



# *Domain Driven Microservices on AWS in Practice*

---

Go from DDD novice to actually understanding  
a real, modern application built with DDD in mind.

**Mikael Vesavuori**





# **Domain Driven Microservices on AWS in Practice**

Mikael Vesavuori

2024

# Contents

Copyright . . . . .	6
Found anything wrong? . . . . .	7
Introduction . . . . .	8
What you'll learn and do . . . . .	9
Why I'm writing this book . . . . .	9
Out of scope . . . . .	10
Audience . . . . .	10
Assumptions . . . . .	11
Structure . . . . .	11
Learning goals . . . . .	12
Note about the reality of writing about something while doing the work at the same time . . . . .	13
On design . . . . .	15
Design does not equal BUFD (Big Up-Front Design) . . . . .	17
You have to design more to be better at it . . . . .	19
DDD Lightning Tour . . . . .	20
What is DDD? . . . . .	22
The "domain" . . . . .	22
The "model" . . . . .	23
The patterns of DDD . . . . .	24
Why DDD and microservices are a match . . . . .	27
How do domains communicate? . . . . .	28
Is DDD still relevant? . . . . .	29
Reasonable objections . . . . .	31
In closing . . . . .	33
Putting DDD in the serverless context . . . . .	34
Not just back-end engineers work with servers these days . . . . .	34
There's a pretty big skills shortage . . . . .	36
You might need to learn a lot of things anew . . . . .	37
Serverless functions "identity crisis" . . . . .	37



---

Project resources . . . . .	40
Getting started . . . . .	41
Scenario . . . . .	42
Functional requirements . . . . .	42
Non-functional requirements . . . . .	42
Integration . . . . .	43
Delivery . . . . .	43
How to follow along . . . . .	44
Structure . . . . .	44
Data modeling . . . . .	45
Commands . . . . .	45
Prerequisites . . . . .	46
Doodling, drawing, whiteboarding flows and concepts . . . . .	47
Strategic DDD . . . . .	50
The domain(s) . . . . .	51
Terminology and language . . . . .	54
Documenting the ubiquitous language . . . . .	56
Delivery . . . . .	58
Setting boundaries . . . . .	60
Core principles . . . . .	61
Bounded Contexts in our example code . . . . .	62
Subdomain or Bounded Context? . . . . .	63
Context maps and relationships . . . . .	66
In summary . . . . .	67
EventStorming . . . . .	68
My solution . . . . .	70
Groundwork . . . . .	72
The logical business flow . . . . .	73
Deciding on a Cloud Architecture . . . . .	75
Going serverless-first . . . . .	75
Databases . . . . .	77
Compute . . . . .	78
Eventing . . . . .	78
API . . . . .	79
Security . . . . .	81
Technical boilerplating . . . . .	82
Lambda handler . . . . .	84



---

Handling the API/event input . . . . .	86
The Data Transfer Object . . . . .	91
Data vs behavior and JavaScript . . . . .	93
Error handling . . . . .	95
Testing . . . . .	97
Positive tests . . . . .	97
Negative tests . . . . .	99
In closing . . . . .	100
Lambda authorizer . . . . .	101
API schema . . . . .	106
Choosing AsyncAPI . . . . .	106
Writing the schema . . . . .	107
Tactical DDD . . . . .	115
Modules . . . . .	116
Demystifying Modules . . . . .	117
Structuring for a Module pattern . . . . .	118
High-level project organization . . . . .	120
Using Clean Architecture as our foundation . . . . .	120
Domain driven Lambdas with a use case approach . . . . .	126
How do we size and relate microservices in serverless DDD? . . . . .	128
Clean architecture-style use cases . . . . .	133
Create slots . . . . .	138
Updating the Display projection . . . . .	139
Get slots . . . . .	141
Reserve a slot . . . . .	143
Unattend no-shows . . . . .	145
Check in . . . . .	147
Check out . . . . .	149
Cancel a slot . . . . .	151
Open slot . . . . .	153
Close slots . . . . .	155
Factories . . . . .	157
Examples of the pattern . . . . .	159
Repositories . . . . .	161
Why Repositories? . . . . .	162
How Repositories are used in the project . . . . .	163
Services . . . . .	171



---

Services in the DDD hierarchy . . . . .	172
Application Services or use cases? . . . . .	174
An application service example . . . . .	175
Domain Services . . . . .	177
Entities . . . . .	185
Splitting data and behavior leads to unmaintainable code . . . . .	187
Rich domain models . . . . .	189
Invariants . . . . .	191
Before we move on . . . . .	191
The Slot entity . . . . .	192
Aggregates . . . . .	208
Do we have Aggregates in the example project? . . . . .	212
How large is an Aggregate? . . . . .	213
What we mean with transactions . . . . .	213
Our domain service as a stand-in . . . . .	215
Examples from our project . . . . .	217
Value Objects . . . . .	222
Creating a TimeSlot as a Value Object . . . . .	224
Domain Events . . . . .	227
Naming, exactness and uniqueness of an event . . . . .	230
Persisting events . . . . .	231
Resiliency . . . . .	232
Emitting events . . . . .	233
The events . . . . .	238
How to go further . . . . .	246
Practice the craft . . . . .	247
Learn to communicate better . . . . .	248
Design more . . . . .	252
Know your sh*t . . . . .	253
References and resources . . . . .	254
Clean Architecture . . . . .	254
DDD . . . . .	254
EventStorming and event modeling . . . . .	255
Event-Driven Architecture . . . . .	255
Resources . . . . .	256
Thank you for reading this book . . . . .	257



## Copyright

© 2024 Mikael Vesavuori. All rights reserved.

No part of this book may be reproduced, distributed, or transmitted in any form or by any means, including but not limited to physical copies, photocopying, recording, electronic books (eBooks), PDFs, digital downloads, or any other electronic or mechanical methods, without the prior written permission of the author, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law, such as under “fair use”.

Cover image adapts photographic material shared by Cassi Josh on Unsplash. All relevant ownership of the original photograph remains with Cassi Josh.

Published by Mikael Vesavuori on LeanPub and Gumroad.



## Found anything wrong?

Writing technical books is challenging. While concepts and ideas may remain relevant for years, practical examples that rely on ever-changing technologies can become outdated quickly.

If you find anything incorrect, not working, or otherwise unusual, I'd greatly appreciate your feedback. I'll do my best to incorporate updates as soon as possible.

Find contact details on my website, [mikaelvesavuori.se](http://mikaelvesavuori.se).



# Introduction

*Welcome to the fast track, taking you from a DDD novice to actually understanding a real, modern application built with it in mind.*

This online book aims to explain and demonstrate how one might practically apply [Domain Driven Design](#) (DDD) to a room-booking application using a microservices pattern that we will build on [Amazon Web Services \(AWS\)](#).

## Information

The application, consisting of four distinct services divided into three domains, is provided in a reference shape and throughout the book we will refer to this example.

**Domain Driven Design** is a software design approach that has been around for almost 20 years, gaining massive renewed attention with the surge of microservices and related technologies in the last decade. DDD focuses on the logical, semantic, and structural sides of software development (heavily leaning on the business end) more than on prescriptive implementation, though it does also provide several good design patterns. DDD is an ideal approach for complex and/or enterprise-leaning software but can be cumbersome for small, self-contained projects. While our own application will not be deeply complex, it is sufficiently advanced to warrant a structured, domain-oriented approach.

**Microservice architecture** is a software architecture style that emphasizes *small, well-defined, loosely coupled services that interact together* over singular, monolithic applications. Technologies like Kubernetes and serverless functions have accelerated the uptake of this style, as it may be hard and potentially expensive in older, non-cloud computing paradigms. Microservices are a good fit with our technology stack as well as helping us enforce clearer boundaries between system components as per DDD.

There is, to be frank, nothing extravagantly special in terms of reasons for choosing **Amazon Web Services**. While the services and their particulars are certainly unique to AWS, there is nothing in the overall architecture that cannot be ported over to Azure, Google Cloud Platform, or other clouds. The specific path we will be taking is centered on [serverless technologies](#) and a [cloud-native way of thinking](#). The AWS platform provides excel-



lent paths for us to build and run the application in this manner.

Throughout the book, several other concepts and methodologies will be introduced to further extend the approach and implementation details of our project.

## What you'll learn and do

The project will demonstrate rich and powerful patterns—binding together serverless, microservices, DDD, Clean Architecture, TypeScript, and more—to present these in a digestible, actionable way. I will attempt to hold back on some “mannerisms” and complexities in the DDD and tech world that may detract from the core lessons I have to impart.

After having read this book—and coded alongside the provided project, if you want—you will have a **hands-on feeling of how a project can go from a scenario (an “ask”) to something that represents a well-structured, domain-oriented application.**

Just like [Vaughn Vernon's \*Implementing Domain-Driven Design\*](#) (2013) drove the OG, [Eric Evans's \*Domain-Driven Design: Tackling Complexity in the Heart of Software\*](#) (2003), to an even more practical level, my intent here is to maximize that kind of push towards an honest-to-God practical reference example. You will be presented with lightweight descriptions of most of the core concepts while fast-tracking (and back-tracking) all the steps I did myself to create our demo application.

I will also of course share resources and references throughout in case you are interested to go deeper into any particular aspect we are touching on.

## Why I'm writing this book

I'm an architect and developer who feels that DDD made me a better professional. I'm neither an “expert” nor a “leading voice” on this—I'm just a guy working in software, helping my organization and our teams to do the best possible work we can. But I would be remiss if I couldn't share my passion for software engineering and how I—and perhaps you too—can connect the dots between the cloud, DDD, and microservices.

Part of the rationale for undertaking this project is because through the years in which



I've encountered, learned, used, and encouraged DDD and Agile design, I have never really had a “full-size” springboard to exemplify just how to do it. Also, because there are many components to this whole package, it's easy to kludge everything and spend too much time on details—techy stuff, sometimes the theory, or whatever else that felt most important that particular day.

Perhaps most importantly, I find it highly relevant in our day and age where there seems sometimes to exist a conflict between developer empowerment (such as expressed through DevOps and Agile) with the very concept of “design” altogether. More on that later.

## Out of scope

This is a book with a broad range. There is an incredible amount of specialist literature and resources to lean into for a multitude of areas that we'll raise here; I'm doing my best to be transparent and link to them. Don't necessarily expect all answers to be given here, you'll probably have better luck just continuing your research elsewhere.

The project itself won't be perfect either. There are always things to improve (or gold-plate, if you wish) and more advanced patterns to bring in. That's OK. However, I feel confident in that the project should be well and good enough to demonstrate with clarity the primary concepts: DDD, microservices, and running it in AWS.

## Audience

I am writing this for several intended audiences:

- **The curious software developer:** Being an open book on the internet, I assume there are throngs of software developers out there who might be interested in learning what is taught here, if not by me, then at least the themes addressed here.
- **Colleagues:** Because we talk a lot about these things and since nothing beats actually showing what we mean.
- **Myself:** As a way to learn more and hone my didactic, communicative, and technical skills.

## Assumptions

My mission here is to bring together all the things that make modern software what it is: DevOps, development, architecture, and an understanding of the cloud and how we deploy and run things there.

You can call yourself whatever or work as whatever, but you will most likely be some kind of developer or engineer, or architect.

You should be familiar with AWS and development in general, and if possible, it is ideal if you know TypeScript. I will not assume that you are a certified AWS professional or an architect.

## Structure

You will find the `Introduction` section first, as expected, where (beyond this page) you'll also get a smooth ride into some of the questions and concerns I've had, and that you may have as well when it comes to the cloud, DDD, microservices, and how they mix. This is followed by some other meta-materials.

The primary book contents start with the `Scenario` section. Here we will look at the requirements set out for our application, as well as inspect the coded example to fit with those requirements, and how you can use it locally on your machine.

Next up is `Strategic DDD`. This section captures many of the core concepts of this first, less implementation-oriented, phase when we build domain-driven systems. You'll start seeing how we can describe a Ubiquitous Language, how we do domain modeling and context mapping, how we use EventStorming to better understand our system-to-be and its flows, and more.

At this point, we may be rearing to go, but we'll first stop by the `Groundwork` section in which we will approach our cloud architecture and basic technical patterns. Since this isn't strictly related to the DDD parts of our implementation, we can instead zoom in on how to set up a cloud-native, serverless frame or boilerplate for our microservices that can be evolved with business logic and Clean Architecture patterns further down the line.

With that part done, finally, we get to go knee-deep in detail in `Tactical DDD`. In this section we'll get to use the vital patterns that separate a decent application from a great



one, seeing for example how we can write Aggregates that make sense in microservices.

Throughout the book, I'll do my best to reference good materials, either online or in literature. The last section, References and resources, does what it sounds like: Providing you with a compilation of further research.

## Learning goals

First and foremost of the learning goals is of course to make you understand how to practically produce a concrete, minimal, but the useful implementation of AWS-based serverless microservices using Domain Driven Design and Clean Architecture and develop it using TypeScript.

## New skills to bring home

You will be able to practically address questions like:

- **How to do strategic DDD** by defining a domain model, outlining bounded contexts, making context maps, specifying a ubiquitous language and more
- **A basic understanding of mapping Bounded Contexts and subdomains** into a serverless microservice implementation
- **Deciding on the type of relationship and integrations between domains and contexts**, including Published Language, Anti-Corruption Layer, Shared Kernel, Open Host Service, Customer/Supplier, Conformist, and Partnership
- **Implementing tactical DDD** with Value Objects, Entities, Aggregates, (application and domain) Services, Factories, Repositories, and Domain Events
- **Ease and improve on DDD with a module structure inspired by Clean Architecture**
- **Understand modern ways to work with events** using AWS EventBridge
- **Persisting data with Repositories and Aggregates**, and handling cross-context transactions

## Understanding problems in common solutions

You will understand how trivial or simplistic implementations can be harmful:

- Distributed monoliths
- Every function being its own bounded context (too large...)
- Microservices that are completely disassociated from contexts (too chatty...)
- Self-implementing an API Gateway (*can* be useful, but is most probably not)
- Unclear APIs, contracts, and events
- Single points of failure and not understanding the “Fallacies of Distributed Computing”

## Note about the reality of writing about something while doing the work at the same time

This is just a short disclaimer to be super-real about the fact that it is somewhat challenging to (on one’s own):

- Concoct (out of thin air) a business case regarding a, to be honest, fictional need;
- Lay out a workshop and learning plan,
- Build the system according to the best of my abilities...
- ...while still doing it (literally) by the book I am writing...
- ...and then to live to write about all of that!

Doing all of the parts means that I also get more or less “perfect information”, which is never the case in real life. We never know all there is to know and how best to approach it.

So with that said, reality in a real production context will pose other problems than those I had when making this material for you. What is hard on one’s own is stuff like having



no one to really ideate with, no one to do another part of the labor, no one to help you role-play business owner, etc. Working with others, however, will give you a somewhat more real problem set, such as discussing, arguing, teaching, and learning with others. Never forget that the social side is *big* for us as tech people.

I feel happy about what I *have* made though, and I hope at the end of it, that you feel the same!

## On design



Figure 1.1: Illustration from Undraw

**Design and architecture are effectively the same, at least in their intents. At some point, you won't have technical problems, only design problems relating to poor models, poor decisions, and poor structure.**

It's probably reasonable to believe most of us building software today began doing so by some combination of trial-and-error and following guided steps. As for me, who didn't graduate in Computer Science—but have in some way or another worked with computers since I was a kid—development is something that was learned on my own. In 1997, at the ripe age of 11, I would read any magazine that contained primitive HTML/web guides until the covers split. And oh boy did I spend time.

Like with any non-programming language, also in programming there is of course a need to understand the *syntax* (“grammar”) and *semantics* (“meaning”) of the language. To some extent, it makes sense to start here. But relatively obscured, in my experience, is the need to understand the *design* of programs.

Learning how to make something—anything, really—on a computer seems often to miss out on the higher-level aspects, such as *design*, i.e. “how do we put it all together in a good way?”. This part is dangerously close to a “trade secret”, and I've experienced this when studying anything from generative music, to painting and concept art, to—in our case—programming. Being new to the subject, we occupy so much of our time on the minor details, all-important though they seem at the time. Knee-deep in paracetamol

pills and questioning the decision to learn programming in the first place, of course, one is not in the ideal position to consider program design. Not “seeing the forest for the trees” is a fitting expression when we are in a learning mode.

### Danger

I think the comparison to *design* in (non-programming) language *might* work, but it’s worth treading with caution: We build programs that are logical constructs in a way regular languages are not.

If you are anything like me, your first (perhaps young) years were a confusing mess of trying to understand *why exactly* programs were structured the way they were, particularly when you began transitioning from single-file coding to “real” programs. I also remember from my own teaching days first-hand how students would be in equal measures entranced and dumbfounded by things even working at all, with what could as well look like a crappy stage magician’s trick and a small length of wire holding it together. “How the hell does this work?”, they would think (or sometimes even say). Curious magic indeed.

I live in the assumption that academically trained engineers will scoff at everything written so far, since they perhaps had to read books like [\*Design Patterns: Elements of Reusable Object-Oriented Software\*](#) (Gamma, Johnson, Vlissides, Helm 1994). Unfortunately, reading is a slow and brittle process, and simply reading does not guarantee understanding or adherence. In my practice and work experience, I find that typically the “academic” (good or bad) aspects of building software are seldom natural parts of the day-to-day conversation.

Perhaps of my personal traits, the design and structure parts have always been interesting, to the point of being my favorite parts of software engineering, and therefore concerns that have naturally occupied quite some time on my end. I took a huge step in my career when I decided to switch fully to formal software engineering in 2015. However, this was nerve-wracking because, as anyone who is self-aware at all will recognize, suddenly you have to deal with people who are skilled experts in the subject. You become very self-critical in approaching such a situation. Years later, for better or worse, I know what many know who have worked in the industry: That the software engineering field, and the people in it, are extremely diverse to the point of being hard to generalize in any



capacity whatsoever.

One thing I do want to generalize about is **the role of design**, or perhaps more fitting the lack of such a role. I've come to the opinion (understanding...?) that somewhat universally, developers/engineers spend a lot less time on design than they ought to do, given the benefits it would give them. A combination of attitudes seems to be pervasive, trying to fend away this reality:

- “Let the architect do it”, or
- “There’s no time and/or budget for design”, or
- “It’s Agile, man, f\*\*k design, it’s part of the code”, or
- Any other derivative of the same general attitude.

### **Danger**

No, all my experiences are not aligned with this, but there is some persistent truth to the above in my professional career. It’s more of a question of quantity of how apparent the attitude becomes rather than if the problem surfaces at all.

## **Design does not equal BUFD (Big Up-Front Design)**

One side of the word “design” that I feel to be misunderstood is around **how design relates to practices such as Agile** and its various forms—which is something every company and context I’ve been at has talked about, but not really delivered on—as well as to how the architect role fits in within modern software development, the DevOps paradigm, and being generally allergic to up-front planning.

It’s nice then to see that this is less a real issue (i.e. not being supported in the orthodox Agile mindset) than it is a perceived issue, perpetuated by misguided business analysts, project/product managers, other peripheral figures, and then the odd engineer here and there. We find evidence for this in how Agile framework co-founder Robert C. Martin tries to set the record straight in his [Clean Agile: Back to Basics](#) as well as in the humble

“[connect the penthouse with the engine room](#)” metaphor stated by Gregor Hohpe. They both express that *design* is something that must be done, Agile or not\*\*, and that modern circumstances do not have to pass on a golden key to some Ivory Tower dude to handle the design on their own.

## **Software design is not only the architect’s territory**

While many years ago I might have understood what an architect did, I personally gradually learned more and more on that side as I started building my own software, distributing open source, and leading technical teams as well as working on architecture proper.

Here too, as above, we see perhaps most clearly in Robert C. Martin’s books, that there is almost a reluctance to talk about architects and architecture and rather spend time on design as a primary concept.

With more complex software needs, more complex technology, and more intricate organizations, it isn’t strange to see that we now have a more multi-faceted range of architects operating today. Yet, this does not in any way relieve the burden (?) of design, especially on the system level, from the engineers.

## **Triplet of skills**

I’m conscious of the over-simplification that will shortly take place, but urge you to consider this point: **Without the basics you can’t do the work, but without design, it will never be good work.**

So:

- Knowledge of proper **syntax** is what, hopefully, makes you able to pocket a paycheck for your work in the first place.
- Knowledge of proper **semantics** is what makes you write good, clean code (Think Uncle Bob’s advice, etc.) on the *local* level, file-by-file.
- Knowledge of proper **design** is what allows you to build *complete* solutions that are competent and grow better over time.

This is a huge topic to unpack, with its layers of psychology, leadership, organization, engineering, and other sub-areas coming into play. Henceforth what I am interested in is stating that **design is important and it's your job as an engineer to handle it**. If you have an architect, then the conversation should likely hover around this topic. How you get to consistently conversing around this area is another problem I will not necessarily address.

## **You have to design more to be better at it**

Nothing has improved my own skills as much as setting up small projects with clear goals. Part of those goals should always be “What do I want to improve on, or learn?”

After having read this book, and maybe even trying to build the application on your own using the guidance here, take some time to reflect on what didn't make sense, what was hard, and what you feel you can do better. Can the language improve? Can the abstractions be better handled? Can you be more expressive with relations between systems?

Reading is not the end goal—designing and reflecting on the results of the process is. And I'm not the only one adamant in stating that domain-driven design is a solid foundation to build our design flow around, as well as informing how we practically develop our software as a team.



## DDD Lightning Tour



Figure 1.2: Illustration from Undraw

Domain Driven Design, DDD for short, was a game changer (and is still somewhat singular) in that it insists on the software not just the engineering part, but also how it logically connects the physical and very real business end to software to accurately represent those ideas. Therefore, DDD was from the start contingent on *language* as a primary tool to create cohesion and allow for expressive and rich modeling. It also came with many prescriptive ideas divided between the higher-level “strategic DDD” and the implementation patterns part of the “tactical DDD”.

Domain Driven Design has grown in the 20-odd years it’s been around to be a foundational part of modern software architecture and shaping the methodology with which many work in software. It seems to have been given an enormous upswing after the microservices pattern become more in vogue some 5-10 years ago.

For me personally, reading about Domain Driven Design—first through articles and then through Eric Evans’ “[blue book](#)” (2004) and Vaughn Vernon’s “[red book](#)” (2013)—made for an exciting summer some years back: It was really obvious (!) that we need to connect the “business” with the implementation. The explosive thing about the books, however, was that they went well and beyond the platitudes of the *statement* (as you read it in the last sentence) to actually detailing patterns, strategies, and ways to get there. And boy does their stories consume a lot of paper and reading time!



Figure 1.3: The three primary books on DDD plus Vlad Khononov’s modern, slimmer take on DDD from 2021.

I guess it’s fair to say that the “problem”, if one can call it that, is that both of the books are big. Like *really* big. I think they fall squarely into the lap of certain types of folks who still enjoy the intellectual exercise and sometimes relatively abstract way of learning that goes with the territory. Thankfully there are complementary resources to pad out your understanding—though as always with this type of literature, it is wise to understand the source material.

Already Vernon wrote in his first book that sometimes DDD is “first embraced as a technical toolset” (Vernon 2013, p. xxi) saying that some refer to this modus as “DDD-Lite”. This will bring forth a number of useful patterns but will miss out on the glue that binds together DDD as a complete concept.

## What is DDD?

The [Wikipedia](#) definition is actually very good and condensed. I'm going to reference it as-is. It says that...

**Domain-driven design (DDD)** is a [software design](#) approach focusing on modelling software to match a [domain](#) according to input from that domain's experts.

In terms of [object-oriented programming](#) it means that the structure and language of software code (class names, [class methods](#), [class variables](#)) should match the [business domain](#). For example, if a software processes loan applications, it might have classes like `LoanApplication` and `Customer`, and methods such as `AcceptOffer` and `Withdraw`.

DDD connects the [implementation](#) to an evolving model.

Domain-driven design is predicated on the following goals:

- placing the project's primary focus on the core [domain](#) and domain logic;
- basing complex designs on a model of the domain;
- initiating a creative collaboration between technical and [domain experts](#) to iteratively refine a conceptual model that addresses particular domain problems.

For a software engineer or architect, the above should at least superficially sound clear and reasonable. Two terms, however, should outline themselves as being in repeated use and being somewhat mysterious or vague in meaning: *domain* and *model*. These are core to understanding Domain Driven Design.

## The “domain”

The domain can be thought of as the principal subject or material of the project. It may be as broad or as narrow as necessary. Eric Evans defines it as:

[...] a sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.



---

— Eric Evans: Domain-Driven Design Reference: Definitions and Pattern Summaries

It should be somewhat clear in most circumstances what the implicit domain boundaries are. DDD eschews implicit boundaries and is particular on boundaries being collaboratively and explicitly defined.

## The “model”

Eric Evans spends quite some space at the start of his book on the notion of a “model” and what model-driven design means. Being model-driven can be likened to virtually being domain-driven. By having a shared understanding, and respecting that there is a need for zooming in/out, we can condense our knowledge to an efficient and useful model that is possible to share with others without losing meaning in the process.

The intangible *domain* can be distilled into a tangible and malleable (*domain*) *model*, which can act as a vehicle for securing shared understanding. It is the “[organized and structured knowledge of the problem](#)”. In total, the domain model can exist as one or more individual pieces of documentation (text, diagrams, code...) as long as it adequately represents the problem in a meaningful, truthful, but necessarily simplified meaning.

There is a beautiful [Borges](#) quote that you may be familiar with:

“In this empire, the art of cartography was taken to such a peak of perfection that the map of a single province took up an entire city and the map of the empire, an entire province. **In time, these oversize maps outlived their usefulness and the college of cartographers drew a map of the empire equal in format to the empire itself, coinciding with it point by point.** The following generations, less obsessed with the study of cartography, decided that this overblown map was useless and somewhat impiously abandoned it to the tender mercies of the sun and seasons. There are still some remains of this map in the western desert, though in very poor shape, the abode of beasts and beggars. No other traces of the geographical disciplines are to be seen throughout the land.”

— Jorge Luis Borges in [A Universal History of Infamy](#) (1946; from <https://www.thepolisblog.org/2012/10/jorge-luis-borges-on-empire-and.html>)

Besides being the nightmare of enterprise architects and surveyors, **grand maps (or schemas) that intend to explain everything can devolve into reprehensible detail.** That's where the very human ability to abstract complex knowledge into models comes in. DDD taps into this quality while being completely open to the ways in which a group might do it.

I am seeing in most traditional organizations that a divide is created, problematically, between domain experts and developers. I find it reprehensible that it's so common to believe developers (and sometimes architects) somehow cannot understand the details of the domain.

The better option is, thus, to do whatever it takes to create an amenable, low-threshold environment where all relevant parties (from business to implementation) can work together. Before this, none of the *ubiquitous languages* or *domain models* will emanate.

## The patterns of DDD

Domain Driven Design is not just something that makes you sound smart, but it's actually a very hands-on toolbox as well. I've reproduced one of Eric Evans' diagrams with colored bubbles to easier see where the tactical and strategic patterns reside. Green (top) is for tactical patterns, more concerning the implementation and code side, and the lilac (bottom) is for strategic ones that deal with the modeling, understanding, and integration of our domains.

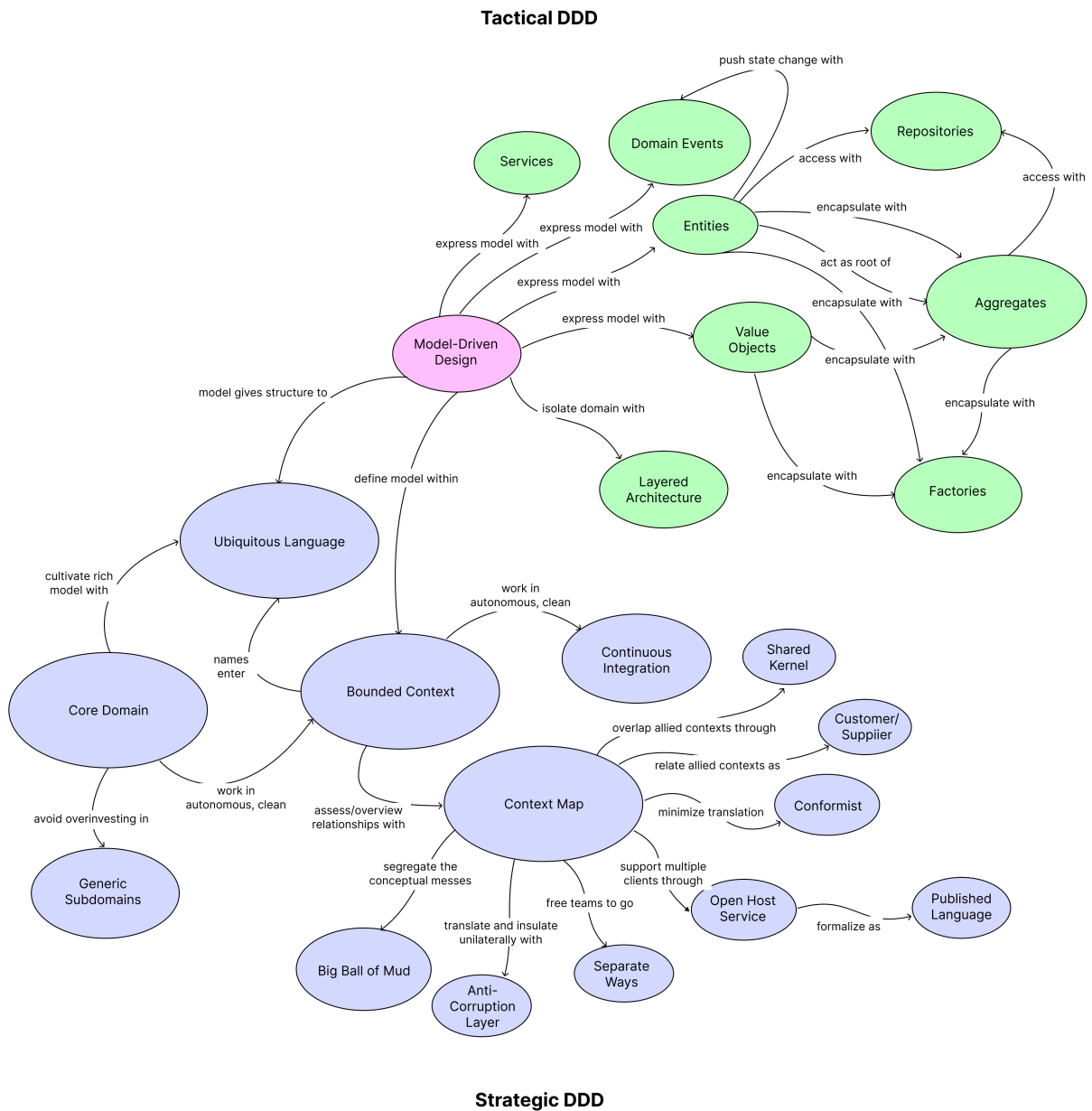


Figure 1.4: How the patterns match up, as presented in Eric Evans's book (2003) and re-drawn by myself.

It may look overwhelming but don't fear!

The central concept is *Model-Driven Design*, which I am sure to come as no surprise—without that, there is nothing more than *either* theory or some sound coding advice.



**Information**

There are of course dedicated sections in this book for strategic and tactical DDD, respectively.

**Strategic DDD in short**

You need to start with the strategic part of the work. In this work, you will uncover the domain(s), its language and terminology, how things relate to one another, and where responsibilities lay (or should lay!).

The work here is collective and should be done with a broad set of constituents, from business to design to programmers and any domain experts. Expect diagrams, post-it notes, coffee, and arguments!

**Tactical DDD in short**

The opposite side of the coin is the tactical work, which instead revolves chiefly around the code and any implementation work. The beauty of DDD is that we are expected to express 1:1 the actual business processes and language through the code and its functionality. DDD provides a small set of patterns to use, all of which are mutually complementary.

## Why DDD and microservices are a match



*Figure 1.5: Illustration from Undraw*

To understand microservices is to be aware of the “prior art”, namely Service Oriented Architecture (SOA) that dates back to the 1990s. Without making this into a history class, the scope of SOA was wider than that of microservices and built on a technical foundation that for natural reasons is not necessarily the basis of modern organizations. It may be fair to call microservices a particular subset of evolved SOA.

### Information

Read more at:

- [Wikipedia](#)
- [IBM](#)

Microservices have grown in popularity as they, among other things, make it easier to build and represent distributed scenarios than building monoliths. As stated by [Amundsen, Nadareishvili, Mitra, and McLarty](#) and referenced at [O’Reilly](#), microservices are:

- Small in size

- Messaging enabled
- Bounded by contexts
- Autonomously developed
- Independently deployable
- Decentralized
- Built and released with automated processes

— [Mac Slocum: Microservices: A quick and simple definition](#)

### Information

For more reading, see for example:

- [Martin Fowler](#)
- [Building Microservices](#)

Microservices (with their above qualities) make it easier to *express, as technical artifacts, the business domain language*.

With the advent of Kubernetes and serverless functions, the practical operations around deploying services also significantly improved. Architects could finally *actually* get those fine-grained services, and developers could finally build them faster and more neatly. This meant that DDD could move out of the enthusiast/nerd/Java/enterprise context and start being applied in broader circumstances.

## How do domains communicate?

Using primarily messaging mechanisms—such as Kafka or AWS EventBridge—we can make the domains and their respective bounded contexts able to communicate with each other. Certainly, you can use traditional request/reply communication via REST APIs (and similar).

What’s worth keeping in mind is that there is little that is “new” with this way compared to traditional service-to-service communication. Ideally, we would have:

- Documentation for the API and events (using a modern schema format like [AsyncAPI](#))
- Microservice discovery catalog (like [Catalogist](#) or [Port](#))

Using the strategic and tactical patterns of DDD we can do the intellectual, as well as technical, labor required to set our path.

## Is DDD still relevant?

So if DDD is ~20 years old, isn't that ancient and archaic by today's standards?

This is a very valid question. However, without going on a philosophical detour, we need to remember that some things in computing and technology change frequently, while others do not. Consider the following:

- [Design Patterns: Elements of Reusable Object-Oriented Software](#) (Gamma, Johnson, Vlissides, Helm) was written in 1994.
- [The Pragmatic Programmer: From Journeyman to Master](#) (Hunt, Thomas) was written in 1999.
- [Refactoring: Improving the Design of Existing Code](#) (Fowler) was written in 1999.
- [Patterns of Enterprise Application Architecture](#) (Fowler, Rice, Foemmel, Hieatt, Mee, Stafford) was written in 2002.
- [Clean Code: A Handbook of Agile Software Craftsmanship](#) (Martin) was written in 2008.

### Information

For more examples (including books going as far back as the 1970s!) see <https://softwareengineering.stackoverflow.com/questions/449117/there-an-expiration-date-for-well-regarded-but-old-books-on-programming>

Principles and broad patterns simply don't age as much as (most) languages and pretty much anything that is only implementation-oriented. It's safe to say that DDD and its



patterns have survived many changes in technology without losing its relevance—though absolutely there are other approaches to DDD popping up.

The bigger issue with DDD and related terms is that with popularity and the acronym going into more widespread use, we start to have a less cohesive understanding of the term. This is not unique to DDD in any way! The sentiment is echoed very well by aryehof on Reddit's thread [“Is Domain Driven Design still the recommended approach for enterprise applications or has any newer approach superseded it?”](#):

**Well what is in the original DDD book is not what tends to be written about or used in practice.** The book is about how to successfully and repeatedly implement plumbing to support a complex problem domain object model. It advocates that standard plumbing so you can *concentrate* on object modeling the problem domain.

“The goal of domain-driven design is to create better software by focusing on a model of the domain rather than technology.”, Eric Evans, Domain Driven Design p148.

**Unfortunately, very very few know how to object model a domain model independent of UI and persistence, so all sorts of alternatives have arisen.** Most notably Event Sourcing/CQRS, or Data Table-based object model architectures (aka Repository Driven Development), or “it’s really just about boundaries and language” (lol). **This is particularly the case in the .NET world where object modeling was never advocated by Microsoft or adopted by the community given its database tooling/wizard orientation.**

So DDD is popular in the same way that “Agile” is. In both cases, their meanings have been corrupted in popular use, and in DDDs by a community that hasn’t even read *the* book (“but hey we read some articles or YouTube”). So something ambiguously labeled “DDD” is often recommended today but does that really help?

With this book, I definitely don’t want to fall over on the wrong side of history!

To be fair, I’ve had a situation like this in mind while writing and I believe you will have a broader yet practical understanding of DDD without us overcomplicating things more than warranted.

### Information

It is my highest recommendation that you also read the source materials, as those are some of the most excellent books and articles I—and I am sure, many out there in the software world—have read on software architecture. My own work with this book and project simply complements and illustrates some of those basics in a practical scenario, rather than extensively elaborating on them.

## Reasonable objections

Let's look at some reasonable objections to our two core subjects. These might seem off-kilter since I'm not changing course as we are just starting, but giving you a taste of when the approach we will work with may be too much.

### **"DDD might be too much?"**

Yes.

This point is raised in at least Evans' book, Vernon's book, and [Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy](#) by Vladik Khononov. So no surprises.

Their point, and mine, is typically that trivial and/or CRUD-oriented systems are good examples of when there is no meaningful reason to pursue the route of DDD. Based on the fact that a lot more software than we sometimes want to accept is just trivial "getting and setting" of basic data, this point should carry across powerfully over a rather wide swath of software engineering projects.

There are ways to make DDD more manageable and I am satisfied with the coded solution we will work on as being representative of such a "lighter-weight" path.

### **"Microservices might be complicating things?"**

Yes.

Microservices have, like every architectural decision and architecture style, their own trade-offs and pros and cons. In the early design stages, it should be clear if there are sufficient reasons to opt for a domain-driven, message-oriented, and decoupled landscape or if another type of solution makes better sense.

### Information

Read more about related concerns at:

- [Architecture tradeoff analysis method](#)
- [Non-functional requirements](#)
- [Application Holotypes](#)
- [Continuous architecture principles](#)

One of the most obvious and significant negative sides of microservices is that their relative autonomy means that you get a linear amount of extraneous “surface area” for each new service: Starting perhaps with the same CI pipes, same scaffolding, same interfaces, and so on. These might be duplicated, or worse, contain tiny differences between them. At the same time, there are ways to handle this (like loading several repos into an IDE workspace, “poor man’s mono repo style”) or simply accept that each service is truly decoupled from the others.

Similar to the previous point (on DDD sometimes being “too much”) it’s sometimes a better proposition to make a monolith, or to at least bundle applications/systems in a coarser fashion. There is nothing controversial about that. However, by doing so you also discard the quality attributes of microservices (independence, scalability, likely more natural representation of bounded contexts, etc.).

During my years working with microservices, as with anything, I have learned that the notion of microservices being *complicated* (or *complicating*) or not is very dependent on an engineer’s or architect’s background and experience. Personally, I’ve grown a lot since working with serverless microservices as they abstract the “right things” while providing the powerful tools I expect in a modern tech environment. Without them, I would not have started my back-end journey the way I did. So as ever: One man’s curse is another’s

gift.

## **In closing**

Any architecture style, framework, and approach will bring something to learn and adapt to. DDD has stuck around successfully for a long time and adapted well to the changes in the technology landscape. Over time we've also seen in actuality how to bring the essence of DDD while making it easier to work with.

The state of the matter is still, as has been the case since 2003, that DDD (in totality) is not a panacea for all software design cases. This should not come as a surprise to anyone even tangentially interested in Domain Driven Design.



## Putting DDD in the serverless context



*Figure 1.6: Illustration from Undraw*

We live in a time of change when it comes to technology and cloud with ever higher degrees of abstraction being presented as commercial products.

I will equate microservices roughly with serverless. Indeed they are not the same, but the scope of this chapter is also generally broader and more imprecise than what we will work on later.

In this chapter, I want to elevate some of the issues that I have seen in understanding and working with serverless, so that we can better understand how the contents of this book attempt to solve real issues for you.

### **Not just back-end engineers work with servers these days**

I am one of those developers who became awestruck when I started to learn about and use serverless functions, in my case back in 2017. At that time these were much more primitive than today. Personally, I didn't want to become an expert on Redis caches or server maintenance, and really avoided lunch conversations with the back-end people. As a front-end developer (and some Node) the things *around* the server weren't really that enticing. You couldn't see what was going on, and it seemed like lots of heavy lifting for pretty much no gain—after all, people like myself were building the majority of the

application! (Or so I thought...)

In the years since, I've had the possibility and fortune to onboard and move many developers into the cloud and various serverless products running on Azure, Google Cloud, and AWS. To be fair, the entire stacks have been primarily serverless. But why were they?

Firstly, of course, because the serverless solution was a good fit (else we would have a real problem!) but also because I had learned first-hand myself what it means to reap the benefits of being able to do *more* with less. And that certainly did not include any of the “[pet management](#)” that went on with the traditional-type back-enders. In a very real and tangible sense, I could support developers with tools that gave them new abilities. I've seen these pay dividends both in small teams as well as in scaling out development teams at Polestar.

The backside of this is that back-end development, which has maybe sometimes been seen as more “privileged” than front-end development (and requires most conventional software engineering background), is now actively conducted by developers who often do not carry the same core skills. This point will touch the next a bit, but the gist is that I have personally seen many examples of (often self-taught; like myself) developers building their back-ends with, often, a very different (objectively worse) sensibility than those who come with traditional engineering backgrounds. It's not that all front-end developers are *bad* (that's not at all the point!), nor is it possible in an extremely wide industry like tech/IT to claim that everyone walked the same path, but I think it is both realistic and relevant to note that such overall enablement will mean an influx of people who lack some of the (sometimes unspoken) necessary skills of back-end engineers. Front-end work has generally had less need for systematic and disciplined “school bench” engineering skills.

### **In short: Full stack will become the norm**

More people are enabled to work full stack because of serverless, but this influx also equates to a wider skills range, in turn ending up with the sector (and we as organizations) having to train people accordingly **if they lack core skills that they may not even be able to identify as deficits in the first place.**

## There's a pretty big skills shortage

As anyone in the IT world can probably attest to, there is a pervasive skills shortage. There are just too few people to do all the work out there! As a continuation of the above, while companies, schools, industry, and even private initiatives try to drive in new blood, the competence curve has to be upped as well, to take new developments and technologies into consideration. I am still repelled by how many schools keep offering courses and training that quickly seem outdated, and how many companies are still holding on to technologies that at least I cannot really understand why they won't build away. There are always reasons, of course, but the more people, etc. you bring in, the harder it will be to change; Basically, it's like keeping digging a deeper hole because it's convenient to not change tasks, i.e. move out of the hole.

There is an interplay between education, market, and industry in which certain technical developments may not be part of these trajectories. That means that in an ugly scenario we have something like the following to happen:

- **2025:** New technology starts being used ("widely enough" for our example)
- **2026:** Architecture group at Company puts the technology as part of its strategic vision; early recruiting for roles with the given tech skill starts; education curricula start updating with the technology
- **2027:** Education starts
- **2028-2030:** The first batch of students learning the technology are graduating; technology is likely to surpass and/or replaced

The above is a made-up, dumb example, but it has some truth to it. Never expect the "traditional paths" to be fast enough to actually cater to your tactical recruiting and skill-building needs.

Serverless gives developers a vast extension of capabilities but good use of it assumes **solid core skills**, ergo: It's likely easier to train someone from scratch to deliver basic value with serverless tech (augmenting the person and abstracting parts of typical engineering) than with older takes on the same things, but the question remains "Is this a good engineer at the end of it?".

**In short: Continuous learning needs to increase**

Strong fundamentals in each candidate/employee/team member and aggressive, continuous learning supported by the organization are key here. Retaining and retraining current developers is probably the best value you can get, rather than trusting an uncertain supply.

**You might need to learn a lot of things anew**

Going down this path will entail having to conceptually and practically accept that many well-used core technologies, like relational SQL databases, API frameworks like Express, and webservers like Nginx are often very different from the cloud-native products you will use when you go serverless. Certainly, they are not 100% different, but adding up each area will end in a relatively significant total amount of work on your side to learn and/or re-learn how to build a complete solution.

It's not completely uncommon that some developers will struggle with adapting, complain about their experiences, or even actively resist learning if everything goes south. If you represent an organization or team, then definitely be level-headed and lucid that all learning is hard; it's just different levels of hard for all of us.

**In short: Serverless is not a one-minute fix**

Expect to learn that you need to support a team making the move to serverless.

**Serverless functions “identity crisis”**

There is a minor identity crisis in the functions-as-a-service world that we can address in this book.

There are two typical styles:

1. Single-purpose utility functions (“workflow-oriented” cases)
2. Entity-oriented functions (“back-end or API” cases)



Worst of all, it's not like it's being very clearly communicated, especially to newcomers. Why I find this detail important is that it points to a certain orthodoxy in the FaaS or microservice world. Are any of these familiar?

- “Microservices don’t share a database”
- “Microservices are single purpose functions”
- “Microservices are no longer than a screen of text”

### Warning

I’ve said and written all of those ideas or principles at some point and I am 100% sure you have seen most of those already if you have read up on this subject. They aren’t incorrect, as much as they are too narrow.

So the fact is that they are not really wrong, but that they have become doctrinal rather than indicative or guiding.

A silly and made-up but very realistic example of a valid question from an engineer could be:

“If a Lambda function can only be ~100 lines long, how do I even get to writing a meaningfully complex service?”

And that’s where the *style* of how we write functions starts mattering.

The style of examples, tutorials, and much of the code out there will use the single-purpose style. It’s very intuitive and nicely contained in for example a data processing example: Something comes in, you do a few snippets of code, and then exit with some status code. Brilliant. Except that a lot of real business use cases have to move beyond that... That’s why we need to spend time looking at some patterns for structuring our code so that we keep the general spirit of serverless functions (well-contained, small, discrete) while supporting richer, more detailed business use cases.

**In short**

Understanding different styles of writing functions are important so that we can apply the correct engineering measures to structuring, writing, and testing our code. For a newbie, the Google wilds will not directly help with explicitly communicating the characteristics of these styles.

## Project resources

The application source code is available on [GitHub](#).

## **Getting started**

In this section I'll describe our scenario, how to follow along, and some thoughts on getting the creative juices flowing.

## Scenario

The expensive and outdated room booking system at your company has been making life miserable for pretty much everyone there. Your team has volunteered to replace the system with a cost-efficient custom-made implementation, with a target of doing so within the space of a couple of weeks. To drive down cost and maintenance you've already settled on using serverless cloud technologies as the core components.

You've just had a brainstorming session and a requirements workshop together with stakeholders from the business and office management side of things, as well as with some front-end developers in the company.

**Now comes the real question: How do you design the system?**

## Functional requirements

For now, these are the identified high-level requirements:

- Reserve a *single* room in a *single* facility (your office) and your time zone.
- Reserve the room in slots of 1 hour at the start of each hour.
- Allow for the cancellation of room reservations.
- Allow for rooms that are not checked in within 10 minutes of their starting time to be canceled automatically.

## Non-functional requirements

The provided non-functional requirements (quality attributes) are:

- The solution needs to have at least some minimum hygiene level of resilience and security.



## **Integration**

When it comes to integration work:

- Assume that the front end should be able to request an updated view on bookings.
- Assume that the front end will provide input.
- The user name.
- The room name or ID.
- The start and end times of the slot.

## **Delivery**

Your team will focus on providing the back end, APIs, and such; the front end is out of scope for your team.

**How will you address this challenge?**

## How to follow along

The general format of the rest of this book will be in the style of a guided tour, meaning that I will write sequentially (post-fact) about the key parts involved in landing a solution I feel encompasses what this book is trying to tell.

This example focuses on overall system design and the goal of setting up an event-driven architecture that fits our scenario.

### Information

In the interest of time and energy, certain features of a full solution are therefore excluded from the scope of this exercise. Also, we should spend less time on details like worrying for conflicting names of rooms, as that is not what we are focusing our cognitive effort on.

You can...

- **Go on a guided tour:** Grab a coffee, just read and follow along with links and references to the work.

or

- **Do this as a full-on exercise:** Clone the repo, run `npm install` and `npm start` in the respective project folders, then read about the patterns and try it out in your own self-paced way.

## Structure

The root of the GitHub code repository will contain the following pertinent bits:

- **code:** Source code for the reference solution, divided into domains and then bounded context

- `data-modeling`: JSON files that show us a rough final state of data that is used between contexts/solutions
- `diagrams`: Diagrams for the solutions

## Data modeling

The `data-modeling` folder contains various forms of data that roughly represent their final shapes.

I've found it a powerful tactic to do this type of hands-on payload modeling work already at the outset of a project, as it's lightweight, fairly easy for non-technical people to understand, and can be a collaborative exercise. You can also actually use the JSON objects when you are writing your actual implementation later!

## Commands

The below commands are those I believe you will want to use. See `package.json` for more commands!

- `npm start`: Runs Serverless Framework in [offline mode](#)
- `npm test`: Tests code
- `npm run docs`: Generates documentation
- `npm run deploy`: Deploys code with Serverless Framework
- `npm run teardown`: Package and build the code with Serverless Framework

### Information

Also note that the code's "[mono repo](#)" structure is more for convenience than a true "decision" as such.

## Prerequisites

*These only apply if you want to deploy code.*

- You have a recent Node version, ideally version 18 or later
- You have an AWS account
- You have sufficient AWS credentials to deploy the required infrastructure (including API Gateway, Lambda, S3 and more)
- You are logged into AWS in your environment
- You follow any additional instructions per each Bounded Context's (microservice's) `README.md`

## Doodling, drawing, whiteboarding flows and concepts



Figure 1.7: Illustration from Undraw

### Information

This section has nothing to do with DDD but has everything to do with encouraging creative processes to function and flow ideally.

Early preparation or “pre-prep” on the concept stage is something I value a lot. I feel it makes it easier to think and work and ask relevant questions.

Making something good is not about being a genius, but being able to feel comfortable in *playing with the substance of the work*. Getting used to it (its constraints, its properties, the stakeholders, priorities...) is therefore very important. I’m sure you might have been in many meetings where you are introduced to a new project and various angles or questions that (later) feel totally natural but didn’t get a single thought aired in that initial meeting. That’s why I like this pre-prep doodling, so you are gradually “getting used” to the new work.

I think it’s a really valid concept to do such [spitballing](#)...

- on **paper**, if you are alone, or
- on a **whiteboard**, if you are with more people.



**Information**

While physical media may seem ancient to some it's much more versatile, it has no real formatting restrictions, and it enables thinking/writing/doodling that is less constrained than that enforced by digital options.

At this stage, it's more about getting your mind used to the business case, any terminology you might know (or need to know!), some flow charts to describe where you might be going, etc.

When it's in the early stages I find that sometimes I need to remind folks (sometimes even myself) that these really *are just* drafts or rough sketches and not a full-fledged way of building something.

Personally, I think this part is really fun—just don't let this part be a massive bookend for the work you are doing but more like a creative exercise to start feeling the materials and substance of the work. As someone who knows DDD, I use this exercise as a way of also trying a tentative “first fit” for various things I encounter in the project into the “tactical” concepts of DDD, such as Entities and Domain Events. More on those later.

Figure 1.8: Picture of some of my first notes for this project.

## Information

**Note:** As a part of my work with writing this book and building its coded implementation, this is the stage in which I really for the first time started to see what the division of services might be.

## Strategic DDD

The “strategic” half of DDD deals as expected with those concerns that are more wide-ranging across the system, more fundamental to its functionality, closer to the business processes, and anything else that will drive the details and requirements for later implementation work.

In this part we will focus on:

- Documenting our **ubiquitous language** (terminology, naming...)
- Drawing out the boundaries of our contexts (systems or applications...)
- Defining the relations between contexts

This isn’t technical work per se, but it dramatically eases the technical work we will do later.

It’s as easy as “No strategic DDD = no DDD”. Let’s get to work!

## The domain(s)



*Figure 1.9: Illustration from Undraw*

It's about time we make it clear what this “domain” thing is, right?

[Vaadin](#) has a fine two-part introduction to DDD and their distillation of the various domain types is short and informative enough for me to want to quote it wholesale:

A **core domain** is what makes an organization special and different from other organizations. An organization cannot succeed (or even exist) without being exceptionally good in its core domain. Because the core domain is so important, it should receive the highest priority, the biggest effort, and the best developers. For smaller domains you may only identify a single core domain, larger domains may have more than one. You should be prepared to implement the features of the core domain from scratch.

A **supporting subdomain** is a subdomain that is necessary for the organization to succeed, but it does not fall into the core domain category. It is not generic either because it still requires some level of specialization for the organization in question. You may be able to start with an existing solution and tweak it or extend it to your specific needs.

A **generic subdomain** is a subdomain that does not contain anything special to the organization but is still needed for the overall solution to work. You

can save a lot of time and work by trying to use off-the-shelf software for your generic subdomains. A typical example would be user identity management.

— [Petter Holmström: DDD Part 1: Strategic Domain-Driven Design](#)

Recollecting what I have read and how I personally think about the *domain* concept, for me the term has always been fairly straightforward and intuitive. If you do struggle with the concept (and its siblings core/supporting/generic subdomain) then remember that these are **logical constructs**, rather than theory-heavy notions. Understanding the relative sizes and relations of an organization or the parts of a project is something that has to be done in-situ, together with the people who:

- Know their way around the project/organization, and
- Will make things hard for you *if they are not part* of making these divisions into the various domain types.

The problem I've found is that for some odd reason people just don't tend to go around all day speaking of "domains". You'll probably walk away empty-handed if you ask the common business person in your organization "Hey there champ, [what's the deal with our domains?](#)". Also, to divide systems into domains after they are fully designed is likely useless too. It should be done, at least coarsely, already in the initial design.

Instead, what you might want to do is to **introduce DDD as a framework, most importantly its ubiquitous language and domain concepts**, and begin discussing and doodling what the intended flows are if the scenario requires building something new. Equally important, and even more so if you are consulting or otherwise external to the organization, is to interview and map how the organization works (with some focus on the technology, if not only to understand how it may differ from the intended business domains).

In essence, you will want to make it clear that all sides have to cooperate to work within the frames of DDD since none of it will come intuitively in organizations as they typically look in the 2000s.



**Information**

Most of the bigger books on DDD include rich scenarios where you can kind of role-play along with the text about how these things might work out.

To document this you will want to think about questions like these:

- How many parts (i.e. *subdomains*) constitute your domain?
- Where do you make your money? Where do you lose your money?
- Where are your organizational pains?
- How does parts of the organization work together?
- Do the subdomains speak the same *ubiquitous language* and are they truly part of the same domain or can they logically move out?
- What is the importance of the respective subdomains?
- How do subdomains interact? (i.e. *relationships*, *context mapping*)
- What leading data (i.e. *aggregates*) exists and who owns it and can modify it?
- What *commands* may change leading data (*aggregates*) and who may create commands?
- How are changes to leading data (*aggregates*) communicated to others? (*domain events*)
- And more!

**Information**

If you use Miro, consider to use a template like this: <https://miro.com/miroverse/strategic-domain-driven-design-template/>

These are some of the common, quite untechnical questions that we need to start with as we start doing DDD. We will see soon start seeing examples of concrete artifacts that can be co-created to describe the domain.

## Terminology and language



Figure 1.10: Illustration from Undraw

The core term in DDD is something that for us non-native English speakers is kind of a mouthful: the *ubiquitous language*.

**“Ubiquitous” in this context means that it’s “everywhere around us” when we are in the domain.** You can think of it as being the common or natural jargon and words that people use when they work together in that context. As such, it’s most likely non-technical in nature (at least in the software sense) and probably heavy on the business side.

Words are the core of DDD because they, as philosopher Ludwig Wittgenstein once wrote, shape the limits of the world for us:

**The limits of my language stand for the limits of my world.** The limits of my language are the limits of my mind. All I know is what I have words for.

— Ludwig Wittgenstein, “Tractatus Logico-Philosophicus” (1921)

Without the words to serve our purposes, our implementations will always suffer from this lack. It is more efficient to discuss and collaborate and build a common shared “worldview” that can later be manifested, than haphazardly building and failing with something that does not match the actual needs because too little grounding was done.

It's not hard to find memes on how product managers, clients, and developers in a range of ways end up screwing each other over, or how software bellyflops into a completely detached state from the realities of the actual business needs. Of course, no one *wanted* to actually end up in that situation, yet they did. I have been part of such failures, and I am sure you have been too.

Beginning with language has many benefits. One of them is that it's non-technical and something we can discuss without a proxy or intermediary. We can come to an understanding of words and processes and we can therefore learn them, thereby ultimately building something that is expressed through those same words and processes. We get to match the logical and semantic constructs with coded incarnations of them.

If it's not clear yet, all of this work has to be done in a cross-functional group where domain experts are part. Domain experts is a broad term but you can think of this as any people who have driving roles and knowledge in your business context. The result is collaborative work, rather than any sub-group having more (or less) privilege than any other. When an accepted set of terms is produced, [word is bond](#).

As quoted by [Martin Fowler](#), Eric Evans writes the following in *Domain Driven Design: Tackling Complexity in the Heart of Software*:

By using the model-based language pervasively and not being satisfied until it flows, we approach a model that is complete and comprehensible, made up of simple elements that combine to express complex ideas. [...]

Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.

Some important features to understand:

- The Ubiquitous Language is the *de facto* standard nomenclature, so other similar words and terms must be avoided.
- The domain model and our subsequent implementations express the Ubiquitous Language.
- The Ubiquitous Language evolves organically over time. As per above, this needs to be reflected similarly in the software artifacts.

- Each domain will likely have its own Ubiquitous Language because of their respective meanings and semantics.
- Ubiquitous “Languages” are not meant to be efficient. Neither are they shared (well, if that doesn’t make sense and is actually the way it is).

## Documenting the ubiquitous language

You can use whatever default means you have concerning documenting, as long as it’s common knowledge that this information is authoritative and co-owned and accessible to all who need to be able to see/edit it.

### Information

Remember that the ubiquitous language is dynamic and ever-changing. It needs to be able to evolve, and we have to be receptive to that. As the language changes, we must similarly assure that the implementation stays intact and semantically aligned. Effectively any language change will also be a technical change.

## Starting to understand the core domain language

Let’s go back to our functional requirements and start sniffing out a language:

- **Reserve** a single **room** in a single **facility** (your office) and your time zone.
- **Reserve** the room in **slots** of 1 hour at the start of each hour.
- Allow for the **cancellation** of **room reservations**.
- Allow for rooms that are not **checked-in** within 10 minutes of their starting time to be **canceled** automatically.

With this, we have now collected Reserve, Room, Slot, Facility, Cancellation, Room ↔ reservation, and Checked-in.

## Filling in the blanks

Unsurprisingly, the language was fairly well-prepared already in the requirements. In a real-life scenario, it would be wise to either discuss/workshop around flows and requirements and peel out the terminology as you go, or to opt for EventStorming to help with that.

In our case, we could, at least after workshopping around the intended flows, see that we are missing for example Checked-out.

## Pruning the language

We can start cleaning the language a bit.

Room reservation sounds too verbose—we can cut this to just Reservation as these are always logically related only to Rooms and their (time) ‘Slots.

Because we only work with a single Facility, we can cut that out too.

Concept	Type
Reserve	Verb
Room	Noun
Slot	Noun
Facility	Noun
Cancellation	Noun
Reservation	Noun
Check in	Verb
Check out	Verb
Checked-in	State
Checked-out	State

*Figure 1.11: Table of language concepts*



**Information**

Later we will look at how EventStorming is a similar (optional, alternative) route we can take to do this and much more.

**The Analytics domain language**

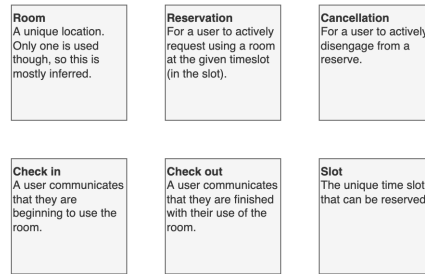
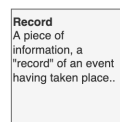
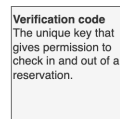
In the Analytics domain, we find one unique concept: The (analytics) Record.

**The Security domain language**

The Security domain has, also, only a single unique concept: The (verification) Code.

**Delivery**

In the reference implementation, this is simply shared as a separate diagram using a basic visual style where each term is described in a short sentence. In a real-life scenario, this format is probably quickly exhausted as definitions may need to be richer and exemplified. Note also how we ascribe each term to specific domains.

**Core domain****Analytics domain****Security domain**

*Figure 1.12: Describing, in short, our Ubiquitous Languages through the domain concepts*

Having this glossary or Ubiquitous Language always ready at hand will be a major benefit, from onboarding to understanding if there are any issues stemming from conceptual misalignment.

## Setting boundaries



*Figure 1.13: Illustration from Undraw*

While the ubiquitous language might be the most pervasive and influential tool in the strategic DDD toolbox, I'd say that **the defining of Bounded Contexts remains the most powerful tool**. Language makes us define the concepts in play, as well as what (and how) they represent something. Setting boundaries on contexts, on the other hand, carve out the landscape, pointing out where responsibilities lay.

Oh, what the Bounded Contexts *are*?

[...] A Bounded Context is a semantic contextual boundary. This means that within the boundary each component of the software model has a specific meaning and does specific things. The components inside a Bounded Context are context-specific and semantically motivated.

— *Domain Driven Design Distilled* (Vernon 2016, p. 11-12)

We use this construct to logically discuss and model our expectations on a given part of the overall problem space. This Bounded Context is then manifested, through programming, into a realized vision of the model. That's the Circle of Life, in DDD terms.

## Core principles

### Information

I will adapt and echo some of the great points made in <https://learn.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries> and in *Domain-Driven Design Distilled*.

### One team, one repository, one Bounded Context

Segregate Bounded Contexts and any work in a meaningful way. Conventionally this is with **one code repository for each team and Bounded Context**. If it's not clear yet, often one Bounded Context translates into one technical solution.

### Bounded Contexts inform you where the system begins and ends

A Bounded Context is always designed from the actual business reality at hand. However, as long as you can meaningfully and accurately decompose parts in their own Bounded Contexts, you are free to do so.

Don't mix up domains, domain languages, or other semantics across multiple divergent needs in the same Bounded Context.

You do not care about anything outside the Bounded Context, only if necessary about any integrations.

### Aggregates are shaped by business needs

An Aggregate, as we will see, roughly equates to what is typically called **leading data**. An Aggregate is the core of most Bounded Contexts. If you cannot decompose the Aggregate into less than its *cohesive* entirety, then you have found the true boundaries of the Aggregate.

## Aggregates don't depend on others

Consequently, services are completely independent of other services and Aggregates and whatnot. All of the required data and behavior are colocated in the Bounded Context, on the Aggregate. Therefore we can always “trust” the way that data is handled, updated, and modified over time. **You do not “split the responsibility” with anyone else.**

## Bounded Contexts in our example code

I will now step through some of the iterations I went through to come to terms with the overall modeling of the Get-A-Room application and somewhat improvise that experience to give you a sense of what it might look like.

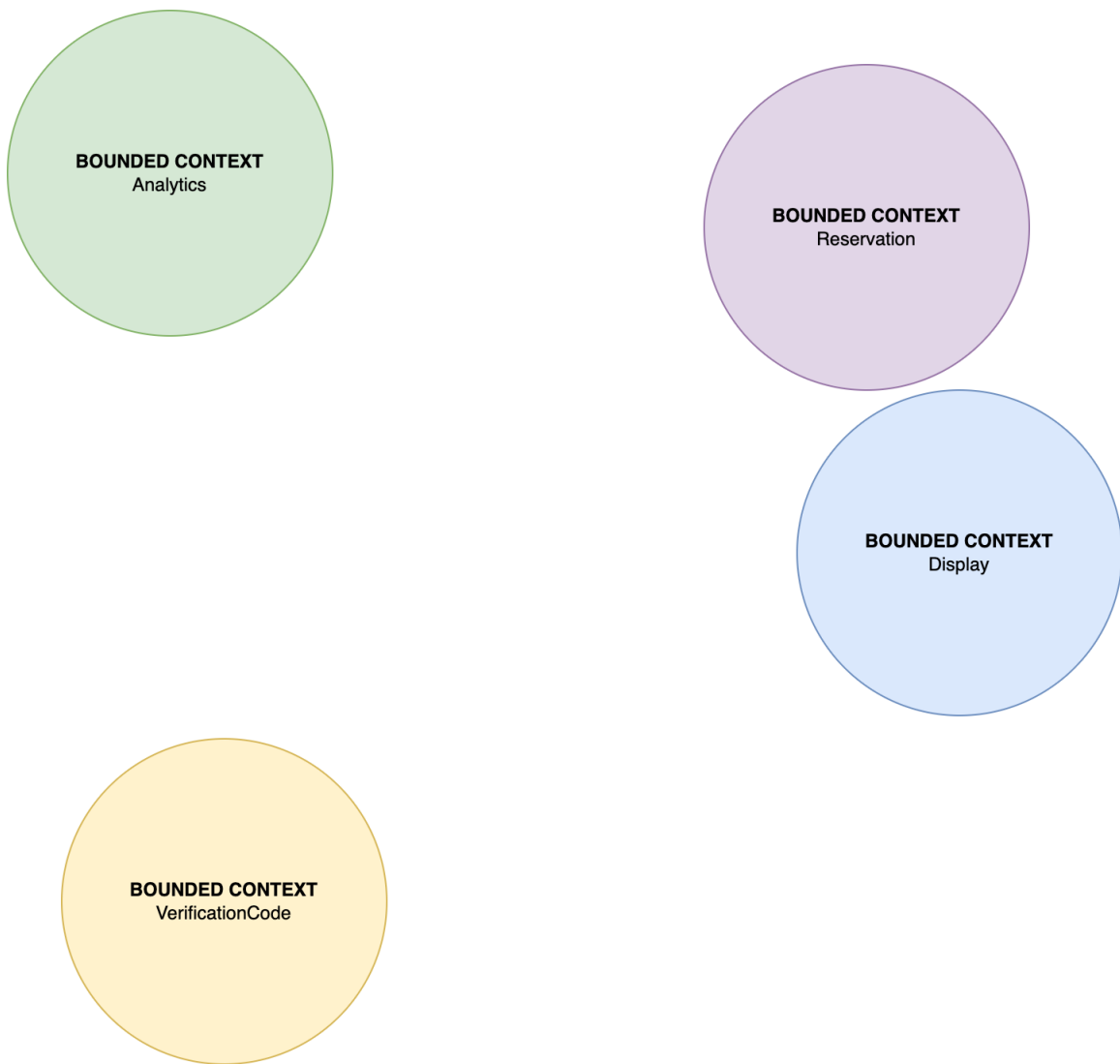
### Information

Consider using some good templates if you are so inclined:

- <https://miro.com/miroverse/the-bounded-context-canvas/>
- <https://github.com/ddd-crew/context-mapping>
- <https://contextmapper.org>

## First attempt: Overall orientation

It was clear early on that there are multiple solutions, or Bounded Contexts, in play in the application/domain. Looking at the overall requirements I could see that analytics and security were completely separate and that the reservation part might be better colocated together. That wasn't necessarily an immediate “score” in my mind but given that their nature of them is intertwined I decided to put them together.



*Figure 1.14: Bounded Contexts established.*

This should not be too controversial, I think.

## Subdomain or Bounded Context?

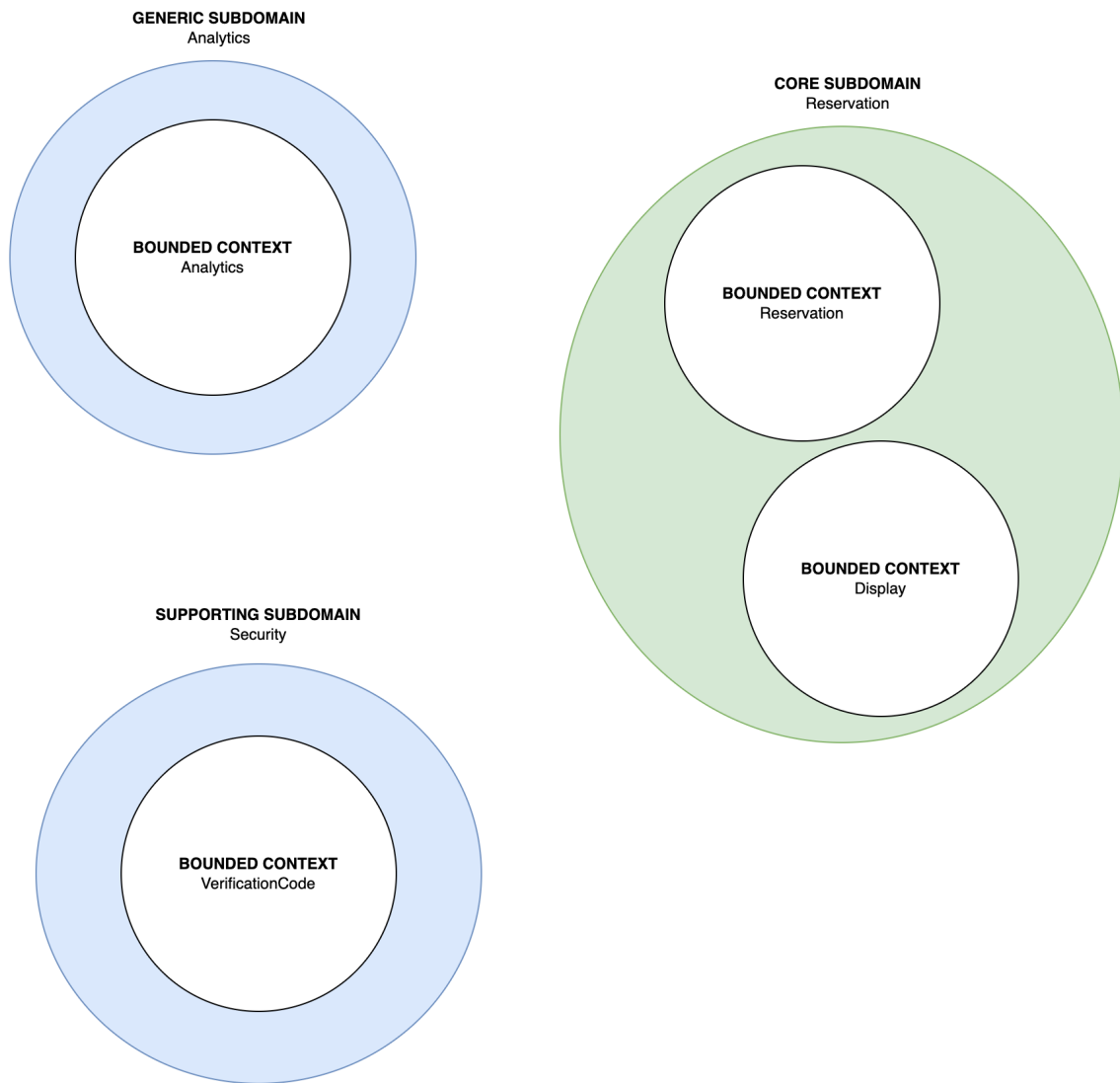
One confusion that Evans sometimes notices in teams is differentiating between bounded contexts and subdomains. In an ideal world, they coincide, but in reality, they are often misaligned. He uses an example of a bank struc-



tured around cash accounts and credit cards. These two subdomains in the banking domain are also bounded contexts. After reorganizing the business around business accounts and personal accounts, there are now two other subdomains, but the bounded contexts stay the same, which means they are now misaligned with the new subdomains. This often results in two teams having to work in the same bounded contexts with an increased risk of ending up with a big ball of mud.

— [Jan Stenberg: Defining Bounded Contexts](#) — [Eric Evans at DDD Europe](#)

As per DDD best practices, we want to have as close alignment as possible between subdomains and Bounded Contexts. Still, because we put Reservation and Display in the same (core) subdomain, that one gets a little fatter. Once again, this is acceptable given that Display will serve as essentially just a read-replica of the Reservation context.



*Figure 1.15: Fitting Bounded Contexts into Subdomains.*

With the addition of subdomains, we have now clarified their relative importance and set the maximum outer boundaries.

- Reservation is the core domain, which we will invest more heavily in
- Security is a supporting subdomain, and we accept that we need to do our own work here for it to be useful in our application
- Analytics is ideally shipped to a commercial-off-the-shelf product or something similar; in Get-A-Room we set up a bare minimum custom implementation

## Context maps and relationships

### Information

For some informative light reading on this subject, see <https://medium.com/ingeniouslysimple/context-mapping-in-domain-driven-design-9063465d2eb8>

In terms of integrations, it should be clear that these somehow need to interact with each other. But how?

- The Reservation context has a **Customer-Supplier** relationship with Display, meaning it supplies data in a way that is easy to represent. In this exact case, it's very close to the original format as it's not that complex.
- Reservation as a whole has a **Conformist** relationship with the Analytics context. This entails that the Analytics service/context dictates how they want data integrated.
- The Reservation context and the VerificationCode context have a **Published language** relationship which is a somewhat convoluted way of expressing that the security side will present a documented API to use.

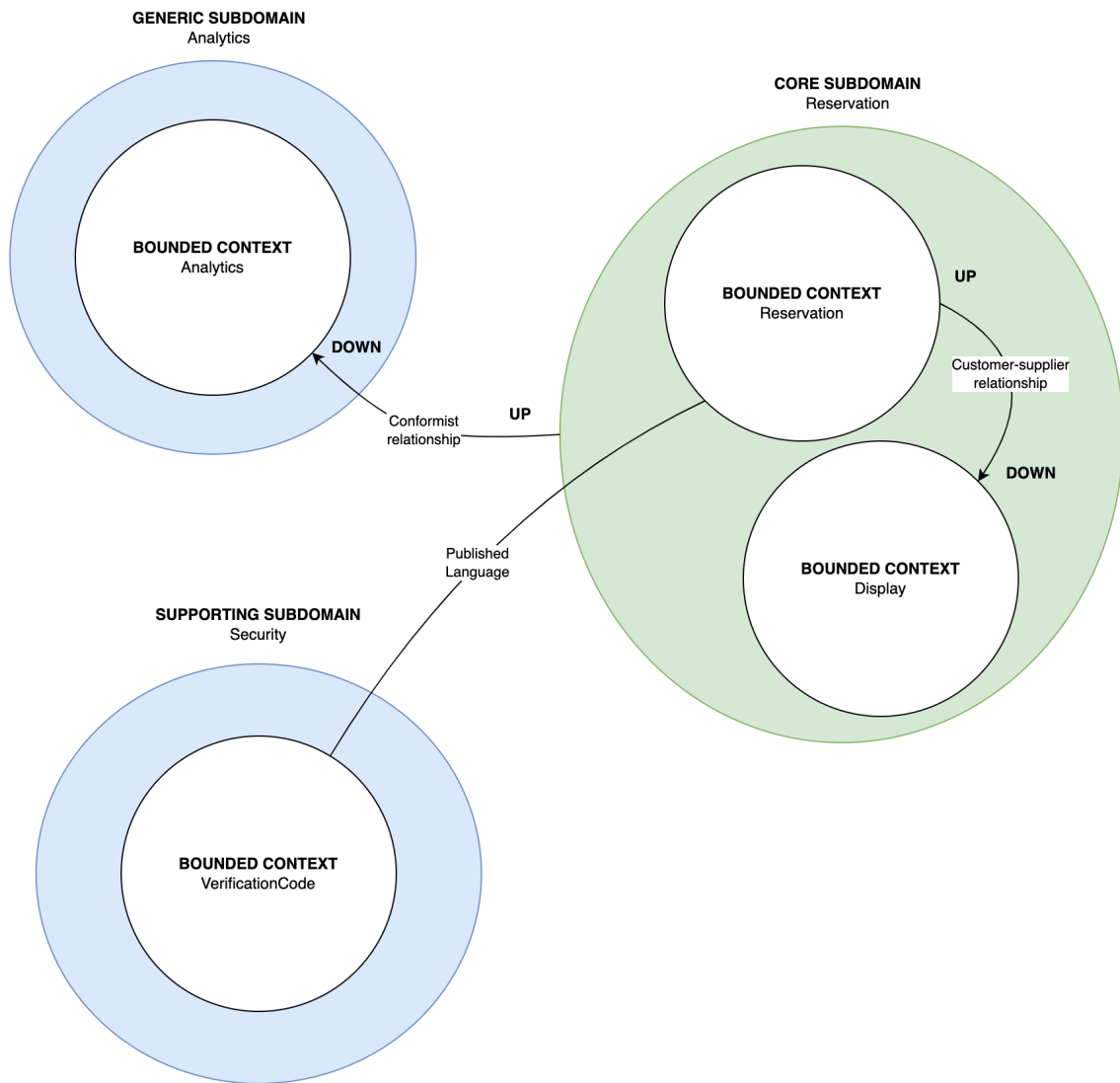
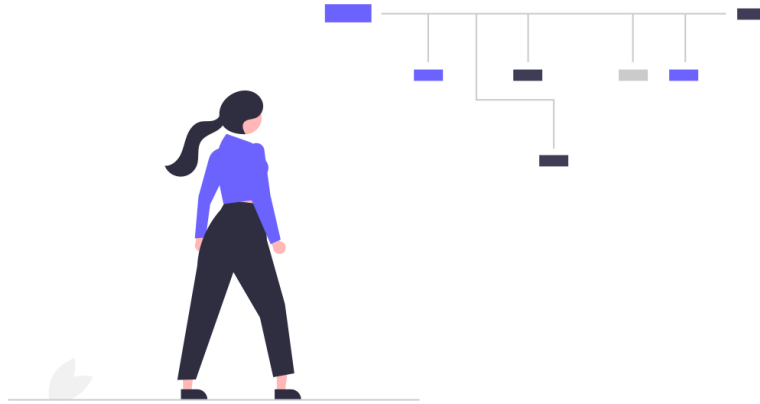


Figure 1.16: Expressing the Relationships in a complete Context Map.

## In summary

Having done all of this modeling, we are well on our way to understanding the domain, how things fit together, how they interact, and being super-clear on which services are worthy of more investment and care.

## EventStorming



*Figure 1.17: Illustration from Undraw*

[EventStorming](#) is a workshop model invented by Alberto Brandolini in 2012 to help facilitate explorative work in the spirit of Domain Driven Design. It's a fast-paced way of coming to a shared understanding among the attendees (who should span the maximum gamut of stakeholders), helping to shape requirements as well as understand the business process.



The original format is very physical, using paper to cover a wall and making use of colored sticky notes and marker pens to, first, come to terms with the current notion of the project/work, and then bit by bit consolidate that view into sequential orderings of the core concepts:

- **Aggregates**, yellow
- **Commands**, blue
- **Domain events**, orange
- **Actors**, yellow (small)
- **Policies** (or business process), purple
- **Views** (“read models”), green
- **External system**, pink

According to Brandolini, the original non-technical, physical format is the preferred format, because (as far as I understood) it's easier to moderate, it's easier to adapt to the current group dynamic, and it simply has fewer constraints than you have with people tethered to their own screens.

### Success

Supposing you want to do this physically, then I can recommend the following article by a person who has done this many times and has tons on tips on what stuff you need to buy: <https://baasie.com/2019/05/08/eventstorming-tools-what-is-in-my-flight-case/> For my part I've purchased the "Super Sticky" 3M Post-its in the "Rio de Janeiro" colorset which seem to pretty closely match the colors used in EventStorming.

It's completely possible to emulate the practicalities of Eventstorming in a remote, digital way. Tools like [Miro](#) work just fine, and you can even do most of this with [Figma](#) or [Excalidraw](#) if you really wanted to. Working by myself I resorted to my trusty old [Diagrams.net](#) as it also made it easy to export for this book and the code repository.

### Information

If you are a Miro user, consider using a ready-made template like this: <https://miro.com/miroverse/storming/>

## My solution

In the below picture you can see the end state of how I addressed this particular project. What is not quite apparent is of course the evolution of terms over time. In my first sketches and first rounds of work, for example, the term Reserve was not used. Instead, it was Book. But you don't really book a room, right, rather you reserve it.

On the good side, this exercise greatly improved even a completely fictional product made by one person! As a DDD apostle, you should always stay wary of any terminology that



reads “Create”, “Read”, “Update” or “Delete”—in my case, it was never much of a problem

My take on this is not necessarily by the book, as we are also missing a key component: The sequential order of these.

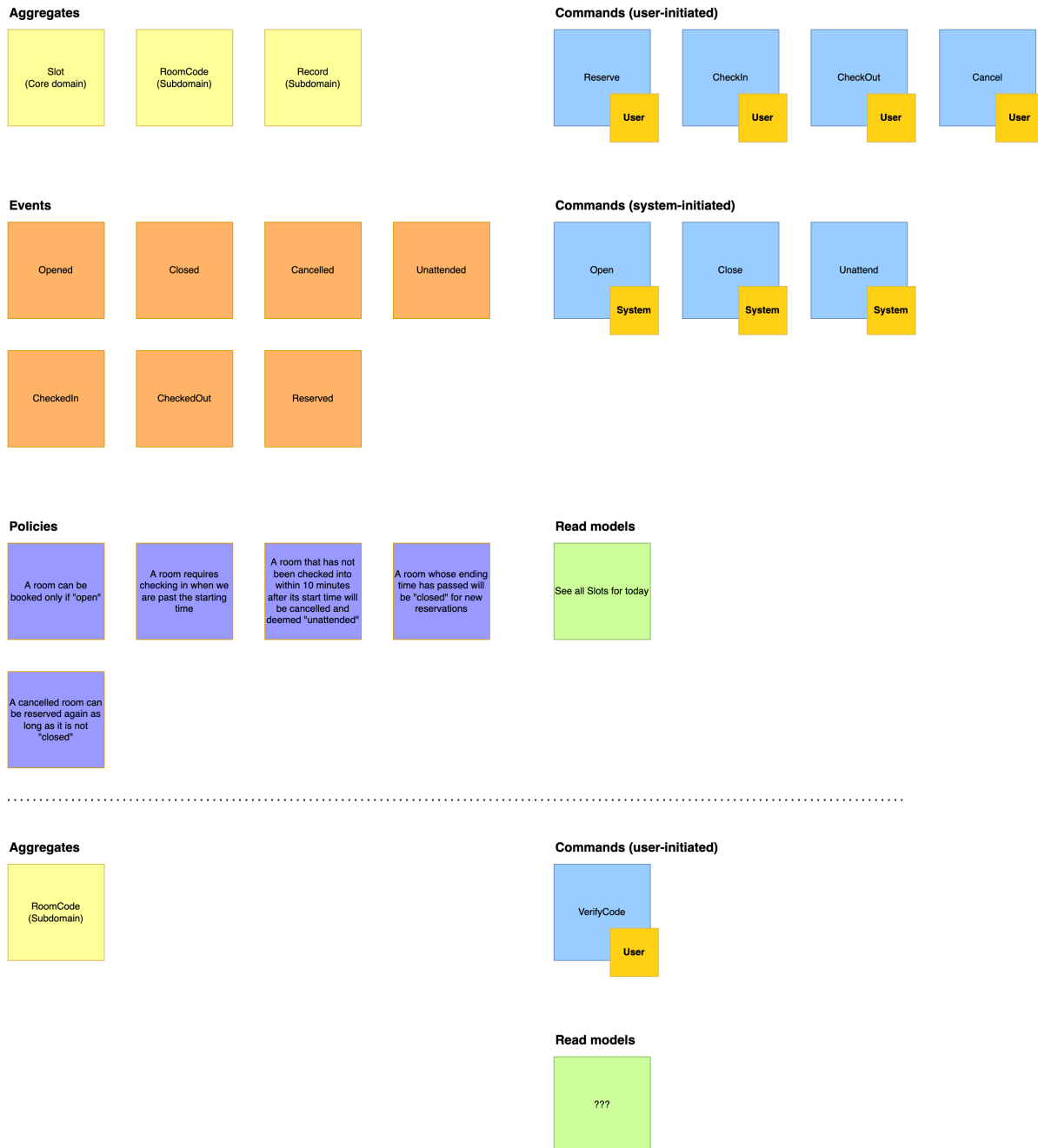


Figure 1.18: Event storming

## Groundwork

This section zooms in on the non-DDD factors that concern our overall engineering, such as choosing a suitable cloud architecture, handling technical boilerplate, and implementing things like:

- The common Lambda handler
- Data Transfer Objects
- Error handling
- Testing approach
- Lambda authorizer
- Writing a basic AsyncAPI schema

## The logical business flow

### Information

You could certainly go full-bore with EventStorming and essentially encapsulate the business flows there. An optional approach could be to do some (lightweight?) [BPMN](#) model, though I would perhaps rather lean back into the full EventStorming approach in that case so we aren't caught in unnecessary complexity spreading around the work.

Already at the start, if possible, it's wise to have some visual and logical model to describe what's going on. This type of flowchart may in complex cases become unwieldy and cumbersome.

Here we see a state chart that represents the key flow, color-coded into the various state it may take. In this case, we are significantly improving the understanding of what a technical implementation would represent.

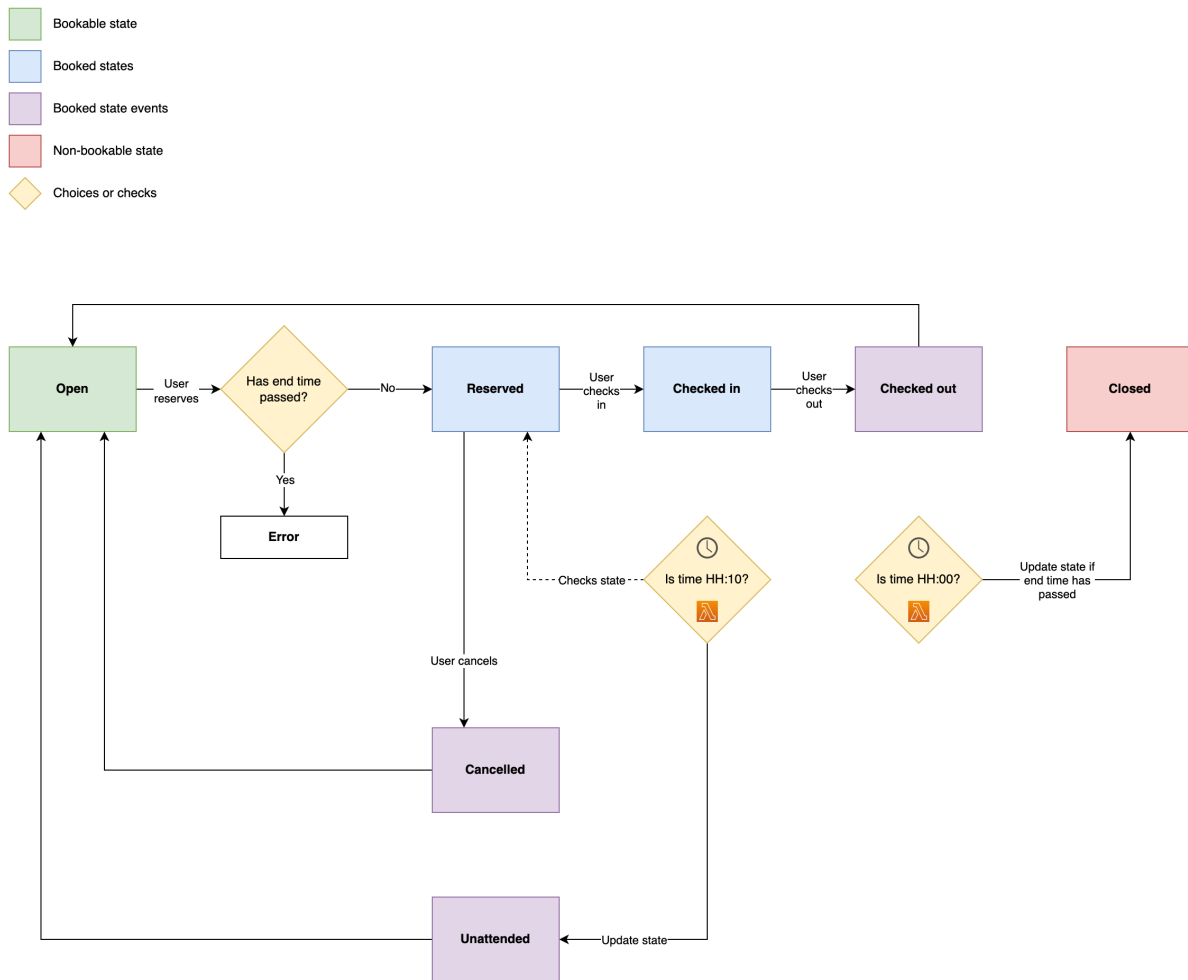


Figure 1.19: Get-A-Room flows.

The diagram as presented here has been modified several times to fit the logical model.

## Deciding on a Cloud Architecture



*Figure 1.20: Illustration from Undraw*

### Going serverless-first

Without being a one-sided advertisement I will still take a few moments to lay out a technical strategy based around [serverless](#). I find that in most cases, given that we can make a technical choice early in the process, it's smart to optimize (as, and if, possible) serverless services.

Some of the well-known benefits of serverless include:

- **Cost-efficiency**, as serverless services can “scale to zero” meaning they cost nothing if no one uses the resources.
- **Less management, maintenance, and toil** since the cloud service provider handles the majority of these parts.
- **Easier to focus on innovation** since the skills required for proper implementation are lower than doing the whole shebang on your own.
- **Makes it easier to adopt modern architecture patterns** like event-driven architecture and microservices because you are essentially forced (well, it just makes a lot more sense) to use API gateways, event buses or topics, and short-lived functions to wire up your solution.

**Warning**

As always there are trade-offs. The most basic ones relate to serverless typically offering fewer configuration options, which *may* be a deal-breaker in some scenarios, and that serverless products indeed become more “black-boxed” and tied to the particular vendor. Dare to question your assumptions on your needs for the above. You’ll probably find that you win a lot more than you lose.

One of my favorite examples of unwanted complexity comes from Yan Cui when trying to answer the oft-mentioned statement “[even simple serverless applications have complex architecture diagrams](#)”. He addresses this perception as a kind of falsity since it has been very common for developers and architects to conflate the logical (business) and physical (infrastructure) views or diagrams of systems, in which case a classical web server hosted on EC2 would look remarkably straightforward, while a Lambda-based microservice solution can quickly start looking rather intimidating.

**Ask yourself what’s more important—having a simple-looking diagram of your application, or actually having a simpler application.**

Because a serverless application might look more complex on paper than its serverful counterpart, only one of these diagrams is a true representation of what you are running in your AWS account. And only one of these diagrams will give you a nasty surprise when you dare to open the box and see what’s inside.

In the spirit of serverless and purpose-oriented functions, the diagram of a serverless solution will much more likely be “[screaming architecture](#)” than an old-school deployment diagram of some network and a virtual machine. A benefit of serverless is that **the diagram itself helps to actually outline what the logical functionality is**, since we can be more expressive and fine-grained with resource usage, for example:

- Dedicated functions for unique business use-cases;
- Logically separated databases for different information objects or Entities;
- Topics or event buses that are solely dedicated to handling the needs of individual functions.

Ergo, the number of cloud resources will indeed grow, but it can also (correctly used) be 100% truthful of what the system performs, rather than the classic 1-4 boxes of “magic providence” that *does all of it*.

## Databases

Our use case does not have a strict need for extensive relational mappings, so we can be broader in our selection and it’s practically fine going with a non-relational solution.

The ideal serverless database option on AWS is [DynamoDB](#), a database that dates back to 2012 and which has gained additional features throughout the years. It’s extremely fast and scalable, and has a stellar reputation and historical performance, but will require a somewhat different mindset when being used.

While there is also [AWS Aurora](#) on the relational spectrum of things, there just does not exist really good relational solutions that are optimized for the cloud that are turn-key the way we expect of serverless.

### Information

DynamoDB is not your grandfather’s database, and it will maybe be challenging to start with if you aren’t already familiar with NoSQL-style databases. Some excellent resources to read up on DynamoDB include:

- [The DynamoDB Guide](#)
- [Single-table vs. multi-table design in Amazon DynamoDB](#)
- [The What, Why, and When of Single-Table Design with DynamoDB](#)
- [Fundamentals of Amazon DynamoDB Single Table Design with Rick Houlihan](#)

## Compute

Once again, looking at the high-level requirements we want something that can run in response to asynchronous events and synchronous API calls. We have no reason to believe it should need to handle long runs, massive amounts of memory, or complex calculations.

The undisputed king of serverless compute platforms has been AWS [Lambda](#) for quite some years now—That’s the same platform we will use here. It satisfies all the conditions mentioned above and is possible to use in a number of languages/runtimes, including TypeScript.

Other viable options could include some of the better-known container services, such as [ECS](#) or [Fargate](#). These however require significantly more plumbing and configuration, plus require some type of containerization actually happening. With Lambda, such work is kept at a minimal level and using [Serverless Framework](#) we’ll continue pushing down the work involved in for example packaging the application.

## Eventing

We need to be able to send (*publish*) and respond (*subscribe*) to events.

The classic choice here would have been [Simple Notification Service \(SNS\)](#). It’s a push-based service, meaning it automatically handles propagating the event to recipients. SNS uses a pay-per-use model and is essentially serverless as the only infrastructure you need is the SNS *topic*.

An optional way of doing this could have been using [Managed Streaming for Apache Kafka \(MSK\)](#). MSK is poll-based, so instead, you have to “ask for” data at intervals to see if something has happened. It hasn’t always been serverless, but nowadays an option has been provided. Nevertheless, MSK makes the most sense if you are already invested in the Kafka space.

In our case, we will opt for still another option: [EventBridge](#). It is superficially a similar type of product as SNS but offers a more convenient developer experience, better event support (such as event catalogs, event archives, and event discovery), better filtering, and the possibility to use more types of subscribers (such as Lambda, SNS, SQS, and API Gateway). Overall it’s a more evolved fit from the more “basic” but still powerful SNS



for our application-centric needs. The EventBridge construct is not a topic, but rather an *event bus*.

From a capability perspective, we might have considered SNS closer if we had very stringent needs around event ordering, exactly-once delivery, or the very high topic count.

### Information

For more reading, please consider checking out this [comparison article by State-Farm](#) or [this article by Ashish Patel](#).

## API

The de facto standard in the modern cloud is that you leverage the native API products to expose your applications, rather than building something on your own with the likes of Express, Fastify, or Kong.

An API gateway acts as the only public interface connected to any other infrastructure, in our case, this would most importantly be our Lambda compute functions which will respond on paths that we have defined in the gateway.

In the case of AWS, the service we are interested in is, unsurprisingly, called [API Gateway](#).

### Is there an option?

It's a bit of a theoretical digression to wander down the path of asking oneself "But *could* I set up another API gateway solution"? In short, the answer is "yes", but then you would most likely (and most effectively) do it in a persistent virtual machine, which is itself hardly serverless, now is it? So, yes, you could do it. But no, we won't.

The API gateway is one of the integration parts that you should leverage maximally when committing to a cloud service provider. Only if you were absolutely sure that you want an open source or multi-cloud solution should you practically consider this option of using a custom API gateway.

## Choosing the type of API Gateway

One of the configuration questions we can look at has to do with which type of (AWS) API Gateway we want. The traditional one is called *REST API* and the newer, lighter-weight one that came out in 2019 is named *HTTP API*. All of this is probably somewhat confusing and you would not be the first one to feel so.

### Information

These are sometimes also called **v1** for REST API and **v2** for HTTP API, which at least for me makes a lot more sense. Do note that both of these types work just fine with actual HTTP(S) and REST; in fact, the naming is just plain bonkers! Read more at <https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html>.

Some of the additional features of the REST API (v1) vs the HTTP API (v2) include:

- AWS X-Ray tracing
- API keys support and per-client throttling
- Caching
- Request validation
- Edge-optimized endpoints

There are certainly benefits to going with the HTTP API as well, for example:

- Much cheaper to run
- They actually do support many (most?) features of v1
- Includes exclusive support for JWT authorization directly on the Gateway (v1 has to use a Lambda authorizer to do the same)
- Has exclusive support for certain integrations

From a non-functional requirements perspective, the items listed in the REST API benefits are things we absolutely want to use, and since price effectively is not a real concern at this small scale, that argument becomes moot.

## Security

As noted in the API section, there exists a few security aspects that we need to keep in mind.

We don't want to double down on security here, but we need to stay mindful of malicious usage. Ways we want to mitigate such usage are:

- Limit scaling and provide a maximum provisioned surface for APIs etc so we cannot get “denial-by-wallet” attacks
- Set permissions to least-privilege
- Secure coding practices
- Use authorization mechanisms, even something simple like an API key
- Use CORS to restrain domains that may call the services

### Information

For a light start on serverless and cloud security, see the following:

- [10 Serverless security best practices](#)
- [Architecting Secure Serverless Applications](#)
- [AWS re:Invent 2021 - Serverless security best practices](#)

## Technical boilerplating

In software architecture, there is the idea (or *strategy*, even) of “[the last responsible moment](#)”—

The **last responsible moment (LRM)** is the strategy of delaying a decision until the moment when the cost of not making the decision is greater than the cost of making it. Design decisions made for software architecture can be among the most important for a software system and they can be among the most difficult to change later.

With traditional software architectures, decisions were made very early in the project. In order to design an evolutionary architecture, it is beneficial to delay a decision until the LRM. This lessens the possibility that a premature decision will be made. Decisions that are made too early are very risky because they may end up being incorrect and then they will result in costly rework.

— *Software Architect’s Handbook* (Ingeno, 2018)

When we talk about software boilerplate, and certainly in the space of doing serverless Node projects, there does indeed exist quite an urgent need to write and set up some degree of boilerplate.

In order to cover concerns like testing, linting, deployment and so on we should make some early (safe) calls. And because we already have some self-constructed constraints, we have a relatively good idea of what type of packages to bring in!

Thus, we will make some high-level definitions already regarding boilerplate:

- [TypeScript](#) as our language
- [Prettier](#) and [ESLint](#) for linting
- [Serverless Framework](#) to deploy and package services
- [Webpack](#) to bundle code
- [AVA](#) for writing and running tests
- [Madge](#), [cfn-diagram](#), and [TypeDoc](#) to generate documentation

- [MikroLog](#) as the logging library
- [Visual Studio Code](#) as the editor
- AWS libraries for their respective services (DynamoDB, EventBridge...)

This should satisfy the majority of our generic concerns in the project. Moreover, each individual service will use pretty much the same set of dependencies and scripts.

None of these deal with highly particular or minute details. If such details start popping up, our architecture should be flexible and pluggable enough to make it possible for us to adapt later.

### Information

If you've been around the block for some time you may have questions about some choices, but you will probably be very familiar with many of the above choices. Architects or technical leaders should always consider familiarity a major plus if a tool supports the functional and non-functional needs it needs to address. However, fight the urge to *always* use the same things—good developers need to try on new tools as often as they can to stay sharp and ahead of the curve.

Most importantly: Right now no one is going to die or complain that we made the above decisions.

Having a clear foundation is something I've seen pay dividends for teams small and large, few and many. As long as we are making humble and unrushed decisions we can make some of these moves on Day 1 without creating too much risk.

## Lambda handler

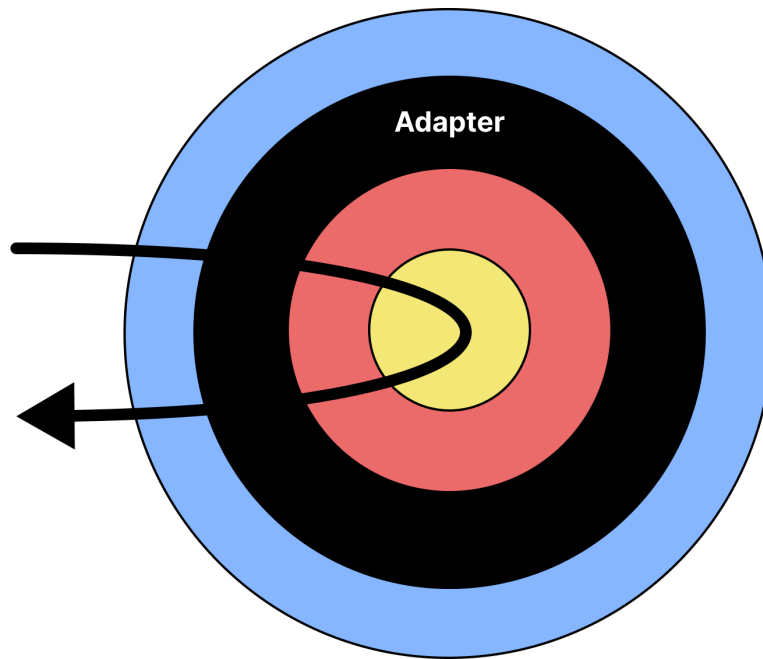


Figure 1.21: Handlers, a type of controller, reside in the Adapter layer.

As I wrote in one of the introductory chapters, a relatively common “misimplementation” is to think of the Lambda handler as the *full extent* of the function. This is all straightforward in trivial contexts, but we gain a significant improvement by being able to remove the pure setup and boilerplate from the business side of things.

The semantic concept of “handler” is somewhat particular to how we talk about *function handlers* or *event handlers*. On a more generic software architecture note, this layer could often be translated into what goes into the “controller” term in the [MVC](#) school. I’ve been known to use the “controller” term and set a dedicated folder in the structure at an earlier stage in my career, but I now refrain from it and go with “[adapters](#)” instead, simply as its an ever wider concept and since we now open for *any* type of driver of our functions.

Enough introduction, let’s go ahead and look at a handler (code/Reservation/Reservation ↵ ↵ /src/infrastructure/adapters/web/ReserveSlot.ts):

```
import {
  APIGatewayProxyEvent,
```

```

    Context,
    APIGatewayProxyResult,
} from "aws-lambda";
import { MikroLog } from "mikrolog";

import { ReserveSlotUseCase } from "../../application/usecases/↵
↵ ReserveSlotUseCase";

import { MissingRequestBodyError } from "../../application/errors/↵
↵ MissingRequestBodyError";
import { UnsupportedVersionError } from "../../application/errors/↵
↵ UnsupportedVersionError";

import { setupDependencies } from "../../utils/setupDependencies";
import { getVersion } from "../../utils/getVersion";
import { setCorrelationId } from "../../utils/userMetadata";

import { metadataConfig } from "../../config/metadata";

/**
 * @description Reserve a slot.
 */
export async function handler(
    event: APIGatewayProxyEvent,
    context: Context
): Promise<APIGatewayProxyResult> {
    try {
        MikroLog.start({
            metadataConfig: { ...metadataConfig, service: "ReserveSlot" },
            event,
            context,
        });
        if (getVersion(event) !== 1) throw new UnsupportedVersionError();

        const body: Record<string, string | number> =
            typeof event.body === "string" ? JSON.parse(event.body) : event.↵
↵ body;
        if (!body || JSON.stringify(body) === "{}")
            throw new MissingRequestBodyError();
        const slotId = body.id as string;
        const hostName = body.host as string;

        setCorrelationId(event, context);

        const dependencies = setupDependencies(metadataConfig("ReserveSlot"↵

```

```

↩️ ));

    const response = await ReserveSlotUseCase(dependencies, {
      slotId,
      hostName,
    });

    return {
      statusCode: 200,
      body: JSON.stringify(response),
    };
  } catch (error: any) {
    return {
      statusCode: 400,
      body: JSON.stringify(error.message),
    };
  }
}

```

At the top, we get the imports, nothing much to add there, and we see that the handler is exported as an async function. This is per Lambda convention.

## Handling the API/event input

I've been somewhat loose on the parameters, as the event is just any old Record (object) but the context is an actual typed AWS context object. This is up for opinion, sure, but I find that the event itself is just easier to deal with when it is untyped and because its structure may significantly change based on which integration mechanism is used—in our case if it's via API Gateway or EventBridge. They ensure this doesn't blow up or bloat *all* of our functions in this service we've made a small `getDTO()` utility function to accurately piece together a fully formed Data Transfer Object from the input. Because it's a utility and not business-oriented we want to avoid any deep considerations or logic in that function, as seen below (code/Analytics/SlotAnalytics/src/infrastructure/↩️ ↩️ `utils/getDTO.ts`):

```

import { AnalyticalRecord } from "../../interfaces/AnalyticalRecord";

/**
 * @description Utility function to get data transfer object from ↩️

```



```

    ↪ either event or request payload.
*/
export function getDTO(event: Record<string, any>): AnalyticalRecord | ↪
    ↪ void {
    if (!event) return;

    // Match for EventBridge case
    if (event?.detail) return createEventBridgeDto(event);

    // Match for typical API GW input
    const body =
        event.body && typeof event.body === "string"
        ? JSON.parse(event.body)
        : event.body;
    if (body) return createApiGatewayDto(body);
    else return;
}

function createEventBridgeDto(event: any) {
    return {
        id: event?.detail?.metadata?.id || "",
        correlationId: event?.detail?.metadata?.correlationId || "",
        event: event?.detail?.data?.event || "",
        slotId: event?.detail?.data?.slotId || "",
        startsAt: event?.detail?.data?.startTime || "",
        hostName: event?.detail?.data?.hostName || "",
    };
}

function createApiGatewayDto(body: any) {
    return {
        id: body.id || "",
        correlationId: body.correlationId || "",
        event: body.event || "",
        slotId: body.slotId || "",
        startsAt: body.startTime || "",
        hostName: body.hostName || "",
    };
}

```

We use the Data Transfer Object, or DTO, simply to carry around a representation of data. We could call this object Input or something if we wanted, but I'll keep it simply as data here.

Back in the handler, you'll see that we start a logger (MikroLog ) so that it's available during our complete function duration (we never know when and if something breaks so let's do that setup at first thing!). See this as the right place for you to set up any other similar components if you have any.

Note also how we wrap the outer perimeter of the handler—being the first thing that is run, after all—in a `try/catch` block. This ensures that we can respond back on the main cases: “All is well”, or “it's a dumpster fire”. More complex examples could absolutely be dynamic and set things like the error code dependent on the error. Once again, here we are keeping at the fundamentals.

## Using unique errors/exceptions

On line 20 we have:

```
if (!data) throw new MissingDataFieldsError();
```

We throw a unique exception (or error) based on the lack of data. Unique errors/exceptions are a good thing to start using, as it also means we can set “identities” on all the failure modes of our application.

## Dependency inversion and injection

On lines 22 and 24 the magic starts happening:

```
const dependencies = setupDependencies();  
  
await AddRecordUseCase(dependencies, data);
```

Notice that there's a dedicated utility function `setupDependencies()` to create various required dependencies. For this particular service, we need only a database.

See [code/Analytics/SlotAnalytics/src/infrastructure/utils/setupDependencies.ts](#) ↩️ :

```
import { Dependencies } from "../../interfaces/Dependencies";

import { createNewDynamoDbRepository } from "../repositories/↵
↵ DynamoDbRepository";
import { makeNewLocalRepository } from "../repositories/LocalRepository↵
↵ ";

/**
 * @description Utility that returns a complete dependencies object
 * based on implementations either "real" infrastructure or mocked ones↵
↵ .
 */
export function setupDependencies(localUse = false): Dependencies {
  const repository = localUse
    ? makeNewLocalRepository()
    : createNewDynamoDbRepository();

  return {
    repository,
  };
}
```

In the other services we use this same pattern but sometimes return more objects depending on the exact needs. In this case, we are receiving either the mock database (for testing and development) or we are getting an instance of DynamoDB. This means we are encapsulating the logic for when we test, rather than spreading this across everything—note that there are still places where we do need to interact prior to tests, but this is the most important bit.

Why bother with this at all? Well, pretty easy. If we want to follow Uncle Bob’s Clean Architecture, as well as following the **D** in **SOLID**, we have to bring lower-level (more concrete; more volatile; less business-oriented) components *into* those that are more business oriented. The magic disconnection we want to create between the infrastructural components (like the database or repository) and the actual use case is now in place.

Note how we just run the use case, injecting it with a set of dependencies making it very easy to replicate and test. We call this pattern **dependency injection** (DI)—more specifically some have called the approach used here “poor man’s DI” or “pure DI”. In my opinion, it’s just the way that makes the most sense: It adds no dependencies, it’s easy to use, and it is completely non-magical. You have this **opinion echoed by people like Khalil Stemmler** as well.

Finally, the correct place to set this “object graph” of dependencies is in what is called the “[composition root](#)”, which in our case is the handler function, just like we see it being used here.

## In closing

So if all these smart patterns are already happening in the handler, are there any bells and whistles left? There sure are! What’s happening in the handler is, no matter how you slice it, completely infrastructural boilerplate. While the `getDTO()` function might need to, well, know, what you actually want, there just isn’t that much “business logic” going here.

Wiring up your handlers this way allows you to be very nimble and totally divorce the connection between the *use case* that orchestrates business logic, and the boilerplate needed to ensure basic conformity with the handler, its API, and all of that. Using DI we also make future testing a lot easier as we can drive the use case with any repository or other dependencies we want.

All in all, for some this might have been obvious and for others, this might be eye-opening, but if nothing else, I definitely saw my own code improve a lot when I started using these patterns.

## The Data Transfer Object

*Shuffling around data can be a mess. The DTO concept helps us tidy things up so they won't look like a toddler's birthday party.*

A bit of a misunderstood gold nugget, which gets a more nuanced use in a DDD context. At its most basic a Data Transfer Object (DTO) could look like this, for example:

```
const myExampleSlotDto = {  
  startTime: "10:00",  
  host: "Mikael",  
};
```

It expresses *something* and it does kind of say it has something to do with a “slot”.

What we don't want with the DTO is to make it other than a basic transferable representation. Why? Because it's just a plain object. It carries state, or data, and nothing else.

### Danger

I have seen examples of where pulling in an ORM or similar to make managing these easier, but it has only driven up complexity instead. Since the DTO is a trivial object to create, just maintaining the definition of the particular DTO and creating it should not be a significant work investment.

Working in a class-oriented fashion, we want to pass around *classes* representing our object (data and behavior together) as long as possible. That's the idea: At some point it needs to serialize into something else, but in that interim nobody should mutate the data.

It might be worthwhile to consider some of the obvious problems of POJOs (Plain Old Javascript Objects; or POCOs or whatever pertains to your language). At this point, you are well aware that what we are working towards is *domain driven microservices*. Contrary to being domain driven is being *anemic*. Wikipedia has this terse and good description:

The **Anemic domain model** is a programming [Anti-pattern](#) where the [domain](#)

[objects](#) contain little or no [business logic](#) like validations, calculations, rules, and so forth. The business logic is thus baked into the architecture of the program itself, making refactoring and maintenance more difficult and time-consuming.

— [Wikipedia: Anemic domain model](#)

Let's look at that a bit more in the words of Martin Fowler:

**The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing.** There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. **The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.** Indeed often these models come with design rules that say that you are not to put any domain logic in the domain objects. Instead, there are a set of service objects which capture all the domain logic, carrying out all the computation and updating the model objects with the results. These services live on top of the domain model and use the domain model for data.

**The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together.** The anemic domain model is really just a procedural style design, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk. What's worse, many people think that anemic objects are real objects, and thus completely miss the point of what object-oriented design is all about.

Now object-oriented purism is all very well, but I realize that I need more fundamental arguments against this anemia. In essence, **the problem with anemic domain models is that they incur all of the costs of a domain model, without yielding any of the benefits.** The primary cost is the awkwardness of mapping to a database, which typically results in a whole layer of O/R mapping.

— [Martin Fowler: AnemicDomainModel](#)

A grave judgment indeed, though it is correct too.

## Data vs behavior and JavaScript

As is probably very clear, we can't really push a class through our API, but we can push out a serialized representation of a plain object. So the need to, at some point, boil our classes with data and behaviors and our domain logic into a representation does exist and that's fine.

On the blog [The Domain Driven Design we find a set of useful tips](#) that start hinting at what we'll see a lot more of in the Tactical DDD part of this book:

- Use private setters. If your properties are defined by the Client directly you will lose the chance to use Domain Events and you will have to validate your Entities by external methods.
- Always validate the status of your entities, your Entities must self-validate.
- Avoid constructors without parameters. Certainly, your objects will need some initialization data to maintain a valid state.
- Think long before you create a Domain Services, they are used as real Silver Bullets by the developers but end up being the biggest causes of the *Anemic Model*.
- Be careful with ORM, they are responsible for creating Domain Objects automatically, producing real containers of public setters and public getters, which leads to an *Anemic Model*.

Let's not steal the thunder from the later sections on Entities, but maybe you are seeing a pattern here: Really do avoid objects that are mutable and that separate data from behavior, at least internally and logically within your own system or service.

Prefer passing instances of classes of Entities or Value Objects rather than DTOs. DTOs do make it easier to do "dumb objects" and are much more portable (the portability is the reason we want them in the first place), especially if you are integrating, say with APIs, but then you lose the behavior. What is the driving need: The data or the behavior? Choose wisely.

**In closing**

The Data Transfer Object is an excellent way to transport data. The battle is around using DTOs at strategic places where it makes sense. We'll see more use of it later, but as some heuristics that make sense to me we can find that the DTO is used as early (example: API input), and as late (as the response going back to the Lambda handler i.e. adapter layer) as possible.



## Error handling

Don't get me wrong: Your code *will* break, so let's just aggressively prepare for handling errors and exceptions rather than pulling out the ukulele and singing Kumbaya.

### Information

Since we are using TypeScript, which is a superset of JavaScript, I'll just take a moment to note that in this language an Error and an Exception are for all intents and purposes the same. If you use another language, feel free to translate the advice "generally" back into your context. Read more at <https://stackoverflow.com/questions/the-difference-between-error-and-exception-in-javascript>

If you are a JavaScript/TypeScript/Node developer, you are most certainly aware of `throw ↵`  
`↵ new Error("omg it broke again! :/")` — Throwing an error with an optional message.

In most languages, there seems to be a more well-established convention to create special errors or exceptions that we can raise when something goes south. I have no hard numbers on this, but I am happy to support that we should indeed do the same for our applications!

A pattern I am using is the one we can see exemplified here (`code/Analytics/SlotAnalytics ↵`  
`↵ /src/application/errors/MissingDataFieldsError.ts`):

```
import { MikroLog } from "mikrolog";

/**
 * @description Used when data input is missing required fields.
 */
export class MissingDataFieldsError extends Error {
  constructor() {
    super();
    this.name = "MissingDataFieldsError";
    const message = `Missing one or more required fields!`;
    this.message = message;

    const logger = MikroLog.start();
```

```
        logger.error(message);  
    }  
}
```

This basic variant can be copied as a foundation for other unique errors as well. I've kept them in the same file ({folders}/application/errors/errors.ts) as well as distinct files in that sort of structure. I have no heavy opinion and I've found that as long as we keep them in a clear application-level location it's all workable regardless.

For certain types of errors you may want to take in input, like so:

```
constructor(inputMessage: string) {  
    super(inputMessage);  
    this.name = 'DemoError';  
    const message = `Something went wrong: ${inputMessage}`;  
    this.message = message;  
  
    const logger = MikroLog.start();  
    logger.error(message);  
}
```

Using custom errors is something that is going to help you significantly as you'll begin to operate your solution.

## Testing

The testing approach used here is (for the theoretically minded) in the mostly “[classical](#)” [testing camp](#), which means that we test behaviors and expected output, but nothing regarding the actual implementation details *within* the thing we test.

Like much programming, **writing good tests should depend on abstractions (interfaces), not concretions**. We can use *test doubles* (mocks, stubs, spies, fakes) to work for us, rather than using infrastructure and implementations that might be problematic. Examples of that could be external services, unfinished services, anything going over a network, persistence technologies, and so on.

Finally, our maxim is to: Write *positive* tests for the “happy flows” and *negative* tests for the “unhappy flows”.

### Positive tests

Positive tests are the tests you are most likely already familiar with. These tests verify that some functionality is delivering as per expectations. Simple as that!

See `code/Analytics/SlotAnalytics/__tests__/usecases/AddRecord.test.ts`.

```
import test from "ava";

import { AddRecordUseCase } from "../../src/application/usecases/↵
↵ AddRecordUseCase";

import { setupDependencies } from "../../src/infrastructure/utils/↵
↵ setupDependencies";

// @ts-ignore
const dependencies = setupDependencies(true);

/**
 * POSITIVE TESTS
 */

test("It should add a record", async (t) => {
  const { repository } = dependencies;
  const data = {
```

```
    id: "abc1234",
    correlationId: "qwerty3901",
    event: "CHECKED_IN",
    slotId: "ldkj2h921",
    startsAt: "20220501",
    hostName: "Somebody",
  };

  await AddRecordUseCase(dependencies, data);

  // @ts-ignore
  const result = repository.dataStore[0];

  t.deepEqual(result, data);
});
```

Most testing libraries are somewhat similar, and while AVA is a less-used framework than say Jest, it retains the same general flow. The top level is for the imports we need, and we'll also call `import test from 'ava';` so that we have access to AVAs utilities.

Let's understand what we are doing by seeing a minimal test in AVA and what that would be:

```
test("It should verify that 1 is 1", (t) => {
  t.is(1, 1);
});
```

It's pretty basic—The above just verifies that the left-side value (1) is the right-side value (2). For many tests, using `is()` or `deepEqual()` (for comparing objects) will be enough.

### Information

Refer to <https://github.com/avajs/ava/blob/main/docs/01-writing-tests.md> for more detailed instructions on how to use AVA.

Remember, unit tests should be easy to understand, stable, and fast to run, and both of those methods make that possible. At the heart of good code and good tests are deterministic input and output. Depending on the exact thing you are testing the nature of

the input/output will be somewhat fluid. For a use case, the input is a set of dependencies and the required input data. The output is the Data Transfer Object that represents the data that will be pushed through the physical API (our Lambda handler, in terms of layers).

## What happens within the test?

Essentially what happens inside of the test itself is whatever is required setup to actually perform our test. This might look different depending on the nature of what you are doing. I sometimes hear that testing would be hard or messy, but in the majority of cases I find the following to be true:

- A developer makes for an excellent tester, in terms of skills needed to perform the job. The opposite is not true.
- Well-structured and well-written code takes trivial effort to test.
- Certain types of tests may be harder to verify. An example: Checking that `console.log()` was called. To deal with this we can:
- Accept that not everything is important to verify. A test (and its reported coverage) can be good enough without having to “extract” verification out into the test scope. Basically, if it ran without breaking, that might sometimes be good enough.
- Accept that you might need to implement more complicated test types like [spies](#). I believe that these should be avoided as often as possible, since 1) Either make your code easier to verify (i.e. rethink, redesign, refactor) or 2) Do it this time and don’t do it again.

## Negative tests

Every non-trivial function, method, or class that is tested should throw a controlled error, ideally a unique one so that you can easily separate them and find out where something went wrong. This should also imply that your user gets that same information, making their life easier too.

The negative tests should map at least to each unique error thrown. For example, given the following...

```
function MessagePrinter(message: string) {  
  if (!message) throw new MissingMessageError();  
  if (message.length <= 5) throw new MessageLengthTooShortError();  
  console.log(message);  
}
```

...it would be advisable to add two separate tests for these “unique” errors in the *negative* section.

### Information

A useful related pattern that deserves to be mentioned and inspired by is the “[guard clause](#)”, a pattern in which we return early on pre-conditions and can effectively cut back on `if-else` statements. The reason I am bringing it up is that we should strive to keep functions and methods as flat (on the left margin) as possible, meaning more readable (and possibly testable!) code.

## In closing

Testing starts at 100%. After you get to—our near 100%—you’ll begin to see the outlines of some of the additional, less obvious checks and tests you have to have.

More importantly for the current context, writing code in a DDD + CA project is going to accelerate your journey to good, testable code because it’s already practically in the same territory as ATDD (acceptance test-driven design).

Ensuring that you structure code in a way that cannot be misunderstood and misused, as well as having complex domain classes (like Entities) that can never be in invalid states is going to be a complete game-changer for developers who are frustrated with poor quality and testing practices.

If good code is what you want, DDD is one very concrete way to get there.

## Lambda authorizer

The Lambda authorizer is somewhat convoluted due to the particulars of how AWS' format works.

Here's the code at `code/Reservation/SlotReservation/src/infrastructure/authorsizers/Authorizer.ts`:

```
import { APIGatewayProxyResult, AuthResponse } from "aws-lambda";

import fetch, { Response } from "node-fetch";

import { AuthorizationHeaderError } from "../../application/errors/AuthorizationHeaderError";
import { InvalidVerificationCodeError } from "../../application/errors/InvalidVerificationCodeError";
import { MissingSecurityApiEndpoint } from "../../application/errors/MissingSecurityApiEndpoint";

const SECURITY_API_ENDPOINT_VERIFY =
  process.env.SECURITY_API_ENDPOINT_VERIFY || "";

/**
 * @description Authorizer that will check the `event.Authorization` header
 * for a slot ID (separated by a pound sign, or "hash tag") and a verification code
 * and validate it against the Security API.
 *
 * @example `Authorization: b827bb85-7665-4c32-bb3c-25bca5d3cc48#abc123` header.
 */
export async function handler(event: EventInput): Promise<AuthResponse> {
  {
    try {
      // @ts-ignore
      if (event.httpMethod === "OPTIONS") return handleCors();
      if (!SECURITY_API_ENDPOINT_VERIFY) throw new MissingSecurityApiEndpoint();

      const { slotId, verificationCode } = getValues(event);
      if (!slotId || !verificationCode) throw new AuthorizationHeaderError();
    }
  }
}
```

```

    // Verify code
    const isCodeValid = await validateCode(slotId, verificationCode);
    if (!isCodeValid) throw new InvalidVerificationCodeError();

    return generatePolicy(verificationCode, "Allow", event.methodArn, "↔");
  } catch (error: any) {
    console.error(error.message);
    const { slotId } = getValues(event);
    const id = slotId ? slotId : "UNKNOWN";
    return generatePolicy(id, "Deny", event.methodArn, {});
  }
}

/**
 * @description Get required values
 */
function getValues(event: EventInput) {
  const header = event.headers["Authorization"] || "";
  const [slotId, verificationCode] = header.split("#");
  return {
    slotId,
    verificationCode,
  };
}

/**
 * @description CORS handler.
 */
function handleCors() {
  return {
    statusCode: 200,
    headers: {
      "Access-Control-Allow-Headers": "Content-Type",
      "Access-Control-Allow-Credentials": true,
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Methods": "OPTIONS,POST,GET",
      Vary: "Origin",
    },
    body: JSON.stringify("OK"),
  } as APIGatewayProxyResult;
}

/**

```



```

* @description Creates the IAM policy for the response.
* @see https://docs.aws.amazon.com/apigateway/latest/developerguide/↵
↵ api-gateway-lambda-authorizer-output.html
*/
const generatePolicy = (
  principalId: string,
  effect: string,
  resource: string,
  data: string | Record<string, any>
) => {
  return {
    principalId,
    context: {
      stringKey: JSON.stringify(data),
    },
    policyDocument: {
      Version: "2012-10-17",
      Statement: [
        {
          Action: "execute-api:Invoke",
          Effect: effect,
          Resource: resource,
        },
      ],
    },
  };
};

/**
* @description Validate a code.
*/
async function validateCode(
  slotId: string,
  verificationCode: string
): Promise<boolean> {
  return await fetch(SEcurity_API_ENDPOINT_VERIFY, {
    body: JSON.stringify({
      slotId,
      code: verificationCode,
    }),
    method: "POST",
  })
    .then((response: Response) => response.json())
    .then((result: boolean) => {
      if (result === true) return true;
    });
}

```

```

        return false;
    })
    .catch((error: any) => {
        console.error(error.message);
        return false;
    });
}

/**
 * @description Very basic approximation of the
 * required parts of the incoming event.
 */
type EventInput = {
    headers: Record<string, string>;
    httpMethod: "GET" | "POST" | "PATCH" | "OPTIONS";
    methodArn: string;
};

```

Most of its contents are pure boilerplate that you can copy between projects to your heart's content.

The particulars in the handler are:

```

if (event.httpMethod === "OPTIONS") return handleCors();

if (!SECURITY_API_ENDPOINT_VERIFY) throw new MissingSecurityApiEndpoint↵
    ↵ ();

const { slotId, verificationCode } = getValues(event);
if (!slotId || !verificationCode) throw new AuthorizationHeaderError();

// Verify code
const isCodeValid = await validateCode(slotId, verificationCode);
if (!isCodeValid) throw new InvalidVerificationCodeError();

return generatePolicy(verificationCode, "Allow", event.methodArn, "");

```

First of all, if this is a call from a source where CORS might be an issue we handle that case. Next, we ensure there is a constant set for our endpoint, or we throw an error. This one is serious if we hit this one, but at least we'll know it's a configuration issue and nothing else.

Then we see how the implementation expects an `Authorization` header to have a specific format with `{SLOT_ID}#{VERIFICATION_CODE}`. Therefore we'll split by the hash, check their presence of them, and throw an error if either is missing.

The actual validation then is nothing more than calling the endpoint that has been configured. If it is incorrect we return an error indicating this. If all is good, then we'll hit the positive branch of `generatePolicy()`.

## API schema

You may remember that back in the strategic DDD section—during the context mapping to be exact—we decided that the `VerificationCode` subdomain and `Reservation` subdomain would have a `Published Language` relationship. Now is the time to address that fact.

The reason we have API schemas is that they serve as human- and machine-readable documentation, and are far better and more portable than any old Word file, Google Doc, or Sharepoint site. **Using a standardized API specification is the way you should document** and it's not that hard either, once you start getting the hang of it!

## Choosing AsyncAPI

In a modern, hybrid landscape where we have both REST and event-driven APIs, it makes sense to reach for the new kid on the block, [AsyncAPI](#), rather than [OpenAPI](#). While OpenAPI is well-known and has a proven track record, it won't cut it in the more confusing technical landscapes of today. With AsyncAPI we get the possibility to not just get some [smooth tooling](#), but we'll also be able to actually [describe a system that has events and asynchronous responses](#).

AsyncAPI is an open source initiative that seeks to improve the current state of Event-Driven Architectures (EDA). Our long-term goal is to make working with EDAs as easy as it is to work with REST APIs. That goes from documentation to code generation, from discovery to event management. Most of the processes you apply to your REST APIs nowadays would be applicable to your event-driven/asynchronous APIs too.

— <https://www.asyncapi.com/docs/tutorials/getting-started>

Before we move on, please know that AsyncAPI does not reinvent the wheel—it actually has some support for both OpenAPI and [JSON Schema](#) for their respective powerful features.

### Information

It's worth mentioning that indeed the API documentation experience for what we have here, a *de facto* RESTish API, could have been done with OpenAPI 3 just as well. We have to approach this in a bit of a hackish way to make it semantic within the boundaries of the standard. AsyncAPI is expected to bring major improvements over time to this overall area, though unclear exactly which set of such improvements, in the next major version. Regardless: You should definitely put familiarity with this new specification high on your list of things to look into.

Anyway, time to get to it.

## Writing the schema

The schema looks like the below. Notice the top-level objects:

- **info**: General useful information like the contact person for the service, the service's name, and description...
- **externalDocs**: Any links to external documentation
- **servers**: Where can you actually run the events and requests?
- **channels**: The addressable components, or “things”, as it were
- **components**: Reusable objects, such as messages
- **schemas**: Definition of inputs and outputs

```
{
  "asyncapi": "2.6.0",
  "info": {
    "title": "VerificationCode",
    "version": "1.0.0",
    "contact": {
      "name": "Sam Person",
      "url": "https://acmecorp.com/docs#owner",
      "email": "sam@acmecorp.xyz"
```

```

    },
    "description": "`VerificationCode` generates and validates codes ↵
    ↵ for slot reservations."
  },
  "externalDocs": {
    "description": "Confluence documentation",
    "url": "https://acmecorp.com/VerificationCode/docs"
  },
  "servers": {
    "production": {
      "url": "https://RANDOM.execute-api.REGION.amazonaws.com/prod",
      "protocol": "http",
      "description": "Production endpoint."
    }
  },
  "channels": {
    "generateCode": {
      "publish": {
        "message": {
          "$ref": "#/components/messages/GenerateCode"
        },
        "bindings": {
          "http": {
            "type": "request",
            "method": "POST"
          }
        }
      },
      "subscribe": {
        "message": {
          "$ref": "#/components/messages/GenerateCodeResponse"
        },
        "bindings": {
          "http": {
            "type": "request",
            "method": "POST"
          }
        }
      }
    },
    "removeCode": {
      "publish": {
        "message": {
          "$ref": "#/components/messages/RemoveCode"
        }
      }
    }
  }
}

```

```

        "bindings": {
            "http": {
                "type": "request",
                "method": "POST"
            }
        },
        "subscribe": {
            "message": {
                "$ref": "#/components/messages/RemoveCodeResponse"
            },
            "bindings": {
                "http": {
                    "type": "request",
                    "method": "POST"
                }
            }
        },
        "verifyCode": {
            "publish": {
                "message": {
                    "$ref": "#/components/messages/VerifyCode"
                },
                "bindings": {
                    "http": {
                        "type": "request",
                        "method": "POST"
                    }
                }
            },
            "subscribe": {
                "message": {
                    "$ref": "#/components/messages/VerifyCodeResponse"
                },
                "bindings": {
                    "http": {
                        "type": "request",
                        "method": "POST"
                    }
                }
            }
        },
        "components": {

```

```

"messages": {
  "GenerateCode": {
    "name": "GenerateCode",
    "title": "GenerateCode",
    "summary": "Generate a verification code for a provided slot ID↵
↵ .",
    "contentType": "application/json",
    "payload": {
      "$ref": "#/components/schemas/SlotIdInput"
    }
  },
  "GenerateCodeResponse": {
    "name": "GenerateCodeResponse",
    "title": "GenerateCodeResponse",
    "summary": "Returns the name of the customer using the given ID↵
↵ .",
    "contentType": "application/json",
    "payload": {
      "$ref": "#/components/schemas/GenerateCodeResponse"
    }
  },
  "RemoveCode": {
    "name": "RemoveCode",
    "title": "RemoveCode",
    "summary": "Remove a verification code for a provided slot ID."↵
↵ ,
    "contentType": "application/json",
    "payload": {
      "$ref": "#/components/schemas/SlotIdInput"
    }
  },
  "RemoveCodeResponse": {
    "name": "RemoveCodeResponse",
    "title": "RemoveCodeResponse",
    "summary": "Returns the name of the customer using the given ID↵
↵ .",
    "contentType": "application/json",
    "payload": {
      "$ref": "#/components/schemas/RemoveCodeResponse"
    }
  },
  "VerifyCode": {
    "name": "VerifyCode",
    "title": "VerifyCode",
    "summary": "Verify a provided code for a given slot ID.",

```



```

        "contentType": "application/json",
        "payload": {
            "$ref": "#/components/schemas/VerifyCodeInput"
        }
    },
    "VerifyCodeResponse": {
        "name": "VerifyCodeResponse",
        "title": "VerifyCodeResponse",
        "summary": "Returns the name of the customer using the given ID↵
↵ .",
        "contentType": "application/json",
        "payload": {
            "$ref": "#/components/schemas/VerifyCodeResponse"
        }
    },
    "schemas": {
        "SlotIdInput": {
            "type": "object",
            "description": "The slot ID to create or remove a verification ↵
↵ code for.",
            "properties": {
                "slotId": {
                    "type": "string",
                    "description": "Slot ID"
                }
            },
            "additionalProperties": false
        },
        "VerifyCodeInput": {
            "type": "object",
            "description": "The slot ID and verification code to verify ↵
↵ together.",
            "properties": {
                "verificationCode": {
                    "type": "string",
                    "description": "8-character verification code"
                },
                "slotId": {
                    "type": "string",
                    "description": "Slot ID"
                }
            },
            "additionalProperties": false
        }
    }
}

```

```
"GenerateCodeResponse": {  
  "type": "string",  
  "description": "Verification code"  
},  
"RemoveCodeResponse": {  
  "type": "null",  
  "description": "Returns HTTP status `204 No content`"  
},  
"VerifyCodeResponse": {  
  "type": "string",  
  "description": "8-character verification code"  
}  
}  
}
```

### Information

Note that you can use both JSON and YAML formats when writing the schema, but because JSON is typically more portable I've gone with that here.

### Success

Absolutely do try out some of the [nice tutorials they have prepared](#).

With some [good IDE tooling](#), you should be able to get a good experience already in your IDE, but you can also copy the schema above into the online [AsyncAPI Studio](#) and get a visualized and live result to work on. Fancy indeed!

Even cooler is that that tooling is available for you to use as well, so you can generate these views during your Continuous Integration step for example.

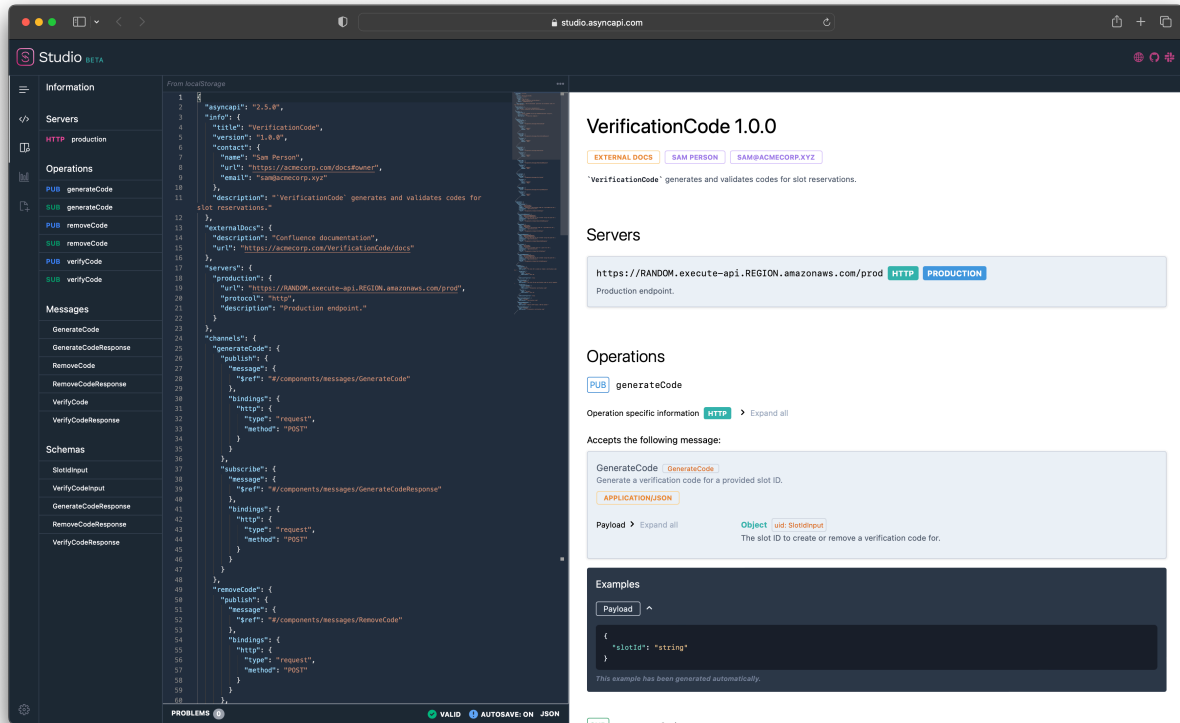
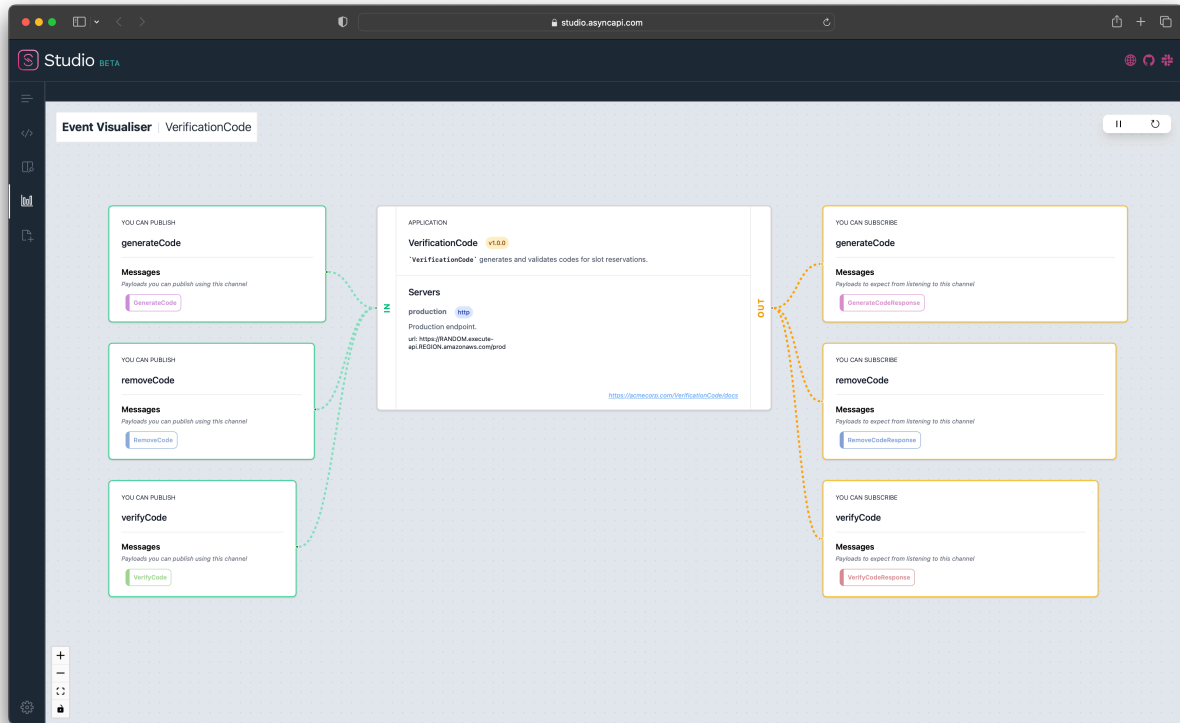


Figure 1.22: Split view with all kinds of information/navigation to the far left, the schema in the middle, and the visualization on the right.



*Figure 1.23: You can even follow the flows of “published” and “subscribed” events. Because we are doing HTTP this of course translates conceptually to requests and responses.*

And with this work behind us, we have now come through on our promise to have a “published language” that describes how our service works. In fact, we have done this even before building the actual API in the first place!

Brilliant.

## Tactical DDD

It's high time to put those "tactical" patterns in Domain Driven Design to work, breathing life into our domain model.

This section describes key passages of the code and the overall implementation rather than rehashing the *complete* and rather extensive code base.

DDD has deep ties to object-oriented programming and its ways of thinking. We will see examples of how encapsulation, inheritance, abstractions, and classes allow us to write more well-structured code so that we can uncover maximum benefits from DDD.

We will inspect and analyze some of the usages of all of the tactical patterns:

- Modules
- Factories
- Repositories
- Services
- Entities
- Aggregates
- Value Objects
- Domain Events

### Warning

In the "blue and red books" you will see that the order that these are presented is different. My reason for presenting the patterns in this particular order is because I believe it makes for easier guiding, starting with those that are more basic and moving into the deeper domain objects halfway through.

## Modules

*The least-discussed but structurally most fundamental pattern concerns our Modules and the structure these put on our work.*

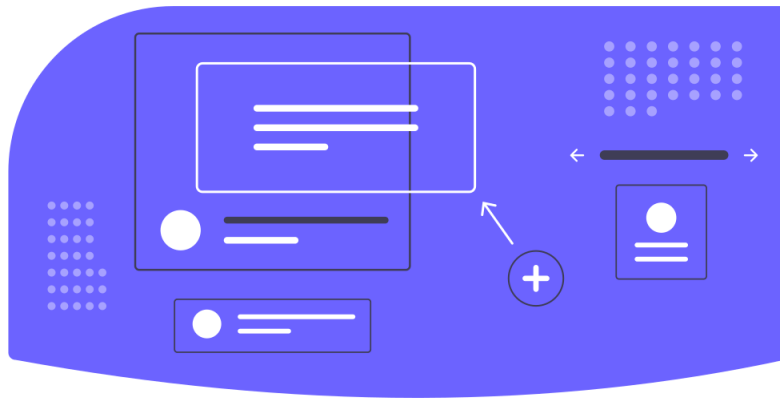


Figure 1.24: Illustration from Undraw

### Success

**TL;DR: Modules** use ubiquitous language to express the technical side of the domain. There is no “one way” to use **Modules**, it’s all mostly about naming and organizing your code. Organize your code in **Modules** by using concepts like namespaces, classes, folder structure, and even how you split responsibilities across microservices. The goal is to accurately express the domain by the way you have structured and named things.

In the DDD context, we use Modules as a logical construct to segregate between concerns when we technically implement our domain model. Modules should precede the Bounded Contexts because Modules typically reside in the same codebase and reflect the *logical model* of our domain. Dividing logical wholes into separate Bounded Contexts can cause problems (Vernon 2013, p. 344). One example of a valid use is to reach for Modules if you need to create a second model in the same Bounded Context (Vernon 2016, p.50).

Again, Modules are local to the code, while Bounded Contexts may constitute one or more

logical solutions. Yet these both (in particular Modules) share the common trade-offs of public interfaces:

[E]ffective modules are deep: a simple public interface encapsulates complex logic. Ineffective modules are shallow: a shallow module’s public interface encapsulates much less complexity than a deep module.

— *Learning Domain Driven Design* (Khononov 2021, p. 223)

This is the most basic tactical pattern, yet it at heart is all about classic programming concepts like “[high cohesion, low coupling](#)” and, as per DDD, expressing the Domain through the naming and functionality.

With all this said, though, the Module pattern itself is not descended from DDD; it is a common pattern that has been around probably since the start of at least object-oriented programming. We use this pattern to **encapsulate** and, sometimes, **name** some part of our application. This can be done by language-specific mechanisms and/or by structuring our code in files and folders.

## Demystifying Modules

In terms of ontology, a **Module can be a namespace or a package, depending on the language** that you are using. For our example code, using TypeScript, [there do exist mechanisms to handle this](#), but they are not completely idiomatic to how the language is typically used. Instead, we will have to do this only at the file and folder level. Generally, it does make sense that we should also see the structure and folders as a related effect of our Modules. Therefore Modules are not simply only a technical matter, but a logical matter.

### Information

See for example [this article by DigitalOcean for more on how the Module pattern works](#) in JavaScript/TypeScript.

Much of DDD wisdom and attempts at concretely structuring files in a DDD-leaning sense will address why one of the most basic tactical things we can implement is packaging

by Module (or features) rather than by layers. You'll perhaps already have experience seeing how many trivial or common projects will use the layered, format-based approach, segmenting folders into their respective types (especially common in front-end projects) or using vague, non-descriptive categories such as helpers. This makes it very hard to understand how objects and functions relate and what their respective hierarchies are. It also becomes hard to discern the domain logic from the overall structure, the Module names, and their usage. All that becomes much easier with Modules.

### Information

For more, from a non-DDD angle, read [this article about why packaging by feature is better than packaging by layers](#). I am also, as an Uncle Bob fanboy, liking [Screaming Architecture](#) quite a bit.

## Structuring for a Module pattern

In DDD you'll hear a lot of arguments against importing outer-level objects (such as services) into deeper-level objects, such as Aggregates. This is sound advice, generally speaking. If we start importing left-right-and-center without discipline we will end up in a really bad place!

**It's worth noting that DDD itself is not prescriptive at all regarding how to set your file structure.** In fact, there is practically nothing in Evans' book about this. Obviously, it does make sense to somehow reflect the "methodology" in how the actual code is organized, but DDD won't save you here, I'm sad to say. Clean Architecture, though, *will* paint a much more exact idea, itself borrowing from the [Ports and Adapters](#) (or *onion/hexagonal architecture*) notion.

There are several examples out in the wild that aim to present various individuals' takes on DDD, in particular, and some Clean Architecture, generally. Sometimes you may find these combined as I have done, but that's typically not quite as common.

Reasons I don't necessarily like some of the other examples out there, include:

- Overbearing amount of boilerplate and folders.



- Typically oriented toward monolithic use cases or indistinct deployment models.
- Related to the above: Over-modularization, where I believe microservices themselves should be the first module boundary.
- Use of decorators; is something that is not standardized in TypeScript (Vandenkam 2021, p. 197-198).
- Use of inversion of control (IoC) libraries and dependency injection (DI) containers/libraries rather than using the language features provided, or simply using regular object-oriented programming. These needs can be handled without external library dependencies by using higher-order functions or passing in dependencies in a functional way.
- Intricate uses of more complex ideas like monads (such as `Either` and `Left/Right`) which adds a higher threshold than necessary for people to start getting value from tactical DDD.

### Information

All of these concerns are addressed and “taken care of” in the example code that goes with this book.

Taking DDD and CA together, we get a pretty powerful toolbox. You should understand that many examples are based on monolithic applications, something I personally very rarely work on. The example here addresses a microservice perspective. The bounded context itself is the main feature, so to speak.

Clean Architecture also changes the structure and naming a bit. We will base our core understanding of application structuring on Clean Architecture and its respective nomenclature, as it’s more prescriptive than regular DDD.

### Warning

As always, “Don’t try to be clever”. DDD is hard enough as it is, so it makes sense to be pragmatic and functional.

## High-level project organization

In our case, the principal module structure for code is:

### Reservation (core subdomain)

- `code/Reservation/Reservation`: The reservation solution and Bounded Context (core subdomain)
- `code/Reservation/Display`: The display solution and Bounded Context (supporting subdomain)

### Analytics (generic subdomain)

- `code/Analytics/Analytics`: The analytics solution and Bounded Context

### Security (supporting subdomain)

- `code/VerificationCode/VerificationCode`: The verification code solution and Bounded Context

Here we've almost completely nailed the 1:1 relationship between Bounded Context and subdomain, as well as have a top-level modularization of solutions/code into these.

## Using Clean Architecture as our foundation

The “Clean Architecture” is a relatively well-known variant of the onion/hexagonal/ports-and-adapters school of architecture.

Many have tried and many have failed when it comes to setting up a folder structure for DDD. For my part, I've found that Robert C. Martin's “clean architecture” is a better (and simpler!) elaboration of where so many developers have tried to find a way. It's not magic, just a very nice mapping (and [blog article](#), and [book](#) for that matter!).

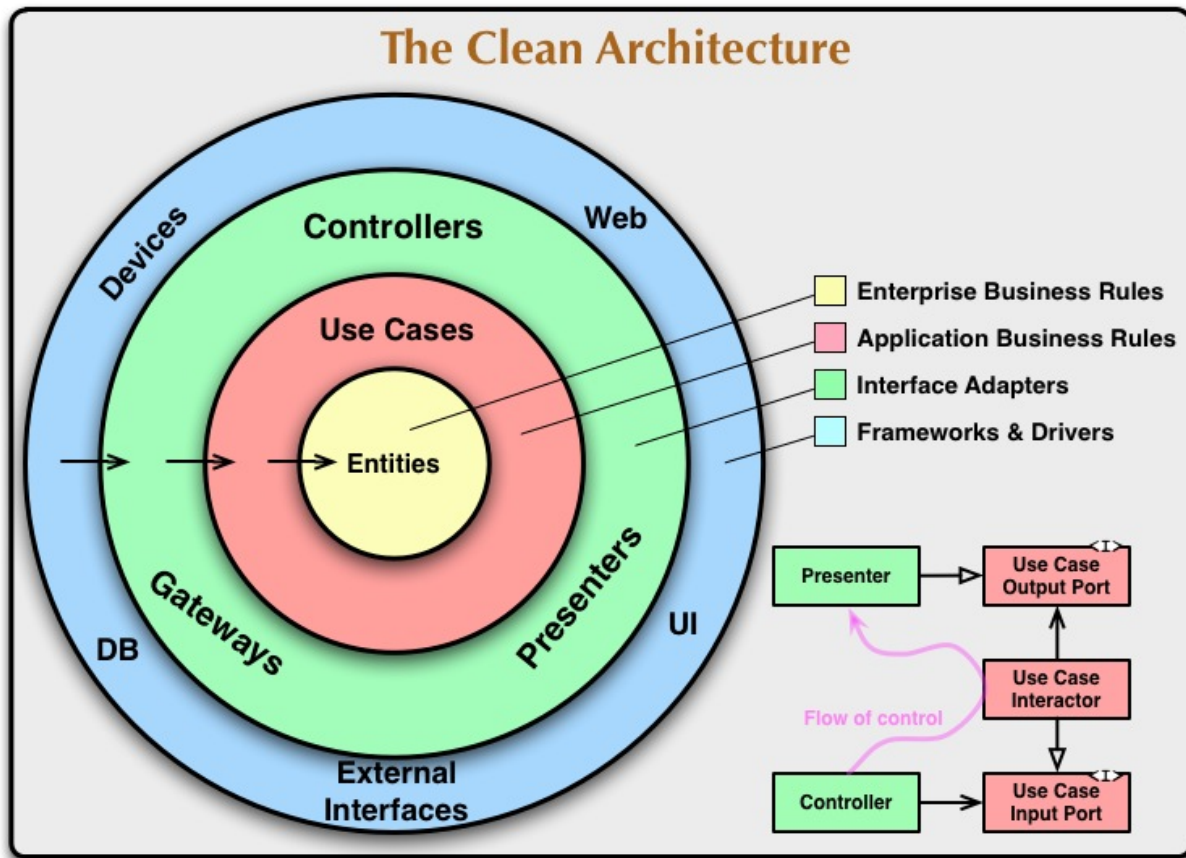


Figure 1.25: From Robert C. Martin's blog. "The Clean Architecture", 10 August 2012, <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

I find it the most immediately effective and neat variant of these, as it:

- Introduces very little in terms of novel concepts;
- Is almost directly compatible with how DDD envisions structure in the software realm;
- Powerfully exploits the *dependency rule* for well-working and testable software.

Robert Martin writes about the *dependency rule* like this:

The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.

The overriding rule that makes this architecture work is *The Dependency Rule*. This rule says that *source code dependencies* can only point *inwards*. Nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle must not be mentioned by the code in the inner circle. That includes functions, and classes, variables, or any other named software entity.

By the same token, data formats used in an outer circle should not be used by an inner circle, especially if those formats are generated by a framework in an outer circle. We don't want anything in an outer circle to impact the inner circles.

— [Robert C. Martin: The Clean Architecture](#)

The intention with all of these ideas for how to structure an application is all well-meaning, but I've also seen and reflected on how a higher level of "layers" or "circles" can complicate things quite quickly.

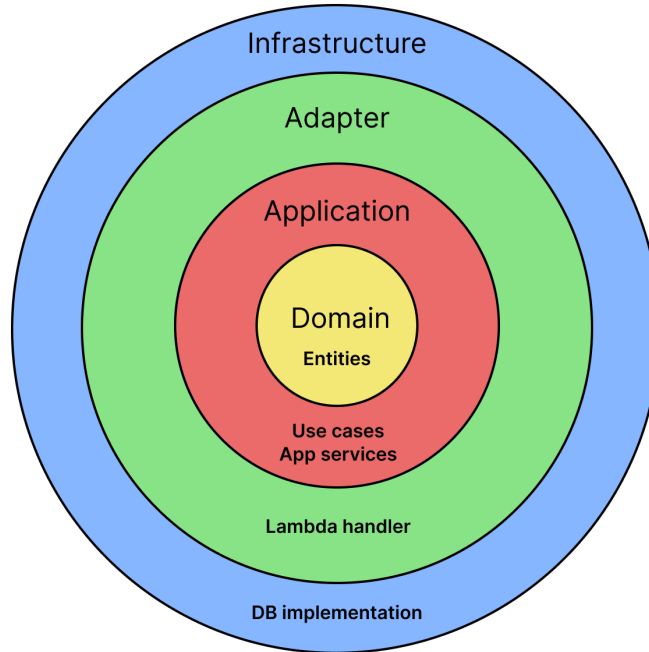
Let's at least look at the levels and some examples of what would go into each, respectively.

- **Entities:** "Business objects of the application"
- **Use cases:** "Use cases orchestrate the flow of data to and from the Entities, and direct those Entities to use their enterprise wide business rules to achieve the goals of the use case"
- **Interface adapters:** "A set of adapters that convert data from the format most convenient for the use cases and Entities, to the format most convenient for some external agency such as the Database or the Web"
- **Frameworks and Drivers:** "Where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside where they can do little harm"

Ultimately: **The farther in something is, the less likely it is to change. Any inner layers must not depend on the outer layers.**

## Adapting the Clean Architecture

I will apply a set of small modifications to this just to juice it up even more. Some of the names from above are too narrow (“entities”) and some are just weird when used in everyday work (“frameworks and drivers”). We can also steer it a smidge towards the DDD nomenclature, and we would arrive at this concept:



*Figure 1.26: Our adjusted model that will follow the overall Clean Architecture outline. The Upper (bigger) names represent the layer name, and the lower (smaller) represents examples of what goes into the layer.*

Or in tabular form with the actual folder names too:

Clean Architecture	Our convention	Folder name
Frameworks & Drivers	Infrastructure	infrastructure/{category}
Interface Adapters	Adapter	infrastructure/adapters
Application Business Rules	Application	application/
Enterprise Business Rules	Domain	domain/

Figure 1.27: Table of compared concepts

### Information

You will notice that here adapters are part of the infrastructure layer rather than being on their own.

If we use a tool like [Madge](#) to generate a diagram of the code, we should be able to see the same [acyclic flow](#) that we want (given that we actually also write the code in the “clean” way!). Below is an example of the Reservation solution.

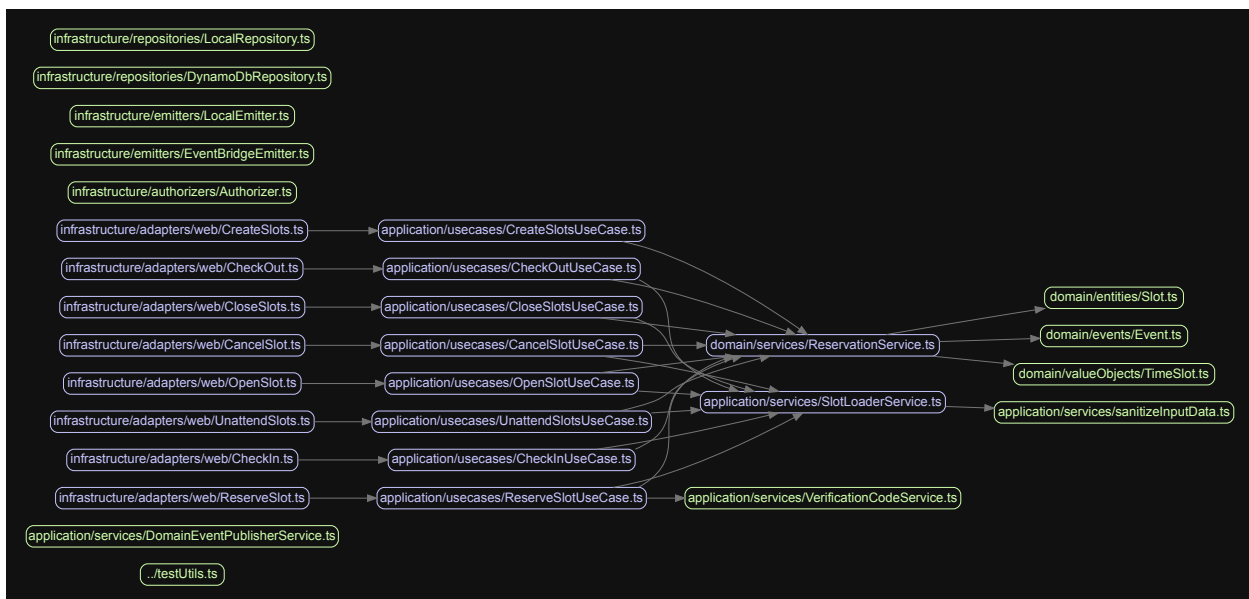


Figure 1.28: Code diagram of the Reservation solution, generated with <https://github.com/pahen/madge>

**Warning**

Note that the above diagram excludes certain code paths—such as those which represent test data, utilities, and interfaces—which can either be used across all levels (or “depths”) or add no meaningful detail to the diagram. They just make the diagram overly busy and are an acceptable omission.

Now, this I am happy with!

**Some words about our layers**

**Infrastructure** The “grown up” way to think about infrastructure is that they are generic functions, classes, and objects that help set up non-domain-related functionality. Good examples of this include Repositories, very generic utility functions, Lambda event handlers (the outer layer), and anything else that has no (or very little) unique value in the specific context.

I’ve been totally happy with not using Clean Architecture’s “frameworks and drivers” nomenclature here but keeping it very flat and simple instead. Those terms didn’t really stick with me or become communicated very well. It’s fine that frameworks and drivers are *part* of the infrastructure, but I’ve personally abandoned packaging under that name.

For me, a useful heuristic has been “Can I move this thing without making essentially no changes and still get it working?”. That does however maybe also say something about the desired level of quality, too...

**Adapters are part of the infrastructure layer**, because, well...they are infrastructure.

**Application** In the application layer, we put anything that is not core to the business, but which *does have* unique value. This should be the first layer where something “new” happens while all the code running before this layer could theoretically be a basic boilerplate.

**We still use the concept of “use cases” and they go into this layer.**

**Domain** Now for the crème de la crème, the secret sauce, and [the figurative room where the magic happens](#). This is, as expected, where all the snazzy unique business logic and domain-orientation truly happens.

**BONUS: Interfaces** Bonus time: Interfaces is an additional folder that I tend to keep at the root—it just collects the types and interfaces. The reason I set this as a root-level item is so that we can effectively do things like:

- Exclude the folder when rendering dependencies
- Put them in the least nested and separate part of the overall structure, as practically every file will have to use some interface or another
- Get them out of the way while still actually putting these in their own place

## Domain driven Lambdas with a use case approach

*The secret sauce for building domain-driven microservices is highly dependent on the “use case” layer.*

How to effectively decompose applications into microservices and bounded contexts is a design exercise that may be challenging and sometimes counter-intuitive, at least if one has been beaten on the head with “FaaS is for single-purpose functions” one too many times. Here, I’ll teach you an approach I’ve found to be well suited to writing testable and good implementations.

### Danger

On this page I will attempt to dispell some misunderstandings and problematic interpretations of structuring work with microservices. Primarily I am concerned about “microservices dogmatism” which is easy to get extra coked-up on when combined with functions-as-a-service. Keeping it super short:

- If you build a “flat landscape” using only Lambdas, then you are close to the “single purpose function” situation. While this is convenient and locally completely suitable in many solutions, as a way to structure an entire



organization's landscape you are in for hell. Some reasons include having to deal with chattiness, latency, many surfaces that need to interact and suboptimal logical cohesion.

- This dogmatism is unrealistic and trivializes the complexity of developing for non-trivial implementation. It also practically eliminates all logical reuse of code and patterns as it cannot be colocated.
- The infrastructure needed for a service is never just Lambda. You need S3 to store the Lambda code, IAM roles, quite realistically an API Gateway and then whatever assortment of candy on top you need to do the work. You need to have a concise notion of what the deployment process is, and what resources it will contain. The “wrapping” on top of that single function is considerable.

You might be interested in [Setting boundaries in your serverless application](#) also.

Features of my approach will mean that:

- We aim for simplicity and clarity of execution using a “one function per use case” type of microservice organization
- Hence, we prefer a functional or procedural style for *most* of the code
- We can use microservices through an API Gateway to expose our well-defined use cases as URL paths
- We use a rich, powerful object-oriented approach for all *complex* components (i.e. Entities and Aggregates) to get the benefits of classic “real engineering master race” (insert meme)
- We'll get the expressive Clean Architecture-style layering
- We set everything in order so we can use the tactical patterns in DDD (which we will come to later)
- We can easily test our application using best practices and a “[classicist](#)” approach, meaning it scales well without any test doubles, “test-speak jargon” or other bullshit

- The approach should be able to translate fluidly into any similar language like C# or Java

### Information

Compared to [Khalil Stemmler's excellent DDD approach](#) the approach presented here is intentionally *simpler* in terms of technical implementation. Something I *have* borrowed from Khalil is [his objection to using Dependency Injection containers](#). Again: We simplify and make the code better for it. You may certainly want to look at his work when, or if, you feel inspired to accelerate my way of doing things. I know I learned a lot from reading his stuff!

### Information

You may also find various Node + DDD projects out in the wild. Without mentioning any names or details (since I've never used them; just inspected them), this approach is better suited to the serverless microservices context (also of course being directly adapted for such a context) and I personally believe my Clean Architecture/DDD-layering stays truer and more conceptually steadfast than what at least I have seen in various projects.

## How do we size and relate microservices in serverless DDD?

There are a number of different takes on how one would relate and size Lambdas, deployments, and how they map to DDD concepts. You will continuously want to question if decisions make

- **local complexity** (i.e. the bounded context or solution itself),
- **global complexity** (i.e. the total landscape),
- **integration complexity** (i.e. how hard it is to make relevant solutions communicate with each other)

better or worse. Your sizing of microservices vs Bounded Contexts vs Aggregates is all part of the same game. Vlad Khononov (and I) would recommend moving towards services that hide significant business logic (ultimately providing something very rich to the user) with the smallest possible surface area (API):

From a system complexity standpoint, a **deep module** reduces the system's global complexity, while a shallow module increases it by introducing a component that doesn't encapsulate its local complexity.

**Shallow services are also the reason why so many microservices-oriented projects fail.** The mistaken definition of a microservice as a service having no more than X lines of code, or as a service that should be easier to rewrite than to modify, concentrate on the individual service while missing the most important aspect of the architecture: the system.

**The threshold upon which a system can be decomposed into microservices is defined by the use cases of the system that the microservices are a part of.**

— *Learning Domain Driven Design* (Khononov 2021, p.224)

For me all of this spells out that use case-oriented APIs, rather than resource-based getter/setter APIs, are what we are aiming for. He also writes more on the actual sizing and boundaries:

Both microservices and bounded contexts are physical boundaries. Microservices, as bounded contexts, are owned by a single team. As in bounded contexts, conflicting models cannot be implemented in a microservice, resulting in complex interfaces. Microservices are indeed bounded contexts. [...]

[T]he relationship between microservices and bounded contexts is not symmetric. Although microservices are bounded contexts, not every bounded context is a microservice. Bounded contexts, on the other hand, denote the boundaries of the largest valid monolith. [...]

[I]f the system is not decomposed into proper bounded contexts or is decomposed past the microservices threshold, it will result in a big ball of mud or a distributed big ball of mud, respectively.

— *Learning Domain Driven Design* (Khononov 2021, p.226-227)

Finally, when it comes to heuristics he writes that:

A more balanced heuristic for designing microservices is to **align the services with the boundaries of business subdomains**. [S]ubdomains are correlated with fine-grained business capabilities. These are the business building blocks required for the company to compete in its business domain(s). [...]

Aligning microservices with subdomains is a safe heuristic that produces optimal solutions for the majority of microservices. That said, there will be cases where other boundaries will be more efficient.

— *Learning Domain Driven Design* (Khononov 2021, p.228-229)

In our case we will see that we did not have to fully follow this advice, however I do see it being a powerful way of closing the loop between:

- The Bounded Context, which is our “designed” and smallest component of the solution
- The subdomain, which acts as the logical container for related components in the domain

If nothing else makes sense, then a sufficiently well-understood subdomain could be packed into a single solution. The beauty of microservices in Lambda, as we will see, is that we can speak of logical monoliths, while still having the individual Lambda functions to work for us.

### Information

For a deeper dive by Eric Evans on different types of bounded contexts and some critique on how one microservice *is not necessarily* one bounded context (which I think it should be), see [Language in Context - Eric Evans - DDD Europe 2019](#).

Personally I find the above sections of Khononov’s book very illuminating, but what in practice does that look like...? :thinking:

The below is what I’ve come to find is the most lucid and rational way to do this when we are building and designing our own system.

## Typical sizing table

### Warning

I will use the broad word **service** to denote the actual code and extent of the “thing” that we are discussing so that we don’t get conflicting terminology.

The way I’ve found to best encapsulate microservices and bounded contexts is that a single Git repository handles a single bounded context which is defined in one configuration file. It may use any number of microservices (i.e. Lambda functions).

One...	...represents
Git repository (classic, single-repo) OR Git repository (mono-repo)	All the code for a single <b>service</b>  All the code for one or more <b>services</b>
Configuration (such as <code>serverless.yml</code> )	Definition for a single <b>service</b>
Service	A single <b>bounded context</b> (typically) OR A single <b>component</b> of the bounded context
Microservice (such as a Lambda function)	A single <b>use case</b>
API gateway	The exposure point for the <b>microservices</b>

Figure 1.29: Comparison

### Information

This approach is valid as far as the bounded context truly is well-defined and self-contained without any excessive territorial encroachment on other contexts!

If the bounded context is wide or simply more coarsely defined, there is absolutely the possibility to relate your service to act as a **component** of the bounded context.

**Warning**

The word “component” isn’t the best, I’m well aware of this, but there is a lack of more descriptive or self-explanatory words.

Of course, under no circumstances should bounded contexts compete about the same logical objects, Aggregates or constructs, nor any attached responsibilities. The bounded context is never bigger than the logical entirety of the context.

Remember that DDD and its terminology is a semantic and logical construct, whereas the code is a technical construct. Therefore any correlation must be handled logically and manually. Nothing forces you to make a single bounded context into a single deployable artifact.

This table, in plain text, could be summarized as:

**A Git repository contains (typically) one microservice that fully represents a single bounded context.** Each microservice is defined by a **single authoritative definition/configuration** and may contain **one or multiple functions that each represent a use case in the bounded context.** An API gateway is the typical way to expose (and protect) the functions.

Let’s say that again: A microservice is not the individual function, it’s the bounded context with *all of its functions*. “Microservices”, hence, refer to the granular style and explicit scope of each bounded context together with the tiny, decoupled deployment artifacts (functions).

Continuing with some of the features I listed at the start of this page, we can attain the following benefits:

- Functions that are truly independent
- Ability to use any number of functions that may or may not interact with each other (or other systems)
- Colocation of code that is logically shared (i.e. bounded context)

- Ability to atomically deploy (i.e. individual functions) or update the whole microservice (i.e. Bounded Context)
- Complete isolation from code and artifacts from other bounded contexts
- Logically and technically scalable solution

## Clean architecture-style use cases

We’ve already touched on structure several times. This time it’s going to be both high-level of how Modules relate as a DDD concept but also how our project is actually divided.

### Information

Read more at <https://www.culttt.com/2014/12/10/modules-domain-driven-design>

This far we have seen how these might work:

- The “blue” ring, as pure infrastructure
- The “green” ring, as our Lambda handler

Now, the “red” ring—use cases—is next up!

## What is a use case?

The software in this layer contains *application specific* business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their *enterprise wide* business rules to achieve the goals of the use case.

We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database,

the UI, or any of the common frameworks. This layer is isolated from such concerns.

We *do*, however, expect that changes to the operation of the application *will* affect the use-cases and therefore the software in this layer. If the details of a use-case change, then some code in this layer will certainly be affected.

— <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

It should sound logical enough. If we look at a very, very basic example it should look like this (code/Reservation/Reservation/src/application/usecases/ReserveSlotUseCase↵↵.ts):

```
import { ReservationService } from "../../domain/services/↵
↵ ReservationService";

import { createVerificationCodeService } from "../services/↵
↵ VerificationCodeService";

import { Dependencies } from "../../interfaces/Dependencies";
import { ReserveOutput } from "../../interfaces/ReserveOutput";
import { SlotInput } from "../../interfaces/Slot";
import { createSlotLoaderService } from "../services/SlotLoaderService"↵
↵ ;

/**
 * @description Use case to handle reserving a slot.
 */
export async function ReserveSlotUseCase(
  dependencies: Dependencies,
  slotInput: SlotInput
): Promise<ReserveOutput> {
  const securityApiEndpoint = process.env.↵
↵ SECURITY_API_ENDPOINT_GENERATE || "";

  const { slotId, hostName } = slotInput;
  const slotLoader = createSlotLoaderService(dependencies.repository);
  const slotDto = await slotLoader.loadSlot(slotId);

  const verificationCodeService =
    createVerificationCodeService(securityApiEndpoint);
  const reservationService = new ReservationService(dependencies);
```



```
return await reservationService.reserve(  
    slotDto,  
    hostName,  
    verificationCodeService  
);  
}
```

There's not that much going on here. We import and use interface definitions, take in dependencies and the record data, and then it's just two commands.

If you have been around the block, maybe you feel one or more of the below:

- “OK, this seems like just adding more chaff into the mix, doesn't it?”
- “Isn't this a *Transaction Script*... WTH mate?”

When it comes to the question of adding more (useless?) code and more layers of abstractions, rather we should see the benefits. Because we broke up the handler and its boilerplate from our business and use case, *\*\*the use case is the first meaningfully testable layer*. That is, the surface for our widest unit testing is the *use case* as it effectively exercises the full flow and we can afford to be totally oblivious about anything in the handler itself. So, yes indeed, it does add a further layer but again we get something better back as a result for that minimal investment.

## Transaction Script is your friend...if done well

Now, let's consider first Martin Fowler's words on the Transaction Script pattern:

Most business applications can be thought of as a series of transactions. A transaction may view some information as organized in a particular way, another will make changes to it. Each interaction between a client system and a server system contains a certain amount of logic. In some cases this can be as simple as displaying information in the database. In others it may involve many steps of validations and calculations.

A Transaction Script organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each

transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures.

— [Martin Fowler/Thoughtworks: Transaction Script](#)

The gist I am trying to tell is that the use case itself should act essentially as a readable, easy-to-follow orchestration of the use case's mechanics. A key difference is that between a novice programmer and a seasoned one, the use case itself must not touch the details, concretions or infrastructure. Instead (as we see) we trust that our commands on abstractions work as intended:

```
const { repository } = dependencies;
await repository.add(record);
```

This pattern scales well, as long as the deeper layers (entities etc.) are doing their job. For this particular case, there isn't an actual Aggregate or Entity involved, just the basic repository.

We can also look at how this scales to a much more complex case. In this case though, you will never see that complexity! It looks almost the same (code/Reservation/↔↔ SlotReservation/src/application/usecases/CreateSlotsUseCase.ts):

```
import { Slot } from "../../domain/aggregates/Slot";
import { Dependencies } from "../../interfaces/Dependencies";

/**
 * @description Use case to handle creating the daily slots.
 */
export async function CreateSlotsUseCase(
  dependencies: Dependencies
): Promise<string[]> {
  const slot = new Slot(dependencies);
  return await slot.makeDailySlots();
}
```

You'll have to trust me on this one for now, but yes, the above is more complex. However, the orchestration that we have to do is not. If the use case, for example, would demand several Entities to interact or connect somehow, or multiple operations to be done, then using the use case file/function/layer is the right place to string together the logic.

Fortunately, though, all the services in Get-A-Room are pretty orthodox to clean domain modelling and tight in their implementation, so they don't do any messy stuff, leaving the use case itself very straightforward.

### **In closing**

The use case is the first meaningfully testable layer. You should prioritize tests for this layer (if you aren't already doing TDD or such). Testing here gives you *a lot* of bang for the buck.

Use cases are good examples of places where a well-working Transaction Script pattern can be used. I find it true that the less lines you have, the better the domain layer is functioning and the tighter you have been on understanding what your system should do.

## Create slots

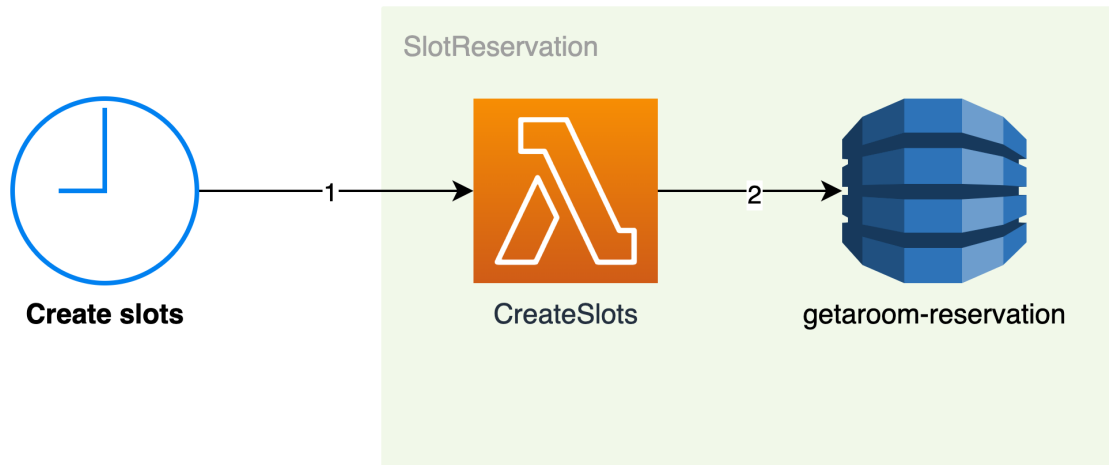


Figure 1.30: Creating slots is a scheduled (“cronjob”) function call in the Reservation service.

The root use case must be the slot creation because without it we don’t have much of anything. It’s fair to call this an “under the hood” type of use case.

For it to do something meaningful, Slots need to be created into the main DynamoDB table so these can be reproduced by the Display service.

This particular solution is trivial, as we only need to allow it to be called as a scheduled function. In our case we call it at 05:00 GMT every Monday through Friday:

In `code/Reservation/Reservation/serverless.yml`:

```

CreateSlots:
  handler: src/infrastructure/adapters/web/CreateSlots.handler
  description: Create new slots
  events:
    # You can activate this to allow for HTTP-based calls
    #- http:
    #  method: GET
    #  path: /CreateSlots
    - schedule: cron(0 5 ? * MON-FRI *)
  
```

The use case itself doesn't do much other than defer to the `ReservationService` (in the file `code/Reservation/Reservation/src/application/usecases/CreateSlotsUseCase.ts`) to create the slots.

```
export async function CreateSlotsUseCase(
  dependencies: Dependencies
): Promise<string[]> {
  const reservationService = new ReservationService(dependencies);

  return await reservationService.makeDailySlots();
}
```

## Updating the Display projection

What about that `Display` service? That one is independently subscribing to relevant Domain Events, so it will update the projection to match what is happening in the leading data source, or Aggregate, which is...`Reservation`!

Let's update `code/Reservation/Display/serverless.yml`:

```
UpdateSlot:
  handler: src/infrastructure/adapters/web/UpdateSlot.handler
  description: Update a room slot projection
  events:
    # Can be activated if you need to do HTTP-based calls or testing
    #- http:
    # method: POST
    # path: /update
    - eventBridge:
  eventBus: ${self:custom.aws.domainBusArn}
  pattern:
  detail-type:
    # User events
    - "Created"
    - "Cancelled"
    - "Reserved"
    - "CheckedIn"
    - "CheckedOut"
    - "Unattended"
    # System events
```

```
- "Closed"
source:
  - prefix: ""
iamRoleStatements:
  - Effect: "Allow"
Action:
  - dynamodb:PutItem
Resource: ${self:custom.aws.databaseArn}
```

As we will see many times, also here the use case is basic. Because our Repository uses an [upsert](#) pattern (update in place) it doesn't matter if the item already exists or not; so it's [idempotent](#) which is a good thing.

In code/Reservation/Display/src/application/usecases/UpdateSlotUseCase.ts:

```
export async function UpdateSlotUseCase(
  dependencies: Dependencies,
  slot: Slot
) {
  const { repository } = dependencies;
  await repository.add(slot);
}
```

## Get slots

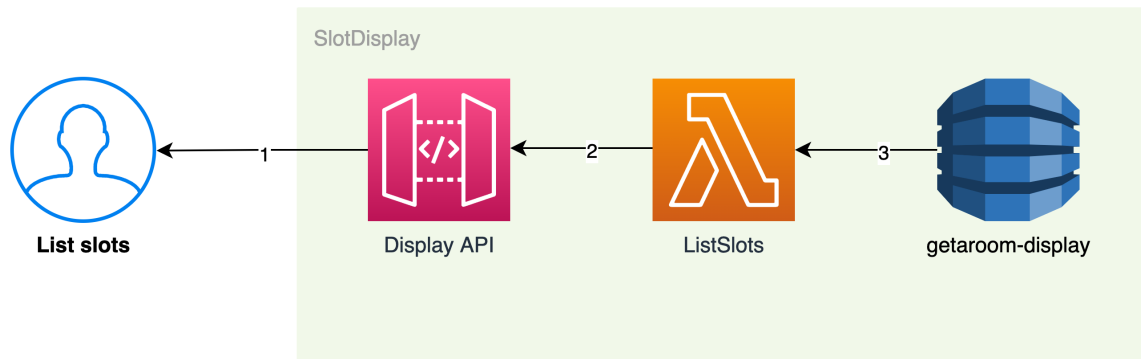


Figure 1.31: The UI-facing Display service can read the projected slots from its own persistence.

The Display service is also not very complicated; it simply returns the current set of slots from its own persistence (DynamoDB). Because it keeps its own projection of slots the data may be eventually consistent, but that won't be a real issue in the type of circumstance we have here.

In `code/Reservation/Display/serverless.yml`:

```

GetSlots:
  handler: src/infrastructure/adapters/web/GetSlots.handler
  description: Get room time slots
  events:
    - http:
        method: GET
        path: /slots
  iamRoleStatements:
    - Effect: "Allow"
      Action:
        - dynamodb:Query
      Resource: ${self:custom.aws.databaseArn}
  
```

The use case is so basic we can fully conduct everything we need in the application-layer use case with the help of our injected Repository (`code/Reservation/Display/↔ src/application/usecases/GetSlotsUseCase.ts`).

```
export async function GetSlotsUseCase(dependencies: Dependencies) {  
  const { repository } = dependencies;  
  return await repository.getSlots();  
}
```



## Reserve a slot

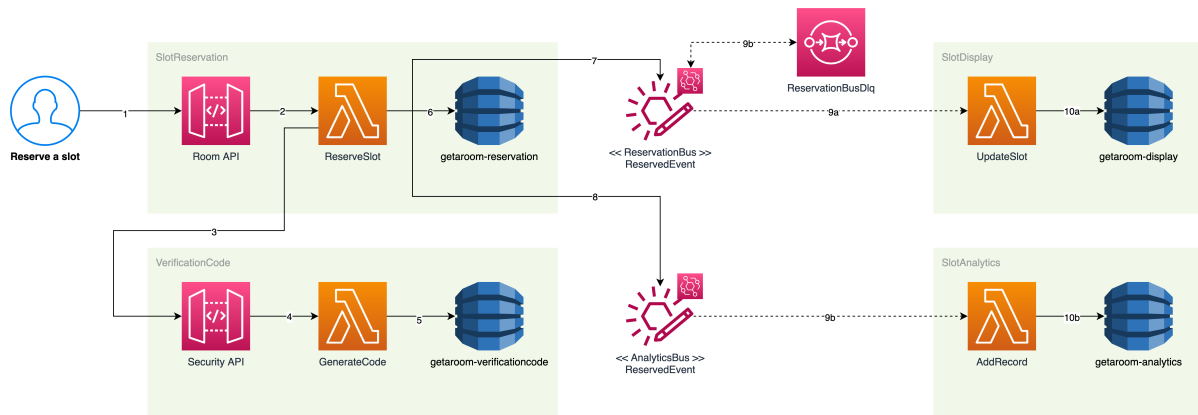


Figure 1.32: Perhaps the most complex use case, passing through all four services.

The big one: Reserve a slot.

It's very important that when we build anything like this, we stay clear on the Bounded Contexts and relationships we mapped previously. Now is not the time to lump together everything in a big box, shake it around, and confuse the borders of our responsibility! Because this happens in the Reservation service, the primary concern we have is to ensure correct functionality within that box.

We need to respect, and use, the API contract provided by the VerificationCode service, as well as ensure that the event pushed to the event buses is correct to the analytics context (because we have promised to give them a specific shape of it). We “own” the event we pass to Display as that just makes more sense. So, those were the responsibilities.

For the actual configuration file (code/Reservation/Reservation/serverless.yml), there's not too much going on, besides enforcing a schema on our API so people won't be calling it willy-nilly.

```
ReserveSlot:
  handler: src/infrastructure/adapters/web/ReserveSlot.handler
  description: Reserve a slot
  events:
    - http:
  method: POST
  path: /ReserveSlot
```

```
request:
schemas:
application/json: ${file(schema/ReserveSlot.validator.json)}
```

This time, the use case is a bit bigger as we need to load a lot more services before passing them to our domain service, `ReservationService` (at `code/Reservation/Reservation` ↩ ↪ `/src/application/usecases/ReserveSlotUseCase.ts`), to actually conduct any business logic.

```
export async function ReserveSlotUseCase(
  dependencies: Dependencies,
  slotInput: SlotInput
): Promise<ReserveOutput> {
  const securityApiEndpoint = process.env.↩ ↪
    SECURITY_API_ENDPOINT_GENERATE || "";

  const { slotId, hostName } = slotInput;
  const slotLoader = createSlotLoaderService(dependencies.repository);
  const slotDto = await slotLoader.loadSlot(slotId);

  const verificationCodeService =
    createVerificationCodeService(securityApiEndpoint);
  const reservationService = new ReservationService(dependencies);

  return await reservationService.reserve(
    slotDto,
    hostName,
    verificationCodeService
  );
}
```

With that, it's still not *a lot*, to be honest. It's always nice being able to do more with less (code)!

## Unattend no-shows

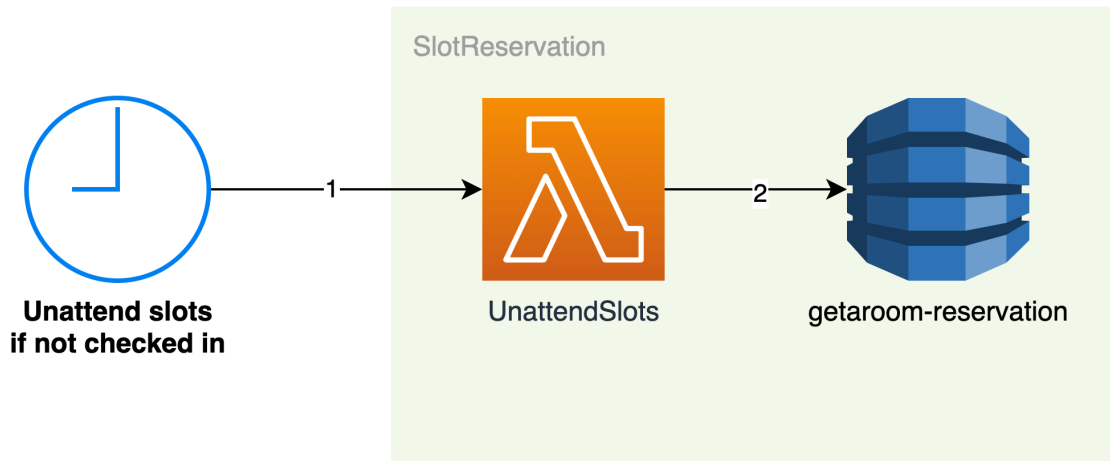


Figure 1.33: Time again (pun intended) for a scheduled “cronjob”!

If any slot is not checked in to within the grace period we need to make them bookable again. Once again a scheduled function can do the heavy lifting, combined with some glue logic on our end.

In code/Reservation/Reservation/serverless.yml:

```

UnattendSlots:
  handler: src/infrastructure/adapters/web/UnattendSlots.handler
  description: Check if any slots are unattended at 10 minutes past the
    ↪ hour
  events:
    # You can activate this to allow for HTTP-based calls
    #- http:
    # method: GET
    # path: /UnattendSlots
    - schedule: cron(10 6-16 ? * MON-FRI *)
  
```

**Warning**

You will note that this logic is not fail-safe. If you were to book a slot at 10:11 for the 10-11 window nothing will “unattend” the slot if you did not check-in. *Good enough* is sometimes just that. And as the requirements are vague and this is no more than a demonstration project, we can leave it at that.

We’ll use a service called `SlotLoaderService`, instead of directly perusing the `Repository` to load the complete set of today’s Slots. Then, we are going to pass them into the `checkForUnattended()` method where the domain service will actually contain the business/domain logic to determine whether or not a slot is attended.

At `code/Reservation/Reservation/src/application/usecases/UnattendSlotsUseCase` ↩  
↩ .ts:

```
export async function UnattendSlotsUseCase(dependencies: Dependencies) ↩  
  ↩ {  
    const slotLoader = createSlotLoaderService(dependencies.repository);  
    const slots = await slotLoader.loadSlots();  
  
    const reservationService = new ReservationService(dependencies);  
    await reservationService.checkForUnattended(slots);  
  }
```

## Check in

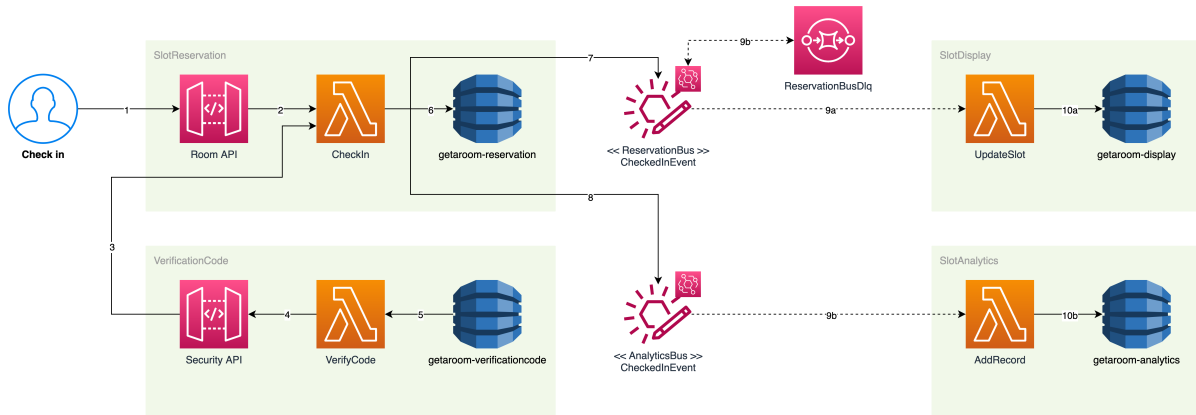


Figure 1.34: Checking in means we have to use all four services.

To check-in, we first need to verify and authorize the calling user, so no one else goes checking in to the room that you've waited so long for. You'll see this in the authorizer↔↔ block. While the implementation of the authorizer itself is rudimentary, just having anything here makes the solution as a whole better.

See code/Reservation/Reservation/serverless.yml:

```

CheckIn:
  handler: src/infrastructure/adapters/web/CheckIn.handler
  description: Check in to slot
  events:
    - http:
  method: POST
  path: /CheckIn
  authorizer:
    name: Authorizer
  resultTtlInSeconds: ${self:custom.config.apiGatewayCachingTtlValue}
  identitySource: method.request.header.Authorization
  type: request
  request:
  schemas:
    application/json: ${file(schema/Id.validator.json)}
  
```

In the use case code, we will load the Slot DTO and pass it to the respective method.

code/Reservation/Reservation/src/application/usecases/CheckInUseCase.ts:

```
export async function CheckInUseCase(
  dependencies: Dependencies,
  slotId: SlotId
) {
  const slotLoader = createSlotLoaderService(dependencies.repository);
  const slotDto = await slotLoader.loadSlot(slotId);

  const reservationService = new ReservationService(dependencies);
  await reservationService.checkIn(slotDto);
}
```

## Check out

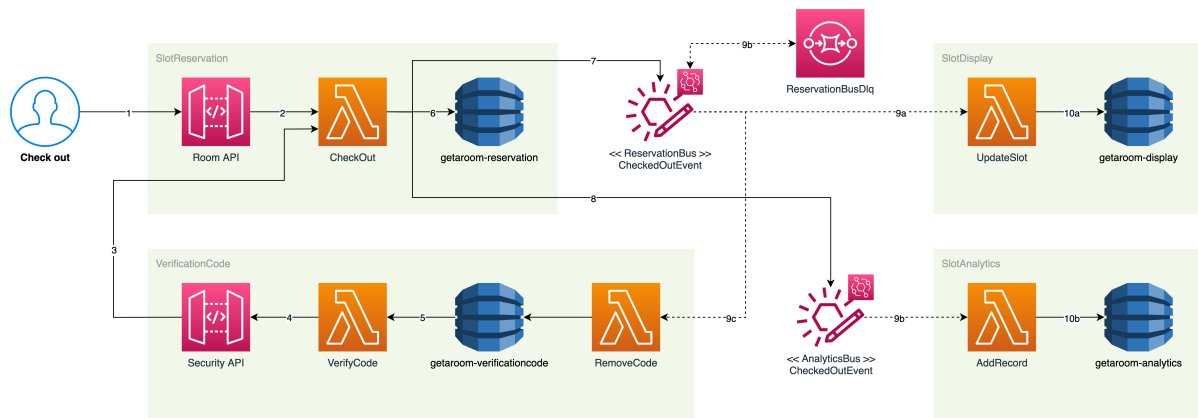


Figure 1.35: Oddly similar to check in? Yep, I feel the same.

Same as with checking in, for checking it we need the authorization of the user and call.

This is code/Reservation/Reservation/serverless.yml:

```
CheckOut:
  handler: src/infrastructure/adapters/web/CheckOut.handler
  description: Check out from slot
  events:
    - http:
      method: POST
      path: /CheckOut
      authorizer:
        name: Authorizer
      resultTtlInSeconds: ${self:custom.config.apiGatewayCachingTtlValue}
      identitySource: method.request.header.Authorization
      type: request
      request:
        schemas:
          application/json: ${file(schema/Id.validator.json)}
```

The use case itself (code/Reservation/Reservation/src/application/usecases/CheckOutUseCase.ts) is practically copy-paste from the check-in case.

```
export async function CheckOutUseCase(
  dependencies: Dependencies,
```

```
    slotId: SlotId
  ) {
    const slotLoader = createSlotLoaderService(dependencies.repository);
    const slotDto = await slotLoader.loadSlot(slotId);

    const reservationService = new ReservationService(dependencies);
    await reservationService.checkOut(slotDto);
  }
```



```

sequenceDiagram
    actor User
    participant RoomAPI as Room API
    participant CancelSlot as CancelSlot
    participant OpenSlot as OpenSlot
    participant getaroom_reservation as getaroom-reservation
    participant getaroom_verificationcode as getaroom-verificationcode
    participant RemoveCode as RemoveCode
    participant SecurityAPI as Security API
    participant VerifyCode as VerifyCode
    participant UpdatesSlot as UpdatesSlot
    participant AddRecord as AddRecord
    participant ReservationBusDisq as ReservationBusDisq
    participant AnalyticsBus as << AnalyticsBus >> CancelledEvent
    participant CanceledEvent as << ReservationBus >> CancelledEvent

    User->>RoomAPI: 1
    RoomAPI->>CancelSlot: 2
    CancelSlot->>OpenSlot: 11
    CancelSlot->>getaroom_reservation: 6
    OpenSlot->>getaroom_reservation: 7
    getaroom_reservation->>getaroom_verificationcode: 8
    getaroom_verificationcode->>RemoveCode: 10c
    RemoveCode->>VerifyCode: 5
    VerifyCode->>SecurityAPI: 4
    SecurityAPI->>User: 3

    getaroom_reservation->>CanceledEvent: 9d
    CanceledEvent->>AnalyticsBus: 9b
    CanceledEvent->>ReservationBusDisq: 9b
    CanceledEvent->>UpdatesSlot: 9a
    CanceledEvent->>AddRecord: 9a
    CanceledEvent->>RemoveCode: 9c

    UpdatesSlot->>getaroom_display as getaroom-display: 10a
    AddRecord->>getaroom_analytics as getaroom-analytics: 10b
  
```

```
application/json: ${file(schema/Id.validator.json)}
```

In our use case, we will again use the convenience service called `SlotLoaderService` ↩, rather than the `Repository`.

This is `code/Reservation/Reservation/src/application/usecases/CancelSlotUseCase` ↩.ts:

```
export async function CancelSlotUseCase(
  dependencies: Dependencies,
  slotId: SlotId
): Promise<void> {
  const slotLoader = createSlotLoaderService(dependencies.repository);
  const slotDto = await slotLoader.loadSlot(slotId);

  const reservationService = new ReservationService(dependencies);
  await reservationService.cancel(slotDto);
}
```

The pattern should be quite familiar by now.

## Open slot

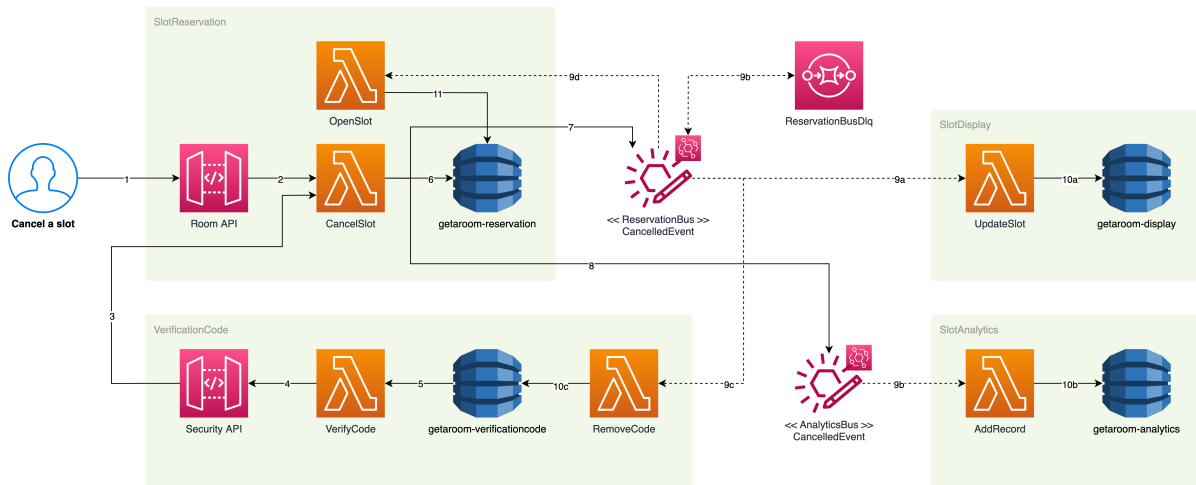


Figure 1.37: The slot opening happens in 9d and 11.

Opening a slot is an intended side effect of canceling the slot. You will see in the diagram at 9d and 11 where the `OpenSlot` Lambda gets triggered. This is also clearly outlined in the configuration file:

In `code/Reservation/Reservation/serverless.yml`:

```
OpenSlot:
  handler: src/infrastructure/adapters/web/OpenSlot.handler
  description: Open a slot
  events:
    # You can activate this to allow for HTTP-based calls
    #- http:
    #  method: POST
    #  path: /OpenSlot
    - eventBridge:
  eventBus: ${self:custom.config.domainBusName} # Create new ↔
    ↪ EventBridge bus
  pattern:
  source:
    - getaroom.reservation.cancelled
  deadLetterQueueArn:
  Fn::GetAtt:
    - ReservationBusDlq
    - Arn
```

```
retryPolicy:  
maximumEventAge: 3600  
maximumRetryAttempts: 3
```

And still another boring, but functional use case with no surprises.

The use case, `code/Reservation/Reservation/src/application/usecases/OpenSlotUseCase` ↩  
↪ `.ts`:

```
export async function OpenSlotUseCase(  
  dependencies: Dependencies,  
  slotId: SlotId  
) {  
  const slotLoader = createSlotLoaderService(dependencies.repository);  
  const slotDto = await slotLoader.loadSlot(slotId);  
  
  const reservationService = new ReservationService(dependencies);  
  await reservationService.open(slotDto);  
}
```

## Close slots

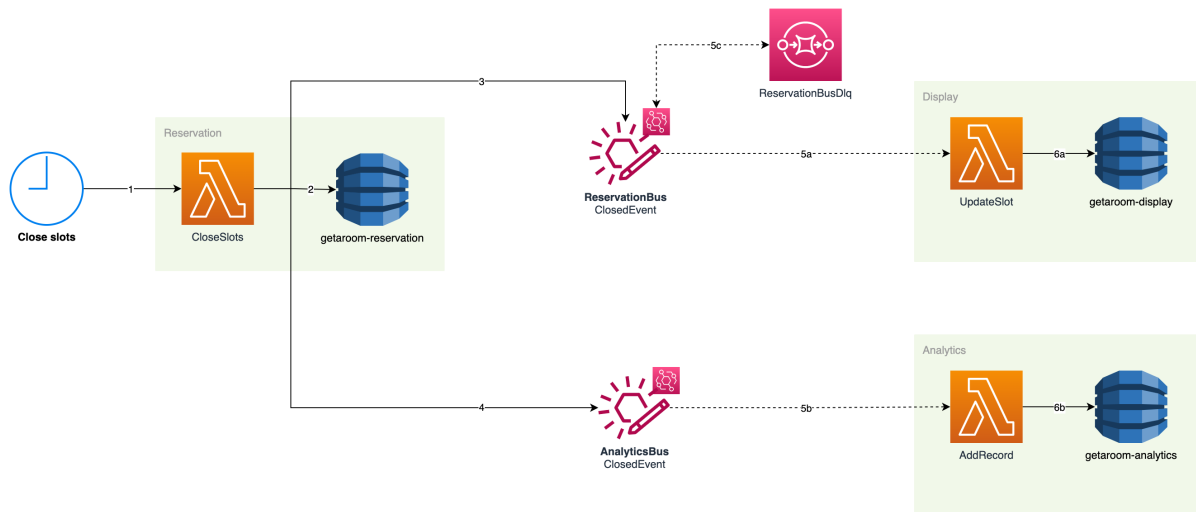


Figure 1.38: To close a slot means we have to ensure the read-copy, or projection, in Display also knows what is going on.

Closing slots is what we have to do when a slot is no longer reservable.

Practically speaking, we will run a scheduled Lambda after each open hour, which is to say: 0700-1700 GMT Monday through Friday. You may ask “what if the slot isn’t actually closed at that time?” The answer is actually pretty simple—since Lambda will only happen *after* the time you want it to run, plus an additional 5-20 seconds of delay before it actually executes at all, you can be sure that you are doing this when it makes logical sense: after the slot has ended.

### Success

If we had a very time-sensitive system this particular solution may have been unacceptably slow, but here it’s not nearly an actual problem.



In code/Reservation/Reservation/serverless.yml:

```

CloseSlots:
  handler: src/infrastructure/adapters/web/CloseSlots.handler
  
```

```
description: Close any slots that have passed their end times
events:
  # You can activate this to allow for HTTP-based calls
  #- http:
  # method: GET
  # path: /CloseSlots
  - schedule: cron(0 7-17 ? * MON-FRI *)
```

Nothing new here... The `checkedForClosed()` will run an internal loop first to check that the provided loops are truly ended and then run the rest of the transactional logic for those slots that we no longer need.

This is `code/Reservation/Reservation/src/application/usecases/CloseSlotsUseCase`    
  `.ts`:

```
export async function CloseSlotsUseCase(dependencies: Dependencies) {
  const slotLoader = createSlotLoaderService(dependencies.repository);
  const slots = await slotLoader.loadSlots();

  const reservationService = new ReservationService(dependencies);
  await reservationService.checkForClosed(slots);
}
```

## Factories

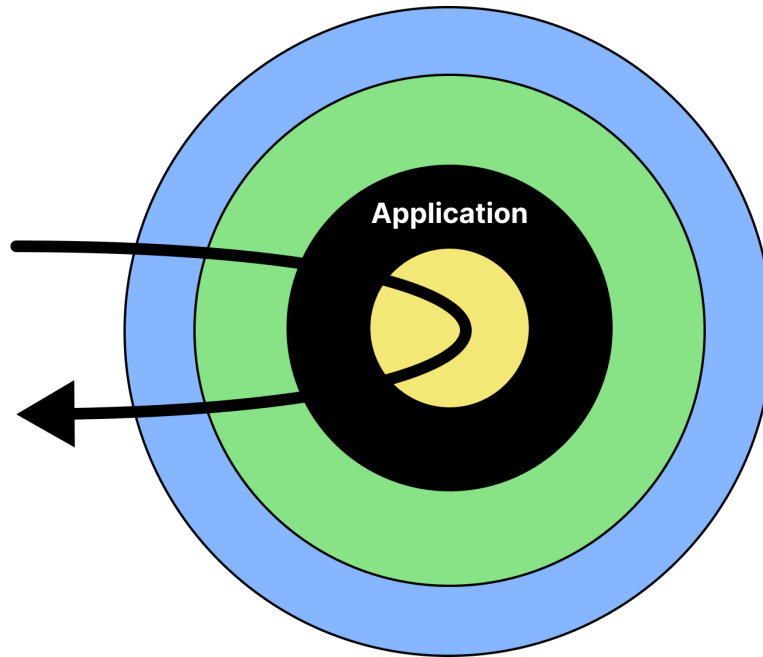
*Factories enable efficient and simple production of (usually) complex objects. Let's use them to simplify our implementation interfaces.*



*Figure 1.39: Illustration from Undraw*

### Success

**TL;DR:** The **Factory** pattern is a classic creational pattern. Some important reasons to use the pattern include that it appropriately encapsulates creation logic, as well as provides a structured way of creating objects in a deterministic manner.



*Figure 1.40: Factories reside in the Application layer.*

Factories encapsulate the creation of, primarily, complex objects such as those in the domain layer. The pattern itself has nothing to do with DDD (instead, please see [Design Patterns: Elements of Reusable Object-Oriented Software](#)). In the context of DDD, we gain even better enforcement of encapsulation, which is especially meaningful when we need to construct an Entity or Aggregate.

Factories help us to hide implementation and construction logic and always return valid invariants of the class (“product”) that we have created. However, invariant logic and validation should as far as possible be deferred to the product being created itself, which makes perfect sense if we are using Factories to create complex objects like Entities and Aggregates that already have such logic baked in.

You can probably imagine a case where the setup of an Aggregate will require pulling lots of parameters, checking validity, and other such stuff—this is a perfect case of hiding that with a Factory. I’ve used Factories several times when I need to create an object that requires complicated asynchronous setups. By using the Factories we can avoid leaking out any of that complexity onto the user.

The way Factories are used in the example is very basic. There is nothing blocking you from applying the Factory pattern to creational methods on Aggregates or Services them-



selves (see for example Vernon 2013, p.391/397).

## Examples of the pattern

To be fair, there are no good uses of “proper” and complex factories in Get-A-Room.

### Success

Often you will find factories in an object-oriented class shape, but here we will use a more TypeScript-idiomatic way of using functions.

Several factories have been used to remove some of the ugly `new SomeClass()` calls. I’ll happily use it whenever I want to avoid letting a user directly access a class, like this (code/Analytics/SlotAnalytics/src/infrastructure/repositories/DynamoDbRepository ↵ ↵ .ts):

```
/**
 * @description Factory function to create a DynamoDB repository.
 */
export function createNewDynamoDbRepository(): DynamoDbRepository {
  return new DynamoDbRepository();
}
```

This also works well in creating concrete instances of services that need some values for setting them up (code/Reservation/Reservation/src/application/services/VerificationCodeService ↵ ↵ .ts):

```
export function createVerificationCodeService(securityApiEndpoint: ↵
  ↵ string) {
  return new ConcreteVerificationCodeService(securityApiEndpoint);
}
```

More on the VerificationCodeService later.

We can also use it to package some important checks or validations we may have, like with

the EventBridge emitter (code/Reservation/SlotReservation/src/infrastructure/emitters↵  
↵ /EventBridgeEmitter.ts):

```
/**
 * @description Factory function to return freshly minted EventBridge ↵
 * ↵ instance.
 */
export const makeNewEventBridgeEmitter = (region: string) => {
  if (!region)
    throw new MissingEnvVarsError(
      JSON.stringify([ { key: "REGION", value: region } ])
    );

  return new EventBridgeEmitter(region);
};
```

While very basic, all of these (especially the two last ones) get the point across; A Factory can hide some of the ugly details involved in creating important objects.

### Information

For an excellent and more in-depth article on factories, see <https://www.culttt.com/2014/12/24/factory-method/typescript/example/> or <https://refactoring.guru/design-patterns/factory-method/typescript/example/>. Overall, I highly recommend checking out the creational patterns at <https://refactoring.guru/design-patterns/creational-patterns>.

## Repositories



*Figure 1.41: Illustration from Undraw*

### Success

**TL;DR:** When it's time to do the inevitable persisting or loading of data, it's a **Repository** you want. Similar to the Factory, a **Repository** makes its core actions (loading, saving) a deterministic and easy-to-use operation. By separating this logic out, we can avoid polluting actual domain logic with this low-level (though important) detail.

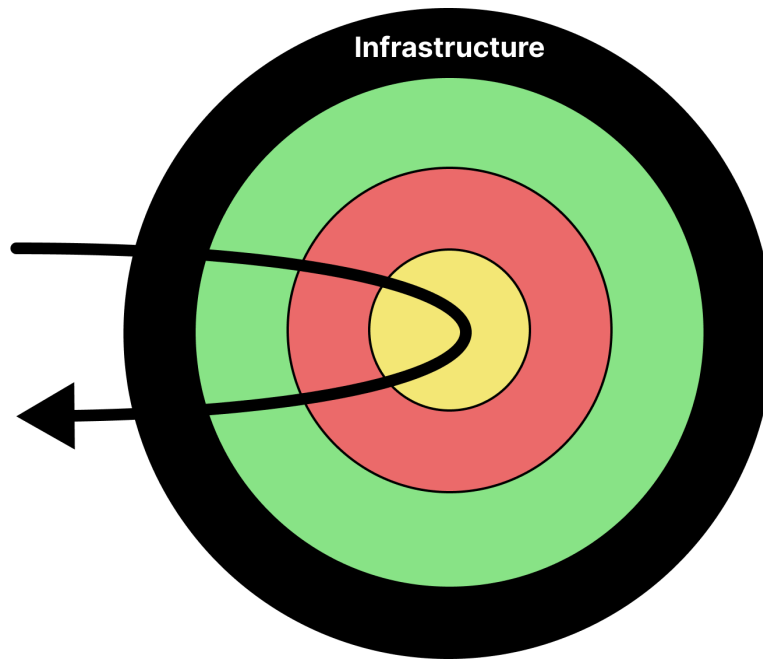


Figure 1.42: Repositories reside in the Infrastructure layer (in my take on DDD + CA).

Good old Repositories! This is by my very unscientific gut feeling maybe the most used and best-known of patterns. Well, at least in terms of its nominal recognition.

## Why Repositories?

Let's start by addressing the need for a Repository. Somehow you will need to **retrieve or store the reference to an Aggregate or Entity or some other domain object**. Using the language of the domain, the Repository will be able to retrieve and return the data. The data, in turn, is typically an Aggregate or Entity which can be *reconstituted* into its programmatic shape (Entity class, etc.) when you've got the data back.

The bad side of being a well-known pattern is that this may have been what has led many traditional back-end developers to be “anemically data-oriented” in their work; seemingly a typical child disease of having been in the hard-knock “relational database school”. As I've previously written, being only structurally data-focused rather than also similarly obsessed about the expected behavior (logic, business rules, etc.) can quickly lead straight down the [anemic domain model](#) hole.

### Danger

Remember that the biggest enemy of DDD is the [anemic domain model](#). Repositories are therefore important in the technical sense to make object persistence work at all, but similarly important is the goal to make Repositories decoupled from any behavior-altering mannerisms: **The Repository is not smart**, your domain objects are! So refrain from making big exercises in data modeling here beyond but is absolutely required to make object retrieval work in the domain model.

The primary place for Repositories is, therefore (as Evans writes; 2013, p.148) in the middle of the object's lifecycle: persisting, loading, and reconstituting the data. The Repository acts as **the only way to retrieve data** and this must not be bypassed.

The typical "by-the-book" way is to use one Repository per higher concept or Aggregate, say, `ReservationRepository` and `SlotRepository`, which would often mean we would need unique Repositories per object. Logically speaking this makes sense as the repository will have to be uniquely implemented based on the specific needs of the Aggregate in question. However, I will now explain why that's *not the way* I am dealing with it in our example code.

## How Repositories are used in the project

Because I am choosing to understand and implement Repositories as an infrastructural feature, rather than as being part of a domain, I do not want Repositories to have knowledge of the actual Entity classes (such as `Slot`) so I do not return the class instance, but the Data Transfer Object that the Aggregate (`Reservation`) can reconstitute itself.

### Information

This model, as far as I know, therefore stays somewhat truer with Robert Martin and his Clean Architecture than with the classic DDD approach.

This opinion is contentious and debated, as witnessed in [this response by Subhash on](#)

## Stack Overflow:

Repositories and their placement in the code structure are a matter of intense debate in DDD circles. It is also a matter of preference, and often a decision taken based on the specific abilities of your framework and ORM.

The issue is also muddled when you consider other design philosophies like Clean Architecture, which advocates using an abstract repository in the domain layer while providing concrete implementations in the infrastructure layer.

— [Stack Exchange: “Which layer do DDD Repositories belong to?”](#)

In the spirit of pragmatism, the approach I am using is more relaxed, going with one Repository per persistence mechanism—DynamoDB and local/mock use. Because the solution itself is one deployable artifact and because there are no overlapping concepts, this is not problematic since there is no confusion or logical overstepping happening.

First of all, let’s see one of the use cases and understand where we are loading the Slot (code/Reservation/Reservation/src/application/usecases/CancelSlotUseCase.ts):

```
import { Reservation } from "../../domain/aggregates/Reservation";

import { createSlotLoaderService } from "../../services/SlotLoaderService"↵
↵ ;

import { Dependencies } from "../../interfaces/Dependencies";
import { SlotId } from "../../interfaces/Slot";

/**
 * @description Use case to handle cancelling a slot.
 */
export async function CancelSlotUseCase(
  dependencies: Dependencies,
  slotId: SlotId
): Promise<void> {
  const reservation = new Reservation(dependencies);
  const slotLoader = createSlotLoaderService(dependencies.repository);
  const slotDto = await slotLoader.loadSlot(slotId);

  await reservation.cancel(slotDto);
}
```

## Success

Don't think too hard about the `SlotLoaderService`. For now, know that it is a higher-order construct on top of the `Repository` itself.

You'll see that we use a `Factory` to vend a new `SlotLoaderService`, which we then use to load the `slotId` we have on hand. With the `Slot`'s `DTO` retrieved we can call the appropriate `Aggregate` method, which itself then may reconstitute the data so that we can make use of the `Slot` `Entity`'s functionality and logic before doing whatever other things it is expected to do.

```
public async cancel(slotDto: SlotDTO): Promise<void> {
  const slot = new Slot().from(slotDto);
  // Rest of code...
}
```

This same pattern is used for all similar use cases.

Now for one of the actual `Repositories`.

See `code/Reservation/Reservation/src/infrastructure/repositories/DynamoDbRepository` ↩  
 ↩ `.ts`:

```
import { randomUUID } from "crypto";
import {
  AttributeValue,
  DynamoDBClient,
  PutItemCommand,
  QueryCommand,
  QueryCommandOutput,
} from "@aws-sdk/client-dynamodb";

import { Repository } from "../../interfaces/Repository";
import { SlotDTO, SlotId } from "../../interfaces/Slot";
import { DynamoItem, DynamoItems } from "../../interfaces/DynamoDb";
import { Event, EventDetail } from "../../interfaces/Event";

import { MissingEnvVarsError } from "../../application/errors/↩
  ↩ MissingEnvVarsError";
```

```

import { getCleanedItems } from "../utils/getCleanedItems";

import testData from "../../testdata/dynamodb/testData.json";

/**
 * @description Factory function to create a DynamoDB repository.
 */
export function createDynamoDbRepository(): DynamoDbRepository {
  return new DynamoDbRepository();
}

/**
 * @description Concrete implementation of DynamoDB repository.
 * @see https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/dynamodb-example-table-read-write.html
 */
class DynamoDbRepository implements Repository {
  docClient: DynamoDBClient;
  tableName: string;
  region: string;

  constructor() {
    this.region = process.env.REGION || "";
    this.tableName = process.env.TABLE_NAME || "";

    if (!this.region || !this.tableName)
      throw new MissingEnvVarsError(
        JSON.stringify([
          { key: "REGION", value: process.env.REGION },
          { key: "TABLE_NAME", value: process.env.TABLE_NAME },
        ])
      );

    this.docClient = new DynamoDBClient({ region: this.region });
  }

  /**
   * @description Create and return expiration time for database item.
   */
  private getExpiryTime(): string {
    const tomorrow = new Date();
    tomorrow.setDate(tomorrow.getDate() + 1);
    tomorrow.setHours(0, 0, 0, 0);
    return Date.parse(tomorrow.toString()).toString().substring(0, 10);
  }
}

```



```

/**
 * @description Load a Slot from the source database.
 */
public async loadSlot(slotId: SlotId): Promise<SlotDTO> {
    const command = {
        TableName: this.tableName,
        KeyConditionExpression: "itemType = :itemType AND id = :id",
        ExpressionAttributeValues: {
            ":itemType": { S: "SLOT" },
            ":id": { S: slotId },
        },
        ProjectionExpression:
            "id, hostName, timeSlot, slotStatus, createdAt, updatedAt",
    };

    const data: QueryCommandOutput | DynamoItems =
        process.env.NODE_ENV === "test"
            ? testData
            : await this.docClient.send(new QueryCommand(command));
    const items =
        (data.Items?.map(
            (item: Record<string, AttributeValue>) => item
        ) as DynamoItem[]) || [];

    return getCleanedItems(items)[0] as unknown as SlotDTO;
}

/**
 * @description Load all Slots for the day from the source database.
 */
public async loadSlots(): Promise<SlotDTO[]> {
    const command = {
        TableName: this.tableName,
        KeyConditionExpression: "itemType = :itemType",
        ExpressionAttributeValues: {
            ":itemType": { S: "SLOT" },
        },
        ProjectionExpression:
            "id, hostName, timeSlot, slotStatus, createdAt, updatedAt",
    };

    const data: QueryCommandOutput | DynamoItems =
        process.env.NODE_ENV === "test"
            ? testData

```

```

        : await this.docClient.send(new QueryCommand(command));
const items =
    (data.Items?.map(
        (item: Record<string, AttributeValue>) => item
    ) as DynamoItem[]) || [];

return getCleanedItems(items);
}

/**
 * @description Add (create/update) a slot in the source database.
 */
public async updateSlot(slot: SlotDTO): Promise<void> {
    const { slotId, hostName, timeSlot, slotStatus, createdAt, ↵
    ↵ updatedAt } =
        slot;

    const expiresAt = this.getExpiryTime();
    const command = {
        TableName: this.tableName,
        Item: {
            itemType: { S: "SLOT" },
            id: { S: slotId },
            hostName: { S: hostName || "" },
            timeSlot: { S: JSON.stringify(timeSlot) },
            slotStatus: { S: slotStatus },
            createdAt: { S: createdAt },
            updatedAt: { S: updatedAt },
            expiresAt: { N: expiresAt },
        },
    };

    if (process.env.NODE_ENV !== "test")
        await this.docClient.send(new PutItemCommand(command));
}

/**
 * @description Add (append) an Event in the source database.
 */
public async addEvent(event: Event): Promise<void> {
    const eventData = event.get();
    const detail: EventDetail = JSON.parse(eventData["Detail"]);
    const data =
        typeof detail["data"] === "string"
        ? JSON.parse(detail["data"])

```

```

        : detail["data"];

const command = {
  TableName: this.tableName,
  Item: {
    itemType: { S: "EVENT" },
    id: { S: randomUUID() },
    eventTime: { S: detail["metadata"]["timestamp"] },
    eventType: { S: data["event"] },
    event: { S: JSON.stringify(eventData) },
  },
};

if (process.env.NODE_ENV !== "test")
  await this.docClient.send(new PutItemCommand(command));
}
}

```

We implement the class based on a base class (abstraction), allowing us to make a dedicated local test variant as well.

The “big two” methods here are `updateSlot()` on line 101 and `addEvent()` on line 126. Yet again, were we to be more orthodox we might have had two Repositories where we can set a clear split between both concerns. Because the Event is a technical construct, yet in the same domain, and because there is no problematic overlap, I’ll happily take the trade-offs in order to have less code duplication and testing needed.

Notice that both methods are “upsert” behaviors where we never create the same item twice but overwrite in place.

### Success

Like anywhere else in the DDD world, avoid terms that are technological and do not carry semantic meaning. Avoid database-fixated words like `create`, `read` and `update`.

Finally, while it may seem like a weird anti-pattern on line 142 with

```
if (process.env.NODE_ENV !== "test")
```

```
await this.docClient.send(new PutItemCommand(command));
```

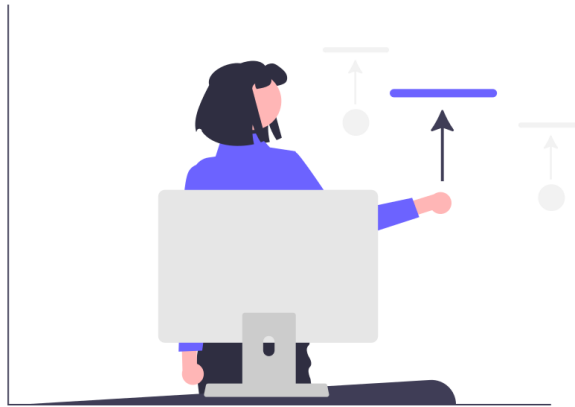
this actually enables unit testing of the majority of the “real” repository without adverse, uncontrolled side effects.

### Information

Microsoft has a lot of good articles on microservices and DDD, for example [this article about Repositories](#).

## Services

*“Service” is an overloaded concept and they’re often over-used in non-DDD contexts. Let’s find out how they are very selectively used in our context.*



*Figure 1.43: Illustration from Undraw*

### Success

**TL;DR: Services** do things that don’t quite fit in Entities or other objects. They are completely stateless. **Application services** are excellent for wrapping non-domain actions like retrieving data from external systems, while **domain services** extend the possibility of acting within the domain. A good example of **domain service** usage is when you need to orchestrate Entities or Aggregates, especially as in our example code we don’t have higher-level Aggregates that can hold such logic.

Services: An overloaded and problematic term. Still, we need them. What did Eric Evans himself actually write about them?

When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as a standalone interface declared as a SERVICE. Define the interface in terms

of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless.

—Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (p. 106)

While we haven't gotten to Entities and Aggregates yet, it's safe to say that **Services** play in the next-highest league, metaphorically speaking.

## Services in the DDD hierarchy

In many projects, you might see services being used very broadly and liberally. This is similar to how in many Node/JS/TS projects you will find tons of helpers, utilities, or other functionally-oriented code. Unwittingly, this way of structuring code will introduce a flattening of hierarchies: Everything is on the same plane, meaning it's hard to understand how pieces fit together and what operates in *which* way on *what*.

Using a more object-oriented approach we can start enforcing a hierarchy like the below:

- Aggregate Root (if needed)
- Aggregate (if needed)
- Entity (if needed)
- Domain Service
- Application Service
- Value Object

### Information

Some of the solutions in the example code are actually basic enough that they need no Entity or higher-level constructs to deal with them (not even services!). As said in the introduction, DDD is sometimes overkilling it by a stretch and then some.

Let's read what Evans writes about layering our services:

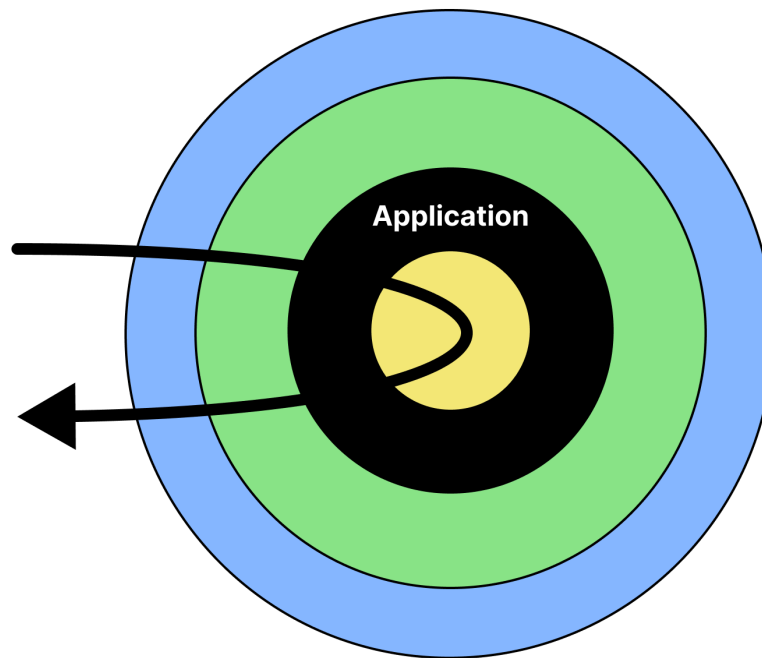
**Application Layer:** Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have a state reflecting the business situation, but it can have a state that reflects the progress of a task for the user or the program.

**Domain Layer:** Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

— Eric Evans (via <https://martinfowler.com/bliki/AnemicDomainModel.html>)

The intuitive difference should be clear, but I've found that it may take a refactoring or two to find the best balance, especially when balancing Domain Services and Aggregates.

## Application Services or use cases?



*Figure 1.44: Application Services reside in the Application layer.*

Application Services and (Clean Architecture) use cases are somewhat equivalent, and we are using both concepts in our example code.

Use cases, like application services, contain no domain-specific business logic; can be used to fetch other domain Entities from external or internal (Repository) sources; may pass off control to Aggregates or Domain Services to execute domain logic; have low [cyclomatic complexity](#).

### Information

See <https://khalilstemmler.com/articles/software-design-architecture/domain-driven-design-vs-clean-architecture/> for more on this.

The way I come to accept both co-existing is like this:

- The use case is strictly equivalent to the **first testable complete unit of code**. This



is where we separate the Lambda infrastructure from the real code itself. This need does not in any way counter the application service notion.

- You can still use application services within the use case as these operate on the same overall conceptual application level and **do things, rather than orchestrate them**.

The main takeaway is that we understand that use cases and Application Services function practically the same, and are positionally equal.

You could, as I have done in other projects, use so-called “[use case interactors](#)” if you’d want to stay consistent with the terminology. In practice, however, I’ve actually only had to use such interactors (or if you’d rather: application services) in my most complex project, [Figmagic](#). I’ve just never had to work on anything else that requires the abstraction, so don’t go expecting that you need it for everything either.

## An application service example

The following is a concrete version of the `VerificationCodeService` used in the Reservation solution. See `code/Reservation/Reservation/src/application/services/VerificationCodeService` ↪ `.ts`.

```
/**
 * @description The `OnlineVerificationCodeService` calls for an online ↪
 * ↪ service
 * to retrieve and passes back a verification code.
 */
class OnlineVerificationCodeService implements VerificationCodeService ↪
  ↪ {
  private readonly securityApiEndpoint: string;

  constructor(securityApiEndpoint: string) {
    this.securityApiEndpoint = securityApiEndpoint;
    if (!securityApiEndpoint) throw new MissingSecurityApiEndpoint();
  }

  /**
   * @description Connect to Security API to generate code.
   */
  async getVerificationCode(slotId: string): Promise<string> {
```

```
const verificationCode = await fetch(this.securityApiEndpoint, {
  body: JSON.stringify({
    slotId: slotId,
  }),
  method: "POST",
}).then((res: Response) => {
  if (res?.status >= 200 && res?.status < 300) return res.json();
});

if (!verificationCode)
  throw new FailedGettingVerificationCodeError("Bad status received↵
↵!");

return verificationCode;
}
```

It has a single public method, `getVerificationCode()`. Using it, one can call an external endpoint and get the implied verification code. Because this is a straightforward and integration-oriented concern, and as we evidently can see there is no business logic here, it's safe to uncontroversially say that—indeed—we are dealing with an application service here.

## Domain Services

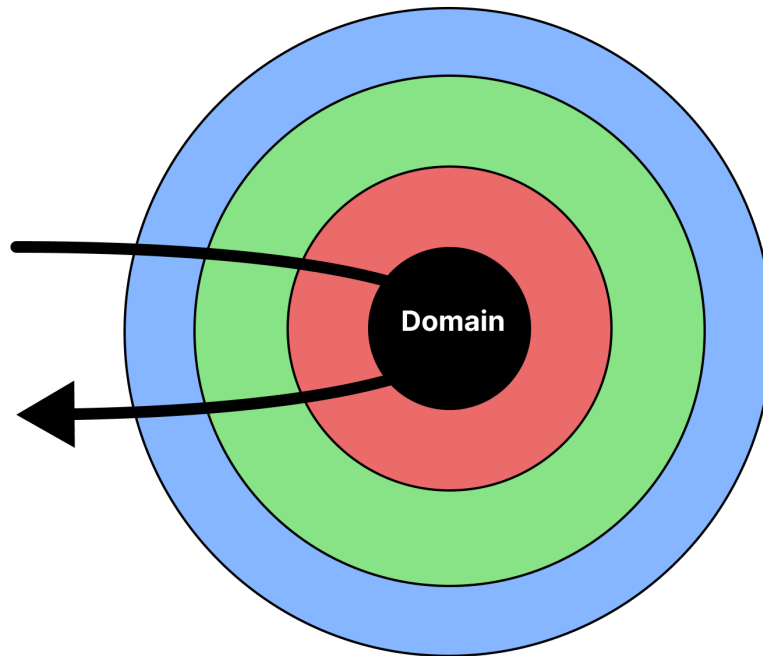


Figure 1.45: Domain Services reside in the Domain layer.

Domain services encapsulate, as expected, domain logic — you'll therefore want this to match the ubiquitous language of your domain. Domain services would be recommended in case you have to interact with multiple Aggregates, for example, otherwise, keep it simple and let it be part of the Aggregate itself.

Next up we are going to check out one of the most important and longest classes in the entire codebase: The `ReservationService`. See `code/Reservation/SlotReservation/↔↔↔ src/domain/services/ReservationService.ts`.

```
import { MikroLog } from "mikrolog";

// Aggregates/Entities
import { Slot } from "../entities/Slot";

// Events
import {
  CancelledEvent,
  CheckedInEvent,
  CheckedOutEvent,
  ClosedEvent,
```

```

    CreatedEvent,
    OpenedEvent,
    ReservedEvent,
    UnattendedEvent,
} from "../events/Event";

// Value objects
import { TimeSlot } from "../valueObjects/TimeSlot";

// Interfaces
import { SlotDTO, Status } from "../../interfaces/Slot";
import { Repository } from "../../interfaces/Repository";
import { Dependencies } from "../../interfaces/Dependencies";
import { ReserveOutput } from "../../interfaces/ReserveOutput";
import { MetadataConfigInput } from "../../interfaces/Metadata";
import { Event } from "../../interfaces/Event";
import { DomainEventPublisherService } from "../../interfaces/↵
    ↵ DomainEventPublisherService";
import { VerificationCodeService } from "../../interfaces/↵
    ↵ VerificationCodeService";

// Errors
import { MissingDependenciesError } from "../../application/errors/↵
    ↵ MissingDependenciesError";

/**
 * @description Acts as the aggregate for Slot reservations (↵
    ↵ representing rooms and
 * their availability), enforcing all the respective invariants ("↵
    ↵ statuses")
 * of the Slot entity.
 */
export class ReservationService {
    private readonly repository: Repository;
    private readonly metadataConfig: MetadataConfigInput;
    private readonly domainEventPublisher: DomainEventPublisherService;
    private readonly logger: MikroLog;

    constructor(dependencies: Dependencies) {
        if (!dependencies.repository || !dependencies.domainEventPublisher)
            throw new MissingDependenciesError();
        const { repository, domainEventPublisher, metadataConfig } = ↵
            ↵ dependencies;

        this.repository = repository;

```

```

    this.metadataConfig = metadataConfig;
    this.domainEventPublisher = domainEventPublisher;
    this.logger = MikroLog.start();
}

/**
 * @description Utility to encapsulate the transactional boilerplate
 * such as calling the repository and event emitter.
 */
private async transact(slotDto: SlotDTO, event: Event, newStatus: ↵
    ↵ Status) {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Make all the slots needed for a single day (same day↵
    ↵ /"today").
 *
 * "Zulu time" is used, where GMT+0 is the basis.
 *
 * @see https://time.is/Z
 */
public async makeDailySlots(): Promise<string[]> {
    const slots: SlotDTO[] = [];

    const startHour = 6; // Zulu time (GMT) -> 08:00 in CEST
    const numberHours = 10;

    for (let slotCount = 0; slotCount < numberHours; slotCount++) {
        const hour = startHour + slotCount;
        const timeSlot = new TimeSlot().startingAt(hour);
        const slot = new Slot(timeSlot.get());
        slots.push(slot.toDto());
    }

    const dailySlots = slots.map(async (slotDto: SlotDTO) => {
        const slot = new Slot().fromDto(slotDto);
        const { slotId, hostName, slotStatus, timeSlot } = slot.toDto();

        const createdEvent = new CreatedEvent({
            event: {
                eventName: "CREATED", // Transient state
                slotId,
                slotStatus,
                hostName,

```

```

        startTime: timeSlot.startTime,
    },
    metadataConfig: this.metadataConfig,
});

    await this.transact(slot.toDto(), createdEvent, slotStatus);
});

await Promise.all(dailySlots);

const slotIds = slots.map((slot: SlotDTO) => slot.slotId);
return slotIds;
}

/**
 * @description Cancel a slot reservation.
 */
public async cancel(slotDto: SlotDTO): Promise<void> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Reserve a slot.
 */
public async reserve(
    slotDto: SlotDTO,
    hostName: string,
    verificationCodeService: VerificationCodeService
): Promise<ReserveOutput> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Check in to a slot.
 */
public async checkIn(slotDto: SlotDTO): Promise<void> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Check out of a slot.
 */
public async checkOut(slotDto: SlotDTO): Promise<void> {
    // Omitted for brevity, clarity, scope
}

```

```

/**
 * @description Re-open a slot.
 */
public async open(slotDto: SlotDTO): Promise<void> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Check for closed slots and set them as being in "↔
↪ closed" invariant state.
 *
 * This is only triggered by scheduled events.
 */
public async checkForClosed(slotDtos: SlotDTO[]): Promise<void> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Close a slot.
 */
private async close(slot: Slot): Promise<void> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Check for unattended slots.
 */
public async checkForUnattended(slotDtos: SlotDTO[]): Promise<void> {
    // Omitted for brevity, clarity, scope
}

/**
 * @description Unattend a slot that has not been checked into.
 */
private async unattend(slot: Slot): Promise<void> {
    // Omitted for brevity, clarity, scope
}
}

```

There's a lot happening there, but it's not quite a [God class](#) either, thank...God?

First of all, the service, even just by glancing at the method names, is clearly handling

domain-specific concerns, such as `unattend()`, `cancel()`, and `makeDailySlots()`.

Most of the code handles roughly similar functionality. For a telling example of the orchestration you might sometimes need, look no further than `makeDailySlots()` on line 70: This is domain logic that would not make sense *inside* the `Slot` but makes perfect sense here in the outer scope. That comment might not make sense yet, but it will after the next couple of pages.

## Constructor

When it gets constructed, it takes a number of dependencies to avoid creating its own imports and links to infrastructural objects. We make properties of the class `private`, and if we can, also `readonly`. In this case, it's no problem to do so. For methods that are called in the use cases they are made public, or else they are private to discourage calling internal functionality from an unwitting outside party.

The constructor had to evolve through a few iterations and it ultimately ended up taking in quite a bit of dependencies and configuration; all in all a good thing since it makes the `ReservationService` less coupled to any infrastructural concerns.

We also have several custom errors that may be thrown if conditions are not valid.

```
private readonly repository: Repository;
private readonly metadataConfig: MetadataConfigInput;
private readonly domainEventPublisher: DomainEventPublisherService;
private readonly logger: MikroLog;

constructor(dependencies: Dependencies) {
  if (!dependencies.repository || !dependencies.domainEventPublisher)
    throw new MissingDependenciesError();
  const { repository, domainEventPublisher, metadataConfig } = ↵
    ↵ dependencies;

  this.repository = repository;
  this.metadataConfig = metadataConfig;
  this.domainEventPublisher = domainEventPublisher;
  this.logger = MikroLog.start();
}
```



## Handling the cancellation

Let's look closer at a use case-oriented method, like `cancel()`. That one looks roughly similar to most of the other operations.

```
public async cancel(slotDto: SlotDTO): Promise<void> {
    const slot = new Slot().fromDto(slotDto);
    const { event, newStatus } = slot.cancel();

    const cancelEvent = new CancelledEvent({
        event,
        metadataConfig: this.metadataConfig
    });

    await this.transact(slot.toDto(), cancelEvent, newStatus);
}
```

The method takes in the Data Transfer Object representation of the `Slot`. We reconstitute it by creating an actual `Slot` Entity object from the DTO and then use the slot's own `cancel()` method, in turn encapsulating the relevant business and validation logic.

Given that nothing broke we can construct the `CancelledEvent` with the local metadata configuration and the event object we receive from the `Slot` itself.

Finally, it's time to run the domain service's `transact()` method that wraps the transactional boilerplate:

```
private async transact(slotDto: SlotDTO, event: Event, newStatus: ↵
    ↵ Status) {
    await this.repository
        .updateSlot(slotDto)
        .then(() => this.logger.log(`Updated status of '${slotDto.slotId}' ↵
            ↵ to '${newStatus}'`));
    await this.repository.addEvent(event);
    await this.domainEventPublisher.publish(event);
}
```

The `domainEventPublisher` will be discussed in the Events section.

**Success**

It might have been even nicer, though more work, to inject some type of service rather than the repository but at some point, we can just be “normal people” and accept the compromise of (in)directly using the repository in the domain layer.

## Entities



Figure 1.46: Illustration from Undraw

### Success

**TL;DR: Entities** are maybe the reason you learned about DDD in the first place. At their heart, the **Entity** is concerned first and foremost about the virtues of conventional OOP and SOLID, and not accepting passing dumb data containers around. Every **Entity** has a unique identity. We use **Entities** to wedge in domain logic on “things” rather than abstract “SomethingServices” and other techno-speak that divorces the domain from the coded implementation. By operating on these things (**Entities**) with clear business/domain logic we solve a lot of poor programming practices. **Entities** and Aggregates are practically the same, with the difference being that an **Entity** is a *thing* while an Aggregate represents a cluster of *things*.

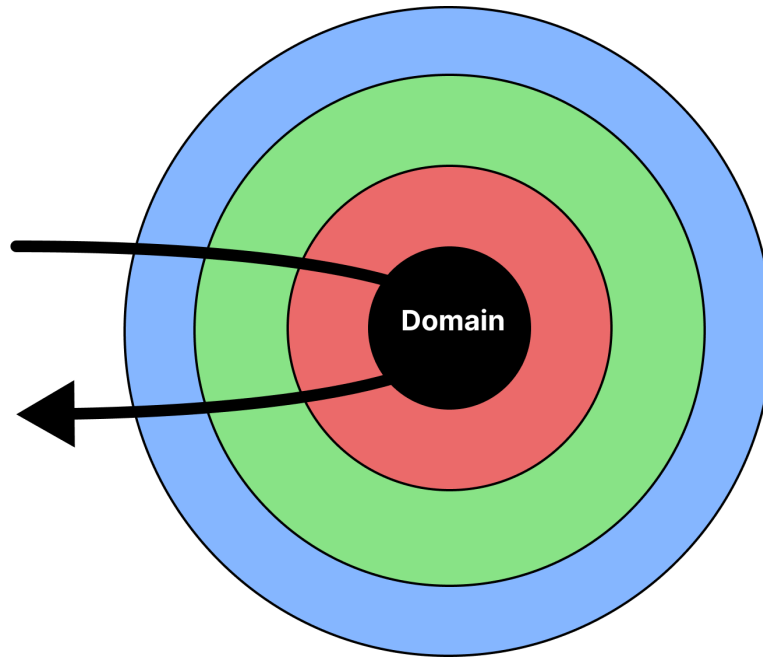


Figure 1.47: Entities reside in the Domain layer.

Entities and Aggregates are perhaps the most “prominent” of the tactical patterns. It’s important to understand that the notion of Entities in database-adjacent contexts and in implementation-oriented tools like Entity Framework *are not the same thing*.

#### Information

**Beware of snake oil salesmen!** DDD has nothing to do with persistence technologies or databases. In fact, when taking a DDD approach and combining it with your required persistence tech, you’ll most likely see that there are no shortcuts—you’ll have to do the modeling and so forth on your own. Tools that “sell” how they map to DDD, like Entity Framework and some Object Relational Mappers, will not help you in any meaningful way.

Both of these concepts are very much related, and it probably makes sense to start with the more general of them: The Entity.

Entities are objects that may mutate (change) over time, and who all have distinct identities. We can think of a `BookClubMember` as something that feels quite right being an

Entity as it implies a person and identity behind it. On the other hand, a `Meeting` may be a simple Value Object (more on these later), as it has neither a unique identity nor will it change after the fact. All in all, it's easy to see how a `BookClubMember` will be a much less simple construct than the `Meeting`.

Our example `BookClubMember` will most likely involve both **data** (such as identity, books read, membership date) and **behavior** (such as updating the member's address). It will also contain its own clear **business rules** attached to such behaviors, where a prospective rule could be something like renewing membership only *after* having paid the member's fee.

Entities are persisted (saved, loaded) with a Repository in the shape of a Data Transfer Object. Before using them in your code, you "turn them into" DTOs or into Entities. DTOs must never be directly mutated.

**Let's make it all ultra clear:** An Entity is an *object*. Most often we represent these as classes. Because a class can contain data we can logically manipulate that data. The way we manipulate the data is through methods on the Entity class that corresponds to our common (ubiquitous) language; We don't let anyone directly manipulate the data on the Entity instance. We can save a representation of the Entity's data (state) with a Repository and we can load back the data and reconstitute it into a valid Entity instance when needed. All of that would happen in the same Bounded Context, in our case, in the same solution (in turn consisting of Lambda functions).

## Splitting data and behavior leads to unmaintainable code

In the world of traditional back-end engineering, you might find something like the below diagram: A service that interacts with several data sources. Because all of these are distinct and separated we have no good idea of who owns and may change, what source. At the bottom we have the faint contours of other services, too.

It's not uncommon that we for example:

- Go with a data synchronization approach, where we duplicate data on our end (this may even be two-way but let's skip that idea for now). While it's easier today in the public cloud to set these things up, many times you'll still face the consequences of having to deal with data decoupled from the business logic (behavior). Further, you

may get a problematic mix of eventual and strong consistency which could break transactional flows.

- Decide to only read back data, making integration easier, but the resiliency and performance worse. An issue here is that at any point where a feature is needed that can update data, you will have a hard time getting that solution to be scalable, secure, and logical as the landscape is now polluted with multiple writers of the same leading data.

Either case will be poor in different ways.

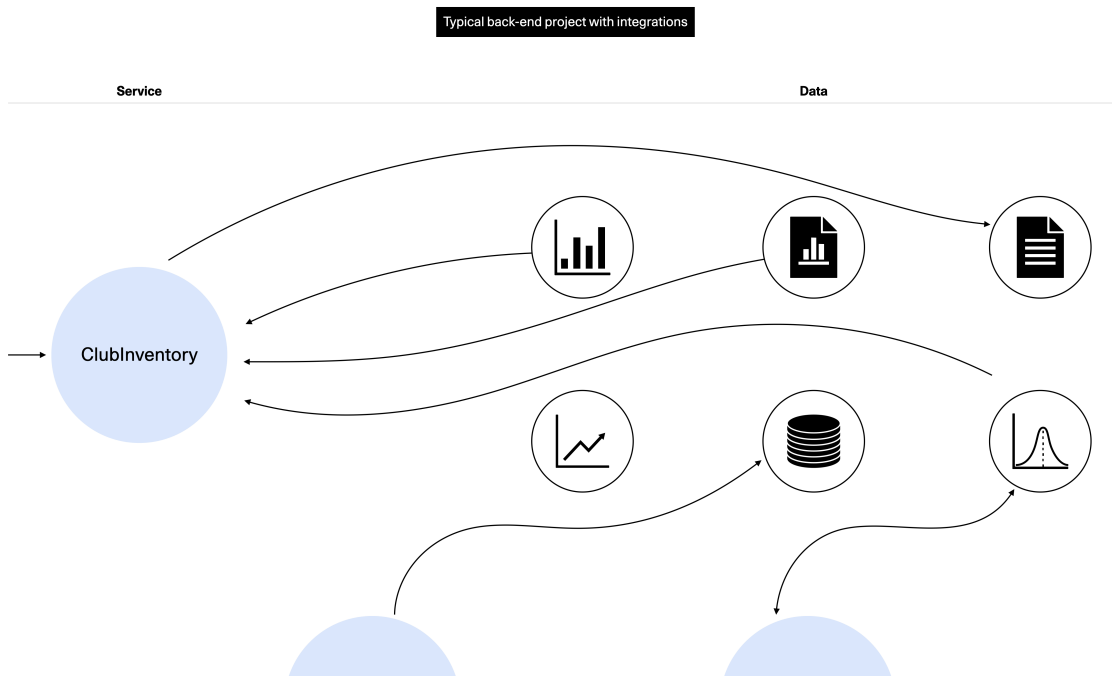


Figure 1.48: Conceptual diagram of tangled integrations where separation of data and behavior leads to uncertainty of who can mutate data in which ways.

While in theory we have decoupling here, in essence, we also have created an even bigger problem: An **anemic domain model**.

### The “anemic domain model”

The [anemic domain model](#) is one that represents objects as shells, or husks, of their true capabilities. They will often be [CRUDdy](#) as they allow for direct mutations through public

getters and setters. It can quickly become hard to understand all the places in a codebase in which the data is manipulated, and how it was done.

### Warning

**Good code does more than just compile.** Some find the criticism around “anemic domain models” academic and roundly wrong. They might argue that their experiences are that it’s just as easy to get things to work, but with less hassle than going full-on with OOP. Remember Robert Martin’s words from [Clean Code: A Handbook of Agile Software Craftsmanship](#): “It is not enough for code to work.” Speaking personally, for me code quality and structure are paramount when building something or when I work with (or coach) other engineers. All this is measurable, once you have access to the code and not just raving to some rando on an internet forum. Using competent tooling you will likely get recommendations to fix issues like this, too. [Good OOP](#) and [refactoring practices](#) are practically institutionalized so this not “just a DDD thing”. The anemic type of objects will maybe *do the job*, but they will become liabilities too. They do not shield the objects from misuse, nor do they express the common language as succinctly.

The opposite of all of this, no surprise, is the “rich domain model”—really no more than a few opinionated ideas on top of your classic object-oriented programming. While that may not technically be the full truth, in our abbreviated version of DDD and the universe, then that explanation is good enough.

## Rich domain models

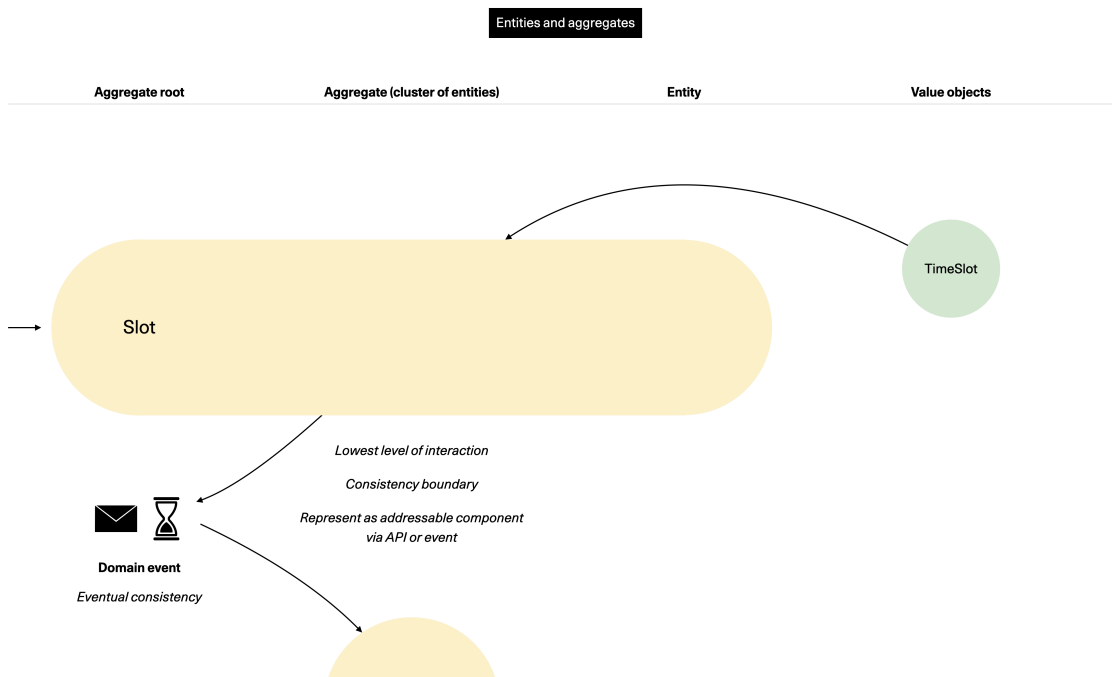
The rich domain model is how Entities solve the question of “who” can do “what” on a specific dataset.

Compared to their anemic brethren, rich domain models (typically Entities and Aggregates) will be easier to understand, will be more resilient to change and disruptions, and are much better encapsulated; we can always know what object can operate on a set of data, and in which ways it does this. **We centralize the majority of our business logic to these domain objects**, and we can entrust them with that because of this encapsula-

tion and overall correctness of behavior.

**A rich model, in the context of our code, is expressive.** It will use a noun (such as a `Book`), rather than a semantic abstraction (say `BookProcessManagerFactory`) and allows us to act on it. Typically this is verb-based — for example `book.recommend()` to correlate with the actual business terms. As we’ve seen many times in this book, we want this to explain 1:1 in our common or ubiquitous language what we are doing.

In the below diagram (note that the use case isn’t the same as in the last diagram!) you can see how a single `Slot` Entity (since it’s the only one, it gets “promoted” to Aggregate Root; more on this in the next section) is the surface that contains all the data and behavior required to create the slot for a room reservation. It also handles the `TimeSlot` Value Object as part of the overall `Slot`. Any changes to the `Slot` gets pushed as a Domain Event so that we can inform other Aggregates or the rest of the technical landscape of ongoing changes.



*Figure 1.49: Diagram for how user interactions to the Slot ensure the complete transactional boundary for any data it holds.*

We expose the operations on the `Slot` as calls one can make to our API Gateway, firing the relevant Lambda functions that will orchestrate, through use cases, the operations. Thus we can be totally sure that specific operations are only permissible in a deterministic



flow, rather than leak it across our complete solution. Any time we are dealing with an Aggregate or Aggregate Root (as we are here, as the Entity is all alone) we publish a Domain Event detailing what happened, such as `SlotReserved`.

## Invariants

Invariants are “consistency rules that must be maintained whenever data changes” (Evans 2004, p. 128). A complete domain model has no holes in it, in other words, there is no possibility for it to be invalid. This is sometimes called the “[always-valid](#)” model. I highly recommend reading that link, as we will keep it short here.

To reach an always-valid domain model, what would you need to keep in mind?

- Your domain model should be valid at all times.
- For that, do the following:
  - [Vladimir Khorikov: Always valid vs not always valid domain model](#)

### Information

See also the following article from Microsoft for more on designing domain-layer validations: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/domain-model-layer-validations>

In plain English, by having moved all the actual domain logic and validations and invariants to the domain layer (where the Entity is the core part), we’re already on a good path. We’ll see in the code samples some of the basic ways we can handle unique invariants that must be enforced.

## Before we move on

Before we go to the code, let’s revisit some highlights.

- Entities are objects that have a unique identity. They are the most closely connected to the domain and its business logic of all DDD concepts.
- Entities represent our “dumb data” as actual “things” (nouns) and makes it smart by enabling us a programmatic way to interact with the data in a logical manner rather than just supplying getters and setters to a POJO/POCO/JSON object.
- Entities typically use verbs to express their commands—its public interface.
- We use the ubiquitous language to name these actions and anything else to do with the Entity.
- Entities must always be valid. *Invariants* is the preferred term for our consistency (and validation) rules.

## The Slot entity

Be ready for one of our biggest and most important classes, the Slot, as seen at code/↵  
 ↵ Reservation/SlotReservation/src/domain/entities/Slot.ts.

```
import { randomUUID } from "crypto";

import { SlotCreateInput, SlotDTO, Status } from "../../interfaces/Slot↵
  ↵ ";
import { TimeSlotDTO } from "../../interfaces/TimeSlot";
import { MakeEventInput } from "../../interfaces/Event";

import { CheckInConditionsNotMetError } from "../../application/errors/↵
  ↵ CheckInConditionsNotMetError";
import { CheckOutConditionsNotMetError } from "../../application/errors↵
  ↵ /CheckOutConditionsNotMetError";
import { CancellationConditionsNotMetError } from "../../application/↵
  ↵ errors/CancellationConditionsNotMetError";
import { ReservationConditionsNotMetError } from "../../application/↵
  ↵ errors/ReservationConditionsNotMetError";

/**
 * @description The `Slot` entity handles the lifecycle
 * and operations of the (time) slots that users can
 * reserve.
 *
 * @example You can create it at once:
```

```

*   `
*   const slot = new Slot({
*     startTime: "2022-07-29T12:00:00.000Z",
*     endTime: "2022-07-29T13:00:00.000Z"
*   });
*   `
*
*   You can also reconstitute a Slot from a SlotDTO
*   loaded from a repository:
*   `
*   const slot = new Slot().fromDto(slotDto);
*   `
*/
export class Slot {
  private slotId: string;
  private hostName: string;
  private timeSlot: TimeSlotDTO;
  private slotStatus: Status;
  private createdAt: string;
  private updatedAt: string;

  constructor(input?: SlotCreateInput) {
    this.slotId = "";
    this.hostName = "";
    this.timeSlot = {
      startTime: "",
      endTime: "",
    };
    this.slotStatus = "OPEN";
    this.createdAt = "";
    this.updatedAt = "";

    if (input) this.make(input);
  }

  /**
   * @description Create a valid, starting-state ("open") invariant of ↵
   * ↵ the Slot.
   */
  private make(input: SlotCreateInput): SlotDTO {
    const { startTime, endTime } = input;
    const currentTime = this.getCurrentTime();

    this.slotId = randomUUID().toString();
    this.hostName = "";
  }

```

```
        this.timeSlot = {
            startTime,
            endTime,
        };
        this.slotStatus = "OPEN";
        this.createdAt = currentTime;
        this.updatedAt = currentTime;

        return this.toDto();
    }

    /**
     * @description Reconstitute a Slot from a Data Transfer Object.
     */
    public fromDto(input: SlotDTO): Slot {
        this.slotId = input["slotId"];
        this.hostName = input["hostName"];
        this.timeSlot = input["timeSlot"];
        this.slotStatus = input["slotStatus"];
        this.createdAt = input["createdAt"];
        this.updatedAt = input["updatedAt"];

        return this;
    }

    /**
     * @description Return data as Data Transfer Object.
     */
    public toDto(): SlotDTO {
        return {
            slotId: this.slotId,
            hostName: this.hostName,
            timeSlot: this.timeSlot,
            slotStatus: this.slotStatus,
            createdAt: this.createdAt,
            updatedAt: this.updatedAt,
        };
    }

    /**
     * @description Remove host name from data.
     */
    public removeHostName(): void {
        this.hostName = "";
    }
}
```

```
/**
 * @description Update host name to new value.
 */
public updateHostName(hostName: string): void {
    this.hostName = hostName;
}

/**
 * @description Updates the common fields to reflect a new `Status`,
 * and also updates the `updatedAt` field.
 *
 */
public updateStatus(status: Status): void {
    this.slotStatus = status;
    this.updatedAt = this.getCurrentTime();
}

/**
 * @description Returns the start time of the time slot.
 */
private getStartTime(): string {
    return this.timeSlot.startTime;
}

/**
 * @description Has the time slot's end time already passed?
 */
public isEnded(): boolean {
    if (this.getCurrentTime() > this.timeSlot.endTime) return true;
    return false;
}

/**
 * @description Check if our 10 minute grace period has ended,
 * in which case we want to open the slot again.
 */
public isGracePeriodOver(): boolean {
    if (
        this.getCurrentTime() >
        this.getGracePeriodEndTime(this.timeSlot.startTime)
    )
        return true;
    return false;
}
```

```
/**
 * @description Returns the end of the grace period until a reserved
 * slot is deemed unattended and returns to open state.
 */
private getGracePeriodEndTime(startTime: string): string {
    const minutes = 10;
    const msPerMinute = 60 * 1000;

    return new Date(
        new Date(startTime).getTime() + minutes * msPerMinute
    ).toISOString();
}

/**
 * @description Returns the current time as an ISO string.
 */
private getCurrentTime(): string {
    return new Date().toISOString();
}

/**
 * @description Can this `Slot` be cancelled?
 */
private canBeCancelled(): boolean {
    if (this.slotStatus !== "RESERVED") return false;
    return true;
}

/**
 * @description Can this `Slot` be reserved?
 */
private canBeReserved(): boolean {
    if (this.slotStatus !== "OPEN") return false;
    return true;
}

/**
 * @description Can this `Slot` be checked in to?
 */
private canBeCheckedInTo(): boolean {
    if (this.slotStatus !== "RESERVED") return false;
    return true;
}
```

```

/**
 * @description Can this `Slot` be checked out of?
 */
private canBeCheckedOutOf(): boolean {
  if (this.slotStatus !== "CHECKED_IN") return false;
  return true;
}

/**
 * @description Can this `Slot` be unattended?
 */
private canBeUnattended(): boolean {
  if (this.slotStatus === "RESERVED") return true;
  return false;
}

/**
 * @description Updates a Slot to be in `OPEN` invariant state by ↵
 * ↵ cancelling the current state.
 *
 * * Can only be performed in `RESERVED` state.
 *
 * * @emits `CANCELLED`
 */
public cancel(): SlotCommand {
  if (!this.canBeCancelled())
    throw new CancellationConditionsNotMetError(this.slotStatus);

  const newStatus = "OPEN";

  this.removeHostName();
  this.updateStatus(newStatus);

  return {
    event: {
      eventName: "CANCELLED", // Transient state
      slotId: this.slotId,
      slotStatus: this.slotStatus,
      hostName: this.hostName,
      startTime: this.getStartTime(),
    },
    newStatus,
  };
}

```

```
/**
 * @description Updates a Slot to be in `RESERVED` invariant state.
 *
 * Can only be performed in `OPEN` state.
 *
 * @emits `RESERVED`
 */
public reserve(hostName: string): SlotCommand {
  if (!this.canBeReserved())
    throw new ReservationConditionsNotMetError(this.slotStatus);

  const newStatus = "RESERVED";

  this.updateHostName(hostName || "");
  this.updateStatus(newStatus);

  return {
    event: {
      eventName: newStatus,
      slotId: this.slotId,
      slotStatus: newStatus,
      hostName: this.hostName,
      startTime: this.getStartTime(),
    },
    newStatus,
  };
}

/**
 * @description Updates a Slot to be in `CHECKED_IN` invariant state.
 *
 * Can only be performed in `RESERVED` state.
 *
 * @emits `CHECKED_IN`
 */
public checkIn(): SlotCommand {
  if (!this.canBeCheckedInTo())
    throw new CheckInConditionsNotMetError(this.slotStatus);

  const newStatus = "CHECKED_IN";
  this.updateStatus(newStatus);

  return {
    event: {
      eventName: newStatus,
```



```

        slotId: this.slotId,
        slotStatus: newStatus,
        hostName: this.hostName,
        startTime: this.getStartTime(),
    },
    newStatus,
};
}

/**
 * @description Updates a Slot to be in `OPEN` invariant state by ↔
 * ↔ checking out from the current state.
 *
 * Can only be performed in `CHECKED_IN` state.
 *
 * @emits `CHECKED_OUT`
 */
public checkOut(): SlotCommand {
    if (!this.canBeCheckedOutOf())
        throw new CheckOutConditionsNotMetError(this.slotStatus);

    const newStatus = "OPEN";
    this.updateStatus(newStatus);
    this.removeHostName();

    return {
        event: {
            eventName: "CHECKED_OUT", // Transient state
            slotId: this.slotId,
            slotStatus: newStatus,
            hostName: this.hostName,
            startTime: this.getStartTime(),
        },
        newStatus,
    };
}

/**
 * @description Updates a Slot to be in "open" invariant state.
 *
 * @emits `OPENED`
 */
public open(): SlotCommand {
    const newStatus = "OPEN";
    this.updateStatus(newStatus);

```

```

    return {
      event: {
        eventName: "OPENED",
        slotId: this.slotId,
        slotStatus: newStatus,
        hostName: "",
        startTime: this.getStartTime(),
      },
      newStatus,
    };
  }

  /**
   * @description Updates a Slot to be in "closed" invariant state.
   *
   * @emits `CLOSED`
   */
  public close(): SlotCommand {
    const newStatus = "CLOSED";
    this.updateStatus(newStatus);

    return {
      event: {
        eventName: newStatus,
        slotId: this.slotId,
        slotStatus: newStatus,
        hostName: this.hostName,
        startTime: this.getStartTime(),
      },
      newStatus,
    };
  }

  /**
   * @description Set a slot as being in `OPEN` invariant state if it ↔
   * ↔ is unattended.
   *
   * State change can only be performed in `RESERVED` state.
   *
   * This is only triggered by scheduled events.
   *
   * @emits `UNATTENDED`
   */
  public unattend(): SlotCommand | void {

```

```

    if (!this.canBeUnattended()) return;

    const newStatus = "OPEN";
    this.updateStatus(newStatus);
    this.removeHostName();

    return {
      event: {
        eventName: "UNATTENDED", // Transient state
        slotId: this.slotId,
        slotStatus: newStatus,
        hostName: this.hostName,
        startTime: this.getStartTime(),
      },
      newStatus,
    };
  }
}

/**
 * @description The finishing command that the `Slot` sends back when ↔
 * ↔ done.
 */
export interface SlotCommand {
  event: MakeEventInput;
  newStatus: Status;
}

```

There's a bunch of private and public methods here, with a slightly higher public method count than on the private side. You'll notice that there are a couple of patterns that keep repeating like those that return `SlotCommand` and those that check rules.

### Information

It might have been more “effective” in a strict, technocratic sense to leave `removeHostName()`, `updateStatus()` and `getCurrentTime()` out as functions and just directly manipulate the values. I am sure you know I will complain about how that breaks our possibility to encapsulate and truly trust our provided mechanisms if we gave even an inch away on this matter.

## The constructor

Let's see:

```
private slotId: string;
private hostName: string;
private timeSlot: TimeSlotDTO;
private slotStatus: Status;
private createdAt: string;
private updatedAt: string;

constructor(input?: SlotCreateInput) {
  this.slotId = '';
  this.hostName = '';
  this.timeSlot = {
    startTime: '',
    endTime: ''
  };
  this.slotStatus = 'OPEN';
  this.createdAt = '';
  this.updatedAt = '';

  if (input) this.make(input);
}
```

Our internal private fields represent the data we store. They can't be retrieved from outside the class which is perfect—this is one of the easiest but most important wins when using DDD or good OOP for that matter. Now, users will have to use our exposed public methods to actually mutate our data.

When constructed, if we lack input, we will assume an almost barren state. We've also set up a basic private `make()` method that will return back the starting-state invariant which we call “open” if slot creation input is passed in.

```
/**
 * @description Create a valid, starting-state ("open") invariant of ↔
 * ↔ the Slot.
 */
private make(input: SlotCreateInput): SlotDTO {
  const { startTime, endTime } = input;
  const currentTime = this.getCurrentTime();
```

```
this.slotId = randomUUID().toString();
this.hostName = '';
this.timeSlot = {
  startTime,
  endTime
};
this.slotStatus = 'OPEN';
this.createdAt = currentTime;
this.updatedAt = currentTime;

return this.toDto();
}
```

## Reconstitute from a DTO

Now for one of the most important private methods: `fromDto()`. This will enable us to create a class representation (Slot Entity) from a Data Transfer Object. It's nothing hard nor magical, just:

```
/**
 * @description Reconstitute a Slot from a Data Transfer Object.
 */
public fromDto(input: SlotDTO): Slot {
  this.slotId = input['slotId'];
  this.hostName = input['hostName'];
  this.timeSlot = input['timeSlot'];
  this.slotStatus = input['slotStatus'];
  this.createdAt = input['createdAt'];
  this.updatedAt = input['updatedAt'];

  return this;
}
```

This acts as our public setter method. In this case we can practically always trust the input but an improvement would be to add validation logic at this point.

By returning a reference to the instance we can allow chaining of commands making the programmatic use a little easier.

## Make into a DTO

There is no way for us to transport a class across systems, so we will have to represent the key data in some way. Luckily this is easy.

```
/**
 * @description Return data as Data Transfer Object.
 */
public toDto(): SlotDTO {
    return {
        slotId: this.slotId,
        hostName: this.hostName,
        timeSlot: this.timeSlot,
        slotStatus: this.slotStatus,
        createdAt: this.createdAt,
        updatedAt: this.updatedAt
    };
}
```

The fields act as a well-known interface/type (`SlotDTO`) and we can now trivially pass this to our persistence mechanism or elsewhere where we don't, or can't, use the actual `Slot` Entity class.

## Use case #1: Domain logic for checking if we can reserve and cancel

Business logic. Domain logic. Both sound *big*. Dangerous. In our case it's literally a check on the expected, valid `slotStatus`.

```
/**
 * @description Can this `Slot` be reserved?
 */
private canBeReserved(): boolean {
    if (this.slotStatus !== 'OPEN') return false;
    return true;
}
```

Now that's some nice, basic logic right there! No need for enums or anything, we just need to check for an open status.

```

/**
 * @description Can this `Slot` be cancelled?
 */
private canBeCancelled(): boolean {
  if (this.slotStatus !== 'RESERVED') return false;
  return true;
}

```

The same goes for the cancellation check, we need to know if we are reserved or not. Both, as seen, return boolean results which makes it a simple-to-understand and expressive check.

Nothing is blocking you to conduct much deeper checking, though that seems overboard in our example code.

## Use case #2: Is the grace period over?

Our Domain Service, ReservationService, calls each Slot's `isGracePeriodOver()` method when checking if we have any reservations that have expired their 10-minute grace period.

```

/**
 * @description Check if our 10 minute grace period has ended,
 * in which case we want to open the slot again.
 */
public isGracePeriodOver(): boolean {
  if (this.getCurrentTime() > this.getGracePeriodEndTime(this.timeSlot.↵
    ↵ startTime)) return true;
  return false;
}

/**
 * @description Returns the end of the grace period until a reserved
 * slot is deemed unattended and returns to open state.
 */
private getGracePeriodEndTime(startTime: string): string {
  const minutes = 10;
  const msPerMinute = 60 * 1000;

  return new Date(new Date(startTime).getTime() + minutes * msPerMinute↵
    ↵ ).toISOString();
}

```

```
}

```

The internal logic is here a tiny bit more elaborate than the super-simple ones from the last example. All the logic around this is neatly stored within the Entity and we are left with a clean, nice public interface to get our answer.

### Use case #3: Reserving a slot

Here's now an example of the actual reservation logic.

```
/**
 * @description Updates a Slot to be in `RESERVED` invariant state.
 *
 * Can only be performed in `OPEN` state.
 *
 * @emits `RESERVED`
 */
public reserve(hostName: string): SlotCommand {
  if (!this.canBeReserved()) throw new ReservationConditionsNotMetError(
    ↪ (this.slotStatus);

  const newStatus = 'RESERVED';

  this.updateHostName(hostName || '');
  this.updateStatus(newStatus);

  return {
    event: {
      eventName: newStatus,
      slotId: this.slotId,
      slotStatus: newStatus,
      hostName: this.hostName,
      startTime: this.getStartTime()
    },
    newStatus
  };
}
```

It will throw an error if it cannot be reserved, which is [cruder than how we could do it](#). Nevertheless, this seems like a reasonable version 1 of our solution. Next, we will set a



new status, update the host name and status internally, and then return a `SlotCommand` which is a type of object that we can create an actual Domain Event from later. Note how, at this point, we have not persisted anything, just made sure that it's all valid, our object is in a regulated and valid state, and that we feed back the basis of our upcoming event for our integration purposes.

## Aggregates

*If you were to super-charge the Entity with transactional responsibilities and the ability to contain a cluster of objects, then you would get the Aggregate.*

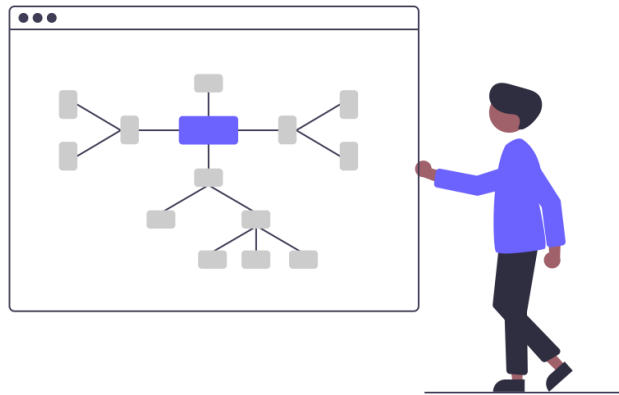
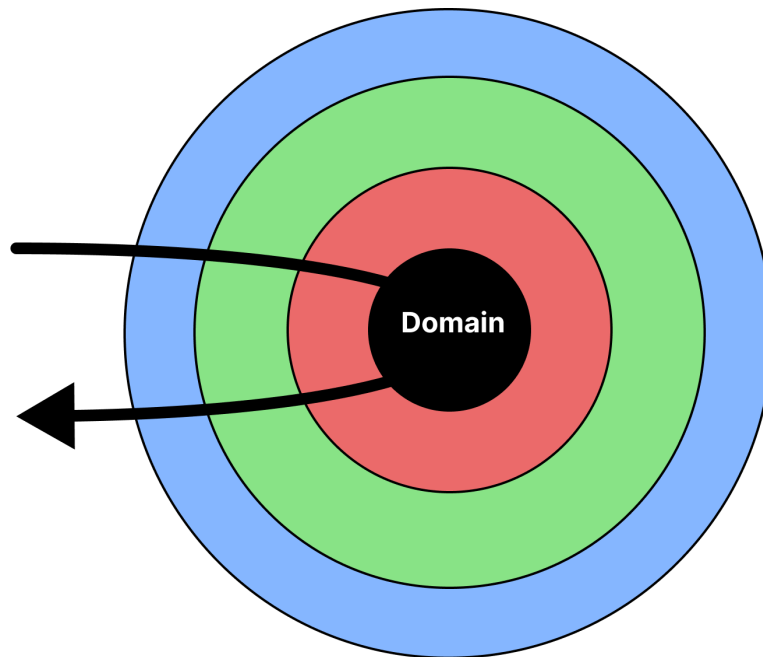


Figure 1.50: Illustration from Undraw

### Success

**TL;DR:** The **Aggregate** can be confusing. It has two common meanings. The “correct” and orthodox one is that the **Aggregate** is simply an Entity that itself “owns” or links other Entities in a logical whole. This entails that **Aggregates**, like Entities, each has their own unique identity. The highest-level Aggregate is called the **Aggregate Root**. There must be no way to access “deeper” Entities without passing the **Aggregate Root**, or whichever other construct is highest. For the secondary, more colloquial meaning it can mean the actual “data object” that we are operating on. While not technically always correct, I find the **Aggregate** term slightly better than saying things like “I will access the X Entity through the API”. At least for me, I find it better at expressing a data source, while Entity is more of a thing. Moreover, the **Aggregate** acts as the *transaction boundary* so it completely deals with all the data that it pertains to. You should never modify more than a single **Aggregate** per database transaction. Any changes to the **Aggregate** result in the **Aggregate** publishing a Domain Event.



*Figure 1.51: Aggregates reside in the Domain layer.*

For the “truth” on the matter of Aggregates, we will look no further than to the Big Blue Book:

An Aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each Aggregate has a root and a boundary. The boundary defines what is inside the Aggregate. The root is a single, specific Entity contained in the Aggregate. The root is the only member of the Aggregate that outside objects are allowed to hold references to, although objects within the boundary may hold references to each other. Entities other than the root have local identities, but that identity needs to be distinguishable only within the Aggregate because no outside object can ever see it out of the context of the root Entity.

— *Domain Driven Design: Tackling Complexity in the Heart of Software* (Evans, p. 126-127)

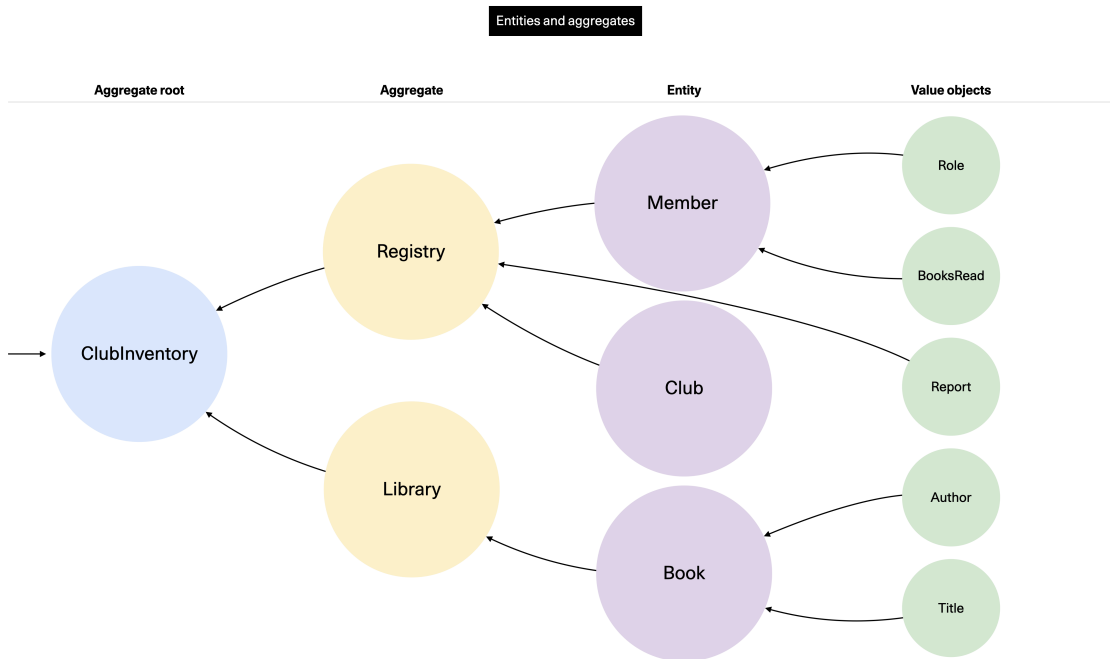
**Danger**

The Aggregate is the most complex pattern, for sure. It is the one I myself have had to contend most with. I hope to get across the fundamentals here, but let it be known that *a lot* of paper has been printed around various ways to think about them, referencing other Aggregate Roots, data modeling, efficient persistence, and so on. Get a coffee, you deserve it and don't sweat it all here and now. Read up and evolve when you have gotten your [sea legs](#).

Revisiting our relations between Aggregates and Entities we see the fundamental items to understand include:

- Aggregates are mostly just Entities with extra responsibilities; more on these in a moment. The opposite case is however not necessarily true.
- Aggregates are the only objects that access and operate on Entities.
- An Aggregate Root is an object that can access the root object (Entity) that itself may collect a group of Entities. The Aggregate Root concept becomes more important and pronounced when you have a rich domain with relations between Entities.

The below conceptual diagram should give you an idea of how this might actually work.



*Figure 1.52: Conceptual demonstration of an deeply-clustered Aggregate Root. This particular model may or may not make actual sense (given that it's simply an example) but we can be quite certain that the orchestration of this will be non-trivial.*

Being an Aggregate means that you add a number of additional characteristics to the Entity's existential features:

- **Consistency enforcement** is Job #1 for the Aggregate. It has to ensure changes are correct and consistent.
- **Acts as a transaction boundary:** Aggregates use their own business/domain logic to modify data. You must not use more than a single Aggregate instance per transaction.
- **Enforces the hierarchy of Entities.** Multiple Entities and/or Value Objects may be part of the same transaction, and updating them must always be done as a shared transaction only *after verification* of rules and checks.
- The rule of thumb for referencing other Aggregates is that **any Entities that must be in a strongly consistent state should be within the same Aggregate boundary**. Anything else is some other Aggregate's job and maybe eventually consistent. Work to minimize Aggregate boundaries to the smallest, logically possible ones.

- **Domain events are emitted** to integrate with other systems (and Aggregates) whenever a transaction is completed.

And as with other object types, **Aggregates use the ubiquitous language to reflect the domain model.**

### Information

See Vlad Khononov's *Learning Domain Driven Design: Aligning Software Architecture and Business Strategy* (2021, p.84-92).

### Information

For more web links on this subject, see:

- [What Are Domain-Driven Design Aggregates?](#)
- [DDD Aggregate](#)

## Do we have Aggregates in the example project?

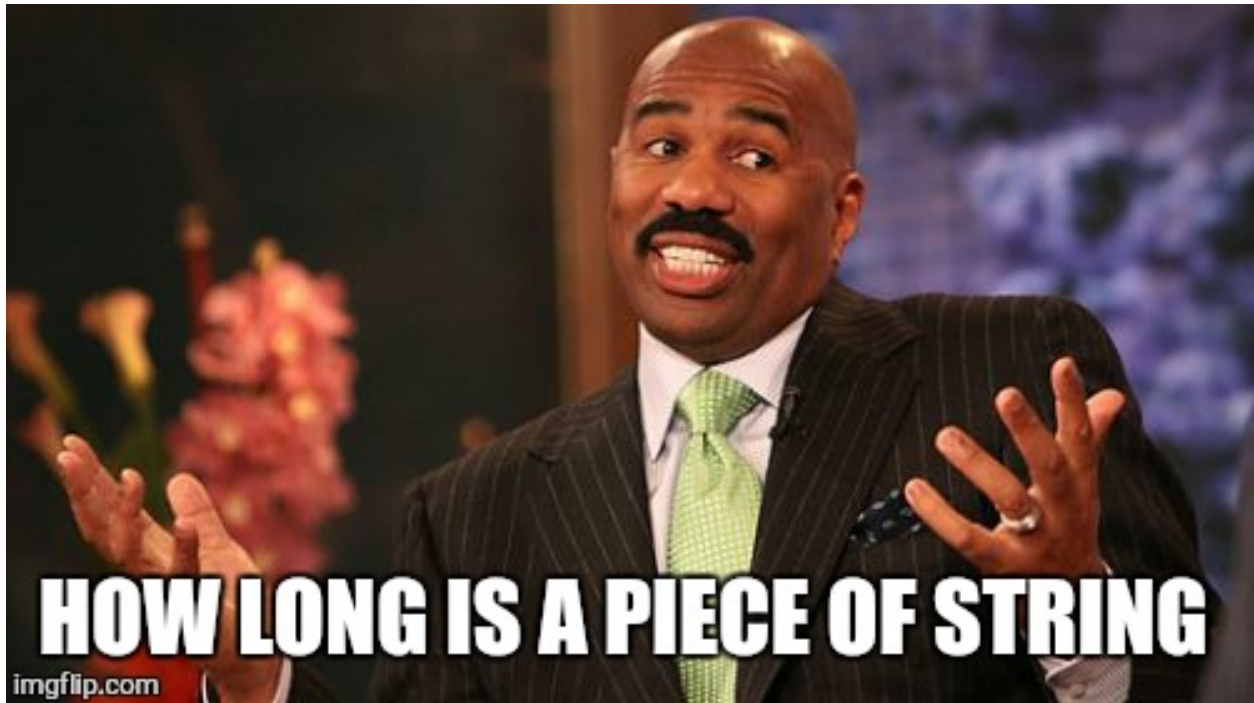
Yes and no.

No, because there is no high-level object of that variety that is containing multiple Entities or similar objects. We only have the “flat” Entity named `Slot`.

Yes, because our only Entity for the above reason automatically becomes the Aggregate Root. For practical reasons, we might not want to use that term all the time when we work, especially not if there is no need for such a concept in a basic domain model like the one in our example project.

We will see later in this section how I am handling this case in the project.

## How large is an Aggregate?



*Figure 1.53: Come on, the question was begging for this meme.*

Vaughn Vernon recommends in *Implementing Domain Driven Design* that you should strive to **design small Aggregates** (p. 355-359). He shows how large-cluster Aggregates will scale and perform poorly, as well as become very complicated to reason about. The technical issues stem from factors such as needing to load more data, possibly from more sources, while also exposing more transactional areas for failure. The bigger the Aggregate the more cumbersome it will become regardless of dimension, whether technical or cognitive.

If a given operation needs to have strong consistency across multiple Aggregates, then that should give a hint that there is poor design at play.

## What we mean with transactions

A *Transaction* is the broad DDD term for committing something from start to (persisted) finish.

Recall how it's already been stated that the Aggregate Root (and Entities, in essence)

serves as a [consistency boundary or transactional boundary](#). What that means in practice is that:

- Anything that has to do with controlling that the data is correct (valid);
- That the data gets persisted in the right way;
- That any nested or clustered objects change together;

is the responsibility of the respective Aggregate. Anything outside the direct responsibility of the Aggregate is someone else's work. You should attempt to shed as much load as possible while staying truthful to the business domain when you decide what work is on the shoulders of an Aggregate. Vernon also writes on that issue:

Just because you are given a use case that calls for maintaining consistency in a single transaction doesn't mean you should do that. Often, in such cases, the business goal can be achieved with eventual consistency between Aggregates. The team should critically examine the use cases and challenge their assumptions, especially when following them as written would lead to unwieldy designs.

— *Implementing Domain Driven Design* (Vernon 2013, p. 359)

Eventual consistency in practice means that we'll offload the change to some other system, rather than stay inside the same process and synchronously await the change. Or worse: have to keep all that logic in "our" solution. Vernon discussed the common question of when to use (and not to use) eventual consistency. Evans answered that someone else's job should always be deemed eventually consistent from "our" angle:

When examining the use case (or story), ask whether it's the job of the user executing the use case to make the data consistent. If it is, try to make it transactionally consistent, but only by adhering to the other rules of Aggregates. If it is another user's job, or the job of the system, allow it to be eventually consistent. That bit of wisdom not only provides a convenient tiebreaker, but it helps us gain a deeper understanding of the domain. It exposes the real system invariants: the ones that must be kept transactionally consistent.

— *Implementing Domain Driven Design* (Vernon 2013, p. 367)

OK, so that sounds good and all but how does that actually work? Easy: Domain Events.



## Aggregates emit Domain Events

We haven't discussed Domain Events in detail yet, as these will come up in an upcoming section, but the way in which the Aggregate informs the rest of the landscape is through events, specifically in the DDD context, Domain Events.

A Domain Event is, in short, an event (or message) pushed to some asynchronous messaging technology where consumers can subscribe to new events unfolding. We give events their own identity, in effect transforming them from just a blob with some data into a fully-fleshed Domain Event that “speaks” our domain's language. By using them we can stitch together interactions across many systems in our landscape without foregoing any of the rich vocabularies we have created through DDD and EventStorming.

Only Aggregates must emit events since they enforce business rules. In practice this should be done post-fact as a result of an operation, for example:

1. User makes a request to our system/service (“Aggregate”)
2. Our system instantiates a class for our Aggregate and fulfills the operation (if valid)
3. Our system emits an event to notify us that the operation has occurred

Let's look more at this later.

## Our domain service as a stand-in

Our code base for the Reservation solution has the following more substantial ingredients:

- A number of use cases
- A number of application services
- The Slot Entity
- The ReservationService Domain Service

**Information**

We inspected the code already in the Services section. For brevity, I will avoid reproducing it here once again. Instead, we will look at selected sections. The code itself is located at `ReservationService.ts` if you want to see the full source.

Before looking too intently at the code itself, I'll clarify the way that I am using a Domain Service to do the Aggregate-type operations on top of our `Slot` Entity.

**Why is this handled in a Domain Service rather than directly in the use case?**

Good question!

In typical DDD fashion we would not want to move the persistence concern (even with a Repository) into the domain layer, but want to keep these in the application layer. However, the actual code that gets executed has more of a domain character than pure application chaff. We can see this in the nature of the code itself—such as orchestrating the Entity and creating events—as well as checking our resulting imports: we directly link to the `Slot` Entity and the events, all of which are in the domain.

Secondly, there is a lot of wiring that needs to be done. By placing all of that into a stateless, separate class rather than in the functionally oriented use cases we can avoid having to rewrite a lot of code.

At the end of the day, it is not about being orthodox but being clear and domain-oriented in our code. I am sure Evans and Vernon and others might find any number of details to complain about, but the way *it actually is* implemented is hopefully clear enough; this is the real goal, not dogmatism.

**Why is this a Domain Service and not an Aggregate?**

Services are something we try to avoid in DDD (as long as we can put behavior on “things” instead) and the uses I understand them to be best for include typical “heavy lifting”, not necessarily being important orchestrators. The `SlotReservation` aggregate has quite a

bit of such orchestration happening on the Slot Entity and more.

The service is stateless and identity-less, so it can't be an Entity or Aggregate.

It's not an Entity because it doesn't handle anything concrete *on* anything.

I really want to avoid injecting Repositories or Domain Event Publishers into the Slot Entity/Aggregate, so something else has to abstract that. However, not even a Domain Layer *should* access such things, but it's generally not seen as a capital offense. :sweat\_smile:

### Information

Here's an example of a Stack Overflow answer that also makes the point that it's acceptable to inject a Repository into a Domain Service: <https://softwareengineering.stackexchange.com/questions/281111/should-i-inject-a-repository-into-a-domain-service>

It *does act* like an Aggregate as it functions as the “entry point” to the Slot Entity that we actually operate on and persist. We also send the Domain Events from here: it, therefore, acts as the transaction boundary.

And that's how we ended up in this compromise. Don't let DDD become dogma. Be humble and realistic and if it makes sense to you and you can explain the reasoning, at the very least we are dealing with a considered and deliberate design which after all is the real goal.

## Examples from our project

### Use case #1: Make daily slots

The first publicly accessible use case is for making daily slots. This one is also one of the longer ones as it has to deal with more setups than the other ones. It is run once per day, no more.

```
/**
 * @description Make all the slots needed for a single day (same day/"↩
 * ↩ today").
 *
 * "Zulu time" is used, where GMT+0 is the basis.
```

```

*
* @see https://time.is/Z
*/
public async makeDailySlots(): Promise<string[]> {
    const slots: SlotDTO[] = [];

    const startHour = 6; // Zulu time (GMT) -> 08:00 in CEST
    const numberHours = 10;

    for (let slotCount = 0; slotCount < numberHours; slotCount++) {
        const hour = startHour + slotCount;
        const timeSlot = new TimeSlot().startingAt(hour);
        const slot = new Slot(timeSlot.get());
        slots.push(slot.toDto());
    }

    const dailySlots = slots.map(async (slotDto: SlotDTO) => {
        const slot = new Slot().fromDto(slotDto);
        const { slotId, hostName, slotStatus, timeSlot } = slot.toDto();

        const createdEvent = new CreatedEvent({
            event: {
                eventName: 'CREATED', // Transient state
                slotId,
                slotStatus,
                hostName,
                startTime: timeSlot.startTime
            },
            metadataConfig: this.metadataConfig
        });

        await this.transact(slot.toDto(), createdEvent, slotStatus);
    });

    await Promise.all(dailySlots);

    const slotIds = slots.map((slot: SlotDTO) => slot.slotId);
    return slotIds;
}

```

The upper half is a loop to produce new Slots using the internal `makeSlot()` method. We are creating `TimeSlot` Value Objects in order to get the correct, valid representation of the time object as we create the Slot.

For the bottom half we'll:

- Loop through the Slots;
- Return [Promises](#) in which we:
- Update the injected Repository with the new Slot;
- Produce a new `CreatedEvent` event with information on the new Slot;
- Emit the event;
- And finally, run the Promises.

That last entire section is where we actually enforce the transactional boundary and hand off to other's to do whatever they might need the event for.

## Use case #2: Check in

The rest of the use cases have a format that resembles the one we look at here, the “check in” case.

```
/**
 * @description Check in to a slot.
 */
public async checkIn(slotDto: SlotDTO): Promise<void> {
    const slot = new Slot().fromDto(slotDto);
    const { event, newStatus } = slot.checkIn();

    const checkInEvent = new CheckedInEvent({
        event,
        metadataConfig: this.metadataConfig
    });

    await this.transact(slot.toDto(), checkInEvent, newStatus);
}
```

We load a Slot based on the ID we have received.

Inside of the Slot Entity, we will destructure some fields, verify that we have the correct slot status (it must be `RESERVED` to work), and then call our private `updateSlot()` method

with the slot data and new status. When that's done it's time to make the correct event (here, the `CheckedInEvent`) and emit that with our private `emitEvents()` method.

All in all, we have ensured the state satisfies our business needs, the new invariant is correctly shaped, made the update, and informed our domain of the change via an event.

### Use case #3: Reserve slot

Reserving a Slot is similar to the above case, but we need to do more this time, including actually getting a verification code for the reservation from a different service in an altogether different solution.

```
/**
 * @description Reserve a slot.
 */
public async reserve(
  slotDto: SlotDTO,
  hostName: string,
  verificationCodeService: VerificationCodeService
): Promise<ReserveOutput> {
  const slot = new Slot().fromDto(slotDto);
  const { event, newStatus } = slot.reserve(hostName);

  const verificationCode = await verificationCodeService.↵
    ↵ getVerificationCode(slotDto.slotId);

  const reserveEvent = new ReservedEvent({
    event,
    metadataConfig: this.metadataConfig
  });

  await this.transact(slot.toDto(), reserveEvent, newStatus);

  return {
    code: verificationCode
  };
}
```

Because this one has to take in a user's input data it becomes very important that we validate the input and sanitize it. That becomes the first thing we do when the DTO is

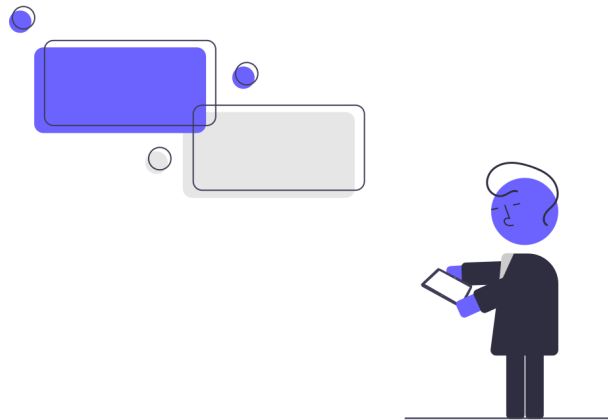
constructed in the Entity.

Next, in the Entity (not seen here) we load the slot data for the requested slot, destructure the data for use, and verify that the slot status is correct or else we throw an error. Then we get a verification code using a private method that will get it from an external service in another (sub)domain. If something goes awry, we throw an error.

Now it's just the home stretch: Update the slot with the correct shape and data, build a `ReservedEvent` and emit it to our domain. Finally, return the `ReserveOutput` object with the verification code we received so that the user can jot it down and use it when the time comes to check-in.

## Value Objects

*Value Objects help us define semi-complex, identity-less objects without us needing to resort to spaghetti code.*

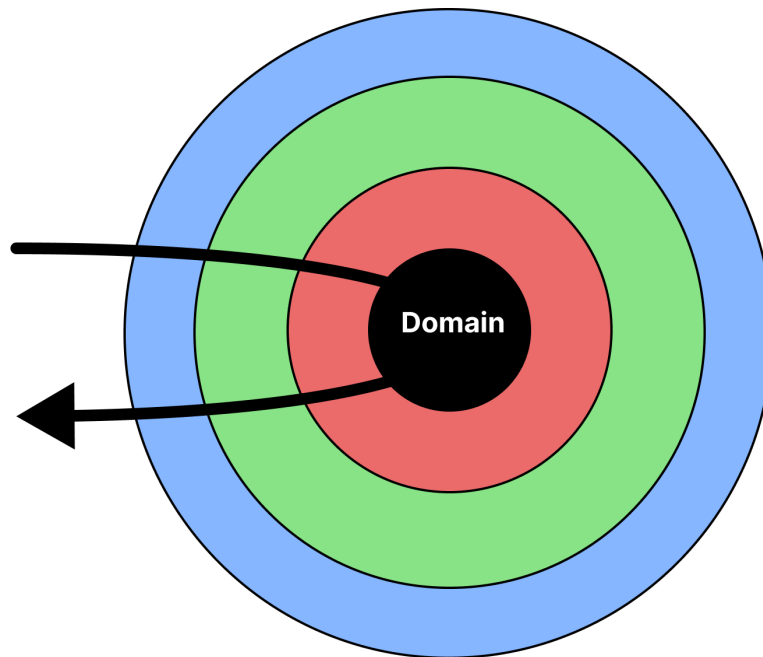


*Figure 1.54: Illustration from Undraw*

### Success

**TL;DR: Value Objects** are like non-unique Entities. You use them in much the same way, except they bear no own identity. An instance of a **Value Object** is equivalent to another instance if they have the same properties and values. They are excellent for containing complex creational logic and work well when combined on Entities that contain **Value Objects** as part of their data.





*Figure 1.55: Value Objects reside in the Domain layer.*

Value objects are a Godsend.

**Value objects are defined by attributes, not by identity.** This makes them great for cases where you want to provide a “vending machine” for non-trivial objects, such as in our case, a `TimeSlot`. The `TimeSlot` itself has no identity, but it does have unique values in non-unique keys/attributes. Because this type of object needs to **always be correctly constructed**, we can delegate the responsibility into (for example) a class that creates such `TimeSlots`. You don’t pass around Value Objects that much, nor update them. Instead, you instantiate new ones—they are 100% replaceable and interchangeable, after all!

This pattern is effective in refactoring, such as when wanting to cut down on [primitive obsession](#).

### **Danger**

Producing non-entity objects might invite one to use “easy” and regressive patterns fished out of the recesses of one’s memory bank. “These aren’t important!” Wrong.

### Information

For more, read <https://medium.com/swlh/value-objects-to-the-rescue-28c563ad97c6>

## Creating a TimeSlot as a Value Object

If there is something I know I need to build more often, it's Value Objects.

Get-A-Room doesn't have very many Value Objects (two, in fact). Let's look at the TimeSlot. This is how it's used at `code/Reservation/SlotReservation/src/domain/aggregates/Slot.ts`:

```
const timeSlot = new TimeSlot();
const currentTime = this.getCurrentTime();

for (let slotCount = 0; slotCount < numberHours; slotCount++) {
  const hour = startHour + slotCount;
  timeSlot.startingAt(hour);
  const { startTime, endTime } = timeSlot.get();
  const newSlot = this.makeSlot({ currentTime, startTime, endTime });
  slots.push(newSlot);
}
```

And the Value Object itself (`code/Reservation/SlotReservation/src/domain/valueObjects/TimeSlot.ts`):

```
import { TimeSlotDTO } from "../../interfaces/TimeSlot";

import { InvalidHourCountError } from "../../application/errors/InvalidHourCountError";

/**
 * @description Handles the creation of valid time objects.
 */
export class TimeSlot {
  private startTime = "";
  private endTime = "";
```

```

/**
 * @description Creates a valid time object. Requires an `hour` ↵
 * ↵ provided as
 * a number as input for the starting hour. Assumes 24 hour clock.
 *
 * All time slots are 1 hour long and provided as ISO strings.
 * @example ```
 * const timeSlot = new TimeSlot();
 * timeSlot.startingAt(8);
 * ```
 */
public startingAt(hour: number): void {
  if (hour > 24) throw new InvalidHourCountError();
  if (hour <= 0) hour = 0;

  const startHour = hour.toString().length === 1 ? `0${hour}` : `${↵
  ↵ hour}`;
  const endHour =
    (hour + 1).toString().length === 1 ? `0${hour + 1}` : `${hour + ↵
  ↵ 1}`;
  const day = new Date(Date.now()).toISOString().substring(0, 10);
  const startTime = new Date(`${day}T${startHour}:00:00`).toISOString(↵
  ↵ ());
  const endTime = new Date(`${day}T${endHour}:00:00`).toISOString();

  this.startTime = startTime;
  this.endTime = endTime;
}

/**
 * @description Returns a `TimeSlotDTO` for the start and end time.
 */
public get(): TimeSlotDTO {
  return {
    startTime: this.startTime,
    endTime: this.endTime,
  };
}
}

```

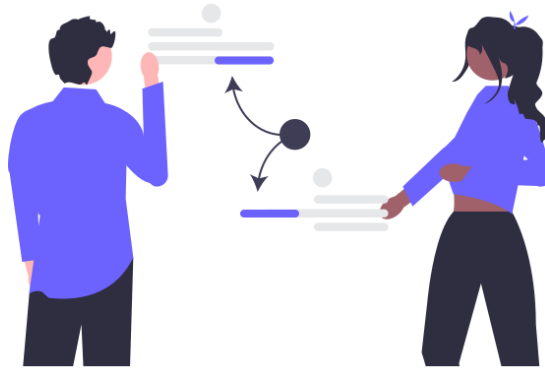
To save on memory we are reusing the same `TimeSlot` instance and calling it several times throughout the loop. This is probably not the right way to do it in certain circumstances, but here I feel it makes sense as we are never relying on the instance of the Value Object

itself, just asking it to return a Data Transfer Object based on the input data. Perhaps this can be seen as acceptable in the limited range of uses that we get to use `TimeSlot` for.

On the plus side, we are neatly encapsulating a lot of tedious detail out of the actual usage contexts. This also ensures that validation is done and that the integrity is correct and can be trusted; You'll see the error handling if we receive an hour count over 24, and how we are resetting any zero values to an acceptable base.

It should be clear that Value Objects can be as simple or complex as possible. Use them whenever you feel that unique data types or values need to be addressed in a controlled manner.

## Domain Events



*Figure 1.56: Illustration from Undraw*

### Success

**TL;DR:** When we say **Events** we in practice always refer to [Domain Events](#). They are a way of decoupling parts of your solution or landscape while expressing what is going on in the terms of ubiquitous language. **Domain Events** can be used with any technology and DDD has no opinion on that side of things.

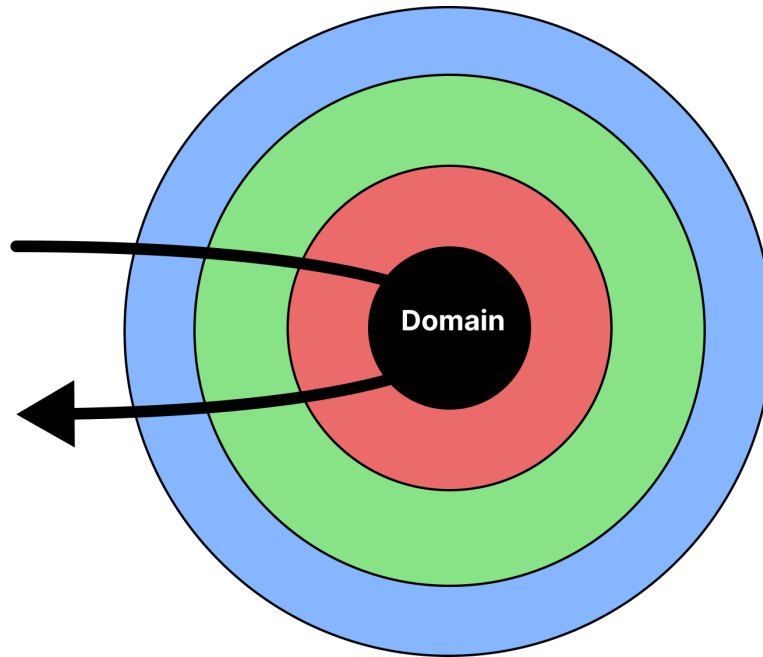


Figure 1.57: (Domain) Events reside in the Domain layer.

*Domain Events* indicate **significant events that have occurred in the domain** and need to be reported to other stakeholders belonging to the domain. Aggregates are responsible for publishing events, though we saw how in our example project it is the Domain Service wrapping the Entity that actually does that work for reasons mentioned in that section. Domain Events drive transactions and can make commands to other systems.

At a high level, events and event-driven architecture mean that we can—and should—decouple systems from each other. This enables us to practically build and sustain an intentional architecture, as promoted by Domain Driven Design, Clean Architecture, and most serious software engineering principles today.

See the diagrams below from [Microsoft](#) for visual clarification:

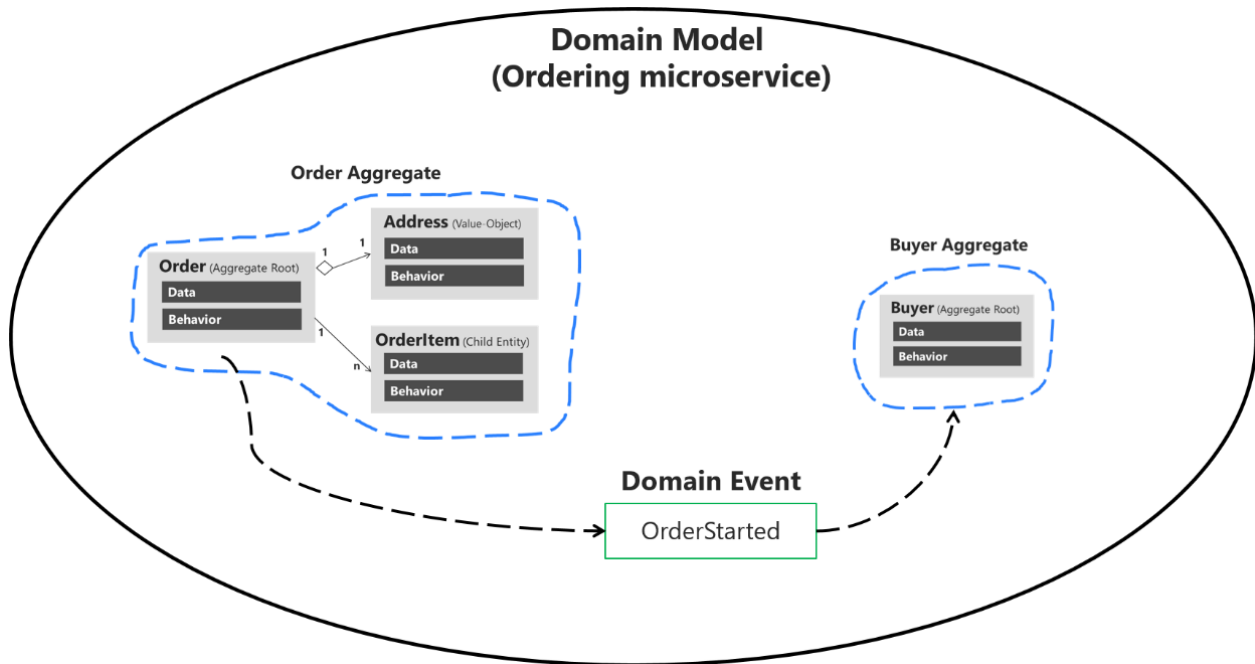


Figure 1.58: Domain events to enforce consistency between multiple Aggregates within the same domain.

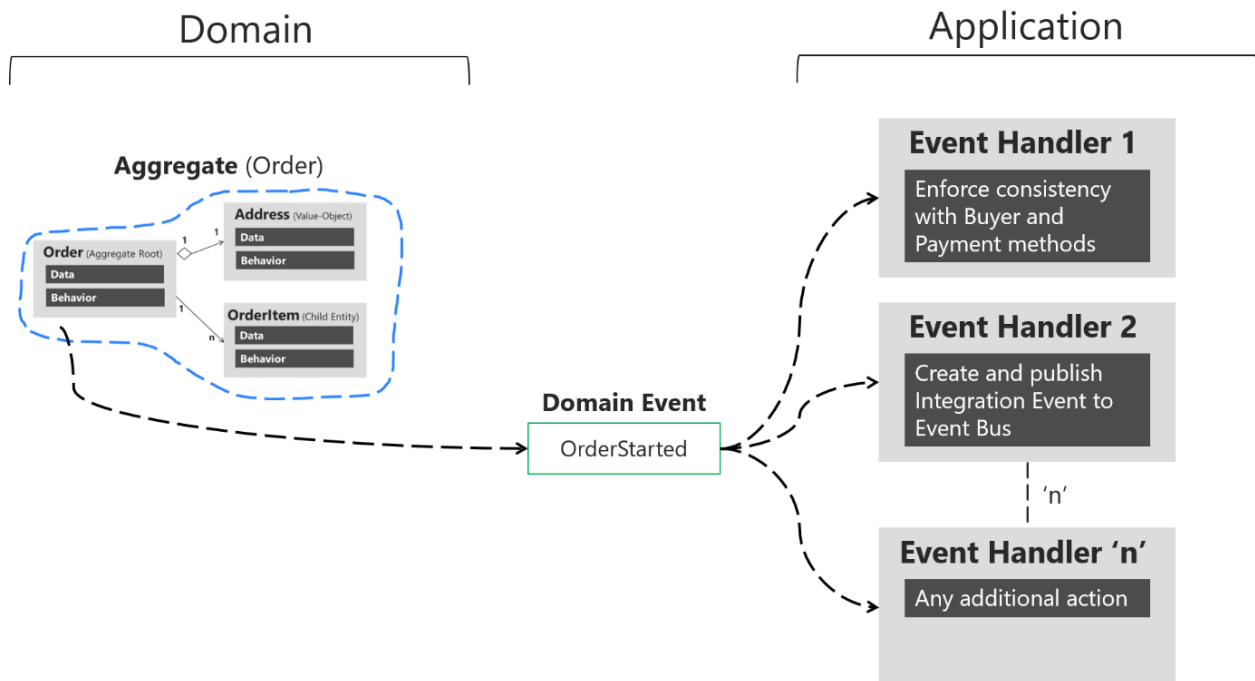


Figure 1.59: Handling multiple actions per domain.

As seen in the diagrams, a typical domain event could be OrderStarted if we are in a com-

mercial domain. This event would be sent to our domain's event bus which all systems in the scope of our domain may subscribe to.

### Information

See more at:

- [Best Practice - An Introduction To Domain-Driven Design](#)
- [Domain Driven Design](#)
- [Domain Driven Design \(Wikipedia\)](#)
- [Domain Driven Design and Development In Practice](#)
- [The Clean Architecture](#)
- [The Clean Architecture - Beginner's Guide](#)

## Naming, exactness and uniqueness of an event

Domain events should translate into clearly named and partitioned and non-overlapping names. Names are, as implied, domain-based and must use nomenclature and language that people understand in the particular domain. Key goals for us include:

- Removing **semantic ambiguity** (not understanding what something refers to)
- Removing **terminological contention** (many contexts claiming the same terms)
- Increasing and enforcing **domain language** (using the same terms that our domain stakeholders use and express)

Domain nomenclature is ultimately *only valid and meaningful within the domain*. Therefore, as a logical consequence, we should not spend time synchronizing nomenclature *across* domains.

**Bad name examples:**



- OrderUpdated
- ErrorOccurred

### Why?

Too broad term; very inspecific; easy to see that others may make claims to the same name; unclear what was actually done. “Order” may not be technically incorrect, but it’s also a term that might be highly contested or have other meanings when traveling across domains. A generic “Error” is not helpful.

### Good name examples:

- SalesOrderDeliveryFieldChanged
- ManufacturingOrderDispatched

### Why?

Very clear demarcation on this being a “sales order” (not a *broad inspecific* “order”); also communicates what exactly was changed.

SalesOrder would be a better example than OrderUpdated also because (we can assume in this fictional case) our system (or Aggregate) controls and enforces this particular type of order.

Note that such work around naming is often more art than science.

## Persisting events

It’s wise to store a history of all events that have occurred. This makes it possible to “play back” the history of a system—well, Aggregate to be exact—and is a foundational component of the [CQRS \(Command Query Responsibility Segregation\) pattern](#).

Personally, I find full-on CQRS to be *a lot* to deal with, and modern cloud architectures can mitigate and improve some of the conditions from which original CQRS evolved. I would however highly advise you to:

- Use [CQS \(Command Query Separation\)](#) when naming. This forms the philosophical underpinning of CQRS itself, meaning that you create a very crisp and elegant nomenclature around events themselves. CQS “weighs” nothing and everyone wins.
- Use an event store to persist all events when you emit Domain Events. This can be as [simple as in our example codebase](#), or become full-blown [event sourcing](#).

The solution used here is manual and is done completely in code, on behalf of the Domain Service (that stands in for the Aggregate orchestration), doing this type of transactional dance (in the case of the Reservation solution)

- Update the Slot table with the updated item
- Update the Slot table with the event
- Emit the Domain Event

### Information

AWS natives will maybe point to a more elegant solution using [DynamoDB streams as an outbox pattern](#), which could definitely work. I am 50/50 which I like the most because doing so would mean you still have to implement some mechanism like a Lambda that can “translate” the DynamoDB table item changes into actual Domain Events.

## Resiliency

The code base uses a trivial handwavy way to set up a Dead Letter Queue (often just abbreviated as DLQ). A full implementation would for example implement a Lambda function that just re-emits the event on the appropriate bus. This could theoretically become mined territory since we want to keep a tight ship regarding who can emit what event to which bus. In this case, we can only use a single Lambda to do that work and it must not contain any business functionality—only re-emit the exact same event!

Further, note that there are differences for DLQs based on which service you are setting them up for, i.e. a Lambda DLQ will be useful when a Lambda function does not respond, but you will still need a separate EventBridge DLQ to guard against failures when a system tries to put something on the EventBridge bus.

This is left to you as an optional exercise should you want to do this.

### Information

See the following for more information:

- [Building resilient serverless patterns by combining messaging services | AWS Compute Blog](#)
- [Improved failure recovery for Amazon EventBridge](#)
- [Amazon EventBridge - Using dead-letter queues \(DLQs\)](#)
- [AWS SQS Docs](#)
- [Designing durable serverless apps with DLQs for Amazon SNS, Amazon SQS, AWS Lambda](#)

## Emitting events

At this point, it should be relatively clear that Domain Events are important and that they should be named well and persisted. But what are they?

In the example project, we use AWS EventBridge, which similarly to other such services, takes in an object as the event payload. There's more to it, but more or less you'll get to stick in whatever object you want (with some size restrictions, etc.).

To work with Domain Events in a controlled manner we'll however need more than just an odd JSON blob.

Our project uses:

- The `SlotCommand` output from the Slot Entity, which dictates the majority of actual content coming from changes.
- `EventEmitter` abstraction that does the infrastructural work. This has both a “local/mock” and an `EventBridge` implementation.
- A `DomainEventPublisher` Application Service that wraps the event emitter (which will emit two events—one for actual use and one for the analytics service—and log out the event).
- An `EmittableEvent` abstraction class that handles all the logic of producing the right shape and metadata and other such laborious things.
- A range of Events (one for each Domain Event) that extends the `EmittableEvent`.

### Success

A Domain Event is therefore always constructed from a `SlotCommand`. The `DomainEventPublisher` is the Application Service that is injected into `ReservationService`.

## The event emitter

This is code/Reservation/SlotReservation/src/infrastructure/emitters/EventBridgeEmitter↵  
↵ .ts.

```
import {
  EventBridgeClient,
  PutEventsCommand,
} from "@aws-sdk/client-eventbridge";

import { EventBridgeEvent } from "../../interfaces/Event";
import { EventEmitter } from "../../interfaces/EventEmitter";

import { MissingEnvVarsError } from "../../application/errors/↵
↵ MissingEnvVarsError";

/**
```

```

    * @description Factory function to return freshly minted EventBridge ↵
    ↵ instance.
    */
    export const createEventBridgeEmitter = (region: string) => {
        if (!region)
            throw new MissingEnvVarsError(
                JSON.stringify([ { key: "REGION", value: region } ])
            );

        return new EventBridgeEmitter(region);
    };

    /**
    * @description An EventBridge implementation of the `EventEmitter`.
    */
    class EventBridgeEmitter implements EventEmitter {
        private readonly eventBridge: EventBridgeClient;

        constructor(region: string) {
            this.eventBridge = new EventBridgeClient({ region });
        }

        /**
        * @description Utility to emit events with the AWS EventBridge ↵
        ↵ library.
        *
        * @see https://docs.aws.amazon.com/eventbridge/latest/APIReference/↵
        ↵ API_PutEvents.html
        * @see https://www.npmjs.com/package/@aws-sdk/client-eventbridge
        */
        public async emit(event: EventBridgeEvent): Promise<void> {
            const command = new PutEventsCommand({ Entries: [event] });
            if (process.env.NODE_ENV !== "test") await this.eventBridge.send(↵
            ↵ command);
        }
    }
}

```

We see that there is a basic Factory there, and then the EventBridgeEmitter just implements the overall EventEmitter which is just a simple interface so we can create other emitter infrastructure in the future. We want to separate the emitters primarily for testing (and local development) reasons so that we can use a local mock rather than the full-blown EventBridge client.

## Domain event publisher service

Now for code/Reservation/Reservation/src/application/services/DomainEventPublisherService↵

↵ .ts:

```
import { MikroLog } from "mikrolog";

import { Event } from "../../interfaces/Event";
import {
  DomainEventPublisherDependencies,
  DomainEventPublisherService,
} from "../../interfaces/DomainEventPublisherService";
import { EventEmitter } from "../../interfaces/EventEmitter";

import { MissingDependenciesError } from "../errors/↵
↵ MissingDependenciesError";
import { MissingEnvVarsError } from "../errors/MissingEnvVarsError";

/**
 * @description Factory function to set up the ↵
↵ DomainEventPublisherService`.
 */
export function createDomainEventPublisherService(
  dependencies: DomainEventPublisherDependencies
) {
  return new ConcreteDomainEventPublisherService(dependencies);
}

/**
 * @description Service to publish domain events.
 */
class ConcreteDomainEventPublisherService
  implements DomainEventPublisherService
{
  private readonly eventEmitter: EventEmitter;
  private readonly analyticsBusName: string;
  private readonly domainBusName: string;
  private readonly logger: MikroLog;

  constructor(dependencies: DomainEventPublisherDependencies) {
    if (!dependencies.eventEmitter) throw new MissingDependenciesError↵
↵ ();
    const { eventEmitter } = dependencies;
```

```

this.eventEmitter = eventEmitter;
this.logger = MikroLog.start();

this.analyticsBusName = process.env.ANALYTICS_BUS_NAME || "";
this.domainBusName = process.env.DOMAIN_BUS_NAME || "";

if (!this.analyticsBusName || !this.domainBusName)
    throw new MissingEnvVarsError(
        JSON.stringify([
            { key: "DOMAIN_BUS_NAME", value: process.env.DOMAIN_BUS_NAME ↵
↵ },
            { key: "ANALYTICS_BUS_NAME", value: process.env.↵
↵ ANALYTICS_BUS_NAME },
        ])
    );
}

/**
 * @description Convenience method to emit an event
 * to the domain bus and to the analytics bus.
 */
public async publish(event: Event): Promise<void> {
    const source = event.get().Source;

    await this.eventEmitter.emit(event.get());
    this.logger.log(`Emitted '${source}' to '${this.domainBusName}'`);

    await this.eventEmitter.emit(
        event.getAnalyticsVariant(this.analyticsBusName)
    );
    this.logger.log(`Emitted '${source}' to '${this.analyticsBusName}'` ↵
↵ );
}
}

```

As written previously, this one adds a layer of extra spice with the multiple emitted events and logging. Other than that it's not much else under the hood. At least it makes it much easier and one step more removed from the real infrastructure.

## The events

The `EmittableEvent` Value Object might look long and daunting, but it's actually very simple. The situation we have to deal with is that the event shape is rather deep meaning it does take some energy to construct it.

This is `code/Reservation/SlotReservation/src/domain/valueObjects/Event.ts`:

```
import { randomUUID } from "crypto";

import {
  EventInput,
  EventDetail,
  EventBridgeEvent,
  EventDTO,
  MakeEventInput,
  MetadataInput,
} from "../../interfaces/Event";
import { Metadata, MetadataConfigInput } from "../../interfaces/↵
  ↵ Metadata";

import { getCorrelationId } from "../../infrastructure/utils/↵
  ↵ userMetadata";

import { MissingMetadataFieldsError } from "../../application/errors/↵
  ↵ MissingMetadataFieldsError";
import { NoMatchInEventCatalogError } from "../../application/errors/↵
  ↵ NoMatchInEventCatalogError";
import { MissingEnvVarsError } from "../../application/errors/↵
  ↵ MissingEnvVarsError";

/**
 * @description Vend a "Event Carried State Transfer" type event with ↵
 * ↵ state
 * that can be emitted with an emitter implementation.
 */
abstract class EmittableEvent {
  private readonly event: EventBridgeEvent;
  private readonly eventName: string;
  private readonly metadataConfig: MetadataConfigInput;

  constructor(eventInput: EventInput) {
    const { event, metadataConfig } = eventInput;
    this.eventBusName = process.env.DOMAIN_BUS_NAME || "";
  }
}
```



```

    this.metadataConfig = metadataConfig;

    if (!this.eventBusName)
        throw new MissingEnvVarsError(
            JSON.stringify([
                { key: "DOMAIN_BUS_NAME", value: process.env.DOMAIN_BUS_NAME ↵
↵ },
            ])
        );

    const eventDTO = this.toDto(event);
    this.event = this.make(eventDTO);
}

/**
 * @description Make an intermediate Data Transfer Object that
 * contains all required information to vend out a full event.
 */
private toDto(eventInput: MakeEventInput): EventDTO {
    const { eventName, slotId, slotStatus } = eventInput;

    const detailType = this.matchDetailType(eventName);
    const timeNow = Date.now();

    return {
        eventBusName: this.eventBusName,
        eventName,
        detailType,
        metadata: {
            ...this.metadataConfig,
            version: eventInput.version || 1,
            id: randomUUID().toString(),
            correlationId: getCorrelationId(),
            timestamp: new Date(timeNow).toISOString(),
            timestampEpoch: `${timeNow}`,
        },
        data: {
            event: eventName,
            slotId,
            slotStatus,
            hostName: eventInput.hostName || "",
            startTime: eventInput.startTime || "",
        },
    };
}

```

```

/**
 * @description Produces a fully formed event that can be used with ↵
 * ↵ AWS EventBridge.
 */
private make(eventDto: EventDTO): EventBridgeEvent {
  const { eventBusName, data, metadata, detailType } = eventDto;
  const { version, id, correlationId } = metadata;
  const source = `${metadata.domain?.toLowerCase()}.${metadata.system↵
  ↵ ?.toLowerCase()}.${detailType.toLowerCase()}`;

  const detail: EventDetail = {
    metadata: this.produceMetadata({ version, id, correlationId }),
    data,
  };

  return {
    EventBusName: eventBusName,
    Source: source,
    DetailType: detailType,
    Detail: JSON.stringify(detail),
  };
}

/**
 * @description Produce correct metadata format for the event.
 * @note The verbose format is used as we cannot make assumptions
 * on users actually passing in fully formed data.
 */
private produceMetadata(metadataInput: MetadataInput): Metadata {
  const { version, id, correlationId } = metadataInput;

  if (
    !version ||
    !this.metadataConfig.lifecycleStage ||
    !this.metadataConfig.domain ||
    !this.metadataConfig.system ||
    !this.metadataConfig.service ||
    !this.metadataConfig.team
  )
    throw new MissingMetadataFieldsError(metadataInput);

  const timeNow = Date.now();

  return {

```

```

        timestamp: new Date(timeNow).toISOString(),
        timestampEpoch: `${timeNow}`,
        id,
        correlationId,
        version,
        lifecycleStage: this.metadataConfig.lifecycleStage,
        domain: this.metadataConfig.domain,
        system: this.metadataConfig.system,
        service: this.metadataConfig.service,
        team: this.metadataConfig.team,
        hostPlatform: this.metadataConfig.hostPlatform,
        owner: this.metadataConfig.owner,
        region: this.metadataConfig.region,
        jurisdiction: this.metadataConfig.jurisdiction,
        tags: this.metadataConfig.tags,
        dataSensitivity: this.metadataConfig.dataSensitivity,
    };
}

/**
 * @description Pick out matching `detail-type` field from event ↔
 * ↔ names.
 * @note Should be refactored to regex solution if this grows.
 */
private matchDetailType(eventName: string) {
    switch (eventName) {
        // User interaction events
        case "CREATED":
            return "Created";
        case "CANCELLED":
            return "Cancelled";
        case "RESERVED":
            return "Reserved";
        case "CHECKED_IN":
            return "CheckedIn";
        case "CHECKED_OUT":
            return "CheckedOut";
        case "UNATTENDED":
            return "Unattended";
        // System interaction events
        case "OPENED":
            return "Opened";
        case "CLOSED":
            return "Closed";
    }
}

```

```

    throw new NoMatchInEventCatalogError(eventName);
}

/**
 * @description Get event payload.
 */
public get() {
    return this.event;
}

/**
 * @description Return modified DTO variant for analytics purposes.
 * Use "Notification" type event without state.
 */
public getAnalyticsVariant(analyticsBusName: string): ↵
    ↵ EventBridgeEvent {
    const analyticsEvent: EventBridgeEvent = JSON.parse(
        JSON.stringify(this.get())
    );
    const detail = JSON.parse(analyticsEvent.Detail);

    analyticsEvent["EventBusName"] = analyticsBusName;
    detail["metadata"]["id"] = randomUUID().toString();
    if (detail.data?.slotStatus) delete detail["data"]["slotStatus"];

    analyticsEvent["Detail"] = JSON.stringify(detail);

    return analyticsEvent;
}
}

/**
 * @description An event that represents the `Created` invariant state.
 */
export class CreatedEvent extends EmittableEvent {
    //
}

/**
 * @description An event that represents the `Cancelled` invariant ↵
    ↵ state.
 */
export class CancelledEvent extends EmittableEvent {
    //
}

```

```

}

/**
 * @description An event that represents the `Reserved` invariant state↵
 * ↵ .
 */
export class ReservedEvent extends EmittableEvent {
  //
}

/**
 * @description An event that represents the `CheckedIn` invariant ↵
 * ↵ state.
 */
export class CheckedInEvent extends EmittableEvent {
  //
}

/**
 * @description An event that represents the `CheckedOut` invariant ↵
 * ↵ state.
 */
export class CheckedOutEvent extends EmittableEvent {
  //
}

/**
 * @description An event that represents the `Unattended` invariant ↵
 * ↵ state.
 */
export class UnattendedEvent extends EmittableEvent {
  //
}

/**
 * @description An event that represents the `Open` invariant state.
 */
export class OpenedEvent extends EmittableEvent {
  //
}

/**
 * @description An event that represents the `Closed` invariant state.
 */
export class ClosedEvent extends EmittableEvent {

```

```
//  
}
```

### Warning

Admittedly the event structure (despite our decoupling of the emitter itself) is tied to EventBridge which is acceptable as we are actually only using EventBridge in our project. If we would support truly different emitters we would perhaps need to add further abstractions on the event shape. In the context of this project, we can accept that as a trivia item.

## Metadata

The `produceMetadata` method does what it says on the box. It's not that complicated but allows us the possibility to vend a metadata object that is always as expected.

## Matching the detail type

Very basic, dumb implementation to match the event name to a recased version.

## DTO

First, we make the `EventDTO`. This has the overall shape we actually require.

## Make method

The `make()` method takes our event DTO and forms it into the `EventBridgeEvent` that can actually be put on our event bus.

## Get method

In order to use the class (remember, data *and* behavior!) rather than a dumb plain object, we'll allow a method to access the current representation.

## Get analytics method

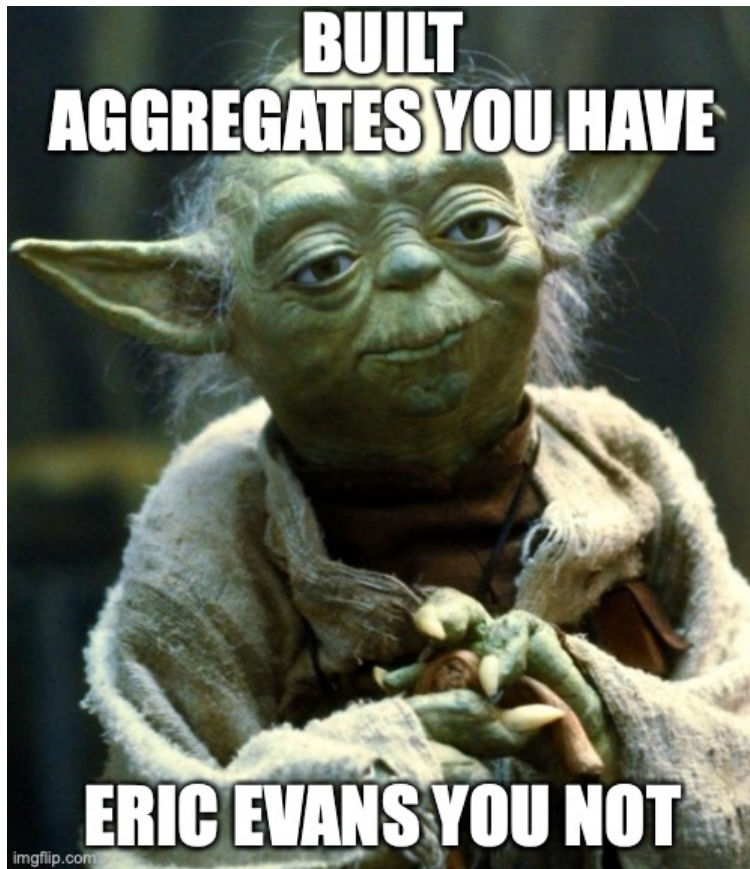
Just as the regular `get()` method, the `getAnalyticsVariant()` method returns a representation of the event. The reasons we want to have this as a specific method are:

- The analytics event bus is not the same as the regular one
- We want to redact the (potentially sensitive) ID
- The analytics context does not need the slot status

## Extended classes

There is nothing unique concerning the classes that we should use, so we can contain the “base” class and make trivial extensions to allow use for the derived classes instead.

## How to go further



*Figure 1.60: I had to make this and now there is no turning back.*

You made it this far, good for you! I'll take the opportunity to describe a number of things you can do to learn more.

If you are anything like me, or most people I suppose, then it's going to take more than one activity and one shot at a project to make it have all the bells and whistles. My proposal is to continue doing hands-on work based on what you've seen. It should be able to sustain you for some time. Then, as you have something of your own in your backpack, you can start to truly take in the full extent of DDD and what has been said and thought.



## **Practice the craft**

### **Run the demo code locally**

I wouldn't be surprised if you "only" read the book. Not an issue!

But do take the time to actually run the code locally and muck about with it to see things change and evolve as you interact with the source code. You might be inexperienced with TypeScript or JavaScript, or any of the other many pieces that make it run, but there just is no way to better understand tactical DDD. Since you already have the heavy lifting done for you, go ahead and feel like a princess and play with it (before burning it down).

### **Add support for multiple rooms, sites, time zones**

Using the reference project, you could start adapting for new requirements, like being able to have multiple rooms and/or sites and/or time zones. Or anything else for that matter.

Identify what changes would be needed, if you need new Aggregates (or an Aggregate Root) or Entities etc. **It should be possible to evolve such functionality without significant rework.**

### **Build the application from scratch**

You could certainly take the starting requirements and build the application to the best of your abilities in your language/runtime and other technologies of choosing.

### **Build your own application**

Why not just go all the way and do what I did (minus writing a book) and build an application based on your own requirements.

Make sure that you keep level-headed and objective and that you try to keep the phases clean — first the high level scenario to guide the project, then conducting the strategic DDD, and only after that proceeding to the tactical implementation work. You may of

course iterate the cycles, but you'll have to role play a "business owner" type character while starting, unless you want it to possibly derail into being any old pet project.

### Success

A spin on this is to actually do a real (low-key) project, maybe for some friends or family, or some affiliation or group you are part of. Looking forward to reading about the *Domain-Driven Bowling Club* or *Domain-Driven Church of Satan (Local Chapter)* in the future!

## Learn to communicate better

### Learn diagramming

Diagramming is a core skill of senior engineers and most architects. They allow, when they are good, an easy-to-digest and precise format. As an author, I find that it's often more accurate than text and faster to finish than writing, too.

The sad truth is that many diagrams that I've seen are miserable because of some combination of (for example):

- Mixed views in the same diagram
- Confusing, overall
- Inconsistent style
- Aesthetically displeasing

While these are just some of the many parameters that could go wrong, **the most pressing** problem is that many times you are happy to have a diagram at all, in the first place\*\*.

So: Diagram everything and make others do it too.

If you've been around the block, you might wonder "Should I just learn UML?". It's a rich and formal way of diagramming, but to be frank, few are orthodox about it. Most

will learn the basics and then stay happy with what they took to heart, ergo, not all of it. But if you *do* want to go full UML, there's the GitHub repo for [Learn UML2.\\* in simple terms](#). On the book side, the classic book on the subject is Martin Fowler's [UML Distilled](#). An outstanding newer book, not beholden to any specific technique, is Jacqui Read's [Communication Patterns: A Guide for Developers and Architects](#) which I *strongly* recommend.

### Information

Lucidchart is a commercial tool that you may or may not want to use. They do however offer a [wide range of tutorials for how to work with many types of diagrams](#). It's a good start.

For models and tools/frameworks beyond UML, consider:

- The [C4 model](#), “Context, Containers, Components, and Code”
- [ContextMapper](#), “A Modeling Framework for Strategic Domain-driven Design”

For software tools, you might want to consider:

- [Excalidraw](#)
- [Diagrams.net](#) (previously Draw.io)
- [WebSequenceDiagrams](#), online sequence diagrams
- [Lucidchart](#)
- [Whimsical](#)
- [Figma](#), which has the [FigJam](#) online whiteboard feature; [more on diagramming here](#)
- [Miro](#)

### Information

My personal picks are **Excalidraw** for online collaborative whiteboarding and **Diagrams.net** for professional work that I can predominantly work on solo and then share as a file asset. Diagrams.net works perfectly fine for both UML-style diagrams and your typical free-form diagrams. No sane person suffers through manually making sequence diagrams in a typical drawing tool; use **WebSequence-Diagrams** instead.

## Write technical documentation

Write. Either for an own project, something at work, an open source project, or whatever really. Just write. We all learn it in school, but—and I know I am hard here—an engineer or architect who cannot write or communicate efficiently is not worth their salt. Sad to say, but that quality is not something I experience a lot...

Different languages have different standards or mannerisms. For our project, we used [JSDoc](#) with a light stylistic mannerism to not re-document anything that's obvious in the TypeScript types. For example, in the case of .NET you might want to look at [DocFX](#).

One of my favorite, and quite extensive, articles on technical writing is InnoQ's [Principles of technical documentation](#).

Obviously technical writing is vital for tech companies. Google has [published several courses](#) you can take, as has GitLab, [Technical Writing Fundamentals](#).

For an interesting approach to technical writing, you should check out [Diátaxis](#).

### Success

It would be strange to not be able to identify “good docs when you see them”. For some examples of good writing, see:

- [Mozilla Developer Network Web Docs](#)
- [Cloudflare Docs](#)

- [Kubernetes Documentation](#)
- [Google Cloud documentation](#)

### **Push for a “ubiquitous language” in an existing project**

As long as we have a business/tech split in a project or organization, whatever side you are on will always be easier to work with—if you are a techie, you could start using the tactical patterns right now. But what good will it make, beyond maybe just being a more disciplined refactoring of whatever code you already have?

That’s why I think you should spend time building the circumstances to enable effective collaboration between the most important groups, so that you can actually create the core semantic artifacts and have these in a documented format.

Starting where you are *now* should make this less scary, and you probably already have an idea of what’s going on around you.

### **Try out EventStorming with colleagues or friends**

With the ubiquitous language defined, perhaps it’s time to scale up and go for the full monty? In that case EventStorming could be an option. The ideas are easy enough, and as always, good facilitation might be the biggest requirement that’s not listed upfront.

Either ask someone who has done this before, or accept that we all learn by doing and **just do it**. The book is cheap, you can always [learn-by-video](#), and there are plenty of good articles describing how to run a workshop.

## Design more

### Design and build solutions

As stated in the beginning of this book, I really see no other way to be good at designing software than, well, by designing software.

While it's easy to get lost in all the specific services popping up in the vast arsenals of AWS and their ilk, you should abstract from individual services to broader *capabilities*. To be better prepared doing so—don't get surprised now—I have a book recommendation. One book that I find has been very good at tying together the last 10-15 years of development practices and modern technologies into one concise overview is [Continuous Architecture in Practice: Software Architecture in the Age of Agility and Devops](#) (Erder, Woods, Pureur 2021). Don't expect to go deep-diving here, but as inspiration to see the big picture that book might do the trick.

Designing is more about process and thought than about a given artifact to represent it. If you want to design as part of “exercise” then consider learning or using tools listed in the Learn diagramming advice.

And of course: Don't forget to build the things you design. Nothing is quite as powerful in conveying the learnings as feeling the pain yourself.

### Present your solutions and decisions

You might want to take the chance to also brush up on actually presenting the work:

- Learn to write [Architecture Decision Records](#), a format that is becoming very popular to describe decisions.
- Same with [Requests for Comments, or RFCs](#), a “new old” way of documenting work *to be* done.
- Get good at [Keynote/Powerpoint/Google Slides etc.](#), because you *will* use this at some point to present your work.

Combine this with the diagramming work as needed.

## Know your sh\*t

### Read up on secondary-source literature

There's excellent and more condensed material available these days than in the days of yore. It doesn't replace the original texts as much as it offers a more accessible path for many readers.

Perhaps this "DDD in 2 hours" type of thing started with Vaughn Vernon's [Domain-Driven Design Distilled](#) (2016). For a not-quite-so-thin book, I recommend Vladik Khononov's [Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy](#) (2021). It's easier to get into (and through!) than the blue and red books and it's very good in its own right. Like this book, it has to skip the deeper stuff, but that's precisely why I think you should start with Khononov's book.

While you wait for the book to be delivered, you can get [Domain Driven Design Quickly](#) by Abel Avram and Floyd Marinescu for free, in the form of a PDF.

There's also a free PDF of [Domain-Driven Design Reference: Definitions and Pattern Summaries](#) by Eric Evans for those reference-type needs you might have.

A stellar resource, especially if you are in the .NET world, is Microsoft's online book [.NET Microservices Architecture for Containerized .NET Applications](#). The link takes you to the (big) section dedicated to DDD. I'm happy to see Microsoft be so clear with how DDD supports modern software development.

### Go to the primary sources

This might feel really odd—having these distinguished sources at the very back of list, like this. Almost sacrilege? No, I don't actually think so. The primary sources are very, very good in my opinion but they require a bit of investment in time.

These are unmistakably "the real deal" and like we saw in an early page, DDD as a concept does not always reflect the true intentions of it. When the day comes for you to go to the root, there is no better way to get things straight than reading Eric Evans' [Domain-Driven Design: Tackling Complexity in the Heart of Software](#) and Vaughn Vernon's [Implementing Domain-Driven Design](#).

## References and resources

Now for a non-exhaustive dump of good stuff you might want to read about if you made it all the way here.



*Figure 1.61: References*

Hopefully you've already found many interesting resources by reading and following along. In this section I will add further resources on the overall subjects we have covered for your reading pleasure.

### Clean Architecture

- [The Clean Architecture](#)
- [Comparison of Domain-Driven Design and Clean Architecture Concepts](#)

### DDD

- [Effective Aggregate Design Part I: Modeling a Single Aggregate](#)
- [Domain Driven Design Quickly](#)
- [Domain-Driven Design Reference - Definitions and Pattern Summaries](#)



- [Serverless Domain Driven Design](#)
- [Uncovering Hidden Business Rules with DDD Aggregates](#)
- [Revisiting the Basics of Domain-Driven Design](#)
- [Why Your Microservices Architecture Needs Aggregates](#)
- [What Are Aggregates In Domain-Driven Design?](#)
- [DDD Aggregates: Consistency Boundary](#)
- [Identify microservice boundaries](#)
- [Using tactical DDD to design microservices](#)
- [An Introduction to Domain Storytelling](#)

## **EventStorming and event modeling**

- [Event Storming - Alberto Brandolini - DDD Europe 2019](#)
- [Event Modeling: What is it?](#)
- [Big Picture Event Storming](#)
- [Visualise coupling between contexts in Big Picture EventStorming](#)
- [Detailed agenda for a DDD Design-Level Event Storming - part 2](#)
- [EventBridge Storming — How to build state-of-the-art Event-Driven Serverless Architectures](#)

## **Event-Driven Architecture**

- [Designing Cloud Native Microservices on AWS \(via DDD/EventStormingWorkshop\)](#)
- [How to use EventBridge as a Cross-Account Event Backbone](#)
- [Serverless Streamed Events](#)

- [AWS re:Invent 2021 - Evolutionary AWS Lambda functions with hexagonal architecture \[REPEAT\]](#)
- [Statefarm: Comparison of AWS Services for Event Driven Architecture](#)

## **Resources**

- [Awesome Domain-Driven Design](#)
- [Awesome Software Architecture](#)
- [The Bounded Context Canvas](#)
- [Domain-Driven Design Crew](#)

## **Thank you for reading this book**

Thank you for investing your time, energy, and money into reading this book. With every book and article I write, I strive to make it as useful as possible. Books allow us to delve deeper—or sometimes broader—into topics than we typically can at work or in short-form articles. Technical books, in particular, are unique creatures: they are both products of their time and, when well-crafted, can become timeless resources within their field. I hope this book remains relevant for (at least a few!) years to come.

I write the books I wish I had read earlier in my career and life. I've tried to be generous with references to other content, such as books and articles that have helped me improve in this subject. There are so many great authors out there and so much knowledge to keep up with.

If you found this book helpful, I would greatly appreciate it if you could rate it on the platform where you purchased it.

Please don't be a stranger! Connect with me on LinkedIn or wherever else I may be when you're reading this.

Once again, thank you, and I hope you found value in the time we spent together.