Dobre praktyki wpływające na czytelność kodu na przykładzie aplikacji ASP.NET Core

Mateusz Bryll
Manager, Engineering

allegro PAY

allegro PAY

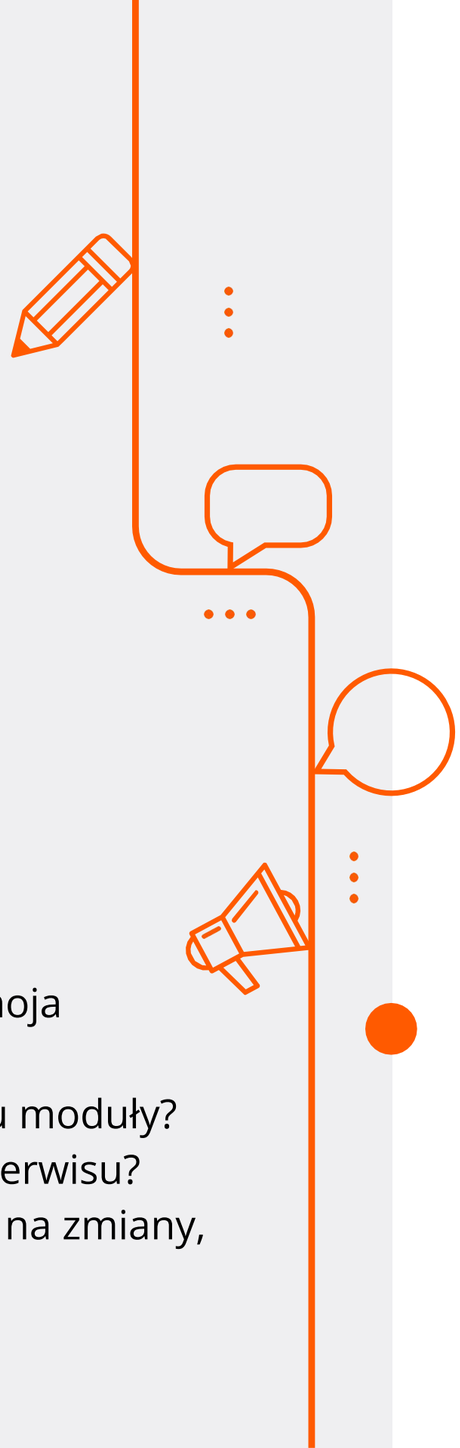
O mnie

Mateusz Bryll, Engineering Manager @ Allegro Pay

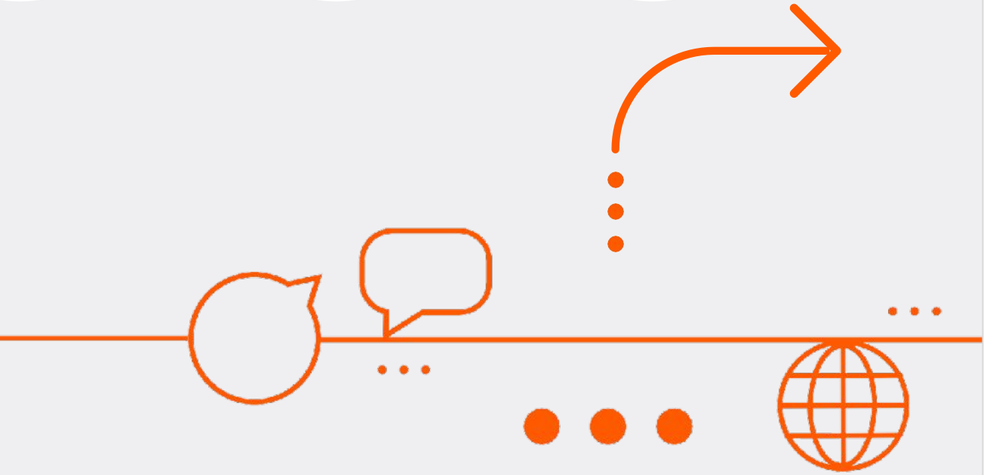
- Jestem liderem zespołu Snatch, który na co dzień opiekuje się dwoma obszarami w Allegro Pay:
  - Onboarding nowych klientów do usługi.
  - Procesy związane z kontem użytkownika po onboardingu.
- Zespół znajduje się w kilku lokalizacjach: Warszawa, Kraków, Wrocław i Poznań.
- Oprócz tego programista, mentor, człowiek orkiestra.
- Więcej na https://mateuszbryll.com/

Agenda

- Teoria (do samodzielnej lektury)
  - Coupling & cohesion
  - Konwencje
  - SOLID
  - Wzorce projektowe
  - Architektura
- Praktyka
  - Co właściwie na pierwszy rzut oka robi moja aplikacja?
  - Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?
  - Jak wydzielić moduł do osobnego mikroserwisu?
  - Kod otwarty na rozszerzenia i zamknięty na zmiany, ale jak?



mateuszbryll.com/put2025



Teoria

Konwencje...

CSharp nie gęś,...

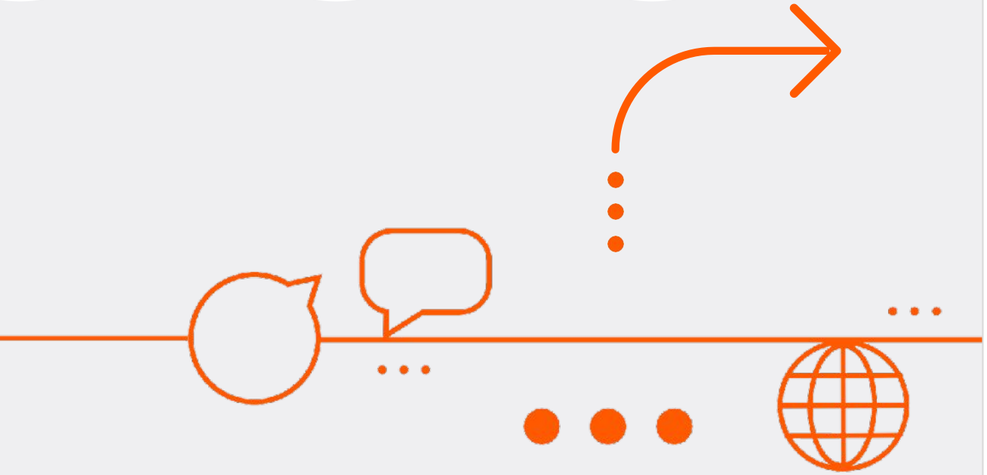
... i swoje konwencje ma. - Parafrazując Mikołaja Reja...

C# jak każdy język programowania posiada swoje konwencje nazywania zmiennych, stałych, pól, właściwości, klas, interfejsów... I możemy tak jeszcze chwilę wymieniać ;)

Pracując w projekcie rozwijanym w tym języku warto znać te powszechne konwencje i dobre praktyki, aby oszczędzić czas seniorom, którzy na pewno zwrócą na to uwagę przy code review ;)

Linki

- Konwencje nazywania
- Konwencje kodowania



Teoria

Coupling & cohesion...

Czy da się mierzyć czytelność?

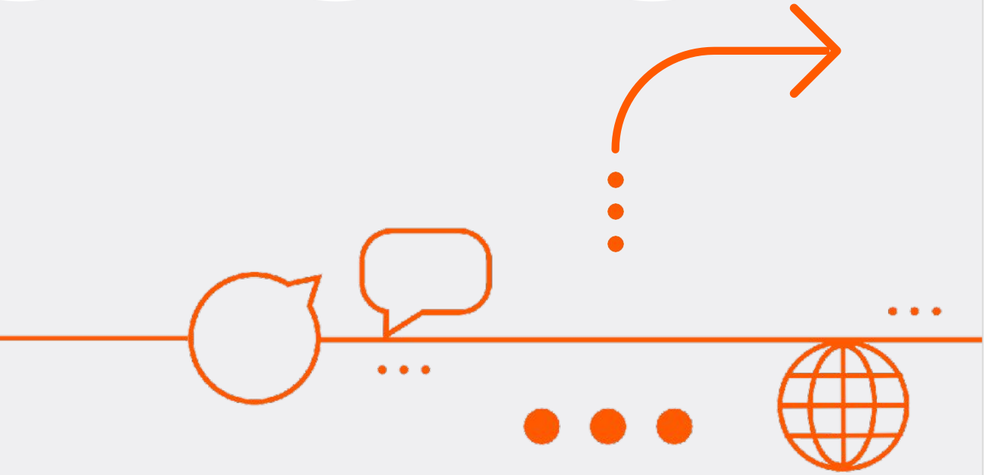
Okazuje się, że tak!
A z perspektywy inżyniera to bardzo dobra wiadomość, gdyż ma on dostęp do odpowiednich metryk które może optymalizować.
Wspomniane metryki to:

- **Coupling** określa jak bardzo moduły są ze sobą powiązane. Wysoki coupling to wysokie powiązanie, niski to małe.
- **Cohesion** określa w jakim stopniu elementy modułu współpracują ze sobą.

W codziennej pracy inżynier stara się minimalizować coupling i maksymalizować cohesion.

Linki

- Coupling & cohesion 1
- Coupling & cohesion 2
- Metryki kodu w Visual Studio
- Microsoft patterns and practices
- [VIDEO] Coupling and Cohesion



Teoria

SOLID...

Od czego zacząć?

Single Responsibility Principle

Open/closed Principle

Liskov Substitution Principle

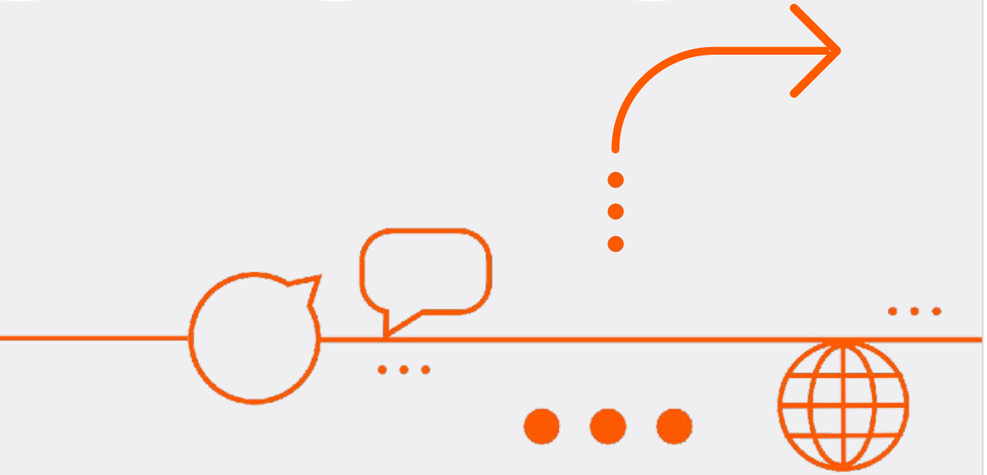
Interface Segregation Principle

Dependency Inversion Principle

Uncle Bob Martin powiedział w tym temacie chyba wszystko, dlatego w tym temacie odsyłam w linkach bezpośrednio do eksperta.

Linki

- [VIDEO] Uncle Bob SOLID principles
- Clean Code
- Agile Software Development



Teoria

Wzorce projektowe...

Po co wynajdywać koło na nowo?

Skoro jakieś rozwiązania powtarzają się ciągle i są dobrze sprawdzone, a przede wszystkim ponadczasowe.To dlaczego się ich nie nauczyć?

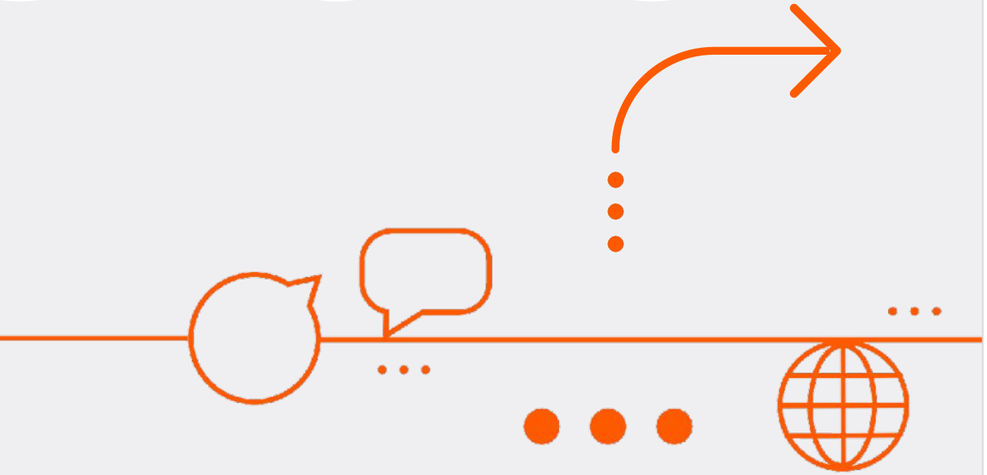
Które stosowałem w swojej praktyce najwięcej?

- Wszystkie! Tak wzorce pojawiają się wszędzie, a to że ich nie widzimy wynika głównie z tego, że ich nie znamy, a co za tym idzie nie zauważamy ich użycia.

Zastosowanie wzorca praktycznie zawsze oznacza czystszy kod ;)

Linki

- ➔ Wiki - wzorce projektowe
- ➔ Design patterns cheat sheet
- ➔ Gang Czworga: Wzorce projektowe
- ➔ Head First Design Patterns



Teoria

Architektura...

Czym jest architektura aplikacji?

“Architekturę w kontekście wytwarzania oprogramowania możemy postrzegać jako metaforę zaczerpniętą ze wznoszenia budynków. U samych podstaw projektowanie nowej budowli oraz rozpoczęcie pracy nad nową aplikacją mają wiele wspólnego. Już w I w. p.n.e. rzymski inżynier i architekt Marcus Vitruvius Pollio opisał cechy jakie powinien posiadać dobry projekt architektoniczny. Są to trwałość (*łac. firmitas*), użyteczność (*łac. utilitas*) oraz piękno (*łac. venustas*). [...]”

Tak zaczyna się moja praca magisterska. Istnieje kilka różnych podejść do architektury aplikacji. W linkach znajdziesz materiały, od których warto zacząć zgłębiać temat.

Linki

- N-Tier
- Clean architecture / Onion arch. / Hexagonal arch.
- [VIDEO] Modularny monolit
- Mikroserwisy
- [VIDEO] CQS, CQRS, Event Sourcing
- [VIDEO] The Art of Destroying Software

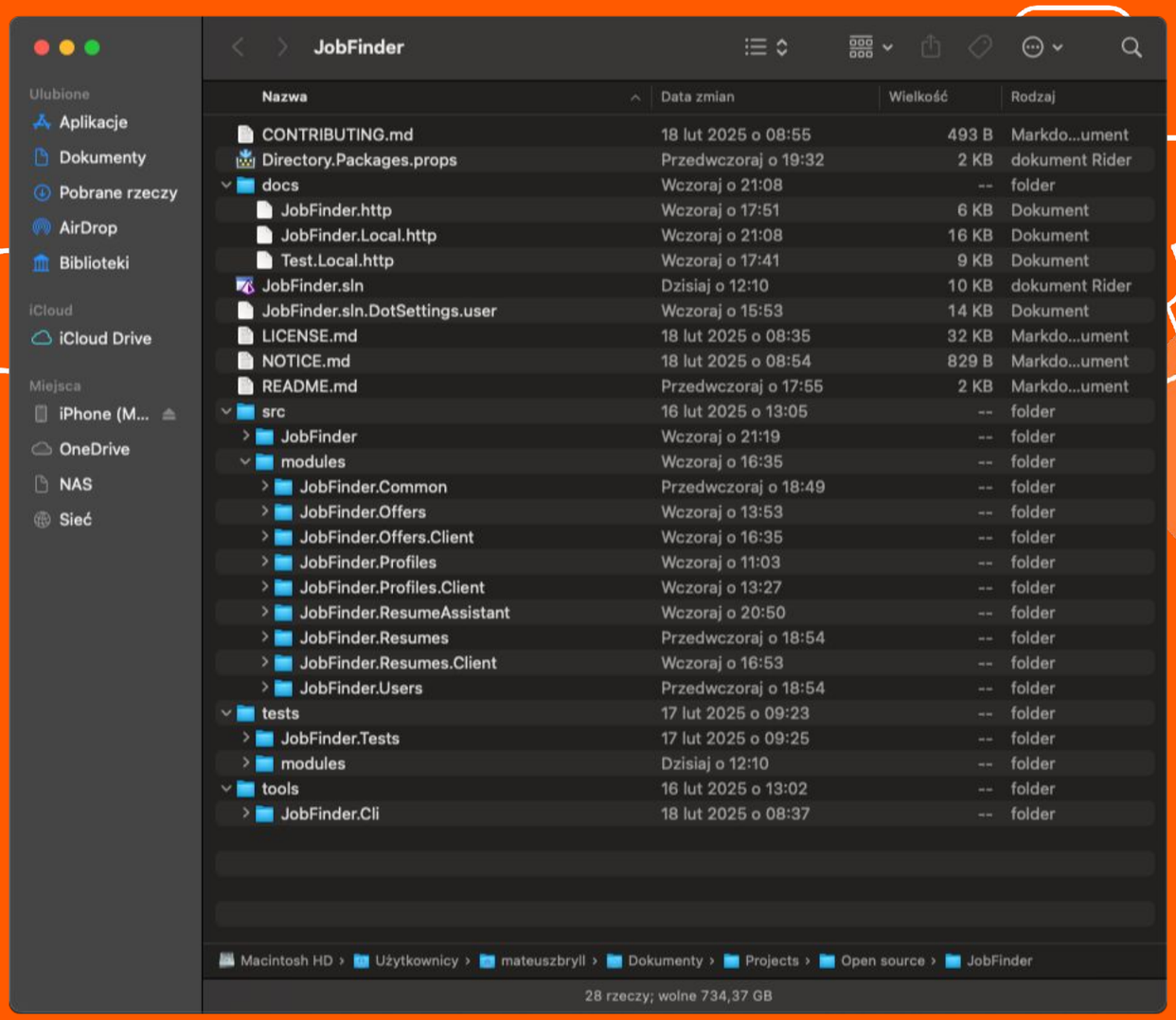
Praktyka...

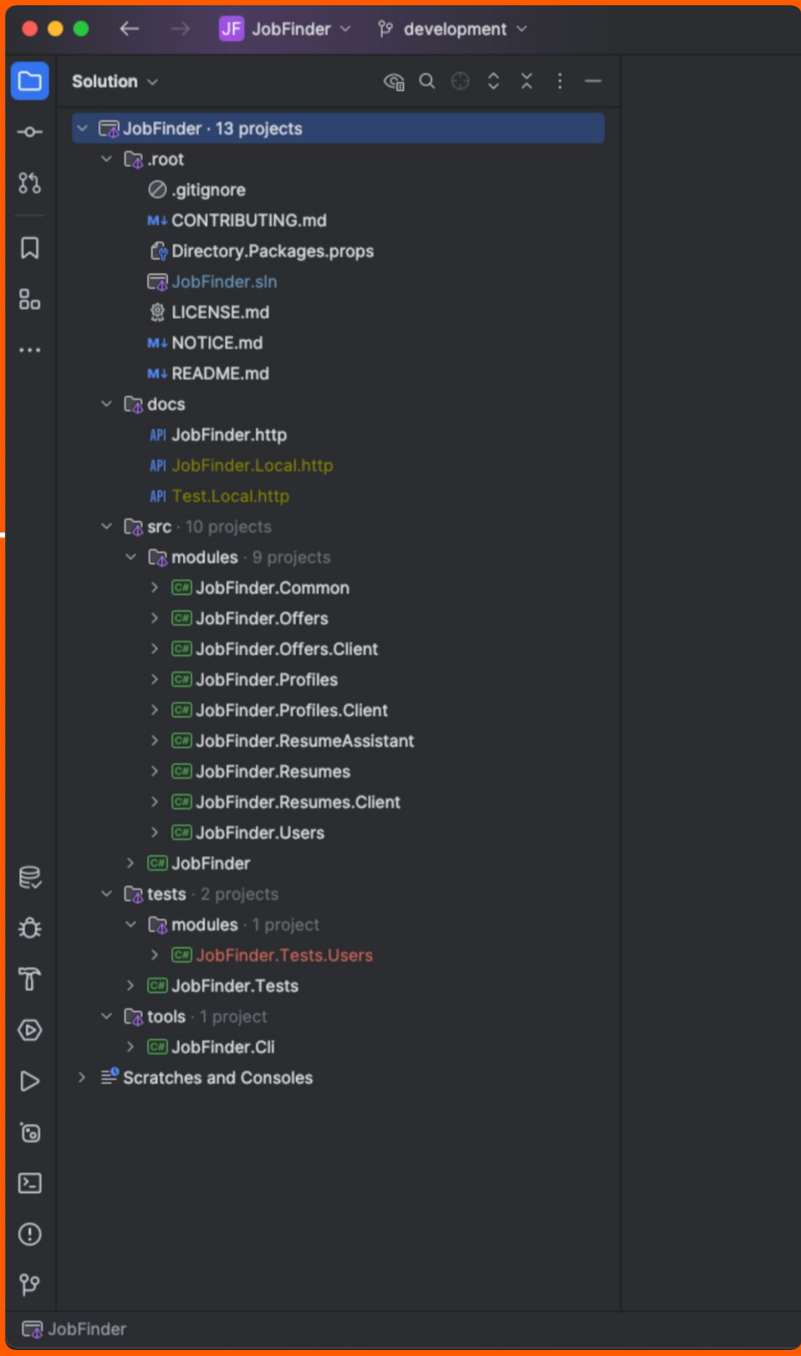
Program.cs

```
1  Console.WriteLine("Hello World!");
```

Struktura projektu

Co właściwie na pierwszy rzut oka robi moja aplikacja?

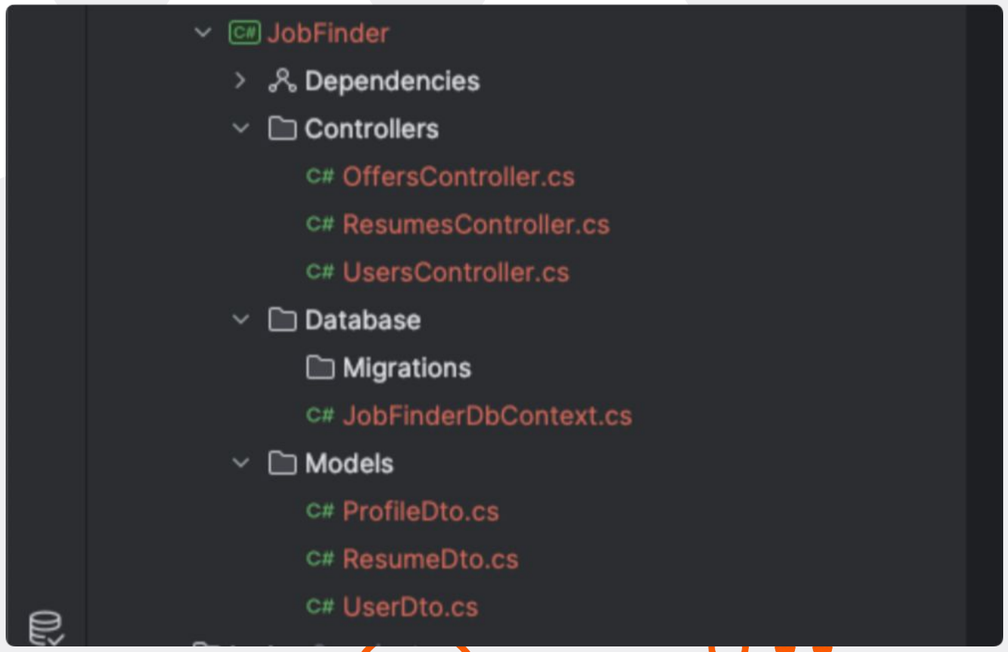




Struktura projektu

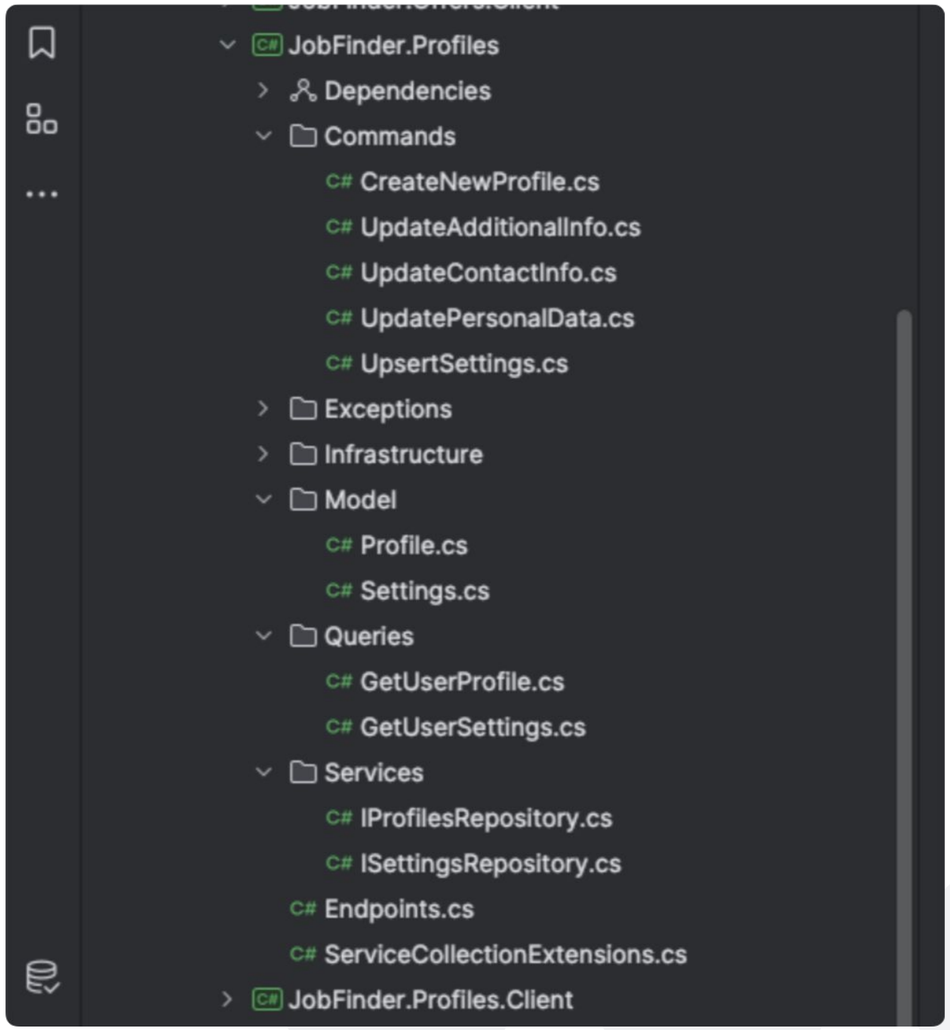
Co właściwie na pierwszy rzut oka robi moja aplikacja?

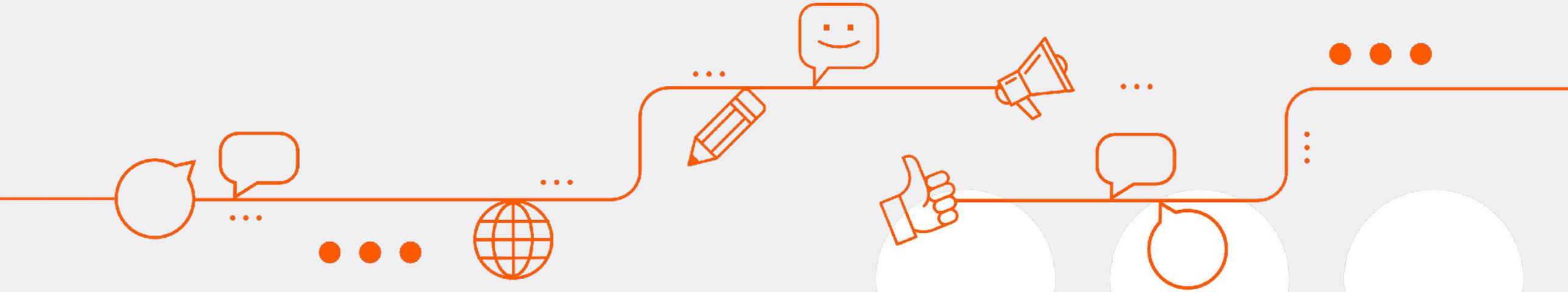
Typowa struktura projektu MVC



VS

Struktura zorientowana na funkcjonalność





Jak podzielić kod na moduły?

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

Program.cs

C# ./src/JobFinder.csproj

Plik **Program.cs** służy do konfiguracji aplikacji Web API. Główny projekt nie posiada własnej domeny – logika biznesowa znajduje się w modułach.

Moduły dodajemy w dwóch krokach:

- ➔ **Add...Module(...);** – rejestracja modułu w kontenerze DI (Dependency Injection).
- ➔ **Use...Module();** – aktywacja modułu i udostępnienie jego endpointów.

JobFinder/Program.cs

```
using JobFinder;
using JobFinder.Offers;
using JobFinder.Profiles;
using JobFinder.ResumeAssistant;
using JobFinder.Resumes;
using JobFinder.Users;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddApplicationInfrastructure(builder.Configuration);

var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
var migrationsAssembly = typeof(Program).Assembly.GetName().Name;

builder.Services.AddUsersModule(builder.Configuration, options =>
    options.UseNpgsql(connectionString, x => x.MigrationsAssembly(migrationsAssembly)));
builder.Services.AddProfilesModule(options =>
    options.UseNpgsql(connectionString, x => x.MigrationsAssembly(migrationsAssembly)));
builder.Services.AddResumesModule(options =>
    options.UseNpgsql(connectionString, x => x.MigrationsAssembly(migrationsAssembly)));
builder.Services.AddOffersModule(options =>
    options.UseNpgsql(connectionString, x => x.MigrationsAssembly(migrationsAssembly)));
builder.Services.AddResumeAssistantModule();

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

app.UseUsersModule();
app.UseProfilesModule();
app.UseResumesModule();
app.UseOffersModule();
app.UseResumeAssistantModule();

app.Run();
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

ServiceCollectionExtensions.cs

C# ./src/modules/JobFinder.Offers.csproj

Każdy moduł rejestruje swoje zależności w kontenerze DI poprzez metodę rozszerzającą **IServiceCollection**.

Rejestracja przykładowego modułu może obejmować:

- Rejestracje DbContext - konfigurację bazy danych modułu.
- Dodanie walidatorów (np. Automatycznie używając FluentValidation).
- Rejestrację klientów innych modułów.
- Rejestrację pozostałych zależności, np. Serwisów, komend, itp.

JobFinder.Offers/ServiceCollectionExtensions.cs

```
// Usings

namespace JobFinder.Offers;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddOffersModule(
        this IServiceCollection services, Action<DbContextOptionsBuilder> builder)
    {
        ArgumentNullException.ThrowIfNull(builder);

        services.AddDbContext<OffersDatabaseContext>(builder);
        services.AddValidatorsFromAssembly(typeof(ServiceCollectionExtensions).Assembly,
            includeInternalTypes: true);

        services.AddProfilesClient();

        services.AddScoped<IOpenApiClientFactory, OpenApiClientFactory>();
        services.AddScoped<IOfferParser, ChatGptOfferParser>();
        services.AddScoped<IOfferRepository, OfferRepository>();

        services.AddScoped<ICommandHandler<CreateOffer>, ParseOfferHandler>();
        services.AddScoped<IQueryHandler<GetOffer, Offer>, GetOfferHandler>();

        return services;
    }
}
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

Endpoints.cs

C# ./src/modules/JobFinder.Offers.csproj

Plik **Endpoints.cs** służy do udostępnienia API modułu. Każdy moduł rejestruje swoje endpointy w metodzie rozszerzającej **IEndpointRouteBuilder**.

Moduły używając .NET Minimal API mogą:

- ➔ Definiować endpointy i metody HTTP jak np. **MapGet(...), MapPost(...), MapPut(...), itp.**.
- ➔ Wstrzykiwać swoje zależności z kontenera DI np. **IQueryBus**, parametry z adresu URL np. **offerId**, czy przykładowo zdeserializowane body żądania.

JobFinder.Offers/Endpoints.cs

```
namespace JobFinder.Offers;

public static class Endpoints
{
    public static IEndpointRouteBuilder UseOffersModule(this IEndpointRouteBuilder builder)
    {
        // ...

        builder.MapGet("/offers/{offerId}", async (HttpContext context, OfferId offerId,
            IQueryBus queryBus) =>
        {
            var userId = context.User.GetUserId();
            var query = new GetOffer(userId, offerId);

            var offer = await queryBus.SendAsync<GetOffer, Offer>(query);

            return Results.Ok(offer);
        }).RequireAuthorization();

        return builder;
    }
}
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

JobFinder.Offers.csproj

C# ./src/modules/JobFinder.Offers.csproj

Aby w projekcie mieć dostęp do interfejsów IServiceCollection oraz IEndpointRouteBuilder w pliku .csproj musimy dodać referencję do frameworka ASP.NET Core.

Robimy to w następujący sposób:

- ➔ Dodajemy w pliku nowy element **<ItemGroup>**
- ➔ Następnie w nim dodajemy element **<FrameworkReference>**.

JobFinder.Offers/JobFinder.Offers.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net9.0</TargetFramework>
        <ImplicitUsings>enable</ImplicitUsings>
        <Nullable>enable</Nullable>
    </PropertyGroup>

    <ItemGroup>
        <FrameworkReference Include="Microsoft.AspNetCore.App" />
    </ItemGroup>

    ...

</Project>
```

Jak obsługiwać błędy?

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

StartupExtensions.cs

C# ./src/JobFinder.csproj

W pliku **StartupExtensions.cs** znajduje się rejestracja i konfiguracja domyślnej obsługi odpowiedzi API serwisu po wystąpieniu wyjątku.

Skonfigurować możemy:

- ➔ **AddProblemDetails(...);** – zachowanie dla wszystkich nieobsłużonych wyjątków.
- ➔ **AddExceptionHandler<...>();** –klasę odpowiedzialną za obsługę wyjątków.
- ➔ Jest to standard opisany w RFC 9457. Możemy zwrócić również **return Results.Problem(...);** z naszego API.

JobFinder/StartupExtensions.cs

```
namespace JobFinder;

internal static class StartupExtensions
{
    public static IServiceCollection AddApplicationInfrastructure(
        this IServiceCollection services,
        IConfiguration configuration)
    {
        // ...

        services.AddProblemDetails(options ⇒
        {
            options.CustomizeProblemDetails = ctx ⇒
            {
                ctx.ProblemDetails.Extensions.Add("instance",
                    $"{ctx.HttpContext.Request.Method} {ctx.HttpContext.Request.Path} \
                        {ctx.HttpContext.Request.Protocol}");
            };
        });
        services.AddExceptionHandler<ApplicationExceptionHandler>();

        // ...
    }
}
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

App...ExceptionHandler.cs

C# ./src/JobFinder.csproj

ASP.NET Core posiada dedykowany serwis ułatwiający nam obsługę wyjątków aplikacji. Aby dodać niestandardową obsługę wyjątków musimy zaimplementować interfejs **IExceptionHandler**, a w konstruktorze naszego handlera możemy użyć serwisu **IProblemDetailsService**, który odpowiednio sformatuje odpowiedź.

Handler może wskazać, że nie obsługuje danego typu wyjątku zwracając **false** z metody **TryHandleAsync** lub **true**, gdy wyjątek został odpowiednio obsłużony.

JobFinder/Infrastructure/ApplicationExceptionHandler.cs

```cs
namespace JobFinder.Infrastructure;

internal class ApplicationExceptionHandler(IProblemDetailsService problemDetailsService) :
    IExceptionHandler
{
    public async ValueTask<bool> TryHandleAsync(
        HttpContext httpContext,
        Exception exception,
        CancellationToken ct)
    {
        if (exception is not DomainException domainException)
        {
            return false;
        }

        httpContext.Response.StatusCode = StatusCodes.Status400BadRequest;
        return await problemDetailsService.TryWriteAsync(new ProblemDetailsContext
        {
            HttpContext = httpContext,
            Exception = domainException,
            ProblemDetails =
            {
                Title = domainException.Title,
                Detail = domainException.Details,
                Type = domainException.Type,
                Status = StatusCodes.Status400BadRequest,
                Extensions = domainException.Extensions
            }
        });
    }
}
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

DomainException.cs

C# ./src/modules/JobFinder.Common.csproj

W poprzednim przykładzie użyty został typ **DomainException** do odfiltrowania obsługiwanych wyjątków. Obok znajduje się jego definicja.

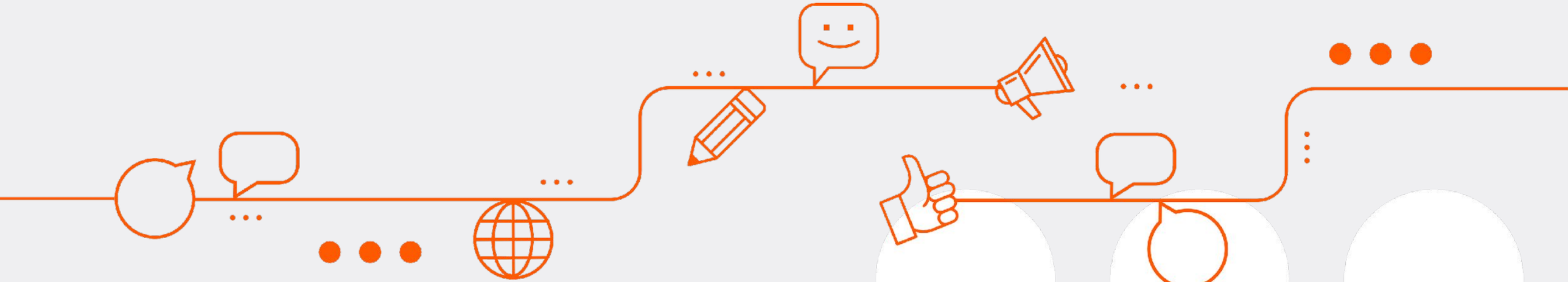
Wprowadzając customowy typ wyjątku:

- ➔ Umożliwiamy domenie naszej aplikacji zwracanie błędów w ustandaryzowany sposób.
- ➔ Jesteśmy otwarci na rozszerzenia - moduły mogą wprowadzać kolejne typy wyjątków, które dziedziczą po DomainException.

JobFinder.Common/DomainException.cs

```
using Humanizer;

namespace JobFinder.Common;

public abstract class DomainException : Exception
{
    public string Details { get; }
    public IDictionary<string, object?> Extensions { get; }
    public string Title => GetType().Name.Humanize();
    public string Type => GetType().FullName!;

    protected DomainException(string details,
        IDictionary<string, object?>? extensions = null) : base(details)
    {
        Details = details;
        Extensions = extensions ?? new Dictionary<string, object?>();
    }
}
```

Jak dodać bazę danych?



Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

OffersDatabaseContext.cs

C# ./src/modules/JobFinder.Offers.csproj

Każdy moduł posiada swój własny **DbContext**, w którym udostępnia tylko dane potrzebne do realizowania logiki biznesowej modułu.

Ważne:

- DbContexty modułów nie są ze sobą w żaden sposób powiązane.
- Dla zachowania czystości kodu zamiast mapować wszystkie encje w metodzie **OnModelCreating(...)**, każda encja ma swój własny plik z konfiguracją mapowania.

JobFinder.Offers/OffersDatabaseContext.cs

```
namespace JobFinder.Offers.Infrastructure.Database;

public sealed class OffersDatabaseContext(DbContextOptions<OffersDatabaseContext> options)
 : DbContext(options)
{
    public DbSet<Offer> Offers { get; set; }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        builder.ApplyConfiguration(new OfferEntityTypeConfiguration());
    }
}
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

OfferEntityTypeConfiguration.cs

C# ./src/modules/JobFinder.Offers.csproj

W osobnej klasie **OfferEntityTypeConfiguration** opisujemy mapowanie encji **Offer** - podejście Code First. Konfiguracja może się różnić w zależności od użytej bazy danych. W przykładzie używana jest baza PostgreSQL.

Ważne:

- ➔ Aby możliwe było użycie konfiguracji w metodzie **ApplyConfiguration(...)** należy zaimplementować interfejs **IEntityTypeConfiguration<TEntity>**.

JobFinder.Offers/OfferEntityTypeConfiguration.cs

```cs
namespace JobFinder.Offers.Infrastructure.Database;

internal sealed class OfferEntityTypeConfiguration : IEntityTypeConfiguration<Offer>
{
    public void Configure(EntityTypeBuilder<Offer> builder)
    {
        builder.ToTable("offers");

        builder.HasKey(x => x.Id);
        builder.Property(x => x.Id)
            .HasColumnType("uuid")
            .HasConversion(
                id => id.Value,
                id => new OfferId(id)
            );

        builder.HasIndex(x => x.UserId);
        builder.Property(x => x.UserId)
            .HasColumnType("uuid")
            .HasConversion(
                id => id.Value,
                id => new UserId(id)
            )
            .IsRequired();

        builder.Property(x => x.Url)
            .HasColumnType("varchar(1000)")
            .IsRequired();

        // ...
    }
}
```

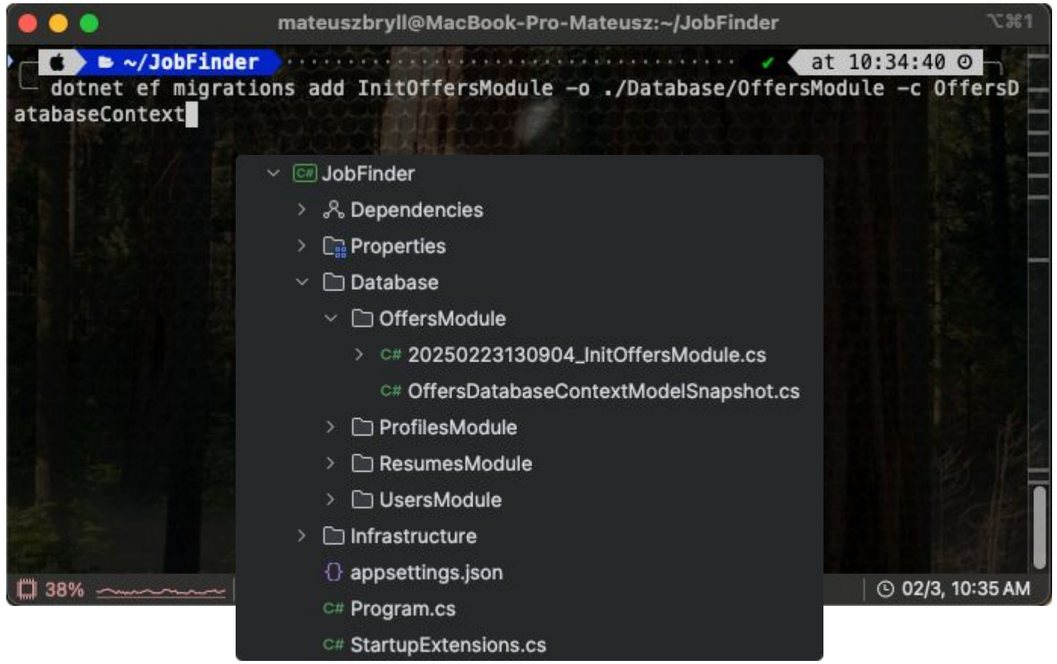
Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

Entity Framework - migracje

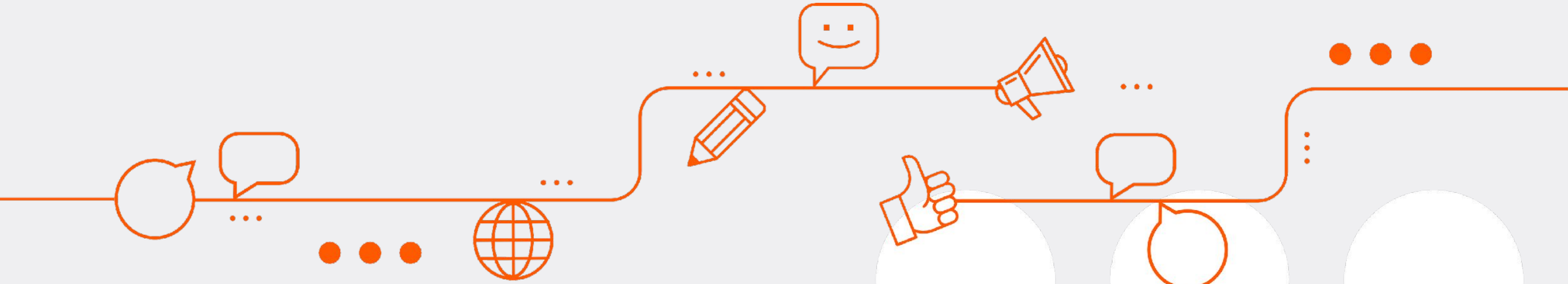
Projekt używa jednej instancji bazy **PostgreSQL**, ale każdy moduł definiuje własne, niezależne tabele. Dlatego w projekcie znaleźć można wiele implementacji **DbContextów**.

Główny projekt (**JobFinder.csproj**) jest odpowiedzialny za skonfigurowanie bazy danych oraz wykonanie migracji. Aby wygenerować kod migracji modułu w głównym projekcie należy:

- ➔ Utworzyć nową migrację komendą **dotnet ef migrations add** wskazując jej nazwę folder docelowy kodu migracji (parametr **-o**) oraz nazwę **DbContextu**, dla którego generujemy migrację (parametr **-c**).
- ➔ Aby wykonać kod migracji wystarczy użyć komendy **dotnet ef database update** wskazując odpowiedni **DbContext** (parametr **-c**).



Jak komunikować się między modułami?



Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

ProfilesClient.cs

C# ./src/modules/JobFinder.Profiles.Client.csproj

W przypadku modularnego monolitu klasa klienta wywołuje bezpośrednio publiczne API innego modułu. W przykładzie obok jest to komenda **GetUserSettings** z modułu **Profiles**.

Ważne:

- ➔ Moduł klienta udostępnia interfejs do komunikacji oraz model danych tzw. **DTOsy**.
- ➔ Należy bardzo uważać, aby nie używać wewnętrznych składowych modułu inaczej niż przez fasadę klienta.
- ➔ Jest to jawna implementacja **wzorca fasady**.

JobFinder.Profiles.Client/IProfilesClient.cs

```
namespace JobFinder.Profiles.Client;

public record OpenAiSettings(string ApiKey, string DefaultModel);

public interface IProfilesClient
{
    Task<OpenAiSettings> GetOpenAiSettingsAsync(UserId userId);
}

internal sealed class ProfilesClient(IQueryBus queryBus) : IProfilesClient
{
    public async Task<OpenAiSettings> GetOpenAiSettingsAsync(UserId userId)
    {
        var userSettings = await queryBus.SendAsync<GetUserSettings, Settings>(
            new GetUserSettings(userId));

        return new OpenAiSettings(userSettings.OpenAiApiKey, userSettings.DefaultModel);
    }
}
```

Jak tworzyć łatwe w utrzymaniu i rozwoju moduły?

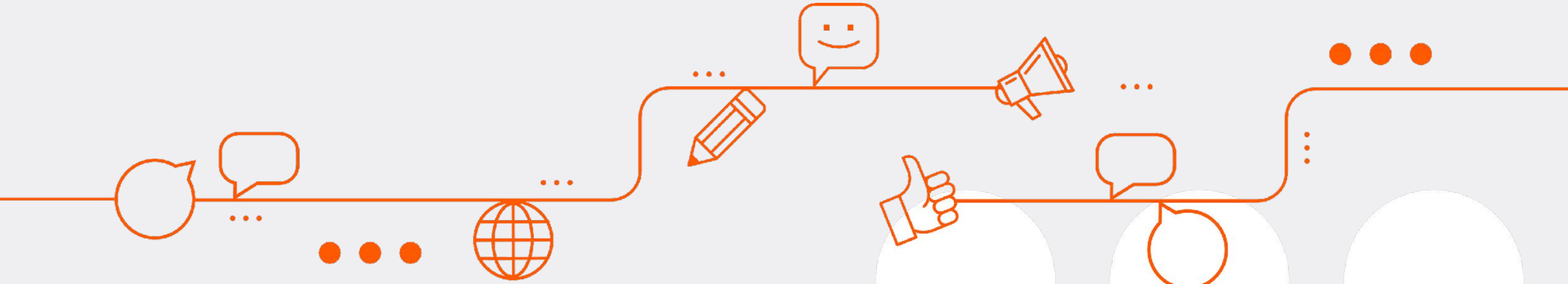
ServiceCollectionExtensions.cs

C# ./src/modules/JobFinder.Profiles.Client.csproj

Moduł klienta udostępnia również odpowiednią metodę pozwalającą zarejestrować moduł kliencki w kontenerze **DI**. W przykładzie obok klient modułu **Profiles** realizuje to w standardowy sposób poprzez udostępnienie metody rozszerzającej interfejs **IServiceCollection** **AddProfilesClient(...)**.

JobFinder.Profiles.Client/ServiceCollectionExtensions.cs

```cs
namespace JobFinder.Profiles.Client;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddProfilesClient(this IServiceCollection services)
    {
        services.AddScoped<IProfilesClient, ProfilesClient>();

        return services;
    }
}
```



Jak wydzielić moduł do mikroserwisu?

Jak w prosty sposób wydzielić moduł do osobnego mikroserwisu?

HttpProfilesClient.cs

C# ./src/modules/JobFinder.Profiles.Client.csproj

W przypadku mikroseriwsu klasa klienta wywołuje publiczne API innego modułu wykonując żądanie HTTP - synchroniczna integracja lub wysyła wiadomość na kolejkę - asynchroniczna integracja. W przykładzie obok klient wykonuje synchroniczne żądanie HTTP, które pod spodem wywołuje komendę **GetUserSettings** z modułu **Profiles**.

Oczywiście moduł **Profiles** musi udostępniać odpowiedni endpoint do integracji. Powinien być on też zabezpieczony - uwierzytelnienie i autoryzacja!

JobFinder.Profiles.Client/HttpProfilesClient.cs

```
namespace JobFinder.Profiles.Client;

internal sealed class HttpProfilesClient(HttpClient httpClient) : IProfilesClient
{
    public async Task<OpenAiSettings> GetOpenAiSettingsAsync(UserId userId)
    {
        var response = await httpClient.GetAsync($"/settings/{userId}");
        response.EnsureSuccessStatusCode();

        var settings = await response.Content.ReadFromJsonAsync<OpenAiSettings>();

        return settings!;
    }
}
```

Jak w prosty sposób wydzielić moduł do osobnego mikroserwisu?

ServiceCollectionExtensions.cs

C# ./src/modules/JobFinder.Profiles.Client.csproj

Zmianie ulega implementacja metody **AddProfilesClient(...)** udostępnianej przez klienta. Oprócz samej rejestracji implementacji klasy klienta w **DI** musimy dodatkowo zarejestrować odpowiednio skonfigurowanego **HttpClienta** do obsługi żądań do modułu **Profiles**.

Ważne:

- ➔ Zgodnie z dokumentacją klienta http rejestrujemy w **DI** metodą **AddHttpClient**.
- ➔ Dodatkowym wyzwaniem w świecie mikroserwisów jest obsługa chwilowych niedostępności innych serwisów.

JobFinder.Profiles.Client/ServiceCollectionExtensions.cs

```
namespace JobFinder.Profiles.Client;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddProfilesClient(this IServiceCollection services,
        IConfiguration configuration)
    {
        var baseAddress = configuration.GetSection("Profiles:BaseAddress").Get<Uri>();
        var apiKey = configuration.GetSection("Profiles:ApiKey").Get<string>();

        var retryPolicy = HttpPolicyExtensions
            .HandleTransientHttpError()
            .OrResult(response => response.StatusCode == HttpStatusCode.NotFound)
            .WaitAndRetryAsync(3, attempt => TimeSpan.FromSeconds(Math.Pow(2, attempt)));

        services.AddHttpClient<HttpProfilesClient>(client =>
        {
            client.BaseAddress = baseAddress;
            client.DefaultRequestHeaders.Add("Authorization", $"APIKEY {apiKey}");
        }).AddPolicyHandler(retryPolicy);

        services.AddScoped<IProfilesClient, HttpProfilesClient>();

        return services;
    }
}
```

Jak w prosty sposób wydzielić moduł do osobnego mikroserwisu?

Program.cs

C# ./src/modules/JobFinder.Profiles.csproj

Nowy mikroserwis w pliku **Program.cs** rejestruje tylko jeden moduł - **Profiles**.

JobFinder.Profiles/Program.cs

```
using JobFinder;
using JobFinder.Profiles;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);
builder.Services.AddApplicationInfrastructure(builder.Configuration);

var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
var migrationsAssembly = typeof(Program).Assembly.GetName().Name;

builder.Services.AddProfilesModule(options =>
    options.UseNpgsql(connectionString, x => x.MigrationsAssembly(migrationsAssembly)));

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

app.UseProfilesModule();

app.Run();
```

BONUS

Własna implementacja Bus'a na przykładzie CommandBus

W projekcie lepiej użyj np. MediatR ;)

allegro PAY

Jak tworzyć kod otwarty na rozszerzenia, a zamknięty na zmiany?

Commands.cs

C# ./deps/JobFinder.Common.csproj

Do implementacji wzorca (pośrednik) potrzebujemy kilku interfejsów:

- **ICommand** – “marker interface”, który musi implementować każda komenda.
- **ICommandHandler** – interfejs ułatwiający implementację wzorca factory method.
- **ICommandHandler<TCommand>** - interfejs, który musi implementować każdy handler.
- **ICommadBus** - interfejs, który musi implementować każda implementacja (synchroniczna, asynchroniczna, wewnątrzprocesowa) command busa.

JobFinder.Common/Commands.cs

```cs
namespace JobFinder.Common.Commands;

public interface ICommand { }

public interface ICommandHandler { }

public interface ICommandHandler<in TCommand> : ICommandHandler where TCommand : ICommand
{
    Task HandleAsync(TCommand command);
}

public interface ICommandBus
{
    Task SendAsync<TCommand>(TCommand command) where TCommand : ICommand;
}
```

Jak tworzyć kod otwarty na rozszerzenia, a zamknięty na zmiany?

InProcessCommandBus.cs

C# ./deps/JobFinder.Common.csproj

Obok przykładowa implementacja wewnątrz procesowej wersji **CommandBus**. Warto zwrócić uwagę na zależność w konstruktorze - wstrzykujemy tutaj **factory method** służące do utworzenia handlera komendy o danym typie.

W celu obsługi komendy implementacja musi:

- ➔ Utworzyć handler komendy używając **factory method**.
- ➔ W przypadku braku handlera dla danej komendy rzucany jest wyjątek.
- ➔ Jeśli udało się utworzyć handler to wywoływana jest jego metoda **HandleAsync(...)**.

JobFinder.Common/InProcessCommandBus.cs

```
namespace JobFinder.Infrastructure;

internal sealed class InProcessCommandBus(Func<Type, ICommandHandler> handlersFactory)
    : ICommandBus
{
    public Task SendAsync<TCommand>(TCommand command) where TCommand : ICommand
    {
        var handler = (ICommandHandler<TCommand>)handlersFactory(typeof(TCommand));
        if (handler is null)
        {
            throw new InvalidOperationException(
                $"Handler for {typeof(TCommand).Name} not found.");
        }

        return handler.HandleAsync(command);
    }
}
```

Jak tworzyć kod otwarty na rozszerzenia, a zamknięty na zmiany?

UpdatePersonalData.cs

C# ./src/modules/JobFinder.Profiles.csproj

Obok przykładowa implementacja komendy z modułu **Profiles**.

Komenda **UpdatePersonalData** jest zwykłym rekordem z parametrami komendy implementującym interfejs **ICommand**.

Poniżej implementacja handlera **UpdatePersonalDataHandler**, który posiada zależność do repozytorium i odpowiada za odpowiednią obsługę komendy.

JobFinder.Profiles/UpdatePersonalData.cs

```cs
namespace JobFinder.Profiles.Commands;

public sealed record UpdatePersonalData(
    UserId UserId, string FirstName, string LastName) : ICommand;

// ...

internal sealed class UpdatePersonalDataHandler(IProfilesRepository repository)
    : ICommandHandler<UpdatePersonalData>
{
    public async Task HandleAsync(UpdatePersonalData command)
    {
        var profile = await repository.GetProfileAsync(command.UserId);
        if (profile is null)
        {
            throw new ProfileNotFoundException(command.UserId);
        }

        profile = profile with
        {
            FirstName = command.FirstName,
            LastName = command.LastName
        };

        await repository.UpdateProfileAsync(profile);
    }
}
```

Jak tworzyć kod otwarty na rozszerzenia, a zamknięty na zmiany?

UpdatePersonalData.cs

C# ./src/modules/JobFinder.Profiles.csproj

Aby uprościć kod handlera odpowiedzialność za walidację danych podanych przez użytkownika w komendzie możemy oddelegować do dedykowanej klasy.

Obok przykładowa implementacja walidatora komendy **UpdatePersonalData** przy użyciu biblioteki **FluentValidation**.

JobFinder.Profiles/UpdatePersonalData.cs

```
using FluentValidation;

namespace JobFinder.Profiles.Commands;

// ...

internal sealed class UpdatePersonalDataValidator : AbstractValidator<UpdatePersonalData>
{
    public UpdatePersonalDataValidator()
    {
        RuleFor(x => x.UserId)
            .NotNull();

        RuleFor(x => x.FirstName)
            .NotEmpty()
            .MaximumLength(255);

        RuleFor(x => x.LastName)
            .NotEmpty()
            .MaximumLength(255);
    }
}

// ...
```

Jak tworzyć kod otwarty na rozszerzenia, a zamknięty na zmiany?

C..B..ValidationDecorator.cs

C# ./deps/JobFinder.Common.csproj

W poprzednim kroku zapewniliśmy, że kod handlera pozostanie czytelny poprzez oddelegowanie reguł walidacji komendy do osobnej klasy.

Aby obsłużyć walidację przy obsłudze komendy, a jednocześnie nie zaciemnić kodu **InProcessCommandBus**, walidowanie komend oddelegowujemy do innej klasy.

Warto zauważyć, że w ten sposób tworzymy **dekorator**, który ma tylko jedną odpowiedzialność i jest niezależny od konkretnej implementacji **ICommandBus**.

JobFinder.Common/CommandBusValidationDecorator.cs

```
namespace JobFinder.Infrastructure;

internal sealed class CommandBusValidationDecorator(
    Func<Type, IValidator> validatorsFactory, ICommandBus innerCommandBus) : ICommandBus
{
    public Task SendAsync<TCommand>(TCommand command) where TCommand : ICommand
    {
        var validator = (IValidator<TCommand>)validatorsFactory(typeof(TCommand));
        var result = validator.Validate(command);
        if (!result.IsValid)
        {
            throw new ValidationException(result);
        }

        return innerCommandBus.SendAsync(command);
    }
}
```

Jak tworzyć kod otwarty na rozszerzenia, a zamknięty na zmiany?

ServiceCollectionExtensions.cs

C# ./deps/JobFinder.Common.csproj

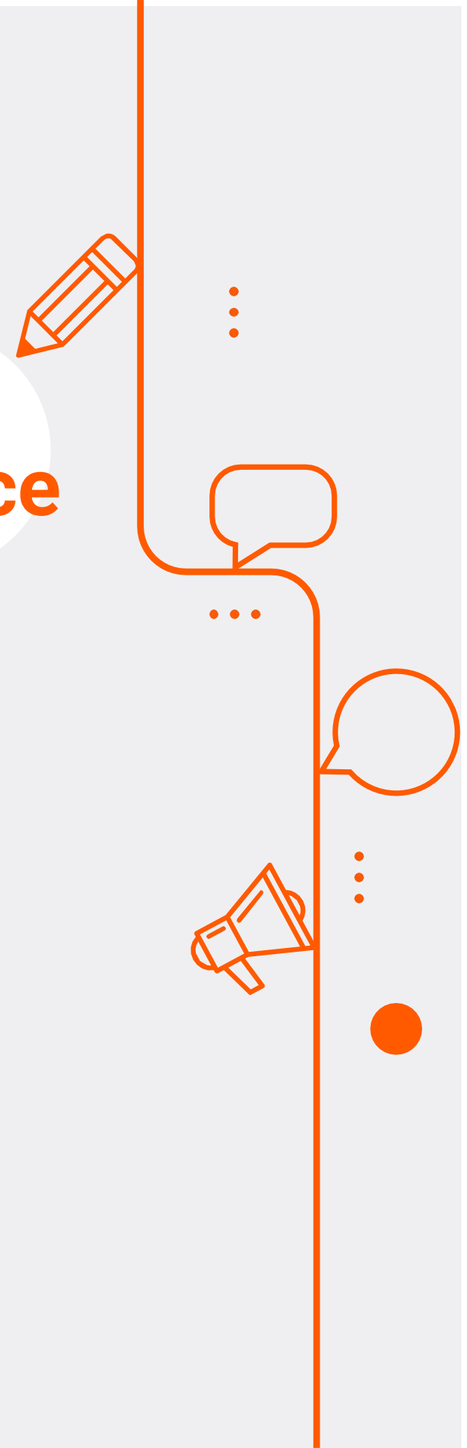
Na koniec pozostała rejestracja wszystkiego w **DI**. W tym celu w pierwszej kolejności dokonujemy rejestracji **InProcessCommandBus** jako **ICommandBus**. Następnie przy użyciu biblioteki **Scrutor** rejestrujemy **dekorator CommandBusValidationDecorator**.

Najtrudniejsze może wydawać się zarejestrowanie dwóch **factory methods**, które umożliwiają utworzenie handlera i walidatora komendy, ale cała rejestracja to tak naprawdę rejestrowanie **wyrażenia lambda**.

JobFinder.Common/ServiceCollectionExtensions.cs

```
namespace JobFinder.Infrastructure;

public static class ServiceCollectionExtensions
{
    public static IServiceCollection AddCommandBus(this IServiceCollection services)
    {
        services.AddScoped<ICommandBus, InProcessCommandBus>();
        services.Decorate<ICommandBus, CommandBusValidationDecorator>();

        services.AddScoped<Func<Type, ICommandHandler>>(provider =>
        {
            return tCommand =>
            {
                var handlerType = typeof(ICommandHandler<>).MakeGenericType(tCommand);
                return (ICommandHandler)provider.GetRequiredService(handlerType);
            };
        });

        services.AddScoped<Func<Type, IValidator>>(provider =>
        {
            return tCommand =>
            {
                var validatorType = typeof(IValidator<>).MakeGenericType(tCommand);
                return (IValidator)provider.GetRequiredService(validatorType);
            };
        });

        return services;
    }
}
```

Pułapki i antywzorce



1. **Zaciemnianie granic modułów**
   Dodawanie bezpośrednich zależności między modułami lub dostęp do wewnętrznych klas innego modułu omijając oficjalne API. Prowadzi to do plątaniny kodu trudnej w utrzymaniu i testowaniu.
2. **Przerost formy nad treścią**
   Zbyt ambitne wprowadzanie skomplikowanych wzorców i architektur (np. mikrousługi w projekcie studenckim o małej skali).
3. **Gigantyczne klasy i metody (antywzorzec “God Object”)**
   Umieszczanie zbyt wielu odpowiedzialności w jednym komponencie (naruszenie zasad SOLID, w szczególności Single Responsibility Principle). Powoduje to trudności w modyfikacji i debugowaniu kodu.
4. **Błędny podział domeny i logiki aplikacji**
   Zbytnia rozbieżność między kontekstem biznesowym a strukturą kodu (np. łączenie niepowiązanych funkcjonalności w jednym module), co utrudnia zrozumienie aplikacji i rozwój w przyszłości.
5. **Nieczytelne API modułów**
   Publiczne metody o niejasnych nazwach, zbyt rozbudowanych sygnaturach, bez jasnego kontraktu.
6. **Brak walidacji i obsługi błędów w jednym miejscu**
   Rozrzucone fragmenty walidacji po różnych warstwach, brak spójnej strategii obsługi wyjątków (np. standardu ProblemDetails).
7. **Powielanie logiki**
   Pisanie tej samej funkcjonalności w różnych miejscach zamiast wspólnego modułu lub biblioteki.

allegro

Rozwiń talent

z e–Xperience!

E–XPERIENCE

6 miesięcy stażu (lipiec-grudzień)

Umowa o pracę

Wsparcie buddy’ego

Rekrutacja trwa! Aplikuj na jobs.allegro.eu

Bądź na bieżąco, zaobserwuj nas na social media

INSTAGRAM

@allegrodobrzetubyc

FACEBOOK

@Allegro Tech
@Allegro Jobs

Dziękuję |

mateuszbryll.com/put2025

allegro PAY