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:

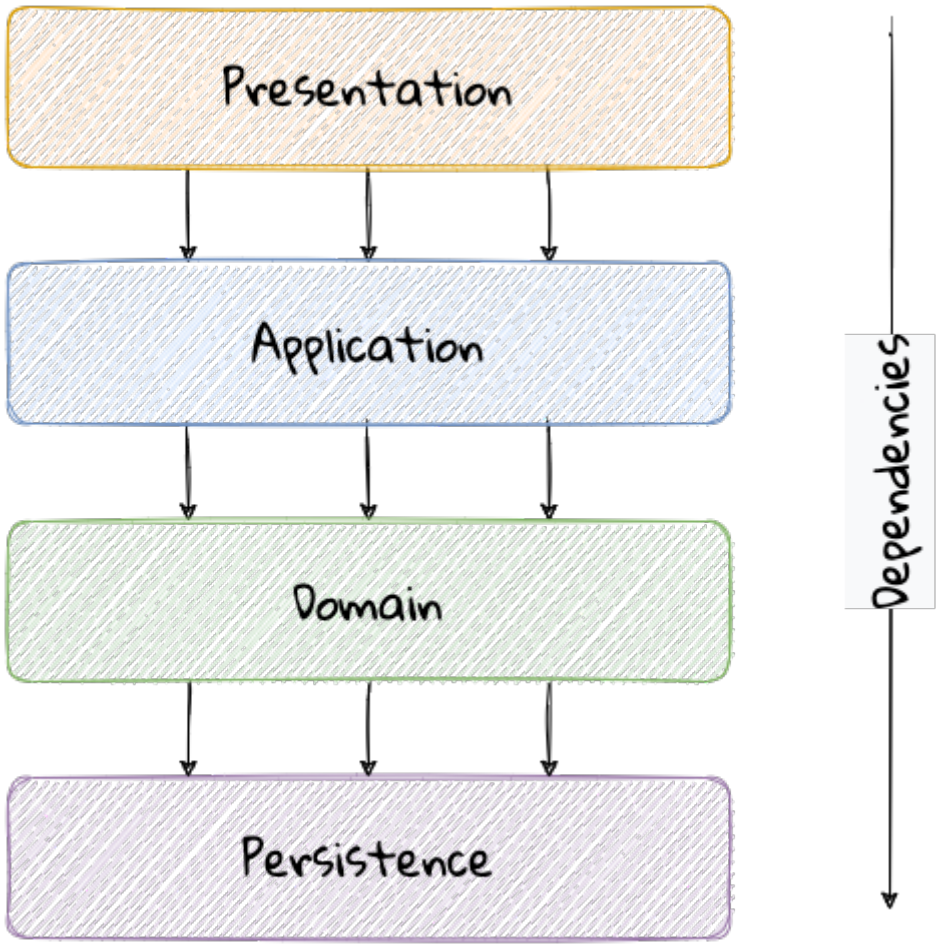

Figure 1. Essence of a layered architecture.

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:

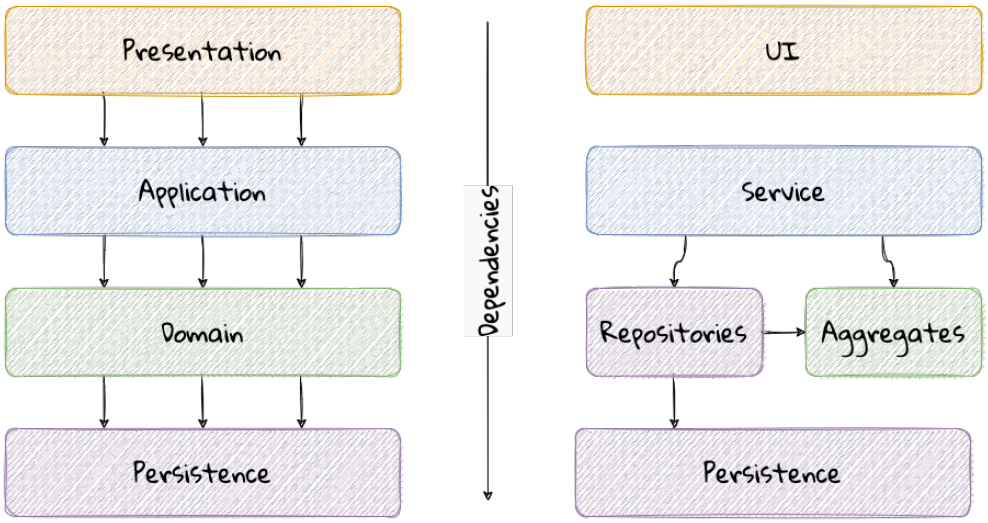

Figure 2. Layered architecture mapped to DDD's tactical design elements.

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 provides a level of isolation, but in simpler cases, it may prove overkill without adding any perceptible benefit other than adherence to an agreed on architectural guidelines. In the layer cake anti-pattern, each layer merely proxies the call to the layer beneath it without adding any value. The example below illustrates this scenario that is fairly common:

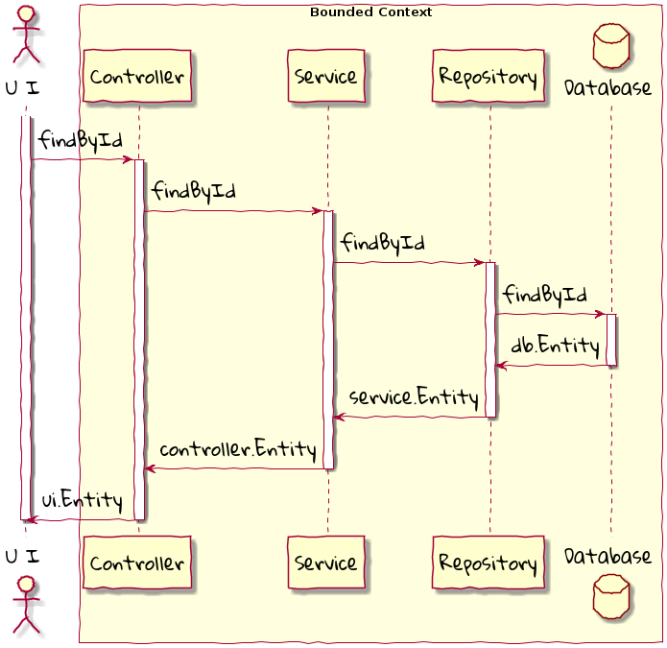

Figure 3. Example of the **layer cake** *anti-pattern to find an entity representation by ID*

Here the `findById` method is replicated in every layer and simply calls the method with the same name in the layer below with no additional logic. This introduces a level of accidental complexity to the solution. Some amount of redundancy in the layering may be unavoidable for the purposes of standardization. It may be best to re-examine the layering guidelines if the *layer cake* occurs prominently in the codebase.

Anemic translation

Another variation of the layer cake we see commonly is one where layers refuse to share input and output types in the name of higher isolation and looser coupling. This makes it necessary to perform translations at the boundary of each layer. If the objects being translated are more or less structurally identical, we have an *anemic translation*. Let's look at a variation of the `findById` example we discussed above.

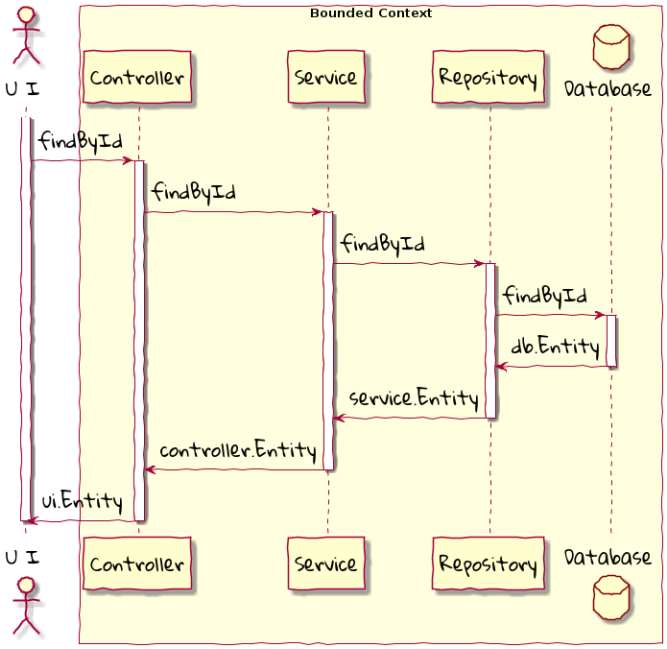

Figure 4. Example of the **anemic translation** *anti-pattern to find an entity representation by ID*

In this case, each layer defines a `Entity` type of its own, requiring a translation between types at each layer. To make matters worse, the structure of the `Entity` type may have seemingly minor variations (for example, `lastName` being referred to as `surname`). While such translations may be necessary across bounded contexts, teams should strive to avoid the need for variations in names and structures of the same concept within a single bounded context. The intentional use of the **ubiquitous language** helps avoid such scenarios.

Layer bypass

When working with a layered architecture, it is reasonable to start by being strict about layers only interacting with the layer immediately beneath it. As we have seen above, such rigid enforcements may lead to an intolerable degree of accidental complexity, especially when applied generically to a large number of use-cases. In such scenarios, it may be worth considering consciously allowing one or more layers to be bypassed. For example, the `controller` layer may be allowed to work directly with the `repository` without using the `service` layer.

This can be a slippery slope. To continue maintaining a level of sanity, teams should consider the

use of a lightweight architecture governance tool like **ArchUnit**[1] to make agreements explicit and afford quick feedback. A simple example of how to use ArchUnit for this purpose is shown here:

```
class LayeredArchitectureTests {
    @ArchTest
    static final ArchRule layer_dependencies_are_respected_with_exception =
layeredArchitecture()

            .layer("Controllers").definedBy("..controller..")
            .layer("Services").definedBy("..service..")
            .layer("Domain").definedBy("..domain..")
            .layer("Repository").definedBy("..repository..")

            .whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
            .whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
            .whereLayer("Domain").mayOnlyBeAccessedByLayers("Services", "Repository",
"Controllers")
            .whereLayer("Repository")
                .mayOnlyBeAccessedByLayers("Services", "Controllers"); ①
}
```

① The Repository layer can be accessed by both the Services and Controllers layers — effectively allowing Controllers to bypass the use of the Services layer.

Onion Architecture

Hexagonal Architecture

Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is an architectural style where software components expose (potentially) reusable functionality over standardized interfaces. The use of standardized interfaces (such as SOAP, REST, gRPC, etc. to name a few) enables easier interoperability when integrating heterogeneous solutions as shown here:

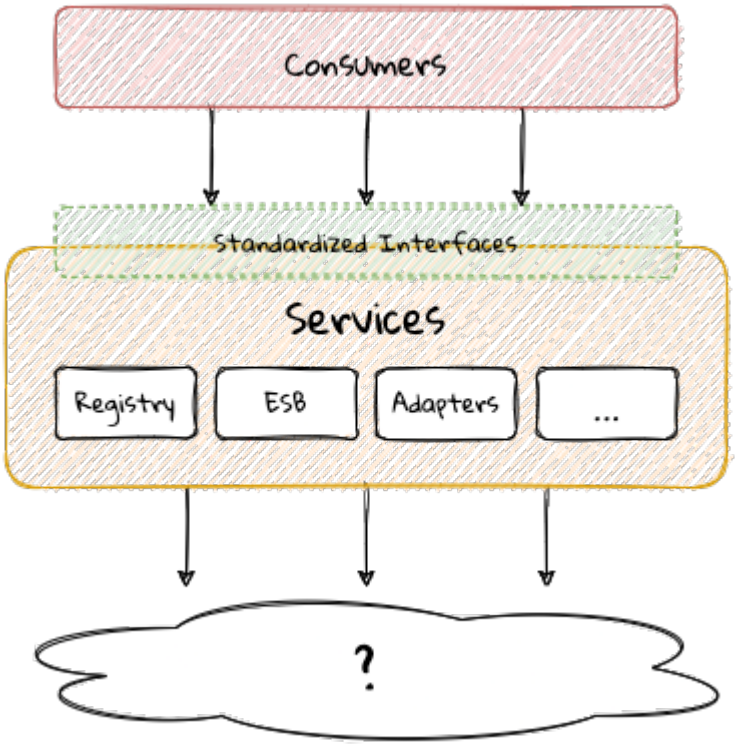

Figure 5. SOA: Expose reusable functionality over standard interfaces.

Previously, the use of non-standard, proprietary interfaces made this kind of integration a lot more challenging. For example, a retail bank may expose inter-account transfer functionality in the form of SOAP web services. While SOA prescribes exposing functionality over standardized interfaces, the focus is more on integrating heterogeneous applications than on implementing them.

At one of the banks we worked at, we exposed a set of over 500 service interfaces over SOAP. Under the covers, we implemented these services using EJB 2.x (a combination of stateless session beans and message-driven beans) hosted on a commercial J2EE application server which also did double duty as an enterprise service bus (ESB). These services largely delegated most if not all the logic to a set of underlying stored procedures within a single monolithic Oracle database using a canonical data model for the entire enterprise! To the outside world, these services were *location transparent*, stateless, *composable* and *discoverable*. Indeed, we advertised this implementation as an example of SOA, and it would be hard to argue that it was not.

This suite of services had evolved organically over the years with no explicit boundaries, concepts from various parts of the organization and generations of people mixed in, each adding their own interpretation of how business functionality needed to be implemented. In essence, the implementation resembled the dreaded big ball of mud which was extremely hard to enhance and maintain.

The intentions behind SOA are noble. However, the promises of reuse, loose coupling are hard to achieve in practice given the lack of concrete implementation guidance on component granularity. It is also true that SOA means many things[2] to different people. This ambiguity leads to most SOA implementations becoming complex, unmaintainable monoliths, centered around technology components like a service bus or the persistence store or both. This is where using DDD to solve a complex problem by breaking it down into subdomains and bounded contexts can be invaluable.

Microservice Architecture

In the last decade or so, microservices have gained quite a lot of popularity with lots of organizations wanting to adopt this style of architecture. In a lot of ways, microservices are an extension of service-oriented architectures — one where a lot of emphasis is placed on creating focused components that deal with doing a limited number of things and doing them right. Sam Newman, the author of the *Building Microservices* book defines microservices as *small*-sized, independently deployable components that maintain their own state and are **modeled around a business domain**. This affords benefits such as adopting a horses for courses approach when modeling solutions, limiting the blast radius, improved productivity and speed, autonomous cross-functional teams, etc. Microservices usually exist as a collective, working collaboratively to achieve the desired business outcomes, as depicted here:

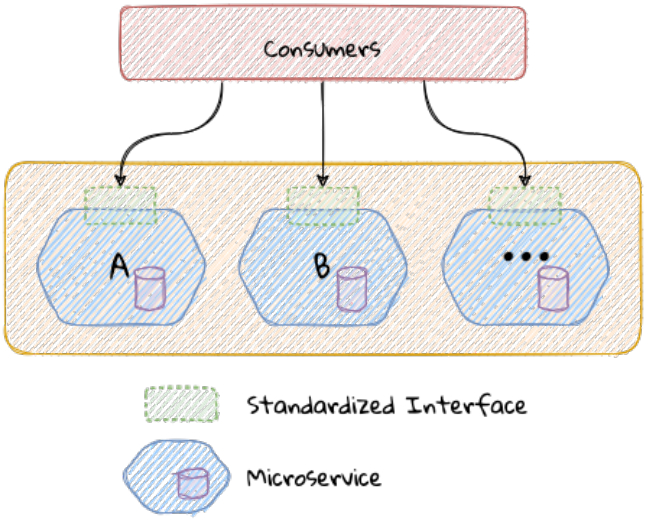

Figure 6. A microservices ecosystem

As we can see, SOA and microservices are very similar from the perspective of the consumers in that they access functionality through a set of standardized interfaces. The microservices approach is an evolution of SOA in that the focus now is on building smaller, self-sufficient, independently deployable components with the intent of avoiding single points of failure (like an enterprise database or service bus), which was fairly common with a number of SOA-based implementations.

While microservices have definitely helped, there still exists quite a lot of ambiguity when it comes to answering how big or small[3] a microservice should be. Indeed, a lot of teams seem to struggle to get this balance right resulting in a distributed monolith[4] — which in a lot of ways can be much worse than even the single process monolith from the SOA days. Again, applying the strategic design concepts of DDD can help create independent, loosely coupled components, making it an ideal companion for the microservices style of architecture.

Event-Driven Architecture (EDA)

Irrespective of the granularity of components (monolith or microservices or something in between), most non-trivial solutions have a boundary, beyond which there may be a need to communicate with external system(s). This communication usually happens through the exchange of messages between systems, causing them to become coupled with each other. Coupling comes in two broad flavors: *afferent* — who depends on you and *efferent* — who you depend on. Excessive

amounts of efferent coupling can make systems very brittle and hard to work with.

Event-driven systems enable authoring solutions that have a relatively low amount of efferent coupling by emitting events when they attain a certain state without caring about who consumes those events. In this regard, it is important to differentiate between message-driven and event-driven systems as mentioned in the *Reactive Manifesto*:

Message-driven versus Event-driven

> A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients.
>
> — Reactive Manifesto

In simpler terms, event-driven systems do not care who the downstream consumers are, whereas in a message-driven system that may not necessarily be true. When we say event-driven in the context of this book, we mean the former.

Typically, event-driven systems eliminate the need for point-to-point messaging with the ultimate consumers by making use of an intermediary infrastructure component usually known as a message broker, event bus, etc. This effectively reduces the efferent coupling from *n* consumers to 1. There are a few variations on how event-driven systems can be implemented. In the context of publishing events, Martin Fowler talks about two broad styles (among other things) — event notifications and event-carried state transfer in his What do you mean by "event-driven"?[5] article. One of the main trade-offs when building an event-driven system is to decide the amount of state (payload) that should be embedded in each event. It may be prudent to consider embedding just enough state indicating changes that occurred as a result of the emitted event to keep the various opposing forces such as producer scaling, encapsulation, consumer complexity, resiliency, etc. We will discuss the related implications in more detail when we cover implementing events in Chapter 5.

Domain-driven design is all about keeping complexity in check by creating these independent bounded contexts. However, independent does not mean isolated. Bounded contexts may still need to communicate with each other. One way to do that is through the use of a fundamental DDD building block — domain events. Event-driven architecture and DDD are thus complementary. It is typical to make use of an event-driven architecture to allow bounded contexts to communicate while continuing to loosely coupled with each other.

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:

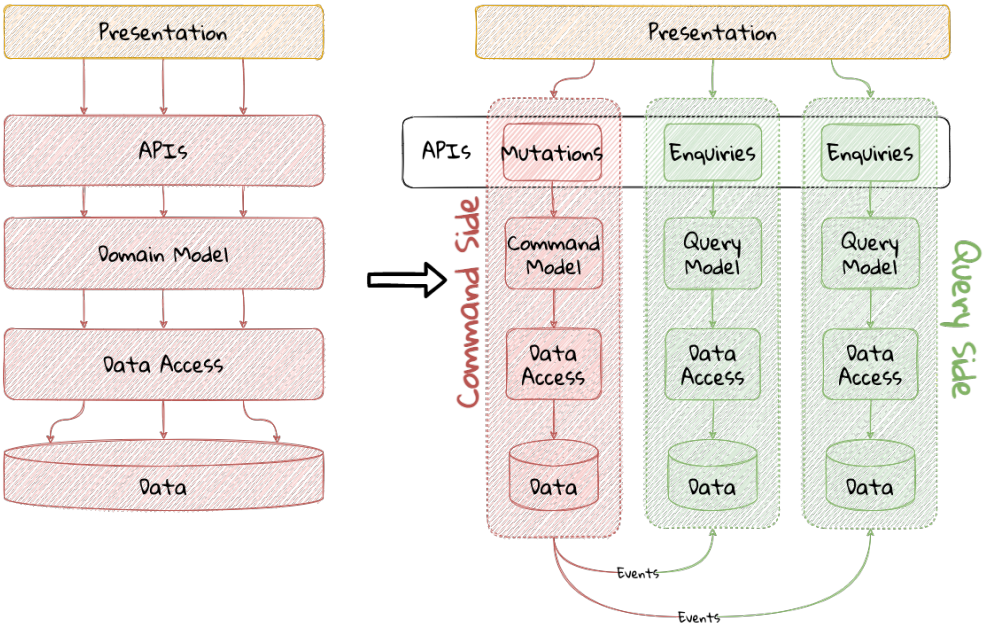

Figure 7. Traditional versus CQRS Architecture

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.

When working with a CQRS based architecture, choosing the persistence mechanism for the command side is a key decision. When working in conjunction with an event-driven architecture, one could choose to persist aggregates as a series of events (ordered in the sequence of their occurrence). This style of persistence is known as event sourcing. We will cover this in more detail in Chapter 5 in the section on event-sourced aggregates.

Serverless Architecture

Serverless architecture is an approach to software design that allows developers to build and run services without having to manage the underlying infrastructure. The advent of AWS Lambda service has popularized this style of architecture, although several other services (like S3 and DynamoDB for persistence, SNS for notifications, SQS for message queuing etc.) have existed long before Lambda was launched. While AWS Lambda provided a compute solution in the form of Functions-as-a-Service (FaaS), these other services are just as essential, if not more, in order to benefit from the serverless paradigm.

In conventional DDD, bounded contexts are formed by grouping related operations around an aggregate, which then informs how the solution is deployed as a unit — usually within th confines of a single process. With the serverless paradigm, each operation (task) is required to be deployed as an independent unit of its own as distributed components. This requires that we look at how we model aggregates and bounded contexts differently — now centered around individual tasks as opposed to a group of related tasks.

Does that mean that the principles of DDD to arrive at a solution do not apply anymore? While serverless introduces an additional dimension of having to treat finely-grained deployable units as first-class citizens in the modeling process, the overall process of applying DDD's strategic and tactical design continue to apply. We will examine this in more detail in Chapter 12 when we refactor the solution we build throughout this book to employ a serverless approach.

Big ball of mud

Thus far, we have examined a catalog of named architecture styles along with their pitfalls and how applying DDD can help alleviate them. On the other extreme, we may encounter solutions that lack a perceivable architecture, infamously termed as the *big ball of mud*.

> A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well-defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.
>
> — Brian Foote and Joseph Yoder

Although Foote and Yoder advise avoiding this style of architecture at all costs, software systems that resemble the big ball of mud continue to be a day-to-day inevitability for a lot of us. DDD provides a set of techniques to help deal with and recover from these near-hopeless situations in a pragmatic manner without potentially having to adopt a big bang approach. Indeed, the focus of this book is to apply these principles to prevent or at least delay further devolution towards the big ball of mud.

[1] https://www.archunit.org/

[2] https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html

[3] https://martinfowler.com/articles/microservices.html#HowBigIsAMicroservice

[4] https://www.infoq.com/news/2016/02/services-distributed-monolith/

[5] https://martinfowler.com/articles/201701-event-driven.html