# Детальная лекция: Паттерн CQRS (C#) с MediatR, Blazor и SQLite

## 1. Введение: Декомпозиция сложности с помощью CQRS

Традиционные подходы к разработке программного обеспечения, основанные на единой модели данных и монолитном слое бизнес-логики, часто сталкиваются с проблемами масштабируемости, производительности и поддержки по мере роста сложности приложения. Паттерн CQRS (Command Query Responsibility Segregation) предлагает элегантное решение этих проблем, разделяя операции, которые изменяют данные (команды), от операций, которые их извлекают (запросы).1 Это разделение является ключевым для создания гибких, производительных и масштабируемых систем, поскольку оно позволяет независимо оптимизировать каждую часть.3

### 1.1. От принципа CQS к паттерну CQRS

Корни CQRS уходят в фундаментальный принцип объектно-ориентированного программирования, известный как CQS (Command-Query Separation), сформулированный Бертраном Мейером. Этот принцип гласит, что каждый метод должен либо изменять состояние объекта (быть **командой**), либо возвращать данные (быть **запросом**), но не делать и то, и другое одновременно.2 Например, метод Add в коллекции является командой, так как он изменяет состояние, а метод Count — запросом, так как он только возвращает данные.2 Паттерн CQRS масштабирует эту идею на архитектурный уровень, полностью разделяя логику и модели данных всего приложения на две отдельные части.2

В традиционном монолитном приложении одна и та же модель данных и один и тот же слой логики используются как для записи (операции, требующие строгой валидации, надежных транзакций и высокого уровня безопасности), так и для чтения (операции, где критически важны скорость и оптимизация для отображения в пользовательском интерфейсе). Это создает внутреннее противоречие. CQRS же разрывает этот порочный круг. Он признает, что цели операций записи и чтения различны, и поэтому позволяет использовать для каждой из них специализированные модели.3 Такое разделение ответственности делает код более прозрачным и облегчает его поддержку, поскольку каждая часть выполняет только одну, четко определенную задачу.1

### 1.2. Роль Медиатора

Для эффективной реализации CQRS необходим механизм, который бы управлял потоком команд и запросов, направляя их к нужным обработчикам. Таким механизмом является паттерн "Медиатор", а библиотека MediatR — его наиболее популярная реализация в экосистеме.NET.1

MediatR выполняет роль центрального диспетчера, который обеспечивает полную декуплировку (развязку) между отправителем (например, Blazor-компонент или контроллер в ASP.NET) и получателем (обработчиком команды или запроса).1 Вместо того чтобы отправитель напрямую создавать и вызывать new AddProductCommandHandler(), он просто отправляет команду или запрос с помощью медиатора (\_mediator.Send(command)). MediatR затем автоматически находит и вызывает соответствующий обработчик.1

Эта архитектура предоставляет несколько ключевых преимуществ:

* **Снижение связанности:** Компоненты приложения не зависят друг от друга напрямую. Контроллер или Blazor-компонент зависит только от интерфейса IMediator, а обработчик — только от своего типа команды или запроса. Это значительно упрощает управление зависимостями и изменениями в коде.1
* **Упрощение тестирования:** Отправитель и обработчик могут быть протестированы независимо, так как они не имеют прямых зависимостей друг от друга.1 Это позволяет писать чистые и сфокусированные юнит-тесты.7
* **Гибкость и расширяемость:** Возможность легко добавить новое поведение к любому запросу с помощью конвейера MediatR (MediatR Pipeline), не изменяя основной код обработчиков. Это позволяет реализовать сквозные задачи, такие как логирование или валидация, централизованно.5

Без MediatR реализация CQRS была бы чрезмерно сложной, требуя ручного управления зависимостями и множеством прямых вызовов. Таким образом, MediatR является не просто вспомогательной библиотекой, а стратегическим выбором, который делает паттерн CQRS управляемым и практичным в реальных проектах.

## 2. Архитектурная модель: Детализация и расширение

### 2.1. Модель команд (The Write Model)

Модель команд отвечает за все операции, которые изменяют состояние системы. Это сердце бизнес-логики приложения.1 Она включает в себя два основных компонента:

* **Команды (Commands)**: Это неизменяемые объекты-сообщения, которые передают *намерение* пользователя выполнить определенное бизнес-действие. Важно, чтобы команды представляли собой осмысленные задачи, а не низкоуровневые операции с данными. Например, вместо команды SetReservationStatusToReserved, более подходящей является BookHotelRoomCommand, так как она лучше отражает бизнес-процесс.1
* **Обработчики команд (Command Handlers)**: Это классы, которые получают команду и выполняют всю необходимую бизнес-логику для ее обработки. Они отвечают за валидацию, взаимодействие с хранилищем данных и изменение состояния системы.1 Часто обработчики команд работают с так называемыми "агрегатами" (Aggregate Root) из Domain-Driven Design (DDD), которые представляют собой кластеры связанных объектов, обеспечивающих целостность данных в рамках одной транзакции.1

### 2.2. Модель запросов (The Read Model)

Модель запросов отвечает исключительно за получение данных. Эта часть приложения не содержит бизнес-логики и никогда не изменяет состояние системы.1 Она состоит из:

* **Запросы (Queries)**: Это неизменяемые объекты, которые передают *требование* на получение данных. Как и команды, они должны быть описательными и отражать потребность клиента в информации, например GetAllProductsQuery.1
* **Обработчики запросов (Query Handlers)**: Это классы, которые извлекают данные из хранилища и преобразуют их в DTO-объекты (Data Transfer Object), оптимизированные для отображения в пользовательском интерфейсе. Поскольку они не содержат бизнес-логики, их можно оптимизировать для максимальной скорости чтения, например, используя кэширование или сложные SQL-запросы без ограничений, накладываемых ORM.1

Разделение моделей записи и чтения позволяет использовать разные схемы данных, что является ключевым преимуществом CQRS.1 В то время как модель записи может использовать нормализованную схему, оптимизированную для транзакций и обеспечения целостности, модель чтения может использовать денормализованные "материализованные представления" (materialized views), чтобы избежать сложных соединений (joins) и обеспечить мгновенное получение данных.1

### 2.3. Сравнение подходов к хранилищу

Представленный в запросе пример с Blazor, MediatR и SQLite демонстрирует базовый, но очень эффективный подход к CQRS, где обе модели — и записи, и чтения — используют одну и ту же базу данных.

#### Единая база данных (Simple CQRS)

В этом случае CQRS-паттерн применяется на логическом уровне, разделяя код, но используя единую физическую базу данных. Это простейшая форма CQRS, которая позволяет получить значительные преимущества от разделения ответственности без излишней сложности.1 Такой подход идеально подходит для большинства приложений, поскольку устраняет необходимость в управлении распределенными системами и проблемами синхронизации данных.

#### Раздельные базы данных (Advanced CQRS)

Более продвинутая форма CQRS предполагает использование отдельных физических баз данных для операций записи и чтения.1

* **База данных для записи (Write Database)**: Оптимизирована для высокой надежности и атомарных транзакций. Обычно это традиционная реляционная СУБД.10
* **База данных для чтения (Read Database)**: Оптимизирована для скорости чтения. Это может быть кэш (например, Redis), поисковый движок (Elasticsearch) или NoSQL-база данных, где данные хранятся в денормализованном виде, готовом для быстрого отображения.10

Такая архитектура позволяет независимо масштабировать каждую часть, что крайне важно для высоконагруженных систем, где количество операций чтения в разы превосходит количество операций записи.2

#### Проблема итоговой согласованности (Eventual Consistency)

При использовании раздельных баз данных возникает проблема "итоговой согласованности" (eventual consistency).1 Это означает, что после изменения данных в базе записи они не сразу отражаются в базе чтения. Возникает временная задержка, в течение которой данные в модели чтения могут быть устаревшими.

Eventual consistency — это компромисс между производительностью, доступностью и строгой согласованностью данных.13 В системах с высокими требованиями к производительности и доступности, таких как социальные сети или торговые платформы, этот компромисс часто оправдан. Однако для приложений, где важна абсолютная согласованность данных в реальном времени (например, финансовые транзакции), этот подход может быть неприемлем.1 Управление итоговой согласованностью требует тщательного проектирования, но дает возможность создавать системы, способные выдерживать колоссальные нагрузки.

## 3. Расширение функционала: Конвейер MediatR

В предоставленном примере отсутствуют сквозные задачи (cross-cutting concerns), такие как валидация или логирование, которые являются неотъемлемой частью любого production-приложения. MediatR предоставляет мощный механизм для централизованного решения этих проблем — конвейер запросов (MediatR Pipeline).

### 3.1. Что такое конвейер MediatR?

Конвейер MediatR — это архитектурный шаблон, который позволяет реализовать логику, "оборачивающую" обработчик запроса. Он основан на паттерне "Декоратор" (Decorator Pattern), где каждая часть конвейера добавляет функциональность к следующему компоненту в цепочке.15 Это достигается путем реализации интерфейса

IPipelineBehavior<TRequest, TResponse>.8

Каждый IPipelineBehavior имеет метод Handle, который принимает входящий запрос и делегат next, представляющий следующий шаг в конвейере. Это позволяет выполнять логику *до* и *после* фактического вызова основного обработчика, обеспечивая централизованное и повторно используемое управление такими задачами, как:

* **Логирование:** Запись информации о запросе и ответе.8
* **Валидация:** Проверка данных перед выполнением бизнес-логики.9
* **Кэширование:** Проверка наличия данных в кэше.2
* **Управление транзакциями:** Обеспечение атомарности операций.5

Конвейер MediatR позволяет разработчику избавиться от дублирования кода в обработчиках и придерживаться принципа единой ответственности (Single Responsibility Principle), поскольку каждый обработчик занимается только своей основной задачей.15

### 3.2. Практическая реализация: Примеры поведения

#### Поведение для логирования (Logging Behavior)

Для реализации централизованного логирования создается класс, который оборачивает каждый запрос.

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,  
 CancellationToken cancellationToken,  
 RequestHandlerDelegate<TResponse> next)  
 {  
 \_logger.LogInformation($" Handling command/query: {typeof(TRequest).Name}");  
 var response = await next();  
 \_logger.LogInformation($" Handled command/query: {typeof(TRequest).Name}");  
 return response;  
 }  
}

Такое поведение регистрируется в файле Program.cs с помощью AddTransient(typeof(IPipelineBehavior<,>), typeof(LoggingBehavior<,>));.8 После этого каждый запрос, отправленный через MediatR, будет автоматически регистрироваться в журнале, что обеспечивает целостный аудит без необходимости вручную добавлять логику в каждый обработчик.8

#### Поведение для валидации (Validation Behavior)

Валидация является одной из наиболее распространенных сквозных задач. Здесь существует два подхода:

**Подход 1: Централизованная валидация в конвейере с FluentValidation**

Этот подход предполагает использование популярной библиотеки FluentValidation для централизованной валидации запросов. Логика проверки отделяется от обработчиков и размещается в отдельных классах-валидаторах.9

1. **Установка пакетов**: Добавление FluentValidation и FluentValidation.DependencyInjectionExtensions.
2. **Регистрация**: Валидаторы регистрируются в DI-контейнере с помощью builder.Services.AddValidatorsFromAssembly(...).9
3. **Поведение**: Создается ValidationBehavior, который получает все валидаторы для данного типа запроса, выполняет их и в случае ошибок выбрасывает FluentValidation.ValidationException.9

Этот метод позволяет очистить обработчики от валидационной логики, делая их более сфокусированными на бизнес-правилах.9

**Подход 2: "Parse, Don't Validate" (Разбирай, а не валидируй)**

Альтернативный, более строгий подход, основанный на принципах DDD, гласит: "Разбирай, а не валидируй".16 Он утверждает, что невалидные данные не должны даже передаваться по системе. Вместо этого, валидация должна происходить непосредственно при создании команды. Если данные не соответствуют бизнес-правилам (например, длина имени продукта меньше трех символов), команда просто не может быть создана.16 Это делает бизнес-правила явными в самой модели и предотвращает ненужную нагрузку на систему, связанную с обработкой невалидных запросов.16

Выбор между этими двумя подходами зависит от философии команды и требований к архитектуре. Валидация в конвейере с FluentValidation является более гибкой и привычной, в то время как "Parse, Don't Validate" обеспечивает более строгую, безопасную и теоретически обоснованную модель.16

В целом, конвейер MediatR — это мощный инструмент, который превращает базовый CQRS-проект в надежную и поддерживаемую систему, готовую к production-среде.

## 4. Тестирование: Обеспечение надежности

Разделение логики на команды и запросы в рамках паттерна CQRS делает систему *более* тестируемой. Вместо тестирования больших, монолитных классов-репозиториев, разработчик получает небольшие, сфокусированные обработчики, которые легко изолировать и тестировать.17

### 4.1. Юнит-тестирование (Unit Testing)

Цель юнит-тестирования — проверить поведение отдельного компонента, такого как обработчик команды или запроса, в полной изоляции.17 В этом случае все внешние зависимости, например

ApplicationDbContext, должны быть заменены на "дублеров" (test doubles) — моки (mocks) или заглушки (stubs).7

Для тестирования обработчика, который зависит от DbContext, можно использовать библиотеку Moq.

1. **Создание мока**: Создается мок IDbContext для имитации поведения реальной базы данных.
2. **Изоляция**: В тест передается мок, а не реальный экземпляр DbContext.
3. **Проверка**: Тест проверяет, что обработчик выполнил свою логику корректно (например, вызвал \_context.Products.Add() и \_context.SaveChangesAsync()).

Этот подход позволяет убедиться, что бизнес-логика внутри обработчика работает правильно, не полагаясь на работоспособность реальной базы данных.7

### 4.2. Интеграционное тестирование (Integration Testing)

В отличие от юнит-тестов, интеграционные тесты проверяют, как различные компоненты системы — HTTP-контроллеры, MediatR, обработчики и **реальная база данных** — работают вместе как единое целое.17

Для интеграционного тестирования ASP.NET Core-приложений рекомендуется использовать WebApplicationFactory.19

1. **Настройка**: Создается тестовый проект, который использует WebApplicationFactory для запуска приложения в памяти.19
2. **Тестирование**: Используя HttpClient, тест отправляет реальные HTTP-запросы к приложению (например, POST /products или GET /products) и проверяет, что ответ соответствует ожиданиям.19
3. **База данных**: Интеграционные тесты часто используют отдельную in-memory базу данных (например, SqliteInMemory для тестов) или контейнер Docker для тестирования с реальной СУБД.6

Цель такого тестирования — убедиться, что весь сквозной "путь" запроса, от его получения до сохранения в БД, работает без сбоев.17 Интеграционные тесты особенно важны в архитектуре CQRS, поскольку они проверяют, что все слои (API, MediatR, обработчики, БД) корректно взаимодействуют между собой.

## 5. Продвинутые концепции: Event Sourcing

### 5.1. Что такое Event Sourcing?

Event Sourcing (ES) — это архитектурный паттерн, который хранит каждое изменение состояния приложения как последовательность неизменяемых событий. Вместо того чтобы сохранять только текущее состояние объекта, система сохраняет все события, которые привели к этому состоянию.11 Например, в банковской системе вместо сохранения текущего баланса счета, Event Sourcing будет хранить все транзакции (DepositEvent, WithdrawalEvent), которые когда-либо произошли. Текущий баланс можно "воспроизвести", применив эти события в хронологическом порядке.21

Журнал событий (Event Store) становится единственным источником истины. Поскольку события неизменяемы, они формируют полный и аудируемый журнал всех изменений, которые когда-либо происходили в системе.11

### 5.2. CQRS + Event Sourcing: Мощное сочетание

Event Sourcing естественным образом дополняет и усиливает паттерн CQRS.21 В такой архитектуре:

* **Модель записи (Write Model)**: Журнал событий становится единственным источником истины для модели записи.1 Когда приходит команда (  
  AddProductCommand), обработчик команды генерирует соответствующее событие (ProductAddedEvent) и сохраняет его в Event Store.
* **Модель чтения (Read Model)**: Модель чтения строит свои оптимизированные представления (materialized views), "воспроизводя" события из Event Store. Например, для создания представления "все продукты" система будет считывать все события ProductAddedEvent и ProductUpdatedEvent и создавать на их основе денормализованные представления, которые будут использоваться для быстрых запросов.1

Это сочетание решает проблему итоговой согласованности в распределенных системах, так как модель чтения всегда может быть перестроена на основе единственного источника истины — журнала событий.1 Однако это также добавляет значительную сложность, связанную с обработкой событий и их синхронизацией.

### 5.3. События и нотификации MediatR

Для реализации событийной архитектуры в рамках одного процесса MediatR предоставляет механизм нотификаций (INotification). В отличие от команд и запросов, у которых есть только один обработчик, нотификации могут иметь несколько обработчиков (INotificationHandler), которые могут реагировать на одно и то же событие.22 Например, после сохранения заказа в базе данных обработчик команды может опубликовать нотификацию

OrderCreatedNotification. Эта нотификация может быть подхвачена несколькими обработчиками, которые работают параллельно: один отправляет email-уведомление клиенту, другой создает запись в логе для аудита, третий обновляет кэш.22

С версии MediatR v12 разработчики могут настраивать стратегию публикации нотификаций:

* ForeachAwaitPublisher: Стратегия по умолчанию, которая выполняет обработчики последовательно.22
* TaskWhenAllPublisher: Выполняет все обработчики параллельно, что может значительно улучшить производительность, особенно в сценариях, где обработчики выполняют независимые длительные операции (например, отправка HTTP-запросов).22

## 6. Сравнение и заключение

6.1. Когда использовать и когда избегать CQRS?

Паттерн CQRS — это мощный инструмент, но он не является универсальным решением для всех задач. Его применение должно быть обосновано и соответствовать требованиям проекта.

**Таблица 1: Сравнение CRUD и CQRS**

|  |  |  |
| --- | --- | --- |
| **Параметр** | **Традиционный CRUD** | **Паттерн CQRS** |
| **Модель данных** | Единая модель для чтения и записи | Раздельные модели (Write/Read) |
| **Бизнес-логика** | Смешанная в одном слое (репозиторий/сервис) | Раздельная: логика записи в обработчиках команд, логика чтения отсутствует |
| **Масштабируемость** | Ограниченная (монолит) | Независимое масштабирование операций чтения и записи |
| **Безопасность** | Ограниченная (одни и те же объекты для всех операций) | Повышенная (возможность применять разные политики к разным моделям) |
| **Сложность** | Низкая, простая кривая обучения | Высокая, сложная кривая обучения |

**Преимущества**

* **Независимое масштабирование:** Операции чтения и записи могут масштабироваться независимо, что крайне важно для высоконагруженных систем, где запросов на чтение гораздо больше, чем на запись.1
* **Гибкость схем данных:** Возможность использовать различные оптимизированные схемы данных для чтения и записи.1
* **Повышенная безопасность:** Легче применять строгие политики безопасности к операциям, которые изменяют данные, так как они четко отделены от операций чтения.1
* **Разделение ответственности:** Четкое разделение кода на модели записи и чтения делает систему более чистой, поддерживаемой и понятной.1

**Недостатки**

* **Повышенная сложность:** CQRS увеличивает количество классов и файлов, что может быть излишним для простых приложений.3
* **Проблема итоговой согласованности:** В системах с раздельными базами данных возникает проблема eventual consistency, что требует дополнительного внимания к синхронизации данных.1
* **Кривая обучения:** Паттерн требует от разработчиков понимания новых концепций, что может замедлить процесс разработки на начальном этапе.3
* **Увеличенный объем кода:** Для реализации простой CRUD-операции требуется больше кода, чем в традиционном подходе.24

### 6.2. CQRS vs. N-Tier Architecture

Традиционная N-уровневая архитектура (N-Tier) объединяет логику чтения и записи в одном слое. Это приводит к тому, что для оптимизации производительности одной части системы приходится масштабировать всю бизнес-логику целиком, что является неэффективным расходованием ресурсов.12 CQRS, напротив, позволяет оптимизировать каждый компонент по отдельности, что делает его более подходящим для "продвинутых веб-решений со сложными совместными контекстами" и большим количеством одновременных пользователей.12

### 6.3. Заключение: Путь разработчика

Паттерн CQRS — это мощный инструмент, который стоит иметь в своем арсенале. Однако его следует применять с осторожностью. Для простых CRUD-приложений (создание, чтение, обновление, удаление) дополнительная сложность, которую он вносит, является избыточной.3

Лучший подход — это эволюционное развитие. Начинайте с простого CQRS, как в предоставленном примере, используя одну базу данных. Это позволит команде освоить принципы разделения ответственности и преимущества использования MediatR. И только тогда, когда возникнут реальные проблемы с производительностью или масштабированием, оправдывающие возросшую сложность, можно переходить к более продвинутым формам, таким как раздельные базы данных или Event Sourcing.18

|  |  |  |
| --- | --- | --- |
| **Тип поведения** | **Назначение** | **Используемые технологии** |
| Логирование | Запись информации о входящих запросах и ответах | ILogger |
| Валидация | Централизованная проверка данных перед выполнением логики | FluentValidation, IPipelineBehavior |
| Управление транзакциями | Обеспечение атомарности операций, затрагивающих несколько сущностей | DbContext |
| Обработка исключений | Централизованная обработка ошибок в едином месте | try/catch |

#### Источники

1. CQRS Pattern - Azure Architecture Center | Microsoft Learn, дата последнего обращения: августа 22, 2025, <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>
2. CQRS Design Pattern C# - Code with Shadman, дата последнего обращения: августа 22, 2025, <https://codewithshadman.com/cqrs-design-pattern-csharp/>
3. 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>
4. 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>
5. 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>
6. gothinkster/aspnetcore-realworld-example-app: ASP.NET Core backend implementation for RealWorld - GitHub, дата последнего обращения: августа 22, 2025, <https://github.com/gothinkster/aspnetcore-realworld-example-app>
7. 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/>
8. MediatR Pipeline Behavior - TheCodeMan | Master .NET ..., дата последнего обращения: августа 22, 2025, <https://thecodeman.net/posts/mediatr-pipeline-behavior>
9. 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>
10. 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/>
11. 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/>
12. 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/>
13. Eventual consistency - Wikipedia, дата последнего обращения: августа 22, 2025, <https://en.wikipedia.org/wiki/Eventual_consistency>
14. 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>
15. 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>
16. Rethinking MediatR Validation: Moving from Pipeline to Domain ..., дата последнего обращения: августа 22, 2025, <https://goatreview.com/rethinking-mediatr-pipeline-validation-pattern/>
17. 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/>
18. 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>
19. 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>
20. How to instantiate Mediatr as part of a Unit Test? - Stack Overflow, дата последнего обращения: августа 22, 2025, <https://stackoverflow.com/questions/55443201/how-to-instantiate-mediatr-as-part-of-a-unit-test>
21. 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>
22. How To Publish MediatR Notifications In Parallel - Milan Jovanović, дата последнего обращения: августа 22, 2025, <https://www.milanjovanovic.tech/blog/how-to-publish-mediatr-notifications-in-parallel>
23. Tackling Complexity in CQRS - Rants on Software Design, дата последнего обращения: августа 22, 2025, <https://vladikk.com/2017/03/20/tackling-complexity-in-cqrs/>
24. Split command/query classes vs monolithic repository? : r/csharp - Reddit, дата последнего обращения: августа 22, 2025, <https://www.reddit.com/r/csharp/comments/1mhdqbc/split_commandquery_classes_vs_monolithic/>