# **ASP.NET Core Middleware Pipeline**

#### **1. Authentication Middleware**

**Purpose**: Validates user credentials like JWT tokens.  
**Usage**: Ensures only authenticated users can access protected endpoints.

builder.Services.AddAuthentication("Bearer").AddJwtBearer(options =>  
{  
 options.Authority = "https://your-auth-provider.com";  
 options.Audience = "your-api";  
});  
app.UseAuthentication();

#### **2. Authorization Middleware**

**Purpose**: Verifies if authenticated users have the necessary permissions (e.g., roles, claims).  
**Usage**: Enables role-based or policy-based access control.

builder.Services.AddAuthorization(options =>  
{  
 options.AddPolicy("AdminOnly", policy => policy.RequireRole("Admin"));  
});  
app.UseAuthorization();

#### **3. CORS Middleware**

**Purpose**: Controls which domains can access your API resources.  
**Usage**: Prevents unauthorized cross-origin requests.

builder.Services.AddCors(options =>  
{  
 options.AddPolicy("AllowSpecificOrigin", policy =>  
 {  
 policy.WithOrigins("https://mywebsite.com").AllowAnyHeader().AllowAnyMethod();  
 });  
});  
app.UseCors("AllowSpecificOrigin");

#### **4. Exception Handling Middleware**

**Purpose**: Handles unhandled exceptions and provides consistent error responses.  
**Usage**: Redirects to an error handler or generates inline error responses.

app.UseExceptionHandler("/error");  
app.Map("/error", (HttpContext context) =>  
{  
 var exception = context.Features.Get<IExceptionHandlerFeature>()?.Error;  
 return Results.Problem(title: "An error occurred", detail: exception?.Message);  
});

#### **5. Logging Middleware**

**Purpose**: Logs request and response details for debugging and monitoring.  
**Usage**: Tracks incoming requests and their responses.

app.UseMiddleware<LoggingMiddleware>();  
app.UseRequestLogging();  
app.Use(async (context, next) => { Console.WriteLine($"Request: {context.Request.Method} {context.Request.Path}"); await next(); });

#### **6. Short-Circuiting Middleware**

**Purpose**: Stops further processing of a request based on specific conditions.  
**Usage**: Rejects requests early, saving resources and improving security.

app.UseMiddleware<ShortCircuitMiddleware>();

#### **7. Routing Middleware**

**Purpose**: Routes incoming requests to the appropriate controllers or endpoints.  
**Usage**: Essential for all ASP.NET Core applications to define and resolve routes.

app.UseRouting();  
app.UseEndpoints(endpoints =>  
{  
 endpoints.MapControllers();  
});

#### **8. Static Files Middleware**

**Purpose**: Serves static files like HTML, CSS, JavaScript, and images.  
**Usage**: Provides access to files in the wwwroot folder or a custom directory.

app.UseStaticFiles();

#### **9. Swagger Middleware**

**Purpose**: Provides API documentation and an interactive testing interface.  
**Usage**: Generates and displays Swagger UI for API exploration.

if (app.Environment.IsDevelopment())  
{  
 app.UseSwagger();  
 app.UseSwaggerUI();  
}

# **Routing in ASP.NET Core Web API**

1. **Routing Basics**:
   * Routing maps incoming HTTP requests to appropriate controller actions.
   * Defined in Program.cs or directly on controllers using attributes.
2. **Types of Routing**:
   * **Conventional Routing**: Centralized route configuration in Program.cs.
   * app.MapControllerRoute(  
      name: "default",  
      pattern: "{controller=Home}/{action=Index}/{id?}");
   * **Attribute Routing**: Defined at the controller or action level using attributes like [Route], [HttpGet], etc.
   * [Route("api/[controller]")]  
     [ApiController]  
     public class HomeController : ControllerBase  
     {  
      [HttpGet("{id}")]  
      public string Get(int id) => "value";  
     }
3. **Route Parameters**:
   * **Required**: Defined as {parameterName}, e.g., {id}.
   * **Optional**: Defined as {parameterName?}, e.g., {id?}.
   * **Constraints**: Enforce specific formats, e.g., {id:int} (integer).
4. **HTTP Method Mapping**:
   * Actions can be mapped to HTTP methods:
     + [HttpGet] for GET.
     + [HttpPost] for POST.
     + [HttpPut] for PUT.
     + [HttpDelete] for DELETE.
5. **Swagger Integration**:
   * Automatically documents and exposes API endpoints.
   * Can be disabled by removing app.UseSwagger() and app.UseSwaggerUI() in Program.cs.

# **Filters in ASP.NET Core Overview**

### **Types of Filters in ASP.NET Core**

1. **Authorization Filters**
   * Run before any other filter and determine whether the user is authorized to access the requested resource.
   * **Use Case:** Check if the user is authenticated before proceeding with the action.

* public class CustomAuthorizationFilter : IAuthorizationFilter  
  {  
   public void OnAuthorization(AuthorizationFilterContext context)  
   {  
   if (!context.HttpContext.User.Identity.IsAuthenticated)  
   {  
   context.Result = new UnauthorizedResult();  
   }  
   }  
  }

1. **Resource Filters**
   * Run after authorization filters but before model binding, useful for caching or resource initialization.
   * **Use Case:** Initialize resources or cache data before executing the action.

* public class CustomResourceFilter : IResourceFilter  
  {  
   public void OnResourceExecuting(ResourceExecutingContext context) { }  
   public void OnResourceExecuted(ResourceExecutedContext context) { }  
  }

1. **Action Filters**
   * Run before and after an action method executes. Typically used for logging, validation, or performance tracking.
   * **Use Case:** Log method execution time or validate model data.

* public class CustomActionFilter : IActionFilter  
  {  
   public void OnActionExecuting(ActionExecutingContext context)  
   {  
   // Logic before action method execution  
   }  
    
   public void OnActionExecuted(ActionExecutedContext context)  
   {  
   // Logic after action method execution  
   }  
  }

1. **Exception Filters**
   * Handle exceptions thrown during action execution and return a custom error response.
   * **Use Case:** Return a standardized error response for unhandled exceptions.

* public class CustomExceptionFilter : IExceptionFilter  
  {  
   public void OnException(ExceptionContext context)  
   {  
   context.Result = new ObjectResult("An error occurred") { StatusCode = 500 };  
   }  
  }

1. **Result Filters**
   * Run before and after the result is executed. Used for modifying or processing the response.
   * **Use Case:** Modify the response before it is sent to the client.

* public class CustomResultFilter : IResultFilter  
  {  
   public void OnResultExecuting(ResultExecutingContext context) { }  
   public void OnResultExecuted(ResultExecutedContext context) { }  
  }

### **Example of Registering Filters in Program.cs**

var builder = WebApplication.CreateBuilder(args);  
  
// Register custom filters  
builder.Services.AddScoped<CustomAuthorizationFilter>();  
builder.Services.AddScoped<CustomActionFilter>();  
builder.Services.AddScoped<CustomExceptionFilter>();  
builder.Services.AddScoped<CustomResourceFilter>();  
builder.Services.AddScoped<CustomResultFilter>();  
  
var app = builder.Build();  
  
// Apply global filters  
app.UseAuthorization();  
app.MapControllers();  
  
app.Run();

### **Applying Filters to Action Methods**

Filters can be applied to actions or controllers using attributes like [ServiceFilter] or [TypeFilter]:

[ServiceFilter(typeof(CustomActionFilter))]  
[HttpGet("api/action")]  
public IActionResult ActionEndpoint()  
{  
 return Ok("Action filter executed");  
}

### **Global Filter Registration**

You can register filters globally, ensuring they are applied to all actions:

builder.Services.AddControllers(options =>  
{  
 options.Filters.Add<CustomActionFilter>();  
});

# **Controller Initialization**

* **Controller Initialization** in ASP.NET Web API happens automatically when a request is made to an endpoint. The framework uses **Dependency Injection** (if configured) to inject dependencies into controllers.
* **Controller** inherits from ApiController, and ASP.NET Web API automatically creates instances of controllers based on route configuration or default routing conventions.
* **Default constructor**: If no constructor is defined, Web API uses a parameterless constructor to initialize the controller.

**Code Example**:

public class MyController : ApiController  
{  
 private readonly IMyService \_service;  
  
 // Custom constructor with dependency injection  
 public MyController(IMyService service)  
 {  
 \_service = service;  
 }  
  
 public MyController() { } // Parameterless constructor (default)  
}

# **Action Methods**

* **Action Methods** are methods inside a controller that handle specific HTTP requests (e.g., GET, POST, PUT, DELETE).
* **Routing**: Web API matches incoming requests to action methods based on the HTTP method (e.g., GET, POST) and route parameters.
* **HTTP Attributes**: Action methods are decorated with attributes like [HttpGet], [HttpPost], etc., to specify which HTTP verbs they respond to.
* **Return Types**: Action methods generally return IHttpActionResult, which provides flexibility in returning different HTTP status codes and data.

**Code Example**:

public class ProductsController : ApiController  
{  
 // GET api/products  
 [HttpGet]  
 public IHttpActionResult GetAllProducts()  
 {  
 var products = new string[] { "Product 1", "Product 2" };  
 return Ok(products); // Returns 200 OK  
 }  
  
 // POST api/products  
 [HttpPost]  
 public IHttpActionResult AddProduct([FromBody] string product)  
 {  
 if (string.IsNullOrEmpty(product))  
 return BadRequest("Product name cannot be empty");  
  
 // Logic to add product (e.g., saving to database)  
 return CreatedAtRoute("DefaultApi", new { id = 1 }, product); // Returns 201 Created  
 }  
}

### **Key Points**:

* **Controller Initialization**:
  + Controllers are instantiated by the Web API framework when a request is received.
  + Custom constructors can be used for Dependency Injection of services or other dependencies.
* **Action Methods**:
  + **Return IHttpActionResult**: Ensures flexibility in returning various HTTP status codes and responses.
  + **Use Attributes** like [HttpGet], [HttpPost], etc., to map methods to HTTP verbs.
  + Parameters can be extracted from the request body, query strings, or route data.