O'REILLY®

Architecture Patterns with Python

Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices

Harry J.W. Percival
& Bob Gregory

O'REILLY®

Architecture Patterns with Python

As Python continues to grow in popularity, projects are becoming larger and more complex. Many Python developers are taking an interest in high-level software design patterns such as hexagonal/clean architecture, event-driven architecture, and the strategic patterns prescribed by domain-driven design (DDD). But translating those patterns into Python isn't always straightforward.

With this hands-on guide, Harry Percival and Bob Gregory from MADE.com introduce proven architectural design patterns to help Python developers manage application complexity—and get the most value out of their test suites.

Each pattern is illustrated with concrete examples in beautiful, idiomatic Python, avoiding some of the verbosity of Java and C# syntax. Patterns include:

- Dependency inversion and its links to ports and adapters (hexagonal/clean architecture)
- Domain-driven design's distinction between Entities, Value Objects, and Aggregates
- Repository and Unit of Work patterns for persistent storage
- Events, commands, and the message bus
- Command-query responsibility segregation (CQRS)
- Event-driven architecture and reactive microservices

Harry Percival has been a Python programmer and fan of TDD and XP since 2009. He's the author of *Test-Driven Development with Python* from O'Reilly, better known by its subtitle, *Obey the Testing Goat*.

Bob Gregory has been building event-driven systems using domain-driven design for more than a decade, in languages including C#, F#, Python, and TypeScript.

"The book the community has been waiting for: what the pivot to modern application architecture looks like in Python! Harry and Bob show how elegant the dependency inversion principle can look in a sleek, dynamic language."

—**Brandon Rhodes**
Author of *Python-patterns.guide*

"Python-native material on how to write maintainable, large-scale systems in Python has been virtually non-existent so far. This book shows that Python is more than adequate for serious software development."

—**Hynek Schlawack**
Pythonista, blogger, and speaker

PROGRAMMING / PYTHON

US $59.99 CAN $79.99
ISBN: 978-1-492-05220-3

Twitter: @oreillymedia
facebook.com/oreilly

Architecture Patterns with Python

Enabling Test-Driven Development, Domain-Driven Design, and Event-Driven Microservices

Harry Percival and Bob Gregory

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Architecture Patterns with Python
by Harry Percival and Bob Gregory

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Ryan Shaw
Development Editor: Corbin Collins
Production Editor: Katherine Tozer
Copyeditor: Sharon Wilkey
Proofreader: Arthur Johnson
Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2020: First Edition

Revision History for the First Edition
2020-03-05: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492052203* for release details.

978-1-492-05220-3

[LSI]

Table of Contents

Preface

You may be wondering who we are and why we wrote this book.

At the end of Harry's last book, *Test-Driven Development with Python* (O'Reilly), he found himself asking a bunch of questions about architecture, such as, What's the best way of structuring your application so that it's easy to test? More specifically, so that your core business logic is covered by unit tests, and so that you minimize the number of integration and end-to-end tests you need? He made vague references to "Hexagonal Architecture" and "Ports and Adapters" and "Functional Core, Imperative Shell," but if he was honest, he'd have to admit that these weren't things he really understood or had done in practice.

And then he was lucky enough to run into Bob, who has the answers to all these questions.

Bob ended up a software architect because nobody else on his team was doing it. He turned out to be pretty bad at it, but *he* was lucky enough to run into Ian Cooper, who taught him new ways of writing and thinking about code.

Managing Complexity, Solving Business Problems

We both work for MADE.com, a European ecommerce company that sells furniture online; there, we apply the techniques in this book to build distributed systems that model real-world business problems. Our example domain is the first system Bob built for MADE, and this book is an attempt to write down all the *stuff* we have to teach new programmers when they join one of our teams.

MADE.com operates a global supply chain of freight partners and manufacturers. To keep costs low, we try to optimize the delivery of stock to our warehouses so that we don't have unsold goods lying around the place.

Ideally, the sofa that you want to buy will arrive in port on the very day that you decide to buy it, and we'll ship it straight to your house without ever storing it.

Getting the timing right is a tricky balancing act when goods take three months to arrive by container ship. Along the way, things get broken or water damaged, storms cause unexpected delays, logistics partners mishandle goods, paperwork goes missing, customers change their minds and amend their orders, and so on.

We solve those problems by building intelligent software representing the kinds of operations taking place in the real world so that we can automate as much of the business as possible.

Why Python?

If you're reading this book, we probably don't need to convince you that Python is great, so the real question is "Why does the *Python* community need a book like this?" The answer is about Python's popularity and maturity: although Python is probably the world's fastest-growing programming language and is nearing the top of the absolute popularity tables, it's only just starting to take on the kinds of problems that the C# and Java world has been working on for years. Startups become real businesses; web apps and scripted automations are becoming (whisper it) *enterprise software*.

In the Python world, we often quote the Zen of Python: "There should be one—and preferably only one—obvious way to do it."[1] Unfortunately, as project size grows, the most obvious way of doing things isn't always the way that helps you manage complexity and evolving requirements.

None of the techniques and patterns we discuss in this book are new, but they are mostly new to the Python world. And this book isn't a replacement for the classics in the field such as Eric Evans's *Domain-Driven Design* or Martin Fowler's *Patterns of Enterprise Application Architecture* (both published by Addison-Wesley Professional) —which we often refer to and encourage you to go and read.

But all the classic code examples in the literature do tend to be written in Java or C++/#, and if you're a Python person and haven't used either of those languages in a long time (or indeed ever), those code listings can be quite…trying. There's a reason the latest edition of that other classic text, Fowler's *Refactoring* (Addison-Wesley Professional), is in JavaScript.

1 `python -c "import this"`

TDD, DDD, and Event-Driven Architecture

In order of notoriety, we know of three tools for managing complexity:

1. *Test-driven development* (TDD) helps us to build code that is correct and enables us to refactor or add new features, without fear of regression. But it can be hard to get the best out of our tests: How do we make sure that they run as fast as possible? That we get as much coverage and feedback from fast, dependency-free unit tests and have the minimum number of slower, flaky end-to-end tests?
2. *Domain-driven design* (DDD) asks us to focus our efforts on building a good model of the business domain, but how do we make sure that our models aren't encumbered with infrastructure concerns and don't become hard to change?
3. Loosely coupled (micro)services integrated via messages (sometimes called *reactive microservices*) are a well-established answer to managing complexity across multiple applications or business domains. But it's not always obvious how to make them fit with the established tools of the Python world—Flask, Django, Celery, and so on.

Don't be put off if you're not working with (or interested in) microservices. The vast majority of the patterns we discuss, including much of the event-driven architecture material, is absolutely applicable in a monolithic architecture.

Our aim with this book is to introduce several classic architectural patterns and show how they support TDD, DDD, and event-driven services. We hope it will serve as a reference for implementing them in a Pythonic way, and that people can use it as a first step toward further research in this field.

Who Should Read This Book

Here are a few things we assume about you, dear reader:

- You've been close to some reasonably complex Python applications.
- You've seen some of the pain that comes with trying to manage that complexity.
- You don't necessarily know anything about DDD or any of the classic application architecture patterns.

We structure our explorations of architectural patterns around an example app, building it up chapter by chapter. We use TDD at work, so we tend to show listings of tests first, followed by implementation. If you're not used to working test-first, it may feel a little strange at the beginning, but we hope you'll soon get used to seeing code "being used" (i.e., from the outside) before you see how it's built on the inside.

We use some specific Python frameworks and technologies, including Flask, SQLAlchemy, and pytest, as well as Docker and Redis. If you're already familiar with them, that won't hurt, but we don't think it's required. One of our main aims with this book is to build an architecture for which specific technology choices become minor implementation details.

A Brief Overview of What You'll Learn

The book is divided into two parts; here's a look at the topics we'll cover and the chapters they live in.

Part I, *Building an Architecture to Support Domain Modeling*

Domain modeling and DDD (Chapters 1 and 7)
: At some level, everyone has learned the lesson that complex business problems need to be reflected in code, in the form of a model of the domain. But why does it always seem to be so hard to do without getting tangled up with infrastructure concerns, our web frameworks, or whatever else? In the first chapter we give a broad overview of *domain modeling* and DDD, and we show how to get started with a model that has no external dependencies, and fast unit tests. Later we return to DDD patterns to discuss how to choose the right aggregate, and how this choice relates to questions of data integrity.

Repository, Service Layer, and Unit of Work patterns (Chapters 2, 4, and 5)
: In these three chapters we present three closely related and mutually reinforcing patterns that support our ambition to keep the model free of extraneous dependencies. We build a layer of abstraction around persistent storage, and we build a service layer to define the entrypoints to our system and capture the primary use cases. We show how this layer makes it easy to build thin entrypoints to our system, whether it's a Flask API or a CLI.

Some thoughts on testing and abstractions (Chapters 3 and 6)
: After presenting the first abstraction (the Repository pattern), we take the opportunity for a general discussion of how to choose abstractions, and what their role is in choosing how our software is coupled together. After we introduce the Service Layer pattern, we talk a bit about achieving a *test pyramid* and writing unit tests at the highest possible level of abstraction.

Part II, *Event-Driven Architecture*

Event-driven architecture (Chapters 8–11)
: We introduce three more mutually reinforcing patterns: the Domain Events, Message Bus, and Handler patterns. *Domain events* are a vehicle for capturing the idea that some interactions with a system are triggers for others. We use a *message bus* to allow actions to trigger events and call appropriate *handlers*. We move on to discuss how events can be used as a pattern for integration between services in a microservices architecture. Finally, we distinguish between *commands* and *events*. Our application is now fundamentally a message-processing system.

Command-query responsibility segregation (Chapter 12)
: We present an example of *command-query responsibility segregation*, with and without events.

Dependency injection (Chapter 13)
: We tidy up our explicit and implicit dependencies and implement a simple dependency injection framework.

Addtional Content

How do I get there from here? (Epilogue)
: Implementing architectural patterns always looks easy when you show a simple example, starting from scratch, but many of you will probably be wondering how to apply these principles to existing software. We'll provide a few pointers in the epilogue and some links to further reading.

Example Code and Coding Along

You're reading a book, but you'll probably agree with us when we say that the best way to learn about code is to code. We learned most of what we know from pairing with people, writing code with them, and learning by doing, and we'd like to re-create that experience as much as possible for you in this book.

As a result, we've structured the book around a single example project (although we do sometimes throw in other examples). We'll build up this project as the chapters progress, as if you've paired with us and we're explaining what we're doing and why at each step.

But to really get to grips with these patterns, you need to mess about with the code and get a feel for how it works. You'll find all the code on GitHub; each chapter has its own branch. You can find a list of the branches on GitHub as well.

Here are three ways you might code along with the book:

- Start your own repo and try to build up the app as we do, following the examples from listings in the book, and occasionally looking to our repo for hints. A word of warning, however: if you've read Harry's previous book and coded along with that, you'll find that this book requires you to figure out more on your own; you may need to lean pretty heavily on the working versions on GitHub.
- Try to apply each pattern, chapter by chapter, to your own (preferably small/toy) project, and see if you can make it work for your use case. This is high risk/high reward (and high effort besides!). It may take quite some work to get things working for the specifics of your project, but on the other hand, you're likely to learn the most.
- For less effort, in each chapter we outline an "Exercise for the Reader," and point you to a GitHub location where you can download some partially finished code for the chapter with a few missing parts to write yourself.

Particularly if you're intending to apply some of these patterns in your own projects, working through a simple example is a great way to safely practice.

At the very least, do a `git checkout` of the code from our repo as you read each chapter. Being able to jump in and see the code in the context of an actual working app will help answer a lot of questions as you go, and makes everything more real. You'll find instructions for how to do that at the beginning of each chapter.

License

The code (and the online version of the book) is licensed under a Creative Commons CC BY-NC-ND license, which means you are free to copy and share it with anyone you like, for non-commercial purposes, as long as you give attribution. If you want to re-use any of the content from this book and you have any worries about the license, contact O'Reilly at *permissions@oreilly.com*.

The print edition is licensed differently; please see the copyright page.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact O'Reilly

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> 800-998-9938 (in the United States or Canada)
> 707-829-0515 (international or local)
> 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/architecture-patterns-python*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

To our tech reviewers, David Seddon, Ed Jung, and Hynek Schlawack: we absolutely do not deserve you. You are all incredibly dedicated, conscientious, and rigorous. Each one of you is immensely smart, and your different points of view were both useful and complementary to each other. Thank you from the bottom of our hearts.

Gigantic thanks also to our Early Release readers for their comments and suggestions: Ian Cooper, Abdullah Ariff, Jonathan Meier, Gil Gonçalves, Matthieu Choplin, Ben Judson, James Gregory, Łukasz Lechowicz, Clinton Roy, Vitorino Araújo, Susan Goodbody, Josh Harwood, Daniel Butler, Liu Haibin, Jimmy Davies, Ignacio Vergara Kausel, Gaia Canestrani, Renne Rocha, pedroabi, Ashia Zawaduk, Jostein Leira, Brandon Rhodes, and many more; our apologies if we missed you on this list.

Super-mega-thanks to our editor Corbin Collins for his gentle chivvying, and for being a tireless advocate of the reader. Similarly-superlative thanks to the production staff, Katherine Tozer, Sharon Wilkey, Ellen Troutman-Zaig, and Rebecca Demarest, for your dedication, professionalism, and attention to detail. This book is immeasurably improved thanks to you.

Any errors remaining in the book are our own, naturally.

Introduction

Why Do Our Designs Go Wrong?

What comes to mind when you hear the word *chaos?* Perhaps you think of a noisy stock exchange, or your kitchen in the morning—everything confused and jumbled. When you think of the word *order*, perhaps you think of an empty room, serene and calm. For scientists, though, chaos is characterized by homogeneity (sameness), and order by complexity (difference).

For example, a well-tended garden is a highly ordered system. Gardeners define boundaries with paths and fences, and they mark out flower beds or vegetable patches. Over time, the garden evolves, growing richer and thicker; but without deliberate effort, the garden will run wild. Weeds and grasses will choke out other plants, covering over the paths, until eventually every part looks the same again—wild and unmanaged.

Software systems, too, tend toward chaos. When we first start building a new system, we have grand ideas that our code will be clean and well ordered, but over time we find that it gathers cruft and edge cases and ends up a confusing morass of manager classes and util modules. We find that our sensibly layered architecture has collapsed into itself like an oversoggy trifle. Chaotic software systems are characterized by a sameness of function: API handlers that have domain knowledge and send email and perform logging; "business logic" classes that perform no calculations but do perform I/O; and everything coupled to everything else so that changing any part of the system becomes fraught with danger. This is so common that software engineers have their own term for chaos: the Big Ball of Mud anti-pattern (Figure P-1).

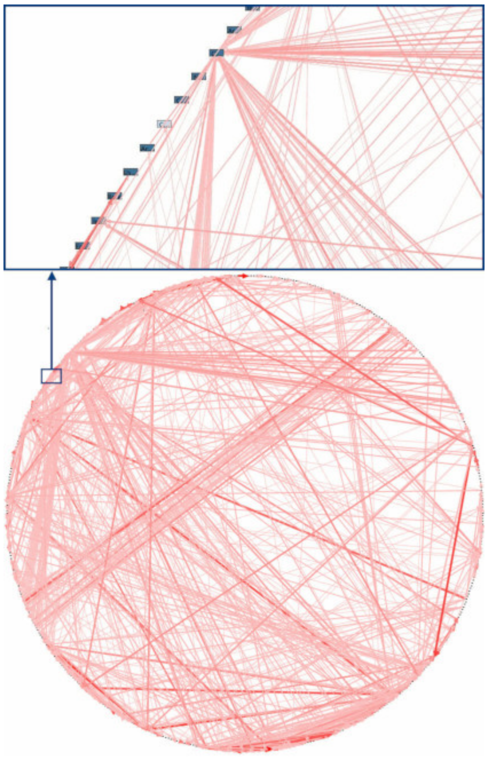


Figure P-1. A real-life dependency diagram (source: "Enterprise Dependency: Big Ball of Yarn" by Alex Papadimoulis)

A big ball of mud is the natural state of software in the same way that wilderness is the natural state of your garden. It takes energy and direction to prevent the collapse.

Fortunately, the techniques to avoid creating a big ball of mud aren't complex.

Encapsulation and Abstractions

Encapsulation and abstraction are tools that we all instinctively reach for as programmers, even if we don't all use these exact words. Allow us to dwell on them for a moment, since they are a recurring background theme of the book.

The term *encapsulation* covers two closely related ideas: simplifying behavior and hiding data. In this discussion, we're using the first sense. We encapsulate behavior by identifying a task that needs to be done in our code and giving that task to a well-defined object or function. We call that object or function an *abstraction*.

Take a look at the following two snippets of Python code:

Do a search with urllib

```python
import json
from urllib.request import urlopen
from urllib.parse import urlencode

params = dict(q='Sausages', format='json')
handle = urlopen('http://api.duckduckgo.com' + '?' + urlencode(params))
raw_text = handle.read().decode('utf8')
parsed = json.loads(raw_text)

results = parsed['RelatedTopics']
for r in results:
    if 'Text' in r:
        print(r['FirstURL'] + ' - ' + r['Text'])
```

Do a search with requests

```python
import requests

params = dict(q='Sausages', format='json')
parsed = requests.get('http://api.duckduckgo.com/', params=params).json()

results = parsed['RelatedTopics']
for r in results:
    if 'Text' in r:
        print(r['FirstURL'] + ' - ' + r['Text'])
```

Both code listings do the same thing: they submit form-encoded values to a URL in order to use a search engine API. But the second is simpler to read and understand because it operates at a higher level of abstraction.

We can take this one step further still by identifying and naming the task we want the code to perform for us and using an even higher-level abstraction to make it explicit:

Do a search with the duckduckgo module

```python
import duckduckgo
for r in duckduckgo.query('Sausages').results:
    print(r.url + ' - ' + r.text)
```

Encapsulating behavior by using abstractions is a powerful tool for making code more expressive, more testable, and easier to maintain.

In the literature of the object-oriented (OO) world, one of the classic characterizations of this approach is called *responsibility-driven design*; it uses the words *roles* and *responsibilities* rather than *tasks*. The main point is to think about code in terms of behavior, rather than in terms of data or algorithms.[1]

Abstractions and ABCs

In a traditional OO language like Java or C#, you might use an abstract base class (ABC) or an interface to define an abstraction. In Python you can (and we sometimes do) use ABCs, but you can also happily rely on duck typing.

The abstraction can just mean "the public API of the thing you're using"—a function name plus some arguments, for example.

Most of the patterns in this book involve choosing an abstraction, so you'll see plenty of examples in each chapter. In addition, Chapter 3 specifically discusses some general heuristics for choosing abstractions.

Layering

Encapsulation and abstraction help us by hiding details and protecting the consistency of our data, but we also need to pay attention to the interactions between our objects and functions. When one function, module, or object uses another, we say that the one *depends on* the other. These dependencies form a kind of network or graph.

In a big ball of mud, the dependencies are out of control (as you saw in Figure P-1). Changing one node of the graph becomes difficult because it has the potential to affect many other parts of the system. Layered architectures are one way of tackling this problem. In a layered architecture, we divide our code into discrete categories or roles, and we introduce rules about which categories of code can call each other.

One of the most common examples is the *three-layered architecture* shown in Figure P-2.

1 If you've come across class-responsibility-collaborator (CRC) cards, they're driving at the same thing: thinking about *responsibilities* helps you decide how to split things up.

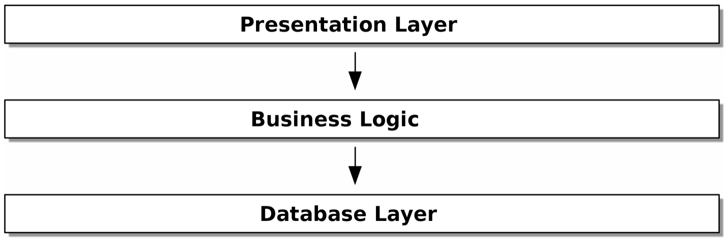


Figure P-2. Layered architecture

Layered architecture is perhaps the most common pattern for building business software. In this model we have user-interface components, which could be a web page, an API, or a command line; these user-interface components communicate with a business logic layer that contains our business rules and our workflows; and finally, we have a database layer that's responsible for storing and retrieving data.

For the rest of this book, we're going to be systematically turning this model inside out by obeying one simple principle.

The Dependency Inversion Principle

You might be familiar with the *dependency inversion principle* (DIP) already, because it's the *D* in SOLID.[2]

Unfortunately, we can't illustrate the DIP by using three tiny code listings as we did for encapsulation. However, the whole of Part I is essentially a worked example of implementing the DIP throughout an application, so you'll get your fill of concrete examples.

In the meantime, we can talk about DIP's formal definition:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Instead, details should depend on abstractions.

But what does this mean? Let's take it bit by bit.

High-level modules are the code that your organization really cares about. Perhaps you work for a pharmaceutical company, and your high-level modules deal with patients

2 SOLID is an acronym for Robert C. Martin's five principles of object-oriented design: single responsibility, open for extension but closed for modification, Liskov substitution, interface segregation, and dependency inversion. See "S.O.L.I.D: The First 5 Principles of Object-Oriented Design" by Samuel Oloruntoba.

and trials. Perhaps you work for a bank, and your high-level modules manage trades and exchanges. The high-level modules of a software system are the functions, classes, and packages that deal with our real-world concepts.

By contrast, *low-level modules* are the code that your organization doesn't care about. It's unlikely that your HR department gets excited about filesystems or network sockets. It's not often that you discuss SMTP, HTTP, or AMQP with your finance team. For our nontechnical stakeholders, these low-level concepts aren't interesting or relevant. All they care about is whether the high-level concepts work correctly. If payroll runs on time, your business is unlikely to care whether that's a cron job or a transient function running on Kubernetes.

Depends on doesn't mean *imports* or *calls*, necessarily, but rather a more general idea that one module *knows about* or *needs* another module.

And we've mentioned *abstractions* already: they're simplified interfaces that encapsulate behavior, in the way that our duckduckgo module encapsulated a search engine's API.

> All problems in computer science can be solved by adding another level of indirection.
>
> —David Wheeler

So the first part of the DIP says that our business code shouldn't depend on technical details; instead, both should use abstractions.

Why? Broadly, because we want to be able to change them independently of each other. High-level modules should be easy to change in response to business needs. Low-level modules (details) are often, in practice, harder to change: think about refactoring to change a function name versus defining, testing, and deploying a database migration to change a column name. We don't want business logic changes to slow down because they are closely coupled to low-level infrastructure details. But, similarly, it is important to *be able* to change your infrastructure details when you need to (think about sharding a database, for example), without needing to make changes to your business layer. Adding an abstraction between them (the famous extra layer of indirection) allows the two to change (more) independently of each other.

The second part is even more mysterious. "Abstractions should not depend on details" seems clear enough, but "Details should depend on abstractions" is hard to imagine. How can we have an abstraction that doesn't depend on the details it's abstracting? By the time we get to Chapter 4, we'll have a concrete example that should make this all a bit clearer.

A Place for All Our Business Logic: The Domain Model

But before we can turn our three-layered architecture inside out, we need to talk more about that middle layer: the high-level modules or business logic. One of the most common reasons that our designs go wrong is that business logic becomes spread throughout the layers of our application, making it hard to identify, understand, and change.

Chapter 1 shows how to build a business layer with a *Domain Model* pattern. The rest of the patterns in Part I show how we can keep the domain model easy to change and free of low-level concerns by choosing the right abstractions and continuously applying the DIP.

PART I

Building an Architecture to Support Domain Modeling

Most developers have never seen a domain model, only a data model.

—Cyrille Martraire, *DDD EU 2017*

Most developers we talk to about architecture have a nagging sense that things could be better. They are often trying to rescue a system that has gone wrong somehow, and are trying to put some structure back into a ball of mud. They know that their business logic shouldn't be spread all over the place, but they have no idea how to fix it.

We've found that many developers, when asked to design a new system, will immediately start to build a database schema, with the object model treated as an afterthought. This is where it all starts to go wrong. Instead, *behavior should come first and drive our storage requirements*. After all, our customers don't care about the data model. They care about what the system *does*; otherwise they'd just use a spreadsheet.

The first part of the book looks at how to build a rich object model through TDD (in Chapter 1), and then we'll show how to keep that model decoupled from technical concerns. We show how to build persistence-ignorant code and how to create stable APIs around our domain so that we can refactor aggressively.

To do that, we present four key design patterns:

- The Repository pattern, an abstraction over the idea of persistent storage
- The Service Layer pattern to clearly define where our use cases begin and end

- The Unit of Work pattern to provide atomic operations
- The Aggregate pattern to enforce the integrity of our data

If you'd like a picture of where we're going, take a look at Figure I-1, but don't worry if none of it makes sense yet! We introduce each box in the figure, one by one, throughout this part of the book.

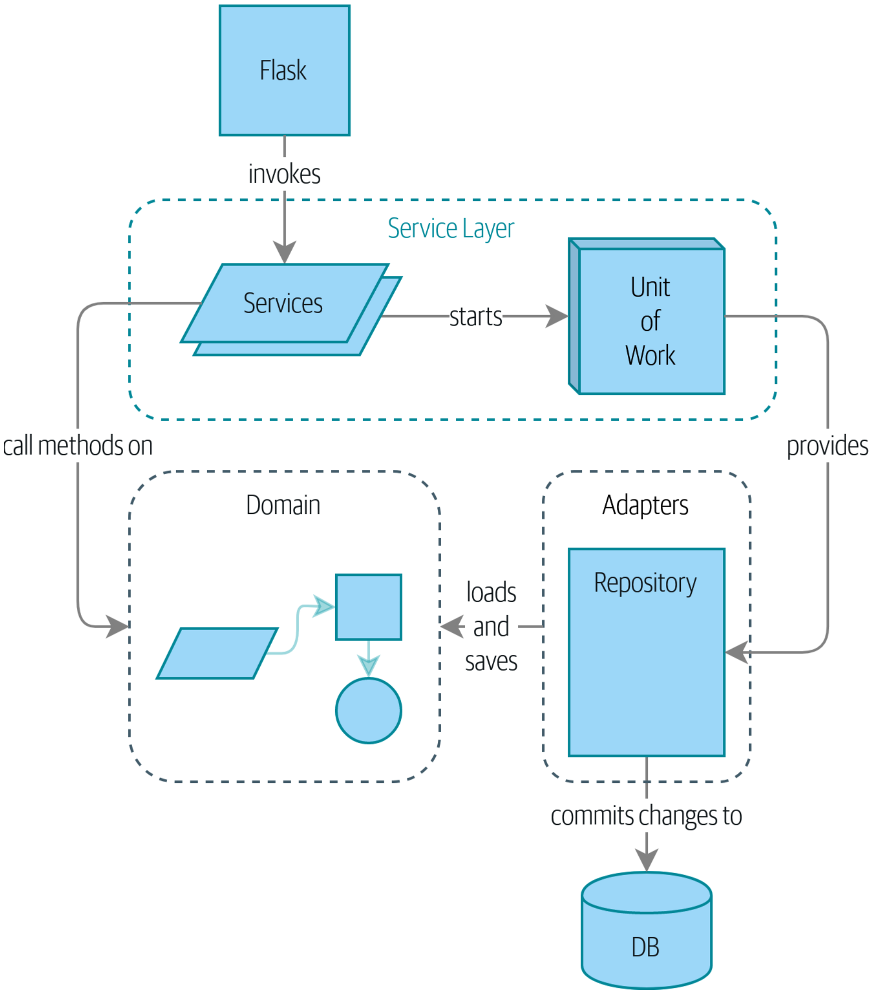


Figure I-1. A component diagram for our app at the end of Part I

We also take a little time out to talk about coupling and abstractions, illustrating it with a simple example that shows how and why we choose our abstractions.

Three appendices are further explorations of the content from Part I:

- Appendix B is a write-up of the infrastructure for our example code: how we build and run the Docker images, where we manage configuration info, and how we run different types of tests.
- Appendix C is a "proof is in the pudding" kind of content, showing how easy it is to swap out our entire infrastructure—the Flask API, the ORM, and Postgres—for a totally different I/O model involving a CLI and CSVs.
- Finally, Appendix D may be of interest if you're wondering how these patterns might look if using Django instead of Flask and SQLAlchemy.

CHAPTER 1

Domain Modeling

This chapter looks into how we can model business processes with code, in a way that's highly compatible with TDD. We'll discuss *why* domain modeling matters, and we'll look at a few key patterns for modeling domains: Entity, Value Object, and Domain Service.

Figure 1-1 is a simple visual placeholder for our Domain Model pattern. We'll fill in some details in this chapter, and as we move on to other chapters, we'll build things around the domain model, but you should always be able to find these little shapes at the core.

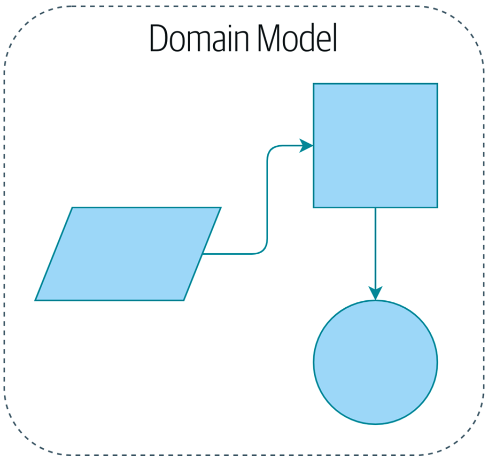


Figure 1-1. A placeholder illustration of our domain model

What Is a Domain Model?

In the introduction, we used the term *business logic layer* to describe the central layer of a three-layered architecture. For the rest of the book, we're going to use the term *domain model* instead. This is a term from the DDD community that does a better job of capturing our intended meaning (see the next sidebar for more on DDD).

The *domain* is a fancy way of saying *the problem you're trying to solve.* Your authors currently work for an online retailer of furniture. Depending on which system you're talking about, the domain might be purchasing and procurement, or product design, or logistics and delivery. Most programmers spend their days trying to improve or automate business processes; the domain is the set of activities that those processes support.

A *model* is a map of a process or phenomenon that captures a useful property. Humans are exceptionally good at producing models of things in their heads. For example, when someone throws a ball toward you, you're able to predict its movement almost unconsciously, because you have a model of the way objects move in space. Your model isn't perfect by any means. Humans have terrible intuitions about how objects behave at near-light speeds or in a vacuum because our model was never designed to cover those cases. That doesn't mean the model is wrong, but it does mean that some predictions fall outside of its domain.

The domain model is the mental map that business owners have of their businesses. All business people have these mental maps—they're how humans think about complex processes.

You can tell when they're navigating these maps because they use business speak. Jargon arises naturally among people who are collaborating on complex systems.

Imagine that you, our unfortunate reader, were suddenly transported light years away from Earth aboard an alien spaceship with your friends and family and had to figure out, from first principles, how to navigate home.

In your first few days, you might just push buttons randomly, but soon you'd learn which buttons did what, so that you could give one another instructions. "Press the red button near the flashing doohickey and then throw that big lever over by the radar gizmo," you might say.

Within a couple of weeks, you'd become more precise as you adopted words to describe the ship's functions: "Increase oxygen levels in cargo bay three" or "turn on the little thrusters." After a few months, you'd have adopted language for entire complex processes: "Start landing sequence" or "prepare for warp." This process would happen quite naturally, without any formal effort to build a shared glossary.

This Is Not a DDD Book. You Should Read a DDD Book.

Domain-driven design, or DDD, popularized the concept of domain modeling,[1] and it's been a hugely successful movement in transforming the way people design software by focusing on the core business domain. Many of the architecture patterns that we cover in this book—including Entity, Aggregate, Value Object (see Chapter 7), and Repository (in the next chapter)—come from the DDD tradition.

In a nutshell, DDD says that the most important thing about software is that it provides a useful model of a problem. If we get that model right, our software delivers value and makes new things possible.

If we get the model wrong, it becomes an obstacle to be worked around. In this book, we can show the basics of building a domain model, and building an architecture around it that leaves the model as free as possible from external constraints, so that it's easy to evolve and change.

But there's a lot more to DDD and to the processes, tools, and techniques for developing a domain model. We hope to give you a taste of it, though, and cannot encourage you enough to go on and read a proper DDD book:

- The original "blue book," *Domain-Driven Design* by Eric Evans (Addison-Wesley Professional)
- The "red book," *Implementing Domain-Driven Design* by Vaughn Vernon (Addison-Wesley Professional)

So it is in the mundane world of business. The terminology used by business stakeholders represents a distilled understanding of the domain model, where complex ideas and processes are boiled down to a single word or phrase.

When we hear our business stakeholders using unfamiliar words, or using terms in a specific way, we should listen to understand the deeper meaning and encode their hard-won experience into our software.

We're going to use a real-world domain model throughout this book, specifically a model from our current employment. MADE.com is a successful furniture retailer. We source our furniture from manufacturers all over the world and sell it across Europe.

1 DDD did not originate domain modeling. Eric Evans refers to the 2002 book *Object Design* by Rebecca Wirfs-Brock and Alan McKean (Addison-Wesley Professional), which introduced responsibility-driven design, of which DDD is a special case dealing with the domain. But even that is too late, and OO enthusiasts will tell you to look further back to Ivar Jacobson and Grady Booch; the term has been around since the mid-1980s.

When you buy a sofa or a coffee table, we have to figure out how best to get your goods from Poland or China or Vietnam and into your living room.

At a high level, we have separate systems that are responsible for buying stock, selling stock to customers, and shipping goods to customers. A system in the middle needs to coordinate the process by allocating stock to a customer's orders; see Figure 1-2.

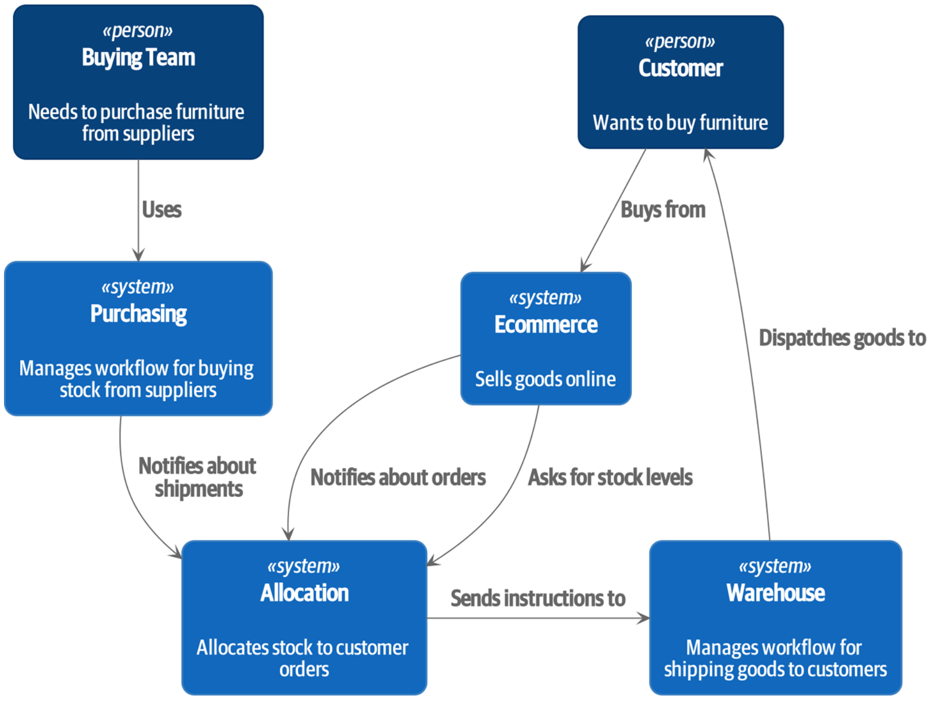


Figure 1-2. Context diagram for the allocation service

For the purposes of this book, we're imagining that the business decides to implement an exciting new way of allocating stock. Until now, the business has been presenting stock and lead times based on what is physically available in the warehouse. If and when the warehouse runs out, a product is listed as "out of stock" until the next shipment arrives from the manufacturer.

Here's the innovation: if we have a system that can keep track of all our shipments and when they're due to arrive, we can treat the goods on those ships as real stock and part of our inventory, just with slightly longer lead times. Fewer goods will appear to be out of stock, we'll sell more, and the business can save money by keeping lower inventory in the domestic warehouse.

But allocating orders is no longer a trivial matter of decrementing a single quantity in the warehouse system. We need a more complex allocation mechanism. Time for some domain modeling.

Exploring the Domain Language

Understanding the domain model takes time, and patience, and Post-it notes. We have an initial conversation with our business experts and agree on a glossary and some rules for the first minimal version of the domain model. Wherever possible, we ask for concrete examples to illustrate each rule.

We make sure to express those rules in the business jargon (the *ubiquitous language* in DDD terminology). We choose memorable identifiers for our objects so that the examples are easier to talk about.

"Some Notes on Allocation" shows some notes we might have taken while having a conversation with our domain experts about allocation.

Some Notes on Allocation

A *product* is identified by a *SKU*, pronounced "skew," which is short for *stock-keeping unit*. *Customers* place *orders*. An order is identified by an *order reference* and comprises multiple *order lines*, where each line has a *SKU* and a *quantity*. For example:

- 10 units of RED-CHAIR
- 1 unit of TASTELESS-LAMP

The purchasing department orders small *batches* of stock. A *batch* of stock has a unique ID called a *reference*, a *SKU*, and a *quantity*.

We need to *allocate order lines* to *batches*. When we've allocated an order line to a batch, we will send stock from that specific batch to the customer's delivery address. When we allocate *x* units of stock to a batch, the *available quantity* is reduced by *x*. For example:

- We have a batch of 20 SMALL-TABLE, and we allocate an order line for 2 SMALL-TABLE.
- The batch should have 18 SMALL-TABLE remaining.

We can't allocate to a batch if the available quantity is less than the quantity of the order line. For example:

- We have a batch of 1 BLUE-CUSHION, and an order line for 2 BLUE-CUSHION.

- We should not be able to allocate the line to the batch.

We can't allocate the same line twice. For example:

- We have a batch of 10 BLUE-VASE, and we allocate an order line for 2 BLUE-VASE.
- If we allocate the order line again to the same batch, the batch should still have an available quantity of 8.

Batches have an *ETA* if they are currently shipping, or they may be in *warehouse stock*. We allocate to warehouse stock in preference to shipment batches. We allocate to shipment batches in order of which has the earliest ETA.

Unit Testing Domain Models

We're not going to show you how TDD works in this book, but we want to show you how we would construct a model from this business conversation.

Exercise for the Reader

Why not have a go at solving this problem yourself? Write a few unit tests to see if you can capture the essence of these business rules in nice, clean code.

You'll find some placeholder unit tests on GitHub, but you could just start from scratch, or combine/rewrite them however you like.

Here's what one of our first tests might look like:

A first test for allocation (test_batches.py)

```python
def test_allocating_to_a_batch_reduces_the_available_quantity():
    batch = Batch("batch-001", "SMALL-TABLE", qty=20, eta=date.today())
    line = OrderLine('order-ref', "SMALL-TABLE", 2)

    batch.allocate(line)

    assert batch.available_quantity == 18
```

The name of our unit test describes the behavior that we want to see from the system, and the names of the classes and variables that we use are taken from the business jargon. We could show this code to our nontechnical coworkers, and they would agree that this correctly describes the behavior of the system.

And here is a domain model that meets our requirements:

First cut of a domain model for batches (model.py)

```
@dataclass(frozen=True)  ❶❷
class OrderLine:
    orderid: str
    sku: str
    qty: int


class Batch:
    def __init__(
        self, ref: str, sku: str, qty: int, eta: Optional[date]  ❷
    ):
        self.reference = ref
        self.sku = sku
        self.eta = eta
        self.available_quantity = qty

    def allocate(self, line: OrderLine):
        self.available_quantity -= line.qty  ❸
```

❶ `OrderLine` is an immutable dataclass with no behavior.[2]

❷ We're not showing imports in most code listings, in an attempt to keep them clean. We're hoping you can guess that this came via `from dataclasses import dataclass`; likewise, `typing.Optional` and `datetime.date`. If you want to double-check anything, you can see the full working code for each chapter in its branch (e.g., chapter_01_domain_model).

❸ Type hints are still a matter of controversy in the Python world. For domain models, they can sometimes help to clarify or document what the expected arguments are, and people with IDEs are often grateful for them. You may decide the price paid in terms of readability is too high.

Our implementation here is trivial: a `Batch` just wraps an integer `available_quantity`, and we decrement that value on allocation. We've written quite a lot of code just to subtract one number from another, but we think that modeling our domain precisely will pay off.[3]

Let's write some new failing tests:

2 In previous Python versions, we might have used a namedtuple. You could also check out Hynek Schlawack's excellent attrs.

3 Or perhaps you think there's not enough code? What about some sort of check that the SKU in the `OrderLine` matches `Batch.sku`? We saved some thoughts on validation for Appendix E.

Testing logic for what we can allocate (test_batches.py)

```
def make_batch_and_line(sku, batch_qty, line_qty):
    return (
        Batch("batch-001", sku, batch_qty, eta=date.today()),
        OrderLine("order-123", sku, line_qty)
    )


def test_can_allocate_if_available_greater_than_required():
    large_batch, small_line = make_batch_and_line("ELEGANT-LAMP", 20, 2)
    assert large_batch.can_allocate(small_line)

def test_cannot_allocate_if_available_smaller_than_required():
    small_batch, large_line = make_batch_and_line("ELEGANT-LAMP", 2, 20)
    assert small_batch.can_allocate(large_line) is False

def test_can_allocate_if_available_equal_to_required():
    batch, line = make_batch_and_line("ELEGANT-LAMP", 2, 2)
    assert batch.can_allocate(line)

def test_cannot_allocate_if_skus_do_not_match():
    batch = Batch("batch-001", "UNCOMFORTABLE-CHAIR", 100, eta=None)
    different_sku_line = OrderLine("order-123", "EXPENSIVE-TOASTER", 10)
    assert batch.can_allocate(different_sku_line) is False
```

There's nothing too unexpected here. We've refactored our test suite so that we don't keep repeating the same lines of code to create a batch and a line for the same SKU; and we've written four simple tests for a new method `can_allocate`. Again, notice that the names we use mirror the language of our domain experts, and the examples we agreed upon are directly written into code.

We can implement this straightforwardly, too, by writing the `can_allocate` method of `Batch`:

A new method in the model (model.py)

```
    def can_allocate(self, line: OrderLine) -> bool:
        return self.sku == line.sku and self.available_quantity >= line.qty
```

So far, we can manage the implementation by just incrementing and decrementing `Batch.available_quantity`, but as we get into `deallocate()` tests, we'll be forced into a more intelligent solution:

This test is going to require a smarter model (test_batches.py)

```
def test_can_only_deallocate_allocated_lines():
    batch, unallocated_line = make_batch_and_line("DECORATIVE-TRINKET", 20, 2)
    batch.deallocate(unallocated_line)
    assert batch.available_quantity == 20
```

In this test, we're asserting that deallocating a line from a batch has no effect unless the batch previously allocated the line. For this to work, our `Batch` needs to understand which lines have been allocated. Let's look at the implementation:

The domain model now tracks allocations (model.py)

```
class Batch:
    def __init__(
        self, ref: str, sku: str, qty: int, eta: Optional[date]
    ):
        self.reference = ref
        self.sku = sku
        self.eta = eta
        self._purchased_quantity = qty
        self._allocations = set()  # type: Set[OrderLine]

    def allocate(self, line: OrderLine):
        if self.can_allocate(line):
            self._allocations.add(line)

    def deallocate(self, line: OrderLine):
        if line in self._allocations:
            self._allocations.remove(line)

    @property
    def allocated_quantity(self) -> int:
        return sum(line.qty for line in self._allocations)

    @property
    def available_quantity(self) -> int:
        return self._purchased_quantity - self.allocated_quantity

    def can_allocate(self, line: OrderLine) -> bool:
        return self.sku == line.sku and self.available_quantity >= line.qty
```

Figure 1-3 shows the model in UML.

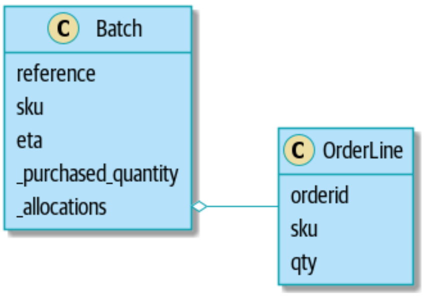


Figure 1-3. Our model in UML

Now we're getting somewhere! A batch now keeps track of a set of allocated `OrderLine` objects. When we allocate, if we have enough available quantity, we just add to the set. Our `available_quantity` is now a calculated property: purchased quantity minus allocated quantity.

Yes, there's plenty more we could do. It's a little disconcerting that both `allocate()` and `deallocate()` can fail silently, but we have the basics.

Incidentally, using a set for `._allocations` makes it simple for us to handle the last test, because items in a set are unique:

Last batch test! (test_batches.py)

```
def test_allocation_is_idempotent():
    batch, line = make_batch_and_line("ANGULAR-DESK", 20, 2)
    batch.allocate(line)
    batch.allocate(line)
    assert batch.available_quantity == 18
```

At the moment, it's probably a valid criticism to say that the domain model is too trivial to bother with DDD (or even object orientation!). In real life, any number of business rules and edge cases crop up: customers can ask for delivery on specific future dates, which means we might not want to allocate them to the earliest batch. Some SKUs aren't in batches, but ordered on demand directly from suppliers, so they have different logic. Depending on the customer's location, we can allocate to only a subset of warehouses and shipments that are in their region—except for some SKUs we're happy to deliver from a warehouse in a different region if we're out of stock in the home region. And so on. A real business in the real world knows how to pile on complexity faster than we can show on the page!

But taking this simple domain model as a placeholder for something more complex, we're going to extend our simple domain model in the rest of the book and plug it into the real world of APIs and databases and spreadsheets. We'll see how sticking rigidly to our principles of encapsulation and careful layering will help us to avoid a ball of mud.

More Types for More Type Hints

If you really want to go to town with type hints, you could go so far as wrapping primitive types by using `typing.NewType`:

Just taking it way too far, Bob

```
from dataclasses import dataclass
from typing import NewType

Quantity = NewType("Quantity", int)
Sku = NewType("Sku", str)
Reference = NewType("Reference", str)
...

class Batch:
    def __init__(self, ref: Reference, sku: Sku, qty: Quantity):
        self.sku = sku
        self.reference = ref
        self._purchased_quantity = qty
```

That would allow our type checker to make sure that we don't pass a `Sku` where a `Reference` is expected, for example.

Whether you think this is wonderful or appalling is a matter of debate.[4]

Dataclasses Are Great for Value Objects

We've used `line` liberally in the previous code listings, but what is a line? In our business language, an *order* has multiple *line* items, where each line has a SKU and a quantity. We can imagine that a simple YAML file containing order information might look like this:

Order info as YAML

```
Order_reference: 12345
Lines:
  - sku: RED-CHAIR
    qty: 25
  - sku: BLU-CHAIR
    qty: 25
  - sku: GRN-CHAIR
    qty: 25
```

4 It is appalling. Please, please don't do this. —Harry

Notice that while an order has a *reference* that uniquely identifies it, a *line* does not. (Even if we add the order reference to the `OrderLine` class, it's not something that uniquely identifies the line itself.)

Whenever we have a business concept that has data but no identity, we often choose to represent it using the *Value Object* pattern. A *value object* is any domain object that is uniquely identified by the data it holds; we usually make them immutable:

OrderLine is a value object

```
@dataclass(frozen=True)
class OrderLine:
    orderid: OrderReference
    sku: ProductReference
    qty: Quantity
```

One of the nice things that dataclasses (or namedtuples) give us is *value equality*, which is the fancy way of saying, "Two lines with the same `orderid`, `sku`, and `qty` are equal."

More examples of value objects

```
from dataclasses import dataclass
from typing import NamedTuple
from collections import namedtuple

@dataclass(frozen=True)
class Name:
    first_name: str
    surname: str

class Money(NamedTuple):
    currency: str
    value: int

Line = namedtuple('Line', ['sku', 'qty'])

def test_equality():
    assert Money('gbp', 10) == Money('gbp', 10)
    assert Name('Harry', 'Percival') != Name('Bob', 'Gregory')
    assert Line('RED-CHAIR', 5) == Line('RED-CHAIR', 5)
```

These value objects match our real-world intuition about how their values work. It doesn't matter *which* £10 note we're talking about, because they all have the same value. Likewise, two names are equal if both the first and last names match; and two lines are equivalent if they have the same customer order, product code, and quantity. We can still have complex behavior on a value object, though. In fact, it's common to support operations on values; for example, mathematical operators:

Math with value objects

```
fiver = Money('gbp', 5)
tenner = Money('gbp', 10)

def can_add_money_values_for_the_same_currency():
    assert fiver + fiver == tenner

def can_subtract_money_values():
    assert tenner - fiver == fiver

def adding_different_currencies_fails():
    with pytest.raises(ValueError):
        Money('usd', 10) + Money('gbp', 10)

def can_multiply_money_by_a_number():
    assert fiver * 5 == Money('gbp', 25)

def multiplying_two_money_values_is_an_error():
    with pytest.raises(TypeError):
        tenner * fiver
```

Value Objects and Entities

An order line is uniquely identified by its order ID, SKU, and quantity; if we change one of those values, we now have a new line. That's the definition of a value object: any object that is identified only by its data and doesn't have a long-lived identity. What about a batch, though? That *is* identified by a reference.

We use the term *entity* to describe a domain object that has long-lived identity. On the previous page, we introduced a `Name` class as a value object. If we take the name Harry Percival and change one letter, we have the new `Name` object Barry Percival.

It should be clear that Harry Percival is not equal to Barry Percival:

A name itself cannot change...

```
def test_name_equality():
    assert Name("Harry", "Percival") != Name("Barry", "Percival")
```

But what about Harry as a *person*? People do change their names, and their marital status, and even their gender, but we continue to recognize them as the same individual. That's because humans, unlike names, have a persistent *identity*:

But a person can!

```
class Person:

    def __init__(self, name: Name):
        self.name = name
```

```
def test_barry_is_harry():
    harry = Person(Name("Harry", "Percival"))
    barry = harry

    barry.name = Name("Barry", "Percival")

    assert harry is barry and barry is harry
```

Entities, unlike values, have *identity equality*. We can change their values, and they are still recognizably the same thing. Batches, in our example, are entities. We can allocate lines to a batch, or change the date that we expect it to arrive, and it will still be the same entity.

We usually make this explicit in code by implementing equality operators on entities:

Implementing equality operators (model.py)

```
class Batch:
    ...

    def __eq__(self, other):
        if not isinstance(other, Batch):
            return False
        return other.reference == self.reference

    def __hash__(self):
        return hash(self.reference)
```

Python's `__eq__` magic method defines the behavior of the class for the == operator.[5]

For both entity and value objects, it's also worth thinking through how `__hash__` will work. It's the magic method Python uses to control the behavior of objects when you add them to sets or use them as dict keys; you can find more info in the Python docs.

For value objects, the hash should be based on all the value attributes, and we should ensure that the objects are immutable. We get this for free by specifying `@frozen=True` on the dataclass.

For entities, the simplest option is to say that the hash is `None`, meaning that the object is not hashable and cannot, for example, be used in a set. If for some reason you decide you really do want to use set or dict operations with entities, the hash should be based on the attribute(s), such as `.reference`, that defines the entity's unique identity over time. You should also try to somehow make *that* attribute read-only.

5 The `__eq__` method is pronounced "dunder-EQ." By some, at least.

This is tricky territory; you shouldn't modify `__hash__` without also modifying `__eq__`. If you're not sure what you're doing, further reading is suggested. "Python Hashes and Equality" by our tech reviewer Hynek Schlawack is a good place to start.

Not Everything Has to Be an Object: A Domain Service Function

We've made a model to represent batches, but what we actually need to do is allocate order lines against a specific set of batches that represent all our stock.

> Sometimes, it just isn't a thing.
>
> —Eric Evans, *Domain-Driven Design*

Evans discusses the idea of Domain Service operations that don't have a natural home in an entity or value object.[6] A thing that allocates an order line, given a set of batches, sounds a lot like a function, and we can take advantage of the fact that Python is a multiparadigm language and just make it a function.

Let's see how we might test-drive such a function:

Testing our domain service (test_allocate.py)

```
def test_prefers_current_stock_batches_to_shipments():
    in_stock_batch = Batch("in-stock-batch", "RETRO-CLOCK", 100, eta=None)
    shipment_batch = Batch("shipment-batch", "RETRO-CLOCK", 100, eta=tomorrow)
    line = OrderLine("oref", "RETRO-CLOCK", 10)

    allocate(line, [in_stock_batch, shipment_batch])

    assert in_stock_batch.available_quantity == 90
    assert shipment_batch.available_quantity == 100


def test_prefers_earlier_batches():
    earliest = Batch("speedy-batch", "MINIMALIST-SPOON", 100, eta=today)
    medium = Batch("normal-batch", "MINIMALIST-SPOON", 100, eta=tomorrow)
    latest = Batch("slow-batch", "MINIMALIST-SPOON", 100, eta=later)
    line = OrderLine("order1", "MINIMALIST-SPOON", 10)

    allocate(line, [medium, earliest, latest])

    assert earliest.available_quantity == 90
```

6 Domain services are not the same thing as the services from the service layer, although they are often closely related. A domain service represents a business concept or process, whereas a service-layer service represents a use case for your application. Often the service layer will call a domain service.

```
    assert medium.available_quantity == 100
    assert latest.available_quantity == 100


def test_returns_allocated_batch_ref():
    in_stock_batch = Batch("in-stock-batch-ref", "HIGHBROW-POSTER", 100, eta=None)
    shipment_batch = Batch("shipment-batch-ref", "HIGHBROW-POSTER", 100, eta=tomorrow)
    line = OrderLine("oref", "HIGHBROW-POSTER", 10)
    allocation = allocate(line, [in_stock_batch, shipment_batch])
    assert allocation == in_stock_batch.reference
```

And our service might look like this:

A standalone function for our domain service (model.py)

```
def allocate(line: OrderLine, batches: List[Batch]) -> str:
    batch = next(
        b for b in sorted(batches) if b.can_allocate(line)
    )
    batch.allocate(line)
    return batch.reference
```

Python's Magic Methods Let Us Use Our Models with Idiomatic Python

You may or may not like the use of `next()` in the preceding code, but we're pretty sure you'll agree that being able to use `sorted()` on our list of batches is nice, idiomatic Python.

To make it work, we implement `__gt__` on our domain model:

Magic methods can express domain semantics (model.py)

```
class Batch:
    ...

    def __gt__(self, other):
        if self.eta is None:
            return False
        if other.eta is None:
            return True
        return self.eta > other.eta
```

That's lovely.

Exceptions Can Express Domain Concepts Too

We have one final concept to cover: exceptions can be used to express domain concepts too. In our conversations with domain experts, we've learned about the possibility that an order cannot be allocated because we are *out of stock*, and we can capture that by using a *domain exception*:

Testing out-of-stock exception (test_allocate.py)

```
def test_raises_out_of_stock_exception_if_cannot_allocate():
    batch = Batch('batch1', 'SMALL-FORK', 10, eta=today)
    allocate(OrderLine('order1', 'SMALL-FORK', 10), [batch])

    with pytest.raises(OutOfStock, match='SMALL-FORK'):
        allocate(OrderLine('order2', 'SMALL-FORK', 1), [batch])
```

Domain Modeling Recap

Domain modeling
: This is the part of your code that is closest to the business, the most likely to change, and the place where you deliver the most value to the business. Make it easy to understand and modify.

Distinguish entities from value objects
: A value object is defined by its attributes. It's usually best implemented as an immutable type. If you change an attribute on a Value Object, it represents a different object. In contrast, an entity has attributes that may vary over time and it will still be the same entity. It's important to define what *does* uniquely identify an entity (usually some sort of name or reference field).

Not everything has to be an object
: Python is a multiparadigm language, so let the "verbs" in your code be functions. For every `FooManager`, `BarBuilder`, or `BazFactory`, there's often a more expressive and readable `manage_foo()`, `build_bar()`, or `get_baz()` waiting to happen.

This is the time to apply your best OO design principles
: Revisit the SOLID principles and all the other good heuristics like "has a versus is-a," "prefer composition over inheritance," and so on.

You'll also want to think about consistency boundaries and aggregates
: But that's a topic for Chapter 7.

We won't bore you too much with the implementation, but the main thing to note is that we take care in naming our exceptions in the ubiquitous language, just as we do our entities, value objects, and services:

Raising a domain exception (model.py)

```
class OutOfStock(Exception):
    pass


def allocate(line: OrderLine, batches: List[Batch]) -> str:
    try:
        batch = next(
        ...
    except StopIteration:
        raise OutOfStock(f'Out of stock for sku {line.sku}')
```

Figure 1-4 is a visual representation of where we've ended up.

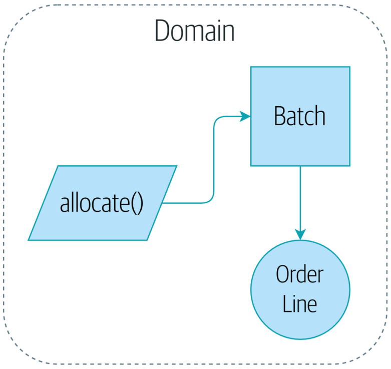


Figure 1-4. Our domain model at the end of the chapter

That'll probably do for now! We have a domain service that we can use for our first use case. But first we'll need a database…

CHAPTER 2

Repository Pattern

It's time to make good on our promise to use the dependency inversion principle as a way of decoupling our core logic from infrastructural concerns.

We'll introduce the *Repository* pattern, a simplifying abstraction over data storage, allowing us to decouple our model layer from the data layer. We'll present a concrete example of how this simplifying abstraction makes our system more testable by hiding the complexities of the database.

Figure 2-1 shows a little preview of what we're going to build: a `Repository` object that sits between our domain model and the database.

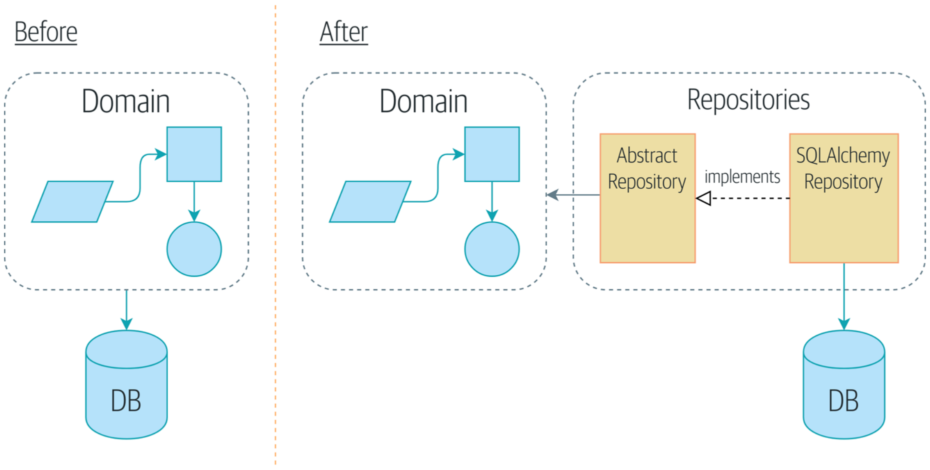


Figure 2-1. Before and after the Repository pattern

The code for this chapter is in the chapter_02_repository branch on GitHub.

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_02_repository
# or to code along, checkout the previous chapter:
git checkout chapter_01_domain_model
```

Persisting Our Domain Model

In Chapter 1 we built a simple domain model that can allocate orders to batches of stock. It's easy for us to write tests against this code because there aren't any dependencies or infrastructure to set up. If we needed to run a database or an API and create test data, our tests would be harder to write and maintain.

Sadly, at some point we'll need to put our perfect little model in the hands of users and contend with the real world of spreadsheets and web browsers and race conditions. For the next few chapters we're going to look at how we can connect our idealized domain model to external state.

We expect to be working in an agile manner, so our priority is to get to a minimum viable product as quickly as possible. In our case, that's going to be a web API. In a real project, you might dive straight in with some end-to-end tests and start plugging in a web framework, test-driving things outside-in.

But we know that, no matter what, we're going to need some form of persistent storage, and this is a textbook, so we can allow ourselves a tiny bit more bottom-up development and start to think about storage and databases.

Some Pseudocode: What Are We Going to Need?

When we build our first API endpoint, we know we're going to have some code that looks more or less like the following.

What our first API endpoint will look like

```
@flask.route.gubbins
def allocate_endpoint():
    # extract order line from request
    line = OrderLine(request.params, ...)
    # load all batches from the DB
    batches = ...
    # call our domain service
    allocate(line, batches)
    # then save the allocation back to the database somehow
    return 201
```

We've used Flask because it's lightweight, but you don't need to be a Flask user to understand this book. In fact, we'll show you how to make your choice of framework a minor detail.

We'll need a way to retrieve batch info from the database and instantiate our domain model objects from it, and we'll also need a way of saving them back to the database.

What? Oh, "gubbins" is a British word for "stuff." You can just ignore that. It's pseudocode, OK?

Applying the DIP to Data Access

As mentioned in the introduction, a layered architecture is a common approach to structuring a system that has a UI, some logic, and a database (see Figure 2-2).

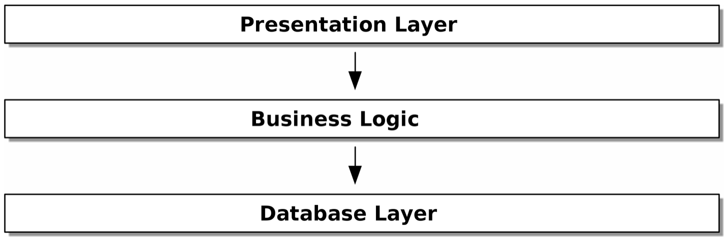


Figure 2-2. Layered architecture

Django's Model-View-Template structure is closely related, as is Model-View-Controller (MVC). In any case, the aim is to keep the layers separate (which is a good thing), and to have each layer depend only on the one below it.

But we want our domain model to have *no dependencies whatsoever*.[1] We don't want infrastructure concerns bleeding over into our domain model and slowing our unit tests or our ability to make changes.

Instead, as discussed in the introduction, we'll think of our model as being on the "inside," and dependencies flowing inward to it; this is what people sometimes call *onion architecture* (see Figure 2-3).

1 I suppose we mean "no stateful dependencies." Depending on a helper library is fine; depending on an ORM or a web framework is not.

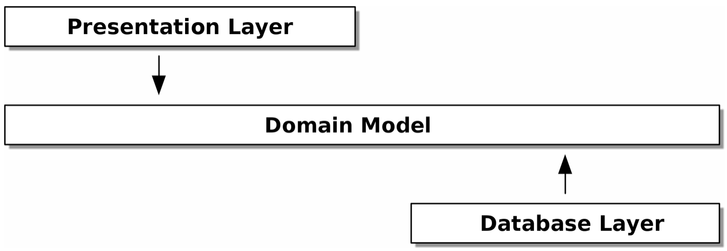


Figure 2-3. Onion architecture

Is This Ports and Adapters?

If you've been reading about architectural patterns, you may be asking yourself questions like this:

> *Is this ports and adapters? Or is it hexagonal architecture? Is that the same as onion architecture? What about the clean architecture? What's a port, and what's an adapter? Why do you people have so many words for the same thing?*

Although some people like to nitpick over the differences, all these are pretty much names for the same thing, and they all boil down to the dependency inversion principle: high-level modules (the domain) should not depend on low-level ones (the infrastructure).[2]

We'll get into some of the nitty-gritty around "depending on abstractions," and whether there is a Pythonic equivalent of interfaces, later in the book. See also "What Is a Port and What Is an Adapter, in Python?" on page 37.

Reminder: Our Model

Let's remind ourselves of our domain model (see Figure 2-4): an allocation is the concept of linking an `OrderLine` to a `Batch`. We're storing the allocations as a collection on our `Batch` object.

2 Mark Seemann has an excellent blog post on the topic.

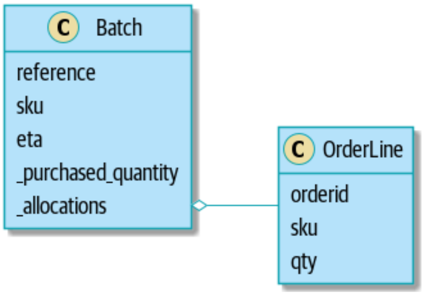


Figure 2-4. Our model

Let's see how we might translate this to a relational database.

The "Normal" ORM Way: Model Depends on ORM

These days, it's unlikely that your team members are hand-rolling their own SQL queries. Instead, you're almost certainly using some kind of framework to generate SQL for you based on your model objects.

These frameworks are called *object-relational mappers* (ORMs) because they exist to bridge the conceptual gap between the world of objects and domain modeling and the world of databases and relational algebra.

The most important thing an ORM gives us is *persistence ignorance*: the idea that our fancy domain model doesn't need to know anything about how data is loaded or persisted. This helps keep our domain clean of direct dependencies on particular database technologies.[3]

But if you follow the typical SQLAlchemy tutorial, you'll end up with something like this:

SQLAlchemy "declarative" syntax, model depends on ORM (orm.py)

```
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship

Base = declarative_base()

class Order(Base):
    id = Column(Integer, primary_key=True)

class OrderLine(Base):
```

3 In this sense, using an ORM is already an example of the DIP. Instead of depending on hardcoded SQL, we depend on an abstraction, the ORM. But that's not enough for us—not in this book!

```
    id = Column(Integer, primary_key=True)
    sku = Column(String(250))
    qty = Integer(String(250))
    order_id = Column(Integer, ForeignKey('order.id'))
    order = relationship(Order)

class Allocation(Base):
    ...
```

You don't need to understand SQLAlchemy to see that our pristine model is now full of dependencies on the ORM and is starting to look ugly as hell besides. Can we really say this model is ignorant of the database? How can it be separate from storage concerns when our model properties are directly coupled to database columns?

Django's ORM Is Essentially the Same, but More Restrictive

If you're more used to Django, the preceding "declarative" SQLAlchemy snippet translates to something like this:

Django ORM example

```
class Order(models.Model):
    pass

class OrderLine(models.Model):
    sku = models.CharField(max_length=255)
    qty = models.IntegerField()
    order = models.ForeignKey(Order)

class Allocation(models.Model):
    ...
```

The point is the same—our model classes inherit directly from ORM classes, so our model depends on the ORM. We want it to be the other way around.

Django doesn't provide an equivalent for SQLAlchemy's classical mapper, but see Appendix D for examples of how to apply dependency inversion and the Repository pattern to Django.

Inverting the Dependency: ORM Depends on Model

Well, thankfully, that's not the only way to use SQLAlchemy. The alternative is to define your schema separately, and to define an explicit *mapper* for how to convert between the schema and our domain model, what SQLAlchemy calls a classical mapping:

Explicit ORM mapping with SQLAlchemy Table objects (orm.py)

```
from sqlalchemy.orm import mapper, relationship

import model  ❶


metadata = MetaData()

order_lines = Table(  ❷
    'order_lines', metadata,
    Column('id', Integer, primary_key=True, autoincrement=True),
    Column('sku', String(255)),
    Column('qty', Integer, nullable=False),
    Column('orderid', String(255)),
)

...

def start_mappers():
    lines_mapper = mapper(model.OrderLine, order_lines)  ❸
```

❶ The ORM imports (or "depends on" or "knows about") the domain model, and not the other way around.

❷ We define our database tables and columns by using SQLAlchemy's abstractions.[4]

❸ When we call the `mapper` function, SQLAlchemy does its magic to bind our domain model classes to the various tables we've defined.

The end result will be that, if we call `start_mappers`, we will be able to easily load and save domain model instances from and to the database. But if we never call that function, our domain model classes stay blissfully unaware of the database.

This gives us all the benefits of SQLAlchemy, including the ability to use `alembic` for migrations, and the ability to transparently query using our domain classes, as we'll see.

When you're first trying to build your ORM config, it can be useful to write tests for it, as in the following example:

Testing the ORM directly (throwaway tests) (test_orm.py)

```
def test_orderline_mapper_can_load_lines(session):  ❶
    session.execute(
        'INSERT INTO order_lines (orderid, sku, qty) VALUES '
```

4 Even in projects where we don't use an ORM, we often use SQLAlchemy alongside Alembic to declaratively create schemas in Python and to manage migrations, connections, and sessions.

```
        '("order1", "RED-CHAIR", 12),'
        '("order1", "RED-TABLE", 13),'
        '("order2", "BLUE-LIPSTICK", 14)'
    )
    expected = [
        model.OrderLine("order1", "RED-CHAIR", 12),
        model.OrderLine("order1", "RED-TABLE", 13),
        model.OrderLine("order2", "BLUE-LIPSTICK", 14),
    ]
    assert session.query(model.OrderLine).all() == expected


def test_orderline_mapper_can_save_lines(session):
    new_line = model.OrderLine("order1", "DECORATIVE-WIDGET", 12)
    session.add(new_line)
    session.commit()

    rows = list(session.execute('SELECT orderid, sku, qty FROM "order_lines"'))
    assert rows == [("order1", "DECORATIVE-WIDGET", 12)]
```

❶ If you haven't used pytest, the `session` argument to this test needs explaining. You don't need to worry about the details of pytest or its fixtures for the purposes of this book, but the short explanation is that you can define common dependencies for your tests as "fixtures," and pytest will inject them to the tests that need them by looking at their function arguments. In this case, it's a SQLAlchemy database session.

You probably wouldn't keep these tests around—as you'll see shortly, once you've taken the step of inverting the dependency of ORM and domain model, it's only a small additional step to implement another abstraction called the Repository pattern, which will be easier to write tests against and will provide a simple interface for faking out later in tests.

But we've already achieved our objective of inverting the traditional dependency: the domain model stays "pure" and free from infrastructure concerns. We could throw away SQLAlchemy and use a different ORM, or a totally different persistence system, and the domain model doesn't need to change at all.

Depending on what you're doing in your domain model, and especially if you stray far from the OO paradigm, you may find it increasingly hard to get the ORM to produce the exact behavior you need, and you may need to modify your domain model.[5] As so often happens with architectural decisions, you'll need to consider a trade-off. As the Zen of Python says, "Practicality beats purity!"

5 Shout-out to the amazingly helpful SQLAlchemy maintainers, and to Mike Bayer in particular.

At this point, though, our API endpoint might look something like the following, and we could get it to work just fine:

Using SQLAlchemy directly in our API endpoint

```
@flask.route.gubbins
def allocate_endpoint():
    session = start_session()

    # extract order line from request
    line = OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

    # load all batches from the DB
    batches = session.query(Batch).all()

    # call our domain service
    allocate(line, batches)

    # save the allocation back to the database
    session.commit()

    return 201
```

Introducing the Repository Pattern

The *Repository* pattern is an abstraction over persistent storage. It hides the boring details of data access by pretending that all of our data is in memory.

If we had infinite memory in our laptops, we'd have no need for clumsy databases. Instead, we could just use our objects whenever we liked. What would that look like?

You have to get your data from somewhere

```
import all_my_data

def create_a_batch():
    batch = Batch(...)
    all_my_data.batches.add(batch)

def modify_a_batch(batch_id, new_quantity):
    batch = all_my_data.batches.get(batch_id)
    batch.change_initial_quantity(new_quantity)
```

Even though our objects are in memory, we need to put them *somewhere* so we can find them again. Our in-memory data would let us add new objects, just like a list or a set. Because the objects are in memory, we never need to call a `.save()` method; we just fetch the object we care about and modify it in memory.

The Repository in the Abstract

The simplest repository has just two methods: `add()` to put a new item in the repository, and `get()` to return a previously added item.[6] We stick rigidly to using these methods for data access in our domain and our service layer. This self-imposed simplicity stops us from coupling our domain model to the database.

Here's what an abstract base class (ABC) for our repository would look like:

The simplest possible repository (repository.py)

```
class AbstractRepository(abc.ABC):

    @abc.abstractmethod  ❶
    def add(self, batch: model.Batch):
        raise NotImplementedError  ❷

    @abc.abstractmethod
    def get(self, reference) -> model.Batch:
        raise NotImplementedError
```

❶ Python tip: `@abc.abstractmethod` is one of the only things that makes ABCs actually "work" in Python. Python will refuse to let you instantiate a class that does not implement all the `abstractmethods` defined in its parent class.[7]

❷ `raise NotImplementedError` is nice, but it's neither necessary nor sufficient. In fact, your abstract methods can have real behavior that subclasses can call out to, if you really want.

6 You may be thinking, "What about `list` or `delete` or `update`?" However, in an ideal world, we modify our model objects one at a time, and delete is usually handled as a soft-delete—i.e., `batch.cancel()`. Finally, update is taken care of by the Unit of Work pattern, as you'll see in Chapter 6.

7 To really reap the benefits of ABCs (such as they may be), be running helpers like `pylint` and `mypy`.

Abstract Base Classes, Duck Typing, and Protocols

We're using abstract base classes in this book for didactic reasons: we hope they help explain what the interface of the repository abstraction is.

In real life, we've sometimes found ourselves deleting ABCs from our production code, because Python makes it too easy to ignore them, and they end up unmaintained and, at worst, misleading. In practice we often just rely on Python's duck typing to enable abstractions. To a Pythonista, a repository is *any* object that has `add(thing)` and `get(id)` methods.

An alternative to look into is PEP 544 protocols. These give you typing without the possibility of inheritance, which "prefer composition over inheritance" fans will particularly like.

What Is the Trade-Off?

> You know they say economists know the price of everything and the value of nothing? Well, programmers know the benefits of everything and the trade-offs of nothing.
>
> —Rich Hickey

Whenever we introduce an architectural pattern in this book, we'll always ask, "What do we get for this? And what does it cost us?"

Usually, at the very least, we'll be introducing an extra layer of abstraction, and although we may hope it will reduce complexity overall, it does add complexity locally, and it has a cost in terms of the raw numbers of moving parts and ongoing maintenance.

The Repository pattern is probably one of the easiest choices in the book, though, if you're already heading down the DDD and dependency inversion route. As far as our code is concerned, we're really just swapping the SQLAlchemy abstraction (`session.query(Batch)`) for a different one (`batches_repo.get`) that we designed.

We will have to write a few lines of code in our repository class each time we add a new domain object that we want to retrieve, but in return we get a simple abstraction over our storage layer, which we control. The Repository pattern would make it easy to make fundamental changes to the way we store things (see Appendix C), and as we'll see, it is easy to fake out for unit tests.

In addition, the Repository pattern is so common in the DDD world that, if you do collaborate with programmers who have come to Python from the Java and C# worlds, they're likely to recognize it. Figure 2-5 illustrates the pattern.

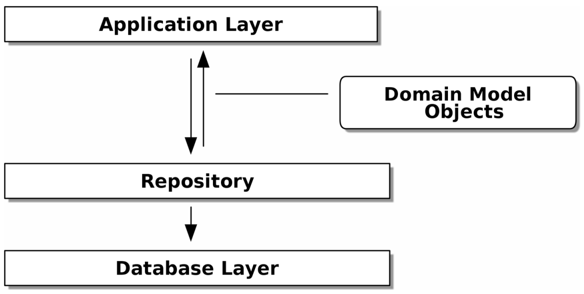


Figure 2-5. Repository pattern

As always, we start with a test. This would probably be classified as an integration test, since we're checking that our code (the repository) is correctly integrated with the database; hence, the tests tend to mix raw SQL with calls and assertions on our own code.

Unlike the ORM tests from earlier, these tests are good candidates for staying part of your codebase longer term, particularly if any parts of your domain model mean the object-relational map is nontrivial.

Repository test for saving an object (test_repository.py)

```
def test_repository_can_save_a_batch(session):
    batch = model.Batch("batch1", "RUSTY-SOAPDISH", 100, eta=None)

    repo = repository.SqlAlchemyRepository(session)
    repo.add(batch)  ❶
    session.commit()  ❷

    rows = list(session.execute(
        'SELECT reference, sku, _purchased_quantity, eta FROM "batches"'  ❸
    ))
    assert rows == [("batch1", "RUSTY-SOAPDISH", 100, None)]
```

❶ `repo.add()` is the method under test here.

❷ We keep the `.commit()` outside of the repository and make it the responsibility of the caller. There are pros and cons for this; some of our reasons will become clearer when we get to Chapter 6.

❸ We use the raw SQL to verify that the right data has been saved.

The next test involves retrieving batches and allocations, so it's more complex:

Repository test for retrieving a complex object (test_repository.py)

```
def insert_order_line(session):
    session.execute(  ❶
        'INSERT INTO order_lines (orderid, sku, qty)'
        ' VALUES ("order1", "GENERIC-SOFA", 12)'
    )
    [[orderline_id]] = session.execute(
        'SELECT id FROM order_lines WHERE orderid=:orderid AND sku=:sku',
        dict(orderid="order1", sku="GENERIC-SOFA")
    )
    return orderline_id

def insert_batch(session, batch_id):  ❷
    ...

def test_repository_can_retrieve_a_batch_with_allocations(session):
    orderline_id = insert_order_line(session)
    batch1_id = insert_batch(session, "batch1")
    insert_batch(session, "batch2")
    insert_allocation(session, orderline_id, batch1_id)  ❸

    repo = repository.SqlAlchemyRepository(session)
    retrieved = repo.get("batch1")

    expected = model.Batch("batch1", "GENERIC-SOFA", 100, eta=None)
    assert retrieved == expected  # Batch.__eq__ only compares reference  ❸
    assert retrieved.sku == expected.sku  ❹
    assert retrieved._purchased_quantity == expected._purchased_quantity
    assert retrieved._allocations == {  ❹
        model.OrderLine("order1", "GENERIC-SOFA", 12),
    }
```

❶ This tests the read side, so the raw SQL is preparing data to be read by the `repo.get()`.

❷ We'll spare you the details of `insert_batch` and `insert_allocation`; the point is to create a couple of batches, and, for the batch we're interested in, to have one existing order line allocated to it.

❸ And that's what we verify here. The first `assert ==` checks that the types match, and that the reference is the same (because, as you remember, `Batch` is an entity, and we have a custom *eq* for it).

❹ So we also explicitly check on its major attributes, including `._allocations`, which is a Python set of `OrderLine` value objects.

Whether or not you painstakingly write tests for every model is a judgment call. Once you have one class tested for create/modify/save, you might be happy to go on and do the others with a minimal round-trip test, or even nothing at all, if they all follow a similar pattern. In our case, the ORM config that sets up the `._allocations` set is a little complex, so it merited a specific test.

You end up with something like this:

A typical repository (repository.py)

```
class SqlAlchemyRepository(AbstractRepository):

    def __init__(self, session):
        self.session = session

    def add(self, batch):
        self.session.add(batch)

    def get(self, reference):
        return self.session.query(model.Batch).filter_by(reference=reference).one()

    def list(self):
        return self.session.query(model.Batch).all()
```

And now our Flask endpoint might look something like the following:

Using our repository directly in our API endpoint

```
@flask.route.gubbins
def allocate_endpoint():
    batches = SqlAlchemyRepository.list()
    lines = [
        OrderLine(l['orderid'], l['sku'], l['qty'])
         for l in request.params...
    ]
    allocate(lines, batches)
    session.commit()
    return 201
```

Exercise for the Reader

We bumped into a friend at a DDD conference the other day who said, "I haven't used an ORM in 10 years." The Repository pattern and an ORM both act as abstractions in front of raw SQL, so using one behind the other isn't really necessary. Why not have a go at implementing our repository without using the ORM? You'll find the code on GitHub.

We've left the repository tests, but figuring out what SQL to write is up to you. Perhaps it'll be harder than you think; perhaps it'll be easier. But the nice thing is, the rest of your application just doesn't care.

Building a Fake Repository for Tests Is Now Trivial!

Here's one of the biggest benefits of the Repository pattern:

A simple fake repository using a set (repository.py)

```python
class FakeRepository(AbstractRepository):

    def __init__(self, batches):
        self._batches = set(batches)

    def add(self, batch):
        self._batches.add(batch)

    def get(self, reference):
        return next(b for b in self._batches if b.reference == reference)

    def list(self):
        return list(self._batches)
```

Because it's a simple wrapper around a `set`, all the methods are one-liners.

Using a fake repo in tests is really easy, and we have a simple abstraction that's easy to use and reason about:

Example usage of fake repository (test_api.py)

```python
fake_repo = FakeRepository([batch1, batch2, batch3])
```

You'll see this fake in action in the next chapter.

> Building fakes for your abstractions is an excellent way to get design feedback: if it's hard to fake, the abstraction is probably too complicated.

What Is a Port and What Is an Adapter, in Python?

We don't want to dwell on the terminology too much here because the main thing we want to focus on is dependency inversion, and the specifics of the technique you use don't matter too much. Also, we're aware that different people use slightly different definitions.

Ports and adapters came out of the OO world, and the definition we hold onto is that the *port* is the *interface* between our application and whatever it is we wish to abstract away, and the *adapter* is the *implementation* behind that interface or abstraction.

Now Python doesn't have interfaces per se, so although it's usually easy to identify an adapter, defining the port can be harder. If you're using an abstract base class, that's the port. If not, the port is just the duck type that your adapters conform to and that your core application expects—the function and method names in use, and their argument names and types.

Concretely, in this chapter, `AbstractRepository` is the port, and `SqlAlchemyRepository` and `FakeRepository` are the adapters.

Wrap-Up

Bearing the Rich Hickey quote in mind, in each chapter we summarize the costs and benefits of each architectural pattern we introduce. We want to be clear that we're not saying every single application needs to be built this way; only sometimes does the complexity of the app and domain make it worth investing the time and effort in adding these extra layers of indirection.

With that in mind, Table 2-1 shows some of the pros and cons of the Repository pattern and our persistence-ignorant model.

Table 2-1. Repository pattern and persistence ignorance: the trade-offs

Pros	Cons
• We have a simple interface between persistent storage and our domain model. • It's easy to make a fake version of the repository for unit testing, or to swap out different storage solutions, because we've fully decoupled the model from infrastructure concerns.	• An ORM already buys you some decoupling. Changing foreign keys might be hard, but it should be pretty easy to swap between MySQL and Postgres if you ever need to.
• Writing the domain model before thinking about persistence helps us focus on the business problem at hand. If we ever want to radically change our approach, we can do that in our model, without needing to worry about foreign keys or migrations until later. • Our database schema is really simple because we have complete control over how we map our objects to tables.	• Maintaining ORM mappings by hand requires extra work and extra code. • Any extra layer of indirection always increases maintenance costs and adds a "WTF factor" for Python programmers who've never seen the Repository pattern before.

Figure 2-6 shows the basic thesis: yes, for simple cases, a decoupled domain model is harder work than a simple ORM/ActiveRecord pattern.[8]

If your app is just a simple CRUD (create-read-update-delete) wrapper around a database, then you don't need a domain model or a repository.

8 Diagram inspired by a post called "Global Complexity, Local Simplicity" by Rob Vens.

But the more complex the domain, the more an investment in freeing yourself from infrastructure concerns will pay off in terms of the ease of making changes.

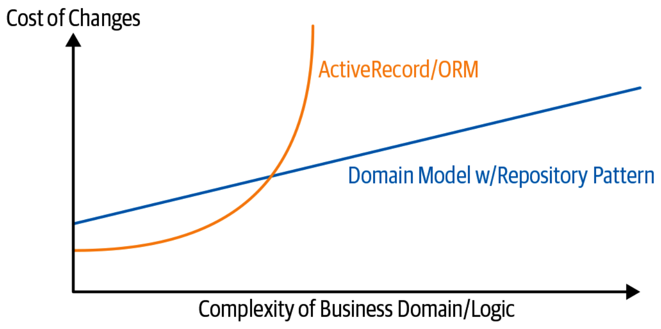


Figure 2-6. Domain model trade-offs as a diagram

Our example code isn't complex enough to give more than a hint of what the right-hand side of the graph looks like, but the hints are there. Imagine, for example, if we decide one day that we want to change allocations to live on the `OrderLine` instead of on the `Batch` object: if we were using Django, say, we'd have to define and think through the database migration before we could run any tests. As it is, because our model is just plain old Python objects, we can change a `set()` to being a new attribute, without needing to think about the database until later.

Repository Pattern Recap

Apply dependency inversion to your ORM
: Our domain model should be free of infrastructure concerns, so your ORM should import your model, and not the other way around.

The Repository pattern is a simple abstraction around permanent storage
: The repository gives you the illusion of a collection of in-memory objects. It makes it easy to create a `FakeRepository` for testing and to swap fundamental details of your infrastructure without disrupting your core application. See Appendix C for an example.

You'll be wondering, how do we instantiate these repositories, fake or real? What will our Flask app actually look like? You'll find out in the next exciting installment, the Service Layer pattern.

But first, a brief digression.

CHAPTER 3

A Brief Interlude: On Coupling and Abstractions

Allow us a brief digression on the subject of abstractions, dear reader. We've talked about *abstractions* quite a lot. The Repository pattern is an abstraction over permanent storage, for example. But what makes a good abstraction? What do we want from abstractions? And how do they relate to testing?

The code for this chapter is in the chapter_03_abstractions branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
git checkout chapter_03_abstractions
```

A key theme in this book, hidden among the fancy patterns, is that we can use simple abstractions to hide messy details. When we're writing code for fun, or in a kata,[1] we get to play with ideas freely, hammering things out and refactoring aggressively. In a large-scale system, though, we become constrained by the decisions made elsewhere in the system.

When we're unable to change component A for fear of breaking component B, we say that the components have become *coupled*. Locally, coupling is a good thing: it's a sign that our code is working together, each component supporting the others, all of them fitting in place like the gears of a watch. In jargon, we say this works when there is high *cohesion* between the coupled elements.

1 A code kata is a small, contained programming challenge often used to practice TDD. See "Kata—The Only Way to Learn TDD" by Peter Provost.

Globally, coupling is a nuisance: it increases the risk and the cost of changing our code, sometimes to the point where we feel unable to make any changes at all. This is the problem with the Ball of Mud pattern: as the application grows, if we're unable to prevent coupling between elements that have no cohesion, that coupling increases superlinearly until we are no longer able to effectively change our systems.

We can reduce the degree of coupling within a system (Figure 3-1) by abstracting away the details (Figure 3-2).

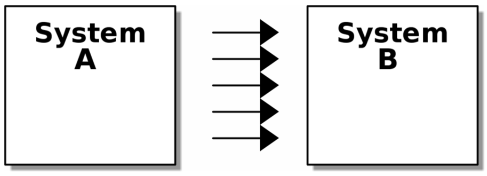


Figure 3-1. Lots of coupling

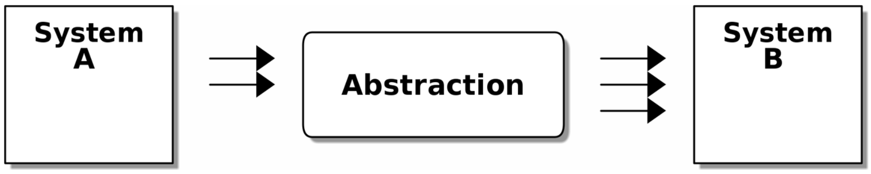


Figure 3-2. Less coupling

In both diagrams, we have a pair of subsystems, with one dependent on the other. In Figure 3-1, there is a high degree of coupling between the two; the number of arrows indicates lots of kinds of dependencies between the two. If we need to change system B, there's a good chance that the change will ripple through to system A.

In Figure 3-2, though, we have reduced the degree of coupling by inserting a new, simpler abstraction. Because it is simpler, system A has fewer kinds of dependencies on the abstraction. The abstraction serves to protect us from change by hiding away the complex details of whatever system B does—we can change the arrows on the right without changing the ones on the left.

Abstracting State Aids Testability

Let's see an example. Imagine we want to write code for synchronizing two file directories, which we'll call the *source* and the *destination*:

- If a file exists in the source but not in the destination, copy the file over.
- If a file exists in the source, but it has a different name than in the destination, rename the destination file to match.
- If a file exists in the destination but not in the source, remove it.

Our first and third requirements are simple enough: we can just compare two lists of paths. Our second is trickier, though. To detect renames, we'll have to inspect the content of files. For this, we can use a hashing function like MD5 or SHA-1. The code to generate a SHA-1 hash from a file is simple enough:

Hashing a file (sync.py)

```python
BLOCKSIZE = 65536

def hash_file(path):
    hasher = hashlib.sha1()
    with path.open("rb") as file:
        buf = file.read(BLOCKSIZE)
        while buf:
            hasher.update(buf)
            buf = file.read(BLOCKSIZE)
    return hasher.hexdigest()
```

Now we need to write the bit that makes decisions about what to do—the business logic, if you will.

When we have to tackle a problem from first principles, we usually try to write a simple implementation and then refactor toward better design. We'll use this approach throughout the book, because it's how we write code in the real world: start with a solution to the smallest part of the problem, and then iteratively make the solution richer and better designed.

Our first hackish approach looks something like this:

Basic sync algorithm (sync.py)

```python
import hashlib
import os
import shutil
from pathlib import Path

def sync(source, dest):
    # Walk the source folder and build a dict of filenames and their hashes
    source_hashes = {}
    for folder, _, files in os.walk(source):
        for fn in files:
            source_hashes[hash_file(Path(folder) / fn)] = fn

    seen = set()  # Keep track of the files we've found in the target

    # Walk the target folder and get the filenames and hashes
    for folder, _, files in os.walk(dest):
        for fn in files:
            dest_path = Path(folder) / fn
            dest_hash = hash_file(dest_path)
            seen.add(dest_hash)

            # if there's a file in target that's not in source, delete it
            if dest_hash not in source_hashes:
                dest_path.remove()

            # if there's a file in target that has a different path in source,
            # move it to the correct path
            elif dest_hash in source_hashes and fn != source_hashes[dest_hash]:
                shutil.move(dest_path, Path(folder) / source_hashes[dest_hash])

    # for every file that appears in source but not target, copy the file to
    # the target
    for src_hash, fn in source_hashes.items():
        if src_hash not in seen:
            shutil.copy(Path(source) / fn, Path(dest) / fn)
```

Fantastic! We have some code and it *looks* OK, but before we run it on our hard drive, maybe we should test it. How do we go about testing this sort of thing?

Some end-to-end tests (test_sync.py)

```
def test_when_a_file_exists_in_the_source_but_not_the_destination():
    try:
        source = tempfile.mkdtemp()
        dest = tempfile.mkdtemp()

        content = "I am a very useful file"
        (Path(source) / 'my-file').write_text(content)

        sync(source, dest)

        expected_path = Path(dest) /  'my-file'
        assert expected_path.exists()
        assert expected_path.read_text() == content

    finally:
        shutil.rmtree(source)
        shutil.rmtree(dest)
```

```
def test_when_a_file_has_been_renamed_in_the_source():
    try:
        source = tempfile.mkdtemp()
        dest = tempfile.mkdtemp()

        content = "I am a file that was renamed"
        source_path = Path(source) / 'source-filename'
        old_dest_path = Path(dest) / 'dest-filename'
        expected_dest_path = Path(dest) / 'source-filename'
        source_path.write_text(content)
        old_dest_path.write_text(content)

        sync(source, dest)

        assert old_dest_path.exists() is False
        assert expected_dest_path.read_text() == content


    finally:
        shutil.rmtree(source)
        shutil.rmtree(dest)
```

Wowsers, that's a lot of setup for two simple cases! The problem is that our domain logic, "figure out the difference between two directories," is tightly coupled to the I/O code. We can't run our difference algorithm without calling the `pathlib`, `shutil`, and `hashlib` modules.

And the trouble is, even with our current requirements, we haven't written enough tests: the current implementation has several bugs (the `shutil.move()` is wrong, for example). Getting decent coverage and revealing these bugs means writing more tests, but if they're all as unwieldy as the preceding ones, that's going to get real painful real quickly.

On top of that, our code isn't very extensible. Imagine trying to implement a `--dry-run` flag that gets our code to just print out what it's going to do, rather than actually do it. Or what if we wanted to sync to a remote server, or to cloud storage?

Our high-level code is coupled to low-level details, and it's making life hard. As the scenarios we consider get more complex, our tests will get more unwieldy. We can definitely refactor these tests (some of the cleanup could go into pytest fixtures, for example) but as long as we're doing filesystem operations, they're going to stay slow and be hard to read and write.

Choosing the Right Abstraction(s)

What could we do to rewrite our code to make it more testable?

First, we need to think about what our code needs from the filesystem. Reading through the code, we can see that three distinct things are happening. We can think of these as three distinct *responsibilities* that the code has:

1. We interrogate the filesystem by using `os.walk` and determine hashes for a series of paths. This is similar in both the source and the destination cases.
2. We decide whether a file is new, renamed, or redundant.
3. We copy, move, or delete files to match the source.

Remember that we want to find *simplifying abstractions* for each of these responsibilities. That will let us hide the messy details so we can focus on the interesting logic.[2]

> In this chapter, we're refactoring some gnarly code into a more testable structure by identifying the separate tasks that need to be done and giving each task to a clearly defined actor, along similar lines to the duckduckgo example.

For steps 1 and 2, we've already intuitively started using an abstraction, a dictionary of hashes to paths. You may already have been thinking, "Why not build up a dictionary for the destination folder as well as the source, and then we just compare two dicts?" That seems like a nice way to abstract the current state of the filesystem:

```
source_files = {'hash1': 'path1', 'hash2': 'path2'}
dest_files = {'hash1': 'path1', 'hash2': 'pathX'}
```

What about moving from step 2 to step 3? How can we abstract out the actual move/copy/delete filesystem interaction?

We'll apply a trick here that we'll employ on a grand scale later in the book. We're going to separate *what* we want to do from *how* to do it. We're going to make our program output a list of commands that look like this:

```
("COPY", "sourcepath", "destpath"),
("MOVE", "old", "new"),
```

Now we could write tests that just use two filesystem dicts as inputs, and we would expect lists of tuples of strings representing actions as outputs.

[2] If you're used to thinking in terms of interfaces, that's what we're trying to define here.

Instead of saying, “Given this actual filesystem, when I run my function, check what actions have happened,” we say, “Given this *abstraction* of a filesystem, what *abstraction* of filesystem actions will happen?”

Simplified inputs and outputs in our tests (test_sync.py)

```
    def test_when_a_file_exists_in_the_source_but_not_the_destination():
        src_hashes = {'hash1': 'fn1'}
        dst_hashes = {}
        expected_actions = [('COPY', '/src/fn1', '/dst/fn1')]
        ...

    def test_when_a_file_has_been_renamed_in_the_source():
        src_hashes = {'hash1': 'fn1'}
        dst_hashes = {'hash1': 'fn2'}
        expected_actions == [('MOVE', '/dst/fn2', '/dst/fn1')]
        ...
```

Implementing Our Chosen Abstractions

That’s all very well, but how do we *actually* write those new tests, and how do we change our implementation to make it all work?

Our goal is to isolate the clever part of our system, and to be able to test it thoroughly without needing to set up a real filesystem. We’ll create a “core” of code that has no dependencies on external state and then see how it responds when we give it input from the outside world (this kind of approach was characterized by Gary Bernhardt as Functional Core, Imperative Shell, or FCIS).

Let’s start off by splitting the code to separate the stateful parts from the logic.

And our top-level function will contain almost no logic at all; it’s just an imperative series of steps: gather inputs, call our logic, apply outputs:

Split our code into three (sync.py)

```
def sync(source, dest):
    # imperative shell step 1, gather inputs
    source_hashes = read_paths_and_hashes(source)  ❶
    dest_hashes = read_paths_and_hashes(dest)  ❶

    # step 2: call functional core
    actions = determine_actions(source_hashes, dest_hashes, source, dest)  ❷

    # imperative shell step 3, apply outputs
    for action, *paths in actions:
        if action == 'copy':
            shutil.copyfile(*paths)
        if action == 'move':
            shutil.move(*paths)
        if action == 'delete':
            os.remove(paths[0])
```

❶ Here's the first function we factor out, `read_paths_and_hashes()`, which isolates the I/O part of our application.

❷ Here is where carve out the functional core, the business logic.

The code to build up the dictionary of paths and hashes is now trivially easy to write:

A function that just does I/O (sync.py)

```
def read_paths_and_hashes(root):
    hashes = {}
    for folder, _, files in os.walk(root):
        for fn in files:
            hashes[hash_file(Path(folder) / fn)] = fn
    return hashes
```

The `determine_actions()` function will be the core of our business logic, which says, "Given these two sets of hashes and filenames, what should we copy/move/delete?". It takes simple data structures and returns simple data structures:

A function that just does business logic (sync.py)

```
def determine_actions(src_hashes, dst_hashes, src_folder, dst_folder):
    for sha, filename in src_hashes.items():
        if sha not in dst_hashes:
            sourcepath = Path(src_folder) / filename
            destpath = Path(dst_folder) / filename
            yield 'copy', sourcepath, destpath

        elif dst_hashes[sha] != filename:
            olddestpath = Path(dst_folder) / dst_hashes[sha]
            newdestpath = Path(dst_folder) / filename
            yield 'move', olddestpath, newdestpath

    for sha, filename in dst_hashes.items():
        if sha not in src_hashes:
            yield 'delete', dst_folder / filename
```

Our tests now act directly on the `determine_actions()` function:

Nicer-looking tests (test_sync.py)

```
def test_when_a_file_exists_in_the_source_but_not_the_destination():
    src_hashes = {'hash1': 'fn1'}
    dst_hashes = {}
    actions = determine_actions(src_hashes, dst_hashes, Path('/src'), Path('/dst'))
    assert list(actions) == [('copy', Path('/src/fn1'), Path('/dst/fn1'))]
...
```

```
def test_when_a_file_has_been_renamed_in_the_source():
    src_hashes = {'hash1': 'fn1'}
    dst_hashes = {'hash1': 'fn2'}
    actions = determine_actions(src_hashes, dst_hashes, Path('/src'), Path('/dst'))
    assert list(actions) == [('move', Path('/dst/fn2'), Path('/dst/fn1'))]
```

Because we've disentangled the logic of our program—the code for identifying changes—from the low-level details of I/O, we can easily test the core of our code.

With this approach, we've switched from testing our main entrypoint function, `sync()`, to testing a lower-level function, `determine_actions()`. You might decide that's fine because `sync()` is now so simple. Or you might decide to keep some integration/acceptance tests to test that `sync()`. But there's another option, which is to modify the `sync()` function so it can be unit tested *and* end-to-end tested; it's an approach Bob calls *edge-to-edge testing*.

Testing Edge to Edge with Fakes and Dependency Injection

When we start writing a new system, we often focus on the core logic first, driving it with direct unit tests. At some point, though, we want to test bigger chunks of the system together.

We *could* return to our end-to-end tests, but those are still as tricky to write and maintain as before. Instead, we often write tests that invoke a whole system together but fake the I/O, sort of *edge to edge*:

Explicit dependencies (sync.py)

```
def sync(reader, filesystem, source_root, dest_root): (1)

    source_hashes = reader(source_root) (2)
    dest_hashes = reader(dest_root)

    for sha, filename in src_hashes.items():
        if sha not in dest_hashes:
            sourcepath = source_root / filename
            destpath = dest_root / filename
            filesystem.copy(destpath, sourcepath) (3)

        elif dest_hashes[sha] != filename:
            olddestpath = dest_root / dest_hashes[sha]
            newdestpath = dest_root / filename
            filesystem.move(olddestpath, newdestpath)

    for sha, filename in dst_hashes.items():
        if sha not in source_hashes:
            filesystem.delete(dest_root/filename)
```

❶ Our top-level function now exposes two new dependencies, a `reader` and a `file system`.

❷ We invoke the `reader` to produce our files dict.

❸ We invoke the `filesystem` to apply the changes we detect.

Although we're using dependency injection, there is no need to define an abstract base class or any kind of explicit interface. In this book, we often show ABCs because we hope they help you understand what the abstraction is, but they're not necessary. Python's dynamic nature means we can always rely on duck typing.

Tests using DI

```
class FakeFileSystem(list): ❶

    def copy(self, src, dest): ❷
        self.append(('COPY', src, dest))

    def move(self, src, dest):
        self.append(('MOVE', src, dest))

    def delete(self, dest):
        self.append(('DELETE', src, dest))


def test_when_a_file_exists_in_the_source_but_not_the_destination():
    source = {"sha1": "my-file" }
    dest = {}
    filesystem = FakeFileSystem()

    reader = {"/source": source, "/dest": dest}
    synchronise_dirs(reader.pop, filesystem, "/source", "/dest")

    assert filesystem == [("COPY", "/source/my-file", "/dest/my-file")]


def test_when_a_file_has_been_renamed_in_the_source():
    source = {"sha1": "renamed-file" }
    dest = {"sha1": "original-file" }
    filesystem = FakeFileSystem()

    reader = {"/source": source, "/dest": dest}
    synchronise_dirs(reader.pop, filesystem, "/source", "/dest")

    assert filesystem == [("MOVE", "/dest/original-file", "/dest/renamed-file")]
```

❶ Bob *loves* using lists to build simple test doubles, even though his coworkers get mad. It means we can write tests like `assert` *`foo`* `not in database`.

❷ Each method in our `FakeFileSystem` just appends something to the list so we can inspect it later. This is an example of a spy object.

The advantage of this approach is that our tests act on the exact same function that's used by our production code. The disadvantage is that we have to make our stateful components explicit and pass them around. David Heinemeier Hansson, the creator of Ruby on Rails, famously described this as "test-induced design damage."

In either case, we can now work on fixing all the bugs in our implementation; enumerating tests for all the edge cases is now much easier.

Why Not Just Patch It Out?

At this point you may be scratching your head and thinking, "Why don't you just use `mock.patch` and save yourself the effort?""

We avoid using mocks in this book and in our production code too. We're not going to enter into a Holy War, but our instinct is that mocking frameworks, particularly monkeypatching, are a code smell.

Instead, we like to clearly identify the responsibilities in our codebase, and to separate those responsibilities into small, focused objects that are easy to replace with a test double.

> You can see an example in Chapter 8, where we `mock.patch()` out an email-sending module, but eventually we replace that with an explicit bit of dependency injection in Chapter 13.

We have three closely related reasons for our preference:

- Patching out the dependency you're using makes it possible to unit test the code, but it does nothing to improve the design. Using `mock.patch` won't let your code work with a `--dry-run` flag, nor will it help you run against an FTP server. For that, you'll need to introduce abstractions.
- Tests that use mocks *tend* to be more coupled to the implementation details of the codebase. That's because mock tests verify the interactions between things: did we call `shutil.copy` with the right arguments? This coupling between code and test *tends* to make tests more brittle, in our experience.
- Overuse of mocks leads to complicated test suites that fail to explain the code.

Designing for testability really means designing for extensibility. We trade off a little more complexity for a cleaner design that admits novel use cases.

Mocks Versus Fakes; Classic-Style Versus London-School TDD

Here's a short and somewhat simplistic definition of the difference between mocks and fakes:

- Mocks are used to verify *how* something gets used; they have methods like `assert_called_once_with()`. They're associated with London-school TDD.
- Fakes are working implementations of the thing they're replacing, but they're designed for use only in tests. They wouldn't work "in real life"; our in-memory repository is a good example. But you can use them to make assertions about the end state of a system rather than the behaviors along the way, so they're associated with classic-style TDD.

We're slightly conflating mocks with spies and fakes with stubs here, and you can read the long, correct answer in Martin Fowler's classic essay on the subject called "Mocks Aren't Stubs".

It also probably doesn't help that the `MagicMock` objects provided by `unittest.mock` aren't, strictly speaking, mocks; they're spies, if anything. But they're also often used as stubs or dummies. There, we promise we're done with the test double terminology nitpicks now.

What about London-school versus classic-style TDD? You can read more about those two in Martin Fowler's article that we just cited, as well as on the Software Engineering Stack Exchange site, but in this book we're pretty firmly in the classicist camp. We like to build our tests around state both in setup and in assertions, and we like to work at the highest level of abstraction possible rather than doing checks on the behavior of intermediary collaborators.[3]

Read more on this in "On Deciding What Kind of Tests to Write" on page 73.

We view TDD as a design practice first and a testing practice second. The tests act as a record of our design choices and serve to explain the system to us when we return to the code after a long absence.

3 Which is not to say that we think the London school people are wrong. Some insanely smart people work that way. It's just not what we're used to.

Tests that use too many mocks get overwhelmed with setup code that hides the story we care about.

Steve Freeman has a great example of overmocked tests in his talk "Test-Driven Development". You should also check out this PyCon talk, "Mocking and Patching Pitfalls", by our esteemed tech reviewer, Ed Jung, which also addresses mocking and its alternatives. And while we're recommending talks, don't miss Brandon Rhodes talking about "Hoisting Your I/O", which really nicely covers the issues we're talking about, using another simple example.

In this chapter, we've spent a lot of time replacing end-to-end tests with unit tests. That doesn't mean we think you should never use E2E tests! In this book we're showing techniques to get you to a decent test pyramid with as many unit tests as possible, and with the minimum number of E2E tests you need to feel confident. Read on to "Recap: Rules of Thumb for Different Types of Test" on page 79 for more details.

So Which Do We Use In This Book? Functional or Object-Oriented Composition?

Both. Our domain model is entirely free of dependencies and side effects, so that's our functional core. The service layer that we build around it (in Chapter 4) allows us to drive the system edge to edge, and we use dependency injection to provide those services with stateful components, so we can still unit test them.

See Chapter 13 for more exploration of making our dependency injection more explicit and centralized.

Wrap-Up

We'll see this idea come up again and again in the book: we can make our systems easier to test and maintain by simplifying the interface between our business logic and messy I/O. Finding the right abstraction is tricky, but here are a few heuristics and questions to ask yourself:

- Can I choose a familiar Python data structure to represent the state of the messy system and then try to imagine a single function that can return that state?
- Where can I draw a line between my systems, where can I carve out a seam to stick that abstraction in?
- What is a sensible way of dividing things into components with different responsibilities? What implicit concepts can I make explicit?

- What are the dependencies, and what is the core business logic?

Practice makes less imperfect! And now back to our regular programming…

CHAPTER 4

Our First Use Case: Flask API and Service Layer

Back to our allocations project! Figure 4-1 shows the point we reached at the end of Chapter 2, which covered the Repository pattern.

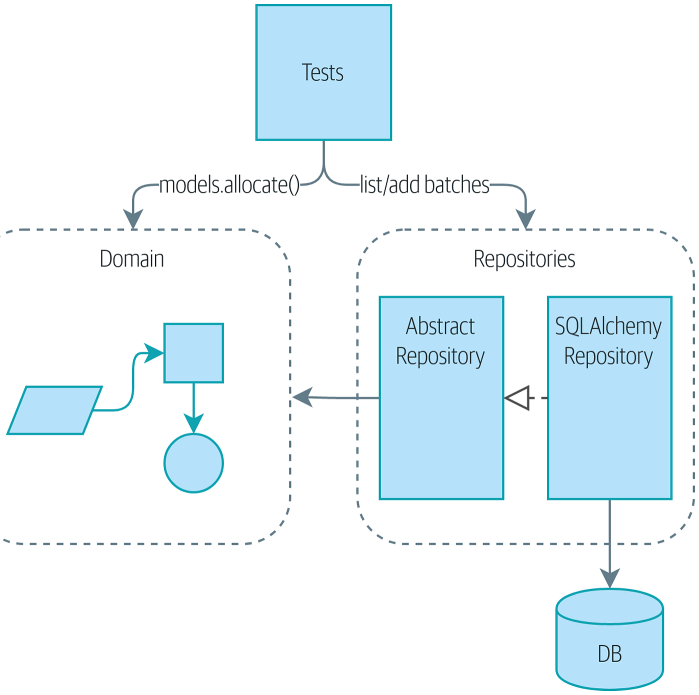


Figure 4-1. Before: we drive our app by talking to repositories and the domain model

In this chapter, we discuss the differences between orchestration logic, business logic, and interfacing code, and we introduce the *Service Layer* pattern to take care of orchestrating our workflows and defining the use cases of our system.

We'll also discuss testing: by combining the Service Layer with our repository abstraction over the database, we're able to write fast tests, not just of our domain model but of the entire workflow for a use case.

Figure 4-2 shows what we're aiming for: we're going to add a Flask API that will talk to the service layer, which will serve as the entrypoint to our domain model. Because our service layer depends on the `AbstractRepository`, we can unit test it by using `FakeRepository` but run our production code using `SqlAlchemyRepository`.

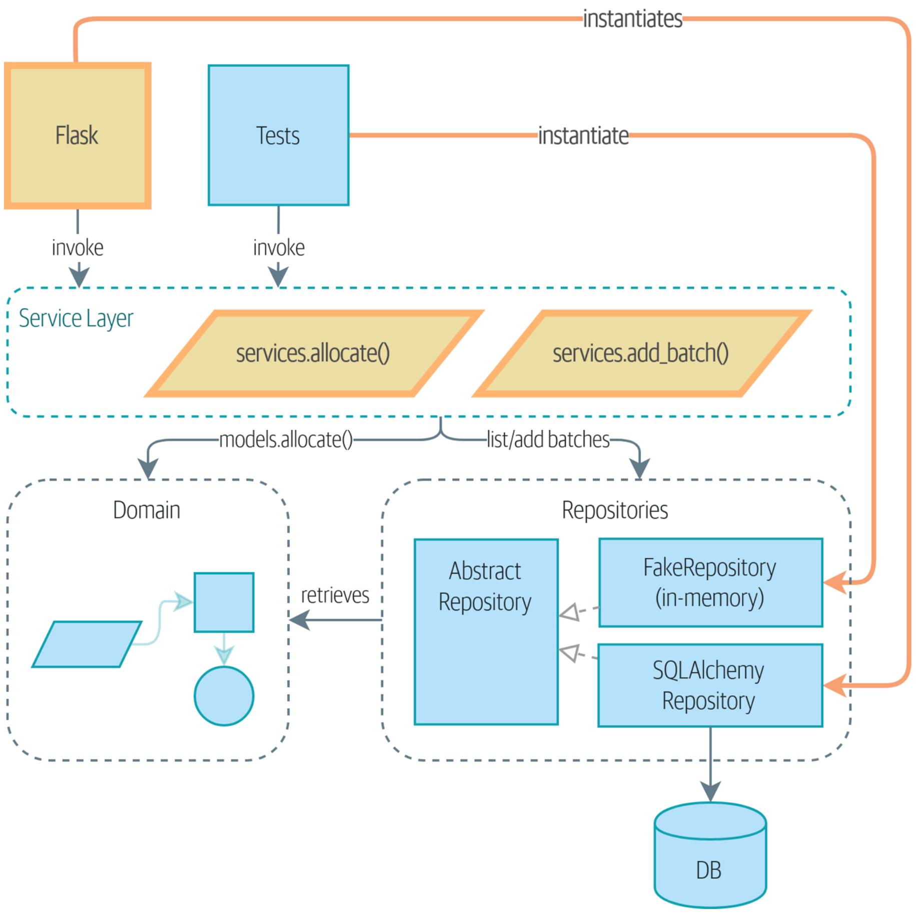


Figure 4-2. The service layer will become the main way into our app

In our diagrams, we are using the convention that new components are highlighted with bold text/lines (and yellow/orange color, if you're reading a digital version).

The code for this chapter is in the chapter_04_service_layer branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_04_service_layer
# or to code along, checkout Chapter 2:
git checkout chapter_02_repository
```

Connecting Our Application to the Real World

Like any good agile team, we're hustling to try to get an MVP out and in front of the users to start gathering feedback. We have the core of our domain model and the domain service we need to allocate orders, and we have the repository interface for permanent storage.

Let's plug all the moving parts together as quickly as we can and then refactor toward a cleaner architecture. Here's our plan:

1. Use Flask to put an API endpoint in front of our `allocate` domain service. Wire up the database session and our repository. Test it with an end-to-end test and some quick-and-dirty SQL to prepare test data.
2. Refactor out a service layer that can serve as an abstraction to capture the use case and that will sit between Flask and our domain model. Build some service-layer tests and show how they can use `FakeRepository`.
3. Experiment with different types of parameters for our service layer functions; show that using primitive data types allows the service layer's clients (our tests and our Flask API) to be decoupled from the model layer.

A First End-to-End Test

No one is interested in getting into a long terminology debate about what counts as an end-to-end (E2E) test versus a functional test versus an acceptance test versus an integration test versus a unit test. Different projects need different combinations of tests, and we've seen perfectly successful projects just split things into "fast tests" and "slow tests."

For now, we want to write one or maybe two tests that are going to exercise a "real" API endpoint (using HTTP) and talk to a real database. Let's call them *end-to-end tests* because it's one of the most self-explanatory names.

The following shows a first cut:

A first API test (test_api.py)

```
@pytest.mark.usefixtures('restart_api')
def test_api_returns_allocation(add_stock):
    sku, othersku = random_sku(), random_sku('other')  ❶
    earlybatch = random_batchref(1)
    laterbatch = random_batchref(2)
    otherbatch = random_batchref(3)
    add_stock([  ❷
        (laterbatch, sku, 100, '2011-01-02'),
        (earlybatch, sku, 100, '2011-01-01'),
        (otherbatch, othersku, 100, None),
    ])
    data = {'orderid': random_orderid(), 'sku': sku, 'qty': 3}
    url = config.get_api_url()  ❸
    r = requests.post(f'{url}/allocate', json=data)
    assert r.status_code == 201
    assert r.json()['batchref'] == earlybatch
```

❶ `random_sku()`, `random_batchref()`, and so on are little helper functions that generate randomized characters by using the `uuid` module. Because we're running against an actual database now, this is one way to prevent various tests and runs from interfering with each other.

❷ `add_stock` is a helper fixture that just hides away the details of manually inserting rows into the database using SQL. We'll show a nicer way of doing this later in the chapter.

❸ *config.py* is a module in which we keep configuration information.

Everyone solves these problems in different ways, but you're going to need some way of spinning up Flask, possibly in a container, and of talking to a Postgres database. If you want to see how we did it, check out Appendix B.

The Straightforward Implementation

Implementing things in the most obvious way, you might get something like this:

First cut of Flask app (flask_app.py)

```
from flask import Flask, jsonify, request
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

import config
import model
import orm
import repository


orm.start_mappers()
get_session = sessionmaker(bind=create_engine(config.get_postgres_uri()))
app = Flask(__name__)

@app.route("/allocate", methods=['POST'])
def allocate_endpoint():
    session = get_session()
    batches = repository.SqlAlchemyRepository(session).list()
    line = model.OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

    batchref = model.allocate(line, batches)

    return jsonify({'batchref': batchref}), 201
```

So far, so good. No need for too much more of your "architecture astronaut" nonsense, Bob and Harry, you may be thinking.

But hang on a minute—there's no commit. We're not actually saving our allocation to the database. Now we need a second test, either one that will inspect the database state after (not very black-boxy), or maybe one that checks that we can't allocate a second line if a first should have already depleted the batch:

Test allocations are persisted (test_api.py)

```
@pytest.mark.usefixtures('restart_api')
def test_allocations_are_persisted(add_stock):
    sku = random_sku()
    batch1, batch2 = random_batchref(1), random_batchref(2)
    order1, order2 = random_orderid(1), random_orderid(2)
    add_stock([
        (batch1, sku, 10, '2011-01-01'),
        (batch2, sku, 10, '2011-01-02'),
    ])
    line1 = {'orderid': order1, 'sku': sku, 'qty': 10}
    line2 = {'orderid': order2, 'sku': sku, 'qty': 10}
    url = config.get_api_url()

    # first order uses up all stock in batch 1
    r = requests.post(f'{url}/allocate', json=line1)
    assert r.status_code == 201
    assert r.json()['batchref'] == batch1

    # second order should go to batch 2
    r = requests.post(f'{url}/allocate', json=line2)
    assert r.status_code == 201
    assert r.json()['batchref'] == batch2
```

Not quite so lovely, but that will force us to add the commit.

Error Conditions That Require Database Checks

If we keep going like this, though, things are going to get uglier and uglier.

Suppose we want to add a bit of error handling. What if the domain raises an error, for a SKU that's out of stock? Or what about a SKU that doesn't even exist? That's not something the domain even knows about, nor should it. It's more of a sanity check that we should implement at the database layer, before we even invoke the domain service.

Now we're looking at two more end-to-end tests:

Yet more tests at the E2E layer (test_api.py)

```
@pytest.mark.usefixtures('restart_api')
def test_400_message_for_out_of_stock(add_stock):  ❶
    sku, smalL_batch, large_order = random_sku(), random_batchref(), random_orderid()
    add_stock([
        (smalL_batch, sku, 10, '2011-01-01'),
    ])
    data = {'orderid': large_order, 'sku': sku, 'qty': 20}
    url = config.get_api_url()
    r = requests.post(f'{url}/allocate', json=data)
    assert r.status_code == 400
    assert r.json()['message'] == f'Out of stock for sku {sku}'


@pytest.mark.usefixtures('restart_api')
def test_400_message_for_invalid_sku():  ❷
    unknown_sku, orderid = random_sku(), random_orderid()
    data = {'orderid': orderid, 'sku': unknown_sku, 'qty': 20}
    url = config.get_api_url()
    r = requests.post(f'{url}/allocate', json=data)
    assert r.status_code == 400
    assert r.json()['message'] == f'Invalid sku {unknown_sku}'
```

❶ In the first test, we're trying to allocate more units than we have in stock.

❷ In the second, the SKU just doesn't exist (because we never called `add_stock`), so it's invalid as far as our app is concerned.

And sure, we could implement it in the Flask app too:

Flask app starting to get crufty (flask_app.py)

```
def is_valid_sku(sku, batches):
    return sku in {b.sku for b in batches}

@app.route("/allocate", methods=['POST'])
def allocate_endpoint():
    session = get_session()
    batches = repository.SqlAlchemyRepository(session).list()
    line = model.OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )

    if not is_valid_sku(line.sku, batches):
        return jsonify({'message': f'Invalid sku {line.sku}'}), 400

    try:
        batchref = model.allocate(line, batches)
    except model.OutOfStock as e:
        return jsonify({'message': str(e)}), 400

    session.commit()
    return jsonify({'batchref': batchref}), 201
```

But our Flask app is starting to look a bit unwieldy. And our number of E2E tests is starting to get out of control, and soon we'll end up with an inverted test pyramid (or "ice-cream cone model," as Bob likes to call it).

Introducing a Service Layer, and Using FakeRepository to Unit Test It

If we look at what our Flask app is doing, there's quite a lot of what we might call *orchestration*—fetching stuff out of our repository, validating our input against database state, handling errors, and committing in the happy path. Most of these things don't have anything to do with having a web API endpoint (you'd need them if you were building a CLI, for example; see Appendix C), and they're not really things that need to be tested by end-to-end tests.

It often makes sense to split out a service layer, sometimes called an *orchestration layer* or a *use-case layer*.

Do you remember the `FakeRepository` that we prepared in Chapter 3?

Our fake repository, an in-memory collection of batches (test_services.py)

```
class FakeRepository(repository.AbstractRepository):

    def __init__(self, batches):
        self._batches = set(batches)

    def add(self, batch):
        self._batches.add(batch)

    def get(self, reference):
        return next(b for b in self._batches if b.reference == reference)

    def list(self):
        return list(self._batches)
```

Here's where it will come in useful; it lets us test our service layer with nice, fast unit tests:

Unit testing with fakes at the service layer (test_services.py)

```
def test_returns_allocation():
    line = model.OrderLine("o1", "COMPLICATED-LAMP", 10)
    batch = model.Batch("b1", "COMPLICATED-LAMP", 100, eta=None)
    repo = FakeRepository([batch])  ❶

    result = services.allocate(line, repo, FakeSession())  ❷❸
    assert result == "b1"


def test_error_for_invalid_sku():
    line = model.OrderLine("o1", "NONEXISTENTSKU", 10)
    batch = model.Batch("b1", "AREALSKU", 100, eta=None)
    repo = FakeRepository([batch])  ❶

    with pytest.raises(services.InvalidSku, match="Invalid sku NONEXISTENTSKU"):
        services.allocate(line, repo, FakeSession())  ❷❸
```

❶ `FakeRepository` holds the `Batch` objects that will be used by our test.

❷ Our services module (*services.py*) will define an `allocate()` service-layer function. It will sit between our `allocate_endpoint()` function in the API layer and the `allocate()` domain service function from our domain model.[1]

❸ We also need a `FakeSession` to fake out the database session, as shown in the following code snippet.

A fake database session (test_services.py)

```
class FakeSession():
    committed = False
```

1 Service-layer services and domain services do have confusingly similar names. We tackle this topic later in "Why Is Everything Called a Service?" on page 66.

```
    def commit(self):
        self.committed = True
```

This fake session is only a temporary solution. We'll get rid of it and make things even nicer soon, in Chapter 6. But in the meantime the fake `.commit()` lets us migrate a third test from the E2E layer:

A second test at the service layer (test_services.py)

```
def test_commits():
    line = model.OrderLine('o1', 'OMINOUS-MIRROR', 10)
    batch = model.Batch('b1', 'OMINOUS-MIRROR', 100, eta=None)
    repo = FakeRepository([batch])
    session = FakeSession()

    services.allocate(line, repo, session)
    assert session.committed is True
```

A Typical Service Function

We'll write a service function that looks something like this:

Basic allocation service (services.py)

```
class InvalidSku(Exception):
    pass


def is_valid_sku(sku, batches):
    return sku in {b.sku for b in batches}

def allocate(line: OrderLine, repo: AbstractRepository, session) -> str:
    batches = repo.list()  ❶
    if not is_valid_sku(line.sku, batches):  ❷
        raise InvalidSku(f'Invalid sku {line.sku}')
    batchref = model.allocate(line, batches)  ❸
    session.commit()  ❹
    return batchref
```

Typical service-layer functions have similar steps:

❶ We fetch some objects from the repository.

❷ We make some checks or assertions about the request against the current state of the world.

❸ We call a domain service.

❹ If all is well, we save/update any state we've changed.

That last step is a little unsatisfactory at the moment, as our service layer is tightly coupled to our database layer. We'll improve that in Chapter 6 with the Unit of Work pattern.

Depend on Abstractions

Notice one more thing about our service-layer function:

```
def allocate(line: OrderLine, repo: AbstractRepository, session) -> str:
```

It depends on a repository. We've chosen to make the dependency explicit, and we've used the type hint to say that we depend on `AbstractRepository`. This means it'll work both when the tests give it a `FakeRepository` and when the Flask app gives it a `SqlAlchemyRepository`.

If you remember "The Dependency Inversion Principle" on page xxi, this is what we mean when we say we should "depend on abstractions." Our *high-level module*, the service layer, depends on the repository abstraction. And the *details* of the implementation for our specific choice of persistent storage also depend on that same abstraction. See Figures 4-3 and 4-4.

See also in Appendix C a worked example of swapping out the *details* of which persistent storage system to use while leaving the abstractions intact.

But the essentials of the service layer are there, and our Flask app now looks a lot cleaner:

Flask app delegating to service layer (flask_app.py)

```
@app.route("/allocate", methods=['POST'])
def allocate_endpoint():
    session = get_session()  ❶
    repo = repository.SqlAlchemyRepository(session)  ❶
    line = model.OrderLine(
        request.json['orderid'],  ❷
        request.json['sku'],  ❷
        request.json['qty'],  ❷
    )
    try:
        batchref = services.allocate(line, repo, session)  ❷
    except (model.OutOfStock, services.InvalidSku) as e:
        return jsonify({'message': str(e)}), 400  ❸

    return jsonify({'batchref': batchref}), 201  ❸
```

❶ We instantiate a database session and some repository objects.

❷ We extract the user's commands from the web request and pass them to a domain service.

❸ We return some JSON responses with the appropriate status codes.

The responsibilities of the Flask app are just standard web stuff: per-request session management, parsing information out of POST parameters, response status codes, and JSON. All the orchestration logic is in the use case/service layer, and the domain logic stays in the domain.

Finally, we can confidently strip down our E2E tests to just two, one for the happy path and one for the unhappy path:

E2E tests only happy and unhappy paths (test_api.py)

```
@pytest.mark.usefixtures('restart_api')
def test_happy_path_returns_201_and_allocated_batch(add_stock):
    sku, othersku = random_sku(), random_sku('other')
    earlybatch = random_batchref(1)
    laterbatch = random_batchref(2)
    otherbatch = random_batchref(3)
    add_stock([
        (laterbatch, sku, 100, '2011-01-02'),
        (earlybatch, sku, 100, '2011-01-01'),
        (otherbatch, othersku, 100, None),
    ])
    data = {'orderid': random_orderid(), 'sku': sku, 'qty': 3}
    url = config.get_api_url()
    r = requests.post(f'{url}/allocate', json=data)
    assert r.status_code == 201
    assert r.json()['batchref'] == earlybatch


@pytest.mark.usefixtures('restart_api')
def test_unhappy_path_returns_400_and_error_message():
    unknown_sku, orderid = random_sku(), random_orderid()
    data = {'orderid': orderid, 'sku': unknown_sku, 'qty': 20}
    url = config.get_api_url()
    r = requests.post(f'{url}/allocate', json=data)
    assert r.status_code == 400
    assert r.json()['message'] == f'Invalid sku {unknown_sku}'
```

We've successfully split our tests into two broad categories: tests about web stuff, which we implement end to end; and tests about orchestration stuff, which we can test against the service layer in memory.

Exercise for the Reader

Now that we have an allocate service, why not build out a service for `deallocate`? We've added an E2E test and a few stub service-layer tests for you to get started on GitHub.

If that's not enough, continue into the E2E tests and *flask_app.py*, and refactor the Flask adapter to be more RESTful. Notice how doing so doesn't require any change to our service layer or domain layer!

If you decide you want to build a read-only endpoint for retrieving allocation info, just do "the simplest thing that can possibly work," which is `repo.get()` right in the Flask handler. We'll talk more about reads versus writes in Chapter 12.

Why Is Everything Called a Service?

Some of you are probably scratching your heads at this point trying to figure out exactly what the difference is between a domain service and a service layer.

We're sorry—we didn't choose the names, or we'd have much cooler and friendlier ways to talk about this stuff.

We're using two things called a *service* in this chapter. The first is an *application service* (our service layer). Its job is to handle requests from the outside world and to *orchestrate* an operation. What we mean is that the service layer *drives* the application by following a bunch of simple steps:

- Get some data from the database
- Update the domain model
- Persist any changes

This is the kind of boring work that has to happen for every operation in your system, and keeping it separate from business logic helps to keep things tidy.

The second type of service is a *domain service*. This is the name for a piece of logic that belongs in the domain model but doesn't sit naturally inside a stateful entity or value object. For example, if you were building a shopping cart application, you might choose to build taxation rules as a domain service. Calculating tax is a separate job from updating the cart, and it's an important part of the model, but it doesn't seem right to have a persisted entity for the job. Instead a stateless TaxCalculator class or a `calculate_tax` function can do the job.

Putting Things in Folders to See Where It All Belongs

As our application gets bigger, we'll need to keep tidying our directory structure. The layout of our project gives us useful hints about what kinds of object we'll find in each file.

Here's one way we could organize things:

Some subfolders

```
.
├── config.py
├── domain  ❶
│   ├── __init__.py
│   └── model.py
├── service_layer  ❷
│   ├── __init__.py
│   └── services.py
├── adapters  ❸
│   ├── __init__.py
│   ├── orm.py
│   └── repository.py
├── entrypoints  ❹
│   ├── __init__.py
│   └── flask_app.py
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── unit
    │   ├── test_allocate.py
    │   ├── test_batches.py
    │   └── test_services.py
    ├── integration
    │   ├── test_orm.py
    │   └── test_repository.py
    └── e2e
        └── test_api.py
```

❶ Let's have a folder for our domain model. Currently that's just one file, but for a more complex application, you might have one file per class; you might have helper parent classes for `Entity`, `ValueObject`, and `Aggregate`, and you might add an *exceptions.py* for domain-layer exceptions and, as you'll see in Part II, *commands.py* and *events.py*.

❷ We'll distinguish the service layer. Currently that's just one file called *services.py* for our service-layer functions. You could add service-layer exceptions here, and as you'll see in Chapter 5, we'll add *unit_of_work.py*.

❸ *Adapters* is a nod to the ports and adapters terminology. This will fill up with any other abstractions around external I/O (e.g., a *redis_client.py*). Strictly speaking, you would call these *secondary* adapters or *driven* adapters, or sometimes *inward-facing* adapters.

❹ Entrypoints are the places we drive our application from. In the official ports and adapters terminology, these are adapters too, and are referred to as *primary*, *driving*, or *outward-facing* adapters.

What about ports? As you may remember, they are the abstract interfaces that the adapters implement. We tend to keep them in the same file as the adapters that implement them.

Wrap-Up

Adding the service layer has really bought us quite a lot:

- Our Flask API endpoints become very thin and easy to write: their only responsibility is doing "web stuff," such as parsing JSON and producing the right HTTP codes for happy or unhappy cases.
- We've defined a clear API for our domain, a set of use cases or entrypoints that can be used by any adapter without needing to know anything about our domain model classes—whether that's an API, a CLI (see Appendix C), or the tests! They're an adapter for our domain too.
- We can write tests in "high gear" by using the service layer, leaving us free to refactor the domain model in any way we see fit. As long as we can still deliver the same use cases, we can experiment with new designs without needing to rewrite a load of tests.
- And our test pyramid is looking good—the bulk of our tests are fast unit tests, with just the bare minimum of E2E and integration tests.

The DIP in Action

Figure 4-3 shows the dependencies of our service layer: the domain model and `AbstractRepository` (the port, in ports and adapters terminology).

When we run the tests, Figure 4-4 shows how we implement the abstract dependencies by using `FakeRepository` (the adapter).

And when we actually run our app, we swap in the "real" dependency shown in Figure 4-5.

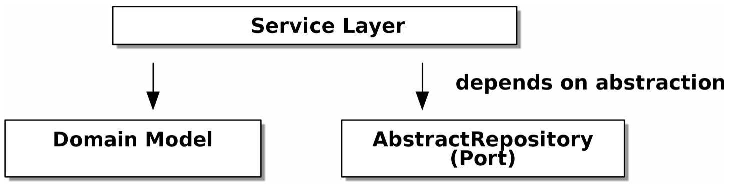


Figure 4-3. Abstract dependencies of the service layer

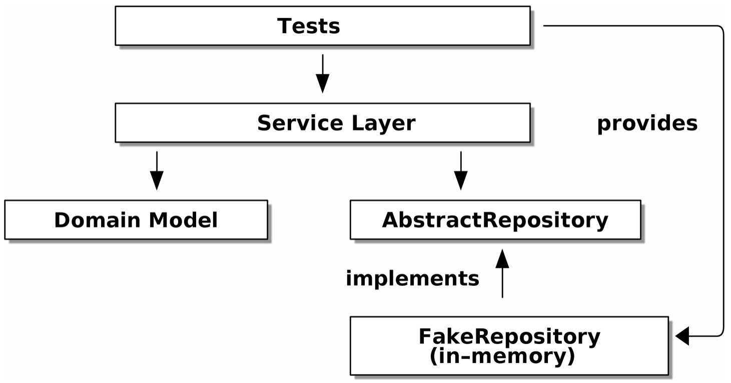


Figure 4-4. Tests provide an implementation of the abstract dependency

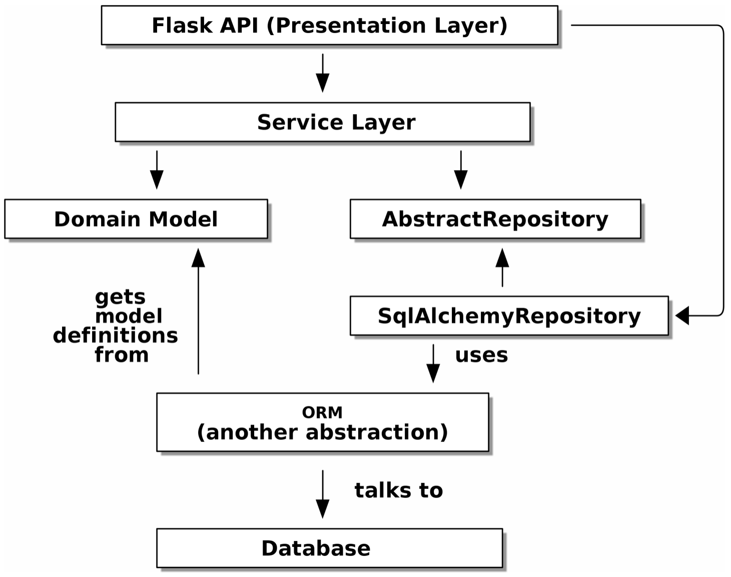


Figure 4-5. Dependencies at runtime

Wonderful.

Let's pause for Table 4-1, in which we consider the pros and cons of having a service layer at all.

Table 4-1. Service layer: the trade-offs

Pros	Cons
• We have a single place to capture all the use cases for our application. • We've placed our clever domain logic behind an API, which leaves us free to refactor. • We have cleanly separated "stuff that talks HTTP" from "stuff that talks allocation." • When combined with the Repository pattern and `FakeRepository`, we have a nice way of writing tests at a higher level than the domain layer; we can test more of our workflow without needing to use integration tests (read on to Chapter 5 for more elaboration on this).	• If your app is *purely* a web app, your controllers/view functions can be the single place to capture all the use cases. • It's yet another layer of abstraction. • Putting too much logic into the service layer can lead to the *Anemic Domain* anti-pattern. It's better to introduce this layer after you spot orchestration logic creeping into your controllers. • You can get a lot of the benefits that come from having rich domain models by simply pushing logic out of your controllers and down to the model layer, without needing to add an extra layer in between (aka "fat models, thin controllers").

But there are still some bits of awkwardness to tidy up:

- The service layer is still tightly coupled to the domain, because its API is expressed in terms of `OrderLine` objects. In Chapter 5, we'll fix that and talk about the way that the service layer enables more productive TDD.
- The service layer is tightly coupled to a `session` object. In Chapter 6, we'll introduce one more pattern that works closely with the Repository and Service Layer patterns, the Unit of Work pattern, and everything will be absolutely lovely. You'll see!

CHAPTER 5

TDD in High Gear and Low Gear

We've introduced the service layer to capture some of the additional orchestration responsibilities we need from a working application. The service layer helps us clearly define our use cases and the workflow for each: what we need to get from our repositories, what pre-checks and current state validation we should do, and what we save at the end.

But currently, many of our unit tests operate at a lower level, acting directly on the model. In this chapter we'll discuss the trade-offs involved in moving those tests up to the service-layer level, and some more general testing guidelines.

Harry Says: Seeing a Test Pyramid in Action Was a Light-Bulb Moment

Here are a few words from Harry directly:

I was initially skeptical of all Bob's architectural patterns, but seeing an actual test pyramid made me a convert.

Once you implement domain modeling and the service layer, you really actually can get to a stage where unit tests outnumber integration and end-to-end tests by an order of magnitude. Having worked in places where the E2E test build would take hours ("wait 'til tomorrow," essentially), I can't tell you what a difference it makes to be able to run all your tests in minutes or seconds.

Read on for some guidelines on how to decide what kinds of tests to write and at which level. The high gear versus low gear way of thinking really changed my testing life.

How Is Our Test Pyramid Looking?

Let's see what this move to using a service layer, with its own service-layer tests, does to our test pyramid:

Counting types of tests

```
$ grep -c test_ test_*.py
tests/unit/test_allocate.py:4
tests/unit/test_batches.py:8
tests/unit/test_services.py:3

tests/integration/test_orm.py:6
tests/integration/test_repository.py:2

tests/e2e/test_api.py:2
```

Not bad! We have 15 unit tests, 8 integration tests, and just 2 end-to-end tests. That's already a healthy-looking test pyramid.

Should Domain Layer Tests Move to the Service Layer?

Let's see what happens if we take this a step further. Since we can test our software against the service layer, we don't really need tests for the domain model anymore. Instead, we could rewrite all of the domain-level tests from Chapter 1 in terms of the service layer:

Rewriting a domain test at the service layer (tests/unit/test_services.py)

```python
# domain-layer test:
def test_prefers_current_stock_batches_to_shipments():
    in_stock_batch = Batch("in-stock-batch", "RETRO-CLOCK", 100, eta=None)
    shipment_batch = Batch("shipment-batch", "RETRO-CLOCK", 100, eta=tomorrow)
    line = OrderLine("oref", "RETRO-CLOCK", 10)

    allocate(line, [in_stock_batch, shipment_batch])

    assert in_stock_batch.available_quantity == 90
    assert shipment_batch.available_quantity == 100


# service-layer test:
def test_prefers_warehouse_batches_to_shipments():
    in_stock_batch = Batch("in-stock-batch", "RETRO-CLOCK", 100, eta=None)
    shipment_batch = Batch("shipment-batch", "RETRO-CLOCK", 100, eta=tomorrow)
    repo = FakeRepository([in_stock_batch, shipment_batch])
    session = FakeSession()

    line = OrderLine('oref', "RETRO-CLOCK", 10)
    services.allocate(line, repo, session)

    assert in_stock_batch.available_quantity == 90
    assert shipment_batch.available_quantity == 100
```

Why would we want to do that?

Tests are supposed to help us change our system fearlessly, but often we see teams writing too many tests against their domain model. This causes problems when they come to change their codebase and find that they need to update tens or even hundreds of unit tests.

This makes sense if you stop to think about the purpose of automated tests. We use tests to enforce that a property of the system doesn't change while we're working. We use tests to check that the API continues to return 200, that the database session continues to commit, and that orders are still being allocated.

If we accidentally change one of those behaviors, our tests will break. The flip side, though, is that if we want to change the design of our code, any tests relying directly on that code will also fail.

As we get further into the book, you'll see how the service layer forms an API for our system that we can drive in multiple ways. Testing against this API reduces the amount of code that we need to change when we refactor our domain model. If we restrict ourselves to testing only against the service layer, we won't have any tests that directly interact with "private" methods or attributes on our model objects, which leaves us freer to refactor them.

Every line of code that we put in a test is like a blob of glue, holding the system in a particular shape. The more low-level tests we have, the harder it will be to change things.

On Deciding What Kind of Tests to Write

You might be asking yourself, "Should I rewrite all my unit tests, then? Is it wrong to write tests against the domain model?" To answer those questions, it's important to understand the trade-off between coupling and design feedback (see Figure 5-1).

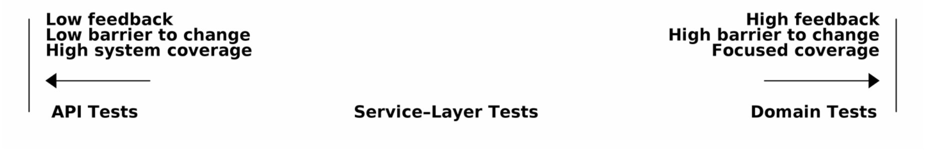


Figure 5-1. The test spectrum

Extreme programming (XP) exhorts us to "listen to the code." When we're writing tests, we might find that the code is hard to use or notice a code smell. This is a trigger for us to refactor, and to reconsider our design.

We only get that feedback, though, when we're working closely with the target code. A test for the HTTP API tells us nothing about the fine-grained design of our objects, because it sits at a much higher level of abstraction.

On the other hand, we can rewrite our entire application and, so long as we don't change the URLs or request formats, our HTTP tests will continue to pass. This gives us confidence that large-scale changes, like changing the database schema, haven't broken our code.

At the other end of the spectrum, the tests we wrote in Chapter 1 helped us to flesh out our understanding of the objects we need. The tests guided us to a design that makes sense and reads in the domain language. When our tests read in the domain language, we feel comfortable that our code matches our intuition about the problem we're trying to solve.

Because the tests are written in the domain language, they act as living documentation for our model. A new team member can read these tests to quickly understand how the system works and how the core concepts interrelate.

We often "sketch" new behaviors by writing tests at this level to see how the code might look. When we want to improve the design of the code, though, we will need to replace or delete these tests, because they are tightly coupled to a particular implementation.

High and Low Gear

Most of the time, when we are adding a new feature or fixing a bug, we don't need to make extensive changes to the domain model. In these cases, we prefer to write tests against services because of the lower coupling and higher coverage.

For example, when writing an `add_stock` function or a `cancel_order` feature, we can work more quickly and with less coupling by writing tests against the service layer.

When starting a new project or when hitting a particularly gnarly problem, we will drop back down to writing tests against the domain model so we get better feedback and executable documentation of our intent.

The metaphor we use is that of shifting gears. When starting a journey, the bicycle needs to be in a low gear so that it can overcome inertia. Once we're off and running, we can go faster and more efficiently by changing into a high gear; but if we suddenly encounter a steep hill or are forced to slow down by a hazard, we again drop down to a low gear until we can pick up speed again.

Fully Decoupling the Service-Layer Tests from the Domain

We still have direct dependencies on the domain in our service-layer tests, because we use domain objects to set up our test data and to invoke our service-layer functions.

To have a service layer that's fully decoupled from the domain, we need to rewrite its API to work in terms of primitives.

Our service layer currently takes an `OrderLine` domain object:

Before: allocate takes a domain object (service_layer/services.py)

```
def allocate(line: OrderLine, repo: AbstractRepository, session) -> str:
```

How would it look if its parameters were all primitive types?

After: allocate takes strings and ints (service_layer/services.py)

```
def allocate(
        orderid: str, sku: str, qty: int, repo: AbstractRepository, session
) -> str:
```

We rewrite the tests in those terms as well:

Tests now use primitives in function call (tests/unit/test_services.py)

```
def test_returns_allocation():
    batch = model.Batch("batch1", "COMPLICATED-LAMP", 100, eta=None)
    repo = FakeRepository([batch])

    result = services.allocate("o1", "COMPLICATED-LAMP", 10, repo, FakeSession())
    assert result == "batch1"
```

But our tests still depend on the domain, because we still manually instantiate `Batch` objects. So, if one day we decide to massively refactor how our `Batch` model works, we'll have to change a bunch of tests.

Mitigation: Keep All Domain Dependencies in Fixture Functions

We could at least abstract that out to a helper function or a fixture in our tests. Here's one way you could do that, adding a factory function on `FakeRepository`:

Factory functions for fixtures are one possibility (tests/unit/test_services.py)

```python
class FakeRepository(set):

    @staticmethod
    def for_batch(ref, sku, qty, eta=None):
        return FakeRepository([
            model.Batch(ref, sku, qty, eta),
        ])

    ...


def test_returns_allocation():
    repo = FakeRepository.for_batch("batch1", "COMPLICATED-LAMP", 100, eta=None)
    result = services.allocate("o1", "COMPLICATED-LAMP", 10, repo, FakeSession())
    assert result == "batch1"
```

At least that would move all of our tests' dependencies on the domain into one place.

Adding a Missing Service

We could go one step further, though. If we had a service to add stock, we could use that and make our service-layer tests fully expressed in terms of the service layer's official use cases, removing all dependencies on the domain:

Test for new add_batch service (tests/unit/test_services.py)

```python
def test_add_batch():
    repo, session = FakeRepository([]), FakeSession()
    services.add_batch("b1", "CRUNCHY-ARMCHAIR", 100, None, repo, session)
    assert repo.get("b1") is not None
    assert session.committed
```

In general, if you find yourself needing to do domain-layer stuff directly in your service-layer tests, it may be an indication that your service layer is incomplete.

And the implementation is just two lines:

A new service for add_batch (service_layer/services.py)

```
def add_batch(
        ref: str, sku: str, qty: int, eta: Optional[date],
        repo: AbstractRepository, session,
):
    repo.add(model.Batch(ref, sku, qty, eta))
    session.commit()


def allocate(
        orderid: str, sku: str, qty: int, repo: AbstractRepository, session
) -> str:
    ...
```

Should you write a new service just because it would help remove dependencies from your tests? Probably not. But in this case, we almost definitely would need an `add_batch` service one day anyway.

That now allows us to rewrite *all* of our service-layer tests purely in terms of the services themselves, using only primitives, and without any dependencies on the model:

Services tests now use only services (tests/unit/test_services.py)

```
def test_allocate_returns_allocation():
    repo, session = FakeRepository([]), FakeSession()
    services.add_batch("batch1", "COMPLICATED-LAMP", 100, None, repo, session)
    result = services.allocate("o1", "COMPLICATED-LAMP", 10, repo, session)
    assert result == "batch1"


def test_allocate_errors_for_invalid_sku():
    repo, session = FakeRepository([]), FakeSession()
    services.add_batch("b1", "AREALSKU", 100, None, repo, session)

    with pytest.raises(services.InvalidSku, match="Invalid sku NONEXISTENTSKU"):
        services.allocate("o1", "NONEXISTENTSKU", 10, repo, FakeSession())
```

This is a really nice place to be in. Our service-layer tests depend on only the service layer itself, leaving us completely free to refactor the model as we see fit.

Carrying the Improvement Through to the E2E Tests

In the same way that adding `add_batch` helped decouple our service-layer tests from the model, adding an API endpoint to add a batch would remove the need for the ugly `add_stock` fixture, and our E2E tests could be free of those hardcoded SQL queries and the direct dependency on the database.

Thanks to our service function, adding the endpoint is easy, with just a little JSON wrangling and a single function call required:

API for adding a batch (entrypoints/flask_app.py)

```python
@app.route("/add_batch", methods=['POST'])
def add_batch():
    session = get_session()
    repo = repository.SqlAlchemyRepository(session)
    eta = request.json['eta']
    if eta is not None:
        eta = datetime.fromisoformat(eta).date()
    services.add_batch(
        request.json['ref'], request.json['sku'], request.json['qty'], eta,
        repo, session
    )
    return 'OK', 201
```

Are you thinking to yourself, POST to */add_batch*? That's not very RESTful! You're quite right. We're being happily sloppy, but if you'd like to make it all more RESTy, maybe a POST to */batches*, then knock yourself out! Because Flask is a thin adapter, it'll be easy. See the next sidebar.

And our hardcoded SQL queries from *conftest.py* get replaced with some API calls, meaning the API tests have no dependencies other than the API, which is also nice:

API tests can now add their own batches (tests/e2e/test_api.py)

```python
def post_to_add_batch(ref, sku, qty, eta):
    url = config.get_api_url()
    r = requests.post(
        f'{url}/add_batch',
        json={'ref': ref, 'sku': sku, 'qty': qty, 'eta': eta}
    )
    assert r.status_code == 201


@pytest.mark.usefixtures('postgres_db')
@pytest.mark.usefixtures('restart_api')
def test_happy_path_returns_201_and_allocated_batch():
    sku, othersku = random_sku(), random_sku('other')
    earlybatch = random_batchref(1)
    laterbatch = random_batchref(2)
    otherbatch = random_batchref(3)
    post_to_add_batch(laterbatch, sku, 100, '2011-01-02')
    post_to_add_batch(earlybatch, sku, 100, '2011-01-01')
    post_to_add_batch(otherbatch, othersku, 100, None)
    data = {'orderid': random_orderid(), 'sku': sku, 'qty': 3}
    url = config.get_api_url()
    r = requests.post(f'{url}/allocate', json=data)
    assert r.status_code == 201
    assert r.json()['batchref'] == earlybatch
```

Wrap-Up

Once you have a service layer in place, you really can move the majority of your test coverage to unit tests and develop a healthy test pyramid.

Recap: Rules of Thumb for Different Types of Test

Aim for one end-to-end test per feature
: This might be written against an HTTP API, for example. The objective is to demonstrate that the feature works, and that all the moving parts are glued together correctly.

Write the bulk of your tests against the service layer
: These edge-to-edge tests offer a good trade-off between coverage, runtime, and efficiency. Each test tends to cover one code path of a feature and use fakes for I/O. This is the place to exhaustively cover all the edge cases and the ins and outs of your business logic.[1]

Maintain a small core of tests written against your domain model
: These tests have highly focused coverage and are more brittle, but they have the highest feedback. Don't be afraid to delete these tests if the functionality is later covered by tests at the service layer.

Error handling counts as a feature
: Ideally, your application will be structured such that all errors that bubble up to your entrypoints (e.g., Flask) are handled in the same way. This means you need to test only the happy path for each feature, and to reserve one end-to-end test for all unhappy paths (and many unhappy path unit tests, of course).

1 A valid concern about writing tests at a higher level is that it can lead to combinatorial explosion for more complex use cases. In these cases, dropping down to lower-level unit tests of the various collaborating domain objects can be useful. But see also Chapter 8 and "Optionally: Unit Testing Event Handlers in Isolation with a Fake Message Bus" on page 147.

A few things will help along the way:

- Express your service layer in terms of primitives rather than domain objects.
- In an ideal world, you'll have all the services you need to be able to test entirely against the service layer, rather than hacking state via repositories or the database. This pays off in your end-to-end tests as well.

Onto the next chapter!

CHAPTER 6

Unit of Work Pattern

In this chapter we'll introduce the final piece of the puzzle that ties together the Repository and Service Layer patterns: the *Unit of Work* pattern.

If the Repository pattern is our abstraction over the idea of persistent storage, the Unit of Work (UoW) pattern is our abstraction over the idea of *atomic operations*. It will allow us to finally and fully decouple our service layer from the data layer.

Figure 6-1 shows that, currently, a lot of communication occurs across the layers of our infrastructure: the API talks directly to the database layer to start a session, it talks to the repository layer to initialize `SQLAlchemyRepository`, and it talks to the service layer to ask it to allocate.

The code for this chapter is in the chapter_06_uow branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_06_uow
# or to code along, checkout Chapter 4:
git checkout chapter_04_service_layer
```

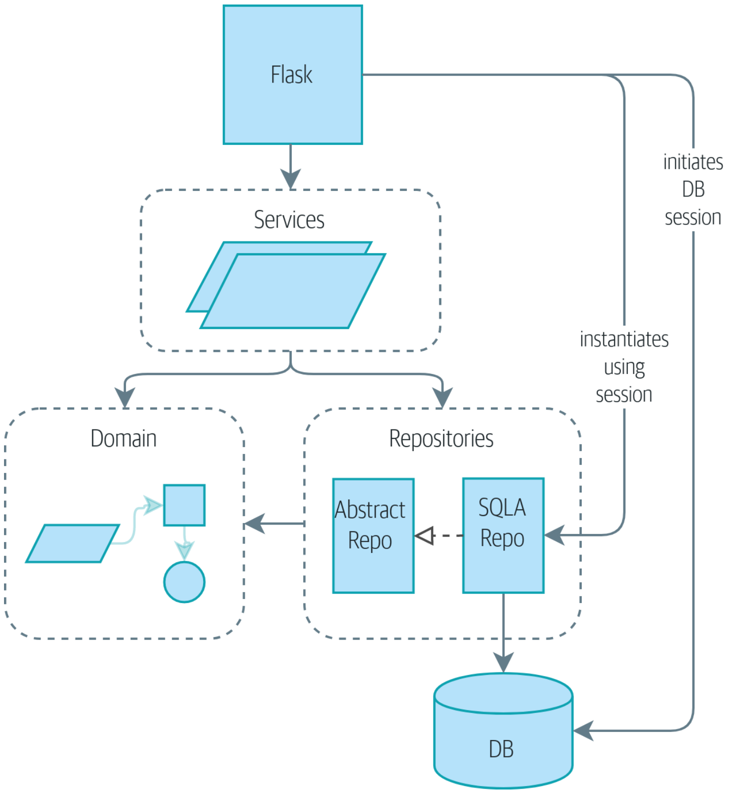


Figure 6-1. Without UoW: API talks directly to three layers

Figure 6-2 shows our target state. The Flask API now does only two things: it initializes a unit of work, and it invokes a service. The service collaborates with the UoW (we like to think of the UoW as being part of the service layer), but neither the service function itself nor Flask now needs to talk directly to the database.

And we'll do it all using a lovely piece of Python syntax, a context manager.

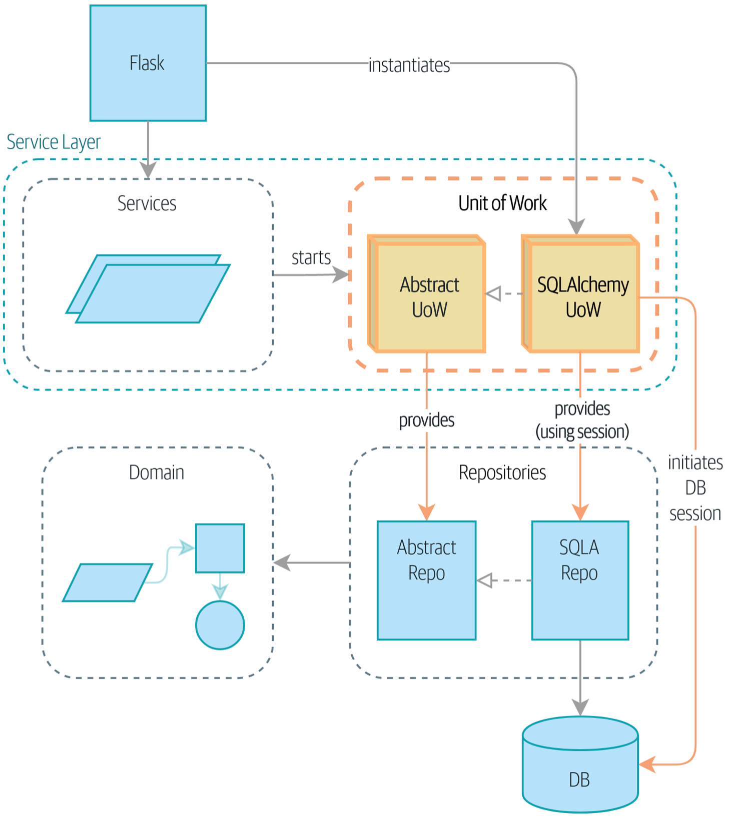


Figure 6-2. With UoW: UoW now manages database state

The Unit of Work Collaborates with the Repository

Let's see the unit of work (or UoW, which we pronounce "you-wow") in action. Here's how the service layer will look when we're finished:

Preview of unit of work in action (src/allocation/service_layer/services.py)

```
def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:  ❶
        batches = uow.batches.list()  ❷
        ...
        batchref = model.allocate(line, batches)
        uow.commit()  ❸
```

❶ We'll start a UoW as a context manager.

❷ `uow.batches` is the batches repo, so the UoW provides us access to our permanent storage.

❸ When we're done, we commit or roll back our work, using the UoW.

The UoW acts as a single entrypoint to our persistent storage, and it keeps track of what objects were loaded and of the latest state.[1]

This gives us three useful things:

- A stable snapshot of the database to work with, so the objects we use aren't changing halfway through an operation
- A way to persist all of our changes at once, so if something goes wrong, we don't end up in an inconsistent state
- A simple API to our persistence concerns and a handy place to get a repository

Test-Driving a UoW with Integration Tests

Here are our integration tests for the UOW:

A basic "round-trip" test for a UoW (tests/integration/test_uow.py)

```
def test_uow_can_retrieve_a_batch_and_allocate_to_it(session_factory):
    session = session_factory()
    insert_batch(session, 'batch1', 'HIPSTER-WORKBENCH', 100, None)
    session.commit()

    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)  ❶
    with uow:
        batch = uow.batches.get(reference='batch1')  ❷
        line = model.OrderLine('o1', 'HIPSTER-WORKBENCH', 10)
        batch.allocate(line)
        uow.commit()  ❸

    batchref = get_allocated_batch_ref(session, 'o1', 'HIPSTER-WORKBENCH')
    assert batchref == 'batch1'
```

1 You may have come across the use of the word *collaborators* to describe objects that work together to achieve a goal. The unit of work and the repository are a great example of collaborators in the object-modeling sense. In responsibility-driven design, clusters of objects that collaborate in their roles are called *object neighborhoods*, which is, in our professional opinion, totally adorable.

❶ We initialize the UoW by using our custom session factory and get back a `uow` object to use in our `with` block.

❷ The UoW gives us access to the batches repository via `uow.batches`.

❸ We call `commit()` on it when we're done.

For the curious, the `insert_batch` and `get_allocated_batch_ref` helpers look like this:

Helpers for doing SQL stuff (tests/integration/test_uow.py)

```
def insert_batch(session, ref, sku, qty, eta):
    session.execute(
        'INSERT INTO batches (reference, sku, _purchased_quantity, eta)'
        ' VALUES (:ref, :sku, :qty, :eta)',
        dict(ref=ref, sku=sku, qty=qty, eta=eta)
    )


def get_allocated_batch_ref(session, orderid, sku):
    [[orderlineid]] = session.execute(
        'SELECT id FROM order_lines WHERE orderid=:orderid AND sku=:sku',
        dict(orderid=orderid, sku=sku)
    )
    [[batchref]] = session.execute(
        'SELECT b.reference FROM allocations JOIN batches AS b ON batch_id = b.id'
        ' WHERE orderline_id=:orderlineid',
        dict(orderlineid=orderlineid)
    )
    return batchref
```

Unit of Work and Its Context Manager

In our tests we've implicitly defined an interface for what a UoW needs to do. Let's make that explicit by using an abstract base class:

Abstract UoW context manager (src/allocation/service_layer/unit_of_work.py)

```
class AbstractUnitOfWork(abc.ABC):
    batches: repository.AbstractRepository  ❶

    def __exit__(self, *args):  ❷
        self.rollback()  ❹

    @abc.abstractmethod
    def commit(self):  ❸
        raise NotImplementedError

    @abc.abstractmethod
    def rollback(self):  ❹
        raise NotImplementedError
```

❶ The UoW provides an attribute called `.batches`, which will give us access to the batches repository.

❷ If you've never seen a context manager, `__enter__` and `__exit__` are the two magic methods that execute when we enter the `with` block and when we exit it, respectively. They're our setup and teardown phases.

❸ We'll call this method to explicitly commit our work when we're ready.

❹ If we don't commit, or if we exit the context manager by raising an error, we do a `rollback`. (The rollback has no effect if `commit()` has been called. Read on for more discussion of this.)

The Real Unit of Work Uses SQLAlchemy Sessions

The main thing that our concrete implementation adds is the database session:

The real SQLAlchemy UoW (src/allocation/service_layer/unit_of_work.py)

```
DEFAULT_SESSION_FACTORY = sessionmaker(bind=create_engine(  ❶
    config.get_postgres_uri(),
))

class SqlAlchemyUnitOfWork(AbstractUnitOfWork):

    def __init__(self, session_factory=DEFAULT_SESSION_FACTORY):
        self.session_factory = session_factory  ❶

    def __enter__(self):
        self.session = self.session_factory()  # type: Session  ❷
        self.batches = repository.SqlAlchemyRepository(self.session)  ❷
        return super().__enter__()

    def __exit__(self, *args):
        super().__exit__(*args)
        self.session.close()  ❸

    def commit(self):  ❹
        self.session.commit()

    def rollback(self):  ❹
        self.session.rollback()
```

❶ The module defines a default session factory that will connect to Postgres, but we allow that to be overridden in our integration tests so that we can use SQLite instead.

❷ The `__enter__` method is responsible for starting a database session and instantiating a real repository that can use that session.

❸ We close the session on exit.

❹ Finally, we provide concrete `commit()` and `rollback()` methods that use our database session.

Fake Unit of Work for Testing

Here's how we use a fake UoW in our service-layer tests:

Fake UoW (tests/unit/test_services.py)

```
class FakeUnitOfWork(unit_of_work.AbstractUnitOfWork):

    def __init__(self):
        self.batches = FakeRepository([])  ❶
        self.committed = False  ❷

    def commit(self):
        self.committed = True  ❷

    def rollback(self):
        pass



def test_add_batch():
    uow = FakeUnitOfWork()  ❸
    services.add_batch("b1", "CRUNCHY-ARMCHAIR", 100, None, uow)  ❸
    assert uow.batches.get("b1") is not None
    assert uow.committed


def test_allocate_returns_allocation():
    uow = FakeUnitOfWork()  ❸
    services.add_batch("batch1", "COMPLICATED-LAMP", 100, None, uow)  ❸
    result = services.allocate("o1", "COMPLICATED-LAMP", 10, uow)  ❸
    assert result == "batch1"
...
```

❶ `FakeUnitOfWork` and `FakeRepository` are tightly coupled, just like the real `UnitofWork` and `Repository` classes. That's fine because we recognize that the objects are collaborators.

❷ Notice the similarity with the fake `commit()` function from `FakeSession` (which we can now get rid of). But it's a substantial improvement because we're now faking out code that we wrote rather than third-party code. Some people say, "Don't mock what you don't own".

❸ In our tests, we can instantiate a UoW and pass it to our service layer, rather than passing a repository and a session. This is considerably less cumbersome.

Don't Mock What You Don't Own

Why do we feel more comfortable mocking the UoW than the session? Both of our fakes achieve the same thing: they give us a way to swap out our persistence layer so we can run tests in memory instead of needing to talk to a real database. The difference is in the resulting design.

If we cared only about writing tests that run quickly, we could create mocks that replace SQLAlchemy and use those throughout our codebase. The problem is that `Session` is a complex object that exposes lots of persistence-related functionality. It's easy to use `Session` to make arbitrary queries against the database, but that quickly leads to data access code being sprinkled all over the codebase. To avoid that, we want to limit access to our persistence layer so each component has exactly what it needs and nothing more.

By coupling to the `Session` interface, you're choosing to couple to all the complexity of SQLAlchemy. Instead, we want to choose a simpler abstraction and use that to clearly separate responsibilities. Our UoW is much simpler than a session, and we feel comfortable with the service layer being able to start and stop units of work.

"Don't mock what you don't own" is a rule of thumb that forces us to build these simple abstractions over messy subsystems. This has the same performance benefit as mocking the SQLAlchemy session but encourages us to think carefully about our designs.

Using the UoW in the Service Layer

Here's what our new service layer looks like:

Service layer using UoW (src/allocation/service_layer/services.py)

```python
def add_batch(
        ref: str, sku: str, qty: int, eta: Optional[date],
        uow: unit_of_work.AbstractUnitOfWork  ❶
):
    with uow:
        uow.batches.add(model.Batch(ref, sku, qty, eta))
        uow.commit()


def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork  ❶
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        batches = uow.batches.list()
        if not is_valid_sku(line.sku, batches):
            raise InvalidSku(f'Invalid sku {line.sku}')
        batchref = model.allocate(line, batches)
        uow.commit()
    return batchref
```

❶ Our service layer now has only the one dependency, once again on an *abstract* UoW.

Explicit Tests for Commit/Rollback Behavior

To convince ourselves that the commit/rollback behavior works, we wrote a couple of tests:

Integration tests for rollback behavior (tests/integration/test_uow.py)

```
def test_rolls_back_uncommitted_work_by_default(session_factory):
    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)
    with uow:
        insert_batch(uow.session, 'batch1', 'MEDIUM-PLINTH', 100, None)

    new_session = session_factory()
    rows = list(new_session.execute('SELECT * FROM "batches"'))
    assert rows == []


def test_rolls_back_on_error(session_factory):
    class MyException(Exception):
        pass

    uow = unit_of_work.SqlAlchemyUnitOfWork(session_factory)
    with pytest.raises(MyException):
        with uow:
            insert_batch(uow.session, 'batch1', 'LARGE-FORK', 100, None)
            raise MyException()

    new_session = session_factory()
    rows = list(new_session.execute('SELECT * FROM "batches"'))
    assert rows == []
```

We haven't shown it here, but it can be worth testing some of the more "obscure" database behavior, like transactions, against the "real" database—that is, the same engine. For now, we're getting away with using SQLite instead of Postgres, but in Chapter 7, we'll switch some of the tests to using the real database. It's convenient that our UoW class makes that easy!

Explicit Versus Implicit Commits

Now we briefly digress on different ways of implementing the UoW pattern.

We could imagine a slightly different version of the UoW that commits by default and rolls back only if it spots an exception:

A UoW with implicit commit... (src/allocation/unit_of_work.py)

```
class AbstractUnitOfWork(abc.ABC):

    def __enter__(self):
        return self

    def __exit__(self, exn_type, exn_value, traceback):
        if exn_type is None:
            self.commit()  ❶
        else:
            self.rollback()  ❷
```

❶ Should we have an implicit commit in the happy path?

❷ And roll back only on exception?

It would allow us to save a line of code and to remove the explicit commit from our client code:

...would save us a line of code (src/allocation/service_layer/services.py)

```
def add_batch(ref: str, sku: str, qty: int, eta: Optional[date], uow):
    with uow:
        uow.batches.add(model.Batch(ref, sku, qty, eta))
        # uow.commit()
```

This is a judgment call, but we tend to prefer requiring the explicit commit so that we have to choose when to flush state.

Although we use an extra line of code, this makes the software safe by default. The default behavior is to *not change anything*. In turn, that makes our code easier to reason about because there's only one code path that leads to changes in the system: total success and an explicit commit. Any other code path, any exception, any early exit from the UoW's scope leads to a safe state.

Similarly, we prefer to roll back by default because it's easier to understand; this rolls back to the last commit, so either the user did one, or we blow their changes away. Harsh but simple.

Examples: Using UoW to Group Multiple Operations into an Atomic Unit

Here are a few examples showing the Unit of Work pattern in use. You can see how it leads to simple reasoning about what blocks of code happen together.

Example 1: Reallocate

Suppose we want to be able to deallocate and then reallocate orders:

Reallocate service function

```
def reallocate(line: OrderLine, uow: AbstractUnitOfWork) -> str:
    with uow:
        batch = uow.batches.get(sku=line.sku)
        if batch is None:
            raise InvalidSku(f'Invalid sku {line.sku}')
        batch.deallocate(line)  ❶
        allocate(line)  ❷
        uow.commit()
```

❶ If `deallocate()` fails, we don't want to call `allocate()`, obviously.

❷ If `allocate()` fails, we probably don't want to actually commit the `deallocate()` either.

Example 2: Change Batch Quantity

Our shipping company gives us a call to say that one of the container doors opened, and half our sofas have fallen into the Indian Ocean. Oops!

Change quantity

```
def change_batch_quantity(batchref: str, new_qty: int, uow: AbstractUnitOfWork):
    with uow:
        batch = uow.batches.get(reference=batchref)
        batch.change_purchased_quantity(new_qty)
        while batch.available_quantity < 0:
            line = batch.deallocate_one()  ❶
        uow.commit()
```

❶ Here we may need to deallocate any number of lines. If we get a failure at any stage, we probably want to commit none of the changes.

Tidying Up the Integration Tests

We now have three sets of tests, all essentially pointing at the database: *test_orm.py*, *test_repository.py*, and *test_uow.py*. Should we throw any away?

```
└── tests
    ├── conftest.py
    ├── e2e
    │   └── test_api.py
    ├── integration
    │   ├── test_orm.py
    │   ├── test_repository.py
    │   └── test_uow.py
    ├── pytest.ini
    └── unit
        ├── test_allocate.py
        ├── test_batches.py
        └── test_services.py
```

You should always feel free to throw away tests if you think they're not going to add value longer term. We'd say that *test_orm.py* was primarily a tool to help us learn SQLAlchemy, so we won't need that long term, especially if the main things it's doing are covered in *test_repository.py*. That last test, you might keep around, but we could certainly see an argument for just keeping everything at the highest possible level of abstraction (just as we did for the unit tests).

Exercise for the Reader

For this chapter, probably the best thing to try is to implement a UoW from scratch. The code, as always, is on GitHub. You could either follow the model we have quite closely, or perhaps experiment with separating the UoW (whose responsibilities are `commit()`, `rollback()`, and providing the `.batches` repository) from the context manager, whose job is to initialize things, and then do the commit or rollback on exit. If you feel like going all-functional rather than messing about with all these classes, you could use `@contextmanager` from `contextlib`.

We've stripped out both the actual UoW and the fakes, as well as paring back the abstract UoW. Why not send us a link to your repo if you come up with something you're particularly proud of?

This is another example of the lesson from Chapter 5: as we build better abstractions, we can move our tests to run against them, which leaves us free to change the underlying details.

Wrap-Up

Hopefully we've convinced you that the Unit of Work pattern is useful, and that the context manager is a really nice Pythonic way of visually grouping code into blocks that we want to happen atomically.

This pattern is so useful, in fact, that SQLAlchemy already uses a UoW in the shape of the `Session` object. The `Session` object in SQLAlchemy is the way that your application loads data from the database.

Every time you load a new entity from the database, the session begins to *track* changes to the entity, and when the session is *flushed*, all your changes are persisted together. Why do we go to the effort of abstracting away the SQLAlchemy session if it already implements the pattern we want?

Table 6-1 discusses some of the trade-offs.

Table 6-1. Unit of Work pattern: the trade-offs

Pros	Cons
• We have a nice abstraction over the concept of atomic operations, and the context manager makes it easy to see, visually, what blocks of code are grouped together atomically. • We have explicit control over when a transaction starts and finishes, and our application fails in a way that is safe by default. We never have to worry that an operation is partially committed. • It's a nice place to put all your repositories so client code can access them. • As you'll see in later chapters, atomicity isn't only about transactions; it can help us work with events and the message bus.	• Your ORM probably already has some perfectly good abstractions around atomicity. SQLAlchemy even has context managers. You can go a long way just passing a session around. • We've made it look easy, but you have to think quite carefully about things like rollbacks, multithreading, and nested transactions. Perhaps just sticking to what Django or Flask-SQLAlchemy gives you will keep your life simpler.

For one thing, the Session API is rich and supports operations that we don't want or need in our domain. Our `UnitOfWork` simplifies the session to its essential core: it can be started, committed, or thrown away.

For another, we're using the `UnitOfWork` to access our `Repository` objects. This is a neat bit of developer usability that we couldn't do with a plain SQLAlchemy `Session`.

Unit of Work Pattern Recap

The Unit of Work pattern is an abstraction around data integrity
: It helps to enforce the consistency of our domain model, and improves performance, by letting us perform a single *flush* operation at the end of an operation.

It works closely with the Repository and Service Layer patterns
: The Unit of Work pattern completes our abstractions over data access by representing atomic updates. Each of our service-layer use cases runs in a single unit of work that succeeds or fails as a block.

This is a lovely case for a context manager
: Context managers are an idiomatic way of defining scope in Python. We can use a context manager to automatically roll back our work at the end of a request, which means the system is safe by default.

SQLAlchemy already implements this pattern
: We introduce an even simpler abstraction over the SQLAlchemy `Session` object in order to "narrow" the interface between the ORM and our code. This helps to keep us loosely coupled.

Lastly, we're motivated again by the dependency inversion principle: our service layer depends on a thin abstraction, and we attach a concrete implementation at the outside edge of the system. This lines up nicely with SQLAlchemy's own recommendations:

> Keep the life cycle of the session (and usually the transaction) separate and external. The most comprehensive approach, recommended for more substantial applications, will try to keep the details of session, transaction, and exception management as far as possible from the details of the program doing its work.
>
> —SQLALchemy "Session Basics" Documentation

CHAPTER 7

Aggregates and Consistency Boundaries

In this chapter, we'd like to revisit our domain model to talk about invariants and constraints, and see how our domain objects can maintain their own internal consistency, both conceptually and in persistent storage. We'll discuss the concept of a *consistency boundary* and show how making it explicit can help us to build high-performance software without compromising maintainability.

Figure 7-1 shows a preview of where we're headed: we'll introduce a new model object called `Product` to wrap multiple batches, and we'll make the old `allocate()` domain service available as a method on `Product` instead.

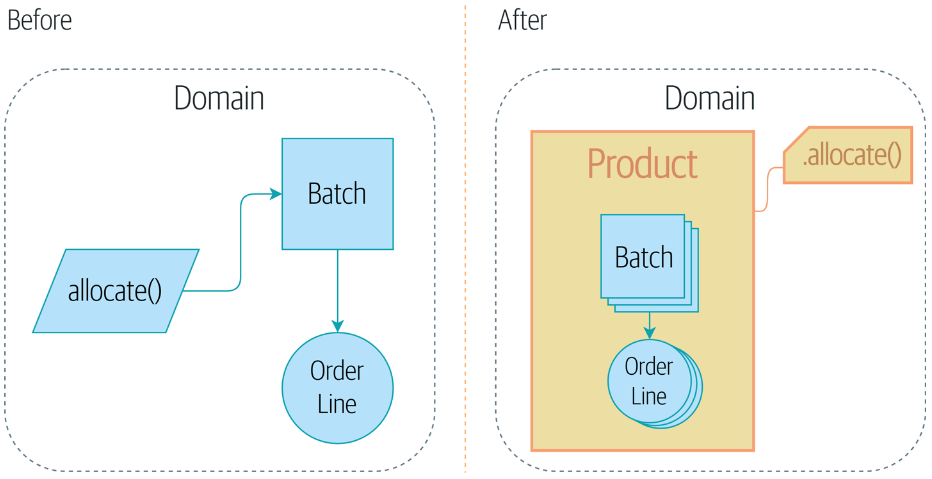


Figure 7-1. Adding the Product aggregate

Why? Let's find out.

The code for this chapter is in the appendix_csvs branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout appendix_csvs
# or to code along, checkout the previous chapter:
git checkout chapter_06_uow
```

Why Not Just Run Everything in a Spreadsheet?

What's the point of a domain model, anyway? What's the fundamental problem we're trying to address?

Couldn't we just run everything in a spreadsheet? Many of our users would be delighted by that. Business users *like* spreadsheets because they're simple, familiar, and yet enormously powerful.

In fact, an enormous number of business processes do operate by manually sending spreadsheets back and forth over email. This "CSV over SMTP" architecture has low initial complexity but tends not to scale very well because it's difficult to apply logic and maintain consistency.

Who is allowed to view this particular field? Who's allowed to update it? What happens when we try to order –350 chairs, or 10,000,000 tables? Can an employee have a negative salary?

These are the constraints of a system. Much of the domain logic we write exists to enforce these constraints in order to maintain the invariants of the system. The *invariants* are the things that have to be true whenever we finish an operation.

Invariants, Constraints, and Consistency

The two words are somewhat interchangeable, but a *constraint* is a rule that restricts the possible states our model can get into, while an *invariant* is defined a little more precisely as a condition that is always true.

If we were writing a hotel-booking system, we might have the constraint that double bookings are not allowed. This supports the invariant that a room cannot have more than one booking for the same night.

Of course, sometimes we might need to temporarily *bend* the rules. Perhaps we need to shuffle the rooms around because of a VIP booking. While we're moving bookings around in memory, we might be double booked, but our domain model should ensure that, when we're finished, we end up in a final consistent state, where the invariants are met. If we can't find a way to accommodate all our guests, we should raise an error and refuse to complete the operation.

Let's look at a couple of concrete examples from our business requirements; we'll start with this one:

> An order line can be allocated to only one batch at a time.
>
> —The business

This is a business rule that imposes an invariant. The invariant is that an order line is allocated to either zero or one batch, but never more than one. We need to make sure that our code never accidentally calls `Batch.allocate()` on two different batches for the same line, and currently, there's nothing there to explicitly stop us from doing that.

Invariants, Concurrency, and Locks

Let's look at another one of our business rules:

> We can't allocate to a batch if the available quantity is less than the quantity of the order line.
>
> —The business

Here the constraint is that we can't allocate more stock than is available to a batch, so we never oversell stock by allocating two customers to the same physical cushion, for example. Every time we update the state of the system, our code needs to ensure that we don't break the invariant, which is that the available quantity must be greater than or equal to zero.

In a single-threaded, single-user application, it's relatively easy for us to maintain this invariant. We can just allocate stock one line at a time, and raise an error if there's no stock available.

This gets much harder when we introduce the idea of *concurrency*. Suddenly we might be allocating stock for multiple order lines simultaneously. We might even be allocating order lines at the same time as processing changes to the batches themselves.

We usually solve this problem by applying *locks* to our database tables. This prevents two operations from happening simultaneously on the same row or same table.

As we start to think about scaling up our app, we realize that our model of allocating lines against all available batches may not scale. If we process tens of thousands of orders per hour, and hundreds of thousands of order lines, we can't hold a lock over the whole `batches` table for every single one—we'll get deadlocks or performance problems at the very least.

What Is an Aggregate?

OK, so if we can't lock the whole database every time we want to allocate an order line, what should we do instead? We want to protect the invariants of our system but allow for the greatest degree of concurrency. Maintaining our invariants inevitably means preventing concurrent writes; if multiple users can allocate `DEADLY-SPOON` at the same time, we run the risk of overallocating.

On the other hand, there's no reason we can't allocate `DEADLY-SPOON` at the same time as `FLIMSY-DESK`. It's safe to allocate two products at the same time because there's no invariant that covers them both. We don't need them to be consistent with each other.

The *Aggregate* pattern is a design pattern from the DDD community that helps us to resolve this tension. An *aggregate* is just a domain object that contains other domain objects and lets us treat the whole collection as a single unit.

The only way to modify the objects inside the aggregate is to load the whole thing, and to call methods on the aggregate itself.

As a model gets more complex and grows more entity and value objects, referencing each other in a tangled graph, it can be hard to keep track of who can modify what. Especially when we have *collections* in the model as we do (our batches are a collection), it's a good idea to nominate some entities to be the single entrypoint for modifying their related objects. It makes the system conceptually simpler and easy to reason about if you nominate some objects to be in charge of consistency for the others.

For example, if we're building a shopping site, the Cart might make a good aggregate: it's a collection of items that we can treat as a single unit. Importantly, we want to load the entire basket as a single blob from our data store. We don't want two requests to modify the basket at the same time, or we run the risk of weird concurrency errors. Instead, we want each change to the basket to run in a single database transaction.

We don't want to modify multiple baskets in a transaction, because there's no use case for changing the baskets of several customers at the same time. Each basket is a single *consistency boundary* responsible for maintaining its own invariants.

> An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes.
>
> —Eric Evans, *Domain-Driven Design blue book*

Per Evans, our aggregate has a root entity (the Cart) that encapsulates access to items. Each item has its own identity, but other parts of the system will always refer to the Cart only as an indivisible whole.

Just as we sometimes use `_leading_underscores` to mark methods or functions as "private," you can think of aggregates as being the "public" classes of our model, and the rest of the entities and value objects as "private."

Choosing an Aggregate

What aggregate should we use for our system? The choice is somewhat arbitrary, but it's important. The aggregate will be the boundary where we make sure every operation ends in a consistent state. This helps us to reason about our software and prevent weird race issues. We want to draw a boundary around a small number of objects—the smaller, the better, for performance—that have to be consistent with one another, and we need to give this boundary a good name.

The object we're manipulating under the covers is `Batch`. What do we call a collection of batches? How should we divide all the batches in the system into discrete islands of consistency?

We *could* use `Shipment` as our boundary. Each shipment contains several batches, and they all travel to our warehouse at the same time. Or perhaps we could use `Warehouse` as our boundary: each warehouse contains many batches, and counting all the stock at the same time could make sense.

Neither of these concepts really satisfies us, though. We should be able to allocate `DEADLY-SPOONs` and `FLIMSY-DESKs` at the same time, even if they're in the same warehouse or the same shipment. These concepts have the wrong granularity.

When we allocate an order line, we're interested only in batches that have the same SKU as the order line. Some sort of concept like `GlobalSkuStock` could work: a collection of all the batches for a given SKU.

It's an unwieldy name, though, so after some bikeshedding via `SkuStock`, `Stock`, `ProductStock`, and so on, we decided to simply call it `Product`—after all, that was the first concept we came across in our exploration of the domain language back in Chapter 1.

So the plan is this: when we want to allocate an order line, instead of Figure 7-2, where we look up all the `Batch` objects in the world and pass them to the `allocate()` domain service…

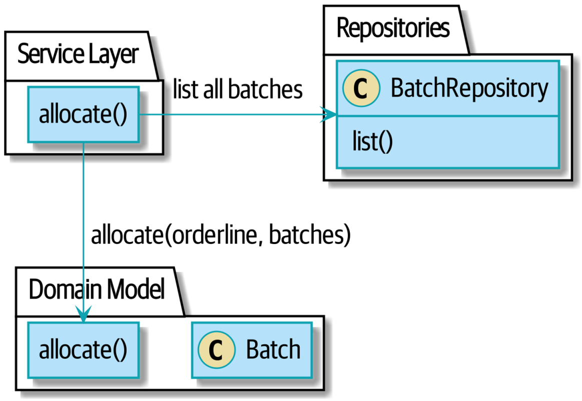


Figure 7-2. Before: allocate against all batches using the domain service

…we'll move to the world of Figure 7-3, in which there is a new `Product` object for the particular SKU of our order line, and it will be in charge of all the batches *for that SKU*, and we can call a `.allocate()` method on that instead.

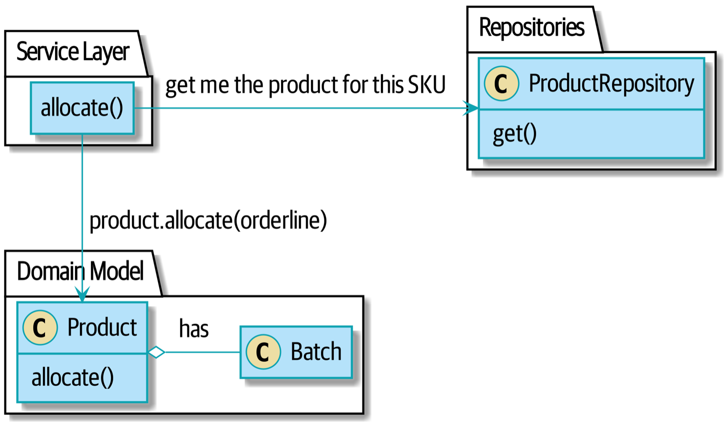


Figure 7-3. After: ask Product to allocate against its batches

Let's see how that looks in code form:

Our chosen aggregate, Product (src/allocation/domain/model.py)

```
class Product:

    def __init__(self, sku: str, batches: List[Batch]):
        self.sku = sku  ❶
        self.batches = batches  ❷

    def allocate(self, line: OrderLine) -> str:  ❸
        try:
            batch = next(
                b for b in sorted(self.batches) if b.can_allocate(line)
            )
            batch.allocate(line)
            return batch.reference
        except StopIteration:
            raise OutOfStock(f'Out of stock for sku {line.sku}')
```

❶ `Product`'s main identifier is the `sku`.

❷ Our `Product` class holds a reference to a collection of `batches` for that SKU.

❸ Finally, we can move the `allocate()` domain service to be a method on the `Product` aggregate.

This `Product` might not look like what you'd expect a `Product` model to look like. No price, no description, no dimensions. Our allocation service doesn't care about any of those things. This is the power of bounded contexts; the concept of a product in one app can be very different from another. See the following sidebar for more discussion.

Aggregates, Bounded Contexts, and Microservices

One of the most important contributions from Evans and the DDD community is the concept of *bounded contexts*.

In essence, this was a reaction against attempts to capture entire businesses into a single model. The word *customer* means different things to people in sales, customer service, logistics, support, and so on. Attributes needed in one context are irrelevant in another; more perniciously, concepts with the same name can have entirely different meanings in different contexts. Rather than trying to build a single model (or class, or database) to capture all the use cases, it's better to have several models, draw boundaries around each context, and handle the translation between different contexts explicitly.

This concept translates very well to the world of microservices, where each microservice is free to have its own concept of "customer" and its own rules for translating that to and from other microservices it integrates with.

In our example, the allocation service has `Product(sku, batches)`, whereas the ecommerce will have `Product(sku, description, price, image_url, dimensions, etc...)`. As a rule of thumb, your domain models should include only the data that they need for performing calculations.

Whether or not you have a microservices architecture, a key consideration in choosing your aggregates is also choosing the bounded context that they will operate in. By restricting the context, you can keep your number of aggregates low and their size manageable.

Once again, we find ourselves forced to say that we can't give this issue the treatment it deserves here, and we can only encourage you to read up on it elsewhere. The Fowler link at the start of this sidebar is a good starting point, and either (or indeed, any) DDD book will have a chapter or more on bounded contexts.

One Aggregate = One Repository

Once you define certain entities to be aggregates, we need to apply the rule that they are the only entities that are publicly accessible to the outside world. In other words, the only repositories we are allowed should be repositories that return aggregates.

The rule that repositories should only return aggregates is the main place where we enforce the convention that aggregates are the only way into our domain model. Be wary of breaking it!

In our case, we'll switch from `BatchRepository` to `ProductRepository`:

Our new UoW and repository (unit_of_work.py and repository.py)

```
class AbstractUnitOfWork(abc.ABC):
    products: repository.AbstractProductRepository

...

class AbstractProductRepository(abc.ABC):

    @abc.abstractmethod
    def add(self, product):
        ...

    @abc.abstractmethod
    def get(self, sku) -> model.Product:
        ...
```

The ORM layer will need some tweaks so that the right batches automatically get loaded and associated with `Product` objects. The nice thing is, the Repository pattern means we don't have to worry about that yet. We can just use our `FakeRepository` and then feed through the new model into our service layer to see how it looks with `Product` as its main entrypoint:

Service layer (src/allocation/service_layer/services.py)

```
def add_batch(
        ref: str, sku: str, qty: int, eta: Optional[date],
        uow: unit_of_work.AbstractUnitOfWork
):
    with uow:
        product = uow.products.get(sku=sku)
        if product is None:
            product = model.Product(sku, batches=[])
            uow.products.add(product)
        product.batches.append(model.Batch(ref, sku, qty, eta))
        uow.commit()


def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f'Invalid sku {line.sku}')
        batchref = product.allocate(line)
        uow.commit()
    return batchref
```

What About Performance?

We've mentioned a few times that we're modeling with aggregates because we want to have high-performance software, but here we are loading *all* the batches when we only need one. You might expect that to be inefficient, but there are a few reasons why we're comfortable here.

First, we're purposefully modeling our data so that we can make a single query to the database to read, and a single update to persist our changes. This tends to perform much better than systems that issue lots of ad hoc queries. In systems that don't model this way, we often find that transactions slowly get longer and more complex as the software evolves.

Second, our data structures are minimal and comprise a few strings and integers per row. We can easily load tens or even hundreds of batches in a few milliseconds.

Third, we expect to have only 20 or so batches of each product at a time. Once a batch is used up, we can discount it from our calculations. This means that the amount of data we're fetching shouldn't get out of control over time.

If we *did* expect to have thousands of active batches for a product, we'd have a couple of options. For one, we could use lazy-loading for the batches in a product. From the perspective of our code, nothing would change, but in the background, SQLAlchemy would page through data for us. This would lead to more requests, each fetching a smaller number of rows. Because we need to find only a single batch with enough capacity for our order, this might work pretty well.

Exercise for the Reader

You've just seen the main top layers of the code, so this shouldn't be too hard, but we'd like you to implement the `Product` aggregate starting from `Batch`, just as we did.

Of course, you could cheat and copy/paste from the previous listings, but even if you do that, you'll still have to solve a few challenges on your own, like adding the model to the ORM and making sure all the moving parts can talk to each other, which we hope will be instructive.

You'll find the code on GitHub. We've put in a "cheating" implementation in the delegates to the existing `allocate()` function, so you should be able to evolve that toward the real thing.

We've marked a couple of tests with `@pytest.skip()`. After you've read the rest of this chapter, come back to these tests to have a go at implementing version numbers. Bonus points if you can get SQLAlchemy to do them for you by magic!

If all else failed, we'd just look for a different aggregate. Maybe we could split up batches by region or by warehouse. Maybe we could redesign our data access strategy around the shipment concept. The Aggregate pattern is designed to help manage some technical constraints around consistency and performance. There isn't *one* correct aggregate, and we should feel comfortable changing our minds if we find our boundaries are causing performance woes.

Optimistic Concurrency with Version Numbers

We have our new aggregate, so we've solved the conceptual problem of choosing an object to be in charge of consistency boundaries. Let's now spend a little time talking about how to enforce data integrity at the database level.

This section has a lot of implementation details; for example, some of it is Postgres-specific. But more generally, we're showing one way of managing concurrency issues, but it is just one approach. Real requirements in this area vary a lot from project to project. You shouldn't expect to be able to copy and paste code from here into production.

We don't want to hold a lock over the entire `batches` table, but how will we implement holding a lock over just the rows for a particular SKU?

One answer is to have a single attribute on the `Product` model that acts as a marker for the whole state change being complete and to use it as the single resource that concurrent workers can fight over. If two transactions read the state of the world for `batches` at the same time, and both want to update the `allocations` tables, we force both to also try to update the `version_number` in the `products` table, in such a way that only one of them can win and the world stays consistent.

Figure 7-4 illustrates two concurrent transactions doing their read operations at the same time, so they see a `Product` with, for example, `version=3`. They both call `Product.allocate()` in order to modify a state. But we set up our database integrity rules such that only one of them is allowed to `commit` the new `Product` with `version=4`, and the other update is rejected.

Version numbers are just one way to implement optimistic locking. You could achieve the same thing by setting the Postgres transaction isolation level to `SERIALIZABLE`, but that often comes at a severe performance cost. Version numbers also make implicit concepts explicit.

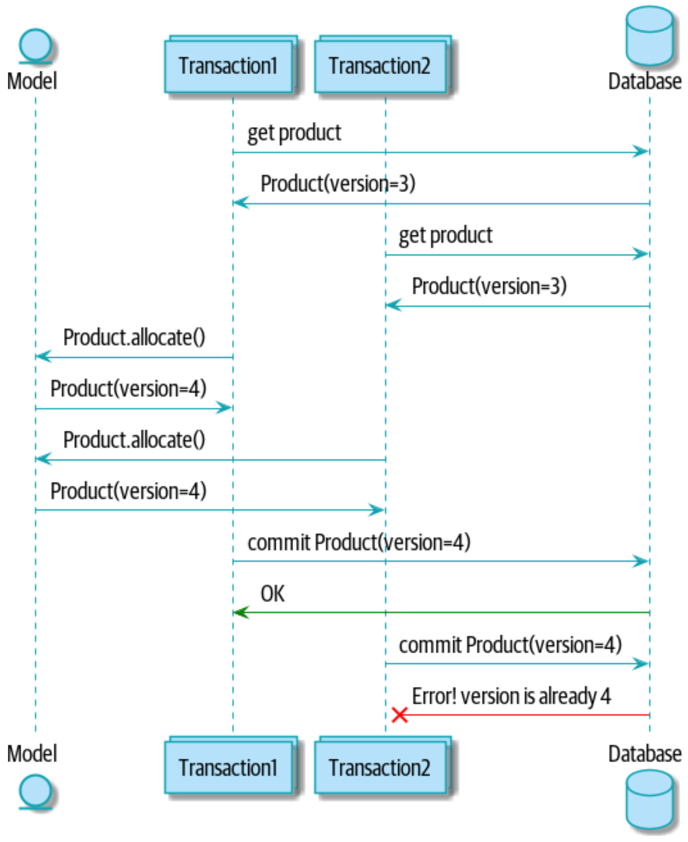


Figure 7-4. Sequence diagram: two transactions attempting a concurrent update on `Product`

Optimistic Concurrency Control and Retries

What we've implemented here is called *optimistic* concurrency control because our default assumption is that everything will be fine when two users want to make changes to the database. We think it's unlikely that they will conflict with each other, so we let them go ahead and just make sure we have a way to notice if there is a problem.

Pessimistic concurrency control works under the assumption that two users are going to cause conflicts, and we want to prevent conflicts in all cases, so we lock everything just to be safe. In our example, that would mean locking the whole `batches` table, or using `SELECT FOR UPDATE`—we're pretending that we've ruled those out for performance reasons, but in real life you'd want to do some evaluations and measurements of your own.

With pessimistic locking, you don't need to think about handling failures because the database will prevent them for you (although you do need to think about deadlocks). With optimistic locking, you need to explicitly handle the possibility of failures in the (hopefully unlikely) case of a clash.

The usual way to handle a failure is to retry the failed operation from the beginning. Imagine we have two customers, Harry and Bob, and each submits an order for `SHINY-TABLE`. Both threads load the product at version 1 and allocate stock. The database prevents the concurrent update, and Bob's order fails with an error. When we *retry* the operation, Bob's order loads the product at version 2 and tries to allocate again. If there is enough stock left, all is well; otherwise, he'll receive `OutOfStock`. Most operations can be retried this way in the case of a concurrency problem.

Read more on retries in "Recovering from Errors Synchronously" on page 158 and "Footguns" on page 226.

Implementation Options for Version Numbers

There are essentially three options for implementing version numbers:

1. `version_number` lives in the domain; we add it to the `Product` constructor, and `Product.allocate()` is responsible for incrementing it.
2. The service layer could do it! The version number isn't *strictly* a domain concern, so instead our service layer could assume that the current version number is attached to `Product` by the repository, and the service layer will increment it before it does the `commit()`.
3. Since it's arguably an infrastructure concern, the UoW and repository could do it by magic. The repository has access to version numbers for any products it

retrieves, and when the UoW does a commit, it can increment the version number for any products it knows about, assuming them to have changed.

Option 3 isn't ideal, because there's no real way of doing it without having to assume that *all* products have changed, so we'll be incrementing version numbers when we don't have to.[1]

Option 2 involves mixing the responsibility for mutating state between the service layer and the domain layer, so it's a little messy as well.

So in the end, even though version numbers don't *have* to be a domain concern, you might decide the cleanest trade-off is to put them in the domain:

Our chosen aggregate, Product (src/allocation/domain/model.py)

```
class Product:

    def __init__(self, sku: str, batches: List[Batch], version_number: int = 0):  ❶
        self.sku = sku
        self.batches = batches
        self.version_number = version_number  ❶

    def allocate(self, line: OrderLine) -> str:
        try:
            batch = next(
                b for b in sorted(self.batches) if b.can_allocate(line)
            )
            batch.allocate(line)
            self.version_number += 1  ❶
            return batch.reference
        except StopIteration:
            raise OutOfStock(f'Out of stock for sku {line.sku}')
```

❶ There it is!

If you're scratching your head at this version number business, it might help to remember that the *number* isn't important. What's important is that the `Product` database row is modified whenever we make a change to the `Product` aggregate. The version number is a simple, human-comprehensible way to model a thing that changes on every write, but it could equally be a random UUID every time.

1 Perhaps we could get some ORM/SQLAlchemy magic to tell us when an object is dirty, but how would that work in the generic case—for example, for a `CsvRepository`?

Testing for Our Data Integrity Rules

Now to make sure we can get the behavior we want: if we have two concurrent attempts to do allocation against the same `Product`, one of them should fail, because they can't both update the version number.

First, let's simulate a "slow" transaction using a function that does allocation and then does an explicit sleep:[2]

time.sleep can reproduce concurrency behavior (tests/integration/test_uow.py)

```
def try_to_allocate(orderid, sku, exceptions):
    line = model.OrderLine(orderid, sku, 10)
    try:
        with unit_of_work.SqlAlchemyUnitOfWork() as uow:
            product = uow.products.get(sku=sku)
            product.allocate(line)
            time.sleep(0.2)
            uow.commit()
    except Exception as e:
        print(traceback.format_exc())
        exceptions.append(e)
```

Then we have our test invoke this slow allocation twice, concurrently, using threads:

An integration test for concurrency behavior (tests/integration/test_uow.py)

```
def test_concurrent_updates_to_version_are_not_allowed(postgres_session_factory):
    sku, batch = random_sku(), random_batchref()
    session = postgres_session_factory()
    insert_batch(session, batch, sku, 100, eta=None, product_version=1)
    session.commit()

    order1, order2 = random_orderid(1), random_orderid(2)
    exceptions = []  # type: List[Exception]
    try_to_allocate_order1 = lambda: try_to_allocate(order1, sku, exceptions)
    try_to_allocate_order2 = lambda: try_to_allocate(order2, sku, exceptions)
    thread1 = threading.Thread(target=try_to_allocate_order1)  ❶
    thread2 = threading.Thread(target=try_to_allocate_order2)  ❶
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()

    [[version]] = session.execute(
        "SELECT version_number FROM products WHERE sku=:sku",
```

2 `time.sleep()` works well in our use case, but it's not the most reliable or efficient way to reproduce concurrency bugs. Consider using semaphores or similar synchronization primitives shared between your threads to get better guarantees of behavior.

```
        dict(sku=sku),
    )
    assert version == 2  ❷
    [exception] = exceptions
    assert 'could not serialize access due to concurrent update' in str(exception)  ❸

    orders = list(session.execute(
        "SELECT orderid FROM allocations"
        " JOIN batches ON allocations.batch_id = batches.id"
        " JOIN order_lines ON allocations.orderline_id = order_lines.id"
        " WHERE order_lines.sku=:sku",
        dict(sku=sku),
    ))
    assert len(orders) == 1  ❹
    with unit_of_work.SqlAlchemyUnitOfWork() as uow:
        uow.session.execute('select 1')
```

❶ We start two threads that will reliably produce the concurrency behavior we want: `read1, read2, write1, write2`.

❷ We assert that the version number has been incremented only once.

❸ We can also check on the specific exception if we like.

❹ And we double-check that only one allocation has gotten through.

Enforcing Concurrency Rules by Using Database Transaction Isolation Levels

To get the test to pass as it is, we can set the transaction isolation level on our session:

Set isolation level for session (src/allocation/service_layer/unit_of_work.py)

```
DEFAULT_SESSION_FACTORY = sessionmaker(bind=create_engine(
    config.get_postgres_uri(),
    isolation_level="REPEATABLE READ",
))
```

Transaction isolation levels are tricky stuff, so it's worth spending time understanding the Postgres documentation.[3]

[3] If you're not using Postgres, you'll need to read different documentation. Annoyingly, different databases all have quite different definitions. Oracle's `SERIALIZABLE` is equivalent to Postgres's `REPEATABLE READ`, for example.

Pessimistic Concurrency Control Example: SELECT FOR UPDATE

There are multiple ways to approach this, but we'll show one. `SELECT FOR UPDATE` produces different behavior; two concurrent transactions will not be allowed to do a read on the same rows at the same time:

`SELECT FOR UPDATE` is a way of picking a row or rows to use as a lock (although those rows don't have to be the ones you update). If two transactions both try to `SELECT FOR UPDATE` a row at the same time, one will win, and the other will wait until the lock is released. So this is an example of pessimistic concurrency control.

Here's how you can use the SQLAlchemy DSL to specify `FOR UPDATE` at query time:

SQLAlchemy with_for_update (src/allocation/adapters/repository.py)

```
    def get(self, sku):
        return self.session.query(model.Product) \
                           .filter_by(sku=sku) \
                           .with_for_update() \
                           .first()
```

This will have the effect of changing the concurrency pattern from

```
    read1, read2, write1, write2(fail)
```

to

```
    read1, write1, read2, write2(succeed)
```

Some people refer to this as the "read-modify-write" failure mode. Read "PostgreSQL Anti-Patterns: Read-Modify-Write Cycles" for a good overview.

We don't really have time to discuss all the trade-offs between `REPEATABLE READ` and `SELECT FOR UPDATE`, or optimistic versus pessimistic locking in general. But if you have a test like the one we've shown, you can specify the behavior you want and see how it changes. You can also use the test as a basis for performing some performance experiments.

Wrap-Up

Specific choices around concurrency control vary a lot based on business circumstances and storage technology choices, but we'd like to bring this chapter back to the conceptual idea of an aggregate: we explicitly model an object as being the main entrypoint to some subset of our model, and as being in charge of enforcing the invariants and business rules that apply across all of those objects.

Choosing the right aggregate is key, and it's a decision you may revisit over time. You can read more about it in multiple DDD books. We also recommend these three online papers on effective aggregate design by Vaughn Vernon (the "red book" author).

Table 7-1 has some thoughts on the trade-offs of implementing the Aggregate pattern.

Table 7-1. Aggregates: the trade-offs

Pros	Cons
• Python might not have "official" public and private methods, but we do have the underscores convention, because it's often useful to try to indicate what's for "internal" use and what's for "outside code" to use. Choosing aggregates is just the next level up: it lets you decide which of your domain model classes are the public ones, and which aren't. • Modeling our operations around explicit consistency boundaries helps us avoid performance problems with our ORM. • Putting the aggregate in sole charge of state changes to its subsidiary models makes the system easier to reason about, and makes it easier to control invariants.	• Yet another new concept for new developers to take on. Explaining entities versus value objects was already a mental load; now there's a third type of domain model object? • Sticking rigidly to the rule that we modify only one aggregate at a time is a big mental shift. • Dealing with eventual consistency between aggregates can be complex.

Aggregates and Consistency Boundaries Recap

Aggregates are your entrypoints into the domain model
By restricting the number of ways that things can be changed, we make the system easier to reason about.

Aggregates are in charge of a consistency boundary
An aggregate's job is to be able to manage our business rules about invariants as they apply to a group of related objects. It's the aggregate's job to check that the objects within its remit are consistent with each other and with our rules, and to reject changes that would break the rules.

Aggregates and concurrency issues go together
When thinking about implementing these consistency checks, we end up thinking about transactions and locks. Choosing the right aggregate is about performance as well as conceptual organization of your domain.

Part I Recap

Do you remember Figure 7-5, the diagram we showed at the beginning of Part I to preview where we were heading?

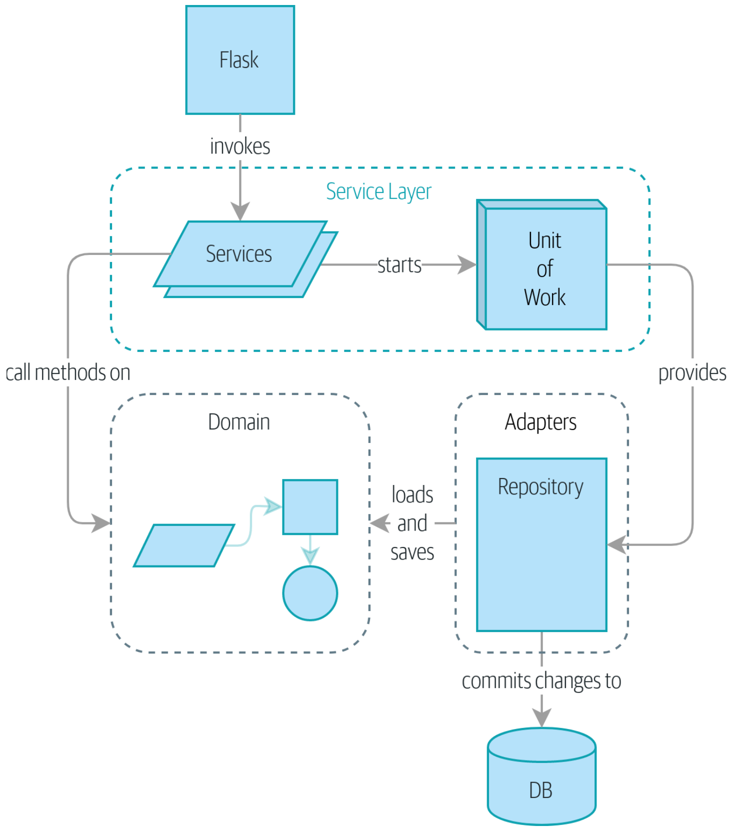


Figure 7-5. A component diagram for our app at the end of Part I

So that's where we are at the end of Part I. What have we achieved? We've seen how to build a domain model that's exercised by a set of high-level unit tests. Our tests are living documentation: they describe the behavior of our system—the rules upon which we agreed with our business stakeholders—in nice readable code. When our business requirements change, we have confidence that our tests will help us to prove the new functionality, and when new developers join the project, they can read our tests to understand how things work.

We've decoupled the infrastructural parts of our system, like the database and API handlers, so that we can plug them into the outside of our application. This helps us to keep our codebase well organized and stops us from building a big ball of mud.

By applying the dependency inversion principle, and by using ports-and-adapters-inspired patterns like Repository and Unit of Work, we've made it possible to do TDD in both high gear and low gear and to maintain a healthy test pyramid. We can test our system edge to edge, and the need for integration and end-to-end tests is kept to a minimum.

Lastly, we've talked about the idea of consistency boundaries. We don't want to lock our entire system whenever we make a change, so we have to choose which parts are consistent with one another.

For a small system, this is everything you need to go and play with the ideas of domain-driven design. You now have the tools to build database-agnostic domain models that represent the shared language of your business experts. Hurrah!

At the risk of laboring the point—we've been at pains to point out that each pattern comes at a cost. Each layer of indirection has a price in terms of complexity and duplication in our code and will be confusing to programmers who've never seen these patterns before. If your app is essentially a simple CRUD wrapper around a database and isn't likely to be anything more than that in the foreseeable future, *you don't need these patterns*. Go ahead and use Django, and save yourself a lot of bother.

In Part II, we'll zoom out and talk about a bigger topic: if aggregates are our boundary, and we can update only one at a time, how do we model processes that cross consistency boundaries?

PART II

Event-Driven Architecture

> I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea.
>
> The big idea is "messaging."…The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
>
> —Alan Kay

It's all very well being able to write *one* domain model to manage a single bit of business process, but what happens when we need to write *many* models? In the real world, our applications sit within an organization and need to exchange information with other parts of the system. You may remember our context diagram shown in Figure II-1.

Faced with this requirement, many teams reach for microservices integrated via HTTP APIs. But if they're not careful, they'll end up producing the most chaotic mess of all: the distributed big ball of mud.

In Part II, we'll show how the techniques from Part I can be extended to distributed systems. We'll zoom out to look at how we can compose a system from many small components that interact through asynchronous message passing.

We'll see how our Service Layer and Unit of Work patterns allow us to reconfigure our app to run as an asynchronous message processor, and how event-driven systems help us to decouple aggregates and applications from one another.

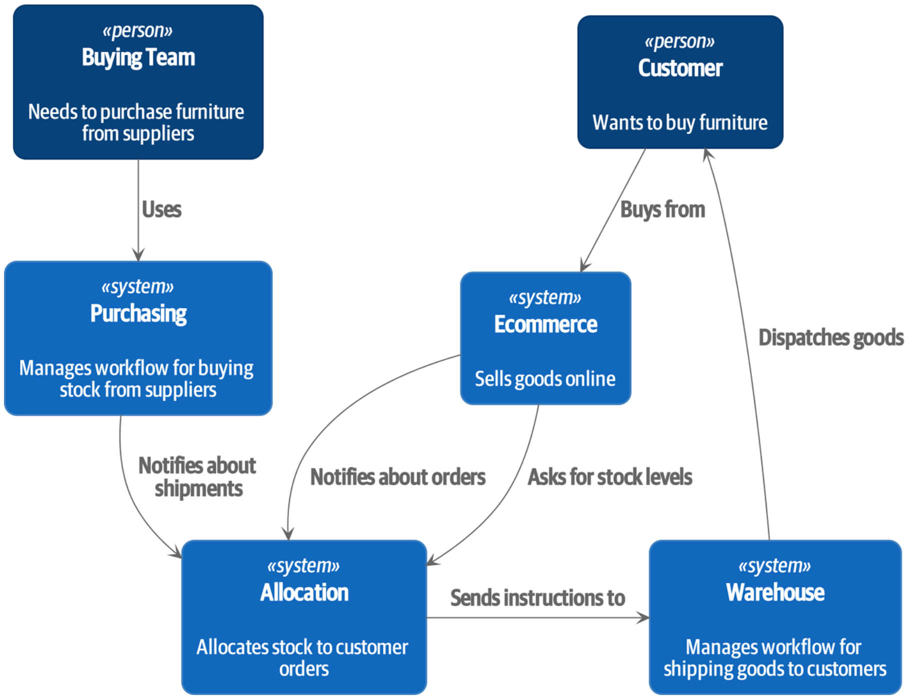


Figure II-1. But exactly how will all these systems talk to each other?

We'll look at the following patterns and techniques:

Domain Events
: Trigger workflows that cross consistency boundaries.

Message Bus
: Provide a unified way of invoking use cases from any endpoint.

CQRS
: Separating reads and writes avoids awkward compromises in an event-driven architecture and enables performance and scalability improvements.

Plus, we'll add a dependency injection framework. This has nothing to do with event-driven architecture per se, but it tidies up an awful lot of loose ends.

CHAPTER 8

Events and the Message Bus

So far we've spent a lot of time and energy on a simple problem that we could easily have solved with Django. You might be asking if the increased testability and expressiveness are *really* worth all the effort.

In practice, though, we find that it's not the obvious features that make a mess of our codebases: it's the goop around the edge. It's reporting, and permissions, and workflows that touch a zillion objects.

Our example will be a typical notification requirement: when we can't allocate an order because we're out of stock, we should alert the buying team. They'll go and fix the problem by buying more stock, and all will be well.

For a first version, our product owner says we can just send the alert by email.

Let's see how our architecture holds up when we need to plug in some of the mundane stuff that makes up so much of our systems.

We'll start by doing the simplest, most expeditious thing, and talk about why it's exactly this kind of decision that leads us to the Big Ball of Mud.

Then we'll show how to use the *Domain Events* pattern to separate side effects from our use cases, and how to use a simple *Message Bus* pattern for triggering behavior based on those events. We'll show a few options for creating those events and how to pass them to the message bus, and finally we'll show how the Unit of Work pattern can be modified to connect the two together elegantly, as previewed in Figure 8-1.

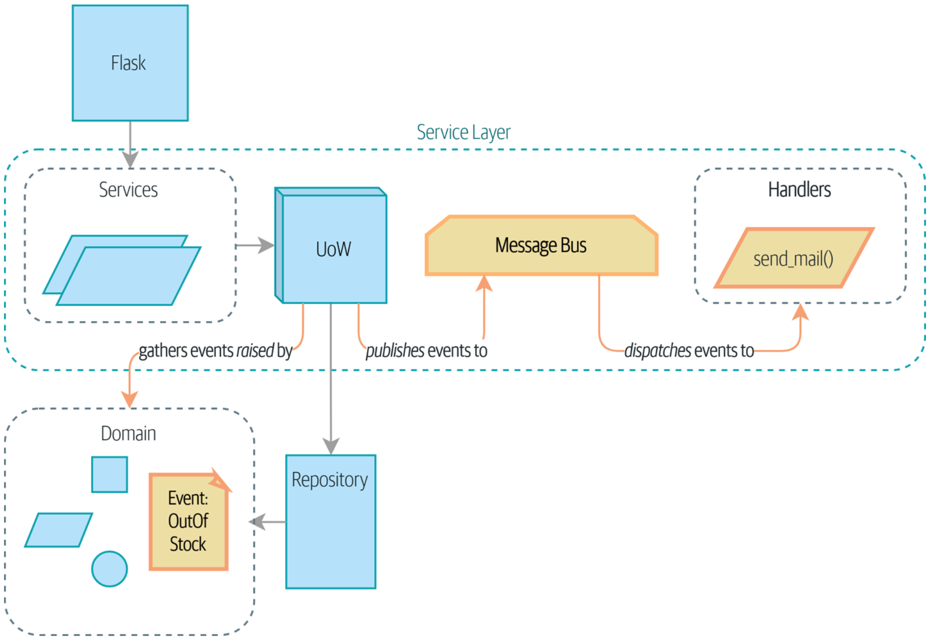


Figure 8-1. Events flowing through the system

The code for this chapter is in the chapter_08_events_and_message_bus branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_08_events_and_message_bus
# or to code along, checkout the previous chapter:
git checkout chapter_07_aggregate
```

Avoiding Making a Mess

So. Email alerts when we run out of stock. When we have new requirements like ones that *really* have nothing to do with the core domain, it's all too easy to start dumping these things into our web controllers.

First, Let's Avoid Making a Mess of Our Web Controllers

As a one-off hack, this *might* be OK:

Just whack it in the endpoint—what could go wrong? (src/allocation/entrypoints/flask_app.py)

```
@app.route("/allocate", methods=['POST'])
def allocate_endpoint():
    line = model.OrderLine(
        request.json['orderid'],
        request.json['sku'],
        request.json['qty'],
    )
    try:
        uow = unit_of_work.SqlAlchemyUnitOfWork()
        batchref = services.allocate(line, uow)
    except (model.OutOfStock, services.InvalidSku) as e:
        send_mail(
            'out of stock',
            'stock_admin@made.com',
            f'{line.orderid} - {line.sku}'
        )
        return jsonify({'message': str(e)}), 400

    return jsonify({'batchref': batchref}), 201
```

…but it's easy to see how we can quickly end up in a mess by patching things up like this. Sending email isn't the job of our HTTP layer, and we'd like to be able to unit test this new feature.

And Let's Not Make a Mess of Our Model Either

Assuming we don't want to put this code into our web controllers, because we want them to be as thin as possible, we may look at putting it right at the source, in the model:

Email-sending code in our model isn't lovely either (src/allocation/domain/model.py)

```
    def allocate(self, line: OrderLine) -> str:
        try:
            batch = next(
                b for b in sorted(self.batches) if b.can_allocate(line)
            )
            #...
        except StopIteration:
            email.send_mail('stock@made.com', f'Out of stock for {line.sku}')
            raise OutOfStock(f'Out of stock for sku {line.sku}')
```

But that's even worse! We don't want our model to have any dependencies on infrastructure concerns like `email.send_mail`.

This email-sending thing is unwelcome *goop* messing up the nice clean flow of our system. What we'd like is to keep our domain model focused on the rule "You can't allocate more stuff than is actually available."

The domain model's job is to know that we're out of stock, but the responsibility of sending an alert belongs elsewhere. We should be able to turn this feature on or off, or to switch to SMS notifications instead, without needing to change the rules of our domain model.

Or the Service Layer!

The requirement “Try to allocate some stock, and send an email if it fails” is an example of workflow orchestration: it’s a set of steps that the system has to follow to achieve a goal.

We’ve written a service layer to manage orchestration for us, but even here the feature feels out of place:

And in the service layer, it’s out of place (src/allocation/service_layer/services.py)

```
def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f'Invalid sku {line.sku}')
        try:
            batchref = product.allocate(line)
            uow.commit()
            return batchref
        except model.OutOfStock:
            email.send_mail('stock@made.com', f'Out of stock for {line.sku}')
            raise
```

Catching an exception and reraising it? It could be worse, but it’s definitely making us unhappy. Why is it so hard to find a suitable home for this code?

Single Responsibility Principle

Really, this is a violation of the *single responsibility principle* (SRP).[1] Our use case is allocation. Our endpoint, service function, and domain methods are all called `allocate`, not `allocate_and_send_mail_if_out_of_stock`.

Rule of thumb: if you can’t describe what your function does without using words like “then” or “and,” you might be violating the SRP.

1 This principle is the *S* in SOLID.

One formulation of the SRP is that each class should have only a single reason to change. When we switch from email to SMS, we shouldn't have to update our `allocate()` function, because that's clearly a separate responsibility.

To solve the problem, we're going to split the orchestration into separate steps so that the different concerns don't get tangled up.[2] The domain model's job is to know that we're out of stock, but the responsibility of sending an alert belongs elsewhere. We should be able to turn this feature on or off, or to switch to SMS notifications instead, without needing to change the rules of our domain model.

We'd also like to keep the service layer free of implementation details. We want to apply the dependency inversion principle to notifications so that our service layer depends on an abstraction, in the same way as we avoid depending on the database by using a unit of work.

All Aboard the Message Bus!

The patterns we're going to introduce here are *Domain Events* and the *Message Bus*. We can implement them in a few ways, so we'll show a couple before settling on the one we like most.

The Model Records Events

First, rather than being concerned about emails, our model will be in charge of recording *events*—facts about things that have happened. We'll use a message bus to respond to events and invoke a new operation.

Events Are Simple Dataclasses

An *event* is a kind of *value object*. Events don't have any behavior, because they're pure data structures. We always name events in the language of the domain, and we think of them as part of our domain model.

We could store them in *model.py*, but we may as well keep them in their own file (this might be a good time to consider refactoring out a directory called *domain* so that we have *domain/model.py* and *domain/events.py*):

2 Our tech reviewer Ed Jung likes to say that the move from imperative to event-based flow control changes what used to be *orchestration* into *choreography*.

Event classes (src/allocation/domain/events.py)

```python
from dataclasses import dataclass

class Event:  ❶
    pass

@dataclass
class OutOfStock(Event):  ❷
    sku: str
```

❶ Once we have a number of events, we'll find it useful to have a parent class that can store common attributes. It's also useful for type hints in our message bus, as you'll see shortly.

❷ `dataclasses` are great for domain events too.

The Model Raises Events

When our domain model records a fact that happened, we say it *raises* an event.

Here's what it will look like from the outside; if we ask `Product` to allocate but it can't, it should *raise* an event:

Test our aggregate to raise events (tests/unit/test_product.py)

```python
def test_records_out_of_stock_event_if_cannot_allocate():
    batch = Batch('batch1', 'SMALL-FORK', 10, eta=today)
    product = Product(sku="SMALL-FORK", batches=[batch])
    product.allocate(OrderLine('order1', 'SMALL-FORK', 10))

    allocation = product.allocate(OrderLine('order2', 'SMALL-FORK', 1))
    assert product.events[-1] == events.OutOfStock(sku="SMALL-FORK")  ❶
    assert allocation is None
```

❶ Our aggregate will expose a new attribute called `.events` that will contain a list of facts about what has happened, in the form of `Event` objects.

Here's what the model looks like on the inside:

The model raises a domain event (src/allocation/domain/model.py)

```python
class Product:

    def __init__(self, sku: str, batches: List[Batch], version_number: int = 0):
        self.sku = sku
        self.batches = batches
        self.version_number = version_number
        self.events = []  # type: List[events.Event]  ❶

    def allocate(self, line: OrderLine) -> str:
        try:
            #...
        except StopIteration:
            self.events.append(events.OutOfStock(line.sku))  ❷
            # raise OutOfStock(f'Out of stock for sku {line.sku}')  ❸
            return None
```

❶ Here's our new `.events` attribute in use.

❷ Rather than invoking some email-sending code directly, we record those events at the place they occur, using only the language of the domain.

❸ We're also going to stop raising an exception for the out-of-stock case. The event will do the job the exception was doing.

We're actually addressing a code smell we had until now, which is that we were using exceptions for control flow. In general, if you're implementing domain events, don't raise exceptions to describe the same domain concept. As you'll see later when we handle events in the Unit of Work pattern, it's confusing to have to reason about events and exceptions together.

The Message Bus Maps Events to Handlers

A message bus basically says, "When I see this event, I should invoke the following handler function." In other words, it's a simple publish-subscribe system. Handlers are *subscribed* to receive events, which we publish to the bus. It sounds harder than it is, and we usually implement it with a dict:

Simple message bus (src/allocation/service_layer/messagebus.py)

```
def handle(event: events.Event):
    for handler in HANDLERS[type(event)]:
        handler(event)


def send_out_of_stock_notification(event: events.OutOfStock):
    email.send_mail(
        'stock@made.com',
        f'Out of stock for {event.sku}',
    )


HANDLERS = {
    events.OutOfStock: [send_out_of_stock_notification],

}  # type: Dict[Type[events.Event], List[Callable]]
```

Note that the message bus as implemented doesn't give us concurrency because only one handler will run at a time. Our objective isn't to support parallel threads but to separate tasks conceptually, and to keep each UoW as small as possible. This helps us to understand the codebase because the "recipe" for how to run each use case is written in a single place. See the following sidebar.

Is This Like Celery?

Celery is a popular tool in the Python world for deferring self-contained chunks of work to an asynchronous task queue. The message bus we're presenting here is very different, so the short answer to the above question is no; our message bus has more in common with a Node.js app, a UI event loop, or an actor framework.

If you do have a requirement for moving work off the main thread, you can still use our event-based metaphors, but we suggest you use *external events* for that. There's more discussion in Table 11-1, but essentially, if you implement a way of persisting events to a centralized store, you can subscribe other containers or other microservices to them. Then that same concept of using events to separate responsibilities across units of work within a single process/service can be extended across multiple processes—which may be different containers within the same service, or totally different microservices.

If you follow us in this approach, your API for distributing tasks is your event classes—or a JSON representation of them. This allows you a lot of flexibility in who you distribute tasks to; they need not necessarily be Python services. Celery's API for distributing tasks is essentially "function name plus arguments," which is more restrictive, and Python-only.

Option 1: The Service Layer Takes Events from the Model and Puts Them on the Message Bus

Our domain model raises events, and our message bus will call the right handlers whenever an event happens. Now all we need is to connect the two. We need something to catch events from the model and pass them to the message bus—the *publishing* step.

The simplest way to do this is by adding some code into our service layer:

The service layer with an explicit message bus (src/allocation/service_layer/services.py)

```
from . import messagebus
...

def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f'Invalid sku {line.sku}')
        try:  ❶
            batchref = product.allocate(line)
            uow.commit()
            return batchref
        finally:  ❶
            messagebus.handle(product.events)  ❷
```

❶ We keep the `try/finally` from our ugly earlier implementation (we haven't gotten rid of *all* exceptions yet, just `OutOfStock`).

❷ But now, instead of depending directly on an email infrastructure, the service layer is just in charge of passing events from the model up to the message bus.

That already avoids some of the ugliness that we had in our naive implementation, and we have several systems that work like this one, in which the service layer explicitly collects events from aggregates and passes them to the message bus.

Option 2: The Service Layer Raises Its Own Events

Another variant on this that we've used is to have the service layer in charge of creating and raising events directly, rather than having them raised by the domain model:

Service layer calls messagebus.handle directly (src/allocation/service_layer/services.py)

```
def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f'Invalid sku {line.sku}')
        batchref = product.allocate(line)
        uow.commit() ❶

        if batchref is None:
            messagebus.handle(events.OutOfStock(line.sku))
        return batchref
```

❶ As before, we commit even if we fail to allocate because the code is simpler this way and it's easier to reason about: we always commit unless something goes wrong. Committing when we haven't changed anything is safe and keeps the code uncluttered.

Again, we have applications in production that implement the pattern in this way. What works for you will depend on the particular trade-offs you face, but we'd like to show you what we think is the most elegant solution, in which we put the unit of work in charge of collecting and raising events.

Option 3: The UoW Publishes Events to the Message Bus

The UoW already has a `try/finally`, and it knows about all the aggregates currently in play because it provides access to the repository. So it's a good place to spot events and pass them to the message bus:

The UoW meets the message bus (src/allocation/service_layer/unit_of_work.py)

```
class AbstractUnitOfWork(abc.ABC):
    ...

    def commit(self):
        self._commit()  ❶
        self.publish_events()  ❷

    def publish_events(self):  ❷
        for product in self.products.seen:  ❸
            while product.events:
                event = product.events.pop(0)
                messagebus.handle(event)

    @abc.abstractmethod
    def _commit(self):
        raise NotImplementedError

...

class SqlAlchemyUnitOfWork(AbstractUnitOfWork):
    ...

    def _commit(self):  ❶
        self.session.commit()
```

❶ We'll change our commit method to require a private `._commit()` method from subclasses.

❷ After committing, we run through all the objects that our repository has seen and pass their events to the message bus.

❸ That relies on the repository keeping track of aggregates that have been loaded using a new attribute, `.seen`, as you'll see in the next listing.

Are you wondering what happens if one of the handlers fails? We'll discuss error handling in detail in Chapter 10.

Repository tracks aggregates that pass through it (src/allocation/adapters/repository.py)

```python
class AbstractRepository(abc.ABC):

    def __init__(self):
        self.seen = set()  # type: Set[model.Product]  ❶

    def add(self, product: model.Product):  ❷
        self._add(product)
        self.seen.add(product)

    def get(self, sku) -> model.Product:  ❸
        product = self._get(sku)
        if product:
            self.seen.add(product)
        return product

    @abc.abstractmethod
    def _add(self, product: model.Product):  ❷
        raise NotImplementedError

    @abc.abstractmethod  ❸
    def _get(self, sku) -> model.Product:
        raise NotImplementedError



class SqlAlchemyRepository(AbstractRepository):

    def __init__(self, session):
        super().__init__()
        self.session = session

    def _add(self, product):  ❷
        self.session.add(product)

    def _get(self, sku):  ❸
        return self.session.query(model.Product).filter_by(sku=sku).first()
```

❶ For the UoW to be able to publish new events, it needs to be able to ask the repository for which `Product` objects have been used during this session. We use a `set` called `.seen` to store them. That means our implementations need to call `super().__init__()`.

❷ The parent `add()` method adds things to `.seen`, and now requires subclasses to implement `._add()`.

❸ Similarly, `.get()` delegates to a `._get()` function, to be implemented by subclasses, in order to capture objects seen.

The use of `._underscorey()` methods and subclassing is definitely not the only way you could implement these patterns. Have a go at the Exercise for the Reader in this chapter and experiment with some alternatives.

After the UoW and repository collaborate in this way to automatically keep track of live objects and process their events, the service layer can be totally free of event-handling concerns:

Service layer is clean again (src/allocation/service_layer/services.py)

```python
def allocate(
        orderid: str, sku: str, qty: int,
        uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f'Invalid sku {line.sku}')
        batchref = product.allocate(line)
        uow.commit()
        return batchref
```

We do also have to remember to change the fakes in the service layer and make them call `super()` in the right places, and to implement underscorey methods, but the changes are minimal:

Service-layer fakes need tweaking (tests/unit/test_services.py)

```python
class FakeRepository(repository.AbstractRepository):

    def __init__(self, products):
        super().__init__()
        self._products = set(products)

    def _add(self, product):
        self._products.add(product)
    def _get(self, sku):
        return next((p for p in self._products if p.sku == sku), None)

...

class FakeUnitOfWork(unit_of_work.AbstractUnitOfWork):
    ...

    def _commit(self):
        self.committed = True
```

Exercise for the Reader

Are you finding all those `._add()` and `._commit()` methods "super-gross," in the words of our beloved tech reviewer Hynek? Does it "make you want to beat Harry around the head with a plushie snake"? Hey, our code listings are only meant to be examples, not the perfect solution! Why not go see if you can do better?

One *composition over inheritance* way to go would be to implement a wrapper class:

A wrapper adds functionality and then delegates (src/adapters/repository.py)

```
class TrackingRepository:
    seen: Set[model.Product]

    def __init__(self, repo: AbstractRepository):
        self.seen = set()  # type: Set[model.Product]
        self._repo = repo

    def add(self, product: model.Product):  ❶
        self._repo.add(product)  ❶
        self.seen.add(product)

    def get(self, sku) -> model.Product:
        product = self._repo.get(sku)
        if product:
            self.seen.add(product)
        return product
```

❶ By wrapping the repository, we can call the actual `.add()` and `.get()` methods, avoiding weird underscorey methods.

See if you can apply a similar pattern to our UoW class in order to get rid of those Java-y `_commit()` methods too. You can find the code on GitHub.

Switching all the ABCs to `typing.Protocol` is a good way to force yourself to avoid using inheritance. Let us know if you come up with something nice!

You might be starting to worry that maintaining these fakes is going to be a maintenance burden. There's no doubt that it is work, but in our experience it's not a lot of work. Once your project is up and running, the interface for your repository and UoW abstractions really don't change much. And if you're using ABCs, they'll help remind you when things get out of sync.

Wrap-Up

Domain events give us a way to handle workflows in our system. We often find, listening to our domain experts, that they express requirements in a causal or temporal way—for example, "When we try to allocate stock but there's none available, then we should send an email to the buying team."

The magic words "When X, then Y" often tell us about an event that we can make concrete in our system. Treating events as first-class things in our model helps us make our code more testable and observable, and it helps isolate concerns.

And Table 8-1 shows the trade-offs as we see them.

Table 8-1. Domain events: the trade-offs

Pros	Cons
• A message bus gives us a nice way to separate responsibilities when we have to take multiple actions in response to a request. • Event handlers are nicely decoupled from the "core" application logic, making it easy to change their implementation later. • Domain events are a great way to model the real world, and we can use them as part of our business language when modeling with stakeholders.	• The message bus is an additional thing to wrap your head around; the implementation in which the unit of work raises events for us is *neat* but also magic. It's not obvious when we call `commit` that we're also going to go and send email to people. • What's more, that hidden event-handling code executes *synchronously*, meaning your service-layer function doesn't finish until all the handlers for any events are finished. That could cause unexpected performance problems in your web endpoints (adding asynchronous processing is possible but makes things even *more* confusing). • More generally, event-driven workflows can be confusing because after things are split across a chain of multiple handlers, there is no single place in the system where you can understand how a request will be fulfilled. • You also open yourself up to the possibility of circular dependencies between your event handlers, and infinite loops.

Events are useful for more than just sending email, though. In Chapter 7 we spent a lot of time convincing you that you should define aggregates, or boundaries where we guarantee consistency. People often ask, "What should I do if I need to change multiple aggregates as part of a request?" Now we have the tools we need to answer that question.

If we have two things that can be transactionally isolated (e.g., an order and a product), then we can make them *eventually consistent* by using events. When an order is canceled, we should find the products that were allocated to it and remove the allocations.

Domain Events and the Message Bus Recap

Events can help with the single responsibility principle
: Code gets tangled up when we mix multiple concerns in one place. Events can help us to keep things tidy by separating primary use cases from secondary ones. We also use events for communicating between aggregates so that we don't need to run long-running transactions that lock against multiple tables.

A message bus routes messages to handlers
: You can think of a message bus as a dict that maps from events to their consumers. It doesn't "know" anything about the meaning of events; it's just a piece of dumb infrastructure for getting messages around the system.

Option 1: Service layer raises events and passes them to message bus
: The simplest way to start using events in your system is to raise them from handlers by calling `bus.handle(some_new_event)` after you commit your unit of work.

Option 2: Domain model raises events, service layer passes them to message bus
: The logic about when to raise an event really should live with the model, so we can improve our system's design and testability by raising events from the domain model. It's easy for our handlers to collect events off the model objects after `commit` and pass them to the bus.

Option 3: UoW collects events from aggregates and passes them to message bus
: Adding `bus.handle(aggregate.events)` to every handler is annoying, so we can tidy up by making our unit of work responsible for raising events that were raised by loaded objects. This is the most complex design and might rely on ORM magic, but it's clean and easy to use once it's set up.

In Chapter 9, we'll look at this idea in more detail as we build a more complex workflow with our new message bus.

CHAPTER 9

Going to Town on the Message Bus

In this chapter, we'll start to make events more fundamental to the internal structure of our application. We'll move from the current state in Figure 9-1, where events are an optional side effect…

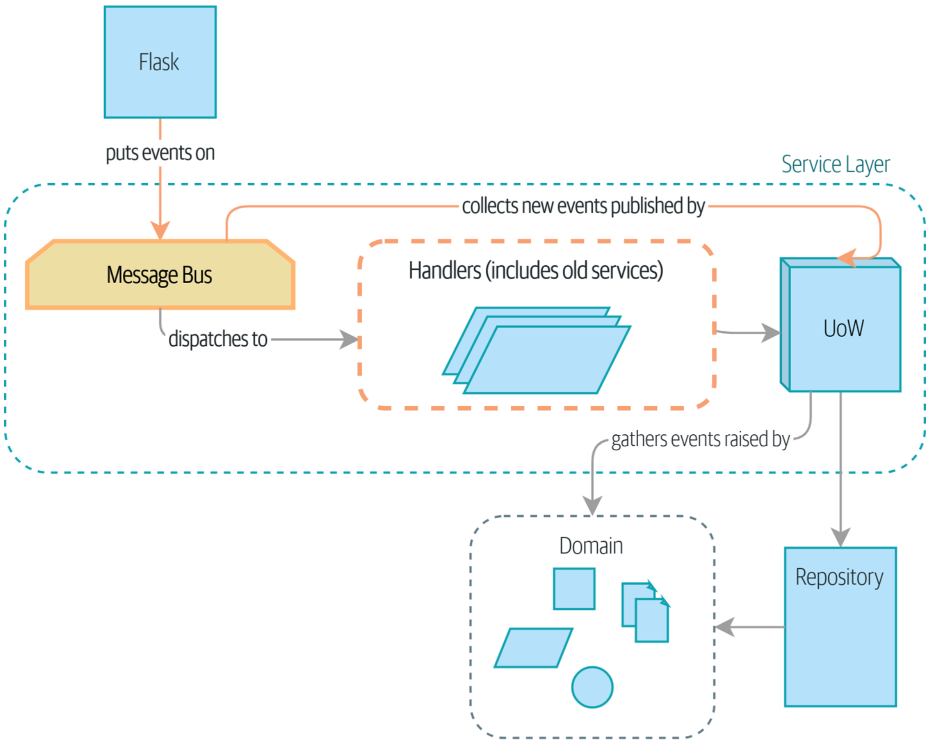


Figure 9-1. Before: the message bus is an optional add-on

…to the situation in Figure 9-2, where everything goes via the message bus, and our app has been transformed fundamentally into a message processor.

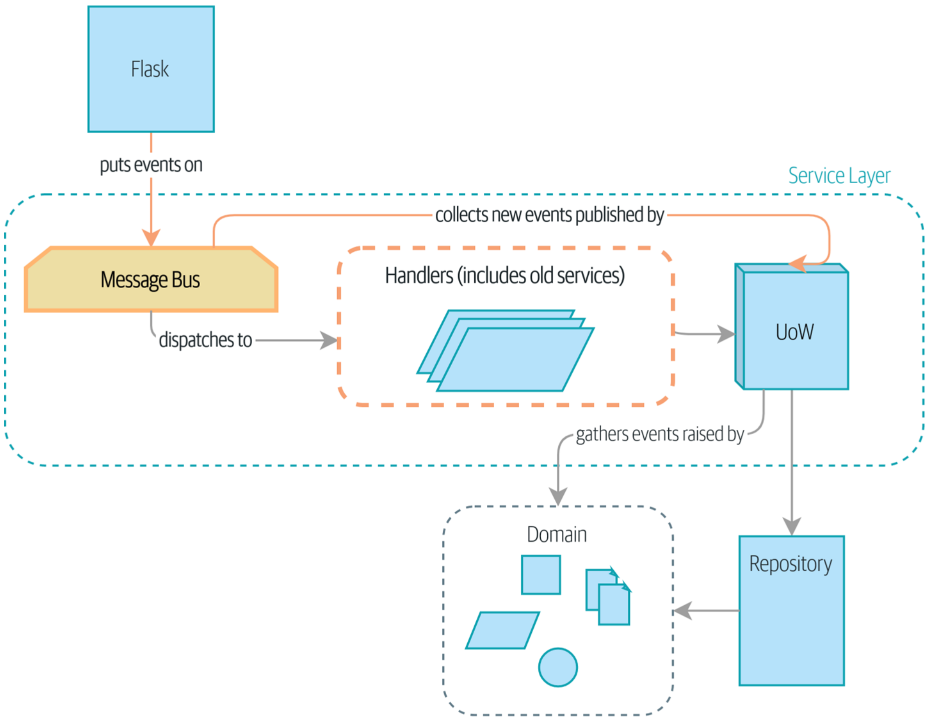


Figure 9-2. The message bus is now the main entrypoint to the service layer

The code for this chapter is in the chapter_09_all_messagebus branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_09_all_messagebus
# or to code along, checkout the previous chapter:
git checkout chapter_08_events_and_message_bus
```

A New Requirement Leads Us to a New Architecture

Rich Hickey talks about *situated software*, meaning software that runs for extended periods of time, managing a real-world process. Examples include warehouse-management systems, logistics schedulers, and payroll systems.

This software is tricky to write because unexpected things happen all the time in the real world of physical objects and unreliable humans. For example:

- During a stock-take, we discover that three `SPRINGY-MATTRESS`es have been water damaged by a leaky roof.
- A consignment of `RELIABLE-FORK`s is missing the required documentation and is held in customs for several weeks. Three `RELIABLE-FORK`s subsequently fail safety testing and are destroyed.
- A global shortage of sequins means we're unable to manufacture our next batch of `SPARKLY-BOOKCASE`.

In these types of situations, we learn about the need to change batch quantities when they're already in the system. Perhaps someone made a mistake on the number in the manifest, or perhaps some sofas fell off a truck. Following a conversation with the business,[1] we model the situation as in Figure 9-3.

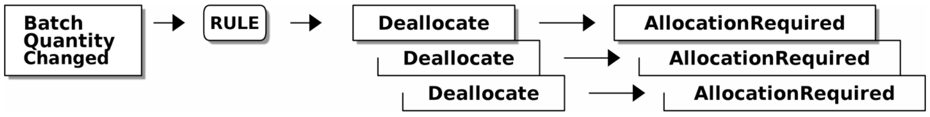


Figure 9-3. Batch quantity changed means deallocate and reallocate

An event we'll call `BatchQuantityChanged` should lead us to change the quantity on the batch, yes, but also to apply a *business rule*: if the new quantity drops to less than the total already allocated, we need to *deallocate* those orders from that batch. Then each one will require a new allocation, which we can capture as an event called `AllocationRequired`.

Perhaps you're already anticipating that our internal message bus and events can help implement this requirement. We could define a service called `change_batch_quantity` that knows how to adjust batch quantities and also how to *deallocate* any excess order lines, and then each deallocation can emit an `AllocationRequired` event that

1 Event-based modeling is so popular that a practice called *event storming* has been developed for facilitating event-based requirements gathering and domain model elaboration.

can be forwarded to the existing `allocate` service, in separate transactions. Once again, our message bus helps us to enforce the single responsibility principle, and it allows us to make choices about transactions and data integrity.

Imagining an Architecture Change: Everything Will Be an Event Handler

But before we jump in, think about where we're headed. There are two kinds of flows through our system:

- API calls that are handled by a service-layer function
- Internal events (which might be raised as a side effect of a service-layer function) and their handlers (which in turn call service-layer functions)

Wouldn't it be easier if everything was an event handler? If we rethink our API calls as capturing events, the service-layer functions can be event handlers too, and we no longer need to make a distinction between internal and external event handlers:

- `services.allocate()` could be the handler for an `AllocationRequired` event and could emit `Allocated` events as its output.
- `services.add_batch()` could be the handler for a `BatchCreated` event.[2]

Our new requirement will fit the same pattern:

- An event called `BatchQuantityChanged` can invoke a handler called `change_batch_quantity()`.
- And the new `AllocationRequired` events that it may raise can be passed on to `services.allocate()` too, so there is no conceptual difference between a brand-new allocation coming from the API and a reallocation that's internally triggered by a deallocation.

All sound like a bit much? Let's work toward it all gradually. We'll follow the Preparatory Refactoring workflow, aka "Make the change easy; then make the easy change":

1. We refactor our service layer into event handlers. We can get used to the idea of events being the way we describe inputs to the system. In particular, the existing `services.allocate()` function will become the handler for an event called `AllocationRequired`.

2 If you've done a bit of reading about event-driven architectures, you may be thinking, "Some of these events sound more like commands!" Bear with us! We're trying to introduce one concept at a time. In the next chapter, we'll introduce the distinction between commands and events.

2. We build an end-to-end test that puts `BatchQuantityChanged` events into the system and looks for `Allocated` events coming out.
3. Our implementation will conceptually be very simple: a new handler for `BatchQuantityChanged` events, whose implementation will emit `AllocationRequired` events, which in turn will be handled by the exact same handler for allocations that the API uses.

Along the way, we'll make a small tweak to the message bus and UoW, moving the responsibility for putting new events on the message bus into the message bus itself.

Refactoring Service Functions to Message Handlers

We start by defining the two events that capture our current API inputs—`AllocationRequired` and `BatchCreated`:

BatchCreated and AllocationRequired events (src/allocation/domain/events.py)

```
@dataclass
class BatchCreated(Event):
    ref: str
    sku: str
    qty: int
    eta: Optional[date] = None

...

@dataclass
class AllocationRequired(Event):
    orderid: str
    sku: str
    qty: int
```

Then we rename *services.py* to *handlers.py*; we add the existing message handler for `send_out_of_stock_notification`; and most importantly, we change all the handlers so that they have the same inputs, an event and a UoW:

Handlers and services are the same thing (src/allocation/service_layer/handlers.py)

```
def add_batch(
        event: events.BatchCreated, uow: unit_of_work.AbstractUnitOfWork
):
    with uow:
        product = uow.products.get(sku=event.sku)
        ...


def allocate(
        event: events.AllocationRequired, uow: unit_of_work.AbstractUnitOfWork
) -> str:
    line = OrderLine(event.orderid, event.sku, event.qty)
    ...


def send_out_of_stock_notification(
        event: events.OutOfStock, uow: unit_of_work.AbstractUnitOfWork,
):
    email.send(
        'stock@made.com',
        f'Out of stock for {event.sku}',
    )
```

The change might be clearer as a diff:

Changing from services to handlers (src/allocation/service_layer/handlers.py)

```
 def add_batch(
-        ref: str, sku: str, qty: int, eta: Optional[date],
-        uow: unit_of_work.AbstractUnitOfWork
+        event: events.BatchCreated, uow: unit_of_work.AbstractUnitOfWork
 ):
     with uow:
-        product = uow.products.get(sku=sku)
+        product = uow.products.get(sku=event.sku)
     ...


 def allocate(
-        orderid: str, sku: str, qty: int,
-        uow: unit_of_work.AbstractUnitOfWork
+        event: events.AllocationRequired, uow: unit_of_work.AbstractUnitOfWork
 ) -> str:
-    line = OrderLine(orderid, sku, qty)
+    line = OrderLine(event.orderid, event.sku, event.qty)
     ...

+
+def send_out_of_stock_notification(
+        event: events.OutOfStock, uow: unit_of_work.AbstractUnitOfWork,
+):
+    email.send(
     ...
```

Along the way, we've made our service-layer's API more structured and more consistent. It was a scattering of primitives, and now it uses well-defined objects (see the following sidebar).

From Domain Objects, via Primitive Obsession, to Events as an Interface

Some of you may remember "Fully Decoupling the Service-Layer Tests from the Domain" on page 75, in which we changed our service-layer API from being in terms of domain objects to primitives. And now we're moving back, but to different objects? What gives?

In OO circles, people talk about *primitive obsession* as an anti-pattern: avoid primitives in public APIs, and instead wrap them with custom value classes, they would say. In the Python world, a lot of people would be quite skeptical of that as a rule of thumb. When mindlessly applied, it's certainly a recipe for unnecessary complexity. So that's not what we're doing per se.

The move from domain objects to primitives bought us a nice bit of decoupling: our client code was no longer coupled directly to the domain, so the service layer could present an API that stays the same even if we decide to make changes to our model, and vice versa.

So have we gone backward? Well, our core domain model objects are still free to vary, but instead we've coupled the external world to our event classes. They're part of the domain too, but the hope is that they vary less often, so they're a sensible artifact to couple on.

And what have we bought ourselves? Now, when invoking a use case in our application, we no longer need to remember a particular combination of primitives, but just a single event class that represents the input to our application. That's conceptually quite nice. On top of that, as you'll see in Appendix E, those event classes can be a nice place to do some input validation.

The Message Bus Now Collects Events from the UoW

Our event handlers now need a UoW. In addition, as our message bus becomes more central to our application, it makes sense to put it explicitly in charge of collecting and processing new events. There was a bit of a circular dependency between the UoW and message bus until now, so this will make it one-way:

Handle takes a UoW and manages a queue (src/allocation/service_layer/messagebus.py)

```
def handle(event: events.Event, uow: unit_of_work.AbstractUnitOfWork):  ❶
    queue = [event]  ❷
    while queue:
        event = queue.pop(0)  ❸
        for handler in HANDLERS[type(event)]:  ❸
            handler(event, uow=uow)  ❹
            queue.extend(uow.collect_new_events())  ❺
```

❶ The message bus now gets passed the UoW each time it starts up.

❷ When we begin handling our first event, we start a queue.

❸ We pop events from the front of the queue and invoke their handlers (the HANDLERS dict hasn't changed; it still maps event types to handler functions).

❹ The message bus passes the UoW down to each handler.

❺ After each handler finishes, we collect any new events that have been generated and add them to the queue.

In *unit_of_work.py*, `publish_events()` becomes a less active method, `collect_new_events()`:

UoW no longer puts events directly on the bus (src/allocation/service_layer/unit_of_work.py)

```
-from . import messagebus  ❶
-


 class AbstractUnitOfWork(abc.ABC):
@@ -23,13 +21,11 @@ class AbstractUnitOfWork(abc.ABC):

     def commit(self):
         self._commit()
-        self.publish_events()  ❷

-    def publish_events(self):
+    def collect_new_events(self):
         for product in self.products.seen:
             while product.events:
-                event = product.events.pop(0)
-                messagebus.handle(event)
+                yield product.events.pop(0)  ❸
```

❶ The `unit_of_work` module now no longer depends on `messagebus`.

❷ We no longer `publish_events` automatically on commit. The message bus is keeping track of the event queue instead.

❸ And the UoW no longer actively puts events on the message bus; it just makes them available.

Our Tests Are All Written in Terms of Events Too

Our tests now operate by creating events and putting them on the message bus, rather than invoking service-layer functions directly:

Handler tests use events (tests/unit/test_handlers.py)

```
class TestAddBatch:

    def test_for_new_product(self):
        uow = FakeUnitOfWork()
-       services.add_batch("b1", "CRUNCHY-ARMCHAIR", 100, None, uow)
+       messagebus.handle(
+           events.BatchCreated("b1", "CRUNCHY-ARMCHAIR", 100, None), uow
+       )
        assert uow.products.get("CRUNCHY-ARMCHAIR") is not None
        assert uow.committed

...

 class TestAllocate:

    def test_returns_allocation(self):
        uow = FakeUnitOfWork()
-       services.add_batch("batch1", "COMPLICATED-LAMP", 100, None, uow)
-       result = services.allocate("o1", "COMPLICATED-LAMP", 10, uow)
+       messagebus.handle(
+           events.BatchCreated("batch1", "COMPLICATED-LAMP", 100, None), uow
+       )
+       result = messagebus.handle(
+           events.AllocationRequired("o1", "COMPLICATED-LAMP", 10), uow
+       )
        assert result == "batch1"
```

A Temporary Ugly Hack: The Message Bus Has to Return Results

Our API and our service layer currently want to know the allocated batch reference when they invoke our `allocate()` handler. This means we need to put in a temporary hack on our message bus to let it return events:

Message bus returns results (src/allocation/service_layer/messagebus.py)

```
 def handle(event: events.Event, uow: unit_of_work.AbstractUnitOfWork):
+   results = []
    queue = [event]
    while queue:
        event = queue.pop(0)
        for handler in HANDLERS[type(event)]:
-           handler(event, uow=uow)
+           results.append(handler(event, uow=uow))
```

```
             queue.extend(uow.collect_new_events())
+    return results
```

It's because we're mixing the read and write responsibilities in our system. We'll come back to fix this wart in Chapter 12.

Modifying Our API to Work with Events

Flask changing to message bus as a diff (src/allocation/entrypoints/flask_app.py)

```
 @app.route("/allocate", methods=['POST'])
 def allocate_endpoint():
     try:
-        batchref = services.allocate(
-            request.json['orderid'],  (1)
-            request.json['sku'],
-            request.json['qty'],
-            unit_of_work.SqlAlchemyUnitOfWork(),
+        event = events.AllocationRequired(  (2)
+            request.json['orderid'], request.json['sku'], request.json['qty'],
         )
+        results = messagebus.handle(event, unit_of_work.SqlAlchemyUnitOfWork())  (3)
+        batchref = results.pop(0)
     except InvalidSku as e:
```

(1) Instead of calling the service layer with a bunch of primitives extracted from the request JSON…

(2) We instantiate an event.

(3) Then we pass it to the message bus.

And we should be back to a fully functional application, but one that's now fully event-driven:

- What used to be service-layer functions are now event handlers.
- That makes them the same as the functions we invoke for handling internal events raised by our domain model.
- We use events as our data structure for capturing inputs to the system, as well as for handing off of internal work packages.
- The entire app is now best described as a message processor, or an event processor if you prefer. We'll talk about the distinction in the next chapter.

Implementing Our New Requirement

We're done with our refactoring phase. Let's see if we really have "made the change easy." Let's implement our new requirement, shown in Figure 9-4: we'll receive as our inputs some new `BatchQuantityChanged` events and pass them to a handler, which in turn might emit some `AllocationRequired` events, and those in turn will go back to our existing handler for reallocation.

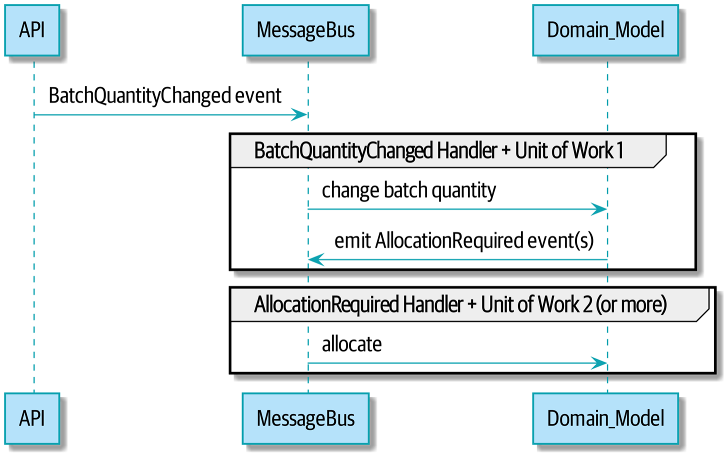

API
MessageBus
Domain_Model
BatchQuantityChanged event
BatchQuantityChanged Handler + Unit of Work 1
change batch quantity
emit AllocationRequired event(s)
AllocationRequired Handler + Unit of Work 2 (or more)
allocate
API
MessageBus
Domain_Model

Figure 9-4. Sequence diagram for reallocation flow

When you split things out like this across two units of work, you now have two database transactions, so you are opening yourself up to integrity issues: something could happen that means the first transaction completes but the second one does not. You'll need to think about whether this is acceptable, and whether you need to notice when it happens and do something about it. See "Footguns" on page 226 for more discussion.

Our New Event

The event that tells us a batch quantity has changed is simple; it just needs a batch reference and a new quantity:

New event (src/allocation/domain/events.py)

```
@dataclass
class BatchQuantityChanged(Event):
    ref: str
    qty: int
```

Test-Driving a New Handler

Following the lessons learned in Chapter 4, we can operate in "high gear" and write our unit tests at the highest possible level of abstraction, in terms of events. Here's what they might look like:

Handler tests for change_batch_quantity (tests/unit/test_handlers.py)

```
class TestChangeBatchQuantity:

    def test_changes_available_quantity(self):
        uow = FakeUnitOfWork()
        messagebus.handle(
            events.BatchCreated("batch1", "ADORABLE-SETTEE", 100, None), uow
        )
        [batch] = uow.products.get(sku="ADORABLE-SETTEE").batches
        assert batch.available_quantity == 100  ❶

        messagebus.handle(events.BatchQuantityChanged("batch1", 50), uow)

        assert batch.available_quantity == 50  ❶


    def test_reallocates_if_necessary(self):
        uow = FakeUnitOfWork()
        event_history = [
            events.BatchCreated("batch1", "INDIFFERENT-TABLE", 50, None),
            events.BatchCreated("batch2", "INDIFFERENT-TABLE", 50, date.today()),
            events.AllocationRequired("order1", "INDIFFERENT-TABLE", 20),
            events.AllocationRequired("order2", "INDIFFERENT-TABLE", 20),
        ]
        for e in event_history:
            messagebus.handle(e, uow)
        [batch1, batch2] = uow.products.get(sku="INDIFFERENT-TABLE").batches
        assert batch1.available_quantity == 10
        assert batch2.available_quantity == 50

        messagebus.handle(events.BatchQuantityChanged("batch1", 25), uow)

        # order1 or order2 will be deallocated, so we'll have 25 - 20
        assert batch1.available_quantity == 5  ❷
        # and 20 will be reallocated to the next batch
        assert batch2.available_quantity == 30  ❷
```

❶ The simple case would be trivially easy to implement; we just modify a quantity.

❷ But if we try to change the quantity to less than has been allocated, we'll need to deallocate at least one order, and we expect to reallocate it to a new batch.

Implementation

Our new handler is very simple:

Handler delegates to model layer (src/allocation/service_layer/handlers.py)

```
def change_batch_quantity(
        event: events.BatchQuantityChanged, uow: unit_of_work.AbstractUnitOfWork
):
    with uow:
        product = uow.products.get_by_batchref(batchref=event.ref)
        product.change_batch_quantity(ref=event.ref, qty=event.qty)
        uow.commit()
```

We realize we'll need a new query type on our repository:

A new query type on our repository (src/allocation/adapters/repository.py)

```
class AbstractRepository(abc.ABC):
    ...

    def get(self, sku) -> model.Product:
        ...

    def get_by_batchref(self, batchref) -> model.Product:
        product = self._get_by_batchref(batchref)
        if product:
            self.seen.add(product)
        return product

    @abc.abstractmethod
    def _add(self, product: model.Product):
        raise NotImplementedError

    @abc.abstractmethod
    def _get(self, sku) -> model.Product:
        raise NotImplementedError

    @abc.abstractmethod
    def _get_by_batchref(self, batchref) -> model.Product:
        raise NotImplementedError
    ...

class SqlAlchemyRepository(AbstractRepository):
    ...

    def _get(self, sku):
        return self.session.query(model.Product).filter_by(sku=sku).first()

    def _get_by_batchref(self, batchref):
        return self.session.query(model.Product).join(model.Batch).filter(
            orm.batches.c.reference == batchref,
        ).first()
```

And on our `FakeRepository` too:

Updating the fake repo too (tests/unit/test_handlers.py)

```
class FakeRepository(repository.AbstractRepository):
    ...

    def _get(self, sku):
        return next((p for p in self._products if p.sku == sku), None)

    def _get_by_batchref(self, batchref):
        return next((
            p for p in self._products for b in p.batches
            if b.reference == batchref
        ), None)
```

We're adding a query to our repository to make this use case easier to implement. So long as our query is returning a single aggregate, we're not bending any rules. If you find yourself writing complex queries on your repositories, you might want to consider a different design. Methods like `get_most_popular_products` or `find_products_by_order_id` in particular would definitely trigger our spidey sense. Chapter 11 and the epilogue have some tips on managing complex queries.

A New Method on the Domain Model

We add the new method to the model, which does the quantity change and deallocation(s) inline and publishes a new event. We also modify the existing allocate function to publish an event:

Our model evolves to capture the new requirement (src/allocation/domain/model.py)

```
class Product:
    ...

    def change_batch_quantity(self, ref: str, qty: int):
        batch = next(b for b in self.batches if b.reference == ref)
        batch._purchased_quantity = qty
        while batch.available_quantity < 0:
            line = batch.deallocate_one()
            self.events.append(
                events.AllocationRequired(line.orderid, line.sku, line.qty)
            )
...

class Batch:
    ...
    def deallocate_one(self) -> OrderLine:
        return self._allocations.pop()
```

We wire up our new handler:

The message bus grows (src/allocation/service_layer/messagebus.py)

```
HANDLERS = {
    events.BatchCreated: [handlers.add_batch],
    events.BatchQuantityChanged: [handlers.change_batch_quantity],
    events.AllocationRequired: [handlers.allocate],
    events.OutOfStock: [handlers.send_out_of_stock_notification],

}  # type: Dict[Type[events.Event], List[Callable]]
```

And our new requirement is fully implemented.

Optionally: Unit Testing Event Handlers in Isolation with a Fake Message Bus

Our main test for the reallocation workflow is *edge-to-edge* (see the example code in "Test-Driving a New Handler" on page 144). It uses the real message bus, and it tests the whole flow, where the `BatchQuantityChanged` event handler triggers deallocation, and emits new `AllocationRequired` events, which in turn are handled by their own handlers. One test covers a chain of multiple events and handlers.

Depending on the complexity of your chain of events, you may decide that you want to test some handlers in isolation from one another. You can do this using a "fake" message bus.

In our case, we actually intervene by modifying the `publish_events()` method on `FakeUnitOfWork` and decoupling it from the real message bus, instead making it record what events it sees:

Fake message bus implemented in UoW (tests/unit/test_handlers.py)

```
class FakeUnitOfWorkWithFakeMessageBus(FakeUnitOfWork):

    def __init__(self):
        super().__init__()
        self.events_published = []  # type: List[events.Event]

    def publish_events(self):
        for product in self.products.seen:
            while product.events:
                self.events_published.append(product.events.pop(0))
```

Now when we invoke `messagebus.handle()` using the `FakeUnitOfWorkWithFakeMessageBus`, it runs only the handler for that event. So we can write a more isolated unit test: instead of checking all the side effects, we just check that `BatchQuantityChanged` leads to `AllocationRequired` if the quantity drops below the total already allocated:

Testing reallocation in isolation (tests/unit/test_handlers.py)

```python
def test_reallocates_if_necessary_isolated():
    uow = FakeUnitOfWorkWithFakeMessageBus()

    # test setup as before
    event_history = [
        events.BatchCreated("batch1", "INDIFFERENT-TABLE", 50, None),
        events.BatchCreated("batch2", "INDIFFERENT-TABLE", 50, date.today()),
        events.AllocationRequired("order1", "INDIFFERENT-TABLE", 20),
        events.AllocationRequired("order2", "INDIFFERENT-TABLE", 20),
    ]
    for e in event_history:
        messagebus.handle(e, uow)
    [batch1, batch2] = uow.products.get(sku="INDIFFERENT-TABLE").batches
    assert batch1.available_quantity == 10
    assert batch2.available_quantity == 50

    messagebus.handle(events.BatchQuantityChanged("batch1", 25), uow)

    # assert on new events emitted rather than downstream side-effects
    [reallocation_event] = uow.events_published
    assert isinstance(reallocation_event, events.AllocationRequired)
    assert reallocation_event.orderid in {'order1', 'order2'}
    assert reallocation_event.sku == 'INDIFFERENT-TABLE'
```

Whether you want to do this or not depends on the complexity of your chain of events. We say, start out with edge-to-edge testing, and resort to this only if necessary.

Exercise for the Reader

A great way to force yourself to really understand some code is to refactor it. In the discussion of testing handlers in isolation, we used something called `FakeUnitOfWorkWithFakeMessageBus`, which is unnecessarily complicated and violates the SRP.

If we change the message bus to being a class,[3] then building a `FakeMessageBus` is more straightforward:

An abstract message bus and its real and fake versions

```
class AbstractMessageBus:
    HANDLERS: Dict[Type[events.Event], List[Callable]]

    def handle(self, event: events.Event):
        for handler in self.HANDLERS[type(event)]:
            handler(event)


class MessageBus(AbstractMessageBus):
    HANDLERS = {
        events.OutOfStock: [send_out_of_stock_notification],

    }


class FakeMessageBus(messagebus.AbstractMessageBus):
    def __init__(self):
        self.events_published = []  # type: List[events.Event]
        self.handlers = {
            events.OutOfStock: [lambda e: self.events_published.append(e)]
        }
```

So jump into the code on GitHub and see if you can get a class-based version working, and then write a version of `test_reallocates_if_necessary_isolated()` from earlier.

We use a class-based message bus in Chapter 13, if you need more inspiration.

Wrap-Up

Let's look back at what we've achieved, and think about why we did it.

3 The "simple" implementation in this chapter essentially uses the *messagebus.py* module itself to implement the Singleton Pattern.

What Have We Achieved?

Events are simple dataclasses that define the data structures for inputs and internal messages within our system. This is quite powerful from a DDD standpoint, since events often translate really well into business language (look up *event storming* if you haven't already).

Handlers are the way we react to events. They can call down to our model or call out to external services. We can define multiple handlers for a single event if we want to. Handlers can also raise other events. This allows us to be very granular about what a handler does and really stick to the SRP.

Why Have We Achieved?

Our ongoing objective with these architectural patterns is to try to have the complexity of our application grow more slowly than its size. When we go all in on the message bus, as always we pay a price in terms of architectural complexity (see Table 9-1), but we buy ourselves a pattern that can handle almost arbitrarily complex requirements without needing any further conceptual or architectural change to the way we do things.

Here we've added quite a complicated use case (change quantity, deallocate, start new transaction, reallocate, publish external notification), but architecturally, there's been no cost in terms of complexity. We've added new events, new handlers, and a new external adapter (for email), all of which are existing categories of *things* in our architecture that we understand and know how to reason about, and that are easy to explain to newcomers. Our moving parts each have one job, they're connected to each other in well-defined ways, and there are no unexpected side effects.

Table 9-1. Whole app is a message bus: the trade-offs

Pros	Cons
• Handlers and services are the same thing, so that's simpler. • We have a nice data structure for inputs to the system.	• A message bus is still a slightly unpredictable way of doing things from a web point of view. You don't know in advance when things are going to end. • There will be duplication of fields and structure between model objects and events, which will have a maintenance cost. Adding a field to one usually means adding a field to at least one of the others.

Now, you may be wondering, where are those `BatchQuantityChanged` events going to come from? The answer is revealed in a couple chapters' time. But first, let's talk about events versus commands.

CHAPTER 10

Commands and Command Handler

In the previous chapter, we talked about using events as a way of representing the inputs to our system, and we turned our application into a message-processing machine.

To achieve that, we converted all our use-case functions to event handlers. When the API receives a POST to create a new batch, it builds a new `BatchCreated` event and handles it as if it were an internal event. This might feel counterintuitive. After all, the batch *hasn't* been created yet; that's why we called the API. We're going to fix that conceptual wart by introducing commands and showing how they can be handled by the same message bus but with slightly different rules.

The code for this chapter is in the chapter_10_commands branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_10_commands
# or to code along, checkout the previous chapter:
git checkout chapter_09_all_messagebus
```

Commands and Events

Like events, *commands* are a type of message—instructions sent by one part of a system to another. We usually represent commands with dumb data structures and can handle them in much the same way as events.

The differences between commands and events, though, are important.

Commands are sent by one actor to another specific actor with the expectation that a particular thing will happen as a result. When we post a form to an API handler, we are sending a command. We name commands with imperative mood verb phrases like "allocate stock" or "delay shipment."

Commands capture *intent*. They express our wish for the system to do something. As a result, when they fail, the sender needs to receive error information.

Events are broadcast by an actor to all interested listeners. When we publish `BatchQuantityChanged`, we don't know who's going to pick it up. We name events with past-tense verb phrases like "order allocated to stock" or "shipment delayed."

We often use events to spread the knowledge about successful commands.

Events capture *facts* about things that happened in the past. Since we don't know who's handling an event, senders should not care whether the receivers succeeded or failed. Table 10-1 recaps the differences.

Table 10-1. Events versus commands

	Event	Command
Named	Past tense	Imperative mood
Error handling	Fail independently	Fail noisily
Sent to	All listeners	One recipient

What kinds of commands do we have in our system right now?

Pulling out some commands (src/allocation/domain/commands.py)

```
class Command:
    pass

@dataclass
class Allocate(Command):  ❶
    orderid: str
    sku: str
    qty: int

@dataclass
class CreateBatch(Command):  ❷
    ref: str
    sku: str
    qty: int
    eta: Optional[date] = None

@dataclass
class ChangeBatchQuantity(Command):  ❸
    ref: str
    qty: int
```

❶ `commands.Allocate` will replace `events.AllocationRequired`.

❷ `commands.CreateBatch` will replace `events.BatchCreated`.

❸ `commands.ChangeBatchQuantity` will replace `events.BatchQuantityChanged`.

Differences in Exception Handling

Just changing the names and verbs is all very well, but that won't change the behavior of our system. We want to treat events and commands similarly, but not exactly the same. Let's see how our message bus changes:

Dispatch events and commands differently (src/allocation/service_layer/messagebus.py)

```
Message = Union[commands.Command, events.Event]


def handle(message: Message, uow: unit_of_work.AbstractUnitOfWork):  ❶
    results = []
    queue = [message]
    while queue:
        message = queue.pop(0)
        if isinstance(message, events.Event):
            handle_event(message, queue, uow)  ❷
        elif isinstance(message, commands.Command):
            cmd_result = handle_command(message, queue, uow)  ❷
            results.append(cmd_result)
        else:
            raise Exception(f'{message} was not an Event or Command')
    return results
```

❶ It still has a main `handle()` entrypoint that takes a `message`, which may be a command or an event.

❷ We dispatch events and commands to two different helper functions, shown next.

Here's how we handle events:

Events cannot interrupt the flow (src/allocation/service_layer/messagebus.py)

```
def handle_event(
    event: events.Event,
    queue: List[Message],
    uow: unit_of_work.AbstractUnitOfWork
):
    for handler in EVENT_HANDLERS[type(event)]:  ❶
        try:
            logger.debug('handling event %s with handler %s', event, handler)
            handler(event, uow=uow)
            queue.extend(uow.collect_new_events())
        except Exception:
            logger.exception('Exception handling event %s', event)
            continue  ❷
```

❶ Events go to a dispatcher that can delegate to multiple handlers per event.

❷ It catches and logs errors but doesn't let them interrupt message processing.

And here's how we do commands:

Commands reraise exceptions (src/allocation/service_layer/messagebus.py)

```
def handle_command(
    command: commands.Command,
    queue: List[Message],
    uow: unit_of_work.AbstractUnitOfWork
):
    logger.debug('handling command %s', command)
    try:
        handler = COMMAND_HANDLERS[type(command)]  ❶
        result = handler(command, uow=uow)
        queue.extend(uow.collect_new_events())
        return result  ❸
    except Exception:
        logger.exception('Exception handling command %s', command)
        raise  ❷
```

❶ The command dispatcher expects just one handler per command.

❷ If any errors are raised, they fail fast and will bubble up.

❸ `return result` is only temporary; as mentioned in "A Temporary Ugly Hack: The Message Bus Has to Return Results" on page 141, it's a temporary hack to allow the message bus to return the batch reference for the API to use. We'll fix this in Chapter 12.

We also change the single `HANDLERS` dict into different ones for commands and events. Commands can have only one handler, according to our convention:

New handlers dicts (src/allocation/service_layer/messagebus.py)

```
EVENT_HANDLERS = {
    events.OutOfStock: [handlers.send_out_of_stock_notification],
}  # type: Dict[Type[events.Event], List[Callable]]

COMMAND_HANDLERS = {
    commands.Allocate: handlers.allocate,
    commands.CreateBatch: handlers.add_batch,
    commands.ChangeBatchQuantity: handlers.change_batch_quantity,
}  # type: Dict[Type[commands.Command], Callable]
```

Discussion: Events, Commands, and Error Handling

Many developers get uncomfortable at this point and ask, "What happens when an event fails to process? How am I supposed to make sure the system is in a consistent state?" If we manage to process half of the events during `messagebus.handle` before an out-of-memory error kills our process, how do we mitigate problems caused by the lost messages?

Let's start with the worst case: we fail to handle an event, and the system is left in an inconsistent state. What kind of error would cause this? Often in our systems we can end up in an inconsistent state when only half an operation is completed.

For example, we could allocate three units of `DESIRABLE_BEANBAG` to a customer's order but somehow fail to reduce the amount of remaining stock. This would cause an inconsistent state: the three units of stock are both allocated *and* available, depending on how you look at it. Later, we might allocate those same beanbags to another customer, causing a headache for customer support.

In our allocation service, though, we've already taken steps to prevent that happening. We've carefully identified *aggregates* that act as consistency boundaries, and we've introduced a *UoW* that manages the atomic success or failure of an update to an aggregate.

For example, when we allocate stock to an order, our consistency boundary is the `Product` aggregate. This means that we can't accidentally overallocate: either a particular order line is allocated to the product, or it is not—there's no room for inconsistent states.

By definition, we don't require two aggregates to be immediately consistent, so if we fail to process an event and update only a single aggregate, our system can still be made eventually consistent. We shouldn't violate any constraints of the system.

With this example in mind, we can better understand the reason for splitting messages into commands and events. When a user wants to make the system do something, we represent their request as a *command*. That command should modify a single *aggregate* and either succeed or fail in totality. Any other bookkeeping, cleanup, and notification we need to do can happen via an *event*. We don't require the event handlers to succeed in order for the command to be successful.

Let's look at another example (from a different, imaginary projet) to see why not.

Imagine we are building an ecommerce website that sells expensive luxury goods. Our marketing department wants to reward customers for repeat visits. We will flag customers as VIPs after they make their third purchase, and this will entitle them to priority treatment and special offers. Our acceptance criteria for this story reads as follows:

```
    Given a customer with two orders in their history,
    When the customer places a third order,
    Then they should be flagged as a VIP.

    When a customer first becomes a VIP
    Then we should send them an email to congratulate them
```

Using the techniques we've already discussed in this book, we decide that we want to build a new `History` aggregate that records orders and can raise domain events when rules are met. We will structure the code like this:

VIP customer (example code for a different project)

```
class History:  # Aggregate

    def __init__(self, customer_id: int):
        self.orders = set() # Set[HistoryEntry]
        self.customer_id = customer_id

    def record_order(self, order_id: str, order_amount: int): ❶
        entry = HistoryEntry(order_id, order_amount)

        if entry in self.orders:
            return

        self.orders.add(entry)

        if len(self.orders) == 3:
            self.events.append(
                CustomerBecameVIP(self.customer_id)
            )


def create_order_from_basket(uow, cmd: CreateOrder): ❷
    with uow:
        order = Order.from_basket(cmd.customer_id, cmd.basket_items)
        uow.orders.add(order)
        uow.commit() # raises OrderCreated


def update_customer_history(uow, event: OrderCreated): ❸
    with uow:
        history = uow.order_history.get(event.customer_id)
        history.record_order(event.order_id, event.order_amount)
        uow.commit() # raises CustomerBecameVIP


def congratulate_vip_customer(uow, event: CustomerBecameVip): ❹
    with uow:
        customer = uow.customers.get(event.customer_id)
        email.send(
            customer.email_address,
            f'Congratulations {customer.first_name}!'
        )
```

❶ The `History` aggregate captures the rules indicating when a customer becomes a VIP. This puts us in a good place to handle changes when the rules become more complex in the future.

❷ Our first handler creates an order for the customer and raises a domain event `OrderCreated`.

❸ Our second handler updates the `History` object to record that an order was created.

❹ Finally, we send an email to the customer when they become a VIP.

Using this code, we can gain some intuition about error handling in an event-driven system.

In our current implementation, we raise events about an aggregate *after* we persist our state to the database. What if we raised those events *before* we persisted, and committed all our changes at the same time? That way, we could be sure that all the work was complete. Wouldn't that be safer?

What happens, though, if the email server is slightly overloaded? If all the work has to complete at the same time, a busy email server can stop us from taking money for orders.

What happens if there is a bug in the implementation of the `History` aggregate? Should we fail to take your money just because we can't recognize you as a VIP?

By separating out these concerns, we have made it possible for things to fail in isolation, which improves the overall reliability of the system. The only part of this code that *has* to complete is the command handler that creates an order. This is the only part that a customer cares about, and it's the part that our business stakeholders should prioritize.

Notice how we've deliberately aligned our transactional boundaries to the start and end of the business processes. The names that we use in the code match the jargon used by our business stakeholders, and the handlers we've written match the steps of our natural language acceptance criteria. This concordance of names and structure helps us to reason about our systems as they grow larger and more complex.

Recovering from Errors Synchronously

Hopefully we've convinced you that it's OK for events to fail independently from the commands that raised them. What should we do, then, to make sure we can recover from errors when they inevitably occur?

The first thing we need is to know *when* an error has occurred, and for that we usually rely on logs.

Let's look again at the `handle_event` method from our message bus:

Current handle function (src/allocation/service_layer/messagebus.py)

```
def handle_event(
    event: events.Event,
    queue: List[Message],
    uow: unit_of_work.AbstractUnitOfWork
):
    for handler in EVENT_HANDLERS[type(event)]:
        try:
            logger.debug('handling event %s with handler %s', event, handler)
            handler(event, uow=uow)
            queue.extend(uow.collect_new_events())
        except Exception:
            logger.exception('Exception handling event %s', event)
            continue
```

When we handle a message in our system, the first thing we do is write a log line to record what we're about to do. For our `CustomerBecameVIP` use case, the logs might read as follows:

```
Handling event CustomerBecameVIP(customer_id=12345)
with handler <function congratulate_vip_customer at 0x10ebc9a60>
```

Because we've chosen to use dataclasses for our message types, we get a neatly printed summary of the incoming data that we can copy and paste into a Python shell to re-create the object.

When an error occurs, we can use the logged data to either reproduce the problem in a unit test or replay the message into the system.

Manual replay works well for cases where we need to fix a bug before we can re-process an event, but our systems will *always* experience some background level of transient failure. This includes things like network hiccups, table deadlocks, and brief downtime caused by deployments.

For most of those cases, we can recover elegantly by trying again. As the proverb says, "If at first you don't succeed, retry the operation with an exponentially increasing back-off period."

Handle with retry (src/allocation/service_layer/messagebus.py)

```
from tenacity import Retrying, RetryError, stop_after_attempt, wait_exponential ❶

...

def handle_event(
    event: events.Event,
    queue: List[Message],
    uow: unit_of_work.AbstractUnitOfWork
):

    for handler in EVENT_HANDLERS[type(event)]:
        try:
            for attempt in Retrying(  ❷
                stop=stop_after_attempt(3),
                wait=wait_exponential()
            ):

                with attempt:
                    logger.debug('handling event %s with handler %s', event, handler)
                    handler(event, uow=uow)
                    queue.extend(uow.collect_new_events())
        except RetryError as retry_failure:
            logger.error(
                'Failed to handle event %s times, giving up!,
                retry_failure.last_attempt.attempt_number
            )
            continue
```

❶ Tenacity is a Python library that implements common patterns for retrying.

❷ Here we configure our message bus to retry operations up to three times, with an exponentially increasing wait between attempts.

Retrying operations that might fail is probably the single best way to improve the resilience of our software. Again, the Unit of Work and Command Handler patterns mean that each attempt starts from a consistent state and won't leave things half-finished.

At some point, regardless of `tenacity`, we'll have to give up trying to process the message. Building reliable systems with distributed messages is hard, and we have to skim over some tricky bits. There are pointers to more reference materials in the epilogue.

Wrap-Up

In this book we decided to introduce the concept of events before the concept of commands, but other guides often do it the other way around. Making explicit the requests that our system can respond to by giving them a name and their own data structure is quite a fundamental thing to do. You'll sometimes see people use the name *Command Handler* pattern to describe what we're doing with Events, Commands, and Message Bus.

Table 10-2 discusses some of the things you should think about before you jump on board.

Table 10-2. Splitting commands and events: the trade-offs

Pros	Cons
• Treating commands and events differently helps us understand which things have to succeed and which things we can tidy up later. • `CreateBatch` is definitely a less confusing name than `BatchCreated`. We are being explicit about the intent of our users, and explicit is better than implicit, right?	• The semantic differences between commands and events can be subtle. Expect bikeshedding arguments over the differences. • We're expressly inviting failure. We know that sometimes things will break, and we're choosing to handle that by making the failures smaller and more isolated. This can make the system harder to reason about and requires better monitoring.

In Chapter 11 we'll talk about using events as an integration pattern.

CHAPTER 11

Event-Driven Architecture: Using Events to Integrate Microservices

In the preceding chapter, we never actually spoke about *how* we would receive the "batch quantity changed" events, or indeed, how we might notify the outside world about reallocations.

We have a microservice with a web API, but what about other ways of talking to other systems? How will we know if, say, a shipment is delayed or the quantity is amended? How will we tell the warehouse system that an order has been allocated and needs to be sent to a customer?

In this chapter, we'd like to show how the events metaphor can be extended to encompass the way that we handle incoming and outgoing messages from the system. Internally, the core of our application is now a message processor. Let's follow through on that so it becomes a message processor *externally* as well. As shown in Figure 11-1, our application will receive events from external sources via an external message bus (we'll use Redis pub/sub queues as an example) and publish its outputs, in the form of events, back there as well.

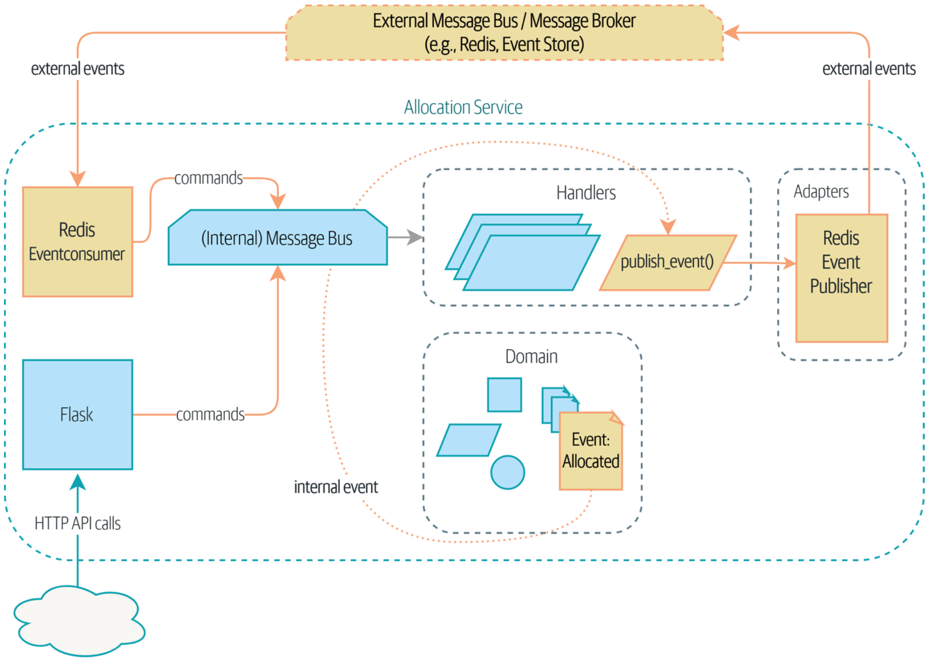


Figure 11-1. Our application is a message processor

The code for this chapter is in the chapter_11_external_events branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_11_external_events
# or to code along, checkout the previous chapter:
git checkout chapter_10_commands
```

Distributed Ball of Mud, and Thinking in Nouns

Before we get into that, let's talk about the alternatives. We regularly talk to engineers who are trying to build out a microservices architecture. Often they are migrating from an existing application, and their first instinct is to split their system into *nouns*.

What nouns have we introduced so far in our system? Well, we have batches of stock, orders, products, and customers. So a naive attempt at breaking up the system might have looked like Figure 11-2 (notice that we've named our system after a noun, *Batches*, instead of *Allocation*).

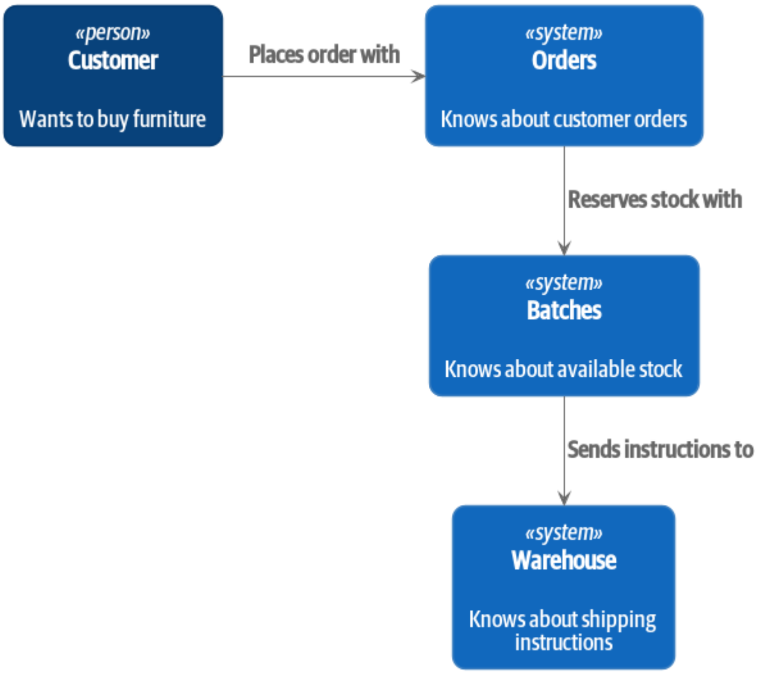


Figure 11-2. Context diagram with noun-based services

Each "thing" in our system has an associated service, which exposes an HTTP API.

Let's work through an example happy-path flow in Figure 11-3: our users visit a website and can choose from products that are in stock. When they add an item to their basket, we will reserve some stock for them. When an order is complete, we confirm the reservation, which causes us to send dispatch instructions to the warehouse. Let's also say, if this is the customer's third order, we want to update the customer record to flag them as a VIP.

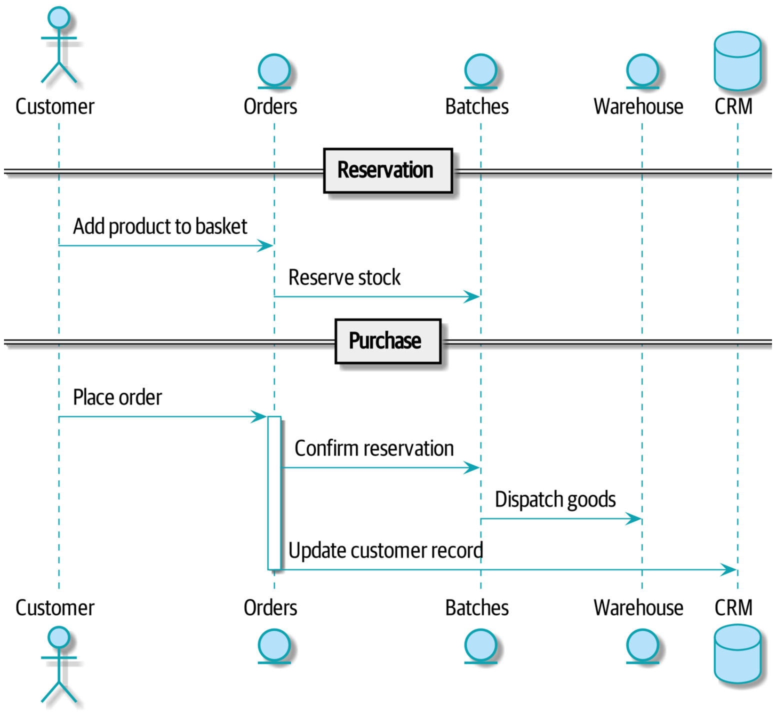


Figure 11-3. Command flow 1

We can think of each of these steps as a command in our system: `ReserveStock`, `ConfirmReservation`, `DispatchGoods`, `MakeCustomerVIP`, and so forth.

This style of architecture, where we create a microservice per database table and treat our HTTP APIs as CRUD interfaces to anemic models, is the most common initial way for people to approach service-oriented design.

This works *fine* for systems that are very simple, but it can quickly degrade into a distributed ball of mud.

To see why, let's consider another case. Sometimes, when stock arrives at the warehouse, we discover that items have been water damaged during transit. We can't sell water-damaged sofas, so we have to throw them away and request more stock from our partners. We also need to update our stock model, and that might mean we need to reallocate a customer's order.

Where does this logic go?

Well, the Warehouse system knows that the stock has been damaged, so maybe it should own this process, as shown in Figure 11-4.

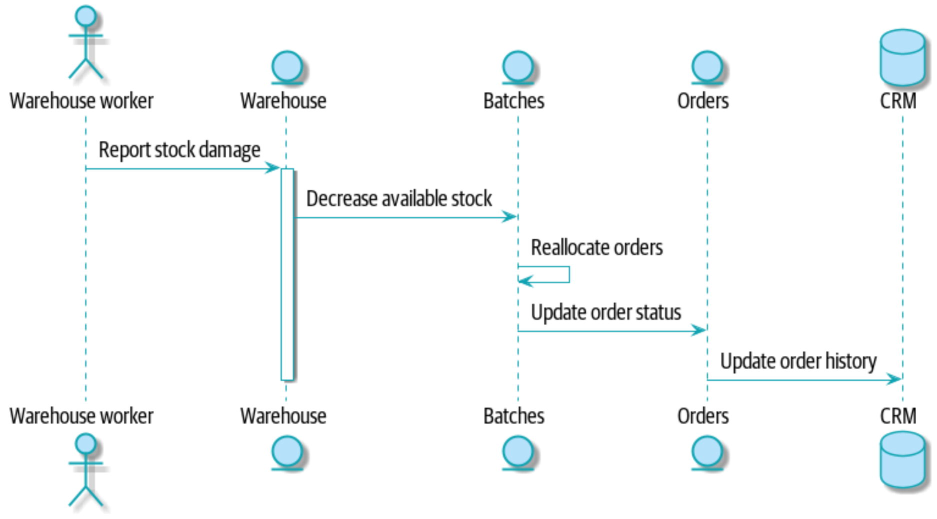


Figure 11-4. Command flow 2

This sort of works too, but now our dependency graph is a mess. To allocate stock, the Orders service drives the Batches system, which drives Warehouse; but in order to handle problems at the warehouse, our Warehouse system drives Batches, which drives Orders.

Multiply this by all the other workflows we need to provide, and you can see how services quickly get tangled up.

Error Handling in Distributed Systems

"Things break" is a universal law of software engineering. What happens in our system when one of our requests fails? Let's say that a network error happens right after we take a user's order for three `MISBEGOTTEN-RUG`, as shown in Figure 11-5.

We have two options here: we can place the order anyway and leave it unallocated, or we can refuse to take the order because the allocation can't be guaranteed. The failure state of our batches service has bubbled up and is affecting the reliability of our order service.

When two things have to be changed together, we say that they are *coupled*. We can think of this failure cascade as a kind of *temporal coupling*: every part of the system has to work at the same time for any part of it to work. As the system gets bigger, there is an exponentially increasing probability that some part is degraded.

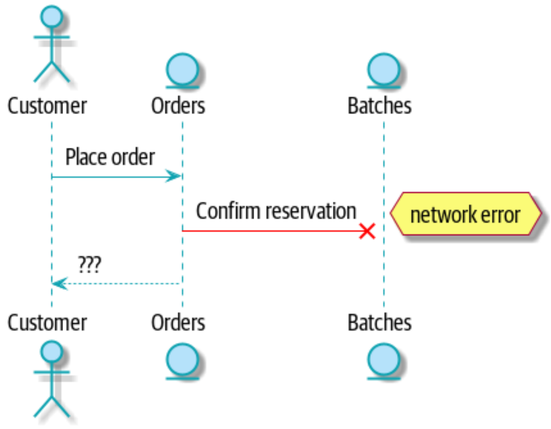


Figure 11-5. Command flow with error

Connascence

We're using the term *coupling* here, but there's another way to describe the relationships between our systems. *Connascence* is a term used by some authors to describe the different types of coupling.

Connascence isn't *bad*, but some types of connascence are *stronger* than others. We want to have strong connascence locally, as when two classes are closely related, but weak connascence at a distance.

In our first example of a distributed ball of mud, we see Connascence of Execution: multiple components need to know the correct order of work for an operation to be successful.

When thinking about error conditions here, we're talking about Connascence of Timing: multiple things have to happen, one after another, for the operation to work.

When we replace our RPC-style system with events, we replace both of these types of connascence with a *weaker* type. That's Connascence of Name: multiple components need to agree only on the name of an event and the names of fields it carries.

We can never completely avoid coupling, except by having our software not talk to any other software. What we want is to avoid *inappropriate* coupling. Connascence provides a mental model for understanding the strength and type of coupling inherent in different architectural styles. Read all about it at connascence.io.

The Alternative: Temporal Decoupling Using Asynchronous Messaging

How do we get appropriate coupling? We've already seen part of the answer, which is that we should think in terms of verbs, not nouns. Our domain model is about modeling a business process. It's not a static data model about a thing; it's a model of a verb.

So instead of thinking about a system for orders and a system for batches, we think about a system for *ordering* and a system for *allocating*, and so on.

When we separate things this way, it's a little easier to see which system should be responsible for what. When thinking about *ordering*, really we want to make sure that when we place an order, the order is placed. Everything else can happen *later*, so long as it happens.

If this sounds familiar, it should! Segregating responsibilities is the same process we went through when designing our aggregates and commands.

Like aggregates, microservices should be *consistency boundaries*. Between two services, we can accept eventual consistency, and that means we don't need to rely on synchronous calls. Each service accepts commands from the outside world and raises events to record the result. Other services can listen to those events to trigger the next steps in the workflow.

To avoid the Distributed Ball of Mud anti-pattern, instead of temporally coupled HTTP API calls, we want to use asynchronous messaging to integrate our systems. We want our `BatchQuantityChanged` messages to come in as external messages from upstream systems, and we want our system to publish `Allocated` events for downstream systems to listen to.

Why is this better? First, because things can fail independently, it's easier to handle degraded behavior: we can still take orders if the allocation system is having a bad day.

Second, we're reducing the strength of coupling between our systems. If we need to change the order of operations or to introduce new steps in the process, we can do that locally.

Using a Redis Pub/Sub Channel for Integration

Let's see how it will all work concretely. We'll need some way of getting events out of one system and into another, like our message bus, but for services. This piece of infrastructure is often called a *message broker*. The role of a message broker is to take messages from publishers and deliver them to subscribers.

At MADE.com, we use Event Store; Kafka or RabbitMQ are valid alternatives. A lightweight solution based on Redis pub/sub channels can also work just fine, and because Redis is much more generally familiar to people, we thought we'd use it for this book.

We're glossing over the complexity involved in choosing the right messaging platform. Concerns like message ordering, failure handling, and idempotency all need to be thought through. For a few pointers, see "Footguns" on page 226.

Our new flow will look like Figure 11-6: Redis provides the `BatchQuantityChanged` event that kicks off the whole process, and our `Allocated` event is published back out to Redis again at the end.

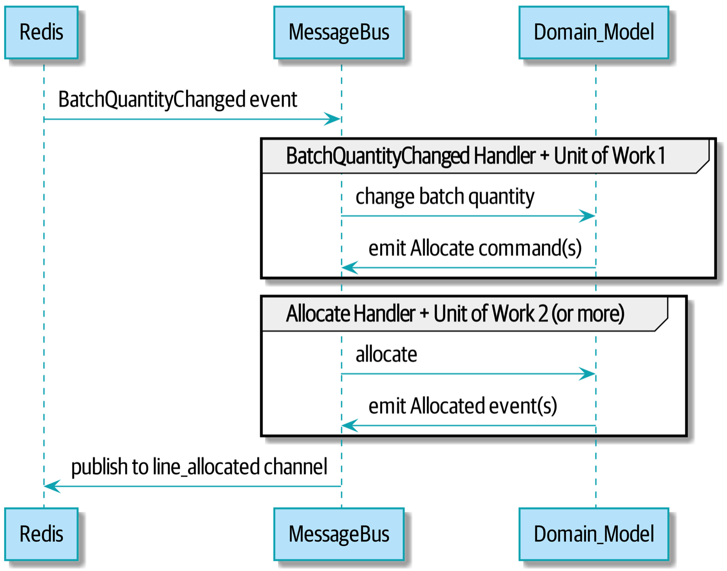

Redis
MessageBus
Domain_Model
BatchQuantityChanged event
BatchQuantityChanged Handler + Unit of Work 1
change batch quantity
emit Allocate command(s)
Allocate Handler + Unit of Work 2 (or more)
allocate
emit Allocated event(s)
publish to line_allocated channel
Redis
MessageBus
Domain_Model

Figure 11-6. Sequence diagram for reallocation flow

Test-Driving It All Using an End-to-End Test

Here's how we might start with an end-to-end test. We can use our existing API to create batches, and then we'll test both inbound and outbound messages:

An end-to-end test for our pub/sub model (tests/e2e/test_external_events.py)

```
def test_change_batch_quantity_leading_to_reallocation():
    # start with two batches and an order allocated to one of them  ❶
    orderid, sku = random_orderid(), random_sku()
    earlier_batch, later_batch = random_batchref('old'), random_batchref('newer')
    api_client.post_to_add_batch(earlier_batch, sku, qty=10, eta='2011-01-02')  ❷
    api_client.post_to_add_batch(later_batch, sku, qty=10, eta='2011-01-02')
    response = api_client.post_to_allocate(orderid, sku, 10)  ❷
    assert response.json()['batchref'] == earlier_batch

    subscription = redis_client.subscribe_to('line_allocated')  ❸

    # change quantity on allocated batch so it's less than our order  ❶
    redis_client.publish_message('change_batch_quantity', {  ❸
        'batchref': earlier_batch, 'qty': 5
    })

    # wait until we see a message saying the order has been reallocated  ❶
    messages = []
    for attempt in Retrying(stop=stop_after_delay(3), reraise=True):  ❹
        with attempt:
            message = subscription.get_message(timeout=1)
            if message:
                messages.append(message)
                print(messages)
            data = json.loads(messages[-1]['data'])
            assert data['orderid'] == orderid
            assert data['batchref'] == later_batch
```

❶ You can read the story of what's going on in this test from the comments: we want to send an event into the system that causes an order line to be reallocated, and we see that reallocation come out as an event in Redis too.

❷ `api_client` is a little helper that we refactored out to share between our two test types; it wraps our calls to `requests.post`.

❸ `redis_client` is another little test helper, the details of which don't really matter; its job is to be able to send and receive messages from various Redis channels. We'll use a channel called `change_batch_quantity` to send in our request to change the quantity for a batch, and we'll listen to another channel called `line_allocated` to look out for the expected reallocation.

❹ Because of the asynchronous nature of the system under test, we need to use the `tenacity` library again to add a retry loop—first, because it may take some time for our new `line_allocated` message to arrive, but also because it won't be the only message on that channel.

Redis Is Another Thin Adapter Around Our Message Bus

Our Redis pub/sub listener (we call it an *event consumer*) is very much like Flask: it translates from the outside world to our events:

Simple Redis message listener (src/allocation/entrypoints/redis_eventconsumer.py)

```
r = redis.Redis(**config.get_redis_host_and_port())


def main():
    orm.start_mappers()
    pubsub = r.pubsub(ignore_subscribe_messages=True)
    pubsub.subscribe('change_batch_quantity')  ❶

    for m in pubsub.listen():
        handle_change_batch_quantity(m)


def handle_change_batch_quantity(m):
    logging.debug('handling %s', m)
    data = json.loads(m['data'])  ❷
    cmd = commands.ChangeBatchQuantity(ref=data['batchref'], qty=data['qty'])  ❷
    messagebus.handle(cmd, uow=unit_of_work.SqlAlchemyUnitOfWork())
```

❶ `main()` subscribes us to the `change_batch_quantity` channel on load.

❷ Our main job as an entrypoint to the system is to deserialize JSON, convert it to a `Command`, and pass it to the service layer—much as the Flask adapter does.

We also build a new downstream adapter to do the opposite job—converting domain events to public events:

Simple Redis message publisher (src/allocation/adapters/redis_eventpublisher.py)

```
r = redis.Redis(**config.get_redis_host_and_port())


def publish(channel, event: events.Event):  ❶
    logging.debug('publishing: channel=%s, event=%s', channel, event)
    r.publish(channel, json.dumps(asdict(event)))
```

❶ We take a hardcoded channel here, but you could also store a mapping between event classes/names and the appropriate channel, allowing one or more message types to go to different channels.

Our New Outgoing Event

Here's what the `Allocated` event will look like:

New event (src/allocation/domain/events.py)

```
@dataclass
class Allocated(Event):
    orderid: str
    sku: str
    qty: int
    batchref: str
```

It captures everything we need to know about an allocation: the details of the order line, and which batch it was allocated to.

We add it into our model's `allocate()` method (having added a test first, naturally):

Product.allocate() emits new event to record what happened (src/allocation/domain/model.py)

```
class Product:
    ...
    def allocate(self, line: OrderLine) -> str:
        ...

            batch.allocate(line)
            self.version_number += 1
            self.events.append(events.Allocated(
                orderid=line.orderid, sku=line.sku, qty=line.qty,
                batchref=batch.reference,
            ))
            return batch.reference
```

The handler for `ChangeBatchQuantity` already exists, so all we need to add is a handler that publishes the outgoing event:

The message bus grows (src/allocation/service_layer/messagebus.py)

```
HANDLERS = {
    events.Allocated: [handlers.publish_allocated_event],
    events.OutOfStock: [handlers.send_out_of_stock_notification],
}  # type: Dict[Type[events.Event], List[Callable]]
```

Publishing the event uses our helper function from the Redis wrapper:

Publish to Redis (src/allocation/service_layer/handlers.py)

```
def publish_allocated_event(
        event: events.Allocated, uow: unit_of_work.AbstractUnitOfWork,
):
    redis_eventpublisher.publish('line_allocated', event)
```

Internal Versus External Events

It's a good idea to keep the distinction between internal and external events clear. Some events may come from the outside, and some events may get upgraded and published externally, but not all of them will. This is particularly important if you get into event sourcing (very much a topic for another book, though).

Outbound events are one of the places it's important to apply validation. See Appendix E for some validation philosophy and examples.

Exercise for the Reader

A nice simple one for this chapter: make it so that the main `allocate()` use case can also be invoked by an event on a Redis channel, as well as (or instead of) via the API.

You will likely want to add a new E2E test and feed through some changes into `redis_eventconsumer.py`.

Wrap-Up

Events can come *from* the outside, but they can also be published externally—our `publish` handler converts an event to a message on a Redis channel. We use events to talk to the outside world. This kind of temporal decoupling buys us a lot of flexibility in our application integrations, but as always, it comes at a cost.

> Event notification is nice because it implies a low level of coupling, and is pretty simple to set up. It can become problematic, however, if there really is a logical flow that runs over various event notifications...It can be hard to see such a flow as it's not explicit in any program text....This can make it hard to debug and modify.
>
> —Martin Fowler, "What do you mean by 'Event-Driven'"

Table 11-1 shows some trade-offs to think about.

Table 11-1. Event-based microservices integration: the trade-offs

Pros	Cons
• Avoids the distributed big ball of mud. • Services are decoupled: it's easier to change individual services and add new ones.	• The overall flows of information are harder to see. • Eventual consistency is a new concept to deal with. • Message reliability and choices around at-least-once versus at-most-once delivery need thinking through.

More generally, if you're moving from a model of synchronous messaging to an async one, you also open up a whole host of problems having to do with message reliability and eventual consistency. Read on to "Footguns" on page 226.

CHAPTER 12

Command-Query Responsibility Segregation (CQRS)

In this chapter, we're going to start with a fairly uncontroversial insight: reads (queries) and writes (commands) are different, so they should be treated differently (or have their responsibilities segregated, if you will). Then we're going to push that insight as far as we can.

If you're anything like Harry, this will all seem extreme at first, but hopefully we can make the argument that it's not *totally* unreasonable.

Figure 12-1 shows where we might end up.

The code for this chapter is in the chapter_12_cqrs branch on GitHub.

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_12_cqrs
# or to code along, checkout the previous chapter:
git checkout chapter_11_external_events
```

First, though, why bother?

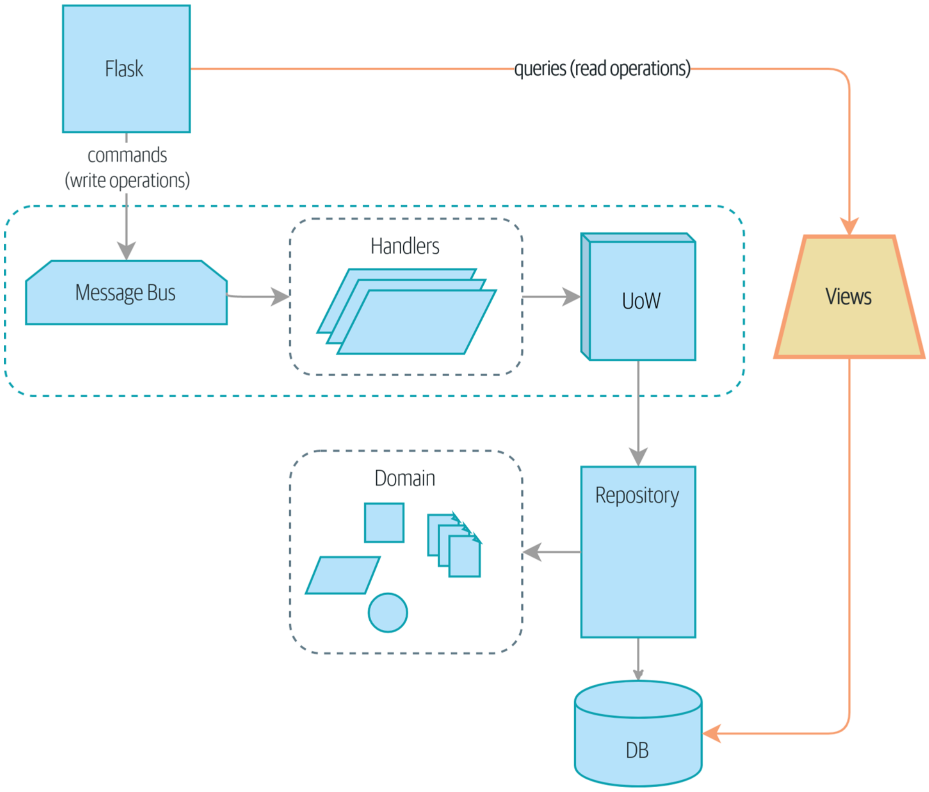


Figure 12-1. Separating reads from writes

Domain Models Are for Writing

We've spent a lot of time in this book talking about how to build software that enforces the rules of our domain. These rules, or constraints, will be different for every application, and they make up the interesting core of our systems.

In this book, we've set explicit constraints like "You can't allocate more stock than is available," as well as implicit constraints like "Each order line is allocated to a single batch."

We wrote down these rules as unit tests at the beginning of the book:

Our basic domain tests (tests/unit/test_batches.py)

```
def test_allocating_to_a_batch_reduces_the_available_quantity():
    batch = Batch("batch-001", "SMALL-TABLE", qty=20, eta=date.today())
    line = OrderLine('order-ref', "SMALL-TABLE", 2)

    batch.allocate(line)

    assert batch.available_quantity == 18

...

def test_cannot_allocate_if_available_smaller_than_required():
    small_batch, large_line = make_batch_and_line("ELEGANT-LAMP", 2, 20)
    assert small_batch.can_allocate(large_line) is False
```

To apply these rules properly, we needed to ensure that operations were consistent, and so we introduced patterns like *Unit of Work* and *Aggregate* that help us commit small chunks of work.

To communicate changes between those small chunks, we introduced the Domain Events pattern so we can write rules like "When stock is damaged or lost, adjust the available quantity on the batch, and reallocate orders if necessary."

All of this complexity exists so we can enforce rules when we change the state of our system. We've built a flexible set of tools for writing data.

What about reads, though?

Most Users Aren't Going to Buy Your Furniture

At MADE.com, we have a system very like the allocation service. In a busy day, we might process one hundred orders in an hour, and we have a big gnarly system for allocating stock to those orders.

In that same busy day, though, we might have one hundred product views per *second*. Each time somebody visits a product page, or a product listing page, we need to figure out whether the product is still in stock and how long it will take us to deliver it.

The *domain* is the same—we're concerned with batches of stock, and their arrival date, and the amount that's still available—but the access pattern is very different. For example, our customers won't notice if the query is a few seconds out of date, but if our allocate service is inconsistent, we'll make a mess of their orders. We can take advantage of this difference by making our reads *eventually consistent* in order to make them perform better.

Is Read Consistency Truly Attainable?

This idea of trading consistency against performance makes a lot of developers nervous at first, so let's talk quickly about that.

Let's imagine that our "Get Available Stock" query is 30 seconds out of date when Bob visits the page for `ASYMMETRICAL-DRESSER`. Meanwhile, though, Harry has already bought the last item. When we try to allocate Bob's order, we'll get a failure, and we'll need to either cancel his order or buy more stock and delay his delivery.

People who've worked only with relational data stores get *really* nervous about this problem, but it's worth considering two other scenarios to gain some perspective.

First, let's imagine that Bob and Harry both visit the page at *the same time*. Harry goes off to make coffee, and by the time he returns, Bob has already bought the last dresser. When Harry places his order, we send it to the allocation service, and because there's not enough stock, we have to refund his payment or buy more stock and delay his delivery.

As soon as we render the product page, the data is already stale. This insight is key to understanding why reads can be safely inconsistent: we'll always need to check the current state of our system when we come to allocate, because all distributed systems are inconsistent. As soon as you have a web server and two customers, you have the potential for stale data.

OK, let's assume we solve that problem somehow: we magically build a totally consistent web application where nobody ever sees stale data. This time Harry gets to the page first and buys his dresser.

Unfortunately for him, when the warehouse staff tries to dispatch his furniture, it falls off the forklift and smashes into a zillion pieces. Now what?

The only options are to either call Harry and refund his order or buy more stock and delay delivery.

No matter what we do, we're always going to find that our software systems are inconsistent with reality, and so we'll always need business processes to cope with these edge cases. It's OK to trade performance for consistency on the read side, because stale data is essentially unavoidable.

We can think of these requirements as forming two halves of a system: the read side and the write side, shown in Table 12-1.

For the write side, our fancy domain architectural patterns help us to evolve our system over time, but the complexity we've built so far doesn't buy anything for reading data. The service layer, the unit of work, and the clever domain model are just bloat.

Table 12-1. Read versus write

	Read side	Write side
Behavior	Simple read	Complex business logic
Cacheability	Highly cacheable	Uncacheable
Consistency	Can be stale	Must be transactionally consistent

Post/Redirect/Get and CQS

If you do web development, you're probably familiar with the Post/Redirect/Get pattern. In this technique, a web endpoint accepts an HTTP POST and responds with a redirect to see the result. For example, we might accept a POST to */batches* to create a new batch and redirect the user to */batches/123* to see their newly created batch.

This approach fixes the problems that arise when users refresh the results page in their browser or try to bookmark a results page. In the case of a refresh, it can lead to our users double-submitting data and thus buying two sofas when they needed only one. In the case of a bookmark, our hapless customers will end up with a broken page when they try to GET a POST endpoint.

Both these problems happen because we're returning data in response to a write operation. Post/Redirect/Get sidesteps the issue by separating the read and write phases of our operation.

This technique is a simple example of command-query separation (CQS). In CQS we follow one simple rule: functions should either modify state or answer questions, but never both. This makes software easier to reason about: we should always be able to ask, "Are the lights on?" without flicking the light switch.

When building APIs, we can apply the same design technique by returning a 201 Created, or a 202 Accepted, with a Location header containing the URI of our new resources. What's important here isn't the status code we use but the logical separation of work into a write phase and a query phase.

As you'll see, we can use the CQS principle to make our systems faster and more scalable, but first, let's fix the CQS violation in our existing code. Ages ago, we introduced an `allocate` endpoint that takes an order and calls our service layer to allocate some stock. At the end of the call, we return a 200 OK and the batch ID. That's led to some ugly design flaws so that we can get the data we need. Let's change it to return a simple OK message and instead provide a new read-only endpoint to retrieve allocation state:

API test does a GET after the POST (tests/e2e/test_api.py)

```
@pytest.mark.usefixtures('postgres_db')
@pytest.mark.usefixtures('restart_api')
def test_happy_path_returns_202_and_batch_is_allocated():
    orderid = random_orderid()
    sku, othersku = random_sku(), random_sku('other')
    earlybatch = random_batchref(1)
    laterbatch = random_batchref(2)
    otherbatch = random_batchref(3)
    api_client.post_to_add_batch(laterbatch, sku, 100, '2011-01-02')
    api_client.post_to_add_batch(earlybatch, sku, 100, '2011-01-01')
    api_client.post_to_add_batch(otherbatch, othersku, 100, None)

    r = api_client.post_to_allocate(orderid, sku, qty=3)
    assert r.status_code == 202

    r = api_client.get_allocation(orderid)
    assert r.ok
    assert r.json() == [
        {'sku': sku, 'batchref': earlybatch},
    ]


@pytest.mark.usefixtures('postgres_db')
@pytest.mark.usefixtures('restart_api')
def test_unhappy_path_returns_400_and_error_message():
    unknown_sku, orderid = random_sku(), random_orderid()
    r = api_client.post_to_allocate(
        orderid, unknown_sku, qty=20, expect_success=False,
    )
    assert r.status_code == 400
    assert r.json()['message'] == f'Invalid sku {unknown_sku}'

    r = api_client.get_allocation(orderid)
    assert r.status_code == 404
```

OK, what might the Flask app look like?

Endpoint for viewing allocations (src/allocation/entrypoints/flask_app.py)

```
from allocation import views
...

@app.route("/allocations/<orderid>", methods=['GET'])
def allocations_view_endpoint(orderid):
    uow = unit_of_work.SqlAlchemyUnitOfWork()
    result = views.allocations(orderid, uow)  ❶
    if not result:
        return 'not found', 404
    return jsonify(result), 200
```

❶ All right, a *views.py*, fair enough; we can keep read-only stuff in there, and it'll be a real *views.py*, not like Django's, something that knows how to build read-only views of our data…

Hold On to Your Lunch, Folks

Hmm, so we can probably just add a list method to our existing repository object:

Views do…raw SQL? (src/allocation/views.py)

```
from allocation.service_layer import unit_of_work

def allocations(orderid: str, uow: unit_of_work.SqlAlchemyUnitOfWork):
    with uow:
        results = list(uow.session.execute(
            'SELECT ol.sku, b.reference'
            ' FROM allocations AS a'
            ' JOIN batches AS b ON a.batch_id = b.id'
            ' JOIN order_lines AS ol ON a.orderline_id = ol.id'
            ' WHERE ol.orderid = :orderid',
            dict(orderid=orderid)
        ))
    return [{'sku': sku, 'batchref': batchref} for sku, batchref in results]
```

Excuse me? Raw SQL?

If you're anything like Harry encountering this pattern for the first time, you'll be wondering what on earth Bob has been smoking. We're hand-rolling our own SQL now, and converting database rows directly to dicts? After all the effort we put into building a nice domain model? And what about the Repository pattern? Isn't that meant to be our abstraction around the database? Why don't we reuse that?

Well, let's explore that seemingly simpler alternative first, and see what it looks like in practice.

We'll still keep our view in a separate *views.py* module; enforcing a clear distinction between reads and writes in your application is still a good idea. We apply command-query separation, and it's easy to see which code modifies state (the event handlers) and which code just retrieves read-only state (the views).

Splitting out your read-only views from your state-modifying command and event handlers is probably a good idea, even if you don't want to go to full-blown CQRS.

Testing CQRS Views

Before we get into exploring various options, let's talk about testing. Whichever approaches you decide to go for, you're probably going to need at least one integration test. Something like this:

An integration test for a view (tests/integration/test_views.py)

```
def test_allocations_view(sqlite_session_factory):
    uow = unit_of_work.SqlAlchemyUnitOfWork(sqlite_session_factory)
    messagebus.handle(commands.CreateBatch('sku1batch', 'sku1', 50, None), uow)  ❶
    messagebus.handle(commands.CreateBatch('sku2batch', 'sku2', 50, today), uow)
    messagebus.handle(commands.Allocate('order1', 'sku1', 20), uow)
    messagebus.handle(commands.Allocate('order1', 'sku2', 20), uow)
    # add a spurious batch and order to make sure we're getting the right ones
    messagebus.handle(commands.CreateBatch('sku1batch-later', 'sku1', 50, today), uow)
    messagebus.handle(commands.Allocate('otherorder', 'sku1', 30), uow)
    messagebus.handle(commands.Allocate('otherorder', 'sku2', 10), uow)

    assert views.allocations('order1', uow) == [
        {'sku': 'sku1', 'batchref': 'sku1batch'},
        {'sku': 'sku2', 'batchref': 'sku2batch'},
    ]
```

❶ We do the setup for the integration test by using the public entrypoint to our application, the message bus. That keeps our tests decoupled from any implementation/infrastructure details about how things get stored.

"Obvious" Alternative 1: Using the Existing Repository

How about adding a helper method to our `products` repository?

A simple view that uses the repository (src/allocation/views.py)

```
from allocation import unit_of_work

def allocations(orderid: str, uow: unit_of_work.AbstractUnitOfWork):
    with uow:
        products = uow.products.for_order(orderid=orderid)  ❶
        batches = [b for p in products for b in p.batches]  ❷
        return [
            {'sku': b.sku, 'batchref': b.reference}
            for b in batches
            if orderid in b.orderids  ❸
        ]
```

❶ Our repository returns `Product` objects, and we need to find all the products for the SKUs in a given order, so we'll build a new helper method called `.for_order()` on the repository.

❷ Now we have products but we actually want batch references, so we get all the possible batches with a list comprehension.

❸ We filter *again* to get just the batches for our specific order. That, in turn, relies on our `Batch` objects being able to tell us which order IDs it has allocated.

We implement that last using a `.orderid` property:

An arguably unnecessary property on our model (src/allocation/domain/model.py)

```
class Batch:
    ...

    @property
    def orderids(self):
        return {l.orderid for l in self._allocations}
```

You can start to see that reusing our existing repository and domain model classes is not as straightforward as you might have assumed. We've had to add new helper methods to both, and we're doing a bunch of looping and filtering in Python, which is work that would be done much more efficiently by the database.

So yes, on the plus side we're reusing our existing abstractions, but on the downside, it all feels quite clunky.

Your Domain Model Is Not Optimized for Read Operations

What we're seeing here are the effects of having a domain model that is designed primarily for write operations, while our requirements for reads are often conceptually quite different.

This is the chin-stroking-architect's justification for CQRS. As we've said before, a domain model is not a data model—we're trying to capture the way the business works: workflow, rules around state changes, messages exchanged; concerns about how the system reacts to external events and user input. *Most of this stuff is totally irrelevant for read-only operations.*

This justification for CQRS is related to the justification for the Domain Model pattern. If you're building a simple CRUD app, reads and writes are going to be closely related, so you don't need a domain model or CQRS. But the more complex your domain, the more likely you are to need both.

To make a facile point, your domain classes will have multiple methods for modifying state, and you won't need any of them for read-only operations.

As the complexity of your domain model grows, you will find yourself making more and more choices about how to structure that model, which make it more and more awkward to use for read operations.

"Obvious" Alternative 2: Using the ORM

You may be thinking, OK, if our repository is clunky, and working with `Products` is clunky, then I can at least use my ORM and work with `Batches`. That's what it's for!

A simple view that uses the ORM (src/allocation/views.py)

```
from allocation import unit_of_work, model

def allocations(orderid: str, uow: unit_of_work.AbstractUnitOfWork):
    with uow:
        batches = uow.session.query(model.Batch).join(
            model.OrderLine, model.Batch._allocations
        ).filter(
            model.OrderLine.orderid == orderid
        )
        return [
            {'sku': b.sku, 'batchref': b.batchref}
            for b in batches
        ]
```

But is that *actually* any easier to write or understand than the raw SQL version from the code example in "Hold On to Your Lunch, Folks" on page 181? It may not look too bad up there, but we can tell you it took several attempts, and plenty of digging through the SQLAlchemy docs. SQL is just SQL.

But the ORM can also expose us to performance problems.

SELECT N+1 and Other Performance Considerations

The so-called `SELECT N+1` problem is a common performance problem with ORMs: when retrieving a list of objects, your ORM will often perform an initial query to, say, get all the IDs of the objects it needs, and then issue individual queries for each object to retrieve their attributes. This is especially likely if there are any foreign-key relationships on your objects.

In all fairness, we should say that SQLAlchemy is quite good at avoiding the `SELECT N+1` problem. It doesn't display it in the preceding example, and you can request eager loading explicitly to avoid it when dealing with joined objects.

Beyond `SELECT N+1`, you may have other reasons for wanting to decouple the way you persist state changes from the way that you retrieve current state. A set of fully normalized relational tables is a good way to make sure that write operations never cause data corruption. But retrieving data using lots of joins can be slow. It's common in such cases to add some denormalized views, build read replicas, or even add caching layers.

Time to Completely Jump the Shark

On that note: have we convinced you that our raw SQL version isn't so weird as it first seemed? Perhaps we were exaggerating for effect? Just you wait.

So, reasonable or not, that hardcoded SQL query is pretty ugly, right? What if we made it nicer…

A much nicer query (src/allocation/views.py)

```
def allocations(orderid: str, uow: unit_of_work.SqlAlchemyUnitOfWork):
    with uow:
        results = list(uow.session.execute(
            'SELECT sku, batchref FROM allocations_view WHERE orderid = :orderid',
            dict(orderid=orderid)
        ))
        ...
```

…by *keeping a totally separate, denormalized data store for our view model*?

Hee hee hee, no foreign keys, just strings, YOLO (src/allocation/adapters/orm.py)

```
allocations_view = Table(
    'allocations_view', metadata,
    Column('orderid', String(255)),
    Column('sku', String(255)),
    Column('batchref', String(255)),
)
```

OK, nicer-looking SQL queries wouldn't be a justification for anything really, but building a denormalized copy of your data that's optimized for read operations isn't uncommon, once you've reached the limits of what you can do with indexes.

Even with well-tuned indexes, a relational database uses a lot of CPU to perform joins. The fastest queries will always be `SELECT * from` *`mytable`* `WHERE` *`key`* `= :`*`value`*.

More than raw speed, though, this approach buys us scale. When we're writing data to a relational database, we need to make sure that we get a lock over the rows we're changing so we don't run into consistency problems.

If multiple clients are changing data at the same time, we'll have weird race conditions. When we're *reading* data, though, there's no limit to the number of clients that can concurrently execute. For this reason, read-only stores can be horizontally scaled out.

Because read replicas can be inconsistent, there's no limit to how many we can have. If you're struggling to scale a system with a complex data store, ask whether you could build a simpler read model.

Keeping the read model up to date is the challenge! Database views (materialized or otherwise) and triggers are a common solution, but that limits you to your database. We'd like to show you how to reuse our event-driven architecture instead.

Updating a Read Model Table Using an Event Handler

We add a second handler to the `Allocated` event:

Allocated event gets a new handler (src/allocation/service_layer/messagebus.py)

```
EVENT_HANDLERS = {
    events.Allocated: [
        handlers.publish_allocated_event,
        handlers.add_allocation_to_read_model
    ],
```

Here's what our update-view-model code looks like:

Update on allocation (src/allocation/service_layer/handlers.py)

```
def add_allocation_to_read_model(
        event: events.Allocated, uow: unit_of_work.SqlAlchemyUnitOfWork,
):
    with uow:
        uow.session.execute(
            'INSERT INTO allocations_view (orderid, sku, batchref)'
            ' VALUES (:orderid, :sku, :batchref)',
            dict(orderid=event.orderid, sku=event.sku, batchref=event.batchref)
        )
        uow.commit()
```

Believe it or not, that will pretty much work! *And it will work against the exact same integration tests as the rest of our options.*

OK, you'll also need to handle `Deallocated`:

```
events.Deallocated: [
    handlers.remove_allocation_from_read_model,
    handlers.reallocate
],

...

def remove_allocation_from_read_model(
        event: events.Deallocated, uow: unit_of_work.SqlAlchemyUnitOfWork,
):
    with uow:
        uow.session.execute(
            'DELETE FROM allocations_view '
            ' WHERE orderid = :orderid AND sku = :sku',
```

Figure 12-2 shows the flow across the two requests.

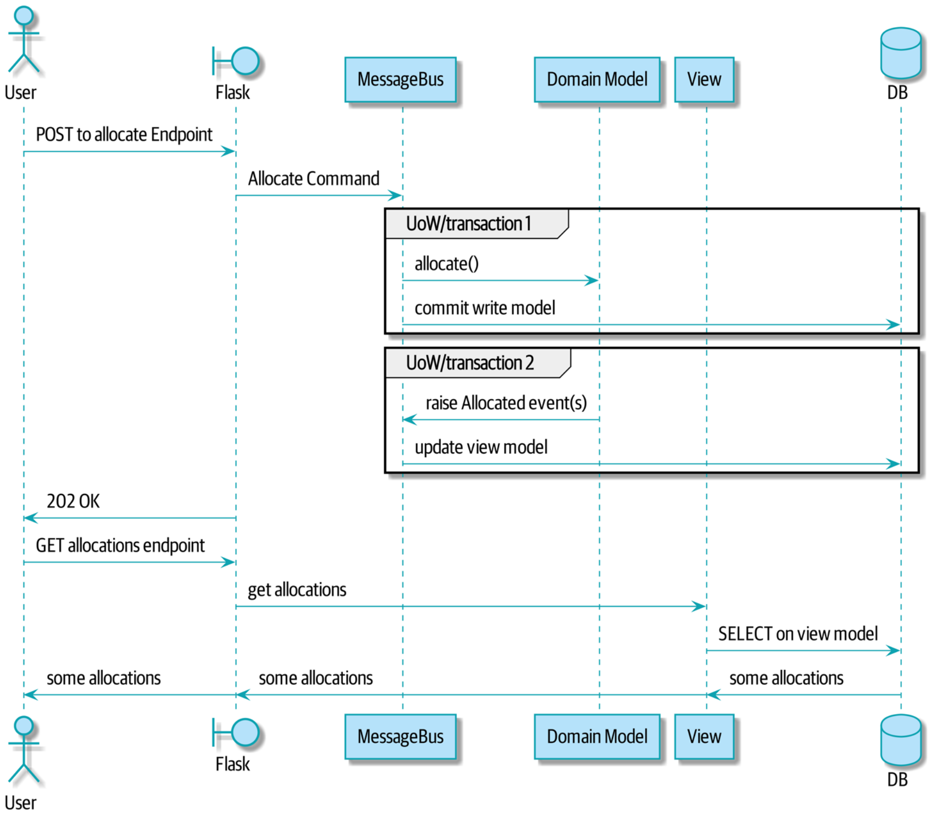


Figure 12-2. Sequence diagram for read model

In Figure 12-2, you can see two transactions in the POST/write operation, one to update the write model and one to update the read model, which the GET/read operation can use.

Rebuilding from Scratch

"What happens when it breaks?" should be the first question we ask as engineers.

How do we deal with a view model that hasn't been updated because of a bug or temporary outage? Well, this is just another case where events and commands can fail independently.

If we *never* updated the view model, and the `ASYMMETRICAL-DRESSER` was forever in stock, that would be annoying for customers, but the `allocate` service would still fail, and we'd take action to fix the problem.

Rebuilding a view model is easy, though. Since we're using a service layer to update our view model, we can write a tool that does the following:

- Queries the current state of the write side to work out what's currently allocated
- Calls the `add_allocate_to_read_model` handler for each allocated item

We can use this technique to create entirely new read models from historical data.

Changing Our Read Model Implementation Is Easy

Let's see the flexibility that our event-driven model buys us in action, by seeing what happens if we ever decide we want to implement a read model by using a totally separate storage engine, Redis.

Just watch:

Handlers update a Redis read model (src/allocation/service_layer/handlers.py)

```
def add_allocation_to_read_model(event: events.Allocated, _):
    redis_eventpublisher.update_readmodel(event.orderid, event.sku, event.batchref)

def remove_allocation_from_read_model(event: events.Deallocated, _):
    redis_eventpublisher.update_readmodel(event.orderid, event.sku, None)
```

The helpers in our Redis module are one-liners:

Redis read model read and update (src/allocation/adapters/redis_eventpublisher.py)

```
def update_readmodel(orderid, sku, batchref):
    r.hset(orderid, sku, batchref)
```

```
def get_readmodel(orderid):
    return r.hgetall(orderid)
```

(Maybe the name *redis_eventpublisher.py* is a misnomer now, but you get the idea.)

And the view itself changes very slightly to adapt to its new backend:

View adapted to Redis (src/allocation/views.py)

```
def allocations(orderid):
    batches = redis_eventpublisher.get_readmodel(orderid)
    return [
        {'batchref': b.decode(), 'sku': s.decode()}
        for s, b in batches.items()
    ]
```

And the *exact same* integration tests that we had before still pass, because they are written at a level of abstraction that's decoupled from the implementation: setup puts messages on the message bus, and the assertions are against our view.

Event handlers are a great way to manage updates to a read model, if you decide you need one. They also make it easy to change the implementation of that read model at a later date.

Exercise for the Reader

Implement another view, this time to show the allocation for a single order line.

Here the trade-offs between using hardcoded SQL versus going via a repository should be much more blurry. Try a few versions (maybe including going to Redis), and see which you prefer.

Wrap-Up

Table 12-2 proposes some pros and cons for each of our options.

As it happens, the allocation service at MADE.com does use "full-blown" CQRS, with a read model stored in Redis, and even a second layer of cache provided by Varnish. But its use cases are quite a bit different from what we've shown here. For the kind of allocation service we're building, it seems unlikely that you'd need to use a separate read model and event handlers for updating it.

But as your domain model becomes richer and more complex, a simplified read model become ever more compelling.

Table 12-2. Trade-offs of various view model options

Option	Pros	Cons
Just use repositories	Simple, consistent approach.	Expect performance issues with complex query patterns.
Use custom queries with your ORM	Allows reuse of DB configuration and model definitions.	Adds another query language with its own quirks and syntax.
Use hand-rolled SQL	Offers fine control over performance with a standard query syntax.	Changes to DB schema have to be made to your hand-rolled queries *and* your ORM definitions. Highly normalized schemas may still have performance limitations.
Create separate read stores with events	Read-only copies are easy to scale out. Views can be constructed when data changes so that queries are as simple as possible.	Complex technique. Harry will be forever suspicious of your tastes and motives.

Often, your read operations will be acting on the same conceptual objects as your write model, so using the ORM, adding some read methods to your repositories, and using domain model classes for your read operations is *just fine*.

In our book example, the read operations act on quite different conceptual entities to our domain model. The allocation service thinks in terms of `Batches` for a single SKU, but users care about allocations for a whole order, with multiple SKUs, so using the ORM ends up being a little awkward. We'd be quite tempted to go with the raw-SQL view we showed right at the beginning of the chapter.

On that note, let's sally forth into our final chapter.

CHAPTER 13

Dependency Injection (and Bootstrapping)

Dependency injection (DI) is regarded with suspicion in the Python world. And we've managed *just fine* without it so far in the example code for this book!

In this chapter, we'll explore some of the pain points in our code that lead us to consider using DI, and we'll present some options for how to do it, leaving it to you to pick which you think is most Pythonic.

We'll also add a new component to our architecture called *bootstrap.py*; it will be in charge of dependency injection, as well as some other initialization stuff that we often need. We'll explain why this sort of thing is called a *composition root* in OO languages, and why *bootstrap script* is just fine for our purposes.

Figure 13-1 shows what our app looks like without a bootstrapper: the entrypoints do a lot of initialization and passing around of our main dependency, the UoW.

If you haven't already, it's worth reading Chapter 3 before continuing with this chapter, particularly the discussion of functional versus object-oriented dependency management.

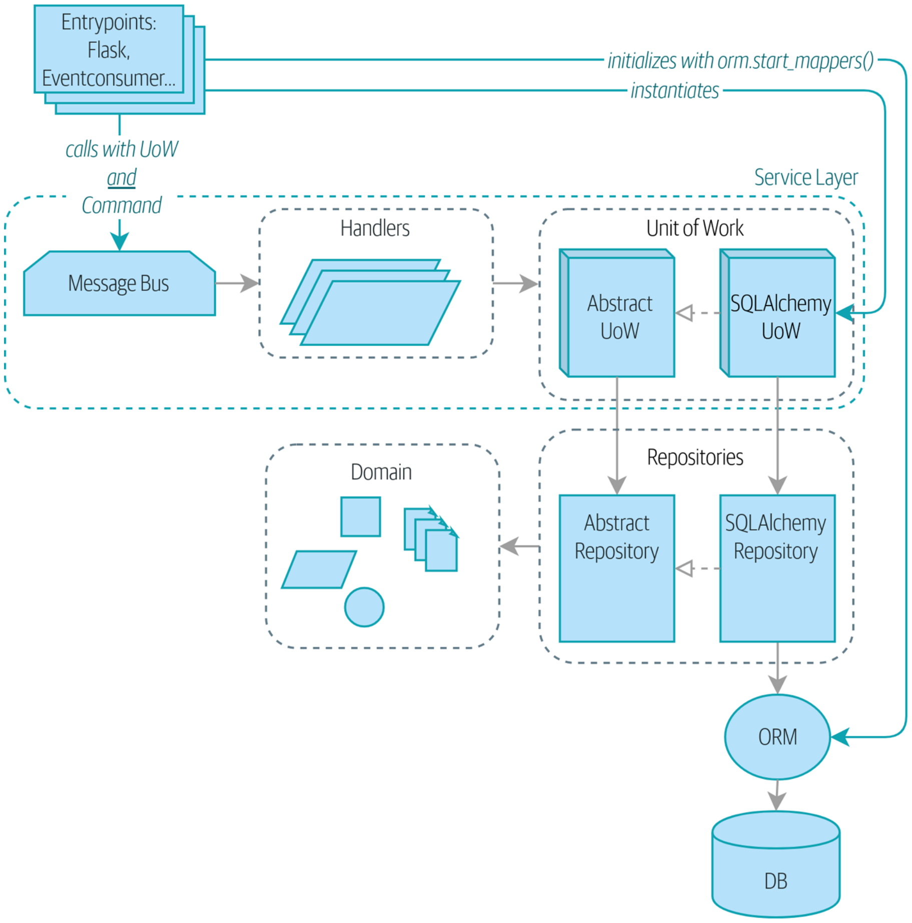


Figure 13-1. Without bootstrap: entrypoints do a lot

The code for this chapter is in the chapter_13_dependency_injection branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_13_dependency_injection
# or to code along, checkout the previous chapter:
git checkout chapter_12_cqrs
```

Figure 13-2 shows our bootstrapper taking over those responsibilities.

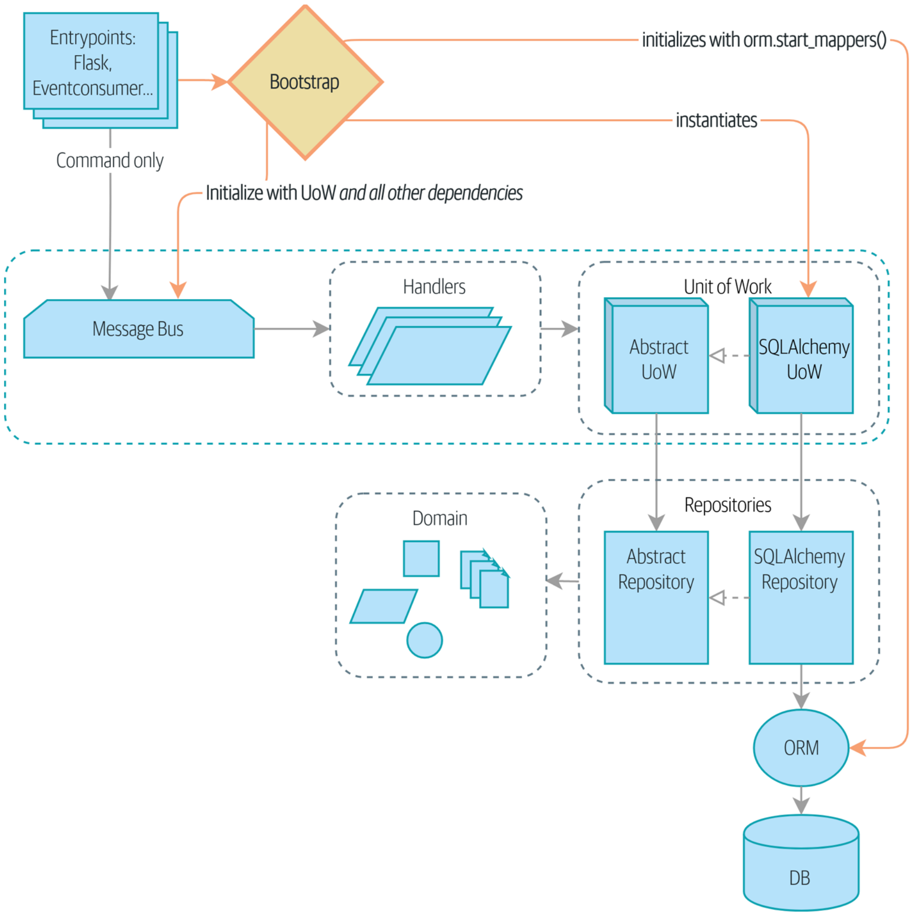


Figure 13-2. Bootstrap takes care of all that in one place

Implicit Versus Explicit Dependencies

Depending on your particular brain type, you may have a slight feeling of unease at the back of your mind at this point. Let's bring it out into the open. We've shown you two ways of managing dependencies and testing them.

For our database dependency, we've built a careful framework of explicit dependencies and easy options for overriding them in tests. Our main handler functions declare an explicit dependency on the UoW:

Our handlers have an explicit dependency on the UoW (src/allocation/service_layer/handlers.py)

```
def allocate(
        cmd: commands.Allocate, uow: unit_of_work.AbstractUnitOfWork
):
```

And that makes it easy to swap in a fake UoW in our service-layer tests:

Service-layer tests against a fake UoW: (tests/unit/test_services.py)

```
    uow = FakeUnitOfWork()
    messagebus.handle([...], uow)
```

The UoW itself declares an explicit dependency on the session factory:

The UoW depends on a session factory (src/allocation/service_layer/unit_of_work.py)

```
class SqlAlchemyUnitOfWork(AbstractUnitOfWork):

    def __init__(self, session_factory=DEFAULT_SESSION_FACTORY):
        self.session_factory = session_factory
        ...
```

We take advantage of it in our integration tests to be able to sometimes use SQLite instead of Postgres:

Integration tests against a different DB (tests/integration/test_uow.py)

```
def test_rolls_back_uncommitted_work_by_default(sqlite_session_factory):
    uow = unit_of_work.SqlAlchemyUnitOfWork(sqlite_session_factory)  ❶
```

❶ Integration tests swap out the default Postgres `session_factory` for a SQLite one.

Aren't Explicit Dependencies Totally Weird and Java-y?

If you're used to the way things normally happen in Python, you'll be thinking all this is a bit weird. The standard way to do things is to declare our dependency implicitly by simply importing it, and then if we ever need to change it for tests, we can monkeypatch, as is Right and True in dynamic languages:

Email sending as a normal import-based dependency (src/allocation/service_layer/handlers.py)

```
from allocation.adapters import email, redis_eventpublisher  ❶
...

def send_out_of_stock_notification(
        event: events.OutOfStock, uow: unit_of_work.AbstractUnitOfWork,
):
    email.send(  ❷
        'stock@made.com',
        f'Out of stock for {event.sku}',
    )
```

❶ Hardcoded import

❷ Calls specific email sender directly

Why pollute our application code with unnecessary arguments just for the sake of our tests? `mock.patch` makes monkeypatching nice and easy:

mock dot patch, thank you Michael Foord (tests/unit/test_handlers.py)

```
    with mock.patch("allocation.adapters.email.send") as mock_send_mail:
        ...
```

The trouble is that we've made it look easy because our toy example doesn't send real email (`email.send_mail` just does a `print`), but in real life, you'd end up having to call `mock.patch` for *every single test* that might cause an out-of-stock notification. If you've worked on codebases with lots of mocks used to prevent unwanted side effects, you'll know how annoying that mocky boilerplate gets.

And you'll know that mocks tightly couple us to the implementation. By choosing to monkeypatch `email.send_mail`, we are tied to doing `import email`, and if we ever want to do `from email import send_mail`, a trivial refactor, we'd have to change all our mocks.

So it's a trade-off. Yes, declaring explicit dependencies is unnecessary, strictly speaking, and using them would make our application code marginally more complex. But in return, we'd get tests that are easier to write and manage.

On top of that, declaring an explicit dependency is an example of the dependency inversion principle—rather than having an (implicit) dependency on a *specific* detail, we have an (explicit) dependency on an *abstraction*:

> Explicit is better than implicit.
>
> —The Zen of Python

The explicit dependency is more abstract (src/allocation/service_layer/handlers.py)

```
def send_out_of_stock_notification(
        event: events.OutOfStock, send_mail: Callable,
):
    send_mail(
        'stock@made.com',
        f'Out of stock for {event.sku}',
    )
```

But if we do change to declaring all these dependencies explicitly, who will inject them, and how? So far, we've really been dealing with only passing the UoW around:

our tests use `FakeUnitOfWork`, while Flask and Redis eventconsumer entrypoints use the real UoW, and the message bus passes them onto our command handlers. If we add real and fake email classes, who will create them and pass them on?

That's extra (duplicated) cruft for Flask, Redis, and our tests. Moreover, putting all the responsibility for passing dependencies to the right handler onto the message bus feels like a violation of the SRP.

Instead, we'll reach for a pattern called *Composition Root* (a bootstrap script to you and me),[1] and we'll do a bit of "manual DI" (dependency injection without a framework). See Figure 13-3.[2]

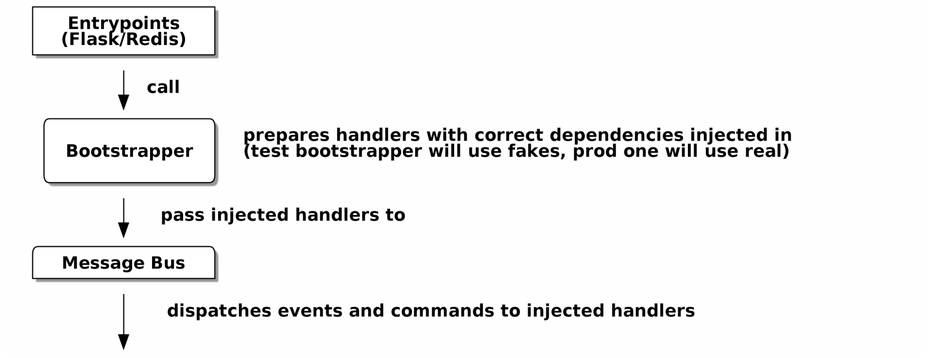

Entrypoints (Flask/Redis)
call
Bootstrapper
prepares handlers with correct dependencies injected in (test bootstrapper will use fakes, prod one will use real)
pass injected handlers to
Message Bus
dispatches events and commands to injected handlers

Figure 13-3. Bootstrapper between entrypoints and message bus

Preparing Handlers: Manual DI with Closures and Partials

One way to turn a function with dependencies into one that's ready to be called later with those dependencies *already injected* is to use closures or partial functions to compose the function with its dependencies:

Examples of DI using closures or partial functions

```
# existing allocate function, with abstract uow dependency
def allocate(
        cmd: commands.Allocate, uow: unit_of_work.AbstractUnitOfWork
):
    line = OrderLine(cmd.orderid, cmd.sku, cmd.qty)
    with uow:
        ...
```

1 Because Python is not a "pure" OO language, Python developers aren't necessarily used to the concept of needing to *compose* a set of objects into a working application. We just pick our entrypoint and run code from top to bottom.

2 Mark Seemann calls this *Pure DI* or sometimes *Vanilla DI*.

```
# bootstrap script prepares actual UoW

def bootstrap(..):
    uow = unit_of_work.SqlAlchemyUnitOfWork()

    # prepare a version of the allocate fn with UoW dependency captured in a closure
    allocate_composed = lambda cmd: allocate(cmd, uow)

    # or, equivalently (this gets you a nicer stack trace)
    def allocate_composed(cmd):
        return allocate(cmd, uow)

    # alternatively with a partial
    import functools
    allocate_composed = functools.partial(allocate, uow=uow)  (1)

# later at runtime, we can call the partial function, and it will have
# the UoW already bound
allocate_composed(cmd)
```

(1) The difference between closures (lambdas or named functions) and `functools.partial` is that the former use late binding of variables, which can be a source of confusion if any of the dependencies are mutable.

Here's the same pattern again for the `send_out_of_stock_notification()` handler, which has different dependencies:

Another closure and partial functions example

```
def send_out_of_stock_notification(
        event: events.OutOfStock, send_mail: Callable,
):
    send_mail(
        'stock@made.com',
        ...


# prepare a version of the send_out_of_stock_notification with dependencies
sosn_composed  = lambda event: send_out_of_stock_notification(event, email.send_mail)

...
# later, at runtime:
sosn_composed(event)  # will have email.send_mail already injected in
```

An Alternative Using Classes

Closures and partial functions will feel familiar to people who've done a bit of functional programming. Here's an alternative using classes, which may appeal to others. It requires rewriting all our handler functions as classes, though:

```
# we replace the old `def allocate(cmd, uow)` with:

class AllocateHandler:

    def __init__(self, uow: unit_of_work.AbstractUnitOfWork):  ❷
        self.uow = uow

    def __call__(self, cmd: commands.Allocate):  ❶
        line = OrderLine(cmd.orderid, cmd.sku, cmd.qty)
        with self.uow:
            # rest of handler method as before
            ...

# bootstrap script prepares actual UoW
uow = unit_of_work.SqlAlchemyUnitOfWork()

# then prepares a version of the allocate fn with dependencies already injected
allocate = AllocateHandler(uow)

...
# later at runtime, we can call the handler instance, and it will have
# the UoW already injected
allocate(cmd)
```

❶ The class is designed to produce a callable function, so it has a *`call`* method.

❷ But we use the `init` to declare the dependencies it requires. This sort of thing will feel familiar if you've ever made class-based descriptors, or a class-based context manager that takes arguments.

Use whichever you and your team feel more comfortable with.

A Bootstrap Script

We want our bootstrap script to do the following:

1. Declare default dependencies but allow us to override them
2. Do the "init" stuff that we need to get our app started
3. Inject all the dependencies into our handlers
4. Give us back the core object for our app, the message bus

Here's a first cut:

A bootstrap function (src/allocation/bootstrap.py)

```
def bootstrap(
    start_orm: bool = True,  ❶
    uow: unit_of_work.AbstractUnitOfWork = unit_of_work.SqlAlchemyUnitOfWork(),  ❷
    send_mail: Callable = email.send,
    publish: Callable = redis_eventpublisher.publish,
) -> messagebus.MessageBus:

    if start_orm:
        orm.start_mappers()  ❶

    dependencies = {'uow': uow, 'send_mail': send_mail, 'publish': publish}
    injected_event_handlers = {  ❸
        event_type: [
            inject_dependencies(handler, dependencies)
            for handler in event_handlers
        ]
        for event_type, event_handlers in handlers.EVENT_HANDLERS.items()
    }
    injected_command_handlers = {  ❸
        command_type: inject_dependencies(handler, dependencies)
        for command_type, handler in handlers.COMMAND_HANDLERS.items()
    }

    return messagebus.MessageBus(  ❹
        uow=uow,
        event_handlers=injected_event_handlers,
        command_handlers=injected_command_handlers,
    )
```

❶ `orm.start_mappers()` is our example of initialization work that needs to be done once at the beginning of an app. We also see things like setting up the `logging` module.

❷ We can use the argument defaults to define what the normal/production defaults are. It's nice to have them in a single place, but sometimes dependencies have some side effects at construction time, in which case you might prefer to default them to `None` instead.

❸ We build up our injected versions of the handler mappings by using a function called `inject_dependencies()`, which we'll show next.

❹ We return a configured message bus ready for use.

Here's how we inject dependencies into a handler function by inspecting it:

DI by inspecting function signatures (src/allocation/bootstrap.py)

```
def inject_dependencies(handler, dependencies):
    params = inspect.signature(handler).parameters  ❶
    deps = {
        name: dependency
        for name, dependency in dependencies.items()  ❷
        if name in params
    }
    return lambda message: handler(message, **deps)  ❸
```

❶ We inspect our command/event handler's arguments.

❷ We match them by name to our dependencies.

❸ We inject them as kwargs to produce a partial.

Even-More-Manual DI with Less Magic

If you're finding the preceding `inspect` code a little harder to grok, this even simpler version may appeal to you.

Harry wrote the code for `inject_dependencies()` as a first cut of how to do "manual" dependency injection, and when he saw it, Bob accused him of overengineering and writing his own DI framework.

It honestly didn't even occur to Harry that you could do it any more plainly, but you can, like this:

Manually creating partial functions inline (src/allocation/bootstrap.py)

```
injected_event_handlers = {
    events.Allocated: [
        lambda e: handlers.publish_allocated_event(e, publish),
        lambda e: handlers.add_allocation_to_read_model(e, uow),
    ],
    events.Deallocated: [
        lambda e: handlers.remove_allocation_from_read_model(e, uow),
        lambda e: handlers.reallocate(e, uow),
    ],
    events.OutOfStock: [
        lambda e: handlers.send_out_of_stock_notification(e, send_mail)
    ]
}
injected_command_handlers = {
    commands.Allocate: lambda c: handlers.allocate(c, uow),
    commands.CreateBatch: \
        lambda c: handlers.add_batch(c, uow),
    commands.ChangeBatchQuantity: \
        lambda c: handlers.change_batch_quantity(c, uow),
}
```

Harry says he couldn't even imagine writing out that many lines of code and having to look up that many function arguments manually. This is a perfectly viable solution, though, since it's only one line of code or so per handler you add, and thus not a massive maintenance burden even if you have dozens of handlers.

Our app is structured in such a way that we always want to do dependency injection in only one place, the handler functions, so this super-manual solution and Harry's `inspect()`-based one will both work fine.

If you find yourself wanting to do DI in more things and at different times, or if you ever get into *dependency chains* (in which your dependencies have their own dependencies, and so on), you may get some mileage out of a "real" DI framework.

At MADE, we've used Inject in a few places, and it's fine, although it makes Pylint unhappy. You might also check out Punq, as written by Bob himself, or the DRY-Python crew's dependencies.

Message Bus Is Given Handlers at Runtime

Our message bus will no longer be static; it needs to have the already-injected handlers given to it. So we turn it from being a module into a configurable class:

MessageBus as a class (src/allocation/service_layer/messagebus.py)

```
class MessageBus:  ❶

    def __init__(
        self,
        uow: unit_of_work.AbstractUnitOfWork,
        event_handlers: Dict[Type[events.Event], List[Callable]],  ❷
        command_handlers: Dict[Type[commands.Command], Callable],  ❷
    ):
        self.uow = uow
        self.event_handlers = event_handlers
        self.command_handlers = command_handlers

    def handle(self, message: Message):  ❸
        self.queue = [message]  ❹
        while self.queue:
            message = self.queue.pop(0)
            if isinstance(message, events.Event):
                self.handle_event(message)
            elif isinstance(message, commands.Command):
                self.handle_command(message)
            else:
                raise Exception(f'{message} was not an Event or Command')
```

❶ The message bus becomes a class…

❷ …which is given its already-dependency-injected handlers.

❸ The main `handle()` function is substantially the same, with just a few attributes and methods moved onto `self`.

❹ Using `self.queue` like this is not thread-safe, which might be a problem if you're using threads, because the bus instance is global in the Flask app context as we've written it. Just something to watch out for.

What else changes in the bus?

Event and command handler logic stays the same (src/allocation/service_layer/messagebus.py)

```
    def handle_event(self, event: events.Event):
        for handler in self.event_handlers[type(event)]:  ❶
            try:
                logger.debug('handling event %s with handler %s', event, handler)
                handler(event)  ❷
                self.queue.extend(self.uow.collect_new_events())
            except Exception:
                logger.exception('Exception handling event %s', event)
                continue


    def handle_command(self, command: commands.Command):
        logger.debug('handling command %s', command)
        try:
            handler = self.command_handlers[type(command)]  ❶
            handler(command)  ❷
            self.queue.extend(self.uow.collect_new_events())
        except Exception:
            logger.exception('Exception handling command %s', command)
            raise
```

❶ `handle_event` and `handle_command` are substantially the same, but instead of indexing into a static `EVENT_HANDLERS` or `COMMAND_HANDLERS` dict, they use the versions on `self`.

❷ Instead of passing a UoW into the handler, we expect the handlers to already have all their dependencies, so all they need is a single argument, the specific event or command.

Using Bootstrap in Our Entrypoints

In our application's entrypoints, we now just call `bootstrap.bootstrap()` and get a message bus that's ready to go, rather than configuring a UoW and the rest of it:

Flask calls bootstrap (src/allocation/entrypoints/flask_app.py)

```
-from allocation import views
+from allocation import bootstrap, views

 app = Flask(__name__)
-orm.start_mappers()  ❶
+bus = bootstrap.bootstrap()


 @app.route("/add_batch", methods=['POST'])
@@ -19,8 +16,7 @@ def add_batch():
     cmd = commands.CreateBatch(
         request.json['ref'], request.json['sku'], request.json['qty'], eta,
     )
-    uow = unit_of_work.SqlAlchemyUnitOfWork()  ❷
-    messagebus.handle(cmd, uow)
+    bus.handle(cmd)  ❸
     return 'OK', 201
```

❶ We no longer need to call `start_orm()`; the bootstrap script's initialization stages will do that.

❷ We no longer need to explicitly build a particular type of UoW; the bootstrap script defaults take care of it.

❸ And our message bus is now a specific instance rather than the global module.[3]

3 However, it's still a global in the `flask_app` module scope, if that makes sense. This may cause problems if you ever find yourself wanting to test your Flask app in-process by using the Flask Test Client instead of using Docker as we do. It's worth researching Flask app factories if you get into this.

Initializing DI in Our Tests

In tests, we can use `bootstrap.bootstrap()` with overridden defaults to get a custom message bus. Here's an example in an integration test:

Overriding bootstrap defaults (tests/integration/test_views.py)

```
@pytest.fixture
def sqlite_bus(sqlite_session_factory):
    bus = bootstrap.bootstrap(
        start_orm=True,  ❶
        uow=unit_of_work.SqlAlchemyUnitOfWork(sqlite_session_factory),  ❷
        send_mail=lambda *args: None,  ❸
        publish=lambda *args: None,  ❸
    )
    yield bus
    clear_mappers()

def test_allocations_view(sqlite_bus):
    sqlite_bus.handle(commands.CreateBatch('sku1batch', 'sku1', 50, None))
    sqlite_bus.handle(commands.CreateBatch('sku2batch', 'sku2', 50, date.today()))
    ...
    assert views.allocations('order1', sqlite_bus.uow) == [
        {'sku': 'sku1', 'batchref': 'sku1batch'},
        {'sku': 'sku2', 'batchref': 'sku2batch'},
    ]
```

❶ We do still want to start the ORM…

❷ …because we're going to use a real UoW, albeit with an in-memory database.

❸ But we don't need to send email or publish, so we make those noops.

In our unit tests, in contrast, we can reuse our `FakeUnitOfWork`:

Bootstrap in unit test (tests/unit/test_handlers.py)

```
def bootstrap_test_app():
    return bootstrap.bootstrap(
        start_orm=False,  ❶
        uow=FakeUnitOfWork(),  ❷
        send_mail=lambda *args: None,  ❸
        publish=lambda *args: None,  ❸
    )
```

❶ No need to start the ORM…

❷ …because the fake UoW doesn't use one.

❸ We want to fake out our email and Redis adapters too.

So that gets rid of a little duplication, and we've moved a bunch of setup and sensible defaults into a single place.

Exercise for the Reader 1

Change all the handlers to being classes as per the DI using classes example, and amend the bootstrapper's DI code as appropriate. This will let you know whether you prefer the functional approach or the class-based approach when it comes to your own projects.

Building an Adapter "Properly": A Worked Example

To really get a feel for how it all works, let's work through an example of how you might "properly" build an adapter and do dependency injection for it.

At the moment, we have two types of dependencies:

Two types of dependencies (src/allocation/service_layer/messagebus.py)

```
    uow: unit_of_work.AbstractUnitOfWork,  ❶
    send_mail: Callable,  ❷
    publish: Callable,  ❷
```

❶ The UoW has an abstract base class. This is the heavyweight option for declaring and managing your external dependency. We'd use this for the case when the dependency is relatively complex.

❷ Our email sender and pub/sub publisher are defined as functions. This works just fine for simple dependencies.

Here are some of the things we find ourselves injecting at work:

- An S3 filesystem client
- A key/value store client
- A `requests` session object

Most of these will have more-complex APIs that you can't capture as a single function: read and write, GET and POST, and so on.

Even though it's simple, let's use `send_mail` as an example to talk through how you might define a more complex dependency.

Define the Abstract and Concrete Implementations

We'll imagine a more generic notifications API. Could be email, could be SMS, could be Slack posts one day.

An ABC and a concrete implementation (src/allocation/adapters/notifications.py)

```python
class AbstractNotifications(abc.ABC):

    @abc.abstractmethod
    def send(self, destination, message):
        raise NotImplementedError

...

class EmailNotifications(AbstractNotifications):

    def __init__(self, smtp_host=DEFAULT_HOST, port=DEFAULT_PORT):
        self.server = smtplib.SMTP(smtp_host, port=port)
        self.server.noop()

    def send(self, destination, message):
        msg = f'Subject: allocation service notification\n{message}'
        self.server.sendmail(
            from_addr='allocations@example.com',
            to_addrs=[destination],
            msg=msg
        )
```

We change the dependency in the bootstrap script:

Notifications in message bus (src/allocation/bootstrap.py)

```diff
 def bootstrap(
     start_orm: bool = True,
     uow: unit_of_work.AbstractUnitOfWork = unit_of_work.SqlAlchemyUnitOfWork(),
-    send_mail: Callable = email.send,
+    notifications: AbstractNotifications = EmailNotifications(),
     publish: Callable = redis_eventpublisher.publish,
 ) -> messagebus.MessageBus:
```

Make a Fake Version for Your Tests

We work through and define a fake version for unit testing:

Fake notifications (tests/unit/test_handlers.py)

```python
class FakeNotifications(notifications.AbstractNotifications):

    def __init__(self):
        self.sent = defaultdict(list)  # type: Dict[str, List[str]]

    def send(self, destination, message):
        self.sent[destination].append(message)
...
```

And we use it in our tests:

Tests change slightly (tests/unit/test_handlers.py)

```
    def test_sends_email_on_out_of_stock_error(self):
        fake_notifs = FakeNotifications()
        bus = bootstrap.bootstrap(
            start_orm=False,
            uow=FakeUnitOfWork(),
            notifications=fake_notifs,
            publish=lambda *args: None,
        )
        bus.handle(commands.CreateBatch("b1", "POPULAR-CURTAINS", 9, None))
        bus.handle(commands.Allocate("o1", "POPULAR-CURTAINS", 10))
        assert fake_notifs.sent['stock@made.com'] == [
            f"Out of stock for POPULAR-CURTAINS",
        ]
```

Figure Out How to Integration Test the Real Thing

Now we test the real thing, usually with an end-to-end or integration test. We've used MailHog as a real-ish email server for our Docker dev environment:

Docker-compose config with real fake email server (docker-compose.yml)

```
version: "3"

services:

  redis_pubsub:
    build:
      context: .
      dockerfile: Dockerfile
    image: allocation-image
    ...

  api:
    image: allocation-image
    ...

  postgres:
    image: postgres:9.6
    ...

  redis:
    image: redis:alpine
    ...

  mailhog:
    image: mailhog/mailhog
    ports:
      - "11025:1025"
      - "18025:8025"
```

In our integration tests, we use the real `EmailNotifications` class, talking to the MailHog server in the Docker cluster:

Integration test for email (tests/integration/test_email.py)

```
@pytest.fixture
def bus(sqlite_session_factory):
    bus = bootstrap.bootstrap(
        start_orm=True,
        uow=unit_of_work.SqlAlchemyUnitOfWork(sqlite_session_factory),
        notifications=notifications.EmailNotifications(),  ❶
        publish=lambda *args: None,
    )
    yield bus
    clear_mappers()


def get_email_from_mailhog(sku):  ❷
    host, port = map(config.get_email_host_and_port().get, ['host', 'http_port'])
    all_emails = requests.get(f'http://{host}:{port}/api/v2/messages').json()
    return next(m for m in all_emails['items'] if sku in str(m))


def test_out_of_stock_email(bus):
    sku = random_sku()
    bus.handle(commands.CreateBatch('batch1', sku, 9, None))  ❸
    bus.handle(commands.Allocate('order1', sku, 10))
    email = get_email_from_mailhog(sku)
    assert email['Raw']['From'] == 'allocations@example.com'  ❹
    assert email['Raw']['To'] == ['stock@made.com']
    assert f'Out of stock for {sku}' in email['Raw']['Data']
```

❶ We use our bootstrapper to build a message bus that talks to the real notifications class.

❷ We figure out how to fetch emails from our "real" email server.

❸ We use the bus to do our test setup.

❹ Against all the odds, this actually worked, pretty much at the first go!

And that's it really.

Exercise for the Reader 2

You could do two things for practice regarding adapters:

1. Try swapping out our notifications from email to SMS notifications using Twilio, for example, or Slack notifications. Can you find a good equivalent to MailHog for integration testing?
2. In a similar way to what we did moving from `send_mail` to a `Notifications` class, try refactoring our `redis_eventpublisher` that is currently just a `Callable` to some sort of more formal adapter/base class/protocol.

Wrap-Up

Once you have more than one adapter, you'll start to feel a lot of pain from passing dependencies around manually, unless you do some kind of *dependency injection.*

Setting up dependency injection is just one of many typical setup/initialization activities that you need to do just once when starting your app. Putting this all together into a *bootstrap script* is often a good idea.

The bootstrap script is also good as a place to provide sensible default configuration for your adapters, and as a single place to override those adapters with fakes for your tests.

A dependency injection framework can be useful if you find yourself needing to do DI at multiple levels—if you have chained dependencies of components that all need DI, for example.

This chapter also presented a worked example of changing an implicit/simple dependency into a "proper" adapter, factoring out an ABC, defining its real and fake implementations, and thinking through integration testing.

DI and Bootstrap Recap

In summary:

1. Define your API using an ABC.
2. Implement the real thing.
3. Build a fake and use it for unit/service-layer/handler tests.
4. Find a less fake version you can put into your Docker environment.
5. Test the less fake "real" thing.
6. Profit!

These were the last patterns we wanted to cover, which brings us to the end of Part II. In the epilogue, we'll try to give you some pointers for applying these techniques in the Real World™.

Epilogue

What Now?

Phew! We've covered a lot of ground in this book, and for most of our audience all of these ideas are new. With that in mind, we can't hope to make you experts in these techniques. All we can really do is show you the broad-brush ideas, and just enough code for you to go ahead and write something from scratch.

The code we've shown in this book isn't battle-hardened production code: it's a set of Lego blocks that you can play with to make your first house, spaceship, and skyscraper.

That leaves us with two big tasks. We want to talk about how to start applying these ideas for real in an existing system, and we need to warn you about some of the things we had to skip. We've given you a whole new arsenal of ways to shoot yourself in the foot, so we should discuss some basic firearms safety.

How Do I Get There from Here?

Chances are that a lot of you are thinking something like this:

"OK Bob and Harry, that's all well and good, and if I ever get hired to work on a green-field new service, I know what to do. But in the meantime, I'm here with my big ball of Django mud, and I don't see any way to get to your nice, clean, perfect, untainted, simplistic model. Not from here."

We hear you. Once you've already *built* a big ball of mud, it's hard to know how to start improving things. Really, we need to tackle things step by step.

First things first: what problem are you trying to solve? Is the software too hard to change? Is the performance unacceptable? Have you got weird, inexplicable bugs?

Having a clear goal in mind will help you to prioritize the work that needs to be done and, importantly, communicate the reasons for doing it to the rest of the team.

Businesses tend to have pragmatic approaches to technical debt and refactoring, so long as engineers can make a reasoned argument for fixing things.

Making complex changes to a system is often an easier sell if you link it to feature work. Perhaps you're launching a new product or opening your service to new markets? This is the right time to spend engineering resources on fixing the foundations. With a six-month project to deliver, it's easier to make the argument for three weeks of cleanup work. Bob refers to this as *architecture tax*.

Separating Entangled Responsibilities

At the beginning of the book, we said that the main characteristic of a big ball of mud is homogeneity: every part of the system looks the same, because we haven't been clear about the responsibilities of each component. To fix that, we'll need to start separating out responsibilities and introducing clear boundaries. One of the first things we can do is to start building a service layer (Figure E-1).

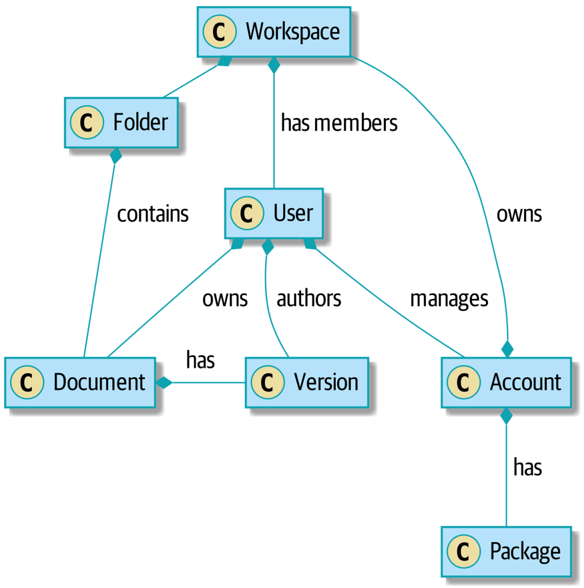


Figure E-1. Domain of a collaboration system

This was the system in which Bob first learned how to break apart a ball of mud, and it was a doozy. There was logic *everywhere*—in the web pages, in manager objects, in helpers, in fat service classes that we'd written to abstract the managers and helpers, and in hairy command objects that we'd written to break apart the services.

If you're working in a system that's reached this point, the situation can feel hopeless, but it's never too late to start weeding an overgrown garden. Eventually, we hired an architect who knew what he was doing, and he helped us get things back under control.

Start by working out the *use cases* of your system. If you have a user interface, what actions does it perform? If you have a backend processing component, maybe each cron job or Celery job is a single use case. Each of your use cases needs to have an imperative name: Apply Billing Charges, Clean Abandoned Accounts, or Raise Purchase Order, for example.

In our case, most of our use cases were part of the manager classes and had names like Create Workspace or Delete Document Version. Each use case was invoked from a web frontend.

We aim to create a single function or class for each of these supported operations that deals with *orchestrating* the work to be done. Each use case should do the following:

- Start its own database transaction if needed
- Fetch any required data
- Check any preconditions (see the Ensure pattern in Appendix E)
- Update the domain model
- Persist any changes

Each use case should succeed or fail as an atomic unit. You might need to call one use case from another. That's OK; just make a note of it, and try to avoid long-running database transactions.

One of the biggest problems we had was that manager methods called other manager methods, and data access could happen from the model objects themselves. It was hard to understand what each operation did without going on a treasure hunt across the codebase. Pulling all the logic into a single method, and using a UoW to control our transactions, made the system easier to reason about.

Case Study: Layering an Overgrown System

Many years ago, Bob worked for a software company that had outsourced the first version of its application, an online collaboration platform for sharing and working on files.

When the company brought development in-house, it passed through several generations of developers' hands, and each wave of new developers added more complexity to the code's structure.

At its heart, the system was an ASP.NET Web Forms application, built with an NHibernate ORM. Users would upload documents into workspaces, where they could invite other workspace members to review, comment on, or modify their work.

Most of the complexity of the application was in the permissions model because each document was contained in a folder, and folders allowed read, write, and edit permissions, much like a Linux filesystem.

Additionally, each workspace belonged to an account, and the account had quotas attached to it via a billing package.

As a result, every read or write operation against a document had to load an enormous number of objects from the database in order to test permissions and quotas. Creating a new workspace involved hundreds of database queries as we set up the permissions structure, invited users, and set up sample content.

Some of the code for operations was in web handlers that ran when a user clicked a button or submitted a form; some of it was in manager objects that held code for orchestrating work; and some of it was in the domain model. Model objects would make database calls or copy files on disk, and the test coverage was abysmal.

To fix the problem, we first introduced a service layer so that all of the code for creating a document or workspace was in one place and could be understood. This involved pulling data access code out of the domain model and into command handlers. Likewise, we pulled orchestration code out of the managers and the web handlers and pushed it into handlers.

The resulting command handlers were *long* and messy, but we'd made a start at introducing order to the chaos.

It's fine if you have duplication in the use-case functions. We're not trying to write perfect code; we're just trying to extract some meaningful layers. It's better to duplicate some code in a few places than to have use-case functions calling one another in a long chain.

This is a good opportunity to pull any data-access or orchestration code out of the domain model and into the use cases. We should also try to pull I/O concerns (e.g., sending email, writing files) out of the domain model and up into the use-case functions. We apply the techniques from Chapter 3 on abstractions to keep our handlers unit testable even when they're performing I/O.

These use-case functions will mostly be about logging, data access, and error handling. Once you've done this step, you'll have a grasp of what your program actually *does*, and a way to make sure each operation has a clearly defined start and finish. We'll have taken a step toward building a pure domain model.

Read *Working Effectively with Legacy Code* by Michael C. Feathers (Prentice Hall) for guidance on getting legacy code under test and starting separating responsibilities.

Identifying Aggregates and Bounded Contexts

Part of the problem with the codebase in our case study was that the object graph was highly connected. Each account had many workspaces, and each workspace had many members, all of whom had their own accounts. Each workspace contained many documents, which had many versions.

You can't express the full horror of the thing in a class diagram. For one thing, there wasn't really a single account related to a user. Instead, there was a bizarre rule requiring you to enumerate all of the accounts associated to the user via the workspaces and take the one with the earliest creation date.

Every object in the system was part of an inheritance hierarchy that included `SecureObject` and `Version`. This inheritance hierarchy was mirrored directly in the database schema, so that every query had to join across 10 different tables and look at a discriminator column just to tell what kind of objects you were working with.

The codebase made it easy to "dot" your way through these objects like so:

```
user.account.workspaces[0].documents.versions[1].owner.account.settings[0];
```

Building a system this way with Django ORM or SQLAlchemy is easy but is to be avoided. Although it's *convenient*, it makes it very hard to reason about performance because each property might trigger a lookup to the database.

Aggregates are a *consistency boundary*. In general, each use case should update a single aggregate at a time. One handler fetches one aggregate from a repository, modifies its state, and raises any events that happen as a result. If you need data from another part of the system, it's totally fine to use a read model, but avoid updating multiple aggregates in a single transaction. When we choose to separate code into different aggregates, we're explicitly choosing to make them *eventually consistent* with one another.

A bunch of operations required us to loop over objects this way—for example:

```
# Lock a user's workspaces for nonpayment

def lock_account(user):
    for workspace in user.account.workspaces:
        workspace.archive()
```

Or even recurse over collections of folders and documents:

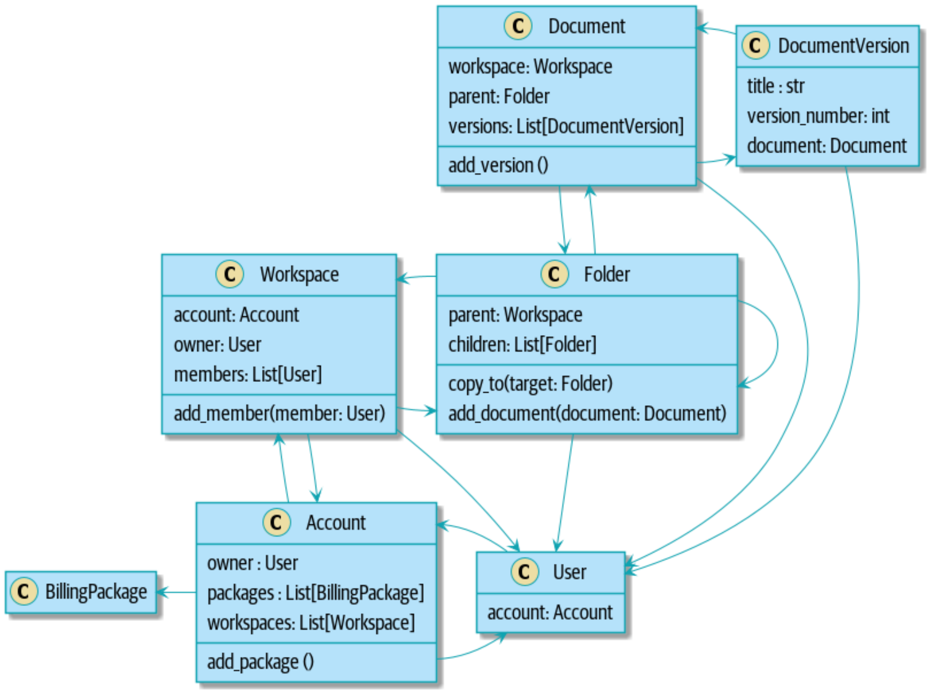
```
def lock_documents_in_folder(folder):

    for doc in folder.documents:
         doc.archive()

     for child in folder.children:
         lock_documents_in_folder(child)
```

These operations *killed* performance, but fixing them meant giving up our single object graph. Instead, we began to identify aggregates and to break the direct links between objects.

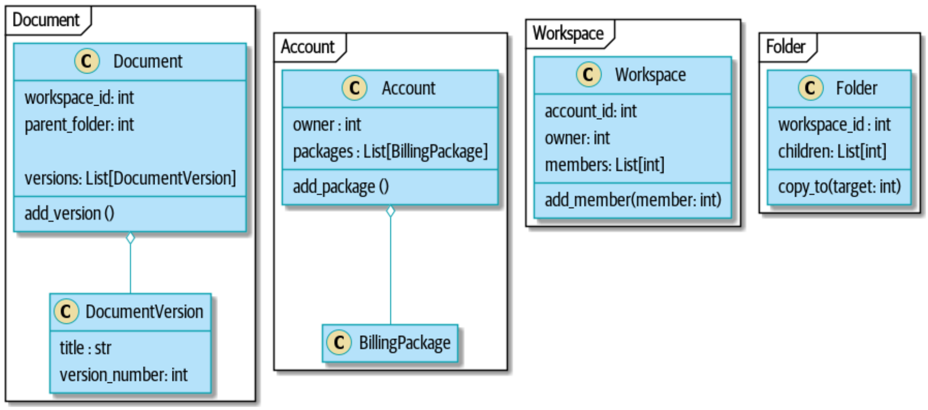
We talked about the infamous `SELECT N+1` problem in Chapter 12, and how we might choose to use different techniques when reading data for queries versus reading data for commands.

Mostly we did this by replacing direct references with identifiers.

Before aggregates:



After modeling with aggregates:



Bidirectional links are often a sign that your aggregates aren't right. In our original code, a `Document` knew about its containing `Folder`, and the `Folder` had a collection of `Documents`. This makes it easy to traverse the object graph but stops us from thinking properly about the consistency boundaries we need. We break apart aggregates by using references instead. In the new model, a `Document` had reference to its `parent_folder` but had no way to directly access the `Folder`.

If we needed to *read* data, we avoided writing complex loops and transforms and tried to replace them with straight SQL. For example, one of our screens was a tree view of folders and documents.

This screen was *incredibly* heavy on the database, because it relied on nested `for` loops that triggered a lazy-loaded ORM.

We use this same technique in Chapter 11, where we replace a nested loop over ORM objects with a simple SQL query. It's the first step in a CQRS approach.

After a lot of head-scratching, we replaced the ORM code with a big, ugly stored procedure. The code looked horrible, but it was much faster and helped to break the links between `Folder` and `Document`.

When we needed to *write* data, we changed a single aggregate at a time, and we introduced a message bus to handle events. For example, in the new model, when we locked an account, we could first query for all the affected workspaces via `SELECT *id* FROM *workspace* WHERE *account_id* = ?`.

We could then raise a new command for each workspace:

```
for workspace_id in workspaces:
    bus.handle(LockWorkspace(workspace_id))
```

An Event-Driven Approach to Go to Microservices via Strangler Pattern

The *Strangler Fig* pattern involves creating a new system around the edges of an old system, while keeping it running. Bits of old functionality are gradually intercepted and replaced, until the old system is left doing nothing at all and can be switched off.

When building the availability service, we used a technique called *event interception* to move functionality from one place to another. This is a three-step process:

1. Raise events to represent the changes happening in a system you want to replace.
2. Build a second system that consumes those events and uses them to build its own domain model.
3. Replace the older system with the new.

We used event interception to move from Figure E-2…

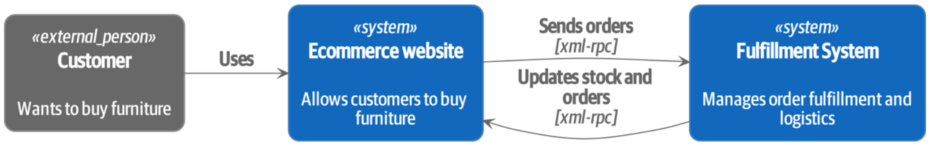


Figure E-2. Before: strong, bidirectional coupling based on XML-RPC

to Figure E-3.

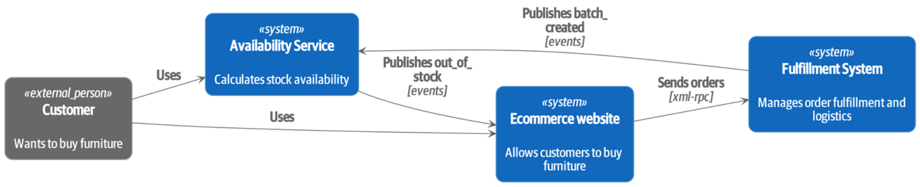


Figure E-3. After: loose coupling with asynchronous events (you can find a high-resolution version of this diagram at cosmicpython.com)

Practically, this was a several month-long project. Our first step was to write a domain model that could represent batches, shipments, and products. We used TDD to build a toy system that could answer a single question: "If I want N units of HAZARDOUS_RUG, how long will they take to be delivered?"

When deploying an event-driven system, start with a "walking skeleton." Deploying a system that just logs its input forces us to tackle all the infrastructural questions and start working in production.

Case Study: Carving Out a Microservice to Replace a Domain

MADE.com started out with *two* monoliths: one for the frontend ecommerce application, and one for the backend fulfillment system.

The two systems communicated through XML-RPC. Periodically, the backend system would wake up and query the frontend system to find out about new orders. When it had imported all the new orders, it would send RPC commands to update the stock levels.

Over time this synchronization process became slower and slower until, one Christmas, it took longer than 24 hours to import a single day's orders. Bob was hired to break the system into a set of event-driven services.

First, we identified that the slowest part of the process was calculating and synchronizing the available stock. What we needed was a system that could listen to external events and keep a running total of how much stock was available.

We exposed that information via an API, so that the user's browser could ask how much stock was available for each product and how long it would take to deliver to their address.

Whenever a product ran out of stock completely, we would raise a new event that the ecommerce platform could use to take a product off sale. Because we didn't know how much load we would need to handle, we wrote the system with a CQRS pattern. Whenever the amount of stock changed, we would update a Redis database with a cached view model. Our Flask API queried these *view models* instead of running the complex domain model.

As a result, we could answer the question "How much stock is available?" in 2 to 3 milliseconds, and now the API frequently handles hundreds of requests a second for sustained periods.

If this all sounds a little familiar, well, now you know where our example app came from!

Once we had a working domain model, we switched to building out some infrastructural pieces. Our first production deployment was a tiny system that could receive a `batch_created` event and log its JSON representation. This is the "Hello World" of event-driven architecture. It forced us to deploy a message bus, hook up a producer and consumer, build a deployment pipeline, and write a simple message handler.

Given a deployment pipeline, the infrastructure we needed, and a basic domain model, we were off. A couple months later, we were in production and serving real customers.

Convincing Your Stakeholders to Try Something New

If you're thinking about carving a new system out of a big ball of mud, you're probably suffering problems with reliability, performance, maintainability, or all three simultaneously. Deep, intractable problems call for drastic measures!

We recommend *domain modeling* as a first step. In many overgrown systems, the engineers, product owners, and customers no longer speak the same language. Business stakeholders speak about the system in abstract, process-focused terms, while developers are forced to speak about the system as it physically exists in its wild and chaotic state.

Case Study: The User Model

We mentioned earlier that the account and user model in our first system were bound together by a "bizarre rule." This is a perfect example of how engineering and business stakeholders can drift apart.

In this system, *accounts* parented *workspaces*, and users were *members* of workspaces. Workspaces were the fundamental unit for applying permissions and quotas. If a user *joined* a workspace and didn't already have an *account*, we would associate them with the account that owned that workspace.

This was messy and ad hoc, but it worked fine until the day a product owner asked for a new feature:

> When a user joins a company, we want to add them to some default workspaces for the company, like the HR workspace or the Company Announcements workspace.

We had to explain to them that there was *no such thing* as a company, and there was no sense in which a user joined an account. Moreover, a "company" might have *many* accounts owned by different users, and a new user might be invited to any one of them.

Years of adding hacks and work-arounds to a broken model caught up with us, and we had to rewrite the entire user management function as a brand-new system.

Figuring out how to model your domain is a complex task that's the subject of many decent books in its own right. We like to use interactive techniques like event storming and CRC modeling, because humans are good at collaborating through play. *Event modeling* is another technique that brings engineers and product owners together to understand a system in terms of commands, queries, and events.

Check out *www.eventmodeling.org* and *www.eventstorming.org* for some great guides to visual modeling of systems with events.

The goal is to be able to talk about the system by using the same ubiquitous language, so that you can agree on where the complexity lies.

We've found a lot of value in treating domain problems as TDD kata. For example, the first code we wrote for the availability service was the batch and order line model. You can treat this as a lunchtime workshop, or as a spike at the beginning of a project. Once you can demonstrate the value of modeling, it's easier to make the argument for structuring the project to optimize for modeling.

Case Study: David Seddon on Taking Small Steps

Hi, I'm David, one of the tech reviewers on this book. I've worked on several complex Django monoliths, and so I've known the pain that Bob and Harry have made all sorts of grand promises about soothing.

When I was first exposed to the patterns described here, I was rather excited. I had successfully used some of the techniques already on smaller projects, but here was a blueprint for much larger, database-backed systems like the one I work on in my day job. So I started trying to figure out how I could implement that blueprint at my current organization.

I chose to tackle a problem area of the codebase that had always bothered me. I began by implementing it as a use case. But I found myself running into unexpected questions. There were things that I hadn't considered while reading that now made it difficult to see what to do. Was it a problem if my use case interacted with two different aggregates? Could one use case call another? And how was it going to exist within a system that followed different architectural principles without resulting in a horrible mess?

What happened to that oh-so-promising blueprint? Did I actually understand the ideas well enough to put them into practice? Was it even suitable for my application? Even if it was, would any of my colleagues agree to such a major change? Were these just nice ideas for me to fantasize about while I got on with real life?

It took me a while to realize that I could start small. I didn't need to be a purist or to get it right the first time: I could experiment, finding what worked for me.

And so that's what I've done. I've been able to apply some *of the ideas in a few places. I've built new features whose business logic can be tested without the database or mocks. And as a team, we've introduced a service layer to help define the jobs the system does.*

If you start trying to apply these patterns in your work, you may go through similar feelings to begin with. When the nice theory of a book meets the reality of your codebase, it can be demoralizing.

My advice is to focus on a specific problem and ask yourself how you can put the relevant ideas to use, perhaps in an initially limited and imperfect fashion. You may discover, as I did, that the first problem you pick might be a bit too difficult; if so, move on to something else. Don't try to boil the ocean, and don't be too *afraid of making mistakes. It will be a learning experience, and you can be confident that you're moving roughly in a direction that others have found useful.*

So, if you're feeling the pain too, give these ideas a try. Don't feel you need permission to rearchitect everything. Just look for somewhere small to start. And above all, do it to solve a specific problem. If you're successful in solving it, you'll know you got something right—and others will too.

Questions Our Tech Reviewers Asked That We Couldn't Work into Prose

Here are some questions we heard during drafting that we couldn't find a good place to address elsewhere in the book:

Do I need to do all of this at once? Can I just do a bit at a time?
: No, you can absolutely adopt these techniques bit by bit. If you have an existing system, we recommend building a service layer to try to keep orchestration in one place. Once you have that, it's much easier to push logic into the model and push edge concerns like validation or error handling to the entrypoints.

  It's worth having a service layer even if you still have a big, messy Django ORM because it's a way to start understanding the boundaries of operations.

Extracting use cases will break a lot of my existing code; it's too tangled
: Just copy and paste. It's OK to cause more duplication in the short term. Think of this as a multistep process. Your code is in a bad state now, so copy and paste it to a new place and then make that new code clean and tidy.

  Once you've done that, you can replace uses of the old code with calls to your new code and finally delete the mess. Fixing large codebases is a messy and painful process. Don't expect things to get instantly better, and don't worry if some bits of your application stay messy.

Do I need to do CQRS? That sounds weird. Can't I just use repositories?
: Of course you can! The techniques we're presenting in this book are intended to make your life *easier*. They're not some kind of ascetic discipline with which to punish yourself.

In our first case-study system, we had a lot of *View Builder* objects that used repositories to fetch data and then performed some transformations to return dumb read models. The advantage is that when you hit a performance problem, it's easy to rewrite a view builder to use custom queries or raw SQL.

How should use cases interact across a larger system? Is it a problem for one to call another?

This might be an interim step. Again, in the first case study, we had handlers that would need to invoke other handlers. This gets *really* messy, though, and it's much better to move to using a message bus to separate these concerns.

Generally, your system will have a single message bus implementation and a bunch of subdomains that center on a particular aggregate or set of aggregates. When your use case has finished, it can raise an event, and a handler elsewhere can run.

Is it a code smell for a use case to use multiple repositories/aggregates, and if so, why?

An aggregate is a consistency boundary, so if your use case needs to update two aggregates atomically (within the same transaction), then your consistency boundary is wrong, strictly speaking. Ideally you should think about moving to a new aggregate that wraps up all the things you want to change at the same time.

If you're actually updating only one aggregate and using the other(s) for read-only access, then that's *fine*, although you could consider building a read/view model to get you that data instead—it makes things cleaner if each use case has only one aggregate.

If you do need to modify two aggregates, but the two operations don't have to be in the same transaction/UoW, then consider splitting the work out into two different handlers and using a domain event to carry information between the two. You can read more in these papers on aggregate design by Vaughn Vernon.

What if I have a read-only but business-logic-heavy system?

View models can have complex logic in them. In this book, we've encouraged you to separate your read and write models because they have different consistency and throughput requirements. Mostly, we can use simpler logic for reads, but that's not always true. In particular, permissions and authorization models can add a lot of complexity to our read side.

We've written systems in which the view models needed extensive unit tests. In those systems, we split a *view builder* from a *view fetcher*, as in Figure E-4.

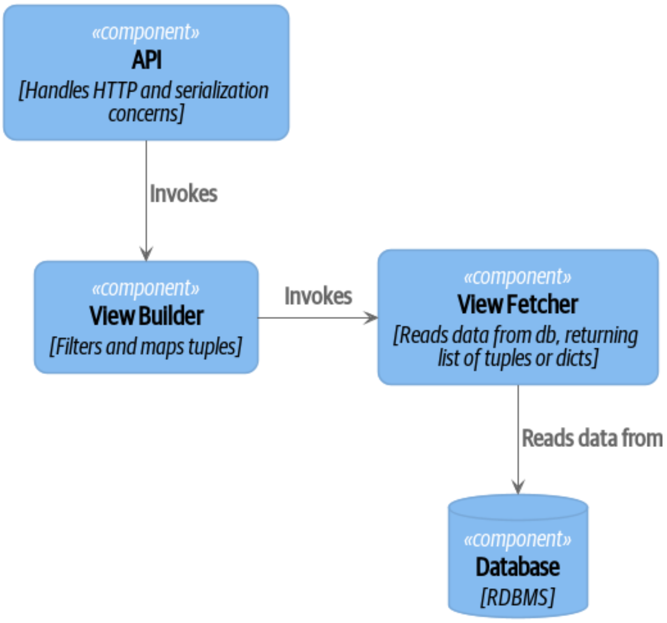

«component»
API
[Handles HTTP and serialization concerns]
Invokes
«component»
View Builder
[Filters and maps tuples]
Invokes
«component»
View Fetcher
[Reads data from db, returning list of tuples or dicts]
Reads data from
«component»
Database
[RDBMS]

Figure E-4. A view builder and view fetcher (you can find a high-resolution version of this diagram at cosmicpython.com)

+ This makes it easy to test the view builder by giving it mocked data (e.g., a list of dicts). "Fancy CQRS" with event handlers is really a way of running our complex view logic whenever we write so that we can avoid running it when we read.

Do I need to build microservices to do this stuff?
: Egads, no! These techniques predate microservices by a decade or so. Aggregates, domain events, and dependency inversion are ways to control complexity in large systems. It just so happens that when you've built a set of use cases and a model for a business process, moving it to its own service is relatively easy, but that's not a requirement.

I'm using Django. Can I still do this?
: We have an entire appendix just for you: Appendix D!

Footguns

OK, so we've given you a whole bunch of new toys to play with. Here's the fine print. Harry and Bob do not recommend that you copy and paste our code into a production system and rebuild your automated trading platform on Redis pub/sub. For reasons of brevity and simplicity, we've hand-waved a lot of tricky subjects. Here's a list of things we think you should know before trying this for real.

Reliable messaging is hard

Redis pub/sub is not reliable and shouldn't be used as a general-purpose messaging tool. We picked it because it's familiar and easy to run. At MADE, we run Event Store as our messaging tool, but we've had experience with RabbitMQ and Amazon EventBridge.

Tyler Treat has some excellent blog posts on his site *bravenewgeek.com*; you should read at least read "You Cannot Have Exactly-Once Delivery" and "What You Want Is What You Don't: Understanding Trade-Offs in Distributed Messaging".

We explicitly choose small, focused transactions that can fail independently

In Chapter 8, we update our process so that *deallocating* an order line and *reallocating* the line happen in two separate units of work. You will need monitoring to know when these transactions fail, and tooling to replay events. Some of this is made easier by using a transaction log as your message broker (e.g., Kafka or EventStore). You might also look at the Outbox pattern.

We don't discuss idempotency

We haven't given any real thought to what happens when handlers are retried. In practice you will want to make handlers idempotent so that calling them repeatedly with the same message will not make repeated changes to state. This is a key technique for building reliability, because it enables us to safely retry events when they fail.

There's a lot of good material on idempotent message handling, try starting with "How to Ensure Idempotency in an Eventual Consistent DDD/CQRS Application" and "(Un)Reliability in Messaging".

Your events will need to change their schema over time

You'll need to find some way of documenting your events and sharing schema with consumers. We like using JSON schema and markdown because it's simple but there is other prior art. Greg Young wrote an entire book on managing event-driven systems over time: *Versioning in an Event Sourced System* (Leanpub).

More Required Reading

A few more books we'd like to recommend to help you on your way:

- *Clean Architectures in Python* by Leonardo Giordani (Leanpub), which came out in 2019, is one of the few previous books on application architecture in Python.
- *Enterprise Integration Patterns* by Gregor Hohpe and Bobby Woolf (Addison-Wesley Professional) is a pretty good start for messaging patterns.
- *Monolith to Microservices* by Sam Newman (O'Reilly), and Newman's first book, *Building Microservices* (O'Reilly). The Strangler Fig pattern is mentioned as a favorite, along with many others. These are good to check out if you're thinking of moving to microservices, and they're also good on integration patterns and the considerations of async messaging-based integration.

Wrap-Up

Phew! That's a lot of warnings and reading suggestions; we hope we haven't scared you off completely. Our goal with this book is to give you just enough knowledge and intuition for you to start building some of this for yourself. We would love to hear how you get on and what problems you're facing with the techniques in your own systems, so why not get in touch with us over at *www.cosmicpython.com*?

APPENDIX A

Summary Diagram and Table

Here's what our architecture looks like by the end of the book:

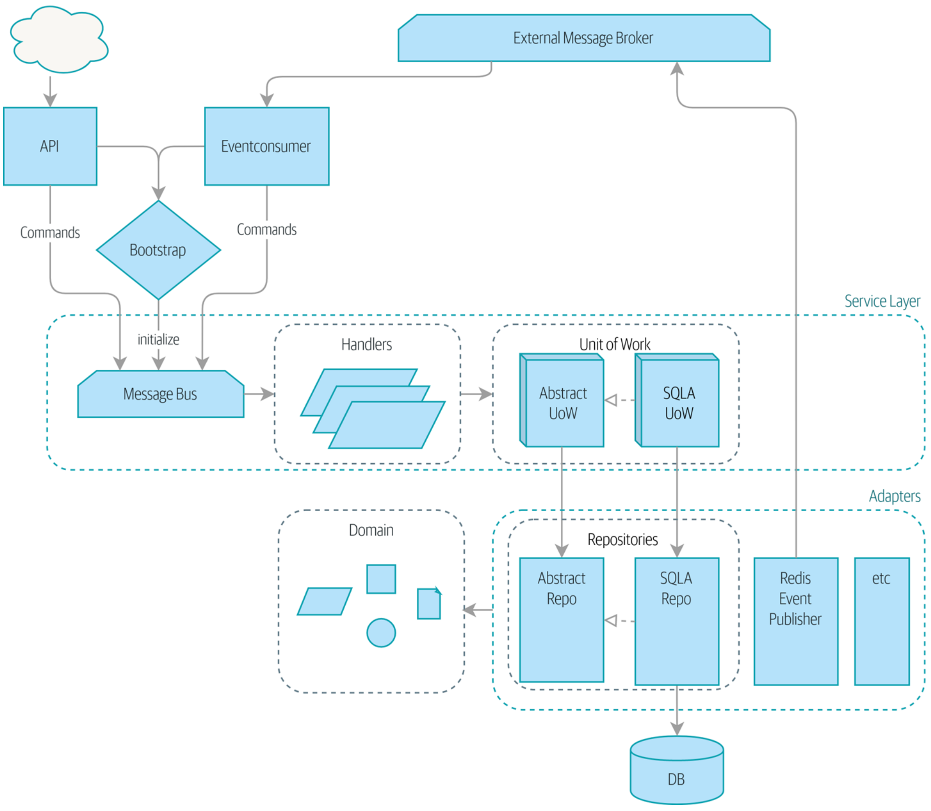


Table A-1 recaps each pattern and what it does.

Table A-1. The components of our architecture and what they all do

Layer	Component	Description
Domain *Defines the business logic.*	Entity	A domain object whose attributes may change but that has a recognizable identity over time.
	Value object	An immutable domain object whose attributes entirely define it. It is fungible with other identical objects.
	Aggregate	Cluster of associated objects that we treat as a unit for the purpose of data changes. Defines and enforces a consistency boundary.
	Event	Represents something that happened.
	Command	Represents a job the system should perform.
Service Layer *Defines the jobs the system should perform and orchestrates different components.*	Handler	Receives a command or an event and performs what needs to happen.
	Unit of work	Abstraction around data integrity. Each unit of work represents an atomic update. Makes repositories available. Tracks new events on retrieved aggregates.
	Message bus (internal)	Handles commands and events by routing them to the appropriate handler.
Adapters (Secondary) *Concrete implementations of an interface that goes from our system to the outside world (I/O).*	Repository	Abstraction around persistent storage. Each aggregate has its own repository.
	Event publisher	Pushes events onto the external message bus.
Entrypoints (Primary adapters) *Translate external inputs into calls into the service layer.*	Web	Receives web requests and translates them into commands, passing them to the internal message bus.
	Event consumer	Reads events from the external message bus and translates them into commands, passing them to the internal message bus.
N/A	External message bus (message broker)	A piece of infrastructure that different services use to intercommunicate, via events.

APPENDIX B

A Template Project Structure

Around Chapter 4, we moved from just having everything in one folder to a more structured tree, and we thought it might be of interest to outline the moving parts.

The code for this appendix is in the appendix_project_structure branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout appendix_project_structure
```

The basic folder structure looks like this:

Project tree

```
.
├── Dockerfile  ❶
├── Makefile  ❷
├── README.md
├── docker-compose.yml  ❶
├── license.txt
├── mypy.ini
├── requirements.txt
├── src  ❸
│   ├── allocation
│   │   ├── __init__.py
│   │   ├── adapters
│   │   │   ├── __init__.py
│   │   │   ├── orm.py
│   │   │   └── repository.py
│   │   ├── config.py
│   │   ├── domain
│   │   │   ├── __init__.py
│   │   │   └── model.py
│   │   ├── entrypoints
│   │   │   ├── __init__.py
│   │   │   └── flask_app.py
│   │   └── service_layer
│   │       ├── __init__.py
│   │       └── services.py
│   └── setup.py ❸
└── tests ❹
    ├── conftest.py ❹
    ├── e2e
    │   └── test_api.py
    ├── integration
    │   ├── test_orm.py
    │   └── test_repository.py
    ├── pytest.ini ❹
    └── unit
        ├── test_allocate.py
        ├── test_batches.py
        └── test_services.py
```

❶ Our *docker-compose.yml* and our *Dockerfile* are the main bits of configuration for the containers that run our app, and they can also run the tests (for CI). A more complex project might have several Dockerfiles, although we've found that minimizing the number of images is usually a good idea.[1]

❷ A *Makefile* provides the entrypoint for all the typical commands a developer (or a CI server) might want to run during their normal workflow: `make build`, `make test`, and so on.[2] This is optional. You could just use `docker-compose` and `pytest` directly, but if nothing else, it's nice to have all the "common commands" in a list somewhere, and unlike documentation, a Makefile is code so it has less tendency to become out of date.

❸ All the source code for our app, including the domain model, the Flask app, and infrastructure code, lives in a Python package inside *src*,[3] which we install using `pip install -e` and the *setup.py* file. This makes imports easy. Currently, the structure within this module is totally flat, but for a more complex project, you'd expect to grow a folder hierarchy that includes *domain_model/*, *infrastructure/*, *services/*, and *api/*.

1 Splitting out images for production and testing is sometimes a good idea, but we've tended to find that going further and trying to split out different images for different types of application code (e.g., Web API versus pub/sub client) usually ends up being more trouble than it's worth; the cost in terms of complexity and longer rebuild/CI times is too high. YMMV.

2 A pure-Python alternative to Makefiles is Invoke, worth checking out if everyone on your team knows Python (or at least knows it better than Bash!).

3 "Testing and Packaging" by Hynek Schlawack provides more information on *src* folders.

❹ Tests live in their own folder. Subfolders distinguish different test types and allow you to run them separately. We can keep shared fixtures (*conftest.py*) in the main tests folder and nest more specific ones if we wish. This is also the place to keep *pytest.ini*.

The pytest docs are really good on test layout and importability.

Let's look at a few of these files and concepts in more detail.

Env Vars, 12-Factor, and Config, Inside and Outside Containers

The basic problem we're trying to solve here is that we need different config settings for the following:

- Running code or tests directly from your own dev machine, perhaps talking to mapped ports from Docker containers
- Running on the containers themselves, with "real" ports and hostnames
- Different container environments (dev, staging, prod, and so on)

Configuration through environment variables as suggested by the 12-factor manifesto will solve this problem, but concretely, how do we implement it in our code and our containers?

Config.py

Whenever our application code needs access to some config, it's going to get it from a file called *config.py*. Here are a couple of examples from our app:

Sample config functions (src/allocation/config.py)

```
import os

def get_postgres_uri():  ❶
    host = os.environ.get('DB_HOST', 'localhost')  ❷
    port = 54321 if host == 'localhost' else 5432
    password = os.environ.get('DB_PASSWORD', 'abc123')
    user, db_name = 'allocation', 'allocation'
    return f"postgresql://{user}:{password}@{host}:{port}/{db_name}"
```

```
def get_api_url():
    host = os.environ.get('API_HOST', 'localhost')
    port = 5005 if host == 'localhost' else 80
    return f"http://{host}:{port}"
```

❶ We use functions for getting the current config, rather than constants available at import time, because that allows client code to modify `os.environ` if it needs to.

❷ *config.py* also defines some default settings, designed to work when running the code from the developer's local machine.[4]

An elegant Python package called *environ-config* is worth looking at if you get tired of hand-rolling your own environment-based config functions.

Don't let this config module become a dumping ground that is full of things only vaguely related to config and that is then imported all over the place. Keep things immutable and modify them only via environment variables. If you decide to use a bootstrap script, you can make it the only place (other than tests) that config is imported to.

Docker-Compose and Containers Config

We use a lightweight Docker container orchestration tool called *docker-compose*. It's main configuration is via a YAML file (sigh):[5]

docker-compose config file (docker-compose.yml)

```
version: "3"
services:

  app:  ❶
    build:
      context: .
      dockerfile: Dockerfile
    depends_on:
      - postgres
    environment:  ❸
      - DB_HOST=postgres  ❹
      - DB_PASSWORD=abc123
      - API_HOST=app
      - PYTHONDONTWRITEBYTECODE=1  ❺
```

4 This gives us a local development setup that "just works" (as much as possible). You may prefer to fail hard on missing environment variables instead, particularly if any of the defaults would be insecure in production.

5 Harry is a bit YAML-weary. It's *everywhere*, and yet he can never remember the syntax or how it's supposed to indent.

```
    volumes:  ❻
      - ./src:/src
      - ./tests:/tests
    ports:
      - "5005:80"  ❼


  postgres:
    image: postgres:9.6  ❷
    environment:
      - POSTGRES_USER=allocation
      - POSTGRES_PASSWORD=abc123
    ports:
      - "54321:5432"
```

❶ In the *docker-compose* file, we define the different *services* (containers) that we need for our app. Usually one main image contains all our code, and we can use it to run our API, our tests, or any other service that needs access to the domain model.

❷ You'll probably have other infrastructure services, including a database. In production you might not use containers for this; you might have a cloud provider instead, but *docker-compose* gives us a way of producing a similar service for dev or CI.

❸ The `environment` stanza lets you set the environment variables for your containers, the hostnames and ports as seen from inside the Docker cluster. If you have enough containers that information starts to be duplicated in these sections, you can use `environment_file` instead. We usually call ours *container.env*.

❹ Inside a cluster, *docker-compose* sets up networking such that containers are available to each other via hostnames named after their service name.

❺ Pro tip: if you're mounting volumes to share source folders between your local dev machine and the container, the `PYTHONDONTWRITEBYTECODE` environment variable tells Python to not write *.pyc* files, and that will save you from having millions of root-owned files sprinkled all over your local filesystem, being all annoying to delete and causing weird Python compiler errors besides.

❻ Mounting our source and test code as `volumes` means we don't need to rebuild our containers every time we make a code change.

❼ The `ports` section allows us to expose the ports from inside the containers to the outside world[6]—these correspond to the default ports we set in *config.py*.

> Inside Docker, other containers are available through hostnames named after their service name. Outside Docker, they are available on `localhost`, at the port defined in the `ports` section.

Installing Your Source as a Package

All our application code (everything except tests, really) lives inside an *src* folder:

The src folder

```
├── src
│   ├── allocation ❶
│   │   ├── config.py
│   │   └── ...
│   └── setup.py ❷
```

❶ Subfolders define top-level module names. You can have multiple if you like.

❷ And *setup.py* is the file you need to make it pip-installable, shown next.

pip-installable modules in three lines (src/setup.py)

```
from setuptools import setup

setup(
    name='allocation',
    version='0.1',
    packages=['allocation'],
)
```

That's all you need. `packages=` specifies the names of subfolders that you want to install as top-level modules. The `name` entry is just cosmetic, but it's required. For a package that's never actually going to hit PyPI, it'll do fine.[7]

6 On a CI server, you may not be able to expose arbitrary ports reliably, but it's only a convenience for local dev. You can find ways of making these port mappings optional (e.g., with *docker-compose.override.yml*).

7 For more *setup.py* tips, see this article on packaging by Hynek.

Dockerfile

Dockerfiles are going to be very project-specific, but here are a few key stages you'll expect to see:

Our Dockerfile (Dockerfile)

```
FROM python:3.8-alpine

❶
RUN apk add --no-cache --virtual .build-deps gcc postgresql-dev musl-dev python3-dev
RUN apk add libpq

❷
COPY requirements.txt /tmp/
RUN pip install -r /tmp/requirements.txt

RUN apk del --no-cache .build-deps

❸
RUN mkdir -p /src
COPY src/ /src/
RUN pip install -e /src
COPY tests/ /tests/

❹
WORKDIR /src
ENV FLASK_APP=allocation/entrypoints/flask_app.py FLASK_DEBUG=1 PYTHONUNBUFFERED=1
CMD flask run --host=0.0.0.0 --port=80
```

❶ Installing system-level dependencies

❷ Installing our Python dependencies (you may want to split out your dev from prod dependencies; we haven't here, for simplicity)

❸ Copying and installing our source

❹ Optionally configuring a default startup command (you'll probably override this a lot from the command line)

One thing to note is that we install things in the order of how frequently they are likely to change. This allows us to maximize Docker build cache reuse. I can't tell you how much pain and frustration underlies this lesson. For this and many more Python Dockerfile improvement tips, check out "Production-Ready Docker Packaging".

Tests

Our tests are kept alongside everything else, as shown here:

Tests folder tree

```
└── tests
    ├── conftest.py
    ├── e2e
    │   └── test_api.py
    ├── integration
    │   ├── test_orm.py
    │   └── test_repository.py
    ├── pytest.ini
    └── unit
        ├── test_allocate.py
        ├── test_batches.py
        └── test_services.py
```

Nothing particularly clever here, just some separation of different test types that you're likely to want to run separately, and some files for common fixtures, config, and so on.

There's no *src* folder or *setup.py* in the test folders because we usually haven't needed to make tests pip-installable, but if you have difficulties with import paths, you might find it helps.

Wrap-Up

These are our basic building blocks:

- Source code in an *src* folder, pip-installable using *setup.py*
- Some Docker config for spinning up a local cluster that mirrors production as far as possible
- Configuration via environment variables, centralized in a Python file called *con-fig.py*, with defaults allowing things to run *outside* containers
- A Makefile for useful command-line, um, commands

We doubt that anyone will end up with *exactly* the same solutions we did, but we hope you find some inspiration here.

APPENDIX C

Swapping Out the Infrastructure: Do Everything with CSVs

This appendix is intended as a little illustration of the benefits of the Repository, Unit of Work, and Service Layer patterns. It's intended to follow from Chapter 6.

Just as we finish building out our Flask API and getting it ready for release, the business comes to us apologetically, saying they're not ready to use our API and asking if we could build a thing that reads just batches and orders from a couple of CSVs and outputs a third CSV with allocations.

Ordinarily this is the kind of thing that might have a team cursing and spitting and making notes for their memoirs. But not us! Oh no, we've ensured that our infrastructure concerns are nicely decoupled from our domain model and service layer. Switching to CSVs will be a simple matter of writing a couple of new `Repository` and `UnitOfWork` classes, and then we'll be able to reuse *all* of our logic from the domain layer and the service layer.

Here's an E2E test to show you how the CSVs flow in and out:

A first CSV test (tests/e2e/test_csv.py)

```
def test_cli_app_reads_csvs_with_batches_and_orders_and_outputs_allocations(
        make_csv
):
    sku1, sku2 = random_ref('s1'), random_ref('s2')
    batch1, batch2, batch3 = random_ref('b1'), random_ref('b2'), random_ref('b3')
    order_ref = random_ref('o')
    make_csv('batches.csv', [
        ['ref', 'sku', 'qty', 'eta'],
        [batch1, sku1, 100, ''],
        [batch2, sku2, 100, '2011-01-01'],
        [batch3, sku2, 100, '2011-01-02'],
    ])
    orders_csv = make_csv('orders.csv', [
        ['orderid', 'sku', 'qty'],
        [order_ref, sku1, 3],
        [order_ref, sku2, 12],
    ])

    run_cli_script(orders_csv.parent)

    expected_output_csv = orders_csv.parent / 'allocations.csv'
    with open(expected_output_csv) as f:
        rows = list(csv.reader(f))
    assert rows == [
        ['orderid', 'sku', 'qty', 'batchref'],
        [order_ref, sku1, '3', batch1],
        [order_ref, sku2, '12', batch2],
    ]
```

Diving in and implementing without thinking about repositories and all that jazz, you might start with something like this:

A first cut of our CSV reader/writer (src/bin/allocate-from-csv)

```
#!/usr/bin/env python
import csv
import sys
from datetime import datetime
from pathlib import Path

from allocation import model

def load_batches(batches_path):
    batches = []
    with batches_path.open() as inf:
        reader = csv.DictReader(inf)
        for row in reader:
            if row['eta']:
                eta = datetime.strptime(row['eta'], '%Y-%m-%d').date()
            else:
                eta = None
            batches.append(model.Batch(
                ref=row['ref'],
                sku=row['sku'],
                qty=int(row['qty']),
                eta=eta
            ))
    return batches



def main(folder):
    batches_path = Path(folder) / 'batches.csv'
    orders_path = Path(folder) / 'orders.csv'
    allocations_path = Path(folder) / 'allocations.csv'

    batches = load_batches(batches_path)

    with orders_path.open() as inf, allocations_path.open('w') as outf:
        reader = csv.DictReader(inf)
        writer = csv.writer(outf)
        writer.writerow(['orderid', 'sku', 'batchref'])
        for row in reader:
            orderid, sku = row['orderid'], row['sku']
            qty = int(row['qty'])
            line = model.OrderLine(orderid, sku, qty)
            batchref = model.allocate(line, batches)
            writer.writerow([line.orderid, line.sku, batchref])



if __name__ == '__main__':
    main(sys.argv[1])
```

It's not looking too bad! And we're reusing our domain model objects and our domain service.

But it's not going to work. Existing allocations need to also be part of our permanent CSV storage. We can write a second test to force us to improve things:

And another one, with existing allocations (tests/e2e/test_csv.py)

```
def test_cli_app_also_reads_existing_allocations_and_can_append_to_them(
        make_csv
):
    sku = random_ref('s')
    batch1, batch2 = random_ref('b1'), random_ref('b2')
    old_order, new_order = random_ref('o1'), random_ref('o2')
    make_csv('batches.csv', [
        ['ref', 'sku', 'qty', 'eta'],
        [batch1, sku, 10, '2011-01-01'],
        [batch2, sku, 10, '2011-01-02'],
    ])
    make_csv('allocations.csv', [
        ['orderid', 'sku', 'qty', 'batchref'],
        [old_order, sku, 10, batch1],
    ])
    orders_csv = make_csv('orders.csv', [
        ['orderid', 'sku', 'qty'],
        [new_order, sku, 7],
    ])

    run_cli_script(orders_csv.parent)

    expected_output_csv = orders_csv.parent / 'allocations.csv'
    with open(expected_output_csv) as f:
        rows = list(csv.reader(f))
    assert rows == [
        ['orderid', 'sku', 'qty', 'batchref'],
        [old_order, sku, '10', batch1],
        [new_order, sku, '7', batch2],
    ]
```

And we could keep hacking about and adding extra lines to that `load_batches` function, and some sort of way of tracking and saving new allocations—but we already have a model for doing that! It's called our Repository and Unit of Work patterns.

All we need to do ("all we need to do") is reimplement those same abstractions, but with CSVs underlying them instead of a database. And as you'll see, it really is relatively straightforward.

Implementing a Repository and Unit of Work for CSVs

Here's what a CSV-based repository could look like. It abstracts away all the logic for reading CSVs from disk, including the fact that it has to read *two different CSVs* (one for batches and one for allocations), and it gives us just the familiar `.list()` API, which provides the illusion of an in-memory collection of domain objects:

A repository that uses CSV as its storage mechanism (src/allocation/service_layer/csv_uow.py)

```
class CsvRepository(repository.AbstractRepository):

    def __init__(self, folder):
        self._batches_path = Path(folder) / 'batches.csv'
        self._allocations_path = Path(folder) / 'allocations.csv'
        self._batches = {}  # type: Dict[str, model.Batch]
        self._load()

    def get(self, reference):
        return self._batches.get(reference)

    def add(self, batch):
        self._batches[batch.reference] = batch

    def _load(self):
        with self._batches_path.open() as f:
            reader = csv.DictReader(f)
            for row in reader:
                ref, sku = row['ref'], row['sku']
                qty = int(row['qty'])
                if row['eta']:
                    eta = datetime.strptime(row['eta'], '%Y-%m-%d').date()
                else:
                    eta = None
                self._batches[ref] = model.Batch(
                ref=ref, sku=sku, qty=qty, eta=eta
            )
        if self._allocations_path.exists() is False:
            return
        with self._allocations_path.open() as f:
            reader = csv.DictReader(f)
            for row in reader:
                batchref, orderid, sku = row['batchref'], row['orderid'], row['sku']
                qty = int(row['qty'])
                line = model.OrderLine(orderid, sku, qty)
                batch = self._batches[batchref]
                batch._allocations.add(line)

    def list(self):
        return list(self._batches.values())
```

And here's what a UoW for CSVs would look like:

A UoW for CSVs: commit = csv.writer (src/allocation/service_layer/csv_uow.py)

```
class CsvUnitOfWork(unit_of_work.AbstractUnitOfWork):

    def __init__(self, folder):
        self.batches = CsvRepository(folder)

    def commit(self):
        with self.batches._allocations_path.open('w') as f:
            writer = csv.writer(f)
            writer.writerow(['orderid', 'sku', 'qty', 'batchref'])
            for batch in self.batches.list():
                for line in batch._allocations:
                    writer.writerow(
                        [line.orderid, line.sku, line.qty, batch.reference]
                    )

    def rollback(self):
        pass
```

And once we have that, our CLI app for reading and writing batches and allocations to CSV is pared down to what it should be—a bit of code for reading order lines, and a bit of code that invokes our *existing* service layer:

Allocation with CSVs in nine lines (src/bin/allocate-from-csv)

```
def main(folder):
    orders_path = Path(folder) / 'orders.csv'
    uow = csv_uow.CsvUnitOfWork(folder)
    with orders_path.open() as f:
        reader = csv.DictReader(f)
        for row in reader:
            orderid, sku = row['orderid'], row['sku']
            qty = int(row['qty'])
            services.allocate(orderid, sku, qty, uow)
```

Ta-da! *Now are y'all impressed or what*?

Much love,

Bob and Harry

APPENDIX D

Repository and Unit of Work Patterns with Django

Suppose you wanted to use Django instead of SQLAlchemy and Flask. How might things look? The first thing is to choose where to install it. We put it in a separate package next to our main allocation code:

```
├── src
│   ├── allocation
│   │   ├── __init__.py
│   │   ├── adapters
│   │   │   ├── __init__.py
...
│   ├── djangoproject
│   │   ├── alloc
│   │   │   ├── __init__.py
│   │   │   ├── apps.py
│   │   │   ├── migrations
│   │   │   │   ├── 0001_initial.py
│   │   │   │   └── __init__.py
│   │   │   ├── models.py
│   │   │   └── views.py
│   │   ├── django_project
│   │   │   ├── __init__.py
│   │   │   ├── settings.py
│   │   │   ├── urls.py
│   │   │   └── wsgi.py
│   │   └── manage.py
│   └── setup.py
└── tests
    ├── conftest.py
    ├── e2e
    │   └── test_api.py
    ├── integration
    │   ├── test_repository.py
...
```

The code for this appendix is in the appendix_django branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout appendix_django
```

Repository Pattern with Django

We used a plug-in called `pytest-django` to help with test database management.

Rewriting the first repository test was a minimal change—just rewriting some raw SQL with a call to the Django ORM/QuerySet language:

First repository test adapted (tests/integration/test_repository.py)

```python
from djangoproject.alloc import models as django_models


@pytest.mark.django_db
def test_repository_can_save_a_batch():
    batch = model.Batch("batch1", "RUSTY-SOAPDISH", 100, eta=date(2011, 12, 25))

    repo = repository.DjangoRepository()
    repo.add(batch)

    [saved_batch] = django_models.Batch.objects.all()
    assert saved_batch.reference == batch.reference
    assert saved_batch.sku == batch.sku
    assert saved_batch.qty == batch._purchased_quantity
    assert saved_batch.eta == batch.eta
```

The second test is a bit more involved since it has allocations, but it is still made up of familiar-looking Django code:

Second repository test is more involved (tests/integration/test_repository.py)

```python
@pytest.mark.django_db
def test_repository_can_retrieve_a_batch_with_allocations():
    sku = "PONY-STATUE"
    d_line = django_models.OrderLine.objects.create(orderid="order1", sku=sku, qty=12)
    d_b1 = django_models.Batch.objects.create(
    reference="batch1", sku=sku, qty=100, eta=None
)
    d_b2 = django_models.Batch.objects.create(
    reference="batch2", sku=sku, qty=100, eta=None
)
    django_models.Allocation.objects.create(line=d_line, batch=d_batch1)
    repo = repository.DjangoRepository()
    retrieved = repo.get("batch1")

    expected = model.Batch("batch1", sku, 100, eta=None)
    assert retrieved == expected  # Batch.__eq__ only compares reference
    assert retrieved.sku == expected.sku
    assert retrieved._purchased_quantity == expected._purchased_quantity
    assert retrieved._allocations == {
        model.OrderLine("order1", sku, 12),
    }
```

Here's how the actual repository ends up looking:

A Django repository (src/allocation/adapters/repository.py)

```
class DjangoRepository(AbstractRepository):

    def add(self, batch):
        super().add(batch)
        self.update(batch)

    def update(self, batch):
        django_models.Batch.update_from_domain(batch)

    def _get(self, reference):
        return django_models.Batch.objects.filter(
            reference=reference
        ).first().to_domain()

    def list(self):
        return [b.to_domain() for b in django_models.Batch.objects.all()]
```

You can see that the implementation relies on the Django models having some custom methods for translating to and from our domain model.[1]

Custom Methods on Django ORM Classes to Translate to/from Our Domain Model

Those custom methods look something like this:

Django ORM with custom methods for domain model conversion (src/djangoproject/alloc/models.py)

```
from django.db import models
from allocation.domain import model as domain_model

class Batch(models.Model):
```

1 The DRY-Python project people have built a tool called mappers that looks like it might help minimize boilerplate for this sort of thing.

```
    reference = models.CharField(max_length=255)
    sku = models.CharField(max_length=255)
    qty = models.IntegerField()
    eta = models.DateField(blank=True, null=True)

    @staticmethod
    def update_from_domain(batch: domain_model.Batch):
        try:
            b = Batch.objects.get(reference=batch.reference)  ❶
        except Batch.DoesNotExist:
            b = Batch(reference=batch.reference)  ❶
        b.sku = batch.sku
        b.qty = batch._purchased_quantity
        b.eta = batch.eta  ❷
        b.save()
        b.allocation_set.set(
            Allocation.from_domain(l, b)  ❸
            for l in batch._allocations
        )

    def to_domain(self) -> domain_model.Batch:
        b = domain_model.Batch(
            ref=self.reference, sku=self.sku, qty=self.qty, eta=self.eta
        )
        b._allocations = set(
            a.line.to_domain()
            for a in self.allocation_set.all()
        )
        return b


class OrderLine(models.Model):
    #...
```

❶ For value objects, `objects.get_or_create` can work, but for entities, you probably need an explicit try-get/except to handle the upsert.[2]

❷ We've shown the most complex example here. If you do decide to do this, be aware that there will be boilerplate! Thankfully it's not very complex boilerplate.

❸ Relationships also need some careful, custom handling.

2 `@mr-bo-jangles` suggested you might be able to use `update_or_create`, but that's beyond our Django-fu.

As in Chapter 2, we use dependency inversion. The ORM (Django) depends on the model and not the other way around.

Unit of Work Pattern with Django

The tests don't change too much:

Adapted UoW tests (tests/integration/test_uow.py)

```
def insert_batch(ref, sku, qty, eta):  ❶
    django_models.Batch.objects.create(reference=ref, sku=sku, qty=qty, eta=eta)

def get_allocated_batch_ref(orderid, sku):  ❶
    return django_models.Allocation.objects.get(
        line__orderid=orderid, line__sku=sku
    ).batch.reference


@pytest.mark.django_db(transaction=True)
def test_uow_can_retrieve_a_batch_and_allocate_to_it():
    insert_batch('batch1', 'HIPSTER-WORKBENCH', 100, None)

    uow = unit_of_work.DjangoUnitOfWork()
    with uow:
        batch = uow.batches.get(reference='batch1')
        line = model.OrderLine('o1', 'HIPSTER-WORKBENCH', 10)
        batch.allocate(line)
        uow.commit()

    batchref = get_allocated_batch_ref('o1', 'HIPSTER-WORKBENCH')
    assert batchref == 'batch1'


@pytest.mark.django_db(transaction=True)  ❷
def test_rolls_back_uncommitted_work_by_default():
    ...

@pytest.mark.django_db(transaction=True)  ❷
def test_rolls_back_on_error():
    ...
```

❶ Because we had little helper functions in these tests, the actual main bodies of the tests are pretty much the same as they were with SQLAlchemy.

❷ The `pytest-django` `mark.django_db(transaction=True)` is required to test our custom transaction/rollback behaviors.

And the implementation is quite simple, although it took me a few tries to find which invocation of Django's transaction magic would work:

UoW adapted for Django (src/allocation/service_layer/unit_of_work.py)

```
class DjangoUnitOfWork(AbstractUnitOfWork):

    def __enter__(self):
        self.batches = repository.DjangoRepository()
        transaction.set_autocommit(False)  ❶
        return super().__enter__()

    def __exit__(self, *args):
        super().__exit__(*args)
        transaction.set_autocommit(True)

    def commit(self):
        for batch in self.batches.seen:  ❸
            self.batches.update(batch)  ❸
        transaction.commit()  ❷

    def rollback(self):
        transaction.rollback()  ❷
```

❶ `set_autocommit(False)` was the best way to tell Django to stop automatically committing each ORM operation immediately, and to begin a transaction.

❷ Then we use the explicit rollback and commits.

❸ One difficulty: because, unlike with SQLAlchemy, we're not instrumenting the domain model instances themselves, the `commit()` command needs to explicitly go through all the objects that have been touched by every repository and manually update them back to the ORM.

API: Django Views Are Adapters

The Django *views.py* file ends up being almost identical to the old *flask_app.py*, because our architecture means it's a very thin wrapper around our service layer (which didn't change at all, by the way):

Flask app → Django views (src/djangoproject/alloc/views.py)

```
os.environ['DJANGO_SETTINGS_MODULE'] = 'djangoproject.django_project.settings'
django.setup()

@csrf_exempt
def add_batch(request):
    data = json.loads(request.body)
    eta = data['eta']
    if eta is not None:
        eta = datetime.fromisoformat(eta).date()
    services.add_batch(
        data['ref'], data['sku'], data['qty'], eta,
        unit_of_work.DjangoUnitOfWork(),
    )
    return HttpResponse('OK', status=201)

@csrf_exempt
def allocate(request):
    data = json.loads(request.body)
    try:
        batchref = services.allocate(
            data['orderid'],
            data['sku'],
            data['qty'],
            unit_of_work.DjangoUnitOfWork(),
        )
    except (model.OutOfStock, services.InvalidSku) as e:
        return JsonResponse({'message': str(e)}, status=400)

    return JsonResponse({'batchref': batchref}, status=201)
```

Why Was This All So Hard?

OK, it works, but it does feel like more effort than Flask/SQLAlchemy. Why is that?

The main reason at a low level is because Django's ORM doesn't work in the same way. We don't have an equivalent of the SQLAlchemy classical mapper, so our `Active Record` and our domain model can't be the same object. Instead we have to build a manual translation layer behind the repository. That's more work (although once it's done, the ongoing maintenance burden shouldn't be too high).

Because Django is so tightly coupled to the database, you have to use helpers like `pytest-django` and think carefully about test databases, right from the very first line of code, in a way that we didn't have to when we started out with our pure domain model.

But at a higher level, the entire reason that Django is so great is that it's designed around the sweet spot of making it easy to build CRUD apps with minimal boilerplate. But the entire thrust of our book is about what to do when your app is no longer a simple CRUD app.

At that point, Django starts hindering more than it helps. Things like the Django admin, which are so awesome when you start out, become actively dangerous if the whole point of your app is to build a complex set of rules and modeling around the workflow of state changes. The Django admin bypasses all of that.

What to Do If You Already Have Django

So what should you do if you want to apply some of the patterns in this book to a Django app? We'd say the following:

- The Repository and Unit of Work patterns are going to be quite a lot of work. The main thing they will buy you in the short term is faster unit tests, so evaluate whether that benefit feels worth it in your case. In the longer term, they decouple your app from Django and the database, so if you anticipate wanting to migrate away from either of those, Repository and UoW are a good idea.
- The Service Layer pattern might be of interest if you're seeing a lot of duplication in your *views.py*. It can be a good way of thinking about your use cases separately from your web endpoints.
- You can still theoretically do DDD and domain modeling with Django models, tightly coupled as they are to the database; you may be slowed by migrations, but it shouldn't be fatal. So as long as your app is not too complex and your tests not too slow, you may be able to get something out of the *fat models* approach: push as much logic down to your models as possible, and apply patterns like Entity, Value Object, and Aggregate. However, see the following caveat.

With that said, word in the Django community is that people find that the fat models approach runs into scalability problems of its own, particularly around managing interdependencies between apps. In those cases, there's a lot to be said for extracting out a business logic or domain layer to sit between your views and forms and your *models.py*, which you can then keep as minimal as possible.

Steps Along the Way

Suppose you're working on a Django project that you're not sure is going to get complex enough to warrant the patterns we recommend, but you still want to put a few steps in place to make your life easier, both in the medium term and if you want to migrate to some of our patterns later. Consider the following:

- One piece of advice we've heard is to put a *logic.py* into every Django app from day one. This gives you a place to put business logic, and to keep your forms, views, and models free of business logic. It can become a stepping-stone for moving to a fully decoupled domain model and/or service layer later.
- A business-logic layer might start out working with Django model objects and only later become fully decoupled from the framework and work on plain Python data structures.

- For the read side, you can get some of the benefits of CQRS by putting reads into one place, avoiding ORM calls sprinkled all over the place.
- When separating out modules for reads and modules for domain logic, it may be worth decoupling yourself from the Django apps hierarchy. Business concerns will cut across them.

We'd like to give a shout-out to David Seddon and Ashia Zawaduk for talking through some of the ideas in this appendix. They did their best to stop us from saying anything really stupid about a topic we don't really have enough personal experience of, but they may have failed.

For more thoughts and actual lived experience dealing with existing applications, refer to the epilogue.

APPENDIX E

Validation

Whenever we're teaching and talking about these techniques, one question that comes up over and over is "Where should I do validation? Does that belong with my business logic in the domain model, or is that an infrastructural concern?"

As with any architectural question, the answer is: it depends!

The most important consideration is that we want to keep our code well separated so that each part of the system is simple. We don't want to clutter our code with irrelevant detail.

What Is Validation, Anyway?

When people use the word *validation*, they usually mean a process whereby they test the inputs of an operation to make sure that they match certain criteria. Inputs that match the criteria are considered *valid*, and inputs that don't are *invalid*.

If the input is invalid, the operation can't continue but should exit with some kind of error. In other words, validation is about creating *preconditions*. We find it useful to separate our preconditions into three subtypes: syntax, semantics, and pragmatics.

Validating Syntax

In linguistics, the *syntax* of a language is the set of rules that govern the structure of grammatical sentences. For example, in English, the sentence "Allocate three units of `TASTELESS-LAMP` to order twenty-seven" is grammatically sound, while the phrase "hat hat hat hat hat hat wibble" is not. We can describe grammatically correct sentences as *well formed*.

How does this map to our application? Here are some examples of syntactic rules:

- An `Allocate` command must have an order ID, a SKU, and a quantity.
- A quantity is a positive integer.
- A SKU is a string.

These are rules about the shape and structure of incoming data. An `Allocate` command without a SKU or an order ID isn't a valid message. It's the equivalent of the phrase "Allocate three to."

We tend to validate these rules at the edge of the system. Our rule of thumb is that a message handler should always receive only a message that is well-formed and contains all required information.

One option is to put your validation logic on the message type itself:

Validation on the message class (src/allocation/commands.py)

```
from schema import And, Schema, Use


@dataclass
class Allocate(Command):

    _schema = Schema({  ❶
        'orderid': int,
         sku: str,
         qty: And(Use(int), lambda n: n > 0)
     }, ignore_extra_keys=True)

    orderid: str
    sku: str
    qty: int

    @classmethod
    def from_json(cls, data):  ❷
       data = json.loads(data)
       return cls(**_schema.validate(data))
```

❶ The `schema` library lets us describe the structure and validation of our messages in a nice declarative way.

❷ The `from_json` method reads a string as JSON and turns it into our message type.

This can get repetitive, though, since we need to specify our fields twice, so we might want to introduce a helper library that can unify the validation and declaration of our message types:

A command factory with schema (src/allocation/commands.py)

```
def command(name, **fields):  ❶
    schema = Schema(And(Use(json.loads), fields), ignore_extra_keys=True)  ❷
    cls = make_dataclass(name, fields.keys())
    cls.from_json = lambda s: cls(**schema.validate(s))  ❸
    return cls

def greater_than_zero(x):
    return x > 0

quantity = And(Use(int), greater_than_zero)  ❹

Allocate = command(  ❺
    orderid=int,
    sku=str,
    qty=quantity
)

AddStock = command(
    sku=str,
    qty=quantity
```

❶ The `command` function takes a message name, plus kwargs for the fields of the message payload, where the name of the kwarg is the name of the field and the value is the parser.

❷ We use the `make_dataclass` function from the dataclass module to dynamically create our message type.

❸ We patch the `from_json` method onto our dynamic dataclass.

❹ We can create reusable parsers for quantity, SKU, and so on to keep things DRY.

❺ Declaring a message type becomes a one-liner.

This comes at the expense of losing the types on your dataclass, so bear that trade-off in mind.

Postel's Law and the Tolerant Reader Pattern

Postel's law, or the *robustness principle*, tells us, "Be liberal in what you accept, and conservative in what you emit." We think this applies particularly well in the context of integration with our other systems. The idea here is that we should be strict whenever we're sending messages to other systems, but as lenient as possible when we're receiving messages from others.

For example, our system *could* validate the format of a SKU. We've been using made-up SKUs like `UNFORGIVING-CUSHION` and `MISBEGOTTEN-POUFFE`. These follow a simple pattern: two words, separated by dashes, where the second word is the type of product and the first word is an adjective.

Developers *love* to validate this kind of thing in their messages, and reject anything that looks like an invalid SKU. This causes horrible problems down the line when some anarchist releases a product named `COMFY-CHAISE-LONGUE` or when a snafu at the supplier results in a shipment of `CHEAP-CARPET-2`.

Really, as the allocation system, it's *none of our business* what the format of a SKU might be. All we need is an identifier, so we can simply describe it as a string. This means that the procurement system can change the format whenever they like, and we won't care.

This same principle applies to order numbers, customer phone numbers, and much more. For the most part, we can ignore the internal structure of strings.

Similarly, developers *love* to validate incoming messages with tools like JSON Schema, or to build libraries that validate incoming messages and share them among systems. This likewise fails the robustness test.

Let's imagine, for example, that the procurement system adds new fields to the `ChangeBatchQuantity` message that record the reason for the change and the email of the user responsible for the change.

Since these fields don't matter to the allocation service, we should simply ignore them. We can do that in the `schema` library by passing the keyword arg `ignore_extra_keys=True`.

This pattern, whereby we extract only the fields we care about and do minimal validation of them, is the Tolerant Reader pattern.

Validate as little as possible. Read only the fields you need, and don't overspecify their contents. This will help your system stay robust when other systems change over time. Resist the temptation to share message definitions between systems: instead, make it easy to define the data you depend on. For more info, see Martin Fowler's article on the Tolerant Reader pattern.

Is Postel Always Right?

Mentioning Postel can be quite triggering to some people. They will tell you that Postel is the precise reason that everything on the internet is broken and we can't have nice things. Ask Hynek about SSLv3 one day.

We like the Tolerant Reader approach in the particular context of event-based integration between services that we control, because it allows for independent evolution of those services.

If you're in charge of an API that's open to the public on the big bad internet, there might be good reasons to be more conservative about what inputs you allow.

Validating at the Edge

Earlier, we said that we want to avoid cluttering our code with irrelevant details. In particular, we don't want to code defensively inside our domain model. Instead, we want to make sure that requests are known to be valid before our domain model or use-case handlers see them. This helps our code stay clean and maintainable over the long term. We sometimes refer to this as *validating at the edge of the system*.

In addition to keeping your code clean and free of endless checks and asserts, bear in mind that invalid data wandering through your system is a time bomb; the deeper it gets, the more damage it can do, and the fewer tools you have to respond to it.

Back in Chapter 8, we said that the message bus was a great place to put cross-cutting concerns, and validation is a perfect example of that. Here's how we might change our bus to perform validation for us:

Validation

```
class MessageBus:

    def handle_message(self, name: str, body: str):
        try:
            message_type = next(mt for mt in EVENT_HANDLERS if mt.__name__ == name)
            message = message_type.from_json(body)
            self.handle([message])
        except StopIteration:
            raise KeyError(f"Unknown message name {name}")
        except ValidationError as e:
            logging.error(
                f'invalid message of type {name}\n'
                f'{body}\n'
                f'{e}'
            )
            raise e
```

Here's how we might use that method from our Flask API endpoint:

API bubbles up validation errors (src/allocation/flask_app.py)

```
@app.route("/change_quantity", methods=['POST'])
def change_batch_quantity():
    try:
        bus.handle_message('ChangeBatchQuantity', request.body)
    except ValidationError as e:
        return bad_request(e)
    except exceptions.InvalidSku as e:
        return jsonify({'message': str(e)}), 400

def bad_request(e: ValidationError):
    return e.code, 400
```

And here's how we might plug it in to our asynchronous message processor:

Validation errors when handling Redis messages (src/allocation/redis_pubsub.py)

```
def handle_change_batch_quantity(m, bus: messagebus.MessageBus):
    try:
        bus.handle_message('ChangeBatchQuantity', m)
    except ValidationError:
       print('Skipping invalid message')
    except exceptions.InvalidSku as e:
        print(f'Unable to change stock for missing sku {e}')
```

Notice that our entrypoints are solely concerned with how to get a message from the outside world and how to report success or failure. Our message bus takes care of validating our requests and routing them to the correct handler, and our handlers are exclusively focused on the logic of our use case.

When you receive an invalid message, there's usually little you can do but log the error and continue. At MADE we use metrics to count the number of messages a system receives, and how many of those are successfully processed, skipped, or invalid. Our monitoring tools will alert us if we see spikes in the numbers of bad messages.

Validating Semantics

While syntax is concerned with the structure of messages, *semantics* is the study of *meaning* in messages. The sentence "Undo no dogs from ellipsis four" is syntactically valid and has the same structure as the sentence "Allocate one teapot to order five,"" but it is meaningless.

We can read this JSON blob as an `Allocate` command but can't successfully execute it, because it's *nonsense*:

A meaningless message

```
{
  "orderid": "superman",
  "sku": "zygote",
  "qty": -1
}
```

We tend to validate semantic concerns at the message-handler layer with a kind of contract-based programming:

Preconditions (src/allocation/ensure.py)

```
"""
This module contains preconditions that we apply to our handlers.
"""

class MessageUnprocessable(Exception):  ❶

    def __init__(self, message):
        self.message = message

class ProductNotFound(MessageUnprocessable):  ❷
   """"
   This exception is raised when we try to perform an action on a product
   that doesn't exist in our database.
   """"

    def __init__(self, message):
        super().__init__(message)
        self.sku = message.sku

def product_exists(event, uow):  ❸
    product = uow.products.get(event.sku)
    if product is None:
        raise ProductNotFound(event)
```

❶ We use a common base class for errors that mean a message is invalid.

❷ Using a specific error type for this problem makes it easier to report on and handle the error. For example, it's easy to map `ProductNotFound` to a 404 in Flask.

❸ `product_exists` is a precondition. If the condition is `False`, we raise an error.

This keeps the main flow of our logic in the service layer clean and declarative:

Ensure calls in services (src/allocation/services.py)

```
# services.py

from allocation import ensure
```

```
def allocate(event, uow):
    line = mode.OrderLine(event.orderid, event.sku, event.qty)
    with uow:
        ensure.product_exists(uow, event)

        product = uow.products.get(line.sku)
        product.allocate(line)
        uow.commit()
```

We can extend this technique to make sure that we apply messages idempotently. For example, we want to make sure that we don't insert a batch of stock more than once.

If we get asked to create a batch that already exists, we'll log a warning and continue to the next message:

Raise SkipMessage exception for ignorable events (src/allocation/services.py)

```
class SkipMessage (Exception):
    """"
    This exception is raised when a message can't be processed, but there's no
    incorrect behavior. For example, we might receive the same message multiple
    times, or we might receive a message that is now out of date.
    """"

    def __init__(self, reason):
        self.reason = reason

def batch_is_new(self, event, uow):
    batch = uow.batches.get(event.batchid)
    if batch is not None:
        raise SkipMessage(f"Batch with id {event.batchid} already exists")
```

Introducing a `SkipMessage` exception lets us handle these cases in a generic way in our message bus:

The bus now knows how to skip (src/allocation/messagebus.py)

```
class MessageBus:

    def handle_message(self, message):
        try:
           ...
       except SkipMessage as e:
           logging.warn(f"Skipping message {message.id} because {e.reason}")
```

There are a couple of pitfalls to be aware of here. First, we need to be sure that we're using the same UoW that we use for the main logic of our use case. Otherwise, we open ourselves to irritating concurrency bugs.

Second, we should try to avoid putting *all* our business logic into these precondition checks. As a rule of thumb, if a rule *can* be tested inside our domain model, then it *should* be tested in the domain model.

Validating Pragmatics

Pragmatics is the study of how we understand language in context. After we have parsed a message and grasped its meaning, we still need to process it in context. For example, if you get a comment on a pull request saying, “I think this is very brave,” it may mean that the reviewer admires your courage—unless they’re British, in which case, they’re trying to tell you that what you’re doing is insanely risky, and only a fool would attempt it. Context is everything.

Validation Recap

Validation means different things to different people
: When talking about validation, make sure you’re clear about what you’re validating. We find it useful to think about syntax, semantics, and pragmatics: the structure of messages, the meaningfulness of messages, and the business logic governing our response to messages.

Validate at the edge when possible
: Validating required fields and the permissible ranges of numbers is *boring*, and we want to keep it out of our nice clean codebase. Handlers should always receive only valid messages.

Only validate what you require
: Use the Tolerant Reader pattern: read only the fields your application needs and don’t overspecify their internal structure. Treating fields as opaque strings buys you a lot of flexibility.

Spend time writing helpers for validation
: Having a nice declarative way to validate incoming messages and apply preconditions to your handlers will make your codebase much cleaner. It’s worth investing time to make boring code easy to maintain.

Locate each of the three types of validation in the right place
: Validating syntax can happen on message classes, validating semantics can happen in the service layer or on the message bus, and validating pragmatics belongs in the domain model.

Once you've validated the syntax and semantics of your commands at the edges of your system, the domain is the place for the rest of your validation. Validation of pragmatics is often a core part of your business rules.

In software terms, the pragmatics of an operation are usually managed by the domain model. When we receive a message like "allocate three million units of `SCARCE-CLOCK` to order 76543," the message is *syntactically* valid and *semantically* valid, but we're unable to comply because we don't have the stock available.

Index

About the Authors

Harry Percival spent a few years being deeply unhappy as a management consultant. Soon he rediscovered his true geek nature and was lucky enough to fall in with a bunch of XP fanatics, working on pioneering the sadly defunct Resolver One spreadsheet. He worked at PythonAnywhere LLP, spreading the gospel of TDD worldwide at talks, workshops, and conferences. He is now with MADE.com.

Bob Gregory is a UK-based software architect with MADE.com. He has been building event-driven systems with domain-driven design for more than a decade.

Colophon

The animal on the cover of *Architecture Patterns with Python* is a Burmese python (*Python bivitattus*). As you might expect, the Burmese python is native to Southeast Asia. Today it lives in jungles and marshes in South Asia, Myanmar, China, and Indonesia; it's also invasive in Florida's Everglades.

Burmese pythons are one of the world's largest species of snakes. These nocturnal, carnivorous constrictors can grow to 23 feet and 200 pounds. Females are larger than males. They can lay up to a hundred eggs in one clutch. In the wild, Burmese pythons live an average of 20 to 25 years.

The markings on a Burmese python begin with an arrow-shaped spot of light brown on top of the head and continue along the body in rectangles that stand out against its otherwise tan scales. Before they reach their full size, which takes two to three years, Burmese pythons live in trees hunting small mammals and birds. They also swim for long stretches of time—going up to 30 minutes without air.

Because of habitat destruction, the Burmese python has a conservation status of Vulnerable. Many of the animals on O'Reilly's covers are endangered; all of them are important to the world.

The color illustration is by Jose Marzan, based on a black-and-white engraving from *Encyclopedie D'Histoire Naturelle*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.