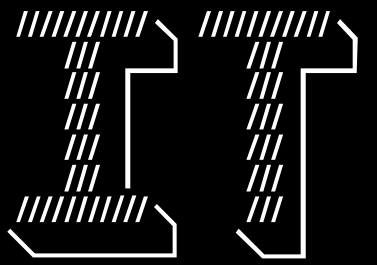
Dependency Pipeline

Andrei Sergeev, Pavel Moskovoy

Райффайзен



Спикер

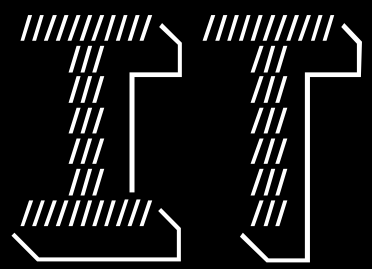
Andrei Sergeev

E-mail:
andrey.s.sergeev@raiffeisen.ru;
a.sergeev.official@gmail.com

Занимается проектированием и разработкой микросервисного back-end продуктовых платформ

В качестве основной платформы и языка разработки использует .NET и C#
Также ориентируется на практики из других языков и платформ (F#, Java, Kotlin, Ruby, Go, etc)
Применяет функциональный подход в разработке

Райффайзен



План

Райффайзен IT

- Что такое инверсия управления (Inversion of Control — IoC) и внедрение зависимостей (Dependency Injection — DI)?
- Типовой подход настройки DI-контейнера в .NET
- Чего хотим достичь?
- Dependency Pipeline
- Итоги

IoC & DI: что почитать

Martin Fowler

Inversion of Control Containers and the Dependency Injection pattern

https://martinfowler.com/articles/injection.html

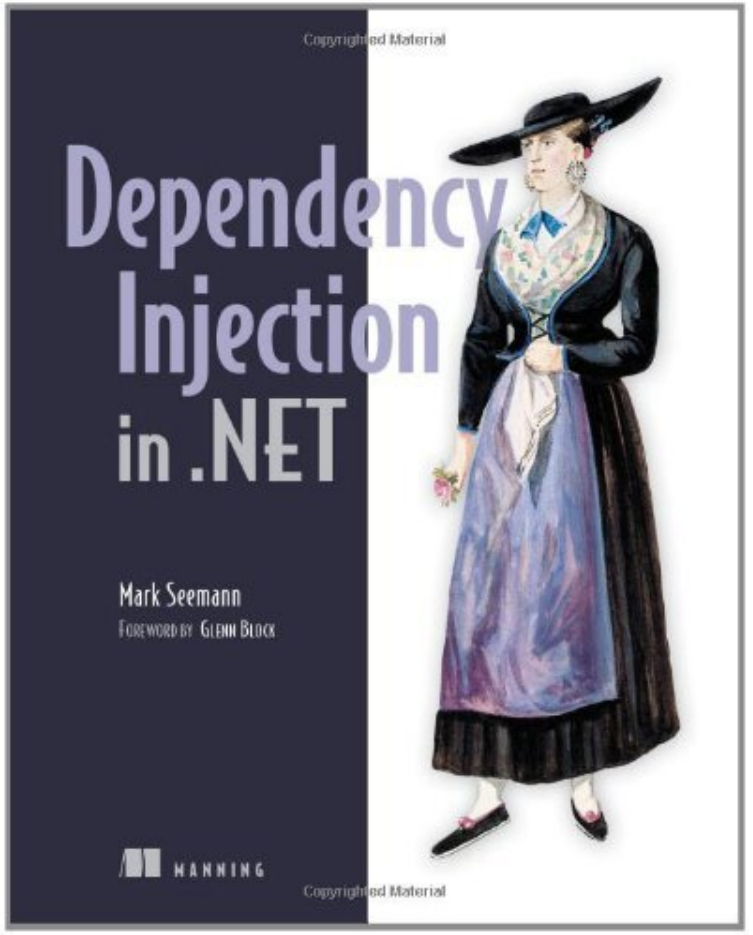
Dependency Injection Inversion

Posted by *Uncle Bob* on 01/17/2010

Dependency Injection is all the rage. There are several frameworks that will help yo
your system. Some use XML (God help us) to specify those dependencies. Others
code. In either case, the goal of these frameworks is to help you create instances v
or Factories.

Mark Seemann about 1 hour later:

Fortunately, I hardly consider this post controversial. In my book, DI is simply a set of patterns and principles that describe how we can write loosely coupled code. These include the GoF principle of programming to interfaces, as well as patterns such as Constructor Injection.



Inversion of Control - IoC

In software engineering, **inversion of control** (**IoC**) is a programming principle. IoC inverts the flow of control as compared to traditional control flow. In IoC, custom-written portions of a computer program receive the flow of control from a generic framework. A software architecture with this design inverts control as compared to traditional procedural programming: in traditional programming, the custom code that expresses the purpose of the program calls into reusable libraries to take care of generic tasks, but with inversion of control, it is the framework that calls into the custom, or task-specific, code.

Inversion of control is used to increase modularity of the program and make it extensible,[1] and has applications in object-oriented programming and other programming paradigms. The term was used by Michael Mattsson in a thesis,[2] taken from there[3] by Stefano Mazzocchi and popularized by him in 1999 in a defunct Apache Software Foundation project, Avalon, then further popularized in 2004 by Robert C. Martin and Martin Fowler.

From: https://en.wikipedia.org/wiki/Inversion_of_control

Dependency Injection - DI

In software engineering, **dependency injection** is a technique in which an object receives other objects that it depends on. These other objects are called dependencies. In the typical "using" relationship the receiving object is called a client and the passed (that is, "injected") object is called a service. The code that passes the service to the client can be many kinds of things[*definition needed*] and is called the injector. Instead of the client specifying which service it will use, the injector tells the client what service to use. The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it.

The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

The intent behind dependency injection is to achieve separation of concerns of construction and use of objects. This can increase readability and code reuse.

From: https://en.wikipedia.org/wiki/Inversion_of_control

Компонент с зависимостями

```
// Example

nullable enable

internal sealed class OrderLogic : IOrderLogic
{
    private readonly IOrderDatabase database;

    public OrderLogic(in IOrderDatabase database)
    {
        this.database = database;
    }

    public async Task<Result<OrderGuid, Error<OrderLogicErrorCode>>> CreateOrder(
        Order order)
    {
        await database.DbContext.Database.BeginTransactionAsync().ConfigureAwait(false);
```

Компонент с зависимостями

Райффайзен

```
// В чем отличие такого компонента от обычного?
```

Компонент с зависимостями

Райффайзен

```
// В чем отличие такого компонента от обычного?
// 1. Создается в коде неявным образом
```

Компонент с зависимостями

Райффайзен

```
// В чем отличие такого компонента от обычного?
// 1. Создается в коде неявным образом
// 2. Зависимости предоставляются также неявно
```

Пример настройки DI-контейнера

```
nullable enable

public void ConfigureServices(IServiceCollection services)
{
    var connection = Configuration.GetConnectionString("OrderConnection");
    services.AddDbContext<OrderContext>(options => options.UseSqlServer(connection));

    services.AddControllers(options => options.EnableEndpointRouting = false);

    services
        .AddSingleton<IOrderProcessingConfig>(Configuration.Get<OrderProcessingConfig>())
        .AddSingleton(Configuration.Get<PlacingServiceClientConfiguration>())
        .AddTransient<IOrderDatabase, OrderContext>()
        .AddSingleton<IPlacingServiceClient, PlacingServiceClient>()
        .AddSingleton<IDateService, DateService>()
        .AddSingleton<IPlacingService, PlacingService>()
        .AddSingleton<IServiceProviderDecorator, ServiceProviderDecorator>()
        .AddHostedService<OrderProcessingService>();
}
```

Пример настройки DI-контейнера


```
// Что не так с этим кодом?
```

Пример настройки DI-контейнера

```
// Что не так с этим кодом?
// 1. Какой компонент какие зависимости использует?
```

Пример настройки DI-контейнера

Райффайзен

```
// Что не так с этим кодом?
// 1. Какой компонент какие зависимости использует?
// 2. Есть ли неиспользуемые зависимости?
```

Пример настройки DI-контейнера

```
// Что не так с этим кодом?
// 1. Какой компонент какие зависимости использует?
// 2. Есть ли неиспользуемые зависимости?
// 3. Возможно, не самый читаемый синтаксис:
Add(Scope)<TDependency>?
```

Пример настройки DI-контейнера

```
        // Что не так с этим кодом?
        // 1. Какой компонент какие зависимости использует?
        // 2. Есть ли неиспользуемые зависимости?
        // 3. Возможно, не самый читаемый синтаксис:
Add(Scope)<TDependency>)?
        // 4. Что еще? К каким ошибкам это может привести?
Есть ли конкретные ошибки в этом коде?
```

Пример настройки DI-контейнера


```
// Что не так с этим кодом?
// 1. Какой компонент какие зависимости использует?
// 2. Есть ли неиспользуемые зависимости?
// 3. Возможно, не самый читаемый синтаксис: Add(Scope)<TDependency>?
// 4. Что еще? К каким ошибкам это может привести? Есть ли конкретные ошибки в этом коде?
```

// А как делаете вы?

Пример настройки DI-контейнера


```
// Что не так с этим кодом?
// 1. Какой компонент какие зависимости использует?
// 2. Есть ли неиспользуемые зависимости?
// 3. Возможно, не самый читаемый синтаксис (Add***)?
// 4. Что еще? К каким ошибкам это может привести? Есть ли конкретные ошибки в этом коде?
```

```
// А как делаете вы?
```

```
// Расскажу, как сделано у нас ==>
```

Что хотим сделать?

```
// 1. Получить «прозрачное» дерево зависимостей
```

Что хотим сделать?

Райффайзен IT

```
// 1. Получить «прозрачное» дерево зависимостей

// 2. Разделить факт регистрации зависимости и определение скоупа регистрации (separation of concerns)
```

Что хотим сделать?

Райффайзен

```
// 1. Получить «прозрачное» дерево зависимостей

// 2. Разделить факт регистрации зависимости и определение скоупа
регистрации (separation of concerns)

// 3. Предоставлять зависимости одного типа с разной конфигурацией
(HttpClient) общим образом без использования workaround TCategory
```

Что хотим сделать?

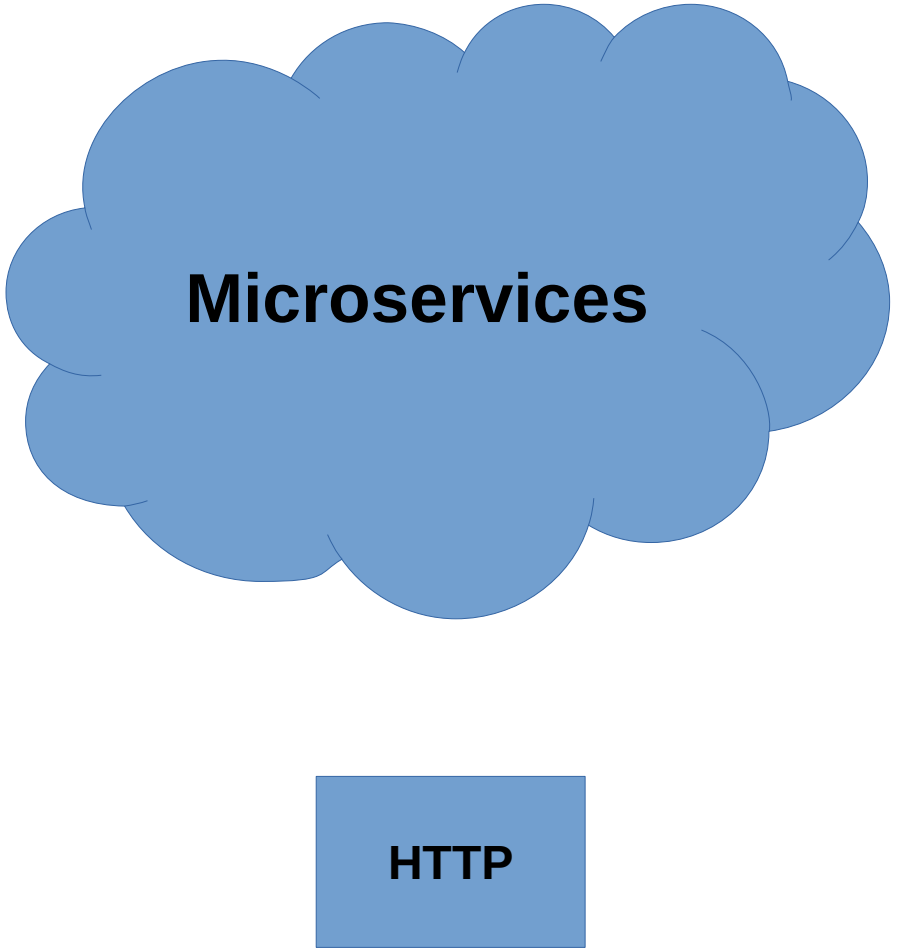
Райффайзен

```
// 1. Получить «прозрачное» дерево зависимостей

// 2. Разделить факт регистрации зависимости и определение скоупа
регистрации (separation of concerns)

// 3. Предоставлять зависимости одного типа с разной конфигурацией
(HttpClient) общим образом без использования workaround TCategory

// 4. Минимизировать число компонентов в инверсии управления, полностью
сохранив внедрение зависимостей
```

Dependency Pipeline


```
// Реализует внедрение зависимостей в pipeline-стиле и
отделяет внедрение зависимостей от инверсии управления
```

Создадим приложение?



Расширения уровня REST HTTP Engine

```
#nullable enable

public static class RestSenderAbstractFactoryExtensions
{
    public static IServiceBuilder<IRestSenderAbstractFactory> UseRest<TMessageInvoker>(
        this IServiceBuilder<TMessageInvoker> messageInvokerBuilder)
        where TMessageInvoker : HttpMessageInvoker
        =>
        messageInvokerBuilder.Map<IRestSenderAbstractFactory>(
            httpMessageInvoker => new RestSenderAbstractFactory(httpMessageInvoker));

    /// <summary>
    /// See "Problem Details for HTTP APIs" https://tools.ietf.org/rfc/rfc7807.txt
    /// </summary>
    public static IServiceBuilder<IRestProblemDetailsAbstractFactory> UseRestRfc7087<TMessageInvoker>(
        this IServiceBuilder<TMessageInvoker> messageInvokerBuilder)
        where TMessageInvoker : HttpMessageInvoker
        =>
        messageInvokerBuilder.Map<IRestProblemDetails>(
                httpMessageInvoker => new RestProblemDetailsAbstractFactory(httpMessageInvoker));
    }
```

Создадим приложение?

Расширения уровня Data

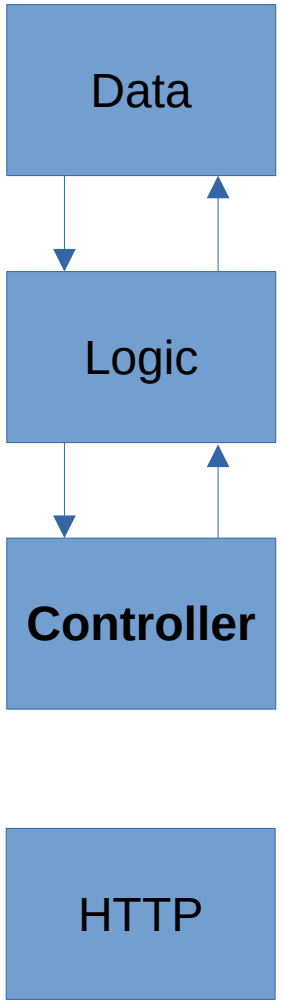
```
#nullable enable

public static class OrderDatabaseServiceBuilderExtensions
{
     public static IServiceBuilder<IOrderDatabase> UseOrderDatabase(
         this IServiceCollection services,
         in DbContextOptions dbContextOptions)
     {
         _ = dbContextOptions
             ?? throw new ArgumentNullException(nameof(dbContextOptions));

         var theDbContextOptions = dbContextOptions;

         return services.UseBuilder<IOrderDatabase>(
             _ => new OrderDbContext(theDbContextOptions));
     }
}
```

Создадим приложение?

Расширения уровня Logic

```
#nullable enable

public static class OrderLogicServiceBuilderExtensions
{
    public static IServiceBuilder<IOrderLogic> ToOrderLogic(
        this IServiceBuilder<IOrderDatabase> databaseBuilder)
        =>
        databaseBuilder.Map<IOrderLogic>(
            database => database switch
            {
                null => throw new InvalidOperationException(
                    "Database must be not null."),

                _ => new OrderLogic(database)
            });
}
```

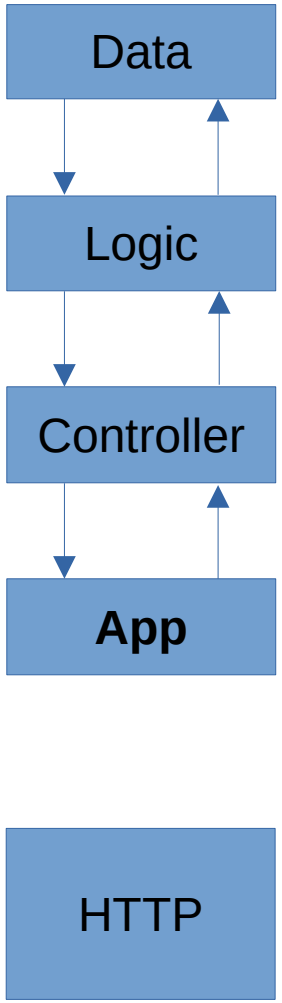
Создадим приложение?



Расширения уровня Infrastructure


```
#nullable enable

public static class OrderInfraServiceBuilderExtensions
{
    public static IServiceBuilder<OrderController> CreateOrderController(
        this IServiceBuilder<IOrderLogic> logicBuilder)
        =>
        logicBuilder.Map(
            orderLogic => orderLogic switch
            {
                null => throw new InvalidOperationException(
                    "Order logic must be not null."),

                _ => new OrderController(orderLogic)
            });
}
```

Создадим приложение?



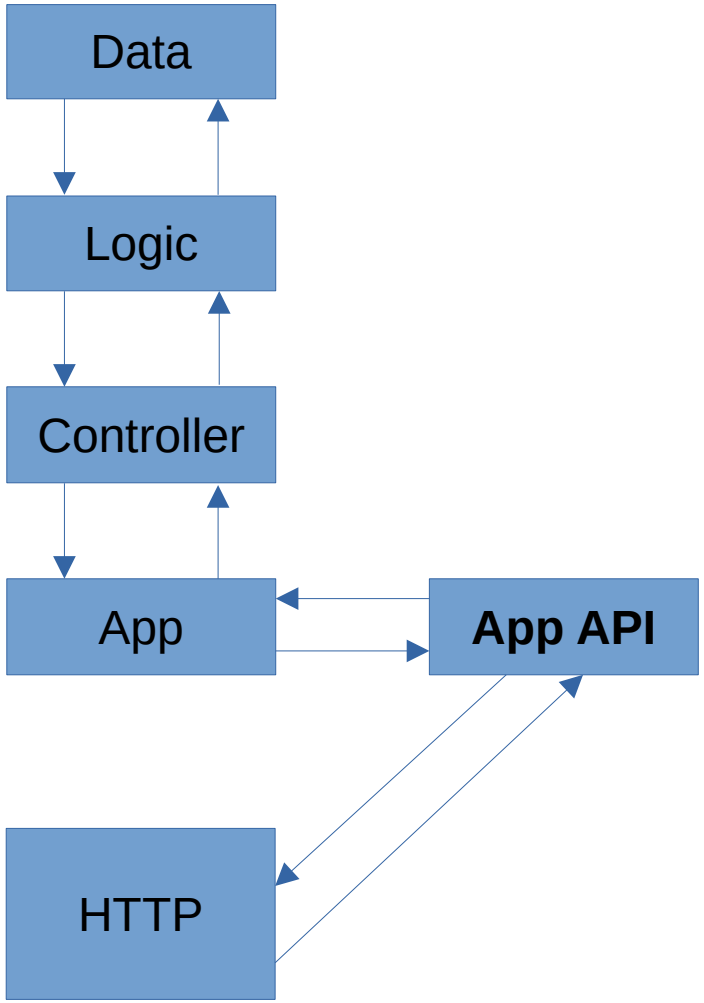
Расширения уровня Application


```
#nullable enable

internal static class DependencyExtensions
{
    public static IServiceBuilder<IControllerActivator> UseControllerActivator(
        this IServiceCollection services, IConfiguration configuration)
        =>
        services
        .UseOrderDatabase(configuration)
        .ToOrderLogic()
        .CreateOrderController()
        .UseControllerActivator();

    private static IServiceBuilder<IOrderDatabase> UseOrderDatabase(
        this IServiceCollection services, in IConfiguration configuration)
        =>
        services
        .UseOrderDatabase(configuration.CreateDbOptions("DbConnection"))
        .Do(b => b.RegisterAsTransient());
```

Внедрение в Application

```
#nullable enable

internal sealed class Startup
{
    public Startup(IWebHostEnvironment env)
        =>
        Configuration = BuildConfiguration(env.ContentRootPath);

    private IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
        =>
        services
        .AddControllers().Services
        .UseControllerActivator(Configuration).RegisterAsSingleton()
        .AddSwagger(Configuration)
        .AddHealthChecks();
```

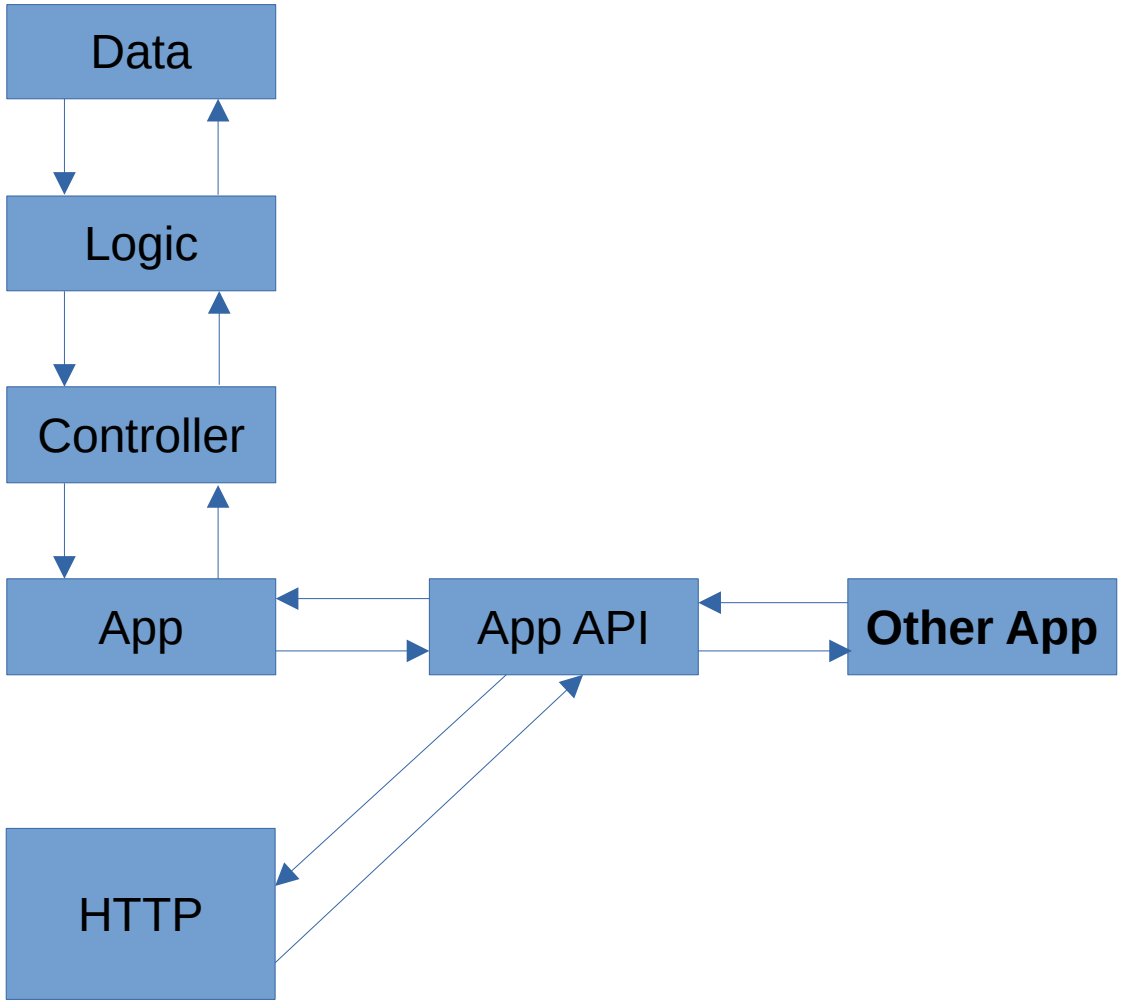
Создадим приложение?



Расширения уровня Application API


```
#nullable enable

public static class OrderApiDependencyExtensions
{
    public static IServiceBuilder<IOrderApi> CreateOrderApi(
        this IServiceBuilder<IRestSenderAbstractFactory> abstractFactoryBuilder)
        =>
        abstractFactoryBuilder.Map(abstractFactory =>
            CreateOrderApi(abstractFactory ?? throw new InvalidOperationException(
                "REST Sender Abstract Factory must be not null.")));

    private static IOrderApi CreateOrderApi(
        in IRestProblemDetailsAbstractFactory abstractFactory)
        =>
        new OrderApi(
            abstractFactory.CreateFactory<OrderJson, OrderGuidJson>().CreatePostSender(),
            abstractFactory.CreateFactory<OrderJson>().CreateGetSender(),
            orderMapper.Default);
}
```

Создадим приложение?



Расширения для подключения API приложения в другом приложении

```
#nullable enable

public static IServiceBuilder<IControllerActivator> UseControllerActivator(
    this IServiceCollection services,
    in IConfiguration configuration)
    => services
    .UseTuple(
        first: services
        .UseHttpClient(configuration.GetSection("SomeApiClient")
            .Get<SomeHttpClientConfiguration>())
            .AddStandardRequestHeadersHandler()
            .UseRestRfc7087().UseSomeServiceApi().Resolve,

        second: services
        .UseHttpClient(configuration.GetSection("AnotherApiClient")
            .Get<AhotherHttpClientConfiguration>())
            .AddStandardRequestHeadersHandler()
            .UseRestRfc7087().CreateAnotherServiceApi().Resolve,

        third: services.UseMbbPolicyDatabase(configuration).Resolve)
    .CreateTheService()
```

Dependency Pipeline

Райффайзен

Implementation

Реализация (Public)

```
#nullable enable

public interface IServiceBuilder<TService> :
    IBuildSupplier<TService>, IServiceCollectionProvider where TService : class
{
    public IServiceCollection RegisterAsTransient() => Services.AddTransient(Build);
    public IServiceCollection RegisterAsScoped() => Services.AddScoped(Build);
    public IServiceCollection RegisterAsSingleton() => Services.AddSingleton(Build);

    public IServiceBuilder<TResultService> Map<TResultService>(
        Func<TService, TResultService> map) where TResultService : class
        =>
        Services.UseBuilder(sp => map.Invoke(Resolve(sp)));

    public IServiceBuilder<TResultService> Pipe<TResultService>(
        Func<IServiceProvider, TService, TResultService> map) where TResultService : class
        =>
        Services.UseBuilder(sp => map.Invoke(sp, Resolve(sp)));

    public ITupleBuilder<TService, TSecondService> And<TSecondService>(
        in IServiceBuilder<TSecondService> second) where TSecondService : class
        =>
        Services.UseTuple(first: this, second: second);
```

Реализация (Internal)

```
#nullable enable

internal sealed class DefaultServiceBuilder<TService> : IServiceBuilder<TService>
    where TService : class
{
    private readonly IServiceCollection services;
    private readonly Resolver<TService> serviceResolver;

    public DefaultServiceBuilder(in IServiceCollection services,
        in Resolver<TService> serviceResolver)
    {
        this.services = services;
        this.serviceResolver = serviceResolver;
    }

    IServiceCollection IServiceCollectionProvider.Services => services;

    TService IBuildSupplier<TService>.Build(IServiceProvider serviceProvider)
        => serviceResolver.Invoke(serviceProvider);
}
```

Реализация (Расширения)

```
#nullable enable

public static class ServiceBuilderExtensions
{
    public static IServiceBuilder<TService> UseBuilder<TService>(
        this IServiceCollection services,
        in Resolver<TService> serviceResolver) where TService : class
        =>
        new DefaultServiceBuilder<TService>(services, serviceResolver);

    public static ITupleBuilder<TFirst, TSecond> UseTuple<TFirst, TSecond>(
        this IServiceCollection services,
        in Resolver<TFirst> first,
        in Resolver<TSecond> second) where TFirst : class where TSecond : class
        =>
        services.UseTuple(services.UseBuilder(first), services.UseBuilder(second));

    public static ITupleBuilder<TFirst, TSecond> UseTuple<TFirst, TSecond>(
        this IServiceCollection services,
        in IServiceBuilder<TFirst> first,
        in IServiceBuilder<TSecond> second)
```

Итоги — что получили

// 1. Дерево зависимостей задается в явном виде «прозрачным» образом: для каждого компонента всегда создаются нужные зависимости, а ненужные зависимости не создаются

Итоги — что получили


```
// 1. Дерево зависимостей задается в явном виде «прозрачным» образом: для каждого компонента всегда создаются нужные зависимости, а ненужные зависимости не создаются

// 2. Разделены факт регистрации зависимости и определение скоупа регистрации (separation of concerns)
```

Итоги — что получили


```
// 1. Дерево зависимостей задается в явном виде «прозрачным» образом: для каждого компонента всегда создаются нужные зависимости, а ненужные зависимости не создаются

// 2. Разделены факт регистрации зависимости и определение скоупа регистрации (separation of concerns)

// 3. Поддерживаются зависимости одного типа с разной конфигурацией (HttpClient) без использования workaround TCategory
```

Итоги — что получили

Райффайзен

// 1. Дерево зависимостей задается в явном виде «прозрачным» образом: для каждого компонента всегда создаются нужные зависимости, а ненужные зависимости не создаются

// 2. Разделены факт регистрации зависимости и определение скоупа регистрации (separation of concerns)

// 3. Поддерживаются зависимости одного типа с разной конфигурацией (HttpClient) без использования workaround TCategory

// ***4. Внедрение зависимостей отделено от инверсии управления***

Точки касания инверсии управления и внедрения зависимостей: точка старта приложения (IControllerActivator), Singleton- и Scoped-зависимости, а также IDisposable для случаев, когда такие зависимости должны освобождаться платформой (DbContext)

Итоги — главный вывод

Райффайзен IT

Единый «движок» для реализации IoC и DI — нарушение принципа Separation of Concerns (SoC)

Итоги — что предлагаем

- ***Использовать разные механизмы для реализации в приложениях инверсии управления и внедрения зависимостей***

  Механизм IoC в популярных фреймворках, используемый для старта приложения должен использоваться по назначению.

  Для остальной части дерева зависимостей должен задействоваться отдельный, механизм (Dependency Pipeline или другой)

Итоги — что предлагаем

Райффайзен

- Использовать разные механизмы для реализации в приложениях инверсии управления и внедрения зависимостей

  …

- **А также:**

  реализовывать отдельные деревья зависимостей на каждом из уровней в библиотеках и приложениях;

  на каждом следующем уровне конструировать дерево зависимостей из предыдущих уровней

Спасибо за внимание

Райффайзен