# Подробное руководство по созданию CRUD-приложения с использованием Blazor, SQLite и CQRS

### I. Введение в архитектурные основы

#### 1.1. Обзор традиционной CRUD-модели и её ограничений

Традиционная многоуровневая архитектура, основанная на операциях Create, Read, Update, Delete (CRUD), является стандартным подходом к созданию приложений. В этой модели единая структура данных и бизнес-логика используются как для чтения, так и для изменения состояния приложения.1 Например, один и тот же контроллер или сервис может содержать методы GetAll(), GetById(), Create(), Update() и Delete(), а также бизнес-логику, необходимую для выполнения этих операций.

Однако, по мере усложнения бизнес-домена, такая архитектура начинает демонстрировать свои недостатки. Одним из основных ограничений является неэффективное использование ресурсов, особенно в системах с несбалансированной нагрузкой, где запросов на чтение значительно больше, чем на запись.2 В таких случаях весь стек, включая сложную бизнес-логику, должен масштабироваться для удовлетворения потребностей в чтении, что приводит к нерациональному расходу ресурсов.3 Кроме того, смешивание логики чтения и записи в одной модели усложняет код, снижает его читаемость и делает систему более трудной для поддержки. В сложных, мультипользовательских средах это также может приводить к конфликтам блокировок при одновременном доступе к данным.4

На первый взгляд, применение сложного архитектурного паттерна, такого как CQRS, к простой CRUD-операции может показаться избыточным и нецелесообразным.2 Действительно, для простейших приложений, где не требуется высокая производительность и масштабируемость, подобное усложнение может не принести существенных выгод, а лишь увеличит накладные расходы на разработку и поддержку.2 Однако, данное руководство позиционируется как демонстрационный проект, цель которого — не просто создать рабочее приложение, а показать, как принципы CQRS позволяют построить более чистую и масштабируемую структуру. Это практическое упражнение, которое подготавливает разработчика к работе со сложными системами, где разделение ответственности становится не роскошью, а необходимостью.4

#### 1.2. Паттерн CQRS (Command Query Responsibility Segregation)

CQRS — это архитектурный паттерн, предложенный Грегом Янгом, который основан на принципе разделения ответственности команд и запросов (CQS).8 Он строго разделяет модель данных и бизнес-логику на две независимые части 9:

* **Команды (Commands)**. Это объекты, которые инкапсулируют намерение пользователя изменить состояние системы (например, CreateProductCommand, UpdateProductCommand). Они не возвращают данные, а лишь сигнализируют о результате выполнения (успех или неудача).4
* **Обработчики команд (Command Handlers)**. Это классы, которые отвечают за выполнение бизнес-логики, связанной с командой. Они изменяют состояние системы и, как правило, не возвращают данные, кроме Unit (пустой ответ) или идентификатора созданной сущности.11
* **Запросы (Queries)**. Это объекты, предназначенные исключительно для извлечения данных. Они не изменяют состояние системы и не имеют побочных эффектов.9
* **Обработчики запросов (Query Handlers)**. Это классы, которые получают данные из хранилища и проецируют их в специальные объекты-переносчики данных (DTO), оптимизированные для слоя представления (View Model).4

Основными преимуществами CQRS являются 8:

* **Независимое масштабирование.** Стороны чтения и записи могут масштабироваться независимо, что позволяет оптимизировать ресурсы для удовлетворения конкретных требований к нагрузке.4
* **Оптимизация.** Модели чтения могут быть оптимизированы для быстрой выборки данных (например, используя денормализованные представления), а модели записи — для обеспечения целостности данных и безопасности.4
* **Четкое разделение ответственности.** Код становится более модульным, понятным и удобным для поддержки, так как каждый компонент выполняет одну четко определенную задачу.4

Зачастую CQRS тесно связан с концепцией "task-based UI" (интерфейс, ориентированный на задачи), который противопоставляется традиционному "CRUD-based UI".8 В то время как CRUD-интерфейс предоставляет одну и ту же форму для всех операций, task-based UI создает отдельные интерфейсы для каждой конкретной бизнес-задачи (например, "Изменить цену товара" или "Отменить заказ"). Такой подход идеально сочетается с CQRS, так как каждая задача пользователя естественным образом соответствует одной команде в системе. Это позволяет мыслить в терминах бизнес-процессов, что является основой предметно-ориентированного проектирования (DDD).15

#### 1.3. Паттерн Mediator и библиотека MediatR

В архитектуре CQRS возникает проблема координации: как обеспечить, чтобы команда или запрос были направлены к правильному обработчику, не создавая жестких зависимостей? Эту проблему решает паттерн "Посредник" (Mediator).17 Посредник действует как центральная шина, через которую компоненты взаимодействуют друг с другом, не зная о существовании друг друга.11

**MediatR** — это популярная реализация паттерна Mediator для.NET, которая выступает в качестве легковесной, внутрипроцессной шины сообщений.18 Она позволяет 5:

* Определять сообщения-запросы, реализующие интерфейс IRequest<TResponse>.
* Определять обработчики, реализующие IRequestHandler<TRequest, TResponse>.
* Отправлять сообщения с помощью интерфейса IMediator.

С помощью MediatR компоненты, такие как Razor-компоненты или контроллеры, могут просто отправлять сообщения, а MediatR берет на себя ответственность за их доставку соответствующим обработчикам, полностью отделяя отправителя от логики обработки.19

**Таблица 1: Сравнение традиционной CRUD-архитектуры и CQRS**

|  |  |  |
| --- | --- | --- |
| Характеристика | Традиционный CRUD | CQRS (Command Query Responsibility Segregation) |
| **Модель данных** | Единая модель для чтения и записи | Раздельные модели: одна для чтения, другая для записи |
| **Масштабируемость** | Вертикальное масштабирование всего стека | Независимое масштабирование чтения и записи |
| **Сложность** | Простота на начальном этапе, но рост сложности в больших системах | Дополнительная сложность на старте, но упрощение в сложных доменах |
| **Оптимизация** | Сложно оптимизировать, так как операции чтения и записи конкурируют за ресурсы | Чтение и запись могут быть оптимизированы независимо, вплоть до использования разных баз данных |
| **Типичный сценарий использования** | Простые приложения с небольшим количеством функций | Сложные системы с высокой нагрузкой, где требования к чтению и записи существенно различаются |

### II. Подготовка и настройка проекта

#### 2.1. Среда разработки: VS Code и.NET CLI

Для данного руководства используется Visual Studio Code (VS Code) и.NET CLI, так как они предоставляют кросс-платформенную среду разработки.

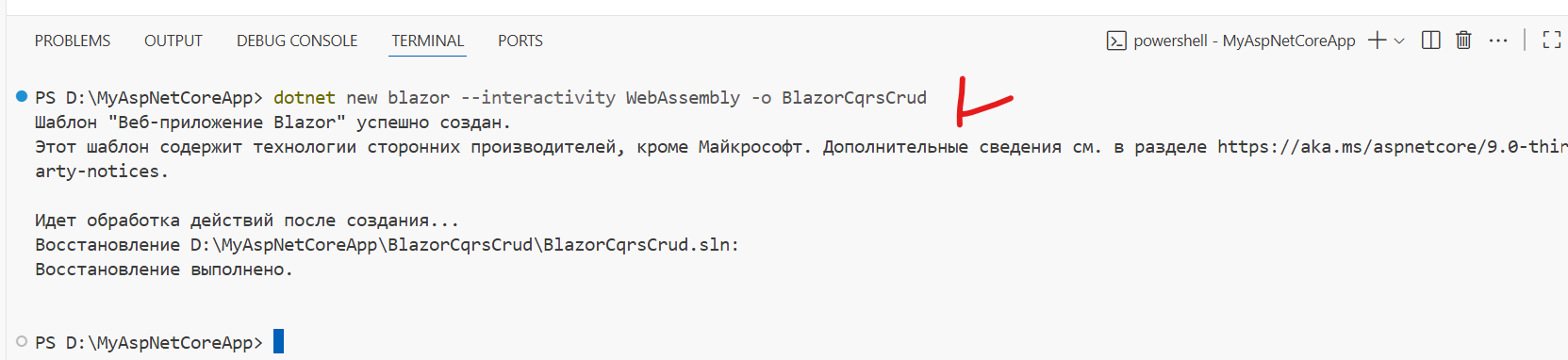
1. **Установка.NET SDK.** Необходимо загрузить и установить.NET 9 SDK (или более новую версию) с официального сайта Microsoft. Для проверки установки следует выполнить команду dotnet --info в терминале.21
2. **Установка VS Code и расширений.** Скачать и установить VS Code. Затем, в магазине расширений VS Code, найти и установить C# Dev Kit.21 Это расширение предоставляет полноценный набор инструментов для разработки на C# и.NET.

#### 2.2. Создание Blazor-проекта

Создание проекта с помощью.NET CLI предоставляет полный контроль над структурой приложения. Для данного руководства будет использоваться шаблон Blazor Web App с опцией WebAssembly-интерактивности. Этот шаблон создает единое решение, содержащее основной проект и дополнительный клиентский проект, который будет компилироваться в WebAssembly, что обеспечивает современный и гибкий подход к разработке.21

Откройте терминал (например, PowerShell или CMD в Windows, Terminal в macOS/Linux) и выполните следующую команду:

dotnet new blazor --interactivity WebAssembly -o BlazorCqrsCrud



Если в проекте отстутвует



Если у нас отсутствует отдельного проекта Shared, нужно немного подкорректировать структуру, чтобы **клиент и сервер могли использовать одни и те же модели и DTO**.

**1. Создание Shared проекта**

1. В корне решения:

dotnet new classlib -o BlazorCqrsCrud.Shared

1. Подключи его к клиенту и серверу:

cd BlazorCqrsCrud

dotnet add reference ../BlazorCqrsCrud.Shared

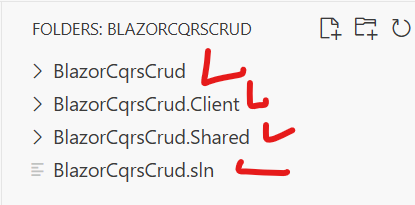
cd ../BlazorCqrsCrud.Client

dotnet add reference ../BlazorCqrsCrud.Shared

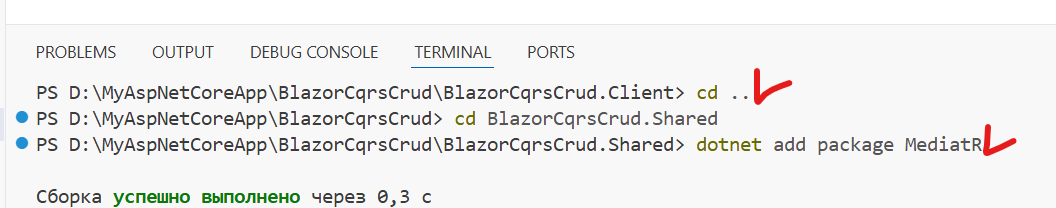


Таким образлом:

* **Shared** — общий проект для моделей и DTO.
* **Server** — хранит DbContext, MediatR handlers и API-контроллеры.
* **Client** — Blazor страницы, которые используют DTO из Shared и обращаются к API сервера.



1. Перейди в папку Shared cd BlazorCqrsCrud.Shared
2. Установи MediatR:



Вот структура проект:

BlazorCqrsCrud/

│

├─ BlazorCqrsCrud.Shared/ ← общие модели и DTO

│ └─ Models/

│ ├─ Product.cs

│ └─ ProductDto.cs

│ └─ Features/

│ └─ Products/

│ ├─ Queries/

│ │ ├─ GetProductListQuery.cs

│ │ └─ GetProductByIdQuery.cs

│ └─ Commands/ ← если будешь добавлять Create/Update/Delete

│

├─ BlazorCqrsCrud.Server/ ← серверная часть

│ ├─ Data/

│ │ ├─ AppDbContext.cs

│ │ └─ AppDbContextFactory.cs

│ └─ Features/

│ └─ Products/

│ ├─ Queries/

│ │ ├─ GetProductListQueryHandler.cs

│ │ └─ GetProductByIdQueryHandler.cs

│ └─ Commands/

│

├─ BlazorCqrsCrud.Client/ ← клиент (Blazor WASM)

│ └─ Pages/

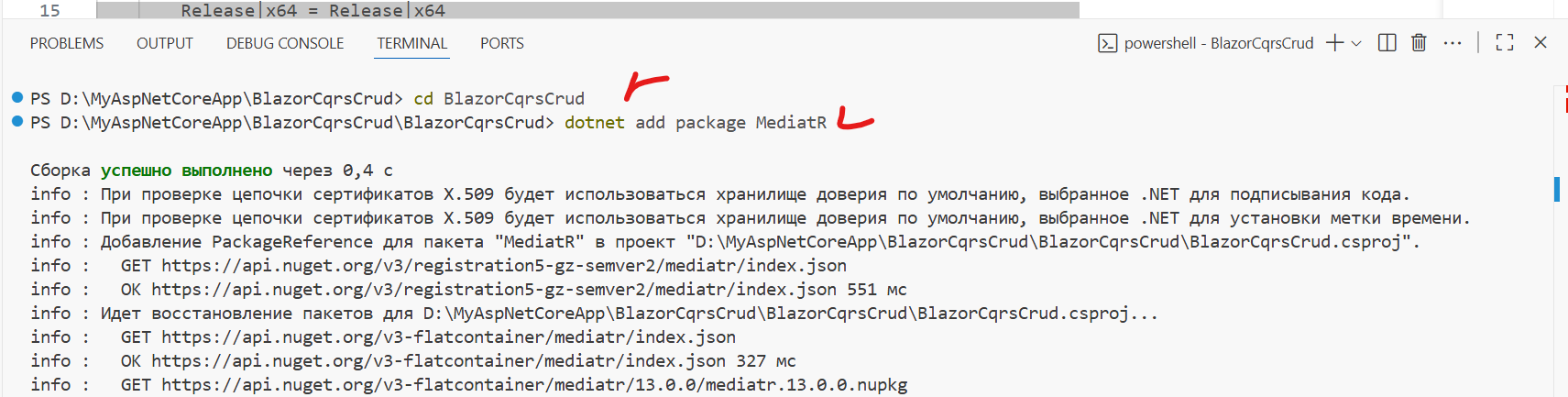
│ └─ Products.razor

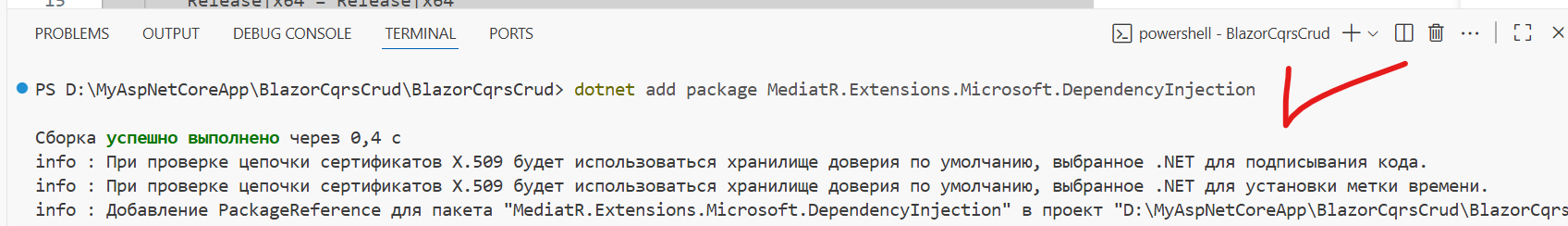
#### 2.3. Управление зависимостями

После создания проекта необходимо добавить необходимые NuGet-пакеты. Перейдите в каталог проекта (cd BlazorCqrsCrud) и выполните следующие команды:

**Таблица 2: Необходимые NuGet-пакеты**

|  |  |  |
| --- | --- | --- |
| Пакет | Команда установки | Назначение |
| MediatR | dotnet add package MediatR | Основная библиотека для реализации паттерна Посредник |
| MediatR.Extensions.Microsoft.DependencyInjection | dotnet add package MediatR.Extensions.Microsoft.DependencyInjection | Расширение для интеграции MediatR с DI-контейнером.NET 5 |
| Microsoft.EntityFrameworkCore.Sqlite | dotnet add package Microsoft.EntityFrameworkCore.Sqlite | Провайдер базы данных EF Core для SQLite 24 |
| Microsoft.EntityFrameworkCore.Tools | dotnet add package Microsoft.EntityFrameworkCore.Tools | Инструменты для работы с миграциями EF Core 25 |
| FluentValidation | dotnet add package FluentValidation | Библиотека для создания гибких правил валидации 26 |
| FluentValidation.DependencyInjectionExtensions | dotnet add package FluentValidation.DependencyInjectionExtensions | Расширение для интеграции FluentValidation с DI 26 |





После установки пакетов необходимо настроить DI-контейнер в файле Program.cs. MediatR предоставляет удобный метод для автоматической регистрации обработчиков и поведений. Вместо ручной регистрации каждого класса IRequestHandler или IPipelineBehavior, MediatR может автоматически сканировать указанную сборку и регистрировать все необходимые зависимости.27 Это значительно упрощает настройку и снижает риск ошибок, особенно в больших проектах.26

В файл Program.cs (серверный часть) добавьте следующий код:

|  |
| --- |
| // ✅ Подключаем EF Core (SQLite, строка в appsettings.json)  builder.Services.AddDbContextFactory<AppDbContext>(options =>      options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));  // ✅ Подключаем MediatR до Build()  builder.Services.AddMediatR(cfg =>      cfg.RegisterServicesFromAssembly(typeof(Program).Assembly));  // ✅ Razor Components  builder.Services.AddRazorComponents()      .AddInteractiveServerComponents()       // 🔥 добавляем поддержку Server render mode      .AddInteractiveWebAssemblyComponents(); // оставляем и wasm (гибрид)  ……  app.MapRazorComponents<App>()      .AddInteractiveServerRenderMode()          // ✅ добавляем серверный режим      .AddInteractiveWebAssemblyRenderMode()     // (оставляем и wasm, если нужно гибридное)      .AddAdditionalAssemblies(typeof(BlazorCqrsCrud.Client.\_Imports).Assembly);  app.Run(); |



III. Устройство слоя данных (Data Layer)

#### 3.1. Модели и сущности

В архитектуре CQRS важно разделять сущности предметной области (Entities) и объекты-переносчики данных (DTO).4 Сущность (Product) представляет собой модель, которая используется в доменном слое и слое данных, тогда как DTO (ProductDto) — это упрощенная модель, используемая для передачи данных в UI. DTO не содержат бизнес-логики и служат для оптимизации запросов и предотвращения утечки деталей доменной модели в слой представления.4

оздай папку Models в BlazorCqrsCrud.Shared и создайте класси Product и ProductDto:

// BlazorCqrsCrud.Shared/Models/Product.cs

namespace BlazorCqrsCrud.Shared.Models

{

    public class Product

    {

        public int Id { get; set; }

        public string Name { get; set; } = string.Empty;

        public decimal Price { get; set; }

    }

}

И соответствующий DTO для запросов:

// BlazorCqrsCrud.Shared/Models/ProductDto.cs

namespace BlazorCqrsCrud.Shared.Models

{

    public class ProductDto

    {

        public int Id { get; set; }

        public string Name { get; set; } = string.Empty;

        public decimal Price { get; set; }

    }

}

#### 3.2. Контекст базы данных (DbContext) и SQLite

Контекст базы данных, реализуемый через класс AppDbContext, является центральным компонентом при взаимодействии с данными в приложениях, построенных с использованием Entity Framework Core (EF Core). В терминологии шаблонов проектирования он реализует паттерн «Единица работы» (*Unit of Work*), обеспечивая согласованное управление набором объектов доменной модели и отслеживание изменений, вносимых в данные в рамках одной логической транзакции. Это позволяет централизовать операции чтения, создания, обновления и удаления (CRUD) и координировать их выполнение с целостностью базы данных.

В данном проекте AppDbContext наследуется от базового класса DbContext и определяет DbSet<Product>, который представляет собой коллекцию сущностей Product, отображаемых на таблицу в базе данных. Наличие DbSet позволяет EF Core автоматически формировать соответствующие SQL-запросы при выполнении операций с объектами Product, обеспечивая прозрачное взаимодействие между объектной моделью приложения и реляционной структурой базы данных.

Для поддержки работы с легковесной, файловой системой хранения данных на стороне сервера выбрана встраиваемая СУБД SQLite. Это решение обусловлено простотой развертывания, отсутствием необходимости во внешнем сервере баз данных и достаточной производительностью для сценариев с умеренной нагрузкой, что делает её подходящим выбором на этапах разработки и тестирования, а также в приложениях с небольшим объёмом данных.

Однако для корректной работы механизма миграций Entity Framework Core требуется возможность создания экземпляра DbContext во время выполнения инструментов разработки (например, при выполнении команд dotnet ef migrations add или dotnet ef database update). Поскольку в стандартной конфигурации ASP.NET Core внедрение зависимостей (Dependency Injection) инициализируется только при запуске приложения, необходим специальный механизм для создания контекста в design-time.

Для решения этой задачи реализован класс AppDbContextFactory, реализующий интерфейс IDesignTimeDbContextFactory<TContext>. Данный интерфейс предоставляет точку расширения, позволяющую EF Core инструментам обнаружить и создать экземпляр контекста без загрузки всего приложения. В методе CreateDbContext настраивается DbContextOptionsBuilder с использованием провайдера UseSqlite, указывающего на файл базы данных app.db. Это гарантирует, что миграции будут применяться к корректному экземпляру базы данных в ходе разработки.

Таким образом, совместное использование AppDbContext и AppDbContextFactory обеспечивает:  
— централизованное управление данными через паттерн *Unit of Work*;  
— поддержку кода миграций и автоматического сопоставления модели (Code First);  
— работоспособность инструментов EF Core в режиме разработки;  
— простоту настройки и развертывания приложения с использованием SQLite в качестве целевой СУБД.

Данный подход соответствует современным практикам разработки .NET-приложений с использованием ORM и способствует поддержке чистой архитектуры, масштабируемости и тестируемости кода.

Создадим контекст AppDbContext:

|  |
| --- |
| using BlazorCqrsCrud.Shared.Models;  using Microsoft.EntityFrameworkCore;  namespace BlazorCqrsCrud.Server.Data  {      public class AppDbContext : DbContext      {          public AppDbContext(DbContextOptions<AppDbContext> options)              : base(options)          {          }          public DbSet<Product> Products { get; set; } = null!;      }  } |

Создадим Фабрика для миграций

|  |
| --- |
| // Server/Data/AppDbContextFactory.cs  using Microsoft.EntityFrameworkCore;  using Microsoft.EntityFrameworkCore.Design;  namespace BlazorCqrsCrud.Server.Data  {      public class AppDbContextFactory : IDesignTimeDbContextFactory<AppDbContext>      {          public AppDbContext CreateDbContext(string[] args)          {              var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>();              optionsBuilder.UseSqlite("Data Source=app.db");              return new AppDbContext(optionsBuilder.Options);          }      }  } |

#### 3.3. Особенности DbContext в Blazor и IDbContextFactory

В серверных Blazor-приложениях (Blazor Server) существует особенность, связанная с жизненным циклом DbContext. По умолчанию DbContext регистрируется как Scoped сервис. В ASP.NET Core Scoped означает, что одна и та же инстанция сервиса будет использоваться на протяжении всего жизненного цикла HTTP-запроса. Однако в Blazor Scoped сервис привязан к так называемому "контуру" (circuit) — состоянию пользовательской сессии, которое может быть долгоживущим.29

DbContext не является потокобезопасным, и его использование в долгоживущих компонентах или при параллельном доступе может привести к проблемам.29 Для решения этой проблемы рекомендуется использовать фабрику

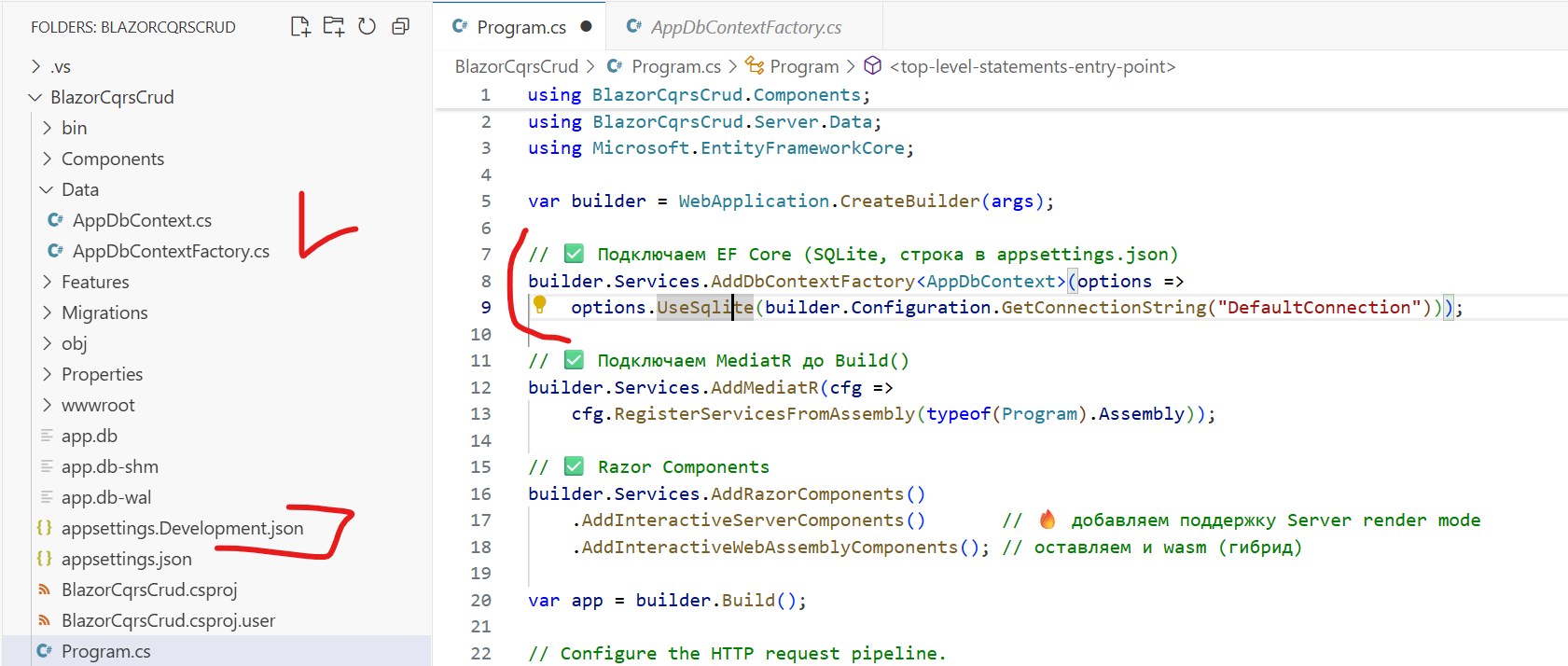
IDbContextFactory<TContext>. Эта фабрика позволяет безопасно создавать новые, кратковременные инстанции DbContext для каждой отдельной операции, что соответствует принципу "один контекст на операцию".29

Настройка DI-контейнера для использования фабрики осуществляется в Program.cs:

// ✅ Подключаем EF Core (SQLite, строка в appsettings.json)

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

    options.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection")));



В appsettings.json и в appsettings.Development.json добавь строку подключения:

{

"ConnectionStrings": {

"DefaultConnection": "Data Source=app.db"

}

}

Вот польный код файла appsettings.json

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

Вот польный код файла appsettings.Development.json

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

#### 3.4. Миграции EF Core

Миграции EF Core позволяют управлять схемой базы данных, синхронизируя её с моделью данных приложения, и являются ключевым инструментом для поддержания схемы в актуальном состоянии.31

1. **Установка инструментов:** Убедитесь, что инструменты EF Core установлены глобально: dotnet tool install --global dotnet-ef.25
2. Создание миграции: Выполните следующую команду в корневой директории проекта, чтобы создать начальную миграцию на основе модели Product:

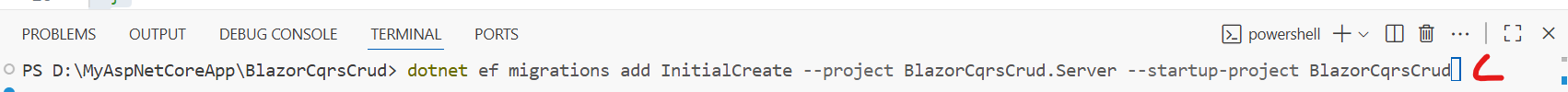
dotnet ef migrations add InitialCreate

1. Применение миграции: После создания миграции примените её к базе данных, что создаст файл app.db и таблицу Products:  
   dotnet ef database update.

Убедись, что в appsettings.json есть "DefaultConnection" → "Data Source=app.db".

Сгенерируй миграцию:

dotnet ef migrations add InitialCreate --project BlazorCqrsCrud.Server --startup-project BlazorCqrsCrud



Примени миграцию:

dotnet ef database update --project BlazorCqrsCrud.Server --startup-project BlazorCqrsCrud



Если ты что-то менял в AppDbContext или моделях, и нужно создать новую миграцию:

dotnet ef migrations add AddProductsTable --project BlazorCqrsCrud --startup-project BlazorCqrsCrud

dotnet ef database update --project BlazorCqrsCrud --startup-project BlazorCqrsCrud

### IV. Реализация стороны чтения (Queries)

#### 4.1. Проектирование запросов

Запросы предназначены исключительно для получения данных. Они должны быть легковесными и не содержать бизнес-логики.

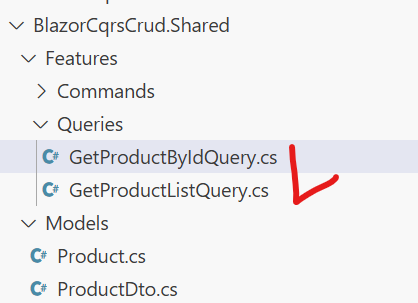
Запрос на получение всех продуктов: Файлы GetProductListQuery.cs и GetProductByIdQuery.cs создать в Shared, чтобы их мог видеть и сервер, и клиент.

Запрос на получение все продукты:

|  |
| --- |
| namespace BlazorCqrsCrud.Shared.Features.Products.Queries  {      public class GetProductListQuery : IRequest<List<ProductDto>> { }  } |

Запрос на получение одного продукта по идентификатору:

|  |
| --- |
| namespace BlazorCqrsCrud.Shared.Features.Products.Queries  {      public class GetProductByIdQuery : IRequest<ProductDto>      {          public int Id { get; set; }      }  } |



После этого очисти и пересобери:

dotnet clean

dotnet build

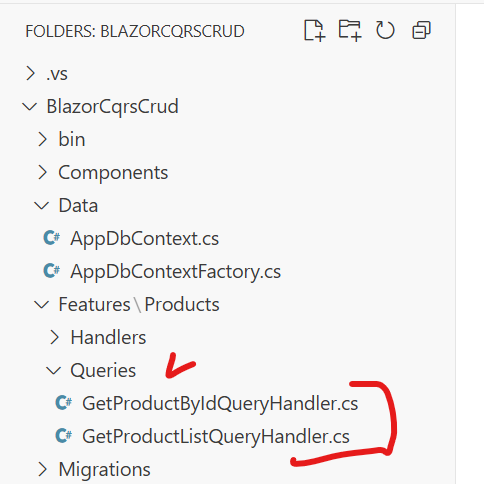
#### 4.2. Обработчики запросов (Query Handlers)

Обработчики запросов инкапсулируют логику получения данных. Они используют IDbContextFactory для создания кратковременного контекста в сервере GetProductListQueryHandler.cs.

|  |
| --- |
| using MediatR;  using Microsoft.EntityFrameworkCore;  using BlazorCqrsCrud.Server.Data;  using BlazorCqrsCrud.Shared.Models;  using BlazorCqrsCrud.Shared.Features.Products.Queries;  namespace BlazorCqrsCrud.Server.Features.Products.Queries  {      public class GetProductListQueryHandler : IRequestHandler<GetProductListQuery, List<ProductDto>>      {          private readonly IDbContextFactory<AppDbContext> \_dbContextFactory;          public GetProductListQueryHandler(IDbContextFactory<AppDbContext> dbContextFactory)          {              \_dbContextFactory = dbContextFactory;          }          public async Task<List<ProductDto>> Handle(GetProductListQuery request, CancellationToken cancellationToken)          {              await using var context = \_dbContextFactory.CreateDbContext();              return await context.Products                  .Select(p => new ProductDto                  {                      Id = p.Id,                      Name = p.Name,                      Price = p.Price                  })                  .ToListAsync(cancellationToken);          }      }  } |

Аналогичным образом создается GetProductByIdQueryHandler.

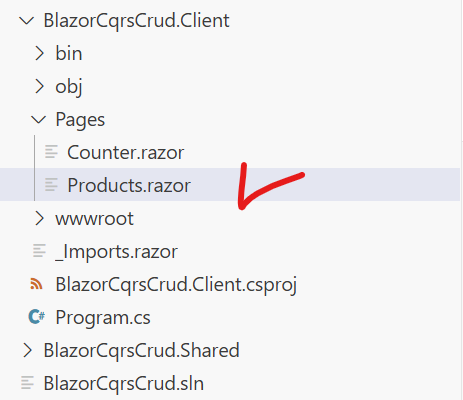
|  |
| --- |
| using MediatR;  using Microsoft.EntityFrameworkCore;  using BlazorCqrsCrud.Server.Data;  using BlazorCqrsCrud.Shared.Models;  using BlazorCqrsCrud.Shared.Features.Products.Queries;  using System.Threading;  using System.Threading.Tasks;  namespace BlazorCqrsCrud.Server.Features.Products.Queries  {      public class GetProductByIdQueryHandler : IRequestHandler<GetProductByIdQuery, ProductDto>      {          private readonly IDbContextFactory<AppDbContext> \_dbContextFactory;          public GetProductByIdQueryHandler(IDbContextFactory<AppDbContext> dbContextFactory)          {              \_dbContextFactory = dbContextFactory;          }          public async Task<ProductDto> Handle(GetProductByIdQuery request, CancellationToken cancellationToken)          {              await using var context = \_dbContextFactory.CreateDbContext();              var product = await context.Products                  .FirstOrDefaultAsync(p => p.Id == request.Id, cancellationToken);              if (product == null) return new ProductDto{};              return new ProductDto              {                  Id = product.Id,                  Name = product.Name,                  Price = product.Price              };          }      }  } |



#### 4.3. Интеграция с Blazor-компонентами

Для взаимодействия с MediatR из Blazor-компонентов необходимо внедрить IMediator. В Blazor WebAssembly проекте (BlazorCqrsCrud.Client) обычно создают папку Pages:

|  |
| --- |
| @page "/products"  @using BlazorCqrsCrud.Shared.Features.Products.Commands  @using BlazorCqrsCrud.Shared.Features.Products.Queries  @using BlazorCqrsCrud.Shared.Models  @using MediatR  @using Microsoft.AspNetCore.Components  @using Microsoft.AspNetCore.Components.Web  @using Microsoft.AspNetCore.Components.Forms  @inject IMediator Mediator  @inject ILogger<Products> Logger  @rendermode RenderMode.InteractiveServer  <h3 class="mb-3">Products</h3>  @if (isLoading)  {      <p>Loading...</p>  }  else  {      @if (!string.IsNullOrWhiteSpace(errorMessage))      {          <div class="alert alert-danger">@errorMessage</div>      }      @if (!string.IsNullOrWhiteSpace(infoMessage))      {          <div class="alert alert-success">@infoMessage</div>      }      <table class="table table-striped align-middle">          <thead>              <tr>                  <th style="width:45%">Name</th>                  <th style="width:25%">Price</th>                  <th style="width:30%">Actions</th>              </tr>          </thead>          <tbody>              @if (products.Count == 0)              {                  <tr><td colspan="3" class="text-muted">No products yet.</td></tr>              }              else              {                  @foreach (var p in products)                  {                      <tr>                          <td>                              @if (editProduct?.Id == p.Id)                              {                                  <InputText @bind-Value="editProduct.Name" class="form-control" />                              }                              else                              {                                  @p.Name                              }                          </td>                          <td>                              @if (editProduct?.Id == p.Id)                              {                                  <InputNumber @bind-Value="editProduct.Price" class="form-control" />                              }                              else                              {                                  @p.Price                              }                          </td>                          <td>                              @if (editProduct?.Id == p.Id)                              {                                  <button class="btn btn-success btn-sm me-2"                                          disabled="@isBusy"                                          @onclick="async () => await SaveEdit()">                                      Save                                  </button>                                  <button class="btn btn-secondary btn-sm"                                          disabled="@isBusy"                                          @onclick="CancelEdit">                                      Cancel                                  </button>                              }                              else                              {                                  <button class="btn btn-primary btn-sm me-2"                                          disabled="@isBusy"                                          @onclick="() => StartEdit(p)">                                      Edit                                  </button>                                  <button class="btn btn-danger btn-sm"                                          disabled="@isBusy"                                          @onclick="async () => await DeleteProduct(p.Id)">                                      Delete                                  </button>                              }                          </td>                      </tr>                  }              }          </tbody>      </table>      <h4 class="mt-4">Add New Product</h4>      <EditForm Model="@newProduct" OnValidSubmit="HandleAdd" FormName="ProductsForm">          <DataAnnotationsValidator />          <ValidationSummary />          <div class="row g-2">              <div class="col-md-6">                  <InputText class="form-control" placeholder="Name" @bind-Value="newProduct.Name" />              </div>              <div class="col-md-3">                  <!-- InputNumber может работать некорректно с decimal? при пустом значении -->                  <InputNumber class="form-control" placeholder="Price" @bind-Value="newProduct.Price" />              </div>              <div class="col-md-3 d-grid">                  <button type="submit" class="btn btn-success" disabled="@isBusy">Add</button>              </div>          </div>      </EditForm>  }  @code {      // --- Состояние страницы ---      private bool isLoading = true;      private bool isBusy = false;      private string? errorMessage;      private string? infoMessage;      private List<ProductDto> products = new();      private ProductDto? editProduct;      // ViewModel для формы добавления нового продукта      private NewProductVm newProduct = new();      // --- Инициализация страницы ---      protected override async Task OnInitializedAsync()      {          Logger.LogInformation("Products page initialized.");          await LoadProducts();      }      // --- Загрузка списка продуктов ---      private async Task LoadProducts()      {          isLoading = true;          errorMessage = null;          infoMessage = null;          Logger.LogInformation("Loading products from database...");          try          {              products = await Mediator.Send(new GetProductListQuery());              Logger.LogInformation("Loaded {Count} products.", products.Count);          }          catch (Exception ex)          {              Logger.LogError(ex, "Error loading products.");              errorMessage = "Failed to load products.";          }          finally          {              isLoading = false;          }      }      // --- Начало редактирования ---      private void StartEdit(ProductDto p)      {          errorMessage = null;          infoMessage = null;          Logger.LogInformation("Start editing product Id={Id}, Name={Name}", p.Id, p.Name);          editProduct = new ProductDto          {              Id = p.Id,              Name = p.Name,              Price = p.Price          };      }      // --- Отмена редактирования ---      private void CancelEdit()      {          Logger.LogInformation("Edit cancelled for product Id={Id}", editProduct?.Id);          editProduct = null;          errorMessage = null;          infoMessage = null;          StateHasChanged();      }      // --- Сохранение изменений ---      private async Task SaveEdit()      {          if (editProduct is null) return;          isBusy = true;          errorMessage = null;          infoMessage = null;          try          {              Logger.LogInformation("Saving changes for product Id={Id}", editProduct.Id);              // Проверка валидации              if (string.IsNullOrWhiteSpace(editProduct.Name))              {                  errorMessage = "Name is required.";                  return;              }              var cmd = new UpdateProductCommand              {                  Id = editProduct.Id,                  Name = editProduct.Name,                  Price = editProduct.Price              };              await Mediator.Send(cmd);              Logger.LogInformation("Product Id={Id} updated.", editProduct.Id);              infoMessage = "Product updated.";              editProduct = null;              await LoadProducts();          }          catch (Exception ex)          {              Logger.LogError(ex, "Error updating product Id={Id}", editProduct?.Id);              errorMessage = "Failed to update product.";          }          finally          {              isBusy = false;          }      }      // --- Добавление нового продукта ---      private async Task HandleAdd(EditContext \_)      {          // Вызов метода добавления          await AddProductCore();      }      private async Task AddProductCore()      {          isBusy = true;          errorMessage = null;          infoMessage = null;          try          {              Logger.LogInformation("Adding new product: Name={Name}, Price={Price}", newProduct.Name, newProduct.Price);              var cmd = new CreateProductCommand              {                  Name = newProduct.Name!,                  Price = newProduct.Price ?? 0m              };              var newId = await Mediator.Send(cmd);              Logger.LogInformation("Product created with Id={Id}", newId);              infoMessage = "Product added.";              newProduct = new(); // сброс формы              await LoadProducts();          }          catch (Exception ex)          {              Logger.LogError(ex, "Error creating product.");              errorMessage = "Failed to add product.";          }          finally          {              isBusy = false;          }      }      // --- Удаление продукта ---      private async Task DeleteProduct(int id)      {          isBusy = true;          errorMessage = null;          infoMessage = null;          try          {              Logger.LogInformation("Deleting product Id={Id}", id);              var cmd = new DeleteProductCommand { Id = id };              await Mediator.Send(cmd);              Logger.LogInformation("Product Id={Id} deleted.", id);              infoMessage = "Product deleted.";              await LoadProducts();          }          catch (Exception ex)          {              Logger.LogError(ex, "Error deleting product Id={Id}", id);              errorMessage = "Failed to delete product.";          }          finally          {              isBusy = false;          }      }      // --- ViewModel для формы Add ---      private sealed class NewProductVm      {          [System.ComponentModel.DataAnnotations.Required(ErrorMessage = "Name is required.")]          public string? Name { get; set; }          [System.ComponentModel.DataAnnotations.Required(ErrorMessage = "Price is required.")]          [System.ComponentModel.DataAnnotations.Range(0.0, double.MaxValue, ErrorMessage = "Price must be >= 0.")]          public decimal? Price { get; set; } = 0m;      }  } |



### V. Реализация стороны записи (Commands)

#### 5.1. Проектирование команд

Команды отвечают за изменение состояния. Названия команд должны отражать бизнес-намерение пользователя, а не низкоуровневые операции, хотя в данном простом примере мы будем использовать общие названия, соответствующие CRUD. Команды создаются в BlazorCqrsCrud.Shared

**Команда создания продукта:**

|  |
| --- |
| using MediatR;  namespace BlazorCqrsCrud.Shared.Features.Products.Commands  {      public class CreateProductCommand : IRequest<int>      {          public string Name { get; set; } = string.Empty;          public decimal Price { get; set; }      }  } |

**Команда обновления продукта:**

|  |
| --- |
| using MediatR;  namespace BlazorCqrsCrud.Shared.Features.Products.Commands  {      public class UpdateProductCommand : IRequest<Unit>      {          public int Id { get; set; }          public string Name { get; set; } = string.Empty;          public decimal Price { get; set; }      }  } |

**Команда удаления продукта:**

|  |
| --- |
| namespace BlazorCqrsCrud.Shared.Features.Products.Commands  {      public class DeleteProductCommand : IRequest<Unit>      {          public int Id { get; set; }      }  } |

#### 

#### 5.2. Обработчики команд (Command Handlers)

Обработчики команд реализуют логику записи. Они также используют IDbContextFactory.

**Обработчик создания:**

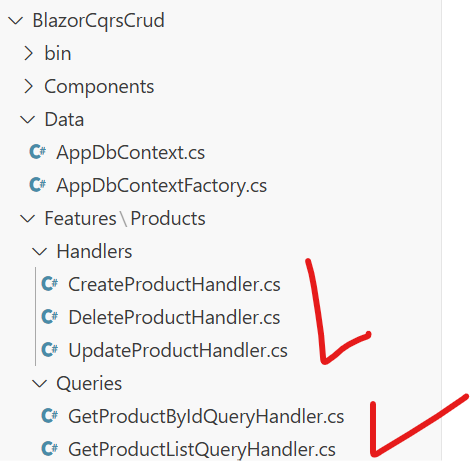
|  |
| --- |
| using BlazorCqrsCrud.Server.Data;  using BlazorCqrsCrud.Shared.Features.Products.Commands;  using BlazorCqrsCrud.Shared.Models;  using MediatR;  using Microsoft.EntityFrameworkCore;  namespace BlazorCqrsCrud.Server.Features.Products.Commands  {      public class CreateProductHandler : IRequestHandler<CreateProductCommand, int>      {          private readonly IDbContextFactory<AppDbContext> \_contextFactory;          private readonly ILogger<CreateProductHandler> \_logger;          public CreateProductHandler(IDbContextFactory<AppDbContext> contextFactory,                                      ILogger<CreateProductHandler> logger)          {              \_contextFactory = contextFactory;              \_logger = logger;          }          public async Task<int> Handle(CreateProductCommand request, CancellationToken cancellationToken)          {              \_logger.LogInformation("Handling CreateProductCommand: Name={Name}, Price={Price}", request.Name, request.Price);              await using var context = await \_contextFactory.CreateDbContextAsync(cancellationToken);              var product = new Product              {                  Name = request.Name,                  Price = request.Price              };              context.Products.Add(product);              \_logger.LogInformation("Product added to context, saving changes...");              var result = await context.SaveChangesAsync(cancellationToken);              \_logger.LogInformation("SaveChangesAsync result: {Result}, Product Id: {Id}", result, product.Id);              return product.Id;          }      }  } |

**Обработчик обновления:**

|  |
| --- |
| using MediatR;  using BlazorCqrsCrud.Server.Data;  using Microsoft.EntityFrameworkCore;  using BlazorCqrsCrud.Shared.Features.Products.Commands;  namespace BlazorCqrsCrud.Server.Features.Products.Commands  {      public class UpdateProductHandler : IRequestHandler<UpdateProductCommand, Unit>      {          private readonly IDbContextFactory<AppDbContext> \_dbContextFactory;          public UpdateProductHandler(IDbContextFactory<AppDbContext> dbContextFactory)          {              \_dbContextFactory = dbContextFactory;          }          public async Task<Unit> Handle(UpdateProductCommand request, CancellationToken cancellationToken)          {              await using var context = \_dbContextFactory.CreateDbContext();              var product = await context.Products.FirstOrDefaultAsync(p => p.Id == request.Id, cancellationToken);              if (product != null)              {                  product.Name = request.Name;                  product.Price = request.Price;                  await context.SaveChangesAsync(cancellationToken);              }              return Unit.Value;          }      }  } |

**Обработчик удаления:**

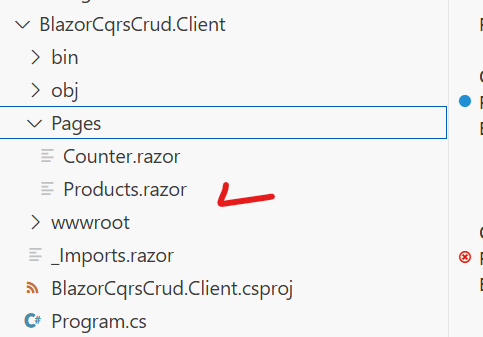
|  |
| --- |
| using MediatR;  using BlazorCqrsCrud.Server.Data;  using Microsoft.EntityFrameworkCore;  using System.Threading;  using System.Threading.Tasks;  using BlazorCqrsCrud.Shared.Features.Products.Commands;  namespace BlazorCqrsCrud.Server.Features.Products.Commands  {      public class DeleteProductHandler : IRequestHandler<DeleteProductCommand, Unit>      {          private readonly IDbContextFactory<AppDbContext> \_dbContextFactory;          public DeleteProductHandler(IDbContextFactory<AppDbContext> dbContextFactory)          {              \_dbContextFactory = dbContextFactory;          }          public async Task<Unit> Handle(DeleteProductCommand request, CancellationToken cancellationToken)          {              await using var context = \_dbContextFactory.CreateDbContext();              var product = await context.Products.FirstOrDefaultAsync(p => p.Id == request.Id, cancellationToken);              if (product != null)              {                  context.Products.Remove(product);                  await context.SaveChangesAsync(cancellationToken);              }              return Unit.Value;          }      }  } |



#### 5.3. Интеграция с Blazor-компонентами

Вызов команды из компонента Blazor осуществляется через IMediator.17

|  |
| --- |
| @page "/products"  @using BlazorCqrsCrud.Shared.Features.Products.Commands  @using BlazorCqrsCrud.Shared.Features.Products.Queries  @using BlazorCqrsCrud.Shared.Models  @using MediatR  @using Microsoft.AspNetCore.Components  @using Microsoft.AspNetCore.Components.Web  @using Microsoft.AspNetCore.Components.Forms  @inject IMediator Mediator  @inject ILogger<Products> Logger  @rendermode RenderMode.InteractiveServer  <h3 class="mb-3">Products</h3>  @if (isLoading)  {      <p>Loading...</p>  }  else  {      @if (!string.IsNullOrWhiteSpace(errorMessage))      {          <div class="alert alert-danger">@errorMessage</div>      }      @if (!string.IsNullOrWhiteSpace(infoMessage))      {          <div class="alert alert-success">@infoMessage</div>      }      <table class="table table-striped align-middle">          <thead>              <tr>                  <th style="width:45%">Name</th>                  <th style="width:25%">Price</th>                  <th style="width:30%">Actions</th>              </tr>          </thead>          <tbody>              @if (products.Count == 0)              {                  <tr><td colspan="3" class="text-muted">No products yet.</td></tr>              }              else              {                  @foreach (var p in products)                  {                      <tr>                          <td>                              @if (editProduct?.Id == p.Id)                              {                                  <InputText @bind-Value="editProduct.Name" class="form-control" />                              }                              else                              {                                  @p.Name                              }                          </td>                          <td>                              @if (editProduct?.Id == p.Id)                              {                                  <InputNumber @bind-Value="editProduct.Price" class="form-control" />                              }                              else                              {                                  @p.Price                              }                          </td>                          <td>                              @if (editProduct?.Id == p.Id)                              {                                  <button class="btn btn-success btn-sm me-2"                                          disabled="@isBusy"                                          @onclick="async () => await SaveEdit()">                                      Save                                  </button>                                  <button class="btn btn-secondary btn-sm"                                          disabled="@isBusy"                                          @onclick="CancelEdit">                                      Cancel                                  </button>                              }                              else                              {                                  <button class="btn btn-primary btn-sm me-2"                                          disabled="@isBusy"                                          @onclick="() => StartEdit(p)">                                      Edit                                  </button>                                  <button class="btn btn-danger btn-sm"                                          disabled="@isBusy"                                          @onclick="async () => await DeleteProduct(p.Id)">                                      Delete                                  </button>                              }                          </td>                      </tr>                  }              }          </tbody>      </table>      <h4 class="mt-4">Add New Product</h4>      <EditForm Model="@newProduct" OnValidSubmit="HandleAdd" FormName="ProductsForm">          <DataAnnotationsValidator />          <ValidationSummary />          <div class="row g-2">              <div class="col-md-6">                  <InputText class="form-control" placeholder="Name" @bind-Value="newProduct.Name" />              </div>              <div class="col-md-3">                  <!-- InputNumber может работать некорректно с decimal? при пустом значении -->                  <InputNumber class="form-control" placeholder="Price" @bind-Value="newProduct.Price" />              </div>              <div class="col-md-3 d-grid">                  <button type="submit" class="btn btn-success" disabled="@isBusy">Add</button>              </div>          </div>      </EditForm>  }  @code {      // --- Состояние страницы ---      private bool isLoading = true;      private bool isBusy = false;      private string? errorMessage;      private string? infoMessage;      private List<ProductDto> products = new();      private ProductDto? editProduct;      // ViewModel для формы добавления нового продукта      private NewProductVm newProduct = new();      // --- Инициализация страницы ---      protected override async Task OnInitializedAsync()      {          Logger.LogInformation("Products page initialized.");          await LoadProducts();      }      // --- Загрузка списка продуктов ---      private async Task LoadProducts()      {          isLoading = true;          errorMessage = null;          infoMessage = null;          Logger.LogInformation("Loading products from database...");          try          {              products = await Mediator.Send(new GetProductListQuery());              Logger.LogInformation("Loaded {Count} products.", products.Count);          }          catch (Exception ex)          {              Logger.LogError(ex, "Error loading products.");              errorMessage = "Failed to load products.";          }          finally          {              isLoading = false;          }      }      // --- Начало редактирования ---      private void StartEdit(ProductDto p)      {          errorMessage = null;          infoMessage = null;          Logger.LogInformation("Start editing product Id={Id}, Name={Name}", p.Id, p.Name);          editProduct = new ProductDto          {              Id = p.Id,              Name = p.Name,              Price = p.Price          };      }      // --- Отмена редактирования ---      private void CancelEdit()      {          Logger.LogInformation("Edit cancelled for product Id={Id}", editProduct?.Id);          editProduct = null;          errorMessage = null;          infoMessage = null;          StateHasChanged();      }      // --- Сохранение изменений ---      private async Task SaveEdit()      {          if (editProduct is null) return;          isBusy = true;          errorMessage = null;          infoMessage = null;          try          {              Logger.LogInformation("Saving changes for product Id={Id}", editProduct.Id);              // Проверка валидации              if (string.IsNullOrWhiteSpace(editProduct.Name))              {                  errorMessage = "Name is required.";                  return;              }              var cmd = new UpdateProductCommand              {                  Id = editProduct.Id,                  Name = editProduct.Name,                  Price = editProduct.Price              };              await Mediator.Send(cmd);              Logger.LogInformation("Product Id={Id} updated.", editProduct.Id);              infoMessage = "Product updated.";              editProduct = null;              await LoadProducts();          }          catch (Exception ex)          {              Logger.LogError(ex, "Error updating product Id={Id}", editProduct?.Id);              errorMessage = "Failed to update product.";          }          finally          {              isBusy = false;          }      }      // --- Добавление нового продукта ---      private async Task HandleAdd(EditContext \_)      {          // Вызов метода добавления          await AddProductCore();      }      private async Task AddProductCore()      {          isBusy = true;          errorMessage = null;          infoMessage = null;          try          {              Logger.LogInformation("Adding new product: Name={Name}, Price={Price}", newProduct.Name, newProduct.Price);              var cmd = new CreateProductCommand              {                  Name = newProduct.Name!,                  Price = newProduct.Price ?? 0m              };              var newId = await Mediator.Send(cmd);              Logger.LogInformation("Product created with Id={Id}", newId);              infoMessage = "Product added.";              newProduct = new(); // сброс формы              await LoadProducts();          }          catch (Exception ex)          {              Logger.LogError(ex, "Error creating product.");              errorMessage = "Failed to add product.";          }          finally          {              isBusy = false;          }      }      // --- Удаление продукта ---      private async Task DeleteProduct(int id)      {          isBusy = true;          errorMessage = null;          infoMessage = null;          try          {              Logger.LogInformation("Deleting product Id={Id}", id);              var cmd = new DeleteProductCommand { Id = id };              await Mediator.Send(cmd);              Logger.LogInformation("Product Id={Id} deleted.", id);              infoMessage = "Product deleted.";              await LoadProducts();          }          catch (Exception ex)          {              Logger.LogError(ex, "Error deleting product Id={Id}", id);              errorMessage = "Failed to delete product.";          }          finally          {              isBusy = false;          }      }      // --- ViewModel для формы Add ---      private sealed class NewProductVm      {          [System.ComponentModel.DataAnnotations.Required(ErrorMessage = "Name is required.")]          public string? Name { get; set; }          [System.ComponentModel.DataAnnotations.Required(ErrorMessage = "Price is required.")]          [System.ComponentModel.DataAnnotations.Range(0.0, double.MaxValue, ErrorMessage = "Price must be >= 0.")]          public decimal? Price { get; set; } = 0m;      }  } |



**Таблица 3: Компоненты CQRS для CRUD-операций**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CRUD-операция | Тип сообщения | Название сообщения | Обработчик | Действие |
| **C**reate | Команда | CreateProductCommand | CreateProductHandler | Создание новой записи в БД |
| **R**ead | Запрос | GetProductListQuery, GetProductByIdQuery | GetProductListQueryHandler, GetProductByIdQueryHandler | Получение данных из БД |
| **U**pdate | Команда | UpdateProductCommand | UpdateProductHandler | Обновление существующей записи в БД |
| **D**elete | Команда | DeleteProductCommand | DeleteProductHandler | Удаление записи из БД |

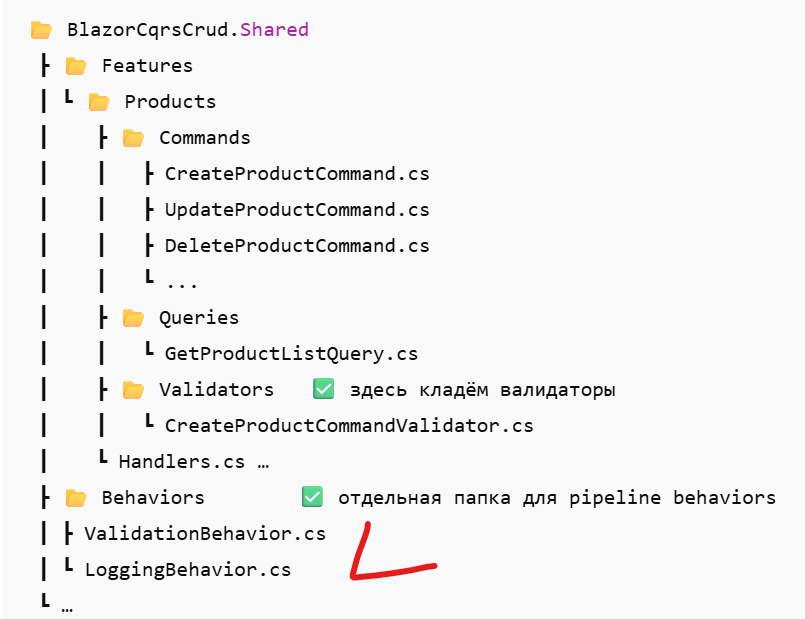
### VI. Кросс-функциональные задачи и улучшения

#### 6.1. MediatR Pipeline Behaviors

MediatR Pipeline Behaviors представляют собой мощный механизм для реализации сквозных задач, таких как логирование, валидация или управление транзакциями.35 Они действуют как middleware-компоненты, перехватывая запросы до их передачи обработчику и после его выполнения.18 Этот подход является реализацией паттерна "Декоратор" и позволяет централизовать общую логику, избегая её дублирования в каждом обработчике.35

#### 6.2. Валидация с FluentValidation

Одним из наиболее распространенных применений пайплайнов является валидация. Использование FluentValidation позволяет отделить правила валидации от доменных объектов и обработчиков.



1. Создание валидатора. Валидатор наследуется от AbstractValidator<TCommand> и содержит правила для каждого свойства.

|  |
| --- |
| using BlazorCqrsCrud.Shared.Features.Products.Commands;  using FluentValidation;  namespace BlazorCqrsCrud.Shared.Features.Products.Validators;  public class CreateProductCommandValidator : AbstractValidator<CreateProductCommand>  {  public CreateProductCommandValidator()  {  RuleFor(x => x.Name)  .NotEmpty().WithMessage("Name is required.");  RuleFor(x => x.Price)  .GreaterThan(0).WithMessage("Price must be greater than 0.");  }  } |

1. **Создание поведения валидации.** ValidationBehavior проверяет, существуют ли валидаторы для данного запроса. Если да, он выполняет валидацию и в случае ошибок выбрасывает ValidationException.

Пример Behaviors/ValidationBehavior.cs

|  |
| --- |
| using FluentValidation;  using MediatR;  namespace BlazorCqrsCrud.Shared.Behaviors;  public class ValidationBehavior<TRequest, TResponse>  : IPipelineBehavior<TRequest, TResponse>  where TRequest : notnull  {  private readonly IEnumerable<IValidator<TRequest>> \_validators;  public ValidationBehavior(IEnumerable<IValidator<TRequest>> validators)  {  \_validators = validators;  }  public async Task<TResponse> Handle(  TRequest request,  RequestHandlerDelegate<TResponse> next,  CancellationToken cancellationToken)  {  if (\_validators.Any())  {  var context = new ValidationContext<TRequest>(request);  var failures = \_validators  .Select(v => v.Validate(context))  .SelectMany(r => r.Errors)  .Where(f => f != null)  .ToList();  if (failures.Count != 0)  throw new ValidationException(failures);  }  return await next();  }  } |

1. **Регистрация.** Валидатор и поведение регистрируются в DI-контейнере.26

builder.Services.AddValidatorsFromAssembly(Assembly.GetExecutingAssembly());  
builder.Services.AddTransient(typeof(IPipelineBehavior<,>), typeof(ValidationBehavior<,>));

1. **Глобальная обработка исключений.** Для корректной обработки ошибок валидации, вызванных пайплайном, настраивается глобальный обработчик исключений, который перехватывает ValidationException и возвращает пользователю ответ с кодом 400 BadRequest.26

Хотя такой подход с валидационным пайплайном является простым и популярным, существует и другая, более строгая парадигма — "Parse, Don't Validate".37 Она утверждает, что невалидное состояние не должно быть представимо в системе. Вместо того чтобы создавать команду с потенциально невалидными данными (

CreateProductCommand(string name)) и затем валидировать её, можно создать Value Object (ProductName) и передавать его в команду (CreateProductCommand(ProductName name)). В этом случае ProductName не может быть создан с пустым или невалидным значением, что делает систему более надежной и предсказуемой на уровне доменной модели. Валидация в пайплайне — это отличный и удобный способ решения проблемы, но принцип "Parse, Don't Validate" представляет собой более глубокий уровень доменного моделирования.

#### 6.3. Логирование и другие поведения

Логирование — еще одна типичная сквозная задача, которая идеально подходит для реализации через MediatR Behaviors. Создание поведения LoggingBehavior позволяет автоматически логировать входящие запросы и исходящие ответы для всех сообщений, проходящих через MediatR.18

Behaviors/LoggingBehavior.cs

|  |
| --- |
| using MediatR;  using Microsoft.Extensions.Logging;  namespace BlazorCqrsCrud.Shared.Behaviors;  public class LoggingBehavior<TRequest, TResponse>  : IPipelineBehavior<TRequest, TResponse>  {  private readonly ILogger<LoggingBehavior<TRequest, TResponse>> \_logger;  public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger)  => \_logger = logger;  public async Task<TResponse> Handle(  TRequest request,  RequestHandlerDelegate<TResponse> next,  CancellationToken cancellationToken)  {  \_logger.LogInformation("Handling {Request}", typeof(TRequest).Name);  var response = await next();  \_logger.LogInformation("Handled {Response}", typeof(TResponse).Name);  return response;  }  } |

Среди других примеров пайплайнов можно упомянуть TransactionBehavior для управления транзакциями или AuthorizationBehavior для централизованной проверки прав доступа.

Регистрация в Program.cs

var builder = WebApplication.CreateBuilder(args);

// Регистрация сервисов

builder.Services.AddScoped<IProductService, ProductService>();

builder.Services.AddSingleton<ILoggingService, LoggingService>();

builder.Services.AddTransient<IEmailSender, EmailSender>();

### VII. Тестирование приложения

#### 7.1. Юнит-тестирование обработчиков

При юнит-тестировании обработчиков MediatR существует распространенная ошибка, которая заключается в том, чтобы просто создать экземпляр обработчика и напрямую вызвать его метод Handle().27 Такой подход тестирует только сам класс, но игнорирует весь пайплайн MediatR. Правильный способ — тестировать поведение, а не класс, отправляя сообщение через посредника (\_mediator.Send()).27 Это гарантирует, что весь путь сообщения, включая пайплайны валидации, логирования и другие, будет проверен.

Для изоляции тестов от внешних зависимостей, таких как DbContext, используются тестовые дублеры (mocks).27 Например, для тестирования

CreateProductHandler можно имитировать IDbContextFactory и убедиться, что метод SaveChangesAsync был вызван с ожидаемыми данными.

#### 7.2. Интеграционное тестирование

Интеграционные тесты проверяют взаимодействие между несколькими компонентами системы, включая реальные зависимости.38 В контексте Blazor-приложения с CQRS это означает тестирование полного пути от UI до базы данных.

Для проведения интеграционных тестов в ASP.NET Core используется пакет Microsoft.AspNetCore.Mvc.Testing.39 Он позволяет создать тестовый веб-хост в памяти, который имитирует запуск реального приложения, включая все сервисы и конфигурацию.39 Тест может отправить HTTP-запрос на тестовый хост, который пройдет через весь стек (Blazor, MediatR, EF Core) и запишет данные в реальную базу данных (в нашем случае, SQLite). Затем тест может проверить, что данные были успешно сохранены.39

### VIII. Выводы и дальнейшие шаги

Применение архитектуры CQRS в сочетании с паттерном Посредник, реализованным через библиотеку MediatR, позволяет создать гибкое, масштабируемое и легко поддерживаемое приложение, даже если его базовая функциональность сводится к CRUD-операциям. Разделение ответственности чтения и записи делает код более читаемым, а использование пайплайнов MediatR позволяет централизовать сквозную логику, такую как валидация и логирование.

Тем не менее, стоит помнить, что CQRS — это не панацея. Как и любой сложный паттерн, он привносит дополнительную сложность в архитектуру, особенно на начальных этапах разработки.2 Это может быть чрезмерным для небольших проектов, где традиционная многоуровневая архитектура с единой моделью будет более чем достаточна.2

Следующим шагом в развитии подобной архитектуры может стать внедрение **Event Sourcing**.14 Этот паттерн предполагает хранение всех изменений состояния системы как последовательности событий, а не как конечного состояния. Event Sourcing и CQRS часто используются вместе, так как они взаимно дополняют друг друга. В такой архитектуре записи (команды) изменяют только события в хранилище, а чтение (запросы) использует денормализованные "материализованные представления", построенные на основе этих событий.4

Связанным с этим понятием является **конечная согласованность (eventual consistency)**.2 В системах с отдельными базами данных для чтения и записи изменения, внесенные командой, могут быть не сразу видны в модели чтения, что приводит к задержке. Конечная согласованность гарантирует, что данные со временем станут согласованными, но не предоставляет немедленной гарантии.41 Это важный компромисс, который необходимо учитывать при проектировании, особенно в приложениях, где требуется строгая согласованность в реальном времени.2

#### Источники

1. Clean ASP.NET Core API using MediatR and CQRS | Setup | by Esam Magdy - Medium, дата последнего обращения: августа 22, 2025, <https://medium.com/@dev.esam2014/mediatr-and-cqrs-291ba0dc5dfe>
2. CQRS Software Architecture Pattern: The Good, the Bad, and the ..., дата последнего обращения: августа 22, 2025, <https://medium.com/@emer.kurbegovic/cqrs-software-architecture-pattern-the-good-the-bad-and-the-ugly-efe48e8dcd14>
3. CQRS for Enterprise Web Development: What's in it for Business ..., дата последнего обращения: августа 22, 2025, <https://www.infoq.com/articles/cqrs-business-kaminski/>
4. CQRS Pattern - Azure Architecture Center | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>
5. Implementing CQRS with MediatR in .NET 8: A Complete Guide - DEV Community, дата последнего обращения: августа 22, 2025, <https://dev.to/adrianbailador/implementing-cqrs-with-mediatr-in-net-8-a-complete-guide-1kof>
6. Command and Query Responsibility Segregation (CQRS) | System Design, дата последнего обращения: августа 22, 2025, <https://www.karanpratapsingh.com/courses/system-design/command-and-query-responsibility-segregation>
7. Applying simplified CQRS and DDD patterns in a microservice - .NET | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/apply-simplified-microservice-cqrs-ddd-patterns>
8. CQRS Design Pattern C# - Code with Shadman, дата последнего обращения: августа 22, 2025, <https://codewithshadman.com/cqrs-design-pattern-csharp/>
9. CQRS - Command Query Responsibility Segregation Design Pattern - GeeksforGeeks, дата последнего обращения: августа 22, 2025, <https://www.geeksforgeeks.org/system-design/cqrs-command-query-responsibility-segregation/>
10. CQRS Pattern C# (How It Works For Developers) - IronPDF, дата последнего обращения: августа 22, 2025, <https://ironpdf.com/blog/net-help/cqrs-pattern-csharp/>
11. CQRS And MediatR : Step-by-Step. Command-Query Responsibility… | by Meisam Alifallhi, дата последнего обращения: августа 22, 2025, <https://medium.com/@araxis/implementing-cqrs-with-mediatr-library-step-by-step-sample-code-2fde29a0afaf>
12. Simple CQRS implementation with raw SQL and DDD - Kamil Grzybek, дата последнего обращения: августа 22, 2025, <https://www.kamilgrzybek.com/blog/posts/simple-cqrs-implementation-raw-sql-ddd>
13. When to use the CQRS design pattern? - architecture - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/8820748/when-to-use-the-cqrs-design-pattern>
14. CQRS + Event Sourcing – Step by Step - Daniel Whittaker, дата последнего обращения: августа 22, 2025, <https://danielwhittaker.me/2020/02/20/cqrs-step-step-guide-flow-typical-application/>
15. TuralSuleymani/the-real-DDD-CQRS-CleanArchitecture: A ... - GitHub, дата последнего обращения: августа 22, 2025, <https://github.com/TuralSuleymani/the-real-DDD-CQRS-CleanArchitecture>
16. Domain Driven Design and CQRS Success Stories : r/dotnet - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/dotnet/comments/1ir7s6o/domain_driven_design_and_cqrs_success_stories/>
17. CQRS and MediatR in ASP.NET Core - Code Maze, дата последнего обращения: августа 22, 2025, <https://code-maze.com/cqrs-mediatr-in-aspnet-core/>
18. MediatR Pipeline Behavior - TheCodeMan | Master .NET ..., дата последнего обращения: августа 22, 2025, <https://thecodeman.net/posts/mediatr-pipeline-behavior>
19. MediatR and Blazor Server? - Jon Hilton, дата последнего обращения: августа 22, 2025, <https://jonhilton.net/mediatr-and-razor-components/>
20. Repository, QCRS, mediator, etc. in Blazor web app - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/Blazor/comments/1fy36o8/repository_qcrs_mediator_etc_in_blazor_web_app/>
21. Blazor tutorial - Build your first web app | .NET, дата последнего обращения: августа 22, 2025, <https://dotnet.microsoft.com/en-us/learn/aspnet/blazor-tutorial/intro>
22. Getting Started with Blazor WebAssembly: A Complete Guide - DEV Community, дата последнего обращения: августа 22, 2025, <https://dev.to/hamza_zeryouh/getting-started-with-blazor-webassembly-a-complete-guide-3h8j>
23. Getting Started with Blazor WebAssembly App in Visual Studio, дата последнего обращения: августа 22, 2025, <https://blazor.syncfusion.com/documentation/getting-started/blazor-webassembly-app>
24. Microsoft.EntityFrameworkCore.Sqlite 9.0.8 - NuGet, дата последнего обращения: августа 22, 2025, <https://www.nuget.org/packages/microsoft.entityframeworkcore.sqlite>
25. EF Core tools reference (.NET CLI), дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/ef/core/cli/dotnet>
26. Validation with MediatR Pipeline and FluentValidation | by Yash ..., дата последнего обращения: августа 22, 2025, <https://medium.com/@yashsnab0/validation-with-mediatr-pipeline-and-fluentvalidation-23c08c7a5aa5>
27. MediatR: How to Quickly Test Your Handlers with Unit Tests, дата последнего обращения: августа 22, 2025, <https://goatreview.com/mediatr-quickly-test-handlers-with-unit-tests/>
28. Microsoft.EntityFrameworkCore 9.0.8 - NuGet, дата последнего обращения: августа 22, 2025, <https://www.nuget.org/packages/microsoft.entityframeworkcore>
29. ASP.NET Core Blazor with Entity Framework Core (EF Core) | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/aspnet/core/blazor/blazor-ef-core?view=aspnetcore-9.0>
30. ASP.NET Core Blazor with Entity Framework Core (EF Core)- Inject DbContext - Medium, дата последнего обращения: августа 22, 2025, <https://medium.com/@drift2galaxy/asp-net-core-blazor-with-entity-framework-core-ef-core-%E5%A6%82%E4%BD%95%E6%B3%A8%E5%85%A5dbcontext-d4fa2ff16822>
31. How to Implement Entity Framework Core Migrations in a .NET Console Application Using C# and VSCode - Ottorino Bruni, дата последнего обращения: августа 22, 2025, <https://www.ottorinobruni.com/how-to-implement-entity-framework-core-migrations-in-a-dotnet-console-application-using-csharp-and-vscode/>
32. Migrations Overview - EF Core | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/>
33. Entity Framework Core Migrations, дата последнего обращения: августа 22, 2025, <https://www.learnentityframeworkcore.com/migrations>
34. CQRS And MediatR Pattern Implementation Using .NET Core 6 Web API - C# Corner, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/cqrs-and-mediatr-pattern-implementation-using-net-core-6-web-api/>
35. Mastering Cross-Cutting Concerns with the Mediator R in .NET | by Ashen Dunusinghe, дата последнего обращения: августа 22, 2025, <https://medium.com/@avdunusinghe/mastering-cross-cutting-concerns-with-the-mediator-r-in-net-43c33ceba582>
36. Enhancing Request Pipelines with MediatR Behaviors - DEV Community, дата последнего обращения: августа 22, 2025, <https://dev.to/stevsharp/enhancing-request-pipelines-with-mediatr-behaviors-48h2>
37. Rethinking MediatR Validation: Moving from Pipeline to Domain ..., дата последнего обращения: августа 22, 2025, <https://goatreview.com/rethinking-mediatr-pipeline-validation-pattern/>
38. Implementing Unit Test .Net Core Application Using CQRS Handler - C# Corner, дата последнего обращения: августа 22, 2025, <https://www.c-sharpcorner.com/article/implementing-unit-test-net-core-application-using-cqrs-handler/>
39. Integration tests in ASP.NET Core | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/aspnet/core/test/integration-tests?view=aspnetcore-9.0>
40. Implementing CQRS and Event Sourcing with C# - DEV Community, дата последнего обращения: августа 22, 2025, <https://dev.to/chakewitz/implementing-cqrs-and-event-sourcing-with-c-3224>
41. Eventual consistency - Wikipedia, дата последнего обращения: августа 22, 2025, <https://en.wikipedia.org/wiki/Eventual_consistency>