CQRS Implementation

Jaume Canals

Spring 2019

Contents

1 Introduction

nucoris is the exploratory prototype of a cloud-native healthcare information system (HIS) that I have developed and used as a learning platform during a sabbatical leave.

CQRS (*Command Query Responsibility Segregation*) is an architectural pattern that separates the models for reading and writing.

This document briefly explains how I have implemented the CQRS pattern in **nucoris**.

2 Overview of CQRS and its Implementation

CQRS is an architectural pattern that **separates the models for reading and writing**, under the assumption that system operations can be divided into:

- **Commands** that modify the system state.
- **Queries** that answer a question by returning some data without modifying state.

The separation can be applied at different levels: at the application layer, the persistence layer or the database, and you may even create a service for commands and a different service for queries in a microservices architecture, each service with a completely different technology stack.

The more levels at which you apply CQRS, the more flexible but the more complex the resulting system becomes. So, there is a tradeoff that in **nucoris** I have resolved as follows:

1) There is a **single NoSQL document database** (Azure Cosmos DB).
   Its usage is explained in detail in the *nucoris Persistence* document, but in a nutshell:
   a. Command handlers and queries on data of a *single patient* use domain repositories.
   b. Queries on *multiple patients* (used by the web app in searches or to populate certain worklists) will interact with specialized repositories that do not depend on the domain entities.
2) There is a **separation at the application layer** between commands and queries.
   a. Command handlers use repositories to read domain entities, interact with the domain entities to perform system state changes, then use the repositories to persist them.
   b. Query handlers may use domain repositories for patient-specific simple queries or specialized repositories for multiple-patient queries that I call "**materialized views**" (see *nucoris Persistence* document for details).
3) The **presentation layer** (*Razor Pages* and a *REST API* served by an ASP.NET Core web application) uses the **mediator pattern** to send commands and queries to a mediator,

which is responsible to invoke the right handler and return a result. Repositories are injected into the handlers as needed so they can fulfill the request.
In simple requests a repository may also be directly injected into the REST API or Razor Page controller.

3 Presentation Layer

3.1 REST API

As an example, here you see how a request to document a new allergy on a patient is implemented in the REST API controller.

Note that in a real application, documenting an allergy might include information on the severity of the reaction, when it was first experienced and other comments such as “only triggered after physical exercise”, but in this prototype we just document the name of the allergen.

```
[Route("api/patients/{patientId:Guid}/allergies")]
public class AllergiesController : ControllerBase
{
    private readonly MediatR.IMediator _mediator;

    public AllergiesController(MediatR.IMediator mediator)
    {
        _mediator = mediator;
    }

    [HttpPost("{allergy}")]
    public async Task<IActionResult> PostAsync(Guid patientId, string allergy)
    {
        var cmd = new application.commands.AddAllergiesCommand(patientId,
                                            new List<string>() { allergy });

        var result = await _mediator.Send(cmd);

        if (result.Successful)
        {
            return Ok();
        }
        else
        {
            return StatusCode(StatusCodes.Status304NotModified,
                   $"Unable to add allergy {allergy} to patient {patientId}");
        }
    }
```

Note: some error control code removed to keep the sample short.

Highlights:

a) I use **MediatR** **mediator pattern** implementation, which is injected into the controller via its constructor. (As a side note, I rely on **Autofac** **IoC container**).
b) In the asynchronous *Post* method implementation, I instantiate the `AddAllergiesCommand` and send it to the mediator, which will find the right handler.

3.2 Views

I use **Razor Pages**, introduced in ASP.NET Core 2.0, to generate the HTML views of **nucoris** web UI. One of them is the *Admitted* page, which shows a list of admitted patients with their name and admission time. Since the query feeding this view contains data from multiple patients, I've created a specific materialized view query for it, named `PatientStateQuery`. The resulting code of the *Get* method of this Razor page model is very similar to the REST API implementation above:

```
public class AdmittedModel : PageModel
{
    private readonly MediatR.IMediator _mediator;

    public List<PatientVM> AdmittedPatients { get; set; }

    public AdmittedModel(MediatR.IMediator mediator)
    {
        _mediator = mediator;
    }

    public async Task OnGetAsync()
    {
        // Get all admitted patients:
        var specification = new
                PatientStateViewSpecification(PatientAdmissionState.Admitted);
        var query = new PatientStateQuery(specification);
        var patients = await _mediator.Send(query);

        this.AdmittedPatients = patients.Select(qp => new PatientVM(qp)).
            OrderBy(vp => vp.DisplayName).ToList();
    }
}
```

Highlights:

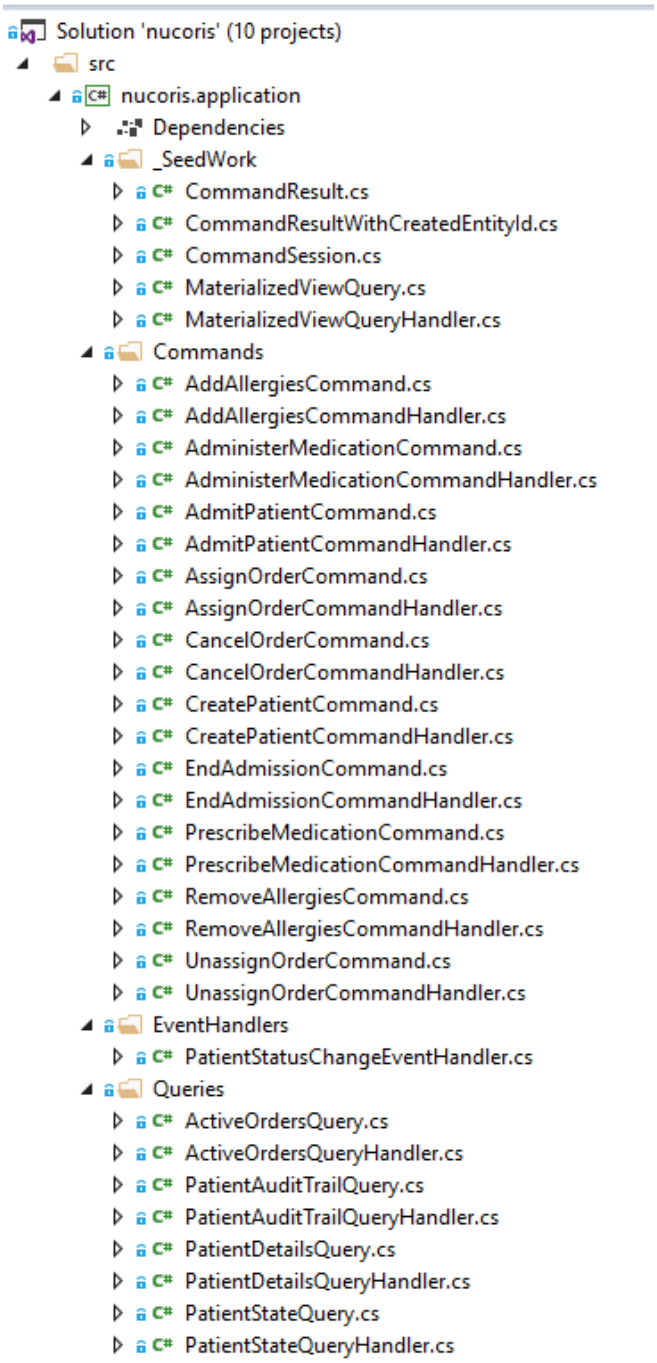
a) As with the REST API, here I use the mediator pattern.
b) I create a specification of the conditions for the query, here restricting it to just *Admitted* patients.
c) The view uses a separate view model object, here instances of `PatientVM`.

4 Application Layer

4.1 Project Organization

If you inspect the *nucoris.application* project you will see:

- **SeedWork** folder with some small reusable or base classes (named after a suggestion from .NET Microservices Architecture Guide)
- **Commands** folder containing both the definition of the commands and their handlers. In a final application you may decide to store commands and handlers separately, or to create a subfolder per area, but in this prototype I've found convenient to keep them together.
- **EventHandlers** containing handlers of domain events. In this prototype there is only one, which I use to maintain the PatientState materialized view (see the *nucoris Persistence* document for details)
- **Queries** folder containing both the definition of the queries and their handlers.



4.2 Commands

A command is a simple POCO class. For example, the `AddAllergiesCommand` we've seen before is defined as (some error-checking code removed to save space):

```
public class AddAllergiesCommand : MediatR.IRequest<CommandResult>
{
    public Guid PatientId { get; }
    public List<string> Allergies { get; }

    public AddAllergiesCommand(Guid patientId, IEnumerable<string> allergies)
    {
        this.PatientId = patientId;
        this.Allergies = allergies.ToList();
    }
}
```

Highlights:

a) Commands implement the "empty" `MediatR.IRequest<T>` interface, so that at startup the IoC container (Autofac in our case) can find them and bind them to their handlers.

4.3 Command Handlers

Let's jump directly into code, it is easy enough:

```
public class AddAllergiesCommandHandler :
             IRequestHandler<AddAllergiesCommand, CommandResult>
    {
        private readonly ICommandSession _commandSession;
        private readonly IPatientRepository _patientRepository;

        public AddAllergiesCommandHandler(
            ICommandSession commandSession,
            IPatientRepository patientRepository)
        {
            _commandSession = commandSession ??
                              throw new ArgumentNullException(nameof(commandSession));
            _patientRepository = patientRepository ??
                              throw new ArgumentNullException(nameof(patientRepository));
        }

        public async Task<CommandResult> Handle(
                     AddAllergiesCommand cmd, CancellationToken cancellationToken)
        {
            var patient = await _patientRepository.GetAsync(cmd.PatientId);

            if (patient == null)
            {
                return new CommandResult(CommandResultCode.EntityNotFound, "Patient not…
            }

            foreach (var allergy in cmd.Allergies)
            {
                patient.Add(new domain.Allergy(allergy));
            }

            _patientRepository.Store(patient);

            var success = await _commandSession.CommitAsync();
```

```
            return new CommandResult(success);
        }
    }
```

Highlights:

a) Command handlers implement `IRequestHandler<Command, CommandResult>`. This is how they are bound to the commands they handle by the IoC registration at startup.
b) The `ICommandSession` (a wrapper over our unit of work), repositories and any other dependency is injected via the constructor.
c) The handler method retrieves the domain entity from the repository implementation, calls the entity methods as needed, stores the changes and calls `CommitAsync` to finish the command session.

4.4 Command Session

`CommandSession` is a **wrapper over our unit of work** and is responsible for **dispatching the domain events** (using again `MediatR.IMediator`) before committing the changes by calling the unit of work *commit* method. Here you have a glimpse of its main method:

```
public CommandSession(
    IUnitOfWork unitOfWork, MediatR.IMediator mediator, IEventRepository eventRepository)
        {
            _unitOfWork = unitOfWork ?? throw new ArgumentNullException("Unit of work");
            _mediator = mediator ?? throw new ArgumentNullException("Mediator");
            _eventRepository = eventRepository ?? throw new ArgumentNullException("Event
repository");
        }

public async Task<bool> CommitAsync()
        {
         // The implementation of the command handler may generate domain events.
         // We are interested to collect and store them as part of the same transaction:

            List<DomainEvent> events = await DispatchEventsAsync();

            _eventRepository.StoreMany(events);

            return await _unitOfWork.CommitAsync();
        }
```

4.5 Query Handlers

Query handlers are implemented similarly to command handlers, except that they are not involved in a command session. Take as an example a query that feeds the view showing all events that constitute the audit trail of a patient:

```
public class PatientAuditTrailQueryHandler :
   IRequestHandler<PatientAuditTrailQuery, nucoris.queries.PatientAuditTrail.Patient>
{
    private readonly IPatientRepository _patientRepository;
    private readonly IEventRepository _eventRepository;

    public PatientAuditTrailQueryHandler(
            IPatientRepository patientRepository,
            IEventRepository eventRepository)
    {
        _patientRepository = patientRepository ??
                throw new ArgumentNullException(nameof(patientRepository));
        _eventRepository = eventRepository ??
                throw new ArgumentNullException(nameof(eventRepository));
    }

    public async Task<nucoris.queries.PatientAuditTrail.Patient> Handle(
        PatientAuditTrailQuery request, CancellationToken cancellationToken)
    {
        Guard.Against.Null(request, nameof(request));
        Guard.Against.Condition(request.PatientId == default(Guid), "PatientId");

        var patient = await _patientRepository.GetAsync(request.PatientId);
        Guard.Against.Null(patient, "Cannot find patient");

        var events = await _eventRepository.GetPatientEventsAsync(request.PatientId);

        var patientAuditTrail = PatientAuditTrail.PatientAuditTrailFactory.
            FromDomain(patient, events);

        return patientAuditTrail;
    }
}
```

Highlights:

a) There is no `ICommandSession` involved, since there is no transaction around queries, nor changes are expected in entities.

b) As a side note, in this example I have not removed the guards, adapted from Steve Smith's GuardClauses project.

c) Domain repositories are used in this case, since the query is simple enough that it can be fulfilled without special repositories. The method `GetPatientEventsAsync`, though, was created specifically for this query and it's not used by commands.

d) The query returns an ad-hoc type, `nucoris.queries.PatientAuditTrail.Patient`. The caller (a Razor Page) may decide to use it directly to build the HTML view, for simplicity, or create an intermediate view model object (in this case, the former option was more convenient).

e) The construction of the returned type is delegated to a factory of this type, to keep the code cleaner since it could be complex sometimes.