**Web API Project Documentation: C# .NET.**

**Prepared By:**

Aurelie Zened

Caren El-Hachem

Contents

[**1.** **Introduction** 1](#_Toc171421604)

[**2.** **Setting Up the Development Environment** 1](#_Toc171421605)

[**Creating a New Web API Project** 1](#_Toc171421606)

[**Adding Dependencies and Packages** 1](#_Toc171421607)

[**3.** **Project Structure** 2](#_Toc171421608)

[**Console application.** 2](#_Toc171421609)

[**Business: Class library.** 2](#_Toc171421610)

[**Common:** **Class library.** 2](#_Toc171421611)

[**Data:** **Class library.** 2](#_Toc171421612)

[**4.** **APIs.** 3](#_Toc171421613)

# **Introduction**

Overview of the project:

* Setting up the development environment and creating a new .NET Web API project.
* Designing RESTful API endpoints and implementing the controllers to handle various ChatBot interactions.
* Integrating with a database to store and retrieve data necessary data.
* Implementing business logic for the interactions.
* Ensuring error handling and logging.

# **Setting Up the Development Environment**

Before we begin, it is essential to ensure that we have the necessary tools and software installed on our development machine:

1. **Visual Studio 2022 or later:** The integrated development environment (IDE) used for .NET development.
2. **.NET 6 SDK:** The software development kit for building and running .NET applications.
3. **PostgreSQL Server:** The database server for storing data.

## **Creating a New Web API Project**

1. Launch Visual Studio and select “Create a new project”, then select “ASP.NET Core Web API” template.
2. Configure the project with the following details:

* **Project Name:** e.g., Chat\_Bot
* **Location:** Choose a directory to save your project
* Click “Create” to generate the project.

## **Adding Dependencies and Packages**

Right-click on the solution in the Solution explorer and select “Manage NuGet Packages”, then search and install the following packages:

* Microsoft.EntityFrameworkCore
* Microsoft.EntityFrameworkCore.SqlServer
* Microsoft.EntityFrameworkCore.Design
* Microsoft.EntityFrameworkCore.Tools: to be able to use Migrations.
* NLog: used for loggings.
* NLog.Extensions.Logging
* NLog.Web.AspNetCore
* Npgsql.EntityFrameworkCore.PostgreSQL
* Swashbuckle.AspNetCore: For API documentation with Swagger

Add any additional packages that your project may require as it evolves.

# **Project Structure**

## **Console application.**

* **Controller:** Handles incoming HTTP requests and defines API endpoints for interacting with functionalities.
* **NLog.config:** Configuration file for NLog, a logging framework for .NET applications. It will specify logging targets, rules and other configurations for logging events in the application.
* **Appsettings.json:** Stores app settings such as database connection strings.
* **Program.cs:** Entry point for the app, responsible for configuring and starting the .NET Core app, typically containing dependency injection.

## **Business: Class library.**

* **Interfaces:** Define operations or capabilities that business logic services must provide.
* **Services:** Concrete implementation of interfaces.

Example: **‘IChatService’** with **‘ChatService’** for the business logic of interactions.

## **Common:** **Class library.**

* **Configuration folder:** Used for configurations like JWT class.
* **DTOs folder:** facilitates communication between different layers of the application.
  + **Request folder:** Contains classes used as request, e.g., **‘LoginRequest’**.
  + **Response folder:** Contains classes used as response DTO, structuring data for the client.
* **Helpers folder:** Contains the **‘StatusCodeHelper’** class for generating API response objects with appropriate status codes and messages.
  + **Key Features:**
    - **StatusCodes Dictionary:** Maps integer status codes to their corresponding messages.
    - **Methods:** **‘GetStatusResponse<T>’** method returns a status code, message and data based on the provided status code.
* **Models folder:** Contains data models defining the structure and relationships of core entities within the application. Each model typically represents a database table.

## **Data:** **Class library.**

* **AppDbContext:** Central point for interacting within the database. It inherits from **‘DbContext’** and is configured through dependency injection via its constructor.
  + **‘OnModelCreating’ method:**
    - Defines the mapping between entity classes and database tables using **‘ToTable’**.
    - Specifies primary key for each entity using **‘HasKey’.**
    - Configures relationships between different entities using **‘HasMany’**,**’WithOne’**,**’HasForeignKey’**, etc.
    - Specifies behavior on deletion using **‘OnDelete’**.
* **Interfaces:** Define methods that repositories must implement, providing abstraction.
* **Repositories:** Concrete implementation of interfaces, providing the actual data access logic.

Example: **‘IChatRepository’** with **‘ChatRepository’** for managing interactions with db.

# **APIs.**

The **‘[Route("api/[controller]")]’** attribute defines the routing pattern for the controller. It specifies that the routes for this controller will start with "api/" followed by the name of the controller, dynamically replacing **‘[controller]’** with the actual controller name.

The **‘[ApiController]’** attribute enhances the controller by ensuring automatic model validation, and returning appropriate HTTP response codes for various outcomes, such as 400 Bad Request for validation errors and 204 No Content for successful actions that do not return data.

When an API endpoint is called, the controller method first processes the request. It then calls a service layer, which contains the business logic of the application. The service layer acts as an intermediary between the controller and the data access layer (repository). The service performs tasks such as data validation, applying business rules, and preparing data for storage or retrieval.

The repository layer is responsible for directly interacting with the database. It contains methods for creating, reading, updating, and deleting data (CRUD operations). The service layer calls the appropriate repository methods to perform these operations. Once the operation is completed, the service returns the result to the controller, which then sends the appropriate HTTP response back to the client.

Example Flow:

1. **Controller Method**: Receives the request and calls the service method.
2. **Service Method**: Validates and processes the request, then calls the repository method.
3. **Repository Method**: Performs the database operation and returns the result to the service.
4. **Service Method**: Returns the processed result to the controller.
5. **Controller Method**: Sends the final response to the client.