



C# BASICS

Training Assignments

Document Code	25e-BM/HR/HDCV/FSOFT
Version	1.1
Effective Date	20/11/2012


Hanoi, 06/2019

RECORD OF CHANGES

No	Effective Date	Change Description	Reason	Reviewer	Approver
1.	01/Oct/2018	Create new	Draft		
2.	01/Jun/2019	Update template	Fsoft template	DieuNT1	

Contents

ASP.NET Core Web API: Quiz Application.....	4
Objectives:	4
Prerequisites:	4
Problem Requirements:	4
Task 1: ASP.NET Core Introduction, Dependency Injection and Basic API	4
Task 2: Routing, Controllers, HTTP Methods, and Model Binding	5
Task 3: Middleware and Error Handling	8
Task 4: Authentication/Authorization, Logging, and Security	9
Task 5: Documenting API with Swagger, Versioning, and Deployment	11
Estimated Time for Each Task:	12
Mark Scale:	13

	CODE:	NWEC.P.L002
	TYPE:	MEDIUM
	LOC:	190
	DURATION:	720 MINUTES

ASP.NET Core Web API: Quiz Application

Objectives:

- » Understand and apply ASP.NET Core for building web APIs.
- » Utilize Dependency Injection (DI) to manage dependencies within the application.
- » Implement CRUD (Create, Read, Update, Delete) operations for the Subject entity.
- » Configure routing to map URLs to controller actions.
- » Develop controllers to handle API requests for quizzes, questions, and answers.
- » Implement HTTP methods (GET, POST, PUT, DELETE) for CRUD operations on these entities.
- » Leverage model binding to automatically map request data to action method parameters.
- » Explore middleware concepts for enhancing application functionality.
- » Implement JWT (JSON Web Token) authentication to secure API endpoints.
- » Configure CORS (Cross-Origin Resource Sharing) to allow access from specific origins.
- » Develop custom middleware for handling and logging exceptions during request processing.
- » Implement user authentication and authorization with ASP.NET Core Identity and JWT.
- » Integrate a logging framework (e.g., Serilog, NLog) to capture application events and user actions.
- » Apply security best practices like input validation, HTTPS, and protection against vulnerabilities.
- » Utilize Swagger/OpenAPI to generate interactive documentation for the API.
- » Implement API versioning to manage changes and support different API versions.
- » Explore deployment strategies for deploying the ASP.NET Core Web API application

Prerequisites:

- » Working environment: Visual Studio Code/Visual Studio 2013 or higher.
- » Delivery: Source code packaged in a compress archive.

Problem Requirements:

This assignment challenges you to develop a comprehensive quiz application using ASP.NET Core and Entity Framework Core (EF Core). You'll explore various functionalities like building a well-structured application, implementing data access with EF Core, creating a RESTful API, and securing user interaction.

Task 1: ASP.NET Core Introduction, Dependency Injection and Basic API

Description:

Based on the Entity Framework Core assignment, you will kickstart the development of a Quiz Application using ASP.NET Core and Entity Framework Core.

Function Requirements:

- **Setup Projects**
 - Ensure the project structure (**NWEC.P.L001**) meets requirements.
 - Verify the connection string in **appsettings.json** and **DbContext** for correct configuration.
- **Dependency Injection (DI):**
 - Configure dependency injection services (**AddTransient, AddScoped, AddSingleton**) in Program.cs for injecting dependencies into controllers, services, and middleware.
 - Inject necessary dependencies into QuestionsController.
- **Create Question API Endpoints:**
 - **GET /questions/{id}**

- Description: Get details of a specific question by ID.
- Response:
 - QuestionViewModel: Content, QuestionType, IsActive
- **GET /questions**
 - Description: Retrieve a list of all questions.
 - Response:
 - List<QuestionViewModel>
- **POST /questions**
 - Description: Create a new question with answer.
 - Action method: public async Task<IActionResult> CreateQuestionWithAnswer(QuestionCreateViewModel questionCreateViewModel)
 - Request Body:
 - QuestionCreateViewModel: Content, QuestionType, IsActive, Answer(ICollection<AnswerCreateViewModel>)
 - AnswerCreateViewModel: Content, IsCorrect, IsActive
 - Response: bool
- **PUT /questions/{id}**
 - Description: Update an existing question by ID with updated answers.
 - Action method: public async Task<IActionResult> UpdateQuestionWithAnswer(Guid id, QuestionEditViewModel questionEditViewModel)
 - Request Body:
 - QuestionEditViewModel: Id, Content, QuestionType, IsActive, Answer(ICollection<AnswerEditViewModel>)
 - AnswerEditViewModel: Id, Content, IsCorrect, IsActive
 - Response: bool
- **DELETE /questions/{id}**
 - Description: Delete a question by ID (consider associated answers before deletion).
 - Response: bool

Hints:

- Organize the project structure into separate folders for DAL, BLL, and API.
- Utilize data annotations or Fluent API to enforce business rules and relationships in entity models.
- Write additional methods in services if necessary

Business Rules:

- Entity models should accurately represent the data structure for quizzes, questions, and answers.
- DbContext configuration should define relationships between entities (e.g., one-to-many, many-to-many).
- Connection string should be stored securely in appsettings.json and accessed via IConfiguration in Program.cs.

Evaluation Criteria:

- Successful creation of ASP.NET Core Web API project with appropriate project structure.
- Correct definition of entity models with required properties, constraints, and relationships.
- DbContext setup and service registration demonstrate understanding of Entity Framework Core configuration.
- Completion of QuestionsController with CRUD operations for Question model.

Submission file:

- Zip solution folder to a zip file
- File: FullName_QuizApp_WebAPI_Task_01_v1.0.zip

Estimated Time: 180 minutes.

Task 2: Routing, Controllers, HTTP Methods, and Model Binding.

Description:

This task focuses on implementing routing, controllers, HTTP methods, and model binding to build RESTful endpoints for managing quizzes, questions, and answers within the Quiz Application.

Functions:

- **Routing Configuration:**
 - Use attribute routing ([Route] attribute) to define custom routes for controller actions.
 - Configure routing to map incoming HTTP requests to appropriate controller actions based on route templates.
- **Dependency Injection (DI):**
 - Configure dependency injection services (AddTransient, AddScoped, AddSingleton) in Program.cs to inject dependencies into controllers, services, and middleware.
 - Inject dependencies for use in controllers
- **Controllers Implementation:**
 - Create controllers (UsersController, RolesController, QuizzesController, QuestionsController) to handle CRUD operations for quizzes, questions, and answers.
 - Implement action methods for HTTP GET, POST, PUT, DELETE operations corresponding to API endpoints.
- **Quiz Application API Reference:**

Quiz API:

- **GET /quizzes/{id}**
 - Description: Get details of a specific quiz by ID.
 - Response:
 - QuizViewModel: Title, Description, Duration, IsActive
- **GET /quizzes**
 - Description: Retrieve a list of all quizzes.
 - Response:
 - List<QuizViewModel>
- **POST /quizzes**
 - Description: Create a new quiz with questions.
 - Action method: public async Task<IActionResult> CreateQuizWithQuestions(QuizCreateViewModel quizCreateViewModel)
 - Request Body:
 - QuizCreateViewModel: Title, Description, Duration, IsActive, QuestionIdWithOrders(ICollection<QuestionIdWithOrderViewModel>)
 - QuestionIdWithOrderViewModel: QuestionId, Order
 - Response: bool
- **PUT /quizzes/{id}**
 - Description: Update an existing quiz by ID with updated questions.
 - Action method: public async Task<IActionResult> UpdateQuizWithQuestions(Guid id, QuizEditViewModel quizEditViewModel)

- Request Body:
 - QuizEditViewModel: Id, Title, Description, Duration, IsActive, QuestionIdWithOrders(ICollection<QuestionIdWithOrderViewModel>)
 - QuestionIdWithOrderViewModel: QuestionId, Order
- Response: bool
- **DELETE /quizzes/{id}**
 - Description: Delete a quiz by ID.
 - Response: bool

User API:

- **GET /users/{id}**
 - Description: Get details of a specific user by ID.
 - Response:
 - UserViewModel: Id, FirstName, LastName, DisplayName, Email, UserName, PhoneNumber, IsActive
- **GET /users**
 - Description: Retrieve a list of all user.
 - Response:
 - List<UserViewModel>
- **POST /users**
 - Description: Create a new user.
 - Action method: public async Task<IActionResult> CreateUser(UserCreateViewModel userCreateViewModel)
 - Request Body:
 - UserCreateViewModel: FirstName, LastName, Email, UserName, PhoneNumber, Password, ConfirmPassword, DateOfBirth, IsActive.
 - Response: bool
- **PUT /quizzes/{id}**
 - Description: Update an existing user by ID.
 - Action method: public async Task<IActionResult> UpdateUser(Guid id, QuizEditViewModel userEditViewModel)
 - Request Body:
 - UserEditViewModel: Id, FirstName, LastName, PhoneNumber, DateOfBirth, IsActive,
 - Response: bool
- **DELETE /users/{id}**
 - Description: Delete a user by ID.
 - Response: bool.
- **POST /users/changePassword**
 - Description: Change password of an user.
 - Action method: public async Task<IActionResult> ChangePassword(ChangePasswordViewModel ChangePasswordViewModel)
 - Request Body:
 - UserCreateViewModel: Id, UserName, CurrentPassword, NewPassword, ConfirmPassword.
 - Response: bool

Role API:

- **GET /roles/{id}**
 - Description: Get details of a specific role by ID.

- Response:
 - RoleViewModel: Id, Name, Description, IsActive
- **GET /roles**
 - Description: Retrieve a list of all roles.
 - Response:
 - List<RoleViewModel>
- **POST /roles**
 - Description: Create a new role.
 - Action method: `public async Task<IActionResult> CreateRole(RoleCreateViewModel roleCreateViewModel)`
 - Request Body:
 - RoleCreateViewModel: Name, Description, IsActive.
 - Response: bool
- **PUT /roles/{id}**
 - Description: Update an existing role by ID.
 - Action method: `public async Task<IActionResult> UpdateRole (Guid id, RoleEditViewModel roleEditViewModel)`
 - Request Body:
 - RoleEditViewModel: Id, Name, Description, IsActive.
 - Response: bool
- **DELETE /roles/{id}**
 - Description: Delete a role by ID.
 - Response: bool.

Hints:

- Use attribute routing ([Route]) to define custom routes for controller actions.
- Implement controller actions to handle HTTP GET, POST, PUT, DELETE requests for different resource endpoints.
- Utilize model binding to bind JSON data from requests to action method parameters.

Business Rules:

- Ensure controllers follow RESTful conventions with separate controllers for different resource types (users, roles, quizzes, questions, answers).
- Use appropriate HTTP methods (GET, POST, PUT, DELETE) for performing CRUD operations on resources.
- Implement model binding to handle request data and enforce data validation rules.

Evaluation Criteria:

- Successful implementation of controllers with action methods for CRUD operations on quizzes, users, roles
- Proper routing configuration to map HTTP requests to controller actions based on route templates.
- Use of HTTP methods and model binding to process request data and interact with the DbContext for data persistence.
- Adherence to RESTful principles and separation of concerns within controllers for different resource types.

Submission file:

- Zip solution folder to a zip file
- File: FullName_QuizApp_WebAPI_Task_02_v1.0.zip

Estimated Time: 180 minutes.

Task 3: Middleware and Error Handling

Description:

In this task, you will explore middleware, and error handling to enhance the robustness and scalability of the Quiz Application.

Function Requirements:

- **Middleware Configuration:**
 - **Authentication Middleware:**

Authentication middleware in ASP.NET Core is used to verify and authenticate incoming requests based on provided credentials (e.g., tokens, cookies). It intercepts requests and checks for valid authentication tokens or session information.

IAuthService:

- **Task<LoginResponseViewModel> LoginAsync(LoginViewModel loginViewModel);**
 - LoginViewModel: UserName, Password.
 - LoginResponseViewModel: UserInformation (JSON string), Token, Expires
- **Task<LoginResponseViewModel> RegisterAsync(RegisterViewModel registerViewModel);**
 - RegisterViewModel: FirstName, LastName, Email, UserName, PhoneNumber, Password, ConfirmPassword, DateOfBirth.
 - LoginResponseViewModel: UserInformation (JSON string), Token, Expires

AuthController:

- **POST /login**
 - Description: Login with UserName and Password.
 - Action method: public async Task<ActionResult> Login(LoginViewModel loginViewModel)
 - Request Body:
 - LoginViewModel: UserName, Password.
 - Response:
 - LoginResponseViewModel: UserInformation (JSON string), Token, Expires
- **POST /register**
 - Description: Register with UserName and Password.
 - Action method: public async Task<ActionResult> Register(RegisterViewModel registerViewModel)
 - Request Body:
 - RegisterViewModel: FirstName, LastName, Email, UserName, PhoneNumber, Password, ConfirmPassword, DateOfBirth.
 - Response:
 - LoginResponseViewModel: UserInformation (JSON string), Token, Expires

Requirements:

- Implement JWT (JSON Web Tokens) authentication middleware.
- Configure token validation parameters (issuer, audience, signing key).
- Use AddAuthentication() method to enable authentication services.

- Use `UseAuthentication()` and `UseAuthorization()` in the request pipeline to enable authentication and authorization.
- **CORS Middleware:**

CORS (Cross-Origin Resource Sharing) middleware in ASP.NET Core is used to configure and enable cross-origin requests from different domains to access resources exposed by the application.

Requirements:

- Configure CORS policies to specify which origins, methods, and headers are allowed.
Origins: <http://localhost:4200>, <https://localhost:4200>
Methods: GET, POST, PUT, DELETE
Headers: "X-Request-Token", "Accept", "Content-Type", "Authorization"
- Use `AddCors()` to register CORS services.
- Use `UseCors()` to apply CORS policies to requests.
- **Error Handling:**
 - Implement global error handling middleware to catch and log exceptions thrown during request processing.
 - Use status codes (e.g., `BadRequest`, `NotFound`, `InternalServerError`) to return appropriate HTTP responses for different error scenarios.

Hints:

- Install the `Microsoft.AspNetCore.Authentication.JwtBearer` package for JWT authentication.
- Configure `TokenValidationParameters` to specify token validation requirements.
- CORS Middleware: Configure specific policies to allow/deny origins, methods, and headers.
- Use `app.UseMiddleware<TMiddleware>()` to add custom middleware components to the ASP.NET Core pipeline.
- Create a custom middleware class implementing `IMiddleware` or using `RequestDelegate` to handle exceptions.
- Catch exceptions using try-catch blocks within the middleware `InvokeAsync()` method.
- (Optional) Log exception details using logging frameworks like `Serilog` or `NLog`.
- Set appropriate HTTP status codes and response messages based on exception types..

Business Rules:

- Ensure tokens are validated against trusted issuers and audience.
- Implement token expiration and refresh mechanisms if needed.
- Use secure methods for handling sensitive user credentials..
- Restrict cross-origin requests to trusted domains for security purposes.
- Use appropriate CORS policies based on application requirements (e.g., allow specific origins only).
- Provide consistent error responses to clients for better user experience.
- Log exception details for troubleshooting and debugging purposes.
- Ensure middleware does not expose sensitive information in error responses.

Evaluation Criteria:

- Implementation of JWT authentication middleware with correct token validation parameters.
- Proper configuration of `AddAuthentication()` and `AddJwtBearer()` in `Program.cs`.
- Correct usage of `UseAuthentication()` and `UseAuthorization()` in the request pipeline.

- Proper configuration of CORS policies in Program.cs.
- Implementation of services.AddCors() and app.UseCors() with correct policy settings.
- Ensure CORS policies align with security and application requirements.
- Implementation of custom exception handling middleware with InvokeAsync() method.
- Proper logging of exception details using logging frameworks.
- Return appropriate HTTP status codes and error messages for different exception scenarios.

Submission file:

- Zip solution folder to a zip file
- File: FullName_QuizApp_Task_03_v1.0.zip

Estimated Time: 180 minutes.

Task 4: Authentication/Authorization, Logging, and Security

Description:

This task focuses on implementing authentication/authorization mechanisms, enabling logging for application events, and addressing security considerations within the Quiz Application.

Function Requirements:

- **Authentication/Authorization for API Endpoints:**
 - Protected Endpoint (Example)
 - Route: /api/quizzes
 - Method: GET
 - Description: Endpoint to fetch quizzes.
 - Authorization: Bearer Token (JWT token obtained after login)
 - Response: Success (200 OK): List of quizzes
 - Permission table:

Controllers	Endpoint	Admin	Editor	User
User	Get all	yes	yes	no
	Get by id	yes	yes	no
	Change password	yes	yes	yes
	Create	yes	no	no
	Update	yes	yes	no
	Delete	yes	no	no
Role	Get all	yes	yes	no
	Get by id	yes	yes	no
	Create	yes	no	no
	Update	yes	yes	no
	Delete	yes	no	no
Quiz	Get all	yes	yes	yes
	Get by id	yes	yes	yes
	Create	yes	yes	no
	Update	yes	yes	no
	Delete	yes	yes	no
Question	Get all	yes	yes	yes
	Get by id	yes	yes	yes

	Create	yes	yes	no
	Update	yes	yes	no
	Delete	yes	yes	no

- **Logging Configuration:**

- Integrate Serilog Frameworks
 - Choose Serilog to handle logging in your ASP.NET Core application.
 - Install the required NuGet packages (Serilog.AspNetCore, Serilog.Extensions.Logging, etc.) and set up the logger in your application startup.
- Configure Logging Sinks
 - Define logging sinks to specify where log messages should be written (e.g., file system, database, console).
 - Common sinks include:
 - File Logging: Configure Serilog or NLog to write log messages to text files on disk.
 - Database Logging (**Optional**): Use Serilog sinks like Serilog.Sinks.MSSqlServer to persist logs to a database table.
- Logging Levels and Enrichers
 - Configure logging levels (e.g., Information, Warning, Error) to control which messages are logged.
 - Use enrichers to add additional context to log messages (e.g., timestamp, user information).
- Exception Handling
 - Implement structured logging for exceptions to capture detailed information about errors.
 - Log exceptions at appropriate levels (e.g., Error) along with stack traces and relevant context.

- **Security Considerations:**

- Input Validation:
 - Apply model validation attributes ([Required], [MaxLength], etc.) to API input models.
 - Use ModelState.IsValid to check model validity and handle validation errors.
- HTTPS Configuration:
 - Redirect HTTP requests to HTTPS using middleware (app.UseHttpsRedirection()).
 - (**Optional**) Obtain an SSL certificate from a certificate authority (CA) or use a self-signed certificate for development.
 - (**Optional**) Configure Kestrel to use HTTPS in Program.cs or Startup.cs.
- (**Optional**) Implement security best practices such as input validation, data sanitization, and protection against common vulnerabilities (SQL injection, XSS attacks).

Hints:

- Use ASP.NET Core Identity for user authentication and role-based authorization.
- Configure JWT authentication for stateless, token-based authentication with custom token validation logic.
- Integrate logging frameworks into the ASP.NET Core application to capture and store log messages.

Business Rules:

- Authentication/authorization mechanisms should restrict access to API endpoints based on user roles and permissions.
- Logging should capture important application events and user actions for auditing and troubleshooting.

- Security practices should be implemented to protect against common security threats and vulnerabilities.

Evaluation Criteria:

- Implementation of authentication middleware to secure API endpoints using ASP.NET Core Identity with JWT.
- Configuration of logging framework to capture application events and user actions.
- Adherence to security best practices to prevent security vulnerabilities (e.g., input validation, HTTPS/TLS usage).

Submission file:

- Zip solution folder to a zip file
- File: FullName_QuizApp_Task_04_v1.0.zip

Estimated Time: 180 minutes.

Task 5: Documenting API with Swagger, Versioning, and Deployment**Description:**

This task focuses on documenting RESTful APIs using Swagger/OpenAPI, implementing API versioning, and exploring deployment strategies for ASP.NET Core applications.

Function Requirements:

- **API Documentation with Swagger/OpenAPI:**
 - Integrate the Swashbuckle.AspNetCore library to generate API documentation.
 - Use Swagger/OpenAPI annotations (e.g., [SwaggerOperation], [SwaggerResponse]) to describe API endpoints, request parameters, and response schemas.
 - Configure Swagger UI to provide an interactive documentation interface for developers.
- **API Versioning:**
 - Implement API versioning to manage backward compatibility and support evolving API designs.
 - Choose a versioning strategy (e.g., URL-based versioning, header-based versioning) and apply it to controllers and actions.
 - Include version information in API responses and request headers for version negotiation.
- **Deployment Strategies:**
 - Explore deployment options for ASP.NET Core applications (**IIS Windows Server**, Docker, Azure App Service).
 - Configure environment-specific settings (e.g., connection strings, API keys) for deployment.
 - Discuss considerations for scalability, performance, and maintenance in different deployment environments.

Hints:

- Use the Swashbuckle.AspNetCore NuGet package to integrate Swagger/OpenAPI into the ASP.NET Core project.

- Apply versioning attributes (e.g., [ApiVersion], [ApiVersioning]) to controllers and actions for version control.

Business Rules:

- API documentation should accurately describe endpoints, request parameters, and response payloads using Swagger/OpenAPI annotations.
- API versioning should be implemented to manage changes and maintain backward compatibility for consumers.

Evaluation Criteria:

- Successful integration of Swagger/OpenAPI for API documentation with interactive Swagger UI.
- Correct implementation of API versioning using chosen versioning strategy (e.g., URL-based, header-based).
- Compliance with best practices for API documentation, versioning, and deployment strategies.

Estimated Time: 180 minutes.

Estimated Time for Each Task:

- Task 1: 180 minutes
- Task 2: 180 minutes
- Task 3: 180 minutes
- Task 4: 180 minutes
- Task 5: 180 minutes

Mark Scale:

OOP design	10%	Function requirements	60%
Business rules	15%	Main function	15%