ASP.NET Core CQRS with MediatR

목차

[ASP.NET Core CQRS with MediatR 1](#_Toc58243124)

[**CQRS and the Mediator Pattern** 4](#_Toc58243125)

[CQRS 4](#_Toc58243126)

[What problem is this trying to solve? 5](#_Toc58243127)

[Trade-offs 5](#_Toc58243128)

[Mediator Pattern 6](#_Toc58243129)

[How MediatR facilitates CQRS and Mediator Patterns 7](#_Toc58243130)

[Setting up an ASP.NET Core API with MediatR 7](#_Toc58243131)

[Project Setup 7](#_Toc58243132)

[Dependencies 7](#_Toc58243133)

[Startup 7](#_Toc58243134)

[Controller 8](#_Toc58243135)

[Data Store 9](#_Toc58243136)

[Separating the Commands and Queries 9](#_Toc58243137)

[Requests with MediatR 10](#_Toc58243138)

[GetValuesQuery 10](#_Toc58243139)

[Calling and Testing our Request 11](#_Toc58243140)

[MediatR Commands 13](#_Toc58243141)

[Calling and Testing our Request 14](#_Toc58243142)

[MediatR Notifications 16](#_Toc58243143)

[Updating our FakeDataStore 16](#_Toc58243144)

[Creating the Notification and Handlers 17](#_Toc58243145)

[Triggering the Notification 18](#_Toc58243146)

[Testing our Notifications 19](#_Toc58243147)

[Building MediatR Behaviors 21](#_Toc58243148)

[Creating our Behavior 21](#_Toc58243149)

[Registering our Behavior 22](#_Toc58243150)

[Testing our Behavior 22](#_Toc58243151)

[Conclusion 24](#_Toc58243152)

이 자료에서는 CQRS 패턴에 대한 간략한 소개와 .NET 라이브러리 MediatR이 아키텍처로 소프트웨어를 구축하는 데 어떻게 도움이 되는지 설명합니다.

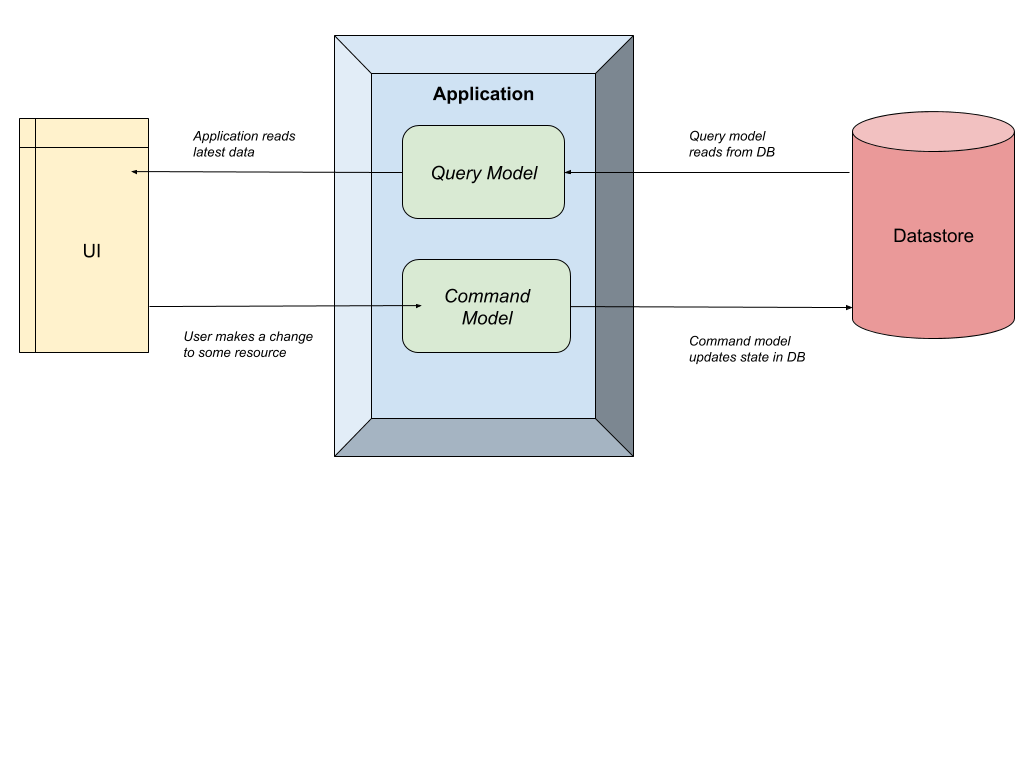
**CQRS and the Mediator Pattern**

MediatR 라이브러리는 CQRS 및 Mediator 패턴의 두 가지 기본 소프트웨어 아키텍처 패턴을 용이하게하기 위해 구축되었습니다. 비슷하지만, 각 패턴의 원리를 이해하는데 잠시 시간을 할애하겠습니다.

### CQRS

CQRS는 "Command Query Responsibility Segregation"을 의미합니다. 두문자에서 알 수 있듯이 명령(저장) 및 쿼리(읽기)의 책임을 다른 모델로 분리하는 것입니다.

일반적으로 사용되는 CRUD 패턴 (Create-Read-Update-Delete)에 대해 생각하면 일반적으로 네 가지 작업을 모두 담당하는 데이터 저장소와 상호 작용하는 사용자 인터페이스가 있습니다. 대신 CQRS는 이러한 작업을 쿼리 용 (일명 "R")과 명령 용 (일명 "CUD")의 두 모델로 분할하도록 했습니다.

다음 이미지는 이것이 어떻게 작동하는지 보여줍니다.

보시다시피 응용 프로그램은 단순히 쿼리 및 명령 모델을 분리합니다. CQRS 패턴은 이러한 분리가 어떻게 발생하는지에 대한 공식적인 요구 사항을 만들지 않습니다. 이는 동일한 애플리케이션의 별도 클래스처럼 간단 할 수 있으며 (마지막 MediatR에서 살펴 보겠지만) 다른 서버에 있는 별도의 물리적 애플리케이션까지 가능합니다. 이 결정은 확장 요구 사항 및 인프라와 같은 요소를 기반으로 하므로 오늘은 그 결정 경로를 다루지 않겠습니다.

요점은 CQRS 시스템을 만들려면 쓰기에서 읽기를 분리하면 된다는 것입니다.

### What problem is this trying to solve?

글쎄요, 일반적인 이유는 종종 시스템을 설계 할 때 데이터 저장소에서 시작하기 때문입니다. 데이터베이스 정규화를 수행하고 기본 및 외래 키를 추가하여 참조 무결성을 적용하고 인덱스를 추가하며 일반적으로 "쓰기 시스템"이 최적화 되었는지 확인합니다. 이것은 SQL Server 또는 MySQL과 같은 관계형 데이터베이스에 대한 일반적인 설정입니다. 다른 경우에는 먼저 읽기 사용 사례에 대해 생각한 다음 이를 데이터베이스에 추가하여 중복 또는 기타 관계형 DB 문제에 대해 걱정하지 않습니다 (종종 "문서 데이터베이스"가 이러한 패턴에 사용됨).

어느 쪽도 잘못된 것이 아닙니다. 그러나 문제는 이것이 읽기와 쓰기 사이에 지속적으로 균형을 맞추고 있으며 결국 한쪽이 "승리"한다는 것입니다. 모든 추가 개발은 양쪽 모두를 분석해야 하고 종종 한쪽이 손상된다는 것을 의미합니다.

CQRS를 사용하면 이러한 고려 사항에서 "파괴"하고 다른 시스템의 영향에 대해 걱정하지 않고 각 시스템에 동일한 디자인과 고려 사항을 부여 할 수 있습니다. 이는 특히 이러한 시스템에서 작업하는 별도의 팀이 있는 경우 성능과 민첩성 모두에서 엄청난 이점을 제공합니다.

### Trade-offs

CQRS는 원칙적으로 훌륭하게 들리지만 소프트웨어의 모든 것과 마찬가지로 항상 절충안이 있습니다.

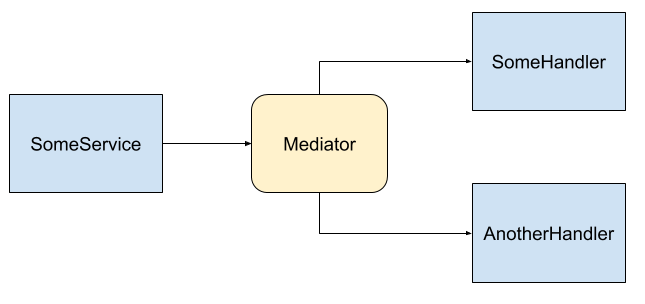
이 중 몇 가지는 다음과 같습니다.

* 별도의 시스템 관리 (애플리케이션 계층이 분할 된 경우)
* 오래된 데이터 (데이터베이스 계층이 분할 된 경우)
* 여러 구성 요소 관리의 복잡성

궁극적으로 특정 사용 사례에 달려 있습니다. 좋은 개발 관행은 우리가 "간단하게 유지"(KISS)하도록 장려하므로 필요한 경우에만 이러한 패턴을 사용합니다. 그렇지 않으면 단순히 조기 최적화 일뿐입니다.

다음 섹션에서는 중재자라는 유사한 패턴에 대해 논의하겠습니다.

### Mediator Pattern

중재자 패턴은 단순히 개체가 서로 상호 작용하는 방식을 캡슐화하는 개체를 정의하는 것입니다. 둘 이상의 객체가 서로에 대한 직접적인 종속성을 갖는 대신, 대신 이러한 상호 작용을 다른 당사자에게 보내는 책임이 있는 "중개자"와 상호 작용합니다. 

위의 이미지에서 SomeService가 중개자에게 메시지를 보내고 중개자는 메시지를 처리하기 위해 여러 서비스를 호출하는 것을 볼 수 있습니다. 파란색 구성 요소 간에 직접적인 종속성이 없습니다.

Mediator 패턴이 유용한 이유는 Inversion of Control과 같은 패턴이 유용하기 때문입니다. 종속성 그래프가 최소화되어 코드가 더 간단하고 테스트하기 쉽기 때문에 "느슨한 결합"이 가능합니다. 즉, 구성 요소에 대한 고려 사항이 적을수록 개발 및 발전이 더 쉬워집니다.

이전 이미지에서 서비스에 직접적인 종속성이 없는지, 메시지 제작자는 이를 처리 할 사람 또는 수를 알지 못합니다. 이것은 메시지 브로커가 "게시 / 구독"패턴에서 작동하는 방식과 매우 유사합니다. 다른 핸들러를 추가하고 싶다면 제작자를 수정할 필요가 없습니다.

이제 몇 가지 이론을 살펴 봤으니 MediatR이 모든 것을 가능하게하는 방법에 대해 이야기 해 보겠습니다.

## How MediatR facilitates CQRS and Mediator Patterns

MediatR을 CQRS 시스템 구축에 도움이 되는 "진행중인"중재자 구현으로 생각할 수 있습니다. 사용자 인터페이스와 데이터 저장소 간의 모든 통신은 MediatR을 통해 이루어집니다.

여기서 "진행 중"이라는 용어는 중요한 제한 사항입니다. 동일한 프로세스의 클래스 내에서 상호 작용을 관리하는 .NET 라이브러리이기 때문에 두 시스템에서 명령과 쿼리를 분리하려는 경우 사용하기에 적합한 라이브러리가 아닙니다. 이러한 상황에서 더 나은 접근 방식은 Kafka 또는 Azure Service Bus와 같은 메시지 브로커를 사용하는 것입니다.

그러나 이 기사의 목적을 위해 우리는 단순한 단일 프로세스 CQRS 시스템을 고수 할 것이므로 MediatR은 완벽하게 적합합니다.

## Setting up an ASP.NET Core API with MediatR

### Project Setup

먼저 Visual Studio를 열고 프로젝트 유형으로 API를 선택하여 새 ASP.NET Core 웹 애플리케이션을 만들어 보겠습니다.

### Dependencies

패키지 관리자 콘솔을 통해 몇 가지 패키지를 설치하겠습니다.

첫째, MediatR 패키지:

PM> install-package MediatR

다음으로 MediatR을 ASP.NET DI 컨테이너와 연결하는 패키지:

PM> install-package MediatR.Extensions.Microsoft.DependencyInjection

### Startup

Startup.cs를 열고 using 문을 추가해 보겠습니다.

PM> using MediatR

다음으로 ConfigureServices를 수정 해 보겠습니다.

PM> services.AddMediatR(typeof(startup));

이제 MediatR이 구성되고 사용할 준비가 되었습니다.

### Controller

이제 모든 것이 설치 되었으므로 MediatR에 메시지를 보낼 새 컨트롤러를 설정하겠습니다.

Controllers 폴더에서 "읽기/쓰기 작업이 있는 API 컨트롤러"를 추가하여 클래스 이름을 기본값 ValuesController.cs로 유지합니다.

그런 다음 다음과 같은 클래스로 끝납니다.

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

**[**ApiController**]**

public class ValuesController : ControllerBase

**{**

// GET: api/<ValuesController>

**[**HttpGet**]**

public IEnumerable**<**string**>** Get**()**

**{**

return new string**[]** **{** "value1", "value2" **}**;

**}**

// GET api/<ValuesController>/5

**[**HttpGet**(**"{id}"**)]**

public string Get**(**int id**)**

**{**

return "value";

**}**

// POST api/<ValuesController>

**[**HttpPost**]**

public void Post**([**FromBody**]** string value**)**

**{**

**}**

// PUT api/<ValuesController>/5

**[**HttpPut**(**"{id}"**)]**

public void Put**(**int id, **[**FromBody**]** string value**)**

**{**

**}**

// DELETE api/<ValuesController>/5

**[**HttpDelete**(**"{id}"**)]**

public void Delete**(**int id**)**

**{**

**}**

**}**

그런 다음 IMediatR 인스턴스를 초기화하는 생성자를 추가해 보겠습니다.

private readonly IMediator \_mediator;

public ValuesController**(**IMediator mediator**)**

**{**

\_mediator = mediator;

**}**

IMediatR 인터페이스를 사용하면 MediatR에 메시지를 보내고 관련 핸들러로 전달할 수 있습니다. 이미 종속성 주입 패키지를 설치했기 때문에 인스턴스가 자동으로 해결됩니다.

### Data Store

일반적으로 실제 데이터베이스와 상호 작용하려고 합니다. 그러나이 기사의 목적을 위해 이 책임을 캡슐화하고 단순히 일부 문자열 값과 상호 작용하는 가짜 클래스를 만들어 보겠습니다.

새 FakeDataStore 클래스를 추가하고 수정 해 보겠습니다.

public class FakeDataStore

**{**

private static List**<**string**>** \_values;

public FakeDataStore**()**

**{**

\_values = new List**<**string**>**

**{**

"a",

"b",

"c"

**}**;

**}**

public void AddValue**(**string value**)**

**{**

\_values.Add**(**value**)**;

**}**

public IEnumerable**<**string**>** GetAllValues**()**

**{**

return \_values;

**}**

**}**

여기서 우리는 단순히 우리의 목적에 충분한 정적 문자열 목록과 상호 작용하고 있습니다.

Startup.cs의 ConfigureServices를 업데이트하여 DataStore를 싱글 톤으로 구성 해 보겠습니다.

services.AddSingleton<FakeDataStore>();

이제 데이터 저장소가 구현 되었으므로 CQRS 용 앱을 설정하겠습니다.

### Separating the Commands and Queries

이 기사는 결국 CQRS에 관한 것이므로 이를 위해 "Commands"와 "Queries"라는 두 개의 새 폴더를 만들어 보겠습니다.

연습 내내이 폴더를 사용하여 모델을 분리합니다. 앞서 언급했듯이 우리는 "in-process"CQRS를 수행하고 있으므로 이를 구성하는 간단한 방법입니다.

다음 섹션에서는 MediatR의 가장 일반적인 사용 인 "요청"에 대해 설명합니다.

## Requests with MediatR

MediatR 요청은 단일 요청이 단일 처리기에 의해 동기 방식으로 처리되는 매우 간단한 요청-응답 스타일 메시지입니다 (C # 내부 비동기 / 대기가 아닌 요청 관점에서 동기). 여기서 좋은 사용 사례는 데이터베이스에서 무언가를 반환하거나 데이터베이스를 업데이트하는 것입니다.

MediatR에는 두 가지 유형의 요청이 있습니다. 하나는 값을 반환하고 하나는 반환하지 않습니다. 종종 이것은 읽기 / 쿼리 (값 반환) 및 쓰기 / 명령 (일반적으로 값을 반환하지 않음)에 해당합니다.

앞서 만든 FakeDataStore를 사용하여 일부 MediatR 요청을 구현할 것입니다.

먼저 FakeDataStore에서 모든 값을 반환하는 요청을 생성 해 보겠습니다.

### GetValuesQuery

이것은 쿼리이므로 GetValuesQuery라는 클래스를 "Queries"폴더에 추가하고 구현해 보겠습니다.

public class GetValuesQuery

**{**

public class Query : IRequest**<**IEnumerable**<**string**>>** **{** **}**

public class Handler : RequestHandler**<**Query, IEnumerable**<**string**>>**

**{**

private readonly FakeDataStore \_db;

public Handler**(**FakeDataStore db**)**

**{**

\_db = db;

**}**

protected override IEnumerable**<**string**>** Handle**(**Query request**)**

**{**

return \_db.GetAllValues**()**;

**}**

**}**

**}**

여기에 약간의 작업이 있으므로 조금씩 분해 해 보겠습니다.

* 먼저 IRequest<IEnumerabl<string>>을 구현하는 Query라는 내부 클래스를 만듭니다. 이것은 단순히 요청이 문자열 목록을 반환한다는 것을 의미합니다.
* 다음으로 RequestHandler<Query, IEnumerable<string>>에서 상속되는 Handler라는 또 다른 내부 클래스를 만듭니다. 즉, 이 클래스는 쿼리를 처리하며 이 경우 문자열 목록을 반환합니다.
* Handler 클래스에서 FakeDataStore에서 값을 반환하는 Handle이라는 단일 메서드를 구현합니다.

몇 가지 추가 참고 사항을 지적 할 가치가 있습니다.

* 내부 클래스와 동일한 클래스에 쿼리와 핸들러를 모두 배치합니다. 그러나, 우리가 원한다면 이것들을 별도의 클래스 나 프로젝트에 확실히 넣을 수 있습니다. 그러나 동일한 클래스에 넣으면 코드베이스를 탐색하는 대신 간단하고 쉽게 발견 할 수 있습니다.
* 여기서도 단순성을 위해 그리고 메모리 내 값 목록을 구현하는 방식으로 인해 동기화 상태를 유지합니다. 그러나 MediatR은 메서드 서명을 변경하여 async/await를 완전히 지원합니다.

### Calling and Testing our Request

요청을 호출하려면 ValuesController에서 Get() 메서드를 수정하기만하면 됩니다.

**[**HttpGet**]**

public async Task**<**IEnumerable**<**string**>>** Get**()**

**{**

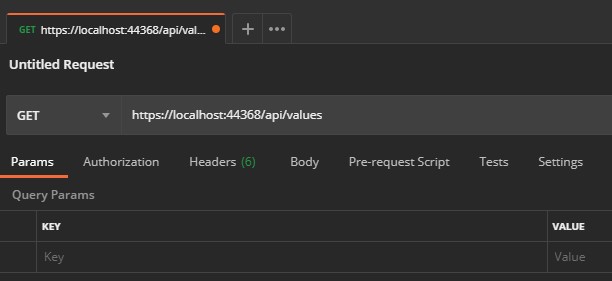
return await \_mediator.Send**(**new Queries.GetValuesQuery.Query**())**;

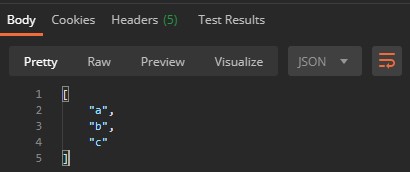
**}**

MediatR에 요청을 보내는 것이 얼마나 간단합니다. FakeDataStore에 대한 종속성을 취하지 않거나 쿼리 처리 방법에 대한 아이디어가 없습니다. 이것은 Mediator 패턴의 원칙 중 하나이며 MediatR을 사용하여 직접 구현 된 것을 볼 수 있습니다.

이제 모든 것이 예상대로 작동하는지 확인하겠습니다.

먼저 CTRL + F5를 눌러 앱을 빌드하고 실행 해 보겠습니다. 브라우저에 "/weatherforecast"응답이 표시되어야 합니다.

그런 다음 Postman을 실행하고 새 요청을 생성하겠습니다. 

그런 다음 "보내기"를 누르면 응답이 표시됩니다. 

환상적입니다! 이것은 우리가 보는 값이 FakeDataStore에 의해 초기화 된 값이기 때문에 MediatR이 올바르게 작동하고 있음을 증명합니다. 방금 CQRS에서 첫 번째 "쿼리"를 구현했습니다 🙂

다음 섹션에서는 값을 반환하지 않는 다른 유형의 MediatR 요청 (예: "명령")에 대해 설명하겠습니다.

## MediatR Commands

첫 번째 "명령"을 만들기 위해 단일 값을 사용하고 FakeDataStore를 업데이트하는 요청을 추가해 보겠습니다.

"Commands"폴더 안에 AddValueCommand라는 클래스를 추가해 보겠습니다.

public class AddValueCommand

**{**

public class Command : IRequest

**{**

public string Value **{** get; set; **}**

**}**

public class Handler : RequestHandler**<**Command**>**

**{**

private readonly FakeDataStore \_db;

public Handler**(**FakeDataStore db**)**

**{**

\_db = db;

**}**

protected override void Handle**(**Command request**)**

**{**

\_db.AddValue**(**request.Value**)**;

**}**

**}**

설명해 드리겠습니다.

* 먼저 IRequest를 구현하는 Command 클래스를 만듭니다. 클래스에는 Value에 대한 단일 속성이 있으며 이는 추가하는 값을 나타냅니다. 이번에는 IRequest 서명에 유형 매개 변수가 없습니다. 이는 우리가 값을 반환하지 않기 때문입니다.
* 다음으로 RequestHandler<Command> 클래스에서 상속되는 Handler 클래스를 만듭니다. 이것은 이전 GetValuesQuery와 유사한 구현입니다. 우리는 이 클래스가 Command 요청을 처리 할 것이라고 말하고 있습니다.
* 그런 다음 FakeDataStore에 값을 추가하여 Handle (Command request) 메서드를 구현합니다.

### Calling and Testing our Request

이제 ValuesController에서 Post 메서드를 수정하여 Command를 호출 해 보겠습니다.

**[**HttpPost**]**

public async Task Post**([**FromBody**]** string value**)**

**{**

await \_mediator.Send**(**new Commands.AddValueCommand.Command

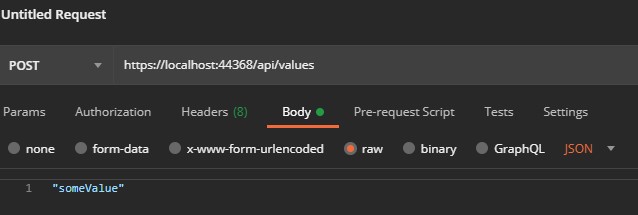
**{**

Value = value

**})**;

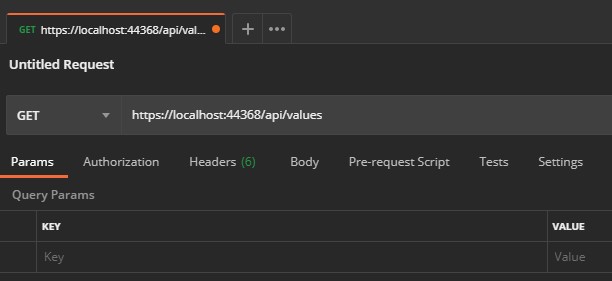
**}**

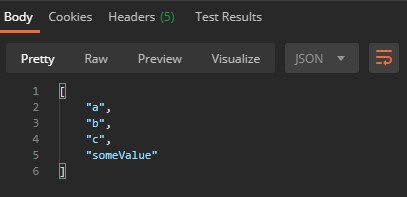
다시 Get 메서드와 매우 유사합니다. 하지만 이번에는 Command에 값을 설정하고 값을 반환하지 않습니다.

명령을 테스트하기 위해 앱을 다시 실행하고 Postman에 새 요청을 추가하겠습니다. 

그런 다음 "보내기"를 누르면 200 OK 상태 응답이 표시되지만 명령에서 반환 된 내용이 없음을 알 수 있습니다.

실제로 작동하는지 테스트하기 위해 GetValues 요청을 다시 실행 해 보겠습니다.



"보내기"를 누르면 새로 추가 된 가치가 표시됩니다. 

이것은 새로운 값으로 MediatR에 메시지를 보내고 상태를 업데이트함으로써 명령이 올바르게 작동하고 있음을 증명합니다. 그러면 쿼리 모델 (예: GetValues)이 새 값으로 업데이트 된 것을 볼 수 있습니다.

이론적으로는 간단 해 보일 수 있지만 단순히 메모리 내 문자열 목록을 업데이트한다는 사실을 넘어서 생각해 보겠습니다. 우리가 하는 일은 구현 방법에 대한 아이디어 없이 간단한 메시지 구성을 통해 데이터 저장소와 통신하는 것입니다. 명령과 쿼리는 다른 데이터 저장소를 가리킬 수 있습니다. 그들은 자신의 요청이 어떻게 처리 될지 모르고 신경도 쓰지 않습니다.

## MediatR Notifications

따라서 단일 핸들러에 의해 처리되는 단일 요청 만 보았습니다. 그러나 여러 핸들러의 단일 요청을 처리하려면 어떻게 해야합니까?

이것이 알림이 들어오는 곳입니다. 이러한 상황에서 우리는 일반적으로 일부 이벤트 후에 발생해야하는 여러 독립적 인 작업을 수행합니다.

예는 다음과 같습니다.

* 이메일 보내기
* 캐시 무효화

이를 설명하기 위해 이전에 만든 AddValueCommand 흐름을 업데이트하여 알림을 게시하고 두 개의 핸들러에서 처리하도록 합니다.

이메일을 보내고 캐시를 무효화 하는 것은 이 문서의 범위를 벗어나지만 알림 동작을 보여주기 위해 단순히 가짜 값 목록을 업데이트하여 무언가가 처리되었음을 나타냅니다.

### Updating our FakeDataStore

FakeDataStore를 열고 새 메소드를 추가해 보겠습니다.

public void EventOccured**(**string value, string evt**)**

**{**

var indexOfValue = \_values.FindIndex**(**val =**>** val.StartsWith**(**value**))**;

\_values**[**indexOfValue**]** = $"{\_values[indexOfValue]}, event: {evt}";

**}**

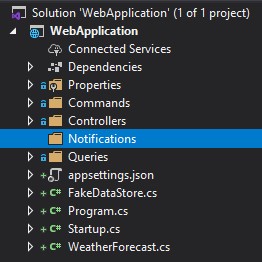
아주 간단하게, 우리는 특정 값을 찾고 그것에 발생한 이벤트를 나타내기 위해 업데이트합니다.

이제 스토어를 수정 했으므로 다음 섹션에서 알림 및 핸들러를 생성 해 보겠습니다.

### Creating the Notification and Handlers

정의하려는 이벤트를 캡슐화하는 알림 메시지를 정의 해 보겠습니다.

먼저 알림이라는 새 폴더를 추가하겠습니다.



이 폴더 안에 ValueAddedNotification이라는 클래스를 추가하겠습니다.

public class ValueAddedNotification : INotification

**{**

public string Value **{** get; set; **}**

**}**

public class EmailHandler : INotificationHandler**<**ValueAddedNotification**>**

**{**

private readonly FakeDataStore \_fakeDataStore;

public EmailHandler**(**FakeDataStore fakeDataStore**)**

**{**

\_fakeDataStore = fakeDataStore;

**}**

public Task Handle**(**ValueAddedNotification notification, CancellationToken cancellationToken**)**

**{**

\_fakeDataStore.EventOccured**(**notification.Value, "Email sent"**)**;

return Task.CompletedTask;

**}**

**}**

public class CacheInvalidationHandler : INotificationHandler**<**ValueAddedNotification**>**

**{**

private readonly FakeDataStore \_fakeDataStore;

public CacheInvalidationHandler**(**FakeDataStore fakeDataStore**)**

**{**

\_fakeDataStore = fakeDataStore;

**}**

public Task Handle**(**ValueAddedNotification notification, CancellationToken cancellationToken**)**

**{**

\_fakeDataStore.EventOccured**(**notification.Value, "Cache invalidated"**)**;

return Task.CompletedTask;

**}**

**}**

자세히 살펴 보겠습니다.

1. 먼저 Value라는 단일 속성과 함께 INotification을 구현하는 ValueAddedNotification이라는 클래스를 만듭니다. 이것은 앞서 본 IRequest와 동일하지만 알림 용입니다.
2. 다음으로 기본적으로 동일한 작업을 수행하는 EmailHandler 및 CacheInvalidationHandler라는 두 개의 처리기를 만듭니다.
   1. INotificationHandler<ValueAddedNotification>을 구현하여 해당 이벤트를 처리 할 수 있음을 나타냅니다.
   2. FakeDataStore에서 EventOccured 메서드를 호출하여 발생한 이벤트를 지정합니다.

실제 사용 사례에서 이들은 다르게 구현되어 외부 종속성을 취하고 의미 있는 작업을 수행 할 수 있지만 여기서는 알림의 동작을 보여 주려고 합니다.

### Triggering the Notification

다음으로 실제로 알림을 트리거해야합니다.

ValuesController를 열고 Post 메서드를 수정 해 보겠습니다.

**[**HttpPost**]**

public async Task Post**([**FromBody**]** string value**)**

**{**

await \_mediator.Send**(**new Commands.AddValueCommand.Command

**{**

Value = value

**})**;

await \_mediator.Publish**(**new Notifications.ValueAddedNotification **{** Value = value **})**;

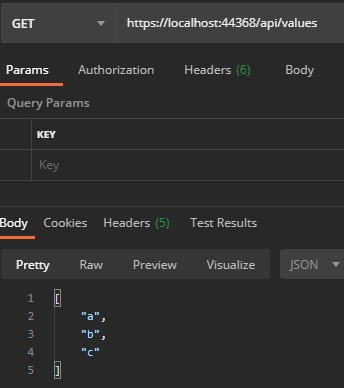
**}**

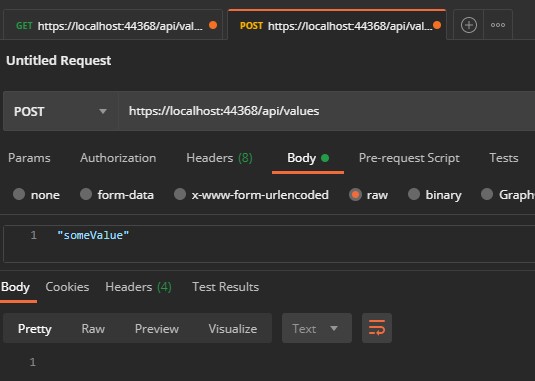
MediatR에 AddValueCommand 요청을 보내는 것 외에도, 이번에는 Publish 메서드를 사용하여 ValueAddedNotification을 MediatR에 보냅니다.

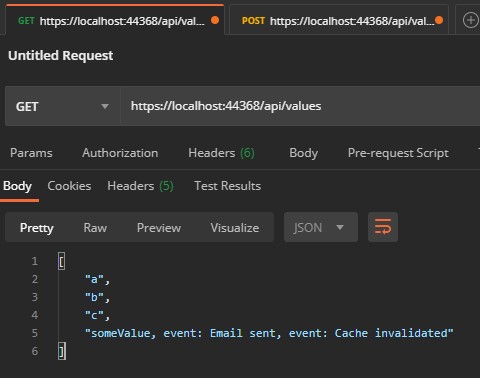
원하는 경우 AddValueCommand의 처리기에서 직접이 작업을 수행 할 수 있지만 간단하게 여기에 배치하겠습니다.

### Testing our Notifications

작동하는지 테스트하기 위해 애플리케이션을 실행하고 GetValues에 대한 요청을 다시 실행 해 보겠습니다.

예상대로 FakeDataStore 생성자에서 초기화하는 세 가지 값이 있습니다.

이제 다른 Postman 요청을 실행하여 새 값을 추가해 보겠습니다.

요청이 성공한 것을 봅니다. 이제 Get Values 요청을 다시 실행 해 보겠습니다.

예상대로 "someValue"를 추가하면 두 이벤트가 모두 발생하고 값을 편집했습니다. 인위적인 예이지만 여기서 중요한 점은 제작자가 다른 사실을 알지 못해도 이벤트를 시작하고 여러 번 처리 할 수 있다는 것입니다.

추가 작업을 수행하기 위해 워크플로를 확장하려면 새 처리기를 추가하기만 하면 됩니다. 알림 자체 또는 해당 알림의 게시를 수정할 필요가 없습니다. 이는 이전의 확장 성 및 우려 사항 분리 지점을 다시 다룹니다.

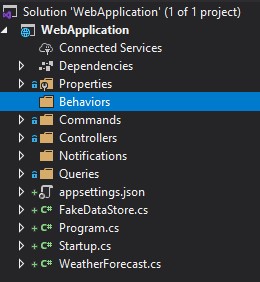
## Building MediatR Behaviors

종종 우리가 응용 프로그램을 구축 할 때 우리는 여러 가지 교차 우려 사항이 있습니다. 여기에는 권한 부여, 유효성 검사 및 로깅이 포함됩니다.

핸들러 전체에서 이 로직을 반복하는 대신 Behaviors를 사용할 수 있습니다. 동작은 요청을 수락하고 일부 작업을 수행 한 다음 (선택적으로) 요청을 전달한다는 점에서 ASP.NET Core 미들웨어와 매우 유사합니다.

로깅을 수행하는 MediatR 동작을 구현하는 방법을 살펴 보겠습니다.

### Creating our Behavior

먼저“Behaviors”라는 다른 솔루션 폴더를 추가하겠습니다. 

다음으로 LoggingBehavior라는 폴더 안에 클래스를 추가해 보겠습니다.

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 {typeof(TRequest).Name}"**)**;

var response = await next**()**;

\_logger.LogInformation**(**$"Handled {typeof(TResponse).Name}"**)**;

return response;

**}**

**}**

코드를 설명해 보겠습니다.

1. 먼저 두 가지 유형 매개 변수 TRequest 및 TResponse를 취하고 IPipelineBehavior<TRequest, TResponse> 인터페이스를 구현하는 LoggingBehavior 클래스를 정의합니다. 간단히 말해,이 동작은 모든 요청에서 작동 할 수 있습니다.
2. 그런 다음 Handle 메서드를 구현하고 next () 대리자를 호출하기 전후에 로깅합니다.

이 로깅 핸들러는 모든 요청에 적용 할 수 있으며 처리 전후에 출력을 기록합니다.

### Registering our Behavior

동작을 등록하기 위해 시작 시 ConfigureServices에 줄을 추가하겠습니다.

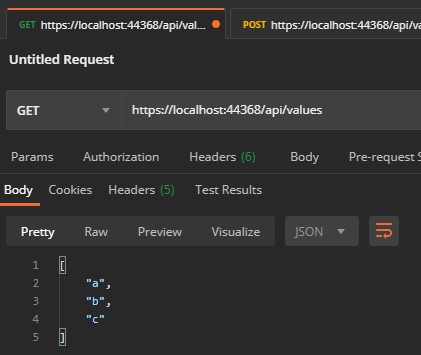
services.AddSingleton(typeof(IPipelineBehavior<,>), typeof(LoggingBehavior<,>));

모든 제네릭 유형 매개 변수에 사용할 수 있는 동작을 지정하기 위해 <,> 표기법을 사용하고 있습니다.

### Testing our Behavior

이번에는 F5 단축키를 사용하여 디버그 모드에서 실행하여 애플리케이션을 실행 해 보겠습니다.

그런 다음 Postman을 열고 Get Values를 다시 실행 해 보겠습니다.

그런 다음 Visual Studio에서 "출력"창을 열고 "다음에서 출력 표시: 웹 애플리케이션 – ASP.NET Core 웹 서버"를 선택하면 몇 가지 흥미로운 메시지가 표시됩니다.

이것은 GetValues 쿼리 핸들러가 호출되기 전후의 로깅 출력입니다.

여기서 중요한 것은 기존 요청이나 핸들러를 수정할 필요가 없다는 것입니다. 우리는 단순히 새로운 동작을 추가하고 연결했습니다.

동일한 방식으로 전체 애플리케이션에 권한 부여 및 유효성 검사를 쉽게 추가 할 수 있는 것처럼 간단하고 간결한 방식으로 교차 절단 문제를 처리하는 좋은 방법이 됩니다.

## Conclusion

이 문서에서는 MediatR을 사용하여 ASP.NET Core에서 CQRS 및 중재자 패턴을 모두 구현하는 방법을 살펴 보았습니다. 우리는 요청과 알림, 그리고 행동과 관련된 교차 우려 사항을 처리하는 방법을 검토했습니다.

MediatR은 읽기 및 쓰기 문제를 분리하고 코드 간의 종속성을 최소화함으로써 단순한 모놀리스에서보다 성숙한 애플리케이션으로 발전해야하는 애플리케이션에서 훌륭한 출발점을 제공합니다.

이를 통해 우리는 가능한 여러 단계를 추가로 취할 수 있습니다.

* 읽기에 다른 데이터베이스를 사용합니다 (아마도 ValueAddedNotification을 확장하여 새 DB에 쓸 두 번째 처리기를 추가 한 다음이 DB에서 읽도록 GetValuesQuery를 수정).
* 읽기 / 쓰기를 별도의 앱으로 분할합니다 (ValueAddedNotification을 수정하여 Kafka/Service Bus에 게시 한 다음 두 번째 앱이 메시지 버스에서 읽도록 함).

이제 우리는 단기간에 일을 과도하게 복잡하게 만들지 않고 필요한 경우 위 단계를 수행 할 수 있는 훌륭한 위치에 앱을 보유하고 있습니다.