Lab Exercise 1: Creating Async Endpoints in Web API Core

Objective: To practice creating asynchronous Web API endpoints using .NET Core, focusing on proper use of async/await for handling I/O-bound operations.

Instructions:

1. **Setup a New Web API Project:**
    - Create a new .NET Core Web API project.
    - Install necessary NuGet packages (e.g., Entity Framework Core, SQL Server, etc.).
2. **Create a Simple Database Context:**
    - Define a database context using Entity Framework Core.
    - Create a model class named Book with properties such as BookId, Name, Author, Description, and Price.
    - Use Code-First Migrations to generate the database.
3. **Create Repository Pattern with Async Methods:**
    - Implement a repository for Product with async methods for GetAllBooks, GetBookById, AddBook, UpdateBook, and DeleteBook.
4. **Create Controller Endpoints:**
    - Create a ProductsController with async actions for the following:
        - **GET** /api/books - Returns all books asynchronously.
        - **GET** /api/ books /{id} - Returns a books by ID asynchronously.
        - **POST** /api/ books - Adds a new books asynchronously.
        - **PUT** /api/ books /{id} - Updates a books by ID asynchronously.
        - **DELETE** /api/ books /{id} - Deletes a books by ID asynchronously.
5. **Testing:**
    - Test the endpoints using Postman or a similar tool.
    - Ensure that all endpoints handle data asynchronously without blocking the main thread.
6. **Handling Exceptions:**
    - Implement proper error handling and logging for the asynchronous operations.
    - Use try-catch blocks to handle exceptions and return meaningful HTTP status codes.

Deliverables:

- Submit the GitHub repository link containing the complete project with async methods implemented.

Lab Exercise 2: Securing Web API with Identity and JWT Tokens

Objective: To secure a Web API using ASP.NET Core Identity for authentication and JWT (JSON Web Token) for authorization.

Instructions:

1. **Setup Identity in Web API Project:**
    - Install the necessary NuGet packages for ASP.NET Core Identity.
    - Configure Identity in the Startup.cs/Program.cs class.
    - Create models for ApplicationUser and configure the database context for Identity.
2. **Generate JWT Tokens:**
    - Create a service for generating JWT tokens after successful user authentication.
    - Define the necessary claims and configure token expiration.
3. **Create Authentication Endpoints:**
    - Implement endpoints in AccountController for user registration and login:
        - **POST** /api/account/register - Register a new user.
        - **POST** /api/account/login - Authenticate user and return JWT token.
4. **Protect API Endpoints:**
    - Add [Authorize] attribute to the BooksController created in Lab Exercise 1 to secure the endpoints.
    - Ensure that only authenticated users with a valid JWT token can access the endpoints.
5. **Testing:**
    - Test the registration and login endpoints to obtain JWT tokens.
    - Use the JWT token to access the secured books endpoints.
6. **Role-Based Authorization (Optional):**
    - Implement role-based authorization, restricting certain endpoints to specific user roles (e.g., Admin, User).
    - Update the JWT generation logic to include user roles in the token.

Deliverables:

- Submit the GitHub repository link containing the complete project with identity and JWT authentication implemented.

Lab Exercise 3: Practicing Advanced Features in Web API

Objective: To explore and practice various advanced features provided by .NET Core in a Web API context, including middleware, dependency injection, configuration, and logging.

Instructions:

1. **Middleware:**
    - Create custom middleware to log request and response information.
    - Implement another middleware to handle global exception handling and return consistent error responses.
2. **Dependency Injection:**
    - Register services using the built-in dependency injection (DI) container.
    - Inject the services into controllers using constructor injection.
    - Create a sample service that implements an interface and demonstrate how DI is used to decouple the implementation from the controller.
3. **Configuration:**
    - Utilize the appsettings.json file to manage application configuration.
    - Create a strongly-typed configuration class to bind settings from appsettings.json.
    - Demonstrate how to inject configuration settings into services or controllers.
4. **Logging:**
    - Implement logging using the built-in logging framework.
    - Create different log levels (Information, Warning, Error) in various parts of the application.
    - Log important events like user authentication, CRUD operations, and errors.
5. **Testing:**
    - Test the middleware to ensure it's logging information correctly.
    - Verify the dependency injection works as expected by swapping out service implementations.
    - Check the logs for various levels and confirm the application configuration is correctly applied.

Deliverables:

- Submit the GitHub repository link containing the complete project demonstrating CORE features with explanations in the README file.

These lab exercises should provide comprehensive practice in core aspects of Web API development using .NET Core, from asynchronous programming and securing APIs to leveraging essential framework features.

Lab Exercise 4: Implementing Cross-Origin Resource Sharing (CORS) in Web API

Objective: To understand and implement Cross-Origin Resource Sharing (CORS) in a Web API to allow controlled access to resources from different origins.

Instructions:

1. **Understanding CORS:**
    - Before diving into the implementation, research and document what CORS is, why it's necessary, and how it impacts web applications.
2. **Basic CORS Setup:**
    - Open the Startup.cs/Program.cs file in your existing Web API project.
    - In the ConfigureServices/Main method, add the CORS services:

```
services.AddCors(options =>
{
  options.AddDefaultPolicy(builder =>
  {
    builder.AllowAnyOrigin()
        .AllowAnyMethod()
        .AllowAnyHeader();
  });
});
```

    - In the Configure method, enable CORS middleware by adding:

```
app.UseCors();
```

3. **Restricting CORS Policy:**
    - Modify the CORS policy to be more restrictive by allowing only specific origins, methods, and headers:

```
services.AddCors(options =>
{
   options.AddPolicy("MyPolicy", builder =>
   {
      builder.WithOrigins("https://example.com")
            .AllowAnyMethod()
            .AllowAnyHeader();
   });
});
```

- Apply this policy globally or to specific controllers/actions by using the [EnableCors("MyPolicy")] attribute.

4. **Testing CORS:**
   - Create a simple frontend application (e.g., using HTML and JavaScript or React) that tries to make API requests to your Web API from a different origin.
   - Test different scenarios:
     - Allowing requests from an allowed origin.
     - Blocking requests from a non-allowed origin.
     - Testing with different HTTP methods and headers.
5. **Dynamic CORS Policy:**
   - Implement a dynamic CORS policy where allowed origins can be read from a configuration file (e.g., appsettings.json) and applied at runtime.
   - Update the CORS configuration to read from the settings and apply the policy dynamically.
6. **Handling Preflight Requests:**
   - Ensure your API correctly handles CORS preflight requests (OPTIONS method) by verifying the response headers like Access-Control-Allow-Origin, Access-Control-Allow-Methods, etc.
   - Test how your API responds to preflight requests from the frontend application.
7. **Advanced CORS Scenarios (Optional):**
   - Explore more advanced scenarios, such as allowing credentials (cookies, HTTP authentication) and managing complex headers.
   - Modify the policy to support these advanced requirements.
8. **Documentation:**

- Document your findings, especially any challenges faced during implementation, and provide a summary of how CORS can impact the security and accessibility of web applications.

Deliverables:

- Submit the GitHub repository link containing the complete project with CORS implemented.
- Include the frontend application used for testing CORS.
- Provide documentation explaining CORS, how it was implemented, and the results of your testing.

This lab exercise will help students gain practical experience in implementing and managing CORS in a Web API, a crucial aspect of web security and cross-origin communication.