ASP.NET Core - True Ultimate Guide

Section 26: Asp.Net Core Web API - Notes

ASP.NET Core Web API and RESTful Principles

1. Understanding Web API

ASP.NET Core Web API allows developers to create HTTP services that can be consumed by a variety of clients, including browsers, mobile applications, and other services. Web APIs are lightweight, stateless, and can be consumed by any device capable of making HTTP requests.

Key Concepts:

- **Stateless Communication**: Each request from a client must contain all the information needed for the server to understand and process the request.
- **JSON/XML Format**: ASP.NET Core Web API primarily communicates using JSON, but it can also support other formats like XML.

2. RESTful Architecture

REST (Representational State Transfer) is an architectural style that defines constraints and principles to create web services. RESTful APIs follow these principles to ensure scalability, simplicity, and performance.

RESTful Principles:

- **Client-Server Architecture**: Separation of concerns between the client and server. The client is responsible for the user interface, while the server handles data storage and processing.
- **Statelessness**: Every interaction between the client and server is independent. The server does not store the client's state.
- **Cacheability**: Responses must explicitly indicate whether caching is allowed. Caching improves performance by reducing the need to repeat requests.
- **Uniform Interface**: All requests are made to a single, well-defined interface using standard HTTP methods such as GET, POST, PUT, and DELETE.
- **Resource Identification**: Resources (data) are identified using URIs (Uniform Resource Identifiers), and they are acted upon using HTTP methods.

3. RESTful Constraints in Web API

- **Resources and URIs**: REST revolves around resources, which can be anything such as a user, product, or order. A resource is identified by a URI.

Example:

GET /api/products/12345

In this example, the resource is product and 12345 is the identifier (ID) for a specific product.

- **Statelessness**: Each HTTP request contains all necessary information, such as headers, request body, etc. The server doesn't store any session data.
- **HTTP Methods in REST**:
    - GET: Retrieve data.
    - POST: Create a new resource.
    - PUT: Update an existing resource.
    - DELETE: Remove a resource.

4. Implementing a Simple Web API with REST Principles

To get started, we'll implement a simple API following RESTful principles. In this example, we'll build a ProductsController to handle a list of products.

Step 1: Create a New ASP.NET Core Web API Project

dotnet new webapi -n ProductApi

Step 2: Define the Product Model In ASP.NET Core, resources are often represented by models. For our API, a Product model might look like this:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Description { get; set; }
}
```

Step 3: Create a ProductsController Controllers in ASP.NET Core Web API are the entry points for handling HTTP requests.

```
[Route("api/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

   private static List<Product> products = new List<Product>

   {

      new Product { Id = 1, Name = "Laptop", Price = 999.99m, Description = "A high-performance laptop" },

      new Product { Id = 2, Name = "Smartphone", Price = 499.99m, Description = "A flagship smartphone" },

   };


   // GET: api/products

   [HttpGet]

   public ActionResult<IEnumerable<Product>> GetProducts()

   {

      return products;

   }


   // GET: api/products/1

   [HttpGet("{id}")]

   public ActionResult<Product> GetProduct(int id)

   {

      var product = products.FirstOrDefault(p => p.Id == id);

      if (product == null)

      {

         return NotFound();

      }

      return product;

   }
```

```
// POST: api/products
[HttpPost]
public ActionResult<Product> CreateProduct(Product newProduct)
{
   newProduct.Id = products.Max(p => p.Id) + 1;
   products.Add(newProduct);
   return CreatedAtAction(nameof(GetProduct), new { id = newProduct.Id }, newProduct);
}

// PUT: api/products/1
[HttpPut("{id}")]
public IActionResult UpdateProduct(int id, Product updatedProduct)
{
   var product = products.FirstOrDefault(p => p.Id == id);
   if (product == null)
   {
      return NotFound();
   }

   product.Name = updatedProduct.Name;
   product.Price = updatedProduct.Price;
   product.Description = updatedProduct.Description;

   return NoContent();  // 204 No Content
}

// DELETE: api/products/1
[HttpDelete("{id}")]
public IActionResult DeleteProduct(int id)
{
```

```
        var product = products.FirstOrDefault(p => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }

        products.Remove(product);
        return NoContent();
    }
}
```

Explanation of Code:

1. **[Route("api/[controller]")]**: This defines the base route for the controller. api/products would map to ProductsController.
2. **[HttpGet]**: Handles HTTP GET requests. GetProducts() returns all products, and GetProduct(int id) retrieves a specific product based on its ID.
3. **[HttpPost]**: Handles HTTP POST requests. CreateProduct() adds a new product to the collection.
4. **[HttpPut("{id}")]**: Handles HTTP PUT requests to update an existing product.
5. **[HttpDelete("{id}")]**: Handles HTTP DELETE requests to remove a product.

5. HTTP Status Codes

In a RESTful API, it's essential to return proper HTTP status codes that indicate the result of the operation:

- **200 OK**: The request was successful (e.g., for GET requests).
- **201 Created**: A resource was successfully created (e.g., for POST requests).
- **204 No Content**: The request was successful, but there is no content to return (e.g., for PUT or DELETE requests).
- **404 Not Found**: The resource was not found (e.g., when requesting a product that doesn't exist).
- **400 Bad Request**: The request was malformed or invalid.

Key Points to Remember:

- ASP.NET Core Web API allows building lightweight, stateless HTTP services that can be consumed by various clients.
- REST is an architectural style that emphasizes scalability, simplicity, and performance through statelessness and resource identification.
- HTTP methods (GET, POST, PUT, DELETE) define operations on resources, and each method has a specific use in REST.
- Controllers in ASP.NET Core Web API handle incoming requests and map them to actions that perform business logic.
- Always return appropriate HTTP status codes to inform clients about the result of their request.

Web API Controllers

In ASP.NET Core, **Web API Controllers** serve as the backbone for handling HTTP requests. Controllers define a set of actions (methods) that respond to HTTP verbs like GET, POST, PUT, and DELETE. These actions interact with the data model and return responses to the client.

1. What is a Web API Controller?

A Web API controller is a class that handles HTTP requests and generates appropriate HTTP responses. It inherits from the ControllerBase class (which we'll cover later) and uses attributes to define routing, HTTP methods, and input/output handling.

Basic Structure of a Web API Controller:

```
[ApiController]

[Route("api/[controller]")]

public class ProductsController : ControllerBase

{

   // Example action methods

}
```

- **[ApiController]**: This attribute makes it easier to build a Web API by handling things like automatic model state validation, binding, and responses.
- **[Route]**: Defines the URI path for the controller. Here, [controller] will automatically be replaced by the controller's name (e.g., ProductsController → api/products).

2. Action Methods in Web API Controllers

Action methods in a controller handle specific HTTP requests (GET, POST, etc.) and correspond to operations on resources.

Example: CRUD Operations for a Products API

Here's a more detailed example of how each action method maps to an HTTP verb:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private static List<Product> products = new List<Product>();

    // GET: api/products
    [HttpGet]
    public ActionResult<IEnumerable<Product>> GetProducts()
    {
        return Ok(products); // Return 200 OK with product list
    }

    // GET: api/products/1
    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = products.FirstOrDefault(p => p.Id == id);
        if (product == null)
        {
            return NotFound(); // Return 404 if not found
        }
        return Ok(product); // Return 200 OK with the product
    }

    // POST: api/products
```

```
[HttpPost]
public ActionResult<Product> CreateProduct(Product newProduct)
{
    newProduct.Id = products.Count + 1;
    products.Add(newProduct);
    return CreatedAtAction(nameof(GetProduct), new { id = newProduct.Id }, newProduct);
}

// PUT: api/products/1
[HttpPut("{id}")]
public IActionResult UpdateProduct(int id, Product updatedProduct)
{
    var product = products.FirstOrDefault(p => p.Id == id);
    if (product == null)
    {
        return NotFound();
    }

    product.Name = updatedProduct.Name;
    product.Price = updatedProduct.Price;
    product.Description = updatedProduct.Description;

    return NoContent(); // 204 No Content, indicating success
}

// DELETE: api/products/1
[HttpDelete("{id}")]
public IActionResult DeleteProduct(int id)
{
    var product = products.FirstOrDefault(p => p.Id == id);
    if (product == null)
```

```
        {
            return NotFound();
        }

        products.Remove(product);
        return NoContent();
    }
}
```

Explanation of the Code:

1. **[HttpGet]**:
    - The GetProducts() method responds to HTTP GET requests to api/products.
    - The GetProduct(int id) method responds to GET requests for a specific product (api/products/1).
    - Returns the products or a 404 response if the product isn't found.
2. **[HttpPost]**:
    - The CreateProduct() method responds to HTTP POST requests to api/products.
    - It creates a new product and returns a 201 (Created) status along with the newly created product.
3. **[HttpPut]**:
    - The UpdateProduct() method responds to HTTP PUT requests to api/products/{id}.
    - Updates an existing product and returns a 204 (No Content) response if the update was successful.
4. **[HttpDelete]**:
    - The DeleteProduct() method responds to HTTP DELETE requests to api/products/{id}.
    - Deletes a product and returns a 204 (No Content) response.

3. Attribute Routing

Routing in ASP.NET Core Web API can be done using **attribute routing**, which allows developers to define routes directly on action methods and controllers.

```
[HttpGet("search/{name}")]

public ActionResult<IEnumerable<Product>> SearchProducts(string name)

{

   var matchedProducts = products.Where(p => p.Name.Contains(name,
StringComparison.OrdinalIgnoreCase)).ToList();

   if (!matchedProducts.Any())

   {

      return NotFound();

   }

   return Ok(matchedProducts);

}
```

Here, the [HttpGet("search/{name}")] attribute defines a custom route. This method allows users to search for products by name using the endpoint api/products/search/{name}.

4. Binding Data to Controllers

Controllers often need to bind incoming HTTP data to method parameters. ASP.NET Core provides several ways to achieve this:

- **From Route**: Bind data from the route parameters.
- **From Query**: Bind data from query strings.
- **From Body**: Bind data from the request body (usually JSON).

Example: Data Binding

```
// Bind id from the route and search from the query string

[HttpGet("{id}")]

public ActionResult<Product> GetProduct(int id, [FromQuery] string search)

{

   // logic...

}
```

In this example:

- The id is automatically bound from the route (api/products/1).
- The search parameter is bound from the query string (api/products/1?search=laptop).

5. Validation in Web API Controllers

Validation ensures that incoming data is correct before performing any business logic. ASP.NET Core provides a built-in validation system through **Data Annotations**.

Example: Validating a Product Model

```
public class Product
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Name is required.")]
    [MaxLength(50)]
    public string Name { get; set; }

    [Range(0.01, 9999.99, ErrorMessage = "Price must be between 0.01 and 9999.99.")]
    public decimal Price { get; set; }

    [StringLength(200)]
    public string Description { get; set; }
}
```

Validating in the Controller

```
[HttpPost]
public ActionResult<Product> CreateProduct(Product newProduct)
{
   if (!ModelState.IsValid)
   {
      return BadRequest(ModelState);  // Return 400 with validation errors
   }

   newProduct.Id = products.Count + 1;
   products.Add(newProduct);
   return CreatedAtAction(nameof(GetProduct), new { id = newProduct.Id }, newProduct);
}
```

- **[Required]**: Ensures that the Name property cannot be empty.
- **[MaxLength]** and **[StringLength]**: Restrict the length of strings.
- **[Range]**: Ensures that Price is within a specific range.

If the validation fails, the controller will return a 400 Bad Request response with detailed error messages.

6. Dependency Injection in Web API Controllers

ASP.NET Core provides built-in **Dependency Injection (DI)**, making it easier to manage dependencies, such as services and data repositories.

Example: Injecting a Service into a Controller

Define a service:

```
public interface IProductService
{
   IEnumerable<Product> GetProducts();
}

public class ProductService : IProductService
{
   public IEnumerable<Product> GetProducts()
```

```
    {
        // Logic to retrieve products
    }
}
```

In the controller, inject the service via the constructor:

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    private readonly IProductService _productService;

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

    [HttpGet]
    public ActionResult<IEnumerable<Product>> GetProducts()
    {
        return Ok(_productService.GetProducts());
    }
}
```

In Program.cs, register the service:

```
eservices.AddScoped<IProductService, ProductService>();
```

Dependency injection helps in writing testable, maintainable code and keeps concerns separated.

Key Points to Remember:

- Web API Controllers handle HTTP requests and send responses. Each action method corresponds to an HTTP verb (GET, POST, PUT, DELETE).
- The [ApiController] attribute adds several helpful features, such as automatic model validation and response handling.
- Routing in Web API can be handled using attribute routing. Custom routes can be defined with parameters.
- ASP.NET Core automatically binds data from the route, query string, and request body to action method parameters.
- Validation can be performed using **Data Annotations**. If validation fails, the controller returns a 400 Bad Request with details.
- ASP.NET Core has built-in dependency injection, which helps to manage services and dependencies cleanly.

API Controllers vs. MVC Controllers

In ASP.NET Core, both **API controllers** and **MVC controllers** are foundational components for handling requests, but they have distinct purposes and are used in different scenarios. Understanding the differences between them helps in choosing the right controller type for specific needs.

1. Overview of API Controllers

API controllers are specialized controllers designed to build RESTful APIs. They primarily handle data exchanges over HTTP using JSON or XML formats. Typically, API controllers are used when the client is a mobile app, a web frontend consuming APIs (like Angular or React), or when the goal is to build a service-oriented architecture.

```
[ApiController]

[Route("api/[controller]")]

public class ProductsController : ControllerBase

{

   [HttpGet]

   public ActionResult<IEnumerable<Product>> GetProducts()

   {

      // Return JSON data

      return Ok(new List<Product>());

   }

}
```

- **[ApiController]** attribute is used to facilitate Web API-specific behavior.
- API controllers return serialized data (JSON or XML) rather than HTML views.

2. Overview of MVC Controllers

MVC controllers are used to handle both requests for HTML pages and data-driven requests in traditional web applications. The MVC (Model-View-Controller) pattern separates concerns by having controllers handle the request, the view render the HTML, and the model represent the data.

```
public class HomeController : Controller
{
   public IActionResult Index()
   {
      // Return an HTML view
      return View();
   }
}
```

- **[Controller]** base class is used, and action methods typically return View() to render HTML views.
- MVC controllers are used in server-rendered applications (like Razor Pages or ASP.NET MVC).

3. Differences Between API and MVC Controllers

Base Class:
API controllers inherit from ControllerBase, which is specifically designed for APIs. It provides core features for handling HTTP requests without view support. On the other hand, MVC controllers inherit from Controller, which includes all the features of ControllerBase plus additional methods for handling views, making it suitable for traditional web applications.

Return Types:
API controllers typically return data in formats like JSON or XML. They often use ActionResult or ActionResult<T> to provide flexible HTTP responses. In contrast, MVC controllers generally return HTML views using the View() method, or other specific formats like PartialView().

[ApiController] Attribute:
API controllers use the [ApiController] attribute, which enhances Web API behavior by adding features like automatic model validation and automatic BadRequest responses. MVC controllers do not use this attribute, as it is tailored to APIs and not needed for rendering views.

View Support:
API controllers do not support returning views such as Razor or cshtml files. They are focused on handling and returning data. MVC controllers, on the other hand, are built to return views (HTML) and handle server-side rendering of web pages.

Purpose:
API controllers are designed for building RESTful services that return data, which is typically consumed by front-end applications or other services. MVC controllers are used for traditional web applications where the server generates HTML views that are sent to the client.

Automatic Model Validation:
In API controllers, model validation is automatic. When the [ApiController] attribute is used, the framework automatically checks the validity of the model and returns a 400 Bad Request if validation fails. In MVC controllers, however, you need to manually check the model's validity by using ModelState.IsValid.

4. ControllerBase vs. Controller

The **ControllerBase** class is a base class for API controllers that provides core features, but **without view support**. On the other hand, **Controller** inherits from ControllerBase and includes additional functionalities for MVC features like views and Razor Pages.

ControllerBase (for API Controllers):

- Focused on returning data (JSON, XML).
- Doesn't include methods for rendering views (View(), PartialView()).
- Recommended for building RESTful services or Web APIs.

```
public class ProductsController : ControllerBase
{
   [HttpGet]
   public ActionResult<IEnumerable<Product>> GetAllProducts()
   {
      return Ok(new List<Product>());
   }
}
```

Controller (for MVC Controllers):

- Inherits all the features of ControllerBase but adds methods for working with views (View(), PartialView()).
- Used for server-side rendering of HTML pages.

```
public class HomeController : Controller
{
   public IActionResult Index()
   {
      return View();  // Renders the Index.cshtml view
   }
}
```

5. Return Types in API Controllers vs. MVC Controllers

The return types differ significantly between the two controllers:

In API Controllers:

- **JSON or XML**: API controllers typically return data serialized in JSON or XML.
- **ActionResult<T>**: Introduced in ASP.NET Core 2.1, it provides type safety while allowing flexible HTTP responses.

Example:

```
[HttpGet]
public ActionResult<IEnumerable<Product>> GetProducts()
{
   return Ok(new List<Product>());
}
```

In MVC Controllers:

- **View**: MVC controllers often return View() for HTML page rendering.

Example:

```
public IActionResult Index()
{
   return View();  // Returns an HTML view
}
```

6. Automatic Model Validation in API Controllers

When the [ApiController] attribute is applied, ASP.NET Core automatically validates incoming data models before calling the action method. If validation fails, a 400 Bad Request response is returned with validation error details. This behavior simplifies model validation in API controllers.

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpPost]
    public IActionResult CreateProduct([FromBody] Product product)
    {
        // No need to manually check ModelState; API controller does it automatically
        return CreatedAtAction(nameof(GetProducts), new { id = product.Id }, product);
    }
}
```

In MVC controllers, you have to explicitly check ModelState.IsValid before processing the model.

```
public IActionResult SaveProduct(Product product)
{
    if (!ModelState.IsValid)
    {
        return View(product);  // Return view with validation errors
    }

    // Save the product and return a success view
    return RedirectToAction("Index");
}
```

7. RESTful API vs MVC

- **RESTful API**: Focuses on exposing resources (like Product, Order, etc.) over HTTP. It deals with data and operations related to resources (e.g., GET all products, POST new product).
    - Operates with different HTTP methods (GET, POST, PUT, DELETE).
    - Returns data in JSON or XML format.
    - Typically consumed by front-end applications (React, Angular) or other services.
- **MVC (Model-View-Controller)**: Primarily for server-side rendered web applications.
    - The controller handles requests and returns views (HTML pages).
    - The model represents the data, and the view is the rendered HTML.

When to Use API Controllers vs. MVC Controllers

- **Use API Controllers** when:
    - You are building a RESTful API.
    - The primary client is a mobile app, a SPA (Single Page Application), or other services consuming JSON/XML.
    - You don't need to return views or HTML content.
- **Use MVC Controllers** when:
    - You are building a server-rendered web application.
    - You need to return HTML views along with data.
    - You are building a traditional web application with Razor pages.

Key Points to Remember:

1. **API Controllers** are designed for building RESTful services, and they typically return data like JSON or XML.
2. **MVC Controllers** are meant for rendering HTML views in traditional web applications.
3. **[ApiController] Attribute**: Adds useful features like automatic model validation and automatic response handling for Web API controllers.
4. **ControllerBase** is used for API controllers and does not support views. **Controller** is used in MVC applications where views are needed.
5. **Automatic Model Validation**: API controllers automatically validate models and return a 400 Bad Request if validation fails, while MVC controllers require manual model validation using ModelState.IsValid.
6. **API Controllers** are often consumed by front-end frameworks like Angular or React, or other systems that expect JSON/XML responses.

Entity Framework Core with Web API

Entity Framework Core (EF Core) is an object-relational mapper (ORM) for .NET that enables developers to work with databases using .NET objects. It abstracts away much of the boilerplate code for database operations like queries, inserts, updates, and deletes. Integrating EF Core with a Web API allows your API to interact with databases in a clean, maintainable way.

In this part, we'll explore how EF Core can be used with a Web API to manage data, focusing on creating, reading, updating, and deleting records (commonly known as CRUD operations).

1. Setup EF Core in an ASP.NET Core Web API

Before using EF Core in a Web API project, you need to add the necessary NuGet packages and configure the database context.

Step 1: Add EF Core NuGet Packages

You need to install the following NuGet packages:

- **Microsoft.EntityFrameworkCore**
- **Microsoft.EntityFrameworkCore.SqlServer** (or another provider like MySQL, PostgreSQL, etc.)

You can install them using the .NET CLI:

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Step 2: Define the Database Context

In EF Core, the **DbContext** class represents a session with the database. It is used to configure the database connection and expose DbSet properties, which represent tables in the database.

```
public class AppDbContext : DbContext
{
   public AppDbContext(DbContextOptions<AppDbContext> options) : base(options)
   {
   }

   // Define a DbSet for the 'Products' table
   public DbSet<Product> Products { get; set; }
}
```

- The AppDbContext class inherits from DbContext.
- The Products DbSet represents the Products table in the database.

Step 3: Configure the Database Connection in appsettings.json

You configure the database connection string in the appsettings.json file:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=ecommerce_db;Trusted_Connection=True;"
  }
}
```

Step 4: Register the DbContext in Program.cs

In the Program.cs file (depending on your ASP.NET Core version), register the DbContext to enable dependency injection.

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


// Register the AppDbContext with the connection string
builder.Services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));


var app = builder.Build();
```

2. Create the Model

A model in EF Core is a class that maps to a database table. EF Core uses **convention-based mapping**, but you can also configure it explicitly using data annotations or Fluent API.

Here's an example of a simple Product model:

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Stock { get; set; }
}
```

- Id: Primary key for the table. By convention, EF Core will treat any property named Id or <ClassName>Id as the primary key.
- Other properties like Name, Price, and Stock map to columns in the Products table.

3. CRUD Operations in Web API with EF Core

Now that the model and DbContext are set up, let's implement the standard CRUD operations (Create, Read, Update, and Delete) in a Web API controller using EF Core.

Step 1: Create the API Controller

```
[ApiController]

[Route("api/[controller]")]

public class ProductsController : ControllerBase

{

   private readonly AppDbContext _context;


   public ProductsController(AppDbContext context)

   {

      _context = context;

   }

}
```

Here, the AppDbContext is injected into the ProductsController through the constructor, allowing it to be used for database operations.

Step 2: Create a Product (POST)

```
[HttpPost]

public async Task<ActionResult<Product>> CreateProduct([FromBody] Product product)

{

   // Add the product to the DbSet

   _context.Products.Add(product);


   // Save changes asynchronously

   await _context.SaveChangesAsync();


   return CreatedAtAction(nameof(GetProductById), new { id = product.Id }, product);

}
```

- The [HttpPost] attribute specifies that this method handles POST requests.

- The CreatedAtAction method returns a 201 status code with the location of the created resource.

Step 3: Read (GET) All Products

```
[HttpGet]
public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
{
   var products = await _context.Products.ToListAsync();
   return Ok(products);
}
```

- The [HttpGet] attribute specifies that this method handles GET requests.
- The ToListAsync method retrieves all the products from the database asynchronously.

Step 4: Read (GET) Product by ID

```
[HttpGet("{id}")]
public async Task<ActionResult<Product>> GetProductById(int id)
{
   var product = await _context.Products.FindAsync(id);

   if (product == null)
   {
      return NotFound();
   }

   return Ok(product);
}
```

- The {id} route parameter captures the product ID from the URL.
- The FindAsync method fetches the product with the given ID.
- If the product is not found, it returns a 404 status code.

Step 5: Update a Product (PUT)

```
[HttpPut("{id}")]
public async Task<IActionResult> UpdateProduct(int id, [FromBody] Product product)
{
   if (id != product.Id)
   {
      return BadRequest();
   }

   _context.Entry(product).State = EntityState.Modified;
   await _context.SaveChangesAsync();

   return NoContent();  // Return 204 No Content on successful update
}
```

- The [HttpPut] attribute specifies that this method handles PUT requests.
- The Entry method is used to mark the product entity as modified in the context.
- The method returns a 204 No Content response when the update is successful.

Step 6: Delete a Product (DELETE)

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteProduct(int id)
{
   var product = await _context.Products.FindAsync(id);

   if (product == null)
   {
      return NotFound();
   }

   _context.Products.Remove(product);
   await _context.SaveChangesAsync();

   return NoContent();
}
```

- The [HttpDelete] attribute specifies that this method handles DELETE requests.
- If the product is found, it is removed from the Products DbSet and the changes are saved.
- The method returns a 204 No Content response when the deletion is successful.

4. Migration: Creating the Database

EF Core uses **migrations** to keep the database schema in sync with the data model. To create the initial database schema, follow these steps:

Step 1: Add Migration

Run the following command in the terminal to create a migration based on the model:

```
dotnet ef migrations add InitialCreate
```

This command creates a migration script that includes the schema changes (e.g., creating the Products table).

Step 2: Update the Database

Run the following command to apply the migration and update the database:

dotnet ef database update

This will create the database (if it doesn't exist) and apply the migration to generate the necessary tables.

5. Using EF Core in Production

For production environments, ensure the following:

- Use appropriate database providers and connection strings for cloud services (e.g., Azure SQL).
- Implement **caching** strategies to reduce the load on the database for frequently accessed data.
- Use **transactions** when performing multiple related database operations to ensure consistency.

Key Points to Remember:

1. **DbContext** represents the session with the database and is the main class for interacting with data using EF Core.
2. **DbSet<T>** represents a table in the database, and each DbSet in the DbContext is mapped to a model class.
3. Use **[FromBody]** to bind the incoming JSON data to the model when creating or updating records.
4. The **SaveChangesAsync** method persists changes to the database.
5. **Migrations** help in syncing the database schema with your model. Use dotnet ef migrations add and dotnet ef database update to manage database updates.
6. API controllers with EF Core use asynchronous methods (like ToListAsync, FindAsync) to ensure non-blocking I/O operations, which is crucial for scalability in web applications.

Different Return Types of Web API Action Methods

In ASP.NET Core Web APIs, action methods (or controller methods) can return different types of results depending on the scenario. Understanding the various return types helps in providing meaningful HTTP responses for different situations (success, failure, errors, etc.). The choice of return type depends on whether the action method will return data, status codes, or a combination of both.

Let's explore the most commonly used return types in Web API:

1. Void

A method can return nothing if no data or status needs to be explicitly returned. This is mostly used for operations like logging where no client feedback is necessary.

```
[HttpPost]
public void LogActivity([FromBody] Activity activity)
{
   // Log activity without returning anything
   _logger.Log(activity);
}
```

- **Use case**: This return type is rare in Web APIs, as APIs generally need to communicate results back to the client.
- **Implicit response**: The response will still return a status code like 200 OK.

2. Primitive Types (e.g., int, string, bool)

Action methods can return simple primitive types, like integers, strings, or booleans. This is useful when you want to send back basic data without a complex structure.

```
[HttpGet("{id}")]
public int GetUserId(int id)
{
   return id;
}
```

- **Use case**: Return basic data, such as an ID, a confirmation message, or a flag (true/false).
- **Default response**: The response type will be a JSON representation of the primitive type.

3. IEnumerable<T> or List<T>

When retrieving a collection of items, it's common to return an IEnumerable<T> or List<T>. This is used for operations like GET /products where multiple items are fetched.

```
[HttpGet]
public IEnumerable<Product> GetProducts()
{
   return _context.Products.ToList();
}
```

- **Use case**: Return lists or collections of objects (e.g., products, users).
- **Default response**: A JSON array of objects is returned to the client.

4. Object

You can return a custom object that represents a specific model or entity. This is ideal when returning structured data such as a single resource (e.g., a product, user, etc.).

```
[HttpGet("{id}")]
public Product GetProduct(int id)
{
   return _context.Products.Find(id);
}
```

- **Use case**: Return a single item or structured data object.
- **Default response**: A JSON object is returned with the properties of the entity.

5. Task / Task<T> (Asynchronous Methods)

In modern web applications, it is common to use asynchronous methods that return a Task or Task<T>. Asynchronous programming helps prevent blocking of threads and allows more efficient use of resources.

```
[HttpGet("{id}")]

public async Task<Product> GetProductAsync(int id)

{

   return await _context.Products.FindAsync(id);

}
```

- **Use case**: Async methods for database operations, HTTP calls, or other I/O-bound tasks.
- **Default response**: A JSON response wrapped in a Task for asynchronous execution.

6. ActionResult<T>

This is a flexible return type that allows you to return either a specific type (like an object) or an HTTP response (like NotFound, BadRequest, etc.). ActionResult<T> is a combination of both ActionResult and the specific return type T.

```
[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProduct(int id)

{

   var product = await _context.Products.FindAsync(id);


   if (product == null)

   {

      return NotFound();

   }


   return product;

}
```

- **Use case**: Allows returning either an object (successful response) or an error status (failure response) in the same method.
- **Default response**: If the result is successful, it returns the object in JSON format. If not, it can return a specific status code like 404 Not Found.

7. IActionResult

IActionResult is a more generic return type that allows you to return any kind of HTTP response (success, error, redirect, etc.). It does not specify the actual type of the returned data, allowing full control over the response.

```
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteProduct(int id)
{
    var product = await _context.Products.FindAsync(id);

    if (product == null)
    {
        return NotFound();
    }

    _context.Products.Remove(product);
    await _context.SaveChangesAsync();

    return NoContent();  // Return 204 No Content
}
```

- **Use case**: Gives you the flexibility to return HTTP status codes or formatted results (e.g., JSON).
- **Default response**: You can specify the response format by using methods like Ok(), BadRequest(), NotFound(), or NoContent().

8. Custom Response Wrappers

In some cases, you might want to return a custom response format, which could include both data and metadata such as status, message, or pagination info.

```
public class ApiResponse<T>
{
    public bool Success { get; set; }
    public T Data { get; set; }
    public string Message { get; set; }
}
```

```
[HttpGet("{id}")]
public async Task<ActionResult<ApiResponse<Product>>> GetProduct(int id)
{
    var product = await _context.Products.FindAsync(id);

    if (product == null)
    {
        return new ApiResponse<Product>
        {
            Success = false,
            Message = "Product not found"
        };
    }

    return new ApiResponse<Product>
    {
        Success = true,
        Data = product,
        Message = "Product retrieved successfully"
    };
}
```

- **Use case**: When you need a consistent response structure across the entire API.
- **Default response**: A custom JSON response with fields like Success, Data, and Message.

9. FileResult / PhysicalFileResult

When returning files (such as PDFs, images, or CSVs), you can use the FileResult or PhysicalFileResult return types.

```
[HttpGet("download/{fileName}")]
public IActionResult DownloadFile(string fileName)
{
```

```
    var filePath = Path.Combine("wwwroot/files", fileName);
    var fileBytes = System.IO.File.ReadAllBytes(filePath);

    return File(fileBytes, "application/octet-stream", fileName);
}
```

- **Use case**: When you need to return a file (e.g., downloading reports, images).
- **Default response**: A file download prompt is shown to the user.

Key Points to Remember:

1. **Primitive Types**: Suitable for returning simple data like strings or numbers. The data will be automatically serialized to JSON.
2. **IEnumerable<T> or List<T>**: Ideal for returning collections of data, such as lists of products or users.
3. **Object**: When you need to return a single entity (e.g., a product or user), use an object return type.
4. **ActionResult<T>**: Provides flexibility, allowing you to return either a specific result or an error status code (like 404 NotFound).
5. **IActionResult**: The most flexible return type, giving you full control over the HTTP response (e.g., Ok, BadRequest, NotFound).
6. **Task / Task<T>**: Use for asynchronous operations to prevent blocking threads in I/O-bound operations (common in database and network calls).
7. **FileResult**: Use when you need to return files from your Web API (e.g., for file downloads).
8. **Custom Wrappers**: Can be used to return consistent response formats, which include data and additional metadata such as success status and error messages.

ControllerBase

ControllerBase is a foundational class in ASP.NET Core MVC and Web API applications. It provides a base class for controllers that handle HTTP requests and responses. While ControllerBase is primarily used for Web API controllers, it is essential to understand its purpose and capabilities as it forms the core of how APIs are structured and function.

Overview of ControllerBase

ControllerBase is part of the Microsoft.AspNetCore.Mvc namespace and serves as the base class for Web API controllers. Unlike Controller, which is used for MVC (Model-View-Controller) applications that involve views, ControllerBase is tailored for building APIs without views.

Key Features:

- **Action Methods**: ControllerBase provides methods that handle HTTP requests and respond with results. These methods are often decorated with HTTP attribute annotations like [HttpGet], [HttpPost], [HttpPut], and [HttpDelete].
- **Response Types**: It includes methods for returning various HTTP responses, including JSON content, status codes, and more.
- **Dependency Injection**: ControllerBase supports dependency injection, allowing you to inject services like repositories or application services directly into the controller.
- **Model Binding and Validation**: It provides built-in support for model binding and validation, making it easy to work with data coming from HTTP requests.
-

Example Usage:

Here's a simple example of how ControllerBase is used in a Web API controller:

```
using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;


namespace MyApi.Controllers

{

  [ApiController]

  [Route("api/[controller]")]

  public class ProductsController : ControllerBase

  {

    private readonly ApplicationDbContext _context;


    public ProductsController(ApplicationDbContext context)

    {

      _context = context;
```

```
    }

    [HttpGet]
    public ActionResult<IEnumerable<Product>> GetProducts()
    {
        var products = _context.Products.ToList();
        return Ok(products); // Returns a 200 OK response with the list of products
    }

    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = _context.Products.Find(id);
        if (product == null)
        {
            return NotFound(); // Returns a 404 Not Found response
        }
        return Ok(product); // Returns a 200 OK response with the product
    }

    [HttpPost]
    public ActionResult<Product> CreateProduct(Product product)
    {
        _context.Products.Add(product);
        _context.SaveChanges();
        return CreatedAtAction(nameof(GetProduct), new { id = product.Id }, product); // Returns a
201 Created response
    }
  }
}
```

Explanation of Key Elements:

1. **Attributes**:
    - [ApiController]: Indicates that the controller responds to Web API requests.
    - [Route("api/[controller]")]: Defines the base route for the controller's endpoints. [controller] is replaced with the controller's name (e.g., Products).
2. **Dependency Injection**:
    - The constructor accepts an ApplicationDbContext instance, demonstrating how services are injected into controllers.
3. **Action Methods**:
    - GetProducts(): Returns a list of products with a 200 OK response.
    - GetProduct(int id): Retrieves a specific product by ID. If not found, it returns a 404 Not Found response.
    - CreateProduct(Product product): Adds a new product to the database and returns a 201 Created response with the newly created product.

Key Methods of ControllerBase:

- **Ok()**: Returns a 200 OK response with optional content.
- **CreatedAtAction()**: Returns a 201 Created response, typically used after a resource has been created.
- **NotFound()**: Returns a 404 Not Found response when a resource is not found.
- **BadRequest()**: Returns a 400 Bad Request response, often used when the client sends invalid data.
- **NoContent()**: Returns a 204 No Content response, used for successful requests that don't return data.

Advantages of Using ControllerBase:

- **No View Support**: Ideal for Web APIs as it does not include view-related features, which are not needed for APIs.
- **Focus on API Responses**: It focuses on handling HTTP requests and responses, making it straightforward for API development.
- **Dependency Injection**: Seamlessly integrates with ASP.NET Core's dependency injection system for service management.

Key Points to Remember:

1. **Purpose**: ControllerBase is used as the base class for Web API controllers, providing essential methods for handling HTTP requests and responses without view support.
2. **Action Methods**: Used to handle various HTTP operations (GET, POST, PUT, DELETE) and return appropriate responses.
3. **Response Methods**: Includes methods like Ok(), NotFound(), BadRequest(), NoContent() for standard HTTP responses.
4. **Dependency Injection**: Supports injection of services (e.g., repositories) directly into controllers.
5. **No View Support**: Focuses on APIs and does not support views, unlike the Controller class used in MVC applications.

Summary

In this section, we've covered several foundational concepts related to ASP.NET Core Web API development, including RESTful principles, the role of Web API controllers, differences between API and MVC controllers, integration with Entity Framework Core, return types for Web API methods, and the use of IActionResult vs ActionResult<T>.

Here's a consolidated overview of each concept:

1. ASP.NET Core Web API and RESTful Principles

- **RESTful Principles**: REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs use HTTP requests to perform CRUD operations and follow principles such as statelessness, resource-based URIs, and standard HTTP methods (GET, POST, PUT, DELETE).
- **Resource-Based**: Resources are entities that APIs expose (e.g., products, users). URIs should represent resources, and actions should be performed using HTTP methods.
- **Stateless**: Each request from a client must contain all necessary information for the server to fulfill the request. The server does not store client state between requests.
- **Uniform Interface**: A consistent, standard way to interact with resources across the API. This includes using standard HTTP methods and status codes.

2. Web API Controllers

- **Purpose**: Web API controllers are responsible for handling HTTP requests and returning responses in the form of JSON or XML data. They inherit from ControllerBase and are typically decorated with [ApiController] to enable Web API-specific features.
- **Attributes**: [Route] defines the route template for the controller, and [ApiController] enables automatic model validation and binding.

3. API Controllers vs MVC Controllers

- **API Controllers**:
    - Inherit from ControllerBase.
    - Designed for handling HTTP requests and returning data (usually in JSON format).
    - Do not include support for rendering views.
- **MVC Controllers**:
    - Inherit from Controller.
    - Designed for handling requests that involve rendering views.
    - Include support for view-related features, such as returning HTML content and using Razor views.

4. Entity Framework Core with Web API

- **Purpose**: EF Core is an Object-Relational Mapper (ORM) that provides a way to interact with databases using .NET objects. It abstracts database operations and enables CRUD operations through LINQ queries.
- **Integration**: Typically involves setting up a DbContext class to manage entities and configure relationships. Controllers use dependency injection to access the DbContext for data operations.

5. Different Return Types of Web API Action Methods

- **IActionResult**:
    - Provides flexibility to return various types of responses and HTTP status codes.
    - Example: Ok(), NotFound(), BadRequest(), Redirect(), File().
- **ActionResult<T>**:
    - Combines IActionResult with a specific return type, allowing you to return both data and status codes from the same action method.
    - Example: Returning a Product object with Ok(product).

6. IActionResult vs ActionResult<T>

- **IActionResult**:
    - Provides control over different HTTP responses and status codes.
    - Suitable for scenarios requiring diverse response types.
- **ActionResult<T>**:
    - Simplifies returning data and status codes together.
    - Ideal for methods returning a specific type along with potential status codes.

7. ControllerBase

- **Purpose**: Serves as the base class for Web API controllers, focusing on handling HTTP requests and responses without view support.
- **Key Methods**: Includes methods like Ok(), NotFound(), BadRequest(), NoContent(), and CreatedAtAction().
- **Dependency Injection**: Supports injecting services into controllers for data operations and business logic.

Key Points to Remember

1. **RESTful Principles**: Focus on resource-based URIs, stateless communication, and uniform interfaces for API design.
2. **Web API Controllers**: Inherit from ControllerBase, handle HTTP requests, and return data responses.
3. **API vs MVC Controllers**: API controllers handle data responses without views, while MVC controllers manage both data and view rendering.
4. **Entity Framework Core**: Used for ORM in ASP.NET Core, enabling CRUD operations and data management.
5. **Return Types**: IActionResult provides flexible responses, while ActionResult<T> combines data and status codes.
6. **ControllerBase**: Core class for Web API controllers, focusing on HTTP request handling and response generation.