

2.1.1. Разработка проекта на ASP.NET Core (Blazor) с использованием SQLite и Clean Architecture

Введение: за пределами кода — понимание «почему»

Сегодня наша лекция будет посвящена не просто написанию кода, а созданию архитектуры — фундаменту, на котором строится любое серьезное программное решение. Мы рассмотрим, как применить принципы **Clean Architecture** к проекту на **ASP.NET Core** с использованием **Blazor** и **SQLite**. Цель нашей встречи — не просто переписать готовый пример, а глубоко понять, какие проблемы решает этот архитектурный подход и почему инвестиции в его освоение окупаются сторицей.

В мире разработки программного обеспечения мы часто сталкиваемся с неизбежным явлением, которое можно сравнить с энтропией в физике: со временем системы становятся более хаотичными и сложными. Традиционные монолитные или жестко связанные архитектуры, где вся логика, данные и пользовательский интерфейс перемешаны, подвержены этой энтропии в наибольшей степени. Изменения в одной части системы могут вызвать непредсказуемые сбои в другой, тестирование становится сложным и медленным, а поддержка и масштабирование превращаются в настоящий кошмар.

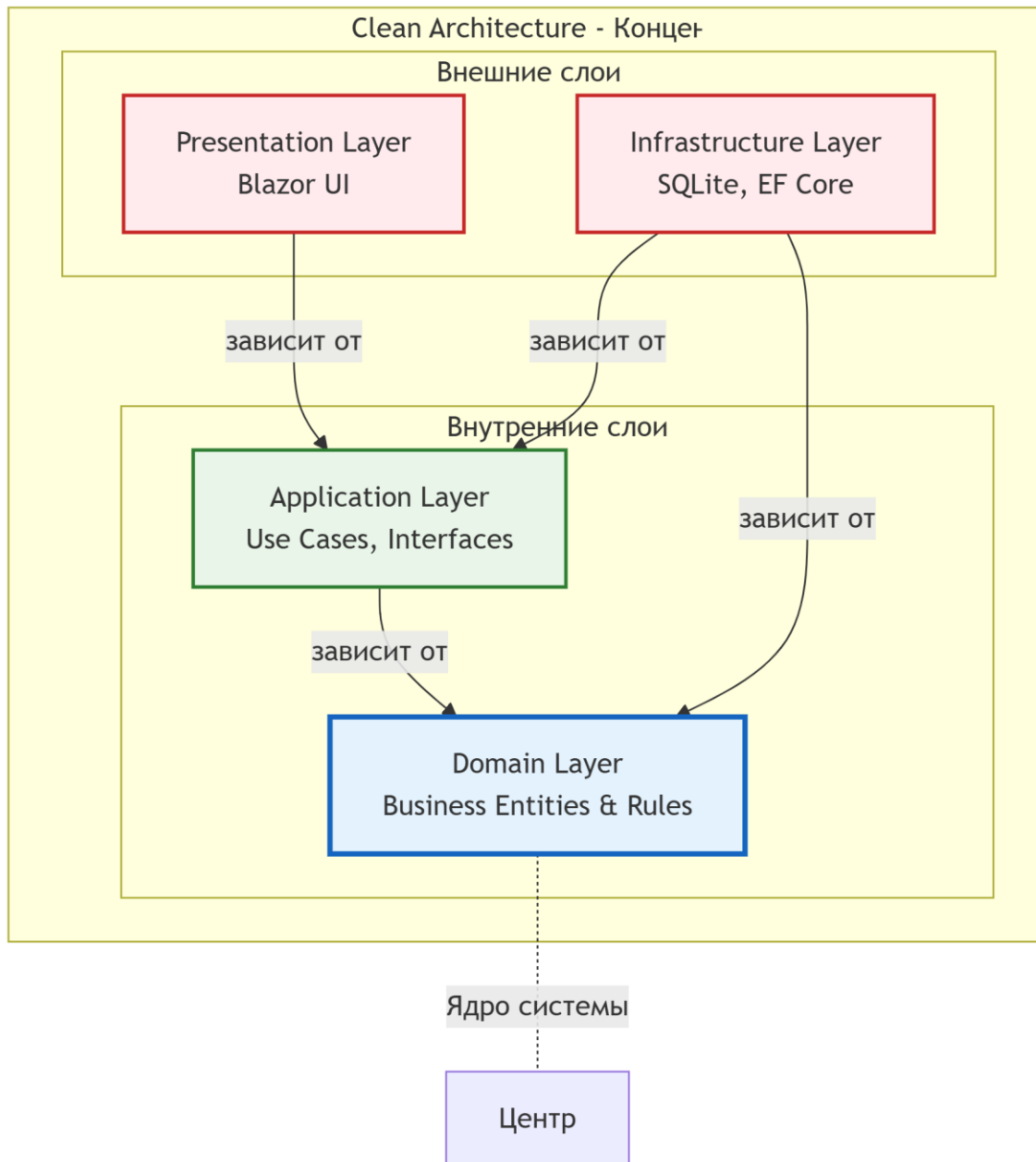
Clean Architecture, предложенная Робертом Мартином, известным как «Дядя Боб», выступает в качестве мощного противоядия от этого хаоса. Ее ключевой принцип — **Правило зависимостей**. Он гласит, что зависимости в системе должны всегда указывать внутрь, от внешних слоев к внутренним. Представьте себе концентрические круги:

- В самом центре — **слой предметной области (Domain)**, содержащий критически важную бизнес-логику.
- Следующий круг — **слой приложения (Application)**, определяющий сценарии использования.
- Далее — **слой инфраструктуры (Infrastructure)**, отвечающий за внешние детали, такие как базы данных, файловые системы и сторонние API.
- И, наконец, самый внешний круг — **слой представления (Presentation)**, который является пользовательским интерфейсом.

Эта структура обеспечивает независимость ядра системы от любых внешних технологий. Внутренние круги не знают ничего о внешних. Таким образом, слой Domain полностью независим от Infrastructure и Presentation, что делает его независимым от фреймворков, базы данных и пользовательского интерфейса. Это не просто академическое упражнение; это практический подход, который дает ощутимые преимущества:

- **Независимость от фреймворков:** Ваша основная бизнес-логика не привязана к ASP.NET Core, Blazor или какой-либо другой технологии. Ее можно использовать в любом другом приложении.
- **Тестируемость:** Центральные бизнес-правила можно тестировать в полной изоляции, не требуя для этого реальной базы данных или веб-сервера.
- **Независимость от пользовательского интерфейса:** UI можно заменить (например, перейти от Blazor к REST API) без необходимости вносить изменения в основной код.
- **Независимость от базы данных:** Вы можете легко сменить SQLite на SQL Server, PostgreSQL или любую другую СУБД, так как ядро приложения взаимодействует с абстракциями, а не с конкретными реализациями.

Сегодня мы будем строить небольшое, но показательное приложение для управления задачами. Наш путь будет проходить именно по этой архитектурной схеме: мы начнем с определения ядра системы и будем двигаться наружу, последовательно добавляя слои и связывая их вместе.



0. Создание решения и проектов

Требования: Установлены .NET 8 SDK, Visual Studio Code с расширением C# Dev Kit.

1. Создание структуры решения

Создаем решение

```
dotnet new sln --name BlazorCleanArchitecture
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell - BlazorCleanArchitecture + v

PS D:\MyAspNetCoreApp> cd BlazorApplication
PS D:\MyAspNetCoreApp\BlazorApplication> md BlazorCleanArchitecture

Directory: D:\MyAspNetCoreApp\BlazorApplication

Mode                LastWriteTime         Length Name
----                -
d-----          27.08.2025    21:12             BlazorCleanArchitecture

PS D:\MyAspNetCoreApp\BlazorApplication> cd BlazorCleanArchitecture
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet new sln --name BlazorCleanArchitecture
Шаблон "Файл решения" успешно создан.

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture>
```

Создаем проекты для каждого слоя

dotnet new blazor -n Presentation -o Presentation --no-https

dotnet new classlib -n Application -o Application

dotnet new classlib -n Domain -o Domain

dotnet new classlib -n Infrastructure -o Infrastructure

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell - BlazorCleanArchitecture + v ... | [?] x

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet new blazor -n Presentation -o Presentation --no-https
Шаблон "Веб-приложение Blazor" успешно создан.
Этот шаблон содержит технологии сторонних производителей, кроме Майкрософт. Дополнительные сведения см. в разделе https://aka.ms/aspnetcore/9.0-third-party-notices.

Идет обработка действий после создания...
Восстановление D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture\Presentation\Presentation.csproj:
Восстановление выполнено.

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet new classlib -n Application -o Application
Шаблон "Библиотека классов" успешно создан.

Идет обработка действий после создания...
Восстановление D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture\Application\Application.csproj:
Восстановление выполнено.

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet new classlib -n Domain -o Domain
Шаблон "Библиотека классов" успешно создан.

Идет обработка действий после создания...
Восстановление D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture\Domain\Domain.csproj:
Восстановление выполнено.

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet new classlib -n Infrastructure -o Infrastructure
Шаблон "Библиотека классов" успешно создан.

Идет обработка действий после создания...
Восстановление D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture\Infrastructure\Infrastructure.csproj:
Восстановление выполнено.

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture>
```

Добавляем проекты в решение

dotnet sln add Presentation/Presentation.csproj

dotnet sln add Application/Application.csproj

dotnet sln add Domain/Domain.csproj

dotnet sln add Infrastructure/Infrastructure.csproj

```
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet sln add Presentation/Presentation.csproj
Проект "Presentation\Presentation.csproj" добавлен в решение.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet sln add Application/Application.csproj
Проект "Application\Application.csproj" добавлен в решение.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet sln add Domain\Domain.csproj
Проект "Domain\Domain.csproj" добавлен в решение.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet sln add Infrastructure/Infrastructure.csproj
Проект "Infrastructure\Infrastructure.csproj" добавлен в решение.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture>
```

BlazorCleanArchitecture

- > .vs
- > Application
- > Domain
- > Infrastructure
- > Presentation
- ≡ .gitignore
- ≡ BlazorCleanArchitecture.sln

Обоснование структуры решения:

Создание многопроектного решения — это фундаментальный шаг в реализации Clean Architecture. Каждый проект представляет собой отдельный слой с четко определенными responsibilities:

- **Presentation** — слой представления, отвечающий за пользовательский интерфейс и взаимодействие с клиентом
- **Application** — слой приложения, содержащий бизнес-логику и use cases
- **Domain** — ядро системы с бизнес-сущностями и правилами
- **Infrastructure** — инфраструктурный слой для работы с внешними системами

Такое разделение обеспечивает соблюдение принципа единственной ответственности (Single Responsibility Principle) и позволяет независимо развивать каждый компонент системы.

2. Настройка зависимостей между слоями

Presentation зависит от Application

```
dotnet add Presentation reference Application
```

```
dotnet add Presentation reference Infrastructure
```

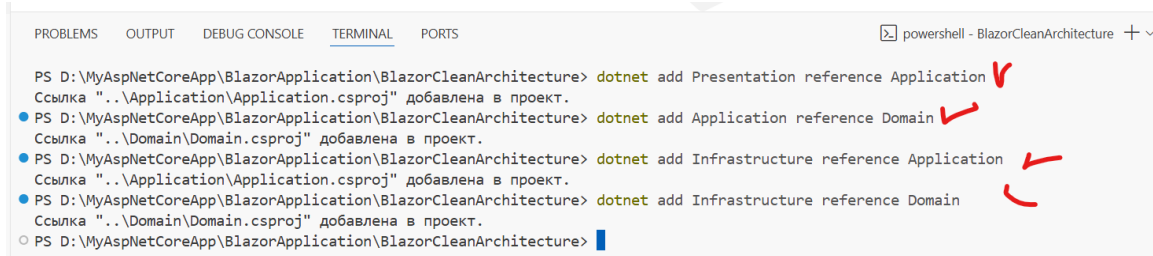
Application зависит от Domain

dotnet add Application reference Domain

Infrastructure зависит от Application и Domain

dotnet add Infrastructure reference Application

dotnet add Infrastructure reference Domain



```
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet add Presentation reference Application
Ссылка "..\Application\Application.csproj" добавлена в проект.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet add Application reference Domain
Ссылка "..\Domain\Domain.csproj" добавлена в проект.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet add Infrastructure reference Application
Ссылка "..\Application\Application.csproj" добавлена в проект.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet add Infrastructure reference Domain
Ссылка "..\Domain\Domain.csproj" добавлена в проект.
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture>
```

Устанавливаем необходимые пакеты

dotnet	add	Infrastructure	package
Microsoft.EntityFrameworkCore.Sqlite			
dotnet	add	Infrastructure	package
Microsoft.EntityFrameworkCore.Design			
dotnet	add	Presentation	package
Microsoft.EntityFrameworkCore.Design			
dotnet	add	Presentation	package
Microsoft.AspNetCore.Components.QuickGrid			



```
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet add Infrastructure package Microsoft.EntityFrameworkCore.Sqlite
```

Обоснование зависимостей:

Настройка зависимостей между проектами — это критически важный шаг, который физически воплощает принципы Clean Architecture и Правило зависимостей (Dependency Rule):

1. **Presentation** → **Application**: Слой представления зависит от слоя приложения, но не наоборот. Это позволяет UI использовать бизнес-логику, но изолирует ядро от деталей реализации интерфейса.
2. **Application** → **Domain**: Слой приложения зависит от доменного слоя, что обеспечивает доступ к бизнес-сущностям и правилам без привязки к инфраструктуре.

3. **Infrastructure → Application & Domain:** Инфраструктурный слой зависит от вышележащих слоев для реализации определенных в них интерфейсов (например, репозиториях). Это классическое проявление принципа инверсии зависимостей (DIP) — зависимости направлены внутрь, к ядру системы.

Обоснование выбора пакетов NuGet:

- **Microsoft.EntityFrameworkCore.Sqlite** — предоставляет провайдер SQLite для EF Core, что позволяет использовать легковесную файловую базу данных, идеальную для разработки и тестирования
- **Microsoft.EntityFrameworkCore.Design** — необходим для работы с миграциями EF Core и инструментами дизайна
- **Microsoft.AspNetCore.Components.QuickGrid** — высокопроизводительный компонент для отображения табличных данных с сортировкой, фильтрацией и пагинацией, что значительно улучшает пользовательский опыт при работе с большими объемами данных

Почему такая структура важна:

1. **Тестируемость:** Каждый слой можно тестировать изолированно. Доменный слой тестируется без зависимостей от базы данных или UI.
2. **Заменяемость компонентов:** Можно легко заменить SQLite на другую СУБД или Blazor на другой UI framework, изменив только соответствующий слой.
3. **Сопровождаемость:** Изменения в одном слое минимально затрагивают другие слои, что уменьшает риски регрессий.
4. **Масштабируемость:** Новые функциональности добавляются в соответствующие слои без нарушения существующей архитектуры.
5. **Разделение ответственности:** Разработчики могут специализироваться на конкретных слоях, что повышает эффективность команды.

Такая организация проекта создает прочный фундамент для разработки сложных enterprise-приложений, которые остаются гибкими и поддерживаемыми на протяжении всего жизненного цикла.

Часть I: Ядро системы — Слой предметной области (Domain Layer)

1.1. Сердце приложения

Слой предметной области, или Domain Layer, является центром нашей архитектуры. Это самый важный слой, поскольку он содержит фундаментальные сущности и бизнес-правила, которые определяют суть нашего приложения. Он полностью изолирован от внешнего мира и не должен иметь никаких зависимостей от других слоев. Здесь мы определяем «что» делает наше приложение, без каких-либо деталей о «как». Это источник истины, живое воплощение бизнес-требований.

1.2. Сущности предметной области — класс TaskItem

В Domain/Entities/TaskItem.cs мы видим определение нашей главной сущности. На первый взгляд, это может показаться обычной моделью данных, но на самом деле это не так. **Сущность предметной области** — это не просто структура с публичными свойствами; это объект, который инкапсулирует как данные, так и поведение, которое ими управляет. Методы MarkAsCompleted(), MarkAsIncomplete() и UpdateDetails() — это не просто удобные вспомогательные функции, это критически важный аспект проектирования.

Рассмотрим альтернативный подход, при котором эти методы были бы вынесены в отдельный класс-помощник или сервис. В этом случае данные (IsCompleted, UpdatedAt) и логика, которая их изменяет, оказались бы в разных частях системы. Это нарушило бы принцип инкапсуляции и сделало бы систему более хрупкой. Например, разработчик, вызывающий метод UpdateDetails(), может забыть обновить свойство UpdatedAt, что приведет к несогласованности данных. Размещение этих методов непосредственно внутри класса TaskItem гарантирует, что бизнес-правила всегда применяются к данным, к которым они относятся. Этот подход, известный как **«богатая доменная модель»**, делает доменную сущность самодостаточной и легко тестируемой в изоляции.

Кроме того, в нашей обновленной доменной модели появился метод Validate(). Это очень важный аспект. Согласно принципам Clean

Architecture, валидация бизнес-правил, которые не зависят от внешних факторов (таких как наличие записи в базе данных), должна находиться как можно ближе к доменной сущности, которую она защищает. Размещение валидации в TaskItem гарантирует, что любая логика, которая создает или изменяет задачу, будет использовать одни и те же правила, независимо от того, где она вызывается. Это предотвращает создание некорректных сущностей и делает систему более надежной.

```
namespace Domain.Entities;
public class TaskItem
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public string Description { get; set; } = string.Empty;
    public bool IsCompleted { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime? UpdatedAt { get; set; }

    // Бизнес-логика в доменной модели
    public void MarkAsCompleted()
    {
        IsCompleted = true;
        UpdatedAt = DateTime.UtcNow;
    }

    public void MarkAsIncomplete()
    {
        IsCompleted = false;
        UpdatedAt = DateTime.UtcNow;
    }

    public void UpdateDetails(string title, string description)
    {
        Title = title;
        Description = description;
    }
}
```

```

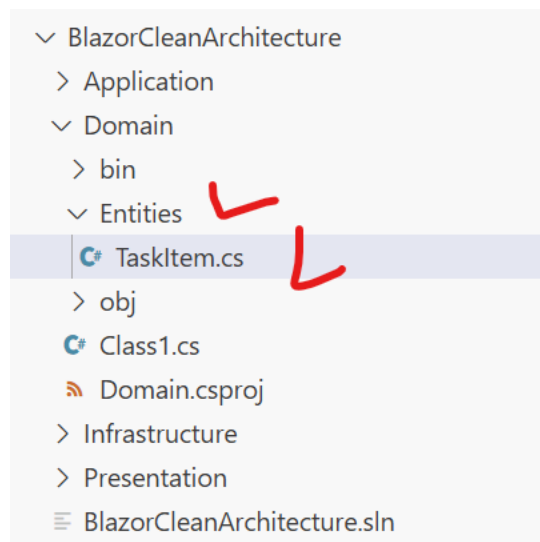
        UpdatedAt = DateTime.UtcNow;
    }

    public void Validate()
    {
        if (string.IsNullOrEmpty(Title))
            throw new ArgumentException("Title is required");

        if (Title.Length > 200)
            throw new ArgumentException("Title cannot exceed 200
characters");

        if (Description.Length > 1000)
            throw new ArgumentException("Description cannot exceed 1000
characters");
    }
}

```



1.3. Абстракции и контракты — Интерфейсы репозитория

В Domain/Interfaces мы определяем два интерфейса: IRepository<T> и ITaskRepository. Эти интерфейсы представляют собой контракты, которые определяют «что» можно делать с данными (например, получать, добавлять, обновлять и удалять), но не «как».

В обновленном интерфейсе IRepository появился новый метод ExistsAsync(int id). Это абстракция, которая позволяет сервисному слою проверить наличие сущности в хранилище, не зная деталей реализации. ITaskRepository был расширен методами SearchTasksAsync(string searchTerm) и GetTasksCountAsync(), которые отражают новые бизнес-сценарии. Эти методы предоставляют специфические для домена запросы, инкапсулируя их в репозитории.

Размещение этих интерфейсов именно в слое Domain является важнейшим архитектурным решением, которое является физическим проявлением **Правила зависимостей**. Слой Application будет зависеть от этих интерфейсов для выполнения операций с данными. Слой Infrastructure будет предоставлять конкретные реализации этих интерфейсов. Таким образом, стрелка зависимости направлена внутрь: Infrastructure (внешний слой) зависит от Domain (внутренний слой), а не наоборот.

```
namespace Domain.Interfaces;
```

```
/// <summary>
```

```
/// Общий интерфейс репозитория для выполнения базовых операций  
с сущностями.
```

```
/// Предоставляет асинхронные операции по доступу и управлению  
данными.
```

```
/// </summary>
```

```
/// <typeparam name="T">Тип сущности, с которой работает  
репозиторий. Должен быть ссылочным типом.</typeparam>
```

```
public interface IRepository<T> where T : class
```

```
{
```

```
    /// <summary>
```

```
    /// Асинхронное получение сущности по её идентификатору.
```

```
    /// </summary>
```

```
        /// <param name="id">Уникальный идентификатор  
сущности.</param>
```

```
        /// <returns>
```

```

    /// Задача, результатом которой является сущность типа
    <typeparamref name="T"/> или <c>null</c>,
    /// если сущность с указанным идентификатором не найдена.
    /// </returns>
    Task<T?> GetByIdAsync(int id);

    /// <summary>
    /// Асинхронное получение всех сущностей из хранилища.
    /// </summary>
    /// <returns>
    /// Задача, результатом которой является коллекция всех сущностей
    типа <typeparamref name="T"/>.
    /// Возвращает пустую коллекцию, если сущности отсутствуют.
    /// </returns>
    Task<IEnumerable<T>> GetAllAsync();

    /// <summary>
    /// Асинхронное добавление новой сущности в хранилище.
    /// </summary>
    /// <param name="entity">Сущность для добавления. Не должна быть
    <c>null</c>.</param>
    /// <returns>Задача, представляющая асинхронную операцию
    добавления.</returns>
    Task AddAsync(T entity);

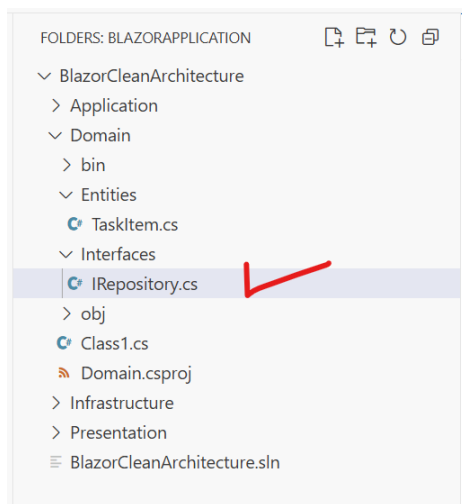
    /// <summary>
    /// Асинхронное обновление существующей сущности в хранилище.
    /// </summary>
    /// <param name="entity">Сущность с обновлёнными данными. Не
    должна быть <c>null</c>.</param>
    /// <returns>Задача, представляющая асинхронную операцию
    обновления.</returns>
    Task UpdateAsync(T entity);

```

```

    /// <summary>
        /// Асинхронная проверка существования сущности по
    идентификатору.
    /// </summary>
        /// <param name="id">Идентификатор сущности для
    проверки.</param>
    /// <returns>
        /// Задача, результатом которой является значение <c>true</c>, если
    сущность существует; иначе — <c>false</c>.
    /// </returns>
    Task<bool> ExistsAsync(int id);
}

```



```

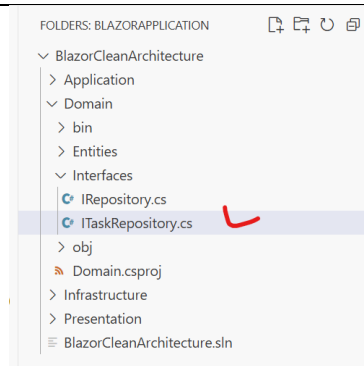
using Domain.Entities;

namespace Domain.Interfaces;

public interface ITaskRepository : IRepository<TaskItem>
{
    Task DeleteAsync(int id);
    Task<IEnumerable<TaskItem>> GetCompletedTasksAsync();
    Task<IEnumerable<TaskItem>> GetPendingTasksAsync();
}

```

```
Task<IEnumerable<TaskItem>> SearchTasksAsync(string  
searchTerm);  
}
```



Часть II: Оркестровка сценариев использования — Слой приложения (Application Layer)

2.1. Командный центр

Слой Application — это следующий уровень в нашей архитектуре. Он содержит специфическую бизнес-логику, которая определяет сценарии использования нашего приложения. Он выступает в роли оркестратора, управляя потоком данных между слоями Presentation и Domain, а также используя инфраструктурные сервисы для выполнения своих задач. Этот слой так же, как и Domain, не зависит от внешних технологий, таких как пользовательский интерфейс или база данных.

2.2. Разделение с помощью объектов передачи данных (DTOs)

В Application/DTOs мы определяем несколько объектов: TaskItemDto, CreateTaskDto и UpdateTaskDto. Они представляют собой простые структуры данных, предназначенные для передачи информации через границы приложения — например, от пользовательского интерфейса к сервису и обратно.

Возникает закономерный вопрос: почему бы не использовать саму сущность TaskItem для передачи данных? Ответ заключается в необходимости **декапсуляции и разделения ответственности**. Доменные сущности TaskItem содержат не только данные, но и важную бизнес-логику. Передача их напрямую в UI может привести к

непреднамеренным побочным эффектам или к тому, что UI будет пытаться манипулировать их состоянием, минуя сервисный слой.

DTOs решают эту проблему, выступая в качестве специального «представления» данных для пользовательского интерфейса. DTO может содержать только подмножество свойств сущности или же агрегировать данные из нескольких сущностей. Это разделение позволяет изменять контракт данных для UI без необходимости вносить правки в основную доменную модель. Кроме того, это повышает безопасность, предотвращая раскрытие внутренних или чувствительных свойств доменной модели.

Обратите внимание, как мы используем современные возможности языка C# для усиления этого подхода. В TaskItemDto мы видим модификатор `required` для свойства `Title`. Это нововведение языка C# 11 создает **компиляторный контракт**, который гарантирует, что свойство `Title` должно быть обязательно инициализировано при создании DTO. Это делает код более безопасным и выразительным, снижая вероятность ошибок, связанных с отсутствием данных.

```
using System.ComponentModel.DataAnnotations;

namespace Application.DTOs;

/// <summary>
/// DTO for returning task details.
/// </summary>
public class TaskItemDto
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Title is required.")]
    [StringLength(200, MinimumLength = 1, ErrorMessage = "Title must be
between 1 and 200 characters.")]
    public required string Title { get; set; }
```

```
[StringLength(1000, ErrorMessage = "Description cannot exceed 1000 characters.")]
```

```
    public string? Description { get; set; } // nullable: null means no description set
```

```
    public bool IsCompleted { get; set; }
```

```
    public DateTime CreatedAt { get; set; }
```

```
    public DateTime? UpdatedAt { get; set; }  
}
```

```
using System.ComponentModel.DataAnnotations;
```

```
namespace Application.DTOs;
```

```
/// <summary>
```

```
/// DTO for creating a new task.
```

```
/// </summary>
```

```
public class CreateTaskDto
```

```
{
```

```
    [Required(ErrorMessage = "Title is required.")]
```

```
    [StringLength(200, MinimumLength = 1, ErrorMessage = "Title must be between 1 and 200 characters.")]
```

```
    public string Title { get; set; } = string.Empty;
```

```
    [StringLength(1000, ErrorMessage = "Description cannot exceed 1000 characters.")]
```

```
    public string? Description { get; set; }
```

```
}
```

```
using System.ComponentModel.DataAnnotations;
```



```

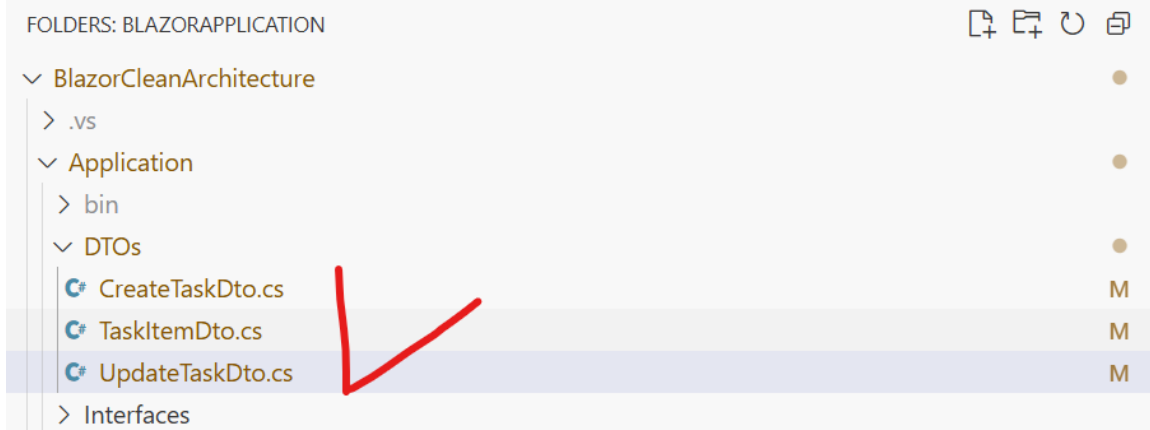
namespace Application.DTOs;

/// <summary>
/// DTO for updating an existing task (supports partial updates).
/// </summary>
public class UpdateTaskDto
{
    [Required(ErrorMessage = "Title is required.")]
    [StringLength(200, MinimumLength = 1, ErrorMessage = "Title must be
between 1 and 200 characters.")]
    public string Title { get; set; } = string.Empty;

    [StringLength(1000, ErrorMessage = "Description cannot exceed 1000
characters.")]
    public string? Description { get; set; }

    public bool? IsCompleted { get; set; } // nullable: if not provided, do not
update status
}

```



Замечание. Кроме того, можно (реализуйте самостоятельно) для DTO, предназначенных для создания и обновления, мы используем тип record (CreateTaskDto и UpdateTaskDto). record'ы идеально подходят для таких объектов. Они являются **неизменяемыми (immutable)** по умолчанию, что означает, что их состояние не может быть изменено после создания. Это гарантирует, что данные, передаваемые между

слоями, остаются целостными и не могут быть случайно изменены в процессе.

0 references

```
public record CreateTaskDto(string Title, string Description);
```

0 references

```
public record UpdateTaskDto(string Title, string Description, bool IsCompleted);
```

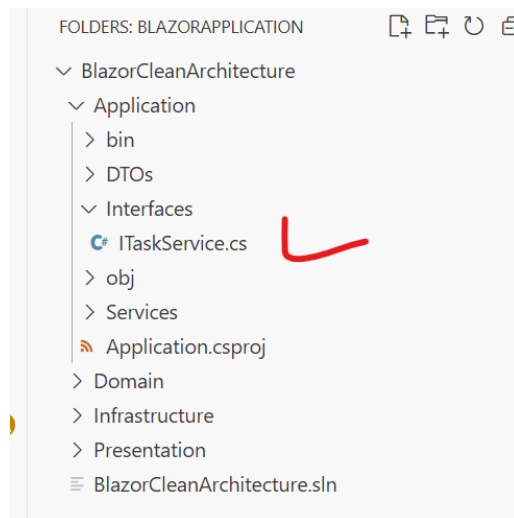
2.3. Определение сервисного контракта — ITaskService

Интерфейс ITaskService в Application/Interfaces определяет публичный API для всех сценариев использования нашего приложения. Это контракт, который слой Presentation будет использовать для взаимодействия с бизнес-логикой. Он абстрагирует все операции, такие как создание, обновление и удаление задач, предоставляя чистый и понятный набор методов. Обновленная версия интерфейса включает новые методы MarkTaskAsIncompleteAsync, SearchTasksAsync и GetTasksCountAsync, что позволяет слою представления работать с расширенным функционалом, таким как фильтрация и поиск, без изменения основной бизнес-логики.

```
using Application.DTOs;

namespace Application.Interfaces;

public interface ITaskService
{
    Task<IEnumerable<TaskItemDto>> GetAllTasksAsync();
    Task<TaskItemDto?> GetTaskByIdAsync(int id);
    Task<TaskItemDto> CreateTaskAsync(CreateTaskDto createTaskDto);
    Task UpdateTaskAsync(int id, UpdateTaskDto updateTaskDto);
    Task DeleteTaskAsync(int id);
    Task MarkTaskAsCompletedAsync(int id);
    Task MarkTaskAsIncompleteAsync(int id);
    Task<IEnumerable<TaskItemDto>> SearchTasksAsync(string
searchTerm);
    Task<int> GetTasksCountAsync();
}
```



2.4. Реализация оркестратора — TaskService

Класс TaskService является реализацией интерфейса ITaskService. Здесь происходит вся оркестровка. Через конструктор, используя **внедрение зависимостей**, он получает экземпляр ITaskRepository — абстракцию, определенную в слое Domain.

Каждый метод в TaskService демонстрирует, как он управляет потоком данных:

1. Он принимает DTOs из внешнего слоя (Presentation).
2. Выполняет необходимую логику, специфичную для приложения (например, преобразует DTO в доменную сущность TaskItem).
3. Использует интерфейс репозитория (_taskRepository) для выполнения операций с данными, не зная, как именно они сохраняются.
4. Возвращает DTOs обратно во внешний слой.

Важно отметить, что TaskService ничего не знает о DbContext, SQLite или Blazor. Его единственная забота — это реализация сценариев использования с помощью доступных ему абстракций. В обновленном коде можно увидеть, как CreateTaskAsync и UpdateTaskAsync вызывают метод task.Validate(). Это показывает, как слой приложения управляет доменной моделью и обеспечивает выполнение бизнес-правил, определенных в ядре.

```
using Application.DTOs;  
using Application.Interfaces;
```

```
using Domain.Entities;
using Domain.Interfaces;

namespace Application.Services;

public class TaskService : ITaskService
{
    private readonly ITaskRepository _taskRepository;

    public TaskService(ITaskRepository taskRepository)
    {
        _taskRepository = taskRepository;
    }

    public async Task<IEnumerable<TaskItemDto>> GetAllTasksAsync()
    {
        var tasks = await _taskRepository.GetAllAsync();
        return tasks.Select(MapToDto);
    }

    public async Task<TaskItemDto?> GetTaskByIdAsync(int id)
    {
        var task = await _taskRepository.GetByIdAsync(id);
        return task is null ? null : MapToDto(task);
    }

    public async Task<TaskItemDto> CreateTaskAsync(CreateTaskDto
createTaskDto)
    {
        var task = new TaskItem
        {
            Title = createTaskDto.Title,
            Description = createTaskDto.Description ?? string.Empty,
            IsCompleted = false,
```

```
        CreatedAt = DateTime.UtcNow
    };

    task.Validate();
    await _taskRepository.AddAsync(task);
    return MapToDto(task);
}

public async Task UpdateTaskAsync(int id, UpdateTaskDto
updateTaskDto)
{
    var task = await _taskRepository.GetByIdAsync(id);
    if (task is null)
        throw new ArgumentException("Task not found");

    task.Title = updateTaskDto.Title;

    if (updateTaskDto.Description is not null)
        task.Description = updateTaskDto.Description;

    // Обновляем статус ТОЛЬКО если значение передано
    if (updateTaskDto.IsCompleted.HasValue)
    {
        if (updateTaskDto.IsCompleted.Value && !task.IsCompleted)
        {
            task.MarkAsCompleted();
        }
        else if (!updateTaskDto.IsCompleted.Value && task.IsCompleted)
        {
            task.MarkAsIncomplete();
        }
        // Если статус уже такой — ничего не делаем
    }
}
```

```
// Всегда обновляем дату изменения при любом обновлении
task.UpdatedAt = DateTime.UtcNow;

task.Validate();
await _taskRepository.UpdateAsync(task);
}

public async Task DeleteTaskAsync(int id)
{
    if (!await _taskRepository.ExistsAsync(id))
        throw new ArgumentException("Task not found");

    await _taskRepository.DeleteAsync(id);
}

public async Task MarkTaskAsCompletedAsync(int id)
{
    await UpdateTaskStatusAsync(id, true);
}

public async Task MarkTaskAsIncompleteAsync(int id)
{
    await UpdateTaskStatusAsync(id, false);
}

        public          async          Task<IEnumerable<TaskItemDto>>>
SearchTasksAsync(string searchTerm)
    {
        var tasks = await _taskRepository.SearchTasksAsync(searchTerm);
        return tasks.Select(MapToDto);
    }

public async Task<int> GetTasksCountAsync()
{
```

```

    return await _taskRepository.GetCountAsync();
}

// Внутренний метод для избежания дублирования
private async Task UpdateTaskStatusAsync(int id, bool isCompleted)
{
    var task = await _taskRepository.GetByIdAsync(id);
    if (task is null)
        throw new ArgumentException("Task not found");

    if (isCompleted && !task.IsCompleted)
    {
        task.MarkAsCompleted();
    }
    else if (!isCompleted && task.IsCompleted)
    {
        task.MarkAsIncomplete();
    }
    else
    {
        // Статус уже такой — всё равно обновляем дату?
        // По желанию: можно пропустить, или обновить UpdatedAt
        task.UpdatedAt = DateTime.UtcNow;
    }

    task.Validate();
    await _taskRepository.UpdateAsync(task);
}

private static TaskItemDto MapToDto(TaskItem task) => new()
{
    Id = task.Id,
    Title = task.Title,
    Description = task.Description,

```

```
        IsCompleted = task.IsCompleted,  
        CreatedAt = task.CreatedAt,  
        UpdatedAt = task.UpdatedAt  
    };  
}
```

2.5. Организация внедрения зависимостей

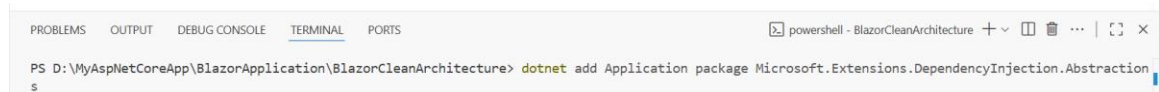
В файле `Application/DependencyInjection.cs` мы видим статический класс с методом расширения `AddApplicationServices`. Это **лучшая практика** для организации внедрения зависимостей. Вместо того чтобы регистрировать все сервисы `Application` напрямую в `Program.cs`, мы инкапсулируем их регистрацию в этом методе. Это делает `Program.cs` более чистым, модульным и легко читаемым, поскольку он теперь содержит вызовы только на уровне слоев, а не отдельных сервисов.

Убедись, что проект **`Application.csproj`** имеет ссылку на `Microsoft.Extensions.DependencyInjection.Abstractions`.

Добавить можно так:

```
dotnet add Application package  
Microsoft.Extensions.DependencyInjection.Abstractions
```

Этот пакет содержит интерфейс `IServiceCollection`.

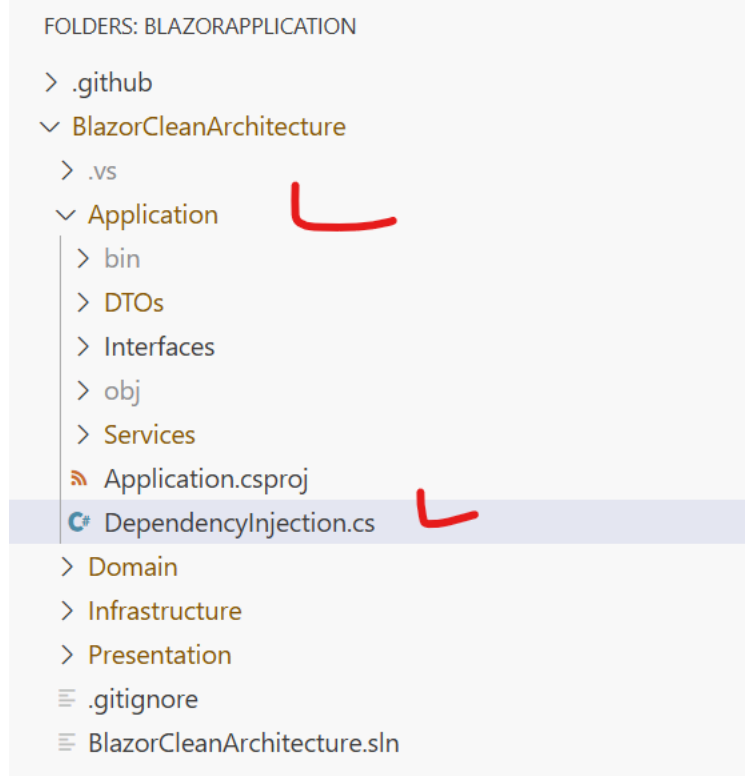


```
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet add Application package Microsoft.Extensions.DependencyInjection.Abstractions
```

```
using Application.Interfaces;  
using Application.Services;  
using Microsoft.Extensions.DependencyInjection;  
  
namespace Application;  
  
public static class DependencyInjection  
{  
    public static IServiceCollection AddApplicationServices(this  
IServiceCollection services)  
    {
```



```
services.AddScoped<ITaskService, TaskService>();  
return services;  
}  
}
```



Часть III: Внешний мир — Слой инфраструктуры (Infrastructure Layer)

3.1. Конкретные реализации

Слой Infrastructure — это самый внешний круг нашей архитектуры. Его основная задача — реализация абстракций, определенных в слоях Domain и Application. Он отвечает за работу с внешними системами, такими как базы данных, файловые системы, сторонние API и сервисы. Этот слой содержит конкретные, зависящие от технологий реализации, которые внутренние слои никогда не должны знать.

3.2. Хранение данных с помощью Entity Framework Core и SQLite

Для нашего проекта мы выбрали SQLite. Это отличный выбор для разработки и небольших приложений, поскольку это простая,

бессерверная, файловая база данных, не требующая сложной настройки. Принципиальный момент здесь заключается в том, что благодаря нашей архитектуре, мы можем легко «поменять порт» — заменить SQLite на более мощную СУБД, такую как SQL Server или PostgreSQL, с минимальными изменениями в коде других слоев. Нам потребуется лишь изменить строку подключения и, возможно, несколько пакетов NuGet, но логика в слоях Application и Domain останется нетронутой.

3.3. Контекст базы данных — ApplicationDbContext

Класс ApplicationDbContext является мостом между Entity Framework Core и нашими доменными сущностями. Он наследуется от DbContext и определяет набор данных (DbSet<TaskItem>), который соответствует сущности TaskItem из нашего домена.

Метод OnModelCreating используется для настройки того, как Entity Framework будет отображать нашу C# модель на схему базы данных. Мы используем его, чтобы явно указать, что свойство Title является обязательным (IsRequired()) и имеет максимальную длину 200 символов (HasMaxLength(200)). Новая строка entity.Property(e => e.CreatedAt).HasDefaultValueSql("datetime('now')") является отличным примером того, как инфраструктурный слой инкапсулирует специфические детали. Мы не меняем доменную модель, чтобы она знала о том, как работать с системным временем базы данных; вместо этого мы настраиваем Entity Framework в слое Infrastructure для автоматической установки этого значения. Это еще раз подчеркивает принцип разделения ответственности.

Следующая таблица наглядно демонстрирует связь между нашей доменной сущностью и итоговой схемой базы данных, которая будет сгенерирована с помощью Entity Framework Core.

Свойство TaskItem	Тип данных (C#)	Ограничения EF Core	Отображение в схеме SQLite
Id	int	HasKey(e => e.Id)	INTEGER PRIMARY KEY

Свойство TaskItem	Тип данных (C#)	Ограничения EF Core	Отображение в схеме SQLite
			AUTOINCREMENT
Title	String	IsRequired(), HasMaxLength(200)	TEXT NOT NULL, VARCHAR(200)
Description	String	HasMaxLength(1000)	TEXT, VARCHAR(1000)
IsCompleted	Bool	По умолчанию	INTEGER (0 или 1)
CreatedAt	DateTime	HasDefaultValueSql (...)	TEXT (ISO 8601)
UpdatedAt	DateTime?	По умолчанию	TEXT

```

using Domain.Entities;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Data;

public class ApplicationDbContext : DbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }

    public DbSet<TaskItem> Tasks => Set<TaskItem>();

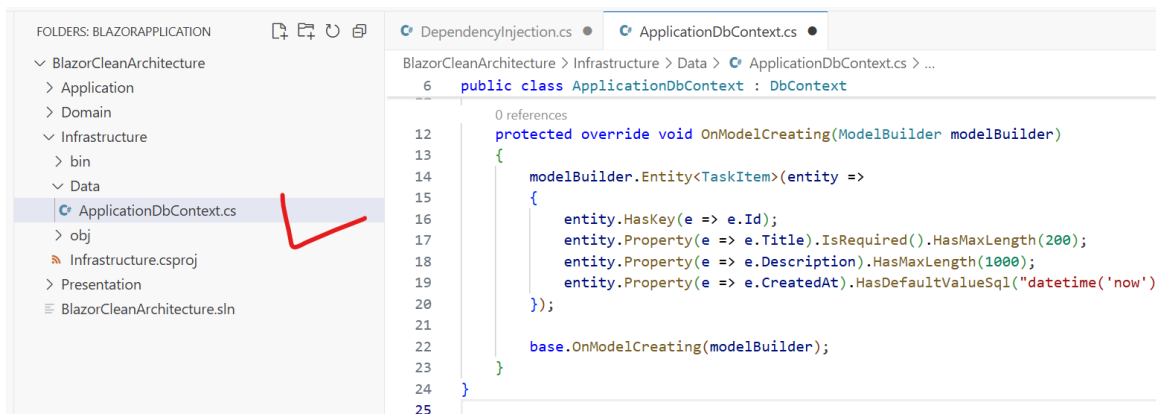
```

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<TaskItem>(entity =>
    {
        entity.HasKey(e => e.Id);
        entity.Property(e => e.Title).IsRequired().HasMaxLength(200);
        entity.Property(e => e.Description).HasMaxLength(1000);
        entity.Property(e
e.CreatedAt).HasDefaultValueSql("datetime('now')");
    });

    base.OnModelCreating(modelBuilder);
}
}

```



3.4. Реализация репозитория — TaskRepository

Класс `TaskRepository` является конкретной реализацией интерфейса `ITaskRepository`, определенного в слое `Domain`. Именно здесь мы предоставляем конкретный способ выполнения операций с данными. Класс принимает `ApplicationDbContext` через внедрение зависимостей и использует его для запросов к базе данных.

При анализе методов, таких как `GetAllAsync()` или `GetCompletedTasksAsync()`, обратите внимание на использование метода `.AsNoTracking()`. Это не случайная деталь, а тонкая, но важная

оптимизация. Entity Framework по умолчанию включает **систему отслеживания изменений**, которая отслеживает все сущности, загруженные в контекст, чтобы знать, какие из них нужно обновить при вызове SaveChangesAsync(). Для операций, которые только считывают данные (например, для отображения в UI) и не будут их изменять, эта система не нужна. Использование .AsNoTracking() отключает ее, что приводит к снижению потребления памяти и увеличению производительности для операций чтения.

Новые методы SearchTasksAsync и ExistsAsync также являются частью этой реализации. SearchTasksAsync использует методы Linq (Where, Contains) для фильтрации данных на уровне базы данных, что намного эффективнее, чем загрузка всех записей в память и их последующая фильтрация. ExistsAsync использует AnyAsync, что также является оптимизированным запросом, который просто проверяет наличие записи, не загружая ее.

```
using Domain.Entities;
using Domain.Interfaces;
using Infrastructure.Data;
using Microsoft.EntityFrameworkCore;

namespace Infrastructure.Repositories;

public class TaskRepository : ITaskRepository
{
    private readonly ApplicationDbContext _context;

    public TaskRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<TaskItem?> GetByIdAsync(int id)
    {
        return await _context.Tasks.FindAsync(id);
    }
}
```

```

    }

    public async Task<IEnumerable<TaskItem>> GetAllAsync()
    {
        return await _context.Tasks
            .AsNoTracking()
            .OrderByDescending(t => t.CreatedAt)
            .ToListAsync();
    }

    public async Task<IEnumerable<TaskItem>>
    GetCompletedTasksAsync()
    {
        return await _context.Tasks
            .Where(t => t.IsCompleted)
            .AsNoTracking()
            .OrderByDescending(t => t.CreatedAt)
            .ToListAsync();
    }

    public async Task<IEnumerable<TaskItem>> GetPendingTasksAsync()
    {
        return await _context.Tasks
            .Where(t => !t.IsCompleted)
            .AsNoTracking()
            .OrderByDescending(t => t.CreatedAt)
            .ToListAsync();
    }

    public async Task<IEnumerable<TaskItem>> SearchTasksAsync(string
searchTerm)
    {
        return await _context.Tasks

```

```
                .Where(t => t.Title.Contains(searchTerm) ||
t.Description.Contains(searchTerm))
                .AsNoTracking()
                .OrderByDescending(t => t.CreatedAt)
                .ToListAsync();
    }

    public async Task AddAsync(TaskItem entity)
    {
        await _context.Tasks.AddAsync(entity);
        await _context.SaveChangesAsync();
    }

    public async Task UpdateAsync(TaskItem entity)
    {
        _context.Tasks.Update(entity);
        await _context.SaveChangesAsync();
    }

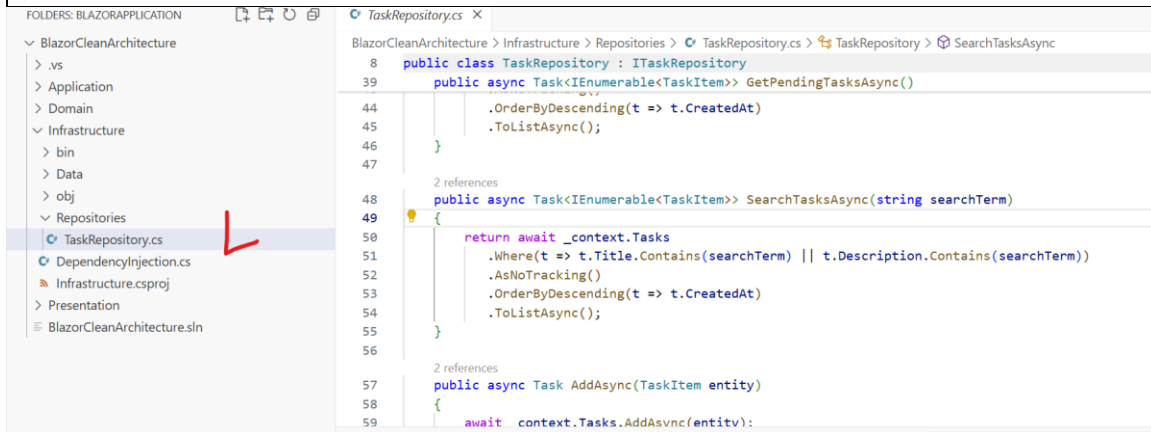
    public async Task DeleteAsync(int id)
    {
        var entity = await _context.Tasks.FindAsync(id);
        if (entity != null)
        {
            _context.Tasks.Remove(entity);
            await _context.SaveChangesAsync();
        }
    }

    public async Task<bool> ExistsAsync(int id)
    {
        return await _context.Tasks.AnyAsync(t => t.Id == id);
    }
}
```

```

public async Task<int> GetCountAsync()
{
    return await _context.Tasks.CountAsync();
}
}

```



3.5. Управление зависимостями — метод расширения AddInfrastructure

В файле Infrastructure/DependencyInjection.cs мы видим статический класс с методом расширения AddInfrastructure. Это **лучшая практика** для организации внедрения зависимостей. Вместо того чтобы регистрировать все сервисы Infrastructure напрямую в Program.cs, мы инкапсулируем их регистрацию в этом методе. Это делает Program.cs более чистым, модульным и легко читаемым, поскольку он теперь содержит вызовы только на уровне слоев, а не отдельных сервисов.

```

using Domain.Interfaces;
using Infrastructure.Data;
using Infrastructure.Repositories;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace Infrastructure;

public static class DependencyInjection
{

```



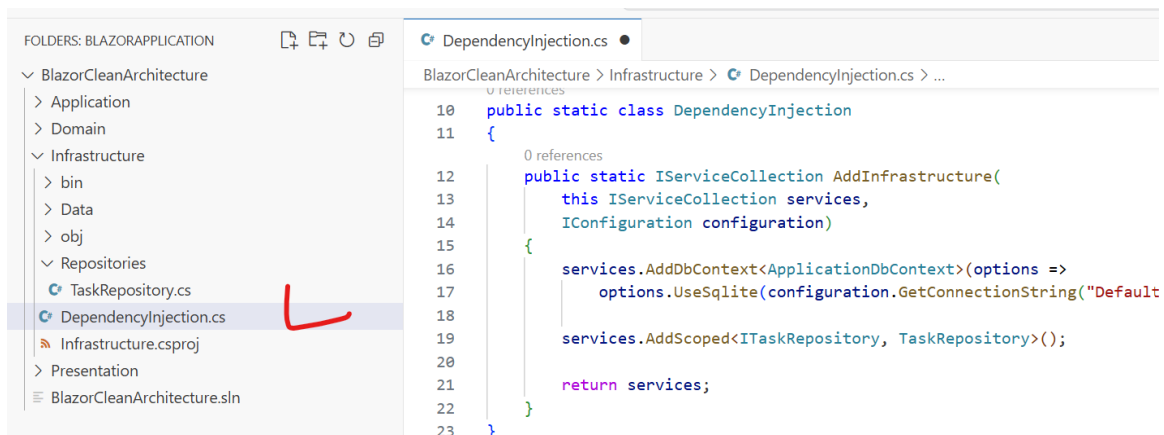
```

public static IServiceCollection AddInfrastructure(
    this IServiceCollection services,
    IConfiguration configuration)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseSqlite(configuration.GetConnectionString("DefaultCon
nection")));

    services.AddScoped<ITaskRepository, TaskRepository>();

    return services;
}
}

```



Часть IV: Пользовательский интерфейс — Слой представления (Presentation Layer)

4.1. Корень композиции — Program.cs

Слой Presentation является самым внешним кругом, точкой входа в наше приложение. Его единственная задача — обработка взаимодействия с пользователем и делегирование задач в слой Application.

Файл Program.cs играет здесь ключевую роль, выступая в качестве **Корня композиции**. Это единственное место во всем приложении, где мы явно связываем интерфейсы с их конкретными реализациями. Мы регистрируем сервисы из слоев Application и Infrastructure.

Обратите внимание, как работает внедрение зависимостей в Program.cs:

1. Мы регистрируем сервисы слоя Application вызовом `builder.Services.AddApplicationServices()`.
2. Мы регистрируем сервисы слоя Infrastructure вызовом `builder.Services.AddInfrastructure()`.
3. И, наконец, мы регистрируем `TaskService`, предоставляя ему интерфейс `ITaskService`.

Это явное связывание является финальным и самым видимым доказательством **Правила зависимостей**. Слой Presentation зависит от обоих внутренних слоев, в то время как слои Application и Infrastructure не имеют никаких ссылок на Presentation. Эта односторонняя зависимость физически закреплена в структуре проекта и гарантирует, что наше ядро остается чистым и независимым.

```
using Application;
using Application.Interfaces;
using Application.Services;
using Infrastructure;
using Infrastructure.Data;
using Microsoft.EntityFrameworkCore;
using Presentation.Components; // App.razor

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

// Register layers
builder.Services.AddApplicationServices();
builder.Services.AddInfrastructure(builder.Configuration);

// Register Application services
builder.Services.AddScoped<ITaskService, TaskService>();
```

```
var app = builder.Build();

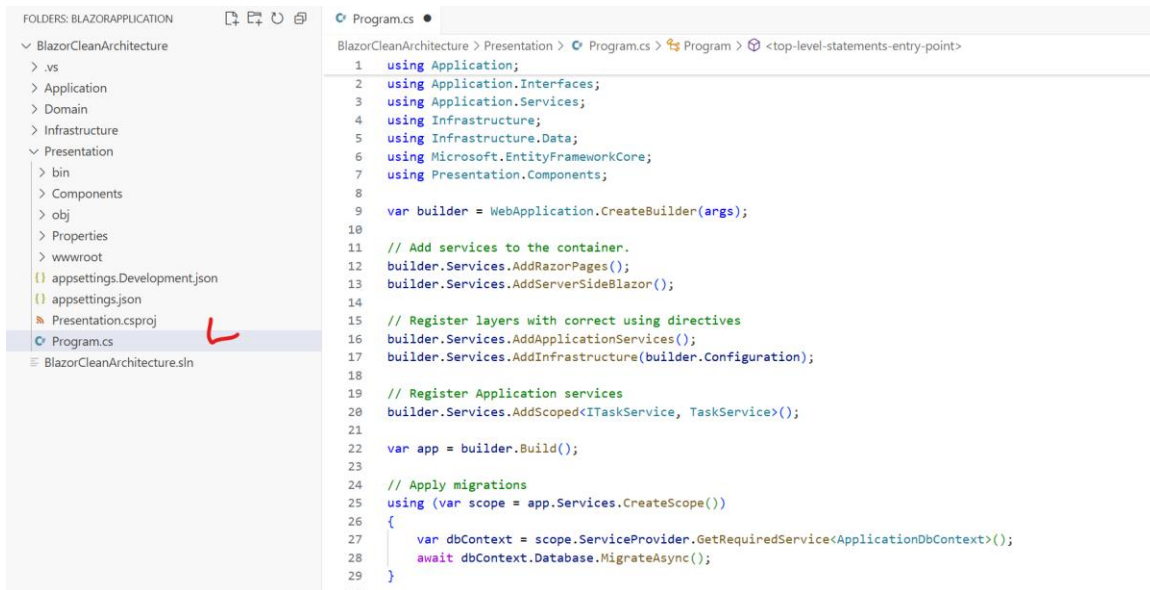
// Apply migrations
using (var scope = app.Services.CreateScope())
{
    var dbContext =
scope.ServiceProvider.GetRequiredService<ApplicationDbContext>();
    await dbContext.Database.MigrateAsync();
}

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAntiforgery();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.Run();
```



Часть V: Операционные и рабочие аспекты

5.1. Миграции базы данных с помощью EF Core

Для управления схемой базы данных мы используем миграции Entity Framework Core. Команды `dotnet ef migrations add` и `dotnet ef database update` позволяют нам создавать и применять изменения в структуре базы данных в контролируемом, версионизируемом виде. Это гарантирует, что схема базы данных всегда соответствует модели, определенной в нашем `DbContext`, и упрощает развертывание обновлений в различных средах. Обратите внимание, что мы теперь используем `dbContext.Database.MigrateAsync()`, который является асинхронной версией метода, что является хорошей практикой.

Создаем миграцию

```
dotnet ef migrations add InitialCreate --project Infrastructure --startup-project Presentation --output-dir Data/Migrations
```

Применяем миграцию

```
dotnet ef database update --project Infrastructure --startup-project Presentation
```



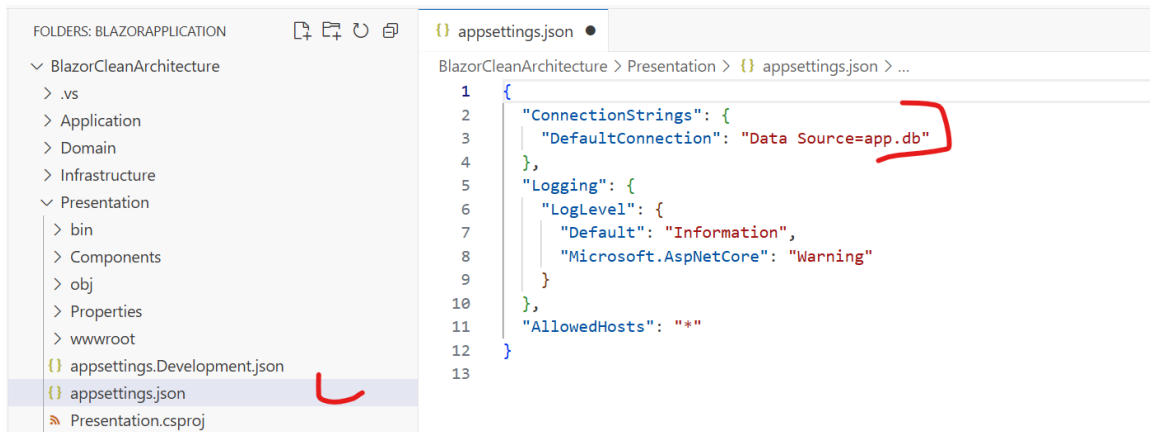
```
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet ef migrations add InitialCreate --project Infrastructure --startup-project Presentation --output-dir Data/Migrations
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet ef database update --project Infrastructure --startup-project Presentation
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Migrations[20411]
Acquiring an exclusive lock for migration application. See https://aka.ms/efcore-docs-migrations-lock for more information if this takes too long.
```

5.2. Управление конфигурацией

Управление конфигурацией в приложении реализовано в соответствии с рекомендованными практиками платформы ASP.NET Core и осуществляется с использованием иерархической системы конфигурационных файлов. Основным механизмом хранения параметров приложения является файл `appsettings.json`, размещённый на уровне Presentation-слоя архитектуры. Данный подход обеспечивает централизованное и гибкое управление настройками, позволяя легко адаптировать поведение приложения под различные среды выполнения без необходимости модификации исходного кода или повторной компиляции.

В файле `appsettings.json` определены ключевые параметры конфигурации, включая строку подключения к базе данных, настройки логирования и список разрешённых хостов:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=app.db"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```



Параметр `ConnectionStrings:DefaultConnection` задаёт путь к локальному файлу базы данных SQLite (`app.db`), используемому в текущей конфигурации. Хранение строки подключения в конфигурационном файле, а не в коде, соответствует принципу внешней конфигурации (*externalized configuration*), что является важным требованием при разработке масштабируемых и поддерживаемых приложений. Это позволяет изменять параметры подключения в зависимости от окружения — разработки, тестирования или производственной среды — без вмешательства в программную логику.

Для поддержки различных сред выполнения ASP.NET Core предоставляет механизм конфигурационных файлов с суффиксами, соответствующими имени среды, например `appsettings.Development.json`, `appsettings.Production.json`. Система конфигурации автоматически загружает базовый файл `appsettings.json`, после чего применяет переопределения из файла, соответствующего текущему значению переменной окружения `ASPNETCORE_ENVIRONMENT`. Таким образом, обеспечивается приоритет конфигурации в порядке: базовая → среда-специфичная → внешние источники.

Ниже приведён пример файла `appsettings.Development.json`, предназначенного для окружения разработки:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=app.dev.db"
  },
  "Logging": {
```

```
"LogLevel": {  
  "Default": "Debug",  
  "Microsoft.AspNetCore": "Trace"  
},  
"DetailedErrors": true,  
"Host": {  
  "EnableDeveloperExceptionPage": true  
}
```

В данном файле повышен уровень детализации логирования до Debug и Trace для диагностики ошибок в процессе разработки, указана альтернативная строка подключения к тестовой базе данных, а также активированы режимы отладки, такие как Developer Exception Page, позволяющие получать подробные сообщения об исключениях.

Кроме файловой конфигурации, ASP.NET Core поддерживает использование других провайдеров конфигурации, включая переменные среды, пользовательские профили, аргументы командной строки и внешние хранилища секретов (secret stores). В производственных условиях рекомендуется использовать защищённые механизмы хранения конфиденциальных данных, такие как Azure Key Vault, AWS Secrets Manager или Hashicorp Vault, что минимизирует риски утечки учётных данных и соответствует требованиям информационной безопасности (например, стандартам OWASP, ISO/IEC 27001).

Также допускается использование переменных окружения для переопределения параметров, что особенно актуально при развертывании в контейнеризованных средах (например, Docker, Kubernetes). Например, строку подключения можно задать через переменную:

```
ConnectionStrings__DefaultConnection="Server=localhost;Database=  
myapp;User=sa;Password=secure;"
```

Обратите внимание, что в именах переменных окружения двойное подчёркивание (__) интерпретируется как разделитель уровней вложенных JSON-объектов.

Таким образом, реализованная система управления конфигурацией обеспечивает:

- Гибкость — возможность адаптации приложения под различные среды;
- Безопасность — исключение хранения чувствительных данных в открытом виде в репозитории;
- Поддерживаемость — простота модификации и расширения конфигурации без изменения кода;
- Соответствие современным практикам DevOps и CI/CD — интеграция с пайплайнами развертывания.

Использование многоуровневой конфигурации является неотъемлемой частью построения надёжных и масштабируемых приложений и соответствует лучшим практикам разработки на платформе .NET.

Часть VI: Blazor — Реализация полноценного CRUD функционала

Теперь, когда наше ядро и инфраструктура готовы, мы можем перейти к созданию полноценного пользовательского интерфейса с использованием Blazor. Мы добавили несколько компонентов для реализации полного цикла CRUD (Create, Read, Update, Delete), а также поиск и фильтрацию.

6.1. Главная страница со списком задач — Tasks.razor

Этот компонент является основным интерфейсом нашего приложения. Он демонстрирует, как мы используем `ITaskService` для получения, фильтрации и управления задачами. Обратите внимание, что весь UI-код работает исключительно с DTO (`TaskItemDto`) и не имеет представления о доменных сущностях или о том, как данные сохраняются. Это — ключевой аспект Clean Architecture в действии. Компонент `@inject ITaskService TaskService` получает экземпляр сервиса через внедрение зависимостей и использует его для всех операций.

Presentation/Components/Tasks.razor

@page "/tasks"

```

@using Application.DTOs
@using Application.Interfaces
@inject ITaskService TaskService
@inject NavigationManager Navigation
@inject IJSRuntime JSRuntime
@using Microsoft.AspNetCore.Components.Forms
@using System.ComponentModel.DataAnnotations
@rendermode RenderMode.InteractiveServer

<h3>Task Management</h3>

<div class="d-flex justify-content-between align-items-center mb-3">
    <div class="btn-group">
        <button @onclick="() => ApplyFilter(null)" class="btn
@((currentFilter == null ? "btn-primary" : "btn-outline-primary"))">
            All (@totalCount)
        </button>
        <button @onclick="() => ApplyFilter(active)" class="btn
@((currentFilter == "active" ? "btn-success" : "btn-outline-success"))">
            Active (@activeCount)
        </button>
        <button @onclick="() => ApplyFilter(completed)" class="btn
@((currentFilter == "completed" ? "btn-warning" : "btn-outline-warning"))">
            Completed (@completedCount)
        </button>
    </div>

    <div class="d-flex gap-2">
        <div class="input-group" style="width: 300px;">
            <input value="@searchTerm"
                @oninput="async (e) => await HandleSearchInput(e)"
                class="form-control" placeholder="Search tasks..." />
            <button @onclick="ClearSearch" class="btn btn-outline-secondary"
title="Clear search">

```

```

        <i class="bi bi-x"></i>
    </button>
</div>

<!-- Кнопка создать -->
<button @onclick="OpenCreateForm" class="btn btn-success">
    <i class="bi bi-plus-circle"></i> Create Task
</button>
</div>
</div>

@if (showCreateForm)
{
    <div class="card mb-3">
        <div class="card-header bg-primary text-white">
            <h5 class="mb-0">Create New Task</h5>
        </div>
        <div class="card-body">
            <EditForm Model="@newTask" OnValidSubmit="@CreateTask"
FormName="createTaskForm">
                <DataAnnotationsValidator />
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label for="title" class="form-label">Title *</label>
                        <InputText id="title" @bind-Value="@newTask.Title"
class="form-control" />
                        <ValidationMessage For="@(() => newTask.Title)" />
                    </div>
                    <div class="col-md-6 mb-3">
                        <label for="description" class="form-
label">Description</label>
                        <InputTextArea id="description" @bind-
Value="@newTask.Description" class="form-control" />
                    </div>
                </div>
            </EditForm>
        </div>
    </div>
}

```

```

</div>
<div class="d-flex gap-2">
    <button type="submit" class="btn btn-success">
        <i class="bi bi-check-circle"></i> Create
    </button>
    <button type="button" @onclick="CancelCreate" class="btn
btn-secondary">
        <i class="bi bi-x-circle"></i> Cancel
    </button>
</div>
</EditForm>
</div>
</div>
}

@if (showEditForm && editingTask != null)
{
    <div class="card mb-3">
        <div class="card-header bg-warning text-white">
            <h5 class="mb-0">Edit Task</h5>
        </div>
        <div class="card-body">
            <EditForm      Model="@editingTask"
OnValidSubmit="@UpdateTask" FormName="editTaskForm">
                <DataAnnotationsValidator />
                <div class="row">
                    <div class="col-md-6 mb-3">
                        <label for="edit-title" class="form-label">Title *</label>
                        <InputText id="edit-title" @bind-
Value="@editingTask.Title" class="form-control" />
                        <ValidationMessage For="@(() => editingTask.Title)" />
                    </div>
                    <div class="col-md-6 mb-3">

```

```

        <label for="edit-description" class="form-label">Description</label>
        <InputTextArea id="edit-description" @bind-Value="@editingTask.Description" class="form-control" />
    </div>
</div>
<div class="form-check mb-3">
    <InputCheckbox id="edit-completed" @bind-Value="@editingTask.IsCompleted" class="form-check-input" />
    <label for="edit-completed" class="form-check-label">Completed</label>
</div>
<div class="d-flex gap-2">
    <button type="submit" class="btn btn-success">
        <i class="bi bi-check-circle"></i> Update
    </button>
    <button type="button" @onclick="CancelEdit" class="btn btn-secondary">
        <i class="bi bi-x-circle"></i> Cancel
    </button>
</div>
</EditForm>
</div>
</div>
}

@if (tasks == null)
{
    <div class="text-center py-5">
        <div class="spinner-border text-primary" role="status">
            <span class="visually-hidden">Loading...</span>
        </div>
        <p class="mt-2">Loading tasks...</p>
    </div>
}

```

[illegible]

```

        <i class="bi bi-circle text-warning" title="Pending"></i>
    }
</td>
<td class="fw-bold">@task.Title</td>
<td>@task.Description</td>
<td>@task.CreatedAt.ToString("MMM dd, yyyy")</td>
<td>
    <div class="btn-group">
        <button @onclick="async () => await
ToggleTaskStatus(task.Id)"
            class="btn @(task.IsCompleted ? "btn-warning" :
"btn-success")"
            title="@ (task.IsCompleted ? "Mark as Incomplete"
: "Mark as Complete")">
            <i class="bi @(task.IsCompleted ? "bi-x-circle" : "bi-
check-circle")"></i>
            @(task.IsCompleted ? "Undo" : "Complete")
        </button>
        <button @onclick="() => EditTask(task)"
            class="btn btn-primary"
            title="Edit">
            <i class="bi bi-pencil"></i> Edit
        </button>
        <button @onclick="async () => await
DeleteTask(task.Id)"
            class="btn btn-danger"
            title="Delete">
            <i class="bi bi-trash"></i> Delete
        </button>
    </div>
</td>
</tr>
}
</tbody>

```

```

        </table>
    </div>

    <!-- Pagination -->
    @if (totalPages > 1)
    {
        <nav aria-label="Task pagination">
            <ul class="pagination justify-content-center">
                <li class="page-item @(currentPage == 1 ? "disabled" : "")">
                    <button class="page-link" @onclick="() =>
ChangePage(currentPage - 1)" aria-label="Previous">
                        <span aria-hidden="true">&laquo;</span>
                    </button>
                </li>

                @for (int i = 1; i <= totalPages; i++)
                {
                    <li class="page-item @(i == currentPage ? "active" : "")">
                        <button class="page-link" @onclick="() =>
ChangePage(i)">@i</button>
                    </li>
                }

                <li class="page-item @(currentPage == totalPages ? "disabled" :
"")">
                    <button class="page-link" @onclick="() =>
ChangePage(currentPage + 1)" aria-label="Next">
                        <span aria-hidden="true">&raquo;</span>
                    </button>
                </li>
            </ul>
        </nav>
    }

```

```

<div class="text-center text-muted mt-2">
    Showing @((currentPage - 1) * pageSize + 1) to
    @Math.Min(currentPage * pageSize, filteredTasks.Count) of
    @filteredTasks.Count tasks
</div>
}

@code {
    private IEnumerable<TaskItemDto>? tasks;
    private List<TaskItemDto> filteredTasks = new();
    private List<TaskItemDto> displayedTasks = new();
    private bool showcreateForm = false;
    private bool showeditform = false;
    private TaskItemDto? editingTask = null;
    private CreateTaskDto newTask = new() { Title = "", Description = "" };
    private string? currentFilter = null;
    private string searchTerm = string.Empty;
    private int totalCount = 0;
    private int activeCount = 0;
    private int completedCount = 0;
    private string active = "active";
    private string completed = "completed";

    private int currentPage = 1;
    private int pageSize = 10;
    private int totalPages = 1;

    private CancellationTokenSource? searchCts;
    private readonly TimeSpan searchDelay =
    TimeSpan.FromMilliseconds(300);

    protected override async Task OnInitializedAsync()
    {
        await LoadAllData();
    }
}

```



```

    }

    private async Task LoadAllData()
    {
        await LoadTasks();
        await LoadCounts();
        ApplyFilterAndSearch();
        UpdatePagination();
    }

    private async Task LoadTasks()
    {
        tasks = await TaskService.GetAllTasksAsync();
    }

    private async Task LoadCounts()
    {
        var allTasks = await TaskService.GetAllTasksAsync();
        totalCount = allTasks.Count();
        activeCount = allTasks.Count(t => !t.IsCompleted);
        completedCount = allTasks.Count(t => t.IsCompleted);
    }

    private void ApplyFilterAndSearch()
    {
        if (tasks == null) return;

        filteredTasks = currentFilter switch
        {
            "active" => tasks.Where(t => !t.IsCompleted).ToList(),
            "completed" => tasks.Where(t => t.IsCompleted).ToList(),
            _ => tasks.ToList()
        };
    }

```

```

        if (!string.IsNullOrEmpty(searchTerm))
        {
            filteredTasks = filteredTasks.Where(t =>
                t.Title.Contains(searchTerm,
StringComparison.OrdinalIgnoreCase) ||
                (!string.IsNullOrEmpty(t.Description) &&
                t.Description.Contains(searchTerm,
StringComparison.OrdinalIgnoreCase)))
                .ToList();
        }

        UpdatePagination();
    }

    private void UpdatePagination()
    {
        totalPages    =    (int)Math.Ceiling(filteredTasks.Count    /
(double)pageSize);
        if (currentPage > totalPages && totalPages > 0)
            currentPage = totalPages;
        else if (currentPage < 1)
            currentPage = 1;

        displayedTasks = filteredTasks
            .Skip((currentPage - 1) * pageSize)
            .Take(pageSize)
            .ToList();
    }

    private async Task ApplyFilter(string? filter)
    {
        currentFilter = filter;
        currentPage = 1;
        searchTerm = string.Empty;
    }

```

```
        await LoadAllData();
        StateHasChanged();
    }

    private void ClearSearch()
    {
        searchTerm = string.Empty;
        currentPage = 1;
        ApplyFilterAndSearch();
        StateHasChanged();
    }

    private void ChangePage(int page)
    {
        if (page >= 1 && page <= totalPages)
        {
            currentPage = page;
            UpdatePagination();
            StateHasChanged();
        }
    }

    private void OpenCreateForm()
    {
        newTask = new CreateTaskDto { Title = "", Description = "" };
        showCreateForm = true;
        showEditForm = false;
    }

    private void CancelCreate()
    {
        showCreateForm = false;
    }
}
```

```
private async Task CreateTask()
{
    try
    {
        await TaskService.CreateTaskAsync(newTask);
        showcreateForm = false;
        await LoadAllData();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error creating task: {ex.Message}");
    }
}

private void EditTask(TaskItemDto task)
{
    editingTask = new TaskItemDto
    {
        Id = task.Id,
        Title = task.Title,
        Description = task.Description,
        IsCompleted = task.IsCompleted,
        CreatedAt = task.CreatedAt
    };
    showEditForm = true;
    showcreateForm = false;
}

private void CancelEdit()
{
    showEditForm = false;
    editingTask = null;
}
```

```
private async Task UpdateTask()
{
    if (editingTask == null) return;

    try
    {
        var updateDto = new UpdateTaskDto
        {
            Title = editingTask.Title,
            Description = editingTask.Description,
            IsCompleted = editingTask.IsCompleted
        };

        await TaskService.UpdateTaskAsync(editingTask.Id, updateDto);
        showEditForm = false;
        editingTask = null;
        await LoadAllData();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Error updating task: {ex.Message}");
    }
}

private async Task ToggleTaskStatus(int taskId)
{
    try
    {
        var task = tasks?.FirstOrDefault(t => t.Id == taskId);
        if (task != null)
        {
            if (task.IsCompleted)
                await TaskService.MarkTaskAsIncompleteAsync(taskId);
            else
```

```

        await TaskService.MarkTaskAsCompletedAsync(taskId);

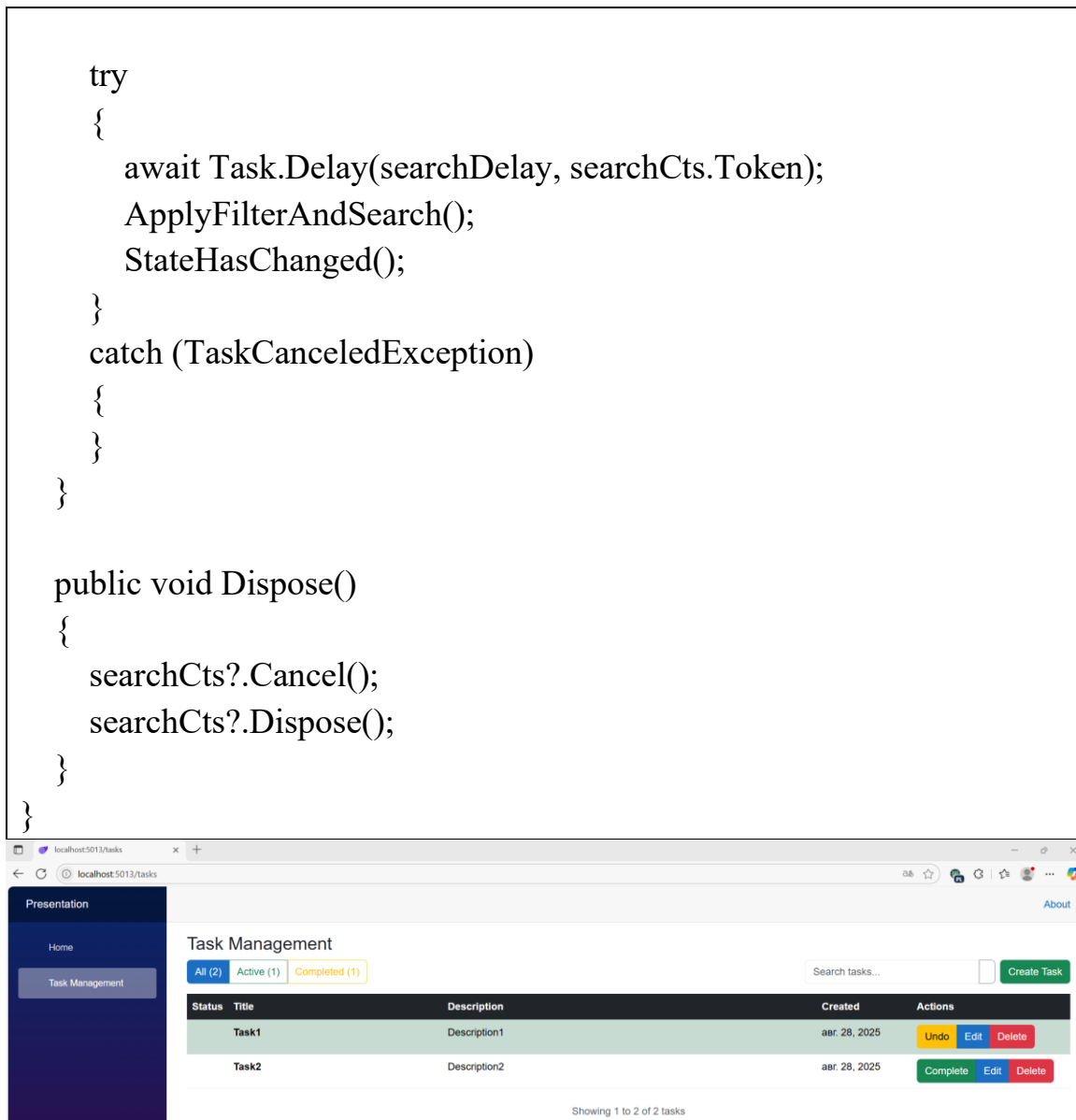
        await LoadAllData();
    }
}
catch (Exception ex)
{
    Console.WriteLine($"Error toggling task status: {ex.Message}");
}
}

private async Task DeleteTask(int taskId)
{
    if (await JSRuntime.InvokeAsync<bool>("confirm", "Are you sure you want to delete this task? This action cannot be undone."))
    {
        try
        {
            await TaskService.DeleteTaskAsync(taskId);
            await LoadAllData();
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error deleting task: {ex.Message}");
        }
    }
}

private async Task HandleSearchInput(EventArgs e)
{
    searchTerm = e.Value?.ToString() ?? string.Empty;

    searchCts?.Cancel();
    searchCts = new CancellationTokenSource();
}

```



6.3. Реализация навигационного интерфейса и визуального оформления с использованием иконок

Для повышения удобства взаимодействия пользователя с приложением и улучшения визуальной составляющей интерфейса в компоненте `Tasks.razor` была реализована навигационная панель действий, а также добавлен пункт меню в боковую навигацию (`NavMenu.razor`). Данные изменения, хотя и носят косметический характер, существенно влияют на юзабилити и воспринимаемое

качество приложения, приближая его к стандартам современных веб-приложений.

Навигационные действия в списке задач

Для каждой задачи в списке реализован групповой набор кнопок, объединённых в элемент управления btn-group с использованием компонентов CSS-фреймворка Bootstrap. Каждая кнопка предоставляет доступ к ключевым операциям: изменение статуса выполнения, редактирование и удаление задачи. Визуальное отображение кнопок адаптируется в зависимости от текущего состояния задачи, что обеспечивает интуитивно понятную обратную связь для пользователя.

```
<div class="btn-group">
  <button @onclick="async () => await ToggleTaskStatus(task.Id)"
    class="btn @(task.IsCompleted ? "btn-warning" : "btn-success")"
    title="@ (task.IsCompleted ? "Mark as Incomplete" : "Mark as
Complete")">
    <i class="bi @(task.IsCompleted ? "bi-x-circle" : "bi-check-
circle")"></i>
    @ (task.IsCompleted ? "Undo" : "Complete")
  </button>
  <button @onclick="() => EditTask(task)"
    class="btn btn-primary"
    title="Edit">
    <i class="bi bi-pencil"></i> Edit
  </button>
  <button @onclick="async () => await DeleteTask(task.Id)"
    class="btn btn-danger"
    title="Delete">
    <i class="bi bi-trash"></i> Delete
  </button>
</div>
```

Каждая кнопка снабжена соответствующей иконкой из библиотеки Bootstrap Icons, что визуально обогащает интерфейс и ускоряет восприятие функциональности. Использование семантически значимых иконок (например, bi-check-circle для завершения задачи, bi-

x-circle для отмены, bi-pencil — редактирование, bi-trash — удаление) соответствует общепринятым паттернам пользовательского интерфейса и способствует снижению когнитивной нагрузки на пользователя.

Цветовая дифференциация кнопок выполнена в соответствии с семантикой действий:

- Зелёный (btn-success) — подтверждение выполнения задачи;
- Жёлтый (btn-warning) — отмена выполнения (перевод в статус "не завершено");
- Синий (btn-primary) — редактирование;
- Красный (btn-danger) — удаление, действие с необратимыми последствиями.

Атрибут title обеспечивает отображение всплывающей подсказки при наведении, что улучшает доступность интерфейса, особенно для пользователей, полагающихся на вспомогательные технологии.

```
<div class="nav-item px-3">  
  <NavLink class="nav-link" href="tasks">  
    <span class="bi bi-list-task" aria-hidden="true"></span> Task  
    Management  
  </NavLink>  
</div>
```

Иконка bi-list-task визуально ассоциируется с управлением списками задач, что способствует быстрому распознаванию назначения пункта меню. Атрибут aria-hidden="true" используется для корректной работы с экранными дикторами — он указывает, что иконка является чисто декоративной и не должна быть озвучена, тогда как текстовая метка "Task Management" обеспечивает доступность для пользователей с нарушениями зрения.

Полная структура NavMenu.razor включает в себя:

- Заголовок приложения (Presentation);
- Ссылку на главную страницу с иконкой bi-house-door-fill;
- Ссылку на раздел управления задачами с соответствующей иконкой;
- Адаптивный механизм раскрытия меню с использованием скрытого чекбокса, что позволяет реализовать поведение "гамбургер-меню" без использования JavaScript.

```

<div class="top-row ps-3 navbar navbar-dark">
  <div class="container-fluid">
    <a class="navbar-brand" href="">Presentation</a>
  </div>
</div>

<input type="checkbox" title="Navigation menu" class="navbar-toggler"
/>

<div class="nav-scrollable" onclick="document.querySelector('.navbar-
toggler').click()">
  <nav class="nav flex-column">
    <div class="nav-item px-3">
      <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
        <span class="bi bi-house-door-fill" aria-hidden="true"></span>
Home
      </NavLink>
    </div>

    <div class="nav-item px-3">
      <NavLink class="nav-link" href="tasks">
        <span class="bi bi-list-task" aria-hidden="true"></span> Task
Management
      </NavLink>
    </div>
  </nav>
</div>

```

Реализованные элементы навигации и визуального оформления способствуют созданию целостного и профессионального пользовательского интерфейса. Использование иконографии из Bootstrap Icons, семантически обоснованной цветовой палитры и доступных атрибутов (title, aria-hidden) соответствует современным стандартам веб-доступности (WCAG) и принципам пользовательского центрированного дизайна. Подобные улучшения, хотя и не влияют на функциональную логику приложения, играют ключевую роль в

формировании положительного впечатления о системе и повышении её эргономических характеристик.

7. Управление миграциями базы данных в Entity Framework Core

Для обеспечения согласованности между объектной моделью предметной области и физической схемой базы данных в проекте используется механизм миграций Entity Framework Core (EF Core). Данный подход позволяет версионировать изменения структуры базы данных, обеспечивая воспроизводимость, контролируемость и безопасность внесения изменений на всех этапах жизненного цикла приложения — от разработки до эксплуатации.

7.1. Создание начальной миграции

На начальном этапе разработки была сформирована первоначальная модель данных, отражающая сущности предметной области (например, Task). Для фиксации этой структуры в виде миграции была выполнена следующая команда в CLI .NET:

```
dotnet ef migrations add InitialCreate --project Infrastructure --startup-project Presentation --output-dir Data/Migrations
```

Пояснение ключевых параметров:

- `migrations add InitialCreate` — создание новой миграции с именем `InitialCreate`, отражающим инициализацию схемы;
- `--project Infrastructure` — указание проекта, содержащего `DbContext` и модели домена;
- `--startup-project Presentation` — определение стартового проекта, необходимого для корректной инициализации конфигурации приложения;
- `--output-dir Data/Migrations` — явное задание каталога для размещения файлов миграций, что способствует структурированности кода и упрощает сопровождение.

В результате выполнения команды в директории `Infrastructure/Data/Migrations` были сгенерированы файлы:

- `<timestamp>_InitialCreate.cs` — основной файл миграции с методами `Up()` (применение изменений) и `Down()` (откат изменений);
- `<timestamp>_InitialCreate.Designer.cs` — автоматически сгенерированный код, содержащий бинарные метаданные миграции;
- Файл входит в состав сборки и используется EF Core для отслеживания состояния базы данных.
 - ✓ Сгенерированный скрипт включает инструкции DDL (Data Definition Language) для создания таблиц, определения первичных и внешних ключей, а также настройки индексов в соответствии с конфигурацией, заданной через Fluent API или атрибуты моделей.

7.2. Применение миграции к базе данных

После создания миграции выполняется обновление целевой базы данных с помощью команды:

```
dotnet ef database update --project Infrastructure --startup-project Presentation
```

Эта команда:

- Подключается к базе данных, указанной в строке подключения (в данном случае — `Data Source=app.db`);
- Проверяет таблицу `__EFMigrationsHistory`, в которой хранится история применённых миграций;
- Применяет все неприменённые миграции в порядке их следования;
- Фиксирует успешное выполнение в таблице миграций.

В случае использования SQLite база данных автоматически создаётся, если файл `app.db` отсутствует. Таким образом обеспечивается полная автономность и переносимость приложения.

7.3. Обновление миграций при изменении модели данных

В процессе разработки структура доменных моделей может изменяться — например, добавляются новые свойства, изменяются типы данных,

вводятся связи между сущностями. В таких случаях необходимо обновить схему базы данных, сохранив при этом существующие данные (если это возможно).

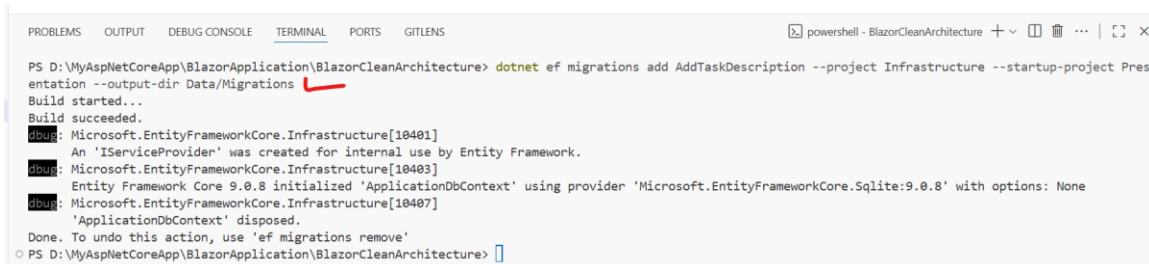
Процедура обновления миграции

1. Внесение изменений в модель. Например, в класс Task была добавлена новая строковая свойство Description:

```
public class Task
{
    public int Id { get; set; }
    public string Title { get; set; } = string.Empty;
    public string? Description { get; set; } // Новое свойство
    public bool IsCompleted { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
}
```

2. Генерация новой миграции
Для фиксации изменений создаётся новая миграция с описательным именем:

```
dotnet ef migrations add AddTaskDescription --project Infrastructure --startup-project Presentation --output-dir Data/Migrations
```



```
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet ef migrations add AddTaskDescription --project Infrastructure --startup-project Presentation --output-dir Data/Migrations
Build started...
Build succeeded.
[debug]: Microsoft.EntityFrameworkCore.Infrastructure[10401]
An 'IServiceProvider' was created for internal use by Entity Framework.
[debug]: Microsoft.EntityFrameworkCore.Infrastructure[10403]
Entity Framework Core 9.0.8 initialized 'ApplicationDbContext' using provider 'Microsoft.EntityFrameworkCore.Sqlite:9.0.8' with options: None
[debug]: Microsoft.EntityFrameworkCore.Infrastructure[10407]
'ApplicationDbContext' disposed.
Done. To undo this action, use 'ef migrations remove'
PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture>
```

В результате будет создан новый файл миграции, содержащий:

- В методе Up() — вызов AddColumn для добавления столбца Description в таблицу Tasks;
- В методе Down() — обратная операция (DropColumn), позволяющая откатить изменения.

3. Применение обновлённой миграции
После проверки корректности сгенерированного скрипта выполняется обновление базы данных:

```
dotnet ef database update --project Infrastructure --startup-project Presentation
```

Если база данных уже содержит данные, EF Core попытается применить изменения без потери информации. Однако в некоторых случаях (например, при изменении типа столбца или удалении обязательного поля) могут потребоваться ручные корректировки миграции или предварительное резервное копирование.

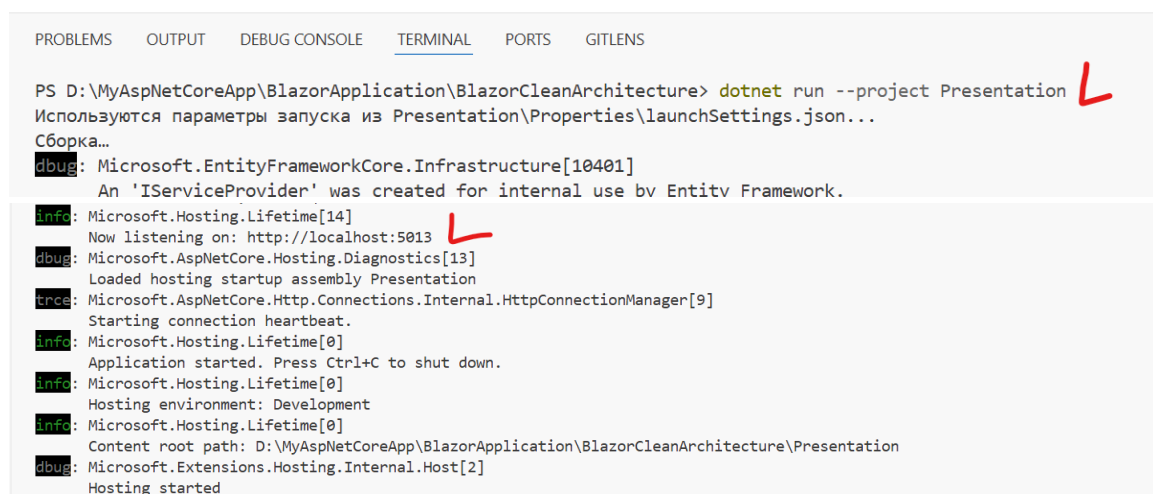
Рекомендации по работе с миграциями

- Используйте осмысленные имена миграций, отражающие суть изменений (например, AddUserEmail, RenameTaskStatusColumn).
- Перед применением миграции в продакшене тестируйте её на копии рабочей базы данных.
- При необходимости можно вручную редактировать сгенерированный код миграции (например, добавить значения по умолчанию, настроить поведение ON DELETE для внешних ключей).
- В CI/CD-пайплайнах применение миграций рекомендуется выполнять через `dotnet ef database update` или с помощью скриптов развертывания.

8. Запуск приложения

После успешного создания и применения миграций приложение может быть запущено с помощью команды:

`dotnet run --project Presentation`



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

PS D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture> dotnet run --project Presentation L
Используются параметры запуска из Presentation\Properties\launchSettings.json...
Сборка...
debug: Microsoft.EntityFrameworkCore.Infrastructure[10401]
      An 'IServiceProvider' was created for internal use by Entity Framework.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5013 L
debug: Microsoft.AspNetCore.Hosting.Diagnostics[13]
      Loaded hosting startup assembly Presentation
trace: Microsoft.AspNetCore.Http.Connections.Internal.HttpConnectionManager[9]
      Starting connection heartbeat.
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\MyAspNetCoreApp\BlazorApplication\BlazorCleanArchitecture\Presentation
debug: Microsoft.Extensions.Hosting.Internal.Host[2]
      Hosting started
```

Данный вызов:

- Компилирует проект Presentation;

- Загружает конфигурацию из appsettings.json и среды-специфичных файлов;
- Настраивает DI-контейнер, регистрируя ApplicationDbContext, сервисы и компоненты;
- Запускает встроенный веб-сервер Kestrel;
- Делает приложение доступным по адресу <https://localhost:5001>.

Пользователь может открыть браузер и перейти в раздел Task Management через навигационное меню, чтобы протестировать функциональность: создание, редактирование, изменение статуса и удаление задач.

Заключение: подведение итогов

Наша лекция подходит к концу. Мы проделали путь от абстрактных архитектурных принципов до конкретной реализации, шаг за шагом построив приложение на основе Clean Architecture.

Давайте еще раз вспомним ключевые моменты:

- **Правило зависимостей** — главный принцип, который заставляет зависимости указывать внутрь, защищая ядро от внешних изменений.
- **Инкапсуляция** — размещение бизнес-логики (MarkAsCompleted(), Validate()) внутри доменных сущностей, что делает их самодостаточными и надежными.
- **Разделение ответственности** — использование DTOs и современных возможностей языка, таких как required и record, для передачи данных через границы слоев, что делает систему более гибкой и безопасной.
- **Внедрение зависимостей** — мощный механизм, который позволяет нам использовать абстракции (интерфейсы) во внутренних слоях и предоставлять их конкретные реализации в самом внешнем слое, в Корне композиции.

Может показаться, что Clean Architecture добавляет избыточную сложность для такого простого приложения. Это правда. Однако эта сложность — это не препятствие, а **стратегическая инвестиция**. Она

создает надежный, гибкий и легко поддерживаемый фундамент. В тот момент, когда ваше приложение начнет расти, когда бизнес-требования станут более сложными, когда вам потребуется сменить базу данных или интерфейс, вы поймете, насколько ценен этот архитектурный подход.

Переход от процедурного к архитектурному мышлению — это один из самых важных шагов в карьере разработчика. Изучение и применение таких паттернов, как Clean Architecture, не только делает вас более компетентным специалистом, но и позволяет создавать программное обеспечение, которое будет служить долго и надежно, выдерживая испытание временем и изменениями.

Источники

1. Clean Architecture with ASP.NET Core | by Thisara dilshan | Medium, дата последнего обращения: августа 25, 2025, https://medium.com/@thisara_dh/clean-architecture-with-asp-net-core-7de60a83edd8
2. Code in Clean vs (Traditional Layered) Architecture .NET - Medium, дата последнего обращения: августа 25, 2025, <https://medium.com/codenx/code-in-clean-vs-traditional-layered-architecture-net-31c4cad8f815>
3. ASP.NET Core Web API using Repository Pattern | by Lokesh prajapat | Medium, дата последнего обращения: августа 25, 2025, <https://medium.com/@lokeshpriajapat742000/asp-net-core-web-api-using-repository-pattern-37479725752a>
4. What is the difference between the clean and the n-tier architectures? - Stack Overflow, дата последнего обращения: августа 25, 2025, <https://stackoverflow.com/questions/22138162/what-is-the-difference-between-the-clean-and-the-n-tier-architectures>
5. Create and Implement 3-Tier Architecture in ASP.Net - C# Corner, дата последнего обращения: августа 25, 2025, <https://www.c-sharpcorner.com/UploadFile/4d9083/create-and-implement-3-tier-architecture-in-Asp-Net/>
6. ASP.NET MVC - Model != BLL or DAL - Dave's Notebook, дата последнего обращения: августа 25, 2025,

<https://davembush.github.io/asp-net-mvc-model-bll-or-dal/>

7. Repository Pattern in ASP.NET Core - Ultimate Guide ..., дата последнего обращения: августа 25, 2025, <https://codewithmukesh.com/blog/repository-pattern-in-aspnet-core/>
8. Repository Pattern in ASP.NET Core REST API - Pragim Tech, дата последнего обращения: августа 25, 2025, <https://www.pragimtech.com/blog/blazor/rest-api-repository-pattern/>
9. neozhu/CleanArchitectureWithBlazorServer: This is a ... - GitHub, дата последнего обращения: августа 25, 2025, <https://github.com/neozhu/CleanArchitectureWithBlazorServer>
10. How to use SQLite in Blazor - All Hands on Tech, дата последнего обращения: августа 25, 2025, <https://www.allhandsontech.com/programming/blazor/how-to-sqlite-blazor/>
11. Blazor & Clean Architecture Masterclass 3h Preview & Special Promo - YouTube, дата последнего обращения: августа 25, 2025, <https://m.youtube.com/watch?v=wW0n4UdjTHA>
12. Add SQLite DB to ASP.NET Core using EF Core code-first - Mobiletonster, дата последнего обращения: августа 25, 2025, <https://mobiletonster.com/blog/code/add-sqlite-db-to-asp.net-core-using-ef-core-code-first>
13. sqlite · GitHub Topics, дата последнего обращения: августа 25, 2025, <https://github.com/topics/sqlite?l=c%23>
14. SQLite Database Provider - EF Core | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/ef/core/providers/sqlite/>
15. How to Implement Entity Framework Core Migrations in a .NET Console Application Using C# and VSCode - Ottorino Bruni, дата последнего обращения: августа 25, 2025, <https://www.ottorinobruni.com/how-to-implement-entity-framework-core-migrations-in-a-dotnet-console-application-using-csharp-and-vscode/>
16. Создавать решения по проектам & - Visual Studio (Windows) | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/ru-ru/visualstudio/ide/creating-solutions->

[and-projects?view=vs-2022](#)

17. Building a Clean Architecture with Blazor, дата последнего обращения: августа 25, 2025, <https://dotnet8.patrickgod.com/posts/building-a-clean-architecture-with-blazor>
18. Use Class Library as BLL, BO and DAL Layers - C# Corner, дата последнего обращения: августа 25, 2025, <https://www.c-sharpcorner.com/blogs/use-class-library-as-bll-bo-and-dal-layers1>
19. Learn to use a 3-tier architecture with C# - Kens Learning Curve, дата последнего обращения: августа 25, 2025, <https://kenslearningcurve.com/tutorials/use-a-3-tier-architecture-with-c/>
20. Clean Architecture in ASP .NET Core Web API | by Mohaned Zekry | Medium, дата последнего обращения: августа 25, 2025, <https://medium.com/@mohanedzekry/clean-architecture-in-asp-net-core-web-api-d44e33893e1d>
21. ASP.NET Web API Repository Pattern Service Layer (Business Logic) - Stack Overflow, дата последнего обращения: августа 25, 2025, <https://stackoverflow.com/questions/22678319/asp-net-web-api-repository-pattern-service-layer-business-logic>
22. Microsoft.EntityFrameworkCore.Sqlite 9.0.8 - NuGet, дата последнего обращения: августа 25, 2025, <https://www.nuget.org/packages/microsoft.entityframeworkcore.sqlite>
23. SQLite in ASP.NET Core with EntityFrameworkCore - Stack Overflow, дата последнего обращения: августа 25, 2025, <https://stackoverflow.com/questions/36488461/sqlite-in-asp-net-core-with-entityframeworkcore>
24. Repository Pattern In ASP.NET Core - C# Corner, дата последнего обращения: августа 25, 2025, <https://www.c-sharpcorner.com/article/repository-pattern-in-asp-net-core/>
25. SQLite & Data Seeding with Entity Framework Core - DEV Community, дата последнего обращения: августа 25, 2025, https://dev.to/_patrickgod/sqlite-data-seeding-with-entity-framework-core-4dna
26. Migrations Overview - EF Core | Microsoft Learn, дата последнего

- обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/>
27. Entity Framework Core Migrations, дата последнего обращения: августа 25, 2025, <https://www.learnentityframeworkcore.com/migrations>
28. How to enable migration in EntityFramework Core Sqlite - Stack Overflow, дата последнего обращения: августа 25, 2025, <https://stackoverflow.com/questions/57145233/how-to-enable-migration-in-entityframework-core-sqlite>
29. Understanding Data Transfer Objects (DTOs) in C# .NET: Best ..., дата последнего обращения: августа 25, 2025, <https://medium.com/@20011002nimeth/understanding-data-transfer-objects-dtos-in-c-net-best-practices-examples-fe3e90238359>
30. Help me understand about DTO in C# .NET : r/csharp - Reddit, дата последнего обращения: августа 25, 2025, https://www.reddit.com/r/csharp/comments/lixlisy/help_me_understand_about_dto_in_c_net/
31. Creating a Business Logic Layer (C#) | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/aspnet/web-forms/overview/data-access/introduction/creating-a-business-logic-layer-cs>
32. asp.net mvc - Difference between Repository and Service Layer? - Stack Overflow, дата последнего обращения: августа 25, 2025, <https://stackoverflow.com/questions/5049363/difference-between-repository-and-service-layer>
33. Build a Blazor movie database app (Part 2 - Add and scaffold a model) | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/aspnet/core/blazor/tutorials/movie-database-app/part-2?view=aspnetcore-9.0>
34. How to Dependency Inject in Blazor Applications - C# Corner, дата последнего обращения: августа 25, 2025, <https://www.c-sharpcorner.com/article/how-to-dependency-inject-in-blazor-applications/>
35. Injecting dependencies into Blazor components, дата последнего обращения: августа 25, 2025, <https://blazor->

university.com/dependency-injection/injecting-dependencies-into-blazor-components/

36. ASP.NET Core fundamentals overview | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-9.0>
37. Blazor Web App 2 program.cs Files And Dependency Injection - Stack Overflow, дата последнего обращения: августа 25, 2025, <https://stackoverflow.com/questions/79486903/blazor-web-app-2-program-cs-files-and-dependency-injection>
38. Understand Dependency Injection in Blazor - C# Corner, дата последнего обращения: августа 25, 2025, <https://www.c-sharpcorner.com/article/understand-dependency-injection-in-blazor/>
39. Blazor Basics: Dependency Injection Best Practices, Use Cases - Telerik.com, дата последнего обращения: августа 25, 2025, <https://www.telerik.com/blogs/blazor-basics-dependency-injection-best-practices-use-cases>
40. ASP.NET Core Blazor dependency injection | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/aspnet/core/blazor/fundamentals/dependency-injection?view=aspnetcore-9.0>
41. ASP.NET Core Blazor with Entity Framework Core (EF Core) | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/aspnet/core/blazor/blazor-ef-core?view=aspnetcore-9.0>
42. Blazor tutorial - Build your first web app | .NET, дата последнего обращения: августа 25, 2025, <https://dotnet.microsoft.com/en-us/learn/aspnet/blazor-tutorial/intro>
43. dotnet sln command - .NET CLI | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-sln>
44. How to Use .NET Core CLI to Create a Multi-Project Solution - NRI, дата последнего обращения: августа 25, 2025, <https://nri-na.com/blog/how-to-use-dot-net-core-cli-create-multi-project/>

45. Tutorial: Create a .NET class library using Visual Studio Code - Microsoft Community, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/dotnet/core/tutorials/library-with-visual-studio-code>
46. Add Class Library In ASP.NET Core Using .NET Core Command-Line Interface (CLI), дата последнего обращения: августа 25, 2025, <https://www.c-sharpcorner.com/article/add-class-library-in-asp-net-core-using-net-core-command-line-interface-cli/>
47. Build and run your first Blazor web app [Pt 2] | Front-end Web Development with .NET for Beginners - YouTube, дата последнего обращения: августа 25, 2025, <https://www.youtube.com/watch?v=llDc88XE--Q>
48. dotnet reference add command - .NET CLI | Microsoft Learn, дата последнего обращения: августа 25, 2025, <https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-reference-add>