Creating APIs Using .Net Core and SQL Server

A Beginner's Guide to Building APIs with C#

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Introduction

Our primary focus will be on utilizing .NET Core to build robust APIs that interact with a SQL Server database, adhering to industry-standard practices and patterns. We'll follow a structured approach that includes using Entity Framework (EF) with a Database First approach, implementing a repository class to manage data operations, and defining controllers to handle HTTP methods. Let's break down each of these components and their roles in our application.

Storing Data in SQL Server

SQL Server will serve as our relational database management system. It's a choice well-suited for enterprises due to its scalability, security features, and robust management tools. SQL Server will host the underlying data for our APIs, which could include tables for entities such as patients, doctors, appointments, and medical records in our healthcare system example.

Using Entity Framework with Database First Approach

Entity Framework (EF) is an object-relational mapper (ORM) that enables .NET developers to work with a database using .NET objects, eliminating the need for most of the data-access code that developers usually need to write. We will use the Database First approach, which involves creating our database schema in SQL Server first and then generating the corresponding data models in our .NET Core application based on that schema.

Implementing a Repository Class

The repository class will act as a mediator between the data access layer and the business logic layer of the application. It will use the DbContext from Entity Framework to query and save data to the database.

Controllers for HTTP Methods

Controllers in the MVC (Model-View-Controller) pattern of .NET Core handle incoming HTTP requests and send responses back to the client. Each controller will manage the routing of API requests to appropriate actions, based on the HTTP methods such as GET, POST, PUT, and DELETE. The controllers will leverage the repository to perform CRUD (Create, Read, Update, Delete) operations on the database via EF. The use of controllers helps to ensure a clean separation of concerns and facilitates the maintenance and scalability of the application.

Build the Database

Before creating the database, it is crucial to identify and define the requirements for the entities and their relationships based on the application's needs. In our healthcare system example, we need the following entities:

- **Patients**: Store personal details and medical history.
- **Doctors**: Include information about specializations and schedules.

Design the Schema

Use a tool like SQL Server Management Studio (SSMS) or a diagram tool to design the database schema. Create tables for each entity and define the relationships (e.g., one-to-many, many-to-many) between them. Below are the SQL commands to create each table:

Patients Table

```
CREATE TABLE Patients (
    PatientID int IDENTITY(1,1) PRIMARY KEY,
    Name nvarchar(255) NOT NULL,
    DateOfBirth datetime NOT NULL,
    Address nvarchar(255) NOT NULL,
    MedicalHistory nvarchar(MAX)
);
```

Doctors Table

```
CREATE TABLE Doctors (
    DoctorID int IDENTITY(1,1) PRIMARY KEY,
    Name nvarchar(255) NOT NULL,
    Specialization nvarchar(255) NOT NULL,
    Schedule nvarchar(255) NOT NULL
);
```

Populate Sample Data

Here are SQL `INSERT` statements for populating the *Patients*, *Doctors*, *Appointments*, and *Medical Records* tables with five records each. These statements reflect realistic yet hypothetical data suitable for your healthcare system.

Patients Table

```
INSERT INTO Patients (Name, DateOfBirth, Address, MedicalHistory)
   VALUES
('John Doe', '1980-05-15', '1234 Elm St', 'Asthma'),
('Jane Smith', '1992-07-04', '5678 Oak St', 'No known allergies'),
('Alice Johnson', '1975-12-11', '9101 Pine St', 'Diabetic'),
('Robert Brown', '1988-10-23', '2342 Maple St', 'Hypertension'),
('Maria Garcia', '2000-03-30', '3456 Spruce St', 'Previous surgery on
   right knee');
```

Doctors Table

```
INSERT INTO Doctors (Name, Specialization, Schedule) VALUES
('Dr. Sarah Lee', 'Cardiology', 'M-F 9AM-5PM'),
('Dr. Emily White', 'Orthopedics', 'M-F 8AM-4PM'),
('Dr. James Black', 'Pediatrics', 'M-F 10AM-6PM'),
('Dr. Jennifer Green', 'Dermatology', 'T-Th 9AM-3PM'),
('Dr. William Davis', 'Neurology', 'M-W-F 8AM-2PM');
```

Create the Project

Open Visual Studio 2022. On the start window, select **Create a new project** to begin the setup process for a new .NET Core application.

1. In the **Create a new project** window, you will see a variety of project templates.
2. Type "API" into the search bar to filter the options.
3. Select **ASP.NET Core Web API**. This template is specifically designed for creating API projects in .NET Core.
4. Click **Next** to continue.
5. In the **Configure your new project** window:
   - **Project Name**: Enter a name for your project, such as *HealthcareApi*.
   - **Location**: Choose or create a directory where your project files will be stored.
   - **Solution Name**: Typically, this is the same as the project name unless you plan to add multiple projects to the same solution.

Build and Run the Project

1. Build the project by selecting **Build** > **Build Solution** from the menu.
2. Run your project by pressing **F5** or clicking the **Start Debugging** button.

Visual Studio will start the application, and a web browser window should open displaying a default API landing page or a Swagger UI page, if included.

Set Up Entity Framework (Database First)

Install Required NuGet Packages

Install the necessary packages using the Package Manager Console:

```
Install-Package Microsoft.EntityFrameworkCore
Install-Package Microsoft.EntityFrameworkCore.SqlServer
Install-Package Microsoft.EntityFrameworkCore.Tools
```

Generate Models and DbContext

Use the `Scaffold-DbContext` command in the Package Manager Console to generate the models and DbContext based on your existing database schema:

```
Scaffold-DbContext "Server=(SERVERNAME); Database=DATABASENAME;
    Trusted_Connection=True; TrustServerCertificate=True;" Microsoft.
    EntityFrameworkCore.SqlServer -OutputDir Models
```

Configure the Connection String

Update the `appsettings.json` file to ensure it contains the correct connection string for your database:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=SERVERNAME;Database=DATABASENAME;
    Trusted_Connection=True;TrustServerCertificate=True;"
  }
}
```

Update OnConfiguring Method

Remove the connection string from the `OnConfiguring` method in your DbContext:

```
protected override void OnConfiguring(DbContextOptionsBuilder
    optionsBuilder){}
```

Register the DbContext

Modify `Program.cs` to register the DbContext in the service container:

```
builder.Services.AddDbContext<CONTEXTNAME>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("
    DefaultConnection")));
```

Create the DAL

Adding a DAL Folder and Creating DoctorRepository Class

Add a DAL folder to the solution and create a new class `DoctorRepository`. Next, inject the `DbContext` into the constructor:

```
private readonly CONTEXTNAME _context;

public DoctorRepository(CONTEXTNAME context)
{
    _context = context;
}
```

Creating Repository Methods

Implement the repository methods to manage data operations:

```
public List<Doctor> GetAllDoctors()
{
    return _context.Doctors.ToList();
}

public Doctor GetDoctorById(int doctorId)
{
    return _context.Doctors.FirstOrDefault(d => d.DoctorId == doctorId)
   ;
}

public void AddDoctor(Doctor doctor)
{
    _context.Doctors.Add(doctor);
    _context.SaveChanges();
}

public void UpdateDoctor(Doctor doctor)
{
    _context.Doctors.Update(doctor);
    _context.SaveChanges();
}

public void DeleteDoctor(int doctorId)
{
    var doctor = _context.Doctors.FirstOrDefault(d => d.DoctorId ==
   doctorId);
    if (doctor != null)
    {
        _context.Doctors.Remove(doctor);
        _context.SaveChanges();
    }
}
```

Explanation of Repository Methods

- **GetAllDoctors()**: Retrieves all doctors from the Doctors table.
- **GetDoctorById(int doctorId)**: Fetches a single doctor based on the provided DoctorID.
- **AddDoctor(Doctor doctor)**: Adds a new doctor to the database and commits the change.
- **UpdateDoctor(Doctor doctor)**: Updates the details of an existing doctor and commits the change.
- **DeleteDoctor(int doctorId)**: Removes a doctor from the database using the provided DoctorID and commits the change.

Creating the Doctor Controller

To create the `DoctorController` in your .NET Core API project, follow these steps carefully:

1. Right-click on the `Controllers` folder in your project's Solution Explorer. If there isn't a `Controllers` folder, you need to create one:
   - Right-click on the project's root directory in the Solution Explorer.
   - Select `Add`, then choose `New Folder`.
   - Name the new folder `Controllers`.
2. Inside the `Controllers` folder, right-click and choose `Add` > `Controller`.
3. From the list of templates, select `API Controller - Empty` and click `Add`.
4. Name the controller `DoctorsController` and confirm by clicking `Add`.

This procedure will set up an empty API controller named `DoctorController` within your project, prepared for further development and integration with the data access layer.

Add Namespace References

Ensure that your controller file includes the necessary using statements to properly reference the `DoctorRepository` and model:

```
using Microsoft.AspNetCore.Mvc;
// Replace with your actual namespace
using YourProjectNamespace.Models;
// Assuming DoctorRepository is within the DAL folder
using YourProjectNamespace.DAL;
```

Integrating DoctorRepository with Dependency Injection

To integrate the `DoctorRepository` into your .NET Core application using Dependency Injection (DI), you need to register it in the DI container. This registration is typically done in the `Program.cs` file for .NET 6 and later versions, or in the `Startup.cs` file for earlier versions.

1. Open the `Program.cs` file in your .NET 6 (or later) project.
2. Add the namespace for `DoctorRepository` if it's not in the same namespace:

```
// Adjust to match your actual namespace
using YourProjectNamespace.DAL;
```

3. Register the `DoctorRepository` in the DI container. This is typically done along with other services configuration. Insert this line in the part where services are being added to the container:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
// Register your repository
builder.Services.AddScoped<DoctorRepository>();
```

Understanding the Dependency Injection Code

This command tells the .NET Core Dependency Injection container to manage the lifecycle of `DoctorRepository` instances using the "Scoped" lifetime.

What does Scoped mean? When you register a service as `Scoped`, it means that a new instance of the service will be created for each request within the application. This is particularly useful for services like data access layers, where you want to maintain a consistent state throughout the execution of a single request, but not beyond it.

In practical terms, registering `DoctorRepository` as `Scoped` ensures that:

- Every incoming HTTP request to the server has its own unique instance of `DoctorRepository`.
- All operations performed by a specific request will use the same instance of `DoctorRepository`, promoting consistency.
- Memory and resource management are optimized because each request cleans up its `DoctorRepository` instance when the request ends, preventing issues like memory leaks or data conflicts across concurrent requests.

This setup is ideal for web applications where each request might have different data context needs or where operations within a request need to be tightly controlled and isolated from other requests.

Inject DoctorRepository into the Controller

Modify the `DoctorsController` class to inject the `DoctorRepository`. This involves adding a private field for the repository and a constructor to initialize this field:

```
public class DoctorsController : ControllerBase
{
    private readonly DoctorRepository _doctorRepository;

    public DoctorsController()
    {
        _doctorRepository = new DoctorRepository();
    }
}
```

Add Get All Doctors Action

Add an action method to retrieve all doctors. This method will utilize the `DoctorRepository` to fetch the data:

```
[HttpGet]
public IActionResult GetAllDoctors()
{
    var doctors = _doctorRepository.GetAllDoctors();
    // Returns a 200 OK response with the doctors data
    return Ok(doctors);
}
```

This setup ensures that the controller is correctly implemented with the necessary references and dependencies. This is essential for maintaining a clean and modular architecture in your .NET Core application, separating concerns and promoting code re-usability.

Testing the API with Swagger

Testing your .NET Core API using Swagger is a straightforward process that allows you to easily interact with the API through a web-based interface. Swagger (also known as OpenAPI) provides a user-friendly way to document and test API endpoints. Here's how to set up and test your `DoctorsController` using Swagger:

Step 1: Enable Swagger in Your .NET Core Application

Before you can test your API, you need to ensure that Swagger is integrated into your application. If it's not already set up, follow these steps to add and configure Swagger:

Add Swagger Middleware If it's not already installed, install the Swagger middleware by adding the NuGet package to your project. You can do this via the NuGet Package Manager in Visual Studio or by using the Package Manager Console:

```
Install-Package Swashbuckle.AspNetCore
```

Configure Swagger in Program.cs Make sure your `Program.cs` file is adding Swagger services:

```
builder.Services.AddScoped<DoctorRepository>();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(); // Adds Swagger
```

Enable Swagger Middleware In the same `Program.cs` file, ensure the middleware is enabled to serve the generated Swagger as a JSON endpoint and the Swagger UI:

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Step 2: Run Your Application

Run your application from Visual Studio by pressing `F5` or clicking on the "Start Debugging" button. Make sure your application is set to run on a web profile and not just the console.

Step 3: Access Swagger UI

Once your application is running, open a web browser and navigate to the root URL of your API (e.g., `https://localhost:5001/` if running locally). If configured as above, the Swagger UI should load automatically and display all the API endpoints defined in your application.

Step 4: Test the GetAllDoctors Endpoint

To test the `GetAllDoctors` endpoint:

Find the Endpoint in Swagger UI Scroll to or search for the `DoctorsController` section in the Swagger UI. You should see the `GET /api/doctors` endpoint listed.

Execute the Request Click on the endpoint in the Swagger UI to expand it, then click the "Try it out" button, followed by the "Execute" button. This sends a request to the `GET /api/doctors` endpoint.

View the Results Swagger will execute the HTTP GET request to your API and display the response. You should see the HTTP status code (200 OK if successful) and the JSON list of doctors returned by your API.

Step 5: Review the Response

Examine the response in the Swagger UI to verify that it contains the expected data. This feedback loop allows you to quickly identify and resolve any issues with the API implementation.

Expanding the DoctorsController with Additional HTTP Methods

After setting up the initial 'GetAllDoctors' method, it's time to extend the 'DoctorsController' so our API can handle individual CRUD operations. These include fetching a doctor by ID, updating a doctor's details, adding a new doctor, and deleting an existing doctor. Below are the implementations for these additional methods

Get Doctor by ID

To retrieve a single doctor by their unique identifier:

```
[HttpGet("{doctorId}")]
public IActionResult GetDoctorById(int doctorId)
{
```

```
    var doctor = _doctorRepository.GetDoctorById(doctorId);
    if (doctor == null)
    {
        return NotFound();
    }
    return Ok(doctor);
}
```

This method uses route data to fetch a doctor's ID and queries the repository. If the doctor is not found, it returns a 404 Not Found status; otherwise, it returns the doctor with a 200 OK status.

Add New Doctor

To streamline the addition of new doctors and ensure that only the necessary data is processed, we employ a Data Transfer Object (DTO). A DTO helps to reduce the risk of over-posting vulnerabilities and clarifies the data requirements of the API.

Defining the DoctorCreationDto

First, define the `DoctorCreationDto` in a separate file within your project's Models or DTOs folder. This DTO will only include properties that are required when creating a new doctor:

```
public class DoctorCreationDto
{
    public string Name { get; set; }
    public string Specialization { get; set; }
    public string Schedule { get; set; }
}
```

This DTO simplifies the model and ensures that the API receives exactly what it needs, no more and no less. This practice maintains the integrity and security of your data.

Add a New Doctor

With the DTO defined, update the API method in the `DoctorsController` to use this DTO instead of directly using the entity model. This method will transform the DTO to the domain model before saving:

```
[HttpPost]
public IActionResult AddDoctor([FromBody] DoctorCreationDto doctorDto)
{
    var doctor = new Doctor
    {
        Name = doctorDto.Name,
        Specialization = doctorDto.Specialization,
        Schedule = doctorDto.Schedule
    };

    _doctorRepository.AddDoctor(doctor);
    return CreatedAtAction(nameof(GetDoctorById), new { doctorId =
   doctor.DoctorId }, doctor);
}
```

This POST method adds a new doctor using the `DoctorCreationDto`. The 'CreatedAtAction' method is used to generate an HTTP 201 Created status code, which also provides a URI to the newly created doctor. Using a DTO not only simplifies the data handling by the API but also adds an additional layer of abstraction, which helps in maintaining clean and manageable code.

This structured approach, employing a DTO, ensures that the API endpoints are secured against over-posting, clarify the input requirements, and facilitate any future changes to the data model without impacting the client-facing contract.

Update Existing Doctor

For updating existing records, it is a good practice to use a separate DTO, such as `UpdateDoctorDto`. This approach provides several advantages:

- **Security**: It prevents over-posting vulnerabilities where a client might inadvertently or maliciously update fields that should not be changed after creation.
- **Flexibility**: Different validation rules can be applied specifically for updates. For example, some fields that are required during creation might be optional during updates or should not be changed at all, like an automatically generated database ID.
- **Clarity**: Separating DTOs for different operations (creating vs. updating) makes the code clearer and easier to maintain, as each DTO can encapsulate only the relevant fields and business logic for its specific operation.

The `UpdateDoctorDto` might exclude certain properties that are immutable after a doctor's record is created, such as an identification number assigned during the creation. Here is how you could define this DTO:

```
public class UpdateDoctorDto
{
    public string Name { get; set; }
    public string Specialization { get; set; }
    public string Schedule { get; set; }
}
```

Implementing the PUT Method Using UpdateDoctorDto

Using the `UpdateDoctorDto`, the API can safely expose only the fields that are allowed to be updated. The following 'PUT' method demonstrates how to use this DTO to update a doctor's details:

```
[HttpPut("{doctorId}")]
public IActionResult UpdateDoctor(int doctorId, [FromBody]
   UpdateDoctorDto updateDto)
{
    var doctor = _doctorRepository.GetDoctorById(doctorId);
    if (doctor == null)
```

```
    {
        return NotFound();
    }

    // Map the DTO to the domain model properties that are allowed to
   be updated
    doctor.Name = updateDto.Name;
    doctor.Specialization = updateDto.Specialization;
    doctor.Schedule = updateDto.Schedule;

    _doctorRepository.UpdateDoctor(doctor);
    return NoContent();  // 204 No Content status is returned after a
   successful update
}
```

This method ensures that updates are performed safely and only on the intended properties. By separating the creation and update DTOs, the API adheres to the principle of least privilege, enhancing both security and maintainability.

This method validates that the ID from the route matches the doctor's ID from the body. If they match and the doctor exists, it updates the doctor and returns a 204 No Content status.

Delete a Doctor

To remove a doctor from the database:

```
[HttpDelete("{doctorId}")]
public IActionResult DeleteDoctor(int doctorId)
{
    var doctor = _doctorRepository.GetDoctorById(doctorId);
    if (doctor == null)
    {
        return NotFound();
    }

    _doctorRepository.DeleteDoctor(doctorId);
    return NoContent();
}
```

This DELETE method checks if the doctor exists and, if so, deletes them. Like updates, a successful deletion typically returns a 204 No Content status.
These methods complete the basic CRUD operations necessary for the 'DoctorsController', allowing it to manage doctor data effectively within the application.

Understanding HTTP Response Types in .NET Core APIs

In .NET Core APIs, controllers often return various types of HTTP responses based on the outcome of the request. Understanding when to use different response types can help in designing more intuitive and standard-compliant APIs. Below are descriptions of common response types:

Ok (200 OK)

The `Ok` method returns a status code of 200, which indicates that the request has succeeded. The `Ok` response is typically used when data is being returned to the client as part of the response body.

NoContent (204 No Content)

The `NoContent` method returns a status code of 204. This response is appropriate when the request has been successfully processed but the response does not need to return any data. It is often used in delete operations or updates where no confirmation data is needed.

NotFound (404 Not Found)

The `NotFound` method returns a status code of 404. This response is used when the requested resource is not found on the server. It is important to provide this feedback to help clients handle errors properly.

CreatedAtAction (201 Created)

The `CreatedAtAction` method returns a status code of 201, which indicates that a new resource has been created. Along with the status code, it also provides a URI to the newly created resource via the Location header. It's commonly used in POST methods that create new entries.

Exercise: Implement CRUD Actions for the Patients Table

Now that you have learned how to implement CRUD operations for the Doctors table using DTOs and best practices for secure API design, it's time to put your skills to the test. Your challenge is to create a complete set of CRUD actions for the Patients table in your .NET Core API. T

Requirements

Your API should include the following functionalities for the Patients table:

- **Create**: Add new patients, ensuring all necessary patient information is included and validated.
- **Read**: Retrieve patient details by ID and also list all patients.
- **Update**: Modify existing patient details. Ensure that only modifiable fields can be updated.
- **Delete**: Remove a patient record when it is no longer needed or upon request.

Hints and Tips

Consider the following pointers to guide your implementation:

- Use DTOs to manage data transfer objects for create and update operations to prevent over-posting vulnerabilities.
- Implement proper validation for patient data to maintain data integrity and comply with healthcare regulations.
- Ensure your API responses are appropriate (e.g., returning `NotFound` for non-existent patient IDs).