.Net Best Practices

This document outlines the best practices for building a robust, scalable, and maintainable Web API in **.NET 8**. It includes practices for structuring the application using **Clean Architecture**, utilizing **Object-Oriented Programming (OOP)** principles, setting up **Dependency Injection (DI)** with extension methods, integrating **middleware**, **logging**, **global exception handling**, **asynchronous programming**, **IDisposable** usage, **Swagger annotations**, and **HttpContextAccessor** for request-specific data, HttpNamedClient etc…

Prepared by: Abdul Razak

1. Clean Architecture

Clean Architecture enforces separation of concerns, allowing you to structure your Web API into layers that ensure maintainability and testability.

Typical Layered Structure:

```
/src
  /MyApp.Api           # API Project (Controllers, Middleware, etc.)
  /MyApp.Application   # Application Layer (Services, DTOs, etc.)
  /MyApp.Domain        # Domain Layer (Entities, Logic, Interfaces)
  /MyApp.Infrastructure # Infrastructure Layer (Database, External Services)
  /MyApp.Tests         # Unit and Integration Tests
```

- **MyApp.Api**: Contains controllers, API-specific logic, middleware, and routing.
- **MyApp.Application**: Contains services, application logic, and DTOs for use cases.
- **MyApp.Domain**: Core business logic, entities, domain services, and interfaces.
- **MyApp.Infrastructure**: Data access, repository implementations, third-party service integrations.
- **MyApp.Tests**: Unit and integration tests for all layers.

Example: Domain Layer Entity

```
// MyApp.Domain/Entities/Product.cs
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

2. Object-Oriented Programming (OOP) Principles

Adhere to core OOP principles like **Encapsulation**, **Inheritance**, **Polymorphism**, and **Abstraction** for a scalable and maintainable API.

Use SOLID Principles

- **S**ingle Responsibility Principle: Each class should have one reason to change.
- **O**pen/Closed Principle: Classes should be open for extension but closed for modification.
- **L**iskov Substitution Principle: Objects should be replaceable with instances of their subtypes.
- **I**nterface Segregation Principle: Clients should not be forced to depend on interfaces they do not use.
- **D**ependency Inversion Principle: High-level modules should not depend on low-level modules, but both should depend on abstractions.

Example:

```
// Liskov Substitution Principle
public class Bird
{
    public virtual void Fly() { }
}

public class Duck : Bird
{
    public override void Fly() { Console.WriteLine("Duck flying."); }
}

public class Ostrich : Bird
{
    public override void Fly() { throw new
InvalidOperationException("Ostriches can't fly."); }
}
```

Example: OOP in Service Layer

```
// Application Layer - Product Service Interface
public interface IProductService
{
    Task<ProductDto> GetProductAsync(int id);
}

// Application Layer - Product Service Implementation
```

```
public class ProductService : IProductService
{
    private readonly IProductRepository _productRepository;

    public ProductService(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }

    public async Task<ProductDto> GetProductAsync(int id)
    {
        var product = await _productRepository.GetProductByIdAsync(id);
        return new ProductDto
        {
            Id = product.Id,
            Name = product.Name,
            Price = product.Price
        };
    }
}
```

3. Dependency Injection (DI) Registration with Extension Method

.NET Core's built-in DI container promotes loose coupling and testability. For cleaner code, use extension methods to register services.

Example: DI Registration Extension Method

```
// MyApp.Api/Extensions/ServiceExtensions.cs
public static class ServiceExtensions
{
    public static IServiceCollection AddApplicationServices(this
IServiceCollection services)
    {
        services.AddScoped<IProductService, ProductService>();
        services.AddScoped<IProductRepository, ProductRepository>();

        return services;
    }
```

```
}
```

Registering in `Program.cs` (or `Startup.cs` for earlier versions):

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

// Register services
builder.Services.AddApplicationServices();
builder.Services.AddDbContext<MyDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnec
tion")));

builder.Services.AddControllers();

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

var app = builder.Build();

app.UseSwagger();
app.UseSwaggerUI();

app.MapControllers();

app.Run();
```

4. Middleware

Middleware provides a way to handle cross-cutting concerns such as logging, exception handling, and authentication.

Example: Custom Logging Middleware

```
// MyApp.Api/Middleware/LoggingMiddleware.cs
public class LoggingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<LoggingMiddleware> _logger;

    public LoggingMiddleware(RequestDelegate next, ILogger<LoggingMiddleware>
logger)
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        _logger.LogInformation($"Request: {context.Request.Method}
{context.Request.Path}");
        await _next(context);
        _logger.LogInformation($"Response: {context.Response.StatusCode}");
    }
}
```

Registering Middleware in `Program.cs`

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

app.UseMiddleware<LoggingMiddleware>();  // Register custom middleware

app.UseRouting();
app.UseAuthorization();

app.MapControllers();

app.Run();
```

5. Logging

Leverage **ILogger** to log essential information for debugging, monitoring, and auditing API requests and actions.

Example: Logging in Controllers

```
public class ProductsController : ControllerBase
{
    private readonly ILogger<ProductsController> _logger;
    private readonly IProductService _productService;

    public ProductsController(ILogger<ProductsController> logger,
IProductService productService)
    {
        _logger = logger;
        _productService = productService;
    }

    [HttpGet("{id}")]
    public async Task<IActionResult> GetProduct(int id)
    {
        _logger.LogInformation("Fetching product with ID {Id}", id);
        var product = await _productService.GetProductAsync(id);

        if (product == null)
        {
            _logger.LogWarning("Product with ID {Id} not found", id);
            return NotFound();
        }

        return Ok(product);
    }
}
```

6. Global Exception Handling

Handling exceptions globally ensures consistent error responses and simplifies maintenance.

Example: Global Exception Middleware

```
// MyApp.Api/Middleware/GlobalExceptionMiddleware.cs
public class GlobalExceptionMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<GlobalExceptionMiddleware> _logger;

    public GlobalExceptionMiddleware(RequestDelegate next,
ILogger<GlobalExceptionMiddleware> logger)
    {
        _next = next;
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An unexpected error occurred");
            context.Response.StatusCode =
(int)HttpStatusCode.InternalServerError;
            await context.Response.WriteAsync("An unexpected error
occurred.");
        }
    }
}
```

Register Global Exception Middleware in `Program.cs`

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

app.UseMiddleware<GlobalExceptionMiddleware>();  // Global exception handler

app.MapControllers();
```

```
app.Run();
```

7. Asynchronous Programming

Using asynchronous methods for I/O-bound operations improves API performance by freeing up threads during database access, API calls, etc.

Example: Asynchronous Repository

```
public class ProductRepository : IProductRepository
{
    private readonly MyDbContext _context;

    public ProductRepository(MyDbContext context)
    {
        _context = context;
    }

    public async Task<Product> GetProductByIdAsync(int id)
    {
        return await _context.Products.FindAsync(id);
    }
}
```

Example: Asynchronous Controller Action

```
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(int id)
{
    var product = await _productService.GetProductAsync(id);
    if (product == null) return NotFound();
    return Ok(product);
}
```

8. IDisposable Usage

Ensure that classes managing unmanaged resources implement `IDisposable` to release those resources when no longer needed.

Example: DbContext with IDisposable

```
public class MyDbContext : DbContext, IDisposable
{
    public MyDbContext(DbContextOptions<MyDbContext> options) : base(options)
{ }

    public DbSet<Product> Products { get; set; }

    public new void Dispose()
    {
        base.Dispose();
    }
}
```

In most cases, DI will handle disposal automatically when services are registered with the correct lifetime (e.g., `Scoped` for DbContext).

9. HttpNamedClient Usage

In .NET 8, you can use **HTTP Named Clients** to configure and manage `HttpClient` instances in a more modular and reusable way. This allows you to set up multiple named clients with specific configurations, such as custom headers, base URLs, and timeouts, which are injected into your services through **Dependency Injection (DI)**. Named clients are particularly useful when you need to interact with different APIs or services with different configurations in a single application.

Steps to Use HTTP Named Clients in .NET 8

1. Setup the `HttpClient` with Named Clients

To configure named HTTP clients, you can use the `IHttpClientFactory` provided by ASP.NET Core. In .NET 8, you can add custom configuration to each named client to define different settings like timeouts, base addresses, or headers.

1.1. Add Named HTTP Clients in Program.cs

To define and configure your HTTP clients, you can register them in the `Program.cs` or `Startup.cs` file (depending on your project setup).

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Register HttpClient with named clients
        builder.Services.AddHttpClient();

        // Register named HttpClients with custom configurations
        builder.Services.AddHttpClient("GitHub", client =>
        {
            client.BaseAddress = new Uri("https://api.github.com/");

client.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClientFactory-
Sample");
            client.Timeout = TimeSpan.FromSeconds(30);
        });

        builder.Services.AddHttpClient("Weather", client =>
        {
            client.BaseAddress = new
Uri("https://api.openweathermap.org/data/2.5/");
            client.DefaultRequestHeaders.Add("API-KEY", "your-api-
```

```
key");
            client.Timeout = TimeSpan.FromSeconds(60);
        });

        var app = builder.Build();
        app.MapGet("/", () => "Hello World!");

        app.Run();
    }
}
```

In the example above:

- `AddHttpClient("GitHub", ...)`: Configures a named client for interacting with GitHub's API.
- `AddHttpClient("Weather", ...)`: Configures a named client for interacting with a weather API.

You can use these named clients wherever necessary.

1.2. Using Named Clients in Services

Once the named HTTP clients are registered, you can inject and use them in your services.

```
public class GitHubService
{
    private readonly HttpClient _httpClient;

    public GitHubService(IHttpClientFactory httpClientFactory)
    {
        _httpClient = httpClientFactory.CreateClient("GitHub"); //
Access named client
    }

    public async Task<string> GetUserReposAsync(string username)
    {
        var response = await
```

```
_httpClient.GetAsync($"users/{username}/repos");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

In this example, `GitHubService` is using the `GitHub` named HTTP client to make requests to the GitHub API.

2. Dependency Injection (DI) of Named Clients

To inject the `GitHubService` into controllers or other services, simply use DI as usual.

```
public class GitHubController : ControllerBase
{
    private readonly GitHubService _gitHubService;

    public GitHubController(GitHubService gitHubService)
    {
        _gitHubService = gitHubService;
    }

    [HttpGet("repos/{username}")]
    public async Task<IActionResult> GetRepos(string username)
    {
        var repos = await _gitHubService.GetUserReposAsync(username);
        return Ok(repos);
    }
}
```

3. Advanced Usage: Configuring HttpClient in an Extension Method

You can further abstract and organize your client configuration by creating extension methods to configure named clients.

```
public static class HttpClientExtensions
{
    public static IServiceCollection AddGitHubHttpClient(this
IServiceCollection services)
    {
        services.AddHttpClient("GitHub", client =>
        {
            client.BaseAddress = new Uri("https://api.github.com/");

client.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClientFactory-
Sample");
            client.Timeout = TimeSpan.FromSeconds(30);
        });

        return services;
    }

    public static IServiceCollection AddWeatherHttpClient(this
IServiceCollection services)
    {
        services.AddHttpClient("Weather", client =>
        {
            client.BaseAddress = new
Uri("https://api.openweathermap.org/data/2.5/");
            client.DefaultRequestHeaders.Add("API-KEY", "your-api-
key");
            client.Timeout = TimeSpan.FromSeconds(60);
        });

        return services;
    }
}
```

Now, you can call these extension methods in `Program.cs`:

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddGitHubHttpClient();
        builder.Services.AddWeatherHttpClient();

        var app = builder.Build();
        app.MapGet("/", () => "Hello World!");
        app.Run();
    }
}
```

4. Handling Timeouts and Errors

You can also configure specific error handling strategies for named clients. For instance, retry policies and timeouts can be customized for individual clients.

```
builder.Services.AddHttpClient("GitHub", client =>
{
    client.BaseAddress = new Uri("https://api.github.com/");

client.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClientFactory-
Sample");
}).AddTransientHttpErrorPolicy(policyBuilder =>
policyBuilder.RetryAsync(3));
```

In this case, if the request fails, it will retry up to 3 times.

5. Using `IHttpClientFactory` in Background Services

You can use the `IHttpClientFactory` to create named clients in background services. For instance, if you need to call APIs periodically or in background tasks:

```
public class BackgroundService : BackgroundService
{
    private readonly IHttpClientFactory _httpClientFactory;

    public BackgroundService(IHttpClientFactory httpClientFactory)
    {
        _httpClientFactory = httpClientFactory;
    }

    protected override async Task ExecuteAsync(CancellationToken
stoppingToken)
    {
        var client = _httpClientFactory.CreateClient("GitHub");
        var response = await
client.GetStringAsync("users/octocat/repos");
        Console.WriteLine(response);
    }
}
```

6. Using `HttpContext` with Named Clients

If you need to customize a named client for each request (e.g., adding request-specific headers or tokens), you can use `IHttpContextAccessor` in combination with `HttpClient`.

```
public class CustomHeaderService
{
    private readonly IHttpClientFactory _httpClientFactory;
    private readonly IHttpContextAccessor _httpContextAccessor;

    public CustomHeaderService(IHttpClientFactory httpClientFactory,
IHttpContextAccessor httpContextAccessor)
    {
        _httpClientFactory = httpClientFactory;
        _httpContextAccessor = httpContextAccessor;
    }

    public async Task<string> GetCustomData()
```

```
    {
        var client = _httpClientFactory.CreateClient("GitHub");

        // Add custom header for the current request
        var token =
_httpContextAccessor.HttpContext.Request.Headers["Authorization"];
        client.DefaultRequestHeaders.Add("Authorization", token);

        var response = await
client.GetStringAsync("users/octocat/repos");
        return response;
    }
}
```

You can combine **HTTP Named Clients** with **`DelegatingHandler`** and **`MessageHandler`** to extend the functionality of your `HttpClient`. These handlers are used to intercept and modify HTTP requests and responses, providing a powerful mechanism for adding cross-cutting concerns such as logging, authentication, and caching.

Key Concepts:

- **`DelegatingHandler`**: A base class that allows you to create custom handlers in the HTTP request pipeline. It is used for modifying or processing HTTP requests and responses.
- **`MessageHandler`**: The base class of `DelegatingHandler`. It is used directly to build the pipeline of message handlers, where `DelegatingHandler` acts as a wrapper for the actual request processing.

By adding **`DelegatingHandler`** to an HTTP client, you can create a chain of handlers that execute in a specific order for each HTTP request.

Steps to Use Named Clients with `DelegatingHandler` in .NET 8

- ### Create a Custom `DelegatingHandler`

First, you create a custom `DelegatingHandler` that can intercept and modify the HTTP request or response.

For example, you could create a logging handler that logs the HTTP request and response details:

```
public class LoggingHandler : DelegatingHandler
{
    private readonly ILogger<LoggingHandler> _logger;

    public LoggingHandler(ILogger<LoggingHandler> logger)
    {
        _logger = logger;
    }

    protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken
cancellationToken)
    {
        // Log request details
        _logger.LogInformation("Request: {Method} {Uri}",
request.Method, request.RequestUri);

        // Call the inner handler (next handler in the pipeline)
        var response = await base.SendAsync(request,
cancellationToken);

        // Log response details
        _logger.LogInformation("Response: {StatusCode}",
response.StatusCode);

        return response;
    }
}
```

- **Register the DelegatingHandler and Named Clients**

Now, register the custom handler (`LoggingHandler`) and the named HTTP clients in the `Program.cs` or `Startup.cs` file.

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Register the custom handler
        builder.Services.AddTransient<LoggingHandler>();

        // Register HttpClient with Named Clients and add the
LoggingHandler to the pipeline
        builder.Services.AddHttpClient("GitHub", client =>
        {
            client.BaseAddress = new Uri("https://api.github.com/");

client.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClientFactory-
Sample");
        })
        .AddHttpMessageHandler<LoggingHandler>(); // Add
LoggingHandler to the pipeline

        builder.Services.AddHttpClient("Weather", client =>
        {
            client.BaseAddress = new
Uri("https://api.openweathermap.org/data/2.5/");
            client.DefaultRequestHeaders.Add("API-KEY", "your-api-
key");
        })
        .AddHttpMessageHandler<LoggingHandler>(); // Add
LoggingHandler to the pipeline

        var app = builder.Build();
        app.MapGet("/", () => "Hello World!");

        app.Run();
    }
}
```

In this example:

- The `LoggingHandler` is added to the pipeline for both named clients (`GitHub` and `Weather`).
- You use `AddHttpMessageHandler<LoggingHandler>()` to inject the handler into the HTTP request pipeline.

- ## Using Named Clients with DelegatingHandler in Services

Now, you can inject and use the named HTTP client in services. When the client makes an HTTP request, the `LoggingHandler` will be invoked as part of the HTTP pipeline.

```
public class GitHubService
{
    private readonly HttpClient _httpClient;

    public GitHubService(IHttpClientFactory httpClientFactory)
    {
        _httpClient = httpClientFactory.CreateClient("GitHub"); //
Access the named client
    }

    public async Task<string> GetUserReposAsync(string username)
    {
        var response = await
_httpClient.GetAsync($"users/{username}/repos");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

In the `GitHubService`, the named client `GitHub` is used, and it automatically uses the `LoggingHandler` because the handler was added during client registration.

- **Add Multiple Handlers in the Pipeline**

You can chain multiple handlers together in the pipeline. Handlers are executed in the order they are added to the pipeline. For example, you might add a custom authentication handler and a logging handler.

```
public class AuthenticationHandler : DelegatingHandler
{
    private readonly string _authToken;

    public AuthenticationHandler(string authToken)
    {
        _authToken = authToken;
    }

    protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken
cancellationToken)
    {
        // Add the authentication token to the request headers
        request.Headers.Authorization = new
System.Net.Http.Headers.AuthenticationHeaderValue("Bearer",
_authToken);

        // Call the next handler in the pipeline
        return await base.SendAsync(request, cancellationToken);
    }
}
```

You can register multiple handlers for the same named client:

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Register custom handlers
        builder.Services.AddTransient<LoggingHandler>();
```

```
        builder.Services.AddTransient<AuthenticationHandler>();

        // Register HttpClient with Named Clients and add handlers to
the pipeline
        builder.Services.AddHttpClient("GitHub", client =>
        {
            client.BaseAddress = new Uri("https://api.github.com/");

client.DefaultRequestHeaders.UserAgent.ParseAdd("HttpClientFactory-
Sample");
        })
        .AddHttpMessageHandler<LoggingHandler>()  // Add
LoggingHandler
        .AddHttpMessageHandler<AuthenticationHandler>(); // Add
AuthenticationHandler

        builder.Services.AddHttpClient("Weather", client =>
        {
            client.BaseAddress = new
Uri("https://api.openweathermap.org/data/2.5/");
            client.DefaultRequestHeaders.Add("API-KEY", "your-api-
key");
        })
        .AddHttpMessageHandler<LoggingHandler>()
        .AddHttpMessageHandler<AuthenticationHandler>();

        var app = builder.Build();
        app.MapGet("/", () => "Hello World!");

        app.Run();
    }
}
```

In this example:

- The `GitHub` client will use both the `LoggingHandler` and `AuthenticationHandler` in that order.
- The `Weather` client will also use both handlers.

- ## Custom Message Handlers for Specific Needs

You can create more sophisticated handlers to meet specific needs, such as caching, retries, or handling specific HTTP status codes.

For instance, a **Retry Handler**:

```
public class RetryHandler : DelegatingHandler
{
    private readonly int _maxRetries;

    public RetryHandler(int maxRetries)
    {
        _maxRetries = maxRetries;
    }

    protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken
cancellationToken)
    {
        int attempt = 0;
        while (attempt < _maxRetries)
        {
            try
            {
                var response = await base.SendAsync(request,
cancellationToken);
                if (response.IsSuccessStatusCode)
                {
                    return response;
                }
                else
                {
                    attempt++;
                    await Task.Delay(1000); // Wait 1 second before
retrying
                }
            }
            catch
```

```
            {
                attempt++;
                if (attempt >= _maxRetries)
                {
                    throw;
                }
                await Task.Delay(1000);
            }
        }

        return new
HttpResponseMessage(System.Net.HttpStatusCode.RequestTimeout);
    }
}
```

You can then add the retry handler to the client:

```
builder.Services.AddHttpClient("GitHub", client =>
{
    client.BaseAddress = new Uri("https://api.github.com/");
})
.AddHttpMessageHandler<LoggingHandler>()
.AddHttpMessageHandler<AuthenticationHandler>()
.AddHttpMessageHandler<RetryHandler>(); // Add RetryHandler
```

Conclusion

Using **Named Clients with `DelegatingHandler`** and **`MessageHandler`** in .NET 8 provides a powerful way to modularize and manage HTTP request and response handling across different clients. By combining these handlers, you can apply cross-cutting concerns like logging, authentication, retries, and error handling in a clean and efficient way. The `HttpClientFactory` along with **`DelegatingHandler`** allows for greater flexibility in configuring HTTP clients without scattering logic throughout your application.

10. Swagger Annotations

Swagger (OpenAPI) is used for documenting your API and making it easier to test and understand.

Install Swashbuckle for Swagger:

```
dotnet add package Swashbuckle.AspNetCore
```

Configure Swagger in `Program.cs`

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

Adding Swagger Annotations to API Methods

```
/// <summary>
/// Retrieves a product by its ID.
/// </summary>
/// <param name="id">The product ID.</param>
/// <returns>A single product.</returns>
/// <response code="200">Returns the product.</response>
/// <response code="404">Product not found.</response>
[HttpGet("{id}")]
[ProducesResponseType(typeof(ProductDto), 200)]
[ProducesResponseType(404)]
public async Task<IActionResult> GetProduct(int id)
{
    var product = await _productService.GetProductAsync(id);
    if (product == null) return NotFound();
    return Ok(product);
}
```

Swagger Operation Annotation

```
/// <summary>
/// Creates a new product.
```

```
/// </summary>
/// <param name="productDto">The product data.</param>
/// <returns>The created product.</returns>
[HttpPost]
[ProducesResponseType(typeof(ProductDto), 201)]
[ProducesResponseType(400)]
[SwaggerOperation(Summary = "Creates a new product", Description = "This
endpoint allows users to create a new product.")]
public async Task<IActionResult> CreateProduct([FromBody] ProductDto
productDto)
{
    var product = await _productService.CreateProductAsync(productDto);
    return CreatedAtAction(nameof(GetProduct), new { id = product.Id },
product);
}
```

10. HttpContextAccessor Usage

Use `IHttpContextAccessor` to access `HttpContext` in non-controller classes (e.g., in services or middleware).

Example: Using `IHttpContextAccessor`

```
public class UserContextService
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    public UserContextService(IHttpContextAccessor httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }

    public string GetCurrentUserId()
    {
        return
_httpContextAccessor.HttpContext?.User?.FindFirst(ClaimTypes.NameIdentifier)?
.Value;
    }
}
```

Register `IHttpContextAccessor` in DI

```
builder.Services.AddHttpContextAccessor(); // Register IHttpContextAccessor
builder.Services.AddScoped<UserContextService

();
```

Design Patterns in .NET Core: Creational, Structural, and Behavioral Patterns

Design patterns are proven solutions to recurring design problems in software development. In this document, we explore **Creational**, **Structural**, and **Behavioral** design patterns in the context of .NET Core. These patterns help create scalable, maintainable, and flexible applications by organizing the system's components in a well-structured manner.

1. Creational Design Patterns in .NET Core

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. These patterns aim to control object creation in a way that is more flexible and efficient.

1.1. Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

Use Case: Useful for services like logging, configuration, and database connections.

Example:

```
public class Logger
{
    private static Logger _instance;
    private static readonly object _lock = new object();

    private Logger() { }

    public static Logger Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new Logger();
                }
                return _instance;
            }
        }
    }

    public void Log(string message)
    {
```

```
        Console.WriteLine($"Log: {message}");
    }
}
```

Usage:

```
Logger.Instance.Log("Application started");
```

1.2. Factory Method Pattern

The Factory Method pattern defines an interface for creating objects, but allows subclasses to alter the type of objects that will be created.

Use Case: Useful for creating different product types, such as varying document formats or shapes.

Example:

```
public abstract class Product
{
    public abstract void Display();
}

public class ConcreteProductA : Product
{
    public override void Display()
    {
        Console.WriteLine("Displaying Product A");
    }
}

public class ConcreteProductB : Product
{
    public override void Display()
    {
        Console.WriteLine("Displaying Product B");
```

```
    }
}

public abstract class Creator
{
    public abstract Product FactoryMethod();
}

public class ConcreteCreatorA : Creator
{
    public override Product FactoryMethod() => new ConcreteProductA();
}

public class ConcreteCreatorB : Creator
{
    public override Product FactoryMethod() => new ConcreteProductB();
}
```

Usage:

```
Creator creator = new ConcreteCreatorA();
Product product = creator.FactoryMethod();
product.Display(); // Displays Product A
```

1.3. Abstract Factory Pattern

Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Use Case: When you need to create a set of related objects that work together, e.g., UI frameworks.

Example:

```
// Abstract product families
public interface IButton
```

```
{
    void Render();
}

public interface ITextbox
{
    void Render();
}

// Concrete product families
public class WinButton : IButton
{
    public void Render() => Console.WriteLine("Rendering Windows
Button");
}

public class WinTextbox : ITextbox
{
    public void Render() => Console.WriteLine("Rendering Windows
Textbox");
}

public class MacButton : IButton
{
    public void Render() => Console.WriteLine("Rendering Mac Button");
}

public class MacTextbox : ITextbox
{
    public void Render() => Console.WriteLine("Rendering Mac
Textbox");
}

// Abstract Factory
public interface IUIFactory
{
    IButton CreateButton();
    ITextbox CreateTextbox();
}
```

```
// Concrete factories
public class WinFactory : IUIFactory
{
    public IButton CreateButton() => new WinButton();
    public ITextbox CreateTextbox() => new WinTextbox();
}

public class MacFactory : IUIFactory
{
    public IButton CreateButton() => new MacButton();
    public ITextbox CreateTextbox() => new MacTextbox();
}
```

Usage:

```
IUIFactory factory = new WinFactory();
IButton button = factory.CreateButton();
ITextbox textbox = factory.CreateTextbox();

button.Render();  // Rendering Windows Button
textbox.Render(); // Rendering Windows Textbox
```

2. Structural Design Patterns in .NET Core

Structural patterns deal with the composition of classes and objects to form larger structures. They focus on how to arrange components to ensure flexible and efficient designs.

2.1. Adapter Pattern

The Adapter pattern allows incompatible interfaces to work together by converting one interface to another.

Use Case: Useful when you need to integrate a third-party library into your system with a different interface.

Example:

```
public interface ITarget
{
    void Request();
}

public class Adaptee
{
    public void SpecificRequest()
    {
        Console.WriteLine("Specific Request");
    }
}

public class Adapter : ITarget
{
    private readonly Adaptee _adaptee;

    public Adapter(Adaptee adaptee)
    {
        _adaptee = adaptee;
    }

    public void Request()
    {
        _adaptee.SpecificRequest();
    }
}
```

Usage:

```
Adaptee adaptee = new Adaptee();
ITarget target = new Adapter(adaptee);
target.Request(); // Specific Request
```

2.2. Composite Pattern

The Composite pattern allows you to compose objects into tree-like structures to represent part-whole hierarchies.

Use Case: Useful for representing tree structures such as file systems or organizational structures.

Example:

```
public abstract class Component
{
    public abstract void Display();
}

public class Leaf : Component
{
    public override void Display()
    {
        Console.WriteLine("Leaf");
    }
}

public class Composite : Component
{
    private readonly List<Component> _children = new
List<Component>();

    public void Add(Component component)
    {
        _children.Add(component);
    }

    public void Remove(Component component)
    {
        _children.Remove(component);
    }

    public override void Display()
```

```
    {
        Console.WriteLine("Composite:");
        foreach (var child in _children)
        {
            child.Display();
        }
    }
}
```

Usage:

```
Leaf leaf1 = new Leaf();
Leaf leaf2 = new Leaf();
Composite composite = new Composite();
composite.Add(leaf1);
composite.Add(leaf2);

composite.Display(); // Composite displays its children (Leaf)
```

2.3. Decorator Pattern

The Decorator pattern allows you to dynamically add behavior to an object at runtime.

Use Case: Add features such as logging, authentication, or caching to an object.

Example:

```
public interface IMessageSender
{
    void Send(string message);
}

public class EmailSender : IMessageSender
{
    public void Send(string message)
    {
```

```
        Console.WriteLine($"Sending Email: {message}");
    }
}

public class SmsSenderDecorator : IMessageSender
{
    private readonly IMessageSender _messageSender;

    public SmsSenderDecorator(IMessageSender messageSender)
    {
        _messageSender = messageSender;
    }

    public void Send(string message)
    {
        Console.WriteLine($"Sending SMS: {message}");
        _messageSender.Send(message);
    }
}
```

Usage:

```
IMessageSender sender = new SmsSenderDecorator(new EmailSender());
sender.Send("Hello World!"); // Sends SMS and Email
```

3. Behavioral Design Patterns in .NET Core

Behavioral patterns focus on communication between objects and how responsibilities are delegated between them.

3.1. Strategy Pattern

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Use Case: Useful when you need to select an algorithm at runtime.

Example:

```
public interface IPaymentStrategy
{
    void Pay(decimal amount);
}

public class CreditCardPayment : IPaymentStrategy
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount:C} via Credit Card.");
    }
}

public class PayPalPayment : IPaymentStrategy
{
    public void Pay(decimal amount)
    {
        Console.WriteLine($"Paid {amount:C} via PayPal.");
    }
}

public class PaymentContext
{
    private readonly IPaymentStrategy _paymentStrategy;

    public PaymentContext(IPaymentStrategy paymentStrategy)
    {
        _paymentStrategy = paymentStrategy;
    }

    public void ExecutePayment(decimal amount)
    {
        _paymentStrategy.Pay(amount);
    }
}
```

Usage:

```
PaymentContext context = new PaymentContext(new PayPalPayment());
context.ExecutePayment(100.0m);  // Paid 100.00 via PayPal
```

3.2. Observer Pattern

The Observer pattern allows objects to subscribe to and receive notifications from a subject when its state changes.

Use Case: Used for event-driven systems or real-time data updates.

Example:

```
public interface IObserver
{
    void Update(string message);
}

public class ConcreteObserver : IObserver
{
    public void Update(string message)
    {
        Console.WriteLine($"Received message: {message}");
    }
}

public class Subject
{
    private readonly List<IObserver> _observers = new
List<IObserver>();

    public void AddObserver(IObserver observer)
    {
        _observers.Add(observer);
    }
```

```
    public void RemoveObserver(IObserver observer)
    {
        _observers.Remove(observer);
    }

    public void NotifyObservers(string message)
    {
        foreach (var observer in _observers)
        {
            observer.Update(message);
        }
    }
}
```

Usage:

```
Subject subject = new Subject();
IObserver observer1 = new ConcreteObserver();
IObserver observer2 = new ConcreteObserver();

subject.AddObserver(observer1);
subject.AddObserver(observer2);
subject.NotifyObservers("Subject state changed!");
```

3.3. Chain of Responsibility Pattern

The Chain of Responsibility pattern allows a request to be passed along a chain of handlers until one of them handles it.

Use Case: Useful when multiple objects can handle a request but the exact handler isn't known upfront.

Example:

```
public abstract class Handler
{
```

```
    protected Handler NextHandler;

    public void SetNext(Handler nextHandler)
    {
        NextHandler = nextHandler;
    }

    public abstract void HandleRequest(string request);
}

public class
```

ConcreteHandlerA : Handler { public override void HandleRequest(string request) { if (request == "A") { Console.WriteLine("Handler A processed request."); } else if (NextHandler != null) { NextHandler.HandleRequest(request); } } }

public class ConcreteHandlerB : Handler { public override void HandleRequest(string request) { if (request == "B") { Console.WriteLine("Handler B processed request."); } else if (NextHandler != null) { NextHandler.HandleRequest(request); } } }

Usage:

```
Handler handlerA = new ConcreteHandlerA();
Handler handlerB = new ConcreteHandlerB();
handlerA.SetNext(handlerB);

handlerA.HandleRequest("B");  // Handler B processed request
```

Conclusion

Design patterns are valuable tools for developing clean, maintainable, and scalable software. In this document, we've explored **Creational**, **Structural**, and **Behavioral** design

patterns, showcasing their implementation in .NET Core. By leveraging these patterns, you can solve common development challenges and create high-quality, flexible applications.