**ASSIGNMENT 2 FRONT SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | Unit 30: Application Development | | |
| **Submission date** |  | **Date Received 1st submission** |  |
| **Re-submission Date** |  | **Date Received 2nd submission** |  |
| **Student Name** | Bui Nguyen Ngoc Han | **Student ID** | BH00150 |
| **Class** | IT0501 | **Assessor name** | Nguyen Thanh Trieu |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice. | | | |
|  |  | **Student’s signature** | Bui Nguyen Ngoc Han |

**Grading grid**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| P4 | P5 | P6 | M3 | M4 | M5 | D2 | D3 |
|  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| **❒ Summative Feedback: ❒ Resubmission Feedback:** | | |
| **Grade:** | **Assessor Signature:** | **Date:** |
| **Lecturer Signature:** | | |

Table of Contents

[**I.** **INTRODUCTION** 5](#_Toc163474260)

[**II.** **CONTENTS** 5](#_Toc163474261)

[**P4 Create a formal questionnaire that effectively reviews your business application, problem definition statement, proposed solution and development strategy. Use this questionnaire as part of a peer-review and document any feedback given** 5](#_Toc163474262)

[**1.** **Formal questionnaire** 5](#_Toc163474263)

[**2.** **The result of survey** 13](#_Toc163474264)

[**2.1.** **Personal information** 13](#_Toc163474265)

[**2.1.1.** **Gender** 13](#_Toc163474266)

[**2.1.2.** **Age group** 13](#_Toc163474267)

[**2.1.3.** **Role** 14](#_Toc163474268)

[**2.2.** **Feedback** 14](#_Toc163474269)

[**P5 Develop a functional business application based on a specified business problem** 20](#_Toc163474270)

[**1.** **Requirement** 20](#_Toc163474271)

[**2.** **Deployment** 21](#_Toc163474272)

[**3.** **Code and explain** 21](#_Toc163474273)

[**3.1.** **Login** 21](#_Toc163474274)

[LoginViewModel 21](#_Toc163474275)

[LoginQuery 22](#_Toc163474276)

[LoginController 23](#_Toc163474277)

[Login view 25](#_Toc163474278)

[**3.2.** **Dashboard** 26](#_Toc163474279)

[HomeController 26](#_Toc163474280)

[Index view 28](#_Toc163474281)

[**3.3.** **Category management** 29](#_Toc163474282)

[CategoryViewModel 29](#_Toc163474283)

[CategoryQuery 31](#_Toc163474284)

[CategoryController 38](#_Toc163474285)

[Category view 44](#_Toc163474286)

[**3.4.** **Course management** 53](#_Toc163474287)

[CoursesViewModel 53](#_Toc163474288)

[CourseQuery 55](#_Toc163474289)

[CoursesController 61](#_Toc163474290)

[Courses view 69](#_Toc163474291)

[**3.5.** **Topic management** 75](#_Toc163474292)

[TopicsViewModel 75](#_Toc163474293)

[TopicQuery 77](#_Toc163474294)

[TopicsController 80](#_Toc163474295)

[Topics view 80](#_Toc163474296)

[**4.** **Result** 81](#_Toc163474297)

[**4.1.** **Login** 81](#_Toc163474298)

[**4.2.** **Dashboard** 81](#_Toc163474299)

[**4.3.** **Category management** 82](#_Toc163474300)

[**4.4.** **Course management** 85](#_Toc163474301)

[**4.5.** **Topic management** 88](#_Toc163474302)

[**P6 Review the performance of your business application against the Problem Definition Statement and initial requirements** 92](#_Toc163474303)

[**1.** **Results of the project** 92](#_Toc163474304)

[**2.** **Test plan** 92](#_Toc163474305)

[**3.** **Test case** 92](#_Toc163474306)

[**M3 Interpret your peer-review feedback and identify opportunities not previously considered** 92](#_Toc163474307)

[**M4 Develop a functional business application based on a specific Software Design Document with supportive evidence of using the preferred tools, techniques and methodologies** 92](#_Toc163474308)

[**M5 Analyse the factors that influence the performance of a business application and use them to undertake a critical review of the design, development and testing stages of your application. Conclude your review by reflectively discussing your previously identified risks** 92](#_Toc163474309)

[**III.** **CONCLUSION** 92](#_Toc163474310)

[**IV.** **REFERENCES** 92](#_Toc163474311)

1. **INTRODUCTION**

To continue the previous report, in this report, I will address the following sections: First, Create a formal questionnaire that effectively reviews your business application, problem definition statement, proposed solution and development strategy. Use this questionnaire as part of a peer-review and document any feedback given. Second, Develop a functional business application based on a specified business problem. Third, Review the performance of your business application against the Problem Definition Statement and initial requirements. And to be able to understand these three parts more deeply, I will provide additional information and knowledge later. These include: First, Interpret your peer-review feedback and identify opportunities not previously considered. The second is Develop a functional business application based on a specific Software Design Document with supportive evidence of using the preferred tools, techniques and methodologies. Finally, Analyze the factors that influence the performance of a business application and use them to undertake a critical review of the design, development and testing stages of your application. Conclude your review by reflectively discuss your previously identified risks.

1. **CONTENTS**

## **P4 Create a formal questionnaire that effectively reviews your business application, problem definition statement, proposed solution and development strategy. Use this questionnaire as part of a peer-review and document any feedback given**

1. **Formal questionnaire**

In this section, I will create a survey questionnaire to collect data to evaluate and develop an internal training management system. Subjects participating in this survey will include: Administrator, Training staff, Trainer. Responses will be recorded for evaluation purposes.

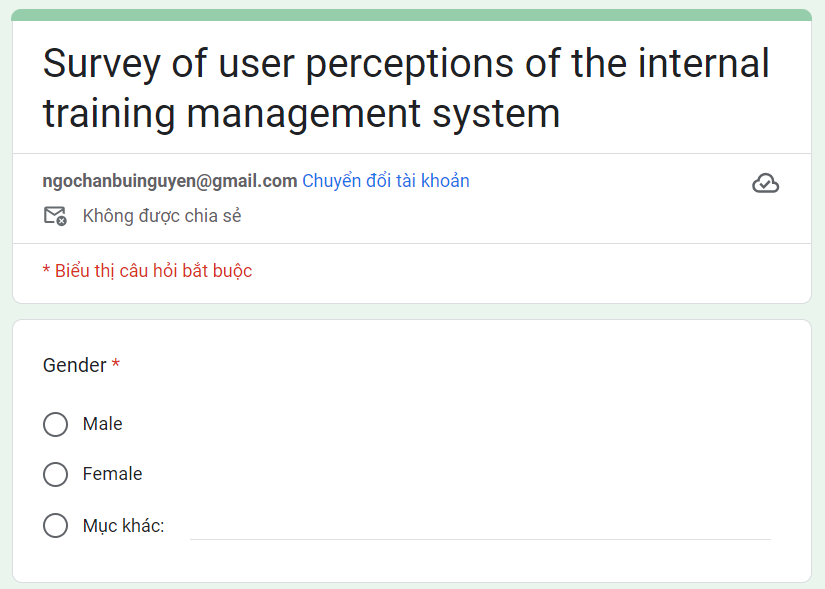
Below is the survey form I created:

Figure 1: Survey 1

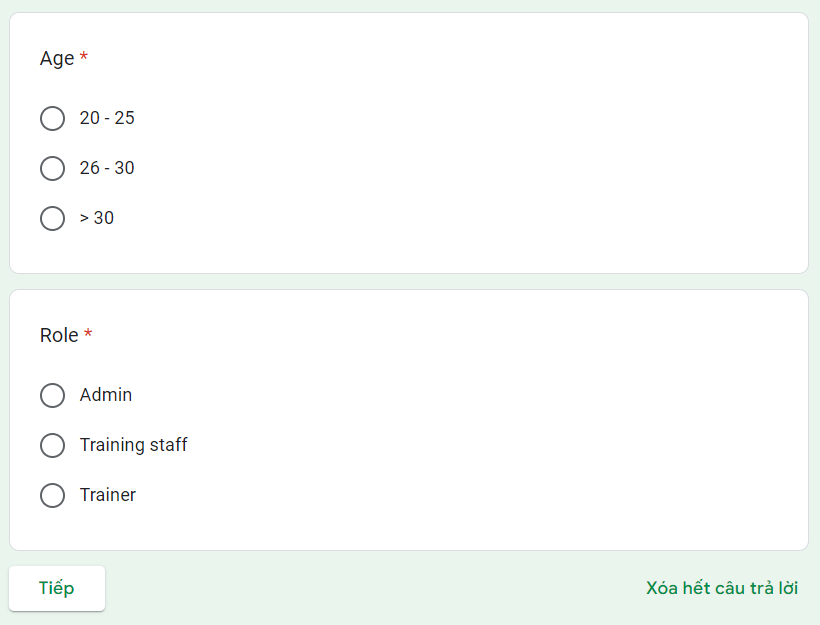


Figure 2: Survey 2

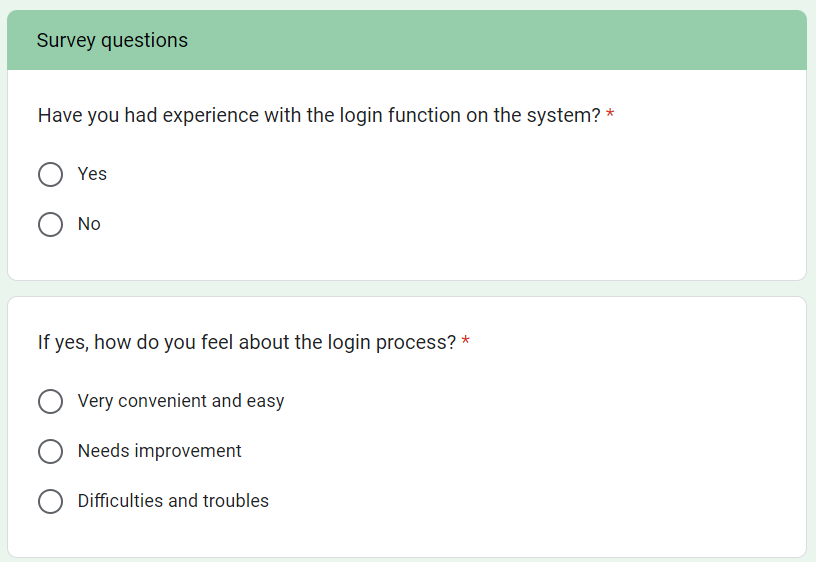


Figure 3: Survey 3

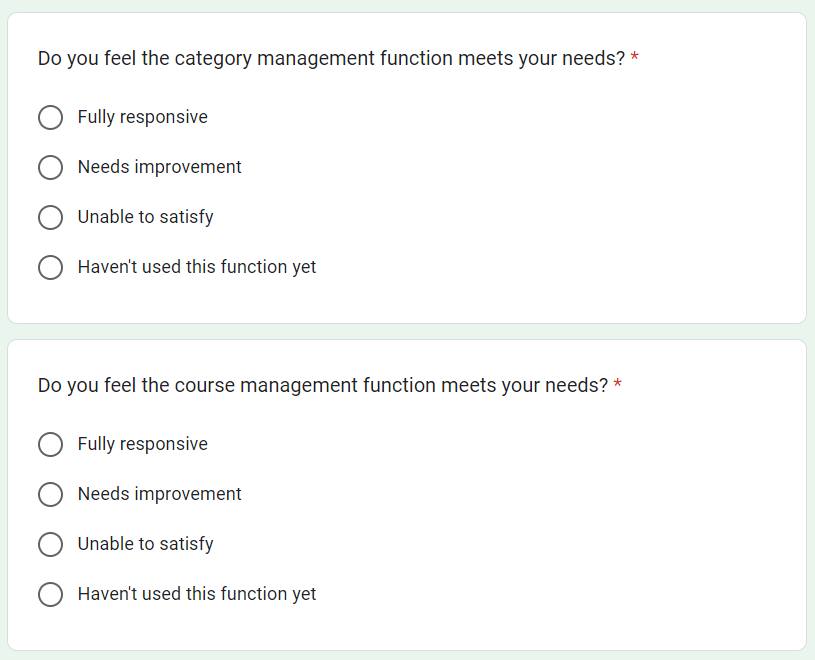


Figure 4: Survey 4

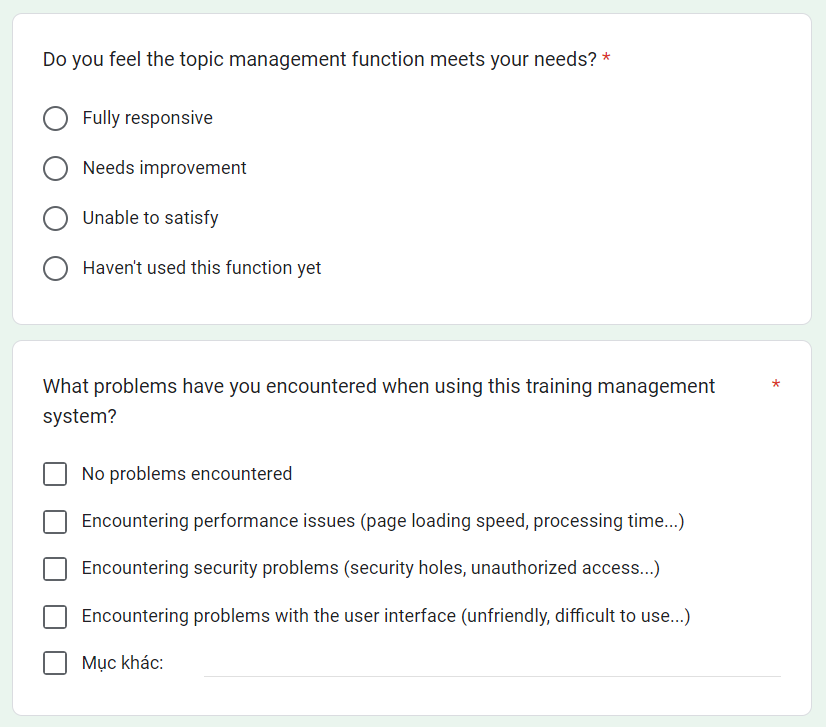


Figure 5: Survey 5

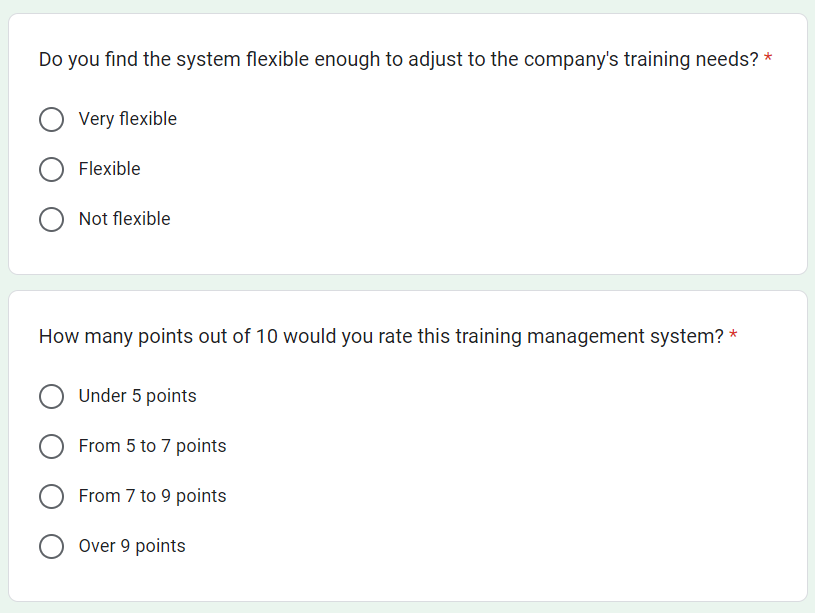


Figure 6: Survey 6

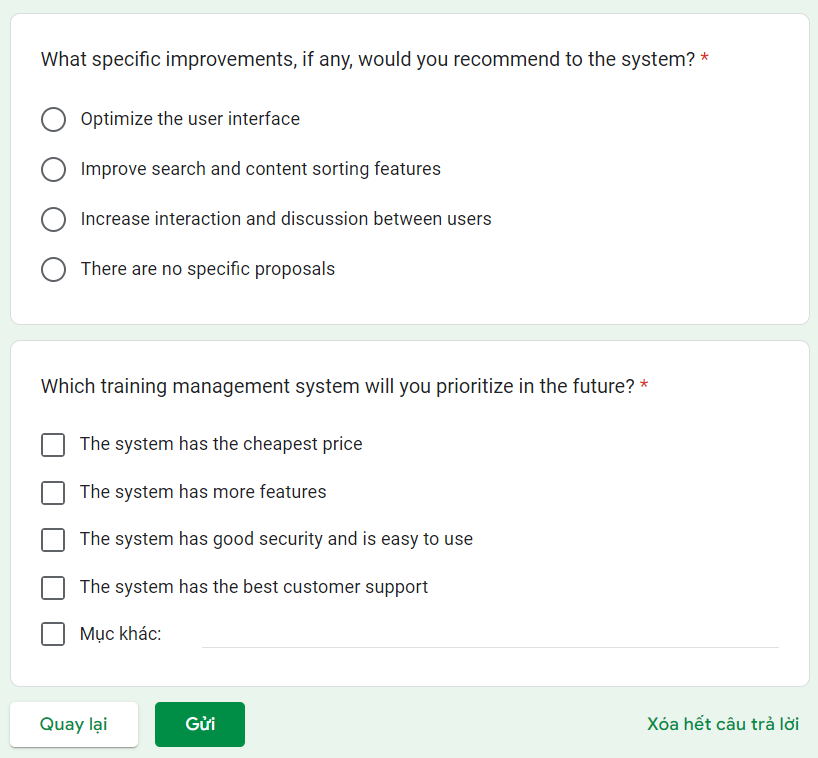


Figure 7: Survey 7

1. **The result of survey**
   1. **Personal information**
      1. **Gender**

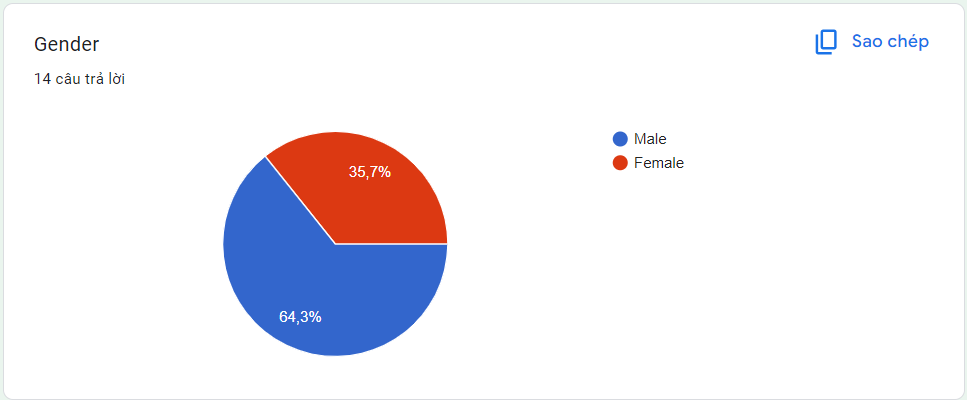
Gender statistics obtained from the survey form show that male gender accounts for 64.3% and female gender accounts for 35.7%:

Figure 8: Gender

* + 1. **Age group**

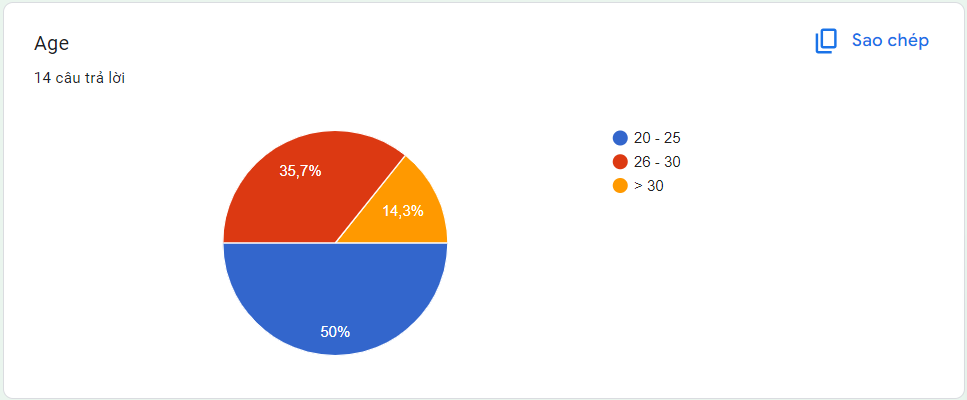
Age statistics obtained from the survey form show that 20-25 years old accounts for 50%, 26-30 accounts for 35.7% and 14.3% for those over 30 years old:

Figure 9: Age

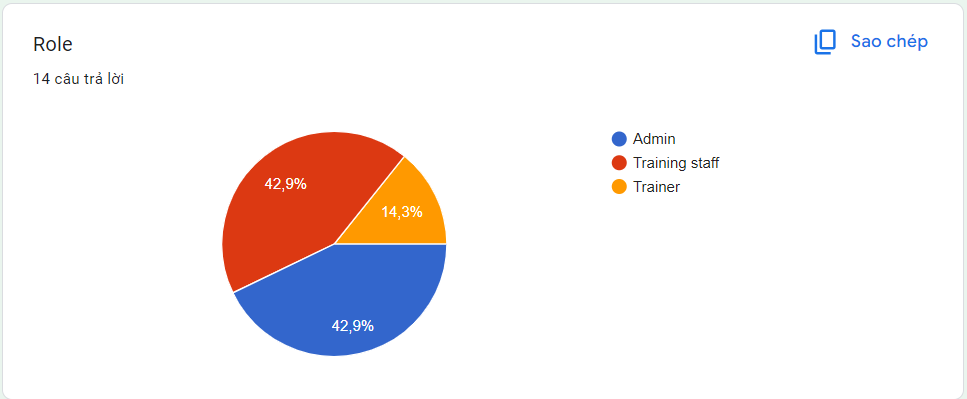
* + 1. **Role**

Figure 10: Role

* 1. **Feedback**
     1. **Have you had experience with the login function on the system?**

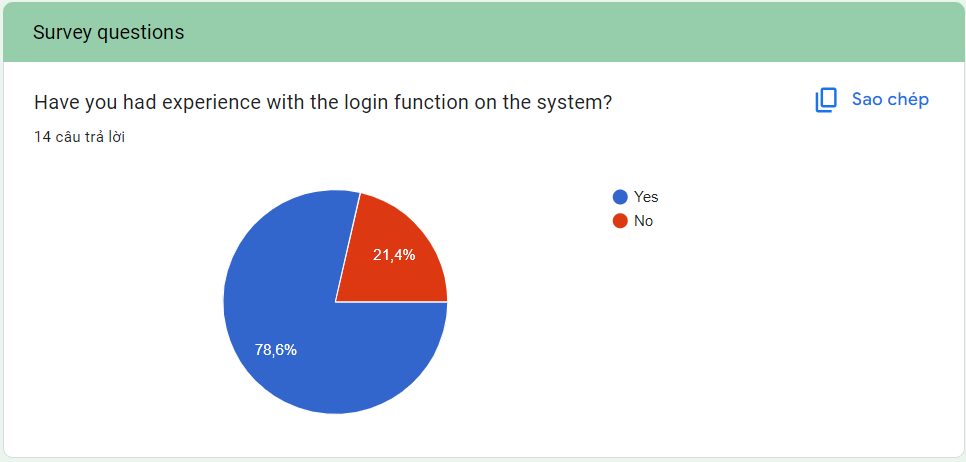
The survey form show that 78.6% for Yes and 21.4% for No

Figure 11: Question 1

* + 1. **If yes, how do you feel about the login process?**

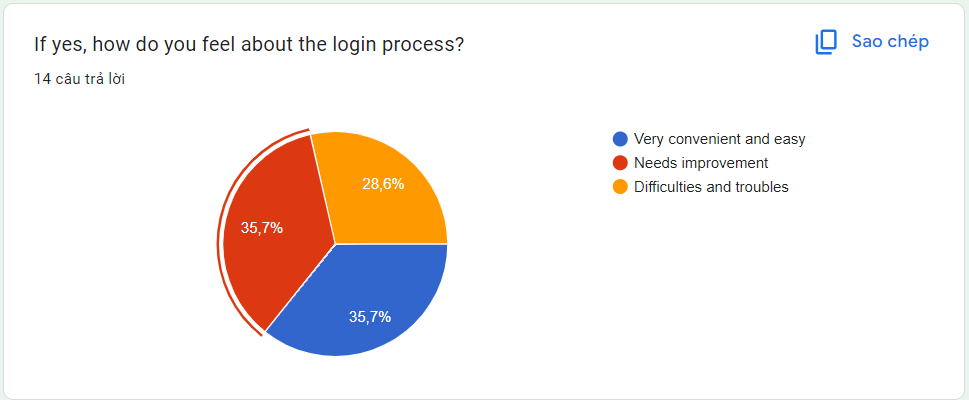
35.7% chose Very convenient and easy and Needs improvement, the remaining 28.6% chose Difficulties and troubles:

Figure 12: Question 2

* + 1. **Do you feel the category management function meets your needs?**

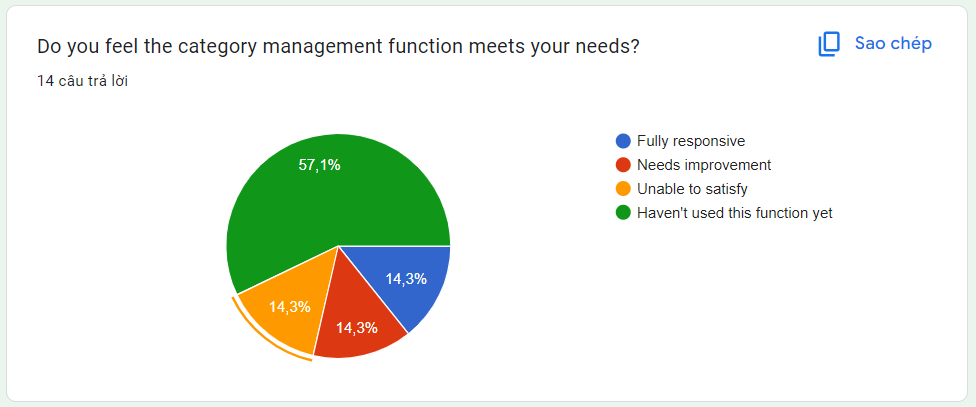
There are 57.1% choosing Haven't used this function yet and there are 14.3% for the 3 answers: Fully responsive, Needs improvement, Unable to satisfy:

Figure 13: Question 3

* + 1. **Do you feel the course management function meets your needs?**

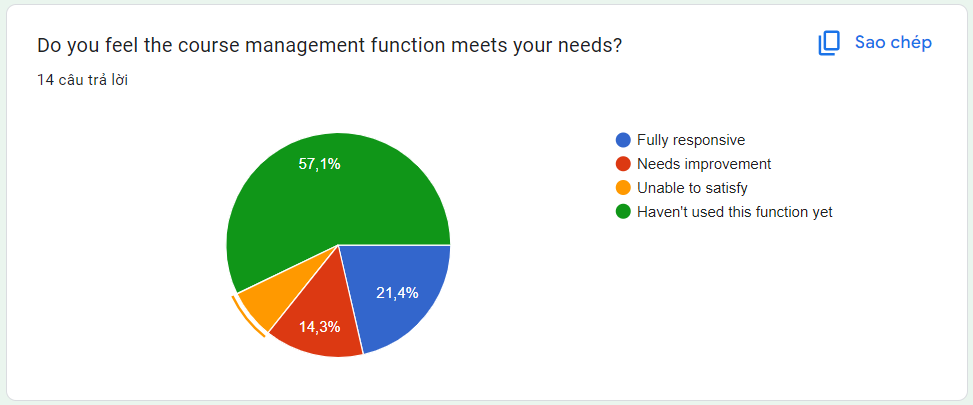
57.1% chose Haven't used this function yet, 21.4% chose Fully responsive, 14.3% chose Needs improvement and the remaining 7.1% chose Unable to satisfy:

Figure 14: Question 4

* + 1. **Do you feel the topic management function meets your needs?**

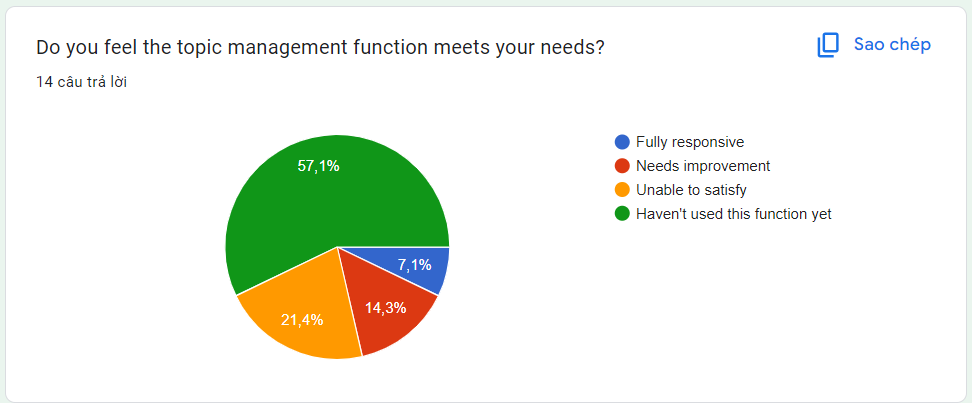
57.1% chose Haven't used this function yet, 21.4% chose Unable to satisfy, 14.3% chose Needs improvement and the remaining 7.1% chose Fully responsive:

Figure 15: Question 5

* + 1. **What problems have you encountered when using this training management system?**

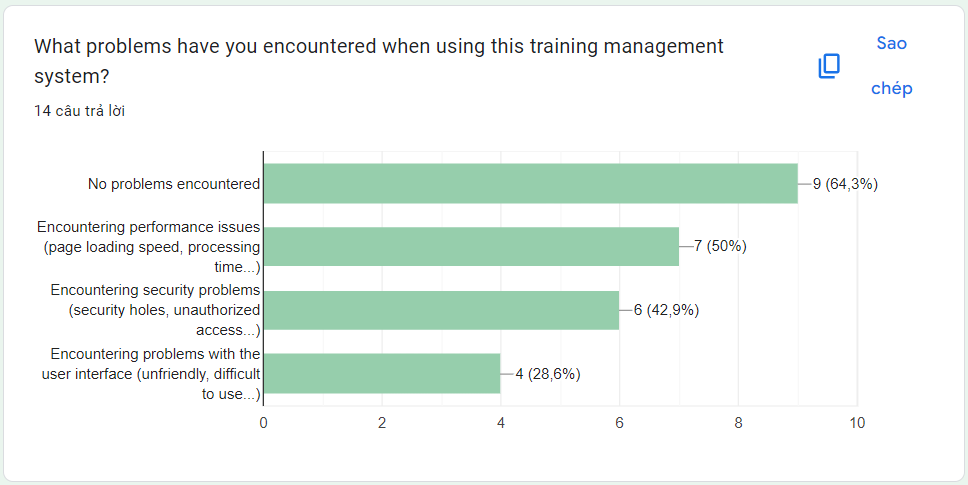
64.3% chose the answer No problems encountered, 50% chose the answer Encountering performance security issues (page loading speed, processing time...), 42.9% chose Encountering problems (security holes, unauthorized access...) and 28.6% chose remaining answers:

Figure 16: Question 6

* + 1. **Do you find the system flexible enough to adjust to the company's training needs?**

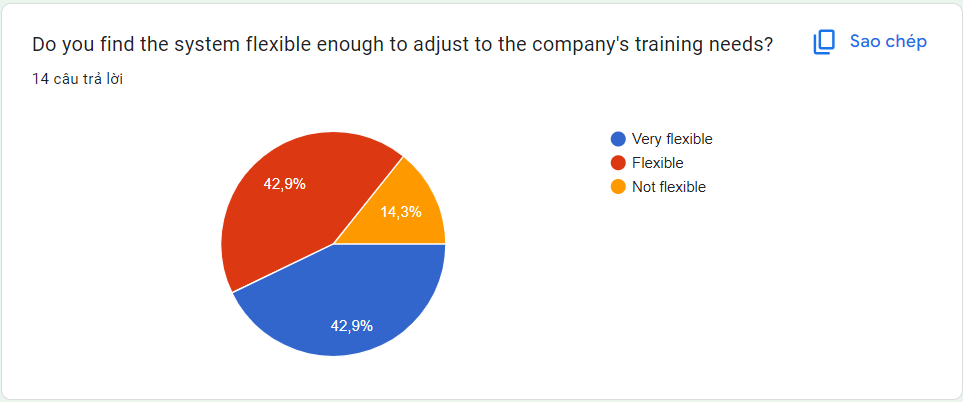
The two answers Very flexible and Flexible both have a percentage of 42.9%, while the answer Not flexible has 14.3%:

Figure 17: Question 7

* + 1. **How many points out of 10 would you rate this training management system?**

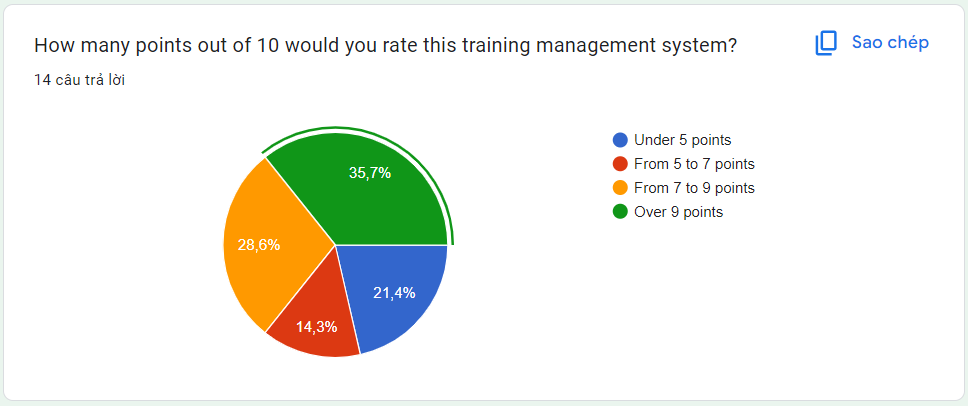
35.7% chose the answer Over 9 points, 28.6% chose the answer From 7 to 9 points, 21.4% chose the answer and the remaining 14.3% chose From 5 to 7 points:

Figure 18: Question 8

* + 1. **What specific improvements, if any, would you recommend to the system?**

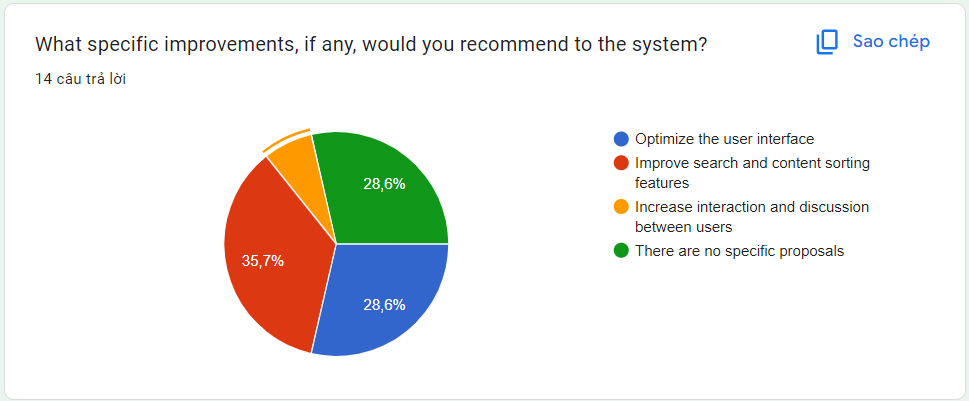
35.7% chose the answer Improve search and content sorting features, 28.6% gave two answers: Optimize the user interface and There are no specific proposals, the remaining 7.1% chose Increase interaction and discussion between users:

Figure 19: Question 9

* + 1. **Which training management system will you prioritize in the future?**

For the question Which training management system will you prioritize in the future?, the answers obtained from the survey form show:

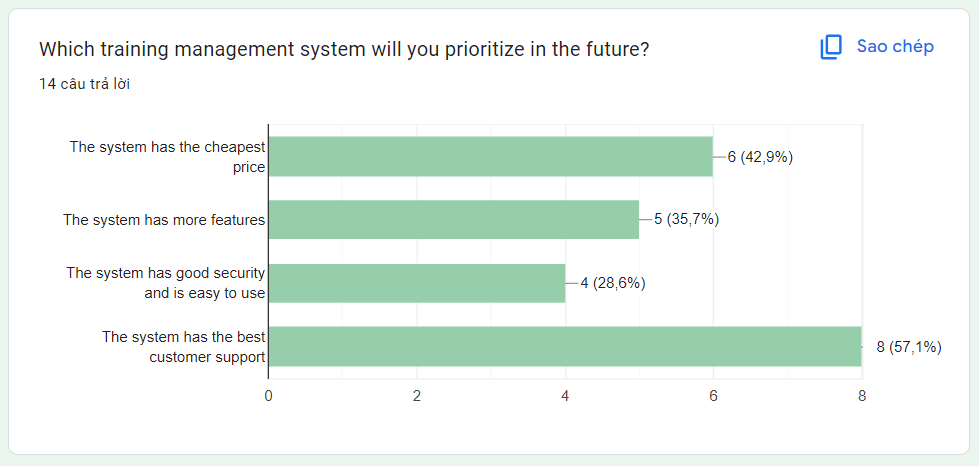
* 57.1% of survey participants chose The system has the best customer support
* 42.9% of survey participants chose the answer The system has the cheapest price
* 35.7% of survey participants chose the answer The system has more features
* And the remaining 28.6% chose the answer The system has good security and is easy to use

Figure 20: Question 10

## **P5 Develop a functional business application based on a specified business problem**

1. **Requirement**

FPT Company wishes to build a continuous learning environment throughout the corporation. They want to develop a web system to manage "Training" activities for the company's internal training program. Regarding requirements, this system can be used to manage trainee accounts, manage trainers, manage course categories, manage courses, manage topics, assign topic to course, assign trainer to topic, assign trainee to course. This is a system used by HR department.

There are three roles in this system, an administrator, training staff and a trainer. The brief description of those roles is as follow.

*Administrator’s role:*

* Can login to the system through the first page of the application
* Can create/edit/delete new user account for trainer/training staff and assign/change (if existing user) username and a password

*Training staff’s role:*

* A registered training staff, who is assigned a user name and a password by the administrator logs in can create trainee accounts by entering details like trainee name, trainee accounts, age, date of birth, education, main programming language, TOEIC score, experience details, department, location, etc.
* After entering successfully all details for trainees, his/her details are then stored in the database. The training staff is given a list of trainees for him to view and search. From the list of trainees, he can also search by trainee account, programming language, TOEIC score…
* Can update, delete trainee accounts
* Can manage course categories such as searching, adding, updating and deleting course categories. Course category includes the information such as course category name and descriptions.
* Can manage courses such as searching, adding, updating and deleting courses. Course includes course name and description.
* Can add topics such as topic name and topic descriptions into a course, add courses into a category.
* Can manage trainer profile such as adding, updating and deleting the information: Trainer name, External or Internal Type, working place, telephone, and email address.
* Can assign trainer to a topic.
* Can assign trainee to a course.

*Trainer’s role:*

* In the same system, the trainer who have been registered by the administrator can login and can update his profile such as Trainer name, External or Internal Type, education, working place, telephone, and email address.
* Can view courses which have a topic he is assigned to.

1. **Deployment**

Below are the deployment steps I will take:

* Step 1: Prepare the environment. Set up the deployment environment, ensuring it meets system requirements. In this project, I chose Visual Studio 2022 to deploy the project.
* Step 2: Deploy the database. Back up and deploy the system's database to the deployment environment. I choose SQLServer to deploy this project.
* Step 3: Deploy the application. Back up and deploy application code to the deployment environment. I use C language to do code implementation.
* Step 4: Testing. Conduct test planning and execute test cases.

1. **Code and explain**
   1. **Login**

##### LoginViewModel

Figure 21: LoginViewModel

This code defines a class named “LoginViewModel”, which is typically used to represent the data associated with a user's login information. Here's an explanation of each part of the code:

* LoginViewModel class: This is a class definition in C. It encapsulates the properties related to a user's login information:
  + Id: Represents the unique identifier of the user.
  + RoleId: Represents the role identifier of the user, which defines their permissions or access level within the system.
  + UserName: Represents the username of the user used for login purposes.
  + Password: Represents the password of the user used for login authentication.
  + Email: Represents the email address associated with the user's account.
  + Phone: Represents the phone number associated with the user's account.
  + Extracode: Represents any additional code or information associated with the user, which might be specific to the application.
* Each property has a getter and setter method, denoted by { get; set; }. These methods allow getting and setting the values of the properties respectively.
* The ? after the string type (string?) indicates that these properties can be assigned a null value. This allows for more flexibility when handling potentially nullable data.

##### LoginQuery

Figure 22: LoginQuery

This code checks the user's login information in the database and returns an object containing the user's information if the login information is valid:

* LoginQuery class: This is a class in the application, containing a method to check the user's login information.
* CheckUserLogin method: This is the method used to check the user's login information. It takes two parameters, username and password, representing the username and password provided by the user.
* LoginViewModel object: This is an object of the LoginViewModel class initialized to store information about the user after logging in.
* SqlConnection object: This is an SqlConnection object used to connect to the database. The GetSqlConnection() method is called to retrieve an SqlConnection object from some Database class.
* SqlCommand object: This is a SqlCommand object used to execute SQL queries on the database. The SQL query in the querySql variable is used to retrieve user information from the Users table.
* SqlParameter object: These are SqlParameter objects used to add parameters to the SQL query. In this case, @username and @password are used to avoid security issues like SQL injection.
* SqlDataReader object: This is a SqlDataReader object used to read rows of data from the result of the SQL query. Data from each row is read and assigned to the corresponding properties of the dataUser object.
* return statement: The method returns the dataUser object, containing information about the user after logging in.

##### LoginController

Figure 23: LoginController

This is a LoginController class in an ASP.NET MVC application, responsible for handling user authentication and session management. Here's an explanation of each part of the code:

* LoginController class: This is a controller class that inherits from the Controller base class provided by ASP.NET MVC framework.
* Index method (GET):
  + This method is decorated with the [HttpGet] attribute, indicating that it handles HTTP GET requests to the URL associated with the action method.
  + It returns an IActionResult, typically representing a view.
  + It initializes a new instance of the LoginViewModel class and passes it to the associated view.
* Index method (POST):
  + This method is decorated with the [HttpPost] attribute, indicating that it handles HTTP POST requests to the URL associated with the action method.
  + It takes a LoginViewModel object model as a parameter, which contains the username and password submitted by the user.
  + It invokes the CheckUserLogin method from a LoginQuery class (which is presumably responsible for interacting with the database to verify user credentials) to authenticate the user.
  + If authentication fails (i.e., if Id or Email properties of the model are empty), it sets an error message in ViewData and returns the view with the model.
  + If authentication succeeds, it stores the user's information in the session (HttpContext.Session) and redirects the user to the Index action of the HomeController.
* Logout method:
  + This method is decorated with the [HttpPost] attribute, indicating that it handles HTTP POST requests to the URL associated with the action method.
  + It removes the user's information from the session if the user is logged in.
  + It redirects the user to the Index action of the LoginController after logout.

##### Login view

Figure 24: Login - View

* @model TrainingFPT.Models.LoginViewModel: This directive specifies the model type that this view expects to receive.
* @{ ... }: This block of code is used to write C code within the Razor view. In this case, it sets the layout of the view and defines the model.
* Layout = "~/Views/Shared/\_LayoutLogin.cshtml";: This line sets the layout file for this view. It specifies the path to the layout file relative to the Views folder.
* @if (ViewData["MessageLogin"] != null) { ... }: This conditional statement checks if there is an error message stored in the ViewData dictionary with the key "MessageLogin". If it exists, it displays the error message in red text.
* @using(Html.BeginForm("Index", "Login", FormMethod.Post, new { @class="user" })) { ... }: This HTML helper method generates the opening <form> tag for the login form. It specifies the action (Index method in LoginController), controller (LoginController), HTTP method (POST), and additional HTML attributes (class="user").
* @Html.TextBoxFor(m => m.UserName, new { @class = "form-control form-control-user" }): This HTML helper method generates an input field for the username, binding it to the UserName property of the LoginViewModel.
* @Html.PasswordFor(m => m.Password, new { @class = "form-control form-control-user" }): This HTML helper method generates a password input field, binding it to the Password property of the LoginViewModel.
* <button type="submit" class="btn btn-primary btn-user btn-block">Login</button>: This is a submit button for the login form. When clicked, it submits the form data to the server for processing.
  1. **Dashboard**

##### HomeController

Figure 25: HomeController

This is a controller class named HomeController in an ASP.NET MVC application. Below is an explanation of each part of the code:

* HomeController class: This is a controller class that inherits from the Controller base class provided by ASP.NET MVC framework.
* Constructor:
  + ILogger<HomeController> \_logger: This private field is of type ILogger<HomeController>, which is used for logging messages related to this controller.
  + HomeController(ILogger<HomeController> logger): This is the constructor of the HomeController class. It takes an instance of ILogger<HomeController> as a parameter and assigns it to the \_logger field.
* Index method:
  + public IActionResult Index(): This is an action method responsible for handling requests to the default homepage of the application.
  + It first checks if the session variable "SessionUsername" (representing the username of the logged-in user) is null or empty.
  + If the username is null or empty, it redirects the user to the login page by calling the Index action of the LoginController.
  + If the username is not null or empty, it returns the default view associated with this action.
* Privacy method:
  + public IActionResult Privacy(): This is an action method responsible for handling requests to the privacy page of the application.
  + It simply returns the view associated with the privacy page.
* Error method:
  + public IActionResult Error(): This is an action method responsible for handling errors that occur within the application.
  + It returns a view for displaying error information, including the request ID and other details.
  + It utilizes ResponseCache attribute to ensure that this action method isn't cached, ensuring that users always receive fresh error information.

##### Index view

Figure 26: Home Index - View

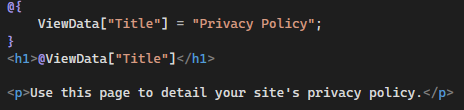
* ViewData["Title"] = "Home Page";: This sets the title of the view to "Home Page". The value of this ViewData property can be used in the layout file to dynamically set the title of the web page.
* ViewData["namePage"] = "Dashboard";: This sets the value of the "namePage" ViewData property to "Dashboard". This value can be used to customize the display or behavior of the dashboard in the layout file or other views.
* ViewBag.report = true;: This sets the value of the "report" property of the ViewBag to true. The ViewBag is a dynamic object used to pass data between the controller and the view. In this case, it seems to indicate that a report is available for display on the dashboard.
* Layout = "~/Views/Shared/\_LayoutMain.cshtml";: This line specifies the layout file for this view. It sets the layout to "\_LayoutMain.cshtml" located in the "Shared" folder within the "Views" directory.

Figure 27: Privacy view

* @{ ... }: This block of code is used to write C code within the Razor view. In this case, it sets the ViewData property to specify the title of the page.
* ViewData["Title"] = "Privacy Policy";: This sets the title of the view to "Privacy Policy". The value of this ViewData property is then used to dynamically display the title of the web page.
* <h1>@ViewData["Title"]</h1>: This line displays the title of the page using the value stored in the ViewData["Title"] property. In this case, it will render "Privacy Policy" as the main heading of the page.
* <p>Use this page to detail your site's privacy policy.</p>: This paragraph provides a brief description of the purpose of the page, instructing users to use it to find detailed information about the site's privacy policy.
  1. **Category management**

##### CategoryViewModel

Figure 28: CategorViewModel

* CategoryViewModel Class:
  + This class represents a view model for managing categories.
  + It contains a property named CategoryDetailList, which is a list of CategoryDetail objects. This property allows passing a list of category details to the view for display or processing.
* CategoryDetail Class:
  + This class represents the details of a category.
  + It contains the following properties:
    - Id: An integer property representing the unique identifier of the category.
    - Name: A string property representing the name of the category. It is marked with the [Required] attribute, indicating that it must have a value.
    - Description: A nullable string property representing the description of the category.
    - Status: A string property representing the status of the category. It is marked with the [Required] attribute, indicating that it must have a value.
    - PosterImage: An IFormFile property representing the image file for the category's poster. It is marked with custom validation attributes (AllowExtensionFile and AllowMaxSizeFile) to ensure that only files with allowed extensions and sizes are accepted.
    - PosterNameImage: A nullable string property representing the name of the poster image.
    - CreatedAt, UpdatedAt, DeletedAt: Nullable DateTime properties representing the creation, update, and deletion timestamps of the category, respectively.
* Validation Attributes:
  + Required: Indicates that the Name and Status properties must have values provided by the user. If not provided, error messages specified in the ErrorMessage property will be displayed.
  + AllowNull: Allows nullable values for properties where a null value is valid.
  + AllowExtensionFile: A custom validation attribute that specifies the allowed file extensions for the PosterImage property.
  + AllowMaxSizeFile: A custom validation attribute that specifies the maximum file size allowed for the PosterImage property.

##### CategoryQuery

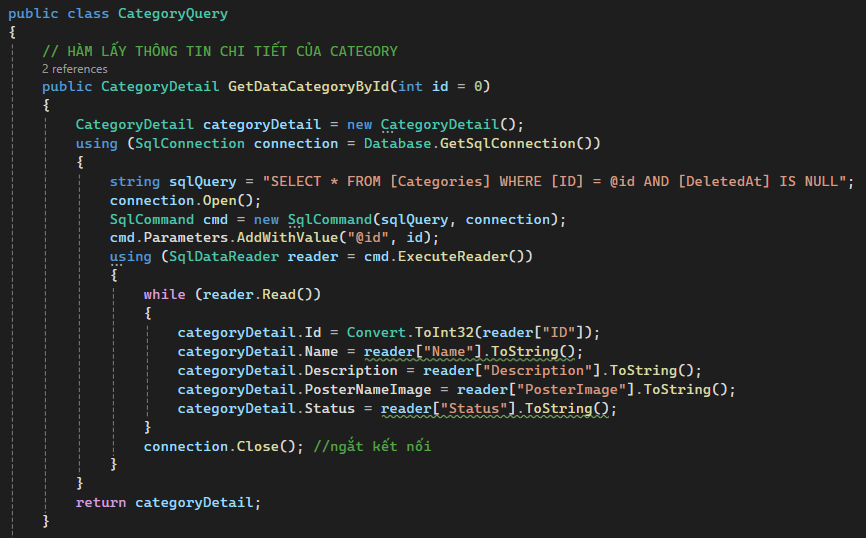


Figure 29: CategoryQuery - GetDataCategoryById

This method GetDataCategoryById retrieves category data from the database based on the provided category ID:

* public CategoryDetail GetDataCategoryById(int id = 0): This is a public method named GetDataCategoryById. It takes an integer parameter id, which represents the ID of the category to retrieve. The parameter has a default value of 0, meaning if no ID is provided, the method will attempt to retrieve data for the category with ID 0.
* CategoryDetail categoryDetail = new CategoryDetail();: This line initializes a new instance of the CategoryDetail class. This object will store the retrieved category data.
* using (SqlConnection connection = Database.GetSqlConnection()) { ... }: This line establishes a connection to the database using the Database.GetSqlConnection() method, which returns an instance of SqlConnection. The using statement ensures that the connection is properly disposed of after its use to avoid resource leaks.
* SQL Query Execution:
  + string sqlQuery = "SELECT FROM [Categories] WHERE [ID] = @id AND [DeletedAt] IS NULL";: This SQL query selects all columns from the Categories table where the ID matches the provided id parameter and the DeletedAt column is NULL.
  + SqlCommand cmd = new SqlCommand(sqlQuery, connection);: This line creates a new SqlCommand object with the SQL query and the database connection.
  + cmd.Parameters.AddWithValue("@id", id);: This line adds a parameter to the SQL command to prevent SQL injection attacks. The value of the @id parameter is set to the id parameter passed to the method.
  + using (SqlDataReader reader = cmd.ExecuteReader()) { ... }: This block executes the SQL command using the ExecuteReader() method, which returns a SqlDataReader object containing the results of the query.
* Data Retrieval:
  + Inside the while (reader.Read()) { ... } loop, the method reads each row of data returned by the query.
  + Data from the reader is assigned to the corresponding properties of the categoryDetail object.
* connection.Close();: This line closes the database connection, releasing the associated resources.
* return categoryDetail;: After retrieving the category data, the method returns the populated categoryDetail object.

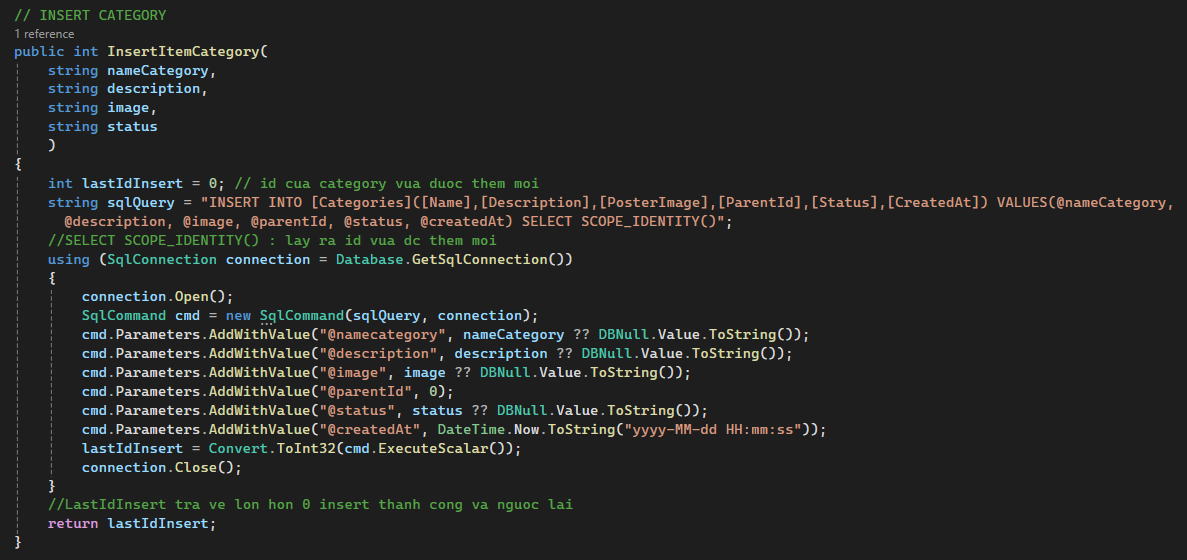
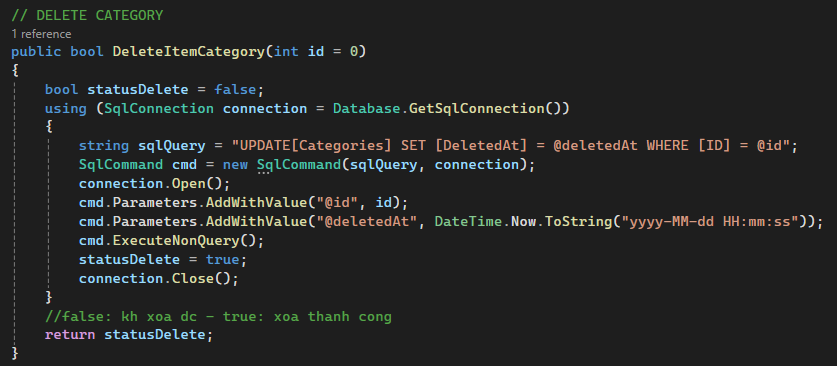
This method InsertItemCategory is responsible for inserting a new category item into the database:

Figure 30: CateoryQuery - Insert

* public int InsertItemCategory(string nameCategory, string description, string image, string status): This is a public method named InsertItemCategory. It takes four string parameters: nameCategory, description, image, and status, representing the attributes of the category to be inserted.
* int lastIdInsert = 0;: This variable will store the ID of the newly inserted category. It's initialized to 0.
* string sqlQuery = "INSERT INTO [Categories]([Name],[Description],[PosterImage],[ParentId],[Status],[CreatedAt]) VALUES(@nameCategory, @description, @image, @parentId, @status, @createdAt) SELECT SCOPE\_IDENTITY()": This SQL query inserts a new record into the Categories table with the provided values and returns the ID of the newly inserted record using SCOPE\_IDENTITY() function.
* using (SqlConnection connection = Database.GetSqlConnection()) { ... }: This line establishes a connection to the database using the Database.GetSqlConnection() method. The using statement ensures that the connection is properly disposed of after its use to avoid resource leaks.
* SQL Command Execution:
  + SqlCommand cmd = new SqlCommand(sqlQuery, connection);: This line creates a new SqlCommand object with the SQL query and the database connection.
  + Parameters are added to the command object to prevent SQL injection attacks.
  + cmd.ExecuteScalar() executes the SQL command and returns the first column of the first row in the result set. In this case, it returns the ID of the newly inserted category.
* lastIdInsert = Convert.ToInt32(cmd.ExecuteScalar());: The returned ID from the SQL query is converted to an integer and assigned to the lastIdInsert variable.
* connection.Close();: This line closes the database connection, releasing the associated resources.
* return lastIdInsert;: The method returns the ID of the newly inserted category. If the insertion was successful, this ID will be greater than 0.

Figure 31: CategoryQuery - Update

This method UpdateCategoryById is responsible for updating an existing category in the database based on the provided category ID:

* public bool UpdateCategoryById(string nameCategory, string description, string posterImage, string status, int id): This is a public method named UpdateCategoryById. It takes five parameters: nameCategory, description, posterImage, status, and id, representing the attributes of the category to be updated and the ID of the category to update.
* bool checkUpdate = false;: This variable will indicate whether the update operation was successful. It's initialized to false.
* string sqlUpdate = "UPDATE [Categories] SET [Name] = @name, [Description] = @description, [PosterImage] = @posterImage, [Status] = @status, [UpdatedAt] = @updatedAt WHERE [ID] = @id AND [DeletedAt] IS NULL";: This SQL query updates the Categories table with the provided values for the specified category ID. It updates the Name, Description, PosterImage, Status, and UpdatedAt columns.
* using (SqlConnection connection = Database.GetSqlConnection()) { ... }: This line establishes a connection to the database using the Database.GetSqlConnection() method. The using statement ensures that the connection is properly disposed of after its use to avoid resource leaks.
* SQL Command Execution:
  + SqlCommand cmd = new SqlCommand(sqlUpdate, connection);: This line creates a new SqlCommand object with the SQL update query and the database connection.
  + Parameters are added to the command object to prevent SQL injection attacks.
  + cmd.ExecuteNonQuery(); executes the SQL command and returns the number of rows affected by the update operation. Since this method is for updating a single category, it's not necessary to return any specific value.
* connection.Close();: This line closes the database connection, releasing the associated resources.
* checkUpdate = true;: If the execution reaches this point without throwing an exception, it means the update operation was successful, so checkUpdate is set to true.
* return checkUpdate;: The method returns true if the update operation was successful, otherwise, it returns false.

This method DeleteItemCategory is responsible for soft-deleting a category from the database based on the provided category ID:

Figure 32: CategoryQuery - Delete

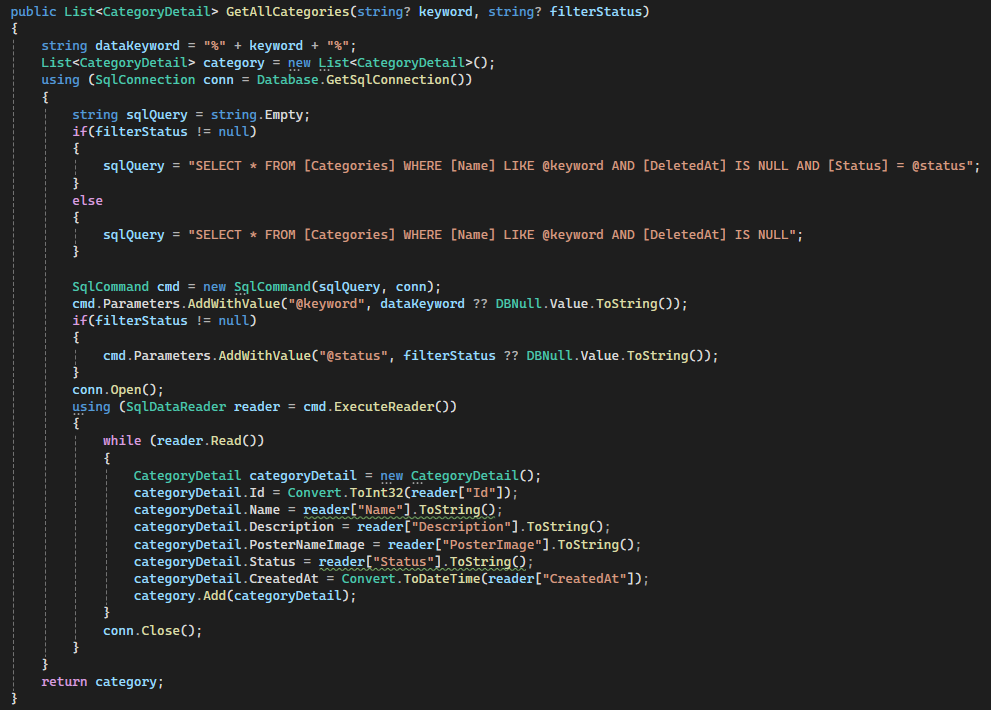
* public bool DeleteItemCategory(int id = 0): This is a public method named DeleteItemCategory. It takes one parameter id, representing the ID of the category to be deleted. The parameter has a default value of 0, meaning if no ID is provided, the method will attempt to delete the category with ID 0.
* bool statusDelete = false;: This variable will indicate whether the deletion operation was successful. It's initialized to false.
* string sqlQuery = "UPDATE [Categories] SET [DeletedAt] = @deletedAt WHERE [ID] = @id";: This SQL query soft-deletes a record from the Categories table by setting the DeletedAt column to the current date and time where the ID matches the provided id.
* using (SqlConnection connection = Database.GetSqlConnection()) { ... }: This line establishes a connection to the database using the Database.GetSqlConnection() method. The using statement ensures that the connection is properly disposed of after its use to avoid resource leaks.
* SQL Command Execution:
  + SqlCommand cmd = new SqlCommand(sqlQuery, connection);: This line creates a new SqlCommand object with the SQL update query and the database connection.
  + Parameters are added to the command object to prevent SQL injection attacks.
  + cmd.ExecuteNonQuery(); executes the SQL command and returns the number of rows affected by the update operation. Since this method is for soft-deleting a single category, it's not necessary to return any specific value.
* connection.Close();: This line closes the database connection, releasing the associated resources.
* statusDelete = true;: If the execution reaches this point without throwing an exception, it means the deletion operation was successful, so statusDelete is set to true.
* return statusDelete;: The method returns true if the deletion operation was successful, otherwise, it returns false.

Figure 33: CategoryQuery - GetAllCategories

This method GetAllCategories is responsible for retrieving a list of categories from the database based on optional search criteria such as keyword and status filter:

* public List<CategoryDetail> GetAllCategories(string? keyword, string? filterStatus): This is a public method named GetAllCategories. It takes two optional parameters: keyword for search keyword and filterStatus for filtering categories by status.
* string dataKeyword = "%" + keyword + "%";: This line initializes a string variable dataKeyword with the search keyword wrapped in wildcard characters %. This allows searching for partial matches in category names.
* List<CategoryDetail> category = new List<CategoryDetail>();: This line initializes an empty list of CategoryDetail objects to store the retrieved category data.
* using (SqlConnection conn = Database.GetSqlConnection()) { ... }: This line establishes a connection to the database using the Database.GetSqlConnection() method. The using statement ensures that the connection is properly disposed of after its use to avoid resource leaks.
* SQL Query Construction:
  + The SQL query is constructed dynamically based on the presence of the filterStatus parameter. If filterStatus is provided, categories are filtered by both name and status. Otherwise, only name-based filtering is applied.
  + sqlQuery variable is assigned the constructed SQL query accordingly.
* SQL Command Execution:
  + SqlCommand cmd = new SqlCommand(sqlQuery, conn);: This line creates a new SqlCommand object with the constructed SQL query and the database connection.
  + Parameters are added to the command object to prevent SQL injection attacks.
  + cmd.ExecuteReader() executes the SQL command and returns a SqlDataReader object containing the results of the query.
* Data Retrieval:
  + Inside the while (reader.Read()) { ... } loop, the method reads each row of data returned by the query.
  + A new CategoryDetail object is created for each row, and the data from the reader is assigned to its properties.
  + The CategoryDetail object is then added to the category list.
* conn.Close();: This line closes the database connection, releasing the associated resources.
* return category;: The method returns the list of CategoryDetail objects containing the retrieved category data.

##### CategoryController

Figure 34: CategoryController - Index

* public IActionResult Index(string SearchString, string Status): This is a public action method named Index. It takes two parameters: SearchString for the search keyword and Status for the category status filter.
* CategoryViewModel categoryViewModel = new CategoryViewModel();: This line initializes a new instance of the CategoryViewModel class, which is a view model used to pass data to the view.
* categoryViewModel.CategoryDetailList = new List<CategoryDetail>();: This line initializes an empty list of CategoryDetail objects within the CategoryViewModel. This list will store the category data retrieved from the database.
* var dataCategory = new CategoryQuery().GetAllCategories(SearchString, Status);: This line executes a query to retrieve category data from the database using the GetAllCategories method of the CategoryQuery class. It passes the search keyword (SearchString) and status filter (Status) as parameters to the method.
* View Data Setting:
  + ViewData["keyword"] = SearchString;: This line sets the value of the keyword ViewData to the SearchString. This ViewData is used to pass data to the view.
  + ViewBag.Status = Status;: This line sets the value of the Status property in the ViewBag. The ViewBag is another way to pass data to the view.
* return View(categoryViewModel);: This line returns a view named Index, passing the categoryViewModel object to the view. The view will render HTML based on the data provided in the categoryViewModel.

Figure 35: CategoryController - Add

HTTP GET Action - Add:

* [HttpGet]: This attribute indicates that the action method responds to HTTP GET requests.
* public IActionResult Add(): This is the action method responsible for rendering the form to add a new category.
* Inside the method:
* CategoryDetail model = new CategoryDetail();: This line initializes a new CategoryDetail object, which represents the model for the form.
* return View(model);: This line returns a view named "Add" with the newly created CategoryDetail object as the model. This view will render the HTML form for adding a new category.

HTTP POST Action - Add:

* [HttpPost]: This attribute indicates that the action method responds to HTTP POST requests.
* [ValidateAntiForgeryToken]: This attribute is used to prevent cross-site request forgery (CSRF) attacks.
* public async Task<IActionResult> Add(CategoryDetail category, IFormFile PosterImage): This is the action method responsible for processing the form submission to add a new category.
* Parameters:
  + category: This parameter represents the category data submitted through the form.
  + PosterImage: This parameter represents the image file uploaded through the form.
* Inside the method:
  + if (ModelState.IsValid) { ... }: This condition checks if the model state is valid, i.e., if the submitted data passes all validation rules defined in the model.
* If the model state is valid:
  + string filePosterImage = UploadFileHelper.UploadFile(PosterImage);: This line uploads the image file using a helper method (UploadFile) and assigns the returned file path to the filePosterImage variable.
  + int idInsertCate = new CategoryQuery().InsertItemCategory(category.Name, category.Description, filePosterImage, category.Status);: This line inserts the category into the database using the InsertItemCategory method of the CategoryQuery class. It passes the category details and the file path of the uploaded image.
* Depending on whether the insertion was successful:
  + If successful (idInsertCate > 0), it sets TempData["saveStatus"] to true.
  + If unsuccessful, it sets TempData["saveStatus"] to false.
  + return RedirectToAction(nameof(CategoryController.Index), "Category");: This line redirects the user to the index action of the CategoryController after adding the category.
* If the model state is not valid, i.e., there are validation errors:
  + It returns the view named "Add" with the same category object. This will display the form again with validation error messages.

HTTP GET Action - Edit:

Figure 36: CategoryController - Edit

* [HttpGet]: This attribute indicates that the action method responds to HTTP GET requests.
* public IActionResult Edit(int id = 0): This is the action method responsible for rendering the form to edit a category.
* Parameters:
  + id: This parameter represents the ID of the category to be edited. It has a default value of 0.
* Inside the method:
  + CategoryDetail categoryDetail = new CategoryQuery().GetDataCategoryById(id);: This line retrieves the category details from the database using the GetDataCategoryById method of the CategoryQuery class, passing the provided category ID.
  + return View(categoryDetail);: This line returns a view named "Edit" with the retrieved CategoryDetail object as the model. This view will render the HTML form for editing the category.

HTTP POST Action - Edit:

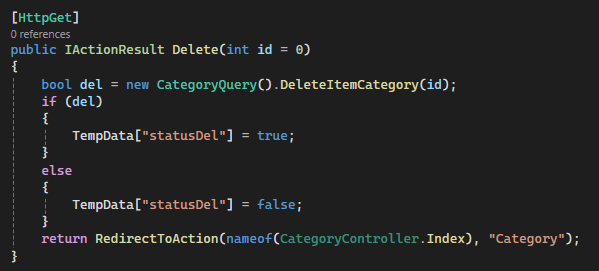
* [HttpPost]: This attribute indicates that the action method responds to HTTP POST requests.
* public IActionResult Edit(CategoryDetail categoryDetail, IFormFile PosterImage): This is the action method responsible for processing the form submission to edit a category.
* Parameters:
  + categoryDetail: This parameter represents the category data submitted through the form.
  + PosterImage: This parameter represents the new image file uploaded through the form.
* Inside the method:
  + var detail = new CategoryQuery().GetDataCategoryById(categoryDetail.Id);: This line retrieves the category details again from the database to obtain the current image file path associated with the category.
  + string uniquePosterImage = detail.PosterNameImage;: This line initializes the uniquePosterImage variable with the current image file path. It will be used to update the category with the existing image path unless a new image is uploaded.
* If a new image is uploaded (categoryDetail.PosterNameImage != null):
  + uniquePosterImage = UploadFileHelper.UploadFile(PosterImage);: This line uploads the new image file using a helper method (UploadFile) and assigns the returned file path to the uniquePosterImage variable.
  + bool update = new CategoryQuery().UpdateCategoryById(categoryDetail.Name, categoryDetail.Description, uniquePosterImage, categoryDetail.Status, categoryDetail.Id);: This line updates the category in the database using the UpdateCategoryById method of the CategoryQuery class, passing the updated category details and the category ID.
* Depending on whether the update was successful:
  + If successful (update is true), it sets TempData["updateStatus"] to true.
  + If unsuccessful, it sets TempData["updateStatus"] to false.
* return RedirectToAction(nameof(CategoryController.Index), "Category");: This line redirects the user to the index action of the CategoryController after editing the category.
* If an exception occurs during the process, it returns the view named "Edit" with the same category object to display any error messages.
* [HttpGet]: This attribute indicates that the action method responds to HTTP GET requests.

Figure 37: CategoryController - Delete

* public IActionResult Delete(int id = 0): This is a public action method named Delete. It takes one optional parameter id, representing the ID of the category to be deleted. The parameter has a default value of 0.
* bool del = new CategoryQuery().DeleteItemCategory(id);: This line invokes the DeleteItemCategory method of the CategoryQuery class to delete the category from the database based on the provided category ID.
* The result of the deletion operation is stored in the del variable. It will be true if the deletion was successful and false otherwise.
* Depending on whether the deletion was successful:
  + If del is true, indicating a successful deletion, TempData["statusDel"] is set to true.
  + If del is false, indicating an unsuccessful deletion, TempData["statusDel"] is set to false.
* return RedirectToAction(nameof(CategoryController.Index), "Category");: This line redirects the user to the index action of the CategoryController after deleting the category.

##### Category view

Figure 38: Category view - Add

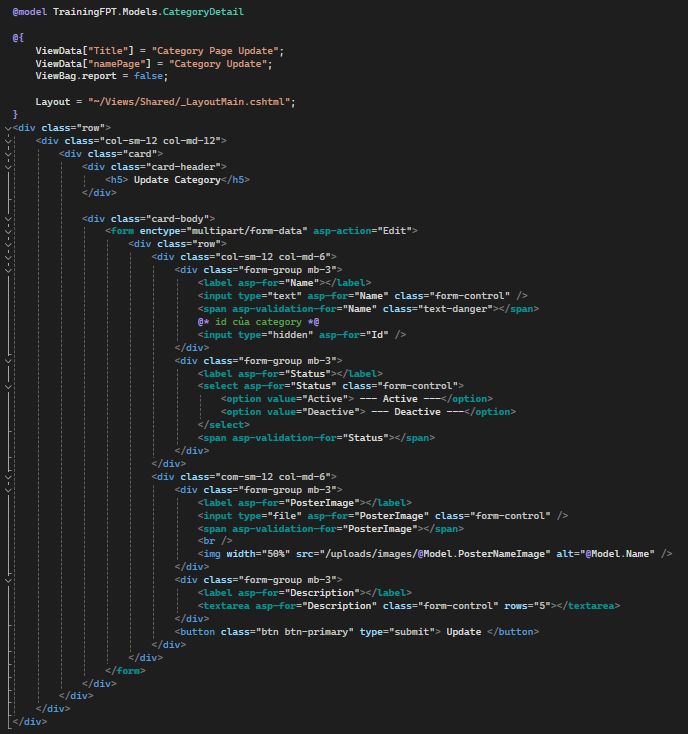


Figure 39: Category view - Edit

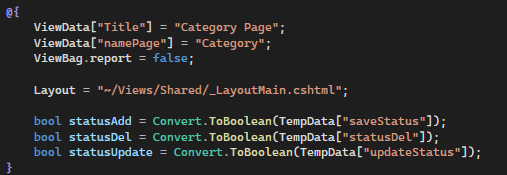
View Data and ViewBag:

Figure 40: Category view - Index 1

* ViewData["Title"] = "Category Page";: Sets the title of the page to "Category Page" using ViewData.
* ViewData["namePage"] = "Category";: Sets the name of the page to "Category" using ViewData.
* ViewBag.report = false;: Sets the value of the ViewBag property named "report" to false.

Layout:

* Layout = "~/Views/Shared/\_LayoutMain.cshtml";: Specifies the layout file (\_LayoutMain.cshtml) to be used for this view. The layout file typically contains the common HTML structure shared across multiple views.

Temporary Data Retrieval:

* bool statusAdd = Convert.ToBoolean(TempData["saveStatus"]);: Retrieves the value of the saveStatus TempData and converts it to a boolean variable named statusAdd.
* bool statusDel = Convert.ToBoolean(TempData["statusDel"]);: Retrieves the value of the statusDel TempData and converts it to a boolean variable named statusDel.
* bool statusUpdate = Convert.ToBoolean(TempData["updateStatus"]);: Retrieves the value of the updateStatus TempData and converts it to a boolean variable named statusUpdate.



Figure 41: Category view - Index 2

<a class="btn btn-primary" href="@Url.Action("Add","Category")"> Add Category + </a>: This line creates a button with a link to the "Add" action of the "Category" controller. It allows users to navigate to the page for adding a new category.

Search and Filter Form:

* <form asp-action="Index" method="get" class="my-3">: This line starts a form element that submits to the "Index" action of the current controller using the GET method.
* Inside the form:
  + Find by name: <input name="SearchString" value="@ViewData["keyword"]" />: This line creates a text input field named "SearchString" for users to enter a search term. The value of the input field is populated with the value stored in the ViewData["keyword"].
  + Filter by:: This text indicates the purpose of the dropdown list that follows.
  + <select name="Status"> ... </select>: This dropdown list allows users to filter categories by their status (e.g., Active or Deactive).
  + Options in the dropdown are generated dynamically based on the value stored in the ViewBag.Status. The selected option is determined by comparing ViewBag.Status with each option's value.
  + <button type="submit" class="btn btn-primary btn-sm"> Search</button>: This button submits the form, triggering a search based on the entered criteria.
  + <a asp-action="Index">Back to lists</a>: This link allows users to reset the search and filter criteria and return to the full list of categories.

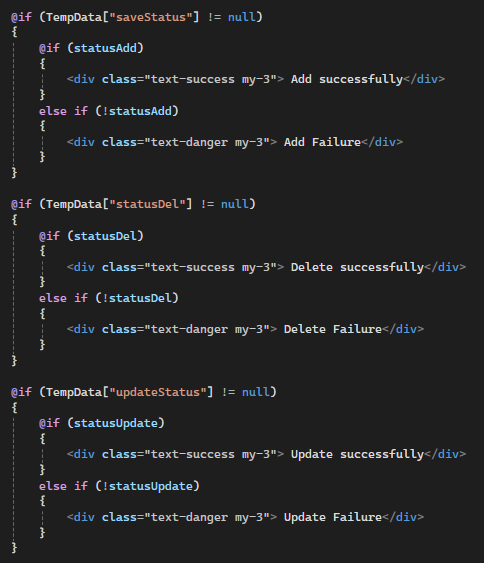


Figure 42: Category view - Index 3

Check TempData for Status Messages:

* @if (TempData["saveStatus"] != null): Checks if the TempData key "saveStatus" is not null, indicating that an add operation was performed.
* @if (TempData["statusDel"] != null): Checks if the TempData key "statusDel" is not null, indicating that a delete operation was performed.
* @if (TempData["updateStatus"] != null): Checks if the TempData key "updateStatus" is not null, indicating that an update operation was performed.

Display Success or Failure Messages:

* If the corresponding TempData value is not null, it checks the boolean variable associated with the respective operation (statusAdd, statusDel, statusUpdate).
* If the boolean variable is true, it displays a success message.
* If the boolean variable is false, it displays a failure message.

Message Display:

* <div class="text-success my-3"> Add successfully</div>: This line displays a success message in green text if the add operation was successful.
* <div class="text-danger my-3"> Add Failure</div>: This line displays a failure message in red text if the add operation failed.
* Similar blocks exist for delete and update operations.

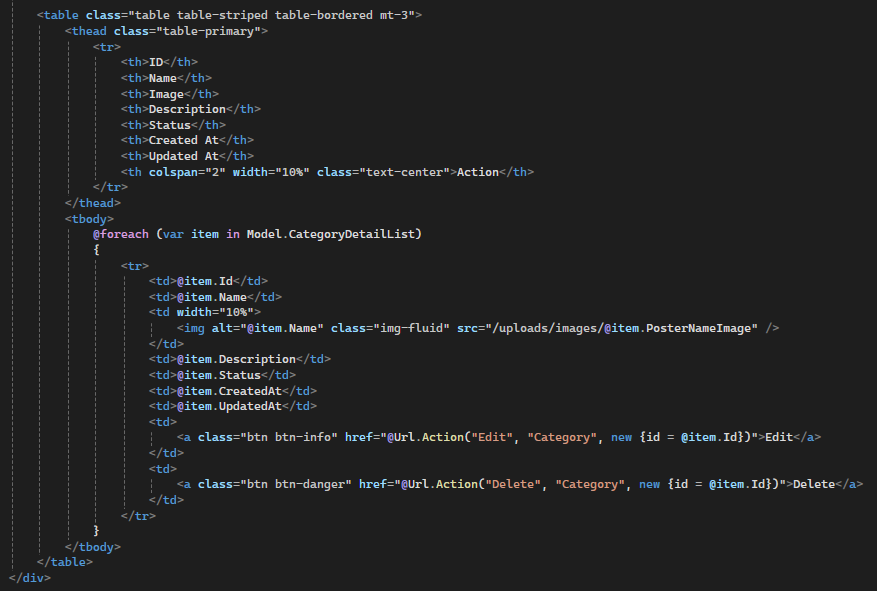


Figure 43: Category view - Index 4

Table Structure:

* <table class="table table-striped table-bordered mt-3">: Defines a table element with the Bootstrap classes for styling, including striped rows and bordered borders. It also has a top margin of 3 units.

Table Header (thead):

* <thead class="table-primary">: Defines the table header row with a blue background color.
* Inside the header row, there are columns representing various attributes of a category:
  + ID: Represents the unique identifier of the category.
  + Name: Represents the name of the category.
  + Image: Represents the image associated with the category.
  + Description: Represents a brief description of the category.
  + Status: Represents the current status of the category.
  + Created At: Represents the date and time when the category was created.
  + Updated At: Represents the date and time when the category was last updated.
  + Action: Represents actions that can be performed on the category (Edit and Delete).

Table Body (tbody):

* <tbody>: Defines the table body where category data will be displayed.
* Inside the body, a foreach loop iterates over each item in the Model.CategoryDetailList, which contains the list of categories to display.
* For each category item, a table row (tr) is generated with corresponding table data cells (td) for each attribute (ID, Name, Image, Description, Status, Created At, Updated At).
* The img element within the "Image" column displays the image associated with the category. The image source is dynamically generated based on the PosterNameImage attribute of the category item.

Action Links:

* Edit: It redirects the user to the edit page of the selected category, passing the category ID as a parameter in the URL.
* Delete: It redirects the user to the delete action of the category, passing the category ID as a parameter in the URL.
  1. **Course management**

##### CoursesViewModel

Figure 44: CoursesViewModel

CoursesViewModel:

* This class represents the view model for courses. It contains a property CourseDetailList which is a list of CourseDetail objects.
* CourseDetailList: Represents a list of detailed course information.

CourseDetail:

* This class represents the details of a course.
* CourseId: Represents the unique identifier of the course. It's decorated with the [Key] attribute indicating it as the primary key.
* CategoryId: Represents the category to which the course belongs. It's required and must be chosen from existing categories.
* NameCourse: Represents the name of the course. It's required.
* Description: Represents a description of the course. It's optional.
* StartDate: Represents the start date of the course. It's required and must be in the specified date format.
* EndDate: Represents the end date of the course. It's optional.
* Image: Represents the image associated with the course. It's required, and file extensions and size are restricted. It's of type IFormFile, indicating it's intended to be used in file uploads.
* ViewImageCourse: Represents the path or URL of the image of the course. It's optional.
* LikeCourse: Represents the number of likes for the course. It's optional.
* StarCourse: Represents the rating of the course in stars. It's optional.
* Status: Represents the status of the course. It's required and must be chosen.
* viewCategoryName: Represents the name of the category associated with the course. It's optional.
* CreatedAt: Represents the date and time when the course was created. It's optional.
* UpdatedAt: Represents the date and time when the course was last updated. It's optional.
* DeletedAt: Represents the date and time when the course was deleted. It's optional.

##### CourseQuery

Figure 45: CourseQuery - GetDataCourseById

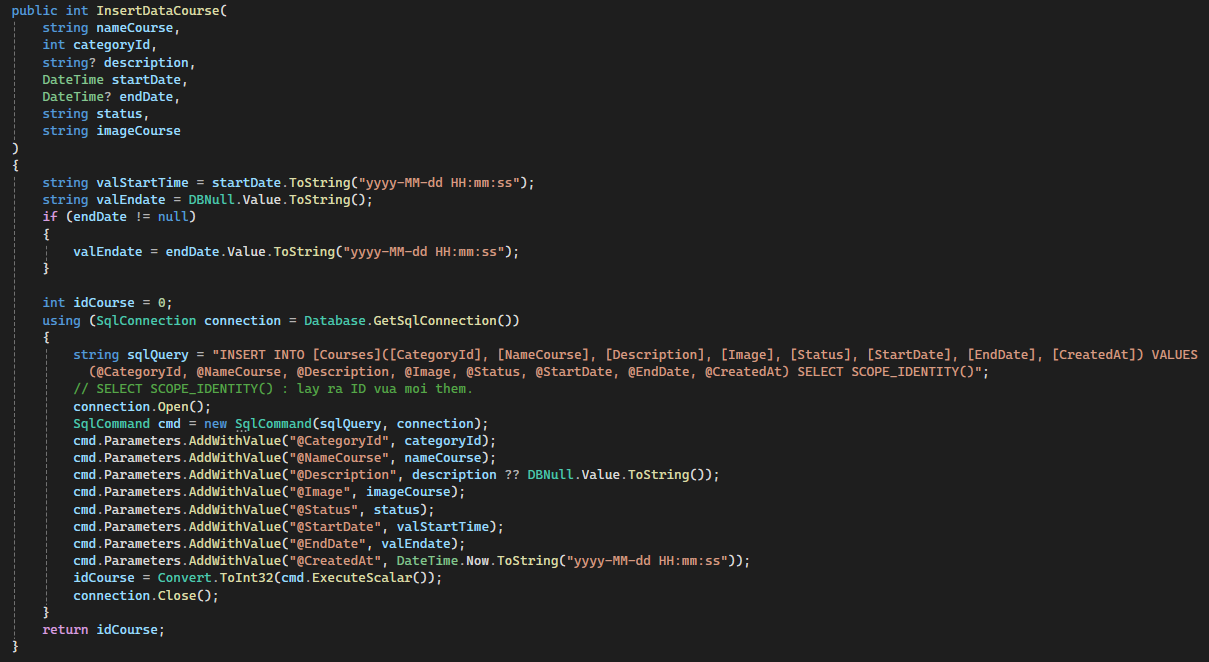
* public CourseDetail GetDataCouseById(int id = 0): This method is declared as public and returns a CourseDetail object. It takes an optional parameter id, representing the ID of the course to fetch. If no ID is provided, it defaults to 0.
* The method establishes a connection to the database using SqlConnection and the Database.GetSqlConnection() method.
* Inside a using block, it ensures proper disposal of resources by automatically closing the connection when done.
* It constructs an SQL query string to select all columns from the "Courses" table where the ID matches the provided id parameter and the course is not marked as deleted ([DeletedAt] IS NULL).
* The SqlCommand object is initialized with the SQL query and the connection.
* The @id parameter is added to the command to prevent SQL injection attacks.
* The ExecuteReader() method is called to execute the SQL command and retrieve data from the database.
* Inside a while loop, it iterates over the result set returned by the query.
* For each row in the result set, it populates the properties of the courseDetail object with data retrieved from the database columns.
* It converts data types as needed, such as converting Id and CategoryId to integers and StartDate and EndDate to DateTime objects.
* After retrieving the necessary data, the database connection is closed.
* The method returns the courseDetail object populated with course information fetched from the database.

Figure 46: CourseQuery - Add

Method Signature:

* Access Modifier: public
* Return Type: int
* Name: InsertDataCourse
* Parameters:
  + nameCourse (string): Name of the course.
  + categoryId (int): ID of the category to which the course belongs.
  + description (string?): Description of the course. It's nullable.
  + startDate (DateTime): Start date of the course.
  + endDate (DateTime?): End date of the course. It's nullable.
  + status (string): Status of the course.
  + imageCourse (string): Image URL or path of the course.

Variable Initialization:

* valStartTime: Converts the startDate parameter to a string in the format "yyyy-MM-dd HH:mm:ss".
* valEndate: Initializes as a string representation of DBNull.Value. If endDate is not null, it is updated with the string representation of endDate in the same format.

Database Interaction:

* Opens a SQL connection using Database.GetSqlConnection() method.
* Constructs an SQL query string to insert data into the Courses table, including parameters for category ID, course name, description, image URL, status, start date, end date, and creation timestamp.
* Adds parameters to the SqlCommand object cmd with values from the method parameters.
* Executes the SQL command using ExecuteScalar() method which returns the newly inserted course's ID using SCOPE\_IDENTITY().
* Closes the database connection.

Return Value:

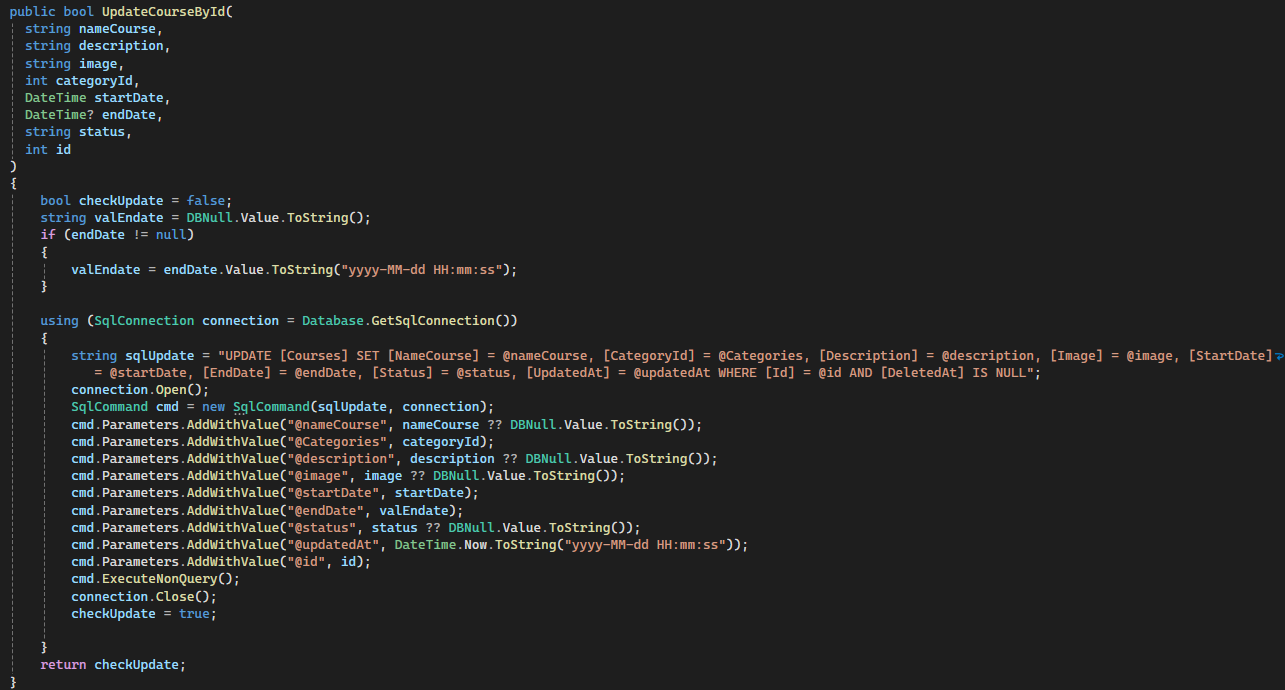
* Returns the ID of the newly inserted course.

Figure 47: CourseQuery - Update

Method Signature:

* Access Modifier: public
* Return Type: bool
* Name: UpdateCourseById
* Parameters:
  + nameCourse (string): Updated name of the course.
  + description (string): Updated description of the course.
  + image (string): Updated image URL or path of the course.
  + categoryId (int): Updated ID of the category to which the course belongs.
  + startDate (DateTime): Updated start date of the course.
  + endDate (DateTime?): Updated end date of the course. It's nullable.
  + status (string): Updated status of the course.
  + id (int): ID of the course to be updated.

Variable Initialization:

* valEndate: Initializes as a string representation of DBNull.Value. If endDate is not null, it is updated with the string representation of endDate in the format "yyyy-MM-dd HH:mm:ss".

Database Interaction:

* Opens a SQL connection using Database.GetSqlConnection() method.
* Constructs an SQL update query string to update the Courses table, setting the provided parameters (nameCourse, categoryId, description, image, startDate, endDate, status, updatedAt) where Id matches the provided id and DeletedAt is NULL.
* Adds parameters to the SqlCommand object cmd with values from the method parameters.
* Executes the SQL command using ExecuteNonQuery() method which executes the query without returning any data.
* Closes the database connection.

Return Value:

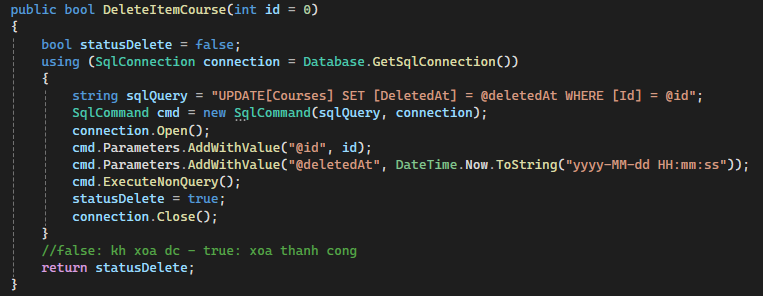
* Returns true if the update was successful, otherwise false.
* Access Modifier: public

Figure 48: CourseQuery - Delete

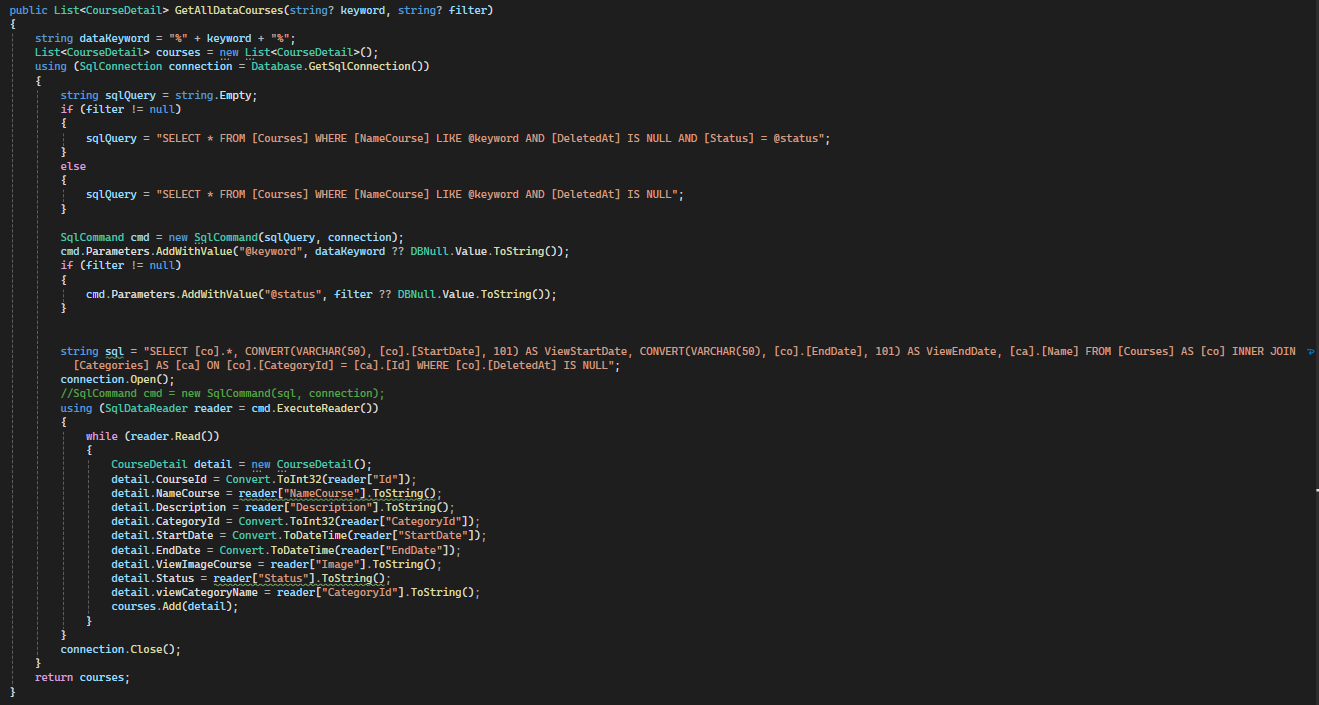
* Return Type: bool
* Name: DeleteItemCourse
* id (int): The ID of the course to be deleted. It has a default value of 0, implying that if no ID is provided, it defaults to 0.
* Database Interaction:
  + Opens a SQL connection using Database.GetSqlConnection() method.
  + Constructs an SQL update query string to set the DeletedAt column to the current timestamp for the course with the specified id.
  + Adds parameters to the SqlCommand object cmd for id and deletedAt (current timestamp).
  + Executes the SQL command using ExecuteNonQuery() method, which executes the query without returning any data.
  + Closes the database connection.
* Returns true if the deletion operation was successful, otherwise false.
* Access Modifier: public

Figure 49: CourseQuery - Search

* Return Type: List<CourseDetail>
* Name: GetAllDataCourses
* keyword (string?): Optional search keyword. It's nullable.
* filter (string?): Optional filter criteria. It's nullable.
* dataKeyword: Constructs a string pattern to be used in SQL LIKE clause by appending wildcard characters (%) to the provided keyword.
* Database Interaction:
  + Opens a SQL connection using Database.GetSqlConnection() method.
  + Constructs an SQL query string based on the presence of a filter:
  + If a filter is provided, it filters courses by name, ensures they are not deleted, and match the specified status.
  + If no filter is provided, it filters courses by name and ensures they are not deleted.
  + Adds parameters to the SqlCommand object cmd for the keyword and filter (if provided).
  + Executes the SQL query using ExecuteReader() method to retrieve data.
  + Reads the data from the result set and populates CourseDetail objects with the retrieved data.
  + Adds populated CourseDetail objects to the courses list.
  + Closes the database connection.
* Returns a list of CourseDetail objects containing course data retrieved from the database.

##### CoursesController

Figure 50: CoursesController - Index

Method Signature:

* Attributes: [HttpGet] indicates that this method handles HTTP GET requests.
* Return Type: IActionResult, which is a generic interface for action results.
* Name: Index
* Parameters:
  + SearchString (string): Represents a search string provided by the user.
  + Status (string): Represents a status filter provided by the user.

Variable Initialization:

* course: Initializes a new instance of the CoursesViewModel class, which presumably contains properties to hold course-related data.
* course.CourseDetailList: Initializes a new empty list to hold CourseDetail objects.

Data Retrieval:

* Calls the GetAllDataCourses method of a CourseQuery object to retrieve course data based on the provided SearchString and Status.
* Iterates over the retrieved data and maps it to CourseDetail objects, adding them to the CourseDetailList property of the course object.

View Data and View Bag:

* Sets ViewData["keyword"] to the value of the SearchString parameter. ViewData is a dictionary used to pass data from the controller to the view.
* Sets ViewBag.Status to the value of the Status parameter. ViewBag is a dynamic property used to pass data from the controller to the view.

View Rendering:

* Returns a view named "Index" along with the course object as the model.

Explanation:

* This method prepares course data to be displayed on the "Index" view. It retrieves course data based on the provided search string and status filter, then passes the data to the view for rendering.
* It also passes additional information such as the search string and status to be used within the view for various purposes like displaying filters or search criteria.



Figure 51: CoursesController - Add 1

[HttpGet] Add():

Figure 52: CoursesController - Add 2

* This method is decorated with [HttpGet] attribute, indicating that it handles HTTP GET requests.
* It initializes a new instance of CourseDetail class to represent the course being added.
* Retrieves all categories from the database using CategoryQuery class and populates a list of SelectListItem objects with category ID and name.
* Passes the list of categories to the view using ViewBag.Categories.
* Returns a view named "Add" along with the empty CourseDetail object.

[HttpPost] Add():

* This method is decorated with [HttpPost] attribute, indicating that it handles HTTP POST requests.
* It also has [ValidateAntiForgeryToken] attribute to prevent CSRF attacks.
* Checks if the model state is valid. If so, it proceeds with the course insertion process.
* Attempts to upload the image file using UploadFileHelper.UploadFile() method, passing the IFormFile object representing the uploaded image.
* Inserts the course data into the database using CourseQuery.InsertDataCourse() method, passing the course details along with the image file path.
* If the insertion is successful (returns a positive idCourse), sets a TempData variable saveStatus to true, otherwise sets it to false.
* Redirects to the Index action of the CoursesController.
* If there's any exception during the process, it catches the exception, returns a message with the exception details as an HTTP response.
* If the model state is not valid, it repeats the process similar to the GET method, retrieving categories and passing them to the view.

Explanation:

* The GET method prepares the view for adding a new course by providing an empty CourseDetail object and a list of categories.
* The POST method handles the form submission, validates the input, uploads the image file, inserts the course data into the database, and redirects to the index page.
* Both methods retrieve categories to populate the dropdown list of categories on the view.
* The POST method handles form submission, performs server-side validation, and database interaction to add a new course.

CSRF Protection:

* The [ValidateAntiForgeryToken] attribute helps protect against CSRF attacks by validating that the anti-forgery token generated in the view matches the one submitted with the request.



Figure 53: CoursesController - Edit

[HttpGet] Edit(int id = 0):

* This method is decorated with [HttpGet] attribute, indicating that it handles HTTP GET requests.
* It retrieves the details of the course with the given id using the CourseQuery.GetDataCouseById() method.
* Retrieves all categories from the database using CategoryQuery.GetAllCategories() method and populates a list of SelectListItem objects with category ID and name.
* Passes the list of categories to the view using ViewBag.Categories.
* Returns a view named "Edit" along with the CourseDetail object retrieved for the given id.

[HttpPost] Edit(CourseDetail courseDetail, IFormFile Image):

* This method is decorated with [HttpPost] attribute, indicating that it handles HTTP POST requests.
* It attempts to retrieve the existing course details based on the provided courseDetail.CourseId using CourseQuery.GetDataCouseById() method.
* Checks if a new image file is uploaded. If so, it uploads the new image using UploadFileHelper.UploadFile() method and assigns the resulting unique image path to uniqueImage.
* Calls the CourseQuery.UpdateCourseById() method to update the course details in the database with the new values provided in courseDetail. If a new image is uploaded, the uniqueImage path is used; otherwise, the existing image path is retained.
* Sets a TempData variable updateStatus based on whether the update operation was successful or not.
* Redirects to the Index action of the CoursesController.

Explanation:

* The GET method prepares the view for editing a specific course by retrieving its details and populating the dropdown list of categories.
* The POST method handles the form submission, validates the input, uploads a new image file if provided, updates the course data in the database, and redirects to the index page.
* Both methods handle exceptions by returning an HTTP response with the exception message.

CSRF Protection:

* There's no explicit [ValidateAntiForgeryToken] attribute in the POST method. It's recommended to include this attribute to protect against CSRF attacks by validating that the anti-forgery token generated in the view matches the one submitted with the request.

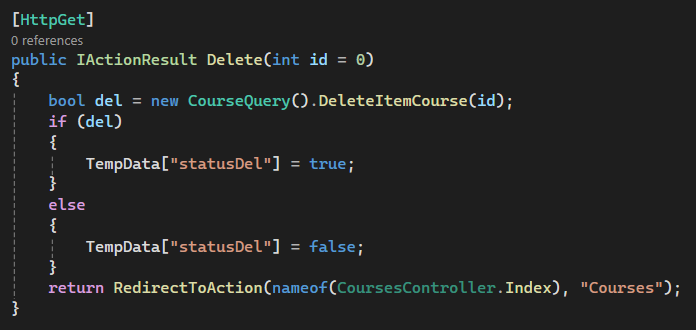
Method Signature:

Figure 54: CoursesController - Delete

* Attributes: [HttpGet] attribute specifies that this method handles HTTP GET requests.
* Return Type: IActionResult, indicating that it returns an action result.
* Name: Delete
* id (int): The ID of the course to be deleted. It has a default value of 0, which means if no ID is provided, it defaults to 0.

Course Deletion:

* It calls the DeleteItemCourse method of the CourseQuery class, passing the id of the course to be deleted.
* The DeleteItemCourse method attempts to delete the course from the database. If the deletion is successful, it returns true; otherwise, it returns false.
* Based on the return value of the DeleteItemCourse method, it sets a TempData variable named statusDel to either true or false to indicate whether the deletion was successful.
* TempData is a temporary storage mechanism provided by ASP.NET MVC or ASP.NET Core to pass data between controller actions and views during the current and subsequent requests.

Redirect:

* After setting the TempData variable, it redirects to the Index action of the CoursesController.
* The Index action typically displays a list of courses, so after deletion, the user is redirected back to the list of courses.

##### Courses view

Figure 55: Courses view - Add 1



Figure 56: Courses view - Add 2

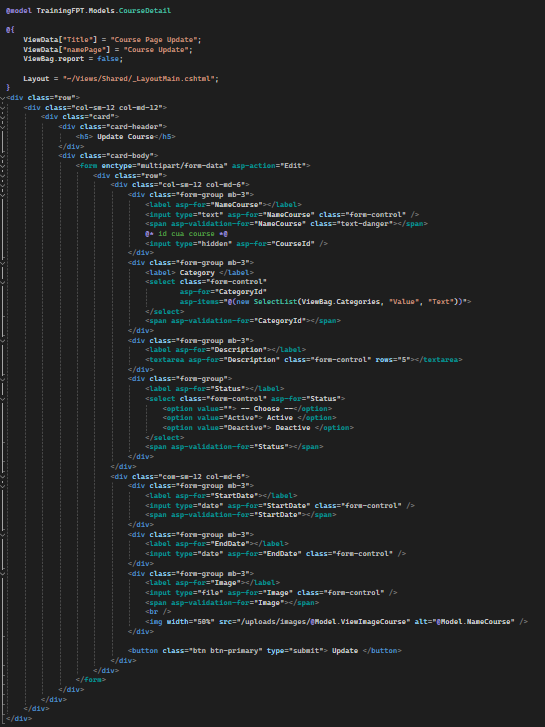


Figure 57: Courses view - Update



Figure 58: Courses view - Index 1

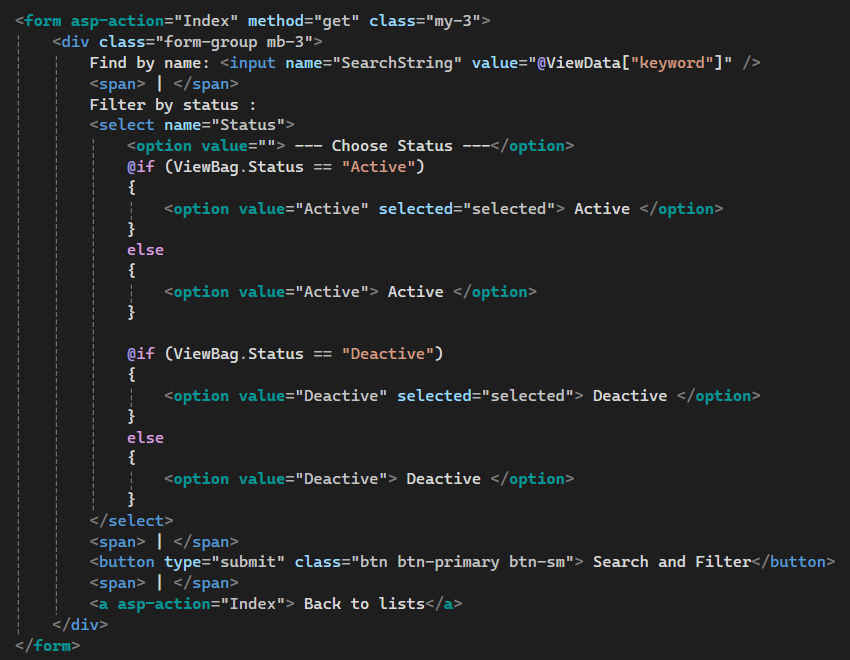


Figure 59: Courses view - Index 2

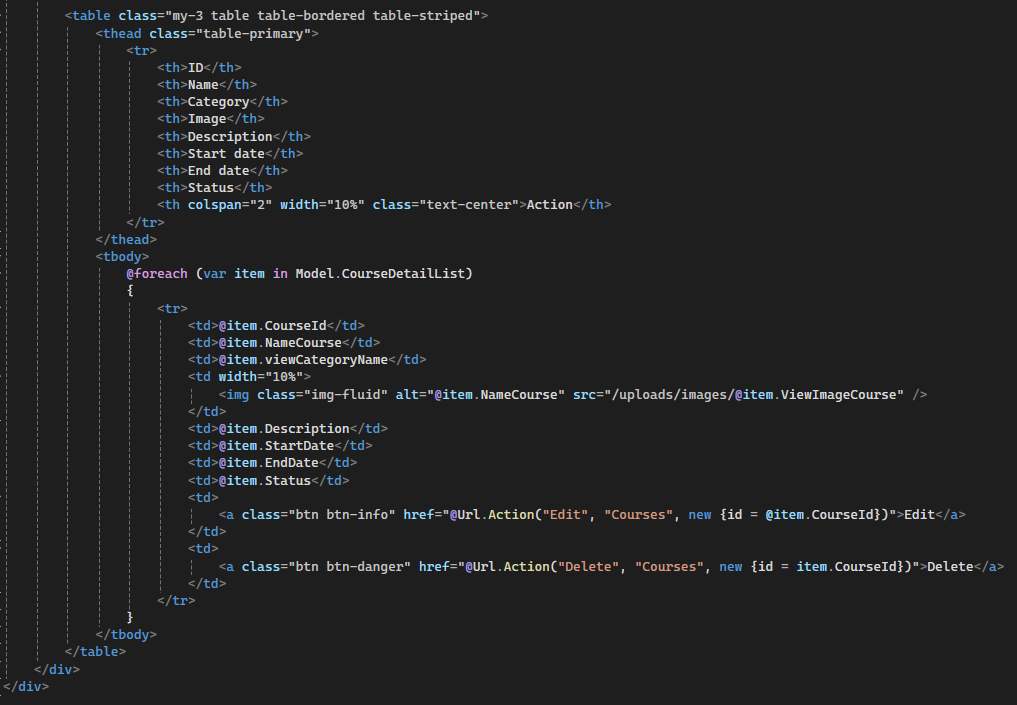


Figure 60: Courses view - Index 3

* 1. **Topic management**

##### TopicsViewModel

Figure 61: TopicsViewModel

TopicsViewModel class:

* This class typically represents the data model used to transfer data between the controller and the view for topics-related functionality.
* It contains a property called TopicDetailList, which is a list of TopicDetail objects.
* This class acts as a container for multiple TopicDetail objects, allowing the view to work with a collection of topics.

TopicDetail class:

* This class represents the details of a topic, likely corresponding to a database table or entity.
* Properties:
  + TopicId: Represents the unique identifier of the topic. It is decorated with the [Key] attribute, indicating it's the primary key.
  + NameTopic: Represents the name of the topic. It is decorated with the [Required] attribute, specifying that it's a required field.
  + CourseId: Represents the ID of the course associated with the topic. It is decorated with the [Required] attribute.
  + Description: Represents the description of the topic. It's nullable.
  + Video: Represents the video file associated with the topic. It's decorated with various validation attributes:
  + [Required]: Specifies that it's a required field.
  + [AllowExtensionFile]: Custom attribute allowing specific file extensions.
  + [AllowMaxSizeFile]: Custom attribute specifying the maximum file size allowed.
  + ViewVideo: Represents the view of the video file.
  + Audio: Represents the audio file associated with the topic. Similar validation attributes as the Video property.
  + ViewAudio: Represents the view of the audio file.
  + DocumentTopic: Represents the document file associated with the topic. Similar validation attributes as the Video property.
  + ViewDocumentTopic: Represents the view of the document file.
  + LikeTopic: Represents the number of likes for the topic. It's nullable.
  + StarTopic: Represents the star rating of the topic. It's nullable.
  + Status: Represents the status of the topic. It is decorated with the [Required] attribute.
  + viewCourseName: Represents the name of the associated course. It's nullable.
  + CreatedAt, UpdatedAt, DeletedAt: Represent the timestamps for creation, last update, and deletion of the topic. They are nullable.

##### TopicQuery

Figure 62: TopicQuery - GetDataTopicById

* Method Signature:
  + Access Modifier: public
  + Return Type: TopicDetail
  + Name: GetDataTopicById
  + id (int): The ID of the topic to retrieve. It has a default value of 0, meaning if no ID is provided, it defaults to 0.
* Initializes a new instance of the TopicDetail class to store the details of the retrieved topic.
* Database Interaction:
  + Opens a SQL connection using Database.GetSqlConnection() method.
  + Constructs an SQL query string to select the topic details based on the provided id and ensuring it's not deleted.
  + Adds a parameter to the SqlCommand object cmd for the id.
  + Executes the SQL query using ExecuteReader() method to retrieve data.
  + Reads the data from the result set and populates the TopicDetail object with the retrieved data.
  + Closes the database connection after reading the data.
* Returns the TopicDetail object containing the details of the topic retrieved from the database. If no topic is found with the provided id, it returns an instance of TopicDetail with default values.
* The SQL query includes a condition to ensure that only non-deleted topics are retrieved ([DeletedAt] IS NULL).
* Data from the SQL result set is mapped to properties of the TopicDetail object.

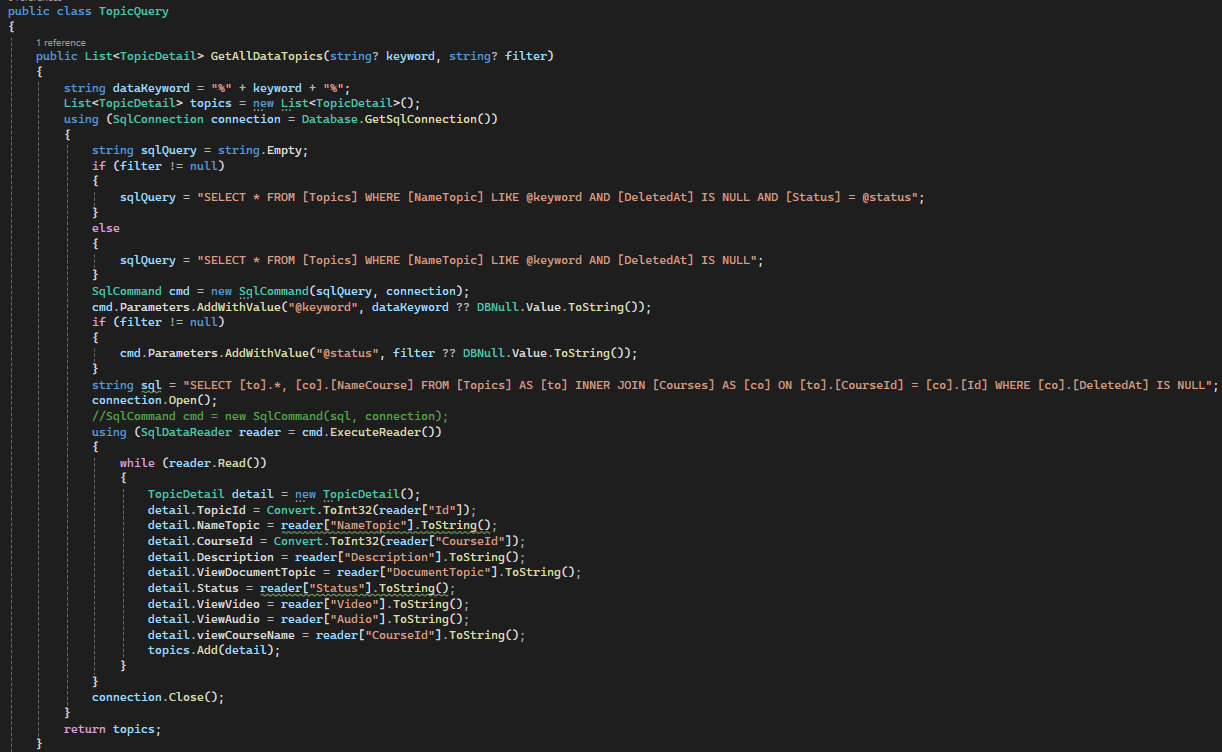


Figure 63: TopicQuery - Search

* Method Signature:
  + Access Modifier: public
  + Return Type: List<TopicDetail>
  + Name: GetAllDataTopics
  + Parameters:
    - keyword (string?): Optional search keyword. It's nullable.
    - filter (string?): Optional filter criteria. It's nullable.
* Variable Initialization:
  + dataKeyword: Constructs a string pattern to be used in SQL LIKE clause by appending wildcard characters (%) to the provided keyword.
  + topics: Initializes an empty list to store TopicDetail objects retrieved from the database.
* Database Interaction:
  + Opens a SQL connection using Database.GetSqlConnection() method.
  + Constructs an SQL query string based on the presence of a filter:
    - If a filter is provided, it filters topics by name, ensures they are not deleted, and match the specified status.
    - If no filter is provided, it filters topics by name and ensures they are not deleted.
  + Adds parameters to the SqlCommand object cmd for the keyword and filter (if provided).
  + Executes the SQL query using ExecuteReader() method to retrieve data.
  + Reads the data from the result set and populates TopicDetail objects with the retrieved data.
  + Adds populated TopicDetail objects to the topics list.
  + Closes the database connection.
* Returns a list of TopicDetail objects containing topic data retrieved from the database.
* The SQL queries include a condition to ensure that only non-deleted topics are retrieved ([DeletedAt] IS NULL).
* The method uses parameterized queries to prevent SQL injection vulnerabilities.
* Data from the SQL result set is mapped to properties of TopicDetail objects.
* DBNull.Value is used for handling nullable database fields. If keyword or filter is null, it's replaced with DBNull.Value.
* The method performs an inner join between the Topics and Courses tables to retrieve additional data about the associated course (CourseId, NameCourse). This allows for displaying course information along with topic details.

##### TopicsController

##### Topics view

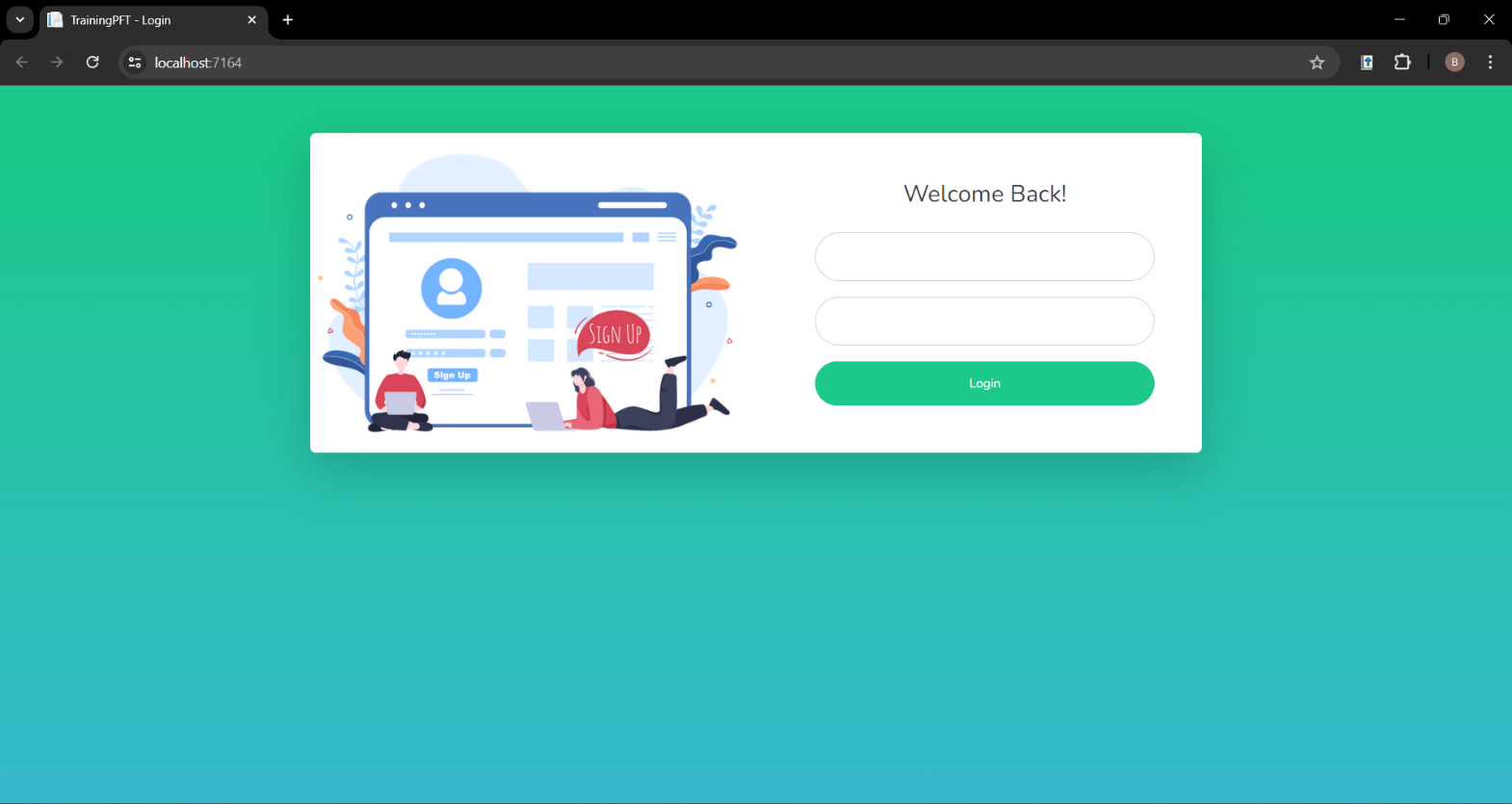
1. **Result**
   1. **Login**

Figure 64: Login

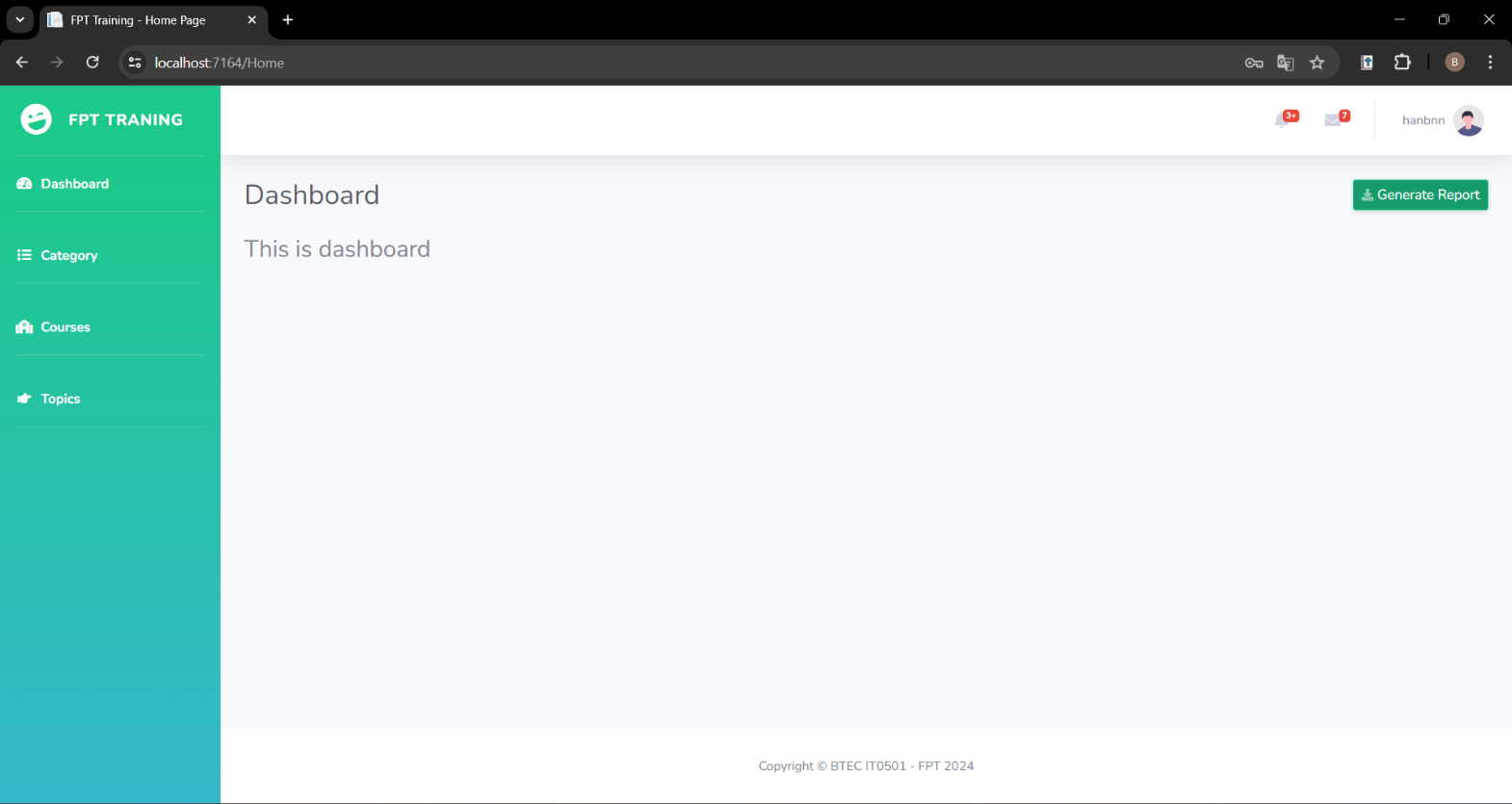
* 1. **Dashboard**

Figure 65: Dashboard

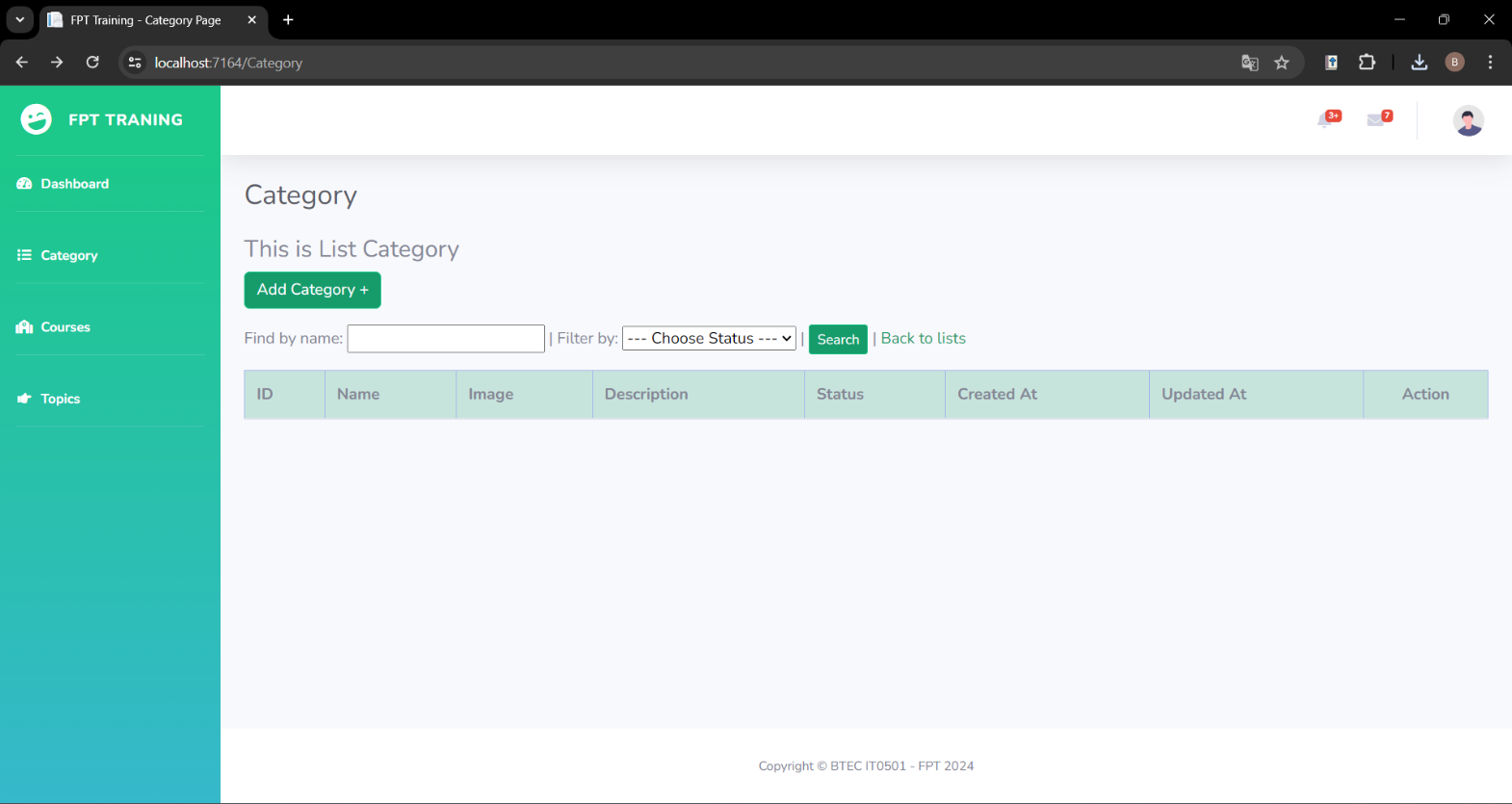
* 1. **Category management**
* Index:

Figure 66: Category index

* Add:

Figure 67: Category add

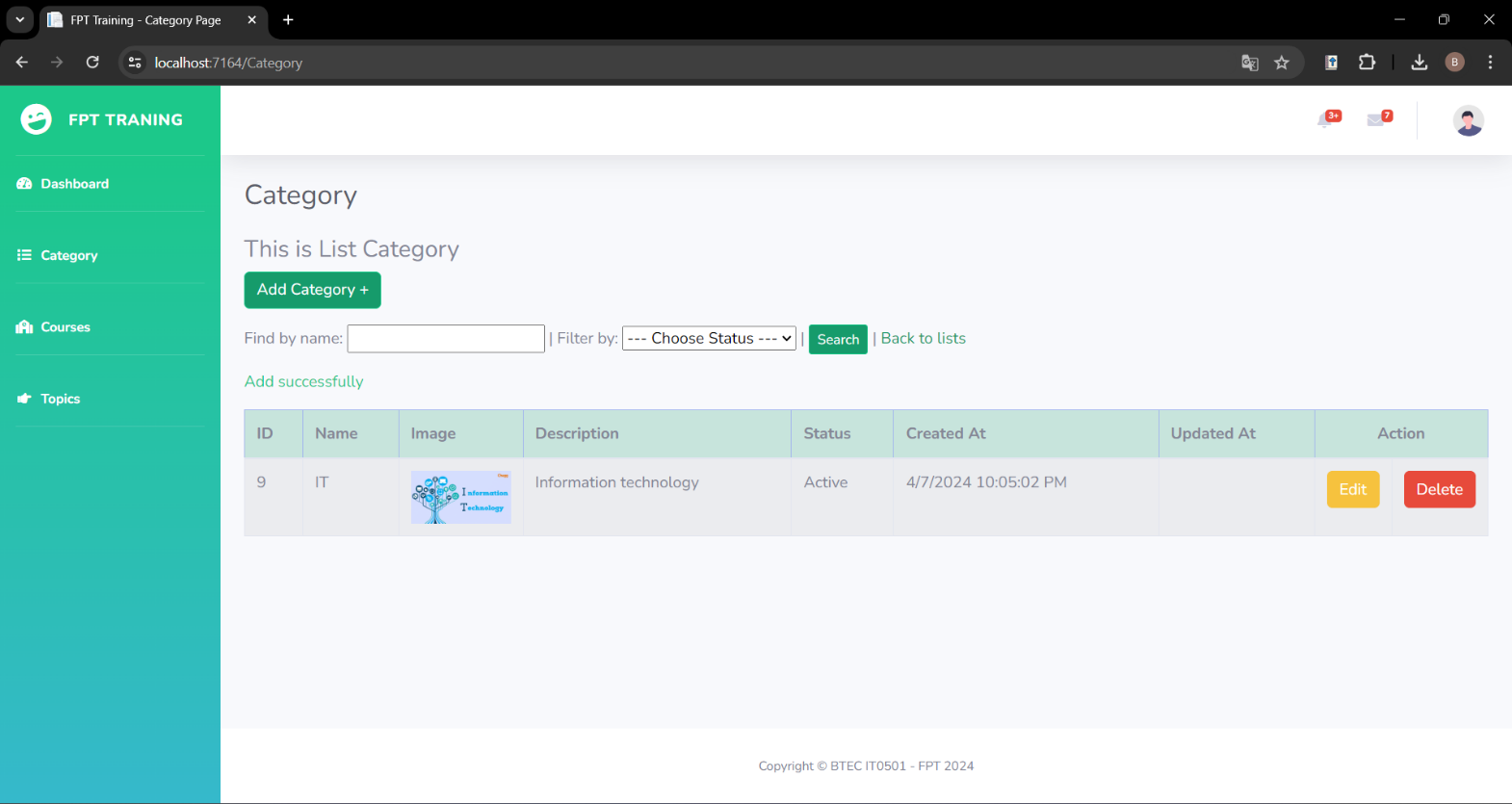
* Interface after adding:

Figure 68: Category interface after adding

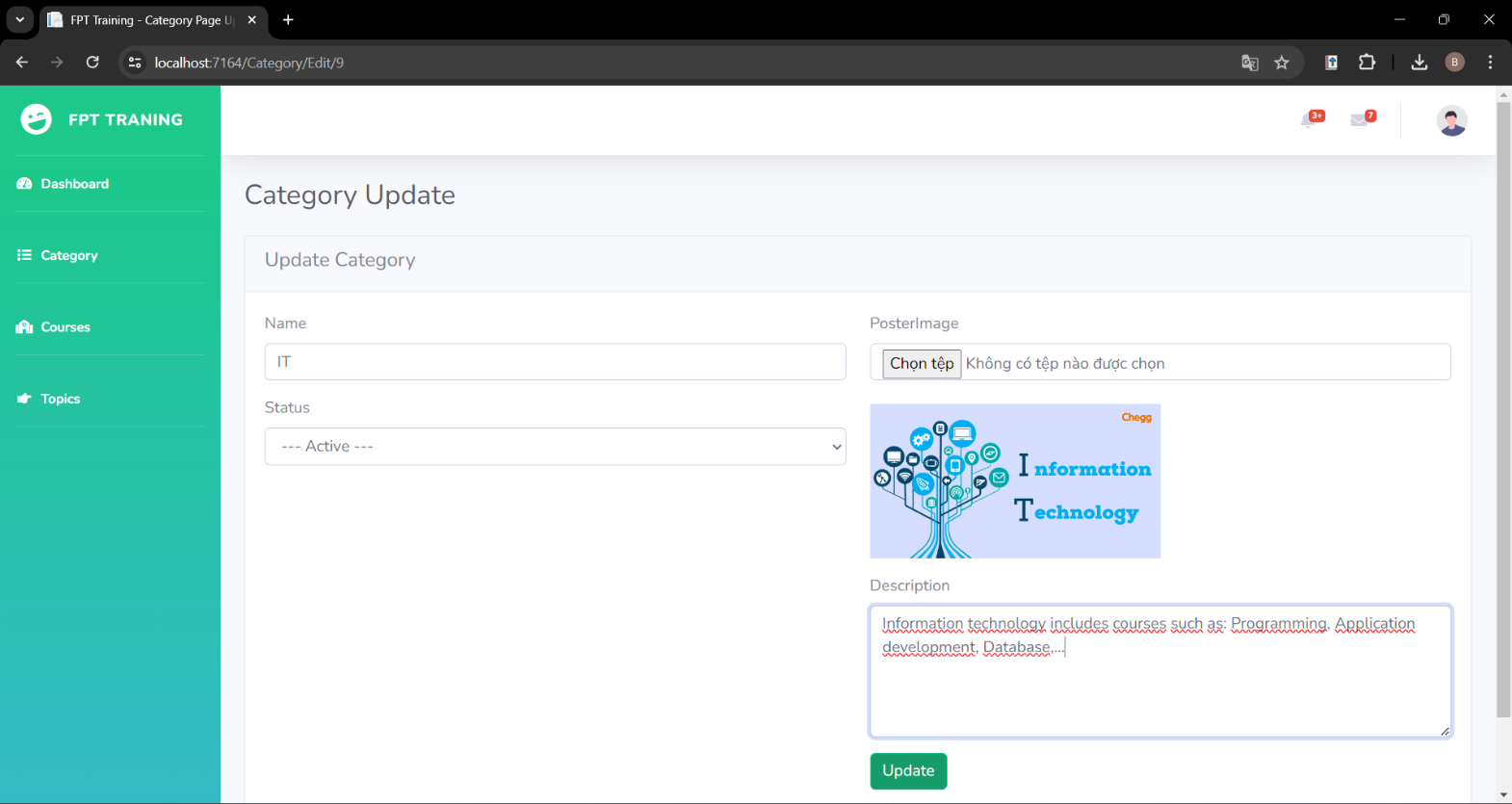
* Update:

Figure 69: Category update

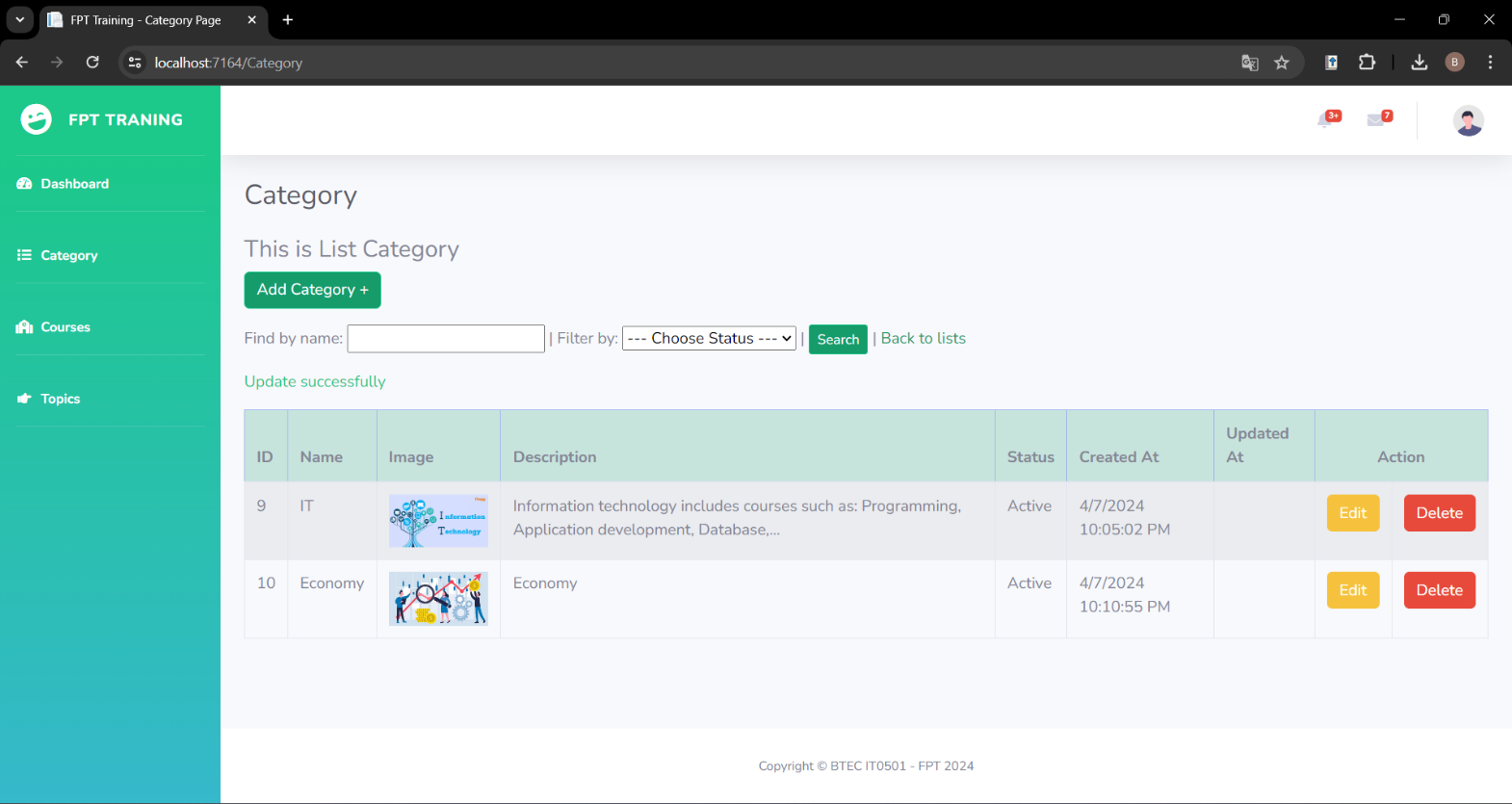
* Interface after update:

Figure 70: Category interface after updating

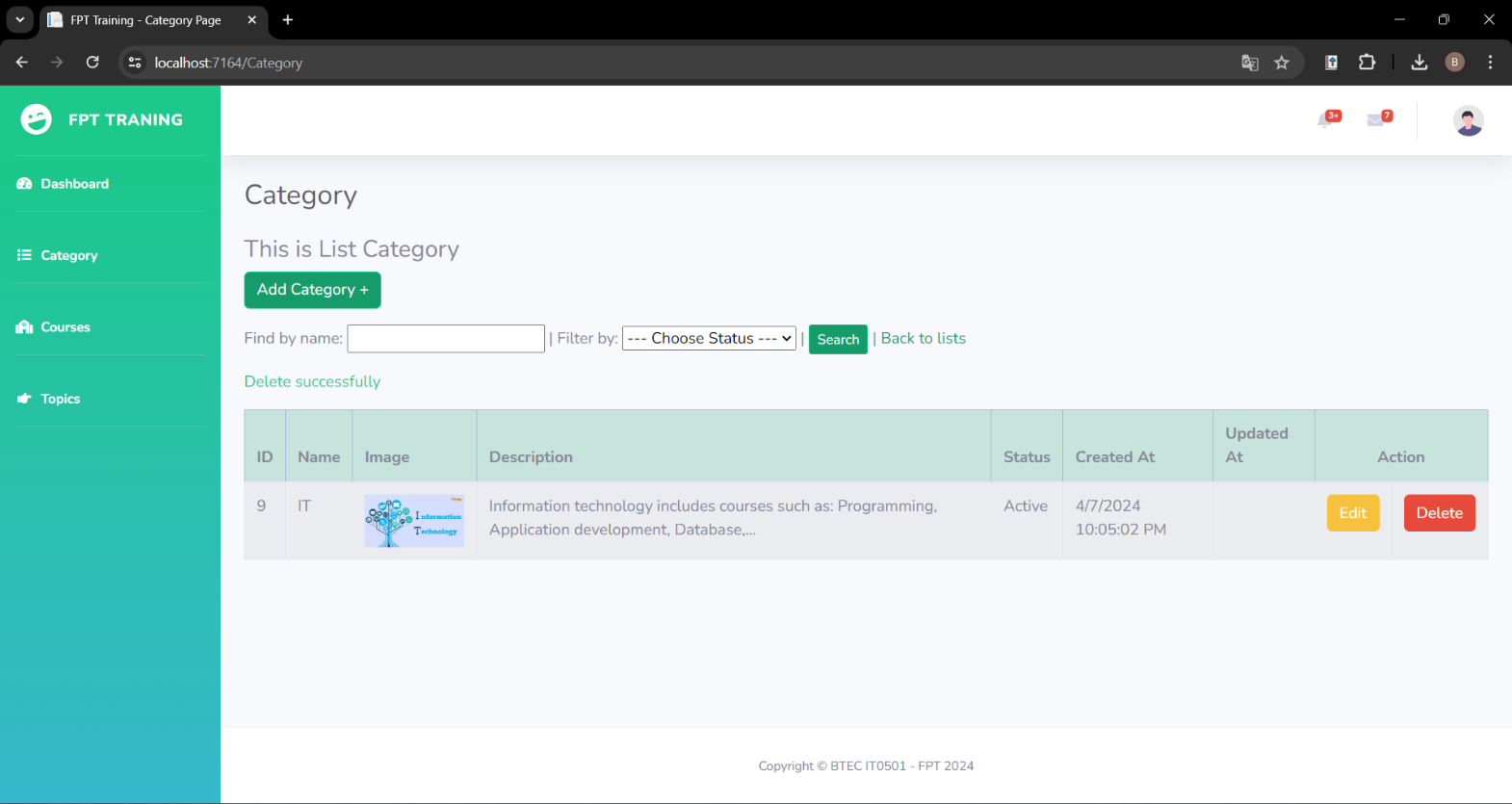
* Delete:

Figure 71: Category delete

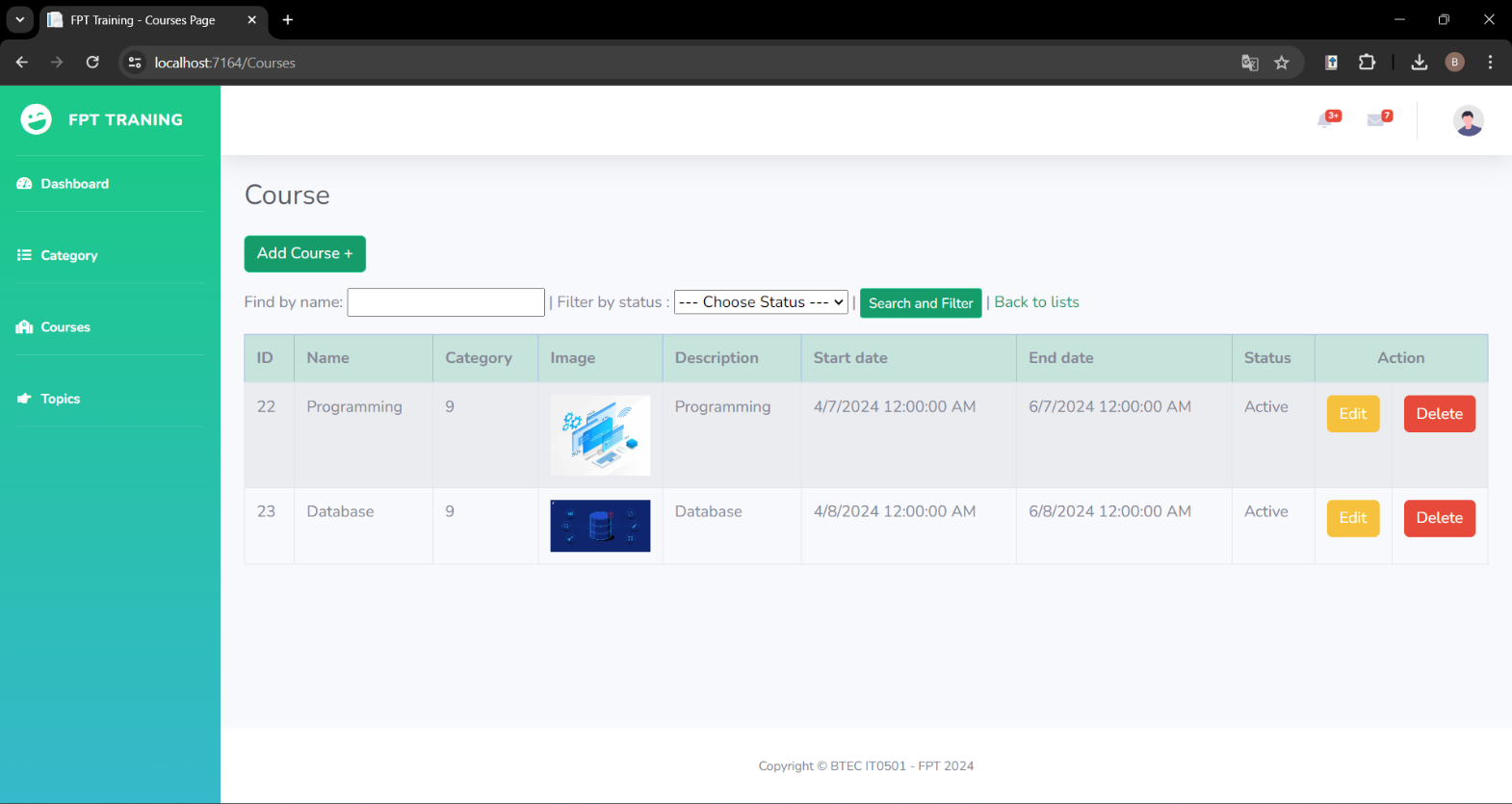
* 1. **Course management**
* Index:

Figure 72: Course index

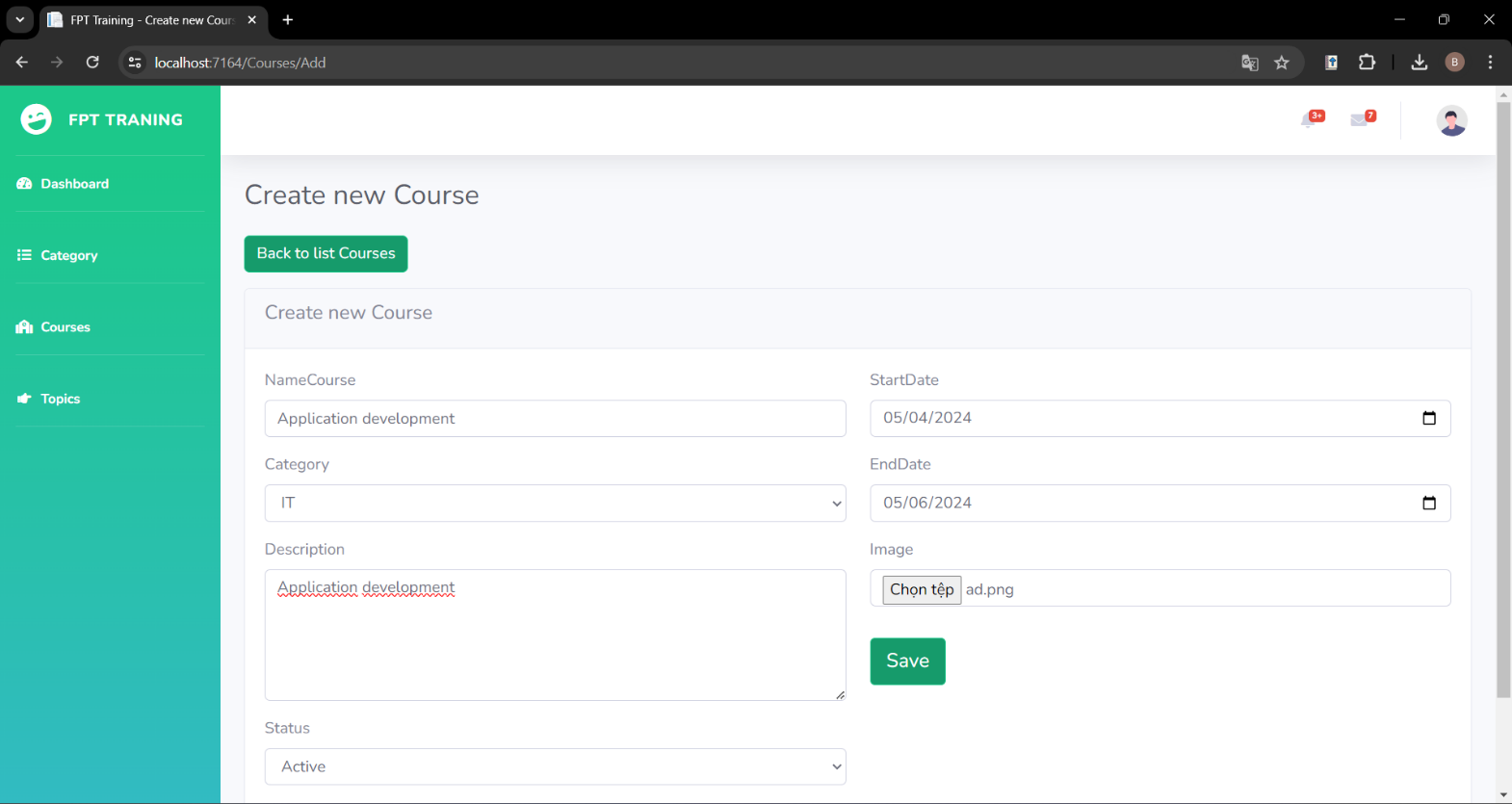
* Add:

Figure 73: Course add

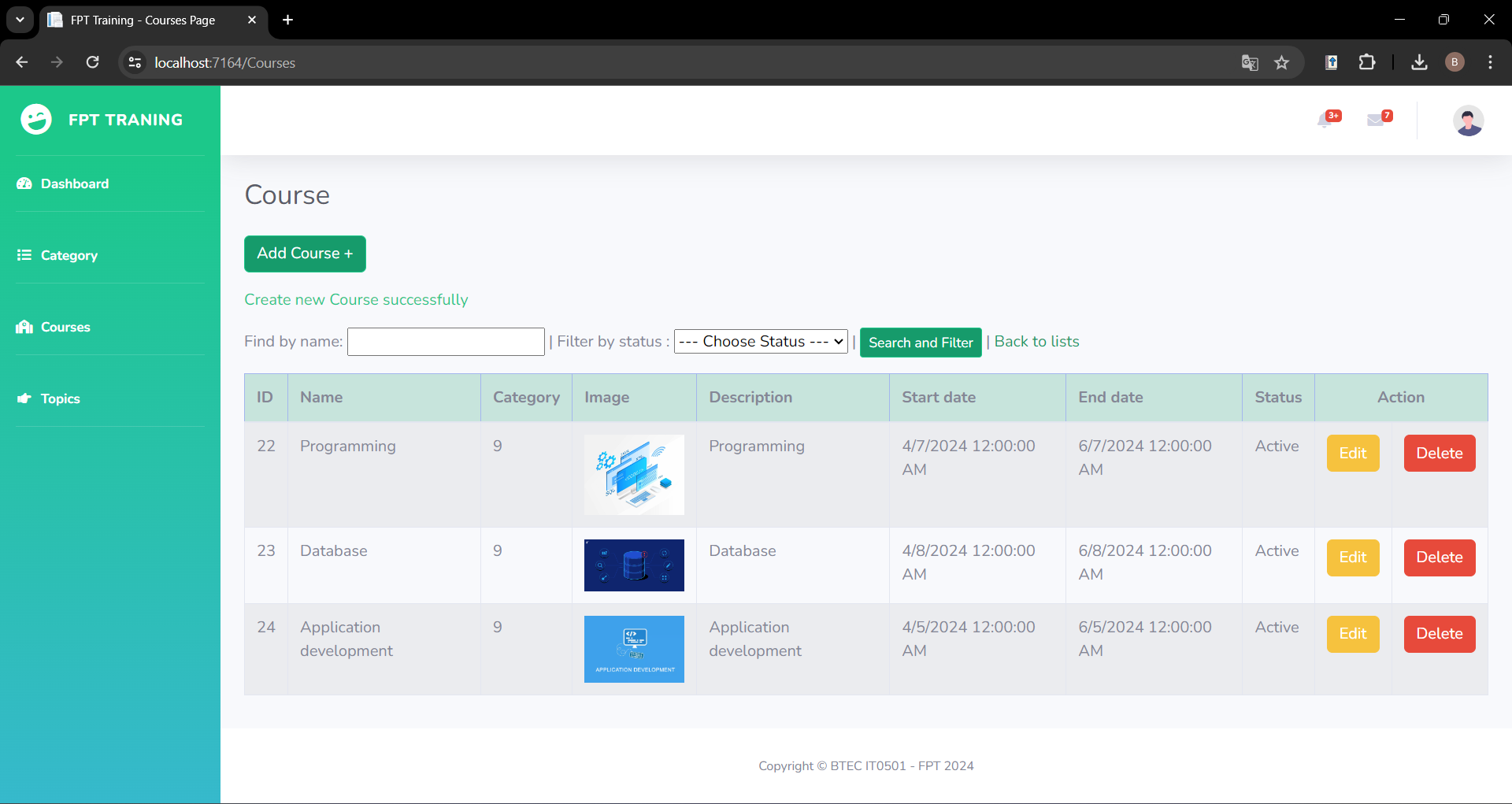
* Interface after adding:

Figure 74: Course interface after adding

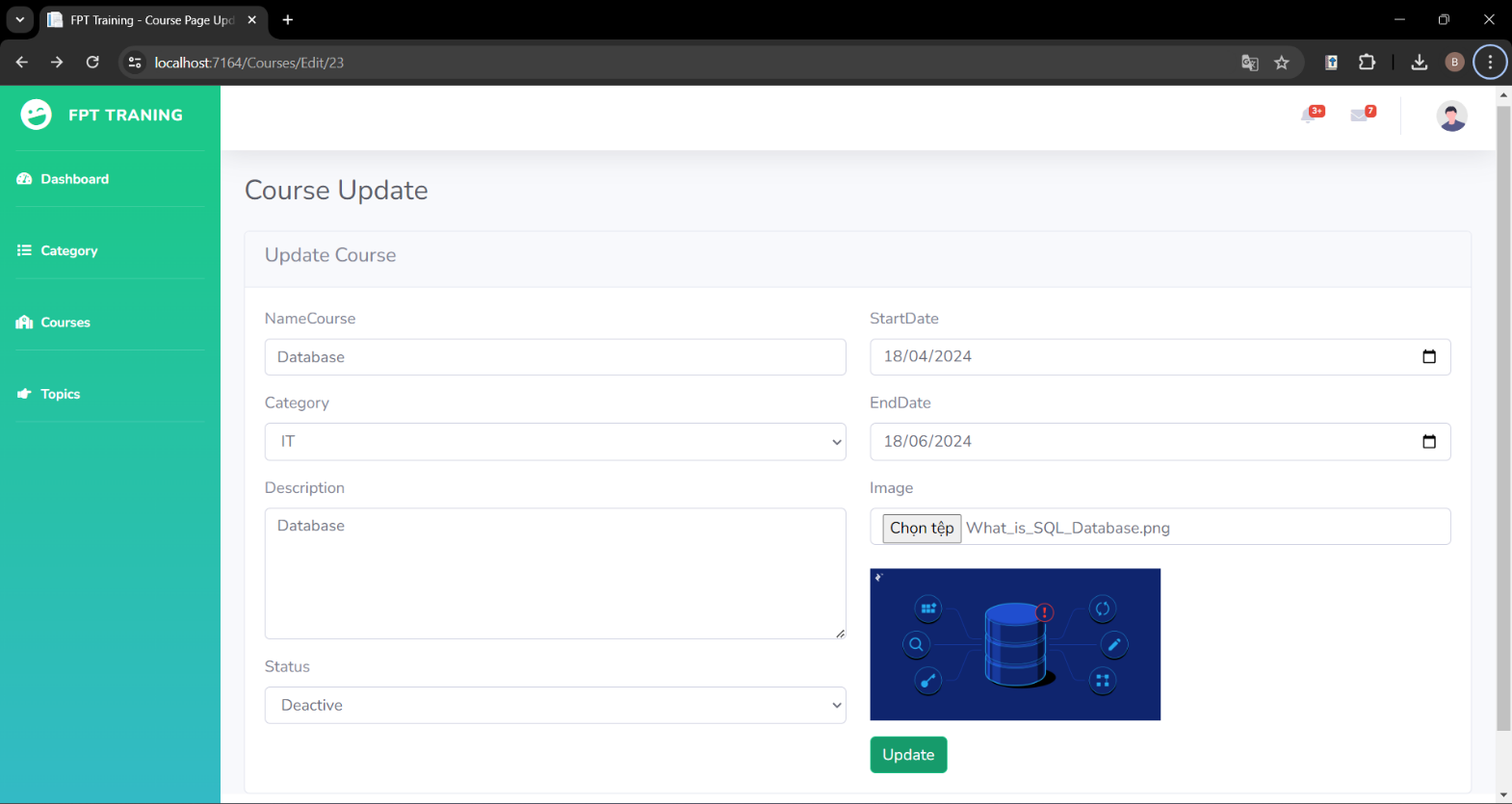
* Update:

Figure 75: Course update

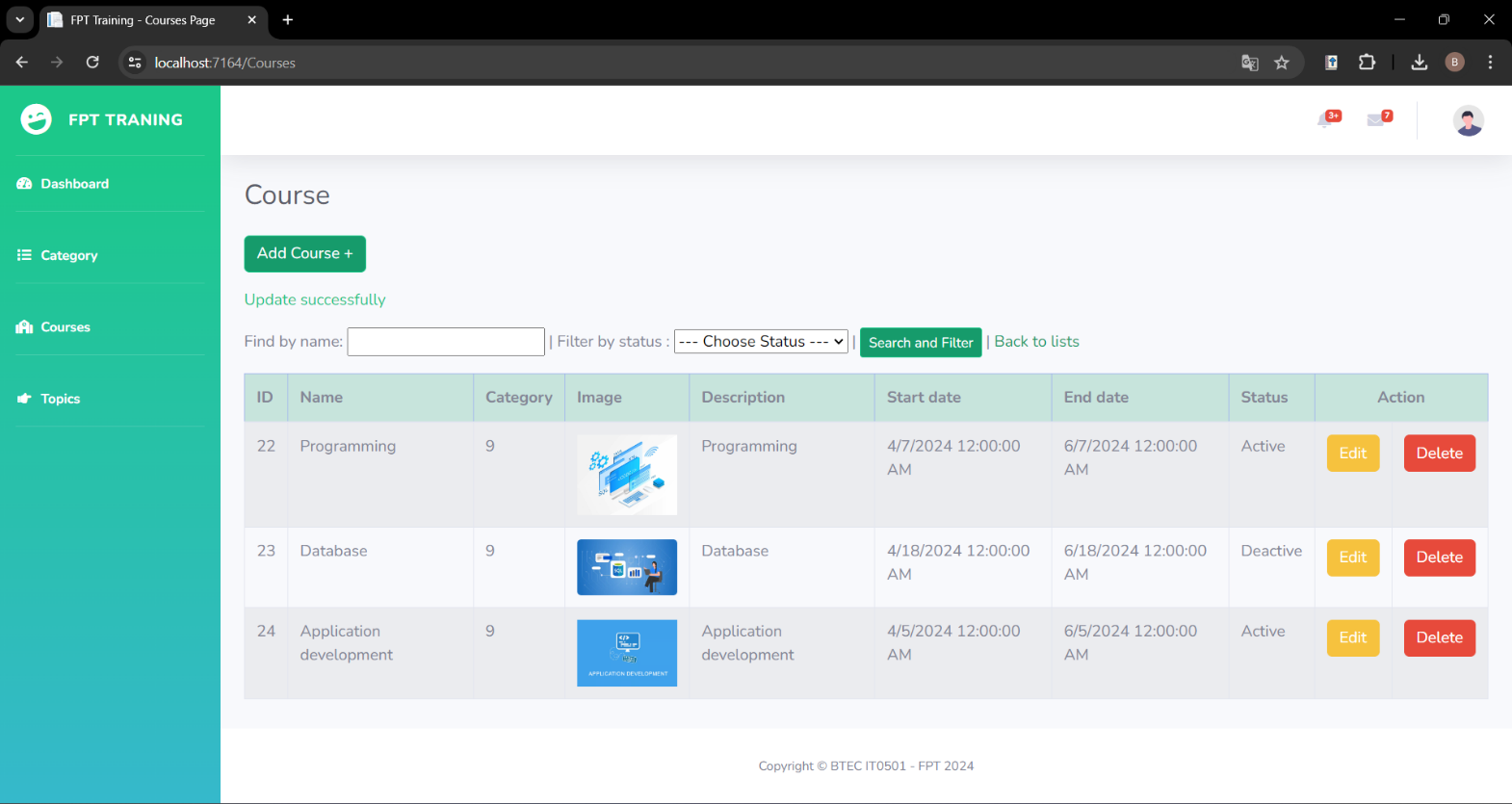
* Interface after update:

Figure 76: Course interface after updating

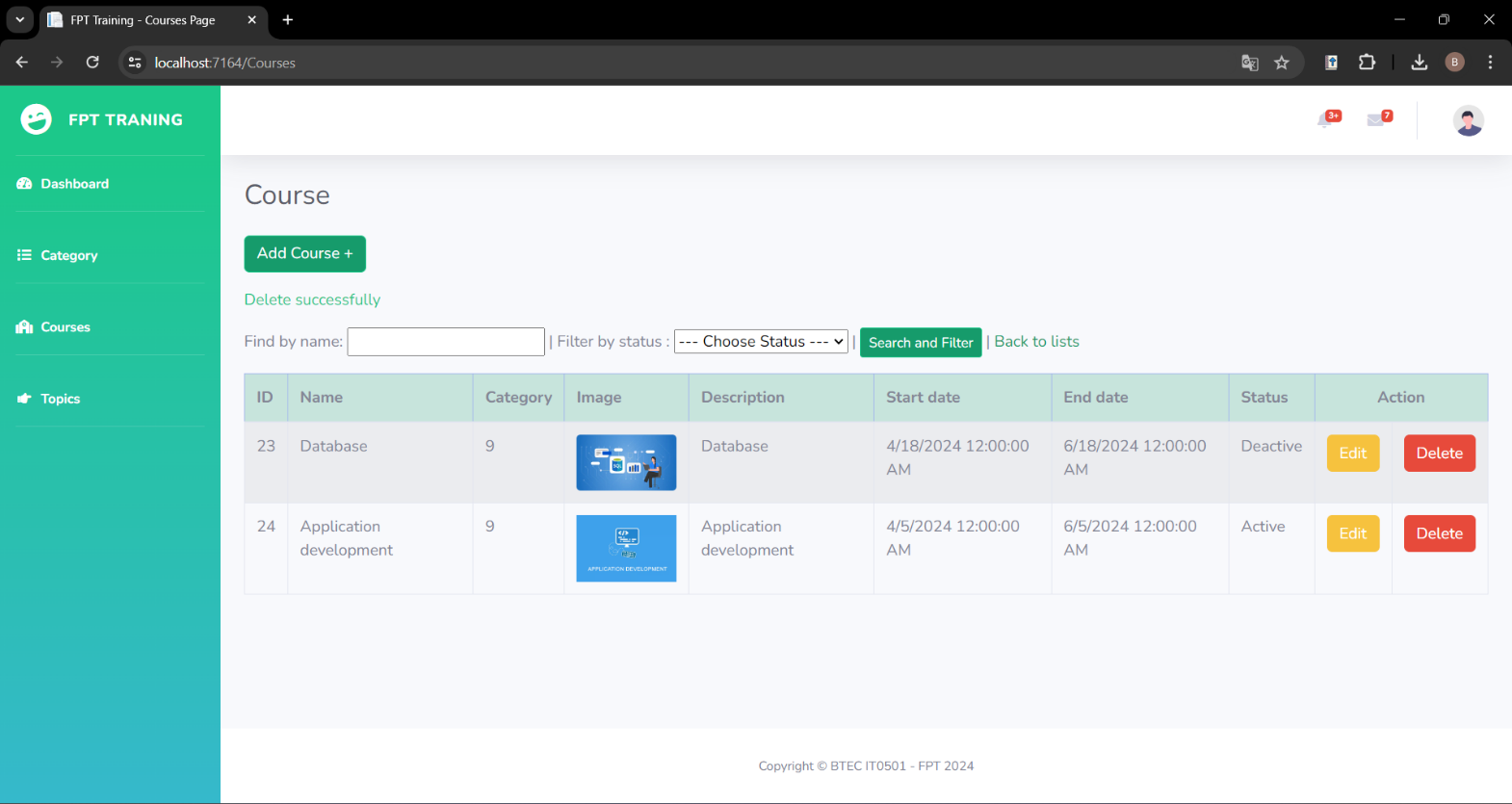
* Delete:

Figure 77: Course delete

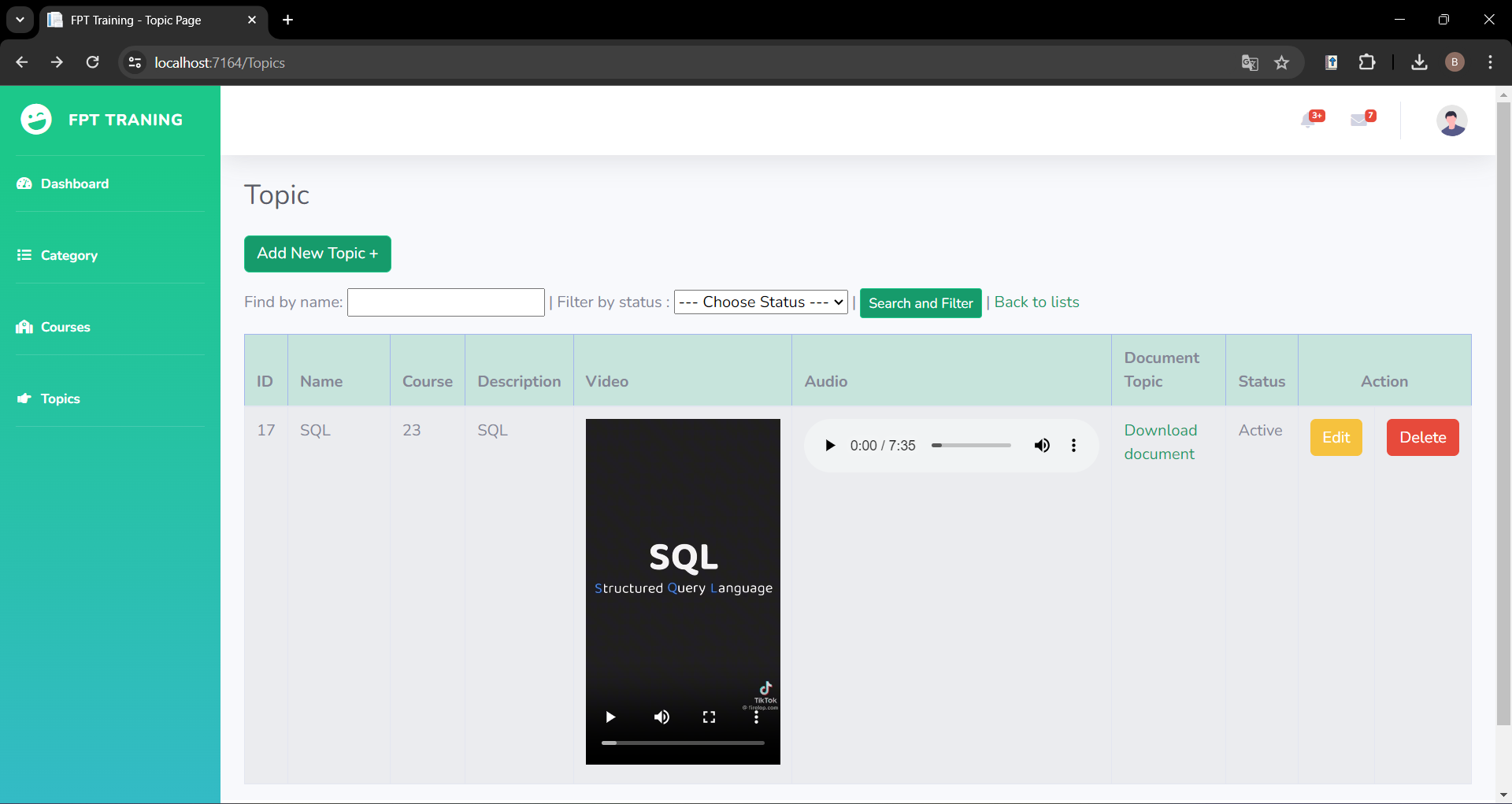
* 1. **Topic management**
* Index:

Figure 78: Topic index

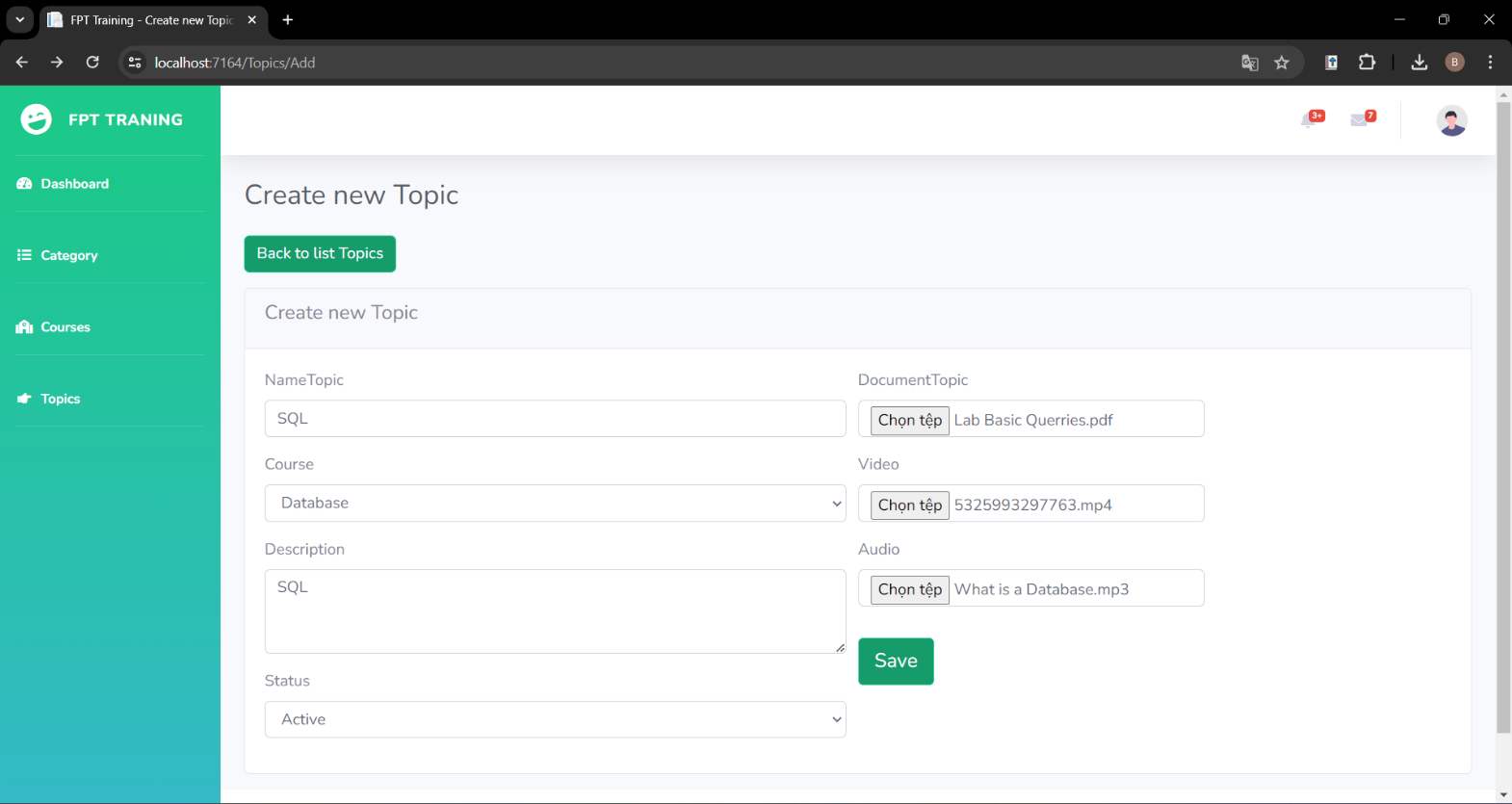
* Add:

Figure 79: Topic add

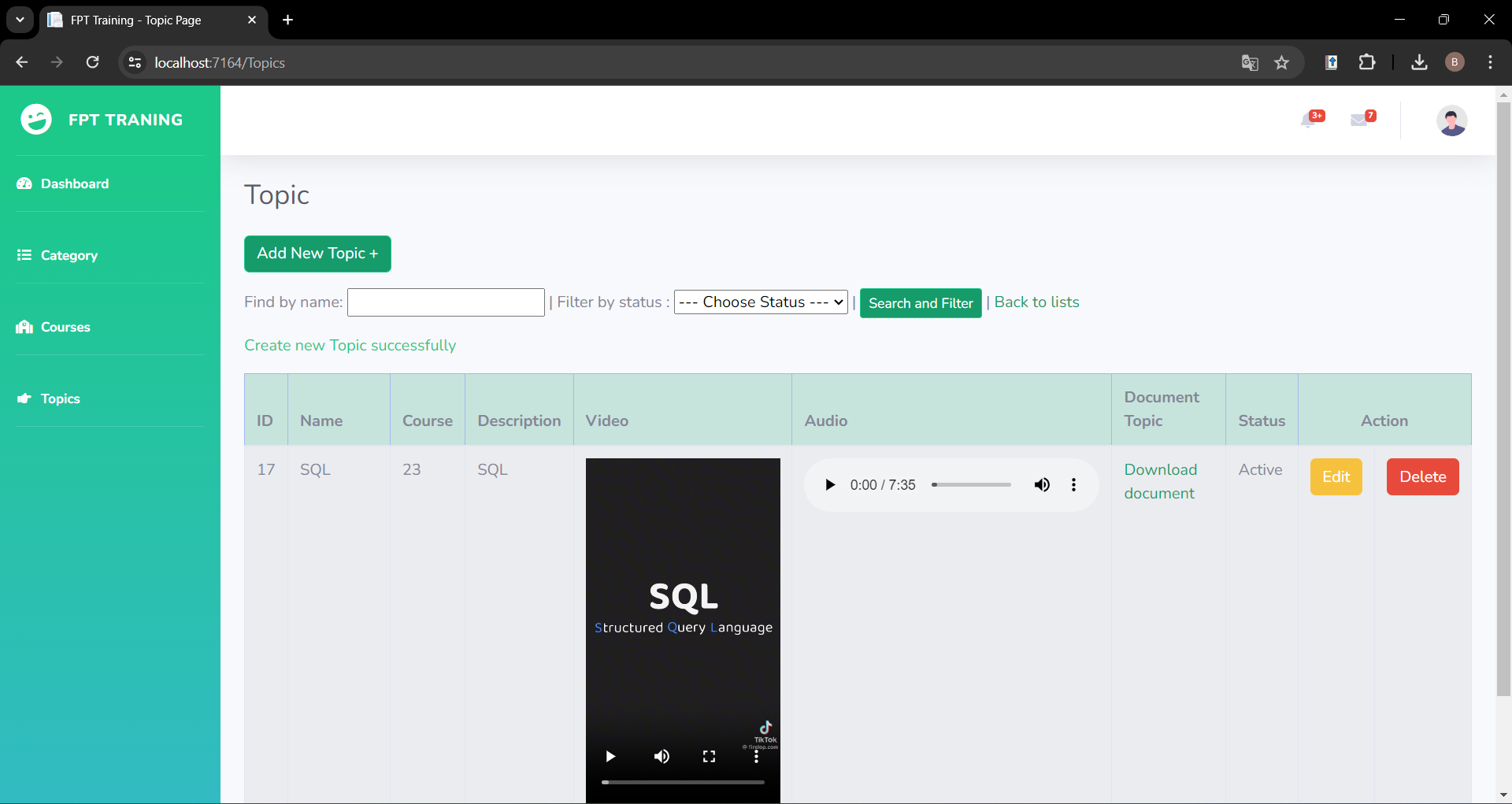
* Interface after adding:

Figure 80: Topic interface after adding

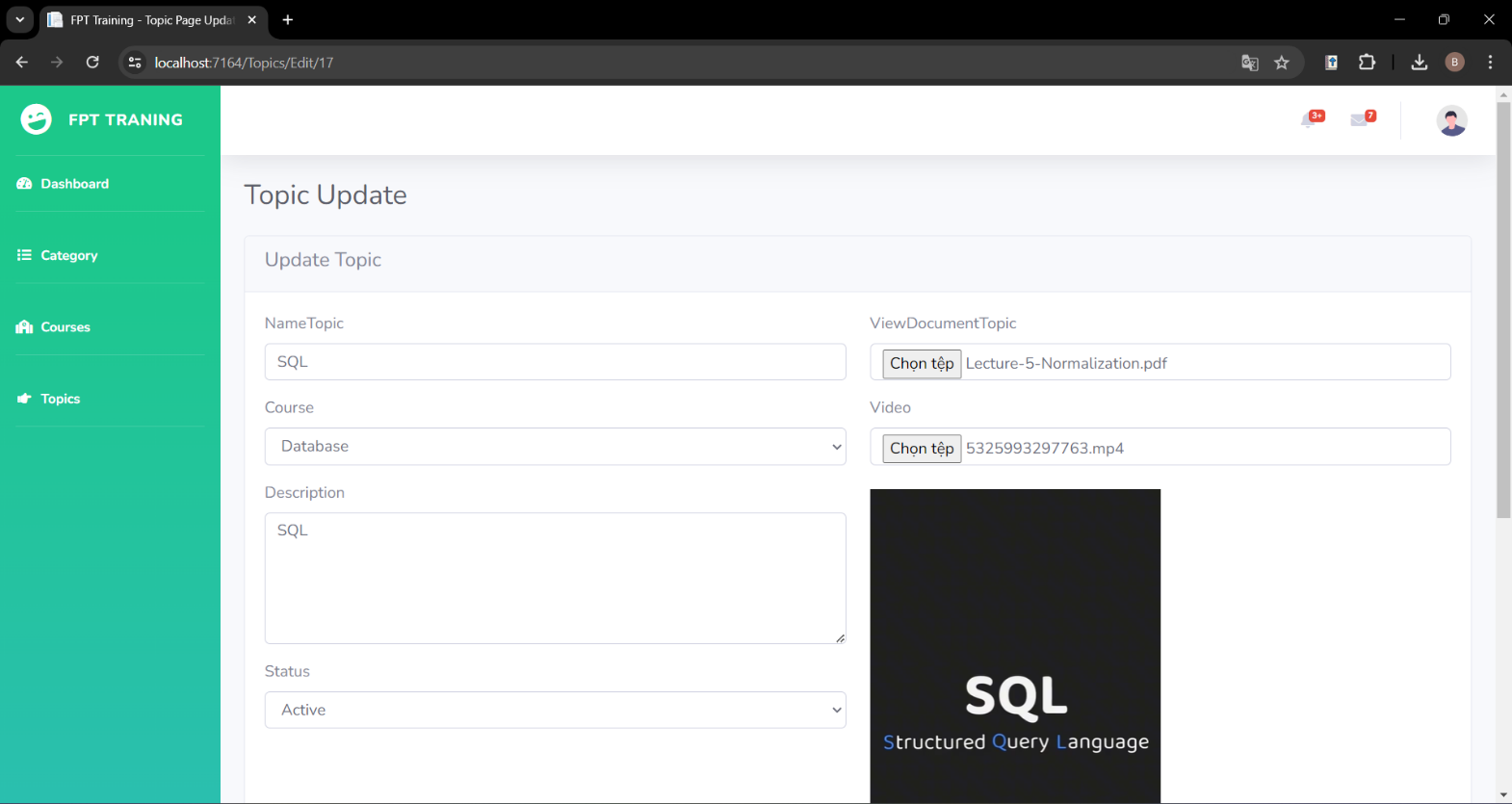
* Update:

Figure 81: Topic update

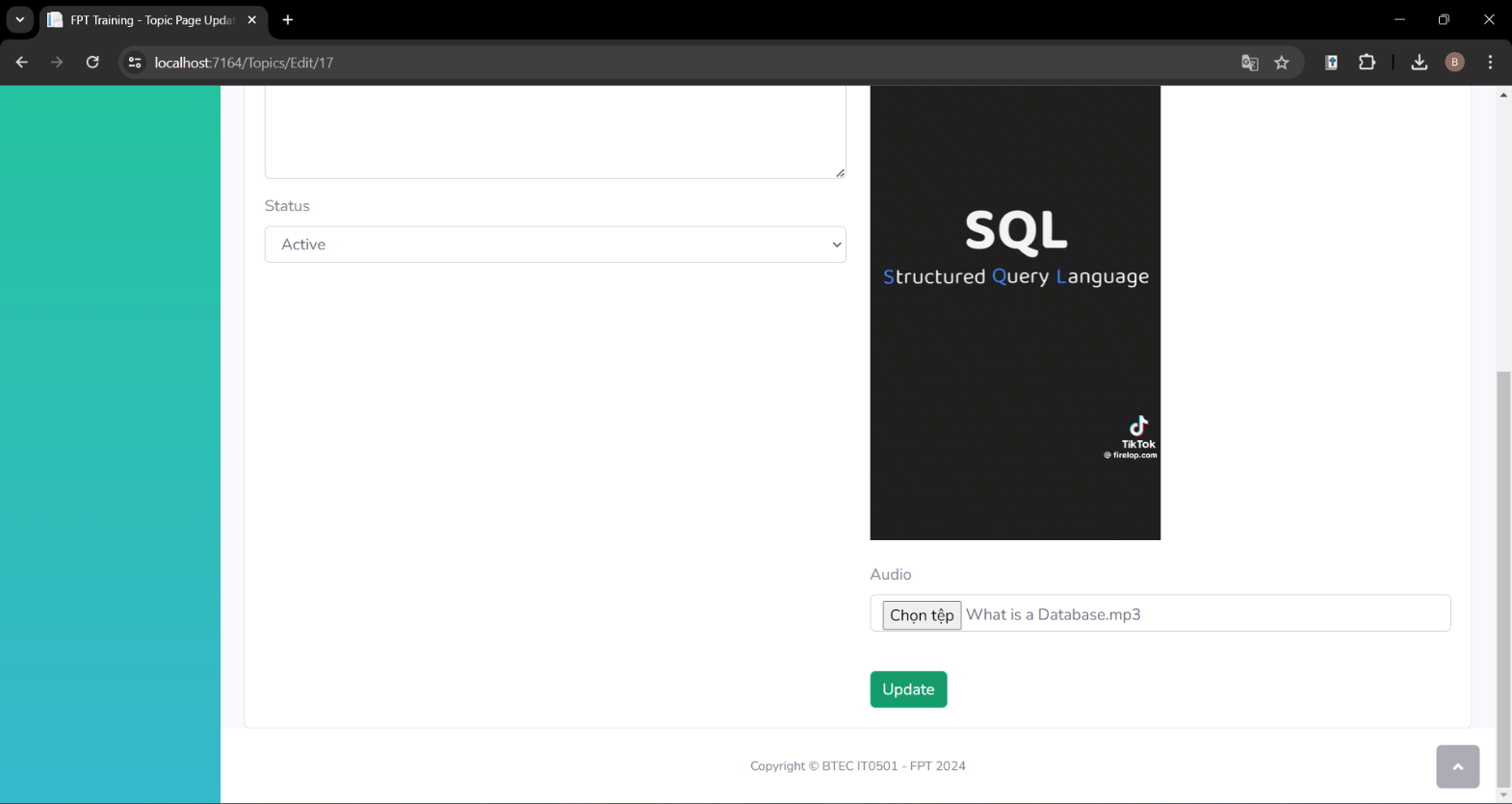


Figure 82: Topic update

* Interface after update:

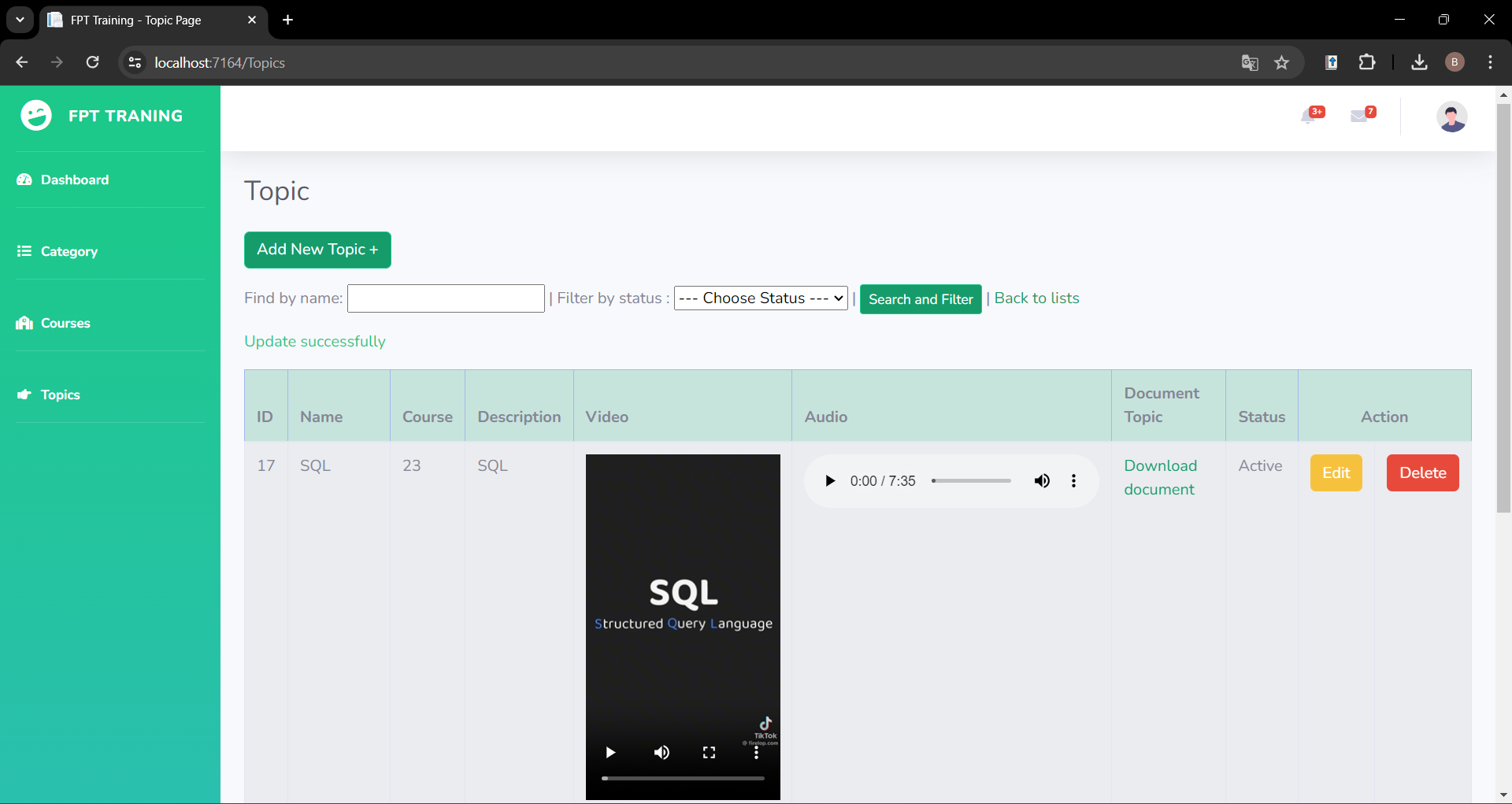


Figure 83: Topic interface after updating

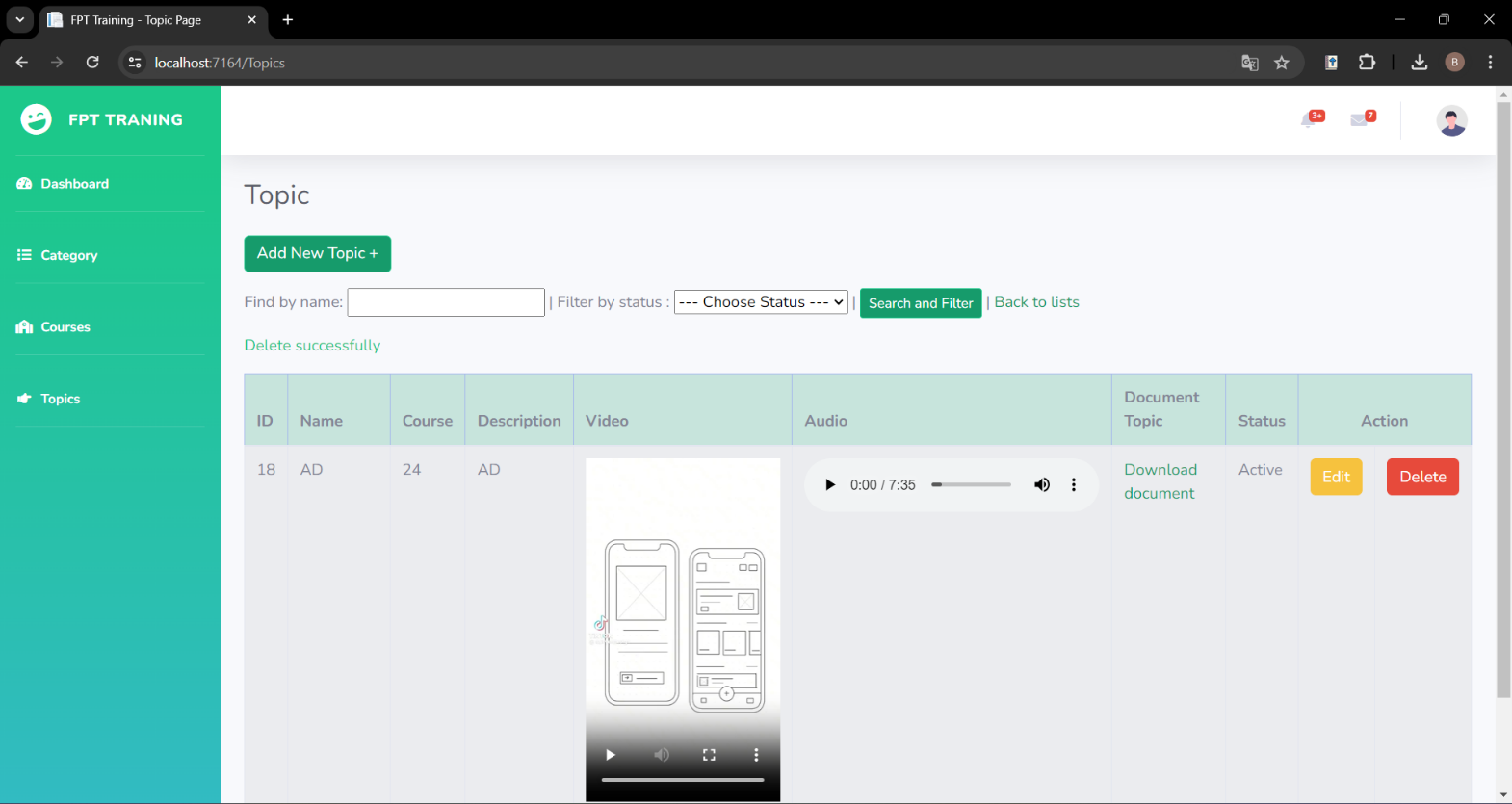
* Delete:

Figure 84: Topic delete

After deleting, the screen will display the message “Delete successfully”

## **P6 Review the performance of your business application against the Problem Definition Statement and initial requirements**

1. **Results of the project**

As in part P5, I have provided images of the system interface, I have implemented the functions of login, category management, course management and topic management. Each management section will include add, edit, and delete functions. In general, the system operates stably and all functions can operate normally. To ensure system quality, in this part I will create a test plan and test the features using a test case table.

1. **Test plan**

A test plan is a document that outlines the approach, scope, resources, and schedule for testing a software application or system. It serves as a guide for the testing process, detailing the objectives, strategies, and methodologies that will be used to ensure the quality and reliability of the software. In this part, I will create a test plan to check the quality of the system.

**Test Plan: Internal Training Management System**

* 1. ***Introduction***

This test plan outlines the testing approach, scope, resources, and schedule for the development of an internal training management system. Functions tested include: login, category management, course management, and topic management.

* 1. ***Objectives***
* Verify the functionality and usability of the internal training management system.
* Make sure the system can login
* Ensure the accuracy and reliability of category, course, and topic management features.
* Identify and address any defects or issues in the system.
  1. ***Scope***

The testing will cover the following areas:

* Login function
* Category Management:
  + Add of categories
  + Edit of categories
  + Delete of categories
* Course Management:
  + Add of courses
  + Edit of courses
  + Delete of courses
* Topic Management:
  + Add of topics
  + Edit of topics
  + Delete of topics
  1. ***Approach***

Testing will be conducted using a combination of manual and automated testing techniques. Test cases will be designed to verify the functionality of each feature, including boundary and edge cases. Both positive and negative test scenarios will be considered.

* 1. ***Schedule***

Testing will commence following the completion of development and continue until all identified issues are resolved. The estimated testing duration is two weeks.

* 1. ***Resources***
* Test Environment: Development environment with access to the internal training management system.
* Test Data: Sample data for category, course, topic management and login.
  1. ***Test Cases***
* Login
* Category Management
  + Verify the adding of a new category.
  + Verify editing an existing category.
  + Verify deletion of a category.
* Course Management
  + Verify the adding of a new course.
  + Verify editing an existing course.
  + Verify deletion of a course.
* Topic Management
  + Verify the adding of a new topic.
  + Verify editing an existing topic.
  + Verify deletion of a topic.
  1. ***Exit Criteria***
* All critical and high-priority defects have been resolved.
* Test coverage of at least 90% for category, course, topic management and login features.
* System performance meets acceptable standards.

1. **Test case**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Test case | Step | Expected output | Actual output | Evalution |
| TC1 | Test Login | 1. Open https://localhost:7164/  2. Enter the correct username and password | Successfully logged in and displayed the Dashboard page. | Successfully logged in and displayed the Dashboard page. | Pass |
| TC2 | Test Login | 1. Open https://localhost:7164/  2. Enter the wrong username and password | Login fails, still displays the login page for the user to re-enter the correct login name and password. | Login fails, still displays the login page for the user to re-enter the correct login name and password. | Pass |
| TC3 | Test Category page | Select “Category” | Displays the Category page | Displays the Category page | Pass |
| TC4 | Test Add Category | 1. Select the Add Category button  2. Enter Name  3. Enter Status  4. Enter PosterImage  5. Enter Description  6. Select the Save button | Displays a notice of successful addition and returns the index page with the newly added Category. | Displays a notice of successful addition and returns the index page with the newly added Category. | Pass |
| TC5 | Test Add Category | 1. Select the Add Category button  2. Enter Status  3. Enter PosterImage  4. Enter Description  5. Select the Save button  (Leave the name blank) | Unable to add new Category.  Display a message asking to enter an additional name. | Unable to add new Category.  Display a message asking to enter an additional name. | Pass |
| TC6 | Test Edit Category | 1. Select the Edit button  2. Enter Name  3. Enter Status  4. Enter PosterImage  5. Enter Description  6. Select the Update button | Displays a successful update message and returns the index page with the newly updated Category. | Displays a successful update message and returns the index page with the newly updated Category. | Pass |
| TC7 | Test Delete Category | Enter the Delete button | Display a successful deletion message and delete the selected Category. | Display a successful deletion message and delete the selected Category. | Pass |
| TC8 | Test Add Course | 1. Select the Add Course button  2. Enter Name  3. Enter Category  4. Enter Image  5. Enter Description  6. Enter Start date  7. Enter End date  8. Enter Status  9. Select the Save button | Displays a notice of successful addition and returns the index page with the newly added Course. | Displays a notice of successful addition and returns the index page with the newly added Course. | Pass |
| TC9 | Test Add Course | 1. Select the Add Course button  2. Enter Name  3. Enter Category  4. Enter Image  5. Enter Description  6. Enter End date  7. Enter Status  8. Select the Save button  ((Leave the Start date blank) | Cannot add new Courses.  Display a message asking to enter additional Start date. | Cannot add new Courses.  Display a message asking to enter additional Start date. | Pass |
| TC10 | Test Edit Course | 1. Select the Edit button  2. Enter Name  3. Enter Category  4. Enter Image  5. Enter Description  6. Enter Start date  7. Enter End date  8. Enter Status  9. Select the Update button | Displays a successful update message and returns the index page with the newly updated Course. | Displays a successful update message and returns the index page with the newly updated Course. | Pass |
| TC11 | Test Delete Course | Enter the Delete button | Display a successful deletion message and delete the selected Course. | Display a successful deletion message and delete the selected Course. | Pass |
| TC12 | Test Add Topic | 1. Select the Add new Topic button  2. Enter Name  3. Enter Course  4. Enter Image  5. Enter Description  6. Enter Status  7. Choose Document topic  8. Choose Video  9. Choose Audio  10. Select the Save button | Displays a notice of successful addition and returns the index page with the newly added Topic. | Displays a notice of successful addition and returns the index page with the newly added Topic. | Pass |
| TC13 | Test Add Topic | 1. Select the Add new Topic button  2. Enter Name  3. Enter Course  4. Enter Image  5. Enter Description  6. Enter Status  7. Choose Document topic  8. Choose Audio  9. Select the Save button  (Leave the video blank) | Cannot add new Topics.  Display a message asking to enter additional Video. | Cannot add new Topics.  Display a message asking to enter additional Video. | Pass |
| TC14 | Test Edit Topic | 1. Select the Edit button  2. Enter Name  3. Enter Course  4. Enter Image  5. Enter Description  6. Enter Status  7. Choose Document topic  8. Choose Video  9. Choose Audio  10. Select the Update button | Displays a successful update message and returns the index page with the newly updated Topic. | Displays a successful update message and returns the index page with the newly updated Topic. | Pass |
| TC15 | Test Delete Topic | Enter the Delete button | Display a successful deletion message and delete the selected Topic. | Display a successful deletion message and delete the selected Topic. | Pass |

1. **Conclusion**

In general, the system achieved passing results in the proposed test cases:

* For the login function, when entering the correct username and password, the user can log in to the home page and vice versa.
* For the Add function of Category, when entering all necessary information in the correct format, a new Category will be added. When input is missing, Category is not added and a message is displayed asking for additional input.
* For the Update function of Category, when entering all necessary information in the correct format, the Category will be updated. On the contrary, when input is missing, the Category is not updated and a message is displayed asking for additional input.
* For the Delete function of Category, the selected Category will be deleted when delete is pressed.
* For the Add function of Course, when entering all necessary information in the correct format, a new Course will be added. When input is missing, the Course is not added and a message asking for additional input is displayed.
* For the Update function of the Course, when you enter all the necessary information in the correct format, the Course will be updated. On the contrary, when input is missing, the Course will not be updated and a message will be displayed asking for additional input.
* For the Delete function of a Course, the selected Course will be deleted when delete is pressed.
* For the Add function of Topic, when entering all necessary information in the correct format, a new Topic will be added. When input is missing, Topic is not added and a message is displayed asking for additional input.
* For the Update function of Topic, when entering all necessary information in the correct format, the Topic will be updated. On the contrary, when input is missing, the Topic will not be updated and a message will be displayed asking for additional input.
* For the Delete function of Topic, the selected Topic will be deleted when pressing delete.

In short, the proposed test cases all returned passing results. However, in order for the system to be more complete, it is necessary to develop some additional functions such as: assigning topics to courses, assigning lecturers to topics, assigning students to courses.

## **M3 Interpret your peer-review feedback and identify opportunities not previously considered**

## **M4 Develop a functional business application based on a specific Software Design Document with supportive evidence of using the preferred tools, techniques and methodologies**

## **M5 Analyse the factors that influence the performance of a business application and use them to undertake a critical review of the design, development and testing stages of your application. Conclude your review by reflectively discussing your previously identified risks**

1. **CONCLUSION**
2. **REFERENCES**