Зависимости

Service Locator, DI, IoC

Неделя 3

О зависимостях

Представим, что у вас есть какой-то DAO (Data Access Object) класс, который позволяет получить вам доступ к таблице пользователей в БД.

```
public class UserDao {

    public User FindById(int id) {
        // execute a sql query to find the user
    }
}
```

Мы хотим реализовать поиск юзеров по id в БД. Что нам нужно?
Для выполнения этого метода нам потребуется подключение к базе данных. Создадим класс с прямой зависимостью

```
public class UserDao {
    private readonly string _connectionString = "Server=localhost;Database=MyDatabase;User
    Id=root;Password=s3cr3t;";
    public User FindById(int id) {
        using (var connection = new SQLConnection(_connectionString)) {
                connection.Open()
               // выполняем запрос типа «select * from users where id =  ?"

        }
    }
}
```

Теперь DAO **зависит** от источника данных

Проблемы

```
public class UserDao {
    private readonly string _connectionString = "Server=localhost;Database=MyDatabase;User
    Id=root;Password=s3cr3t;";
    public User FindById(int id) {
        using (var connection = new SQLConnection(_connectionString)) {
                connection.Open()
               // выполняем запрос типа «select * from users where id =  ?"

        }
    }
}
```

– Что будет, если мы захотим создать новый класс **ProductDAO**?
(*Ответ:* он будет также зависеть от источника данных, код для получения подключения придется копировать)

– Для каждого отдельного запроса создается новый источник данных, что ресурсозатратно

– Код сложно тестировать из-за прямой зависимости от строки подключения

Идея!

А давайте, вынесем все в глобальный класс Application

```
public static class Application
{
    private static IDbConnection _dataSource;

    public static IDbConnection GetDataSource()
    {
        if (_dataSource == null)
        {
            var connection = new SqlConnection("Server=localhost;Database=MyDatabase;User
Id=root;Password=s3cr3t;");
            connection.Open();
            _dataSource = connection;
        }
        return _dataSource;
    }
}
```

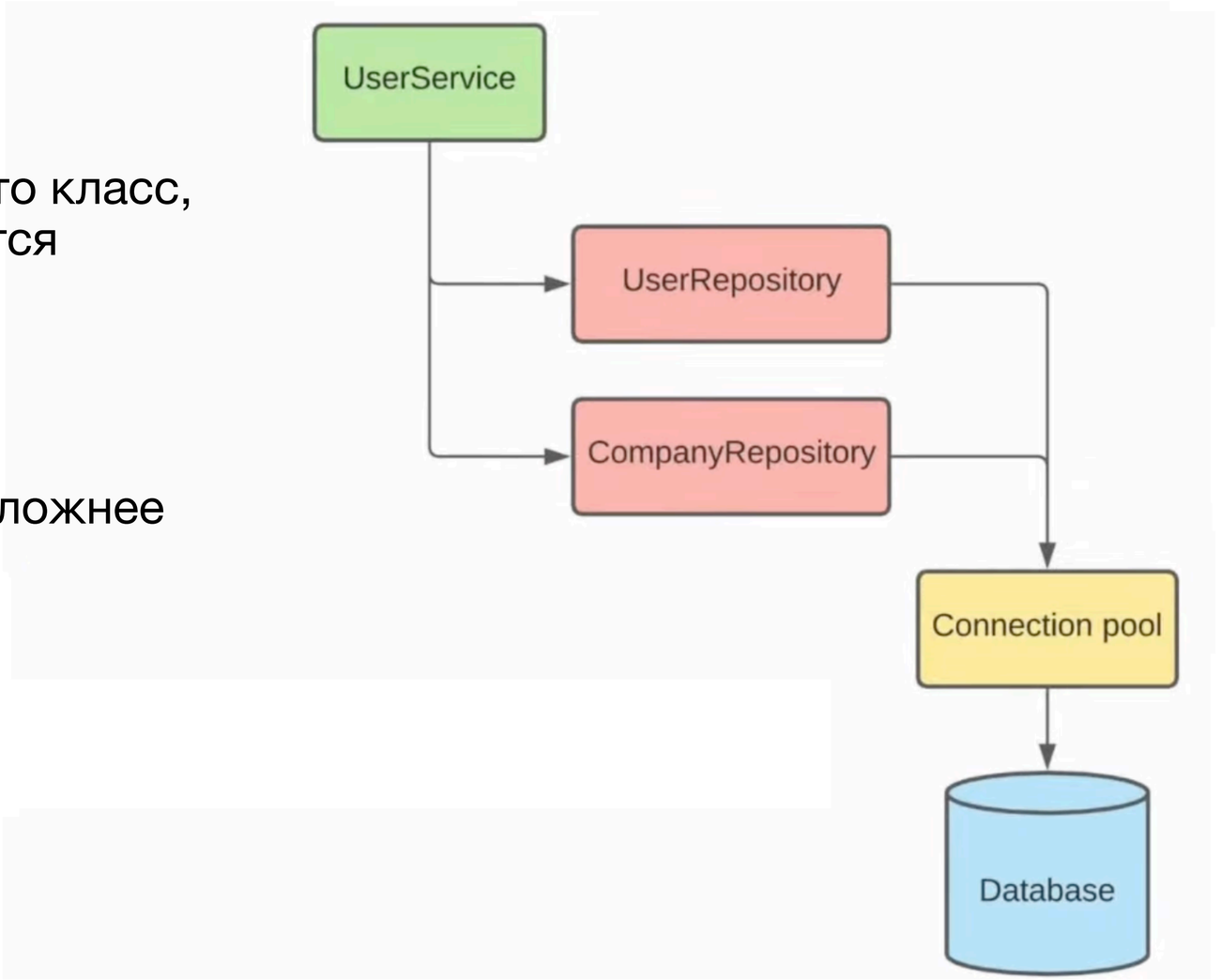
Код в DAO сократиться, мы получим один объект источника данных для всех методов, но класс Application вскоре (скорее всего) разрастется до гигантских масштабов

Еще одна проблема

Цепочка зависимостей

Когда нам потребуется изменить какой-то класс, все зависимые классы также подвергнутся изменениям.

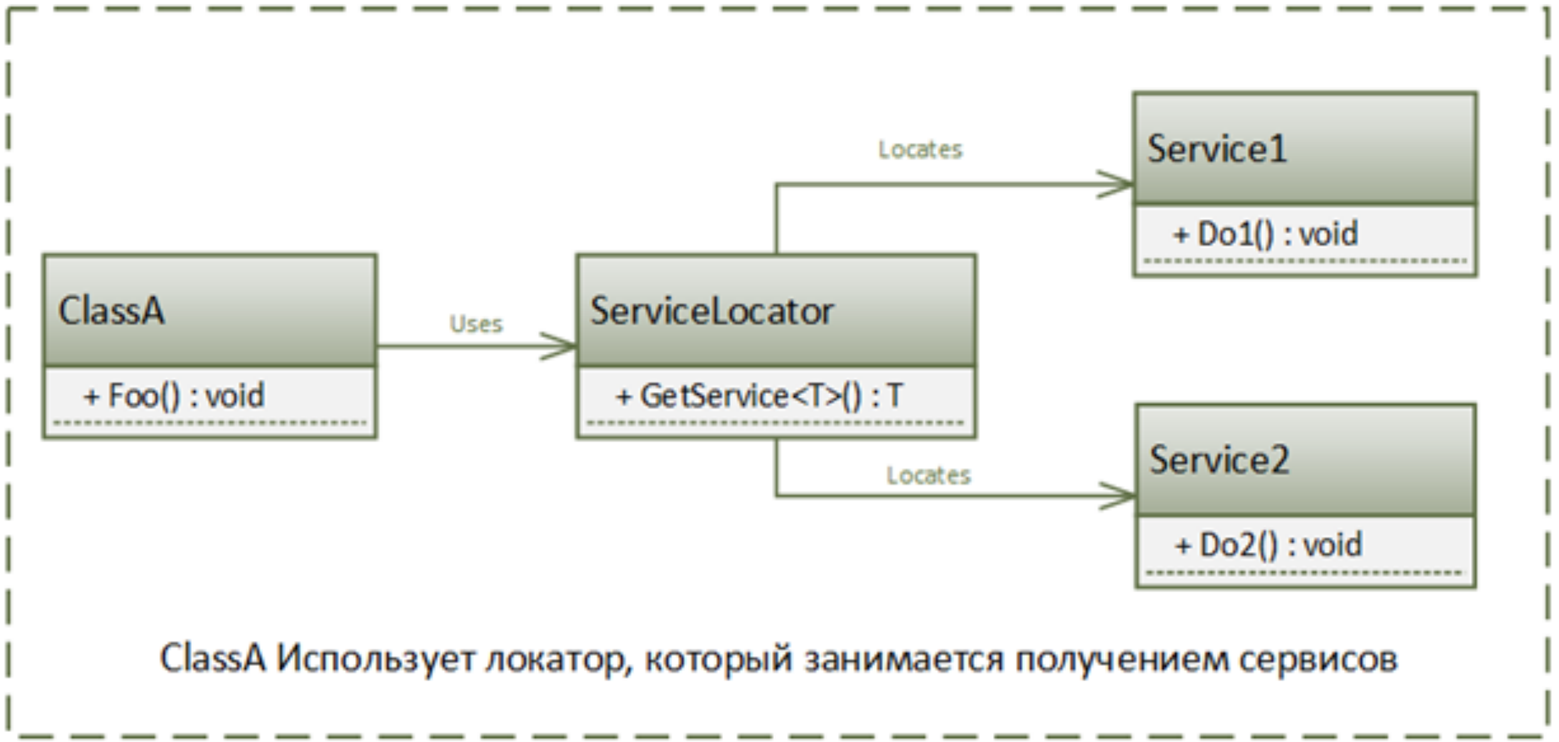
Вывод: чем больше зависимостей, тем сложнее поддерживать приложение.



Service Locator

Паттерн, с которого началось развитие DI-контейнеров

Суть паттерна: за создание («нахождение») сервисов отвечает объект-локатор



Service Locator

Можно описать так

```
// Статический "локатор"
public static class ServiceLocator
{
    public static object GetService(Type type) {}
    public static T GetService<T>() {}
}
```

Или так

```
// Сервис локатор в виде интерфейса
public interface IServiceLocator
{
    T GetService<T>();
}
```

Service Locator

Использовать можно так:

```
class EditEmployeeViewModel
{
    private IServiceLocator _serviceLocator;
    public EditEmployeeViewModel(IServiceLocator serviceLocator)
    {
        _serviceLocator = serviceLocator;
    }
}
```

Или так

```
class EditEmployeeViewModel
{
    private void OkCommandHandler()
    {
        ValidateEmployee(_employee);
        var repository = _serviceLocator.GetService<IRepository>();
        repository.Save(_employee);
    }
}
```

Или так…

```
class EditEmployeViewModel
{
    private readonly IRepository _repository;
    private readonly ILogger _logger;
    private readonly IMailSender _mailSender;

    public EditEmployeViewModel(IServiceLocator locator)
    {

        _repository = locator.GetService<IRepository>();
        _mailSender = locator.GetService<IMailSender>();
        _logger = locator.GetService<ILogger>();
    }
}
```

Основные минусы

Паттерн Service Locator полезен в некоторых случаях, но у него есть недостатки:

1. Класс, в который внедряются зависимости, знает о контейнере зависимостей - это в свою очередь ухудшает портируемость кода

2. По контрактам, которые класс предоставлят, нельзя понять, какие именно зависимости он использует - из-за этого ухудшается тестируемость кода

Инверсия контроля (IoC), DI

Управление выполнением программы передается фреймворку, а не программисту.

Dependency Injection (DI) – одна из реализаций инверсии управления.
Простой пример: передача объекта через конструктор класса.

DI-фреймворк самостоятельно может внедрить нужные зависимости, создать объекты там, где они нужны.

```
public class UserDao {

    private DataSource dataSource;

    private UserDao(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public User findById(Integer id) {
        using (Connection connection = dataSource.getConnection()) {
                PreparedStatement selectStatement =
                  connection.prepareStatement("select * from users where id =  ?");
                // TODO execute the select etc.
        }
    }
```

* UserDao теперь не знает ни о классе приложения, ни о том, как создать источник данных.

DI в .NET

Будем использовать пакет **Microsoft.Extensions.DependencyInjection**

Для его добавления в проект можно использовать как UI-средства, предоставляемые IDE, так и обычный терминал. Рассмотрим способ с терминалом:

1. Перейти в каталог с проектом
2. Запустить терминал
3. В терминале выполнить команду `dotnet add package Microsoft.Extensions.DependencyInjection`
4. После скачивания пакета он будет добавлен в проект

При организации внедрения зависимостей через данный пакет работа делится на следующие этапы:

1. Создание коллекции зависимостей
2. Регистрация зависимостей в коллекции
3. Построение контейнера зависимостей
4. Разрешение зависимостей

DI в .NET

Будем использовать пакет **Microsoft.Extensions.DependencyInjection**

Аналогично с сервис-локатором, первоначально нужно зарегистрировать все сервисы

```
var services = new ServiceCollection();

services.AddSingleton<CarService>(); // регистрируем сервис для управления автомобилями
services.AddSingleton<CustomersStorage>(); // регистрируем сервис управления покупателями
services.AddSingleton<PedalCarFactory>(); // регистрируем сервис создания педальных авто
services.AddSingleton<HandCarFactory>(); // регистрируем сервис создания авто с ручным
приводом
```

Для получения сервисов из коллекции, необходимо создать объект-провайдер

```
var serviceProvider = services.BuildServiceProvider(); // строим контейнер зависимостей
var customers = serviceProvider.GetService<CustomersStorage>();
```

Демо с использованием Spring Framework (для джавистов)

https://github.com/LucyRez/SpringDemoDependencies/tree/master

Ресурсы

1. Презентация адаптирована с этой большой статьи: https://habr.com/ru/articles/490586/