# Application of the domain driven design in cloud native computing

## Abstract

The number of cloud-based systems using domain-driven design has been increasing in recent years. Every software project has a set of attributes, the most important of which are the amounts of data it operates, performance requirements, business logic and technical complexity. The techniques DDD proposes are useful for projects that have a lot of complex business rules. DDD won't help you if you work with big data, need to achieve outstanding performance, or program against hardware systems. The only purpose DDD concepts serve is to tackle business logic complexity.

## Introduction

## Patterns

DDD techniques that enable to find common patterns that can be reuse in cloud applications.

Command Query Separation (CQS) was originally defined by Bertrand Meyer in his book Object-Oriented Software Construction. The fundamental idea is that object's methods should be divided into two categories:  
- Queries: free of side effects, which return a result and do not change the state of the system;  
- Commands: Change the state of a system but do not return a value;

Command and Query Responsibility Segregation (CQRS) was introduced by Greg Young. It is based on the CQS principle. It can be considered an architectural pattern that separates the models for reading and writing data based on commands and events plus optionally on asynchronous messages. In many cases, CQRS is related to more advanced scenarios. A more advanced CQRS system might also use Event-Sourcing (ES), which stores events in the domain model instead of the current-state data (César de la Torre, Bill Wagner, Mike Rousos).

The separation aspect of CQRS is achieved by grouping query operations in one layer and commands in another layer. Each layer has its own data model (note that we say model, not necessarily a different database) and is built using its own combination of patterns and technologies. More importantly, the two layers can be within the same tier or microservice, as in the example (ordering microservice) used for this guide. Or they could be implemented on different microservices or processes so they can be optimized and scaled out separately without affecting one another.

CQRS means having two objects for a read/write operation where in other contexts there is one. There are reasons to have a denormalized reads database, which you can learn about in more advanced CQRS literature. But we are not using that approach here, where the goal is to have more flexibility in the queries instead of limiting the queries with constraints from DDD patterns like aggregates.

An example of this kind of service is the ordering microservice from the eShopOnContainers reference application. This service implements a microservice based on a simplified CQRS approach. It uses a single data source or database, but two logical models plus DDD patterns for the transactional domain, as shown in Figure 7-2.

Diagram showing a high level Simplified CQRS and DDD microservice.

Figure 7-2. Simplified CQRS- and DDD-based microservice

The Logical "Ordering" Microservice includes its Ordering database, which can be, but doesn't have to be, the same Docker host. Having the database in the same Docker host is good for development, but not for production.

The application layer can be the Web API itself. The important design aspect here is that the microservice has split the queries and ViewModels (data models especially created for the client applications) from the commands, domain model, and transactions following the CQRS pattern. This approach keeps the queries independent from restrictions and constraints coming from DDD patterns that only make sense for transactions and updates, as explained in later sections.