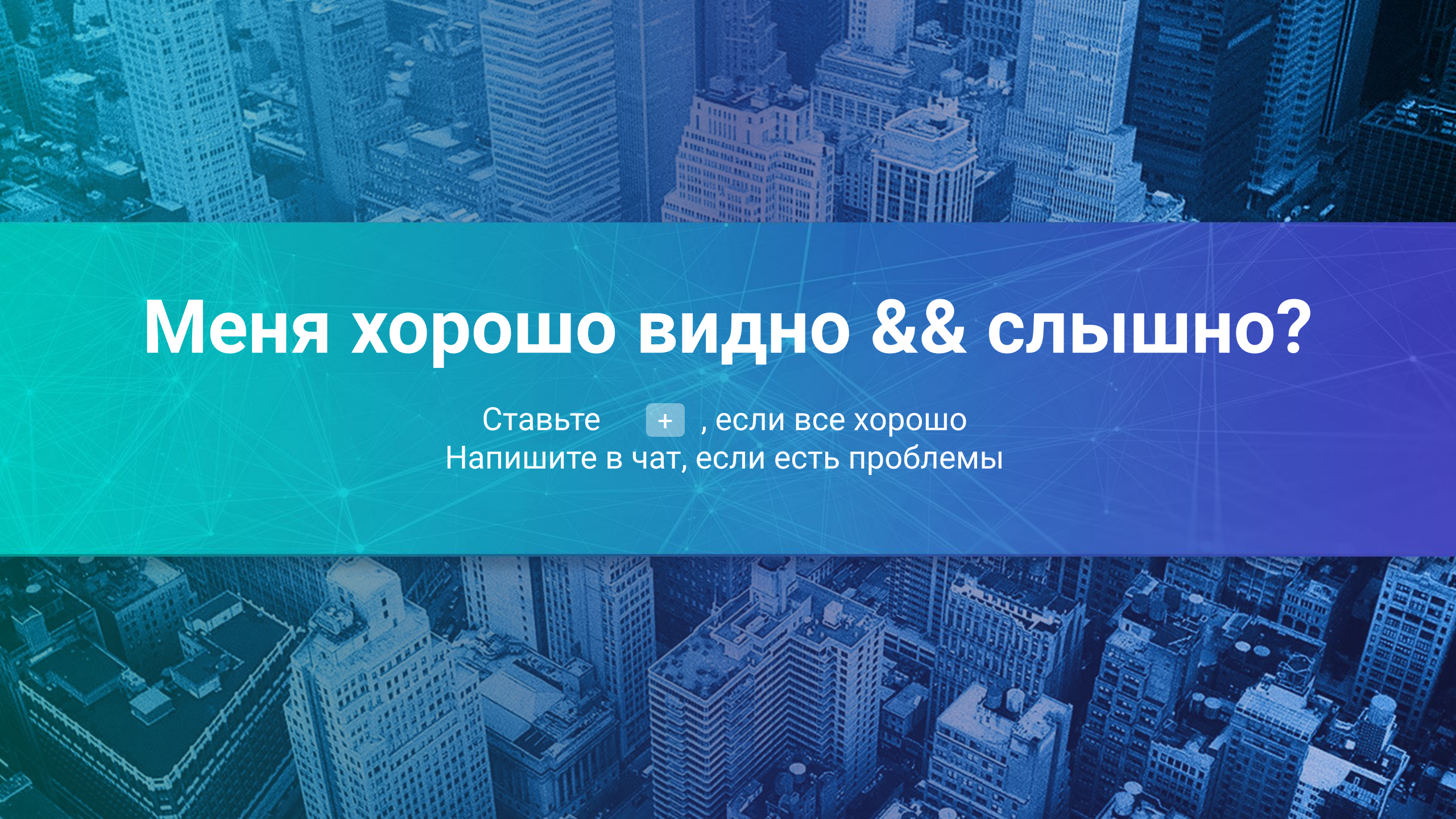
Проверить, идет ли запись!



Меня хорошо видно && слышно?

Ставьте + , если все хорошо
Напишите в чат, если есть проблемы

Стандартный и нестандартные DI контейнеры в ASP.NET Core

Гранковский Андрей

Архитектор направления
Альфа-Банк

https://www.linkedin.com/in/agrankovskiy/

Преподаватель

Гранковский Андрей

- 8 лет опыта в разработке программного обеспечения и из них последние 6 лет в качестве .NET разработчика, в том числе, как Full-stack разработчик.
- В 2014 году закончил МГТУ им. Н.Э. Баумана
- Работал в таких компаниях, как Райффайзенбанк, ЦИАН, Локо-банк
- Имею сертификаты MCP, MCSD: Programming in C#
- Люблю разработку на C#, архитектуру, DDD, тестирование и Agile, стараюсь ориентироваться, как в backend, так и во frontend разработке

Правила вебинара

Активно участвуем

Задаем вопрос в чат

Вопросы вижу в чате, могу ответить не сразу

Цели вебинара

1. Повторить преимущества DI/IOC принципа и основные возможности DI-контейнера для ASP.NET Core

2. Изучить жизненный цикл объектов в DI - контейнере

3. Изучить способы конфигурации нестандартных DI контейнеров и дополнительные инструменты

Смысл | Зачем вам это уметь

1. DI - контейнеры - важнейший механизм для построения расширяемой архитектуры Web-приложений

2. Стандартный DI контейнер подходит для большей части проектов и активно используется

3. Для проектов, где нужны продвинутые инструменты могут понадобиться другие контейнеры

Маршрут вебинара

Best Practices/DI/IOC

↓

DI-контейнер ASP.NET Core

↓

Жизненный цикл объектов в DI-контейнере

↓

Нестандартные DI-контейнеры и расширения

Репозиторий с примером

Тайминг: 1 минута

Репозиторий с проектом для занятия, кому удобнее смотреть у себя - клонируем

https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Напишите в чат +, если репозиторий доступен


DI/IOC

Маршрут вебинара

Best Practices/DI/IOC

↓

DI-контейнер ASP.NET Core

↓

Жизненный цикл объектов в DI-контейнере

↓

Нестандартные DI-контейнеры и расширения

Best Practices

Вопрос

Тайминг: 1 минута

Кто уже делал дополнительное задание про Employees CRUD в первом ДЗ?

Напишите в чат + или -

Вопрос

Тайминг: 1 минута

Как вы считаете много ли кода приходится на заполнение и маппинг данных из одних объектов в другие, например из Models в Domain Entity и наоборот?

Напишите в чат сколько это в процентах по вашему мнению 20%, 30% и т.д.

Минутка Best Practices

Такого кода очень много, многие операции бизнес-логики сводятся к простому маппингу в существующие или новые объекты.

Использование инициализаторов ведет к ошибкам, так как нарушает инкапсуляцию создания и изменения объекта

Субъективно > 50% ошибок вызвано ошибками в Create/Update операциях из-за копипаста, контроллеры и сервисы получаются “толстыми” - в итоге много плохого кода

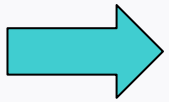
Минутка Best Practices

Стараемся выносить маппинг и создание объектов в отдельные компоненты (Мапперы, Фабрики) и/или использовать конструкторы сущностей/агрегатов

Инициализация и мапперы

```
employee.FirstName = model.FirstName;
employee.LastName = model.LastName;
employee.AppliedPromocodesCount = model.AppliedPromocodesCount;
employee.Email = model.Email;
employee.Roles = await _rolesRepository.GetByCondition( predicate: x :Role =>
    model.RoleNames.Contains(x.Name)) as List<Role>;
```

```
//Нарушаем инкапсуляцию создания объекта
Employee employee = new Employee()
{
    Id = Guid.NewGuid(),
    FirstName = model.FirstName,
    LastName = model.LastName,
    Email = model.Email,
    AppliedPromocodesCount = model.AppliedPromocodesCount,
    Roles = await _rolesRepository // IRepository<Role>
        .GetByCondition( predicate: x :Role => model.RoleNames.Contains(x.Name)) as List<Role>
};
```



```
var roles  = await _rolesRepository.GetByCondition( predicate: x :Role =>
    model.RoleNames.Contains(x.Name)) as List<Role>;

employee = EmployeeMapper.MapFromModel(model, roles, employee);
```

Инициализация и мапперы

Плюсы:

1. Лучше Single Responsibility;
2. Соблюдаем инкапсуляцию при создании объектов;
3. Меньше багов
4. Легче покрыть unit-тестами
5. Код бизнес-логики становится читаемее в разы, в итоге лучше поддержка

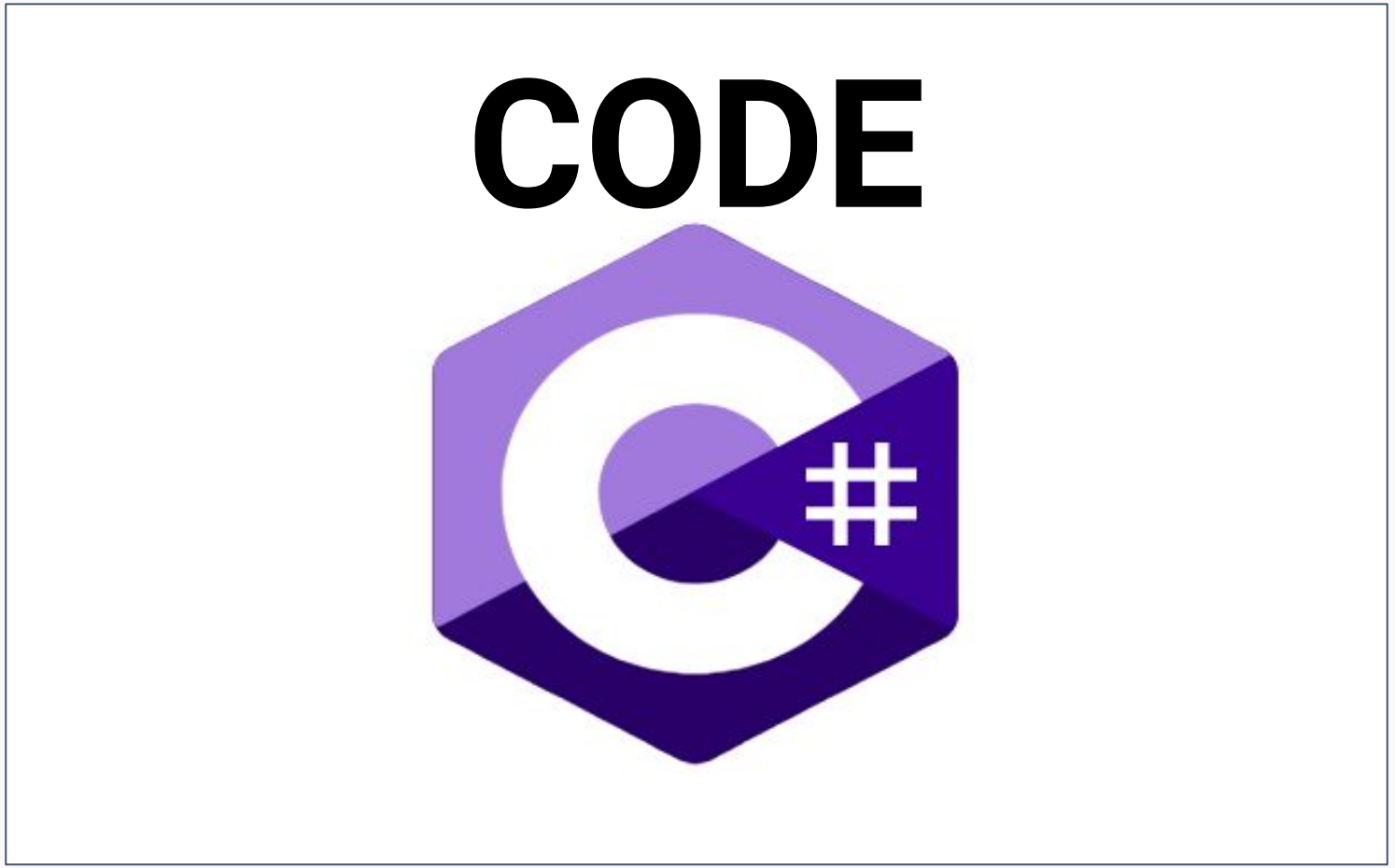
Минусы

1. Иногда можем смешивать операции создания/обновления, тогда используем конструкторы и отдельные Edit методы внутри класса сущности/агрегата или специальную фабрику
2. Если все делать правильно, то нужно использовать классы-зависимости, например, IEmployeeFactory, в итоге много компонентов, но можно обойтись статическими классами для этого, обычно их достаточно

Про Automapper

Эти проблемы частично решает Automapper, но обычно в сторону простых моделей от Entities, плюс создаем сильную связь с этой библиотекой, но в целом вариант хороший

Инициализация и мапперы



https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

EmployeesBestPracticesController


DI/IOC

Вопрос

Тайминг: 2 минуты

Что вообще такое зависимость класса?

Напишите в чат или -, если нужно пояснить

Зависимости классов

Зависимость — это любой объект, который требуется другому объекту.

Зависимости классов

```
var roles  = await _rolesRepository.GetByCondition( predicate: x :Role  =>
    model.RoleNames.Contains(x.Name)) as List<Role>;

var employee = EmployeeStaticMapper.MapFromModel(model, roles);

try
{
    await _employeeRepository.CreateAsync(employee);
}
catch (Exception e)
{
    return BadRequest(e.Message);
}
```

Зависимость

Зависимость

```
var rolesRepository = new InMemoryRepository<Role>(FakeDataFactory.Roles);

var roles  = await rolesRepository.GetByCondition( predicate: x :Role  =>
    model.RoleNames.Contains(x.Name)) as List<Role>;
```

Зависимость

Вопрос

Какие есть проблемы/преимущества у данных вариантов зависимостей?
Напишите в чат по каждому виду: цифра - пояснение

```
var roles  = await _rolesRepository.GetByCondition( predicate: x :Role  =>
    model.RoleNames.Contains(x.Name)) as List<Role>;
```

1

```
var employee = EmployeeStaticMapper.MapFromModel(model, roles);
```

2

```
var rolesRepository = new InMemoryRepository<Role>(FakeDataFactory.Roles);

var roles  = await rolesRepository.GetByCondition( predicate: x :Role  =>
    model.RoleNames.Contains(x.Name)) as List<Role>;
```

3

Пример проблем с зависимостями из жизни

Пример проблем с зависимостями из жизни

Есть Web-приложение, в нем есть функция генерации .pdf файла отчета на основе отчета MS SQL Reporting Service

1. Пользователь в меню приложения выбирает отчет и настраивает входные параметры;
2. Идет вызов ReportController/МетодДляНужногоОтчета
3. Там происходит сбор входных параметров и передача их в MS SQL Reporting Service, конфиг для доступа лежит в web.config. Доступ к серверу отчетов сделан в статическом классе;
4. Отчетов более 300 штук, в каждом методе вызывается статический класс
5. Появляется задача перенести эти настройки в БД, в приложении настройки уже везде пробрасываются через DI;
6. Как итог надо переписать 300 методов, чтобы перевести работу с настройками на объект из контейнера и заменить статический класс обычным, можно было это сделать заранее, а не писать статический класс, чтобы сделать быстро;

Dependency Injection

```
var rolesRepository = new InMemoryRepository<Role>(FakeDataFactory.Roles);

var roles  = await rolesRepository.GetByCondition( predicate: x :Role  =>
    model.RoleNames.Contains(x.Name)) as List<Role>;
```

```
public EmployeesController(IRepository<Employee> employeeRepository, IRepository<Role> rolesRepository)
{
    _employeeRepository = employeeRepository;
    _rolesRepository = rolesRepository;
}
```

Чтобы явно знать от каких классов зависит другой класс мы используем инъекции в конструктор, мы точно знаем, что нужно классу для работы, а объявить эти зависимости попросим другой класс - Composition Root, можно задействовать полиморфизм, увеличим гибкость программы, сделать ее модульной

SOLID

S - Single Responsibility principle
O - Open/Closed principle
L - Liskov substitution principle
I - Interface segregation principle
D - Dependency inversion principle

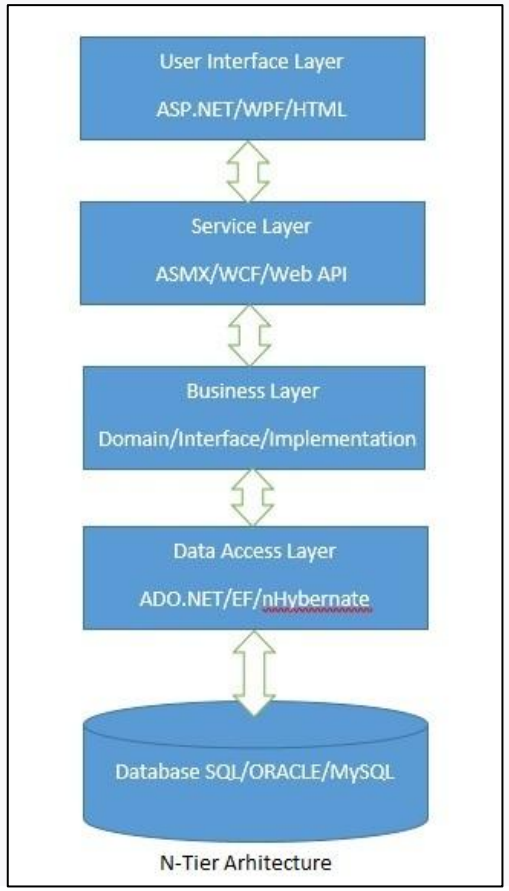
Вопрос

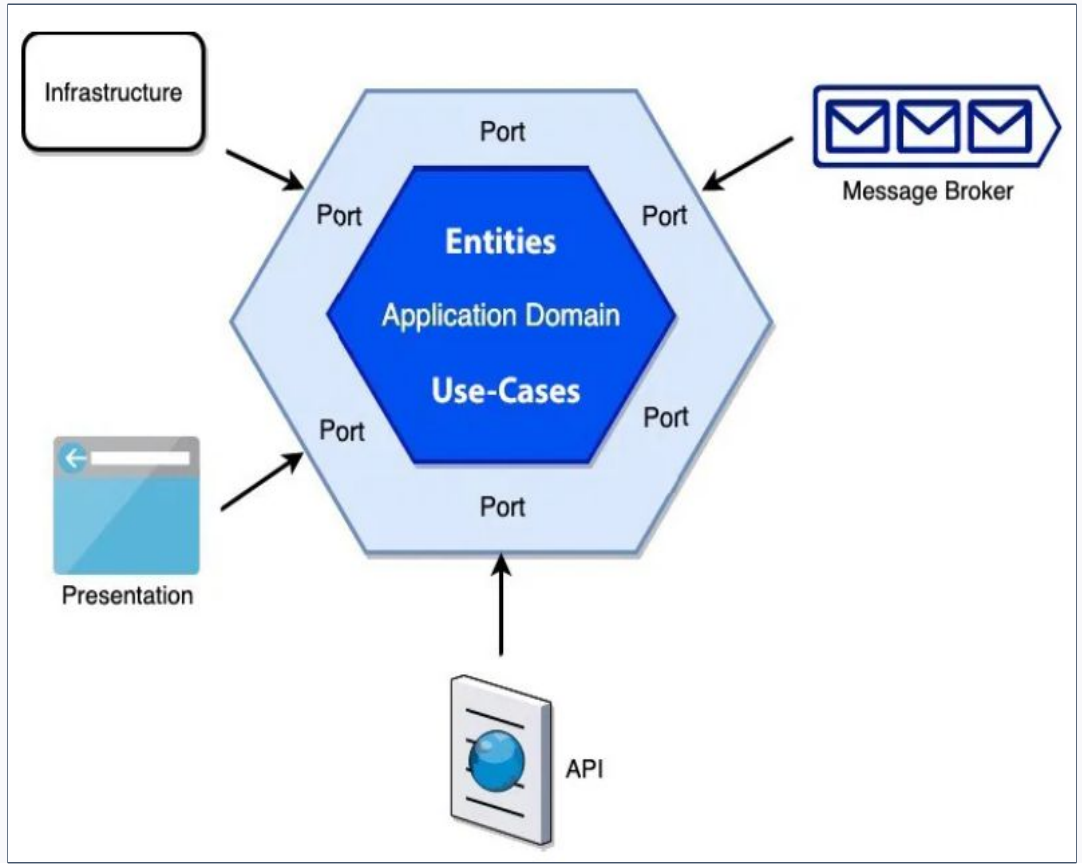
Тайминг: 3 минуты

В чем отличие Dependency Injection и почему говорят еще про Dependency Inversion и Inversion of control?
Напишите в чат или -, +-, если кажется, что это одно и то же

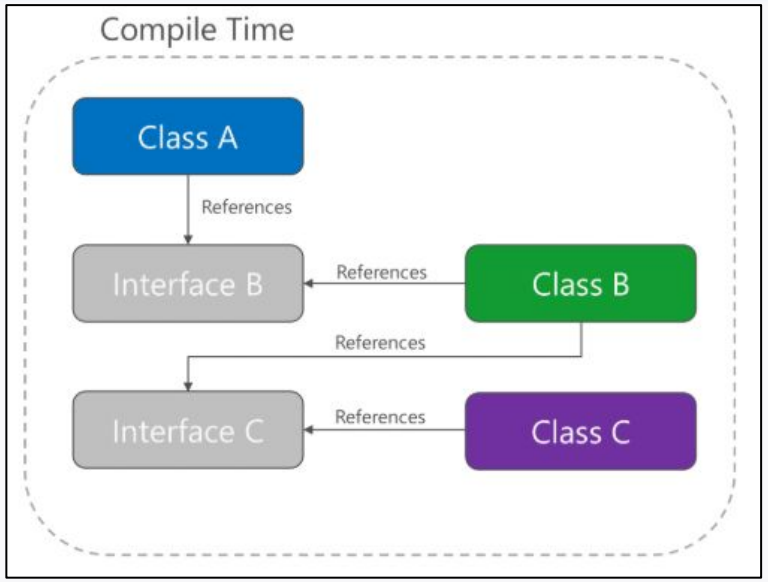
Многоуровневая и гексагональная архитектура

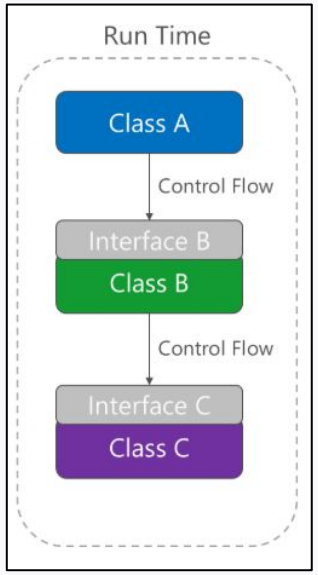
В чем разница у этих архитектур с точки зрения DI/IOC или ее нет, ведь и там и там интерфейсы можно подкладывать? Напишите в чат или -





DI/IOC





Формулировка:

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.
- Зависимости идут в направлении противоположному потоку управления (Поэтому Inversion Of Control)

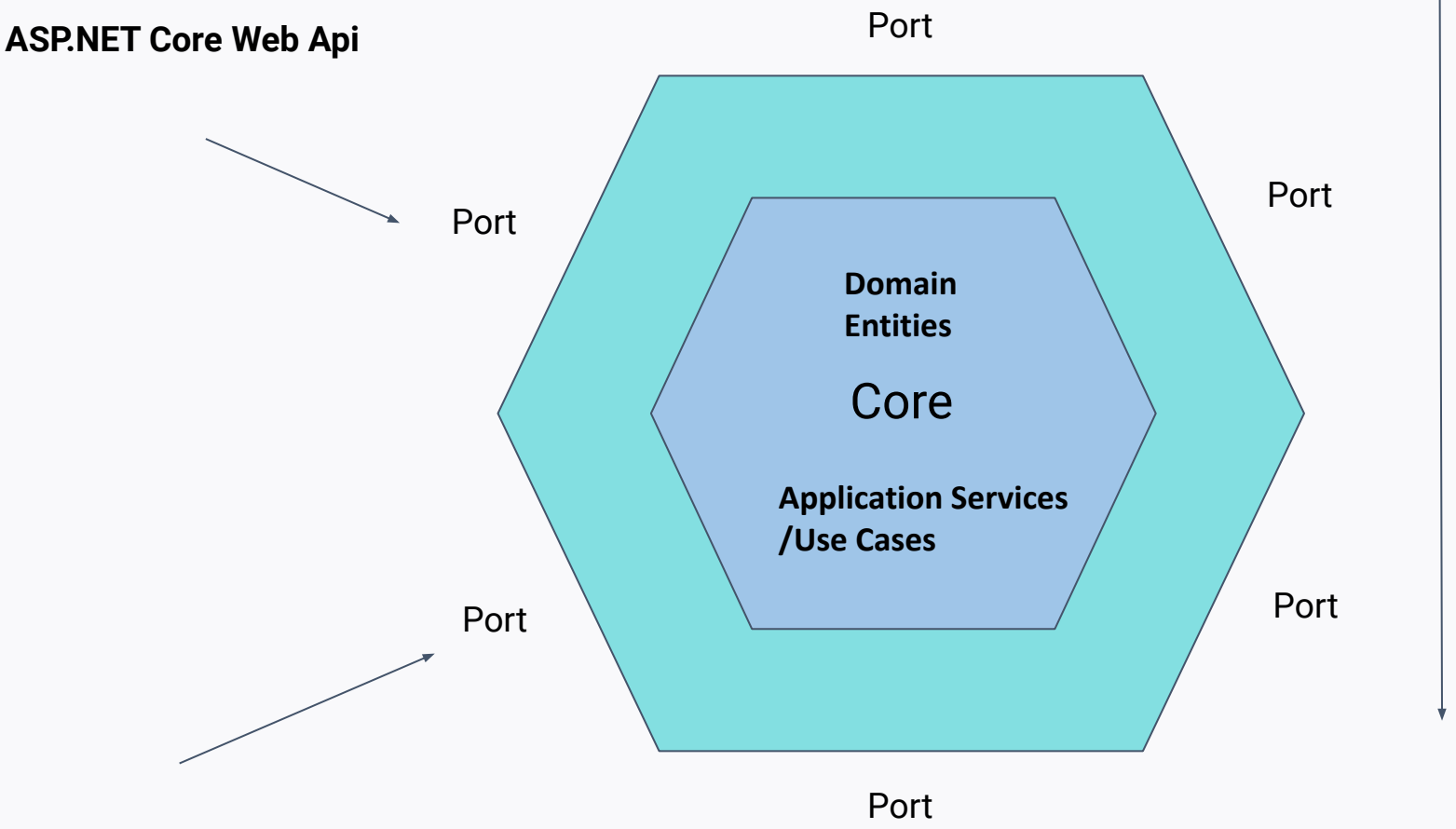
Контейнер

Контейнер зависимостей (DI-контейнер) - это только инструмент создания/жизненного цикла и инъекции зависимостей, он ничего не знает про IOC, если конечно мы не имеем в виду под этим перенос инициализации зависимостей в Composition Root

Будет ли наша архитектура соблюдать IOC зависит от нас

IOC - про связи компонентов, а не про контейнеры

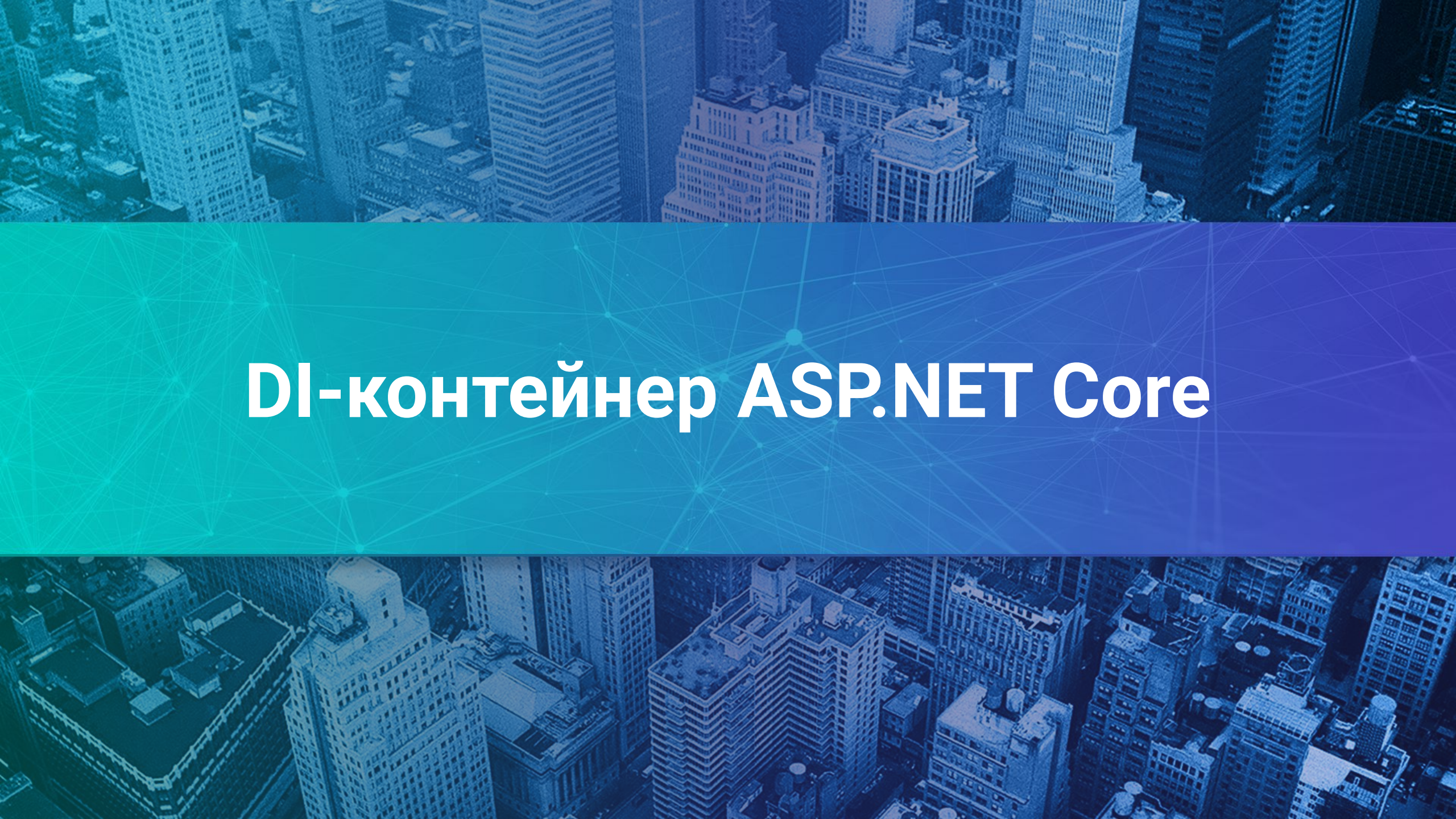
Гексагональная архитектура



- Вызываем API
  GET api/users/1
- Вызываем метод
  GetUserById(1) интерфейса
  IUserUseCasesService
- Вызываем метод
  FindUserById(1) интерфейса
  IUserRepository
- Нашли пользователя в БД и вернули ответ

Интерфейсы определены в Core

Запрос идет сверху - вниз, от UI до реализации репозитория, но бизнес-логика использует интерфейс, который определен ниже;
Зависимости “снаружи-внутри”, хотя поток выполнения программы идет как обычно



DI-контейнер ASP.NET Core

Маршрут вебинара

Best Practices/DI/IOC

↓

DI-контейнер ASP.NET Core

↓

Жизненный цикл объектов в DI-контейнере

↓

Нестандартные DI-контейнеры и расширения

ASP.NET Core и DI

DI - это основа архитектуры ASP.NET Core и отличие от предыдущего ASP.NET, так как любой элемент внутренней инфраструктуры может быть изменен, как и пользовательские компоненты.
Реализация находится в пакете
Microsoft.Extensions.DependencyInjection

Возможности DI-контейнера ASP.NET Core

- Встроенный контейнер зависимостей предназначен для платформы ASP.NET Core и большинства клиентских приложений;
- Это фактически самый быстрый контейнер в .NET
- Встроенный контейнер поддерживает основные инструменты, которые нужны:
  i. Внедрение в конструкторы
  ii. Использование реализации по умолчанию и Generic-реализации
  iii. Управление временем жизни объекта (3 основных режима) и Scope объекта
  iv. Легкие инструменты расширения и замены контейнера
  v. Внедрение, как платформенных служб, так и клиентских
  vi. Абстракции контейнера - это основа гибкой архитектуры ASP.NET Core

Возможности DI-контейнера ASP.NET Core

Контейнер не поддерживает функции, которые на самом деле не нужны для большинства приложений:

1. Инъекции в свойство;
2. Подконтейнеры и другие средства реализации плагинной архитектуры;
3. Динамический резолв зависимостей по соглашению (вот это полезная фича)
4. Некоторые другие функции…

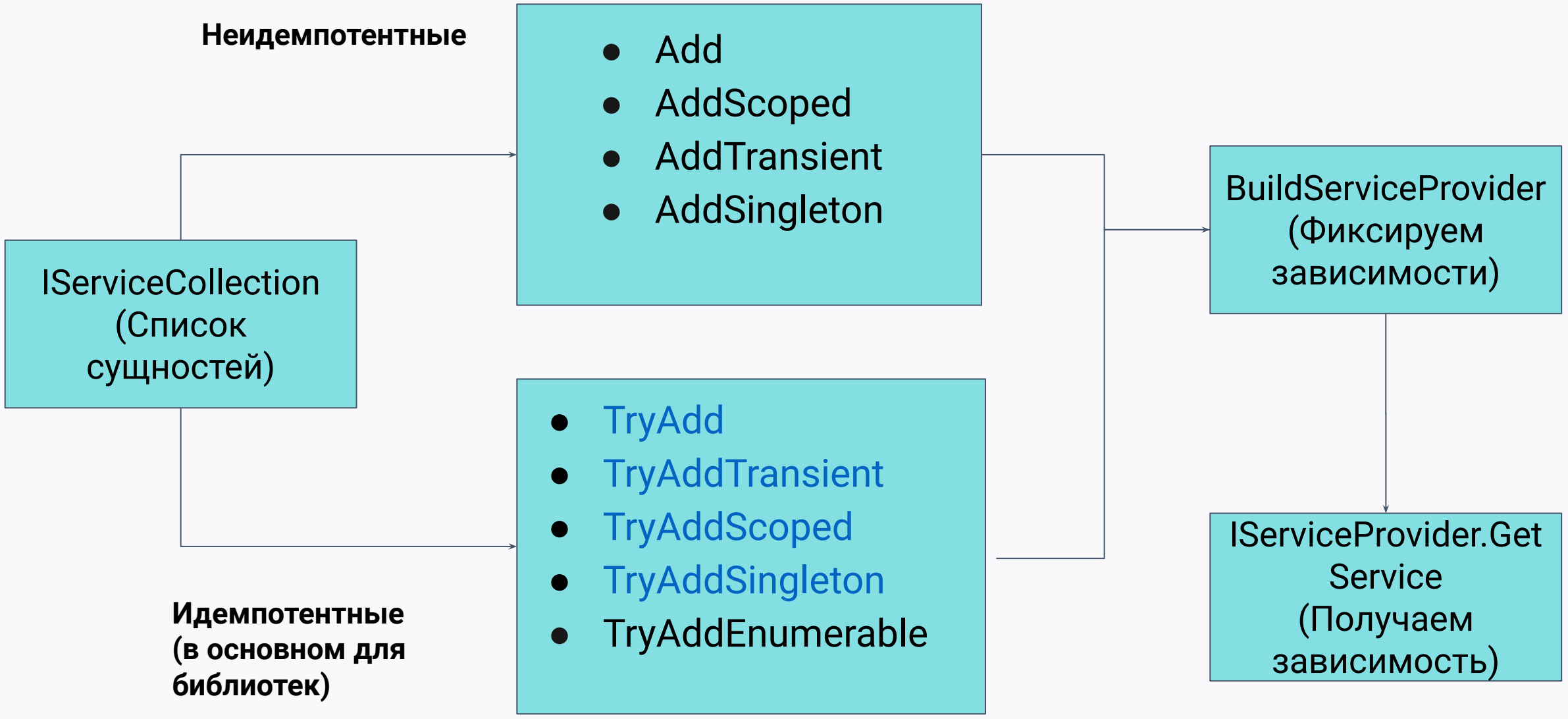
Он поддерживает большинство функций, которые нужны для микросервисов и средних приложений

Где конфигурируем зависимости?

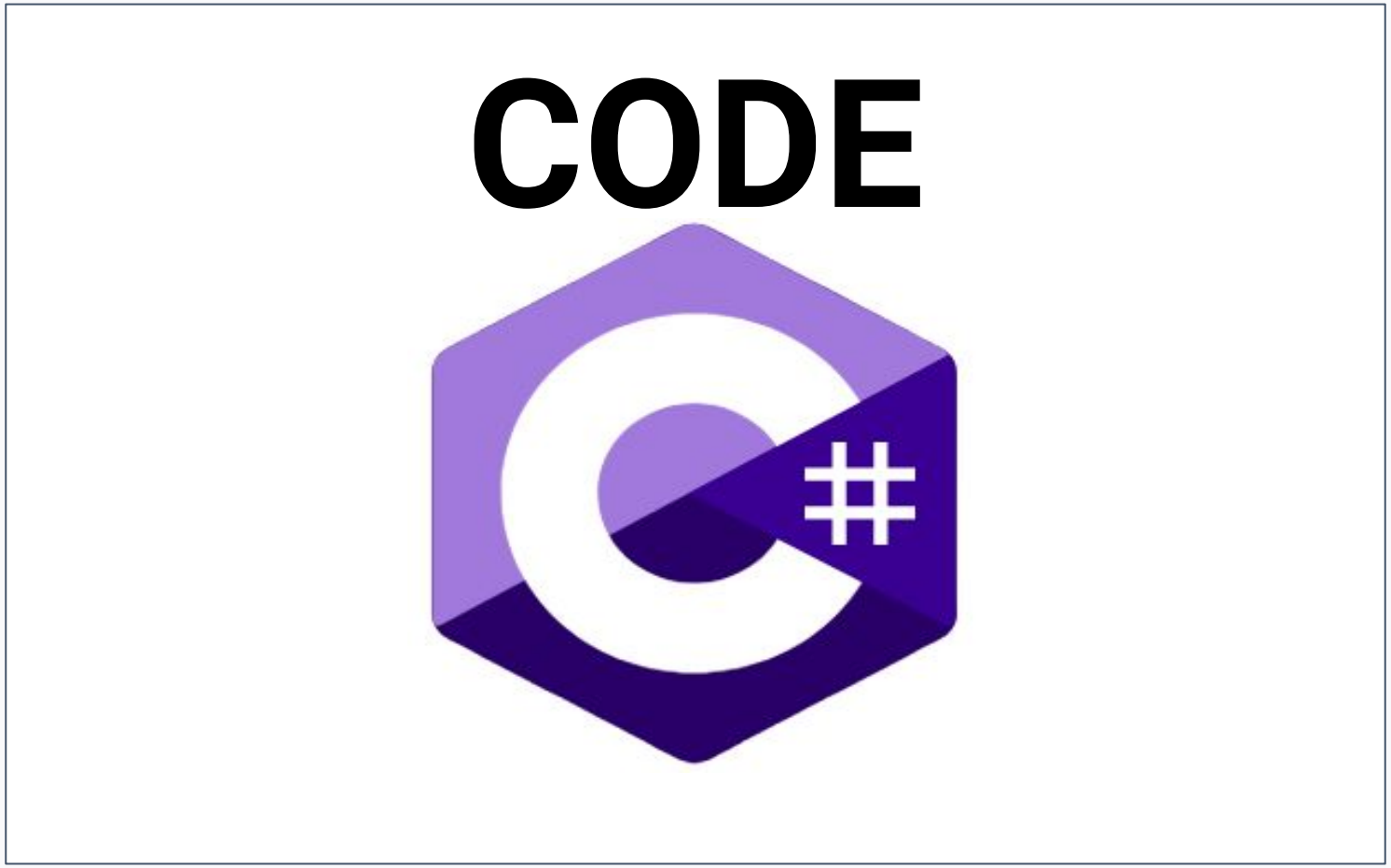
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers( configure: x :MvcOptions => x.SuppressAsyncSuffixInActionNames = false);
    services.AddScoped(typeof(IRepository<Employee>), implementationFactory: (x :IServiceProvider ) =>
        new InMemoryRepository<Employee>(FakeDataFactory.Employees));
    services.AddScoped(typeof(IRepository<Role>), implementationFactory: (x :IServiceProvider ) =>
        new InMemoryRepository<Role>(FakeDataFactory.Roles));
}
```

IServiceCollection - основная абстракция для работы с сервисами, которые хотим зарегистрировать в контейнере
Вызов конфигурации происходит при старте приложения, зависимости разрешаются на каждый запрос в зависимости от жизненного цикла объекта

Основные методы и сущности



Как работать с контейнером изолированно



https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Жизненный цикл объектов в DI-контейнере

Маршрут вебинара

Best Practices/DI/IOC

↓

DI-контейнер ASP.NET Core

↓

Жизненный цикл объектов в DI-контейнере

↓

Нестандартные DI-контейнеры и расширения

Жизненный цикл объектов в DI-контейнере

Три вида жизненного цикла зависимостей

Transient

Scoped

Singleton

Вопрос

Тайминг: 2 минуты

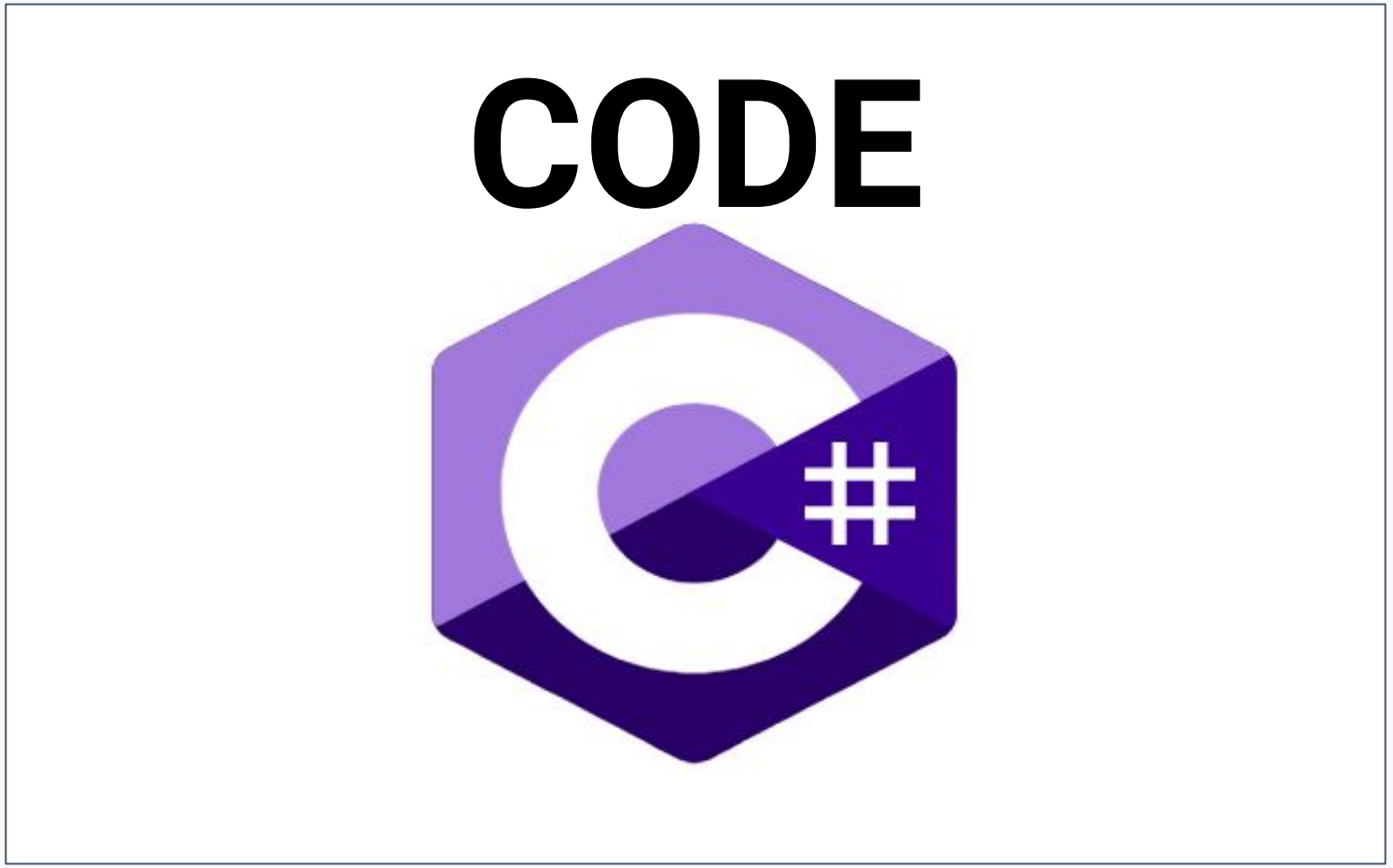
Что такое Transient и зачем нам может понадобиться Transient зависимость?

Напишите в чат или -, если надо пояснить

Transient в ASP.NET Core

Зависимость создается каждый раз, когда она нам нужна, хорошо подходит для Stateless компонентов и если есть многопоточность

Transient зависимости

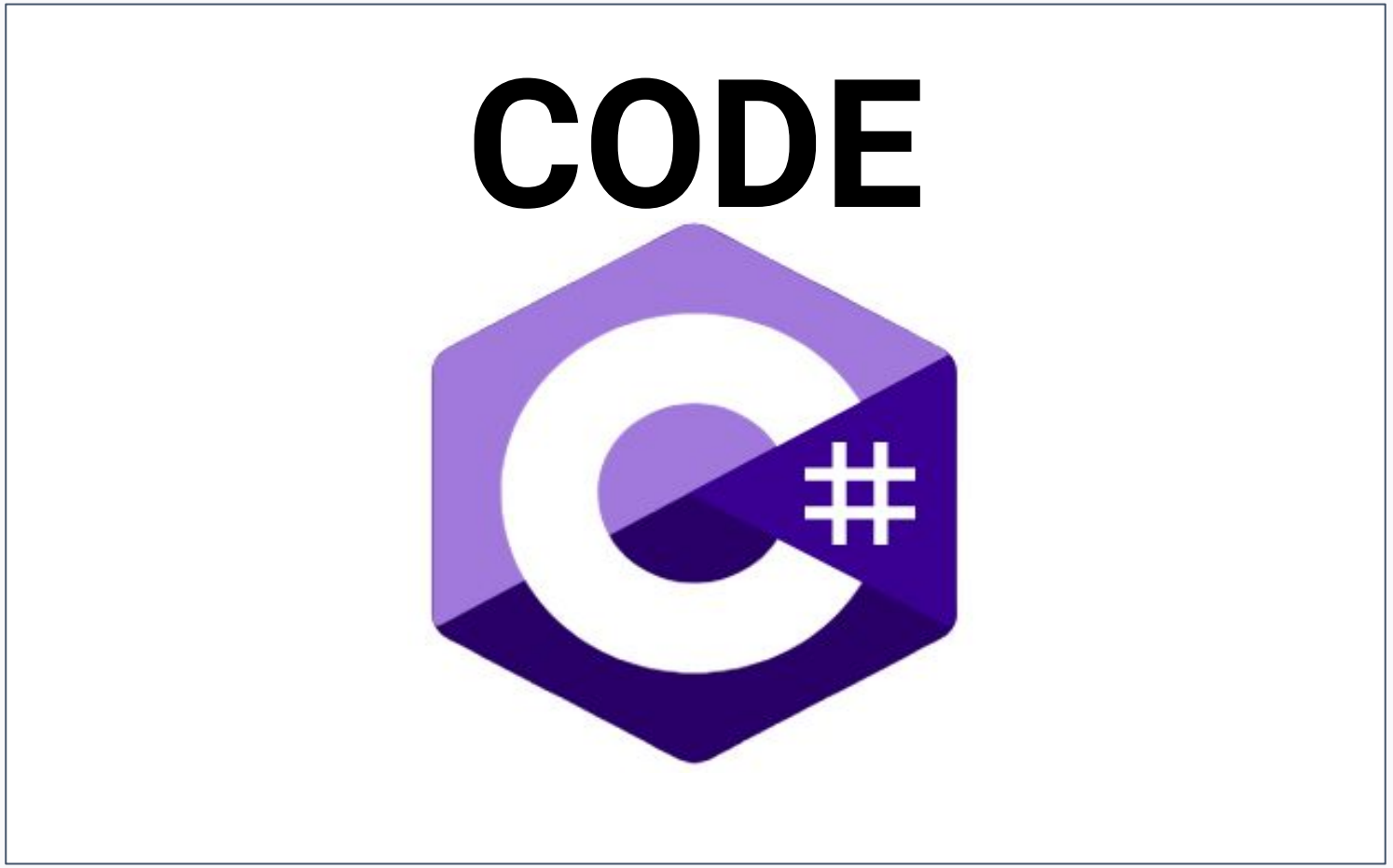


https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Singleton в ASP.NET Core

Один экземпляр на все запросы, то есть будет создан один раз, не очень при многопоточной работе, у него не должно быть изменяемого состояния
Нельзя использовать со Scoped

Singleton зависимости



https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Вопрос

Тайминг: 2 минуты

Как работает Scoped для ASP.NET Core?

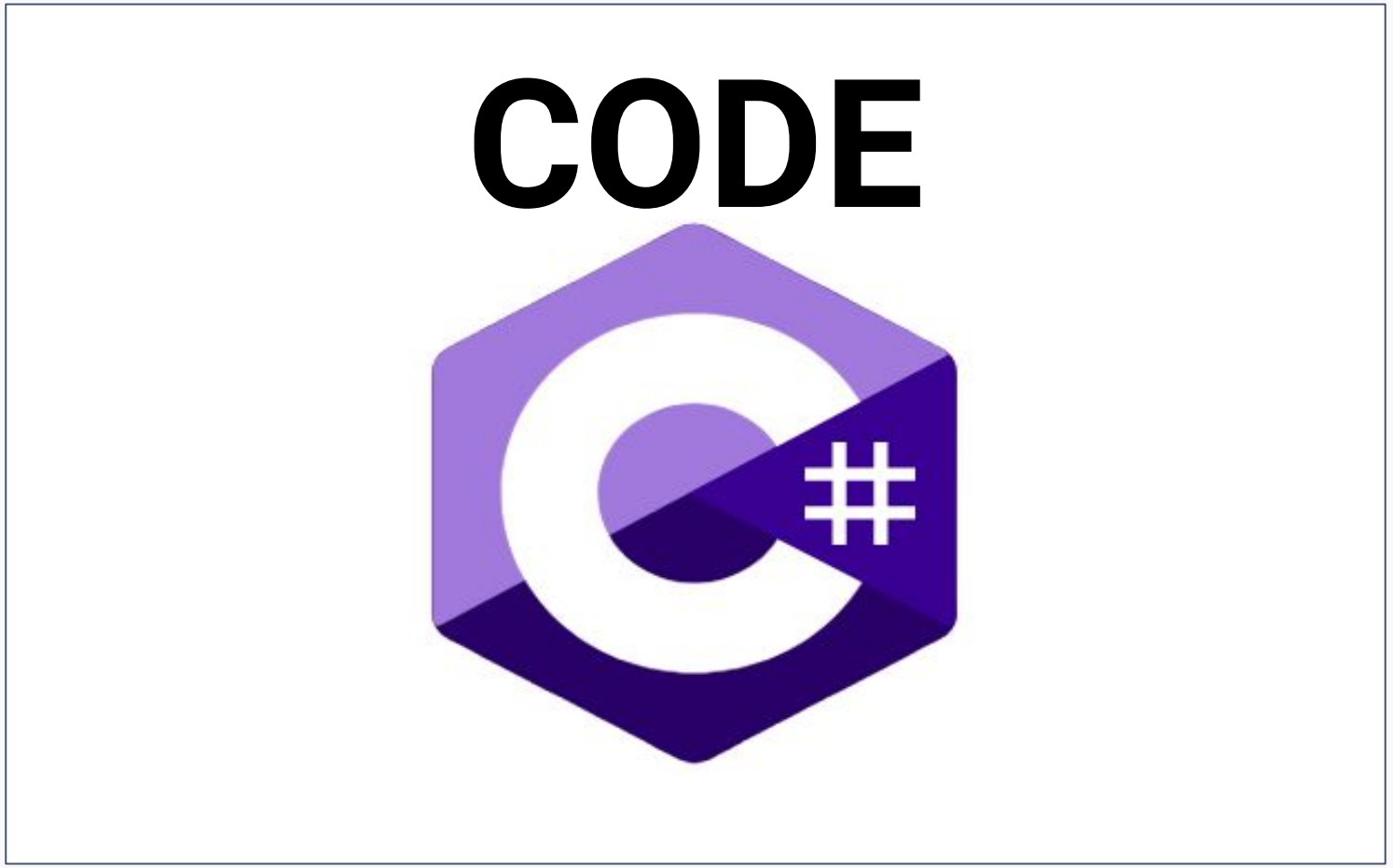
Напишите в чат или -, если надо пояснить

Scoped в ASP.NET Core

Основное, что надо знать про Scoped в ASP.NET Core:

Будет создан один экземпляр каждой зависимости пока не закончился запрос, то есть мы не вернули ответ клиенту
Все Disposed объекты будут жить до конца Scope

Как создаем Scope



https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Зачем создавать новый Scope?

Например, если все зависимости в рамках запроса разрешены, как Scoped, а нам нужна новая, иногда такое нужно при работе с EF, так как DbContext обычно существует на запрос, чтобы фиксировать транзакции в рамках запроса

Если нужен новый DbContext, то может понадобиться создать Scope, но этого лучше не делать, так как это инфраструктурный код

Про жизненный цикл

1. При старте приложения ASP.NET Core собираем провайдер
2. Провайдер будет существовать пока не остановим приложение
3. То есть все Singleton зависимости будут существовать до остановки приложения, поэтому он IDisposable
4. Для Scoped объекты привязаны к запросу или using scope, в итоге будут собраны и вызван Dispose

Нестандартные DI-контейнеры и расширения

Маршрут вебинара

Best Practices/DI/IOC

↓

DI-контейнер ASP.NET Core

↓

Жизненный цикл объектов в DI-контейнере

↓

Нестандартные DI-контейнеры и расширения

Зачем менять DI контейнер в ASP.NET Core

- Инъекция в свойство;
- Инъекция по имени;
- Дочерние контейнеры;
- Настраиваемое управление временем существования;
- Регистрация на основе соглашения;
- Регистрация с помощью модулей, когда вы можете указать класс, который инкапсулирует конфигурацию

DI контейнеры в .NET

- Autofac
- Castle Windsor
- Lamar
- LightInject
- Ninject
- SimpleInjector
- Spring.NET
- Unity
- LinFu *(inactive)*
- Managed Extensibility Framework (MEF) *(abandoned / deprecated)*
- PicoContainer.NET *(abandoned / deprecated)*
- S2Container.NET *(abandoned / deprecated)*
- StructureMap *(abandoned / deprecated)*

https://www.claudiobernasconi.ch/2019/01/24/the-ultimate-list-of-net-dependency-injection-frameworks/

Сравнение контейнеров

Можно посмотреть по ссылке:
https://danielpalme.github.io/IocPerformance

https://habr.com/ru/post/302240/

Если кратко

Autofac достаточно производительный и является одним из самых популярных для ASP.NET MVC и совместим с ASP.NET Core, у него хорошая документация и поддержка
SimpleInjector также достаточно популярен и совместим с ASP.NET Core

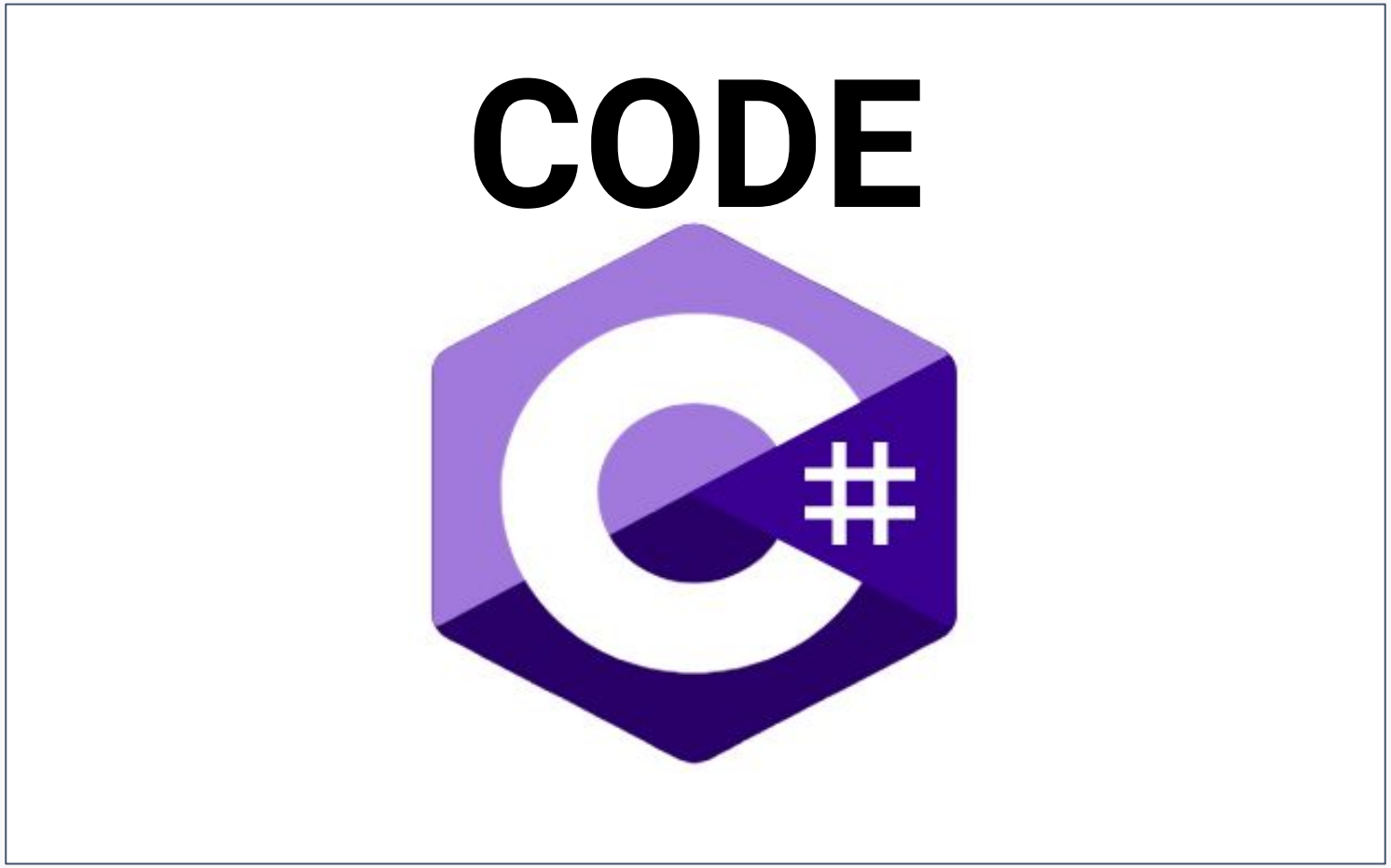
Ninject, Castle Windsdor и Unity уже нет особого смысла рассматривать

Но они все равно проигрывают по производительности стандартному контейнеру Microsoft.Extension.DependencyInjection

Контейнер Autofac

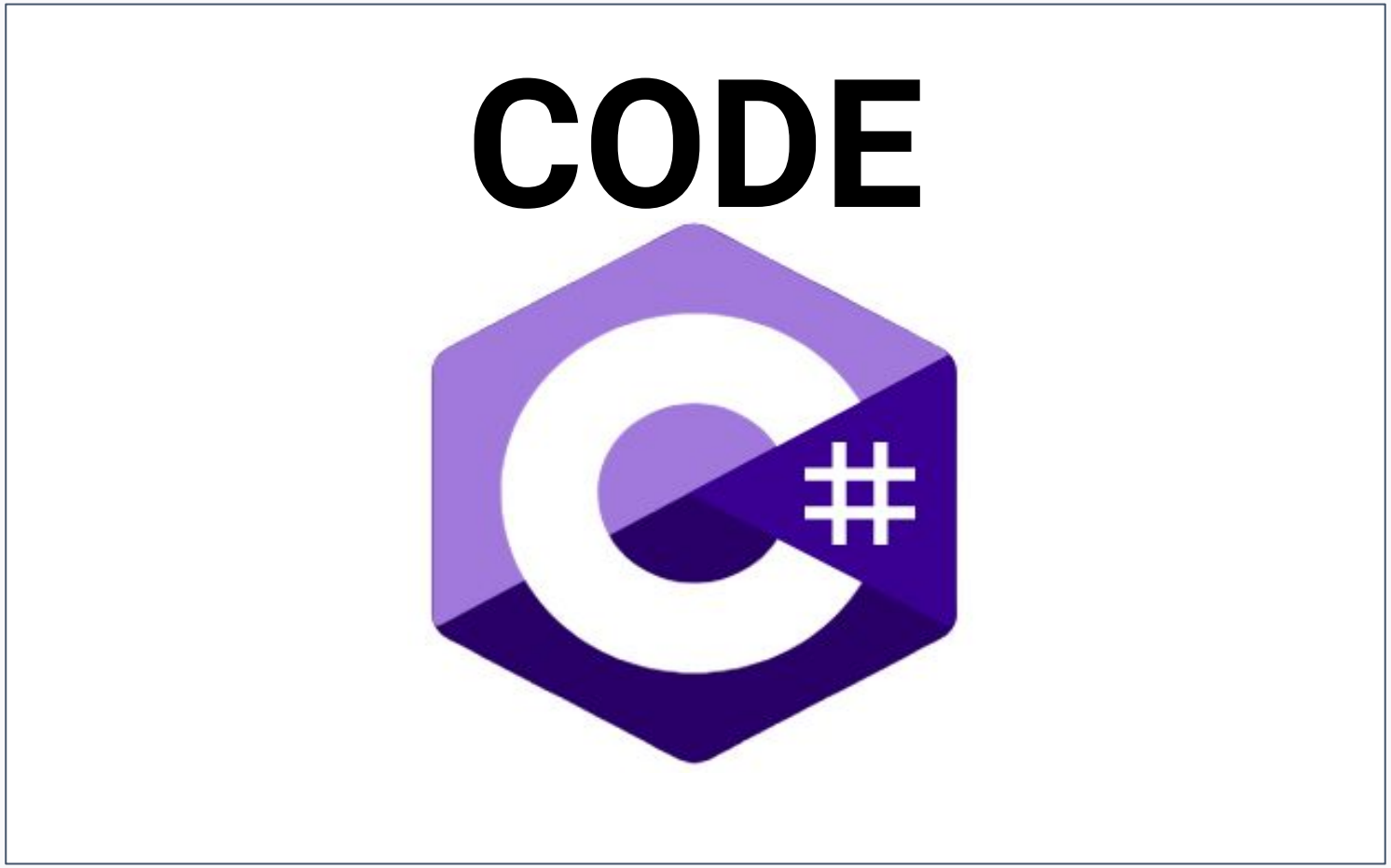
1. Очень популярен для ASP.NET MVC и имеет хорошую интеграцию с Core и документацию;
2. Есть регистрация модулей;
3. Есть инъекция в свойство;
4. Можно делать подконтейнеры;

Подключаем Autofac



https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Динамическое разрешение зависимостей через модули



https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Вопрос

Тайминг: 2 минуты

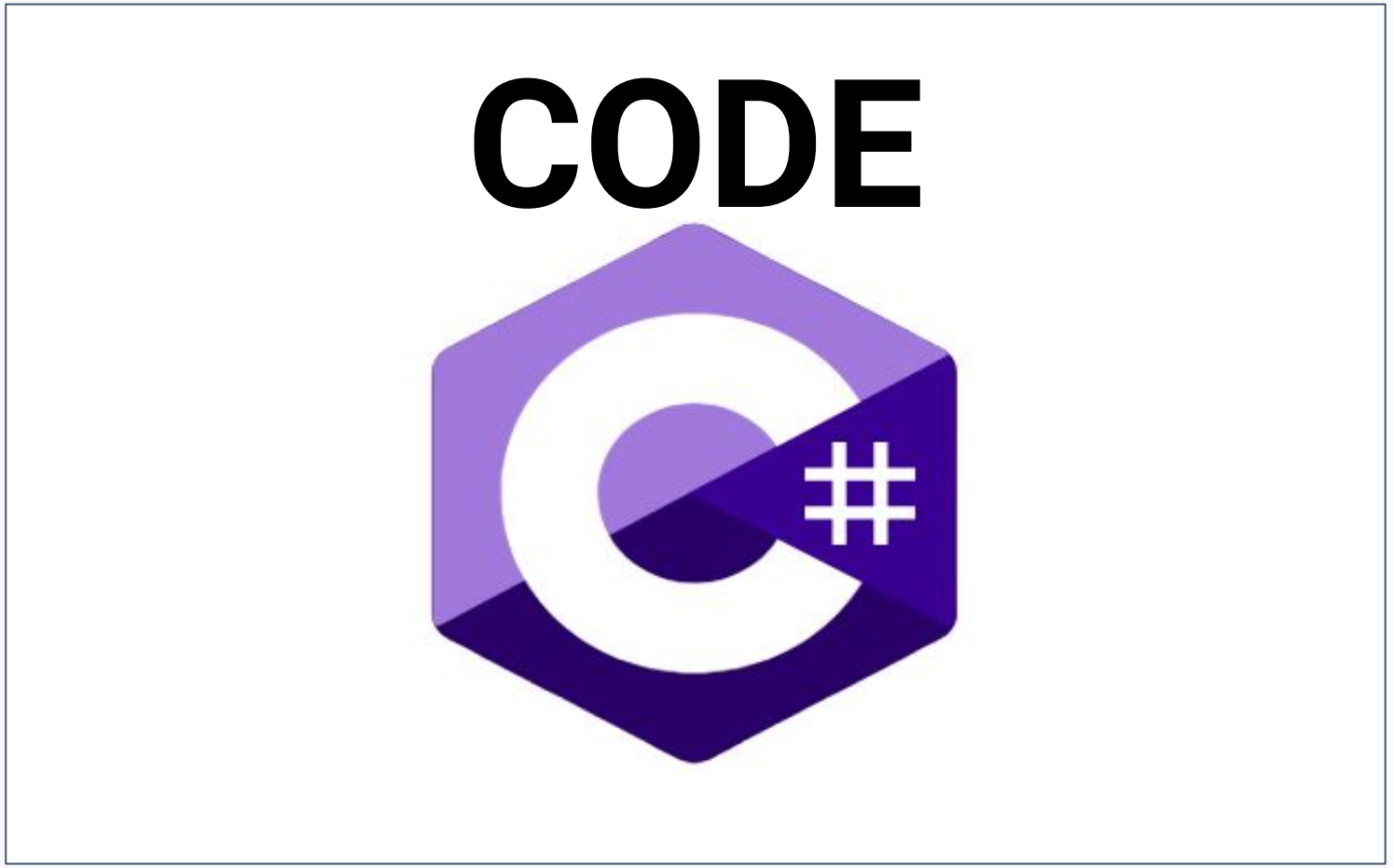
Зачем нам может понадобиться инъекция в свойство?

Напишите в чат или -, если надо пояснить

Инъекция в свойство

Может быть полезно если у нас есть базовый контроллер, который написали сами и контроллеры, от которых он наследует, чтобы не менять конструкторы всех наследуемых контроллеров можно какой-то параметр внедрить через свойство

Инъекция в свойство



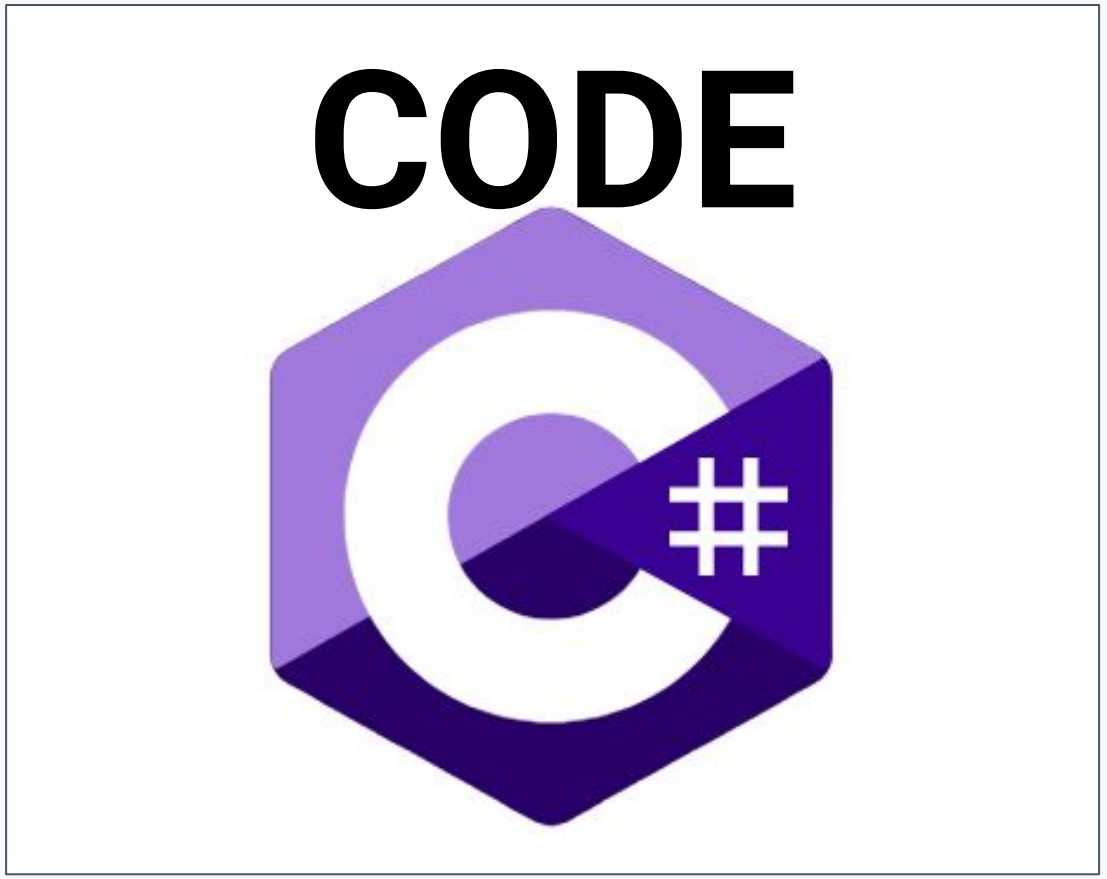
https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Динамическое разрешение зависимостей через фабрику

Если мы уже собрали контейнер, то в него просто так не добавить зависимости, допустим нужно выбрать реализацию в Runtime, либо вспоминаем про ServiceLocator, что не очень хорошо, либо пишем свою фабрику и через нее используем контейнер

Например, нужно построить дерево зависимостей по параметру запроса

Динамическое разрешение зависимостей через фабрику



Попробуйте по этим ссылкам настроить сами после занятия

https://stackoverflow.com/questions/54127414/using-factory-pattern-with-asp-net-core-dependency-injection

https://espressocoder.com/2018/10/08/injecting-a-factory-service-in-asp-net-core/

https://gitlab.com/devgrav/otus.teaching.promocodefactory.demo.di

Конфигурация по соглашению

На самостоятельную проработку

Чтобы не писать каждый раз Add каждого сервиса было бы удобно регистрировать по соглашению о наименовании, например, для всех классов, заканчивающихся на Service добавлять все одной строчкой или использующих Marker интерфейс

Конфигурация по соглашению

Можно использовать специальный контейнер вместо стандартного, но это может быть тяжелое решение, поэтому есть библиотека Scrutor, которая не вносит особых изменений в инфраструктуру, но добавляет фичи по динамической регистрации зависимостей

Это может быть актуально для CQRS подхода, где нам нужно регистрировать много обработчиков команд, также для этого полезна библиотека Mediatr, но это только часть ее назначения

Конфигурация по соглашению

Можно посмотреть в документации Autofac

https://autofaccn.readthedocs.io/en/latest/register/scanning.html

Scrutor

Удобное расширение для ASP.NET Core контейнера DI

https://github.com/khellang/Scrutor

https://andrewlock.net/using-scrutor-to-automatically-register-your-services-with-the-asp-net-core-di-container/

Выводы

1. Повторить преимущества DI/IOC принципа и основные возможности DI-контейнера для ASP.NET Core

2. Изучили жизненный цикл объектов в DI -контейнере

3. Изучить способы конфигурации нестандартных DI контейнеров и дополнительные инструменты

Список материалов для изучения

Внедрение зависимостей в .NET. Марк Симан
https://www.ozon.ru/context/detail/id/22104901/

Чистая архитектура. Роберт Мартин
https://www.ozon.ru/context/detail/id/144499396/


Заполните, пожалуйста,
опрос о занятии по ссылке
https://otus.ru/polls/15890/
Лучше всего написать что-то текстом!)

Спасибо за внимание!
Приходите на следующие вебинары

Гранковский Андрей

Архитектор направления
Альфа-Банк

https://www.linkedin.com/in/agrankovsky/