PROFESSIONAL SOFTWARE ENGINEERING

PSE SWE LE 4 und 5 - Domain Driven Design

DDD with Python

Dominik Neumann


Knowledge Foundation
@ Reutlingen University
BUILD A PYTHON APPLICATION
GUIDED BY DDD PATTERNS

OUTLINE

We follow many chapters from the book „Architecture Patterns with Python“ by Harry Percival and Bob Gregory:

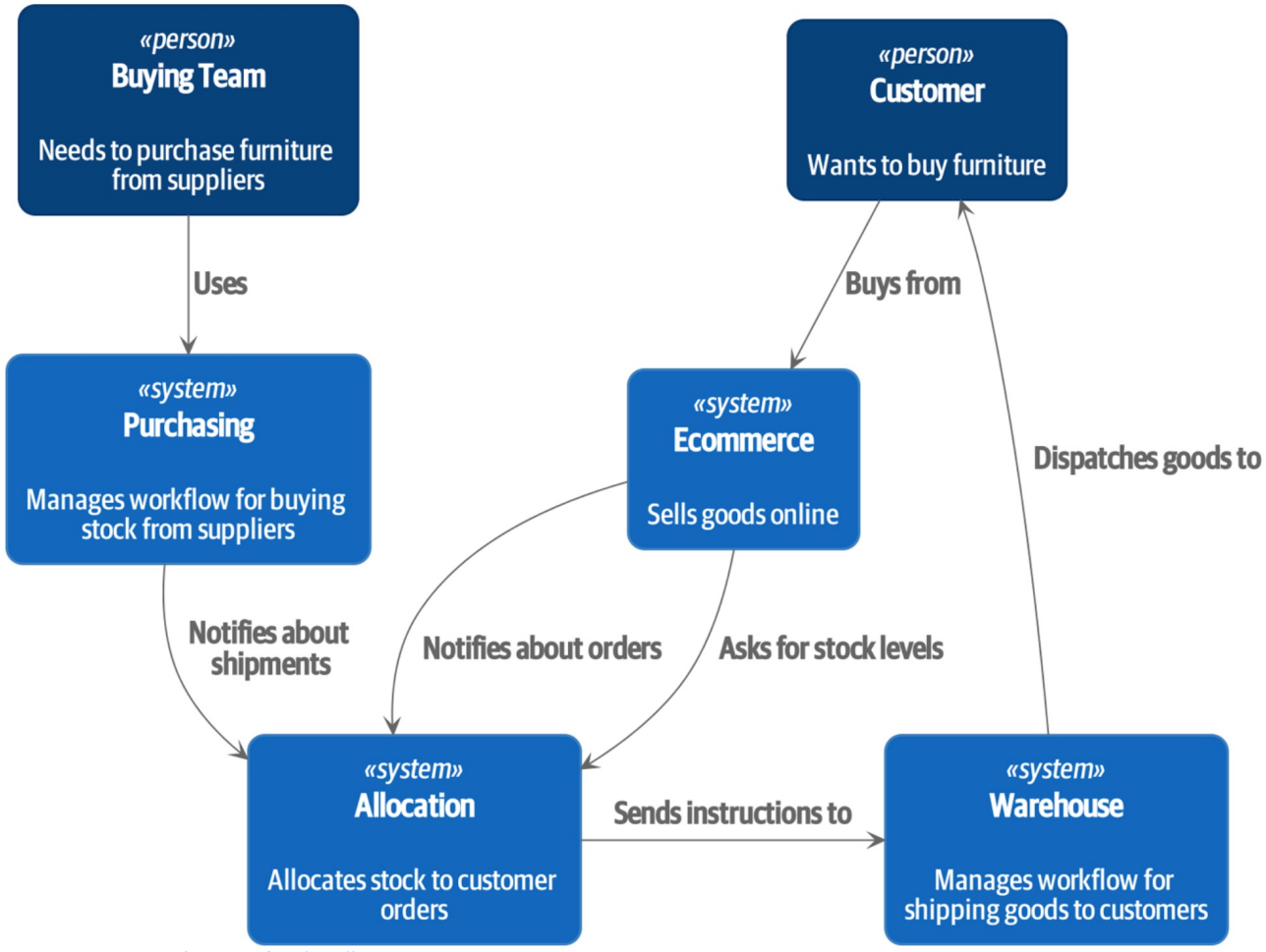
- https://www.cosmicpython.com

They introduce proven architectural design patterns to help Python developers manage application complexity:

- DDD: Entities, Value Objects, and Aggregates
- DDD: Repository
- Unit of Work patterns for persistent storage
- DDD: Events, commands, and the message bus
- CRQS: Command-query responsibility segregation
- Event-driven architecture and reactive microservices

PROBLEM SPACE

- Furniture Retailer with 4 domains:
  - Purchasing
  - Allocation
  - Ecommerce
  - Warehouse



Source: https://www.cosmicpython.com/book/chapter_01_domain_model.html

THE PROBLEM

The business decides to implement an exciting new way of allocating stock:

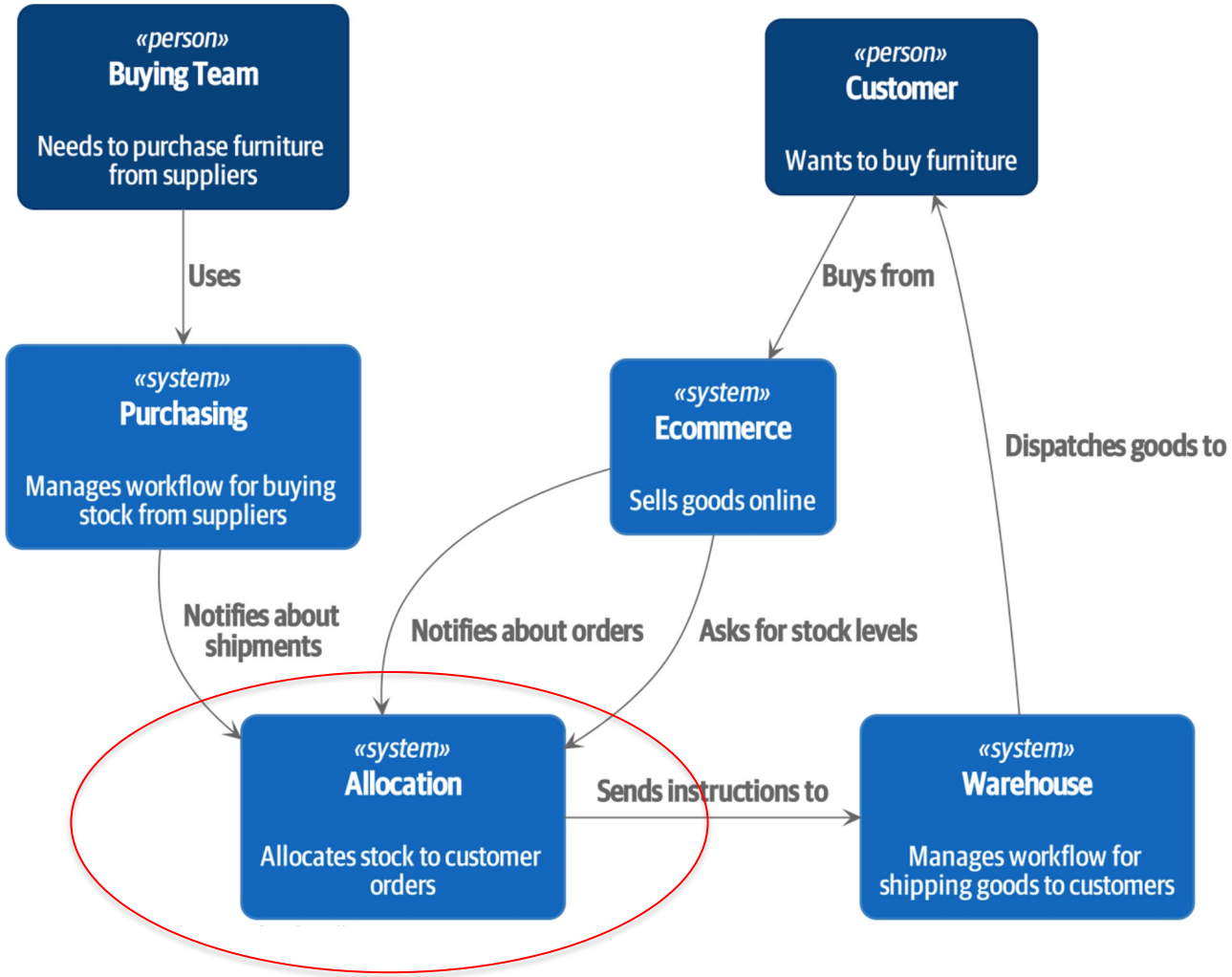
- Until now, the business has been presenting stock and lead times based on what is physically available in the warehouse. If and when the warehouse runs out, a product is listed as "out of stock" until the next shipment arrives from the manufacturer.

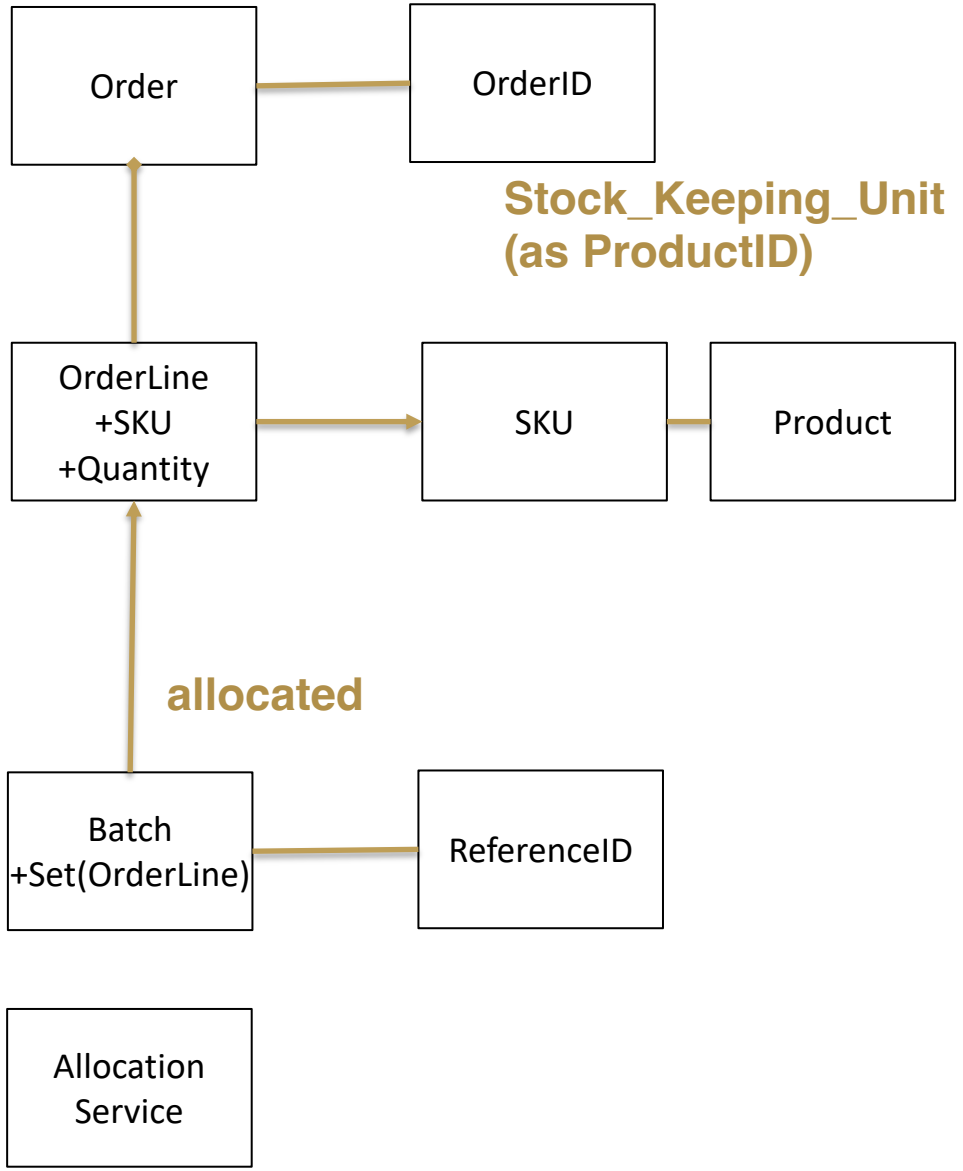
- Here's the innovation: if we have a system that can keep track of all our shipments and when they're due to arrive, we can treat the goods on those ships as real stock and part of our inventory, just with slightly longer lead times. Fewer goods will appear to be out of stock, we'll sell more, and the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

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

Source: https://www.cosmicpython.com/book/chapter_01_domain_model.html

THE ALLOCATION SYSTEM



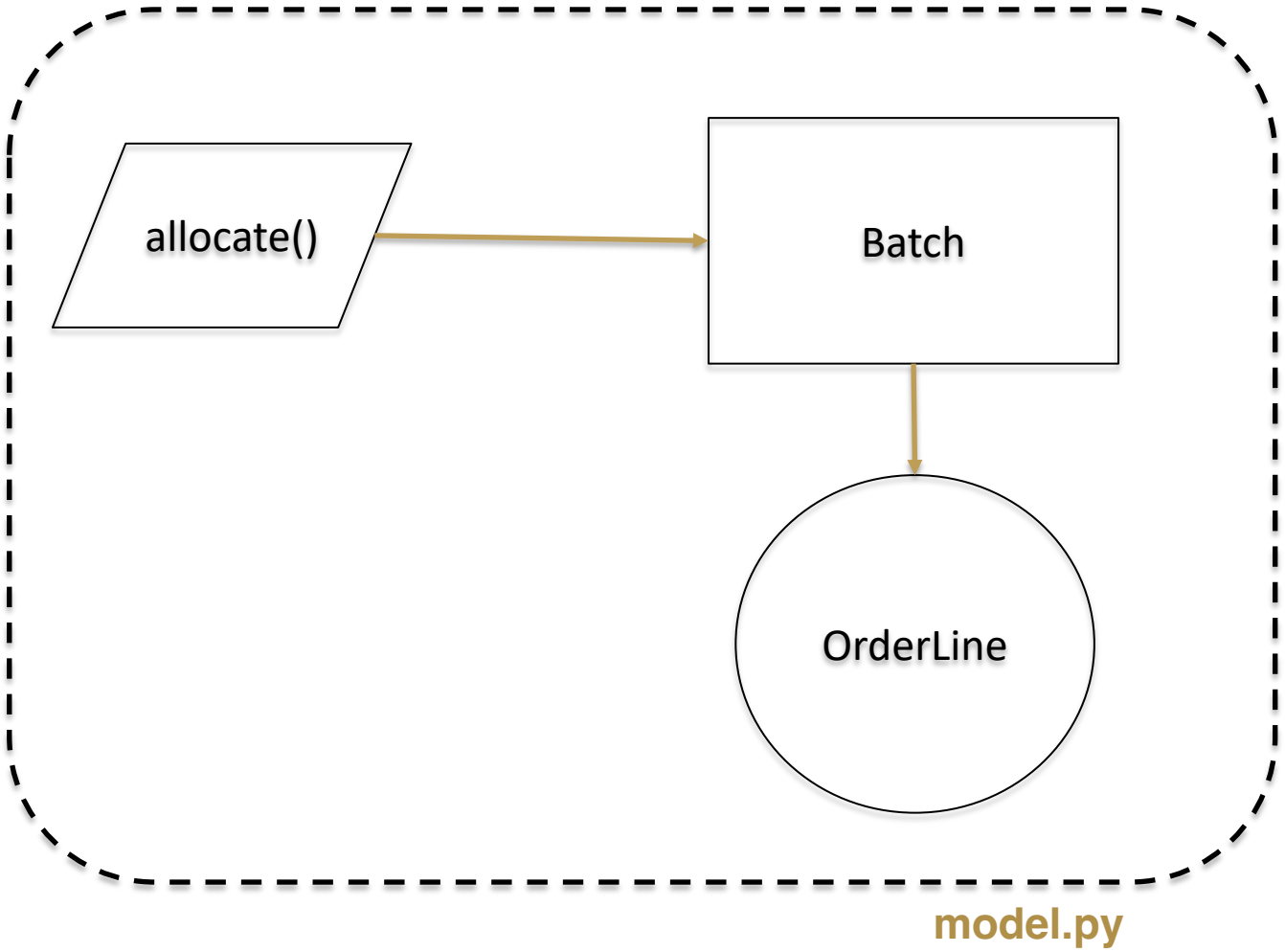


DOMAIN SERVICE

- Not everything has to be an object.
- „**Sometimes, it just isn't a thing.**“ (Eric Evans, Domain-Driven Design)
- Evans discusses the idea of Domain Service operations that don't have a natural home in an entity or value object. A thing that allocates an order line, given a set of batches, sounds a lot like a function.
- Domain services are not the same thing as the services from the service layer, but 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.

Source: https://www.cosmicpython.com/book/chapter_01_domain_model.html

DOMAIN MODEL.

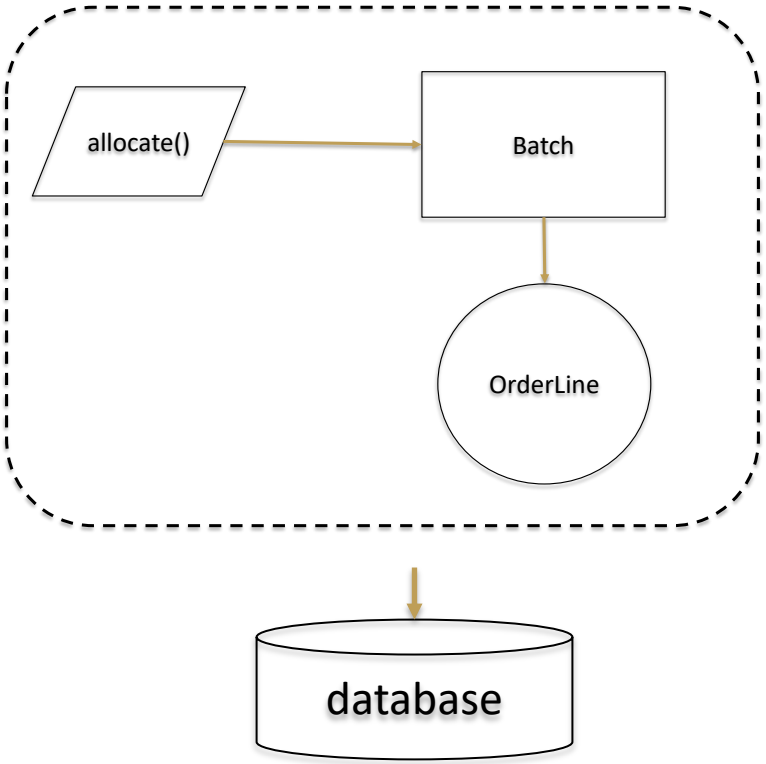


Source: https://www.cosmicpython.com/book/chapter_01_domain_model.html

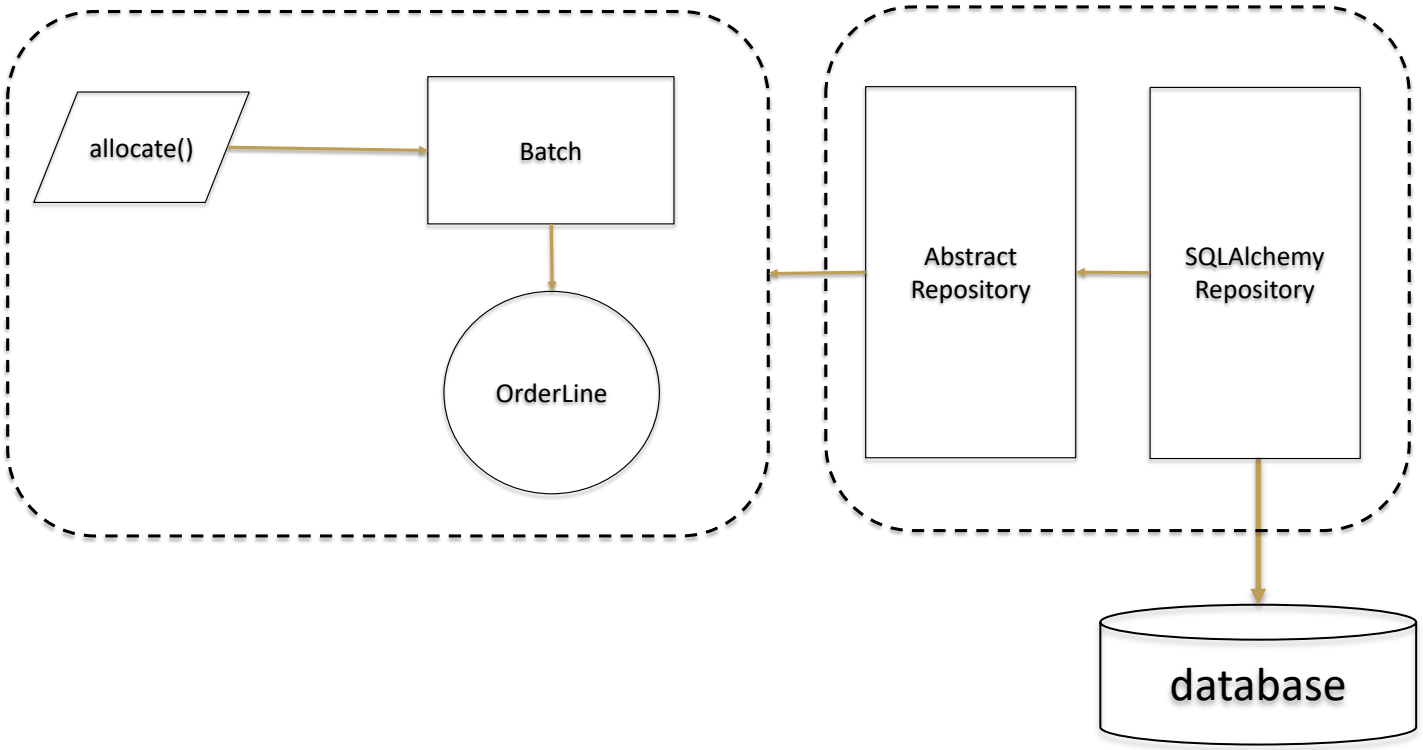
REPOSITORY PATTERN

- Use Dependency Inversion Principle (DIP) to decouple domain model from database storage logic. (decoupling our core logic from infrastructural concerns)
- Repository pattern allowing us to decouple our business layer from the data layer.

Before using the Repository Pattern

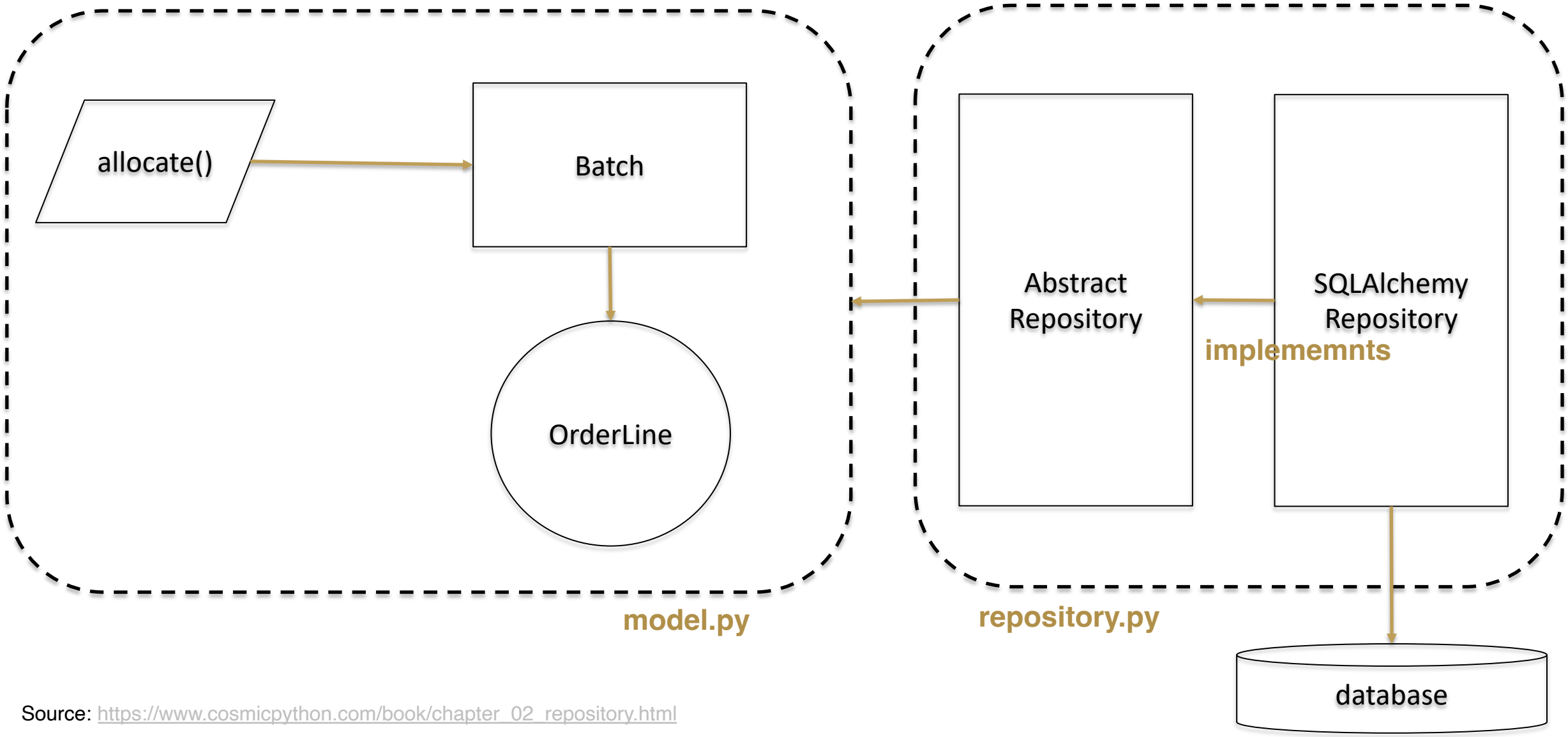


After using the Repository Pattern



Source: https://www.cosmicpython.com/book/chapter_02_repository.html

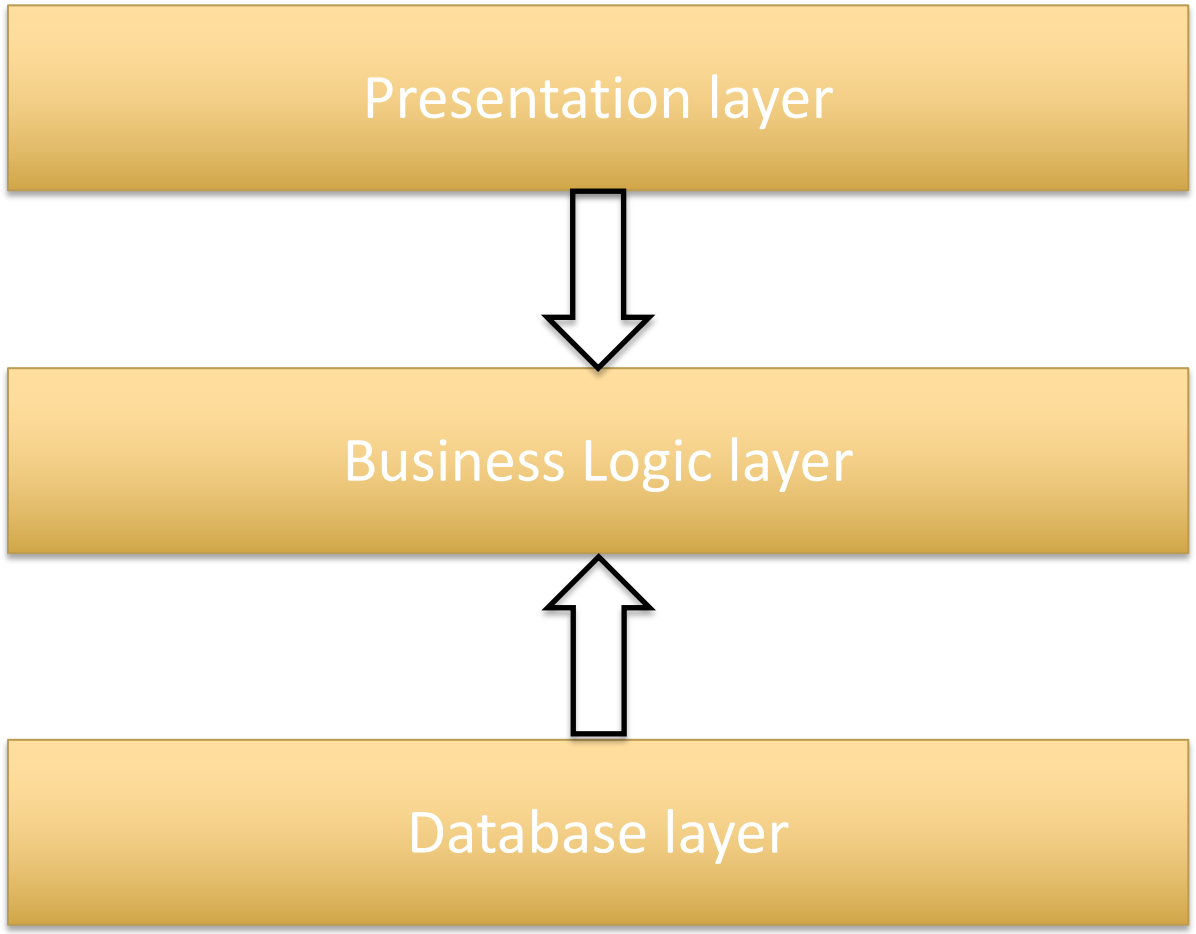
REPOSITORY PATTERN



Source: https://www.cosmicpython.com/book/chapter_02_repository.html

DEPENDENCY INVERSION

- Move from classical layered architecture to onion architecture (or port and adaptors or hexagonal architecture)



- domain model should not have any contact to technical stuff
- domain model doesn't need to know anything about how data is loaded or persisted.

Source: https://www.cosmicpython.com/book/chapter_02_repository.html

ORM SHOUD DEPEND ON MODEL NOT OTHERWISE

Inverting the Dependency

Classical ORM (declarative style)

```
from sqlalchemy import Integer, String,

from sqlalchemy.orm import DeclarativeBase
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import mapped_column

#declarative base
class class Base(DeclarativeBase): pass

# an example mapping using the base
class User(Base):
    __tablename__       = "user"
    id: Mapped[int]     = mapped_column(
                            primary_key=True)
    name: Mapped[str]   = mapped_column(
                            String(50))
    fullname: Mapped[str] = mapped_column(
                            String(30))
```

Source: https://www.cosmicpython.com/book/chapter_02_repository.html

ORM depends on model (imperative style)

```
from sqlalchemy import Table, Column, Integer,
                       String,

from sqlalchemy.orm import registry
from model import User

mapper_registry = registry()

user_table = Table(
                "user",
                mapper_registry.metadata,
                Column("id", Integer,
                       primary_key=True),
                Column("name", String(50)),
                Column("fullname", String(50)),
                )

mapper_registry.map_imperatively(User, user_table)
```

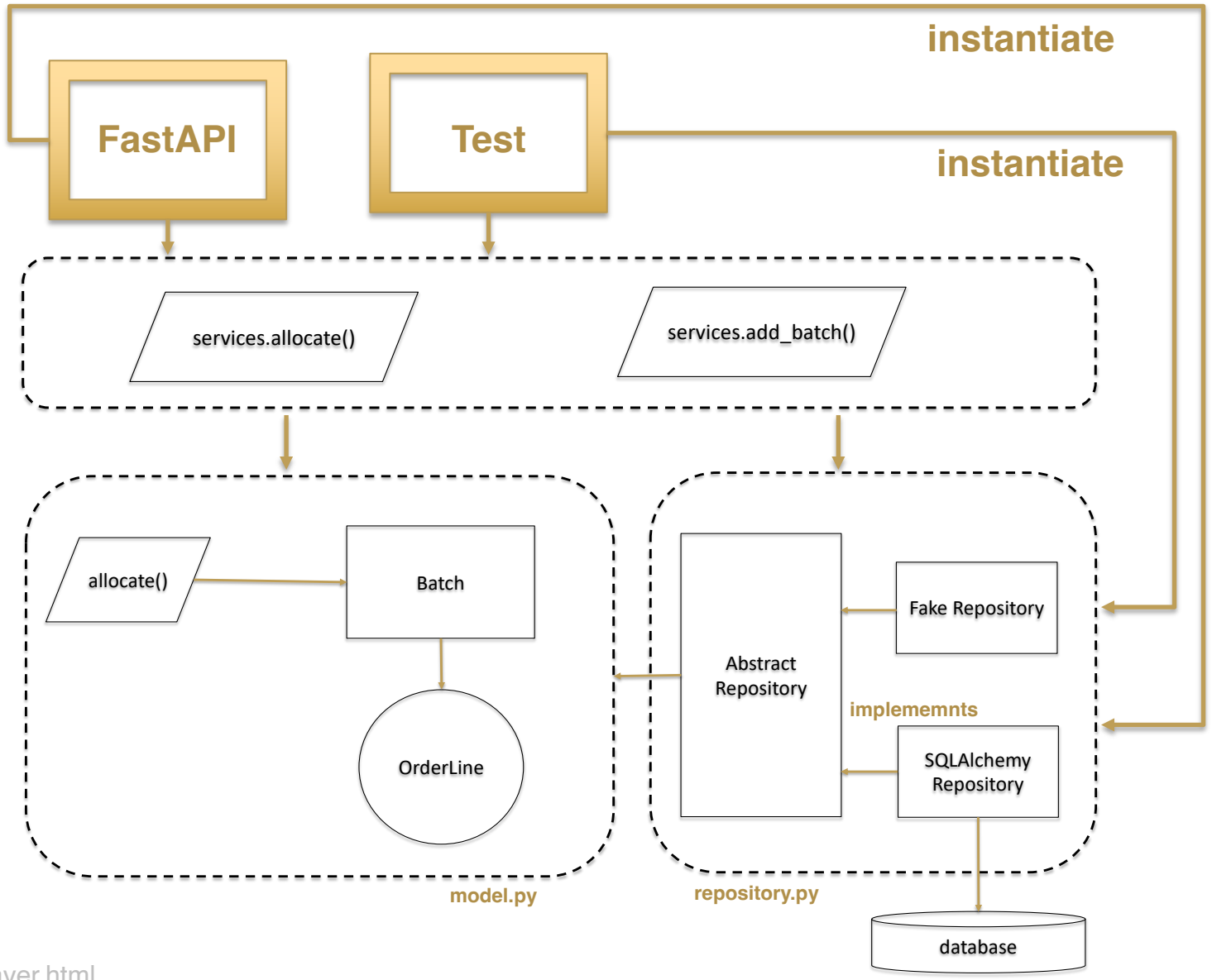
REPOSITORY PATTERN

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

Source: https://www.cosmicpython.com/book/chapter_02_repository.html

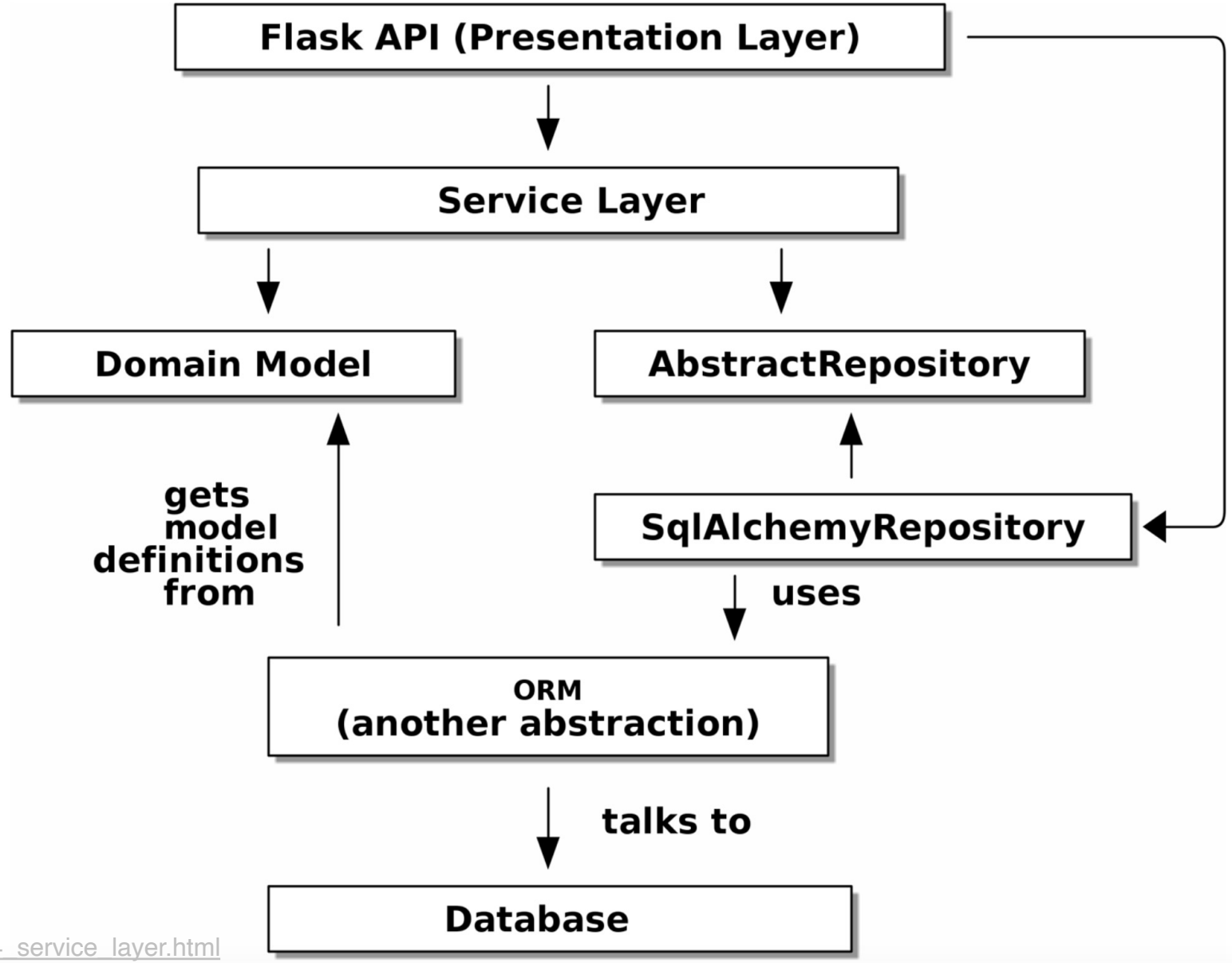
SERVICE LAYER PATTERN

- FastAPI Microservice instantiate SQLAlchemy repository
- Test instantiate Fake repository



Source: https://www.cosmicpython.com/book/chapter_04_service_layer.html

LAYERED ARCHITECTURE



Source: https://www.cosmicpython.com/book/chapter_04_service_layer.html

UNIT OF WORK

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

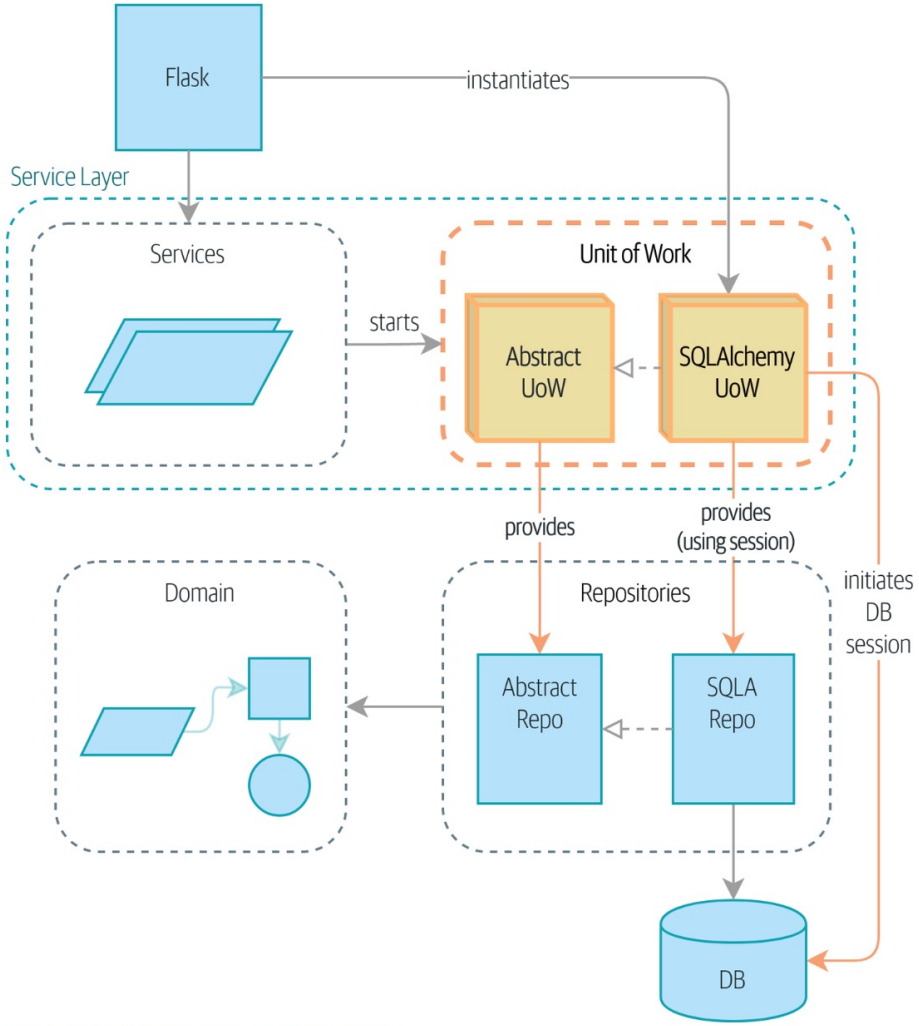
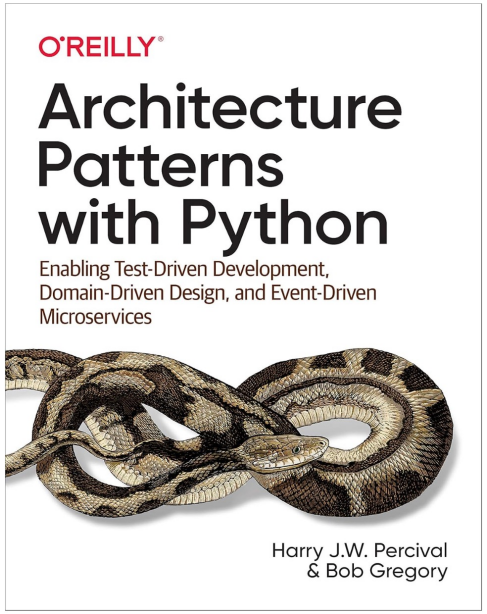


Figure 2. With UoW: UoW now manages database state

FURTHER READING

DDD WITH PYTHON



2020 – Percival & Gregory
"Architecture Patterns with Python"

"covers a lot of modern architecture pattern on top of DDD"