Списание „Математика, компютърни науки и образование“ Том 7, кн. 2, 2024

Journal of Mathematics, Computer Science and Education Vol. 7, No. 2, 2024

DOI: 10.54664/URDG9808

ИМПЛЕМЕНТАЦИЯ НА ШАБЛОНИТЕ ЗА СОФТУЕРЕН ДИЗАЙН ПОСРЕДНИК (MEDIATOR) И CQRS С БИБЛИОТЕКАТА MEDIATR В .NET

Пламенна Петрова, Златко Върбанов

IMPLEMENTATION OF THE SOFTWARE DESIGN PATTERNS MEDIATOR AND CQRS WITH THE LIBRARY MEDIATR IN .NET

Plamenna Petrova, Zlatko Varbanov

Abstract: This paper explores the features of the popular open-source library from the .NET ecosystem MediatR, which implements the behavioral software design pattern Mediator to achieve loose coupling and better communication between the components of developed applications. It introduces an ‘in-process’ mediator, which aids in building systems, based on the Command and Query Responsibility Segregation software design pattern (CQRS). The MediatR library offers the ability to handle both synchronous and asynchronous requests, responses, commands, notifications, and events. It provides a straightforward approach for managing command and request handlers, simplifying the adoption of CQRS. Using the MediatR library improves the modularity and testability of .NET applications by promoting adherence to SOLID principles in the code structure. This facilitates complex interactions between different parts of applications, the integration of new functionalities and the maintenance of existing ones.

Keywords: Software Design Patterns, GoF, Mediator, Command and Query Responsibility Segregation (CQRS), MediatR, .NET, C#, API, SOLID principles, Domain-driven Design (DDD), Clean Architecture, Separation of concerns

ВЪВЕДЕНИЕ

MediatR е мощна библиотека, която предоставя многобройни възможности на разработчиците на уеб приложения, което я прави основен инструмент в създаването на съвременен софтуер. Едно от ключовите предимства на интеграцията на MediatR в проекти, базирани на .NET платформата, е постигането на по-чист код. Използването на MediatR опростява цялостната кодова база чрез намаляване на зависимостите и улесняване на отделянето на подателя и получателя на заявки [1]. Това преодолява тясното свързване между компонентите и подобрява организацията и поддръжката на приложенията. Имплементацията на поведенческия шаблон за софтуерен дизайн Посредник (Mediator) от MediatR позволява изпращане на съобщения за действие между различни части на приложението без те да взаимодействат пряко помежду си [6]. По този начин се улеснява разширяването на кодовата база, като същевременно се насърчава модулността и се подобрява четимостта на кода. Освен това, MediatR безпроблемно имплементира и шаблона за софтуерен дизайн за Разделяне на отговорността на команди и заявки (CQRS). CQRS разделя операциите за четене (заявки) от тези за запис (команди), осигурявайки ясно разграничаване между извличането и модифицирането на данни [4]. С поддръжката на MediatR за CQRS, командите, заявките и известията могат да бъдат успешно обработвани по централизиран начин, което допринася за по-добра мащабируемост на приложенията, позволявайки да се оптимизират поотделно сценарии с тежка наситеност на операции за четене и запис, като се достигат оптимални нива на

производителност [1]. С интеграцията на MediatR софтуерните разработчици могат да рационализират процеса на обработка на команди, заявки и известия, да повишат качеството на кода си и да проектират по-ефективни системи чрез разделяне на отговорностите въз основа на крайни цели.

Изложение

шаблонът за софтуерен дизайн Посредник (Mediator) е поведенчески шаблон за дизайн, който насърчава слабото свързване между обектите чрез капсулиране на техните взаимодействия в обект в ролята на посредник, контролиращ и координиращ комуникацията между тях, без знанието им един за друг [3]. Чрез въвеждане на обект-посредник, обектите могат да комуникират чрез него, намалявайки сложността и улеснявайки поддържането на кода.

Шаблонът Посредник (Mediator) се състои от следните участници:

Посредник	Интерфейс или абстрактен клас, който дефинира комуникационния протокол между обектите. Обикновено включва методи за регистриране, изпращане и получаване на съобщения от обектите.
Конкретен посредник	Конкретна имплементация на интерфейса на посредника, който познава обектите, с които посредникът взаимодейства, и техните комуникационни протоколи.
Колега	Интерфейс или абстрактен клас, който определя поведението на обектите, които комуникират чрез посредника. Обикновено включва методи за изпращане и получаване на съобщения.
Конкретен колега	Конкретна имплементация на интерфейса на колегата, който познава своя посредник и го използва за комуникация с други колеги.

Към предимствата на шаблона Посредник (Mediator) спадат:

- Намалено свързване: обектите не достъпват информация за други обекти, а само посредникът [12]
- Опростена комуникация: обектите комуникират през една точка, с което се спазва принципа на единичната отговорност и се намалява сложността на взаимодействие [12]
- Повишена гъвкавост: посредникът може да промени начина, по който обектите комуникират, без те да бъдат засегнати [12]
- Спазване на принципа „отворен/затворен“ (open/closed) : могат да бъдат въведени нови посредници, без да се налага промяната на действителни компоненти [13]

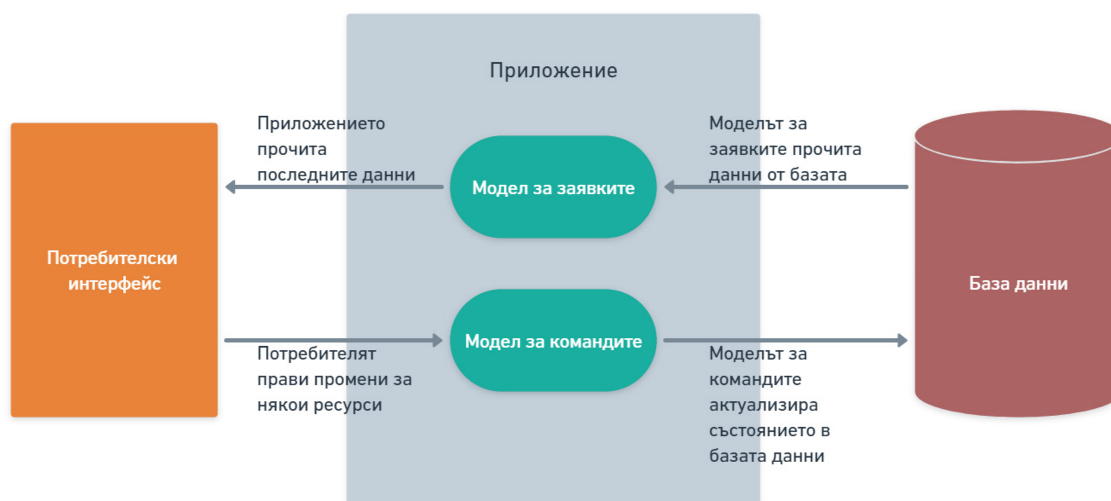
Въпреки това за шаблона Посредник (Mediator) могат да бъдат посочени някои недостатъци, които са:

- Увеличена сложност: логиката за посредника може да се усложни, ако трябва да бъдат обработвани много обекти и комуникационни протоколи. [12]
- Единична точка на отказ: ако посредникът се провали, комуникацията между обектите ще бъде отказана [12]
- Разход на производителност: посредникът добавя допълнителен слой на комуникация, което може да забави системата [12]

В обобщение, шаблонът Посредник (Mediator) служи за намаляване на свързването и опростяване на комуникацията между обектите, като отделя техните взаимодействия и ги централизира в обект-посредник [11], поради което имплементацията му помага в изграждането на кор-

поративни приложения, където е необходимо да се управляват и следят взаимодействията между множество обекти.

CQRS е съкращение от Command and Query Responsibility Segregation и означава разделяне на отговорността на команди и заявки. Той е софтуерен шаблон за дизайн, който разделя операциите за четене и запис на системата в различни модели. В софтуерна архитектура, базирана на CQRS, операциите за запис (команди) и операциите за четене (заявки) се обработват отделно, като се прилагат съответните модели за всяка операция [9]. Традиционните архитектурни шаблони често използват един и същ модел на данни или обект за трансфер на данни както за заявки, така и за запис в източник на данни. Този подход работи добре за основните операции за създаване, четене, актуализиране и изтриване, още познати като CRUD (Create-Read-Update-Delete) операции, но може да стане ограничаващ с развитието на приложенията и нарастването на сложността на софтуерните изисквания [7]. CQRS разрешава проблема с усложняването на логическия модел, разделяйки отговорностите за обработка на команди и заявки. По този начин всеки модел може да бъде оптимизиран за изпълнение на специфична цел, с което се подобряват общата производителност и скалируемост [8]. CQRS също така осигурява гъвкавост и по-ефективна обработка на данни, улеснявайки добавянето на нови функции или извършването на промени в системата, без да се нарушава структурата на други модули [16].



Фиг. 1. *Принципи на работа на шаблона за софтуерен дизайн CQRS*

Схемата показва разделянето на модела за заявки от модела за команди в приложението. Шаблонът CQRS не поставя официални изисквания за начина, по който ще се извърши това разделяне. То може да варира от клас, намиращ се в същото приложение до отделни физически приложения на различни сървъри, в зависимост от фактори като изисквания за мащабиране и инфраструктура [10]. Командите задействат манипулатори, които запазват обекти в база данни и връщат идентификатори. Заявките си служат с манипулатори за извличане на данни, превръщайки ги в обекти за трансфер на данни, които ще бъдат върнати като отговор на клиента. Поделянето на операциите в модели за заявки и команди предоставя възможност за свързване с различни бази данни за четене и запис, въпреки че е достатъчно да се използва единствено проста база данни в паметта, като се поддържа логическо разпределение на работния поток [7].

Към предимствата на шаблона CQRS спадат още:

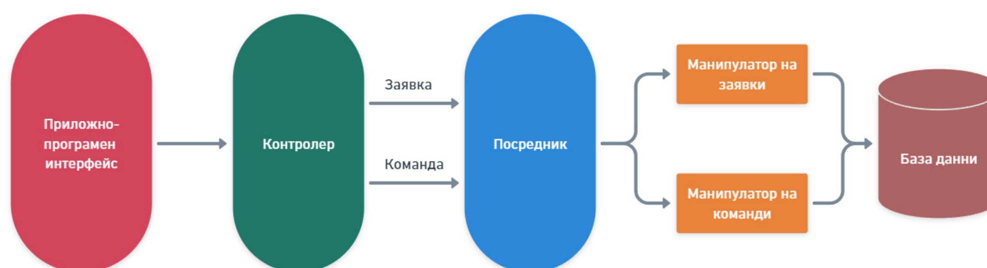
- Подобрена конкурентност и паралелизъм: с въвеждането на специални модели за всяка операция, CQRS гарантира, че паралелните операции са защитени и че ще се запази целостта на данните. Това се явява полезно в сценарии, при които трябва да се изпълняват едновременно множество операции [7].

- Подобрена сигурност: разделянето на отговорности в CQRS помага за осигуряване на достъп до данни. В резултат от дефинирането на ясни граници между операциите за четене и запис, може да се имплементират по-подробни механизми за контрол на достъпа, които ще подобрят цялостната сигурност на приложението [7].
- Фокус върху домейна (основната логика и правила): CQRS позволява дизайна на системата да се фокусира повече върху бизнес логиката и поведението, вместо да решава проблеми с достъпа до данни [15].

Недостатъците на шаблона CQRS се състоят в:

- Повишена сложност и обем на кода: Имплементацията на шаблона CQRS често води до значително увеличаване на необходимия код. Тази сложност възниква от необходимостта да се управляват отделните модели и манипулатори за операции за четене и запис, което може да бъде предизвикателство за поддръжка и отстраняване на грешки [7].
- Проблеми със съгласуваността на данните: CQRS може да доведе до евентуални проблеми със съгласуваността на данните, при които моделите на заявките може да не отразяват веднага най-новите промени, направени от командите. Затова съгласуваността на данните в двата модела налага внимателно планиране и изпълнение [5].
- Допълнителна натовареност: моделите за четене и запис трябва да се синхронизират с помощта на CQRS, което изисква допълнително натоварване на системата и може да усложни нейната синхронизация [5].

В .NET приложенията крайните точки на контролерите или минималните приложно-програмни интерфейси трябва да се съсредоточават върху обработката на входящи заявки, насочването им към подходящите услуги или компоненти на бизнес логика и връщане на отговори. Добра практика е да се разтоварва сложната бизнес логика, валидирането на данните и други тежки задачи към отделни класове услуги или библиотеки [14]. Придържането към принципа на разделяне на проблеми подобрява тестването и скалируемостта на приложенията. Библиотеката MediatR е създадена именно с тази цел. Тя имплементира шаблона Посредник (Mediator) за постигане на слаба свързаност и по-добра комуникация между компонентите в разработваните приложения, въвеждайки процесен посредник, който помага за изграждането на системи, базирани на шаблона за разделяне на отговорността на команди и заявки CQRS [10]. Имплементацията на шаблона Посредник (Mediator) насърчава по-организирана и управляема кодова база чрез централизиране на комуникационната логика. В контекста на CQRS, шаблонът Посредник (Mediator) е особено полезен. CQRS разделя операциите за четене и запис, а шаблонът Посредник (Mediator) може да помогне при координирането на тези операции, действайки като свързващо звено между командите и заявките [7]. С комбинацията на шаблоните Посредник (Mediator) и CQRS от библиотеката MediatR може да се осигури по-чиста архитектура, при която командите се обработват отделно от заявките, което е предпоставка за по-лесно поддържане и мащабиране на системите [2].



Фиг. 2. *Работен поток на библиотеката MediatR*

MediatR предлага работа със синхронни и асинхронни заявки, отговори, команди, известия и събития [10]. Библиотеката предоставя различни видове посредници, включително за заявки или отговори, известия и поведение на конвейера [14]. Работният поток на MediatR е показан на Фигура 2 и той включва изпращането на команди и заявки от крайните точки на приложно-програмните интерфейси към процесния посредник, обработването на всяка заявка или команда от манипулатори, които са отговорни за изпълнението им и връщането на отговори, както и взаимодействие с база данни.

При конструирането на команди и заявки е препоръчително да се използва специалния тип record в C#, предназначен за създаване на неизменяеми типове данни, тъй като в архитектура, базирана на CQRS, командите и заявките често са неизменяеми обекти. Структурата на record за създаване на команда или заявка има следният синтаксис:

```
        public record CreateProductCommand(CreateProductRequest CreateProduc-
tRequest) : IRequest<ProductResponse>;
```

Показано е дефинирането на команда, при което се извършва имплементация на генеричния интерфейс IRequest от библиотеката, който обозначава командата или заявката, която ще бъде обработена от съответния манипулатор. Указват се и специфичният тип на обекта, който ще бъде подаден на командата, и типът на отговора, който ще бъде върнат след нейната обработка.

Манипулаторите на команди и заявки имплементират генеричния интерфейс IRequestHandler, на който се задават като параметри типът record за командата или заявката, както и типът на обекта за трансфер на данни за резултата от обработващата логика, което е представено в следващия фрагмент от код:

```
      public class CreateProductHandler : IRequestHandler<CreateProductCommand,
ProductResponse>
        {
            private readonly IProductRepository _productRepository;
            private readonly ILogger<CreateProductHandler> _logger;
            private readonly IMapper _mapper;

            public CreateProductHandler(IProductRepository productRepository,
ILogger<CreateProductHandler> logger, IMapper mapper)
            {
                _productRepository = productRepository;
                _logger = logger;
                _mapper = mapper;
            }

            public async Task<ProductResponse> Handle(CreateProductCommand
createProductCommand, CancellationToken cancellationToken)
            {
                var productToCreate = _mapper.Map<Product>(createProductComma
nd.CreateProductRequest);
                _productRepository.Add(productToCreate);
                await _productRepository.SaveChangesAsync(cancellationToken);

                _logger.LogInformation(„Created a new product");

                return _mapper.Map<ProductResponse>(productToCreate);
            }
        }
```

Освен зависимостите от параметри за заявка и отговор на генеричния интерфейс, се инжектира и зависимост за хранилище, реализирано чрез структурния шаблон за софтуерен дизайн Repository, което служи като посредник между домейн логиката и базата данни и предоставя абстракция за управление на данните, улесняваща тяхното съхранение, извличане и модифициране, без да се разкрива сложността на самата база данни. В допълнение се инжектират услуга за отчитане на събития и услуга за преобразуване на данни между различни обекти. В метода Handle данните от командата се трансформират в нов обект, който се добавя към хранилището, и промените се съхраняват в базата данни асинхронно. След успешното създаване на новия обект се записва ново събитие, а методът връща окончателния отговор от манипулацията на командата.

Логиката за създаване и манипулация на заявките следва същата аналогия както при командите, като се насърчава кеширането на заявките в паметта спрямо уникален идентификатор. От страна на контролерите методът Send от интерфейса IMediator, идващ от библиотеката, се извиква, за да се предаде логика към конкретен манипулатор, на който ще се прикачи обект за команда или заявка като аргумент:

```
[HttpGet(„{id:guid}“, Name = „ProductById“)]
public async Task<ActionResult<ProductResponse>> GetById(Guid id)
{
    var getProductByIdQuery = new GetProductByIdQuery(id);
    var productById = await _mediator.Send(getProductByIdQuery);

    return productById != null ? Ok(productById) : NotFound();
}
```

В посочения пример инжектираният посредник задейства манипулатор за обработка на заявка, след което се връща отговор с търсения обект, ако той съществува.

Известията в MediatR за разлика от разгледаните команди и заявки предоставят механизъм за изпращане на информация или събития към множество обработчици без очакване на отговор. Те са от съществено значение за уведомяване на различни компоненти в приложението за предстоящи събития или за активиране на конкретни действия в зависимост от настъпване на събитие. Известията са полезни в микросървисната архитектура, тъй като улесняват разпределената комуникация и преодоляването на ограничения, отнасящи се до части от програмата, които разчитат или трябва да повлияят на много други части на системата, като същевременно поддържат и асинхронна обработка. Стремейки се към слабо свързване между компонентите и позволявайки излъчването на събития, известията подобряват гъвкавостта и скалируемостта на приложенията [14].

Същото важи както за модулността така и за придържането към SOLID принципите в структурата на кода, които водят до опростяване на сложните взаимодействия между различните части на приложенията, интегрирането на нови функции и поддръжката на съществуващите. Те се постигат чрез имлементацията на отделните характеристики на библиотеката и инжектиране на междусекторни проблеми като регистриране, валидиране и оторизация в конвейера за заявки или отговори, чиято логика е повторно използваема. В тази връзка при проектирането на приложно-програмни интерфейси с помощта на MediatR се предлага конструирането на няколко основни слоеве с цел осигуряване на структурирана и ефективна архитектура:

- Създаване на домейн модели: дефинират се надеждни домейн модели, които капсулират основните обекти и бизнес логиката на приложението.
- Създаване на слой за връзка с базата данни: включва имплементацията на шаблона Repository за комуникация с базата данни, конфигурация на релационни връзки и съпоставяния между обекти. Конкретните имплементации на шаблона Repository се интегрират с MediatR за обработка на команди, заявки, събития и известия.

- Създаване на слой за идентичността на потребителите: разработва се слой за управление на идентичността, който да управлява автентикацията на потребителите, оторизацията и контрола на достъпа.
- Създаване на инфраструктурен слой за конфигурация на услугите и помощни разширителни методи.
- Създаване на слой за работа с отделните характеристики на MediatR – заявки, команди, събития, известия и валидатори.
- Конфигурация на приложно-програмния интерфейс: задават се крайните точки за насочване на заявки или команди към подходящите манипулатори на MediatR
- Разработване и свързване с клиентското приложение: осигурява се взаимодействие между приложно-програмния интерфейс и клиентското приложение.
- Компонентно тестване: изпълняват се компонентни тестове за потвърждение на функционалността на отделните компоненти, включително манипулатори и конкретни имплементации на шаблона Repository.
- Интеграционно тестване: изпълняват се интеграционни тестове с цел проверка на оперативната съвместимост на цялата система, включително крайни точки на приложно-програмния интерфейс, взаимодействието с базата данни и обработката на съобщения за логически операции от MediatR.

ЗАКЛЮЧЕНИЕ

Изграждането на мащабируеми и лесно поддържани приложения изисква солидна архитектурна основа. Чистата архитектура излага принципи, които целят да отделят основната логика на приложението от неговия механизъм за доставка, като по този начин го правят по-лесно за управление и развитие. Един от инструментите, които могат да помогнат за осъществяване на това е именно библиотеката MediatR. Интеграцията на MediatR в .NET приложенията улеснява имплементацията на шаблоните за софтуерен дизайн Посредник (Mediator) и Разделяне на отговорността на команди и заявки (CQRS), предоставяйки ефективен начин за внедряване на чиста архитектура чрез насърчаване на слабото свързване, придържането към SOLID принципите и ясното разделение между бизнес логиката и входните точки на приложението. Работата с тези шаблони допринася за създаването на добре организирани, модулни и мащабируеми системи. MediatR намалява зависимостите между компонентите чрез централизиране на обработката на команди, заявки и известия, въвеждайки поведение на конвейера за успешно справяне с междусекторни проблеми. В резултат кодовата база става по-чиста и по-структурирана, което улеснява поддръжката и разширяването на приложенията. Чрез разделяне на операциите за четене и запис, CQRS подсилва производителността и скалируемостта на системите, позволявайки поотделна оптимизация на специфичните операции. В заключение, комбинацията на шаблоните Посредник (Mediator) и CQRS в библиотеката MediatR представлява мощен инструмент за създаване на модерни и ефективни уеб приложения. Библиотеката подпомага изграждането на системи с добре организирана архитектура, която е лесна за поддръжка, мащабиране и разширяване, осигурявайки оптимална производителност и гъвкавост за съвременния софтуер.

ЛИТЕРАТУРА

[1.] **Alina Bo. (2023).** MediatR. (Юли, 2023). Извлечено на 10.11.2024 г. от https://alinabo.com/mediatr

[2.] **Arthur C. Codex. (2024).** Using MediatR for Clean Architecture in ASP.NET Core APIs. (Април, 2024). Извлечено на 10.11.2024 от https://reintech.io/blog/using-mediatr-clean-architecture-aspnet-core

[3.] **Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1995).** Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley Professional Computing Series. Addison Wesley.

[4.] **Dmytro Hruzin, Oleksandr Lytvynov. (2024).** On The Migration of Domain Driven Design to CQRS with Event Sourcing Software Architecture. Information Technology Computer Science Software Engineering and Cyber Security. DOI: 10.32782/IT/2024-1-7. (Юни, 2024), 50–60.

[5.] **Sardar Mudassar Ali Khan. (2023).** CQRS Architectural Design Pattern Used In Software Development. https://dev.to/sardarmudassaralikhan/cqrs-architectural-design-pattern-56nm

[6.] **Edson Martinez. (2024).** Implementing the CQRS and Mediator pattern in a .NET 8 Web API. (Май, 2024). Извлечено на 10.11.2024 г. от https://medium.com/@EdsonMZ/implementing-the-cqrs-and-mediator-pattern-in-a-net-8-web-api-8c0319a4525c

[7.] **Mukesh Murugan. (2024).** CQRS and MediatR in ASP.NET Core - Building Scalable Systems. (Май, 2024). Извлечено на 10.11.2024 от https://codewithmukesh.com/blog/cqrs-and-mediatr-in-aspnet-core/

[8.] **Ramin Sharifi. (2024).** CQRS Pattern (.net core). (Януари, 2024). Извлечено на 10.11.2024 г. от https://www.linkedin.com/pulse/cqrs-pattern-net-core-ramin-sharifi-m6c7f

[9.] **Inderjit Singh. (2023).** Implement CQRS Design Pattern with MediatR in ASP.NET Core 6 (C#) (Септември, 2023). Извлечено на 10.11.2024 г. от https://medium.com/@inderjit.fullstack.dev/implement-cqrs-design-pattern-with-mediatr-in-asp-net-core-6-c-dc192811694e

[10.] **Marinko Spasojevic. (2024).** CQRS and MediatR in ASP.NET Core. (Април, 2024). Извлечено на 10.11.2024 г. от https://code-maze.com/cqrs-mediatr-in-aspnet-core/

[11.] **Jack Poorte. (2024).** C# Mediator Design Pattern (Март, 2024). Извлечено на 10.11.2024 г. от https://www.dofactory.com/net/mediator-design-pattern 4.3.05. Mediator. (Март, 2023). Извлечено на 10.11.2024 г. от https://wikidocs.net/186227

Mediator. Извлечено на 10.11.2024 г. от https://refactoring.guru/design-patterns/mediator

Simplifying complexity with MediatR and Minimal APIs. (Юли, 2023). Извлечено на 10.11.2024 от https://q.agency/blog/simplifying-complexity-with-mediatr-and-minimal-apis/

What is CQRS? Извлечено на 10.11.2024 от https://www.confluent.io/learn/cqrs/

[12.] **Пашковський, Б., Слабінога, М., Романів, М. (2024).** Оптимізація роботи веб-застосунків засобами горизонтального масштабування з використанням архітектури CQRS. Вісник Херсонського Національного Технічного Університету. 272–278. DOI: 10.35546/Kntu2078-4481.2024.1.38. // Pashkovsykiy, B., Slabinoga, M., Romaniv, M. (2024). Optimizatsiya roboti veb-zastosunkiv zasobami gorizontalynogo masshtabuvannya z vikoristannyam arhitekturi CQRS. Visnik Hersonsykogo Natsionalynogo Tehnichnogo Universitetu

ИНФОРМАЦИЯ ЗА АВТОРИТЕ

Пламенна Петрова – студент в магистърската програма Уеб технологии и разработване на софтуер, Факултет „Математика и информатика“, Великотърновски университет „Св. св. Кирил и Методий“; Програмист на софтуерни приложения в Тремол ООД, e-mail: plamennavp@abv.bg

Златко Върбанов – Доц. д-р във Факултет „Математика и информатика“, Катедра „Информационни технологии“, Великотърновски университет „Св. св. Кирил и Методий“, България, e-mail: zl.varbanov@ts.uni-vt.bg

ABOUT THE AUTHORS

Plamenna Petrova – student in the Master's Degree Program in Web Technologies and Software Development, Faculty of Mathematics and Informatics, "St. Cyril and St. Methodius" University of Veliko Tarnovo; Software Developer at Tremol Ltd., email: plamennavp@abv.bg

Zlatko Varbanov – Associate Professor, PhD, Faculty of Mathematics and Informatics, "St. Cyril and St. Methodius" University of Veliko Tarnovo, e-mail: zl.varbanov@ts.uni-vt.bg