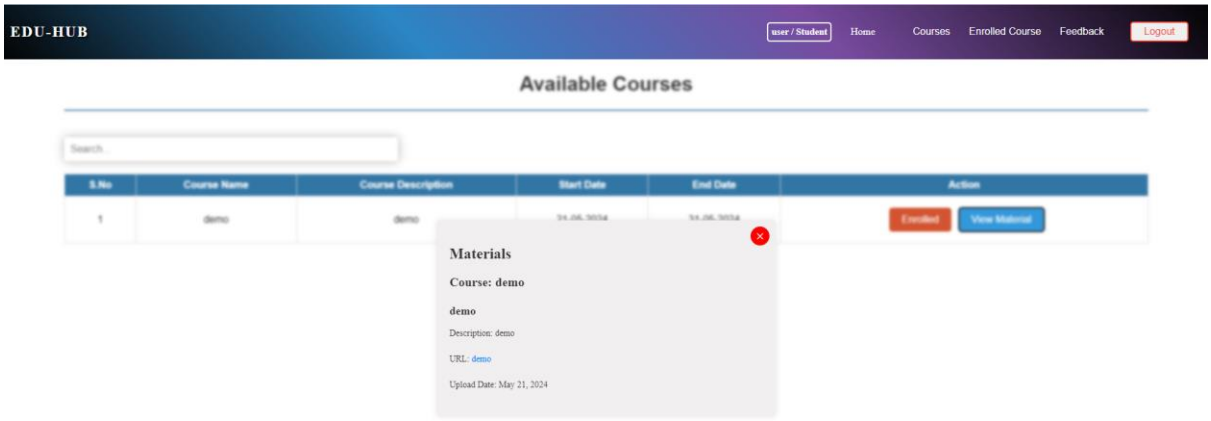
Upon successful login, if the user is an student, the (studentnav component) will be displayed. If the user is a student, the (usernav component) will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.

Courses

On hovering over the "Courses" item in the navbar, a submenu should appear with options to "View Courses".

Clicking on “Enroll” will navigate to the “**enrolled**” component, which displays the form to apply enroll with heading as "enrolment form"

ViewMaterial:



Clicking on “View Material” will display the “Material” component, which displays the form to view the material with heading as "Materials"

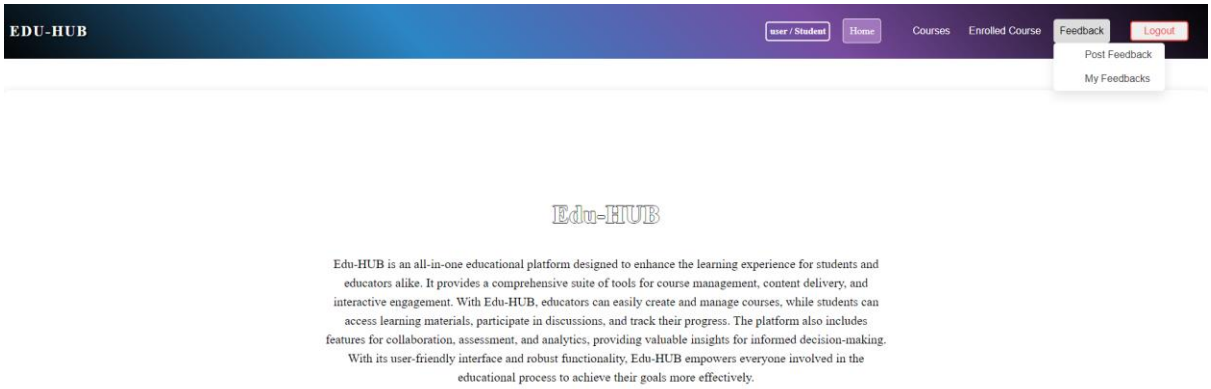
EnrolledCourses:

On hovering over the "Enrolled Course" item in the navbar, a submenu should appear with options to "enroll".

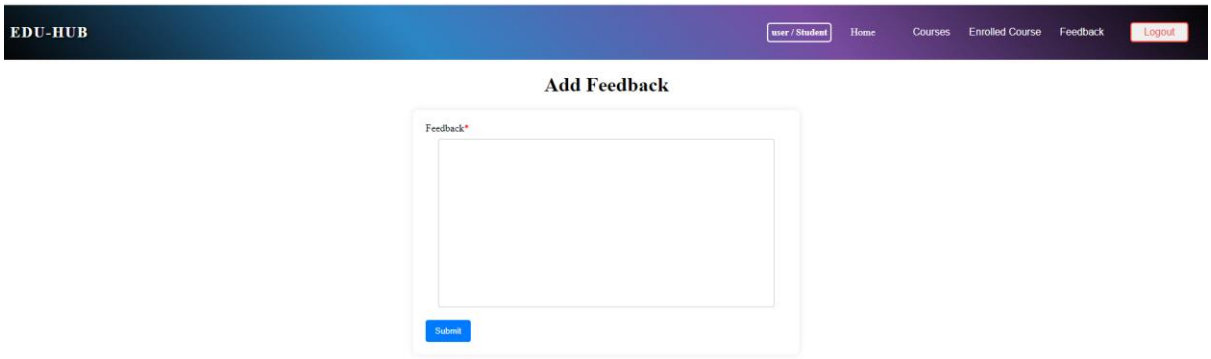
Clicking on “unenroll” will unroll the enrolled course.

Feedback:

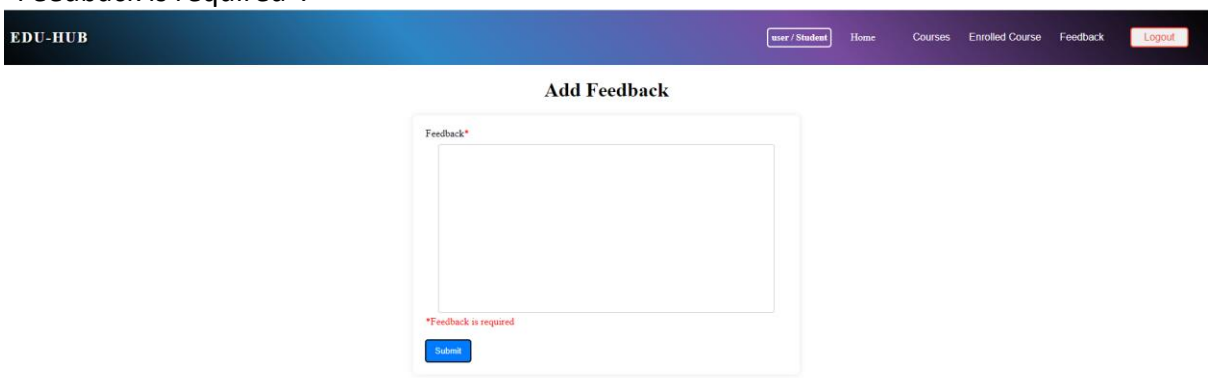
On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".



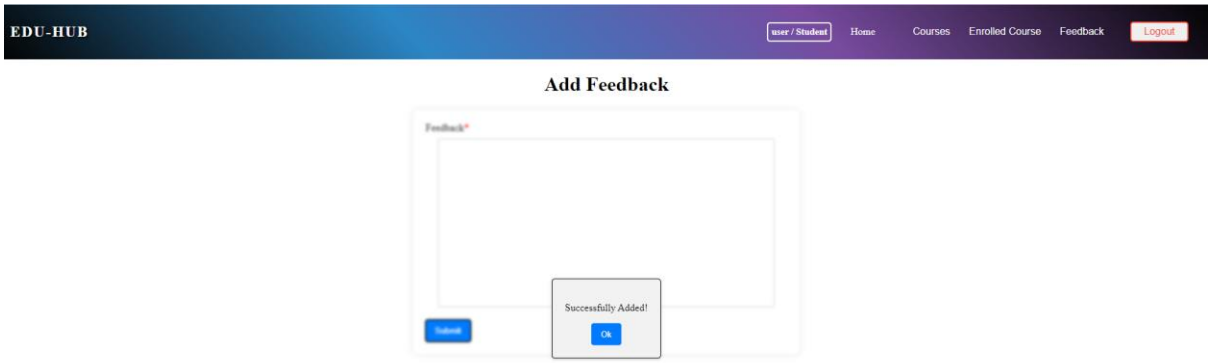
Clicking on "Post Feedback" will navigate to the **useraddfeedback** component, which displays the form to post feedback with heading as "Add Feedback"



Clicking the "Submit" button with empty textarea will display a validation message stating "Feedback is required".

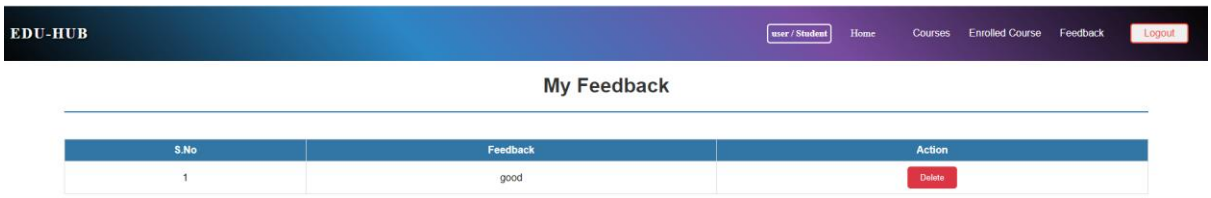


Upon clicking the "Submit" button, if the operation is successful, a popup message saying "Successfully Added!" should be displayed.

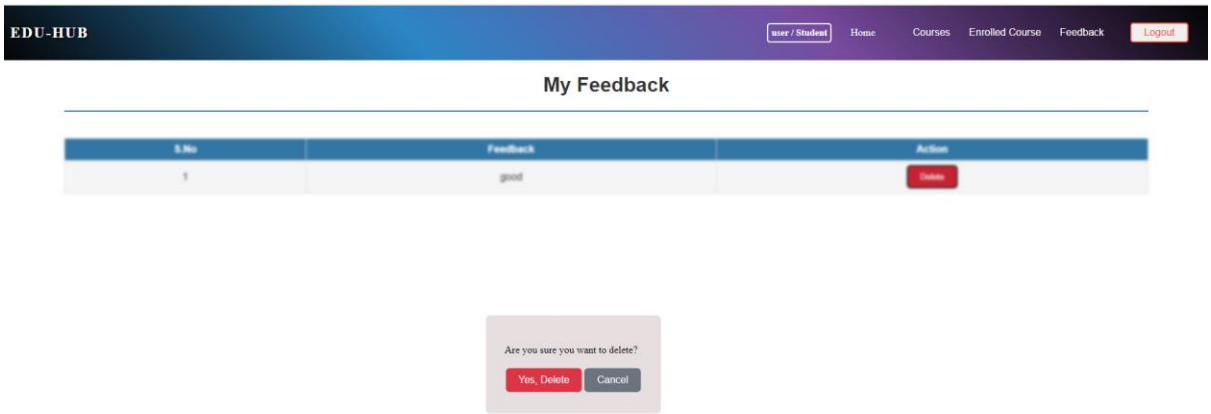


Clicking the "Ok" button will close the popup, and the same **useraddfeedback component** will be displayed.

User view Feedback:
On hovering over the "Feedback" item in the navbar, a submenu should appear with options to "Post Feedback" and "My Feedbacks".
Clicking on "My Feedbacks" will navigate to the **userviewfeedback component**, which displays posted feedback with heading as "My Feedback"

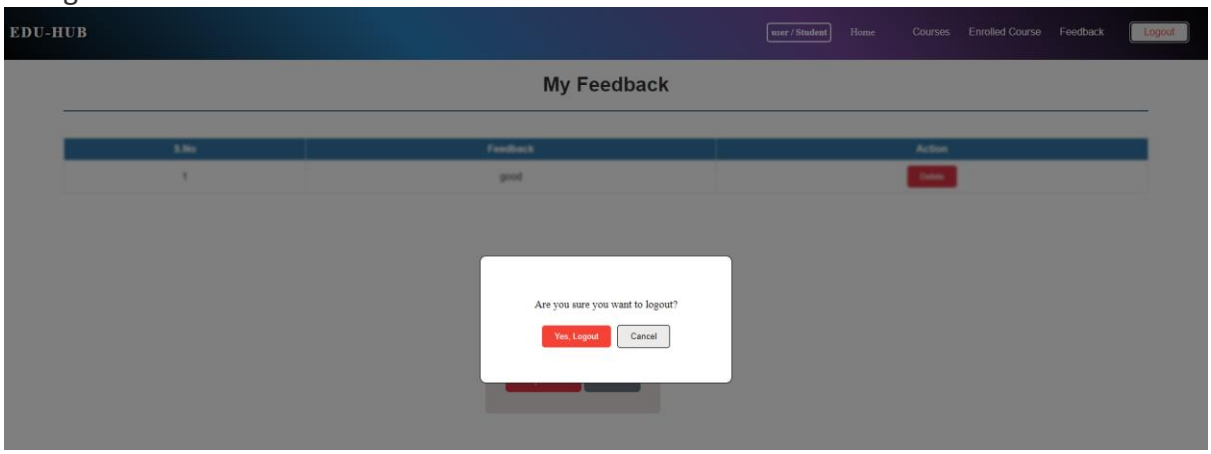


On clicking the "Delete" button, a pop-up should be displayed with confirmatory message to delete the data.

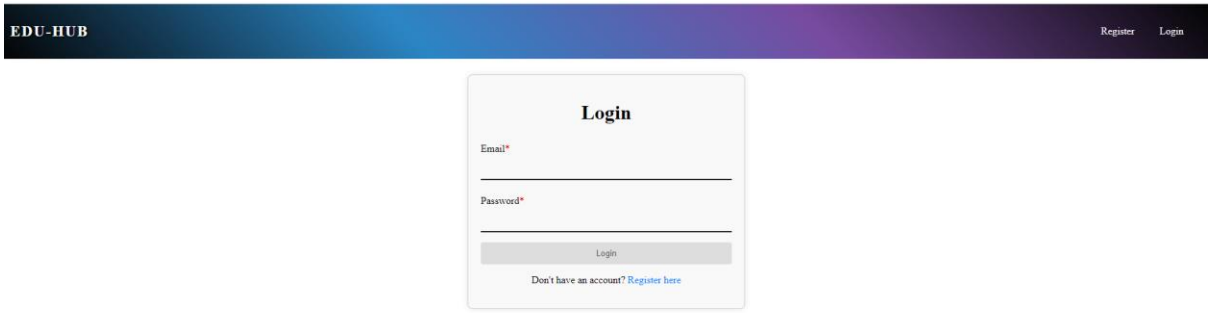


Clicking "Yes, Delete" will delete the feedback posted, and the change will be automatically reflected.

On clicking the "Logout" button, a pop-up should be displayed with confirmatory message to logout the user.

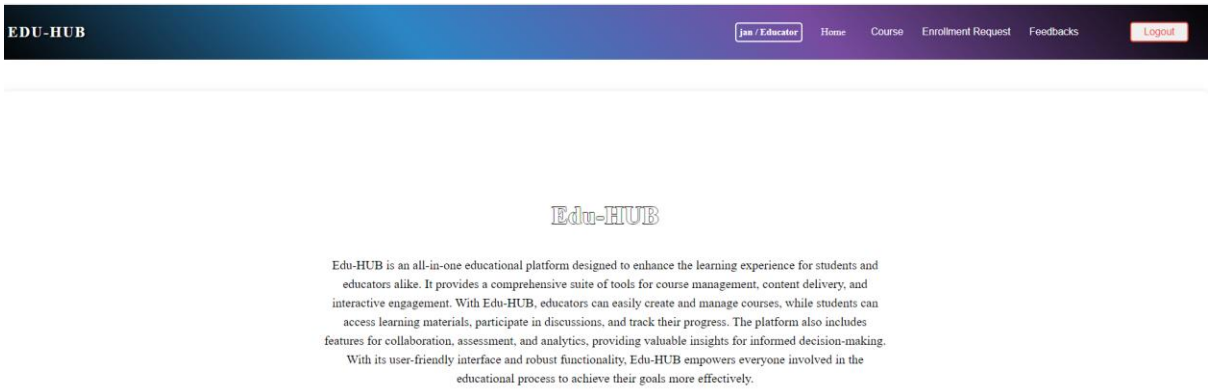


Clicking "Yes, Logout" will navigate to the login component.



<u>Educator side:</u>

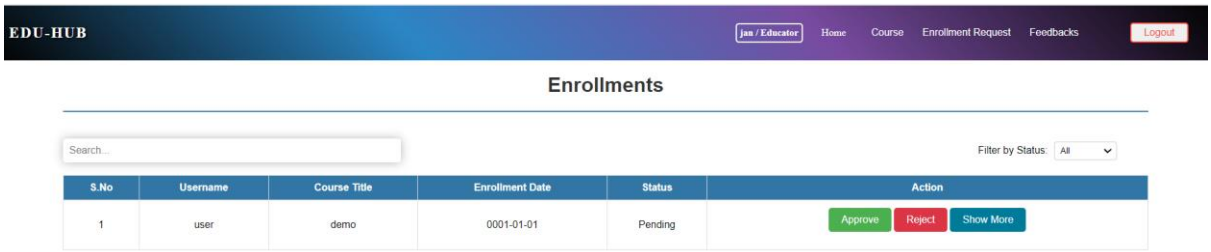
Home Component: This page is used to display the information about the loan hub application. On clicking the '**Home**' tab, user can view the information about the application.



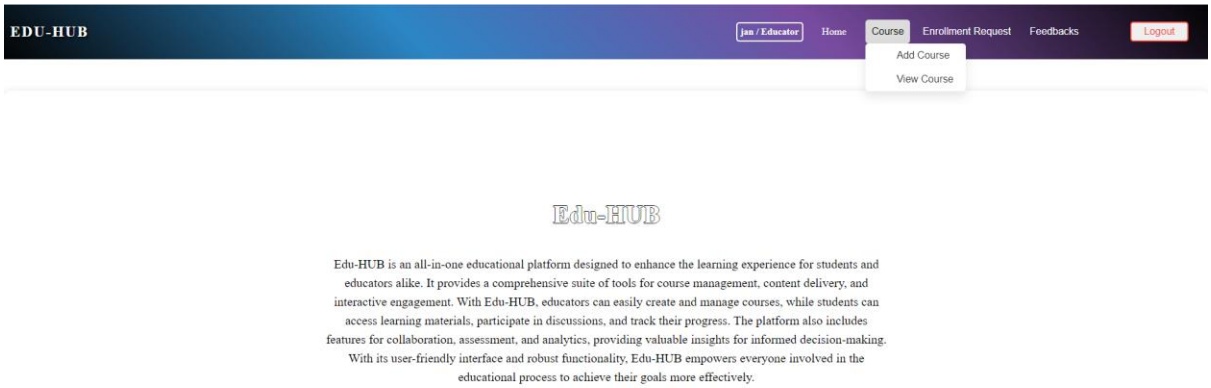
Upon successful login, if the user is an manager, the (**adminnav** component) will be displayed. If the user is a employee, the (**usernav** component) will be displayed. Additionally, the role-based navigation bar will also display login information such as the username and role.

Educators can navigate to other pages by clicking on the menu available in the navigation bar.

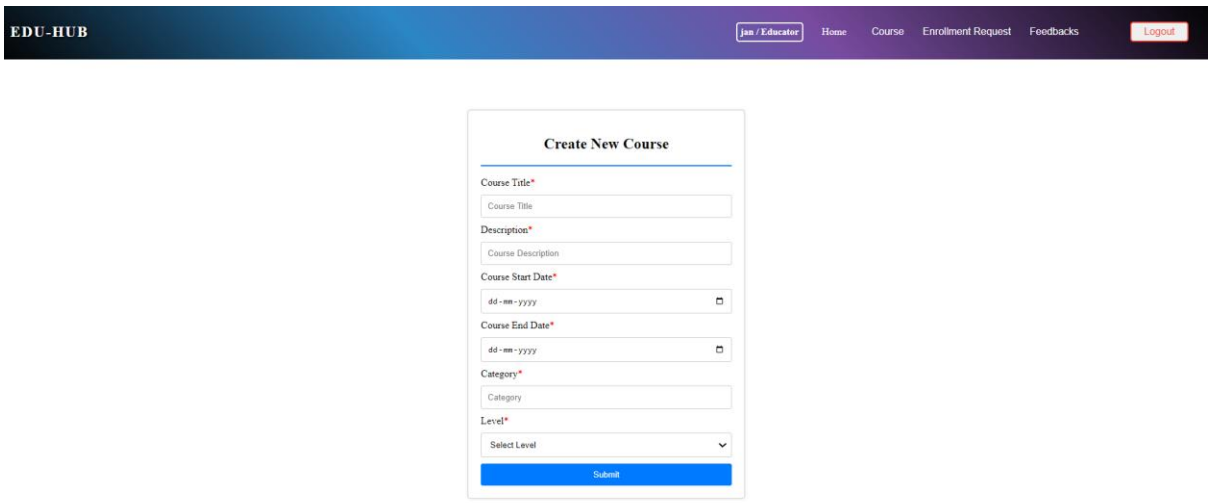
By clicking the “Enrollment Request” option will navigate to the “**Enrollment Request**” component, where the educator can approve or reject the request.



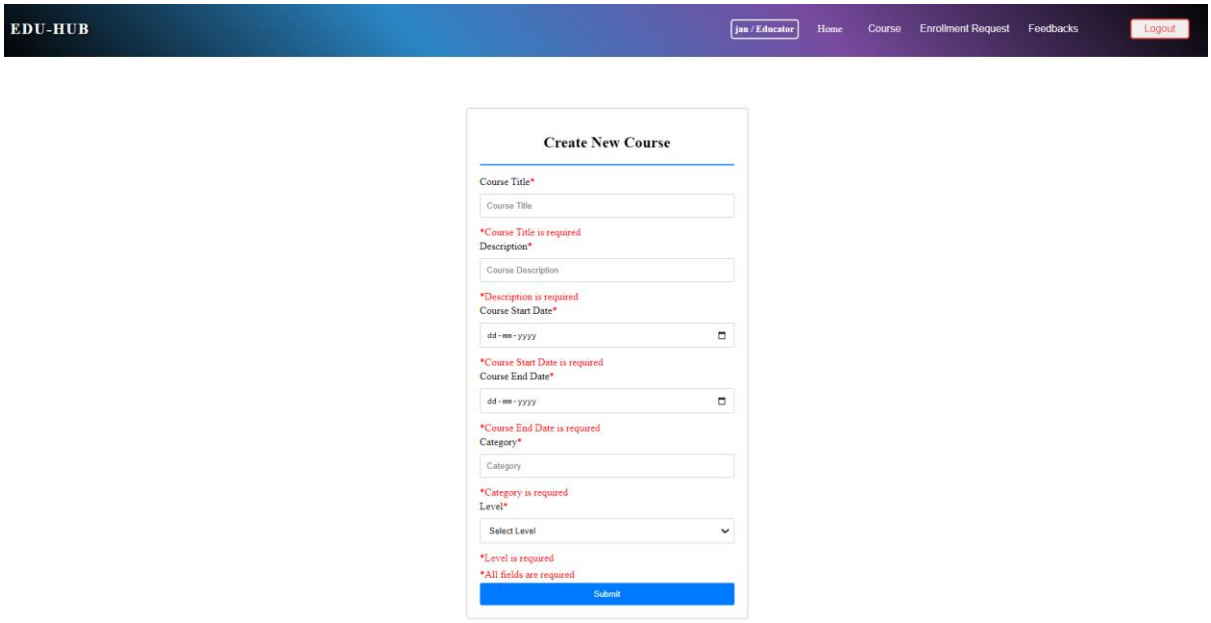
By clicking the “Course” option will navigate to the “**Course**” component, where the educator can add course and view courses.

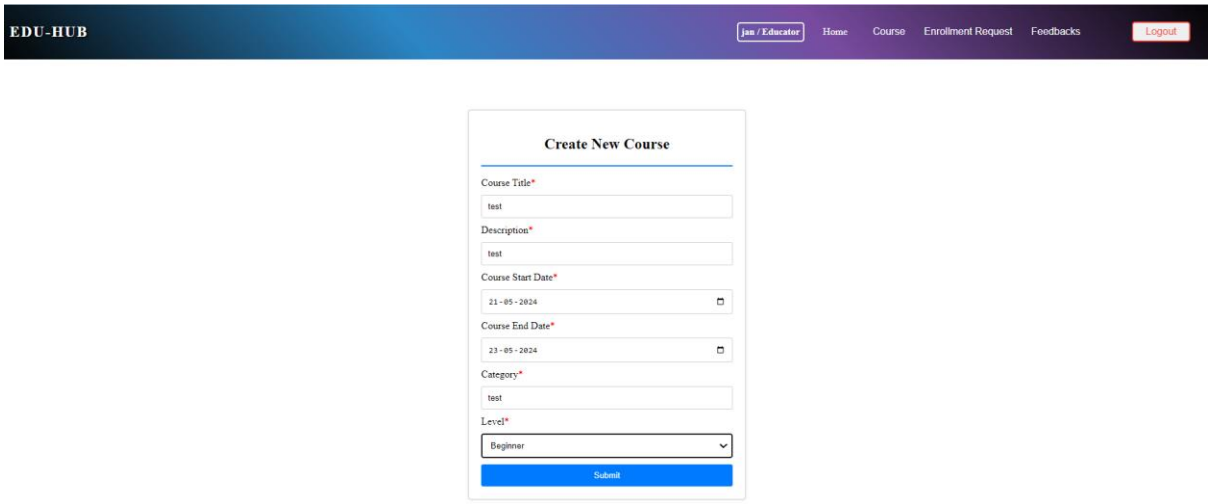


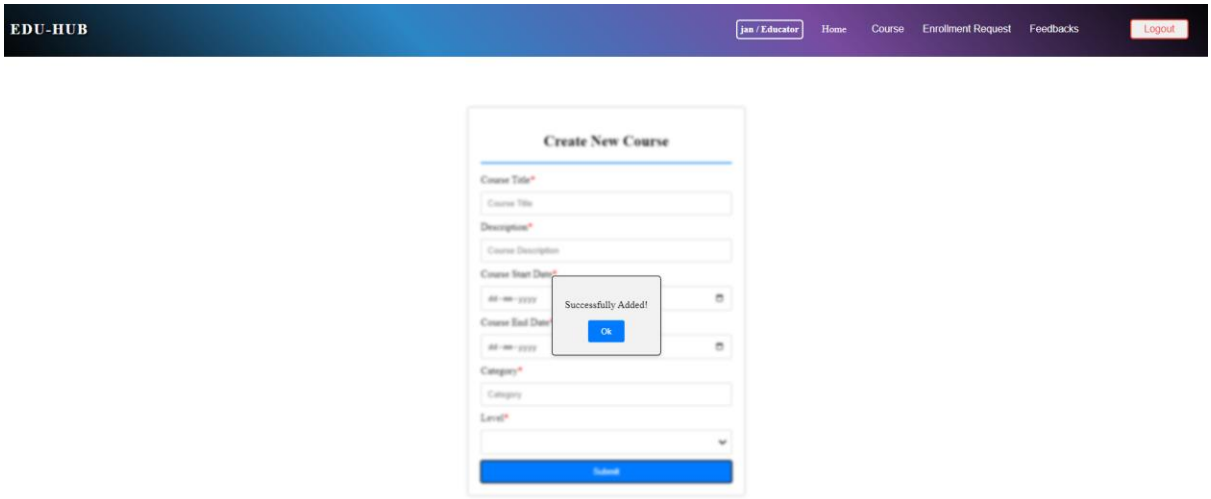
Clicking on “Add Course” will navigate to the AddCourse form component



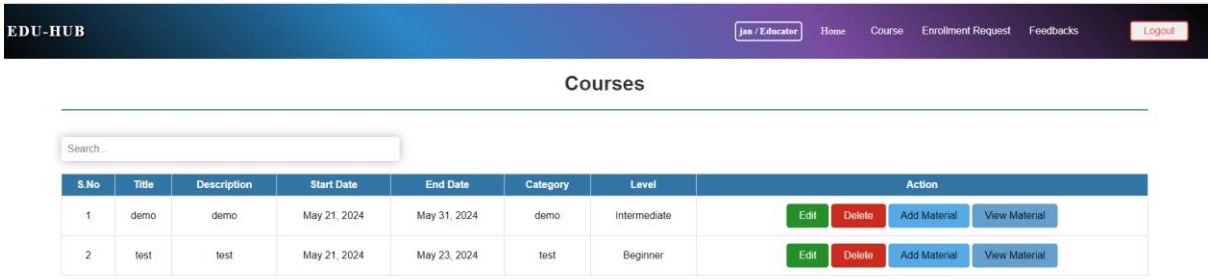
Validations when the form is empty.



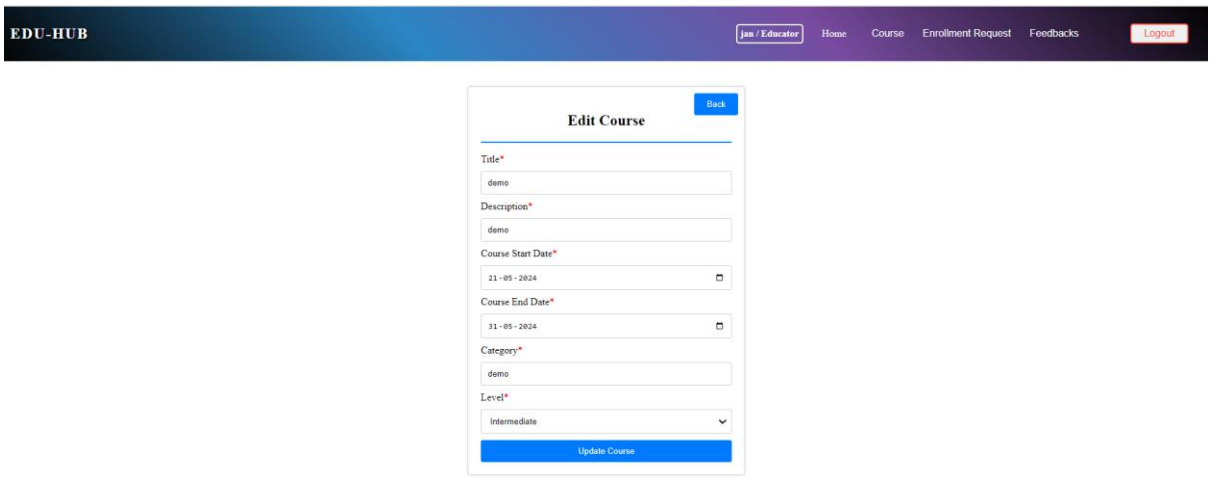




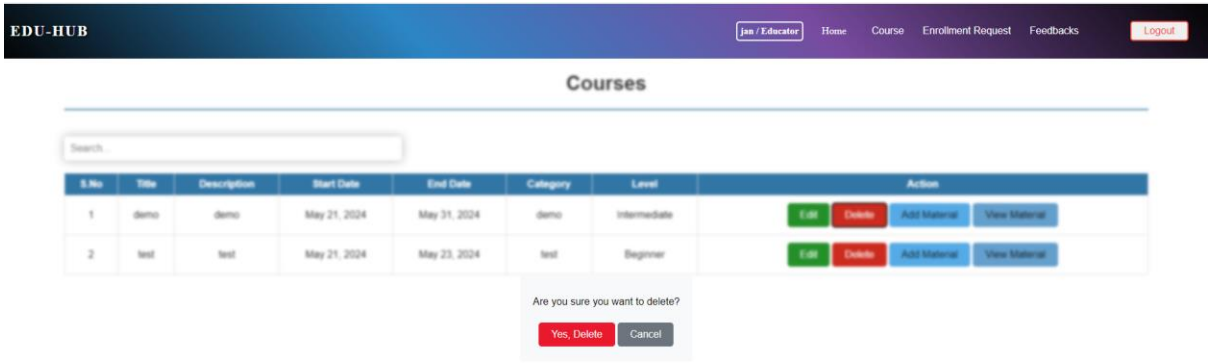
When Clicking the “View Course” it navigates to the list of courses.



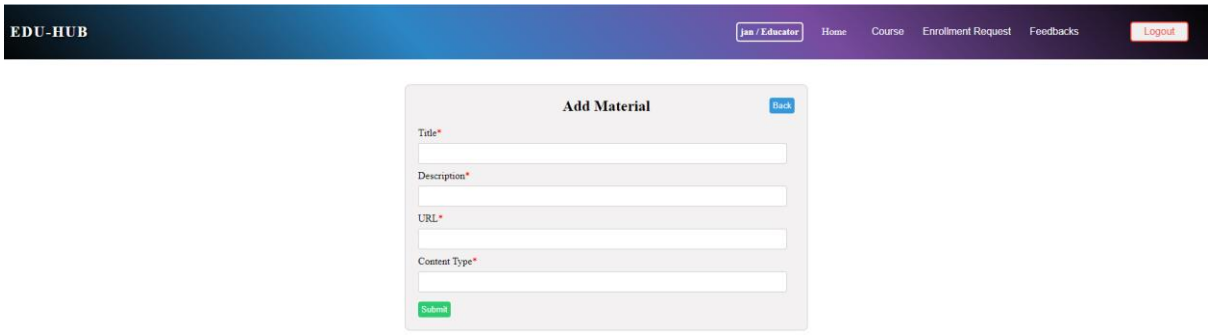
When clicking the “edit” button it must navigate to the update form component.



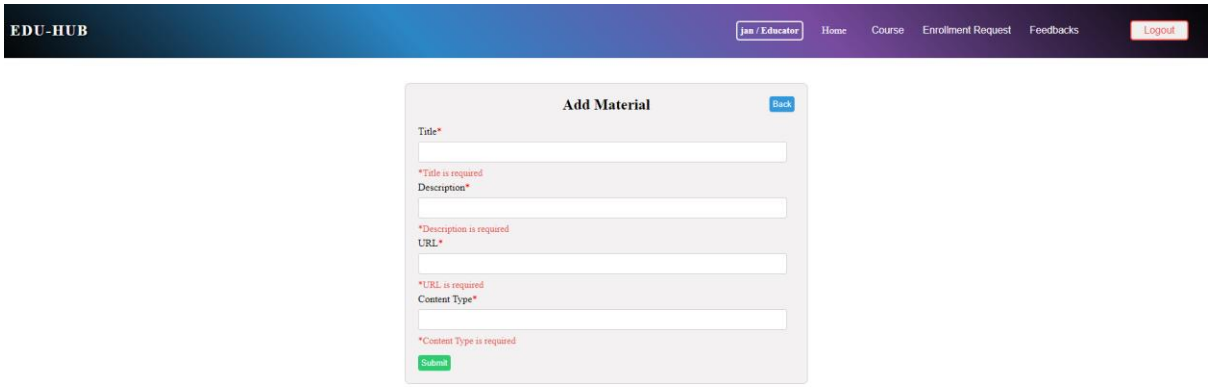
When clicking the ”delete” button it must navigate to the delete form component.

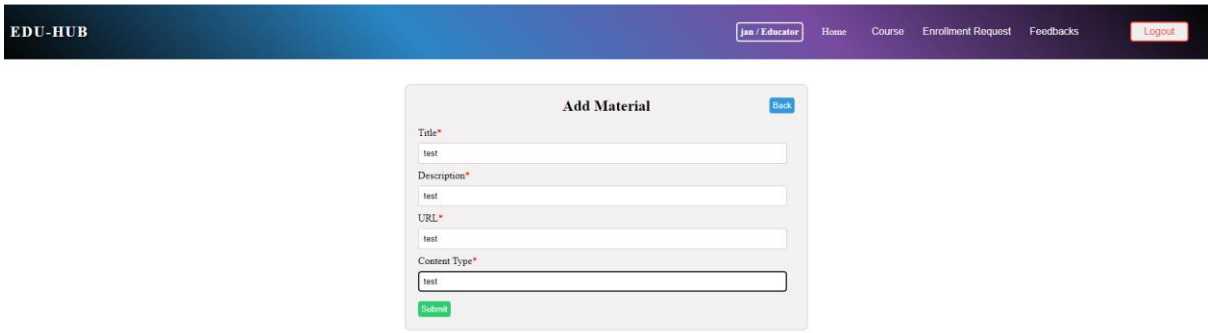


When clicking the “Add Material” button it must navigate to the MaterialForm component.

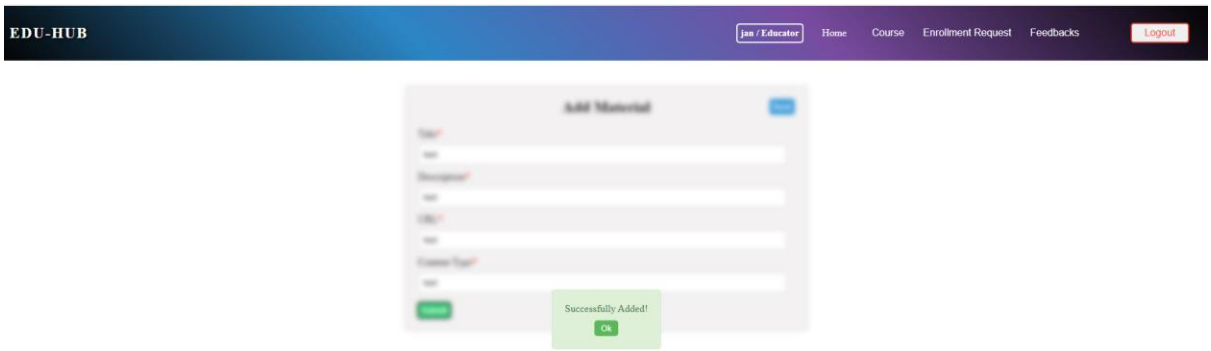


Validations of the form:

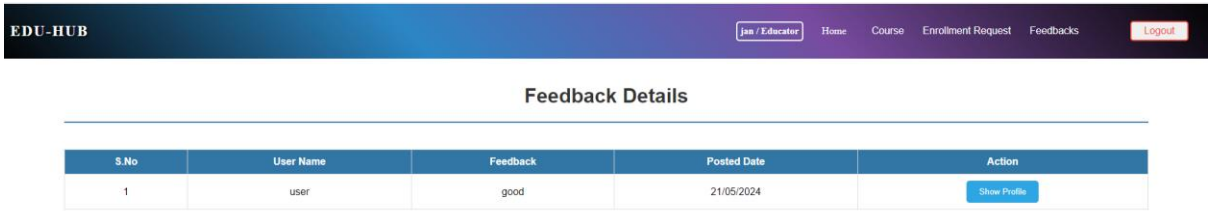




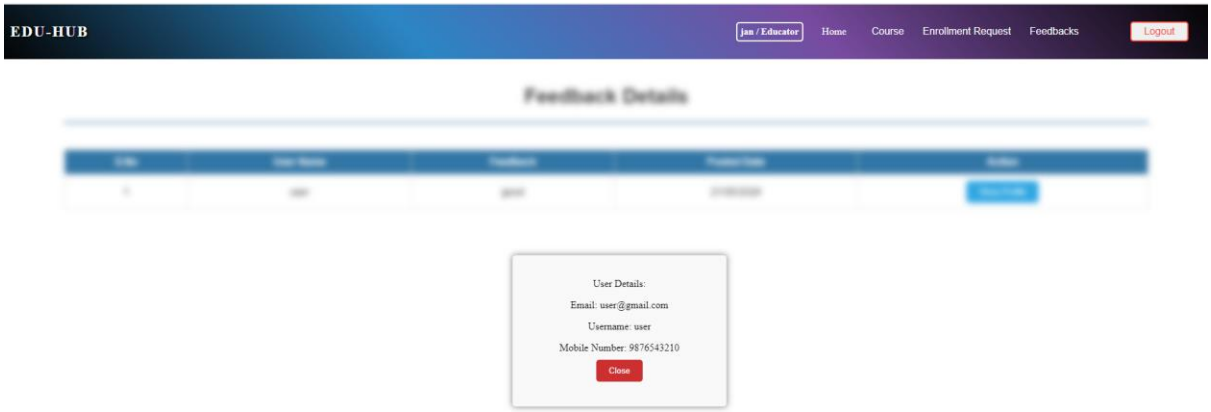
After adding the Material.



Clicking on "Feedbacks" from the navbar will navigate to the **adminviewfeedback** component, which displays all feedbacks posted by all employees.

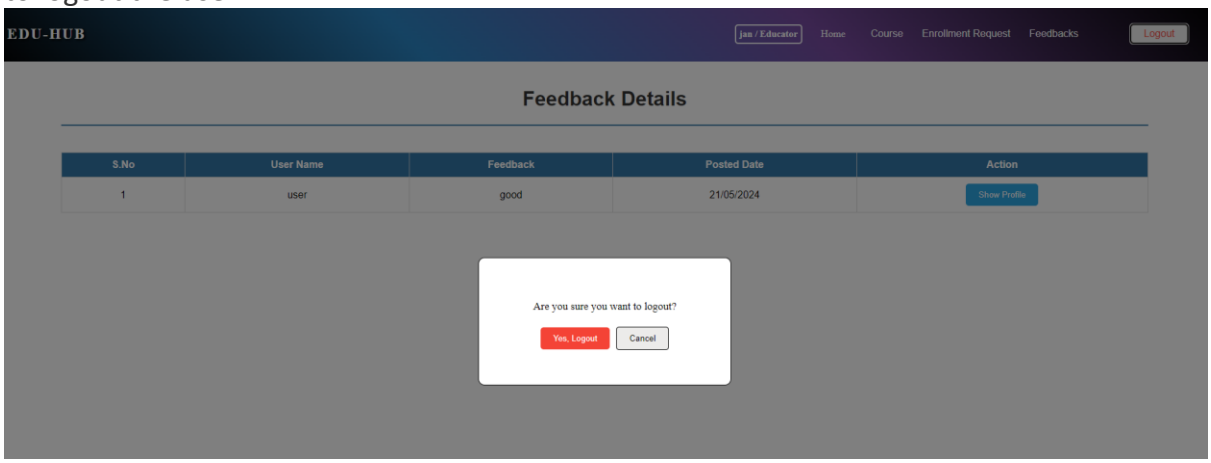


Clicking on "Show Profile" will display additional details about the user in pop-up modal.



On clicking the "Close" button, will close the pop-up modal displayed.

On clicking the "Logout" button, a pop-up should be displayed with confirmatory message to logout the user.



Clicking "Yes, Logout" will navigate to the login component.