CRAFTING MINIMAL API IN .NET 8

SCALABLE AND RESILIENT API WITH A FUNCTIONAL APPROACH

LUDMAL DE SILVA

Contents

About the Author

I've been enjoying programming for over 20 years, mostly with .NET and C#, and I have a real knack for RESTful API design and Microservices architecture. Plus, I've contributed to the community by writing several open source projects and libraries in C# and Python. It's been a fantastic journey, and I'm always excited to share and learn more! This book contains all the knowledge I gained while developing APIs over the past decade.

This book is not a detailed step-by-step guide, but rather a practical guide with best practices for designing and developing a scalable API. I haven't discussed security in detail, which is a separate topic that deserves its own book. I hope you will enjoy reading this book and, most importantly, gain knowledge for writing APIs.

Email: ludmal@gmail.com
Blog: https://ludmal.com

SECTION

01

Introduction to minimal api

The .NET has a long history of APIs. It began with the .NET Framework, which offered a wide range of class libraries and runtime environment for creating and running applications. Later, with the release of .NET Framework 3.0, Windows Communication Foundation (WCF) was introduced, enabling developers to build distributed systems using different protocols.

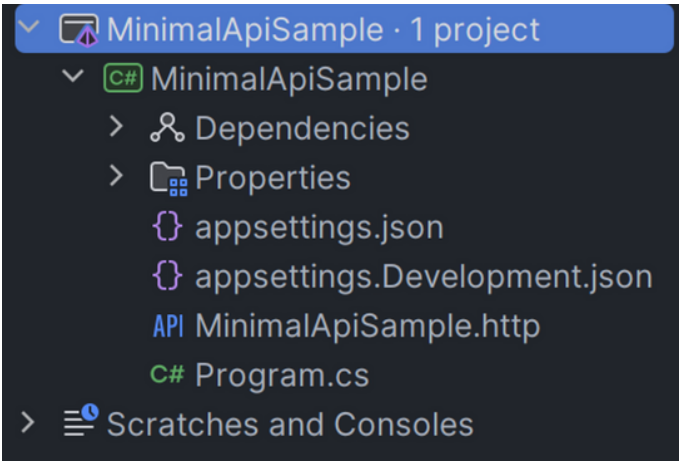
In 2016, .NET Core was introduced as a framework that can work on different platforms like Windows, macOS, and Linux. This made it easier for developers to create APIs that can run on multiple platforms.

In recent years, there has been a shift towards creating simpler and lighter APIs. This resulted in the introduction of the Minimal API in .NET 6. The goal of the Minimal API is to make API development easier by reducing the amount of unnecessary code. Microsoft has put a lot of effort into improving the performance of the API with each release, and with the introduction of .NET 8, it has become one of the fastest APIs frameworks available.

Key principles of API design are consistency, simplicity, discoverability, and predictability. APIs should be intuitive to use, have clear and meaningful names, and provide a consistent experience to developers.

1.1 Project Structure

After creating a Minimal API project using Visual Studio/Code or Rider, it will initially generate a default project structure with the Program.cs file and other necessary files like appsettings.json, launchsettings.json, and {projectname}.http files.



The Program.cs file in a .NET 8 Minimal API project serves as the entry point for the application. It contains the CreateBuilder method, which is responsible for creating and configuring the web host. The web host is responsible for hosting the Minimal API and handling incoming HTTP requests.

In the Program.cs file, you can configure various aspects of the web host, such as setting up logging, configuring the application's services, configuring the behavior of your Minimal API, including defining routes, handling requests, and configuring middleware. Let's explore the details in the sections below.

1.2 CreateBuilder vs CreateSlimBuilder

The CreateBuilder method in .NET Minimal API is responsible for creating and configuring the web host, which hosts the Minimal API and handles incoming HTTP requests.

The CreateSlimBuilder method is an alternative to the CreateBuilder method. It allows you to create and configure the web host for hosting the Minimal API in a simpler and lightweight way. This method is designed for scenarios where you want a more minimalistic approach and is especially useful for Native AOT(Explains below). It provides a stripped-down version of the web host configuration, removing unnecessary features and components for your specific use case.

By using CreateSlimBuilder, you can have more control over the configuration of your Minimal API project, only including the essential components and services that you need. This can help improve performance and reduce unnecessary overhead.

CreateEmptyBuilder is used to create a web host with no dependencies. This is ideal if you simply need to test out a few features and increase the performance. However you must ensure to include all the dependencies that web host require.

1.3 Ahead Of Time (AOT) [New in Net 8]

Ahead of Time (AOT) compilation is a feature in .NET 8 that lets developers compile their applications into machine code before runtime. This leads to faster startup times and better performance. AOT compilation removes the need for Just-in-Time (JIT) compilation during runtime, making the application more efficient. It's great for situations with limited runtime environments, like mobile devices or containers.

1.4 Environments

In the Program.cs file, you can use the Environments class to determine the current environment of the application. For instance, you can check if the environment is development or production.

```
//If running in Development
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
//or running in Production
if (app.Environment.IsProduction())
{
   app.UseSwagger();
   app.UseSwaggerUI();
}
//or running in Custom environment
if (app.Environment.IsEnvironment("SIT"))
{
   app.UseSwagger();
   app.UseSwaggerUI();
}
```

To set up this configuration, use the Environment class. It checks the ASPNETCORE_ENVIRONMENT variable to determine the environment type. To change the environment, update it through the command line. Or we could simply update in launchSettings.json file.

```
dotnet run --environment Production
```

1.5 Configuration

Application configuration is achieved through one or multiple configuration providers. These configuration providers are responsible for retrieving configuration data from various sources in the form of key-value pairs. They can tap into a diverse range of configuration sources, enabling flexibility in how configuration data is obtained

- Settings files, such as appsettings.json
- Environment variables
- Azure Key Vault
- Directory files
- In-memory .NET objects
- Custom Providers

```
var app = WebApplication.Create(args);
var version = app.Configuration["Version"] ?? "0v";
app.MapGet("/ping", () => version);
app.Run();
```

Option Pattern

The Option pattern in API configuration allows developers to retrieve values from a configuration, like appsettings.json, using a configuration class. This pattern offers a simple way to access configuration values within an API and helps with organizing and managing configuration settings efficiently. In the example below, we define settings for the GitHub API in the GitHubSettings class. These settings are configured in the appsettings.json file. We can then easily bind and inject these settings into the services where we need to access them.

```
//removed code for brevity
builder.Services.Configure<GitHubSettings>(
builder.Configuration.GetSection(nameof(GitHubSettings)));

var app = builder.Build();
app.UseHttpsRedirection();
app.MapGet("/apikey", (IOptions<GitHubSettings>
githubSettings) => Results.Ok(githubSettings.Value.ApiKey));

app.Run();

public class GitHubSettings
{
public string ApiKey { get; set; } = string.Empty;
public string BaseUrl { get; set; } = string.Empty;
}
```

Here is the appsettings.json file

```
{
 "Logging": {
    "LogLevel": {
         "Default": "Information",
         "Microsoft.AspNetCore": "Warning"
       }
  },
 "AllowedHosts": "*",
 "GitHubSettings": {
    "ApiKey": "26xeywt3t",
    "BaseUrl": "https://api.github.com"
  }
}
```

Just a note, we usually use nameof(GitHubSettings) which is a good practice. But in this case, be careful. If you ever change the name of the GitHubSettings class automatically, it might cause problems with the configuration because it's written as plain text in the appsettings.json file. So, keep in mind before making changes.

Convention Over Configuration — Convention over configuration is a software development principle that says we should use default settings and assumptions if we don't specify any. This means developers can use sensible defaults and standards, which makes coding easier and faster because there's less need for manual setup and configuration.

launchSettings.json

The launchSettings.json file in a Minimal API project is used to configure the launch settings for the application. It allows developers to specify various options such as the environment variables, command-line arguments, and other settings that control the execution and debugging of the application during development.

appsettings.{Environment}.json

The appsettings.{environment}.json file serves as a repository for environment-specific configuration settings within the API. By default, you'll find appsettings.development.json and appsettings.production.json files, but you have the freedom to establish custom environments and their corresponding appsettings files. Consequently, when you execute your application with the designated environment name, the configuration system will automatically retrieve values from the associated environment-specific appsettings files.

secrets.json

The secrets.json file is used to store configuration settings that should be kept secret, such as API keys or connection strings. It allows developers to securely store and access sensitive information in the API without exposing it in the codebase. This will not be included in your commits and it is only for you in your development machine.

1.6 Swagger for documentation

Swagger is integrated seamlessly with .NET Minimal API for generating interactive API documentation. It automatically generates documentation that allows developers to explore and test API endpoints. This clear and comprehensive documentation facilitates the development and adoption of the Minimal API. For instance, you can enable the documentation in the Development environment as shown in the example below. The UseSwaggerUI function provides a user-friendly interface for all the API operations. It is best practice to enable Swagger only on dvelopment mode.

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

1.7 IOC

Inversion of Control (IOC) container is used to manage dependencies and provide instances of objects to the application. IOC promotes loose coupling, modular design, and easier maintenance. It enables easy resolution and injection of dependencies, improving separation of concerns and facilitating unit testing with mock objects. Here are the top most benefits of IOC.

Key benefits of IOC (Inversion of Control) are:

1. **Improved Modularity:** IOC promotes loose coupling between components, allowing for better modularity in your application. This makes it easier to understand, maintain, and update individual components without affecting the entire application.
2. **Easier Testing:** IOC facilitates unit testing by allowing dependencies to be easily mocked or replaced with test doubles. This helps isolate the behavior of individual components during testing, leading to more reliable and efficient tests.
3. **Flexibility and Extensibility:** With IOC, you can easily replace or extend components without modifying the code that depends on them. This flexibility enables you to adapt your application to changing requirements or add new features without extensive code modifications.
4. **Decoupling and Dependency Management:** IOC helps manage dependencies between components by decoupling them. This reduces the dependencies between different parts of your application, making it easier to understand and maintain. It also simplifies dependency management, as dependencies can be easily resolved and injected into components.

Here is a sample IOC usage

```
public interface IGreetingService
{
  string Greet(string name);
}

public class EnglishGreetingService : IGreetingService
{
  public string Greet(string name) => $"Hello, {name}!";
}

public class SpanishGreetingService : IGreetingService
{
  public string Greet(string name) => $"Hola, {name}!";
}

//We simply use the EnglishGreeting service
builder.Services.AddScoped<IGreetingService,
EnglishGreetingService>();

app.MapGet("/greeting", (IGreetingService greetingService) =>
Results.Ok(greetingService.Greet("World")));
//Results will be Hello World!
```

Keyed Service Registration [New in Net 8]

With .Net 8, we have the ability to use keyed services. One of the main advanatages of doing that is to decide what implementation to use at runtime. Here is an example:

```
builder.Services.AddKeyedSingleton<IGreetingService,
EnglishGreetingService>("English");
builder.Services.AddKeyedSingleton<IGreetingService,
SpanishGreetingService>("Spanish");

app.MapGet("/greet",
([FromKeyedServices("English")]IGreetingService english,
[FromKeyedServices("Spanish")]IGreetingService spanish) =>
$"English:{english.GetGreeting()} Spanish:
{spanish.GetGreeting()}").WithName("GreetingEndpoint");
```

Lifetime of IOC

There are three lifetimes of IOC (Inversion of Control) registrations available:

1. **Singleton**: A single instance of the registered service is created and shared across the entire application. It is reused for each request or usage of the service.
2. **Scoped**: A new instance of the registered service is created for each scope. A scope typically corresponds to a single HTTP request in a web application. The same instance is reused within that scope, but different scopes will have different instances.
3. **Transient**: A new instance of the registered service is created every time it is requested. This means that every usage or request for the service will receive a new instance.

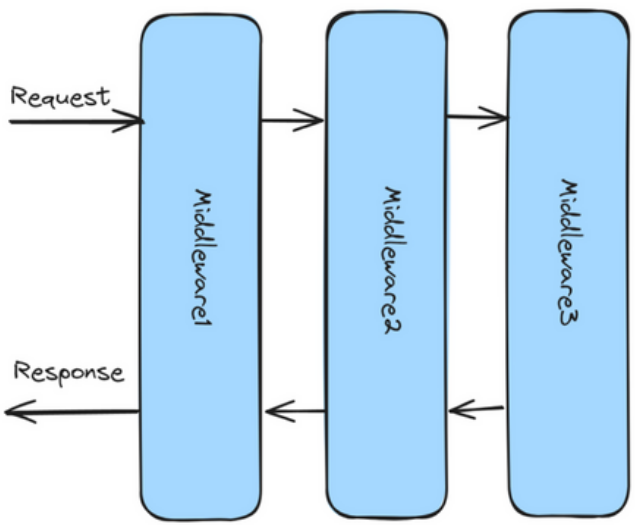
These lifetimes provide flexibility in managing the lifecycle and behavior of the services within an application. Choosing the appropriate lifetime depends on the specific requirements and dependencies of the services being registered.

> DRY — Don't repeat yourself

1.8 Middlware/Filters

Middlewares

A middleware in Api is a component that sits in the request/response pipeline and adds functionality to the handling of HTTP requests. It intercepts and processes requests and responses, allowing for operations such as logging, authentication, and error handling to be performed. It implements a chain of responsiblity pattern as shown below.



Common usages of middleware in .NET are:

1. **Authentication Middleware:** Used to handle authentication and authorization for incoming requests. It can validate credentials, issue and validate tokens, and enforce access control rules.
2. **Logging Middleware:** Used to log information about incoming requests and outgoing responses. It can capture request details, response status codes, and any errors or exceptions that occur during the request/response cycle.
3. **Error Handling Middleware:** Used to handle exceptions and errors that occur during the processing of a request. It can catch and handle exceptions, log error details, and return appropriate error responses to the client.
4. **Caching Middleware:** Used to cache responses and improve performance by serving cached data for subsequent identical requests. It can store and retrieve cached responses based on specific caching rules and expiration policies.

These middleware components can be added to the middleware pipeline in a specific order (remember order is important) to perform various tasks and modify the behavior of the request/response flow. In addition to the above you can write your own middleware. In the following example, you will see a custom middleware to handle Api key validation.

```
public class ApiKeyValidationMiddleware
{
  private readonly RequestDelegate _next;
  public ApiKeyValidationMiddleware(RequestDelegate next)
  {
    _next = next;
  }

  public async Task InvokeAsync(HttpContext context)
  {
    var apiKey = context.Request.Query["api-key"];
    if (string.IsNullOrWhiteSpace(apiKey))
    {
      throw new UnauthorizedAccessException();
    }

    // this will call the next middleware in the pipeline.
    await _next(context);
  }
}

//We typicaly write an extension method to be more
//descriptive to the startup class
public static class ApiKeyValidationMiddlewareExtensions
{
  public static IApplicationBuilder UseApiKeyValidation(this
IApplicationBuilder builder)
 {
   return builder.UseMiddleware<ApiKeyValidationMiddleware>
();
 }
}

//Add in the startup
var app = builder.Build();
app.UseHttpsRedirection();
//Here we add the Api key validation middleware
app.UseApiKeyValidation(); //👈 Add this
```

Middleware must use depencies in constructor and it is constructed once per application lifetime.

Filters

EndpointFilters in minimal apis provide a way to add cross-cutting concerns to the request/response pipeline of an API. Filters can be used to perform tasks such as logging, validation, custom business logic and more, allowing for modular and reusable code across different routes.
Here is an example of a filter for Minimal API:

```
public class LoggingFilter: IEndpointFilter{
protected readonly ILogger Logger;
public LogginFilter(ILoggerFactory loggerFactory) {
  Logger = loggerFactory.CreateLogger<LoggingFilter>();
}
public async ValueTask<object?> InvokeAsync(
EndpointFilterInvocationContext context,
EndpointFilterDelegate next)
{
var result = await next(context);
Logger.LogInformation("{result}", result);
return result;
}
```

The EndpointFilterInvocationContext object grants access to the parameters linked with a specific request directed towards the endpoint, accessible through the GetArguments method.

The registration of the filter is accomplished by employing a delegate that accepts an EndpointFilterInvocationContext as input and yields an EndpointFilterDelegate as its output.

The filter is executed before the endpoint handler, and in cases where multiple AddEndpointFilter invocations are made on a handler, filter code BEFORE the await next(context); executed FIFO(First In First Out) and AFTER the await next(context); excuted FILO (First In Last Out) order.

To use this filter, you can apply it to an endpoint or a group of endpoints using the AddEndpointFilter method:

```
app.MapGet("/api/endpoint", () => "Hello, World!")
.AddEndpointFilter
<LoggingFilter>();
```

This will apply the LoggingFilter to the /api/endpoint endpoint, and the filter will be executed before and after the endpoint execution, logging the appropriate messages.

In a Minimal API, middleware and endpoint filters both provide ways to add cross-cutting concerns to the request/response pipeline. Here are some guidelines on when to use middleware vs endpoint filters:

Middleware:

- Use middleware when you need to perform an operation over entire request pipeline.
- Middleware is typically used to handle tasks such as authentication, logging, error handling, and request/response transformations.
- Middleware operates at a lower level in the pipeline and has access to the raw HTTP request and response.
- Middleware can be added globally to the application or applied to specific routes using the app.Use or app.Map methods.

Endpoint Filters:

- Use endpoint filters when you need to apply specific actions or behavior to individual endpoints or groups of endpoints.
- Endpoint filters are applied directly to endpoints using the AddEndpointFilter method.
- Endpoint filters are executed only for the targeted endpoints, allowing for more fine-grained control.
- Endpoint filters are typically used for tasks such as validation, authorization, and logging specific to an endpoint.
- Endpoint filters have access to the endpoint context with Request data and can modify or inspect the endpoint's metadata and execution context.

In summary, middleware is used for things that affect the entire system or pipeline, while endpoint filters are used for things that only affect specific endpoints or groups.

1.9 Routes

Routes in Minimal API provide a way to define the endpoints and their corresponding handlers for handling incoming HTTP requests. They allow developers to specify the URL pattern, HTTP verb, and the handler function or method to be executed when a request matches the defined route.

Here is a simple example of defining routes in Minimal API:

```
app.MapGet("/api/todos", () => "Get all todos");
app.MapPost("/api/todos", () => "Create a new todo");
app.MapPut("/api/todos/{id}", (int id) => $"Update todo with
ID: {id}");
app.MapDelete("/api/todos/{id}", (int id) => $"Delete todo
with ID: {id}");
```

In this example, we have defined four routes for handling different HTTP verbs. The routes /api/todos and /api/todos/{id} are used for retrieving, creating, updating, and deleting todo items, with the appropriate handlers implemented as lambda expressions. In the next section, "Architecting Minimal API," we will talk more about how to handle these request operations.

Minimal Api supports all the major http verbs like Get, Post, Patch, Put, Delete.

Handlers

In Minimal API, route handlers are the functions or methods that are executed when a request matches a defined route. They define the logic for processing the request and generating the response, allowing developers to implement custom behavior and business logic for each endpoint. For example, the following is a static function to handle a Get endpoint.

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", HelloHandler.Hello);
app.Run();

class HelloHandler
{
 public static string Hello() { return "Hello world!"; }
}
```

Similarly you can use lambda expression, instance method, location function or static method as handlers.

Route Groups

.NET Minimal API introduces the concept of route groups, which allows developers to group related endpoints under a common route prefix. This helps organize and structure the API by grouping related functionality together. Route groups provide a clean and concise way to define multiple endpoints with a shared base route, reducing duplication and improving maintainability.

```
app.Map("/api", api =>
{
api.MapGet("/todos", () => "Get all todos");
api.MapPost("/todos", () => "Create a new todo");
api.MapPut("/todos/{id}", (int id) => $"Update todo with ID:
{id}");
api.MapDelete("/todos/{id}", (int id) => $"Delete todo with
ID: {id}");
});
```

Short Circuit

Short circuiting in Minimal API allows you to quickly handle and respond to requests without further processing based on certain conditions. It helps optimize the request pipeline by bypassing unnecessary middleware and route handlers, improving performance and reducing latency. It skips all the middleware in between EndpointRoutingMiddlware and EndpointMiddleware

```
app.MapGet("/", () => "Hello World!")
    .ShortCircuit(); // 👈 Add this
```

The advantage of using ShortCircuit is mainly to avoid using unnecessary memory or processing power for requests that don't require Authorization, Cors, or other middleware . For example, requests like favicon.ico or robots.txt.

Link Generator

The Link Generator in Minimal API provides a convenient way to generate URLs for different endpoints within the API, allowing developers to easily create hyperlinks to other resources. For example, you can use the Link Generator to generate a URL for the /api/todos endpoint as follows:

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

app.MapGet("/todos", () => "List of all the todos")
.WithName("TodoList");

app.MapGet("/", (LinkGenerator linker) =>
$"The link to get all the Todos is {
 linker.GetPathByName("TodoList", values: null)
}");
```

Parameters

Parameters are mainly request data mainly from query or as in body. A binding source determines where parameters come from. Binding sources can be explicit or inferred based on the HTTP method and parameter type.

Binding Sources:

- Query string
- Route values
- Body (as JSON)
- Header
- Form values
- Custom

Route parameters are mainly from the query string or route path and they can be captured as part of the pattern defined in the route.

```
app.MapGet("/todos/{todoId}/files/{fileId}",
  (int todoId, int fileId) => $"The todo id is {todoId} and
file id is {fileId}");
```

Wildcard routes

```
app.MapGet("/todos/{*all}", (string all) => $"Incoming params
{all}"); // 👈 Catches all the requests

/*Example
/todos/hello
/todos/new
/todo/update
*/
```

Route constraints

```
app.MapGet("/todos/{id:int}", (int id) =>
db.Todos.FirstOrDefault(id));//👈 Ensure id is int
```

Parameter Binding

POST will bind the parameters from the body automatically while GET, HEAD, OPTIONS and DELETE will not. Here are some sample parameter binding with attributes

```
app.MapPost("/todos/", (Todo todo) => $"Creating
{todo.Text}");
//FromBody
app.MapGet("/todos/", ([FromBody]Todo todo) => $"Exceuted");
//FromQuery
app.MapGet("/todos/search", ([FromQuery(Name="q")]string
keyword) => $"Executed");
//FromService
app.MapGet("/todos/search", ([FromService]ITodoService
todoService) => $"Executed"");
//FromHeader
app.MapGet("/todos/search", ([FromHeader(Name="x-cache-
key")]string cacheKey) => $"Executed");
```

AsParameters attribute

This is particularly important to others. This enables simple parameter bindings to types. Query parameters or form values can be bound to type object as shown in the following example.

```
app.MapGet("/todos/search", ([AsParameters]SearchParameters
params) => $"Searching for: {params.Keyword},
{params.Count}");

//In the above example, SearchParameters
//object will be bound from the querystring values

public record SearchParameters(string Keyword, int Count);
//Example api call /* "/todos/search?Keyword=shirt&Count=1
```

Responses

There are different types of return values for handling responses. These include returning raw strings or data, using Results unions for standardized responses, and using HTTP-specific classes like Results.Ok() or Results.BadRequest() for more structured and descriptive responses.

```
//Returns an object
app.MapGet("/hello", () => "Hello World" );
//Returns an object
app.MapGet("/hello", () => new { Message = "Hello World" });
```

Typically, IResult is used when returning results as shown in the following example

```
app.MapGet("/hello", () => Results.Ok("Hello World"));
```

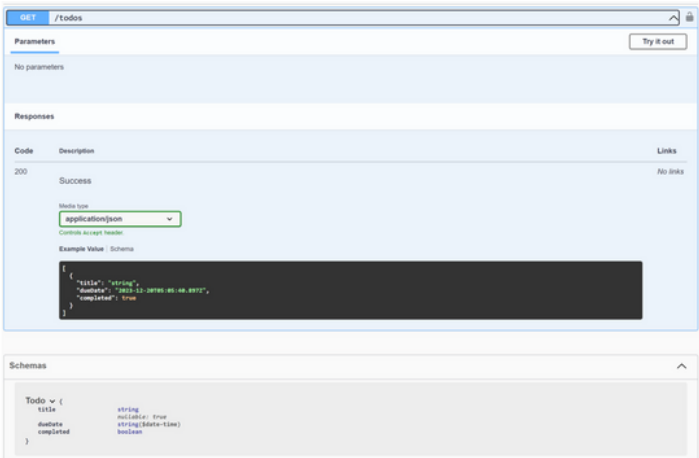
However it is better to use TypedResults instead of Results. They have the following advantages:

- Provides schema data for OpenAI(Swagger) documentation which can help the client developer
- Helps unit tested since it returns strongly typed objects
- Increase code readability

Here is an example:

```
app.MapGet("/todos", () => Enumerable.Range(1,10)
.Select(x => new Todo()
{
  Title = $"Todo {x}",
  DueDate = DateTime.Now.AddDays(x),
  Completed = false
}));
app.Run();

public class Todo {
 public string Title { get; set; } = string.Empty;
 public DateTime DueDate { get; set; }
 public bool Completed { get; set; }
}
```

Here is an example:



When we need to return multiple results, we use Results with generic union types. Let's consider the following scenario:

```
public static class TodoApi
{
  public static IResult GetTodos() {
    var todos = new Collection<Todo>();
    return todos.Any()
      ? Results.Ok(todos) : Results.NotFound();
 }
}
```

By using a TypedResults union type, we can easily improve the code to make use of strongly-typed objects and generate schema documentation.

```
public static class TodoApi
{
public static Results<Ok<List<Todo>>, NotFound> GetTodos()
{
var todos = new List<Todo>();
return todos.Any()
? TypedResults.Ok(todos) : TypedResults.NotFound();
}
}
```

1.10 Cancellation Token

The cancellation token is important in APIs as it allows for graceful cancellation of long-running operations, improving responsiveness and resource utilization in the API. It provides a mechanism to cancel or stop the execution of a request or operation based on external factors or user actions, ensuring efficient handling of requests and preventing unnecessary resource consumption.

```
app.MapGet("/todos", (CancellationToken cancellationToken) =>
{
return Results.Ok(LongRunningOperation(cancellationToken));

// Simulate long running operation
object? LongRunningOperation(CancellationToken
cancellationToken1) {
}
});
```

1.11 Rate Limiting

Throttling or Rate limiting in Minimal API refers to the practice of limiting the number of requests a client can make within a certain time period. It helps prevent abuse and ensures fair usage of API resources. By implementing throttling mechanisms, developers can control the rate at which clients can access their APIs, improving performance and protecting against potential overloading or denial of service attacks.

Typically these requests are handled via Api Gateway policies. However, .Net provides built-in middleware to handle this effectively.

```
//Following Rate Limiter allows maximum of 4 request
//within 12 seconds time window
//FixedWindowLimiter
builder.Services.AddRateLimiter(r => r
.AddFixedWindowLimiter(policyName: "fixed", options =>
{
options.PermitLimit = 4; // 4 requests
options.Window = TimeSpan.FromSeconds(12); //within 12seconds
}));

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

app.MapGet("/", () => Results.Ok($"Hello World"))
.RequireRateLimiting("fixed");
```

In addition to the above FixedWindowLimiter there are 3 more rate limiter strategies:

- **Sliding Window:** The sliding window is a rate limiter implementation that limits the number of requests allowed within a specified time window. However, the implementation is different to the FixedWindowLimiter with a segment per window.
- **Token Bucket:** The token bucket limiter, unlike the sliding window limiter, replenishes a fixed number of tokens at regular intervals and ensures that the accumulated tokens never exceed the specified token bucket limit.
- **Concurrency:** The concurrency limiter restricts how many requests can happen at the same time, with each request decrementing the limit by one and completing requests increasing it by one. Unlike other request limiters that control the total number of requests within a time frame, the concurrency limiter only restricts the simultaneous requests and doesn't impose a limit on the total requests over time.

SECTION

02

Architecting Minimal Api

Architecting an API requires a lot of experience and careful consideration. Over the past decade, many patterns have emerged for designing APIs using .net technologies. The evolution of APIs gained momentum with the shift from Monolithic to Microservices. However, when choosing the right architecture pattern for designing the API, we should avoid overcomplicating things and strive to use the principles to come up with the simplest approach.

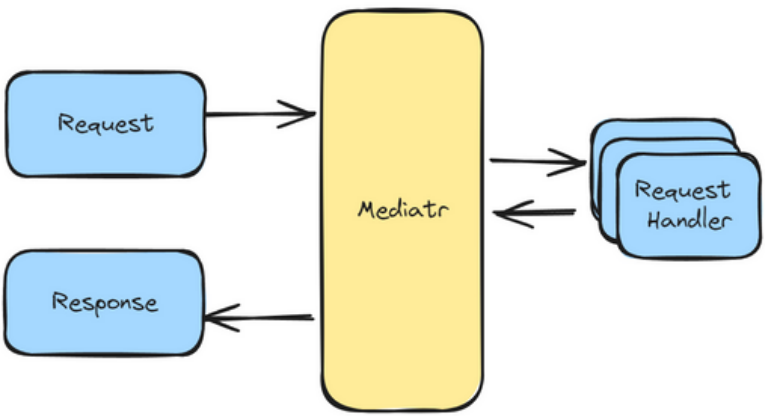
I have often observed that some frameworks complicate the implementation with excessive abstractions. What I propose and aim to introduce is a simple yet effective approach to designing any API layer.

This approach aims to reduce complexity, improve performance, support scalability, and enhance maintainability for smaller-scale API projects.

KISS - Keep it Simple Stupid
YAGNI — You aren't gonna need it

2.1 Request/Response

The Request/Response pattern is a fundamental pattern in API design. It involves clients sending requests to the API, and the API responding with appropriate responses. This pattern allows for clear communication and interaction between clients and the API, enabling the exchange of data and the execution of desired actions. It helps ensure that the API can handle different types of requests and provides consistent and predictable responses to clients. It is the simplest yet most effective pattern I've seen over the last few years.



In its simplest form, the client makes a call to an API endpoint with the Request data. The RequestDispatcher then dispatches the request to the relevant RequestHandler and returns the Response.

In the following examples, I will show you how to implement a basic API endpoint operation using a Request/Response pattern. This pattern can be easily implemented using C# and .net build in middlewares, however there is a great library for implementing this pattern. — https://github.com/jbogard/MediatR

Implementing Request/Response pattern

Let's create a MinimalApi project and add the following nuget package:

```
dotnet add package MediatR --version 10.0.1
dotnet add package
MediatR.Extensions.Microsoft.DependencyInjection --version
10.0.1
```

And Let's hook that up in the project

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddMediatR(typeof(Program)); //👈 Register
all Mediator classes
```

In the following example, a complete implementation has been done with request, request handler and response object. Also explains each object in detail below.

```
app.MapPost("/todos", (ISender sender, CreateTodoRequest
request) => sender.Send(request));
app.Run();

public record CreateTodoRequest(string Title, DateTime
DueDate) : IRequest<CreateTodoResponse>;

public record CreateTodoResponse(int Id, bool Success);

public sealed class CreateTodoRequestHandler :
IRequestHandler<CreateTodoRequest, CreateTodoResponse>
{
private readonly ILogger<CreateTodoRequestHandler> _logger;
private readonly IDataContext _dataContext;
public
CreateTodoRequestHandler(ILogger<CreateTodoRequestHandler>
logger, IDataContext dataContext)
{
_logger = logger;
_dataContext = dataContext;
}

public async Task<CreateTodoResponse>
Handle(CreateTodoRequest request, CancellationToken
cancellationToken)
{
var created = await _dataContext.Create<CreateTodoRequest>
(request);
return new CreateTodoResponse(1, created);
} }
```

CreateTodoRequest: Request object which includes data when passing through the Api. i.e. Body. This class must implement IRequest interface and pass the generic type. If there is no return object simply using the interface without generic type.

CreateTodoResponse: Response object which returns as the reponse. This does not required any implementation rather a simple Data transfer object (DTO)

CreateTodoRequestHandler: Request handler is responsible for handling the request, do all the logic required and return the response back. This class must implement IRequestHandler interface with the relevent genric types. Also note the ILogger<> and IDataContext in the constructor. IDataContext is injected from the IOC container (ensure you have registered it in the program.cs file). Logging is very imporant in every aspect of the code, however we should carefulty selects the log level to ensure we don't bloat the logs database.

Handle in CreateTodoRequestHandler**:** This is the main method to handle the request and do all the necessary business logic and returns the response object.

When logging, use the template pattern instead of string interpolation. This is to prevent unnecessary memory allocation and to enable searching through the logs. eg. _logger.LogInformation("Creating todo with data {data}", JsonSerializer.Serialize(request));

2.2 Validations

Model validation in Minimal API ensures that the data submitted by the client meets the specified criteria and constraints before processing it further.

FluentValidation is a popular validation library that can be used in Minimal API to easily define and enforce validation rules for incoming requests. It provides a fluent and expressive syntax for defining validation rules, allowing developers to validate request models and ensure data integrity in a concise and readable manner.

To use FluentValidation in Minimal API, first, install the FluentValidationNuGet packages. Then, follow these steps:

```
dotnet add package FluentValidation
dotnet add package
FluentValidation.DependencyInjectionExtensions
```

1. Create a validator class that inherits from AbstractValidator<T>, where T is the request model you want to validate.
2. In the validator class, define the validation rules using the RuleFor method.
3. Register the validator in the application services using the AddValidatorsFromAssembly method.

Here's an example of how to implement FluentValidation in Minimal API:

```
using FluentValidation;
using Microsoft.AspNetCore.Mvc;

public record CreateTodoRequest(string Title, DateTime DueDate);

public class CreateTodoRequestValidator :
AbstractValidator<CreateTodoRequest>
{
public CreateTodoRequestValidator()
{
RuleFor(x => x.Title).NotEmpty().WithMessage("Title is required");
RuleFor(x => x.DueDate).LessThan(DateTime.Now).WithMessage("Due date
must be in the future");
}}

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddValidatorsFromAssemblyContaining<Program>();

app.MapPost("/todo", (CreateTodoRequest todo,
IValidator<CreateTodoRequest> validator) =>
{
    var result = validator.Validate(todo);
    return result.IsValid ? Results.Ok(todo) :
Results.BadRequest(result.Errors);
});

app.Run();
```

In this example, we define a CreateTodoRequest record as the request model. We create a CreateTodoRequestValidator class that inherits from AbstractValidator<CreateTodoRequest> to define the validation rules. The validation rules state that the Title property must not be empty and the DueDate property must be in the past. And then we inject the IValidator<CreateTodoRequest> in endpoint and call validate(request) to ensure if the request is valid. This is the most basic usage of fluent validation library.

However, validations should be automatically injected into the pipeline. This ensures better separation of concerns. This can be done easily with Mediatr pipelines. It works same as middlewares, except these are configured in the meditr library request/response process.

To implement the validation process via Mediatr pipelines:

1. Create a Pipeline Behaviour and register as Meditr as IPipelineBehavior
2. Create a Middleware to handle the ValidationException and return as Status400BadRequest
3. Register the middleware in the pipeline

```
//Meditr validation behaviour
public class ValidationBehaviour<TRequest, TResponse>(
IEnumerable<IValidator<TRequest>> validators,
ILogger<TRequest> logger) : IPipelineBehavior<TRequest,
TResponse> where TRequest : notnull, IRequest<TResponse>
{
public async Task<TResponse> Handle(TRequest request,
CancellationToken cancellationToken,
RequestHandlerDelegate<TResponse> next)
{
if (validators.Any())
{
var context = new ValidationContext<TRequest>(request);
var errorsDictionary = validators
.Select(x => x.Validate(context))
.SelectMany(x => x.Errors)
.Where(x => x != null)
.GroupBy( x => x.PropertyName, x => x.ErrorMessage,
(propertyName, errorMessages) => new { Key = propertyName,
Values = errorMessages.Distinct().ToArray()
})
.ToDictionary(x => x.Key, x => x.Values);
if (errorsDictionary.Any()) {
throw new ValidationException( "Validation Failed",
errorsDictionary.Select(x => new ValidationFailure(x.Key,
string.Join(",",x.Value))));
}
}
else
logger.LogDebug("No Validators found");
return await next(); }
}
```

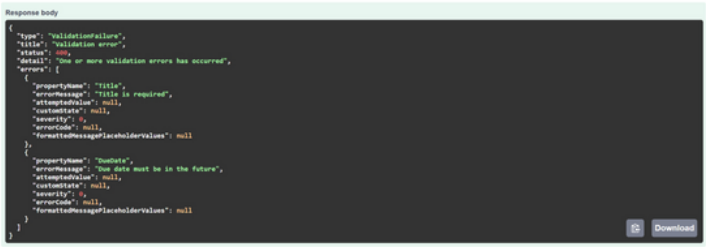
To register middleware in the program.cs file, we typically create an extension class. This is one of the best practices to register middlewares.

```
public sealed class
ValidationExceptionHandlingMiddleware(RequestDelegate next)
{
public async Task InvokeAsync(HttpContext context)
{
try
{
await next(context);
}
catch (ValidationException exception)
{
var problemDetails = new ProblemDetails
{ Status = StatusCodes.Status400BadRequest,
Type = "ValidationFailure",
Title = "Validation error",
Detail = "One or more validation errors has occurred"
};

if (exception.Errors is not null)
{
problemDetails.Extensions["errors"] = exception.Errors;
}

context.Response.StatusCode =
StatusCodes.Status400BadRequest;
await context.Response.WriteAsJsonAsync(problemDetails);
}
}
}

public static class ValidatorMiddlewareExtentions
{
public static IApplicationBuilder UseValidationBehaviour(
this IApplicationBuilder builder)
{
return
builder.UseMiddleware<ValidationExceptionHandlingMiddleware>
();
}
}
```

Finally, register the IPipelineBehavior in IOC container and the validation middleware.

```
builder.Services.AddTransient(typeof(IPipelineBehavior<,>),
typeof(ValidationBehaviour<,>));
app.UseValidationBehaviour();
```

As shown in the screenshot below, the validations are returned as a Status400BadRequest along with the relevant error messages.



Separation of concerns is vital to a scalable and maintainable api.

DIE — Duplication Is Evil

2.3 Mappers

Mapping means converting data from one object type to another. It is commonly used to make data transfer between different parts of an application easier. For instance, if you have to associate the properties from a Model to Entity or Entity to DTO, instead of manually writing the code, we can simply utilize a library to handle the mapping for us.

AutoMapper is a widely used library in .NET that makes it easy to map between different object types, saving time and effort in writing mapping code. It can be effectively used in Minimal API to simplify the mapping process and enhance code readability and maintainability.

To use AutoMapper in Minimal API, you need to install the following NuGet package.

```
dotnet add package
AutoMapper.Extensions.Microsoft.DependencyInjection
```

Steps:

1. Create a mapping profile class that inherits from Profile.
2. In the mapping profile class, use the CreateMap method to define the mapping between the source and destination types.
3. Register the mapping profile in the application services using the AddAutoMapper method.

Here's an example of how to implement AutoMapper in Minimal API:

```
using AutoMapper;
// Register the MappingProfile program.cs
builder.Services.AddAutoMapper(typeof(Program));

public record TodoDto(string Title, bool Completed);
public class TodoDtoProfile : Profile {
public TodoDtoProfile() {
CreateMap<Todo, TodoDto>();
}
}
```

In this example, we define a TodoDto record as the destination type and a Todo class as the source type. We create a TodoDtoProfile class that inherits from Profile to define the mapping between the Todo and TodoDto types using the CreateMap method.

To use the mapping, you can inject IMapper into your route handlers or services and use the Map method to perform the mapping. Here's an example:

```
app.MapGet("/todos", (ITodoService todoService, IMapper
mapper) =>
{
var todos = todoService.GetTodos();
var todoDtos = mapper.Map<IEnumerable<TodoDto>>(todos);

return TypedResults.Ok(todoDtos);
});
```

In this example, we inject ITodoService and IMapper into the route handler. We retrieve the todos from the ITodoService and then use the Map method of the IMapper to map the todos to a collection of TodoDto objects.

2.4 Services

Services provide a way to encapsulate and organize the logic and functionality of your application. They help promote separation of concerns and modularity, allowing you to easily manage and test different parts of your codebase.

Typically, services are injected to Request Handlers and perform relevant operations. This is one of the best practices in Request/Response design. Consider the following scenario:

```
public sealed class CreateTodoRequestHandler(
ILogger<CreateTodoRequestHandler> logger,
IDataContext dataContext,
ITodoService todoService) :
IRequestHandler<CreateTodoRequest, CreateTodoResponse>
{
private readonly ILogger<CreateTodoRequestHandler> _logger =
logger;
private readonly ITodoService _todoService = todoService;

public async Task<CreateTodoResponse>
Handle(CreateTodoRequest request, CancellationToken
cancellationToken)
{
if (_todoService.IsExist(request.Title))
return new CreateTodoResponse(0, false);
var created = await dataContext.Create<CreateTodoRequest>
(request);
return new CreateTodoResponse(1, created);
}
}

public interface ITodoService {
bool IsExist(string title);
}

public class TodoService : ITodoService {
public bool IsExist(string title)
{ //Logic to check if todo exists return true;
}
}
```

In the above example, ITodoService interface is defined to create a TodoService class which encapsulates the logic to ensure the Todo item is unique. And typically we inject the class in the constructor. In a world of Test driven development or similar, programming to interface makes your code more testable.

OCP - Open Close Principle — Always program to interfaces and not to implementations

2. 5 Error Handling

When working with APIs, it's important to handle exceptions and errors correctly and let the client know about them. In Minimal API, one way to handle errors is by using middleware. This allows you to catch exceptions and return the right HTTP status codes and error messages.

You can also use the Results class to send specific error results like Results.BadRequest() or Results.NotFound(). This helps provide more helpful error responses to the client.

Let's look at the implementation of the Error handling using middleware.

1. Create a base class exception and derived exceptions

```
public class BusinessException<TData>(string message) :
Exception(message)
{
public TData Data
{ get; set; } = default!;
}

public class TodoNotFoundException : BusinessException<Todo>
{
public TodoNotFoundException(Todo todo) : base("Todo not
found")
{
Data = todo;
}
}
```

2. Create the Error handling middleware and extensions class

```
public sealed class BusinessValidationsMiddleware<TData>
(RequestDelegate next)
{
public async Task InvokeAsync(HttpContext context)
{
try
{
await next(context);
}
catch (BusinessException<TData> exception)
{
var problemDetails = new ProblemDetails {
Status = StatusCodes.Status400BadRequest,
Type = "BusinessValidation",
Title = "Business validation error",
Detail = "One or more validation errors has occurred" };

if (exception.Data is not null)
{
problemDetails.Extensions["data"] = exception.Data;
}
context.Response.StatusCode =
StatusCodes.Status400BadRequest;

await context.Response.WriteAsJsonAsync(problemDetails);
}
}
}

public static class BusinessValidationsMiddlewareExtensions {

public static IApplicationBuilder
UseBusinessValidationMiddleware<TData>(this
IApplicationBuilder builder) {
return
builder.UseMiddleware<BusinessValidationsMiddleware<TData>>
();
}
}
```

3. Use the TodoNotFoundException in route handlers.

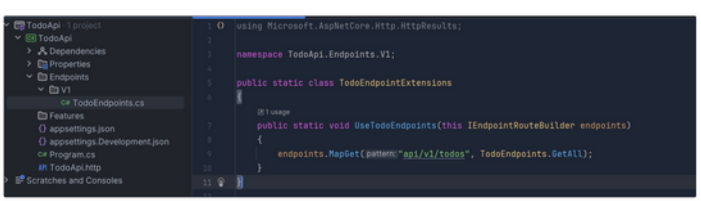
```
app.UseBusinessValidationMiddleware<Todo>();

app.MapGet("/todos/{title}", (string title, ITodoService
todoService) => {
var todo = todoService.IsExist(title);
if (!todo) throw new TodoNotFoundException(new Todo()
{
Title = title
});

return TypedResults.Ok(new Todo()
{
Title = title
});
});
app.Run();
```

There are many benefits of using this error handling pattern, such as separation of concerns, adaptability, streamlined exceptions, and more. However, it is generally not recommended to throw exceptions within the application layer. Sometimes, though, you need to strike a balance between productivity and following the rules. We will be talking about discriminated unions which help avoid throwing exceptions in the application.

> Software development is all about trade off

2.6 Versioning

Versioning in Minimal API allows developers to manage the different versions of their APIs, ensuring compatibility and smooth transitions for clients. It provides a way to introduce breaking changes or new features while maintaining backward compatibility with existing clients. There are many nuget packages which support API versioning. But I recommend to keeping it very simple in your folder structure as shown below.



In the example above, we put the Endpoint file in the Endpoint/V1 folder. This represents the API version. Additionally, in the route builder, we can include the version as api/v1/todos.

This is the simplest way and I find it elegant. In addition to the above, there are many ways to handle versioning in APIs. i.e. Header, Body or even via query string. However it is out of the scope of this book.

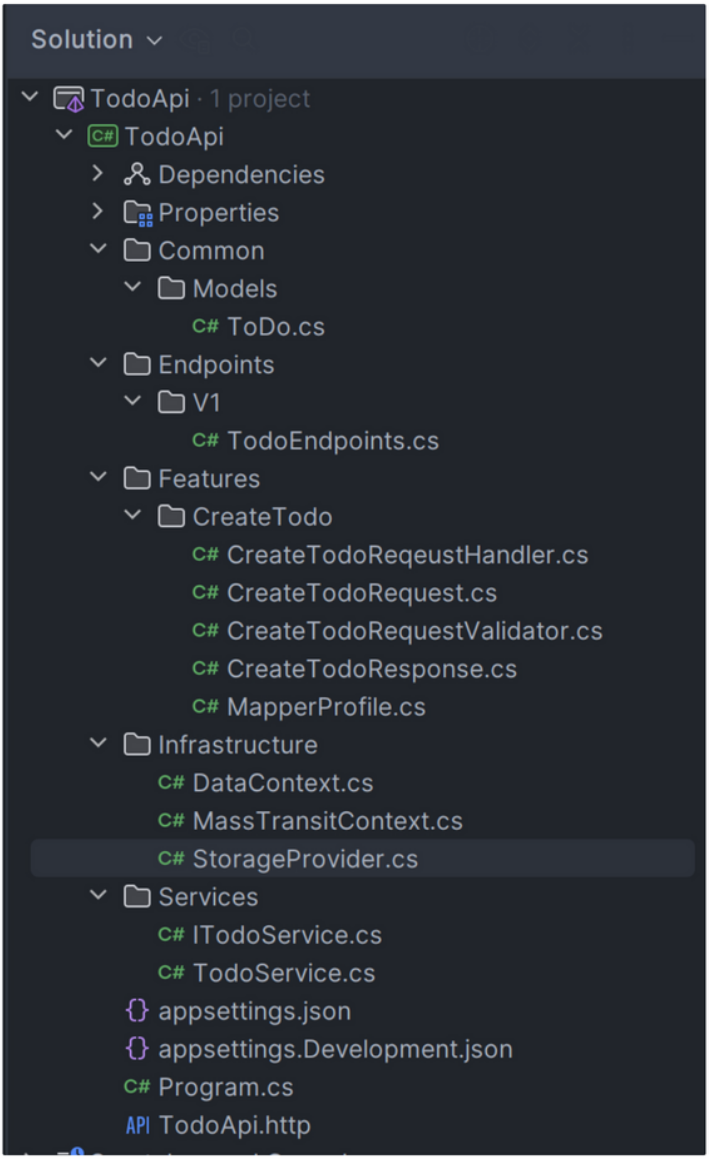
> KISS - Keep it Simple Stupid (Most of time the simplest solution is the best solution)

2.7 Organizing the project

In organizing the Minimal API project folder structure, it is important to establish a clear and logical organization to ensure maintainability and scalability of the codebase. There is a better approach to this by Clean architecture pattern.

However, I think sometimes it's better to not stick to the usual structure and instead choose what works best for your project or team. For example, it's common to have separate projects for Presentation, Infrastructure, Domain, etc., in the same solution. This can be too much to navigate, especially if the DLLs are not meant to be shared. In a world of microservices, it's recommended to have one single API project with the required folder structure. One thing to keep in mind is that it's always recommended to have separate projects for DTOs (these can be contracts to the API and later pushed to a company NuGet) and Tests.

Here is a recommended folder structure:



There is no exact rule, but what works best for your team.

One of the approaches I take when writing routes, is to separate it completely from the program.cs file. For example:

```
public static class TodoEndpointExtensions
{
public static void UseTodoEndpoints(this
IEndpointRouteBuilder endpoints)
{
endpoints.MapGet("api/v1/todos", TodoEndpoints.GetAll);
}
}

public static class TodoEndpoints
{
public static List<Todo> GetAll(HttpContext context) => [];
}
```

As shown in the above example, we could simply write an extension class to expose to program.cs routes pipelines.

```
app.UseTodoEndpoints(); 👈
app.Run();
```

In summary, it can be difficult to create a solid strategy. Excessive nesting in the folder structure can complicate the lives of developers. We should strive to keep the structure simple and easy to navigate.

Yak Shaving : A term for a job or task you do that lead to more jobs and distracts you from your goal

SECTION

03

A Functional approach to C#

Functional programming in C# focuses on writing code that is based on pure functions, immutable data, and avoids side effects, resulting in more predictable and easier-to-debug code. F# is a functional language in .Net, however we could leverage C# to enhance our code to be more functional. In the next sections, I will explain a few but impressive functional concepts. You could use a C# functional library like, LangExt — https://github.com/louthy/language-ext. However, I would be using lightweight nuget packages or sometimes write my own C# code.

Writing in a functional style leads to more expressive code. See the following example where we generate some random names and split & validate them before return in a dictionary:

```
//generate list of names
var names = new List<string>(); Enumerable.Range(1,
10).ToList().ForEach(i =>
{ names.Add($"{i},{
Faker.NameFaker.Name()
}");
});

//Object Oriented Approach
var filteredDictionary = new Dictionary<string, string>();
foreach (var name in names) {
     var split = name.Split(",");
     var id = split[0];
     var firstName = name.Split(" ")[0];
     if(firstName.Length > 8)
     {
       filteredDictionary.Add(id, split[1]);
     }
}

//Functional approach
filteredDictionary = names
 .Select(name => name.Split(","))
 .Where(split => split.Length >= 2)
 .Select(split => new { Id = split[0], FirstName =
split[1].Trim().Split(" ")[0] })
 .Where(entry => entry.FirstName.Length > 8)
.ToDictionary(entry => entry.Id, entry => entry.FirstName);
```

Yes, Linq is functional. C# is becoming more focused on writing code in a functional way. This can be seen in the latest C# releases, which introduce support for Record types and switch expressions. Writing C# with a functional approach is a popular technique nowadays. Railway Oriented Programming is one approach that promotes the use of functional styles in C#.

Now that you have seen how expressive functional code is, let's explore the pros and cons of using functional code.

Pros:

1. **Immutability:** Immutable data structures in functional programming guarantee that data cannot be changed after it is created. This removes the possibility of unexpected side effects and makes the code more predictable, easier to understand, and less likely to have bugs.
2. **Modular and Reusable:** Functional programming encourages breaking down complex problems into smaller functions that can be combined together. These functions can be reused in different parts of the codebase, making the code cleaner and easier to maintain.
3. **Concurrency:** Functional programming promotes the use of pure functions, which do not depend on shared mutable state. This simplifies writing concurrent and parallel code, as functions can be executed separately without concerns about race conditions or data conflicts.
4. **Testability:** Functional programming focuses on using pure functions that do not have any side effects and only rely on their input parameters. This makes it simpler to write unit tests for these functions since their behavior is predictable and can be tested independently.

Cons:

1. **Mutability:** There are situations where a mutable state is needed or can improve performance. Functional programming limits mutability, which can sometimes lead to more complicated code or performance trade-offs in specific cases.
2. **Performance & Memory:** Functional programming often uses recursion and higher-order functions, which can add extra work compared to imperative or object-oriented programming. This can affect performance in certain cases, particularly when working with big datasets or code that needs to be fast.
3. **Steep Learning Curve:** Functional programming is a distinct approach to programming that revolves around the use of immutable data, higher-order functions, and recursion. This can pose a challenge for developers accustomed to imperative or object-oriented programming paradigms.
4. **Lack of Tooling and Libraries:** While functional programming has gained popularity in recent years, it still has a smaller ecosystem compared to imperative or object-oriented programming. This means that there may be fewer libraries, frameworks, and tools specifically designed for functional programming in C#.

Let’s dive deep and learn a few functional approaches.

3.1 Pure Functions

A pure function is a function that, given the same input, always produces the same output and has no side effects. It does not modify any external state or interact with any external resources. Pure functions are predictable, easy to test, and promote code that is more maintainable and less prone to bugs.

```
static int Add(int a, int b)
{
return a + b;
}

Func<string, string> Update => (string) => $"{string}
Updated";
```

3.2 Higher Order Functions (HOF)

Higher-order functions are functions that can take other functions as arguments and/or give functions as results. They are an important concept in functional programming and can make your code more clear and flexible.

```
Func<int, int, int> add = (a, b) => a + b;
Func<int, int, int> multiply = (a, b) => a * b;
int resultAdd = add(3, 4);
int resultMultiply = multiply(3, 4);
```

Real world scenario on validation:

```
public static class Customer {
private static readonly Func<string, bool> IsNameLongEnough =
name => name.Length > 8;
private static readonly Func<string, bool> HasLastName = name
=> name.Split(" ").Length > 1;
private static readonly Func<string, bool> MustStartsWith =
name => name.StartsWith("A");

public static bool ValidateName(string name) =>
IsNameLongEnough(name) && HasLastName(name) &&
MustStartsWith(name);

public static bool ValidateNameByPredicates(string name) =>
name.Validate(IsNameLongEnough, HasLastName, MustStartsWith);
}
public static class Extensions {
public static bool Validate<TInput>(this TInput input, params
Func<TInput, bool>[] predicates) => predicates.All(p =>
p(input));
}
```

Predicate is a Func<TType, bool>. In the above scenario, you can use Predicate<TType> instead of Func<TType, bool>. However, the point is to demonstrate that using Func allows you to do much more.

Extensions are an important part of functional programming in C#. You can use Generic types to fully leverage their advantages.

3.3 Discriminated Unions

A Discriminated Union is a data structure in functional programming that allows for the creation of composite types with multiple variants, enabling pattern matching and precise modeling of complex data.It is useful in scenarios where you need to handle different types of results or outcomes. Here's an example using OneOf nuget package

We have used the following Nuget package to implement discriminated unions, but it is very simple to implement your own.

```
dotnet add package OneOf
```

```
using OneOf;

public OneOf<int, string, bool> ValidateAndReturn(string
input)
{
return input
switch
{
var str when int.TryParse(str, out var intValue) => intValue,
var str when bool.TryParse(str, out var boolValue) =>
boolValue, var str => str, _ => throw new
ArgumentException("Invalid input type")
};
}

var result = ValidateAndReturn("123");
if (result.TryPickT0(out int intValue)) {
Console.WriteLine($"Result is an integer: {intValue}"); }
else if (result.TryPickT1(out string stringValue)) {
Console.WriteLine($"Result is a string: {stringValue}"); }
else if (result.TryPickT2(out bool boolValue)) {
Console.WriteLine($"Result is a boolean: {boolValue}"); }
```

Another example using pattern matching.

```
private static OneOf<bool, Exception> Create()
{
try
{
//Created successfully
return true;
}
catch (Exception e)
{
return e;
}
}

public static void Run()
{
var result = Create();
 result.Switch(success => Console.WriteLine("Created
successfully"), error => Console.WriteLine($"Error occurred:
{error.Message}"));
}
```

3.4 Pattern Matching

One of my favorite things in recent C# is pattern matching using Switch statements. It's a powerful feature that lets developers write shorter and more expressive code for conditions, checking types, and breaking down objects. Let's look at a few examples of pattern matching.

```
public string Execute(ICommand command) => (command)
switch {
OpenCommand o => "Open Command",
CloseCommand o => "Close Command",
ExecuteCommand o => "Execute Command",
_ => throw new ArgumentOutOfRangeException(nameof(command),
command, null)
};
```

Pattern matching with properties:

```
public string Execute(ICommand command) => (command)
switch
{
{ Valid: true } and {Value: > 10 }=> "Open Command",
{ Valid: false } and {Value: <= 10 } => "Close Command",
ExecuteCommand o => "Execute Command",
_ => throw new ArgumentOutOfRangeException(nameof(command),
command, null)
};
```

Pattern matching with Tuples:

```
private static string FormatMobileNumber(string number) =>
(StartsWithPlus6: number.StartsWith("+6"), StartsWith6:
number.StartsWith("6"), StartsWith04:
number.StartsWith("04"))
switch {
(StartsWithPlus6:true, _, _) => number,
(StartsWithPlus6:false, StartsWith6:true, _) =>
$"+61{number[1..]}",
(StartsWithPlus6:false, StartsWith6:false, StartsWith04:true)
=> $"+61{number[1..]}", _ => number

};
```

Note the use of "_" in switch expressions. These are called discards and are important if you don't care about the value. This also helps reduce memory allocations.

3.5 Options (Maybe)

Used to avoid the null mistake which they called a billion dollar mistake. Option is a way to represent a value that may or may not be present. It is used to handle cases where a value could be absent or null without causing null reference exceptions or having to explicitly check for null values. The Option type promotes safer and more predictable code by forcing you to handle both cases: when a value is present (Some) and when it's absent (None).

We can easily implement Options pattern in c# as shown in the following example. There are many C# functional libraries however to which provide these functionality. i.e. LangExt.

```
public interface IOption<T> {
    TResult Match<TResult>(Func<T, TResult> onSome, Func<TResult>
onNone);
}
public class Some<T> : IOption<T>{
    private T _data;
    private Some(T data) {
        _data = data;
    }
    public static IOption<T> Of(T data) => new Some<T>(data);
    public TResult Match<TResult>(Func<T, TResult> onSome, Func<TResult>
_) =>
        onSome(_data);
}
public class None<T> : IOption<T>{
    public TResult Match<TResult>(Func<T, TResult> _, Func<TResult>
onNone) =>
        onNone();
}
```

```
public static void Run()
{
var address = FindAddress("2000");
var addressLabel = address.Match( Some: address => $"
{address.StreetName}, {address.Suburb}, {address.Postcode}", None: () =>
"Address not found" );
}

private static IOption<Address> FindAddress(string postcode) => postcode
switch
{
"2000" => new Address("George Street", "Sydney", "2000"), "3000" =>
Some<Address>.Of(Address("Collins Street", "Melbourne", "3000")), _ =>
new None<Address>()

};
```

3.6 Recursion

Recursion is a powerful concept in functional programming that allows a function to call itself, often with different input values, until a specific condition is met. This approach is particularly useful for solving problems that can be divided into smaller, self-referential subproblems, and it promotes code that is concise and elegant.

Here's a simple example of getting fibonacci sequence in recursion:

```
static int Fibonacci(int n)
{
if (n <= 1) {
return n;
}
else
{
return Fibonacci(n - 1) + Fibonacci(n - 2);
}
}
```

3.7 Record vs Class

The Record type is a new type in C# 9 that provides several features out of the box, making it more convenient to work with than the Class type. Here are the difference between Record and Class and when to use:

Records:

- Records are value-based types that emphasize immutability and equality by default.
- They are primarily used for modeling data and representing immutable objects.
- Records have value semantics, meaning that two records with the same data are considered equal.
- Records provide built-in implementations of the Equals(), GetHashCode(), and ToString() methods based on their properties.
- Records support positional and named properties, making them convenient for data transfer objects (DTOs) and data models.
- Records can be deconstructed using the deconstruct pattern, allowing easy extraction of properties.
- Records can be used in pattern matching expressions, making them useful in functional programming scenarios.
- Records are ideal for scenarios where immutability and equality are desired, such as in functional programming or when working with data models.

Classes:

- Classes are reference-based types that allow for mutable state and provide more flexibility.
- They are used for encapsulating data and behavior, implementing abstractions, and building complex object hierarchies.
- Classes have reference semantics, meaning that two class instances with the same data are considered distinct unless overridden.
- Classes require explicit implementation of the Equals(), GetHashCode(), and ToString() methods for custom equality comparisons.
- Classes can have properties, fields, methods, events, and more, allowing for complex behavior and encapsulation of state.
- Classes can be inherited and extended to create hierarchies and implement inheritance and polymorphism.
- Classes are ideal for scenarios where mutable state and behavior encapsulation are needed, such as in object-oriented programming or when building complex systems.

Records are value-based, immutable, and provide built-in equality and string representation, while classes are reference-based, mutable, and provide more flexibility for encapsulating behavior and state.

Deconstructing a record type can be done with a one-liner using the Deconstruct method. Here's an example:

```
public record Person(string FirstName, string LastName);
var person = new Person("John", "Doe");

// Deconstruct into a tuple with properties.
var (firstName, lastName) = person;
```

In this example, the Deconstruct method is automatically generated by the compiler for the Person record type. It allows us to destructure the person object into its individual properties firstName and lastName in a single line.

Record type also support with keyword. Since Record types are meant to be immutable, we can simply copy the old values and add the new values to construct a new instance.

```
public record Person(string FirstName, string LastName)
{
public Person With(string? FirstName = null, string? LastName
= null)
{
return this with
{
FirstName = FirstName ?? this.FirstName,
LastName = LastName ?? this.LastName
};
}
}
```

SECTION

04

Patterns for scalable and resilience Api

Designing a scalable and resilient API is more of an art than a bunch of patterns. There are many aspects to consider when designing an API. However, with the experience and knowledge we have acquired by designing these systems over many decades, we have come up with some patterns to solve problems that occur frequently. The patterns explained here are important for designing scalable and resilient APIs, but not limited to that. There are many more patterns, but the goal is to familiarize you with these basic patterns so that you can use them and improve the design of your API.

4.1 Concurrent operations

Concurrent operations in C# allow multiple tasks or operations to be executed simultaneously, improving performance and efficiency in multi-threaded applications. C# provides various mechanisms for handling concurrent operations, such as multi-threading, async/await, and parallel programming.

In multi-threading, multiple threads are created to execute different tasks concurrently. For example, you can use the Thread class to create and start multiple threads that perform different operations simultaneously.

```
void DoTask1()
{
// Perform task 1
}
void DoTask2()
{
// Perform task 2
}

// Create and start two threads
Thread thread1 = new Thread(DoTask1);
Thread thread2 = new Thread(DoTask2);
thread1.Start();
thread2.Start();
```

Async/await is another approach for handling concurrent operations in C#. It allows you to write asynchronous code that can execute multiple tasks concurrently without blocking the main thread. For example, you can use the Task class and the await keyword to asynchronously execute multiple tasks.

```
async Task DoTask1Async()
{
// Perform task 1 asynchronously
}

async Task DoTask2Async()
{
// Perform task 2 asynchronously
}

// Execute multiple tasks asynchronously
Task task1 = DoTask1Async();
Task task2 = DoTask2Async();
await Task.WhenAll(task1, task2);
```

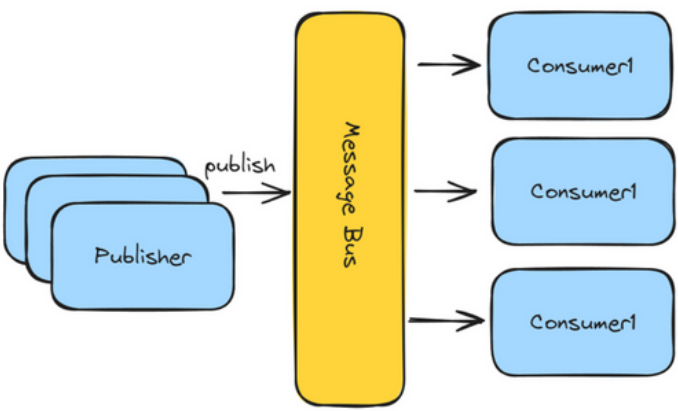
Parallel programming is another way to handle concurrent operations in C#. It allows you to divide a task into smaller sub-tasks that can be executed in parallel on multiple threads. For example, you can use the Parallel class to parallelize a loop and execute its iterations concurrently.

```
void ProcessData(int[] data)
{
Parallel.For(0, data.Length, i =>
{
// Process each element of the data array concurrently
});
}
```

These examples demonstrate different approaches to handle concurrent operations in C#, providing flexibility and performance improvements in multi-threaded scenarios.

4.2 Pub/Sub Pattern

Publish/Subscribe(pub/sub), is a messaging pattern used in distributed systems to enable communication between components or services. It is a vital solution in Microservices architecture. It provides a way for publishers to send messages, or events, to multiple subscribers who are interested in receiving those messages.



In C#, you can implement the Pub/Sub pattern using various messaging frameworks or libraries, such as RabbitMQ, Azure Service Bus, MassTransit, or even custom implementations using in-memory message brokers.

Here's an example of how you can implement Pub/Sub using the MediatR library in C# which we used in the above examples:

1. Define the messages (events) that you want to publish and subscribe to. For example, let's define a TodoCreated event:

```
public class TodoCreated : INotification
{
public int TodoId { get; set;
}
// Other properties...
}
```

2. Create the event handlers (subscribers) that will handle the published events. For example, let's create an TodoCreatedHandler:

```
public class TodoCreatedHandler :
INotificationHandler<TodoCreated>
{
public Task Handle(
TodoCreated notification,
CancellationToken cancellationToken)
{
// Handle the TodoCreated event
Console.WriteLine($"Todo {notification.TodoId}");
return Task.CompletedTask;
}
}
```

3. Configure the MediatR pipeline and register the event handlers. This can be done in the ConfigureServices method of your Startup class:

```
public void ConfigureServices(IServiceCollection services)
{
// Other service registrations...
services.AddMediatR(typeof(Startup));
services.AddTransient<INotificationHandler<TodoCreated>,
TodoCreatedHandler>();
}
```

4. Publish the event when it occurs. For example, let's publish an TodoCreated event:

```
public class TodoService : ITodoService
{
private readonly IMediator _mediator;
public TodoService(IMediator mediator)
{
_mediator = mediator;
}

public async Task CreateTodo(Todo order)
{
// Create the todo...
var todoCreated = new TodoCreated {
TodoId = todo.Id
};
await _mediator.Publish(todoCreated);
}
}
```

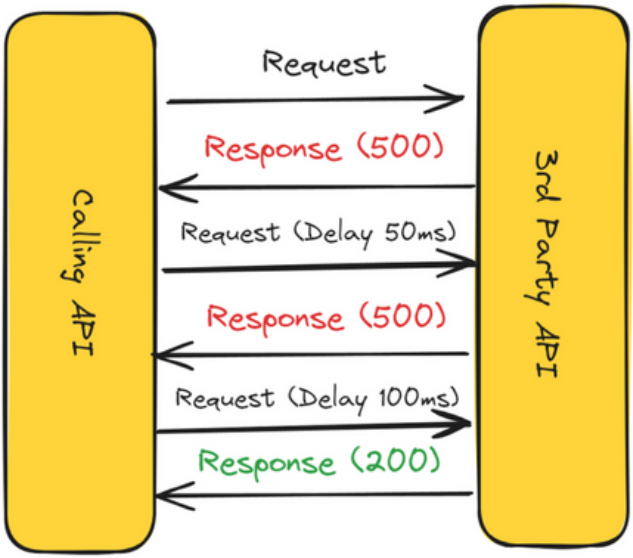
In this example, when a todo item is created, the TodoService publishes an TodoCreated event using the IMediator interface provided by MediatR. The event will be sent to all registered event handlers (TodoCreatedHandler in this case).

When the event is received by the event handler, the Handle method of the handler is executed, allowing you to perform any necessary logic or actions based on the event.

You can decouple components or services in your application, allowing them to communicate asynchronously and independently. True scalability comes alone when we introduce a message bus such as Azure Service Bus or RabbitMQ in between which acts as a container which holds all the messages. This promotes better scalability, modularity, and flexibility in your system architecture.

4.3 Retry Pattern

The Retry pattern is used to handle transient failures in distributed systems by retrying failed operations. It provides a mechanism to automatically retry an operation that has failed due to a temporary issue, such as a network timeout or a database connection failure. This pattern helps improve the resilience and availability of the system.



Polly is a popular library in C# that provides support for implementing the Retry pattern. With **Polly**, you can easily configure the retry behavior, including the number of retry attempts, the delay between retries, and the conditions under which a retry should be performed.

You could easily add by using the Polly nuget package.

```
dotnet add package Polly
```

```
var retryPolicy = Policy
.Handle<SomeException>()
.WaitAndRetry(
retryCount: 3,
sleepDurationProvider: retryAttempt =>
TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
);
try
{
retryPolicy.Execute(() => {
// Perform the operation that may fail and need to be retried
});
}
catch (SomeException ex)
{
// Handle the exception or take appropriate action
}
```

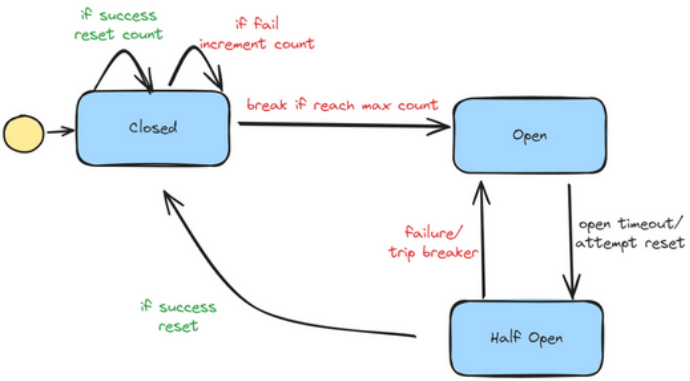
In this example, the Handle method is used to specify the type of exception that should trigger a retry. The WaitAndRetry method is then called to configure the retry policy with the number of retry attempts (retryCount) and the sleep duration between retries (sleepDurationProvider). The sleepDurationProvider function is used to calculate the delay between retries, with each retry attempt waiting for an exponentially increasing duration.

By using the Retry pattern and Polly library, you can easily add resilience to your code by automatically retrying failed operations, reducing the impact of transient failures and improving the overall reliability of your system.

4.4 Circuit Breaker Pattern

The Circuit Breaker pattern is a design pattern used in software development to handle and recover from failures in distributed systems. It is especially helpful in situations where a service or resource may become unavailable or start experiencing delays.

The pattern works by monitoring the availability of a service or resource. When the service is healthy and responsive, the circuit is closed, allowing requests to be executed as normal. However, if the service starts to fail or become unresponsive, the circuit is opened, preventing further requests from being executed. This helps to protect the system from cascading failures and provides an opportunity for the service to recover.



Here is an example of how to use Polly's Circuit Breaker policy:

```
var circuitBreakerPolicy = Policy
.Handle<SomeException>()
.CircuitBreaker(
exceptionsAllowedBeforeBreaking: 3,
durationOfBreak: TimeSpan.FromSeconds(30)
);

try
{
 circuitBreakerPolicy.Execute(() => {
// Call the service or execute the operation
// that may throw SomeException
});
}
catch (SomeException ex)
{
// Handle the exception or take appropriate action
}
```

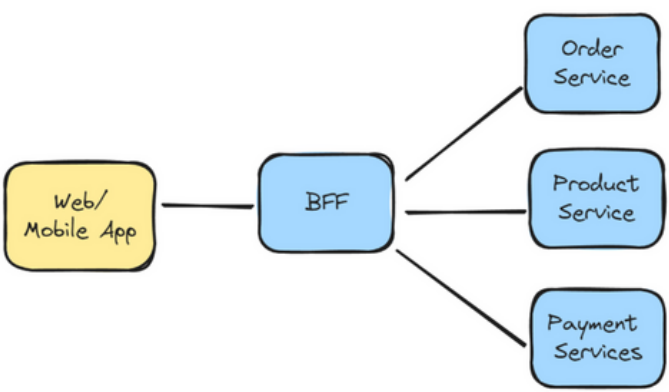
In this example, the Handle method is used to specify the type of exception that should trigger the circuit breaker. The CircuitBreaker method is then called to configure the circuit breaker policy with the number of allowed exceptions before breaking (exceptionsAllowedBeforeBreaking) and the duration of the break (durationOfBreak).

When the circuit breaker is open, any additional calls to the service or operation protected by the policy will fail right away without carrying out the actual operation. This helps to stop further requests to the failing service and gives it a chance to recover. After the specified duration of the break, the circuit breaker will move to a half-open state, allowing one request to be executed. If that request is successful, the circuit breaker will close again, enabling normal operation. If the request fails, the circuit breaker will open again, extending the break duration.

Using Polly's Circuit Breaker policy can greatly improve the resiliency and reliability of your applications by handling and recovering from failures in a controlled manner.

4.5 BFF Pattern

The Backend for Frontend (BFF) pattern is an architectural pattern where a dedicated backend service is created to cater to the specific needs of a frontend application. This allows the frontend to have a tailored API that provides the required data and functionality efficiently. For example, a mobile app may have its own BFF service that provides optimized endpoints and data structures for mobile-specific features, improving performance and user experience.



Pros :

1. **Tailored API:** The BFF pattern allows the frontend application to have a dedicated backend service that provides optimized endpoints and data structures specifically designed for the frontend's needs. This results in improved performance and user experience.
2. **Micro-Frontends:** BFF enables the implementation of micro-frontends, where different parts of the frontend can be developed independently and have their own dedicated BFF services. This promotes modularity and scalability in frontend development.
3. **Reduced Overfetching/Underfetching:** With a dedicated BFF, the frontend can request only the data it needs for a specific view or feature, reducing unnecessary data transfer and improving efficiency.
4. **Improved Development Velocity:** The BFF pattern allows frontend and backend teams to work independently and at their own pace. This can lead to faster development cycles and quicker time-to-market for new features or updates.

Cons :

1. **Increased Complexity:** Introducing a dedicated backend service for each frontend can lead to increased complexity in the overall architecture and deployment. Managing multiple BFF services and their interactions requires careful coordination and maintenance.
2. **Duplication of Effort:** With separate backend services for each frontend, there may be some duplication of effort in terms of logic, validation, and data retrieval. This can result in increased development and maintenance overhead.
3. **Potential for Inconsistent APIs:** If not properly managed, the BFF pattern can lead to inconsistencies in the APIs exposed to the frontend. Different BFF services may have variations in naming conventions, data structures, or error handling, which can create confusion and inconsistency in the frontend codebase.
4. **Increased Network Traffic:** Having separate backend services for each frontend can potentially increase network traffic, especially if multiple frontend applications are making requests simultaneously. This can impact scalability and performance, requiring careful consideration of network optimization strategies.

It's important to carefully evaluate the requirements and constraints of your specific application before deciding to adopt the BFF pattern. While it offers benefits in terms of tailored APIs and micro-frontends, it also introduces additional complexity and coordination challenges in the architecture.

4.6 Saga Pattern

The Saga pattern is a design pattern used in distributed systems to handle long-running transactions or business processes that span multiple services. It ensures that all the steps in the process are either completed successfully or rolled back if any step fails.

For example, in an e-commerce application, a Saga can be used to handle the process of placing an order, which involves multiple services such as inventory management, payment processing, and shipping. If any step fails, the Saga can initiate compensating actions to undo the changes made in the previous steps and maintain data consistency.

This is vital in distributed microservices architecture where an operation can be spaned in multiple microsservices.

Pros:

1. **Ensures Data Consistency:** The Saga pattern makes sure that data consistency is maintained across multiple services involved in a long-running transaction or business process. If any step fails, actions can be taken to undo the changes made in the previous steps, keeping the data consistent.
2. **Handling Distributed Transactions:** Saga helps in managing transactions that involve multiple services running independently. It coordinates the execution of multiple steps and ensures that the transaction is either successfully completed or rolled back.
3. **Scalability and Availability:** Saga allows for scalability and availability in distributed systems. Each step can be executed independently, enabling parallel processing and reducing the load on individual services. In case of failure, only the affected steps need to be retried or rolled back, minimizing the impact on the entire process.
4. **Fault Tolerance:** Saga promotes fault tolerance in distributed systems. If any step fails, compensating actions can be triggered to revert the changes and restore the system to a consistent state. This helps in recovering from failures and maintaining system integrity.

Cons:

1. **Increased Complexity:** Using the Saga pattern makes the system architecture more complicated. It needs careful coordination and management of multiple steps and services in a long-running transaction. This complexity can make development, testing, and debugging harder.
2. **Compensating Actions:** Designing and implementing compensating actions for each step in the Saga can be complex and error-prone. It requires a thorough understanding of the dependencies and side effects of each step to ensure proper rollback and data consistency.
3. **Synchronization and Coordination:** Coordinating the execution of multiple steps and ensuring their proper sequence and synchronization can be challenging. It requires mechanisms for communication, coordination, and error handling between different services involved in the Saga.
4. **Performance Overhead:** Saga introduces additional overhead in terms of communication, coordination, and data management. The need for compensating actions and the potential for retries or rollbacks can impact performance, especially in scenarios with high concurrency or high transaction volumes.

It is crucial to think about the pros and cons and what your system needs before choosing to use the Saga pattern. Although it provides advantages in terms of data consistency and fault tolerance, it also brings in complexity and performance considerations that require careful handling.

Here is a sample implementation:

```
public interface ISagaStep {
void Execute();
void Compensate();
}

public class Step1 : ISagaStep {
public void Execute()
{
// Perform Step 1
}

public void Compensate()
{
// Compensate for Step 1
}
}

public class Step2 : ISagaStep
{
public void Execute()
{
// Perform Step 2
}
public void Compensate()
{
// Compensate for Step 2
}
}

public class Saga
{
private readonly List<ISagaStep> steps;
public Saga(List<ISagaStep> steps)
{
this.steps = steps;
}
public void ExecuteSaga()
{
foreach (ISagaStep step in steps) {
try {
step.Execute();
}
catch
{
// If any step fails, compensate for the previous steps CompensateSteps();
throw;
}
}
}
private void CompensateSteps() {
for (int i = steps.Count - 1; i >= 0; i--) {
try { steps[i].Compensate();
} catch {
// Ignore compensation failures
}
}
}
}
// Usage
List<ISagaStep> steps = new List<ISagaStep> { new Step1(), new Step2() };
Saga saga = new Saga(steps);
saga.ExecuteSaga();
```

That’s all - Tell us what you think

We would absolutely love to hear your feedback. What did you get out of reading this book?
How easy/hard was it to follow? Is there something that you would like to see in the next book?

https://docs.google.com/forms/d/e/1FAIpQLSf0q-R289Erw3d0zULTkV61k0NzwIjtIMiKghdrs6J_SevBrg/viewform?usp=sf_link