O'REILLY®

Enterprise Architecture Patterns with Python

How to Apply DDD, Ports and Adapters, and Enterprise Architecture Design Patterns in a Pythonic Way

Early Release

RAW & UNEDITED

Harry J.W. Percival
& Bob Gregory

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, he found himself asking a bunch of questions about architecture — 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 we minimise the number of integration and end-to-end tests we 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 e-commerce company who sell furniture online - where 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 operate a global supply chain of freight partners and manufacturers. To try and keep costs low, we try to optimise 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 3 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 that represents the kind 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 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 have 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:[1] "there should be one—and preferably only one—obvious way to do it." 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're going to discuss in this book are new, but they are mostly new to the Python world. And this book won't be a replacement for the classics in the field like Eric Evans' *Domain-Driven Design* or Martin Fowler's *Patterns of Enterprise Application Architecture* (both of 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), it can make them quite trying. There's a reason the latest edition of that other classic text, Refactoring is in JavaScript.

So we hope this book will make for a lightweight introduction to some of the key architectural patterns that support domain-driven design (DDD) and event-driven microservices, that it will serve as a reference for implementing them in a Pythonic way, and that it will serve as a first step for those who want to do further research in this field.

Who Should Read This Book

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

We assume you've been close to some reasonably complex Python applications.

We assume you've seen some of the pain that comes with trying to manage that complexity.

We do *not* assume that you already 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 test-driven development (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 (version 3) frameworks and technologies, like 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 where specific technology choices become minor implementation details.

A Brief Overview of What You'll Learn

Part 1: Dependency Inversion and Domain Modelling

Chapter 1: Domain Modelling and DDD

> 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 it, without getting tangled up with infrastructure concerns, with our web frameworks, or whatever else? In this chapter we give a broad overview of *domain modelling* and DDD, and show how to get started with a model that has no external dependencies, and fast unit tests.

Chapter 2, 4 and 5: Repository, Service Layer and Unit of Work Patterns

> 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 very thin entrypoints to our system, be it a Flask API or a CLI.

Chapter 3: An Aside on Coupling and Abstractions

> After presenting the first abstraction (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.

Chapter 6: Aggregate Pattern

> A brief return to the world of DDD, where we discuss how to choose the right *Aggregate*, and how this choice relates to questions of data integrity

Part 2: Event-Driven Architecture

Chapters 7, 8 and 9: Event-Driven Architecture

> We introduce three more mutually-reinforcing patterns, starting with the concept of *Domain Events*, 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 add the distinction between *Commands* and *Events*. Our application is now fundamentally a message-processing system.

Chapter 10: CQRS

> An example of *command-query responsibility segregation*, with and without events.

Chapter 11 Dependency Injection

> We tidy up our explicit and implicit dependencies, and implement a very simple dependency injection framework.

Epilogue (Chapter 12): How Do I Get There From Here?

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 attempt to provide a few pointers in this last chapter 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 recreate 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), which we build up as we go, and the narrative of the book is as if you're pairing with us as we go, 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 actually 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 them here: *https://github.com/python-leap/code/branches/all*

Here's three different ways you might code along with the book:

- Start your own repo and try and build up the app as we do,

following the examples from listings in the book, and occasionally looking to our repo for hints.

- Try to apply these 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 lower effort, in each chapter we'll 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.

If you want to go all the way to town, why not try and build up the code as you read along? Particularly if you're intending to apply some of these patterns in your own projects, then working through a simple example can really help you to get some safe practice.

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.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

O’Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning

Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

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 *http://www.oreilly.com/catalog/<catalog page>*.

To comment or ask technical questions about this book, send email to

bookquestions@oreilly.com.

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

```
python -c "import this"
```

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, and order by complexity.

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 utils modules. We find that our sensibly layered architecture has collapsed into itself like an over-soggy trifle. Chaotic software systems are characterised by a sameness of function: API handlers that have domain knowledge, and send emails and perform logging; “business logic” classes that perform no calculations but do perform IO; 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.

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

The term *encapsulation* covers two closely related ideas: simplifying behavior, and hiding data. In this book, when we say "encapsulation" 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.

Take a look at the following two snippets of Python code, Example P-1 and Example P-2:

Example P-1. Do a search with urllib

```
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'])
```

Example P-2. Do a search with requests

```
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 of these 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 use an even higher-level abstraction to make it explicit:

Example P-3. Do a search with the duckduckgo module

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

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

> **NOTE**
>
> This approach is inspired by the OO practice of responsibility-driven design. which would use 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. If you've come across CRC cards, they're driving at the same thing.

Layering

Encapsulation helps 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. 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.

For example most people are familiar with the three layered architecture (see Figure P-1):

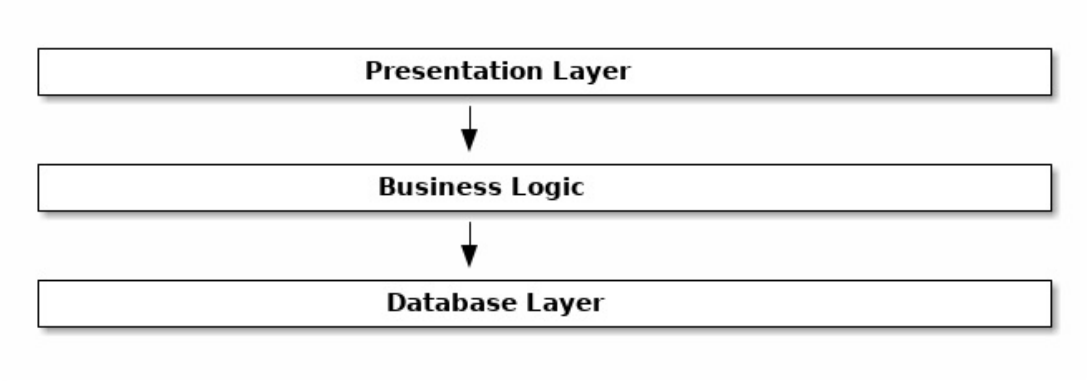
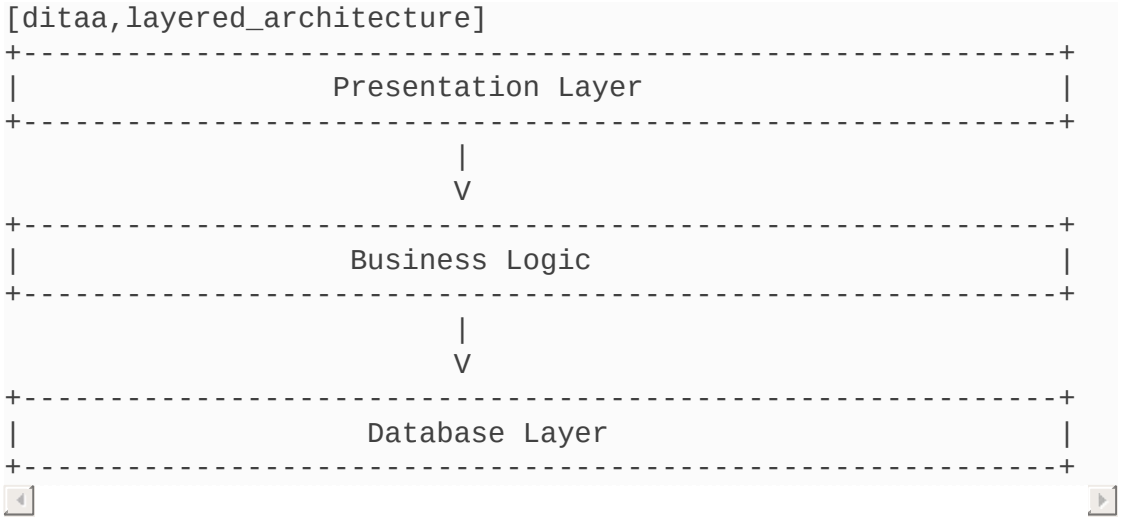


Figure P-1. Layered architecture



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, or 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 data 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 already, because it's the D in the SOLID[1] mnemonic. Formally, the DIP says:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. 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 organisation really cares about. Perhaps you work for a pharmaceutical company, and your high-level modules deal with patients 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 organisation doesn't care about. It's unlikely that your HR department gets excited about file systems, or network sockets. It's not often that you can discuss SMTP, or HTTP, or AMQP with your finance team. For our non-technical 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 more 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 some behavior, in the way that our duckduckgo module encapsulated a search engine's API. In a traditional-OO language you might use an abstract base class or an interface to define an abstraction. In Python you can (and we sometimes do) use ABCs, but you can also 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.

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

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

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 need. Low-level modules (details) are often, in practice, harder to change: think about refactoring to change a function name vs defining, testing and deploying a database migration to change a column name. We don't want business logic changes to be slowed 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 in 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? We'll come to that in Chapter 4, but before we can turn our three-layered architecture inside out, we need to talk more about that middle layer, the business logic.

One of the most common reasons that our designs go wrong is that business logic becomes spread out throughout the layers of our application, hard to identify, understand and change.

The next few chapters discuss some application architecture patterns that allow us to keep our business layer, the domain model, free of dependencies and easy to maintain.

1 Uncle Bob's five principles of object-oriented design: Single responsibility, Open for extension but closed for modification, Liskov substitution, Interface segregation, and Dependency Inversion. There's a good overview, with examples, at *https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design*

Chapter 1. Domain Modelling

In this chapter we'll look into how we can model business processes with code, in a way that's highly compatible with TDD. We'll discuss *why* domain modelling matters, and we'll look at a few key patterns for modelling domains: Entities, Value Objects, and Domain Services.

What is a Domain Model?

In the [Link to Come], 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.* We currently work for an online retailer of furniture. Depending on which system I'm 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 some 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 how 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.

THIS IS NOT A DDD BOOK. YOU SHOULD READ A DDD BOOK.

Domain-driven design, or DDD, is where the concept of domain modelling was popularized,[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, like Entity, Aggregate and Value Objects (see Chapter 5), and Repository pattern (in the next chapter) all 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 some problem. If we get that model right, then our software delivers value and makes new things possible.

If we get it 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 the processes, tools and techniques for developing a domain model. We hope to give you a taste for 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, 2003)
- Or, some people prefer the "red book", *Implementing Domain-Driven Design*, by Vaughn Vernon (Addison-Wesley, 2013).

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 between 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 doo-hickey 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.

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.

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. There's a system in the middle that needs to coordinate the process by allocating stock to a customer's orders; see Figure 1-1.

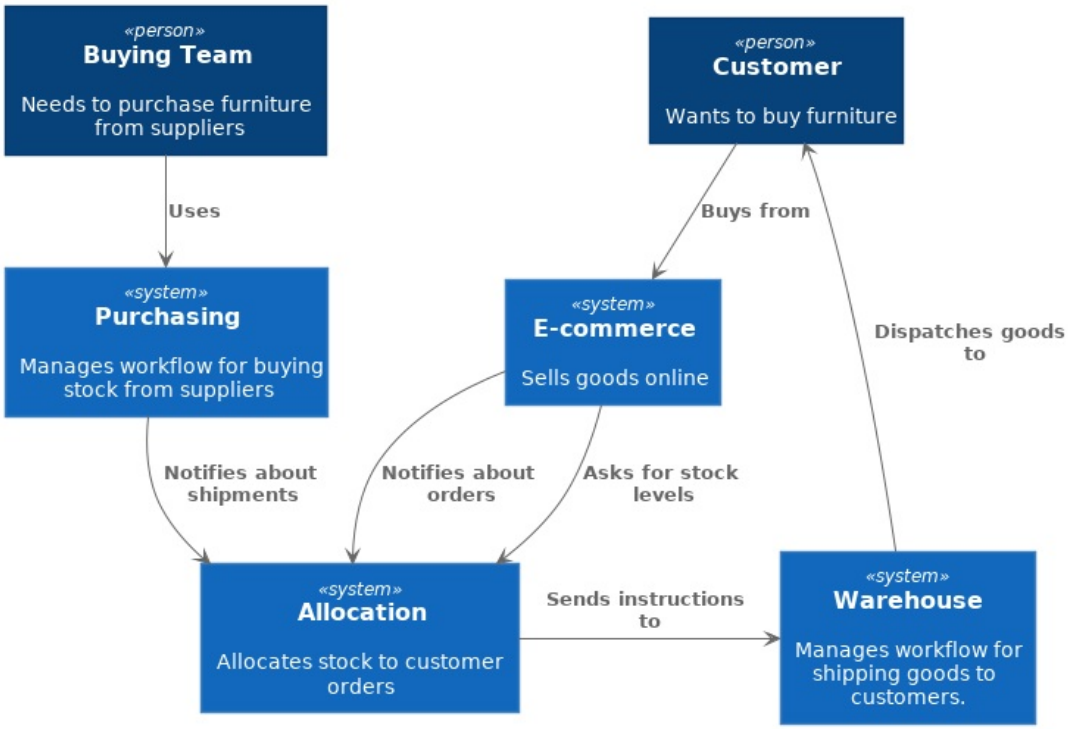


Figure 1-1. Context diagram for the allocation service

```
[plantuml, allocation_context_diagram]
@startuml Allocation Context Diagram
!includeurl https://raw.githubusercontent.com/RicardoNiepel/C4-
PlantUML/master/C4.puml
!includeurl https://raw.githubusercontent.com/RicardoNiepel/C4-
PlantUML/master/C4_Context.puml

System(systema, "Allocation", "Allocates stock to customer
orders")

Person(customer, "Customer", "Wants to buy furniture")
Person(buyer, "Buying Team", "Needs to purchase furniture from
suppliers")

System(procurement, "Purchasing", "Manages workflow for buying
stock from suppliers")
System(ecom, "E-commerce", "Sells goods online")
System(warehouse, "Warehouse", "Manages workflow for shipping
goods to customers.")
```

```
Rel(buyer, procurement, "Uses")
Rel(procurement, systema, "Notifies about shipments")
Rel(customer, ecom, "Buys from")
Rel(ecom, systema, "Asks for stock levels")
Rel(ecom, systema, "Notifies about orders")
Rel_R(systema, warehouse, "Sends instructions to")
Rel_U(warehouse, customer, "Dispatches goods to")

@enduml
```

For the purposes of this book, we're imagining a situation where 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.

The innovation is: if we have a system that can keep track of all our shipments and when they're due to arrive, then 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 modelling.

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 we 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.

Here are some notes we might have taken while having a conversation with our domain experts about 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*.

  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 which they call 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 1 unit of stock to a batch, the *available quantity* is reduced.

  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.

  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.

  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.

EXERCISE FOR THE READER

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

We've got some placeholder unit tests here, but you could just start from scratch, or combine/rewrite these however you like:

https://github.com/python-leap/code/tree/chapter_01_domain_model_exercise

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.

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

Example 1-1. A first test for allocation (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
```

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 non-technical co-workers, and they would agree that this correctly describes the behavior of the system.

And here is a domain model that meets our requirements:

Example 1-2. First cut of a domain model for batches (model.py)

```
@dataclass(frozen=True)  ❶
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
```

1. `OrderLine` is an immutable dataclass[2] with no behavior.
2. 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 modelling our domain precisely will pay off.

Let's write some new failing tests:

Example 1-3. 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`.

Example 1-4. 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:

Example 1-5. 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:

Example 1-6. A decent first cut of the domain model (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 1-2 shows the model in diagram form.

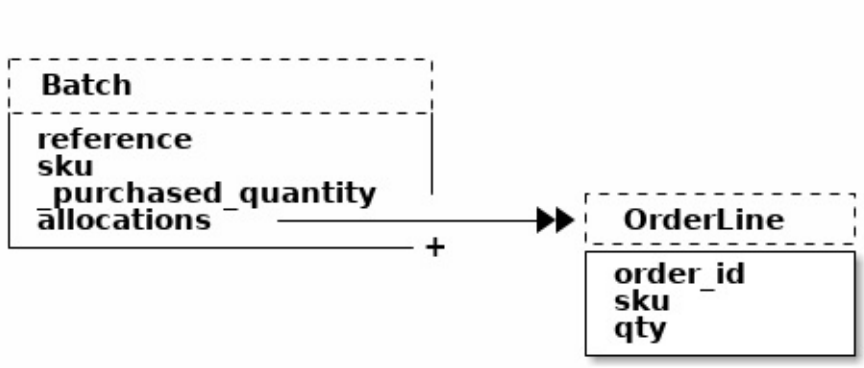


Figure 1-2. Our Model

```
[ditaa, model_diagram]
+=====================+
| Batch               |
+---------------------+
| reference           |
| sku                 |
| _purchased_quantity |       +=============+
| allocations ------------->>| OrderLine   |
+---------------------+       +-------------+
                              | order_id    |
                              | sku         |
                              | qty         |
                              +-------------+
```

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 - allocated quantity. Using a set here makes it simple for us to handle the last test, because items in a set are unique.

Example 1-7. 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
```

Perhaps you think this model is too trivial to bother with object-orientation, but throughout this book, we're going to extend our simple domain model, and plug it into the real world of APIs and databases and spreadsheets, and 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 as far as wrapping primitive types using `typing.NewType`:

Example 1-8. 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.available_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[3] is a matter of debate.

Dataclasses Are Great for Value Objects

We've used the *line* liberally in the previous code listings, but what is a line? In the 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:

Example 1-9. Order info as YAML

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

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 some data but no identity, we often choose to represent it using a Value Object. A Value Object is any domain object that is uniquely identified by the data it holds; we usually make them immutable.

Example 1-10. OrderLine is a Value Object.

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

Introduced in Python 3.7, `Dataclasses` are a neat way to represent value objects; if you're on Python 2, you could use `namedtuples` instead. Either technique will give you *value equality* which is the fancy way of saying "two lines with the same orderid, sku and qty are equal."

Example 1-11. 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 intuitions 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 name 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.

Example 1-12. Maths 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 orderid, 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 only identified 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":

Example 1-13. 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 recognise them as the same individual. That's because humans, unlike names, have a persistent *identity*.

Example 1-14. 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 recognisably 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:

Example 1-15. 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.

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; more info in the Python docs.

For Value Objects, the hash should be based on all the value attributes. For entities, the hash should either be `None`, or it should be based on the attribute(s), like `.reference`, that define identity over time.

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 Services[4] operations that don't have a natural home in an Entity or Value Object. 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 multi-paradigm language and just make it a function.

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

Example 1-16. 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
    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:

Example 1-17. 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()` above, 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:

Example 1-18. 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

One final concept to cover, which is the idea that exceptions can be used to express domain concepts too. In our conversations with the 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 using a *domain exception*:

Example 1-19. Testing out of stock exception (test_allocate.py)

```
def test_raises_out_of_stock_exception_if_cannot_allocate():
    batch = Batch('batch1', 'HEAVY-SPOON', 100, eta=today)
```

```
    different_sku_line = OrderLine('oref', 'SMALL-FORK', 10)

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

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 like we do our Entities, Value Objects and Services.

Example 1-20. 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}')
```

That'll probably do for now! We have a Domain Service which we can use for our first use case. But first we'll need a database.

DOMAIN MODELLING WRAP-UP

Domain modelling

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 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 multi-paradigm language, so let the "verbs" in your code be functions. Classes called "Manager" or "Builder" or "Factory" are a code smell.

This is the time to apply your best OO design principles

revise SOLID. has-a vs is-a. composition over inheritance. etc etc.

You'll also want to think about consistency boundaries and Aggregates

But that's a topic for Chapter 6.

---

1 DDD did not originate domain modelling. Eric Evans refers to *Object Design* from Rebecca Whirfs-Brock and Alan McKean, 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.

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

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

4 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.

Chapter 2. Repository Pattern

In this chapter, we'll start to make good on our promise to apply the dependency inversion principle as a way of decoupling our core logic from infrastructural concerns.

We'll introduce the *Repository*, a simplifying abstraction over data storage, allowing us to decouple our model layer from the data layer. We'll see 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` class that sits between `SqlAlchemy` (our ORM) and our Domain Model's `Batch` classes.

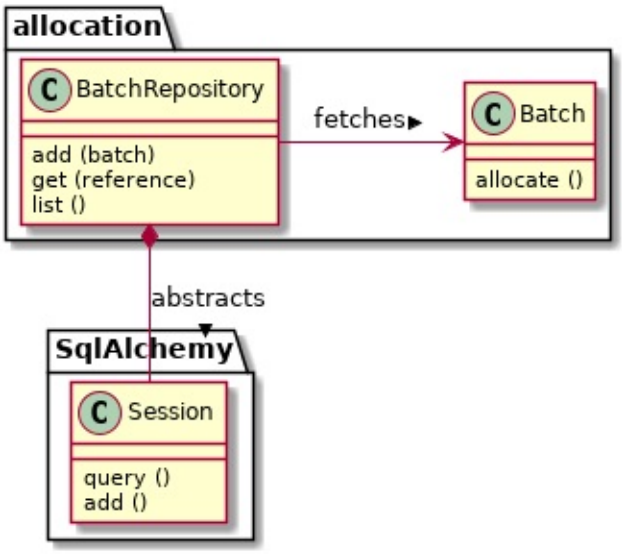


Figure 2-1. Batch, Repository and SqlAlchemy

```
[plantuml, chapter_02_class_diagram]
```

```
@startuml

package allocation {

  class BatchRepository {
    add (batch)
    get (reference)
    list ()
  }

  class Batch {
    allocate ()
  }


}

package SqlAlchemy {
  class Session {
    query ()
    add ()
  }

}

  BatchRepository *-- Session : abstracts >
  BatchRepository -> Batch : fetches >

@enduml
```

Persisting our Domain Model

In the previous chapter 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 we'll need to 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 idealised domain model to external state.

We expect to be working in an agile manner, so our priority is to get to an MVP 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 Example 2-1 [1].

Example 2-1. 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'll need a way to retrieve batch info from the DB and instantiate our domain model objects from it, and we'll also need a way of saving them back to the database.

Applying the Dependency Inversion Principle to the Database

As mentioned in Introduction: Why Do Our Designs Go Wrong?, the "layered architecture" is a common approach to structuring a system that has a UI, some logic, and a database (see Figure 2-2).

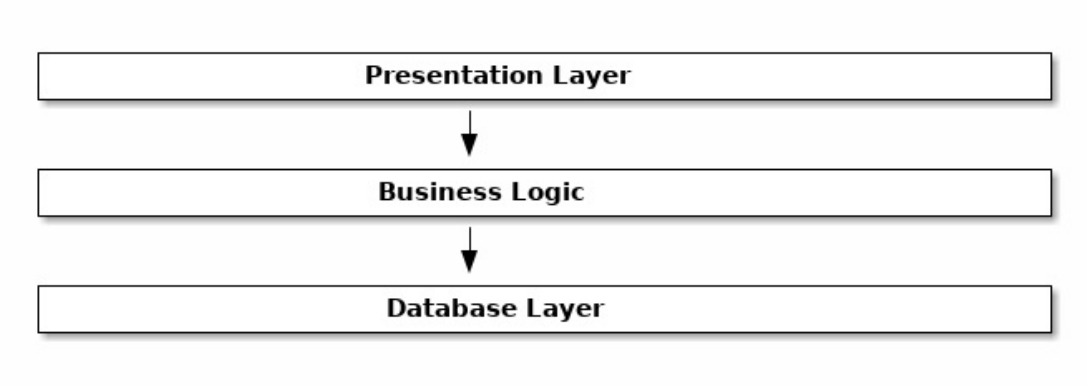


Figure 2-2. Layered Architecture

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…

But we want our domain model to have *no dependencies whatsoever*[2]. We don't want infrastructure concerns bleeding over into our domain model and slowing down our unit tests or our ability to make changes.

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

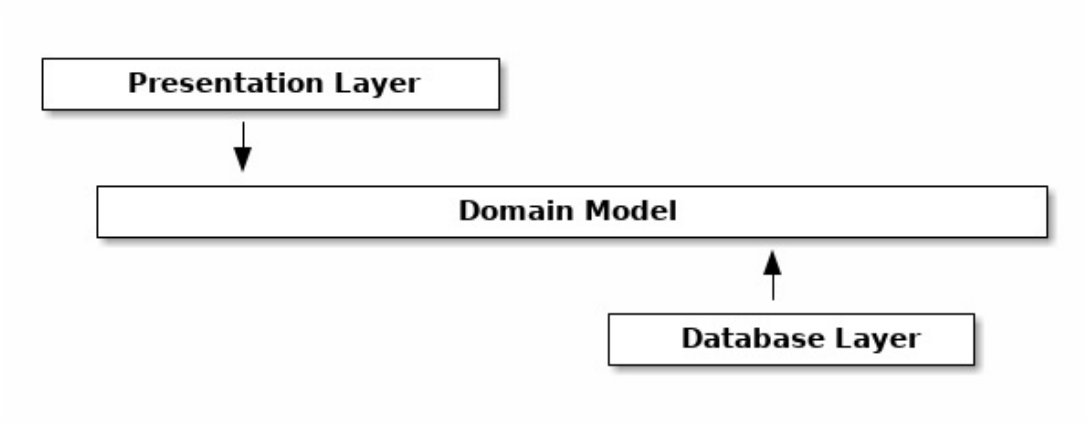


Figure 2-3. Onion Architecture

```
[ditaa, onion_architecture]
+-------------------------+
|   Presentation Layer    |
+-------------------------+
           |
           V
   +-----------------------------------------------------+
   |                    Domain Model                     |
   +-----------------------------------------------------+
                                      ^
                                      |
                                +-------------------+
                                |   Database Layer  |
                                +-------------------+
```

IS THIS PORTS AND ADAPTERS?

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

> *"Is this Ports and Adapters? Or is it Hexagonal Architecture? Is it the same as the 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).[3]

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.

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:

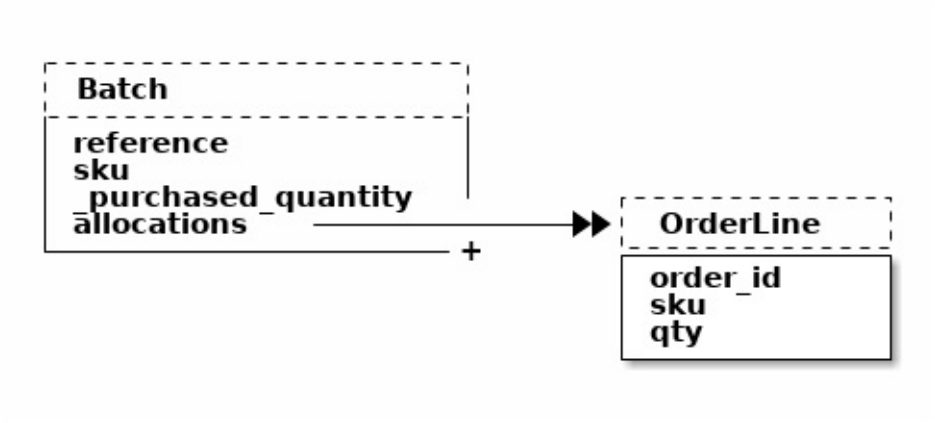


Figure 2-4. Our Model

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

The "Normal" ORM Way: Model Depends on ORM.

In 2019 it's unlikely that your team 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 because they exist to bridge the conceptual gap between the world of objects and domain modelling, 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 are loaded or persisted. This helps to keep our domain clean of direct dependencies on particular databases technologies.[4]

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

Example 2-2. 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):
    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 SQLAlchemy snippet above translates to something like this:

Example 2-3. 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 C for some examples of how you 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 an explicit *mapper* for how to convert between the schema and our domain model:

https://docs.sqlalchemy.org/en/latest/orm/mapping_styles.html#classical-mappings

Example 2-4. 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 using SQLAlchemy's abstractions.

❸ And 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, then 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 some tests for it, as in Example 2-5:

Example 2-5. Testing the ORM directly (throwaway tests) (test_orm.py)

```
def test_orderline_mapper_can_load_lines(session):  ❶
    session.execute(  ❶
        'INSERT INTO order_lines (orderid, sku, qty) VALUES '
        '("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've not 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 we'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 an additional abstraction called the repository pattern, which will be easier to write tests against, and will provide a simple, common 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 with architectural decisions, there is a trade-off you'll need to consider. As the Zen of Python says, "Practicality beats purity!"

At this point though, our API endpoint might look something like Example 2-6, and we could get it to work just fine.

Example 2-6. Using SQLAlchemy directly in our API endpoint

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

    # extract order line from request
    line = OrderLine(
        request.params['order_id'],
        request.params['sku'],
        request.params['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 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?

Example 2-7. You've got 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, and since 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 for our repository would look like:

Example 2-8. The simplest possible repository (repository.py)

```
class AbstractRepository(abc.ABC):

    @abc.abstractmethod  ❶
    def add(self, batch):
```

```
        raise NotImplementedError  ❷

    @abc.abstractmethod
    def get(self, reference):
        raise NotImplementedError
```

> **WARNING**
>
> 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 often 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 tend to 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.

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

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

> **NOTE**
>
> To really reap the benefits of ABCs (such as they may be) you'll want to be running some helpers like `pylint` and `mypy`.

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 tradeoffs of nothing.*
>
> —Rich Hickey

Whenever we introduce an architectural pattern in this book, we'll always be trying to 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 be reducing complexity overall, it does add complexity locally, and it has a cost in terms raw numbers of moving parts and ongoing maintenance.

Repository pattern is probably one of the easiest choices in the book though, if you've 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`) which 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 very simple abstraction over our storage layer, which we control. It would make it very easy to make fundamental changes to the way we store things (see Appendix B), and as we'll see, it is very easy to fake out for unit tests.

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

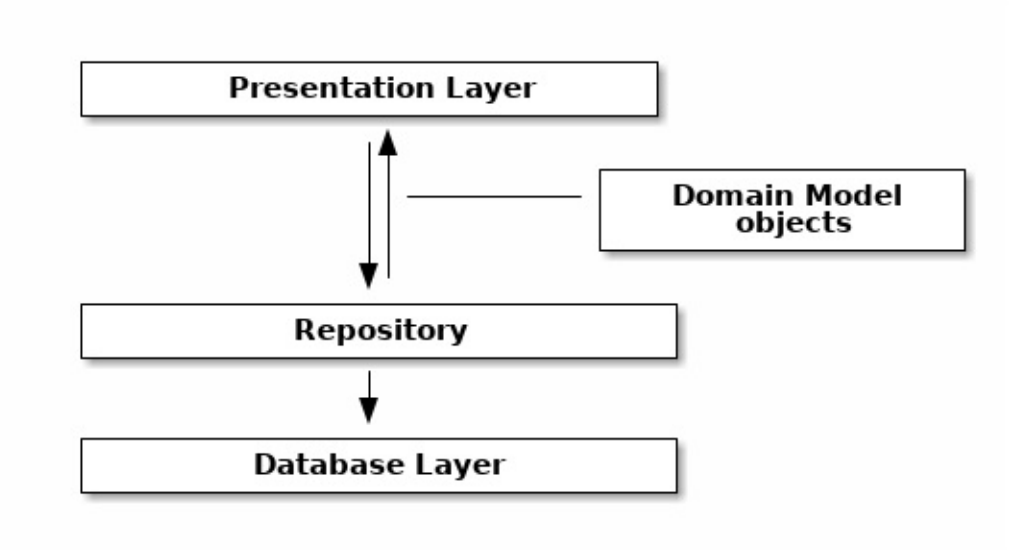


Figure 2-5. Repository pattern

```
[ditaa, repository_pattern_diagram]
  +------------------------------+
  |      Presentation Layer      |
  +------------------------------+
                 |^
                 ||          +------------------+
                 ||----------|   Domain Model   |
                 ||          |      objects     |
                 ||          +------------------+
                 V|
  +------------------------------+
  |          Repository          |
  +------------------------------+
                 |
                 V
  +------------------------------+
  |        Database Layer        |
  +------------------------------+
```

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.

TIP

> 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.

Example 2-9. 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 5.

❸ And 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:

Example 2-10. 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)  ❸
    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",
"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 different 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.

Whether or not you painstakingly write tests for every model is a judgement 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 roundtrip 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 Example 2-11:

Example 2-11. 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 Example 2-12:

Example 2-12. 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.” 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?

https://github.com/python-leap/code/tree/chapter_02_repository_exercise

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 *repository pattern*.

Example 2-13. A simple fake repository using a set (repository.py)

```
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 2-14. Example usage of fake repository (test_api.py)

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

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

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

> **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.
>
> 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 very easy to create a `FakeRepository` for testing, and it makes it easy to swap fundamental details of your infrastructure without disrupting your core application. See Appendix B for an example.

Bearing the Rich Hickey quote in mind, in each chapter we're going to try and summarise the costs and benefits of each architectural pattern we introduce. We want to be very 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 *Repository Pattern* and our *Persistence Ignorant Model*.

Table 2-1. Repository Pattern and Peristence Ignorance: The Trade-Offs

Pros	Cons
• We have a simple interface between persistent storage and our domain	• OTOH an ORM already buys you quite a lot of decoupling. if

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.
- Writing the domain model before thinking about persistence helps us focus on the business problem at hand.
- Our database schema is really simple because we have complete control over how we map our objects to tables.

you're using an ORM it should be pretty easy to swap between, eg, mysql and postgres

- how likely is it that you're going to want to change databases on your project?
- Maintaining the ORM mappings by hand is 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 Repository Pattern before

---

You'll be wondering, how do we actually instantiate these repositories, fake or real? What will our flask app actually look like? We'll find out in the next exciting instalment!

But first, a word from our sponsors…

---

1 we've used Flask because it's lightweight, but you don't need to understand Flask to understand this book. One of the main points we're trying to make is that your choice of web framework should be a minor implementation detail

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

3 Mark Seeman has an excellent blog post on the topic, which we recommend.

4 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!

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

6 You may be thinking, what about `list` or `delete` or `update`, but in the ideal world, we only modify our model objects one at a time, and delete is usually handled as a soft-delete, ie `batch.cancel()`. Finally, update is taken care of by the Unit of Work, as we'll see in

Chapter 5.

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 is an abstraction over permanent storage for example. But what makes a good abstraction? What do we want from them? And how do they relate to testing?

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, fitting in place like the gears of a watch.

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 some changes at all. This is the problem with the ball of mud pattern: as the application grows, the 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):

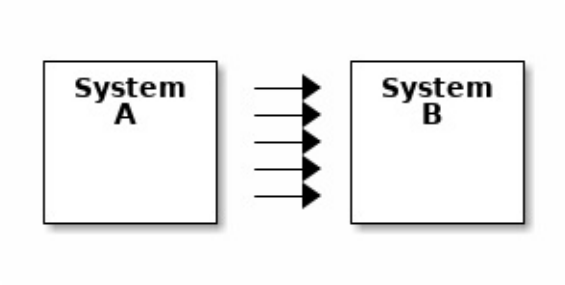


Figure 3-1. Lots of coupling

```
[ditaa,coupling_illustration1]
+--------+      +--------+
| System | ---> | System |
|   A    | ---> |   B    |
|        | ---> |        |
|        | ---> |        |
|        | ---> |        |
+--------+      +--------+
```

Figure 3-2. Less coupling

```
[ditaa,coupling_illustration2]
+--------+                          +--------+
| System | ---> /-------------\      | System |
|   A    | ---> |             |      |   B    |
|        | ---> | Abstraction | ---> |        |
|        | ---> |             |      |        |
|        | ---> \-------------/      |        |
+--------+                          +--------+
```

In both diagrams, we have a pair of subsystems, with the one dependent on the other. In the first diagram, there is a high degree of coupling

between the two because of reasons. If we need to change system B, there's a good chance that the change will ripple through to system A.

In the second, though, we have reduced the degree of coupling by inserting a new, simpler, abstraction. This abstraction serves to protect us from change by hiding away the complex details of whatever system B does.

Abstracting State Aids Testability

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

- If a file exists in the source, but not the destination, copy the file over.
- If a file exists in the source, but has a different name than in the destination, rename the destination file to match.
- If a file exists in the destination but not 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. In order to detect renames, we'll have to inspect the content of files. For this we can use a hashing function like md5 or SHA. The code to generate a SHA hash from a file is simple enough.

Example 3-1. Hashing a file (sync.py)

```
BLOCKSIZE = 65536

def hash_file(path):
    hasher = hashlib.sha1()  ❷
    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 interesting business logic. When we have to tackle a problem from first principles, we usually try to write a simple implementation, and then refactor towards 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:

Example 3-2. Basic sync algorithm (sync.py)

```
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  ❶

    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* okay, but before we run it on our hard drive, maybe we should test it? How do we go about testing this sort of thing?

Example 3-3. 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 very simple cases! The problem is that our domain logic, "figure out the difference between two directories," is tightly coupled to the IO code. We can't run our difference algorithm without calling the pathlib, shutil, and hashlib modules.

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 hard to read and write.

Choosing the right abstraction(s)

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

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

1. We interrogate the filesystem using `os.walk` and determine hashes for a series of paths. This is actually very similar in both the source and the destination cases.
2. We decide 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 that we can focus on the interesting logic.

> **NOTE**
>
> 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 from the prologue.

For (1) and (2), we've already intuitively started using an abstraction, a dictionary of hashes to paths, and you may already have been thinking, "why not use build up a dictionary for the destination folder as well as the source, then we just compare two dicts?" That seems like a very 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're going to 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 2 filesystem dicts as inputs, and expect lists of tuples of strings representing actions as outputs.

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?"

Example 3-4. 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.

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

Example 3-5. Split our code into three (sync.py)

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

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

    # 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])

...

def read_paths_and_hashes(root):  ❶
    hashes = {}
    for folder, _, files in os.walk(root):
        for fn in files:
```

```
            hashes[hash_file(Path(folder) / fn)] = fn
    return hashes


def determine_actions(src_hashes, dst_hashes, src_folder,
dst_folder):  ❷
    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
```

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

❷ The core of our "business logic," which says, "given these two sets of hashes and filenames, what should we copy/move/delete?" takes simple data structures and returns simple data structures.

❸ And our top-level module now contains almost no logic whatseover, it's just an imperative series of steps: gather inputs, call our logic, apply outputs.

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

Example 3-6. Nicer looking tests (test_sync.py)

```
    @staticmethod
    def
test_when_a_file_exists_in_the_source_but_not_the_destination():
        src_hashes = {'hash1': 'fn1'}
        dst_hashes = {}
        actions = list(determine_actions(src_hashes, dst_hashes,
Path('/src'), Path('/dst')))
```

```
        assert actions == [('copy', Path('/src/fn1'),
Path('/dst/fn1'))]

    @staticmethod
    def test_when_a_file_has_been_renamed_in_the_source():
        src_hashes = {'hash1': 'fn1'}
        dst_hashes = {'hash1': 'fn2'}
        actions = list(determine_actions(src_hashes, dst_hashes,
Path('/src'), Path('/dst')))
        assert 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 IO, we can easily test the core of our code.

Testing Edge-to-Edge with Fakes

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 IO, sort of *edge-to-edge*.

Example 3-7. Explicit dependencies (sync.py)

```
def synchronise_dirs(reader, filesystem, source_root, dest_root):
❶

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

    for sha, filename in src_hashes.items():
        if sha not in dst_hashes:
            sourcepath = source_root / filename
            destpath = dest_root / filename
```

```
            filesystem.copy(destpath, sourcepath) ❸

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

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

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

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

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

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

Example 3-8. 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 co-workers 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 your tests act on the exact same function that's used by your production code. The disadvantage is that we have to make our stateful components explicit and we have to pass them around. DHH famously described this as "test 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 some of our readers will be scratching their heads 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 are a code smell.

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

There's a few, closely related reasons for that:

1. 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.

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

2. 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 I call `shutil.copy` with the right arguments? This coupling between code and test *tends* to make tests more brittle in our experience.

   Martin Fowler wrote about this in his 2007 blog post Mocks Aren't Stubs

3. Over-use of mocks leads to complicated test suites that fail to

explain the code.

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.

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 over-mocked tests in his talk Test Driven Development: That's Not What We Meant

SO WHICH DO WE USE IN THIS BOOK? FCIS OR DI?

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 [Link to Come] for more exploration of making our dependency injection more explicit and centralised.

Wrap-up: "Depend on Abstractions."

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 IO. Finding the right abstraction is tricky, but here's a few heuristics and questions to ask yourself:

- Can I choose a familiar Python datastructure to represent the state of the messy system, and 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 are the dependencies and what is the core "business" logic?

Practice makes less-imperfect!

And now back to our regular programming…

---

1 We'll talk about TDD kata soon, but if you're new to the idea check out *http://www.peterprovost.org/blog/2012/05/02/kata-the-only-way-to-learn-tdd/*

Chapter 4. our First Use Case: Flask API and Service Layer.

Back to our allocations project!

In this chapter, we'll discuss the difference 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 the entire workflow for a use case.

By the end of this chapter, we'll have added a Flask API, that will talk to the Service Layer, which will serve as the entrypoint to our Domain Model. By making the service layer depend on the `AbstractRepository`, we'll be able to unit test it using `FakeRepository`, and then run it in real life using `SqlAlchemyRepository`. Figure 4-1 is a class diagram showing where we're heading.

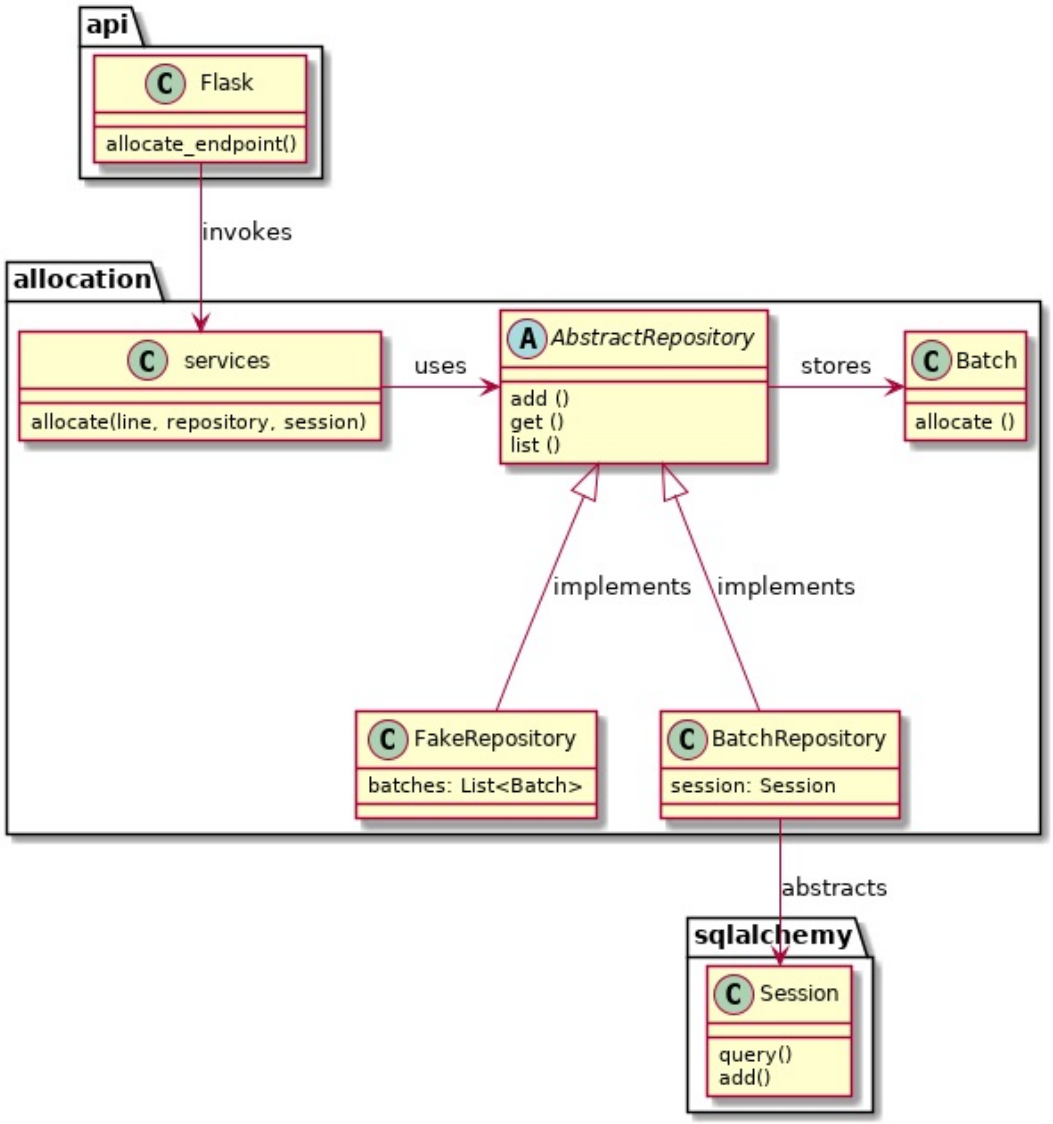


Figure 4-1. Our target architecture for the end of this chapter

```
[plantuml, chapter_03_class_diagram]
@startuml

package api {

  class Flask {
    allocate_endpoint()
  }
}

package sqlalchemy {
  class Session {
    query()
```

```
    add()
  }
}

package allocation {

  class services {
    allocate(line, repository, session)
  }

  abstract class AbstractRepository {
    add ()
    get ()
    list ()
  }

  class Batch {
    allocate ()
  }


  class FakeRepository {
    batches: List<Batch>
  }

  class BatchRepository {
    session: Session
  }

}

services -> AbstractRepository: uses
AbstractRepository -> Batch : stores

AbstractRepository <|-- FakeRepository : implements
AbstractRepository <|-- BatchRepository : implements
Flask --> services : invokes

BatchRepository ---> Session : abstracts
@enduml
```

Connecting our Application to the Real World

Like any good agile team, we're hustling to try and 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 try and plug all the moving parts together as quickly as we can, and then refactor towards a cleaner architecture. Here's our plan:

- 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.
- Refactor out a *Service Layer* to serve as an abstraction to capture the use case, and sit between Flask and our Domain Model. Build some service-layer tests and show how they can use the `FakeRepository`.
- 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.
- Add an extra service called `add_stock` so that our service-layer tests and end-to-end tests no longer need to go directly to the storage layer to set up test data.

A First End-To-End (E2E) Test

No-one is interested in getting into a long terminology debate about what counts as an E2E test vs a functional test vs an acceptance test vs an integration test vs unit tests. 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.

Example 4-1 shows a first cut:

Example 4-1. 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
    batch1, batch2, batch3 = random_batchref(1),
random_batchref(2), random_batchref(3)
    add_stock([  ❷
        (batch1, sku, 100, '2011-01-02'),
        (batch2, sku, 100, '2011-01-01'),
        (batch3, 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'] == batch2
```

❶ `random_sku()`, `random_batchref()` etc are little helper functions that add generate some randomised characters using the `uuid` module. Because we're running against an actual database now, this is one way to prevent different 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 find a nicer way of doing this later in the chapter.

❸ *config.py* is a module for getting configuration information. Again, this is an unimportant detail, and everyone has different ways of solving these problems, but if you're curious, you can find out more in Appendix A.

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

The Straightforward Implementation

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

Example 4-2. First cut 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-boxey), or maybe one that checks we can't allocate a second line if a first should have already depleted the batch:

Example 4-3. 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 get a commit in.

Error Conditions That Require Database Checks

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

Supposing 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:

Example 4-4. 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:

Example 4-5. 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 aren't anything to do with having a web API endpoint (you'd need them if you were building a CLI for example, see Appendix B), 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 *orchestration layer* or *use case layer*.

Do you remember the `FakeRepository` that we prepared in the last chapter?

Example 4-6. 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:

Example 4-7. Unit testing with fakes at the services 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` (code below) holds the `Batch` objects that will be used by our test.

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

❸ We also need a `FakeSession` to fake out the database session, see below:

Example 4-8. A fake database session (test_services.py)

```
class FakeSession():
    committed = False

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

(The fake session is only a temporary solution. We'll get rid of it and make

things even nicer in the next chapter, Chapter 5)

MOCKS VS FAKES; CLASSIC STYLE VS LONDON SCHOOL TDD

Couldn't we have used a mock (from `unittest.mock`) instead of building our own `FakeSession`, or instead of `FakeRepository`? What's the difference between a fake and a mock anyway?

We tend to find that building our own fakes is an excellent way of exercising design pressure against our abstractions. If our abstractions are nice and simple, then they should be easy to fake.

In fact in the case of `FakeRepository`, because our fake has actual behavior, using a magic mock from `unittest.mock` wouldn't really help.

In the case of `FakeSession`, the `session` object isn't one of our own abstractions, so the argument doesn't apply; in fact, a `unittest.mock` mock would have been just fine, but out of habit we avoided using one; in any case, we'll be getting rid of it in the next chapter.

In general we try and avoid using mocks, and the associated `mock.patch`. Whenever we find ourselves reaching for them, we often see it as an indication that something is missing from our design. You'll see a good example of that in [Link to Come] when we mock out an email-sending module, but eventually we replace it with an explicit bit of dependency injection. That's discussed in [Link to Come].

Regarding the definition of fakes vs mocks, the short but simplistic answer is:

- 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 only designed for use in tests; they wouldn't work "in real life", like our in-memory repository. 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, promise we're done with the test double terminology nitpicks now.)

What about London-school vs classic-style TDD? You can read more about those two in Martin Fowler's article just cited, as well as on stackoverflow, but in this book we're pretty firmly in the classicist camp. We like to build our tests around state, both in setup and assertions, and we like to work at the highest level of abstraction possible rather than doing checks on the behavior of intermediary collaborators.[1].

Read more on this shortly, in the high gear vs low gear section.

The fake `.commit()` lets us migrate a third test from the E2E layer:

Example 4-9. 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 get to a service function that looks something like Example 4-10:

Example 4-10. Basic allocation service (services.py)

```
class InvalidSku(Exception):
    pass


def is_valid_sku(sku, batches):  ❷
    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 And if all is well, we save/update any state we've changed.

That last step is a little unsatisfactory at the moment, our services layer is tightly coupled to our database layer, but we'll improve on that in the next chapter.

"DEPEND ON ABSTRACTIONS"

Notice one more thing about our service-layer function:

Example 4-11. the service depends on an abstraction (services.py)

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

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``[2] 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 "The Dependency Inversion Principle", This is what we mean when we says 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 the diagram at the end of the chapter, [Link to Come].

See also Appendix B where we show a worked example of swapping out the *details* of which persistent storage system to use, while leaving the abstractions intact.

Still, the essentials of the services layer are there, and our Flask app now looks a lot cleaner, Example 4-12:

Example 4-12. 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
```

We see that the responsibilities of the Flask app are much more minimal, and more focused on just the web stuff:

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 And 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:

Example 4-13. E2E tests now only happy + 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')
    batch1, batch2, batch3 = random_batchref(1),
random_batchref(2), random_batchref(3)
    add_stock([
        (batch1, sku, 100, '2011-01-02'),
        (batch2, sku, 100, '2011-01-01'),
        (batch3, 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'] == batch2


@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.

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:

Example 4-14. Counting different types of test

```
▫grep -c test_ test_*.py
test_allocate.py:4
test_batches.py:8
test_services.py:3

test_orm.py:6
test_repository.py:2

test_api.py:4
```

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

Should Domain Layer Tests Move to the

Service Layer?

We could take this a step further. Since we can test the our software against the service layer, we don't really need tests for the domain model any more. Instead, we could rewrite all of the domain-level tests from chapter one in terms of the service layer.

Example 4-15. Rewriting a domain test at the service layer (test_services.py)

```
# 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([warehouse_batch, shipment_batch])
    session = FakeSession()

    line = OrderLine('oref', "RETRO-CLOCK", 10)

    services.allocate(line, repo, session)

    assert warehouse_batch.available_quantity == 90
```

Why would we want to do that?

Tests are supposed to help us change our system fearlessly, but very 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 some 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.

Every line of code that we put in a test is like a blob of glue, holding the system in a particular shape.

As we get further into the book, we'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 restricting ourselves to only testing 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 more free to refactor them.

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 the question,

it’s important to understand the trade-off between coupling and design feedback (see Figure 4-2.)

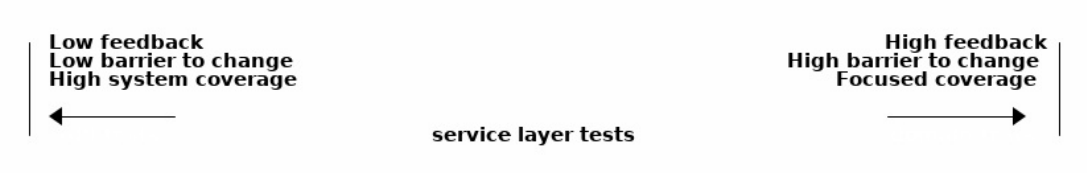


Figure 4-2. The test spectrum

```
[ditaa, test_spectrum_diagram]
| Low feedback
High feedback |
| Low barrier to change                                  High
barrier to change|
| High system coverage
Focused coverage |
|
|
| <---------
---------->  |
| API tests                 service-layer tests
domain tests |
```

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 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 DB 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.

Low and High 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 for the lower-coupling and high-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 out a new project, or when we hit a particularly gnarly problem, we will drop back down to writing tests against the domain

model, so that we get better feedback and executable documentation of our intent.

The metaphor we use is that of shifting gears. When starting off 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 we're forced to slow down by a hazard, we again drop down to a low gear until we can pick up speed again.

DIFFERENT TYPES OF TEST: RULES OF THUMB

- Write one end-to-end test per feature[3] to demonstrate that the feature exists and is working. This might be written against an HTTP api. These tests cover an entire feature at a time.
- Write the bulk of the tests for your system against the service layer. This offers a good trade-off between coverage, run-time, and efficiency. These tests tend to cover one code path of a feature and use fakes for IO.
- Maintain a small core of tests written against your domain model. These tests have highly-focused coverage, and are more brittle, but have the highest feedback. Don't be afraid to delete these tests if the functionality is later covered by tests at the service layer.

Fully Decoupling the Service Layer Tests From the Domain

We still have some 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:

Example 4-16. Before: allocate takes a domain object (services.py)

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

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

Example 4-17. After: allocate takes strings and ints (services.py)

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

We rewrite the tests in those terms as well:

Example 4-18. Tests now use primitives in function call (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`:

Example 4-19. Factory functions for fixtures are one possibility (test_services.py)

```
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, then 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:

Example 4-20. Test for new add_batch service (test_services.py)

```
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
```

And the implementation is just two lines

Example 4-21. A new service for add_batch (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:
    ...
```

> **NOTE**
>
> 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.

> **TIP**
>
> 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.

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.

Example 4-22. Services tests now only use services (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 only depend on the services 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 services-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 can be free of those hardcoded SQL queries and the direct dependency on the database.

The service function means adding the endpoint is very easy, just a little json-wrangling and a single function call:

Example 4-23. API for adding a batch (flask_app.py)

```
@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
```

> **NOTE**
>
> 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 RESTey, 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 very nice:

Example 4-24. API tests can now add their own batches (test_api.py)

```
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')
    batch1, batch2, batch3 = random_batchref(1),
random_batchref(2), random_batchref(3)
    post_to_add_batch(batch1, sku, 100, '2011-01-02')
    post_to_add_batch(batch2, sku, 100, '2011-01-01')
    post_to_add_batch(batch3, 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'] == batch2
```

EXERCISE FOR THE READER

We've now got services for `add_batch` and `allocate`, 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 here:

https://github.com/python-leap/code/tree/chapter_04_service_layer_exercise

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!

TIP

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 vs writes in [Link to Come].

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," things like 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 B), or the tests! They're an adapter for our domain too.
- We can write tests in "high gear" 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 abstract dependencies of our service layer:

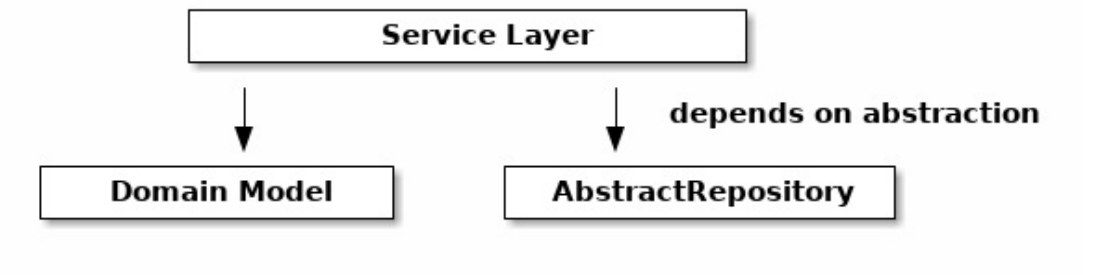


Figure 4-3. Abstract Dependencies of the service layer

```
[ditaa, service_layer_diagram_abstract_dependencies]
        +----------------------------+
        |         Service Layer      |
        +----------------------------+
           |                   |
           |                   | depends on abstraction
           V                   V
+------------------+     +--------------------+
|   Domain Model   |     | AbstractRepository |
+------------------+     +--------------------+
```

When we run the tests, we implement the abstract dependencies using `FakeRepository`, as in Figure 4-4:

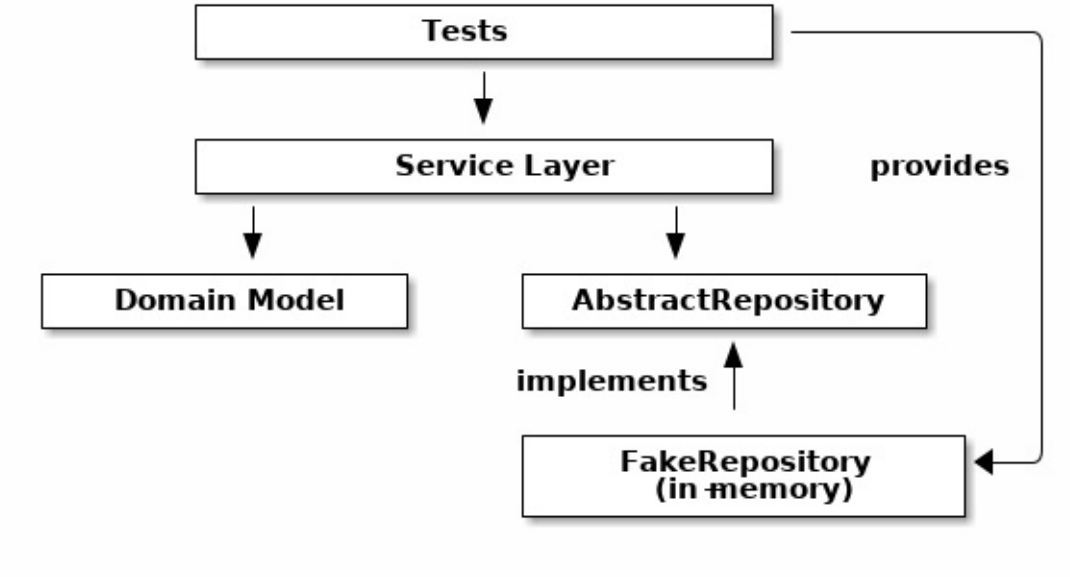


Figure 4-4. Tests provide an implementation of the abstract dependency

```
[ditaa, service_layer_diagram_test_dependencies]
        +-----------------------------+
        |           Tests             |-------------\
        +-----------------------------+             |
                       |                            |
                       V                            |
        +-----------------------------+             |
        |         Service Layer       |    provides |
        +-----------------------------+             |
           |                       |                |
           V                       V                |
+------------------+     +--------------------+     |
|   Domain Model   |     | AbstractRepository |     |
+------------------+     +--------------------+     |
                                   ^                |
                         implements |                |
                                    |                |
                         +----------------------+    |
                         |    FakeRepository    |<--/
                         |      (in-memory)     |
                         +----------------------+
```

And when we actually run our app, we swap in the “real” dependency, Figure 4-5:

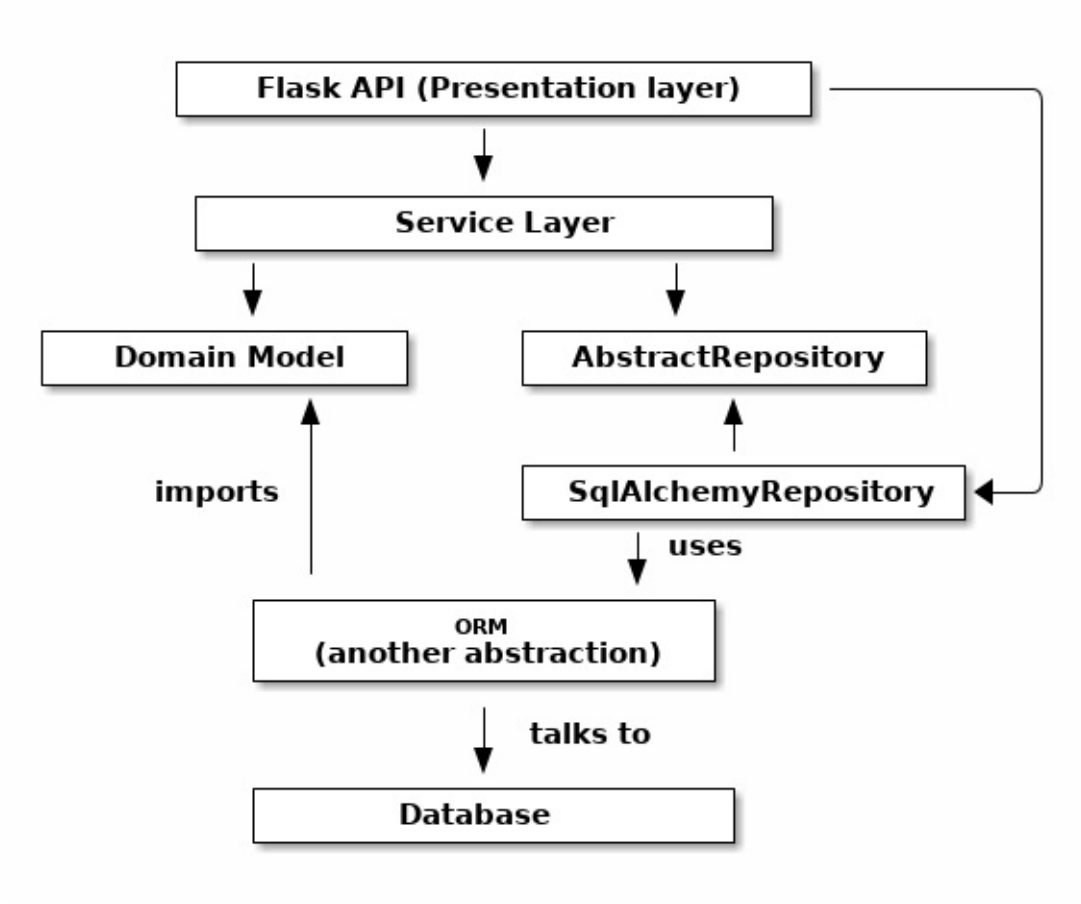
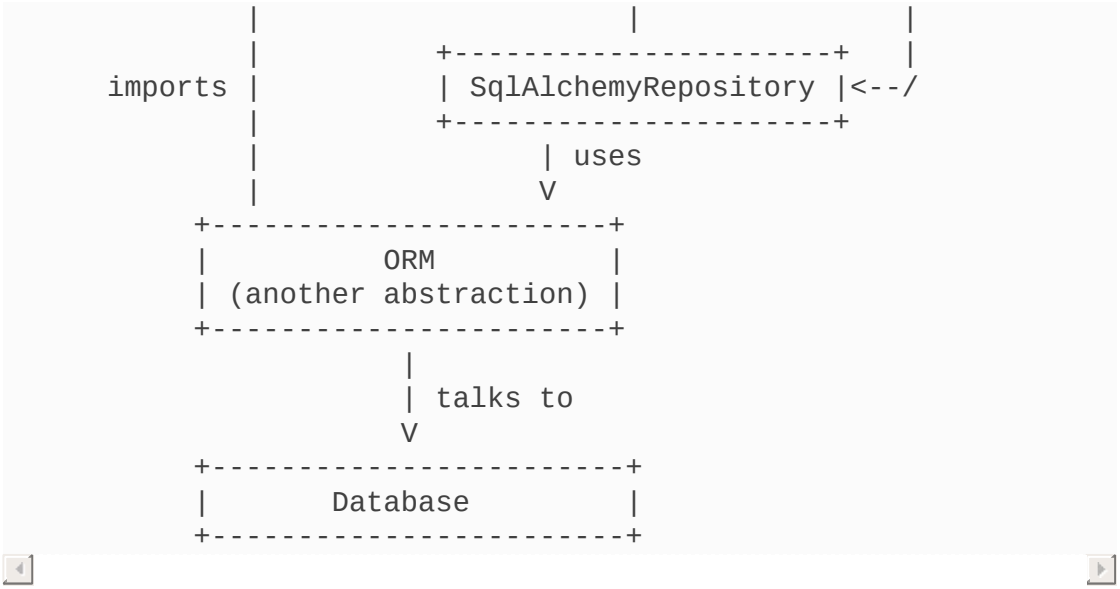


Figure 4-5. Dependencies at runtime

```
[ditaa, service_layer_diagram_runtime_dependencies]
       +---------------------------------+
       | Flask API (Presentation layer) |-----------\
       +---------------------------------+           |
                       |                             |
                       V                             |
        +------------------------------+             |
        |         Service Layer        |             |
        +------------------------------+             |
           |                       |                 |
           V                       V                 |
+------------------+     +--------------------+      |
|   Domain Model   |     | AbstractRepository |      |
+------------------+     +--------------------+      |
              ^                       ^              |
```



Wonderful. But there's still a bit of awkwardness we'd like to get rid of. The service layer is tightly coupled to a `session` object. In the next chapter, we'll introduce one more pattern that works closely with Repository and Service Layer, the Unit of Work pattern, and everything will be absolutely lovely. You'll see!

Table 4-1. Service Layer: The Trade-Offs

Pros	Cons
• We've got 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 *Repository Pattern* and a `FakeRepository`, we've got a nice way of writing tests at a	• 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 once 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

higher level than the Domain Layer; we can test more of our workflow without needing to go to integration tests.	of your controllers and down to the model layer, without needing to add an extra layer in between (aka “fat models, thin controllers”)

---

1 Which is not to say that we think the London School people are wrong. There are some insanely smart people that work that way. It’s just not what we’re used to

2 Is this Pythonic? Depending on who you ask, both abstract base classes and type hints are hideous abominations, and serve only to add useless, unreadable cruft to your code; beloved only by people who wish that Python was Haskell, which it will never be. “beautiful is better than ugly,” “simple is better than complex,” and “readability counts…” Or, perhaps they make explicit something that would otherwise be implicit (“explicit is better than implicit”). For the purposes of this book, we’ve decided this argument carries the day. What you decide to do in your own codebase is up to you.

3 what about happy path and unhappy path? We say, error-handling is a feature, so yes you need one E2E test for error handling, but probably not one unhappy-path test per feature

Chapter 5. Unit of Work Pattern

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

If the Repository is our abstraction over the idea of persistent storage, the Unit of Work is our abstraction over the idea of *atomic operations*. It will allow us to finally, fully, decouple our Service Layer from the data layer.

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

The Unit of Work Collaborates with Repository(-ies)

The unit of work will give us access to our repository(-ies), and then keep track of what objects were loaded and what the latest state is.[1] This gives us two useful things:

1) It gives us a stable snapshot of the database to work with, so that the objects we use aren't changing halfway through an operation.

2) It gives us a way to persist all of our changes at once so that if something goes wrong, we don't end up in an inconsistent state.

In other words, it allows us to manage concurrency and atomicity.

For now, that translates straight into a database transaction, but by giving ourselves our own abstraction, we can make it mean more things, as we'll

see when we get to [Link to Come].

In the last chapter, the service layer was tightly coupled to the SQLAlchemy session object, so we'll fix that.

But we'll also use be giving ourself a tool for explicitly saying that some work needs to work as an atomic unit. We either do all of it, or none of it. An error part of the way along should lead to any interim work being reverted.

What's a nice, Pythonic way of expressing that a block of code should run as a coherent whole, with some setup at the beginning, and some tidy-up at the end, some different handling for error and success cases? Something like `try/except/finally`?

A context manager.

Test-Driving a UoW with Integration Tests

Here's a test for a new `UnitofWork` (or UoW, which we pronounce "you-wow"). It's a context manager that allows us to start a transaction, retrieve and get things from repos, and commit:

Example 5-1. A basic "roundtrip" test for a Unit of Work (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


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'
```

❶ We initialise the Unit of Work 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 And we call `commit()` on it when we're done.

Unit of Work and Its Context Manager

In our tests we've implicitly defined an interface for what a unit of work needs to do, let's make that explicit by using an abstract base class:

Example 5-2. the Unit of Work context manager in the abstract (src/allocation/unit_of_work.py)

```
class AbstractUnitOfWork(abc.ABC):

    def __enter__(self):  ❶
        return self  ❷

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

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

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

    def init_repositories(self, batches:
repository.AbstractRepository):  ❺
        self._batches = batches

    @property
    def batches(self) -> repository.AbstractRepository:  ❺
        return self._batches
```

❶ 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. They're our setup and teardown phases.

❷ The enter returns `self`, because we want access to the `uow` instance and its attributes and methods, inside the `with` block.

❸ It provides a way to explicitly commit our work

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).

❺ The other thing we provide is an attribute called `.batches`, which

will give us access to the batches repository. The `init_repositories()` method is needed because different subclasses will want to initialise repositories in slightly different ways, this just gives us a single place to do that.

The Real Unit of Work Uses Sqlalchemy Sessions

Example 5-3. the real SQLAlchemy Unit of Work (src/allocation/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 = session_factory()  # type: Session  ❷

self.init_repositories(repository.SqlAlchemyRepository(self.session
))  ❷

    def commit(self):  ❸
        self.session.commit()

    def rollback(self):  ❸
        self.session.rollback()
```

❶ The module defines a default session factory that will connect to postgres, but we allow that to be overriden in our integration tests, so that we can use SQLite instead.

❷ The init is responsible for starting a database session, and starting a real repository that can use that session

❸ 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 Unit of Work in our service layer tests

Example 5-4. Fake unit of work (tests/unit/test_services.py)

```
class FakeUnitOfWork(unit_of_work.AbstractUnitOfWork):

    def __init__(self):
        self.init_repositories(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 Unit of Work and Repository classes

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 3rd party code. Some people say "don't mock what you don't own".

❸ And in our tests, we can instantiate a UoW and pass it to our service layer, instead of a repository and a session, which is considerably less cumbersome.

Using the UoW in the Service Layer

And here's what our new service layer looks like:

Example 5-5. Service layer using UoW (src/allocation/services.py)

```
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 only has the one dependency, once again on an *abstract* Unit of Work.

Explicit Tests for Commit/Rollback Behaviour

To convince ourselves that the commit/rollback behavior works, we wrote a couple of tests:

Example 5-6. 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 == []
```

> **TIP**
>
> 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, ie the same engine. For now we're getting away with using SQLite instead of Postgres, but in Chapter 6 we'll switch some of the tests to using the real DB. It's convenient that our UoW class makes that easy!

Explicit vs Implicit Commits

A brief digression on different ways of implementing the UoW pattern.

We could imagine a slightly different version of the UoW, which commits by default, and only rolls back if it spots an exception:

Example 5-7. 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
        self.session.close()  ❸
```

❶ should we have an implicit commit in the happy path?
❷ and roll back only on exception?
❸ and maybe close sessions too?

It would allow us to save a line of code, and remove the explicit commit from our client code:

Example 5-8. ... would save us a line of code (src/allocation/services.py)

```
def add_batch(ref: str, sku: str, qty: int, eta: Optional[date],
start_uow):
    with start_uow() as uow:
        uow.batches.add(model.Batch(ref, sku, qty, eta))
        # uow.commit()
```

This is a judgement call, but we tend to prefer requiring the explicit commit so that we have to choose when to flush state.

Although it's 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 "always-rollback" to "only-rollback-on-error," because the former feels easier to understand; rollback rolls back to the last commit, so either the user did one, or we blow their changes away. Harsh but simple.

As to the option of using `session.close()`, we have played with that in the past, but we always end up having to look up the SQLAlchemy docs to find out exactly what it does. And besides, why not leave the session open for the next time? But you should experiment and figure out your own preferences here.

Examples: Using UoW to Group Multiple Operations Into an Atomic Unit

Here's 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

Supposing we want to be able to deallocate and then reallocate orders?

Example 5-9. 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 do `allocate()`, obviously.

❷ But 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!

Example 5-10. 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
        model.allocate(line)
        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?

Example 5-11.

```
└── 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 feel 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 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 do is try to implement a UoW from scratch. 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 initialise 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`.

https://github.com/python-leap/code/tree/chapter_05_uow_exercise

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?

Wrap-Up

Hopefully we've convinced you that the Unit of Work is a useful pattern, and hopefully you'll agree 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 unit-of-work 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 db, 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?

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.

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 lifecycle 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.*

UNIT OF WORK PATTERN: WRAP-UP

Unit of Work 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

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 which 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 rollback our work at the end of 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.

Table 5-1. Unit of Work: The Trade-Offs

Pros	Cons
• We've got a nice abstraction over the concept of atomic operations, and the context manager makes it very 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	• 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

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 it
- And we'll see in later chapters, atomicity isn't only about transactions, it can help us to work with events and the message bus.

actually 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.

---

1 You may have come across the word *collaborators*, which is what responsibility-driven design would use to describe the relationship between the unit of work and the repository. In the terminology, clusters of objects that collaborate in their roles are called *object neighborhoods* which is, in our professional opinion, totally adorable.

Chapter 6. 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 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.

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 addresss?

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 forward over e-mail. 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

If we were writing a hotel booking system, we might have the constraint that no two bookings can exist for the same hotel room on the same night. This supports the invariant that no room is double booked.

Of course, sometimes we might need to temporarily *bend* the rules. Perhaps we need to shuffle the rooms around due to a VIP booking. While we're moving people around, 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. Either that, or we raise an error and refuse to complete the operation.

Let's look at a couple of concrete examples from our business requirements

> - *An order line can only be allocated to one batch at a time.*
>
> —the business

This is a business rule that implements a constraint. The constraint is that an order line is allocated to either zero or one batches, 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 doing that. By

nominating an aggregate to be “in charge of” all batches, we’ll have a single place where we can enforce this constraint.

Invariants and Concurrency

Let’s look at another one of our business rules:

> - *I 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. The invariant is that we never oversell stock by allocating two customers to the same physical cushion. Every time we update the state of the system, our code needs to ensure that we don’t break the invariants.

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 happening simultaneously on the same row or same table.

As we start to think about scaling up our app, we realise that our model of allocating lines against all available batches may not scale. If we’ve got 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.

In the rest of this chapter, we'll first discuss choosing an aggregate and demonstrate its usefulness for managing invariants at the conceptual level, as the single entrypoint in our code for modifying batches and allocations. In that role, it's defending us from programmer error.

Then we'll return to the topic of concurrency and discuss how the aggregate can enforce invariants at a lower level. In that role, it'll be defending us against concurrency / data integrity bugs.

Choosing the Right Aggregate

> *An AGGREGATE is a cluster of associated objects that we treat as a unit for the purpose of data changes.*
>
> —Eric Evans, DDD blue book

Even if it weren't for the data integrity concerns, as a model gets more complex and grows more different Entity and Value Objects, all of which start pointing to each other, it can be hard to keep track of who can modify what. Especially when we have *collections* in the model like 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.

TIP

Just like 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 are "private".

Some sort of `Order` object might suggest itself, but that's more about order lines, and we're more concerned about something that provides some sort of conceptual unity for collections of batches.

When we allocate an order line, we're actually only interested in batches that have the same SKU as the order line. Some sort of concept of `GlobalSkuStock` or perhaps just simply `Product` — after all, that was the first concept we came across in our exploration of the domain language back in Chapter 1.

Let's go with that for now, and see how it looks

Example 6-1. Our chosen Aggregate, Product (src/allocation/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 It holds a reference to a collection of `batches` for that sku

❸ And finally, we can move the `allocate()` domain service to being a method on `Product`.

> **NOTE**
>
> 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 microservices and bounded contexts, the concept of Product in one app can be very different from another.[1]

> **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 services, 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, better to have several different 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 rules for translating that to and from other microservices it integrates with.
>
> Whether or not you've got 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.

1 Aggregate = 1 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.

In our case, we'll switch from `BatchRepository` to `ProductRepository`:

Example 6-2. Our new UoW and Repository (unit_of_work.py and repository.py)

```
class _UnitOfWork:
    def __init__(self, session):
        self.session = session
        self.products = repository.ProductRepository(session)


#...

class ProductRepository:
    #...

    def get(self, sku):
        return
self.session.query(model.Product).filter_by(sku=sku).first()
```

And our service layer evolves to use `Product` as its main entrypoint:

Example 6-3. Service layer (src/allocation/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
```

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 like we did.

Of course you could cheat and copy/paste from the listings above, 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.

https://github.com/python-leap/code/tree/chapter_06_aggregate_exercise

We've put in a "cheating" implementation in that delegates to the existing `allocate()` function, so you should be able to evolve that towards the real thing.

We've marked a couple of tests with `@pytest.skip()`, come back to then when you're done and you've read the rest of this chapter, to have a go at implementing version numbers. Bonus points if you can get SQLAlchemy to do them for you by magic!

Version Numbers

We've got our new aggregate and we're using it in all the right places, the remaining question is: how will we actually enforce our data integrity rules? 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? The answer is to have a single attribute on the Product model which acts as a marker for the whole state change being complete, and we use it as the single resource that concurrent workers can fight over: if two transactions both read the state of the world for `batches` at the same time, and they both want to update the `allocations` tables, we force both of them to also try and update the `version_number` in the `products` table, in such a way that only one of them can win and the

world stays consistent.

There are essentially 3 options for implementing version numbers:

1. `version_number` lives in domain, we add it to the `Product` constructor, and `Product.allocate()` is responsible for incrementing it.
2. The services 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. Or, 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[2].

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 tradeoff is to put them in the domain.

Example 6-4. Our chosen Aggregate, Product (src/allocation/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!

```
we're just setting _something_ so the db complains, could use
uids,
also discuss similarity with eventsourcing version numbers.
```

Testing for our Data Integrity Rules

Now to actually 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:

Example 6-5. An integration test for concurrency behavior (tests/integration/test_uow.py)

```
def
test_concurrent_updates_to_version_are_not_allowed(postgres_sessio
n_factory):
    sku, batch = random_ref('s'), random_ref('b')
    session = postgres_session_factory()
    insert_batch(session, batch, sku, 100, eta=None,
product_version=3)
```

```
    session.commit()

    exceptions = []
    o1, o2 = random_ref('o1'), random_ref('o2')
    target1 = lambda: try_to_allocate(o1, sku, exceptions)
    target2 = lambda: try_to_allocate(o2, sku, exceptions)
    t1 = threading.Thread(target=target1)  ❶
    t2 = threading.Thread(target=target2)  ❶
    t1.start()
    t2.start()
    t1.join()
    t2.join()

    [[version]] = session.execute(
        "SELECT version_number FROM products WHERE sku=:sku",
        dict(sku=sku),
    )
    assert version == 4  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
```

❶ We set up two threads that will reliably produce the concurrency behavior we want: `read1, read2, write1, write2`. (see below for the code being run in each thread).

❷ We assert that the version number has only been incremented once.

❸ We can also check on the specific exception if we like

❹ And we can make sure that only one allocation has gotten through.

Example 6-6. time.sleep can reliably produce 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)
```

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:

Example 6-7. Set isolation level for session (src/allocation/unit_of_work.py)

```
DEFAULT_SESSION_FACTORY = sessionmaker(bind=create_engine(
    config.get_postgres_uri(),
    isolation_level="SERIALIZABLE",
))
```

Transaction isolation levels are tricky stuff, it's worth spending time understanding the documentation.

SELECT FOR UPDATE Can Also Help

An alternative to using the `SERIALIZABLE` isolation level is to use SELECT FOR UPDATE, which will produce different behavior: two concurrent transactions will not be allowed to do a read on the same rows at the same time.

Example 6-8. SqlAlchemy with_for_update (src/allocation/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)
```

In our simple case, it's not obvious which to prefer. In a more complex scenario, `SELECT FOR UPDATE` might lead to more deadlocks, while `SERIALIZABLE` having more of an "optimistic locking" approach and might lead to more failures, but the failures might be more recoverable. So, as usual, the right solution will depend on circumstances.

RECAP: AGGREGATES AND CONSISTENCY BOUNDARIES

Choose the right aggregate

bla

Something something transactions

bla bla.

Table 6-1. Aggregates: The Trade-Offs

Pros	Cons
	• Yet another new

- Python might not have “official” public and private methods, but the underscores convention belies the fact that it’s useful to try and separate 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.
- Modelling our operations around explicit consistency boundaries helps to 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.

concept for new developers to take on. Explaining Entities vs Value Objects was already a mental load, now there’s a third type of domain model object?

- Sticking rigidly to the rule that we only modify one aggregate at a time is a big mental shift.
- Dealing with eventual consistency between aggregates can be complex.

---

1 Well, either that, or it’s just a bad name. but `SKUStock` would be so *awkward*!

2 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, eg for a CsvRepository?

Appendix A. 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.

Example A-1 shows the folder structure:

Example A-1. Project tree

```
.
├── docker-compose.yml  ❶
├── Dockerfile  ❶
├── license.txt
├── Makefile  ❷
├── mypy.ini
├── README.md
├── requirements.txt
├── src  ❸
│   ├── allocation
│   │   ├── config.py
│   │   ├── flask_app.py
│   │   ├── model.py
│   │   ├── orm.py
│   │   ├── repository.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 can also run the tests (for CI). A more complex project might have several Dockerfiles, although we've found that minimising 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. 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 go out of date.

❸ All the actual source code for our app, including the domain model, the flask app, and infrastructure code, lives in a Python package inside *src*,[2] 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 including *domain_model/*, *infrastructure/*, *services/*, *api/*

❹ Tests live in their own folder, with subfolders to 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*.

> **TIP**
>
> The pytest docs are really good on test layout and importability.

Let's look at a few of these 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:

- 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
- and different settings for 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*_. Example A-2 shows a couple of examples from our app:

Example A-2. 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[3].

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[4]), Example A-3:

Example A-3. 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
    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 some other infrastructure services like a database. In production you may 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 Protip: if you're mounting volumes to share source folders between your local dev machine and the container, the `PYTHONDONTWRITEBYTECODE` env var 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 And the `ports` section allows us to expose the ports from inside the containers to the outside world[5]--these correspond to the default ports we set in *config.py*.

> **NOTE**
>
> 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, as in Example A-4:

Example A-4. 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. See Example A-5.

Example A-5. pip-installable modules in 3 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, this

is all you need.

Dockerfile

Dockerfiles are going to be very project-specific, but here's a few key stages you'll expect to see:

Example A-6. Our Dockerfile (Dockerfile)

```
FROM python:3.7-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/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
❸ Copying and installing our source
❹ Optionally configuring a default startup command (you'll probably override this a lot from the command-line)

> **TIP**

One thing to note is that we install things in the order of how frequently they are likely to change. This allows us to maximise docker build cache reuse. I can't tell you how much pain and frustration belies this lesson.

Tests

Our tests are kept alongside everything else, as in Example A-7:

Example A-7. 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.

We've not needed to make tests pip-installable, but if you have difficulties with import paths, you might find it helps.

---

1 Splitting out images for prod and test 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 (eg web api vs pubsub 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 More on *src* folders: https//hynek.me/articles/testing-packaging/

3 You might prefer to fail hard if an env var is not set, but this gives us a local dev setup that "just works" (as much as possible).

4 Harry hates YAML. He says he can never remember the syntax or how it's supposed to indent.

5 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, eg with docker-compose.override.yml

Appendix B. 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 on from Chapter 5.

Just as we finish building out our Flask API and getting it ready for release, the business come to us apologetically saying they're not ready to use our API and could we build a thing that reads just batches and orders from a couple of CSVs and outputs a third 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 some E2E test to show you how the CSVs flow in and out:

Example B-1. A first CSV test (tests/e2e/test_csv.py)

```
def
test_cli_app_reads_csvs_with_batches_and_orders_and_outputs_alloca
tions(
        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:

Example B-2. 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 actually not looking too bad! And we're re-using our domain model objects and our domain service….

But it's actually 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:

Example B-3. 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 our Unit of Work.

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's actually quite 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 just gives us the familiar `.list()` API which gives us the illusion of an in-memory collection of domain objects.

Example B-4. A repository that uses CSV as its storage mechanism (src/allocation/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 Unit of Work for CSVs would look like:

Example B-5. A Unit of Work for CSVs: commit = csv.writer.

(src/allocation/csv_uow.py)

```
class CsvUnitOfWork(unit_of_work.AbstractUnitOfWork):

    def __init__(self, folder):
        self.init_repositories(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 just 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:

Example B-6. Allocation with CSVs in 9 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 C. Repository and Unit of Work Patterns with Django

Supposing you wanted to use Django instead of SQLAlchemy and Flask, how might things look?

First thing is to choose where to install it. I put it in a separate package next to our main allocation code:

Example C-1.

```
├── src
│   ├── allocation
│   │   ├── config.py
│   │   ├── model.py
│   │   ├── repository.py
│   │   ├── services.py
│   │   └── unit_of_work.py
│   ├── djangoproject
│   │   ├── alloc
│   │   │   ├── apps.py
│   │   │   ├── __init__.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
#...
```

Repository Pattern with Django

I used a plugin 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:

Example C-2. First repository test adapted (tests/integration/test_repository.py)

```
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:

Example C-3. Second repository test is more involved (tests/integration/test_repository.py)

```
@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_batch1 =
django_models.Batch.objects.create(reference="batch1", sku=sku,
qty=100, eta=None)
    d_batch2 =
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:

Example C-4. A Django repository. (src/allocation/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.

Custom Methods on Django ORM Classes to Translate To/From our Domain Model

> **NOTE**
>
> As in Chapter 2, we use dependency inversion. The ORM (Django) depends on the model, and not the other way around

Those custom methods look something like this:

Example C-5. Django ORM with custom methods for domain model conversion (src/djangoproject/alloc/models.py)

```
from django.db import models
from allocation import model as domain_model

class Batch(models.Model):
    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 need an explict try-get/except to handle the upsert.

❷ I'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.

Unit of Work Pattern with Django

The tests don't change too much

Example C-6. 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 body of the tests are pretty much the same as they were with SQLA

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 goes to find what actual invocation of Django's transaction magic would work:

Example C-7. Unit of Work adapted for Django (src/allocation/unit_of_work.py)

```
class DjangoUnitOfWork(AbstractUnitOfWork):
```

```
    def __init__(self):
        self.init_repositories(repository.DjangoRepository())

    def __enter__(self):
        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 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 got through all the objects that have been touched by every repository and manually updated 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 btw).

Example C-8. 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)
```

Conclusions: Would You Bother?

OK it works but it does feel like more effort than Flask/SQLAlchemy. Why is that, and when might you still choose Django?

- it's hard because the ORM doesn't work in the same way. We

don’t have an equivalent of the SQLAlchemy classical mapper, so our ActiveRecord and our domain model can’t be the same object. Instead we have to build a manual translation layer behind the repository instead. That’s more work (although once it’s done the ongoing maintenance burden shouldn’t be too high).

- it’s also hard because you need to integrate `pytest-django` and think carefully about test databases etc

So why might you still do it?

- when migrating an existing project that has Django?
- or because you want the Django Admin? (but we’d have to say that’s likely to be a bad idea, it goes against the grain of wanting to decouple your model and business logic from the ORM…)