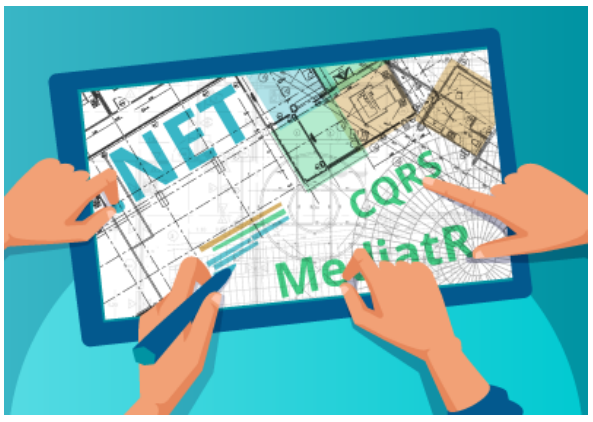
Building Microservices On .NET Core – Part 2 Shaping microservice internal architecture with CQRS and MediatR

21/01/2019

In first article in our series about building microservices in .NET core we are going to focus on internal architecture of a typical microservice. There are many options to consider depending on microservice type. Some services in your system will be typical CRUD so there is no use debating on their design (unless they are critical from performance and scalability perspective).

In this article we will design internal architecture of non-trivial microservice that is responsible for both managing its data state and exposing it to the external world. Basically our microservice will be responsible for creation and various modifications of its data and also will expose API that will allow other services and applications to query for this data.

Source code for complete solution can be found on our Github.



Overview of CQRS

Imagine `ProductService` class. It implements all operations we could do with products. It is insurance product in my example but in this context it doesn't matter. Every change in code needs an investigation how it works and what could be the side effect. It causes the code grows, becomes difficult to deal with and it is time consuming to start.

In many applications there are a lot of giant classes implementing logic and containing everything that could be done with object of given type. How to refactor them to split the code and share the functionality? Simplifying, we can distinguish two main data operations. Operations can change data or read it. So the natural way is separation of them by this category. Operations that change data (**commands**) can be distinguished from operations that just read data (**queries**). In most systems the differences between the reads and writes are essential. When you are doing a read you are not doing any validation or business logic. But you are often use caching. The models for reading and writing operations are (or need to be) also mostly different.

CQRS – **Command Query Responsibility Segregation** is the pattern that require to separate the code and models that performs a query logic from the code and models that performs commands.

Back to our example – the `ProductService` shared according to rules above becomes now:

- `FindAllProductsQuery` that returns `IEnumerable<ProductDto>` (could be also implemented as another model – `FindAllProductsResult` with collection of `ProductDto`)
- `FindProductByCodeQuery` that returns `ProductDto`
- `CreateProductDraftHandler` with input `ProductDraftDto` and adds product to our system.

We have one model shared by the queries above but in case of need to have different data in results the models should be separated (and it is often so).

So we have two pieces now: command or query class and the result class.

How to connect them?
How to know what type is an input/output of each query/command?
It is time to introduce a **mediator**. The job the mediator does in that situation is to tie these pieces together into the single request.

.NET Core 2.x and MediatR

I think we can see some code now. We use **MediatR** library that helps us to implement **CQRS pattern** in our ProductService. **MediatR** is some kind of 'memory bus' – the interface to communication between different parts of our application.

We can use Package Manager Console to add **MediatR** to the project typing:

```
Install-Package MediatR
```

Next we are registering it in DI container just by adding code `services.AddMediatR(); in ConfigureServices` method of `Startup` class.

To create query message with **MediatR** we need to add class implementing **IRequest** interface and specify the response type our query class is expecting:

```
    }
```

How to define input model? It is the parameters of controller action:

```
// GET api/products/{code}
[HttpGet("{code}")]
public async Task<ActionResult> GetByCode([FromRoute]string code)
{
     var result = await mediator.Send(new FindProductByCodeQuery{ ProductCode = code });
     return new JsonResult(result);
 }
```

Now we can send our message with **MediatR**. The controller is pretty slim. There is no logic here. The only responsibility of it is to send the client JSON response. To prepare the response we send the mediator message – call Send method of the IMediator object (injected from DI container – see below). We send `FindProductByCodeQuery` object with property ProductCode set.

```
private readonly IMediator mediator;

public ProductsController(IMediator mediator)
{
    this.mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
}
```

And here we need to define another piece of our **CQRS** solution. It is something that handles the request. The class that will answer to each message of give type.

And one more time **MediatR** makes it easy to do:

```
public class FindProductByCodeHandler : IRequestHandler<FindProductByCodeQuery, ProductDto>
{
    private readonly IProductRepository productRepository;

    public FindProductByCodeHandler(IProductRepository productRepository)
    {
        this.productRepository = productRepository ?? throw new ArgumentNullException(nameof(productRepository));
    }
}
```

As we can see handler implements `IRequestHandler` interface with definition of input and output types:

```
public interface IRequestHandler<in TRequest, TResponse> where TRequest : IRequest<TResponse>
  {
    Task<TResponse> Handle(TRequest request, CancellationToken cancellationToken);
  }
```

In our example `FindProductByCodeHandler` definition gives us (and the mediator) information that it 'knows' how to respond to `FindProductByCodeQuery` messages and that it returns `ProductDto` object. Now we need to define how to handle the message. The interface defines the Handle method which we should implement. We will go to `IProductRepository` and retrieve requested object:

```
public async Task<ProductDto> Handle(FindProductByCodeQuery request, CancellationToken cancellationToken)
        {
            var result = await productRepository.FindOne(request.ProductCode);

            return result != null ? new ProductDto
            {
                Code = result.Code,
                Name = result.Name,
                Description = result.Description,
                Image = result.Image,
                MaxNumberOfInsured = result.MaxNumberOfInsured,
                Questions = result.Questions != null ? ProductMapper.ToQuestionDtoList(result.Questions) : null,
                Covers = result.Covers != null ? ProductMapper.ToCoverDtoList(result.Covers) : null
            } : null;
        }
```

Mapping to the result type is also performed in handler class. If it is needed, we could use e.g. AutoMapper or implement some custom mapper. We could also add cachi
and any other logic that is necessary to prepare the response.

Now we have functionality of getting Product by ProductCode ready. Let's test. We use xUnit to test our .NET Core 2.x applications.

Testing while using **MediatR** and **CQRS** is quite simple. We created in `ProductsControllerTest` short method which tests the controller using bus:

```
[Fact]
public async Task GetAll_ReturnsJsonResult_WithListOfProducts()
{
    var client = factory.CreateClient();

    var response = await client.DoGetAsync<List>("/api/Products");

    True(response.Count > 1);
}
```

We should also test our handler, one of test in FindProductsHandlersTest:

```
[Fact]
public async Task FindProductByCodeHandler_ReturnsOneProduct()
{
    var findProductByCodeHandler = new FindProductByCodeHandler(productRepository.Object);

    var result = await findProductByCodeHandler.Handle(new Api.Queries.FindProductByCodeQuery { ProductCode = TestProductFactory.Travel().Code},
new System.Threading.CancellationToken());

    Assert.NotNull(result);
}
```

productRepository is mock of `IProductRepository` and it is defined in the following way:

```
private Mock productRepository;

private List products = new List
{
    TestProductFactory.Travel(),
    TestProductFactory.House()
};

public FindProductsHandlersTest()
{
    productRepository = new Mock();


    productRepository.Setup(x => x.FindAll()).Returns(Task.FromResult(products));
    productRepository.Setup(x => x.FindOne(It.Is(s => products.Select(p => p.Code).Contains(s)))).Returns(Task.FromResult(products.First()));
    productRepository.Setup(x => x.FindOne(It.Is(s => !products.Select(p => p.Code).Contains(s)))).Returns(Task.FromResult(null));
}
```

Implementation of commands is definitely the same. There is no place here to show the example but go to the full source code on GitHub (here are examples of command) where you could review all the code, organization of project etc.

Summary

I strongly recommend trying to work with **MediatR** library. It is easy to setup so gives us possibility to quick start and discover what the library and especially **CQRS pattern** gives us. I hope my text shows it keeps all things separated, each class has its own responsibility, input and output models are well-fitting and controllers are as clean as possible.

If we create different, individual requests and handlers instead of one big interface we can change any part of service functionality with no side effect. We can easily change behavior of handlers (logic!) until it still returns object of correct type – it will have no impact on the controller.

We can create new functionality by adding new request-handler pair. Or delete other by removing them. If we are new in longtime developed system – we just need to investigate small part of it – only where our maintenance is necessary.

CQRS could be also implemented in microservices architecture – the query command and/or the command handler could be implemented as separated microservices. We could implement command queue also. Different models could be used to reading and writing and microservices could use different data models. Operations could be scaled by running different number of handlers of command or query type.

Of course **CQRS** does not resolve all the problems ☺ I think definition of thousands of events does not make our system easy maintainable. And if it doesn't respond to development challenge do not use it.

Tags:

.NET • ASC LAB • Banking software • cqrs • Insurance software • MediatR • Software development • Software engineering • Software House

1 Add comment

Join the discussion...

1 0 1

1

Subscribe newest oldest most voted

Sepp

Guest

Of course CQRS does not resolve all the problems ☺ You are correct. Also authorization of resources, pagination and filtering are more complex use cases with MediatR. But it is doable with polymorphism and base classes. I currently build up an API that is using MediatR and also highly recommend it. You need to grasp the principles of the Query and QueryHandler paradigms. Also how those classes can be discovered by the IoC container (especially when they are not in the same assembly but in a separate project/assembly). After that it's a process of wiring it up and building meaning... Read more »

5 Reply 02/07/2019 12:36

Gusti

Guest

Sorry, I still can't see what MediatR is good for, except to avoid the constructor explosion ☹ What disturbs me the most is that new FindProductByCodeQuery(...) from var result = await mediator.Send(new FindProductByCodeQuery{ ProductCode = code }); I was told new is bad... It turns out it's not so bad. Also, you wrote: "So we have two pieces now: command or query class and the result class. How to connect them? [...] The job the mediator does in that situation is to tie these pieces together into the single request." Which confuses me even more... At some moment you say:... Read more »

0 Reply 19/10/2019 22:39