Table of Contents

Where and How Does DDD Fit? (15 pages)

> We won't be distracted by comparison if we are captivated with purpose.
>
> — Bob Goff

Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Over the years, we have accumulated a series of architecture styles to help us deal with system complexity. In this chapter we will examine how DDD compares with several of these architecture styles and how/where it fits in the overall scheme of things when crafting a software solution.

Architecture Styles

Layered Architecture

The layered architecture is one of the most common architecture styles where the solution is typically organized into four broad categories: **presentation**, **application**, **domain** and **persistence**. Each of the layers provides a solution to a particular concern it represents as shown here:

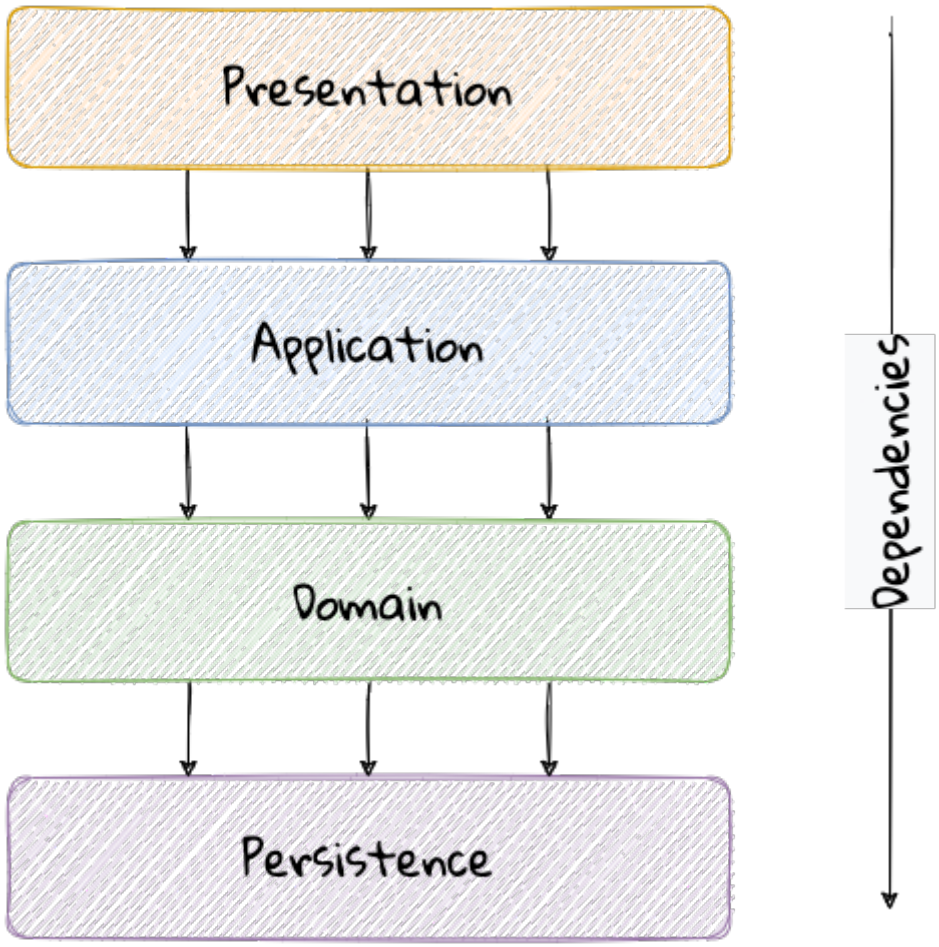


Figure 1. Essence of a layered architecture.

The main idea behind the layered architecture is a separation of concerns — where the dependencies between layers are unidirectional (from the top to the bottom). For example, the domain layer can depend on the persistence layer, not the other way round. In addition, any given layer typically accesses the layer immediately beneath it without bypassing layers in between. For example, the presentation layer may access the domain layer only through the application layer.

This structure enables looser coupling between layers and allows them to evolve independently of each other. The idea of the layered architecture fits very well with domain-driven design's tactical design elements as depicted here:

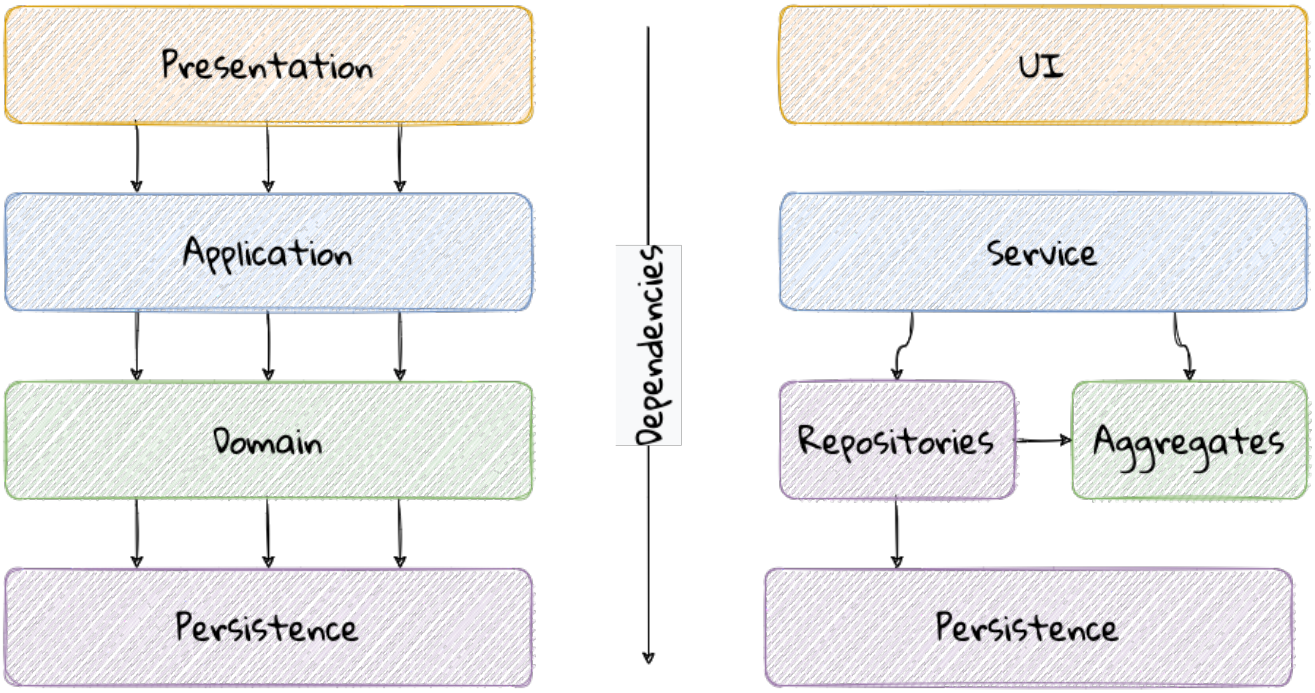


Figure 2. Layered architecture mapped to DDD's tactical design elements.

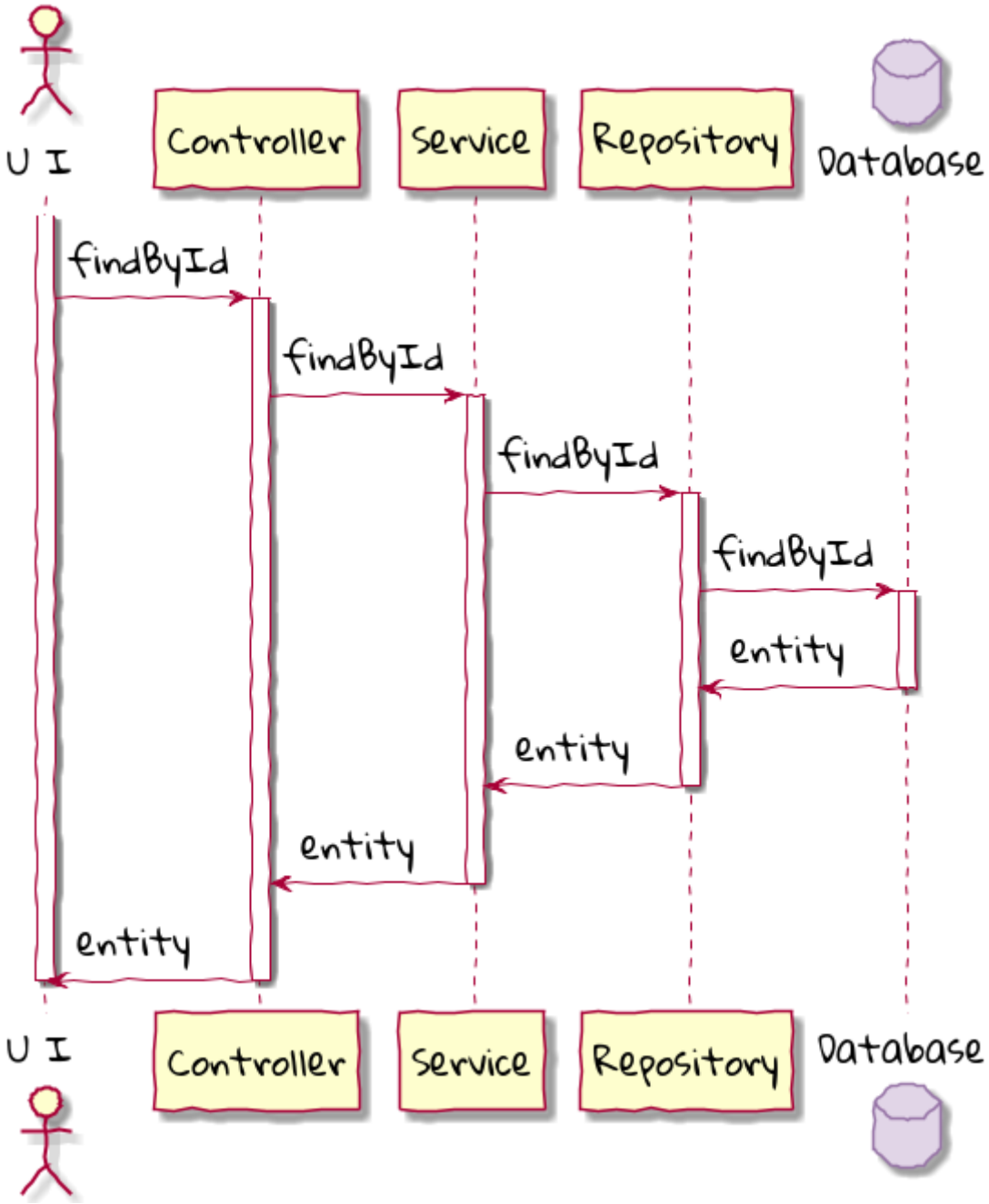
DDD actively promotes the use of a layered architecture, primarily because it makes it possible to focus on the domain layer in isolation of other concerns like how to information gets displayed, how end-to-end flows are managed, how data is stored and retrieved, etc. From that perspective, solutions that apply DDD tend to naturally be layered as well.

However, any architecture approach we choose comes with its set of tradeoffs and limitations. We discuss some of these here.

Considerations

Layer cake anti-pattern

Sticking to a fixed set of layers



Inter-layer translation:

Layer bypass

Onion Architecture

Hexagonal Architecture

Service Oriented Architecture

Microservice Architecture

Event-Driven Architecture (EDA)

Command Query Responsibility Segregation (CQRS)

In traditional applications, a single domain, data/persistence model is used to handle all kinds of operations. With CQRS, we create distinct models to handle updates (commands) and enquiries. This is depicted in the following diagram:

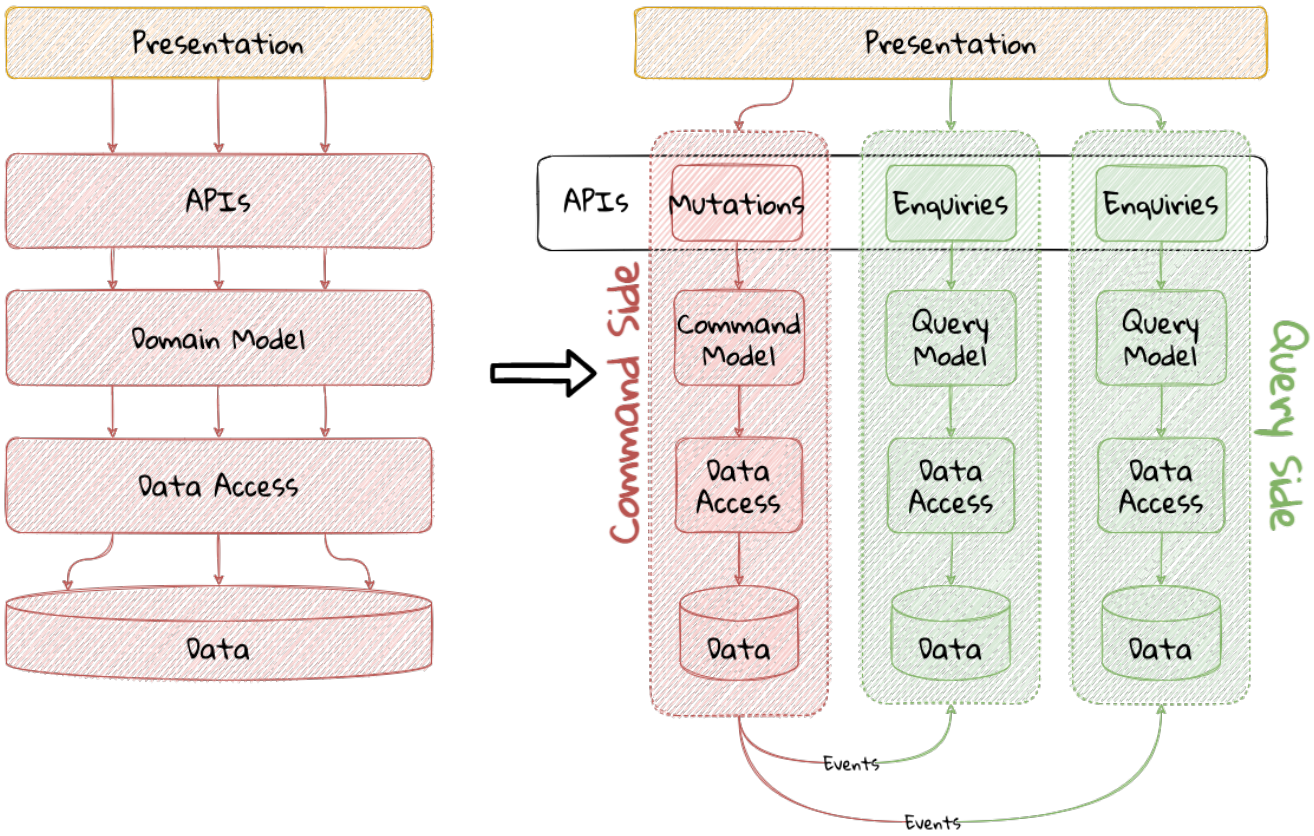


Figure 3. Traditional versus CQRS Architecture

We depict multiple query models above because it is possible (but not necessary) to create more than one query model, depending on the kinds of query use cases that need to be supported.

For this to work predictably, the query model(s) need to be kept in sync with the write models (we will examine some of the techniques to do that in detail later.

When to use CQRS?

The traditional, single-model approach works well for simple, CRUD-style applications, but starts to become unwieldy for more complex scenarios. We discuss some of these scenarios below:

- **Volume imbalance between read and writes**: In most systems, read operations often outnumber write operations by significant orders of magnitude. For example, consider the number of times a trader checks stock prices vs. the number of times they actually transact (buy or sell stock trades). It is also usually true that write operations are the ones that make businesses money. Having a single model for both reads and writes in a system with a majority

of read operations can overwhelm a system to an extent where write performance can start getting affected.

- **Need for multiple read representations**: When working with relatively complex systems, it is not uncommon to require more than one representation of the same data. For example, when looking at personal health data, one may want to look at a daily, weekly, monthly view. While these views can be computed on the fly from the *raw* data, each transformation (aggregation, summarization, etc.) adds to the cognitive load on the system. Several times, it is not possible to predict ahead of time, the nature of these requirements. By extension, it is not feasible to design a single canonical model that can provide answers to all these requirements. Creating domain models specifically designed to meet a focused set of requirements can be much easier.
- **Different security requirements**: Managing authorization and access requirements to data/APIs when working a single model can start to become cumbersome. For example, higher levels of security may be desirable for debit operations in comparison to balance enquiries. Having distinct models can considerably ease the complexity in designing fine-grained authorization controls.
- **More uniform distribution of complexity**: Having a model dedicated to serve only command-side use cases means that they can now be focused towards solving a single concern. For query-side use cases, we create models as needed that are distinct from the command-side model. This helps spread complexity more uniformly over a larger surface area — as opposed to increasing the complexity on the single model that is used to serve all use cases. It is worth noting that the essence of domain-driven design is mainly to work effectively with complex software systems and CQRS fits well with this line of thinking.

Serverless architecture

Big ball of mud