**Demonstrating Swagger Installation and Usage with WebAPI**

Swagger (OpenAPI) provides a standardized, language-agnostic interface to REST APIs, allowing both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or network traffic inspection. Swashbuckle.AspNetCore is a popular library for integrating Swagger into ASP.NET Core Web APIs.

**1. Nuget Package to Download: Swashbuckle.AspNetCore**

**Steps:**

1. **Create a new ASP.NET Core Web API project:**
   * Open Visual Studio.
   * Select "Create a new project."
   * Search for and select "ASP.NET Core Web API."
   * Click "Next," provide a name for your project (e.g., MyWebApiSwaggerDemo), and click "Create."
2. **Install the Swashbuckle.AspNetCore NuGet Package:**
   * In Visual Studio, right-click on your project in Solution Explorer.
   * Select "Manage NuGet Packages..."
   * Go to the "Browse" tab.
   * Search for Swashbuckle.AspNetCore.
   * Select the latest stable version and click "Install."

**2. Usage of ProducesResponseType to Web API Method**

[ProducesResponseType] attribute improves the Swagger documentation by explicitly declaring the possible HTTP response types and their associated data models for an API action.

**Example:**

Modify a controller action (e.g., WeatherForecastController.cs or create a new one):

C#

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

using System.Net; // For HttpStatusCode

namespace MyWebApiSwaggerDemo.Controllers

{

[ApiController]

[Route("[controller]")]

public class ProductsController : ControllerBase

{

private static readonly List<Product> \_products = new List<Product>

{

new Product { Id = 1, Name = "Laptop", Price = 1200.00m },

new Product { Id = 2, Name = "Mouse", Price = 25.00m }

};

[HttpGet]

[ProducesResponseType(typeof(IEnumerable<Product>), (int)HttpStatusCode.OK)] // Success

[ProducesResponseType((int)HttpStatusCode.NotFound)] // Not found (if applicable)

public IActionResult GetProducts()

{

if (!\_products.Any())

{

return NotFound();

}

return Ok(\_products);

}

[HttpGet("{id}")]

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult GetProductById(int id)

{

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

if (product == null)

{

return NotFound();

}

return Ok(product);

}

[HttpPost]

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

public IActionResult AddProduct([FromBody] Product product)

{

if (product == null || string.IsNullOrWhiteSpace(product.Name))

{

return BadRequest("Product data is invalid.");

}

product.Id = \_products.Max(p => p.Id) + 1;

\_products.Add(product);

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

}

[HttpDelete("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult DeleteProduct(int id)

{

var productToRemove = \_products.FirstOrDefault(p => p.Id == id);

if (productToRemove == null)

{

return NotFound();

}

\_products.Remove(productToRemove);

return NoContent();

}

}

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

**3. AddSwaggerGen and UseSwaggerUI**

These methods are crucial for configuring Swagger generation and enabling the Swagger UI in your ASP.NET Core application.

**Steps:**

1. **Configure Startup.cs (for .NET 5.0 and earlier) or Program.cs (for .NET 6.0 and later):**

**For .NET 5.0 and earlier (Startup.cs):**

C#

// Inside Startup.cs

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

// Add SwaggerGen service

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "My Web API", Version = "v1" });

});

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

// Enable Swagger and Swagger UI

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My Web API V1");

// To serve Swagger UI at application's root (http://localhost:<port>/)

// c.RoutePrefix = string.Empty;

});

}

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

**For .NET 6.0 and later (Program.cs):**

C#

// Inside Program.cs

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "My Web API", Version = "v1" });

});

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My Web API V1");

// To serve Swagger UI at application's root (http://localhost:<port>/)

// c.RoutePrefix = string.Empty;

});

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

**4. WebAPI Listing on Browser**

**Steps:**

1. **Run your Web API project.**
   * Press F5 in Visual Studio or run dotnet run from your project directory in the terminal.
2. **Navigate to the Swagger UI in your browser:**
   * Once the application starts, it will typically open a browser tab. If you haven't changed c.RoutePrefix, the Swagger UI will be available at http://localhost:<port>/swagger.
   * You should see a well-formatted page listing all your API endpoints, their methods, parameters, and the ProducesResponseType information you added. You can also test your API directly from this interface.

**Demonstrating the Usage of Postman Tool to Hit WebAPI Methods**

Postman is a popular API client that makes it easy to create, share, test, and document APIs.

**1. Structure in Postman Tool**

* **Sidebar:** Contains Collections, APIs, Environments, Mock Servers, Monitors, and History.
* **Workspace:** The main area where you work on requests.
* **Request Tab:** Each tab represents an individual API request.
* **Request Builder:** Where you define your HTTP method, URL, headers, and body.
* **Response Viewer:** Displays the response from the server.

**2. Headers with Authorization**

Headers provide meta-information about the request or response. Authorization headers are crucial for securing APIs.

**Steps:**

1. **Open Postman.**
2. **Create a New Request:** Click the + button on the tabs bar or "File" > "New" > "Request."
3. **Enter the API URL:** For example, http://localhost:<port>/Products.
4. **Select HTTP Method:** Choose GET, POST, PUT, DELETE, etc.
5. **Add Authorization Header:**
   * Go to the "Headers" tab in the Request Builder.
   * Add a Key: Authorization
   * Add a Value: Bearer <Your\_JWT\_Token> (if using JWT authentication) or Basic <base64\_encoded\_username:password> (for basic authentication).
   * **Alternatively, use the "Authorization" tab:** Postman provides a dedicated "Authorization" tab where you can select common authorization types (Bearer Token, Basic Auth, OAuth 2.0, etc.) and Postman will construct the header for you.

**3. Body as JSON**

For POST and PUT requests, you often send data in the request body, typically as JSON.

**Steps:**

1. **Select HTTP Method:** Choose POST or PUT.
2. **Go to the "Body" tab.**
3. **Select "raw" and then "JSON" from the dropdown.**
4. **Enter your JSON payload** in the text area.

**Example (for POST /Products):**

JSON

{

"name": "Keyboard",

"price": 75.50

}

**4. Option to Choose the Type of Request**

This refers to the HTTP verb (GET, POST, PUT, DELETE, PATCH, OPTIONS, HEAD). It's a dropdown menu right next to the URL input field in the Request Builder.

**5. Request Collection and How to Add a New Request in the Collection**

Collections allow you to organize your API requests into folders, making it easier to manage and share them.

**Steps:**

1. **Create a New Collection:**
   * In the sidebar, click on "Collections."
   * Click the + icon next to "Collections" or the "New Collection" button.
   * Give it a name (e.g., "My Web API Tests").
2. **Add a Request to a Collection:**
   * Once you have a request open in a tab, click the "Save" button.
   * In the "SAVE REQUEST" dialog, choose your created collection from the dropdown.
   * Give the request a name (e.g., "Get All Products").
   * Click "Save."
   * Alternatively, right-click on your collection in the sidebar and select "Add Request."

**6. Tabs in the Center Pane that Corresponds to the Request**

* Each time you open a saved request from a collection or create a new request, it opens in a new tab in the center pane.
* This allows you to work on multiple API requests simultaneously without losing your work.
* You can easily switch between tabs to manage different requests.

**Demonstrating the Usage of Route and Explain Name Attribute in Http Requests**

Routing in ASP.NET Core maps incoming HTTP requests to action methods on controllers.

**1. Importance of User-Friendly Name to Action Method**

The Name attribute in routing allows you to define a user-friendly name for a route. This name can then be used to generate URLs for that route without hardcoding the URL path, making your application more robust to URL changes.

**Example:**

Consider a ProductsController with an action to get a product by ID:

C#

[ApiController]

[Route("api/[controller]")] // Controller-level route prefix

public class ProductsController : ControllerBase

{

// ... (Product class and \_products list from previous example)

[HttpGet("{id}", Name = "GetProductByIdRoute")] // Route template and Name attribute

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult GetProduct(int id)

{

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

if (product == null)

{

return NotFound();

}

return Ok(product);

}

[HttpPost(Name = "AddProductRoute")] // Route template and Name attribute

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

public IActionResult CreateProduct([FromBody] Product product)

{

if (product == null || string.IsNullOrWhiteSpace(product.Name))

{

return BadRequest("Product data is invalid.");

}

product.Id = \_products.Max(p => p.Id) + 1;

\_products.Add(product);

// Using CreatedAtRoute with the route name for URL generation

return CreatedAtRoute("GetProductByIdRoute", new { id = product.Id }, product);

}

}

**Benefits of Name attribute:**

* **Decoupling:** If the URL path for GetProductById changes (e.g., from api/Products/{id} to api/V2/Products/{id}), you only need to update the Route attribute. Any code that uses CreatedAtRoute("GetProductByIdRoute", ...) will automatically generate the correct URL.
* **Readability:** Using a descriptive name can make your code easier to understand.
* **Testability:** Helps in testing scenarios where you need to verify URL generation.

**2. Explain the Usage of ActionName to Have More Than 1 Method with the Same Action Verb**

By default, ASP.NET Core routing uses the action method's name to match incoming requests. However, sometimes you might want to have multiple methods with the same HTTP verb (e.g., two GET methods) but differentiate them based on their purpose or parameters, without changing their actual method name. The [ActionName] attribute allows you to assign a different name for routing purposes.

**Example:**

C#

[ApiController]

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

public class UsersController : ControllerBase

{

private static readonly List<string> \_users = new List<string> { "Alice", "Bob", "Charlie" };

[HttpGet] // Matches GET /api/Users

public IActionResult GetUsers()

{

return Ok(\_users);

}

[HttpGet("search")] // Matches GET /api/Users/search?name=...

[ActionName("FindUserByName")] // Route name for this action

public IActionResult SearchUserByName([FromQuery] string name)

{

var user = \_users.FirstOrDefault(u => u.Contains(name, StringComparison.OrdinalIgnoreCase));

if (user == null)

{

return NotFound();

}

return Ok(user);

}

[HttpGet("{id:int}")] // Matches GET /api/Users/{id}

[ActionName("GetUserById")] // Route name for this action

public IActionResult GetUserById(int id)

{

// This is a simplified example, usually you'd retrieve by ID from a data store

if (id >= 0 && id < \_users.Count)

{

return Ok(\_users[id]);

}

return NotFound();

}

}

In this example:

* We have three GET methods.
* GetUsers is the default GET for the controller.
* SearchUserByName is accessible via /api/Users/search and has an ActionName for potentially generating URLs or for clarity in routing.
* GetUserById is accessible via /api/Users/{id} and also has an ActionName.

ActionName is primarily useful when:

* You want to expose an action with a different name than its method name.
* You have multiple actions with the same HTTP verb that would otherwise conflict in routing if not for different route templates or ActionName.
* When using Url.Action or CreatedAtAction in client-side code where you refer to the action by its ActionName.

**Demonstrating Creation of a Simple WebAPI - With Read, Write Actions**

We've already covered this extensively in the Swagger and Routing sections. The ProductsController example provided earlier already demonstrates common Read (GET) and Write (POST, DELETE) actions.

Here's a recap of the ProductsController for clarity, specifically focusing on Read/Write:

**Controllers/ProductsController.cs:**

C#

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

using System.Net; // For HttpStatusCode

namespace MyWebApiSwaggerDemo.Controllers

{

[ApiController]

[Route("api/[controller]")] // Base route for the controller: /api/Products

public class ProductsController : ControllerBase

{

// In a real application, this would be a database or repository

private static readonly List<Product> \_products = new List<Product>

{

new Product { Id = 1, Name = "Laptop", Price = 1200.00m },

new Product { Id = 2, Name = "Mouse", Price = 25.00m },

new Product { Id = 3, Name = "Monitor", Price = 300.00m }

};

// READ All Products

// GET /api/Products

[HttpGet]

[ProducesResponseType(typeof(IEnumerable<Product>), (int)HttpStatusCode.OK)]

public IActionResult GetProducts()

{

return Ok(\_products);

}

// READ Single Product by ID

// GET /api/Products/{id}

[HttpGet("{id}", Name = "GetProductById")] // Name for URL generation

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult GetProductById(int id)

{

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

if (product == null)

{

return NotFound($"Product with ID {id} not found.");

}

return Ok(product);

}

// WRITE (CREATE) a new Product

// POST /api/Products

[HttpPost]

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

public IActionResult AddProduct([FromBody] Product product)

{

if (product == null || string.IsNullOrWhiteSpace(product.Name))

{

return BadRequest("Product data is invalid.");

}

// Assign a new ID (in a real app, this would be handled by the database)

product.Id = \_products.Any() ? \_products.Max(p => p.Id) + 1 : 1;

\_products.Add(product);

// Return 201 Created and the URL to the newly created resource

return CreatedAtRoute("GetProductById", new { id = product.Id }, product);

}

// WRITE (UPDATE) an existing Product

// PUT /api/Products/{id}

[HttpPut("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult UpdateProduct(int id, [FromBody] Product updatedProduct)

{

if (updatedProduct == null || updatedProduct.Id != id || string.IsNullOrWhiteSpace(updatedProduct.Name))

{

return BadRequest("Invalid product data or ID mismatch.");

}

var existingProduct = \_products.FirstOrDefault(p => p.Id == id);

if (existingProduct == null)

{

return NotFound($"Product with ID {id} not found.");

}

existingProduct.Name = updatedProduct.Name;

existingProduct.Price = updatedProduct.Price;

return NoContent(); // 204 No Content for successful update

}

// WRITE (DELETE) a Product

// DELETE /api/Products/{id}

[HttpDelete("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult DeleteProduct(int id)

{

var productToRemove = \_products.FirstOrDefault(p => p.Id == id);

if (productToRemove == null)

{

return NotFound($"Product with ID {id} not found.");

}

\_products.Remove(productToRemove);

return NoContent(); // 204 No Content for successful deletion

}

}

// Model class

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

This controller demonstrates the core CRUD (Create, Read, Update, Delete) operations which are the fundamental "Read" and "Write" actions for a Web API.

To get this information into a PDF, you can simply copy the entire response from this chat and paste it into a document editor of your choice (e.g., Google Docs, Microsoft Word, LibreOffice Writer). From there, you can use the "Print to PDF" or "Save as PDF" option to create a PDF file.

I can provide you with the information and steps to achieve your objectives, but I cannot directly "make it a PDF of all files and send to me." My capabilities are limited to generating text-based responses.

However, you can easily copy and paste the information I provide into a document editor (like Microsoft Word, Google Docs, or LibreOffice Writer) and then save or print it as a PDF yourself.

Here's a breakdown of the demonstrations you requested:

**Demonstrating Swagger Installation and Usage with WebAPI**

Swagger (OpenAPI) provides a standardized, language-agnostic interface to REST APIs, allowing both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or network traffic inspection. Swashbuckle.AspNetCore is a popular library for integrating Swagger into ASP.NET Core Web APIs.

**1. Nuget Package to Download: Swashbuckle.AspNetCore**

**Steps:**

1. **Create a new ASP.NET Core Web API project:**
   * Open Visual Studio.
   * Select "Create a new project."
   * Search for and select "ASP.NET Core Web API."
   * Click "Next," provide a name for your project (e.g., MyWebApiSwaggerDemo), and click "Create."
2. **Install the Swashbuckle.AspNetCore NuGet Package:**
   * In Visual Studio, right-click on your project in Solution Explorer.
   * Select "Manage NuGet Packages..."
   * Go to the "Browse" tab.
   * Search for Swashbuckle.AspNetCore.
   * Select the latest stable version and click "Install."

**2. Usage of ProducesResponseType to Web API Method**

[ProducesResponseType] attribute improves the Swagger documentation by explicitly declaring the possible HTTP response types and their associated data models for an API action.

**Example:**

Modify a controller action (e.g., WeatherForecastController.cs or create a new one):

C#

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

using System.Net; // For HttpStatusCode

namespace MyWebApiSwaggerDemo.Controllers

{

[ApiController]

[Route("[controller]")]

public class ProductsController : ControllerBase

{

private static readonly List<Product> \_products = new List<Product>

{

new Product { Id = 1, Name = "Laptop", Price = 1200.00m },

new Product { Id = 2, Name = "Mouse", Price = 25.00m }

};

[HttpGet]

[ProducesResponseType(typeof(IEnumerable<Product>), (int)HttpStatusCode.OK)] // Success

[ProducesResponseType((int)HttpStatusCode.NotFound)] // Not found (if applicable)

public IActionResult GetProducts()

{

if (!\_products.Any())

{

return NotFound();

}

return Ok(\_products);

}

[HttpGet("{id}")]

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult GetProductById(int id)

{

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

if (product == null)

{

return NotFound();

}

return Ok(product);

}

[HttpPost]

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

public IActionResult AddProduct([FromBody] Product product)

{

if (product == null || string.IsNullOrWhiteSpace(product.Name))

{

return BadRequest("Product data is invalid.");

}

product.Id = \_products.Max(p => p.Id) + 1;

\_products.Add(product);

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

}

[HttpDelete("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult DeleteProduct(int id)

{

var productToRemove = \_products.FirstOrDefault(p => p.Id == id);

if (productToRemove == null)

{

return NotFound();

}

\_products.Remove(productToRemove);

return NoContent();

}

}

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

**3. AddSwaggerGen and UseSwaggerUI**

These methods are crucial for configuring Swagger generation and enabling the Swagger UI in your ASP.NET Core application.

**Steps:**

1. **Configure Startup.cs (for .NET 5.0 and earlier) or Program.cs (for .NET 6.0 and later):**

**For .NET 5.0 and earlier (Startup.cs):**

C#

// Inside Startup.cs

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

// Add SwaggerGen service

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "My Web API", Version = "v1" });

});

}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

// Enable Swagger and Swagger UI

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My Web API V1");

// To serve Swagger UI at application's root (http://localhost:<port>/)

// c.RoutePrefix = string.Empty;

});

}

app.UseRouting();

app.UseAuthorization();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

});

}

**For .NET 6.0 and later (Program.cs):**

C#

// Inside Program.cs

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "My Web API", Version = "v1" });

});

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "My Web API V1");

// To serve Swagger UI at application's root (http://localhost:<port>/)

// c.RoutePrefix = string.Empty;

});

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();

**4. WebAPI Listing on Browser**

**Steps:**

1. **Run your Web API project.**
   * Press F5 in Visual Studio or run dotnet run from your project directory in the terminal.
2. **Navigate to the Swagger UI in your browser:**
   * Once the application starts, it will typically open a browser tab. If you haven't changed c.RoutePrefix, the Swagger UI will be available at http://localhost:<port>/swagger.
   * You should see a well-formatted page listing all your API endpoints, their methods, parameters, and the ProducesResponseType information you added. You can also test your API directly from this interface.

**Demonstrating the Usage of Postman Tool to Hit WebAPI Methods**

Postman is a popular API client that makes it easy to create, share, test, and document APIs.

**1. Structure in Postman Tool**

* **Sidebar:** Contains Collections, APIs, Environments, Mock Servers, Monitors, and History.
* **Workspace:** The main area where you work on requests.
* **Request Tab:** Each tab represents an individual API request.
* **Request Builder:** Where you define your HTTP method, URL, headers, and body.
* **Response Viewer:** Displays the response from the server.

**2. Headers with Authorization**

Headers provide meta-information about the request or response. Authorization headers are crucial for securing APIs.

**Steps:**

1. **Open Postman.**
2. **Create a New Request:** Click the + button on the tabs bar or "File" > "New" > "Request."
3. **Enter the API URL:** For example, http://localhost:<port>/Products.
4. **Select HTTP Method:** Choose GET, POST, PUT, DELETE, etc.
5. **Add Authorization Header:**
   * Go to the "Headers" tab in the Request Builder.
   * Add a Key: Authorization
   * Add a Value: Bearer <Your\_JWT\_Token> (if using JWT authentication) or Basic <base64\_encoded\_username:password> (for basic authentication).
   * **Alternatively, use the "Authorization" tab:** Postman provides a dedicated "Authorization" tab where you can select common authorization types (Bearer Token, Basic Auth, OAuth 2.0, etc.) and Postman will construct the header for you.

**3. Body as JSON**

For POST and PUT requests, you often send data in the request body, typically as JSON.

**Steps:**

1. **Select HTTP Method:** Choose POST or PUT.
2. **Go to the "Body" tab.**
3. **Select "raw" and then "JSON" from the dropdown.**
4. **Enter your JSON payload** in the text area.

**Example (for POST /Products):**

JSON

{

"name": "Keyboard",

"price": 75.50

}

**4. Option to Choose the Type of Request**

This refers to the HTTP verb (GET, POST, PUT, DELETE, PATCH, OPTIONS, HEAD). It's a dropdown menu right next to the URL input field in the Request Builder.

**5. Request Collection and How to Add a New Request in the Collection**

Collections allow you to organize your API requests into folders, making it easier to manage and share them.

**Steps:**

1. **Create a New Collection:**
   * In the sidebar, click on "Collections."
   * Click the + icon next to "Collections" or the "New Collection" button.
   * Give it a name (e.g., "My Web API Tests").
2. **Add a Request to a Collection:**
   * Once you have a request open in a tab, click the "Save" button.
   * In the "SAVE REQUEST" dialog, choose your created collection from the dropdown.
   * Give the request a name (e.g., "Get All Products").
   * Click "Save."
   * Alternatively, right-click on your collection in the sidebar and select "Add Request."

**6. Tabs in the Center Pane that Corresponds to the Request**

* Each time you open a saved request from a collection or create a new request, it opens in a new tab in the center pane.
* This allows you to work on multiple API requests simultaneously without losing your work.
* You can easily switch between tabs to manage different requests.

**Demonstrating the Usage of Route and Explain Name Attribute in Http Requests**

Routing in ASP.NET Core maps incoming HTTP requests to action methods on controllers.

**1. Importance of User-Friendly Name to Action Method**

The Name attribute in routing allows you to define a user-friendly name for a route. This name can then be used to generate URLs for that route without hardcoding the URL path, making your application more robust to URL changes.

**Example:**

Consider a ProductsController with an action to get a product by ID:

C#

[ApiController]

[Route("api/[controller]")] // Controller-level route prefix

public class ProductsController : ControllerBase

{

// ... (Product class and \_products list from previous example)

[HttpGet("{id}", Name = "GetProductByIdRoute")] // Route template and Name attribute

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult GetProduct(int id)

{

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

if (product == null)

{

return NotFound();

}

return Ok(product);

}

[HttpPost(Name = "AddProductRoute")] // Route template and Name attribute

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

public IActionResult CreateProduct([FromBody] Product product)

{

if (product == null || string.IsNullOrWhiteSpace(product.Name))

{

return BadRequest("Product data is invalid.");

}

product.Id = \_products.Max(p => p.Id) + 1;

\_products.Add(product);

// Using CreatedAtRoute with the route name for URL generation

return CreatedAtRoute("GetProductByIdRoute", new { id = product.Id }, product);

}

}

**Benefits of Name attribute:**

* **Decoupling:** If the URL path for GetProductById changes (e.g., from api/Products/{id} to api/V2/Products/{id}), you only need to update the Route attribute. Any code that uses CreatedAtRoute("GetProductByIdRoute", ...) will automatically generate the correct URL.
* **Readability:** Using a descriptive name can make your code easier to understand.
* **Testability:** Helps in testing scenarios where you need to verify URL generation.

**2. Explain the Usage of ActionName to Have More Than 1 Method with the Same Action Verb**

By default, ASP.NET Core routing uses the action method's name to match incoming requests. However, sometimes you might want to have multiple methods with the same HTTP verb (e.g., two GET methods) but differentiate them based on their purpose or parameters, without changing their actual method name. The [ActionName] attribute allows you to assign a different name for routing purposes.

**Example:**

C#

[ApiController]

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

public class UsersController : ControllerBase

{

private static readonly List<string> \_users = new List<string> { "Alice", "Bob", "Charlie" };

[HttpGet] // Matches GET /api/Users

public IActionResult GetUsers()

{

return Ok(\_users);

}

[HttpGet("search")] // Matches GET /api/Users/search?name=...

[ActionName("FindUserByName")] // Route name for this action

public IActionResult SearchUserByName([FromQuery] string name)

{

var user = \_users.FirstOrDefault(u => u.Contains(name, StringComparison.OrdinalIgnoreCase));

if (user == null)

{

return NotFound();

}

return Ok(user);

}

[HttpGet("{id:int}")] // Matches GET /api/Users/{id}

[ActionName("GetUserById")] // Route name for this action

public IActionResult GetUserById(int id)

{

// This is a simplified example, usually you'd retrieve by ID from a data store

if (id >= 0 && id < \_users.Count)

{

return Ok(\_users[id]);

}

return NotFound();

}

}

In this example:

* We have three GET methods.
* GetUsers is the default GET for the controller.
* SearchUserByName is accessible via /api/Users/search and has an ActionName for potentially generating URLs or for clarity in routing.
* GetUserById is accessible via /api/Users/{id} and also has an ActionName.

ActionName is primarily useful when:

* You want to expose an action with a different name than its method name.
* You have multiple actions with the same HTTP verb that would otherwise conflict in routing if not for different route templates or ActionName.
* When using Url.Action or CreatedAtAction in client-side code where you refer to the action by its ActionName.

**Demonstrating Creation of a Simple WebAPI - With Read, Write Actions**

We've already covered this extensively in the Swagger and Routing sections. The ProductsController example provided earlier already demonstrates common Read (GET) and Write (POST, DELETE) actions.

Here's a recap of the ProductsController for clarity, specifically focusing on Read/Write:

**Controllers/ProductsController.cs:**

C#

using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;

using System.Linq;

using System.Net; // For HttpStatusCode

namespace MyWebApiSwaggerDemo.Controllers

{

[ApiController]

[Route("api/[controller]")] // Base route for the controller: /api/Products

public class ProductsController : ControllerBase

{

// In a real application, this would be a database or repository

private static readonly List<Product> \_products = new List<Product>

{

new Product { Id = 1, Name = "Laptop", Price = 1200.00m },

new Product { Id = 2, Name = "Mouse", Price = 25.00m },

new Product { Id = 3, Name = "Monitor", Price = 300.00m }

};

// READ All Products

// GET /api/Products

[HttpGet]

[ProducesResponseType(typeof(IEnumerable<Product>), (int)HttpStatusCode.OK)]

public IActionResult GetProducts()

{

return Ok(\_products);

}

// READ Single Product by ID

// GET /api/Products/{id}

[HttpGet("{id}", Name = "GetProductById")] // Name for URL generation

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult GetProductById(int id)

{

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

if (product == null)

{

return NotFound($"Product with ID {id} not found.");

}

return Ok(product);

}

// WRITE (CREATE) a new Product

// POST /api/Products

[HttpPost]

[ProducesResponseType(typeof(Product), (int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

public IActionResult AddProduct([FromBody] Product product)

{

if (product == null || string.IsNullOrWhiteSpace(product.Name))

{

return BadRequest("Product data is invalid.");

}

// Assign a new ID (in a real app, this would be handled by the database)

product.Id = \_products.Any() ? \_products.Max(p => p.Id) + 1 : 1;

\_products.Add(product);

// Return 201 Created and the URL to the newly created resource

return CreatedAtRoute("GetProductById", new { id = product.Id }, product);

}

// WRITE (UPDATE) an existing Product

// PUT /api/Products/{id}

[HttpPut("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.BadRequest)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult UpdateProduct(int id, [FromBody] Product updatedProduct)

{

if (updatedProduct == null || updatedProduct.Id != id || string.IsNullOrWhiteSpace(updatedProduct.Name))

{

return BadRequest("Invalid product data or ID mismatch.");

}

var existingProduct = \_products.FirstOrDefault(p => p.Id == id);

if (existingProduct == null)

{

return NotFound($"Product with ID {id} not found.");

}

existingProduct.Name = updatedProduct.Name;

existingProduct.Price = updatedProduct.Price;

return NoContent(); // 204 No Content for successful update

}

// WRITE (DELETE) a Product

// DELETE /api/Products/{id}

[HttpDelete("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.NotFound)]

public IActionResult DeleteProduct(int id)

{

var productToRemove = \_products.FirstOrDefault(p => p.Id == id);

if (productToRemove == null)

{

return NotFound($"Product with ID {id} not found.");

}

\_products.Remove(productToRemove);

return NoContent(); // 204 No Content for successful deletion

}

}

// Model class

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

}

}

This controller demonstrates the core CRUD (Create, Read, Update, Delete) operations which are the fundamental "Read" and "Write" actions for a Web API.

To get this information into a PDF, you can simply copy the entire response from this chat and paste it into a document editor of your choice (e.g., Google Docs, Microsoft Word, LibreOffice Writer). From there, you can use the "Print to PDF" or "Save as PDF" option to create a PDF file.