Внедрение зависимостей
Dependency Injection (DI)

Вопросы по ДЗ

Внедрение зависимостей

- Это паттерн проектирования
- **Передает классу его зависимости через конструктор, метод или свойство**
- Делает код более гибким
- Делает код тестируемым

Что не так с кодом?

```
public class RegistrationService
{
    public void RegisterUser()
    {
        /*
         * Логика регистрации...
         */
        var emailSender = new ASESEmailSender();
        emailSender.Send("Вы успешно зарегистрированы");
    }
}
```

“ASES” - Amazon Simple Email Service

Что не так с кодом?

```
public class RegistrationService
{
    private readonly ASESEmailSender _emailSender;

    public RegistrationService(ASESEmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    public void Register()
    {
        // ...
        _emailSender.Send("Вы успешно зарегистрированы");
    }
}
```

“ASES” - Amazon Simple Email Service

Что не так с кодом?

```
public class RegistrationService
{
    private readonly IEmailSender _emailSender;

    public RegistrationService(IEmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    public void Register()
    {
        // ...
        _emailSender.Send("Вы успешно зарегистрированы");
    }
}
```

SOLID

DIP – **D**ependency **I**nversion **P**rinciple

DIP – Принцип инверсии зависимости

1. *Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

   В общем, все должны зависеть от абстракций.

2. *…*

DIP – Принцип инверсии зависимости

- Абстракция – интерфейс или абстрактный класс, предоставляющий контракт, который обязаны соблюдать всего его реализаторы
- Деталь – конкретный класс, поведение которого невозможно изменить

Абстракция и ее реализации

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

```
public class ASESEmailSender : IEmailSender
{
    public void Send(string message)
    {
        // тут логика отправки письма
    }
}

public class MailgunEmailSender : IEmailSender
{
    public void Send(string message)
    {
        // тут логика отправки письма
    }
}
```

Заглушка для теста

```
public class EmailSenderStub : IEmailSender
{
    private readonly ILogger<EmailSenderStub> _logger;

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

    public void Send(string message)
    {
        _logger.LogDebug("Отправка сообщения вызвана, но пропущена");
    }
}
```

Слабосвязанный код

Использование интерфейсов приводит к слабосвязанному (loosely coupled) дизайну, поскольку класс `RegistrationService` знает лишь об интерфейсе `IEmailSender` и не знает о конкретной реализации этого интерфейса.

DI-контейнер

Представляет собой программную библиотеку, реализующую:

1. регистрацию зависимостей
2. их разрешение
3. и управление их временем жизни

DI-контейнеры. Что было раньше?

Старый ASP.NET:

- Autofac
- Ninject
- Lamar
- Castle

Microsoft.Extensions.DependencyInjection

- Новый DI-контейнер, встроенный в ASP.NET Core
- Еще его называются “коробочный DI”
- Ныне стандарт в мире ASP.NET Core

Время жизни зависимости

Dependency Lifetime

Время жизни: Singleton

- Живет пока жив DI-контейнер
- В ASP.NET Core время жизни Singletone зависимости соответствует времени жизни веб-приложения
- Применим **только** с потокобезопасными классами
- Примеры
  - `HttpClient`
  - `MemoryCache`
  - клиент MongoDb

Вспомним паттерн проектирования Singleton

Singleton (одиночка) – это паттерн проектирования, который делает две вещи:

- Дает гарантию, что у класса будет всего один экземпляр класса.
- Предоставляет глобальную точку доступа к экземпляру данного класса.

Простой пример реализации паттерна Singleton

```
public class MyRandom
{
    public static MyRandom Instance { get; } = new MyRandom();

    public int Next() => Random.Shared.Next();
}
```

Использование:

```
int rnd = MyRandom.Instance.Next();
```

Время жизни Singletone

$\neq$

Паттерн проектирования Singleton

Singleton: Противопоказания

- Когда у компонента отсутствует потокобезопасность. Поскольку экземпляр Singleton потенциально совместно используется сразу несколькими потребителями, он должен быть способен обрабатывать параллельный доступ.
- Когда время жизни одной из зависимостей компонента ожидаемо короче времени жизни остальных зависимостей, возможно, из-за того, что она не обладает потокобезопасностью.

Регистрация Singleton зависимости в веб-приложении

```
var builder = WebApplication.CreateBuilder(args);
                                //Абстракция    //Реализация
builder.Services.AddSingleton<IEmailSender, ASESEmailSender>();
```

Разрешение Singleton зависимости

```
var emailSender = app.Services.GetService<IEmailSender>();
```

Какой тип будет у emailSender?

Разрешение Singleton зависимости

Или просто:

```
public class RegistrationService
{
    public RegistrationService(IEmailSender emailSender)
    {
        /// ...
    }
}
```

Разрешение зависимости в Minimal API

```
builder.Services.AddSingleton<RegistrationService>();

//...

app.MapGet("/register", (RegistrationService service) => service.RegisterUser());
```

Время жизни: Scoped

- Создается новый экземпляр на каждый запрос (скоуп)
- Время жизни соответствует времени жизни запроса (скоупа)
- Подходит для потоконебезопасных классов
- Примеры
  - `DbContext` (Entity Framework Core)
  - Класс `Random`

Регистрация Scoped зависимости в веб-приложении

```
var builder = WebApplication.CreateBuilder(args);
…

                          /* Абстракция */ /* Реализация */
builder.Services.AddScoped<IEmailSender,  ASESEmailSender>();
```

Регистрация Scoped зависимости в веб-приложении

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

                          //Реализация
builder.Services.AddScoped<Random>();
//Так тоже можно, но нарушает DIP
```

Разрешение Scoped зависимости

```
using (var scope = app.Services.CreateScope())
{
    var services = scope.ServiceProvider;
    var sender = services.GetRequiredService<IEmailSender>();
    // ...
}
```

Разрешение Scoped зависимости

Или просто:

```
public class RegistrationService
{
    public RegistrationService(IEmailSender emailSender)
    {
        /// ...
    }
}
```

Разрешение Scoped зависимости

Или просто:

```
public class RegistrationService
{
    public RegistrationService(IEmailSender emailSender, Random random)
    {
        /// ...
    }
}
```

Scoped: Недостатки

Может выделять неоправданно много ресурсов, т. к. на каждый запрос создается новый экземпляр зависимости.

Singleton – антипаттерн?

Время жизни: Transient

- Новый экземпляр создается при **каждом** разрешении зависимости
- Обычно безболезненно может быть заменен на `Scoped`
- Адекватный кейс применения найти не удалось

Подробнее про Transient: https://bit.ly/3HEpn4E

Анти-паттерн: Service Locator (локатор сервисов)

- Класс скрывает очевидность своих зависимостей
  - Это затрудняет тестирование
  - Лишает проверки корректности зависимостей при старте веб-приложения (для сервисов)
- В ASP.NET Core Service Locator это
  - **`IServiceProvider`**
  - Также Service Locator можно создать, вызвав `IServiceScopeFactory.CreateScope`
  - Про разницу между IServiceProvider и IServiceScopeFactory читайте здесь

Анти-паттерн: Service Locator (локатор сервисов)

```
public class RegistrationService
{
    private readonly IEmailSender _emailSender;
    private readonly Random _random;

    public RegistrationService(IServiceProvider serviceProvider)
    {
        _emailSender = serviceProvider.GetRequiredService<IEmailSender>();
        _random = serviceProvider.GetRequiredService<Random>();
    }

    // ...
}
```

Ликвидация зависимости

- Если ваш класс использует неуправляемые ресурсы, то просто реализуйте интерфейс `IDisposable` и реализуйте высвобождение захваченных ресурсов в методе `Dispose()`.
- Например, метод `Dispose()` в `DbContext` освобождает контекст, возвращая его в пул контекстов.

Оборотная сторона DIP

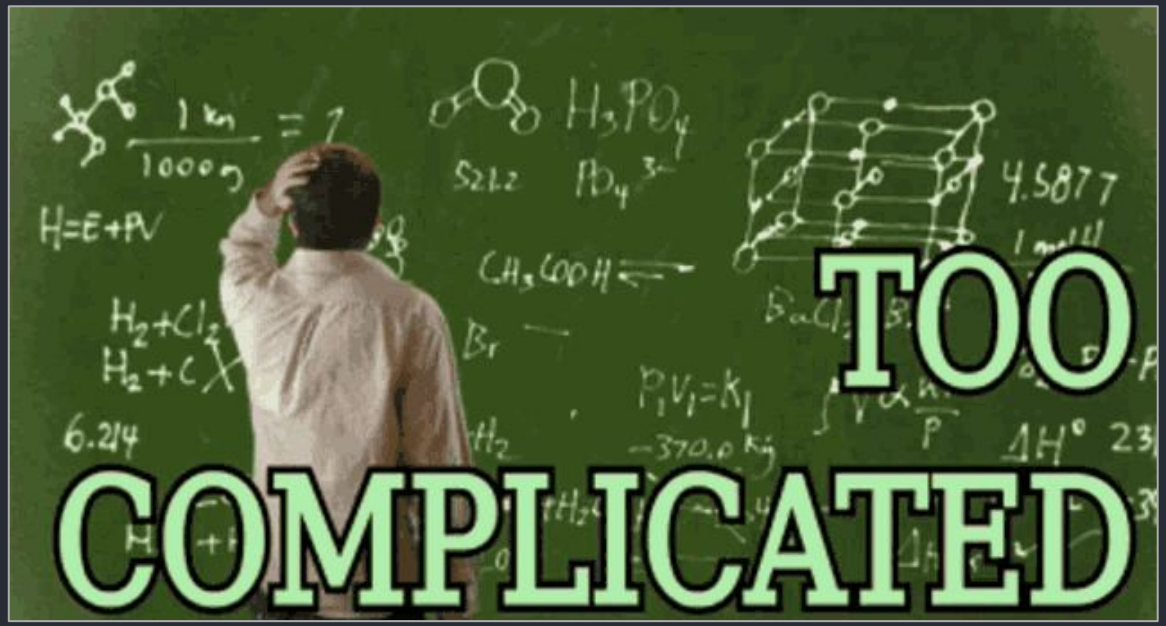
- Наличие интерфейсов образует дополнительный уровень абстракции, что затрудняет понимание системы. Понять логику исполнения становится довольно сложно.
- Было бы странно `List<T>` появилась бы зависимость вида `IListGrowingPolicy`, которая отвечала бы за способ роста списка! Подобные решения обеспечивают гибкость не там, где нужно, и могут подрывать инкапсуляцию.

Anti-DIP

Принцип инверсии сознания, или DI головного мозга. Интерфейсы выделяются для каждого класса и пачками передаются через конструкторы.

Anti-DIP

Понять, где находится логика, становится практически невозможно.



Где грань DIP?

- На этапе проектирования интерфейсы могут сильно усложнять процесс проектирования
- Для доменных сервисов, вероятно, абстракции не нужны
- Думаем и решаем грозит ли сервису изменение
- В стартапах

DEMO

Scoped vs Transient

Несовместимость Lifetime

- В Singleton-зависимость нельзя передать Scoped-зависимость, т. к. Singleton существуют вне скоупа.
- Зато в Singleton-зависимость можно передать Transient-зависимость – просто будет создан уникальный экземпляр Transient-зависимости, т. к. Transient-зависимостям не нужен скоуп.
  - Но лучше так не делать, т. к. никто не гарантирует, что Transient зависимость будет потокобезопасна, поэтому придется самостоятельно заботиться об этом

Несовместимость Lifetime

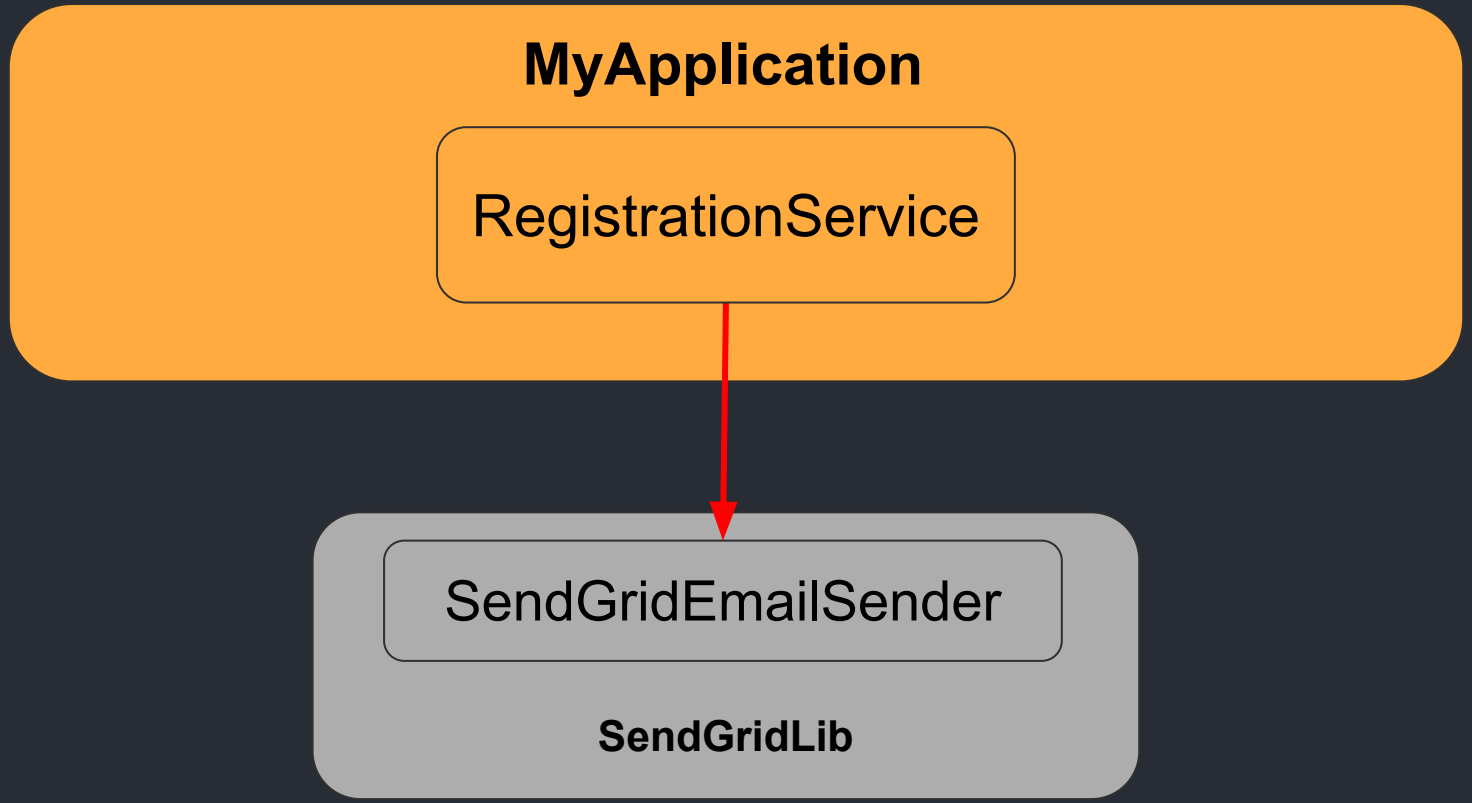
DEMO

DIP. Advanced

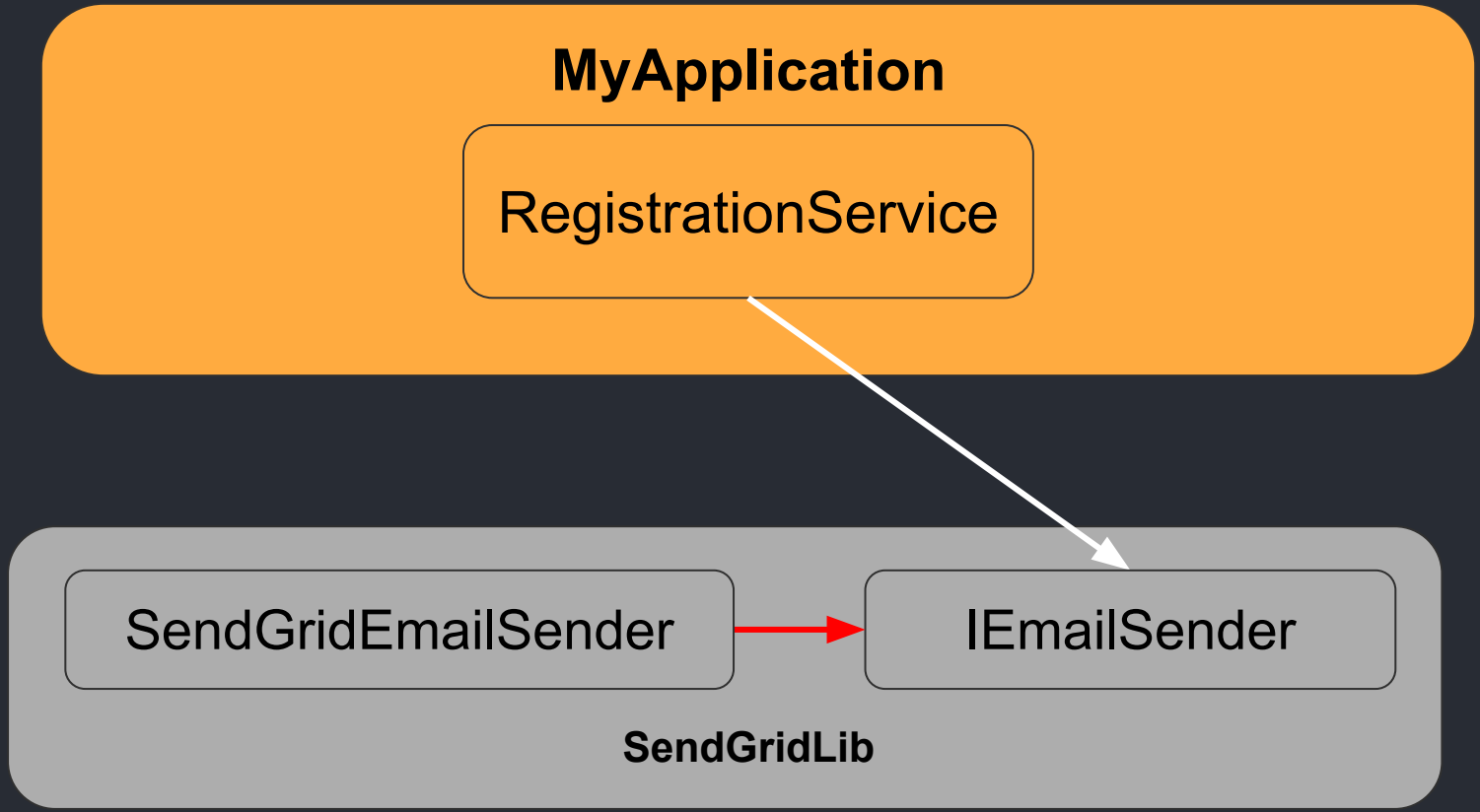
DIP – Принцип инверсии зависимости

- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.
- **Модули верхнего уровня не должны зависеть от модулей нижнего уровня.** И те и другие должны зависеть от абстракций.

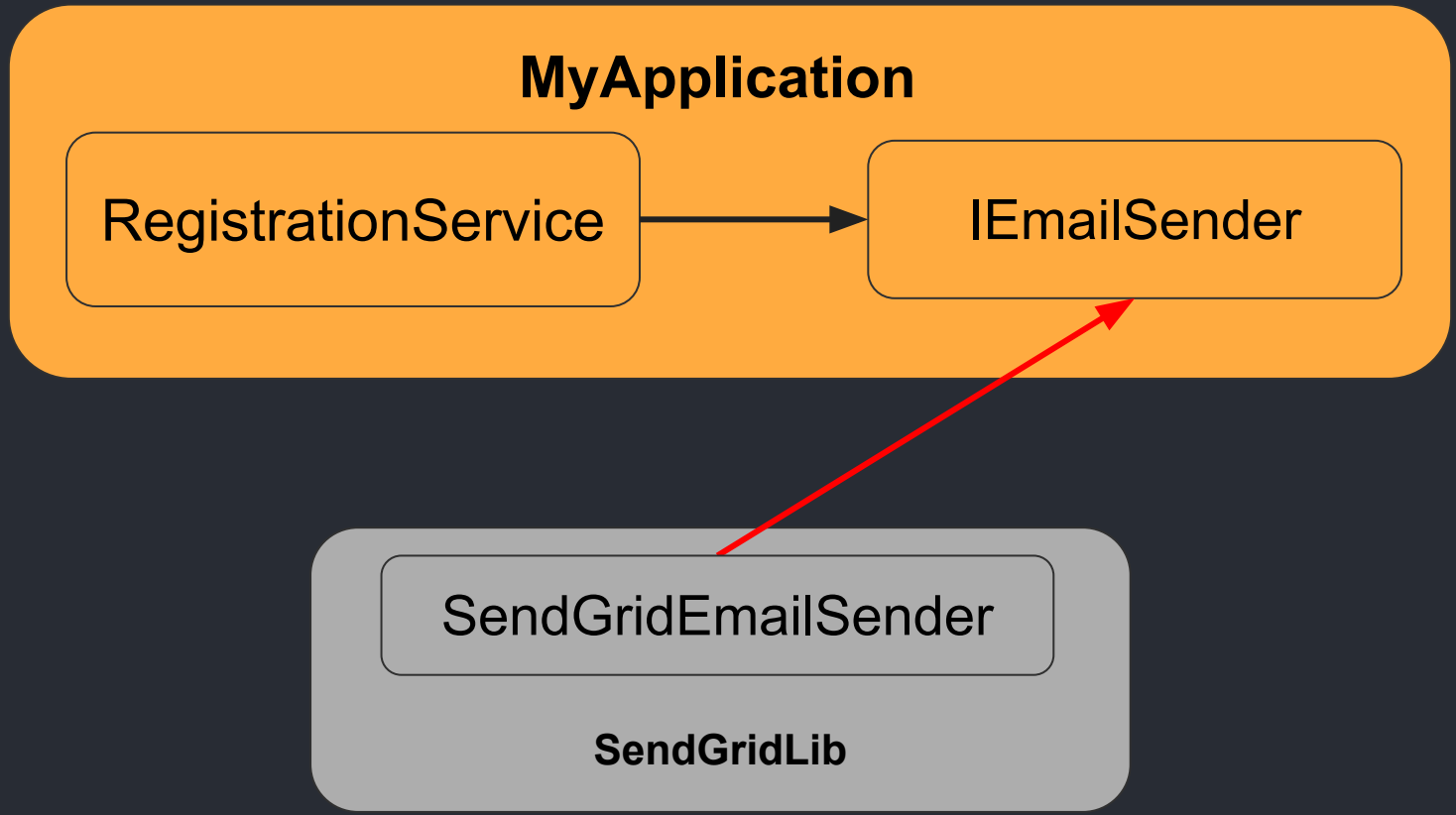
Совсем без инверсии зависимости



Все еще без инверсии зависимостей



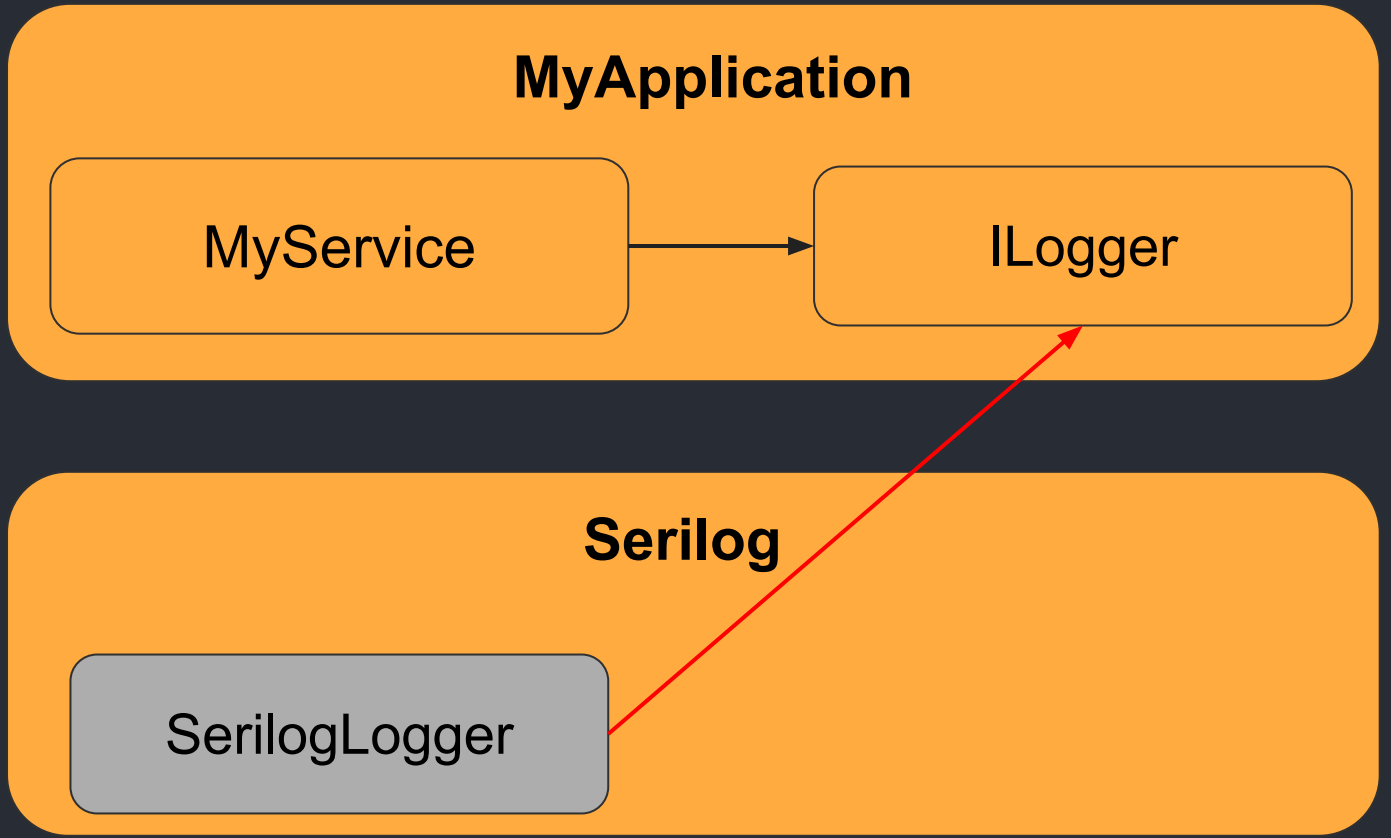
Инвертируем зависимости



Инверсия зависимостей: Пример

- Веб-приложение имеет в кач-ве зависимости интерфейс `ILogger`, он определен на уровне веб-приложения
- Класс `SerilogLogger` из библиотеки Serilog реализует интерфейс `ILogger`
- В итоге получается, что деталь нижнего уровня (`SerilogLogger`) зависит от абстракции высокого уровня (`ILogger`)

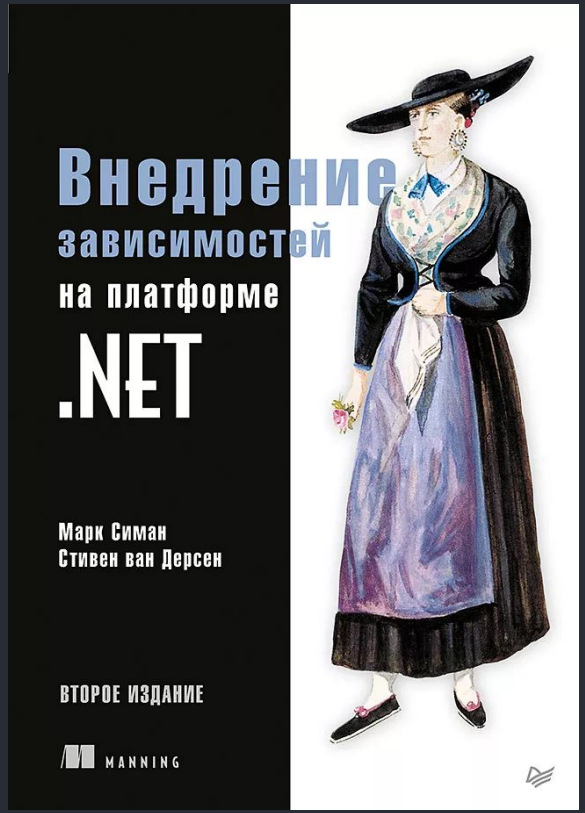
Инверсия зависимостей: Пример



Итоги

- Перед тем, как добавить библиотеку в свою программу проверьте ее потокобезопасность и только после этого примите решение о времени жизни зависимости
- Вы можете выбрать время жизни Singleton только в случае 100% гарантии потокобезопасности. В противном случае лучше выбрать Scoped или Transient.

Внедрение зависимостей на платформе .NET



Домашнее задание

1. Реализуйте сервис отправки Email, используя библиотеку MailKit *.
2. При каждом добавлении нового товара в каталог, отправляйте об этом письмо через созданный сервис.
3. Не забудьте про DIP.

* Для авторизации можете использовать следующие данные:

Сервер: smtp.beget.com (порт 25)
Логин: asp2022gb@rodion-m.ru
Пароль: 3drtLSa1

Спасибо