Vinod Kumar Kayartaya vinod@vinod.co

Building REST APIs with .NET Core

REST APIs (Representational State Transfer APIs) are a popular way to build web services that allow clients (e.g., web apps, mobile apps) to interact with server-side data using standard HTTP methods (GET, POST, PUT, DELETE). ASP.NET Core, part of the .NET ecosystem, provides a robust framework for creating RESTful APIs. This material covers setting up a project and implementing CRUD (Create, Read, Update, Delete) operations with data validation, using an in-memory data structure.

---

1. Overview of REST APIs in .NET Core

What is ASP.NET Core Web API?

- ASP.NET Core Web API is a framework for building HTTP-based services in .NET Core/.NET.
- It supports REST principles: stateless operations, resource-based URLs, and standard HTTP methods.
- Features include routing, model binding, dependency injection, and built-in support for JSON.

Why Use In-Memory Data?

- For simplicity and learning, we'll use a `List<T>` instead of a database. This avoids external dependencies while focusing on API mechanics.

---

2. Setting Up an ASP.NET Core Web API Project

Prerequisites

- **.NET SDK**: Installed on your machine (e.g., .NET 8 as of March 2025). Download from dotnet.microsoft.com.
- **IDE**: Visual Studio, VS Code, or any text editor (optional but recommended).
- **Command Line**: For .NET CLI commands.

Steps to Create the Project

1. **Create a New Project** Open a terminal and run:

   ```
   dotnet new webapi -o MyApiProject
   cd MyApiProject
   ```

   - `webapi`: Template for a minimal Web API project.
   - `-o MyApiProject`: Output directory name.

2. **Explore the Project Structure**

   - `Program.cs`: Main entry point, configures the app and services.
   - `appsettings.json`: Configuration settings.
   - `Controllers/WeatherForecastController.cs`: A sample controller (we'll replace this).

3. **Run the Project**

```
dotnet run
```

- Launches the API at `https://localhost:5001` (or a similar port).
- Test the default endpoint (`/weatherforecast`) using a browser or a tool like Postman.

Clean Up the Template

- Delete `WeatherForecastController.cs` and `WeatherForecast.cs` (we'll create our own).

---

3. Implementing CRUD Operations

Model Definition

We'll create a simple `Product` model to represent our data.

```
// Models/Product.cs
namespace MyApiProject.Models
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
}
```

In-Memory Data Store

We'll use a static `List<Product>` as our data store.

```
// Services/ProductService.cs
namespace MyApiProject.Services
{
    public class ProductService
    {
        private static List<Product> _products = new List<Product>
        {
            new Product { Id = 1, Name = "Laptop", Price = 999.99m },
            new Product { Id = 2, Name = "Mouse", Price = 19.99m }
        };

        public List<Product> GetAll() => _products;

        public Product GetById(int id) => _products.FirstOrDefault(p => p.Id ==
id);
```

```
        public void Add(Product product)
        {
            product.Id = _products.Max(p => p.Id) + 1; // Simple ID generation
            _products.Add(product);
        }

        public bool Update(int id, Product product)
        {
            var existing = GetById(id);
            if (existing == null) return false;
            existing.Name = product.Name;
            existing.Price = product.Price;
            return true;
        }

        public bool Delete(int id)
        {
            var product = GetById(id);
            if (product == null) return false;
            _products.Remove(product);
            return true;
        }
    }
}
```

Register the Service

In `Program.cs`, add the service to the dependency injection container:

```
// Program.cs
var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddSingleton<ProductService>(); // Register as singleton

var app = builder.Build();

// Configure the HTTP request pipeline
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Create the API Controller

Define a controller to handle CRUD operations.

```
// Controllers/ProductsController.cs
using Microsoft.AspNetCore.Mvc;
using MyApiProject.Models;
using MyApiProject.Services;

namespace MyApiProject.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class ProductsController : ControllerBase
    {
        private readonly ProductService _productService;

        public ProductsController(ProductService productService)
        {
            _productService = productService;
        }

        // GET: api/products
        [HttpGet]
        public ActionResult<List<Product>> GetAll()
        {
            return Ok(_productService.GetAll());
        }

        // GET: api/products/1
        [HttpGet("{id}")]
        public ActionResult<Product> GetById(int id)
        {
            var product = _productService.GetById(id);
            if (product == null) return NotFound();
            return Ok(product);
        }

        // POST: api/products
        [HttpPost]
        public ActionResult<Product> Create([FromBody] Product product)
        {
            _productService.Add(product);
            return CreatedAtAction(nameof(GetById), new { id = product.Id },
product);
        }

        // PUT: api/products/1
        [HttpPut("{id}")]
        public ActionResult Update(int id, [FromBody] Product product)
        {
            if (id != product.Id) return BadRequest();
            if (!_productService.Update(id, product)) return NotFound();
            return NoContent();
        }

        // DELETE: api/products/1
        [HttpDelete("{id}")]
        public ActionResult Delete(int id)
        {
            if (!_productService.Delete(id)) return NotFound();
            return NoContent();
        }
    }
}
```

HTTP Methods and Routes

Method	Route	Action
GET	`api/products`	Retrieve all products
GET	`api/products/{id}`	Retrieve one product
POST	`api/products`	Create a product
PUT	`api/products/{id}`	Update a product
DELETE	`api/products/{id}`	Delete a product

4. Data Validation

Adding Validation Rules

Use **data annotations** from `System.ComponentModel.DataAnnotations` to enforce rules on the `Product` model.

```
// Models/Product.cs
using System.ComponentModel.DataAnnotations;

namespace MyApiProject.Models
{
    public class Product
    {
        public int Id { get; set; }

        [Required(ErrorMessage = "Name is required")]
        [StringLength(50, MinimumLength = 2, ErrorMessage = "Name must be between
2 and 50 characters")]
        public string Name { get; set; }

        [Range(0.01, 10000.00, ErrorMessage = "Price must be between 0.01 and
10000.00")]
        public decimal Price { get; set; }
    }
}
```

Handling Validation in the Controller

ASP.NET Core automatically validates models and returns errors if they fail.

Update the `Create` Action

```
[HttpPost]
public ActionResult<Product> Create([FromBody] Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState); // Returns validation errors
    }
    _productService.Add(product);
    return CreatedAtAction(nameof(GetById), new { id = product.Id }, product);
}
```

Update the `Update` Action

```
[HttpPut("{id}")]
public ActionResult Update(int id, [FromBody] Product product)
{
    if (id != product.Id) return BadRequest();
    if (!ModelState.IsValid) return BadRequest(ModelState);
    if (!_productService.Update(id, product)) return NotFound();
    return NoContent();
}
```

Testing Validation

- **Invalid POST Request** (e.g., missing `Name`):

  ```
  {
      "price": 10.99
  }
  ```

  Response (HTTP 400):

  ```
  {
      "errors": {
          "Name": ["Name is required"]
      },
      "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
      "title": "One or more validation errors occurred.",
  ```

```
    "status": 400
}
```

---

5. Running and Testing the API

Run the Project

```
dotnet run
```

- API runs at `https://localhost:5001` (or similar).

Test with Tools

Use **Postman**, **cURL**, or a browser:

1. **GET All**: `GET https://localhost:5001/api/products`
   - Response: List of products in JSON.
2. **POST**: `POST https://localhost:5001/api/products`
   - Body: `{"name": "Keyboard", "price": 49.99}`
   - Response: 201 Created with the new product.
3. **GET by ID**: `GET https://localhost:5001/api/products/3`
4. **PUT**: `PUT https://localhost:5001/api/products/3`
   - Body: `{"id": 3, "name": "Updated Keyboard", "price": 59.99}`
5. **DELETE**: `DELETE https://localhost:5001/api/products/3`

---

6. Practical Application Example

This API manages a simple product catalog. The in-memory `ProductService` simulates a data layer, and validation ensures data integrity. Here's the full flow:

- Start with pre-populated products.
- Use POST to add a new product.
- Use GET to retrieve products.
- Use PUT to update a product (with validation).
- Use DELETE to remove a product.

Summary Table

Topic	Description	Key Feature
Project Setup	Create Web API with .NET CLI	`dotnet new webapi`
CRUD Operations	Implement GET, POST, PUT, DELETE	HTTP methods, controller actions
Data Validation	Enforce rules on model	Data annotations, `ModelState`

Topic	Description	Key Feature
In-Memory Data	Use `List<T>` instead of a database	Simple, no external dependency

Exercises

1. Add a new endpoint `GET api/products/search?name={query}` to filter products by name (case-insensitive).
2. Extend the `Product` model with a `Category` property and add validation for it (e.g., required, max length 30).
3. Modify the API to return a custom error message when a product isn't found (instead of just 404).