Improve Scalability and Focus on Business Logic with CQRS in Application Architecture

by Benny Bottema - Monday, March 21, 2016

http://www.bennybottema.com/2016/03/21/improve-scalability-and-focus-on-business-logic-with-cqrs-in-application-architecture/

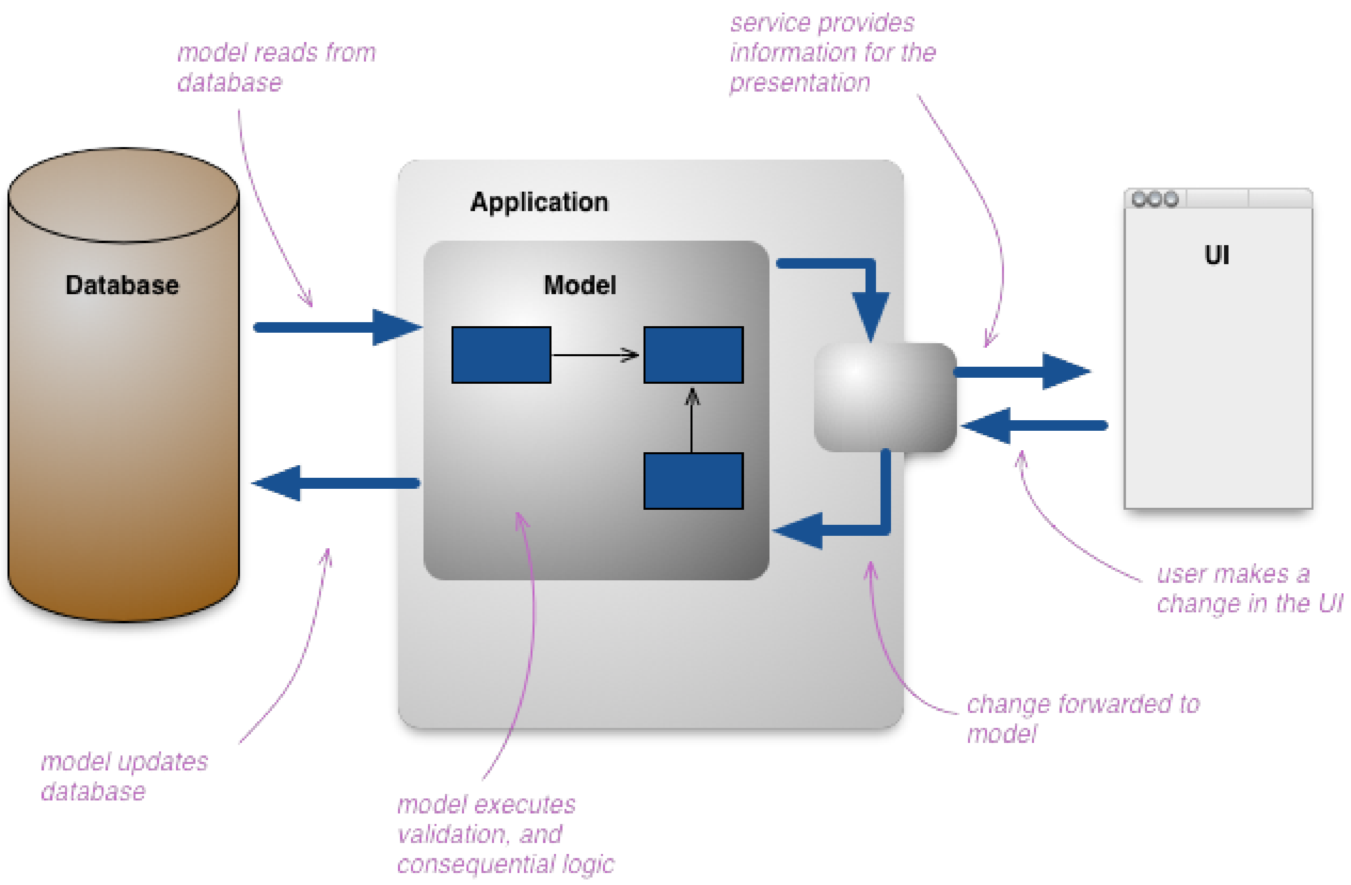
Contents

What is it

Command Query Responsibility Segregation (and Martin Fowler) or **CQRS** is a variation on the Command–query separation princple or CQS in short. CQS states that every method should be either an action or a query, but not both. It is a separation of responsibility where you clearly divide methods that change state from methods that report on state. CQRS is different in that it specializes in separating actions from queries for data CRUD manipulation specifically.

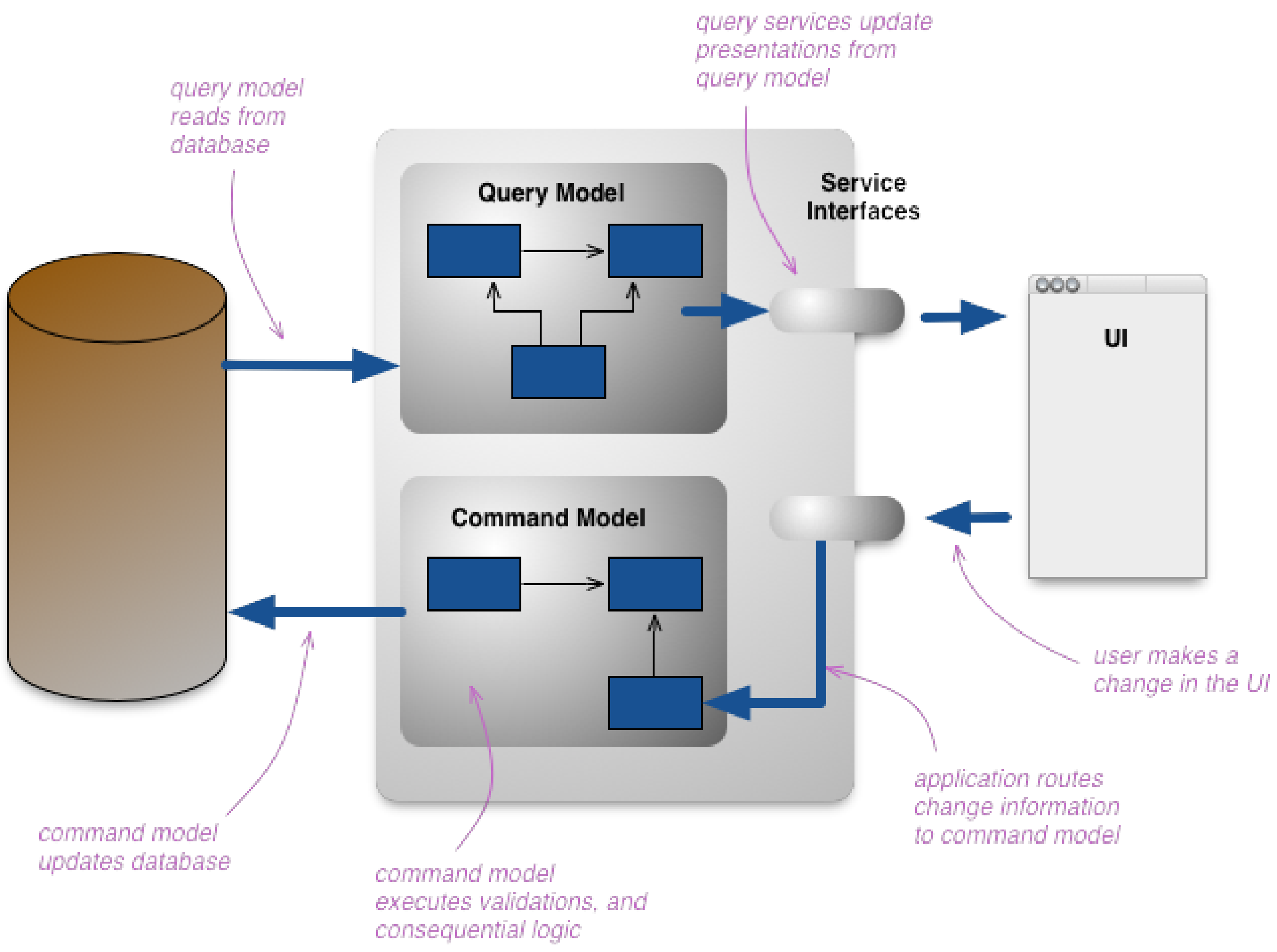
CQRS is an application architecture pattern or principle that is very simple in essence: split your read and write application paths.

Take a REST service for example, traditionally, you call the REST service, services goes to business layer, next integration layer and then all the way back again: synchronously the same path back. It also applies to asynchronous messaging systems such as JMS, because the application reading from a queue or topic and posting back a reply might still handle the business logic and persistence integration in a synchronous fashion using a single path.


model reads from database
service provides information for the presentation
Application
Model
Database
UI
user makes a change in the UI
change forwarded to model
model updates database
model executes validation, and consequential logic

Traditional CRUD model

With CQRS, the model is split up. In case of REST, there would be at least two services now: one for performing the action and one for performing the query of the result (unless you create some sort of synchronous read/writing blocking facade in front of the CQRS interface).


query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

CRUD with CQRS

Why would you need it

Why would you go out of your way to split *that* up? To quote Microsoft's excellent article on the subject:

> The most common business benefits that you might gain from applying the CQRS pattern are enhanced scalability, the simplification of a complex aspect of your domain, increased flexibility of your solution, and greater adaptability to changing business requirements.

Because you separate these concerns, it becomes easier to maintain and understand and you can scale these concerns separately as well.

When to use it

> Although we have outlined some of the reasons why you might decide to apply the CQRS pattern to some of the bounded contexts in your system, it is helpful to have some rules of thumb to help

> identify the bounded contexts that might benefit from applying the CQRS pattern.
>
> In general, applying the CQRS pattern may provide the most value in those bounded contexts that are collaborative, complex, include ever-changing business rules, and deliver a significant competitive advantage to the business. Analyzing the business requirements, building a useful model, expressing it in code, and implementing it using the CQRS pattern for such a bounded context all take time and cost money. You should expect this investment to pay dividends in the medium to long term. It is probably not worth making this investment if you don't expect to see returns such as increased adaptability and flexibility in the system, or reduced maintenance costs.

So you may have a complex domain model that is easier to understand if you separate read / write concerns. Or if you expect a lot of read / writes and you want to be more flexible in scaling these.

Martin Fowler posts a warning though:

> Despite these benefits, **you should be very cautious about using CQRS**. Many information systems fit well with the notion of an information base that is updated in the same way that it's read, adding CQRS to such a system can add significant complexity. I've certainly seen cases where it's made a significant drag on productivity, adding an unwarranted amount of risk to the project, even in the hands of a capable team. So while CQRS is a pattern that's good to have in the toolbox, beware that it is difficult to use well and you can easily chop off important bits if you mishandle it.

CQRS adds complexity to your application, but it removes a lot of complexity in the right situation.

Microsoft's article quote some concerns as well:

> Although you can list a number of clear benefits to adopting the CQRS pattern in specific scenarios, you may find it difficult in practice to convince your stakeholders that these benefits warrant the additional complexity of the solution.
>
> "In my experience, the most important disadvantage of using CQRS/event sourcing and DDD is the fear of change. This model is different from the well-known three-tier layered architecture that many of our stakeholders are accustomed to."
> —Pawe? Wilkosz (Customer Advisory Council)
>
> "The learning curve of developer teams is steep. Developers usually think in terms of relational

> database development. From my experience, the lack of business, and therefore domain rules and specifications, became the biggest hurdle.”
> —José Miguel Torres (Customer Advisory Council)
>
> Adopting the CQRS pattern helps you to focus on the business and build task-oriented UIs.

Things to watch out for

A couple of problems arise from this pattern:

Synchronous read + write behavior becomes complex

If need to perform an action *and* read back the result, then you end up working around the pattern to make ends meet… literally. In this case you need a blocking mechanism that initiates the action while asynchronously waits for the result to come in to report back. The real solution is to provide a published interface that doesn’t return data from actions. CQRS system is a natural fit in an EDA landscape.

Stale data

As reading and writing to a system using CQRS becomes an asynchronous process, the problem of stale data arises. Some sort of synchronizing behavior should be present to keep up with the ever changing state of a CQRS system. Be it a blocking write/read mechanism, pulling or a batch job.