# Внедрение Зависимостей (Dependency Injection, DI) в ASP.NET Core: От Основ к Продвинутым Архитектурам

### Введение: Фундамент Современной Разработки на C#

Внедрение зависимостей (Dependency Injection, DI) представляет собой один из наиболее фундаментальных и широко используемых паттернов проектирования в современной разработке на платформе.NET. Интегрированный непосредственно в фреймворк ASP.NET Core, он обеспечивает механизм, который способствует созданию гибких, слабосвязанных и легко тестируемых приложений. В рамках данной лекции будет произведен глубокий анализ концепции DI, рассмотрены ее теоретические основы, практические аспекты применения, а также освещены лучшие практики и распространенные архитектурные ошибки. Материал предназначен как для начинающих разработчиков, стремящихся понять принципы построения масштабируемого кода, так и для опытных специалистов, желающих систематизировать свои знания и изучить продвинутые сценарии.

### Раздел 1: Теоретические Основы. Что такое DI и зачем оно нужно?

#### 1.1. Проблема жёстких зависимостей: new как "клей" в коде

До повсеместного распространения DI стандартной практикой было создание зависимостей внутри класса, использующего их, с помощью оператора new.1 Например, класс ReportService мог напрямую создавать экземпляр EmailService, необходимый для отправки отчетов. Этот подход, известный как сильная или "жесткая" связанность (tight coupling), приводит к ряду серьезных проблем.1

Представьте, что ReportService — это машина, а EmailService — ее деталь. Если каждая машина ReportService самостоятельно производит свою деталь (new EmailService()), то при необходимости заменить ее на новую версию (SendGridService) придется пересобрать каждую машину вручную. Это трудоемко и рискованно.1 Кроме того, такую машину невозможно протестировать без использования реальной детали, что делает модульное тестирование невозможным.2

#### 1.2. Принципы, лежащие в основе DI: IoC и DIP



Чтобы понять DI, нужно разобраться с двумя ключевыми принципами:

* **Принцип Инверсии Управления (IoC - Inversion of Control):** Это архитектурная концепция, которая переворачивает традиционный поток управления.3 Вместо того чтобы ваш класс (ReportService) сам создавал и управлял своими зависимостями (EmailService), он передает эту ответственность внешнему "контейнеру" или фреймворку. Контейнер берет на себя управление жизненным циклом объектов и "внедряет" их в ваши классы.
* **Принцип Инверсии Зависимостей (DIP - Dependency Inversion Principle):** Это один из пяти принципов SOLID. Он гласит, что высокоуровневые модули не должны зависеть от низкоуровневых. Вместо этого, и те, и другие должны зависеть от абстракций (например, интерфейсов).1 В C# это означает, что ваш  
  ReportService должен зависеть от интерфейса IEmailService, а не от конкретного класса SmtpEmailService.4

Таким образом, внедрение зависимостей (Dependency Injection, DI) — не просто инструмент, а фундаментальная часть архитектуры современных приложений на ASP.NET Core. Оно тесно связано с проектированием системы и невозможно без понимания ключевых принципов и паттернов. Разберёмся, как DI работает в .NET, какие принципы его лежат в основе и как правильно его использовать.

Можно выделить три ключевых аспекта:

1. Принципы проектирования — такие как *Inversion of Control (IoC)* и *Dependency Inversion Principle (DIP)*.
2. Паттерны — в первую очередь *Dependency Injection*.
3. Инструменты — встроенный контейнер DI в ASP.NET Core и сторонние IoC-контейнеры (например, Autofac, Microsoft.Extensions.DependencyInjection).

**Inversion of Control (IoC) — инверсия управления**

Inversion of Control (IoC) — это архитектурный принцип, при котором управление созданием и жизненным циклом объектов передаётся внешнему компоненту (например, контейнеру зависимостей), а не оставляется на усмотрение самого класса.

Вместо того чтобы класс сам создавал свои зависимости, он получает их извне — от фреймворка или контейнера. Это инвертирует традиционный поток управления: теперь не класс управляет зависимостями, а система управляет классом.

**Пример без IoC (жёсткая связь):**

|  |
| --- |
| public class UserService  {  private readonly AuthService \_authService;  public UserService()  {  \_authService = new AuthService(); // Жёсткая зависимость!  }  public void RegisterUser(string username, string password)  {  \_authService.Login(username, password);  }  } |

Такой подход делает класс негибким:

* Его невозможно протестировать без реальной реализации AuthService.
* Невозможно заменить реализацию (например, на мок для тестов).
* Класс привязан к конкретной реализации.

**Пример с IoC:**

|  |
| --- |
| public class UserService  {  private readonly IAuthService \_authService;  public UserService(IAuthService authService)  {  \_authService = authService; // Зависимость передаётся извне  }  public void RegisterUser(string username, string password)  {  \_authService.Login(username, password);  }  } |

Теперь UserService не знает, как создаётся IAuthService. Он просто использует абстракцию. Кто-то другой (например, контейнер DI) отвечает за создание и передачу нужной реализации.

**Dependency Inversion Principle (DIP) — принцип инверсии зависимостей**

DIP — один из пяти принципов SOLID, и он напрямую связан с IoC и DI.

Высокоуровневые модули не должны зависеть от низкоуровневых. Оба должны зависеть от абстракций.

Это означает:

* Классы верхнего уровня (например, сервисы) не должны напрямую зависеть от конкретных реализаций.
* Вместо этого они должны зависеть от интерфейсов (абстракций).
* Реализации (низкоуровневые классы) также должны реализовывать эти интерфейсы.

**Пример без DIP (жёсткая связь):**

|  |
| --- |
| public class UserService  {  private readonly EmailService \_emailService;  public UserService(EmailService emailService)  {  \_emailService = emailService;  }  public void SendWelcomeMessage(string email)  {  \_emailService.Send(email, "Welcome!");  }  } |

Здесь UserService напрямую зависит от EmailService. Если мы захотим использовать SmsService или PushNotificationService, придётся переписывать UserService.

**Пример с DIP:**

|  |
| --- |
| public interface IMessageService  {  void Send(string destination, string message);  }  public class EmailService : IMessageService  {  public void Send(string destination, string message)  {  // Отправка по email  }  }  public class UserService  {  private readonly IMessageService \_messageService;  public UserService(IMessageService messageService)  {  \_messageService = messageService;  }  public void SendWelcomeMessage(string email)  {  \_messageService.Send(email, "Welcome!");  }  } |

Теперь UserService зависит только от интерфейса IMessageService. Мы можем легко подменить реализацию — например, использовать SmsService или MockMessageService в тестах.

**Так в чём разница между IoC и DIP?**

|  |  |
| --- | --- |
|  |  |
| Архитектурный принцип, при котором управление объектами передаётся внешнему контейнеру. | Принцип проектирования, утверждающий, что модули должны зависеть от абстракций, а не от конкретик. |
| Этообщая концепция— DI, Service Locator, Template Method и др. являются её реализациями. | Эточастный случай IoC, применяемый к зависимостям. |
| Отвечает на вопрос:*«Кто создаёт объекты?»* | Отвечает на вопрос:*«От чего зависит класс?»* |

DI — это способ реализации IoC, а DIP — принцип, который делает DI осмысленным.

#### 1.3. DI как практический паттерн

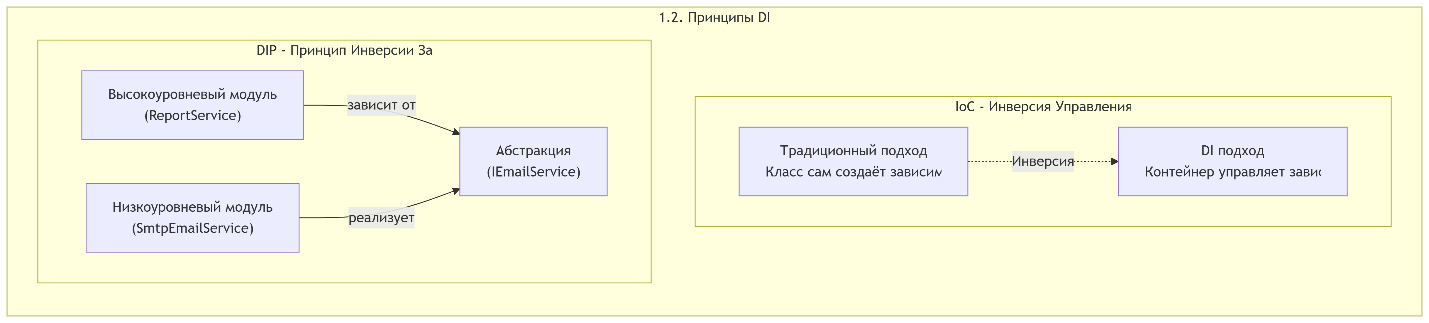
Внедрение зависимостей (DI) является конкретным паттерном, который реализует принципы IoC и DIP. Его суть в том, что зависимости класса предоставляются ему извне, обычно через конструктор.5

Паттерн DI состоит из четырех основных ролей 6:

1. **Клиент (Client):** Класс, которому нужна зависимость. Например, ReportService.
2. **Интерфейс сервиса (Service-Interface):** Абстракция, которая определяет контракт. Например, IEmailService.
3. **Сервис (Service):** Конкретная реализация интерфейса. Например, SmtpEmailService.
4. **Инжектор (Injector):** Компонент, который связывает Клиента и Сервис, создавая и передавая экземпляр Сервиса в Клиент. В ASP.NET Core эту роль выполняет **DI-контейнер**.8

Процесс DI состоит из трех шагов:

1. **Абстракция:** Вы определяете зависимость через интерфейс, например IEmailService.
2. **Регистрация:** В центральном месте (Program.cs) вы говорите контейнеру: "Когда кто-то попросит IEmailService, дай ему SmtpEmailService". builder.Services.AddScoped<IEmailService, SmtpEmailService>();6
3. **Внедрение:** Ваш класс-клиент (ReportService) объявляет в конструкторе, что ему нужен IEmailService. Контейнер автоматически создает экземпляр SmtpEmailService и передает его в конструктор ReportService.9



#### 1.4. Типы внедрения зависимостей

Существуют три основных способа внедрения зависимостей 9:

* **Внедрение через конструктор (Constructor Injection):** Самый распространенный и рекомендуемый способ в ASP.NET Core. Зависимости передаются как параметры в конструктор класса.9 Этот подход делает зависимости явными, что улучшает читаемость и надежность кода.4 **Пример.** builder.Services.AddScoped<IEmailService, EmailService>();

Класс, использующий внедрение:

|  |
| --- |
| public class OrderService  {  private readonly IEmailService \_emailService;  public OrderService(IEmailService emailService)  {  \_emailService = emailService;  }  public void ProcessOrder()  {  // Логика обработки заказа  \_emailService.Send("Заказ выполнен.");  }  } |

✅ Преимущества: зависимости обязательны, легко тестировать, явная сигнатура.

* **Внедрение через метод (Method Injection):** Зависимости передаются через публичный метод. Этот подход используется реже, например, в Minimal APIs.4

Пример (Minimal API):

|  |
| --- |
| var builder = WebApplication.CreateBuilder(args);  // Регистрация сервиса  builder.Services.AddSingleton<NotificationService>();  var app = builder.Build();  // Зависимость автоматически внедряется через параметр метода  app.MapGet("/notify/{id}", (string id, NotificationService service) =>  {  service.NotifyUser(id, "Ваш запрос обработан.");  return Results.Ok();  });  app.Run(); |

⚠️ Важно: Такое внедрение работает только для обработчиков маршрутов в Minimal API. Не рекомендуется использовать для бизнес-логики вне контекста HTTP-запроса.

* **Внедрение через свойство (Property Injection):** Зависимости присваиваются публичному свойству. Встроенный DI-контейнер ASP.NET Core не поддерживает этот способ "из коробки", так как он скрывает зависимости и усложняет тестирование. Пример Регистрация и использование (вручную или с помощью стороннего DI):

Пример (ручное внедрение — не рекомендуется):

|  |
| --- |
| var emailService = new EmailService();  var reportService = new ReportService();  reportService.EmailService = emailService; // Внедрение через свойство  reportService.GenerateReport(); |

❌ Проблемы: зависимость необязательная, может быть не установлена → риск NullReferenceException.

|  |
| --- |
| var builder = new ContainerBuilder();  builder.RegisterType<EmailService>().As<IEmailService>();  builder.RegisterType<ReportService>().PropertiesAutowired();  var container = builder.Build(); |

✅ Autofac позволяет включить PropertiesAutowired(), но даже там рекомендуется отдавать приоритет внедрению через конструктор.

Рекомендации

✅ Используй внедрение через конструктор — это стандарт для ASP.NET Core.

⚠️ Внедрение через метод — допустимо в Minimal API, но не для внутренних классов.

❌ Внедрение через свойство — избегай его, особенно с встроенным DI-контейнером. Используй только при крайней необходимости (например, совместимость с устаревшим кодом).

#### 1.5. Архитектурные преимущества

Использование DI влечет за собой ряд значительных архитектурных преимуществ 6:

* **Слабая связанность (Loose Coupling):** Классы зависят от абстракций, а не от конкретных реализаций, что позволяет легко заменять одни реализации на другие.1
* **Тестируемость (Testability):** Возможность подмены реальных зависимостей на "моки" (mocks) или "стабы" (stubs) значительно упрощает модульное тестирование, без необходимости использования реальных баз данных или внешних сервисов.
* **Масштабируемость и гибкость:** Слабая связанность упрощает добавление новых функциональностей, что делает архитектуру более адаптивной к будущим изменениям.6
* **Параллельная разработка:** Поскольку интерфейс четко определяет границы, разные команды могут параллельно разрабатывать Клиента и Сервис, не мешая друг другу.

#### 1.6. Роль DI в архитектурных паттернах

DI является краеугольным камнем таких архитектурных паттернов, как **Clean Architecture** и **Onion Architecture**.12 Эти архитектуры стремятся создать ядро приложения, полностью независимое от внешних слоев, таких как база данных или веб-сервисы.12 DI-контейнер, находящийся в самом внешнем слое, "связывает" компоненты во время выполнения, внедряя конкретные реализации в ядро приложения.12

### Раздел 2: Время Жизни Сервисов (Service Lifetimes)

"Время жизни" (service lifetime) определяет, как долго экземпляр сервиса будет существовать после его создания и будет ли он использоваться повторно.5 Встроенный DI-контейнер ASP.NET Core предоставляет три основных варианта времени жизни:

Transient, Scoped и Singleton.5

Давайте рассмотрим их на простом примере: у нас есть сервис CounterService, который считает, сколько раз был запрошен его экземпляр.

public interface ICounter  
{  
 int GetCount();  
}  
  
public class CounterService : ICounter  
{  
 private int \_count;  
 public CounterService()  
 {  
 \_count = new Random().Next(1000); // Случайное число для демонстрации нового экземпляра  
 }  
 public int GetCount() => \_count;  
}

#### 2.1. Transient — Новый экземпляр на каждый запрос

Сервис с временем жизни Transient создает новый экземпляр каждый раз, когда он запрашивается из DI-контейнера. Даже в рамках одного HTTP-запроса, если сервис запрашивается дважды, будут созданы два разных экземпляра.

* **Когда использовать:** Для легковесных, безстатусных сервисов, которым не требуется хранить информацию между вызовами, например, для утилит или хэшеров паролей.
* **Регистрация:** builder.Services.AddTransient<ICounter, CounterService>();  
  14
* **Что происходит:**
  + **Запрос №1:** В одном и том же контроллере мы запрашиваем ICounter дважды.
    - controller.Counter1 получает CounterService с ID 123.
    - controller.Counter2 получает CounterService с ID 456.
    - В рамках одного запроса создаются два разных объекта.
  + **Запрос №2:** Приходит новый запрос, и мы снова запрашиваем ICounter.
    - controller.Counter получает CounterService с ID 789.
    - Создается новый объект, не связанный с предыдущими.

#### 2.2. Scoped — Один экземпляр на "область видимости"

Scoped-сервис создается один раз **на область видимости** (scope) и повторно используется внутри этой области.5 В контексте веб-приложения, областью видимости является **один HTTP-запрос**.5 Это означает, что один и тот же экземпляр будет доступен всем компонентам, которые участвуют в обработке данного запроса (контроллерам, сервисам, репозиториям).5

Scoped является наиболее распространенным временем жизни для веб-приложений.14

* **Когда использовать:** Для сервисов, которые должны поддерживать единое состояние в рамках одного запроса. Идеально подходит для DbContext из Entity Framework Core, который является "единицей работы" (Unit of Work) и должен быть один на каждый запрос.15
* **Регистрация:** builder.Services.AddScoped<ICounter, CounterService>();  
  14
* **Что происходит:**
  + **Запрос №1:** В одном и том же контроллере мы запрашиваем ICounter дважды.
    - controller.Counter1 получает CounterService с ID 123.
    - controller.Counter2 получает **тот же самый** CounterService с ID 123.
    - В рамках одного запроса используется один и тот же объект.
  + **Запрос №2:** Приходит новый запрос.
    - controller.Counter получает CounterService с ID 456.
    - Создается новый объект, который будет использоваться в течение этого нового запроса.

#### 2.3. Singleton — Один экземпляр на всё приложение

Singleton-сервис создается только один раз за всю жизнь приложения — при первом запросе или при запуске приложения — и затем повторно используется всеми последующими запросами и компонентами.5

* **Когда использовать:** Для сервисов, которые :
  + Дороги в создании (например, кэш-сервисы, пулы соединений).
  + Должны хранить глобальное состояние, доступное всем частям приложения.
  + **Обязательно** должны быть **потокобезопасными** (thread-safe), поскольку к ним могут обращаться одновременно из разных потоков.5
* **Регистрация:** builder.Services.AddSingleton<ICounter, CounterService>();  
  14
* **Что происходит:**
  + **Запрос №1:** controller.Counter получает CounterService с ID 123.
  + **Запрос №2:** controller.Counter получает **тот же самый** CounterService с ID 123.
  + Независимо от количества запросов, всегда используется один и тот же экземпляр.

#### 2.4. Детальное сравнение и примеры поведения

|  |  |  |  |
| --- | --- | --- | --- |
| Критерий | Transient | Scoped | Singleton |
| Создание объекта | При каждом запросе сервиса | Один раз на каждый scope (например, HTTP-запрос) | Один раз за всё время жизни приложения |
| Совместное использование | Никогда — каждый получает новый экземпляр | Общие в пределах одного scope (запроса) | Общие между всеми запросами и потоками |
| Типичное использование | Лёгкие, безымянные, без состояния сервисы (например, утилиты) | Сервисы, привязанные к запросу (например,DbContext) | Глобальные, потокобезопасные сервисы (логгеры, кэш, конфигурация) |
| Потокобезопасность | Не требуется (короткоживущие экземпляры) | Желательна при асинхронности, но обычно не критична | Обязательна (один экземпляр на все потоки) |
| Когда использовать | По умолчанию, если нет особых требований | Для большинства сервисов в веб-приложениях |  |

**Реализация DI в VS Code**

Для того чтобы реализовать теорию о времени жизни сервисов, вам нужно создать простой веб-проект на ASP.NET Core в Visual Studio Code.

**Шаг 1: Создание проекта**

1. **Откройте терминал в VS Code**. Вы можете использовать встроенный терминал, нажав Ctrl + ` (обратная кавычка).
2. **Создайте новый проект веб-приложения**. Введите следующую команду:

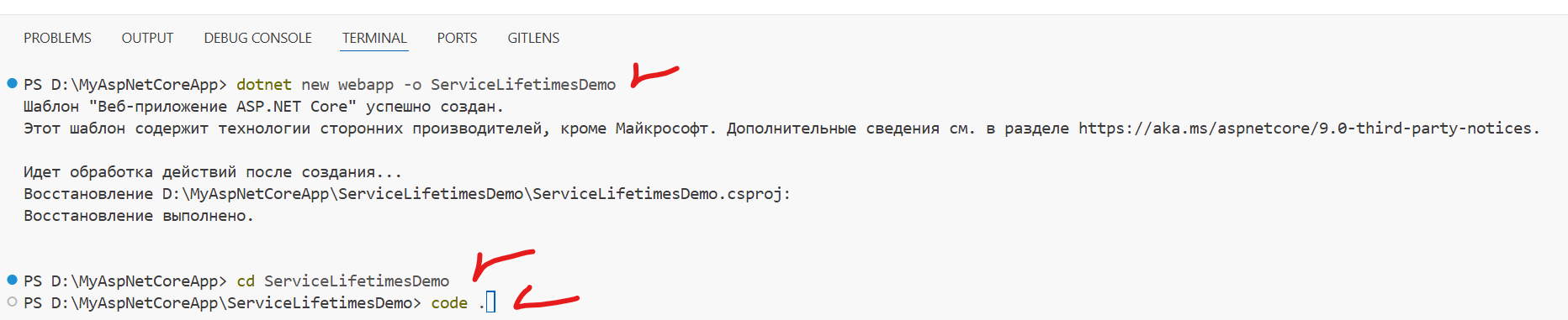
dotnet new webapp -o ServiceLifetimesDemo

cd ServiceLifetimesDemo

Эта команда создаст новый проект с именем ServiceLifetimesDemo и перейдет в его директорию.

1. **Откройте папку проекта в VS Code**.

code .



**Шаг 2: Структура проекта**

Наш проект будет состоять из трех основных частей:

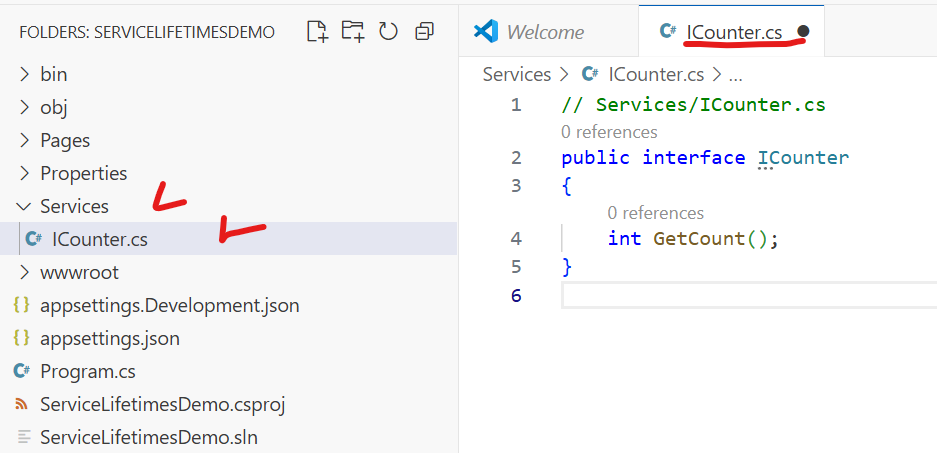
* **Интерфейс** ICounter и его **реализация** CounterService.
* **Контроллер** HomeController, который будет использовать наш сервис.
* **Файл конфигурации** Program.cs, где мы будем регистрировать сервисы с разным временем жизни.

**Шаг 3: Реализация кода**

**3.1. Создание интерфейса и сервиса**

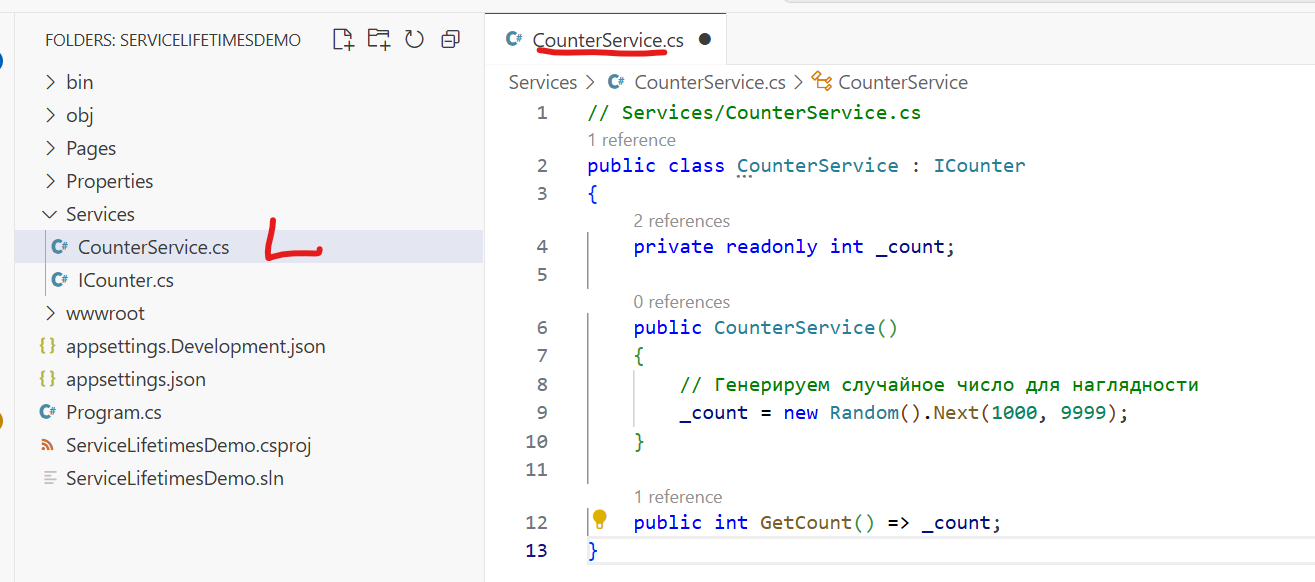
1. Создайте новую папку в корне проекта с именем Services.
2. Внутри папки Services создайте файл ICounter.cs и добавьте следующий код:

|  |
| --- |
| // Services/ICounter.cs  public interface ICounter  {  int GetCount();  } |



1. В той же папке создайте файл CounterService.cs и добавьте его реализацию:

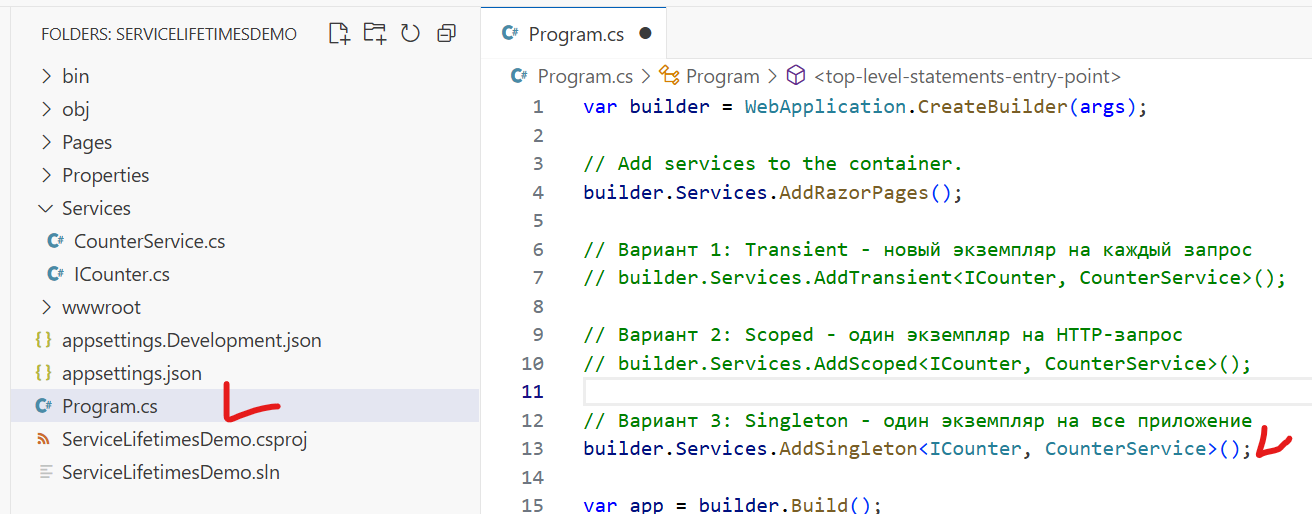
|  |
| --- |
| // Services/CounterService.cs  public class CounterService : ICounter  {  private readonly int \_count;  public CounterService()  {  // Генерируем случайное число для наглядности  \_count = new Random().Next(1000, 9999);  }  public int GetCount() => \_count;  } |



**3.2. Настройка Program.cs**

1. Откройте файл Program.cs. Это файл, где происходит настройка и регистрация всех сервисов.
2. В секции регистрации сервисов (builder.Services) добавьте один из следующих вариантов, чтобы протестировать каждое время жизни. Вы можете закомментировать или раскомментировать нужный вариант.

|  |
| --- |
| // Вариант 1: Transient - новый экземпляр на каждый запрос  // builder.Services.AddTransient<ICounter, CounterService>();  // Вариант 2: Scoped - один экземпляр на HTTP-запрос  // builder.Services.AddScoped<ICounter, CounterService>();  // Вариант 3: Singleton - один экземпляр на все приложение  builder.Services.AddSingleton<ICounter, CounterService>(); |

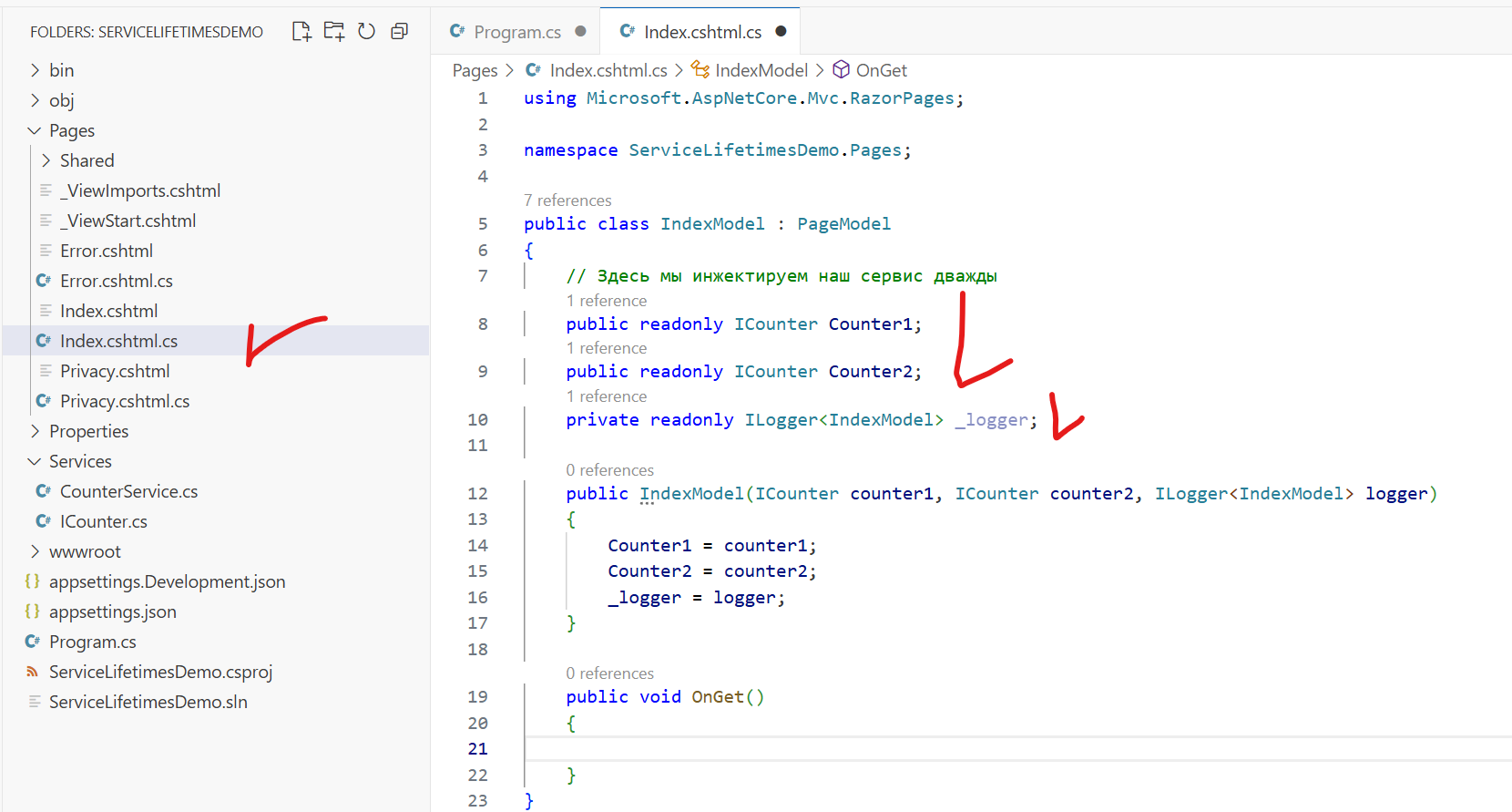


**3.3. Создание Razor Page для отображения результата**

Мы будем использовать Razor Page для простоты, так как они уже настроены в проекте по умолчанию.

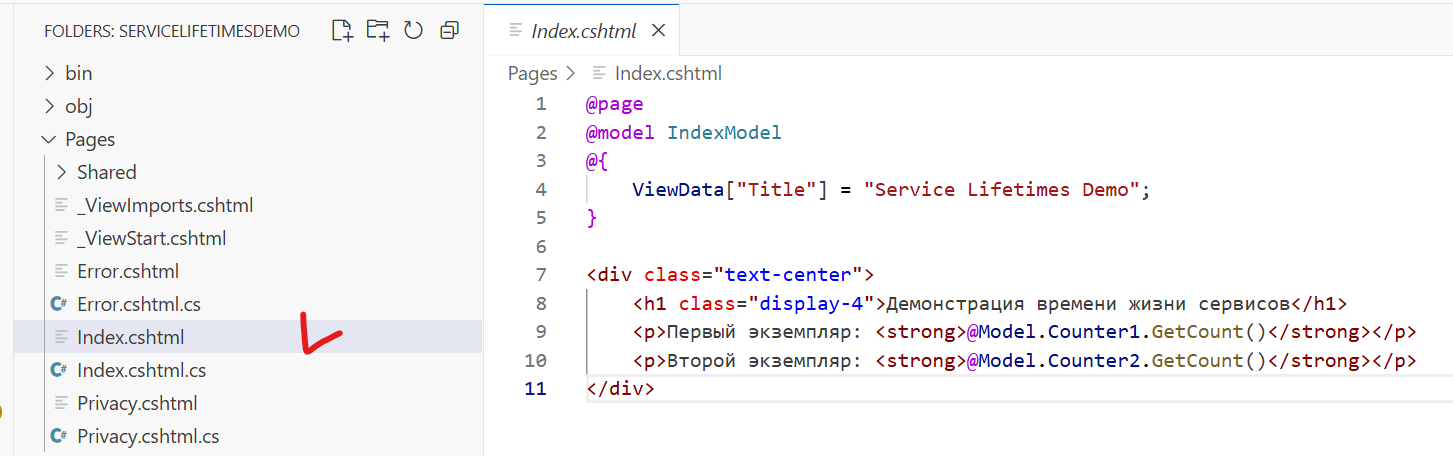
1. Откройте файл Pages/Index.cshtml.cs. Это модель страницы.
2. Внутри класса IndexModel добавьте два поля для нашего сервиса и их внедрение через конструктор:

|  |
| --- |
| using Microsoft.AspNetCore.Mvc.RazorPages;  namespace ServiceLifetimesDemo.Pages;  public class IndexModel : PageModel  {      // Здесь мы инжектируем наш сервис дважды      public readonly ICounter Counter1;      public readonly ICounter Counter2;      private readonly ILogger<IndexModel> \_logger;      public IndexModel(ICounter counter1, ICounter counter2, ILogger<IndexModel> logger)      {          Counter1 = counter1;          Counter2 = counter2;          \_logger = logger;      }      public void OnGet()      {      }  } |



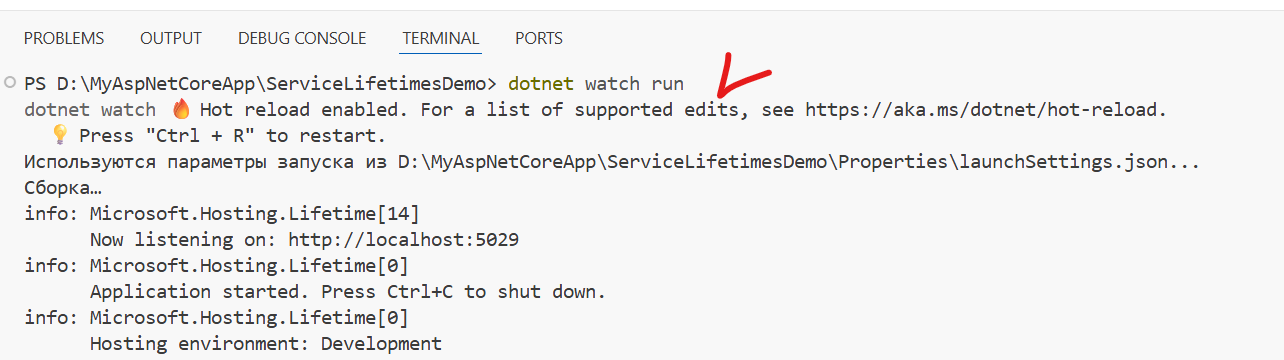
1. Откройте файл Pages/Index.cshtml. Это представление страницы.
2. Замените содержимое на следующий код, чтобы отобразить значения счетчиков:

|  |
| --- |
| @page  @model IndexModel  @{      ViewData["Title"] = "Service Lifetimes Demo";  }  <div class="text-center">      <h1 class="display-4">Демонстрация времени жизни сервисов</h1>      <p>Первый экземпляр: <strong>@Model.Counter1.GetCount()</strong></p>      <p>Второй экземпляр: <strong>@Model.Counter2.GetCount()</strong></p>  </div> |



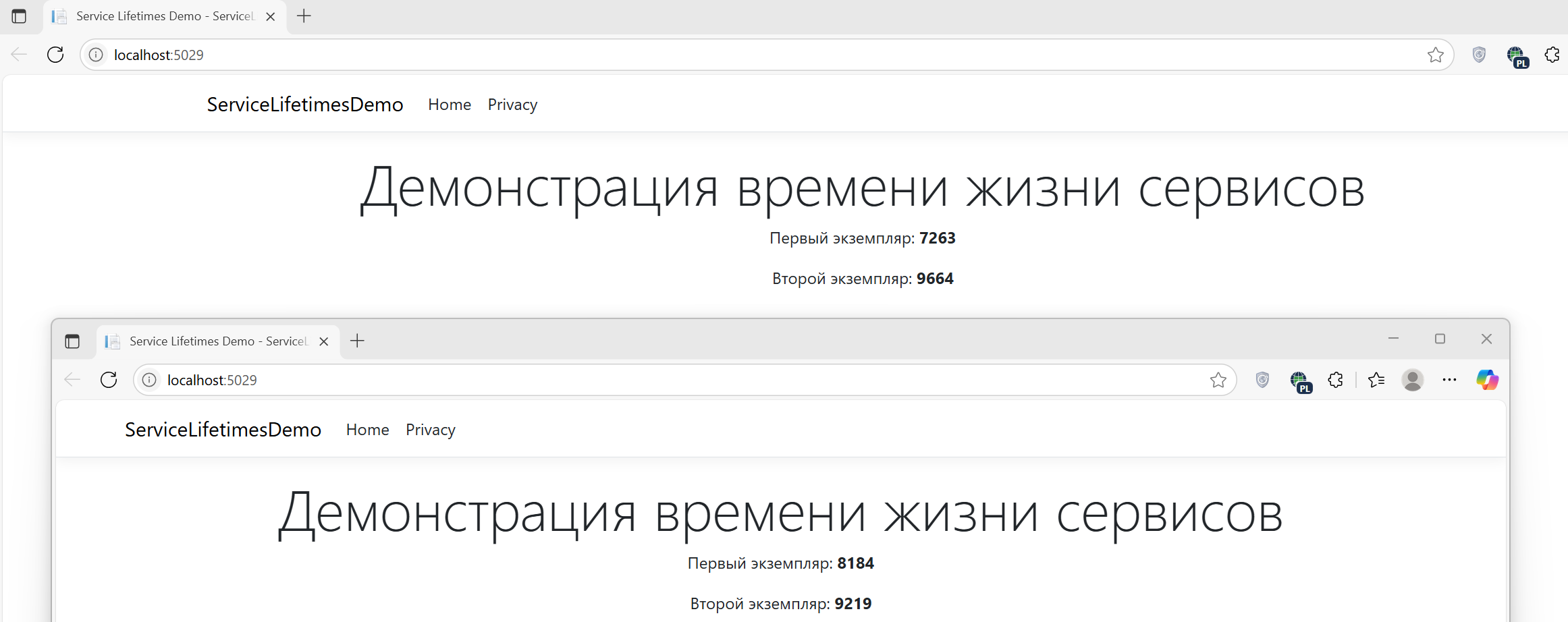
**Шаг 4: Запуск и тестирование**

1. Сохраните все файлы.
2. В терминале VS Code введите команду для запуска приложения:
3. dotnet watch run

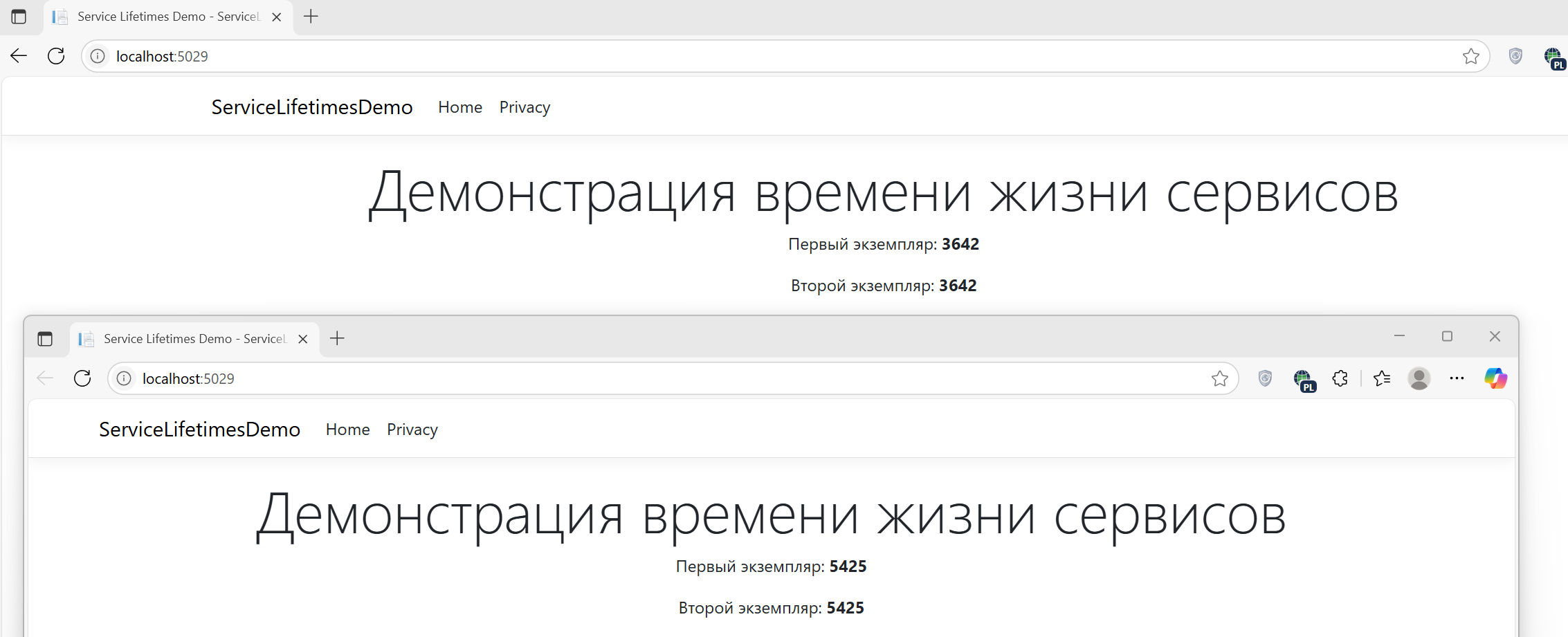


Команда watch будет автоматически перезапускать приложение при каждом изменении кода.

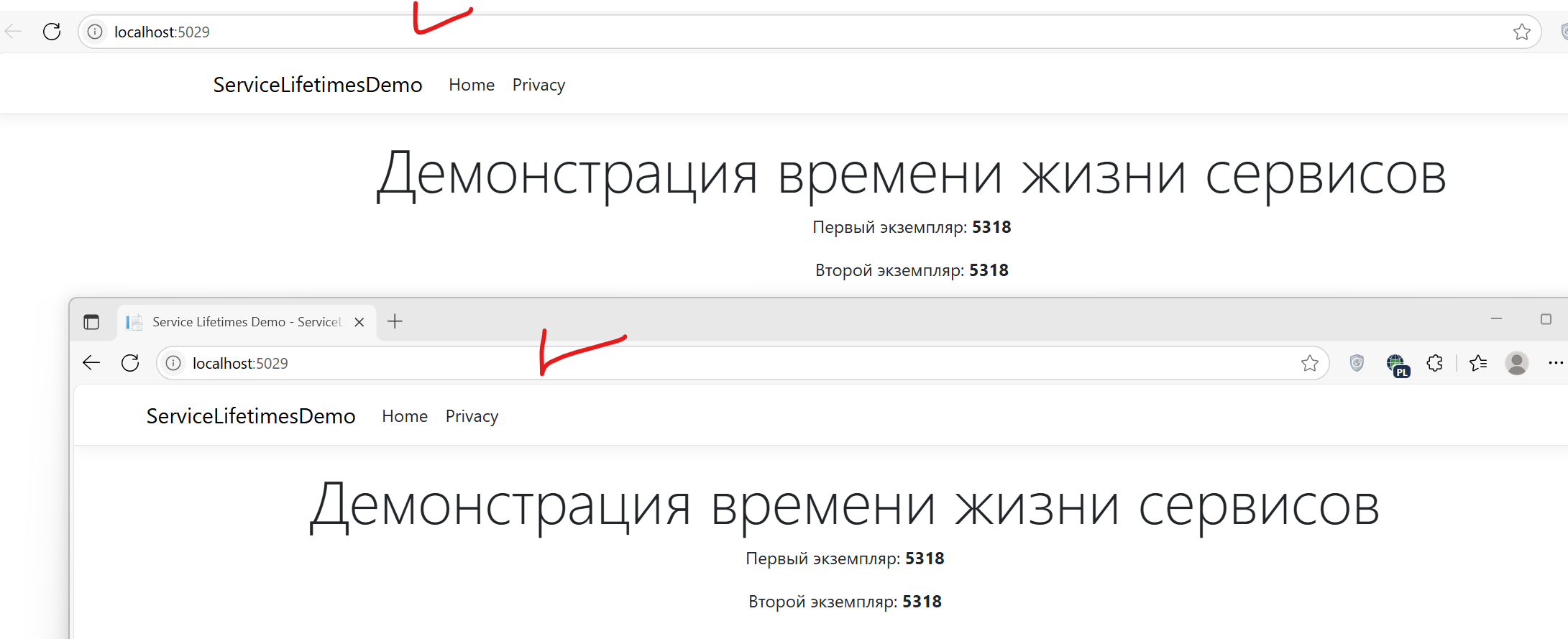
1. Перейдите в браузер по адресу https://localhost:5001 (или по адресу, указанному в терминале).
2. **Обновите страницу несколько раз**, и вы увидите, как ведут себя числа:
   * **Transient:** При каждом обновлении страницы и даже при каждом внедрении в одном и том же запросе, числа будут **разными**.



* + **Scoped:** При каждом обновлении страницы число будет **новым**, но в рамках одного запроса (т.е., на одной странице) оно будет **одинаковым**.



* + **Singleton:** Независимо от обновлений и запросов, число всегда будет **одним и тем же**.



Эта пошаговая инструкция позволит вам на практике увидеть, как работают разные "времена жизни" сервисов. Поэкспериментируйте, раскомментируя разные варианты в Program.cs, чтобы полностью понять разницу.

### Раздел 3: Регистрация и Использование Зависимостей

#### 3.1. Центральная точка: Конфигурация в Program.cs (Startup.cs)

В современных приложениях ASP.NET Core, построенных на.NET 6 и выше, регистрация всех сервисов происходит в файле Program.cs.5 Сервисы добавляются в коллекцию builder.Services с помощью методов расширения, таких как AddTransient, AddScoped и AddSingleton. В более ранних версиях.NET (до.NET 6) эта конфигурация выполнялась в методе ConfigureServices класса Startup.cs.

#### 3.2. Стандартный подход: Внедрение через конструктор

Внедрение через конструктор — это основной и наиболее рекомендуемый способ получения зависимостей в ASP.NET Core.6 Он заключается в том, что все необходимые сервисы объявляются в качестве параметров конструктора класса.9

**Пример:**

public interface IEmailSender  
{  
 void SendEmail(string to, string subject);  
}  
  
public class SmtpEmailSender : IEmailSender  
{  
 public void SendEmail(string to, string subject)  
 {  
 Console.WriteLine($"Отправка письма через SMTP: {to}");  
 }  
}  
  
// Регистрация в Program.cs  
builder.Services.AddScoped<IEmailSender, SmtpEmailSender>();  
  
// Внедрение в конструктор контроллера  
public class UserController : ControllerBase  
{  
 private readonly IEmailSender \_emailSender;  
  
 // Конструктор, через который внедряется зависимость  
 public UserController(IEmailSender emailSender)  
 {  
 \_emailSender = emailSender;  
 }  
  
 [HttpGet]  
 public IActionResult SendTestEmail()  
 {  
 \_emailSender.SendEmail("test@example.com", "Тестовое письмо");  
 return Ok();  
 }  
}

#### 3.3. Примеры регистрации и внедрения

##### 3.3.1. Регистрация простого класса без интерфейса

Вы можете зарегистрировать класс напрямую, без использования интерфейса. Это возможно, но не рекомендуется, поскольку такой подход снижает гибкость и делает код более связанным.

// 1. Простой класс  
public class NotificationService  
{  
 public void Send(string message)  
 {  
 Console.WriteLine($"Уведомление отправлено: {message}");  
 }  
}  
  
// 2. Регистрация в Program.cs  
builder.Services.AddScoped<NotificationService>();  
  
// 3. Внедрение в контроллер (не рекомендуется)  
public class HomeController : ControllerBase  
{  
 private readonly NotificationService \_notificationService;  
  
 public HomeController(NotificationService notificationService)  
 {  
 \_notificationService = notificationService;  
 }  
}

##### 3.3.2. Регистрация через интерфейс (предпочтительный способ)

Это стандартный и рекомендуемый подход, который позволяет вашему коду зависеть от абстракции (интерфейса), а не от конкретной реализации.16 Это значительно упрощает замену реализаций и тестирование.9

// 1. Абстракция и конкретная реализация  
public interface INotificationService  
{  
 void Send(string message);  
}  
  
public class EmailNotificationService : INotificationService  
{  
 public void Send(string message)  
 {  
 Console.WriteLine($"Отправка email: {message}");  
 }  
}  
  
// 2. Регистрация в Program.cs  
builder.Services.AddScoped<INotificationService, EmailNotificationService>();  
  
// 3. Внедрение в контроллер  
public class HomeController : ControllerBase  
{  
 private readonly INotificationService \_notificationService;  
  
 public HomeController(INotificationService notificationService)  
 {  
 \_notificationService = notificationService;  
 }  
}

##### 3.3.3. Множественные реализации одного интерфейса

Когда у одного интерфейса есть несколько реализаций, по умолчанию DI-контейнер будет использовать последнюю зарегистрированную. Для динамического выбора или использования всех реализаций можно применять следующие подходы.

Для нашего примера мы будем использовать один и тот же интерфейс и две его реализации. Сначала создадим проект в VS Code с названием ServiceLifetimesDemo. Для этого в терминале вводим команду:

dotnet new webapp -o ServiceLifetimesDemo

cd ServiceLifetimesDemo

Все последующие примеры кода будут реализовываться внутри этого проекта.

**Шаг 1: Подготовка интерфейса и реализаций**

Создайте интерфейс IExternalLogger и два класса, которые его реализуют: FileLogger и DatabaseLogger.

|  |
| --- |
| // Services/IExternalLogger.cs  public interface IExternalLogger  {  void Log(string message);  }  // Services/FileLogger.cs  public class FileLogger : IExternalLogger  {  public void Log(string message)  {  Console.WriteLine($"FileLogger: Запись в файл - {message}");  }  }  // Services/DatabaseLogger.cs  public class DatabaseLogger : IExternalLogger  {  public void Log(string message)  {  Console.WriteLine($"DatabaseLogger: Запись в базу данных - {message}");  }  } |

**1. Использование IEnumerable<T>**

Этот подход позволяет получить **коллекцию всех зарегистрированных реализаций** интерфейса. Он идеально подходит, когда вам нужно применить одно и то же действие ко всем доступным сервисам, например, отправить одно сообщение всем подписчикам или выполнить несколько этапов обработки данных.

**Регистрация в Program.cs**

Обе реализации регистрируются в контейнере служб с использованием метода AddScoped<TService, TImplementation>(), что обеспечивает их создание в пределах одного HTTP-запроса:

// Program.cs

builder.Services.AddScoped<IExternalLogger, FileLogger>();

builder.Services.AddScoped<IExternalLogger, DatabaseLogger>();

**Использование в сервисе**

При регистрации нескольких реализаций одного интерфейса контейнер DI сохраняет их в виде упорядоченной коллекции. При запросе экземпляра типа IEnumerable<IExternalLogger> контейнер возвращает перечисление, содержащее все зарегистрированные реализации, в порядке их регистрации.

Таким образом, возможность получения коллекции реализаций через IEnumerable<T> позволяет реализовать паттерн массового оповещения (broadcasting), когда одно и то же действие применяется ко всем зарегистрированным обработчикам.

4. Сервисная оркестрация: класс MyService

Для централизованного управления вызовом методов на всех реализациях интерфейса IExternalLogger вводится посреднический класс — MyService. Данный класс инкапсулирует логику итерации по коллекции зарегистрированных логгеров.

// Services/MyService.cs

using System.Collections.Generic;

public class MyService

{

private readonly IEnumerable<IExternalLogger> \_loggers;

public MyService(IEnumerable<IExternalLogger> loggers)

{

\_loggers = loggers;

}

public void LogToAll(string message)

{

foreach (var logger in \_loggers)

{

logger.Log(message);

}

}

}

Конструктор класса принимает IEnumerable<IExternalLogger>, что инициирует разрешение всех зарегистрированных в DI-контейнере реализаций. Метод LogToAll последовательно вызывает метод Log для каждого элемента коллекции. Не забудь зарегистрировать MyService как сервис:

builder.Services.AddScoped<MyService>();

Теперь Program.cs будет содержать:

builder.Services.AddScoped<IExternalLogger, FileLogger>();

builder.Services.AddScoped<IExternalLogger, DatabaseLogger>();

builder.Services.AddScoped<MyService>();

Шаг 5: Использование в обработчике (например, в IndexModel)

Открой Pages/Index.cshtml.cs и внедри MyService:

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace ServiceLifetimesDemo.Pages;

public class IndexModel : PageModel

{

    private readonly ILogger<IndexModel> \_logger;

    private readonly MyService \_myService;

    public IndexModel(ILogger<IndexModel> logger, MyService myService)

    {

        \_logger = logger;

        \_myService = myService;

    }

    public void OnGet()

    {

        \_myService.LogToAll("Тестовое сообщение при загрузке страницы");

    }

}

Шаг 6: Запуск проекта dotnet run и **получим**

FileLogger: Запись в файл - Тестовое сообщение при загрузке страницы

DatabaseLogger: Запись в базу данных - Тестовое сообщение при загрузке страницы

Следует отметить, что попытка внедрить IExternalLogger как единичный сервис (не в виде IEnumerable<T>) приведёт к исключению времени выполнения:

InvalidOperationException: Unable to resolve service for type 'IExternalLogger' while attempting to activate...

Это обусловлено тем, что контейнер DI не может однозначно определить, какая из реализаций должна быть выбрана при отсутствии дополнительных дескрипторов (например, ключей или условий). Следовательно, механизм IEnumerable<T> является единственным корректным способом получения всех зарегистрированных реализаций одного интерфейса в стандартной реализации DI .NET.

6. Практическое применение и преимущества

Представленный подход обеспечивает:

* Гибкость архитектуры — новые реализации интерфейса могут быть добавлены без изменения клиентского кода.
* Соблюдение принципа открытости/закрытости (OCP) — система открыта для расширения, но закрыта для модификации.
* Снижение связанности — клиентский код зависит от абстракций, а не от конкретных реализаций.

Типичные сценарии применения:

* Множественная доставка уведомлений (email, SMS, push).
* Параллельная валидация данных через независимые валидаторы.
* Аудит и логирование в различные хранилища.

**2. Использование фабричного метода (Func<T>)**

Этот подход дает вам **полный контроль над динамическим выбором** конкретной реализации в момент её запроса. Вместо внедрения коллекции, мы внедряем функцию-фабрику.

**Регистрация в Program.cs**

Сначала регистрируем каждую реализацию как отдельный сервис. Затем регистрируем нашу фабричную функцию, которая будет получать эти сервисы и возвращать нужный в зависимости от переданного ключа.

// Program.cs

|  |
| --- |
| // 1. Регистрируем все реализации напрямую  builder.Services.AddScoped<FileLogger>();  builder.Services.AddScoped<DatabaseLogger>();  builder.Services.AddScoped<MyService>();  // 2. Регистрируем Func-фабрику для динамического выбора  builder.Services.AddScoped<Func<string, IExternalLogger>>(serviceProvider => loggerType =>  {      return loggerType.ToLower() switch      {          "file" => serviceProvider.GetRequiredService<FileLogger>(),          "database" => serviceProvider.GetRequiredService<DatabaseLogger>(),          \_ => throw new ArgumentException("Неизвестный тип логгера.", nameof(loggerType))      };  }); |

**Использование в сервисе**

Теперь мы можем внедрить Func<string, IExternalLogger> и использовать её для получения нужного логгера по строковому ключу.

|  |
| --- |
| // Services/MyService.cs  using System;  public class MyService  {      private readonly Func<string, IExternalLogger> \_loggerFactory;      public MyService(Func<string, IExternalLogger> loggerFactory)      {          \_loggerFactory = loggerFactory;      }      public void Log(string message, string type)      {          var logger = \_loggerFactory(type);          logger.Log(message);      }  } |

Использование в обработчике (например, в IndexModel)

Открой Pages/Index.cshtml.cs и внедри MyService:

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;  namespace ServiceLifetimesDemo.Pages;  public class IndexModel : PageModel  {      private readonly ILogger<IndexModel> \_logger;      private readonly MyService \_myService;      public IndexModel(ILogger<IndexModel> logger, MyService myService)      {          \_logger = logger;          \_myService = myService;      }      public void OnGet()      {          \_myService.LogToAll("Тестовое сообщение при загрузке страницы");      }  } |

Запуск проекта dotnet run и **получим**

DatabaseLogger: Запись в базу данных - Тестовое сообщение при загрузке страницы

**3. Использование "ключевых" сервисов (Keyed services)**

Начиная с .NET 8, платформа предоставляет встроенную поддержку ключевых сервисов (keyed services) — механизма, позволяющего регистрировать несколько реализаций одного интерфейса с уникальным идентификатором (ключом) и внедрять их по этому ключу. Этот подход решает проблему неоднозначности при разрешении зависимостей, возникающую при множественной регистрации одного интерфейса, и делает код более читаемым, типобезопасным и поддерживаемым по сравнению с ручными фабриками или IEnumerable<T>.

3.2. Преимущества ключевых сервисов

|  |  |
| --- | --- |
|  |  |
| ✅Явное разрешение зависимостей | Можно однозначно указать, какая реализация нужна, без ручного управления фабриками. |
| ✅Поддержка всех времен жизни | Работает сSingleton,Scoped,Transient. |
| ✅Интеграция с DI-контейнером | Полностью поддерживается встроенным провайдером. |
| ✅Улучшенная читаемость | Код становится понятнее: ключ явно указывает на назначение сервиса. |
| ✅Снижение boilerplate-кода | Не нужно писать ручные фабрики илиswitch-выражения. |

3.3. Регистрация ключевых сервисов в Program.cs

Для регистрации используется метод AddKeyedScoped<TService, TImplementation>(key) (или аналоги: AddKeyedSingleton, AddKeyedTransient):

**Регистрация в Program.cs**

Используем методы AddKeyedSingleton, AddKeyedScoped или AddKeyedTransient, указывая уникальный ключ.

// Program.cs

|  |
| --- |
| builder.Services.AddKeyedScoped<IExternalLogger, FileLogger>("FileLogger");  builder.Services.AddKeyedScoped<IExternalLogger, DatabaseLogger>("DatabaseLogger");  builder.Services.AddScoped<MyService>(); |

**📌** Пояснение:

* IExternalLogger — интерфейс, реализации которого регистрируются.
* FileLogger, DatabaseLogger — конкретные реализации.
* "FileLogger", "DatabaseLogger" — строковые ключи, по которым в дальнейшем можно получить нужную реализацию.

🔹 Ключи должны быть уникальными в пределах одного типа. 🔹 Регистрация происходит по интерфейсу, но с привязкой к ключу.

3.4. Использование в сервисе: атрибут [FromKeyedServices]

Чтобы внедрить конкретную реализацию по ключу, используется атрибут [FromKeyedServices("ключ")]:

📁 Services/MyService.cs

|  |
| --- |
| using Microsoft.Extensions.DependencyInjection;  public class MyService  {      private readonly IExternalLogger \_fileLogger;      private readonly IExternalLogger \_databaseLogger;      public MyService(          [FromKeyedServices("FileLogger")] IExternalLogger fileLogger,          [FromKeyedServices("DatabaseLogger")] IExternalLogger databaseLogger)      {          \_fileLogger = fileLogger;          \_databaseLogger = databaseLogger;      }      public void Log()      {          \_fileLogger.Log("Это сообщение будет записано в файл.");          \_databaseLogger.Log("Это сообщение будет записано в базу данных.");      }  } |

📌 Важные моменты:

* Атрибут [FromKeyedServices] указывает DI-контейнеру:
* "Возьми реализацию IExternalLogger, зарегистрированную под ключом FileLogger".

Необходимо добавить:

using Microsoft.Extensions.DependencyInjection;

иначе атрибут будет неизвестен.

Обе реализации внедряются независимо, и каждая может использоваться по своему назначению.

Внедри MyService в IndexModel. Открой файл: Pages/Index.cshtml.cs

Измени его так, чтобы он принимал MyService через конструктор.

📁 Pages/Index.cshtml.cs

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;  namespace ServiceLifetimesDemo.Pages;  public class IndexModel : PageModel  {      private readonly ILogger<IndexModel> \_logger;      private readonly MyService \_myService;      public IndexModel(ILogger<IndexModel> logger, MyService myService)      {          \_logger = logger;          \_myService = myService;      }      public void OnGet()      {          \_logger.LogInformation("Загружена главная страница.");          \_myService.Log();      }  } |

**Запусти проект**

В терминале:dotnet run

Открой в браузере: https://localhost:5001 (или http://localhost:5000)

Получим:

FileLogger: Запись в файл - Это сообщение будет записано в файл.

DatabaseLogger: Запись в базу данных - Это сообщение будет записано в базу данных.

**4. Условная регистрация (Conditional Registration)**

Условная регистрация — это подход, при котором выбор конкретной реализации интерфейса происходит один раз при запуске приложения, на основе конфигурации (например, из appsettings.json, переменных окружения или аргументов командной строки).

Этот способ идеально подходит для сценариев, где:

* Нужно использовать только одну реализацию из нескольких.
* Выбор зависит от режима работы приложения (например, Development, Production).
* Конфигурация определяет, куда писать логи, как отправлять уведомления и т.д.

✅ Ключевая идея: Регистрируем только одну реализацию в DI-контейнере, исходя из настроек. Всё остальное приложение работает с интерфейсом — и не знает, какая реализация используется.

🎯 Преимущества условной регистрации

|  |  |
| --- | --- |
|  |  |
| Простота и ясность | Внедряется только одна реализация — нет неоднозначности. |
| Зависимость от конфигурации | Легко переключать поведение без перекомпиляции. |
| Низкое потребление ресурсов | В памяти только нужная реализация, нет лишних объектов. |
| Полная интеграция с DI | Работает со всеми временами жизни (Scoped,Singleton,Transient). |
| Подходит для production-сценариев | Часто используется в реальных приложениях. |

📦 4.1. Регистрация в Program.cs

Используй конфигурацию приложения (builder.Configuration) для чтения настроек, затем примени if, switch или другую логику, чтобы зарегистрировать нужную реализацию.

**Пример: выбор логгера по настройке**

|  |
| --- |
| // 1. Читаем тип логгера из конфигурации  var loggerType = builder.Configuration["LoggerType"]; // Например: "FileLogger"  // 2. Условная регистрация реализации IExternalLogger  switch (loggerType?.ToLower())  {      case "filelogger":          builder.Services.AddScoped<IExternalLogger, FileLogger>();          break;      case "databaselogger":          builder.Services.AddScoped<IExternalLogger, DatabaseLogger>();          break;      default:          // Реализация по умолчанию          builder.Services.AddScoped<IExternalLogger, FileLogger>();          break;  }  // Регистрируем сервис, который использует логгер  builder.Services.AddScoped<MyService>(); |

🔍 Пояснение:

* builder.Configuration["LoggerType"] читает значение из:
  + appsettings.json
  + Переменных окружения
  + Аргументов командной строки
* Используется ToLower() для устойчивости к регистру.
* Только одна реализация регистрируется в контейнере.
* Если значение неизвестно — используется реализация по умолчанию.

📄 Пример конфигурации: appsettings.json

|  |
| --- |
| {  "Logging": {  "LogLevel": {  "Default": "Information",  "Microsoft.AspNetCore": "Warning"  }  },  "AllowedHosts": "\*",  "LoggerType": "FileLogger"  } |

Можно также задать через переменную окружения:

export LoggerType=FileLogger

🔌 4.2. Использование в сервисе

После условной регистрации, сервис просто внедряет интерфейс — и получает нужную реализацию автоматически.

**Пример:** MyService.cs

// Services/MyService.cs

public class MyService

{

private readonly IExternalLogger \_logger;

public MyService(IExternalLogger logger)

{

\_logger = logger;

}

public void DoWork()

{

// Нам не важно, какая реализация используется —

// DI-контейнер уже внедрил правильную!

\_logger.Log("Сообщение из MyService");

}

}

💡 Важно: Весь код работает с IExternalLogger — это и есть инверсия управления (IoC). Реализация скрыта за интерфейсом.

🧪 4.3. Внедрение в IndexModel

Как и раньше, внедри MyService в PageModel.

|  |
| --- |
| // Pages/Index.cshtml.cs  using Microsoft.AspNetCore.Mvc;  using Microsoft.AspNetCore.Mvc.RazorPages;  namespace ServiceLifetimesDemo.Pages;  public class IndexModel : PageModel  {  private readonly ILogger<IndexModel> \_logger;  private readonly MyService \_myService;  public IndexModel(ILogger<IndexModel> logger, MyService myService)  {  \_logger = logger;  \_myService = myService;  }  public void OnGet()  {  \_logger.LogInformation("Загружена главная страница.");  \_myService.DoWork(); // Использует логгер, выбранный по конфигурации  }  } |

▶️ 4.4. Запуск и проверка

1. Запусти приложение: dotnet run
2. Открой https://localhost:5001
3. Проверь вывод в консоли:

* Если LoggerType = "FileLogger" → увидишь:

FileLogger: Запись в файл - Сообщение из MyService

* Если LoggerType = "DatabaseLogger" → увидишь:

DatabaseLogger: Сообщение из MyService

**💡 Советы и лучшие практики**

|  |  |
| --- | --- |
|  |  |
| Используй enum для типов | Чтобы избежать опечаток: |

1. public enum LoggerType { File, Database }

var type = Enum.TryParse<LoggerType>(builder.Configuration["LoggerType"], out var result)

? result

: LoggerType.File;

1. Добавь валидацию | Проверяй, поддерживается ли запрошенная реализация:

if (!Enum.IsDefined(typeof(LoggerType), type))

throw new InvalidOperationException($"Неподдерживаемый тип логгера: {type}");

1. Документируй настройки | Укажи в README или appsettings.json.example, какие значения допустимы. | | Не используй для динамического выбора в рантайме | Если нужно менять реализацию во время выполнения — лучше подойдут keyed services или Func<T>. |

**⚖️ Сравнение подходов**

|  |  |
| --- | --- |
|  |  |
| IEnumerable<T> | Когда нужно выполнить операциювсеми реализациями(например, оповестить всех). |
| Func<string, T> / фабрики | Когда выбор зависит отпараметров времени выполнения(например,userId,region). |
| [FromKeyedServices] | Когда нужноявно выбрать одну из нескольких реализацийпо ключу. |
| Условная регистрация | Когда выборфиксирован при стартеи зависит от конфигурации. ✅ |

✅ Заключение

Условная регистрация — это простой, эффективный и производительный способ управлять зависимостями на основе конфигурации. Он:

* Позволяет гибко настраивать поведение приложения.
* Поддерживает инверсию зависимостей (DI).
* Минимизирует сложность кода.
* Широко используется в production-приложениях.

💡 Это фундаментальный паттерн в .NET — знай его и применяй, когда поведение приложения зависит от среды (dev, staging, prod).

Каждый из этих подходов имеет свои преимущества. **IEnumerable<T>** отлично подходит для выполнения массовых операций, **Func<T>** — для сложной, динамической логики выбора, **"ключевые" сервисы** — для прямого и явного выбора, а **условная регистрация** — для конфигурационных сценариев.

### Раздел 4: Практические Сценарии Использования DI

#### 4.1. Работа с базой данных: DbContext и Scoped lifetime

DbContext из Entity Framework Core является scoped-сервисом, который предназначен для работы с одной "единицей работы" (HTTP-запросом).15 Внедрение DbContext через DI — это стандартная и рекомендуемая практика, которая позволяет контейнеру управлять его жизненным циклом, создавая его в начале запроса и утилизируя в конце.15

**Пример:**

// Регистрация DbContext в Program.cs  
builder.Services.AddDbContext<AppDbContext>();  
  
// Использование в репозитории  
public class UserRepository : IUserRepository  
{  
 private readonly AppDbContext \_context;  
  
 public UserRepository(AppDbContext context)  
 {  
 \_context = context;  
 }  
}

DI гарантирует, что все репозитории, запрошенные в рамках одного HTTP-запроса, будут работать с одним и тем же экземпляром AppDbContext.15

#### 4.2. Особенности работы с scoped-сервисами в фоновых задачах (BackgroundService)

BackgroundService — это долгоживущий сервис, который по своей природе является Singleton. Прямое внедрение scoped-сервисов, таких как DbContext, в его конструктор **приводит к ошибке**, известной как **"плененная зависимость" (Captive Dependency)**.19 Это происходит, потому что

singleton сервис будет удерживать ссылку на короткоживущую scoped зависимость на протяжении всей жизни приложения, что может вызвать утечки памяти и ошибки потокобезопасности.22

Решение: Ручное управление областью видимости

Для корректной работы с scoped-сервисами внутри singleton-сервисов необходимо вручную создавать область видимости (scope). Это можно сделать с помощью IServiceScopeFactory, который инжектируется в конструктор.

**Пример с IServiceScopeFactory:**

public class MyBackgroundTask : BackgroundService  
{  
 private readonly IServiceScopeFactory \_scopeFactory;  
  
 public MyBackgroundTask(IServiceScopeFactory scopeFactory)  
 {  
 \_scopeFactory = scopeFactory;  
 }  
  
 protected override async Task ExecuteAsync(CancellationToken stoppingToken)  
 {  
 // Создаем новую "область видимости" для каждой итерации  
 using (var scope = \_scopeFactory.CreateScope())  
 {  
 // Здесь мы находимся внутри нового "scope"  
 var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();  
 // Используем dbContext, он будет утилизирован в конце using блока  
 }  
 }  
}

Альтернативное решение: IDbContextFactory

Другой подход — использовать IDbContextFactory, который регистрируется как singleton и позволяет создавать scoped экземпляры DbContext по запросу.

public class MyBackgroundTask : BackgroundService  
{  
 private readonly IDbContextFactory<AppDbContext> \_contextFactory;  
  
 public MyBackgroundTask(IDbContextFactory<AppDbContext> contextFactory)  
 {  
 \_contextFactory = contextFactory;  
 }  
  
 protected override async Task ExecuteAsync(CancellationToken stoppingToken)  
 {  
 // Создаем новый экземпляр DbContext, когда это нужно  
 await using var dbContext = await \_contextFactory.CreateDbContextAsync();  
 // Используем dbContext  
 }  
}

#### 4.3. Внедрение зависимостей в Middleware

Классы Middleware, как и BackgroundService, являются Singleton по умолчанию. Следовательно, прямое внедрение scoped сервисов в их конструктор не допускается из-за риска Captive Dependency.23 Решение заключается во внедрении scoped сервисов в метод Invoke или InvokeAsync, так как этот метод вызывается в рамках каждого HTTP-запроса и, следовательно, находится в правильной scoped области видимости.9

### Раздел 5: DI-контейнер (IoC-контейнер)

Встроенный DI-контейнер ASP.NET Core (IServiceProvider) — это библиотека, которая автоматизирует паттерн DI. Он выполняет роль "инжектора" на уровне всего приложения, управляя сложным графом зависимостей.7 Его основные функции:

* **Регистрация сопоставлений (Register Mappings):** Контейнер хранит "карту", связывающую абстракции (интерфейсы) с их конкретными реализациями.
* **Управление жизненным циклом (Manage Lifetimes):** Контейнер отвечает за создание и уничтожение экземпляров сервисов в соответствии с их зарегистрированным временем жизни.
* **Разрешение (Resolve):** Когда класс запрашивает зависимость, контейнер рекурсивно создает ее экземпляр вместе со всеми ее собственными зависимостями, пока весь "граф объектов" не будет полностью разрешен.7

Встроенный контейнер прост и эффективен. Для более сложных сценариев можно использовать сторонние контейнеры, такие как **Autofac** 24 или **DryIoc** 23, которые предоставляют расширенные возможности, например, автоматическое сканирование сборок или поддержку именованных сервисов до.NET 8.24

### Раздел 6: Тестирование и Архитектурные Ограничения

#### 6.1. Повышение тестируемости: Мокинг и стабы

Одним из главных преимуществ DI является значительное упрощение модульного тестирования.2 Благодаря тому, что класс зависит от абстракций (интерфейсов), его зависимости можно легко заменить на фиктивные объекты: "моки" (mocks) или "стабы" (stubs).2

* **Зачем это нужно:** Представьте, что вы тестируете сервис OrderService, который сохраняет заказы в базу данных. В обычном тесте вам нужно было бы настроить реальную базу. С DI вы можете заменить IOrderRepository на "моковый" объект, который просто будет имитировать сохранение данных, не обращаясь к реальной базе. Это делает тесты быстрыми, надежными и независимыми от внешних факторов.
* **Как это работает:** Библиотеки для мокинга (например, Moq) позволяют создавать фиктивные реализации интерфейсов. Вы говорите моку, что он должен делать, когда его методы вызываются. Например: mockUserRepository.Setup(x => x.GetUserById(1)).Returns(new User());

#### 6.2. Интеграционное тестирование с WebApplicationFactory

WebApplicationFactory<T> — это мощный инструмент для написания интеграционных тестов в ASP.NET Core. Он позволяет запустить весь стек приложения в памяти, включая DI-контейнер, и затем отправлять к нему HTTP-запросы. Для тестирования можно создать кастомную фабрику, которая переопределяет конфигурацию сервисов, позволяя заменить реальные зависимости на моки для конкретных тестов.

#### 6.3. Ограничения встроенного DI-контейнера

Встроенный контейнер ASP.NET Core является простым и эффективным, но имеет некоторые ограничения 25:

* **Отсутствие property injection:** Встроенный контейнер не поддерживает автоматическое внедрение зависимостей в публичные свойства класса.9
* **Исторические ограничения:** До появления keyed services в.NET 8 встроенный контейнер не имел простого механизма для именованных сервисов, что требовало использования фабрик или коллекций IEnumerable<T>.24

### Раздел 7: Лучшие Практики и Типичные Ошибки

#### 7.1. Антипаттерны DI

Неправильное использование DI может привести к появлению антипаттернов, которые сводят на нет все его преимущества.

* **Service Locator:** Прямое обращение к IServiceProvider в коде для получения сервисов (serviceProvider.GetService<IService>()) является антипаттерном.7 Такой подход скрывает зависимости класса, делает его жестко связанным с DI-контейнером и снижает тестируемость. Правильной практикой является внедрение зависимостей через конструктор.7
* **Циклические зависимости:** Возникают, когда ServiceA зависит от ServiceB, а ServiceB — от ServiceA.17 Контейнер не может разрешить этот граф и выдает ошибку. Это не проблема DI, а "запах" в дизайне, который указывает на нарушение принципа единой ответственности (SRP). Решение состоит в рефакторинге и выделении общего посредника или интерфейса.28

#### 7.2. Ошибка "Пленённой зависимости" (Captive Dependency)

Ошибка "Пленённой зависимости" — это серьезная проблема, возникающая при неверной конфигурации жизненных циклов.21 Она происходит, когда долгоживущий сервис (Singleton) захватывает в качестве зависимости короткоживущий сервис (Scoped или Transient).21 Поскольку Singleton создается один раз и держит ссылку на зависимость, эта зависимость будет существовать на протяжении всего времени работы приложения, а не в течение своей "запланированной" короткой жизни.22

**Почему это опасно:** Например, Singleton сервис ReportService получает в конструкторе Scoped сервис AppDbContext.21 В результате, все HTTP-запросы будут использовать один и тот же экземпляр DbContext, что может привести к ошибкам потокобезопасности, потере данных и утечкам памяти.22 В окружении разработки встроенный DI-контейнер ASP.NET Core имеет проверку ValidateScopes 29, которая автоматически обнаруживает подобные ошибки и выбрасывает исключение, что является ценной защитой.22

**Как исправить:**

1. Сделать оба сервиса Scoped (или Transient), чтобы их жизненные циклы совпадали.21
2. Использовать фабрику (IServiceScopeFactory или IDbContextFactory) для ручного создания короткоживущей зависимости по мере необходимости.21

#### 7.3. Таблица: Основные антипаттерны и их решения

|  |  |  |  |
| --- | --- | --- | --- |
| Антипаттерн | Описание | Почему это плохо | Решение |
| **Service Locator** | Прямое обращение к IServiceProvider в коде.7 | Скрывает зависимости, делает код зависимым от контейнера, снижает тестируемость.27 | Внедрение через конструктор.27 |
| **Captive Dependency** | Singleton сервис внедряет Scoped или Transient зависимость.22 | Утечки памяти, ошибки потокобезопасности, некорректное состояние.22 | Согласовать жизненные циклы или использовать фабрику для ручного создания.21 |
| **Циклические зависимости** | Два сервиса зависят друг от друга по кругу.17 | Контейнер не может разрешить зависимости, плохой дизайн.17 | Рефакторинг, разделение ответственности.28 |

### Раздел 8: DI в Различных Типах Приложений и Заключение

#### 8.1. DI в разных типах приложений

**1. Внедрение зависимостей в ASP.NET Core Web API: архитектурная роль и управление жизненным циклом**

**1.1. Общая концепция внедрения зависимостей в Web API**

В ASP.NET Core Web API внедрение зависимостей (Dependency Injection, DI) является фундаментальным механизмом, обеспечивающим реализацию принципа инверсии управления (Inversion of Control, IoC) и способствующим созданию слабо связанных, тестируемых и поддерживаемых компонентов. DI интегрирован на уровне фреймворка и позволяет передавать зависимости в компоненты (в первую очередь — в контроллеры) через конструктор, минимизируя жёсткую привязку к конкретным реализациям.

Механизм DI в Web API базируется на интерфейсе IServiceProvider и контейнере, поставляемом по умолчанию (DefaultServiceProvider), который управляет созданием и уничтожением экземпляров сервисов в соответствии с их зарегистрированным временем жизни.

**1.2. Внедрение зависимостей в контроллеры**

Контроллеры в ASP.NET Core Web API являются типичными получателями зависимостей. Через конструктор в них могут внедряться сервисы, отвечающие за:

* бизнес-логику (IUserService, IOrderService),
* доступ к данным (IRepository<T>),
* внешние интеграции (IEmailService, IExternalApiService),
* логирование (ILogger<T>).

**Пример:**

[ApiController]

[Route("api/[controller]")]

public class UsersController : ControllerBase

{

private readonly IUserService \_userService;

private readonly ILogger<UsersController> \_logger;

public UsersController(IUserService userService, ILogger<UsersController> logger)

{

\_userService = userService;

\_logger = logger;

}

[HttpGet]

public async Task<IActionResult> GetUsers()

{

\_logger.LogInformation("Получение списка пользователей");

var users = await \_userService.GetAllAsync();

return Ok(users);

}

}

В данном примере:

* IUserService — абстракция бизнес-слоя.
* ILogger<UsersController> — встроенный сервис логирования.
* Зависимости разрешаются автоматически через DI-контейнер при создании экземпляра контроллера.

Такой подход обеспечивает:

* слабую связанность между компонентами,
* удобство тестирования (возможность подмены зависимостей моками),
* централизованное управление жизненным циклом.

**1.3. Управление жизненным циклом сервисов в контексте HTTP-запроса**

Одной из ключевых особенностей Web API является привязка времени жизни сервисов к жизненному циклу HTTP-запроса. ASP.NET Core поддерживает три основные области видимости (lifetimes):

|  |  |
| --- | --- |
|  |  |
| Transient | Новый экземпляр создаётся каждый раз при запросе зависимости. Подходит для лёгких, иммутабельных сервисов. |
| Scoped | Один экземпляр наодин HTTP-запрос. Создаётся при начале запроса и уничтожается при его завершении. |
| Singleton | Один экземпляр на всё приложение. Создаётся при запуске и живёт до завершения процесса. |

**Особое значение имеет** scoped**-область:**

* Сервисы, зарегистрированные как scoped, создаются один раз на запрос, но могут использоваться многократно в рамках этого запроса (например, в контроллере, сервисе, репозитории).
* Это предотвращает создание дубликатов и обеспечивает согласованность состояния (например, при использовании DbContext в Entity Framework Core).

**Пример регистрации:**

builder.Services.AddScoped<IUserService, UserService>();

builder.Services.AddTransient<IEmailService, EmailService>();

builder.Services.AddSingleton<ICacheService, MemoryCacheService>();

**1.4. Архитектурные преимущества использования DI в Web API**

|  |  |
| --- | --- |
|  |  |
| Модульность | Сервисы могут быть заменены без изменения контроллеров. |
| Тестируемость | Возможность использования моков (например, через Moq) в unit- и integration-тестах. |
| Централизованная инициализация | Все зависимости регистрируются в одном месте —Program.csилиStartup.cs. |
| Управление ресурсами | Контейнер DI автоматически вызываетDispose()дляIDisposable-сервисов в конце скоупа. |

**1.5. Практические рекомендации**

* Используйте интерфейсы для определения контрактов сервисов.
* Избегайте service locator pattern — всегда внедряйте зависимости через конструктор.
* Не храните состояние в singleton-сервисах без синхронизации (риск race condition).
* Для scoped-сервисов убедитесь, что они не "утекают" в фоновые задачи (например, через Task.Run), где контекст запроса недоступен.

**1.6. Заключение**

Внедрение зависимостей в ASP.NET Core Web API играет ключевую архитектурную роль, обеспечивая гибкость, масштабируемость и тестируемость приложений. Благодаря привязке scoped-сервисов к жизненному циклу HTTP-запроса, DI становится эффективным инструментом управления контекстом выполнения, особенно при работе с базами данных, транзакциями и пользовательскими сессиями.

Таким образом, правильное использование DI позволяет строить чистые, масштабируемые и сопровождаемые API, соответствующие современным стандартам разработки программного обеспечения.

**2. Внедрение зависимостей в Blazor Server: архитектурные особенности и управление жизненным циклом сервисов**

**2.1. Общие принципы внедрения зависимостей в Blazor Server**

Blazor Server — это технология, реализующая интерактивные веб-приложения на стороне сервера с использованием .NET и WebAssembly-подобной модели компонентного программирования. В отличие от традиционных веб-фреймворков, где каждый HTTP-запрос обрабатывается независимо, Blazor Server поддерживает долгоживущее соединение между клиентом и сервером посредством протокола SignalR. Это принципиально изменяет поведение контейнера внедрения зависимостей (Dependency Injection, DI), особенно в отношении сервисов с областью видимости *scoped*.

Центральной особенностью Blazor Server является интеграция с встроенным контейнером DI ASP.NET Core, позволяющая использовать механизм инъекции зависимостей не только в сервисах бизнес-логики, но и непосредственно в Razor-компонентах. Это обеспечивает высокую степень модульности, тестируемости и соответствует принципу инверсии зависимостей (Dependency Inversion Principle, DIP) из SOLID.

**4.2. Механизм инъекции зависимостей в компонентах: директива** @inject

Для внедрения сервисов в Razor-компоненты Blazor предоставляет специальную директиву @inject, синтаксис которой определяется следующим образом:

@inject [Тип] [ИмяПеременной]

Данная директива эквивалентна добавлению параметра в конструктор компонента, однако, в отличие от явной конструкторной инъекции, @inject упрощает синтаксис и повышает читаемость компонентов, особенно при работе с несколькими зависимостями.

**Пример использования:**

@page "/counter"

@inject IExternalLogger Logger

@inject NavigationManager Navigation

<h3>Счётчик</h3>

<p>Текущее значение: @currentCount</p>

<button @onclick="Increment">Увеличить</button>

@code {

private int currentCount = 0;

private void Increment()

{

currentCount++;

Logger.Log($"Счётчик увеличен до {currentCount}");

}

}

В данном примере:

* IExternalLogger — абстракция, реализующая логику записи событий.
* NavigationManager — встроенный сервис ASP.NET Core, отвечающий за навигацию между маршрутами.

Семантически директива @inject инициирует разрешение зависимости через IServiceProvider, ассоциированный с текущим контекстом выполнения. При этом компилятор Razor трансформирует директиву в поле класса, инициализируемое через DI-контейнер.

**2.3. Особенности жизненного цикла сервисов в Blazor Server**

Одной из ключевых архитектурных особенностей Blazor Server является расширение смысла области видимости scoped по сравнению с традиционными веб-приложениями.

**2.3.1. Контекст** scoped **в ASP.NET Core (традиционное поведение)**

В стандартных ASP.NET Core-приложениях (например, MVC или Web API) сервисы, зарегистрированные с областью видимости scoped, существуют в течение одного HTTP-запроса. При завершении запроса экземпляр сервиса уничтожается.

**2.3.2. Контекст** scoped **в Blazor Server**

В Blazor Server область видимости scoped привязывается к сессии пользователя, то есть к длительности SignalR-соединения. Это означает, что:

* Каждый пользователь получает собственный экземпляр scoped-сервиса.
* Экземпляр сохраняется в памяти на протяжении всей сессии.
* Состояние сервиса сохраняется между обновлениями UI и обработкой событий.

Таким образом, scoped-сервисы в Blazor Server функционируют как состоятельные компоненты уровня сессии, что делает их идеальным инструментом для управления пользовательским состоянием (user state management).

**2.4. Практическое применение: управление состоянием пользователя**

Рассмотрим пример использования scoped-сервиса для хранения пользовательских настроек.

**Определение сервиса:**

public class UserPreferences

{

public string Theme { get; set; } = "light";

public int FontSize { get; set; } = 14;

}

**Регистрация в контейнере DI:**

builder.Services.AddScoped<UserPreferences>();

**Использование в компоненте:**

@page "/settings"

@inject UserPreferences Preferences

<h3>Настройки</h3>

<p>Тема: @Preferences.Theme</p>

<p>Размер шрифта: @Preferences.FontSize</p>

<button @onclick="ToggleTheme">Сменить тему</button>

@code {

private void ToggleTheme()

{

Preferences.Theme = Preferences.Theme == "light"

? "dark"

: "light";

}

}

**Анализ поведения:**

* При первом обращении пользователя к любому компоненту, использующему UserPreferences, DI-контейнер создаёт новый экземпляр.
* Все последующие обращения в рамках одной сессии используют тот же экземпляр.
* Изменения состояния (например, переключение темы) сохраняются автоматически.
* При разрыве соединения (выход пользователя) экземпляр уничтожается, и память освобождается.

**4.5. Рекомендации по проектированию**

При использовании scoped-сервисов в Blazor Server следует учитывать следующие аспекты:

|  |  |
| --- | --- |
|  |  |
| Потребление памяти | Избегать хранения больших объёмов данных вscoped-сервисах, так как каждый пользователь удерживает отдельный экземпляр. |
| Потокобезопасность | При наличии асинхронных операций необходимо обеспечивать потокобезопасный доступ к состоянию (например, черезlockилиSemaphoreSlim). |
| Тестирование | Для модульного тестирования следует использоватьIServiceScopeдля симуляции пользовательской сессии. |
| Выбор области видимости | <ul><li>Singleton— для общих ресурсов (кэш, настройки приложения).</li><li>Scoped— для состояния пользователя.</li><li>Transient— для лёгких, независимых операций.</li></ul> |

**2.6. Ограничения и потенциальные риски**

Несмотря на преимущества, использование scoped-сервисов в Blazor Server сопряжено с рядом рисков:

* Утечка памяти: при длительных сессиях и большом количестве пользователей возможен значительный рост потребления памяти.
* Сложность масштабирования: состояние хранится на сервере, что затрудняет горизонтальное масштабирование без использования внешних хранилищ (например, Redis).
* Проблемы с восстановлением сессии: при обрыве соединения состояние теряется, если не реализован механизм персистентности.

**2.7. Заключение**

Blazor Server предоставляет мощный механизм внедрения зависимостей, интегрированный с жизненным циклом пользовательской сессии. Директива @inject позволяет декларативно внедрять сервисы в компоненты, обеспечивая высокую степень абстракции и модульности. Благодаря расширенной семантике scoped-сервисов, Blazor Server поддерживает управление состоянием на уровне сессии пользователя, что делает его подходящим для разработки сложных интерактивных приложений.

Однако данная модель требует тщательного проектирования с учётом потребления ресурсов и масштабируемости. В проектах с высокой нагрузкой рекомендуется комбинировать scoped-сервисы с внешними механизмами хранения состояния (например, через IStateManager или базу данных).

Вывод: Blazor Server трансформирует традиционную модель DI, превращая scoped-сервисы в инструмент управления пользовательской сессией, что расширяет возможности архитектурного проектирования, но требует осознанного подхода к управлению ресурсами.

**3. Внедрение зависимостей в Worker Services: особенности управления областями видимости**

**3.1. Архитектурная специфика Worker Services**

Worker Services в .NET представляют собой приложения, предназначенные для выполнения фоновых задач в течение продолжительного времени, таких как обработка очередей, опрос баз данных, отправка уведомлений или взаимодействие с внешними API. В отличие от веб-приложений (например, ASP.NET Core Web API или Blazor), Worker Services не привязаны к HTTP-контексту, что существенно влияет на поведение механизма внедрения зависимостей (Dependency Injection, DI).

Несмотря на использование общего контейнера DI (IServiceProvider), предоставляемого ASP.NET Core, отсутствие встроенного жизненного цикла запроса делает невозможным автоматическое создание и управление *scoped*-областями (областями видимости), которые в веб-приложениях привязаны к HTTP-запросу.

**3.2. Проблема использования scoped-сервисов в фоновых задачах**

Сервисы, зарегистрированные с областью видимости scoped, предполагают существование логической единицы выполнения, в рамках которой их экземпляр остаётся неизменным. В веб-приложениях такой единицей является HTTP-запрос. Однако в Worker Services:

❌ Нет автоматической привязки scoped-сервисов к контексту выполнения.

Попытка внедрить scoped-сервис напрямую в BackgroundService (наследника IHostedService) приводит к ошибке времени выполнения:

InvalidOperationException: Cannot consume scoped service 'IService' from singleton 'IHostedService'.

Это происходит потому, что IHostedService (включая BackgroundService) регистрируется как singleton, а внедрение scoped-сервиса в singleton нарушает правила контейнера DI.

**3.3. Решение: ручное создание областей видимости**

Для корректного использования scoped-зависимостей (например, DbContext, IRepository<T>, IUserContext) в Worker Services необходимо вручную создавать область видимости с помощью IServiceScope.

**Алгоритм:**

1. Внутри фонового метода (ExecuteAsync) получить IServiceScopeFactory.
2. Создать новую область с помощью CreateScope().
3. Через serviceProvider в этой области разрешить scoped-зависимости.
4. Обеспечить освобождение ресурсов (вызов Dispose при выходе из области).

**Пример реализации:**

public class Worker : BackgroundService

{

private readonly ILogger<Worker> \_logger;

private readonly IServiceProvider \_serviceProvider;

public Worker(ILogger<Worker> logger, IServiceProvider serviceProvider)

{

\_logger = logger;

\_serviceProvider = serviceProvider;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

\_logger.LogInformation("Worker запущен.");

while (!stoppingToken.IsCancellationRequested)

{

try

{

// Создаём новую область видимости

using var scope = \_serviceProvider.CreateScope();

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

var processor = scope.ServiceProvider.GetRequiredService<IMessageProcessor>();

await processor.ProcessAsync(dbContext, stoppingToken);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Ошибка при выполнении фоновой задачи.");

}

// Пауза между итерациями

await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);

}

\_logger.LogInformation("Worker остановлен.");

}

}

**3.4. Обоснование подхода**

* IServiceProvider внедряется как singleton, что допустимо.
* IServiceScope создаётся динамически в рамках каждой итерации, обеспечивая:
  + Изоляцию состояния.
  + Корректное управление временем жизни scoped-сервисов.
  + Автоматическое освобождение ресурсов (например, подключения к БД).
* Каждая итерация обработки может рассматриваться как логическая единица работы, аналогичная HTTP-запросу.

**3.5. Практические рекомендации**

|  |  |
| --- | --- |
|  |  |
| ✅ Всегда используйтеusingпри созданииIServiceScope | Гарантирует вызовDispose()и освобождение неуправляемых ресурсов. |
| ✅ Выполняйте тяжёлые операции внутри области | Чтобы избежать утечек памяти и "зависших" соединений. |
| ✅ Не сохраняйте ссылки наscoped-сервисы за пределами области | Это приведёт к ошибкам и неопределённому поведению. |
| ✅ ИспользуйтеCancellationToken | Для корректной остановки при завершении приложения. |
| ✅ Рассмотрите использованиеIHostedServiceдля нескольких фоновых задач | Каждая может управлять своей областью. |

**3.6. Альтернативные сценарии**

* Transient-сервисы могут использоваться напрямую, так как создаются при каждом запросе.
* Singleton-сервисы — допустимы, но требуют потокобезопасности при изменении состояния.
* Для обработки сообщений из очереди (например, RabbitMQ, Kafka) — каждое сообщение может обрабатываться в отдельной IServiceScope.

**3.7. Заключение**

Worker Services, будучи долгоживущими процессами, требуют явного управления областями видимости при использовании scoped-зависимостей. Отсутствие автоматического контекста запроса делает невозможным применение стандартного механизма DI без дополнительных мер.

✅ Ручное создание IServiceScope через IServiceScopeFactory является стандартным и рекомендованным подходом, обеспечивающим:

* Безопасное использование scoped сервисов,
* Корректное управление ресурсами,
* Соответствие принципам DI и IoC.

Таким образом, правильное проектирование фоновых задач включает осознанное разделение жизненных циклов сервисов и организацию изолированных областей выполнения, что позволяет строить надёжные и масштабируемые фоновые приложения в экосистеме .NET.

**4. Консольные приложения на основе** IHost**: реализация полноценного внедрения зависимостей вне веб-контекста**

**4.1. Введение**

Традиционно консольные приложения ассоциировались с простыми сценариями выполнения — запуском скриптов, обработкой данных или автоматизацией задач без сложной архитектуры. Однако с появлением унифицированной модели хостинга (Generic Host) в .NET Core 2.1 и её дальнейшим развитием в .NET 5+, 6+, 8+ стало возможным строить сложные, модульные и сопровождаемые консольные приложения с использованием тех же механизмов, что и в ASP.NET Core.

Ключевым элементом этой архитектуры является интерфейс IHost, который предоставляет централизованное управление жизненным циклом приложения, конфигурацией, логированием и внедрением зависимостей (Dependency Injection, DI). Это означает, что даже в отсутствие веб-сервера или HTTP-запросов консольное приложение может использовать полноценный DI-контейнер, аналогичный тому, что используется в Web API, Blazor или Worker Services.

**4.2. Архитектурная основа:** IHost **и** Host.CreateDefaultBuilder

Для активации механизмов DI в консольном приложении используется статический метод Host.CreateDefaultBuilder(), который инициализирует экземпляр IHostBuilder с уже настроенными компонентами:

* Конфигурация (IConfiguration) — из appsettings.json, переменных окружения, аргументов командной строки.
* Логирование (ILogger) — интеграция с провайдерами логирования (Console, Debug, EventLog и др.).
* Внедрение зависимостей (IServiceCollection, IServiceProvider).
* Управление жизненным циклом (IHostedService).

Убедись, что в твой .csproj-файл добавлена ссылка на Microsoft.Extensions.Hosting:

|  |
| --- |
| <ItemGroup>  <PackageReference Include="Microsoft.Extensions.DependencyInjection" Version="9.0.8" />  <PackageReference Include="Microsoft.Extensions.Hosting" Version="9.0.8" />  </ItemGroup> |

Если его нет — добавь вручную или установи через CLI:

dotnet add package Microsoft.Extensions.Hosting

**Пример базовой структуры:**

|  |
| --- |
| using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Hosting;  using Microsoft.Extensions.Logging;  var host = Host.CreateDefaultBuilder(args)  .ConfigureServices((context, services) =>  {  // Пример регистрации сервисов  services.AddSingleton<ILogger, ConsoleLogger>();  services.AddScoped<IUserDataService, UserDataService>();  services.AddTransient<IReportGenerator, PdfReportGenerator>();  // Регистрация фоновой задачи  services.AddHostedService<ApplicationService>();  })  .Build();  // Запуск приложения  await host.RunAsync(); |

**Создай нужные классы**

Убедись, что у тебя есть реализации интерфейсов, например:

using Microsoft.Extensions.Logging;

public class ConsoleLogger : ILogger

{

public IDisposable? BeginScope<TState>(TState state) => null;

public bool IsEnabled(LogLevel logLevel) => true;

public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception? exception, Func<TState, Exception?, string> formatter)

{

Console.WriteLine($"[{logLevel}] {formatter(state, exception)}");

}

}

IUserDataService **и** UserDataService.cs

public interface IUserDataService { }

public class UserDataService : IUserDataService { }

IReportGenerator **и** PdfReportGenerator.cs

public interface IReportGenerator { }

public class PdfReportGenerator : IReportGenerator { }

ApplicationService.cs

**4.3. Внедрение зависимостей в консольном приложении**

После построения хоста DI-контейнер становится полностью функциональным. Все зарегистрированные сервисы могут быть:

* Автоматически внедрены в реализации IHostedService.
* Получены вручную через IServiceProvider.
* Управляемы в соответствии с их временем жизни: transient, scoped, singleton.

**Пример сервиса с инъекцией:**

using Microsoft.Extensions.Hosting;

public class ApplicationService : IHostedService

{

private readonly ILogger \_logger;

public ApplicationService(ILogger logger)

{

\_logger = logger;

}

public async Task StartAsync(CancellationToken cancellationToken)

{

\_logger.Log(LogLevel.Information, 0, "ApplicationService запущен.", null, (s, e) => s);

await Task.Delay(1000, cancellationToken); // имитация работы

}

public Task StopAsync(CancellationToken cancellationToken)

{

\_logger.Log(LogLevel.Information, 0, "ApplicationService остановлен.", null, (s, e) => s);

return Task.CompletedTask;

}

}

Здесь зависимости (ILogger, IUserDataService) автоматически разрешаются контейнером DI при создании экземпляра ApplicationService.

**4.4. Управление областями видимости (scoped services)**

Как и в других типах приложений, использование scoped-сервисов требует явного создания области с помощью IServiceScope, поскольку в консольных приложениях отсутствует автоматический контекст запроса.

**Пример ручного создания области:**

using var scope = host.Services.CreateScope();

var dbContext = scope.ServiceProvider.GetRequiredService<AppDbContext>();

var processor = scope.ServiceProvider.GetRequiredService<IDataProcessor>();

await processor.ProcessAsync(dbContext);

✅ Каждая такая область гарантирует, что scoped-сервисы будут созданы и уничтожены корректно, а ресурсы (например, соединение с БД) — освобождены.

**4.5. Преимущества использования** IHost **в консольных приложениях**

|  |  |
| --- | --- |
|  |  |
| Модульность | Возможность разделения логики на независимые сервисы через интерфейсы. |
| Тестируемость | Лёгкая замена зависимостей моками в unit-тестах. |
| Централизованная конфигурация | ПоддержкаIConfiguration, включаяappsettings.json, environment variables. |
| Управление жизненным циклом | ПоддержкаIHostedServiceдля фоновых задач и корректного завершения. |
| Интеграция с экосистемой .NET | ИспользованиеILogger,IOptions,Health Checks,EF Coreи др. |

**4.6. Практические сценарии применения**

Такой подход особенно эффективен в следующих случаях:

* Скрипты миграции базы данных (например, с использованием Entity Framework Core).
* Обработка сообщений из очереди (RabbitMQ, Kafka, Azure Service Bus).
* Пакетная обработка данных (файлов, логов, экспорта/импорта).
* Автоматизированные задачи администрирования.
* Тестовые утилиты, имитирующие поведение микросервисов.

**4.7. Сравнение с традиционными консольными приложениями**

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Управление зависимостями | Ручное создание объектов | Автоматическое внедрение через DI |
| Конфигурация | Жёстко закодирована или парсится вручную | Интеграция сIConfiguration |
| Логирование | Console.WriteLine() | Структурированное логирование черезILogger |
| Жизненный цикл | Нет управления остановкой | ПоддержкаCancellationToken,StopAsync |
| Архитектура | Процедурная | Модульная, ориентированная на DI |

**4.8. Заключение**

Внедрение зависимостей в консольных приложениях .NET более не является архитектурным компромиссом. Благодаря интерфейсу IHost и Generic Host, разработчик может создать полноценную инфраструктуру приложения с использованием DI-контейнера, даже в отсутствие веб-интерфейса.

✅ Таким образом, консольное приложение, построенное на IHost, перестаёт быть простым скриптом и превращается в структурированное, масштабируемое и легко тестируемое приложение, соответствующее современным стандартам разработки программного обеспечения.

Этот подход особенно рекомендуется для любых нетривиальных сценариев, где важны поддерживаемость, расширяемость и архитектурная чистота.

#### 8.2. Сквозной пример: DI в проекте с Clean Architecture

Давайте подробно разберем, как реализовать сквозной пример с **Dependency Injection (DI)** в проекте на **ASP.NET Core** с использованием **Clean Architecture**, включая **Blazor** для пользовательского интерфейса. Этот процесс покажет, как каждый слой приложения взаимодействует с другими, не зная их конкретных реализаций, благодаря инверсии зависимостей.

**Шаг 1: Подготовка проекта**

Сначала нам нужно создать решение с несколькими проектами, чтобы реализовать Clean Architecture. Мы будем использовать **.NET CLI**, что является стандартным подходом в Visual Studio Code.

**1.1. Создание решения и проектов с помощью .NET CLI**

1. Откройте терминал в Visual Studio Code (или любой другой терминал).
2. Создайте новую папку для вашего решения и перейдите в нее:

mkdir CleanArchitectureExample

cd CleanArchitectureExample

1. Создайте пустое решение:

dotnet new sln

1. Создайте проекты для каждого слоя: **Слой Core**: dotnet new classlib -o Core

**Назначение**: Этот проект будет содержать наши основные бизнес-правила и сущности. Это самый "внутренний" слой.

**Слой Application**: dotnet new classlib -o Application

**Назначение**: Здесь будут находиться интерфейсы для репозиториев, обработчики команд и запросов.

**Слой Infrastructure**: dotnet new classlib -o Infrastructure

**Назначение**: Этот слой будет содержать конкретные реализации интерфейсов из Application.

**Слой WebUI**: dotnet new blazorwasm -o WebUI --no-hosted

**Важно**: Флаг --no-hosted создает только клиентский Blazor-проект, что нам и нужно для этого примера.

**Назначение**: Это самый внешний слой, отвечающий за пользовательский интерфейс.

**1.2. Добавление проектов в решение и настройка зависимостей**

Теперь нужно добавить созданные проекты в файл решения и настроить зависимости между ними.

1. Добавьте все проекты в файл решения:

dotnet sln add Core

dotnet sln add Application

dotnet sln add Infrastructure

dotnet sln add WebUI

1. Настройте зависимости (ссылки на проекты):

Проект Application должен ссылаться на Core: dotnet add Application reference Core

Проект Infrastructure должен ссылаться на Core и Application:

dotnet add Infrastructure reference Core

dotnet add Infrastructure reference Application

Проект WebUI должен ссылаться на Application и Infrastructure:

dotnet add WebUI reference Application

dotnet add WebUI reference Infrastructure

После завершения этого шага ваша структура проекта готова.

**Шаг 2: Реализация слоев**

Теперь, когда структура проекта готова, мы можем реализовать каждый из слоев, следуя принципам Clean Architecture.

**2.1. Слой Core**

Этот слой содержит основные сущности приложения. Создайте простой класс, который будет представлять пользователя.

1. В проекте Core создайте папку Entities.
2. В этой папке создайте файл User.cs и добавьте следующий код:

namespace Core.Entities

{

public class User

{

public int Id { get; set; }

public string Name { get; set; }

public string Email { get; set; }

}

}

**2.2. Слой Application**

Здесь мы определяем интерфейс нашего репозитория, который будет работать с сущностью User.

1. В проекте Application создайте папку Interfaces.
2. В этой папке создайте файл IUserRepository.cs и добавьте следующий код:

using Core.Entities;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace Application.Interfaces

{

public interface IUserRepository

{

Task<User> GetUserByIdAsync(int id);

Task<IEnumerable<User>> GetAllUsersAsync();

Task AddUserAsync(User user);

}

}

**2.3. Слой Infrastructure**

Этот слой содержит конкретную реализацию IUserRepository с использованием, например, Entity Framework Core и In-Memory базы данных для простоты.

**В проекте Infrastructure** установите пакеты NuGet, используя .NET CLI:

cd Infrastructure

dotnet add package Microsoft.EntityFrameworkCore.InMemory

cd ..

1. Создайте папку Data.
2. В этой папке создайте файл AppDbContext.cs (наш DbContext) и добавьте следующий код:

using Core.Entities;

using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Data

{

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

public DbSet<User> Users { get; set; }

}

}

1. Создайте файл UserRepository.cs, который будет реализовывать IUserRepository:

using Application.Interfaces;

using Core.Entities;

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace Infrastructure.Data

{

public class UserRepository : IUserRepository

{

private readonly AppDbContext \_context;

// Конструктор, в который DI-контейнер внедрит AppDbContext

public UserRepository(AppDbContext context)

{

\_context = context;

}

public async Task AddUserAsync(User user)

{

\_context.Users.Add(user);

await \_context.SaveChangesAsync();

}

public async Task<IEnumerable<User>> GetAllUsersAsync()

{

return await \_context.Users.ToListAsync();

}

public async Task<User> GetUserByIdAsync(int id)

{

return await \_context.Users.FirstOrDefaultAsync(u => u.Id == id);

}

}

}

Обратите внимание, что UserRepository запрашивает AppDbContext через конструктор. Это и есть **внедрение зависимости**.

**Шаг 3: Настройка DI в слое WebUI**

Это самый важный шаг. Здесь мы связываем все наши слои и регистрируем зависимости в DI-контейнере.

**3.1. Установка пакетов и регистрация сервисов в Program.cs**

1. **В проекте WebUI** установите пакеты NuGet, используя .NET CLI:

cd WebUI

dotnet add package Microsoft.EntityFrameworkCore.InMemory

cd ..

**Откройте файл WebUI/Program.cs**.

1. В этом файле мы регистрируем наши сервисы. Замените стандартный код на следующий, добавив регистрацию DbContext и репозитория:

using Microsoft.AspNetCore.Components.Web;

using Microsoft.AspNetCore.Components.WebAssembly.Hosting;

using WebUI;

using Infrastructure.Data; // Импорт слоя Infrastructure

using Microsoft.EntityFrameworkCore;

using Application.Interfaces; // Импорт слоя Application

var builder = WebAssemblyHostBuilder.CreateDefault(args);

builder.RootComponents.Add<App>("#app");

builder.RootComponents.Add<HeadOutlet>("head::after");

// 1. Регистрация DbContext (scoped)

// In-memory базу данных мы используем для простоты примера

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseInMemoryDatabase("UsersDb"));

// 2. Регистрация IUserRepository и его конкретной реализации UserRepository (scoped)

// Это связующее звено, которое говорит DI-контейнеру, что для интерфейса

// IUserRepository нужно использовать класс UserRepository.

builder.Services.AddScoped<IUserRepository, UserRepository>();

await builder.Build().RunAsync();

* **AddDbContext**: Мы регистрируем AppDbContext. DI-контейнер создаст экземпляр AppDbContext, который будет **scoped** (один на каждый компонент в Blazor).
* **AddScoped<IUserRepository, UserRepository>()**: Эта строка говорит DI-контейнеру: "Когда кто-то запросит IUserRepository, создай и верни экземпляр UserRepository".

**Шаг 4: Использование DI в компоненте Blazor**

Теперь, когда зависимости зарегистрированы, мы можем использовать их в любом компоненте Blazor.

1. **В проекте WebUI** откройте файл Pages/Index.razor.

Внедрите IUserRepository в этот компонент, используя директиву @inject.

Напишем простой код для отображения списка пользователей.

@page "/"

@using Application.Interfaces

@using Core.Entities

@inject IUserRepository UserRepository

@using System.Collections.Generic

@using System.Threading.Tasks

<PageTitle>Index</PageTitle>

<h1>Hello, users!</h1>

<p role="status">Count: @users?.Count()</p>

@if (users == null)

{

<p><em>Загрузка...</em></p>

}

else

{

<table class="table">

<thead>

<tr>

<th>ID</th>

<th>Имя</th>

<th>Email</th>

</tr>

</thead>

<tbody>

@foreach (var user in users)

{

<tr>

<td>@user.Id</td>

<td>@user.Name</td>

<td>@user.Email</td>

</tr>

}

</tbody>

</table>

}

@code {

private IEnumerable<User> users;

protected override async Task OnInitializedAsync()

{

// Добавляем тестовые данные для демонстрации

await UserRepository.AddUserAsync(new User { Id = 1, Name = "Alice", Email = "alice@example.com" });

await UserRepository.AddUserAsync(new User { Id = 2, Name = "Bob", Email = "bob@example.com" });

// Запрашиваем данные через внедренный репозиторий

users = await UserRepository.GetAllUsersAsync();

}

}

**Шаг 5: Запуск и тестирование**

1. **Запустите проект** из корневой папки решения, используя .NET CLI:

cd WebUI

dotnet run

1. Терминал покажет, что проект запущен и работает на определенном порту (например, http://localhost:5000).
2. Откройте указанный URL в браузере. Вы увидите страницу Blazor с таблицей, содержащей пользователей "Alice" и "Bob". Это означает, что DI-контейнер успешно:
   * Создал экземпляр Index.razor.
   * Обнаружил, что Index.razor запрашивает IUserRepository.
   * Создал экземпляр UserRepository (поскольку мы так настроили в Program.cs).
   * Создал экземпляр AppDbContext (поскольку UserRepository его запросил).
   * Внедрил все эти зависимости, позволяя компоненту Index.razor работать с данными, не зная, как именно они хранятся.

Таким образом, DI обеспечивает бесшовную интеграцию и **инверсию управления** (Inversion of Control), позволяя каждому слою работать только с тем, что ему нужно, не зная о конкретной реализации.

#### 8.3. Заключительные рекомендации

Внедрение зависимостей — это не просто инструмент, а фундаментальная часть современной разработки на.NET. Чтобы по-настоящему раскрыть его потенциал, следует придерживаться нескольких ключевых принципов:

* **Всегда предпочитайте внедрение через конструктор.** Это стандартная и лучшая практика.
* **Программируйте к абстракциям.** Использование интерфейсов вместо конкретных классов обеспечивает гибкость и тестируемость.
* **Тщательно выбирайте время жизни сервисов.** Transient для легковесных утилит, Scoped для сервисов, привязанных к запросу, и Singleton только для глобальных, потокобезопасных ресурсов.
* **Избегайте антипаттернов.** Service Locator и "плененная зависимость" являются сигналами о проблемах в архитектуре.

Овладение DI позволяет создавать не просто работающий, а архитектурно выверенный, поддерживаемый и легко расширяемый код.

#### Источники

1. ASP.NET - Writing Clean Code in ASP.NET Core with Dependency ..., дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/archive/msdn-magazine/2016/may/asp-net-writing-clean-code-in-asp-net-core-with-dependency-injection>
2. Unit Testing in .NET Core - Mastering Mocking - C# Corner, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/unit-testing-in-net-core-mastering-mocking/>
3. Do you know the best dependency injection container? | SSW.Rules, дата последнего обращения: августа 22, 2025, <https://www.ssw.com.au/rules/the-best-dependency-injection-container/>
4. Can someone explain when to use Singleton, Scoped and Transient with some real life examples? : r/csharp - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/csharp/comments/1acwtar/can_someone_explain_when_to_use_singleton_scoped/>
5. Service Lifetimes in ASP.NET Core: Transient vs Scoped vs Singleton, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/service-lifetimes-in-asp-net-co-transient-vs-scoped-vs-singleton/>
6. Dependency Injection In .NET Core - C# Corner, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/dependency-injection-in-net-core/>
7. Dependency injection in ASP.NET Core | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-9.0>
8. Exploring Keyed Services Dependency Injection | by sharmila subbiah - Medium, дата последнего обращения: августа 22, 2025, <https://medium.com/@malarsharmila/exploring-keyed-services-dependency-injection-7183e9f819db>
9. Dependency Injection in ASP.NET Core Explained - codewithmukesh, дата последнего обращения: августа 22, 2025, <https://codewithmukesh.com/blog/dependency-injection-in-aspnet-core-explained/>
10. Mocking dependencies in ASP.NET Core tests · Cezary Piątek Blog, дата последнего обращения: августа 22, 2025, <https://cezarypiatek.github.io/post/mocking-dependencies-in-asp-net-core/>
11. Advanced Dependency Injection Techniques in C# - TechOnDiapers, дата последнего обращения: августа 22, 2025, <https://tech-on-diapers.hashnode.dev/advanced-dependency-injection-techniques-in-c>
12. Clean Architecture in ASP.NET Core - NDepend Blog, дата последнего обращения: августа 22, 2025, <https://blog.ndepend.com/clean-architecture-for-asp-net-core-solution/>
13. Understanding Service Lifetimes in .Net Core - ScholarHat, дата последнего обращения: августа 22, 2025, <https://www.scholarhat.com/tutorial/net/service-lifetimes-in-net-core>
14. When to Use Transient, Scoped, or Singleton in .NET Apps - Understanding Service Lifetimes - codewithmukesh, дата последнего обращения: августа 22, 2025, <https://codewithmukesh.com/blog/when-to-use-transient-scoped-singleton-dotnet/>
15. Managing the Entity Framework Core DbContext in .NET: Using ..., дата последнего обращения: августа 22, 2025, <https://sasangaedirisinghe.medium.com/managing-the-entity-framework-core-dbcontext-in-net-using-statement-vs-dependency-injection-798caefbb298>
16. Adding a implementation rather than its interface in ASP.NET Core DI - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/49175697/adding-a-implementation-rather-than-its-interface-in-asp-net-core-di>
17. Dealing with Multiple Implementations of a Service in ASP.NET Core & ABP Dependency Injection, дата последнего обращения: августа 22, 2025, <https://abp.io/community/articles/dealing-with-multiple-implementations-of-a-service-in-asp.net-core-abp-dependency-injection-ysfp4ho2>
18. asp.net core - Do we need interfaces for dependency injection? - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/43079277/do-we-need-interfaces-for-dependency-injection>
19. Inject a DbContext Instance Into BackgroundService in .NET Core, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/inject-a-dbcontext-instance-into-backgroundservice-in-net-core/>
20. How to register multiple implementations of the same interface in Asp.Net Core?, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/39174989/how-to-register-multiple-implementations-of-the-same-interface-in-asp-net-core>
21. What is Captive Dependency in .Net core ? | by Sudip Ranabhat | Medium, дата последнего обращения: августа 22, 2025, <https://medium.com/@sudipdevdev/what-is-captive-dependency-in-net-core-ebad7b5e838>
22. Captive Dependency - ploeh blog, дата последнего обращения: августа 22, 2025, <https://blog.ploeh.dk/2014/06/02/captive-dependency/>
23. ASP.NET Core — Autofac 7.0.0 documentation, дата последнего обращения: августа 22, 2025, <https://autofac.readthedocs.io/en/latest/integration/aspnetcore.html>
24. dependency injection - How is AutoFac better than Microsoft ..., дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/63407601/how-is-autofac-better-than-microsoft-extensions-dependencyinjection>
25. ASP.NET Core: Built-in IoC Container - Tutorials Teacher, дата последнего обращения: августа 22, 2025, <https://www.tutorialsteacher.com/core/internals-of-builtin-ioc-container-in-aspnet-core>
26. is injecting a iserviceprovider instance considered a anti-pattern : r/dotnet - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/dotnet/comments/tipu7i/is_injecting_a_iserviceprovider_instance/>
27. c# - Is IDependencyResolver an anti-pattern? - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/5653783/is-idependencyresolver-an-anti-pattern>
28. In ASP.Net Core , how does one resolve circular references?, дата последнего обращения: августа 22, 2025, <https://softwareengineering.stackexchange.com/questions/343943/in-asp-net-core-how-does-one-resolve-circular-references>
29. ASP.NET Core Dependency Injection: What is the IServiceProvider and how is it Built?, дата последнего обращения: августа 22, 2025, <https://www.stevejgordon.co.uk/aspnet-core-dependency-injection-what-is-the-iserviceprovider-and-how-is-it-built>