Instructor example

Web University by Harsha Vardhan

What is ASP.NET Core Minimal API, and how does it differ from traditional Web API?

ASP.NET Core Minimal API is a lightweight and simplified approach to building web APIs in ASP.NET Core. It aims to reduce the amount of boilerplate code required to create APIs by leveraging the new minimalistic programming model. Unlike traditional Web API, which typically involves multiple files and configurations, Minimal API allows developers to define the API routes and handlers in a single file, making it easier to understand and maintain.

How to implement CRUD operations using Asp.Net Core Minimal API? Explain with sample code.

To create minimal API CRUD operations in ASP.NET Core 6, follow these steps:

1. Create a new ASP.NET Core 6 project in Visual Studio:

- Open Visual Studio.
- Click on "Create a new project."
- Select "ASP.NET Core Empty" template.
- Configure other project details and click on "Create."

2. Define your data model:

Create a class that represents the entity you want to perform CRUD operations on. For example, a "Product" class with properties like Id, Name, Price, etc.

3. Implement the API endpoints:

Open the project's Program.cs file.

Add the necessary services for your application. For example, you'll typically need to add a database context using builder.Services.AddDbContext.

Here's an example of creating minimal API CRUD operations for a "Product" entity:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services
builder.Services.AddDbContext<ApplicationDbContext>(options =>
{
 options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
});

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
 c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1" });
});

// Create database and migrate
using (var app = builder.Build())
{
using (var scope = builder.Services.CreateScope())
{
  var services = scope.ServiceProvider;
  var dbContext = services.GetRequiredService<ApplicationDbContext>();
  dbContext.Database.Migrate();
}

// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();

// Specify the Swagger JSON endpoint.
```

```
app.UseSwaggerUI(c =>
{
 c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
 c.RoutePrefix = string.Empty;
});
```

4. Define API endpoints for GET, POST, PUT and DELETE operations

```
app.MapGet("/api/products", async (ApplicationDbContext dbContext) =>
{
  var products = await dbContext.Products.ToListAsync();
  return Results.Ok(products);
});

app.MapGet("/api/products/{id}", async (int id, ApplicationDbContext dbContext) =>
{
  var product = await dbContext.Products.FindAsync(id);
  if (product != null)
  {
    return Results.Ok(product);
  }
  else
  {
    return Results.NotFound();
  }
});

app.MapPost("/api/products", async (Product product, ApplicationDbContext  dbContext) =>
{
  dbContext.Products.Add(product);
  await dbContext.SaveChangesAsync();
  return Results.Created($"/api/products/{product.Id}", product);
});

app.MapPut("/api/products/{id}", async (int id, Product updatedProduct, ApplicationDbContext dbContext) =>
{
  var product = await dbContext.Products.FindAsync(id);
  if (product != null)
  {
    product.Name = updatedProduct.Name;
    product.Price = updatedProduct.Price;
    // Update other properties as needed

    await dbContext.SaveChangesAsync();
    return Results.Ok(product);
  }
  else
  {
  return Results.NotFound();
  }
});

app.MapDelete("/api/products/{id}", async (int id, ApplicationDbContext  dbContext) =>
{
  var product = await dbContext.Products.FindAsync(id);
  if (product != null)
  {
    dbContext.Products.Remove(product);
    await dbContext.SaveChangesAsync();
    return Results.NoContent();
  }
  else
  {
    return Results.NotFound();
  }
});

app.Run();
}
```

In this example, the endpoints for listing products, retrieving a product by id, creating a product, updating a product, and deleting a product are defined using app.MapGet, app.MapPost, app.MapPut, and app.MapDelete respectively. The endpoints use the ApplicationDbContext to access the database and perform CRUD operations. Additionally, Swagger is configured to generate documentation for the API.

How do you handle request routing and parameter binding in ASP.NET Core Minimal API?

In ASP.NET Core Minimal API, request routing and parameter binding can be handled using the MapMethods extension method within the Configure method. The MapMethods method allows you to specify the HTTP verb, route pattern, and handler function for each API endpoint.

The MapMethod can be MapGet, MapPost, MapPut and MapDelete.

For example, to handle a GET request with a route parameter, you can define a route pattern with a placeholder for the parameter and bind it to the handler function using lambda syntax:

```
app.MapGet("/api/users/{id}", new[] { "GET" }, (int id) =>
{
  // Handle the GET request with the specified route parameter
  // Access the 'id' parameter in the handler function
  // Perform necessary operations and return a response
});
```

ASP.NET Core Minimal API uses a built-in route parameter binding feature that automatically maps the route parameter to the corresponding parameter in the handler function. You can use various parameter types (such as int, string, DateTime, etc.) depending on the type of data expected.

How do you perform model validation in ASP.NET Core Minimal API?

To perform model validation in ASP.NET Core Minimal API, you can utilize the model binding and validation features provided by the framework. Follow these steps:

Define a request model class with the desired properties and validation attributes. For example:

```
public class UserRequest
{
  [Required]
  public string Name { get; set; }

  [Range(18, 99)]
  public int Age { get; set; }
}
```

In the handler function for the API endpoint, add a parameter of the defined request model type. ASP.NET Core will automatically bind and validate the request data against the model.

Check the ModelState.IsValid property to determine if the model validation passed or not. If the model is invalid, you can return a BadRequest response with the validation errors.

Here's an example of a POST endpoint with model validation:

```
app.MapPost("/api/users", new[] { "POST" }, (UserRequest user) =>
{
  if (!ModelState.IsValid)
  {
    return Results.BadRequest(ModelState);
  }

  // Process the valid user request and return a response
});
```

ASP.NET Core will automatically perform validation based on the defined attributes, such as Required or Range, and populate the ModelState with any validation errors. Returning a BadRequest result with the ModelState will include the validation errors in the response.

How can you implement authentication and authorization in ASP.NET Core Minimal API?

To implement authentication and authorization in ASP.NET Core Minimal API, you can leverage the authentication and authorization middleware provided by the framework. Here's a high-level overview of the steps involved:

Install the required NuGet packages for the authentication and authorization middleware. For example, Microsoft.AspNetCore.Authentication and Microsoft.AspNetCore.Authorization.

Configure the desired authentication scheme(s) in the services collection. This typically involves setting up authentication options, such as JwtBearer or cookies, and specifying the authentication provider details.

Use the UseAuthentication and UseAuthorization middleware in the request pipeline. Ensure that the UseAuthentication middleware is placed before the routing middleware to authenticate incoming requests.

Apply the necessary authorization attributes to the API endpoints or handler functions to restrict access. For example, [Authorize] attribute can be used to specify that only authenticated users are allowed to access an endpoint.

Here's an example of configuring JWT authentication and applying authorization to an endpoint:

```
// Services
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer(options =>
{
  // Configure JWT authentication options
});

builder.Services.AddAuthorization();

// Request pipeline
var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();

app.MapMethods("/api/protected", new[] { "GET" }, () =>
{
  // This endpoint requires authorization
}).RequireAuthorization();

app.Run();
```

In this example, the JWT authentication scheme is configured in ConfigureServices, and the UseAuthentication and UseAuthorization middleware are applied in Configure. The /api/protected endpoint requires authorization, and the [Authorize] attribute can also be applied to the handler function for the same effect.

Your submission

Cesar David Guerrero Regino

What is ASP.NET Core Minimal API, and how does it differ from traditional Web API?

How to implement CRUD operations using Asp.Net Core Minimal API? Explain with sample code.

How do you handle request routing and parameter binding in ASP.NET Core Minimal API?

How do you perform model validation in ASP.NET Core Minimal API?

How can you implement authentication and authorization in ASP.NET Core Minimal API?