Clarified CQRS

Udi Dahan - The Software Simplist

After listening how the community has interpreted Command-Query Responsibility Segregation I think that the time has come for some clarification. Some have been tying it together to Event Sourcing. Most have been overlaying their previous layered architecture assumptions on it. Here I hope to identify CQRS itself, and describe in which places it can connect to other patterns.

Why CQRS

Before describing the details of CQRS we need to understand the two main driving forces behind it: collaboration and staleness.

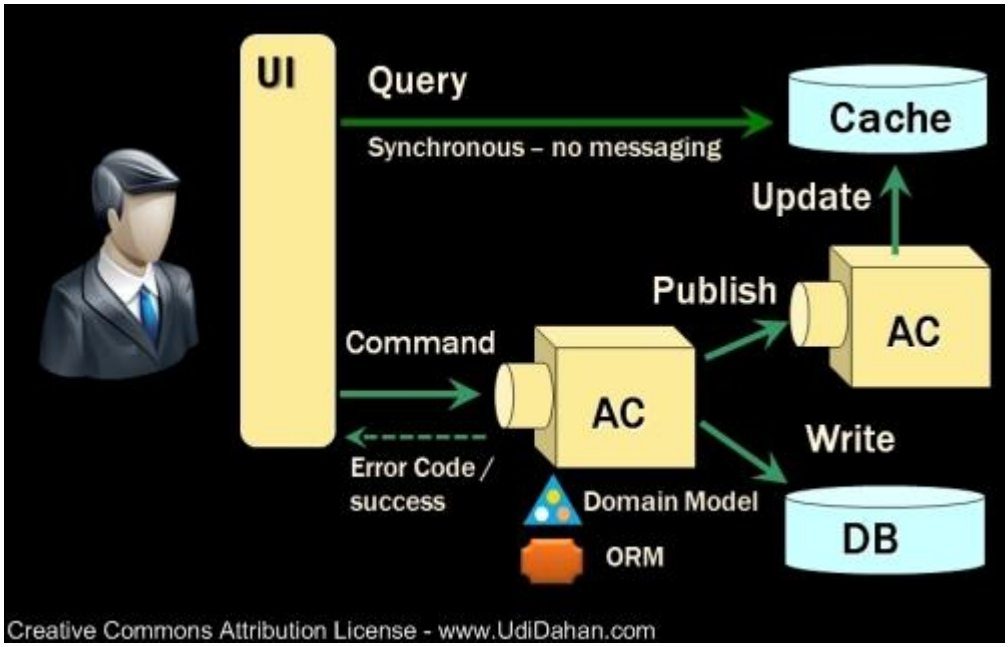
Collaboration refers to circumstances under which multiple actors will be using/modifying the same set of data – whether or not the intention of the actors is actually to collaborate with each other. There are often rules which indicate which user can perform which kind of modification and modifications that may have been acceptable in one case may not be acceptable in others. We'll give some examples shortly. Actors can be human like normal users, or automated like software.

Staleness refers to the fact that in a collaborative environment, once data has been shown to a user, that same data may have been changed by another actor – it is stale. Almost any system which makes use of a cache is serving stale data – often for performance reasons. What this means is that we cannot entirely trust our users decisions, as they could have been made based on out-of-date information.

Standard layered architectures don't explicitly deal with either of these issues. While putting everything in the same database may be one step in the direction of handling collaboration, staleness is usually exacerbated in those architectures by the use of caches as a performance-improving afterthought.

A picture for reference

I've given some talks about CQRS using this diagram to explain it:



The boxes named AC are Autonomous Components. We'll describe what makes them autonomous when discussing commands. But before we go into the complicated parts, let's start with queries:

Queries

If the data we're going to be showing users is stale anyway, is it really necessary to go to the master database and get it from there? Why transform those 3rd normal form structures to domain objects if we just want data – not any rule-preserving behaviors? Why transform those domain objects to DTOs to transfer them across a wire, and who said that wire has to be exactly there? Why transform those DTOs to view model objects?

In short, it looks like we're doing a heck of a lot of unnecessary work based on the assumption that reusing code that has already been written will be easier than just solving the problem at hand. Let's try a different approach:

How about we create an additional data store whose data can be a bit out of sync with the master database – I mean, the data we're showing the user is stale anyway, so why not reflect in the data store itself. We'll come up with an approach later to keep this data store more or less in sync.

Now, what would be the correct structure for this data store? How about just like the view model? One table for each view. Then our client could simply SELECT * FROM MyViewTable (or possibly pass in an ID

in a where clause), and bind the result to the screen. That would be just as simple as can be. You could wrap that up with a thin facade if you feel the need, or with stored procedures, or using AutoMapper which can simply map from a data reader to your view model class. The thing is that the view model structures are already wire-friendly, so you don't need to transform them to anything else.

You could even consider taking that data store and putting it in your web tier. It's just as secure as an in-memory cache in your web tier. Give your web servers SELECT only permissions on those tables and you should be fine.

Query Data Storage

While you can use a regular database as your query data store it isn't the only option. Consider that the query schema is in essence identical to your view model. You don't have any relationships between your various view model classes, so you shouldn't need any relationships between the tables in the query data store.

So do you actually need a *relational* database?

The answer is no, but for all practical purposes and due to organizational inertia, it is probably your best choice (for now).

Scaling Queries

Since your queries are now being performed off of a separate data store than your master database, and there is no assumption that the data that's being served is 100% up to date, you can easily add more instances of these stores without worrying that they don't contain the exact same data. The same mechanism that updates one instance can be used for many instances, as we'll see later.

This gives you cheap horizontal scaling for your queries. Also, since your not doing nearly as much transformation, the latency per query goes down as well. Simple code is fast code.

Data modifications

Since our users are making decisions based on stale data, we need to be more discerning about which things we let through. Here's a scenario explaining why:

Let's say we have a customer service representative who is one the phone with a customer. This user is looking at the customer's details on the screen and wants to make them a 'preferred' customer, as well as modifying their address, changing their title from Ms to Mrs, changing their last name, and indicating that they're now married. What the user doesn't know is that after opening the screen, an event arrived from the billing department indicating that this same customer doesn't pay their bills – they're delinquent. At this point, our user submits their changes.

Should we accept their changes?

Well, we should accept some of them, but not the change to 'preferred', since the customer is delinquent. But writing those kinds of checks is a pain – we need to do a diff on the data, infer what the changes mean, which ones are related to each other (name change, title change) and which are separate, identify which data to check against – not just compared to the data the user retrieved, but compared to the current state in the database, and then reject or accept.

Unfortunately for our users, we tend to reject the whole thing if any part of it is off. At that point, our users have to refresh their screen to get the up-to-date data, and retype in all the previous changes, hoping that this time we won't yell at them because of an optimistic concurrency conflict.

As we get larger entities with more fields on them, we also get more actors working with those same entities, and the higher the likelihood that something will touch some attribute of them at any given time, increasing the number of concurrency conflicts.

If only there was some way for our users to provide us with the right level of granularity and intent when modifying data. That's what commands are all about.

Commands

A core element of CQRS is rethinking the design of the user interface to enable us to capture our users' intent such that making a customer preferred is a different unit of work for the user than indicating that the customer has moved or that they've gotten married. Using an Excel-like UI for data changes doesn't capture intent, as we saw above.

We could even consider allowing our users to submit a new command even before they've received confirmation on the previous one. We could have a little widget on the side showing the user their pending commands, checking them off asynchronously as we receive confirmation from the server, or marking them with an X if they fail. The user could then double-click that failed task to find information about what happened.

Note that the client *sends* commands to the server – it doesn't publish them. Publishing is reserved for events which state a fact – that something has happened, and that the publisher has no concern about what receivers of that event do with it.

Commands and Validation

In thinking through what could make a command fail, one topic that comes up is validation. Validation is different from business rules in that it states a context-independent fact about a command. Either a command is valid, or it isn't. Business rules on the other hand are context dependent.

In the example we saw before, the data our customer service rep submitted was valid, it was only due to the billing event arriving earlier which required the command to be rejected. Had that billing event not arrived, the data would have been accepted.

Even though a command may be valid, there still may be reasons to reject it.

As such, validation can be performed on the client, checking that all fields required for that command are there, number and date ranges are OK, that kind of thing. The server would still validate all commands that arrive, not trusting clients to do the validation.

Rethinking UIs and commands in light of validation

The client can make of the query data store when validating commands. For example, before submitting a command that the customer has moved, we can check that the street name exists in the query data store.

At that point, we may rethink the UI and have an auto-completing text box for the street name, thus ensuring that the street name we'll pass in the command will be valid. But why not take things a step further? Why not pass in the street ID instead of its name? Have the command represent the street not as a string, but as an ID (int, guid, whatever).

On the server side, the only reason that such a command would fail would be due to concurrency – that someone had deleted that street and that that hadn't been reflected in the query store yet; a fairly exceptional set of circumstances.

Reasons valid commands fail and what to do about it

So we've got a well-behaved client that is sending valid commands, yet the server still decides to reject them. Often the circumstances for the rejection are related to other actors changing state relevant to the processing of that command.

In the CRM example above, it is only because the billing event arrived first. But "first" could be a millisecond before our command. What if our user pressed the button a millisecond earlier? Should that actually change the **business outcome**? Shouldn't we expect our system to behave the same when observed from the outside?

So, if the billing event arrived second, shouldn't that revert preferred customers to regular ones? Not only that, but shouldn't the customer be notified of this, like by sending them an email? In which case, why not have this be the behavior for the case where the billing event arrives first? And if we've already got a notification model set up, do we really need to return an error to the customer service rep? I mean, it's not like they can do anything about it **other than notifying the customer**.

So, if we're not returning errors to the client (who is already sending us valid commands), maybe all we need to do on the client when sending a command is to tell the user "thank you, you will receive confirmation via email shortly". We don't even need the UI widget showing pending commands.

Commands and Autonomy

What we see is that in this model, commands don't need to be processed immediately – they can be queued. How fast they get processed is a question of Service-Level Agreement (SLA) and not architecturally significant. This is one of the things that makes that node that processes commands autonomous from a runtime perspective – we don't require an always-on connection to the client.

Also, we shouldn't need to access the query store to process commands – any state that is needed should be managed by the autonomous component – that's part of the meaning of autonomy.

Another part is the issue of failed message processing due to the database being down or hitting a deadlock. There is no reason that such errors should be returned to the client – we can just rollback and try again. When an administrator brings the database back up, all the message waiting in the queue will then be processed successfully and our users receive confirmation.

The system as a whole is quite a bit more robust to any error conditions.

Also, since we don't have queries going through this database any more, the database itself is able to keep more rows/pages in memory which serve commands, improving performance. When both commands and queries were being served off of the same tables, the database server was always juggling rows between the two.

Autonomous Components

While in the picture above we see all commands going to the same AC, we could logically have each command processed by a different AC, each with its own queue. That would give us visibility into which queue was the longest, letting us see very easily which part of the system was the bottleneck. While this is interesting for developers, it is critical for system administrators.

Since commands wait in queues, we can now add more processing nodes behind those queues (using the distributor with NServiceBus) so that we're only scaling the part of the system that's slow. No need to waste servers on any other requests.

Service Layers

Our command processing objects in the various autonomous components actually make up our service layer. The reason you don't see this layer explicitly represented in CQRS is that it isn't really there, at least not as an identifiable logical collection of related objects – here's why:

In the layered architecture (AKA 3-Tier) approach, there is no statement about dependencies between objects within a layer, or rather it is implied to be allowed. However, when taking a command-oriented view on the service layer, what we see are objects handling different types of commands. Each

command is independent of the other, so why should we allow the objects which handle them to depend on each other?

Dependencies are things which should be avoided, unless there is good reason for them.

Keeping the command handling objects independent of each other will allow us to more easily version our system, one command at a time, not needing even to bring down the entire system, given that the new version is backwards compatible with the previous one.

Therefore, keep each command handler in its own VS project, or possibly even in its own solution, thus guiding developers away from introducing dependencies in the name of reuse (it's a fallacy). If you do decide **as a deployment concern**, that you want to put them all in the same process feeding off of the same queue, you can ILMerge those assemblies and host them together, but understand that you will be undoing much of the benefits of your autonomous components.

Whither the domain model?

Although in the diagram above you can see the domain model beside the command-processing autonomous components, it's actually an implementation detail. There is nothing that states that all commands *must* be processed by the same domain model. Arguably, you could have some commands be processed by transaction script, others using table module (AKA active record), as well as those using the domain model. Event-sourcing is another possible implementation.

Another thing to understand about the domain model is that it now isn't used to serve queries. So the question is, why do you need to have so many relationships between entities in your domain model?

(You may want to take a second to let that sink in.)

Do we really need a collection of orders on the customer entity? In what command would we need to navigate that collection? In fact, what kind of command would need *any* one-to-many relationship? And if that's the case for one-to-many, many-to-many would definitely be out as well. I mean, most commands only contain one or two IDs in them anyway.

Any aggregate operations that may have been calculated by looping over child entities could be pre-calculated and stored as properties on the parent entity. Following this process across all the entities in our domain would result in isolated entities needing nothing more than a couple of properties for the IDs of their related entities – "children" holding the parent ID, like in databases.

In this form, commands could be entirely processed by a single entity – viola, an aggregate root that is a consistency boundary.

Persistence for command processing

Given that the database used for command processing is not used for querying, and that most (if not all) commands contain the IDs of the rows they're going to affect, do we really need to have a column for every single domain object property? What if we just serialized the domain entity and put it into a single column, and had another column containing the ID? This sounds quite similar to key-value storage that is available in the various cloud providers. In which case, would you really need an object-relational mapper to persist to this kind of storage?

You could also pull out an additional property per piece of data where you'd want the "database" to enforce uniqueness.

I'm not suggesting that you do this in all cases – rather just trying to get you to rethink some basic assumptions.

Let me reiterate

How you process the commands is an implementation detail of CQRS.

Keeping the query store in sync

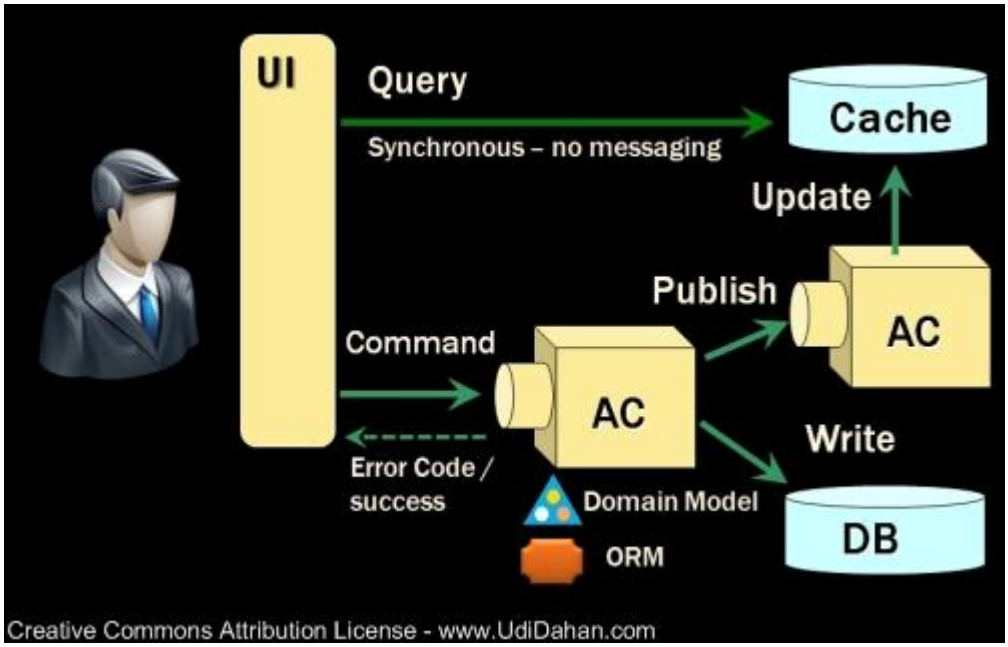
After the command-processing autonomous component has decided to accept a command, modifying its persistent store as needed, it publishes an event notifying the world about it. This event often is the "past tense" of the command submitted:

MakeCustomerPerferredCommand -> CustomerHasBeenMadePerferredEvent

The publishing of the event is done transactionally together with the processing of the command and the changes to its database. That way, any kind of failure on commit will result in the event not being sent. This is something that should be handled by default by your message bus, and if you're using MSMQ as your underlying transport, requires the use of transactional queues.

The autonomous component which processes those events and updates the query data store is fairly simple, translating from the event structure to the persistent view model structure. I suggest having an event handler per view model class (AKA per table).

Here's the picture of all the pieces again:



Bounded Contexts

While CQRS touches on many pieces of software architecture, it is still not at the top of the food chain. CQRS if used is employed within a bounded context (DDD) or a business component (SOA) – a cohesive piece of the problem domain. The events published by one BC are subscribed to by other BCs, each updating their query and command data stores as needed.

UI's from the CQRS found in each BC can be "mashed up" in a single application, providing users a single composite view on all parts of the problem domain. Composite UI frameworks are very useful for these cases.

Summary

CQRS is about coming up with an appropriate architecture for multi-user collaborative applications. It explicitly takes into account factors like data staleness and volatility and exploits those characteristics for creating simpler and more scalable constructs.

One cannot truly enjoy the benefits of CQRS without considering the user-interface, making it capture user intent explicitly. When taking into account client-side validation, command structures may be somewhat adjusted. Thinking through the order in which commands and events are processed can lead to notification patterns which make returning errors unnecessary.

While the result of applying CQRS to a given project is a more maintainable and performant code base, this simplicity and scalability require understanding the detailed business requirements and are not the result of any technical "best practice". If anything, we can see a plethora of approaches to apparently similar problems being used together – data readers and domain models, one-way messaging and synchronous calls.

Although this article is over 3000 words, I know that it doesn't go into enough depth on the topic - it takes about 3 days out of the 5 of my Advanced Distributed Systems Design course to cover everything in enough depth. Still, I hope it has given you the understanding of why CQRS is the way it is and possibly opened your eyes to other ways of looking at the design of distributed systems.