A guide to CQRS : The missing Link

# Introduction

This guideline is somewhat the “missing-link” in our architectural documentation on how to use a CQRS[[1]](#footnote-1)-pattern based framework (or shouldn’t we rather say CQRS-pattern based library …).

**The document is build up into next parts:**

* I will start this journey on how to correctly use the CQRS-pattern in our FAVV (BeCert) related applications by discussing the well known **“ONION-application-architecture”** and how it’s components and boundaries should apply to the CQRS-pattern we use in our applications.
* Next I will map this “ONION-application-architecture” to a more concrete implementation of CQRS, this by implementing the different boundaries as real artifacts within a “real-life” application which mimics the initial create of a Certificate (strongly simplified of course). I will present this demo case by using a recent implementation of the CQRS framework, which is already ported to **.NET Standard and .NET Core !** The CQRS library is called **CQRSLite** (which was initially created by the founder of CQRS : ***Greg Young***, you can find a copy here:
  + <https://github.com/gautema/CQRSlite>

|  |  |
| --- | --- |
|  | ***Important remark:***  *Although the goal of this document is not an extensive tutorial on how to CQRSLite, it’s worth mentioning that this “lightweight framework” (or should we rather say component library) could be taken as the base for our future CQRS-implementations in .NET Core/Standard related projects. The CQRSLite framework has some serious advantages in contrast to our current CQRS implementation, this is not only the fact that conceived with .NET Standard in mind (it is totally compatible with the .NET Standard sample domain model I created) but it has some nice “build-in” features like async handling (yes, we can put a messaging layer on top !), snapshots, cancelable query and command handlers and CacheRepository. And more important, it has a far less “footprint” then the CQRS framework (Bricks) we currently use. We should also rather call it a “CQRS-component-library” instead of a framework. Of course the framework should be “pluggable”, compatible and work side-by side with our current application, we may achieve this by writing a factory or strategy above the concrete implementation.* |

# Definition of the ONION application architecture

|  |
| --- |
|  |
| The onion architecture is comprised of multiple concentric layers which have strict boundaries between them. The first is the **UI-layer** which should only know by the **Application services layer.**  The **Application services layer** typically offers **“services”** to the **UI-layer.** These services are called **Application Services[[2]](#footnote-2)** and in most cases these are **Webservice** based (WebAPI,WCF) service implementations. The **UI-Layer** doesn’t know anything about the underlaying domain model and communication between UI-Layer and Application Services is typically done through DTOs.[[3]](#footnote-3) We will come back to the use of DTOs later in this document …  Next layer in our ONION architecture is the so-called **Infrastructure layer** which holds a collection of **Repositories**, **Factories**  and **Domain Services.** Repositories are typically used to create/persist instances of Aggregates (write-side) or Repositories can also be wrappers round so-called “thin-data layer” to handle the read-side of our application. Both “read” (or query) and “write” (or command) side will be discussed later in this document. **Factories** are only used when creating complex (composed) Aggregates and **Domain-Services** are used to communicate with the outside-world (outside the domain layer), e.g. a domain-service may be used to notify clients about changes in the domain model, we will see an example of this later in this document.  Finally, the most inner-layer of our ONION architecture is the **Domain layer** itself. This layer contains the domain-model of a specific bound-context and may be composed by **Entities**, **Aggregates, ValueObjects** and **Domain Events.[[4]](#footnote-4)** |

# Stereotypical Architecture

|  |
| --- |
|  |
| Before diving into the details of CQRS, let’s briefly talk about the **Stereotypical Architecture** first. The Stereotypical Architecture, as shown above, is the most common architecture we used to implement (and @FAVV, most applications are based on this architecture).  This architecture has **a client (UI) layer** which requests and **returns DTOs** to the **Application Layer** by means of **Application Services**. In between is a so-called **“Remote Façade”** (some kind of web based Api) which acts as a gateway between the client and the backend.  Next in the **Application (or business) layer DTOs get mapped to domain models (or domain models to DTOs)**, validated and persisted in the database.  ***Benefit*** *of the Stereotypical Architecture is it’s “simplicity”, but in mean time* ***drawback*** *of this simplicity is that it may neither* ***scale*** *or* ***perform*** *conveniently in heavy loading scenario’s. This mainly because of the use of a single model (both for the read and write side) and mapping from/to DTOs may be painfull to. I’ll come back to a better approach by separating the model into “read” and “write” model and create a separate flow for handling “reads” and “writes”. I’ll come back to this when discussing the basics of a CQRS based application.* |

|  |  |
| --- | --- |
|  | ***Important remark:***  *Although CQRS based frameworks may be more convenient in heavy loading scenario’s, the* ***Stereotypical Architecture*** *will work for most of the applications. CQRS brings complexity and should only be applied sparely when really needed or in specific area’s of the application, surely not in all of them !* |

# CQRS Architecture

## Separation of Concerns : Read-Side vs Write-Side

|  |  |
| --- | --- |
| Read-Side (Query) | Write-Side (Command) |
|  |  |
| In CQRS there is a strict separation on how to “read” data and how to “persist” data, each resulting in a specific model optimized either for “read” or “write”.  Read-Side (represented by queries) model will typically store data in denormalized format, optimized for fast-querying the data from the underlying datastore. As shown above, read-side will also typically omit the domain model and rather directly read (through a thin read layer) from the Data Storage. Depending on the application needs, we also may store multiple projections of the same data (e.g. mobile clients may have to present data in a different way then web applications do). The persistence technology used in the read-side may typically be a NO-SQL or document based approach, so the created projections can be stored as-is and queries are fast. Another benefit from separating the read from the write side is that you can scale the read-side independently from the write-side (as in most application the number of reads are far more important then the number of writes, so it may be beneficial to “scale-out” the read part depending on the circumstances).  Write-Side (represented by commands) is a whole other story. Typically clients will send messages (mapped from (partial) DTOs and represented by commands) to the Application Layer. Next domain model (in the form of an Aggregate) will be created from the supplied command object. Next depending on the underlying persistence technology, the changed Aggregate state may be persisted as a stream of Events or in a normalized (mostly in 3NF format) SQL-based database. We typically talk about a CQRS/ES implementation in case of event streaming.  *I think we all agree that applying CQRS is a total different approach then applying the more classic Stereotypical Architecture. Although CQRS brings complexity, holds many different technologies and certainly in case of CQRS/ES possible headaches in case of model-changes (as when model changes, the way events are replayed to form the Aggregate changes to …), it may be very efficient in heavy-load scenario’s, or situation when object state must be brought back to a certain point in time.* | |

## Synchronization principle between Read and Write Side

|  |
| --- |
|  |
| Well, we know now that in CQRS we have a read-side which is totally separated and isolated from a write-side, and both sides may even be using total different persistent technologies and even be hosted on different platforms and servers. But … we have one missing piece here, namely: how can we be sure that when “persisting” data on the “write-side” (using Events Store or SQL-3NF-Db) that we get the same result when querying the “read-side” ? Well this is done by sending and external-event after data is persisted. Though, be aware that as we have no transactional unit, read and write side may be temporarily “out-of-sync”, this is also called “eventual consistency” (as the arrow “Eventually” mentions). |

|  |  |
| --- | --- |
|  | ***Very very Important Remark !!!***  *Having the issue of “eventual consistency” in CQRS means that (as I already mentioned a few times) we should have a* ***PROPER MESSAGAING TRANSPORT layer (NServiceBus)****, so in case of transient errors (temporary outages), messages are stored on queues and won’t be lost when the system comes up again, as these messages will be resend as soon as the system comes up again !!!* |

# CQRS Best Practice

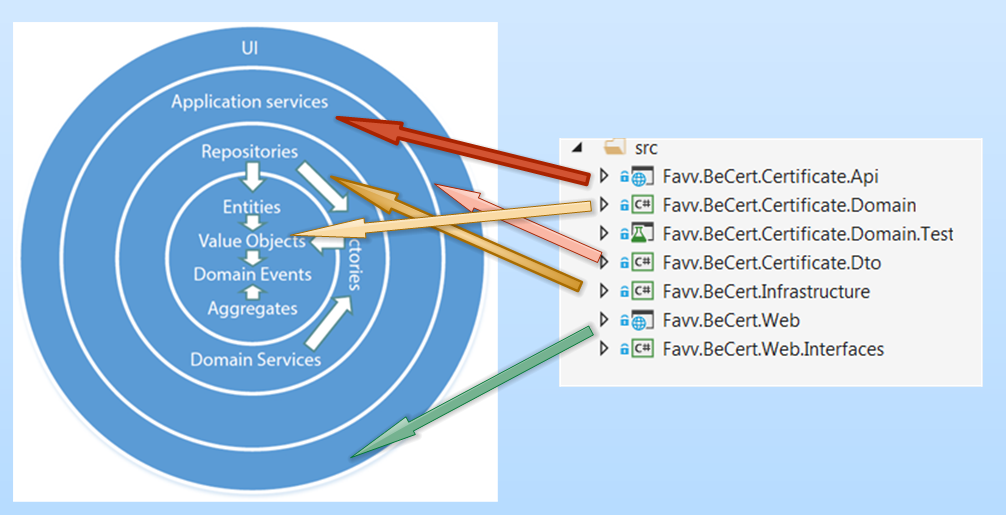
## Introduction

As we now fully understand the theory behind CQRS 😊, let’s look how the CQRS-pattern conforms to the ONION structure and how we should structure our Visual Studio project accordingly, without breaking the strict Boundaries as imposed by the ONION-application-architecture !

## Sample Application

|  |
| --- |
|  |
| The image above shows a simple web application I’ve build (in .NET Core). This application will create a “minimal” Certificate by supplying the enterprise number of the operator, the product category, product and country for which the Certificate applies to. Please note that this is a very simplified representation and in the sample we have to provide all info manually (in a real-world implementation items like product category, product and country should be selectable from a drop down list). Anyways … as goal of this document is proper CQRS design (and not proper UI-design ) this simple user interface suffices …  Next, in upcoming §, we will go through the details of the initial Certificate Creation process, but first have a look at the common CQRS-structure I’ve applied in Visual Studio and how this applies to the “Boundaries” of an ONION-based architecture design ! |

## ONION Architecture Conform Application Structure



Before I go into the details of the proposed application structure (in our case modeled in Visual Studio), let’s look how our structure complies with the proposed ONION Architecture by looping through them in a logical order:

* **Favv.BeCert.Certificate.Web**
  + This project contains our client-facing web application (user interface / presentation layer) and applies to the outer-most layer in our ONION Architecture (UI). This Layer only “knows” about the “Application Services” layer and does not know anything about the underlaying Infrastructure and Domain Layer. The communication between the UI and the Web is commonly realized through well-defined data contracts, aka DTOs.
* **Favv.BeCert.Certificate.Api**
  + This project represents the “Application Services” layer, typically represented by some kind of web-service architecture (SOAP, REST or WCF). This Api layer executes services for the UI/PL in the form of providing endpoints to request (GET), create (POST), update (PUT), partially-update (PATCH) or delete (DELETE) DTOs. So the Web application only knows about DTOs that it get “serviced” from the Application Services Layer.

|  |  |
| --- | --- |
|  | ***Important note:***  *Very important but not visually mentioned is to note that in our case the Application Services Layer will have a reference to the* ***CQRS-library****, more specific, the Application Services Layer will get task-based “Request Messages” from the UI/PL (in the form of DTOs) and will next “Map” the incoming DTO to the appropriate Command which will then be taken and process by the CQRS-Command Handling framework.* |

* **Favv.BeCert.Certificate.Dto**
  + As already mentioned, the DTO project defines the “contract” between the UI/PL and the Application Service Layer and defines how data should be formatted when communicating between these layers.
* **Favv.BeCert.Infrastructure**
  + Infrastructure is all about how to load and persist the domain logic layer components (Aggregates to be more precise). Infrastructure contains Factories, Services and Repositories which are responsible for creating, validating and persisting the core business layer of our application.
* **Favv.BeCert.Domain.Test**
  + There is always a “debate” on where a development team should apply proper (unit) testing. We can argue if “common” components like APIs and Repositories should be separately unit-tested, but IMHO, if 1 layer should be properly unit-tested (and possibly by applying TDD[[5]](#footnote-5) principle, then the “Domain Layer” is the right candidate !

|  |  |
| --- | --- |
|  | ***Important note:***  *Proper testing of the Domain Layer should be in close relation with the business and product owner of the application.*  *Even more, the unit-testing scenario’s should rather be established by the business (or by intermediary of the product owner) and should in no means be “invented” by the development team, at least that is my opinion ! :😉* |

## CQRS Architecture : Write-Side and Read-Side

|  |  |
| --- | --- |
|  | Before we go into the details of the Visual Studio solution, it is very important to explain the applied structure.  As already mentioned in §4 (when I talked about the common CQRS Architecture), there is a distinction between “read” and “write” which is, as you can see from the image on the left quit “symmetric”.  This means that both “read” and “write” side will have their events, handlers, model and infrastructure layer and it is a good idea to put that “explicitly” in the Visual Studio Solution project ! |

|  |  |
| --- | --- |
|  | ***Important note:***  *In the image above you can see that our Domain model has both “read” and “write” side. In this example I’ve put both “read” and “write” in a single solution but you could even get a bigger isolation when creating a separate solution for both read- and write side !* |

## Dissecting the Sample Application : Write-Side

### Introduction

In upcoming § I will go through the details of each layer, as an use case I will take the sample of a certificate creation request in BeCert this will explain the **“Why’s”** and **“When’s”** of applying CQRS in our projects. So let’s start with the UI/PL. This part will be specific for the “Write” side, next § will be specific for the “Read” side.

|  |  |
| --- | --- |
|  | ***Note:***  *Please note that the proposed use case of certification creation request is a very simplified version of the initial certificate creation request in BeCert. The proposed certificate creation request process for this example takes the Enterprise Number, Product Category, Product and Country as input parameters and when the certificate is created in the event store, the status Created is returned to the user.* |

### UI/PL Layer

#### 5.4.2.1. Project Structure

|  |  |  |
| --- | --- | --- |
|  | | * On the right is the structure of our Web project, which represents the User Interface / Presentation Layer of our solution. * For this demo, I use Razor-Pages (instead of MVC …). There is a single Razor-Page in our application which represents the creation of a Certificate. * The application also contains a “Services” folder, which holds a proxy to our Application Services Layer (which will be detailed later …) * The View Models folder holds a single View Model which contains the data-context for a new Certificate and the necessary logic to “talk” with the Application Services Layer. * Finally we have a “Startup” which does some initialization stuff … * Web only references “DTOs” project. |
|  | ***Note:***  *I distinguish between* ***UI/PL.*** *UI represents* ***the User Interface****, PL the* ***Presentation Logic****, represented by* ***View Models****. The View Models can be seen as the mediator between the User Interface and the Application Services layer, which will come more clear further in the document …* | |

#### 5.4.2.2. Startup

|  |  |
| --- | --- |
|  | |
| One thing worth noting in our startup class is the initialization of our ApiClient, more precise setup of the URL where the base address of the web service is. | |
|  | ***Note:***  *For sake of simplicity, the “URL” is “hard-coded”, in real we should store the URL in a (secured) application configuration file.* |

#### 5.4.2.3. Api Http Client Proxy

|  |
| --- |
|  |
|  |
| Our web application does not have a “hard-reference” to the Application Services Layer (Api in our case), but holds a “Http Proxy” and sets the endpoint URL for the Api services it can call. |

#### 5.4.2.4. Send the Certificate Creation Message Request

|  |
| --- |
|  |
|  |
| Our work flow starts after the user fill’s in the “Certificate” request form and hits the “Save” button. Next the VM (which is set as the Data-Context of the View) holds the certificate request values as entered by the User. Next the SaveAsync() method is called on the VM … |

#### 5.4.2.5. ViewModel : Map to DTO

|  |  |
| --- | --- |
|  | |
| Next, in our View Model we map our user’s request (which is part of the VM) to the “data-contract” which is known by our Application Service Layer. Next we call the “RequestNewCertificate” (POST) endpoint on our Api Proxy with the DTO as argument. This will send the request to the Application Service Layer (represented by the API) and we await the result ! | |
|  | ***Note:***  *The attentive reader may note that the DTO model in this sample is not different from the VM model (on property base). But in more complex situations this may be the case. And even when DTO and VM contract are equal, a VM should still be used because VM also contains “behavior” (is not only a getter;setter property-bag but also has methods like the Save method) and is used as the Data Context for our view !* |

### Application Services Layer

#### Project Structure

|  |
| --- |
|  |
| In our simple example, the Application Service Layer is a REST-based .NET Core Web-API project, containing a single controller called “Certificate Controller”, having a single method called “CreateCertificateAsync”. |

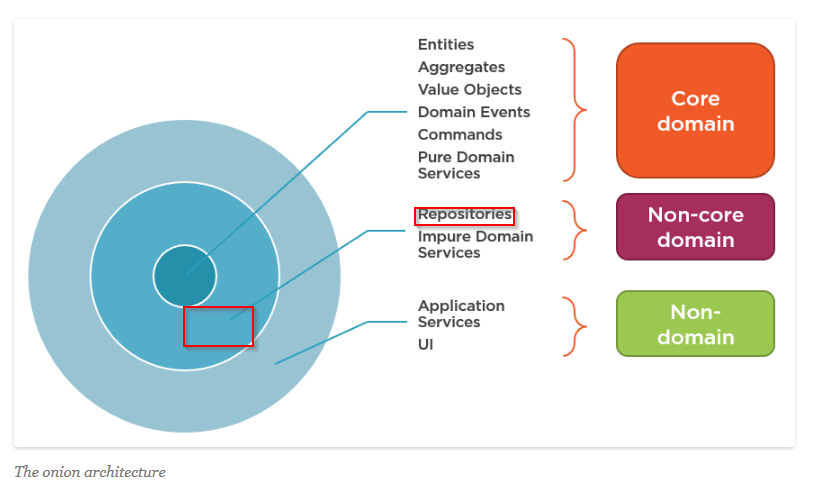
#### Command Creation & CQRS-service bus delegation

|  |
| --- |
|  |
| Most important part in the Application Sevices layer is the transformation of the incoming DTO request message into the CreateCertificateItemCommand and delegation of the command to the underlying CQRS-CommandSender service bus which will delegate the Command to the right “CommandHandler”. |

### The Infrastructure & Domain Layer

#### Project Structure

|  |  |
| --- | --- |
|  |  |
| The **\_commandSender.SendAsync(command)** call in the **Application Services Layer** (as explained in the former §) will invoke the appropriate **Command Handler** which is defined in the **Domain Layer** project.  The may sound a bit confusing, and it is too … because there is a lot of ongoing discussion on the internet whether **Commands** and related **Command Handler** belong either to the **Application Services Layer** or the **Domain Layer.**  Personally (and this is also the main idea in one of the best discussions I found concerning the position of **commands** and **command handlers** in the ONION architecture, which link you can find here[[6]](#footnote-6)), I personally think, although both commands and command handlers are defined in the Domain Layer, **they do not belong to the core of the domain layer** but exist rather **on the edge** between the **Application Services Layer** and the **Domain Logic Layer.** So on the left is the original picture (as taken from the provided URL), the right is the correction and what I think should be the correct interpretation of where commands and joining handlers belong to:   |  |  | | --- | --- | | Initial (non-correct) Interpretation | Corrected Interpretation | | [The onion architecture](https://i1.wp.com/enterprisecraftsmanship.com/wp-content/uploads/2019/01/1.png) |  | | |

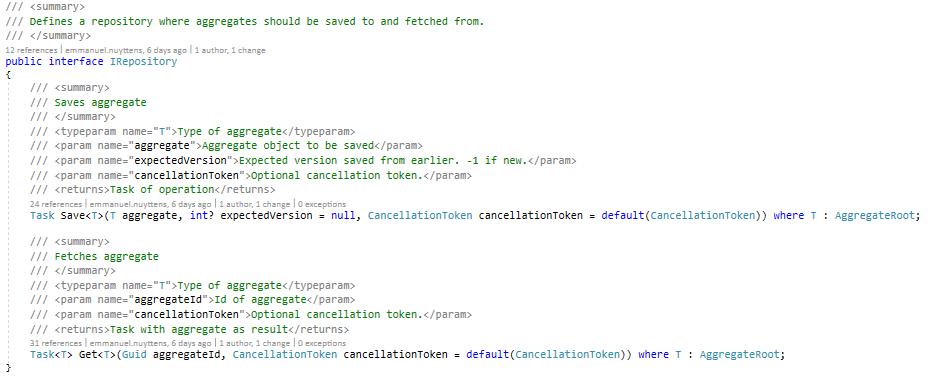


But … wait a minute … the attentive reader ask next question*: “What about our* ***repositories then*** *which belong to the* ***Infrastructure Layer****, where do this fit in then ?****”.***

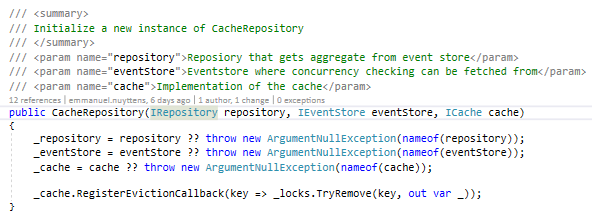
That is a correct question, as we didn’t explicitly mention the Infrastructure Layer, because although the the **Infrastructure Layer** is a concentric layer in the onion model, it is rather **implicit** for our **CQRS** implementation, let me explain this through next picture:

|  |
| --- |
|  |
| I think the picture above makes it more clear about where the responsibilities are. Command Handler will use repository (which is part of infrastructure) to load the current state of the aggregate and repository will also be used to persist current state of the aggregate after updates have been applied. In our case, the persistence will lead to events stored in an event sourcing database. |

In the CQRS-framework (or should I rather say CQRS-library) that I use, repositories are implicit (so we do not have something like certificate repository or so). In our case, the CQRSLite framework provides an **Interface** called **IRepository**, which resembles next:



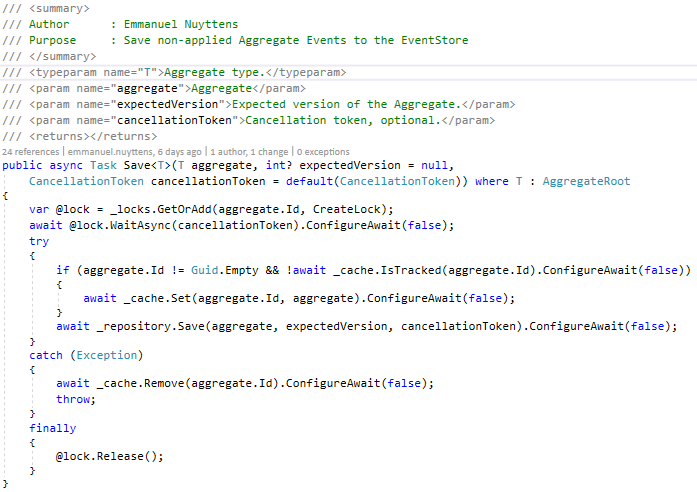
Repository interface is used to load or persist (save) aggregates from or to the (event)store. Internally, the system uses a **CacheRepository** implementation, which “decorates” a **Repository** implementation as shown below:



And next we have 2 methods, one for retrieving **events** from the **event store** (which will reconstruct the current state of the **Aggregate)** as shown below:



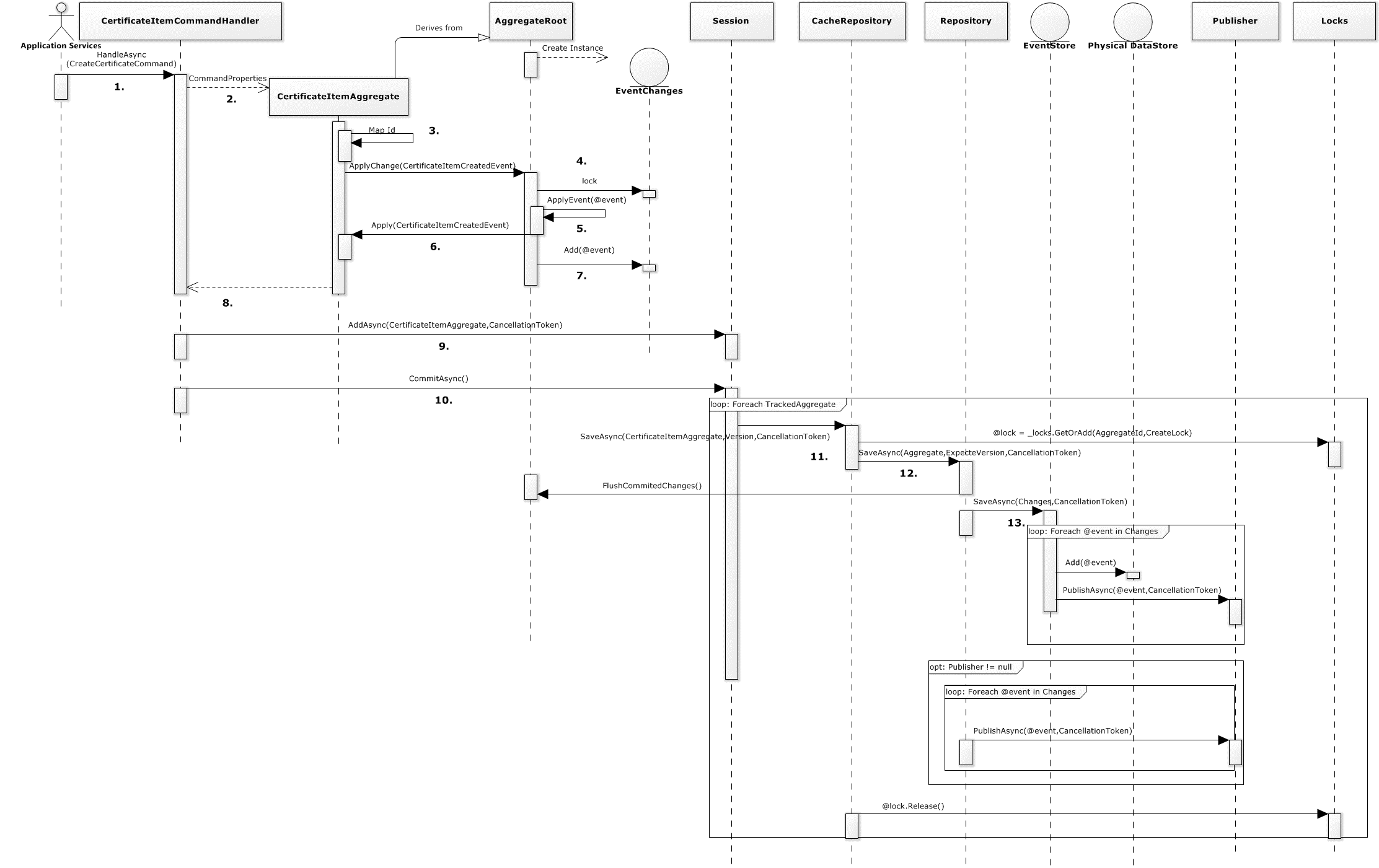
The second will take all unsaved events from the aggregate and save them in the events store:



#### Command Handler

Ok, where were we ? as this was quite a bit of deviation, but I hope that some of the confusion that exists is gone now ;-).

So, we were at the point that the command handler was trigger from the command (create certificate request) in the API Controller of the Application Services layer, so we are right in the command handling code block right now: There are a lot of steps involved in the process flow of the “write-side” of CQRS. Let’s summarize these steps by presenting a sequence diagrams which starts when the CreateCertificateCommand is received by the Command Handler:



|  |  |  |
| --- | --- | --- |
| Step | Info | Component |
| 1. | The **CreateCertificateCommand** is received by the **CertificateItemCommandHandler** and the **HandleAsync** method is resolved from the **CQRS-framework.**    This command holds the certificate properties which are needed to create a minimal Certificate. | CertificateItemCommandHandler |
| 2. | The **CertificateItemCommandHandler** creates an instance of the **CertificateItemAggregate** (which is a specialization of an Aggregate) and the **CommandProperties** are send to the **Aggregate**: | CertificateItemCommandHandler |
| 3. | The constructor of the **CertificateItemAggregate** is called, the **Aggregate Id** is mapped and the **ApplyChange()** method is called on the base **Aggregate** class supplying a **new CertificateItemCreatedEvent.** | CertificateItem |
| 4. | In the **AggregateRoot,** we first put a **lock** on the list of **changed events.** Next we call the private **ApplyEvent** method of the **AggregateRoot**. | AggregateRoot |
| 5. | From the supplied @event type, the method uses the CQRS-framework to figure out which “Apply” method should be called on the concrete Aggregate object (CertificateItem in our case). | AggregateRoot |
| 6. | The properties of the event are mapped to the current property values of our CertificateItem Aggregate. | CertificateItem |
| 7. | Back in the AggregateRoot, the **CertificateItemCreated** event is added to the list of changed events. | AggregateRoot |
| 8. | The Certificate Aggregate is returned to the Command Handler. | CertificateItemCommandHandler |
| 9. | Next, still in the HandleAsync Method the **AddAsync** method is called on the **Session** object. The Session object implements **ISession** and has methods to **add Aggregates to the Session**, get a specific A**ggregate** from the session and **persist** the aggregates to the underlying data store (event store). The code first checks if the aggregate is already tracked, if not then it is added to the list of tracked aggregates. | Session |
| 10. | When back in the HandleAsync method of the CommandHandler, the final method on **the Session** object is called: **CommitAsync.**  This method will create a task for each tracked aggregate and write (add-only) the changed events (stored in \_changes veriable) from the aggregate to the eventstore. | Session |
| 11. | First we put a lock on the Aggregate, so we are sure to be the only one changing it. Next we add the Aggregate to the Cache (if not already in …). Next we call the **SaveAsync()** method on the decorated **Repository** object. After commiting all changes, we release the lock on the Aggregate.   |  |  | | --- | --- | |  | ***Important Note:***  *The* ***Session*** *object will first call the* ***SaveAsync()*** *method on the* ***CacheRepository*** *which in turn decorates the real* ***Repository*** *which is responsible for delegating the* ***changed events*** *to the underlying* ***event store*** *to be finally persisted in the* ***event store database*** *!* | | CacheRepository |
| 12. | In the **Repository** we first get all **uncommited changes events** for the current **Aggregate** (in our demo case there is only 1, but there may be many …). After we get all the uncommited events of the Aggregate we call the **SaveAsync()** method on the **EventStore.** Sothe Regular repository deals with storing uncommited Aggregate Events to the underlying event store, and next **notifying the read-model** via the **IEventPublisher** implementation (internal application bus). It also deals with **loading all historical events** for a particular Aggregate by using the **IEventStore** implementation. | Repository |
| 13. | Any events that are stored in the current session and which are persisted in the **EventStore** through the event store’s **SaveAsync** method are next also **internally published** within the **domain model** using an **internal process publisher** by using the **PublishAsync** method. This method will **resolve the appropriate EventHandler** which is **CertificateItemCreatedEventHandler** which will be responsible for **updating the readmodel** and **publish the event(s) on the message queue** to notify the **subscribed clients (mostly UIs)** that something has changed in the domain model. | EventStore |

|  |  |
| --- | --- |
|  | ***Important Note:***  *The attentive reader may have noted that the* ***publish method*** *(which notifies the read-model on changes) can be triggered in 2 locations : either in the* ***repository*** *(after all events are commited to the eventstore) or in the* ***event-store*** *itself (where each addition to the eventstore will trigger a publish). That’s why that* ***null-test*** *is made in the repository before activation of the publish !* |

## Dissecting the Sample Application : Read-Side

### Internal Publish

In former § we discussed the **“write-model”** which is responsible for persisting the data in the EventStore Database (or regular database in case of no event sourcing).

In this section I will discuss the **“read-model”** which **comes in play after the new events have been stored in the event store.**

Any events that are stored in the current session and that end up being persisted into the **event store** are next **internally published** within the **domain model**, this through an **internal process publisher,** this can be seen in the image below by using the **PublishAsync** method.

Next there are **(internal) event handlers** that act on those **published events**, this is what we will discuss in next §.

|  |
| --- |
| Event Store: (Part of CQRS framework) |
|  |
|  |

|  |  |
| --- | --- |
|  | ***Important Note:***  *As mentioned before, we could also activate the Publish() method in the Repository SaveAsync() method instead too ! The only difference is that the publish is done on another level, namely a PRE publish in case we do publishing from the repository and a POST publish when executing the publishing from the event-store.* |

### Internal Event Handlers : Update Read Model and Send notification Message

|  |
| --- |
|  |
| In our sample demo for the **CertificateItemCreatedEvent,** we have 2 domain internal **event handlers** which will act upon the **PublishAsync** method as described in §[Introduction](#_Introduction).  These event handlers are called **CertificateItemDetailsViewEventHandler** and **CertificateItemListViewEventHandler** and will be detailed next ! |

#### CertificateItemDetailsViewEventHandler

|  |
| --- |
|  |
| This event handler will first update our first instance (projection-1) of the read-model, which holds specific projections for detail-views. After the read-database-projection has been physically updated (here represented by a simple in-memory-db), a message is published on the interprocess message bus to inform subscribed clients that something has changed in the domain model. The **interprocess message bus** is part of the CQRS-model. |

#### CertificateItemListViewEventHandler

|  |
| --- |
|  |
| A similar event hander is used for the second projection of the read-model, which is specific for list-views. |

### The process of Querying the “Read-Database-Projections”

Once the read-model is updated (as explained in §5.6.2.) and the notification message(s) are sent to the **message bus** then clients who subscribed to those notification message should be able to receive them and act upon them (by re-reading the read-model-projection for example).



[TODO: Add image/code sample which explains how clients can subscribe to notifications of the CQRS framework when something has changed.]

[TODO: Add image/code sample which explains how we can query the read-model projection databases.]

### Commands vs DTOs vs ViewModels

A **DTO[[7]](#footnote-7)** is a data contract made for the client and it’s structure should map as close as possible to the representations need in the user interface.

A Command is an instruction to do something, and we could be “tempted” to “leak” the command structure into the outside world (outside the application service layer of our ONION model), and so let the client (the UI) create the command and omit the mapping to the corresponding DTO, but this is not a good practice to-do so. Commands and DTOs are different things with different goals !

**Commands** are serializable method calls on the methods in the domain model, whereas **DTOs** are the data contracts between the Backend-API and the Presentation Layer of the calling Client.



[TODO: Add an image which explains difference between DTOs and Commands.]

**ViewModels** on the other hand are closely related with DTOs and may even be used interchangeably. This is certainly true for the “ReadViewModel” which is returned from the “thin data access layer” and part of the “Read-Side” of our CQRS application.

But … there is another type of View Model-type in our application, this one is mostly used in POST (Create) , PUT (Update) , PATCH (Partially Update) or DELETE (Delete) operations and is used as the “Data Context” of the View. This type of View Model is not only a getter-setter “property bag” but also contains behavior and informs the linked View when something has changed. This View Model is part of the MVVM-pattern which separates the “view” part of a data contract from the “behavioral part” and in some circumstances a DTO may be mapped to a View Model after it has been loaded from the data store (this is for example the case when applying Updates (PUT/PATCH)).



[TODO: Add an image which explains MVVM-pattern in the CQRS framework.]

|  |
| --- |
|  |
| A |

# How to build a production-ready Event Store



[TODO: Explain how we should create and maintain a production-ready event-store.]

# The Future use of CQRS at FAVV



[TODO: Explain how we should use the CQRS pattern in the future for both new and existing apps.]

1. Command Query Responsibility Segregation [↑](#footnote-ref-1)
2. There is also the notion of “Domain Services” these are another kind of services which offer services for the domain layer and have nothing to do with the Application Services. [↑](#footnote-ref-2)
3. Data Transfer Objects [↑](#footnote-ref-3)
4. Detailed description of the domain model or DDD (Domain-Driven-Design) more specifically is out of scope for this document. [↑](#footnote-ref-4)
5. Test Driven Development [↑](#footnote-ref-5)
6. https://enterprisecraftsmanship.com/2019/01/31/cqrs-commands-part-domain-model/ [↑](#footnote-ref-6)
7. Data Transfer Object [↑](#footnote-ref-7)