Cosmic Python

Manage Complexity

Wannes Rombouts

Meetup Python Toulouse - 06/12/2023

Don’t do this

… they said.

Technical but not code.

Hopefully practical.

Applications

as opposed to

scripts / libraries

Who’s doing what?

Greenfield / legacy?

Big / small?

Common problems

Where is the business logic?

Abstract-overengineering

Recursive imports

business.py

Move things elsewhere when they start obscuring the logic.

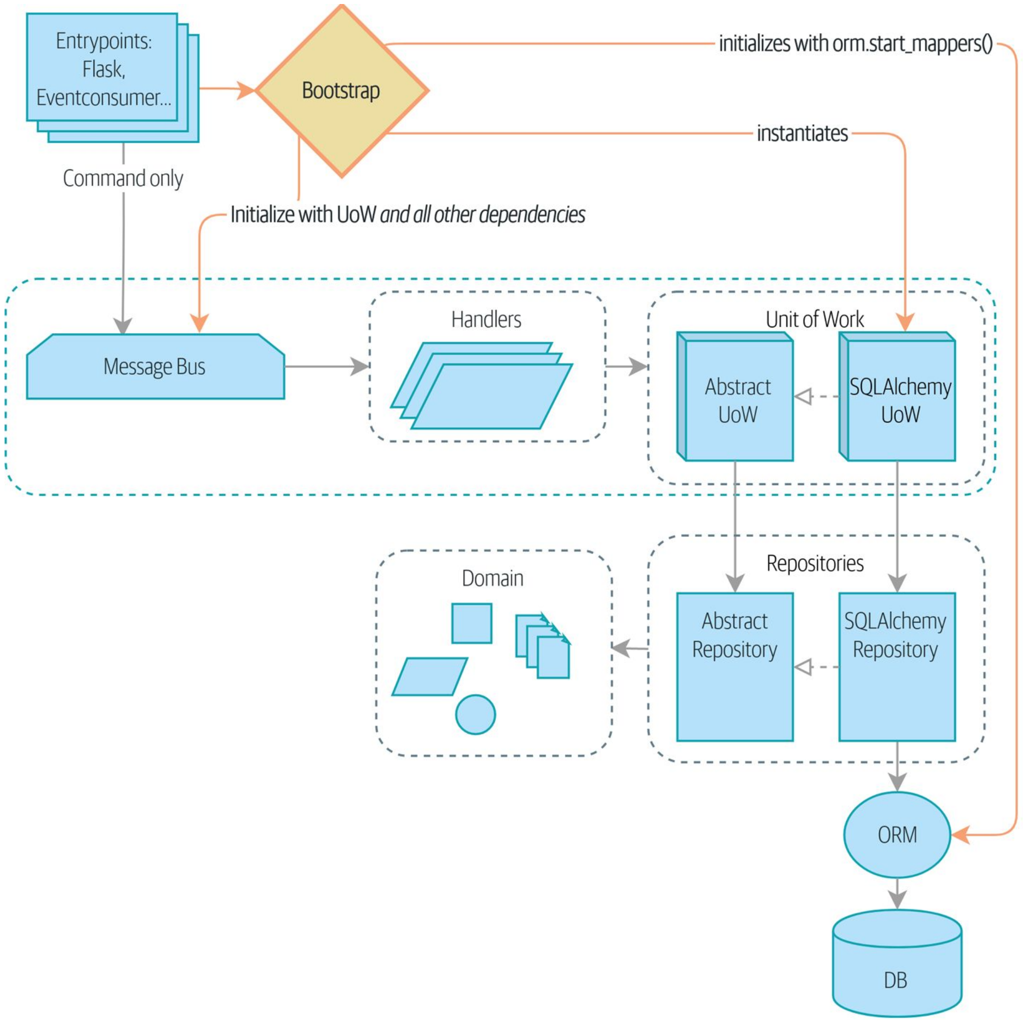
“**Abstract only** what you are **not working** on.”
– Wannes 2016

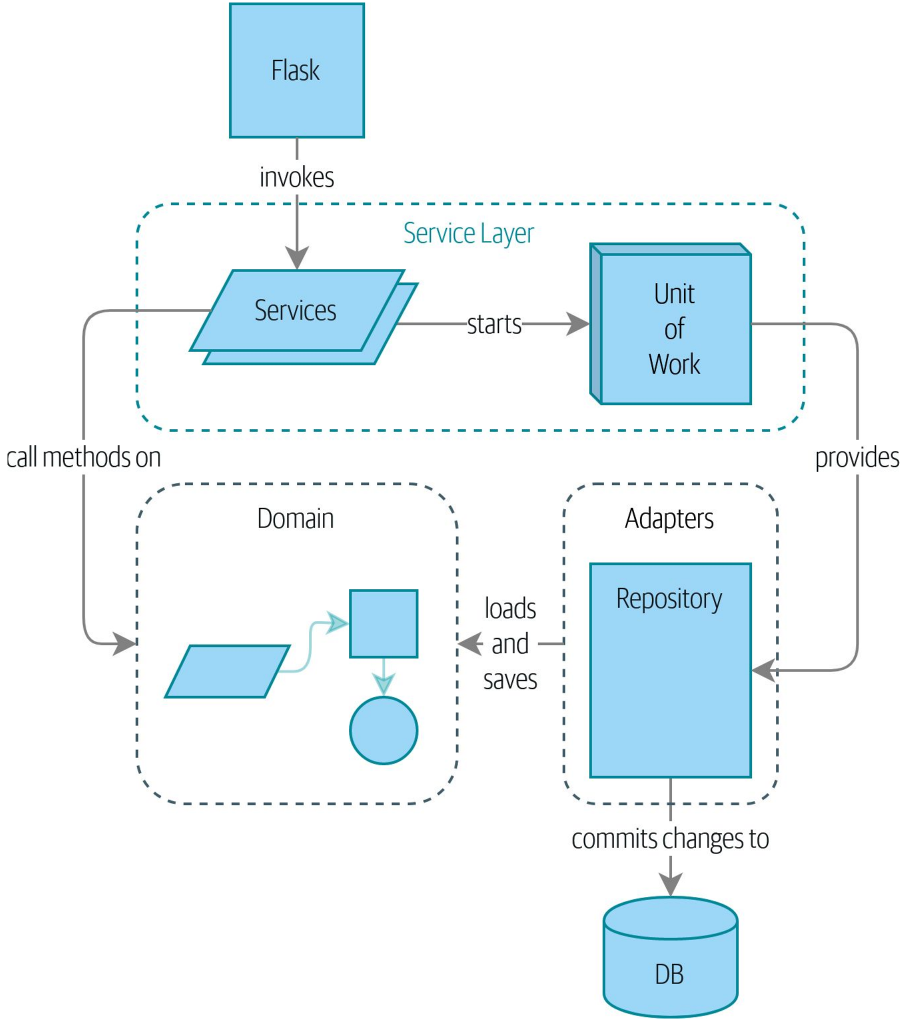
Cosmic Python

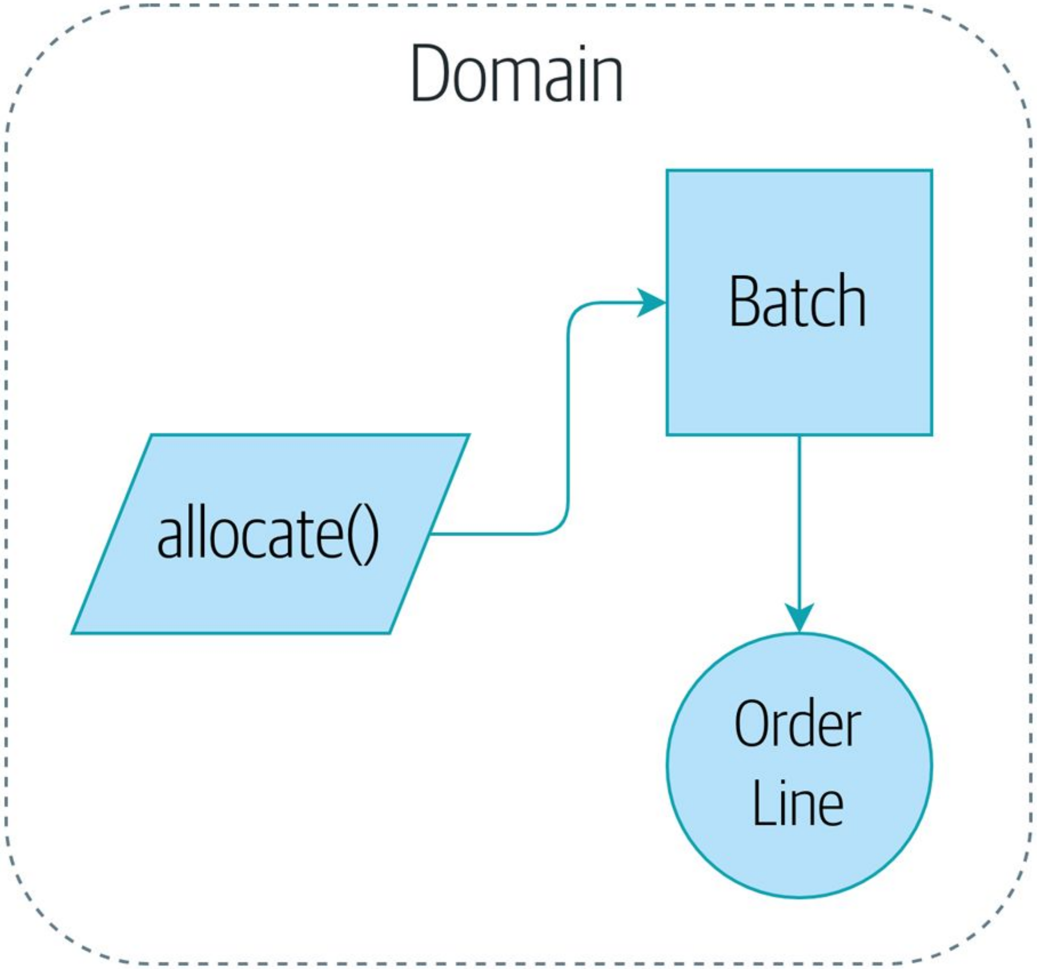
Architecture Patterns With Python

by Harry J.W. Percival & Bob Gregory

2020


Entrypoints:
Flask,
Eventconsumer...
Bootstrap
initializes with orm.start_mappers()
instantiates
Command only
Initialize with UoW *and all other dependencies*
Message Bus
Handlers
Unit of Work
Abstract UoW
SQLAlchemy UoW
Domain
Repositories
Abstract Repository
SQLAlchemy Repository
ORM
DB


Flask
invokes
Service Layer
Services
starts
Unit of Work
call methods on
provides
Domain
Adapters
Repository
loads and saves
commits changes to
DB


Domain
Batch
allocate()
Order Line

Ahah! Moment

DDD, Ports & adapters,
Hexagonal architecture,
Clean architecture,
etc.

business.py

Move things elsewhere when they start obscuring the logic.

1 - Vocab & theory

- Dependency Injection
- Dependency Inversion
- Low level vs. high level logic

2 - better business.py

- Domain modeling
- Working with ORMs
- Repository pattern

3 - orchestration

- What about the real world? Flask.
- Example of dependency inversion

4 - (maybe) event driven

- Why would we do this?
- What is a message bus?
- How far can we push this?

Vocab & theory

Dependencies

importing, inheriting, using

“Knowing about”

Encapsulation
Abstraction

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

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

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

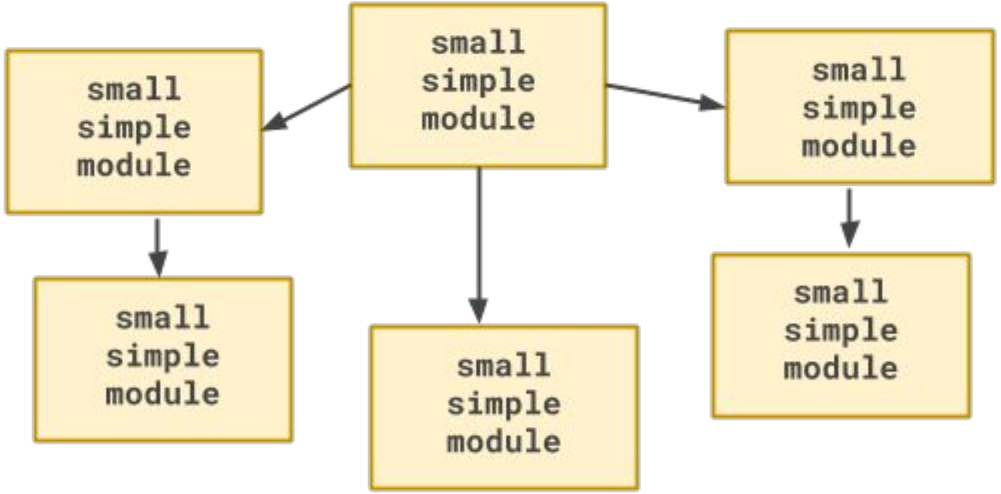
```python
import requests

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

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

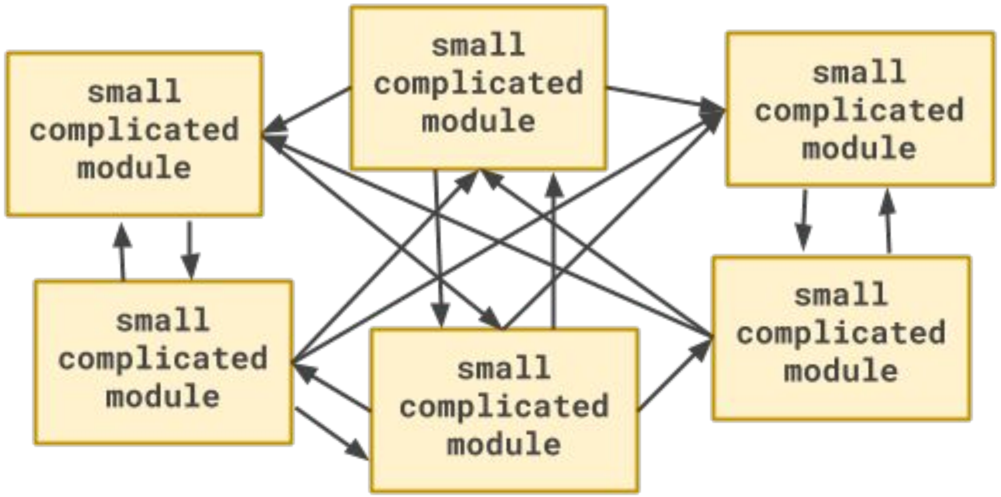
```python
import duckduckpy

for r in duckduckpy.query('Sausages').related_topics:
      print(r.first_url, ' - ', r.text)
```

Modularity

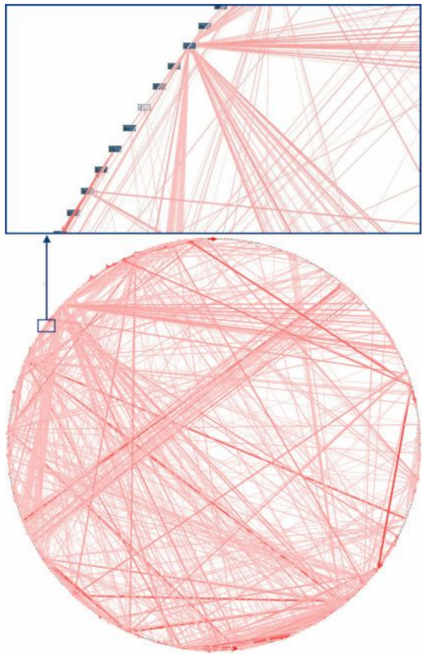
BIG
COMPLICATED
SYSTEM

We want modularity,
we apply **separation of concerns**

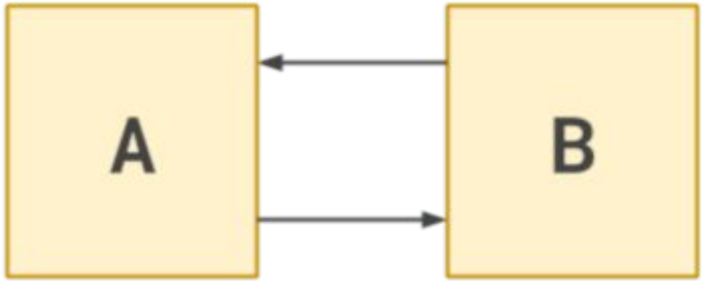

small simple module
small simple module
small simple module
small simple module
small simple module
small simple module

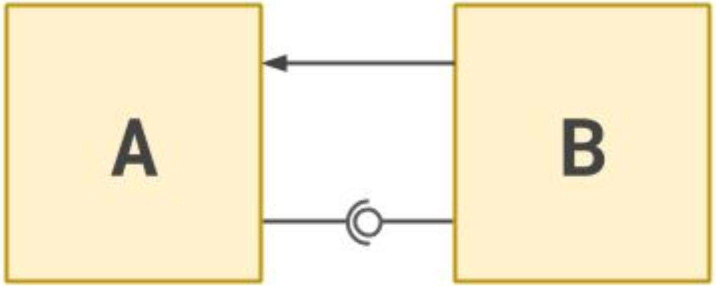


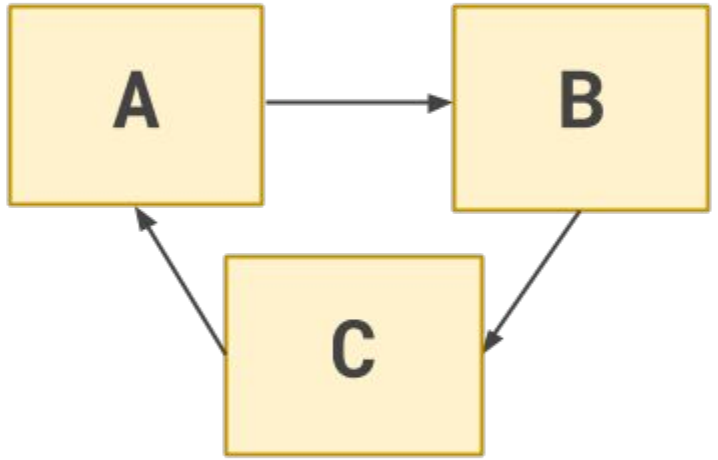
BIG COMPLICATED SYSTEM

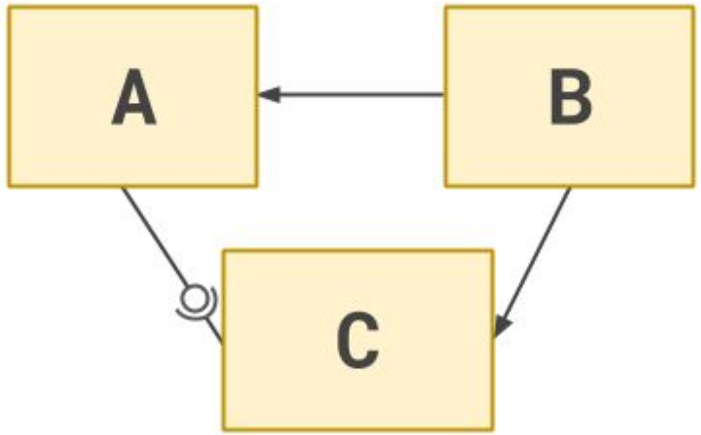


Inverting Control


A
B


A
B


A
B
C

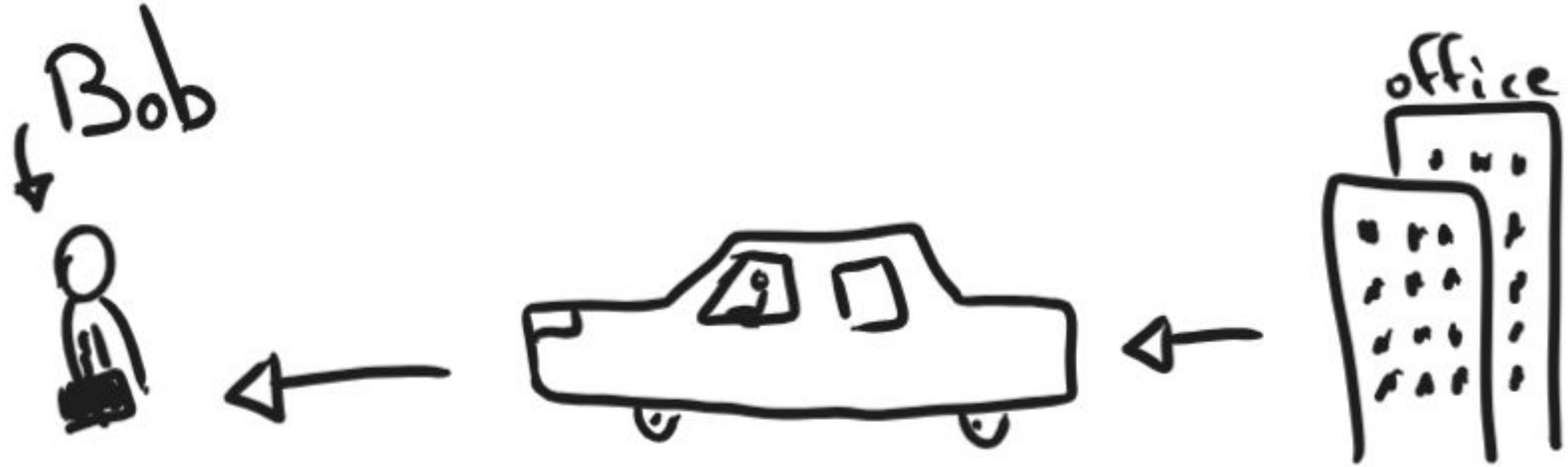

A
B
C

A simple problem

Bob goes to work

Bob
TAXI
office

Bob needs to find a car



Work sends a car to bob

Dependency Injection

Separate **Creation** and **Use,**
one class should not do both.

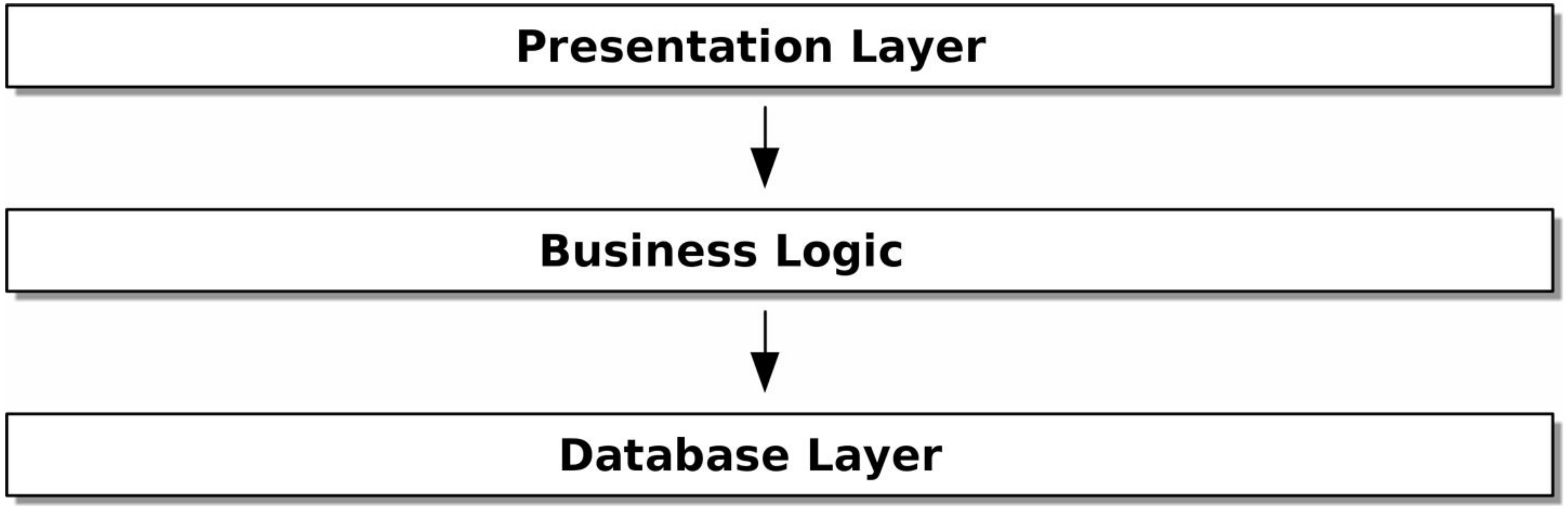
Bob uses the car, he doesn’t own it.

Dependency Inversion

High level modules should not depend on low level modules.

High level - business

Low level - tech

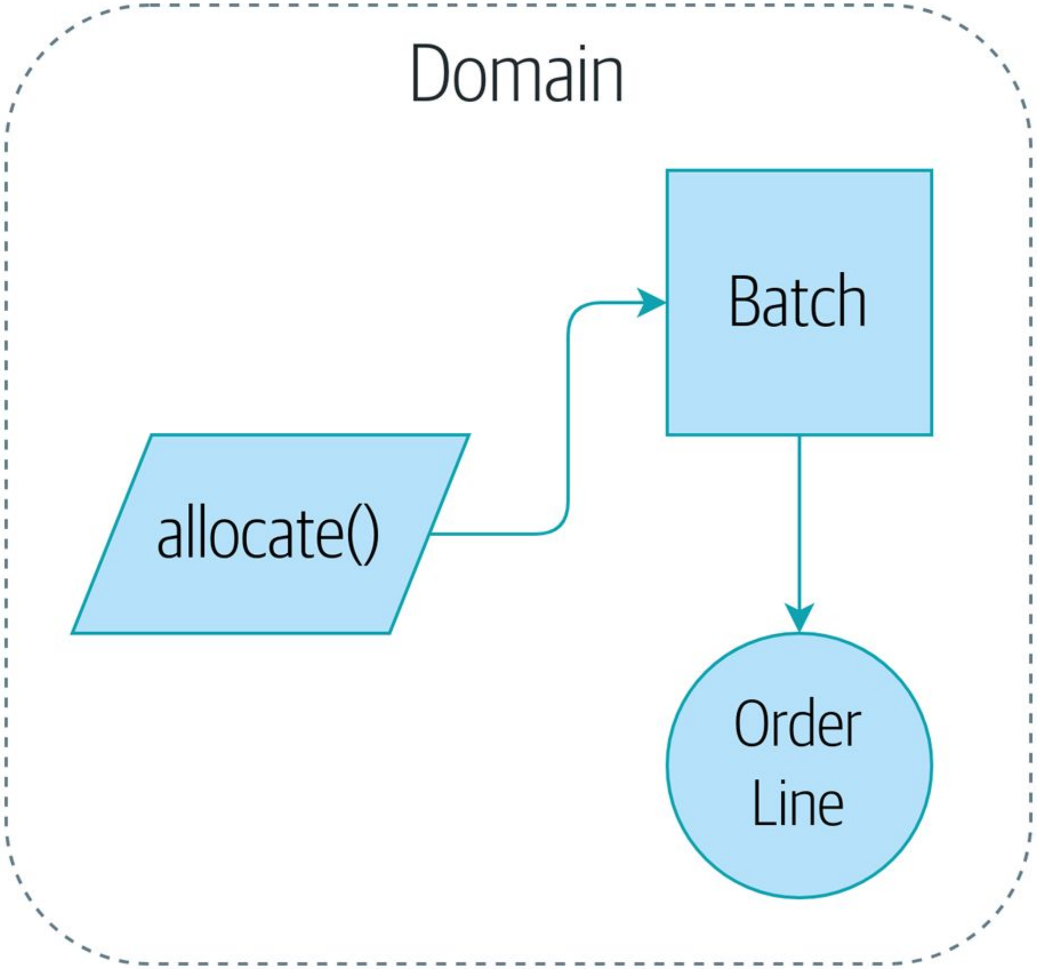

Presentation Layer
Business Logic
Database Layer

2 - better business.py

- Domain modeling
- Working with ORMs
- Repository pattern

KISS business logic

A simple domain without dependencies

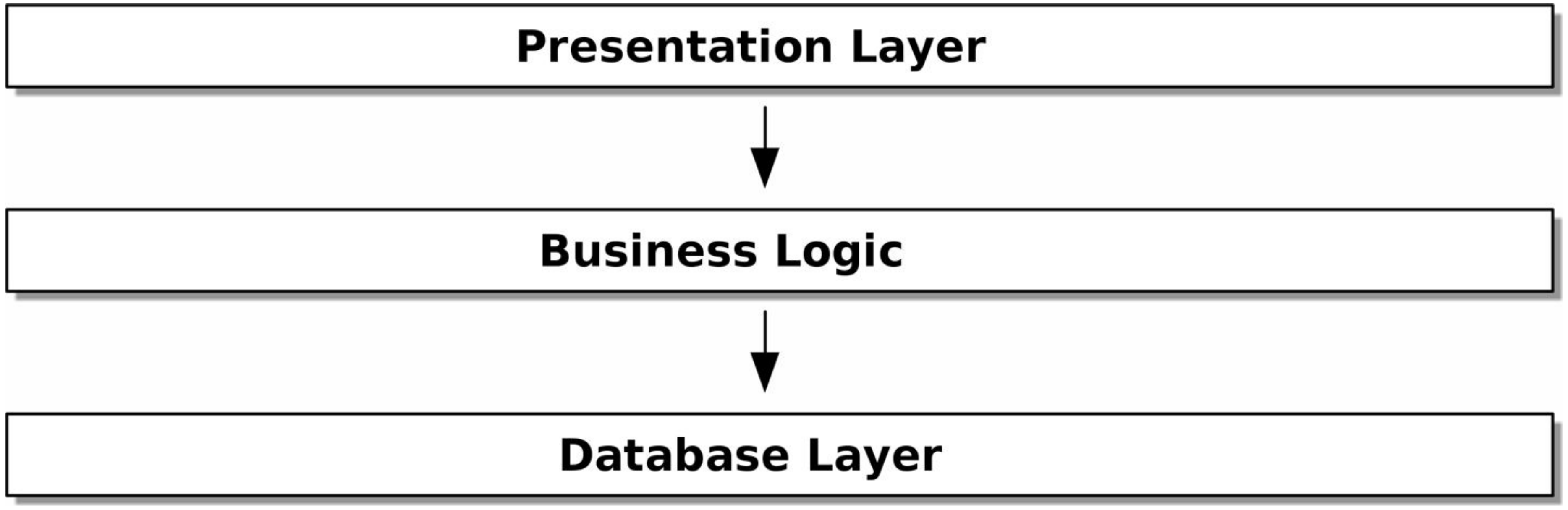

Domain
Batch
allocate()
Order
Line

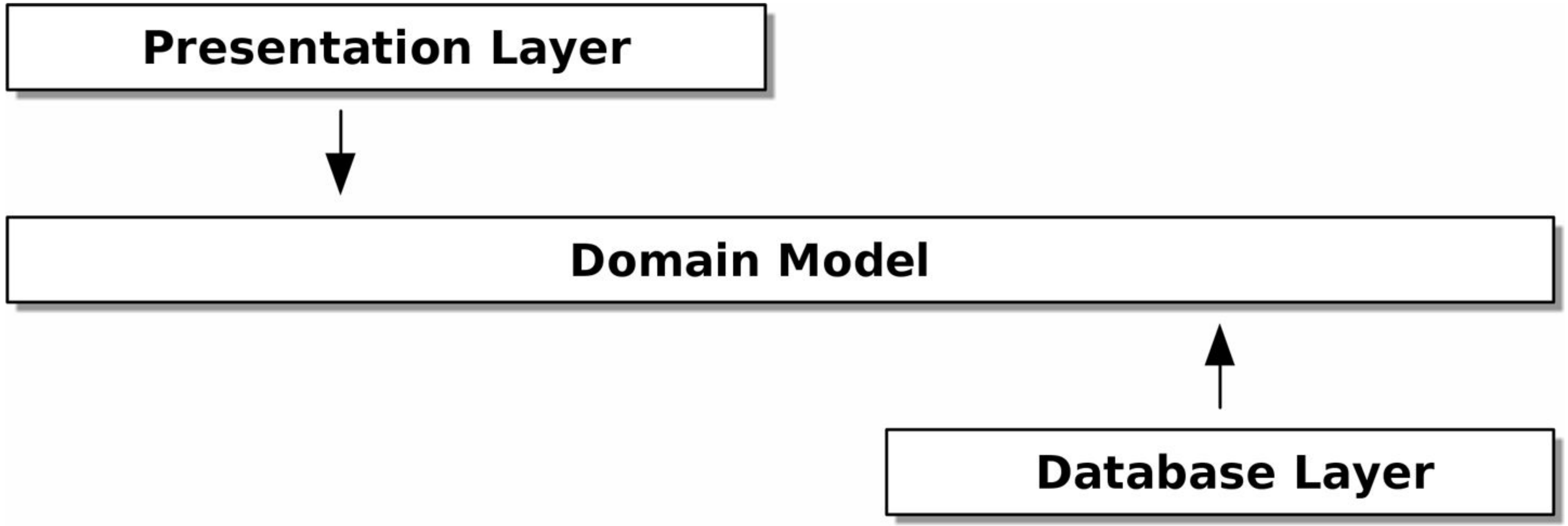
```python
@dataclass(frozen=True)
class OrderLine:
    orderid: OrderReference
    sku: ProductReference
    qty: Quantity

class Batch:
    def __init__(
        self, ref: str, sku: str, qty: int
    ):
        self.reference = ref
        self.sku = sku
        self.available_quantity = qty

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

How do we store this?

A simple domain without dependencies

```
@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
```


Presentation Layer
Business Logic
Database Layer


Presentation Layer
Domain Model
Database Layer

“Normal” ORM

Model depends on database
Django, typical SQLAlchemy, etc.

```python
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)
```

Inverting the dependency

Have the ORM depend
on the model.

```
from sqlalchemy.orm import registry

import model

mapper_registry = registry()

order_lines = Table(  #(2)
    "order_lines",
    mapper_registry.metadata,
    Column("id", Integer, primary_key=True, autoincrement=True),
    Column("sku", String(255)),
    Column("qty", Integer, nullable=False),
    Column("orderid", String(255)),
)

mapper_registry.map_imperatively(model.OrderLine,order_lines)
```

It’s harder with django.

Back to flask !

```python
@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
```

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

    # extract order line from request
    line = OrderLine(request.params, ...)

    # 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
```

Repository Pattern

Why do we need databases?

```python
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)
```

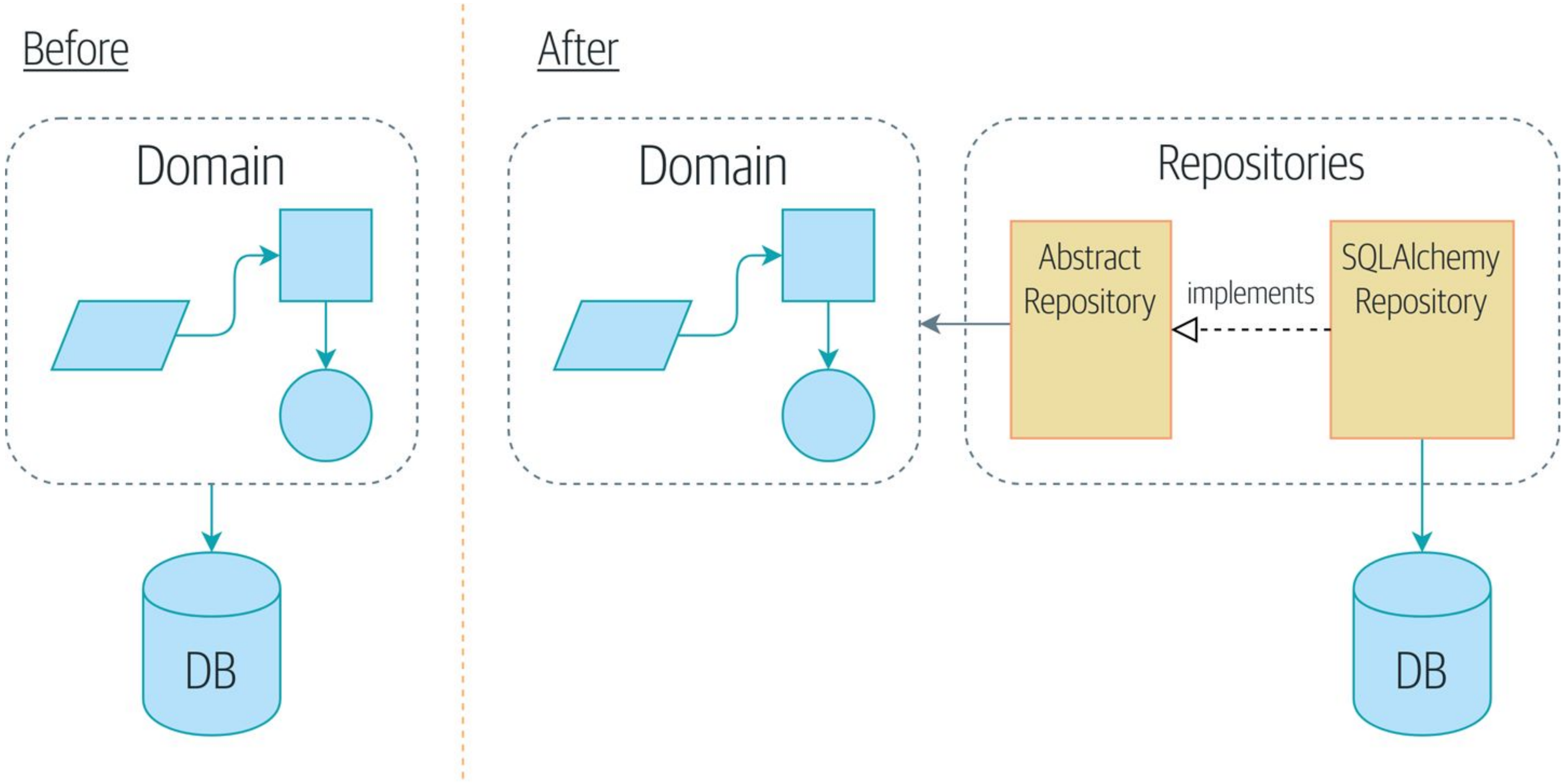
Infinite memory is nice :-)

But no joke

```python
class FakeRepository(AbstractRepository):

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

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

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

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

SqlAlchemyRepository

```
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()
```


Before
Domain
DB
After
Domain
Repositories
Abstract Repository
implements
SQLAlchemy Repository
DB

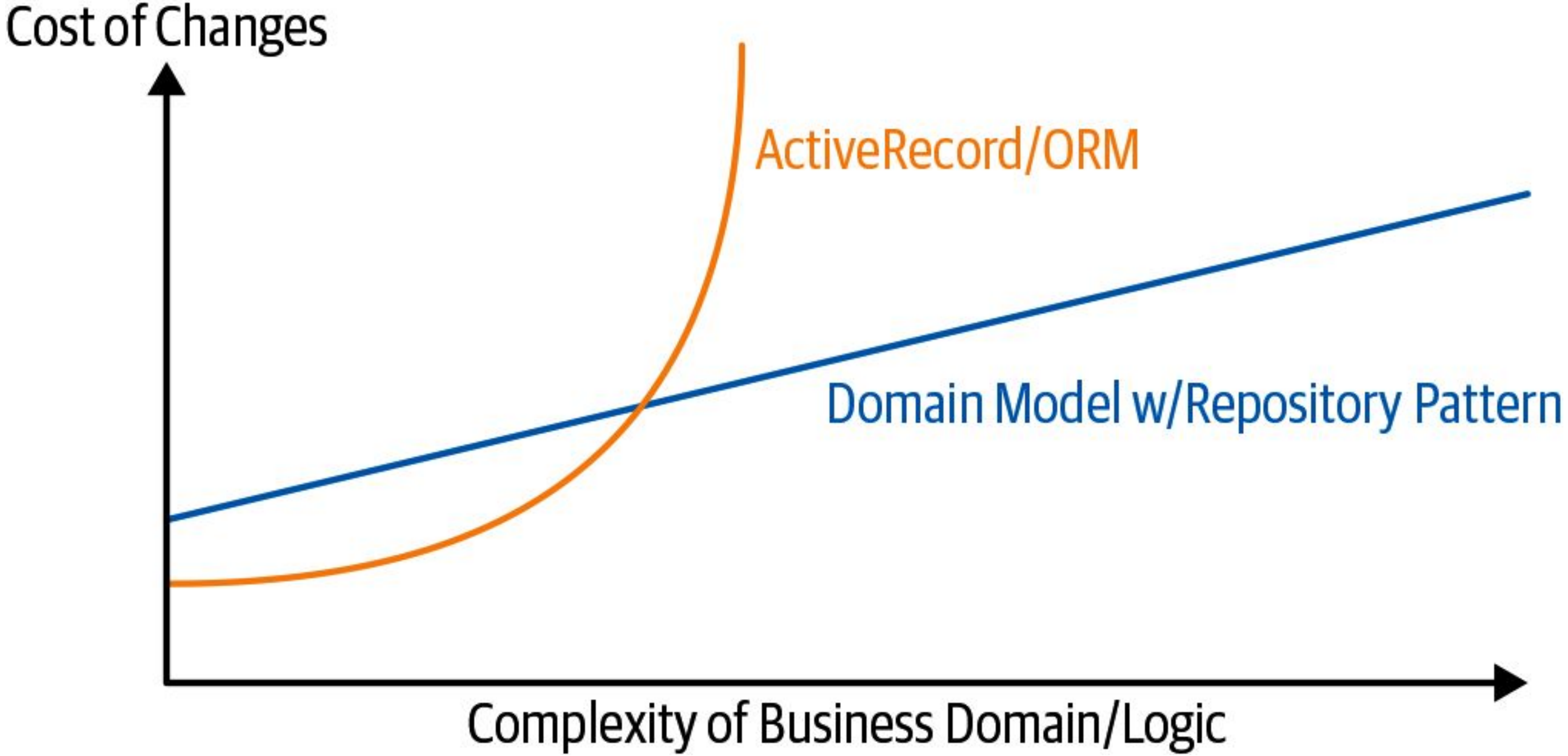
Pro / con

Pro:

- Simple interface
- Easy to Fake
- Focus on business first, storage second
- Simple schema

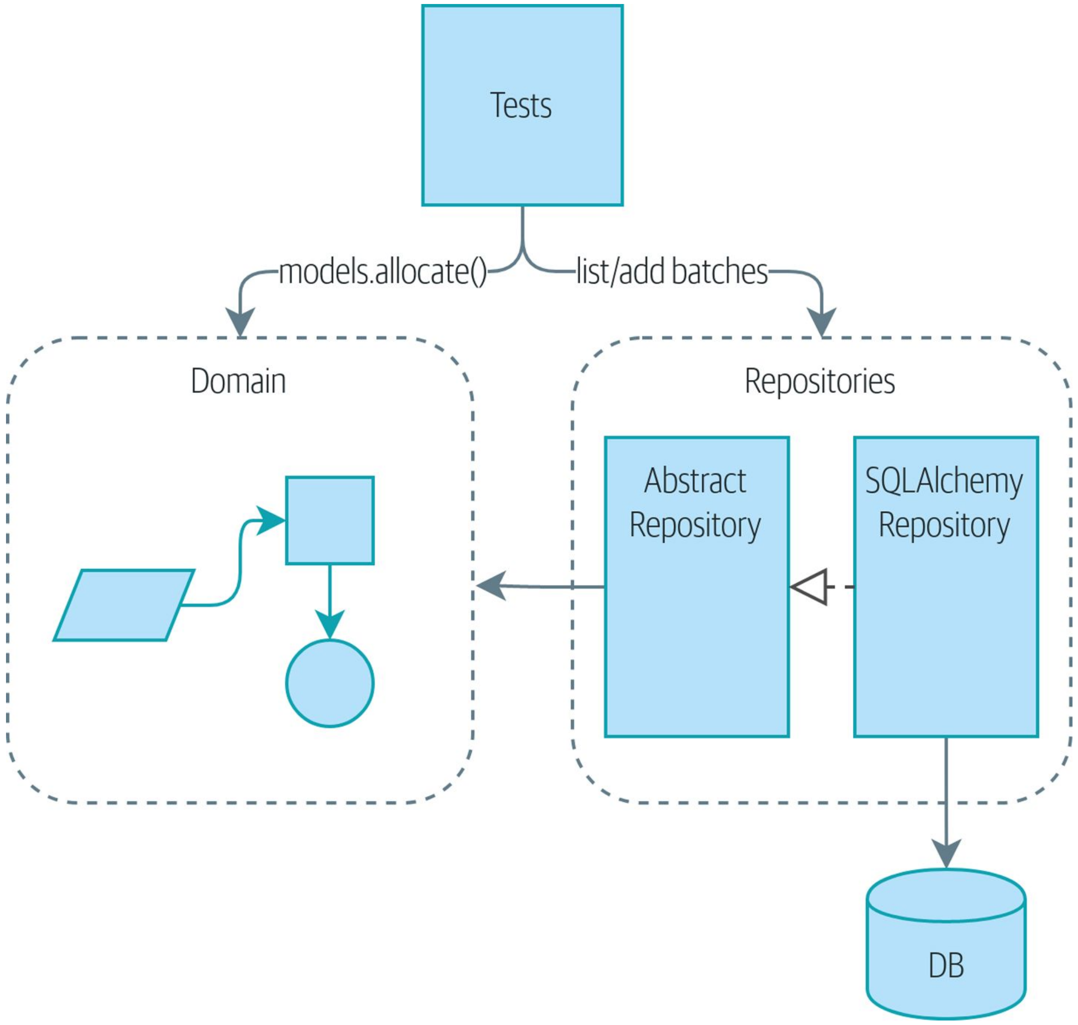
Con:

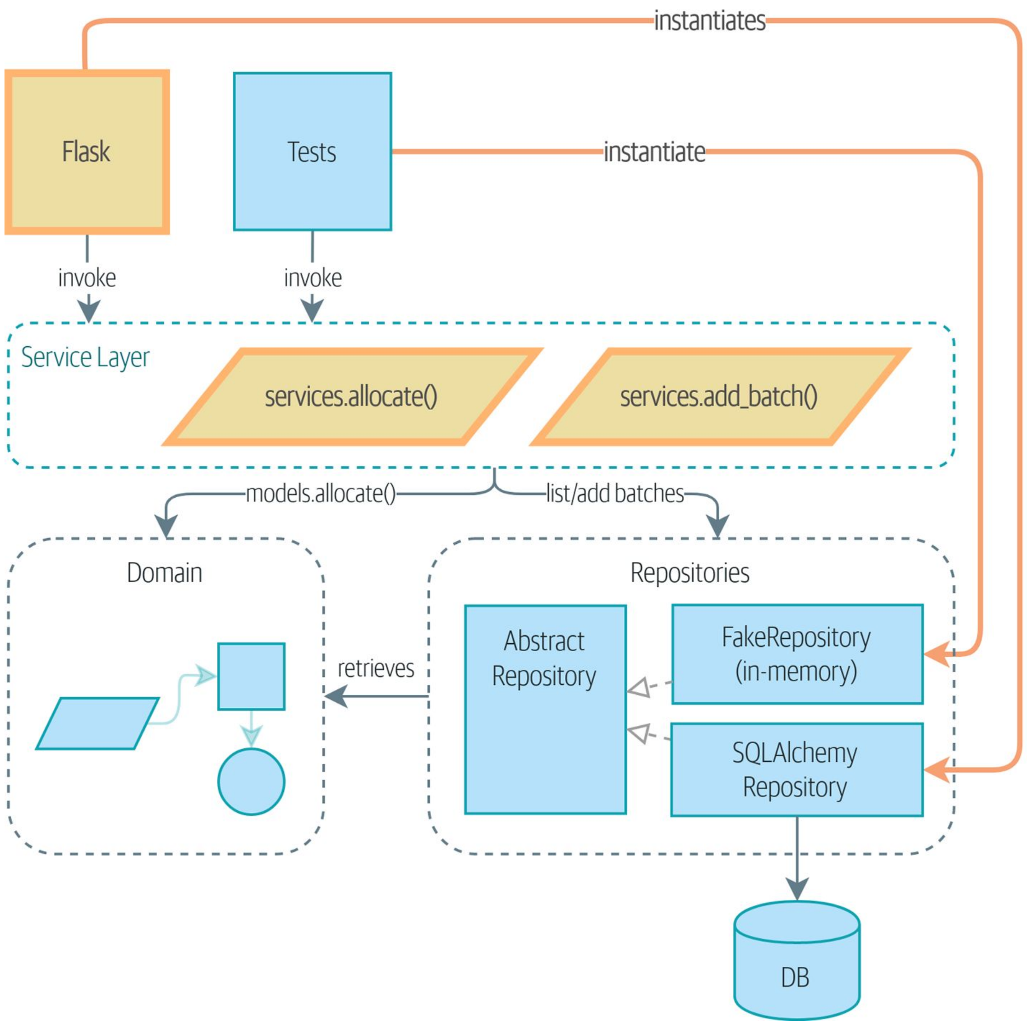
- ORM already gives decoupling
- Doing ORM mappings by hand is extra work and extra code
- WTF – maintenance costs, onboarding


Cost of Changes
ActiveRecord/ORM
Domain Model w/Repository Pattern
Complexity of Business Domain/Logic

3 - orchestration

- What about the real world? Flask.
- Example of dependency inversion
- How to manage transactions


Tests
models.allocate()
list/add batches
Domain
Repositories
Abstract Repository
SQLAlchemy Repository
DB


instantiates
Flask
Tests
instantiate
invoke
invoke
Service Layer
services.allocate()
services.add_batch()
models.allocate()
list/add batches
Domain
Repositories
retrieves
Abstract Repository
FakeRepository (in-memory)
SQLAlchemy Repository
DB

Business logic (domain)
Interfacing logic (repos)

Orchestration logic (service)

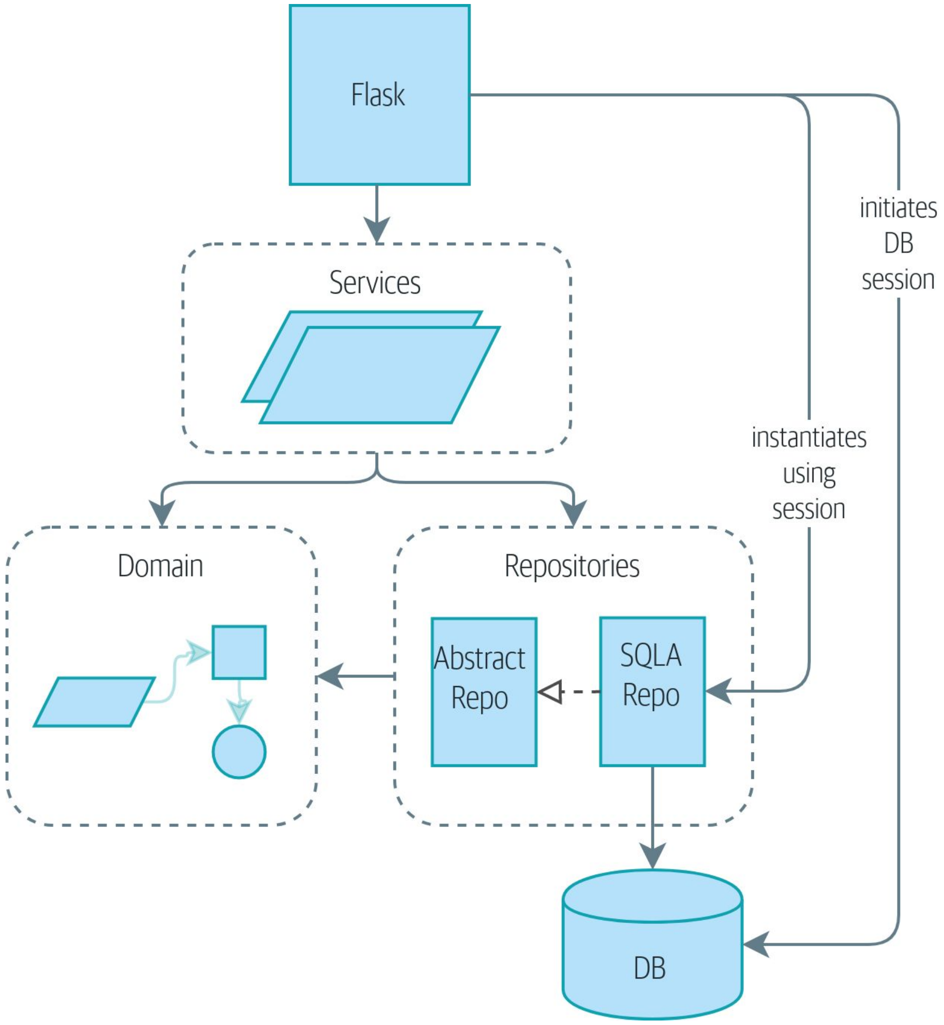
Orchestration, use-cases

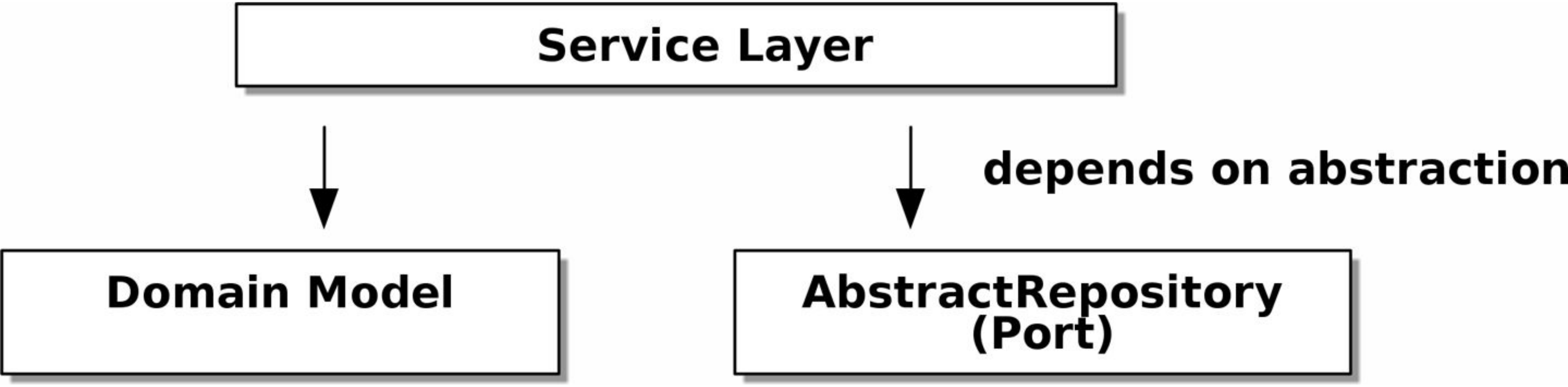
Drives the app.
Called “application layer”
in DDD, Eric Evans.

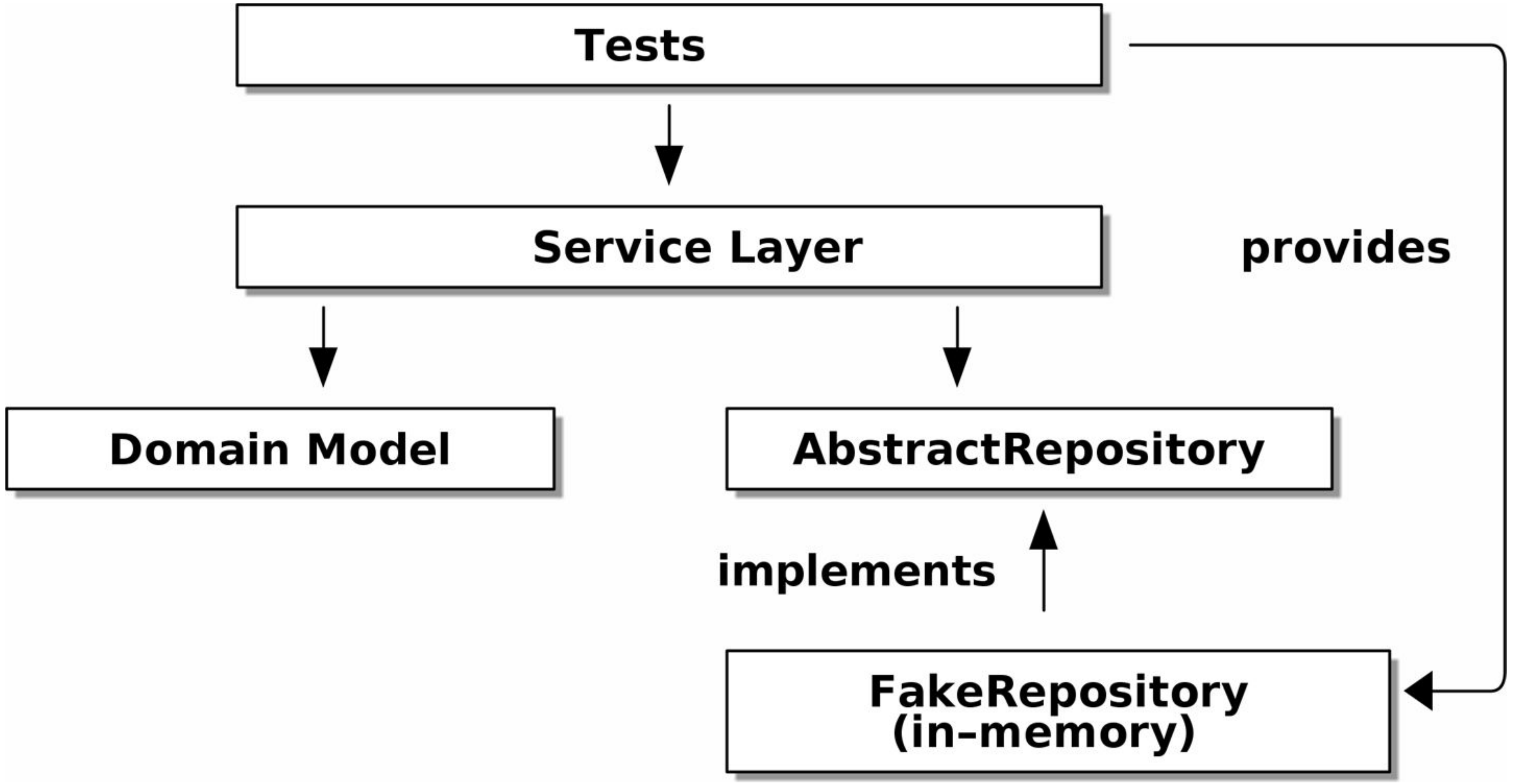
```python
class InvalidSku(Exception):
    pass


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


def allocate(
    line: OrderLine, repo: AbstractRepository, session
) -> str:
    batches = repo.list()  #(1)
    if not is_valid_sku(line.sku, batches):  #(2)
        raise InvalidSku(f"Invalid sku {line.sku}")
    batchref = model.allocate(line, batches)  #(3)
    session.commit()  #(4)
    return batchref
```

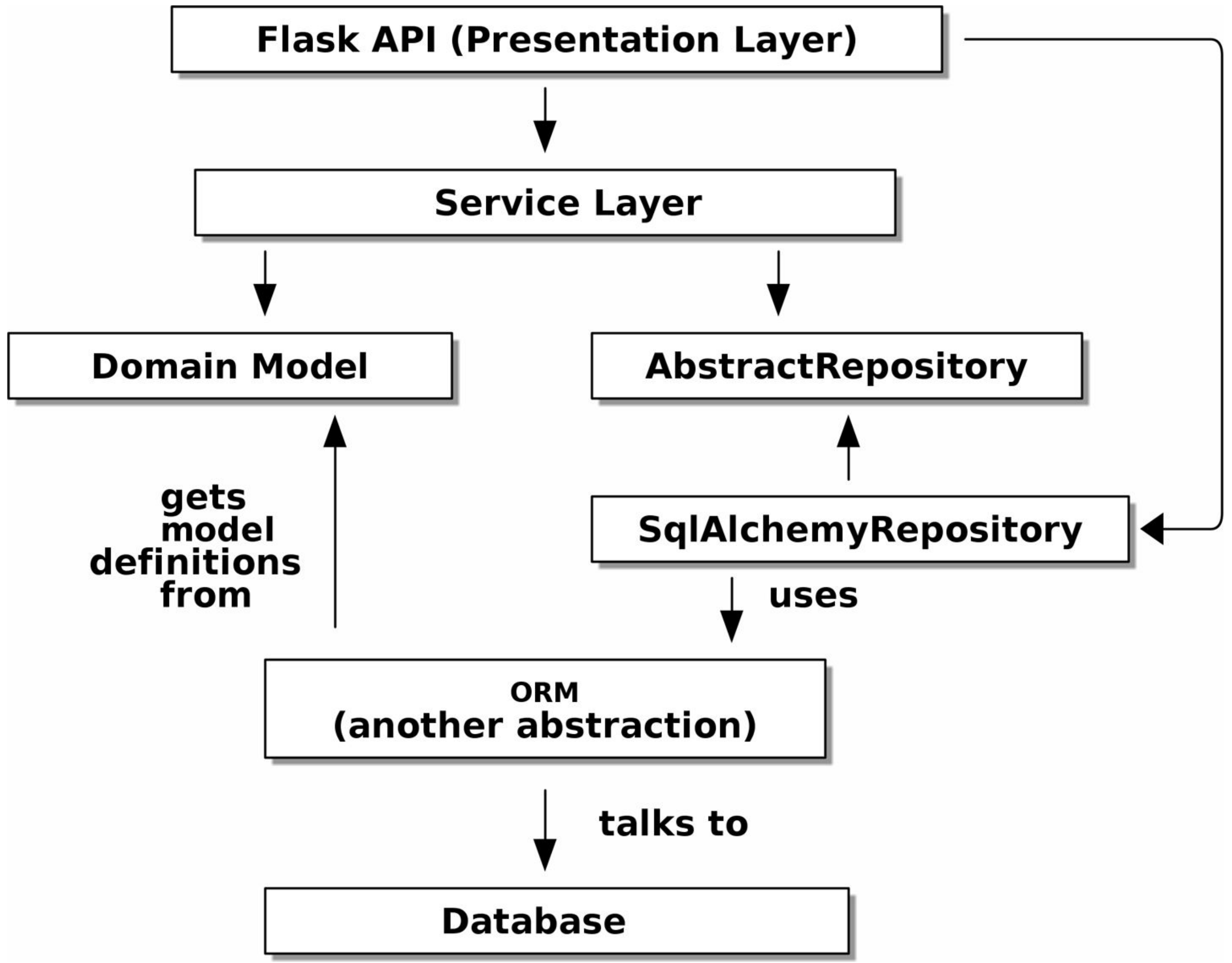
Both flask, CLI, other :

- 1. Fetch from DB
- 2. Make some checks
- 3. Call Domain
- 4. Save / update


Flask
Services
Domain
Repositories
Abstract Repo
SQLA Repo
DB
instantiates using session
initiates DB session


Service Layer
depends on abstraction
Domain Model
AbstractRepository (Port)


Tests
Service Layer
provides
Domain Model
AbstractRepository
implements
FakeRepository
(in-memory)


Flask API (Presentation Layer)
Service Layer
Domain Model
AbstractRepository
gets model definitions from
SqlAlchemyRepository
uses
ORM
(another abstraction)
talks to
Database

Pro / con

Pro:

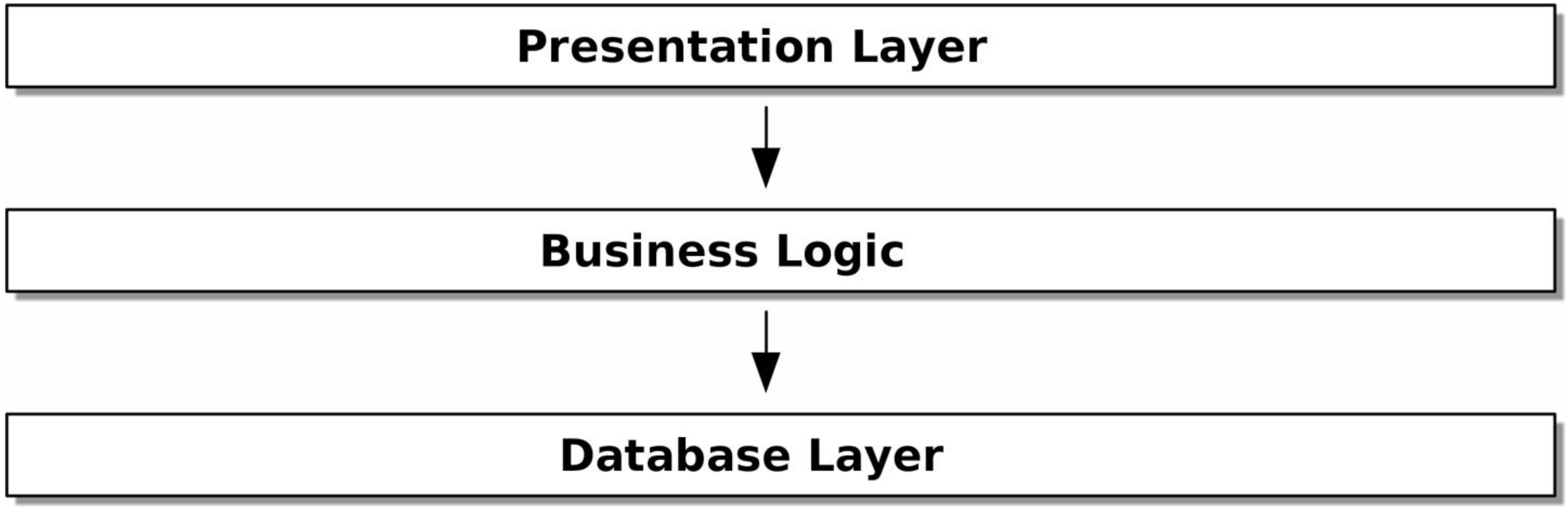
- Single place for use cases
- Domain logic is behind an “API” (services)
- Separate HTTP stuff from Domain stuff
- Services + FakeRepository == easy tests

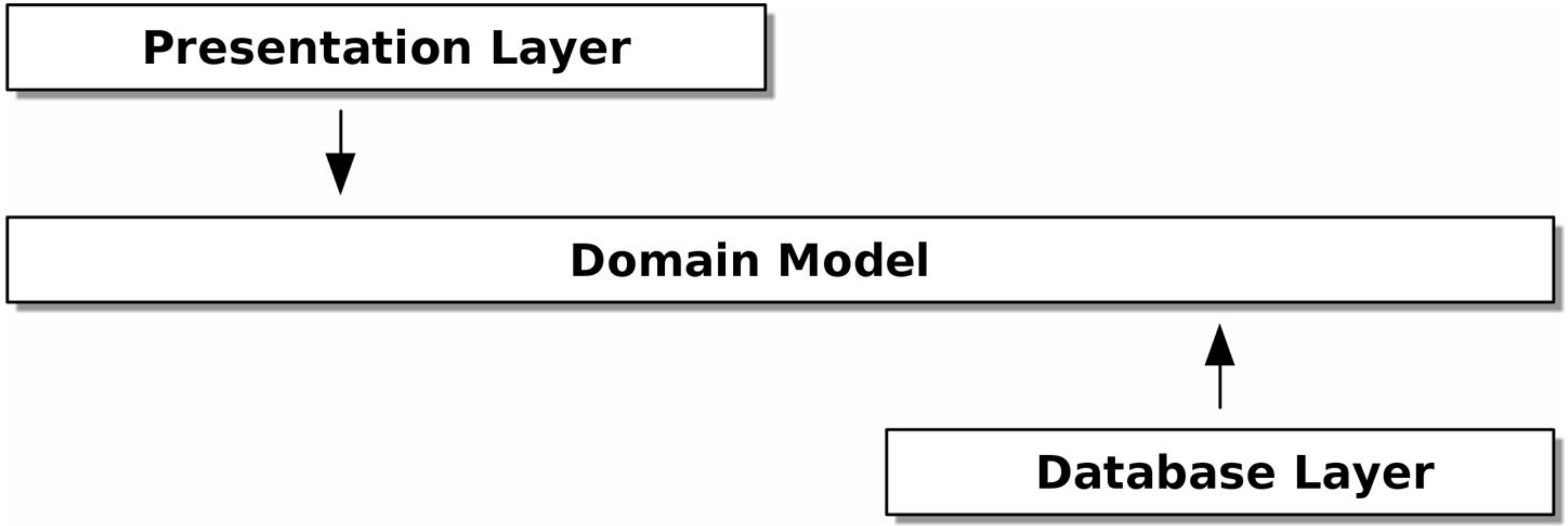
Con:

- Controllers/views are good enough :)
  IF your app is ONLY web. (no cli, no manual access, no notebooks, etc.)
- It’s another layer of abstraction
- It’s a trap : logic tends to end up in the service layer instead of the domain layer.

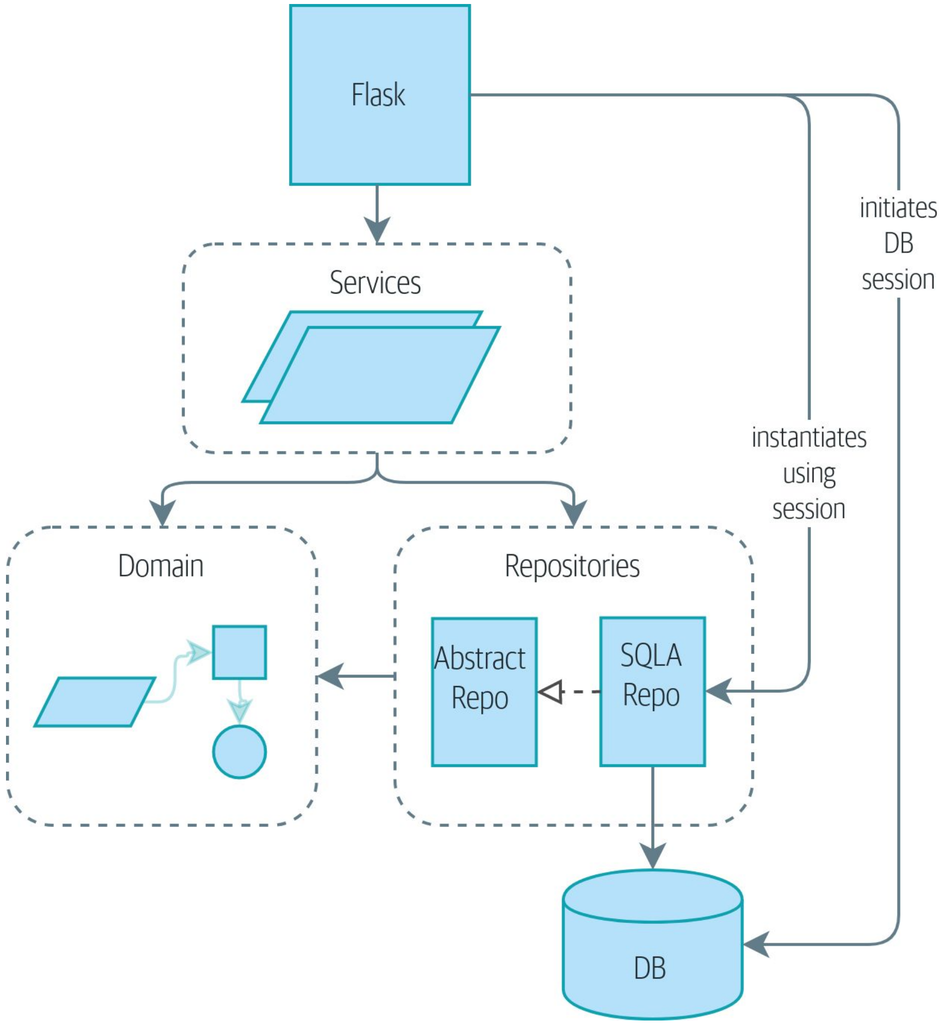
(my) **Main take-away**

A domain without dependencies
is super fun to work with.


Presentation Layer
Business Logic
Database Layer


Presentation Layer
Domain Model
Database Layer

Questions?


Flask
initiates DB session
Services
instantiates using session
Domain
Repositories
Abstract Repo
SQLA Repo
DB

4 - (maybe) event driven

- Why would we do this?
- What is a message bus?
- How far can we push this?

Send an email if out of stock.

Flask ?
service layer ?
domain model ?

Flask

```python
@app.route("/allocate", methods=["POST"])
def allocate_endpoint():
    line = model.OrderLine(
        request.json["orderid"],
        request.json["sku"],
        request.json["qty"],
    )
    try:
        uow = unit_of_work.SqlAlchemyUnitOfWork()
        batchref = services.allocate(line, uow)
    except (model.OutOfStock, services.InvalidSku) as e:
        send_mail(
            "out of stock",
            "stock_admin@made.com",
            f"{line.orderid} - {line.sku}"
        )
        return {"message": str(e)}, 400

    return {"batchref": batchref}, 201
```

Domain

```
 def allocate(self, line: OrderLine) -> str:
     try:
         batch = next(b for b in sorted(self.batches) if
b.can_allocate(line))
         #...
     except StopIteration:
         email.send_mail("stock@made.com", f"XXX {line.sku}")
         raise OutOfStock(f"Out of stock for sku {line.sku}")
```

Service

```python
def allocate(
    orderid: str, sku: str, qty: int,
    uow: unit_of_work.AbstractUnitOfWork,
) -> str:
    line = OrderLine(orderid, sku, qty)
    with uow:
        product = uow.products.get(sku=line.sku)
        if product is None:
            raise InvalidSku(f"Invalid sku {line.sku}")
        try:
            batchref = product.allocate(line)
            uow.commit()
            return batchref
        except model.OutOfStock:
            email.send_mail("stock@made.com", f"XXX{line.sku}")
            raise
```

We are trying to do two things:
allocate_and_send_email_if_xxx()

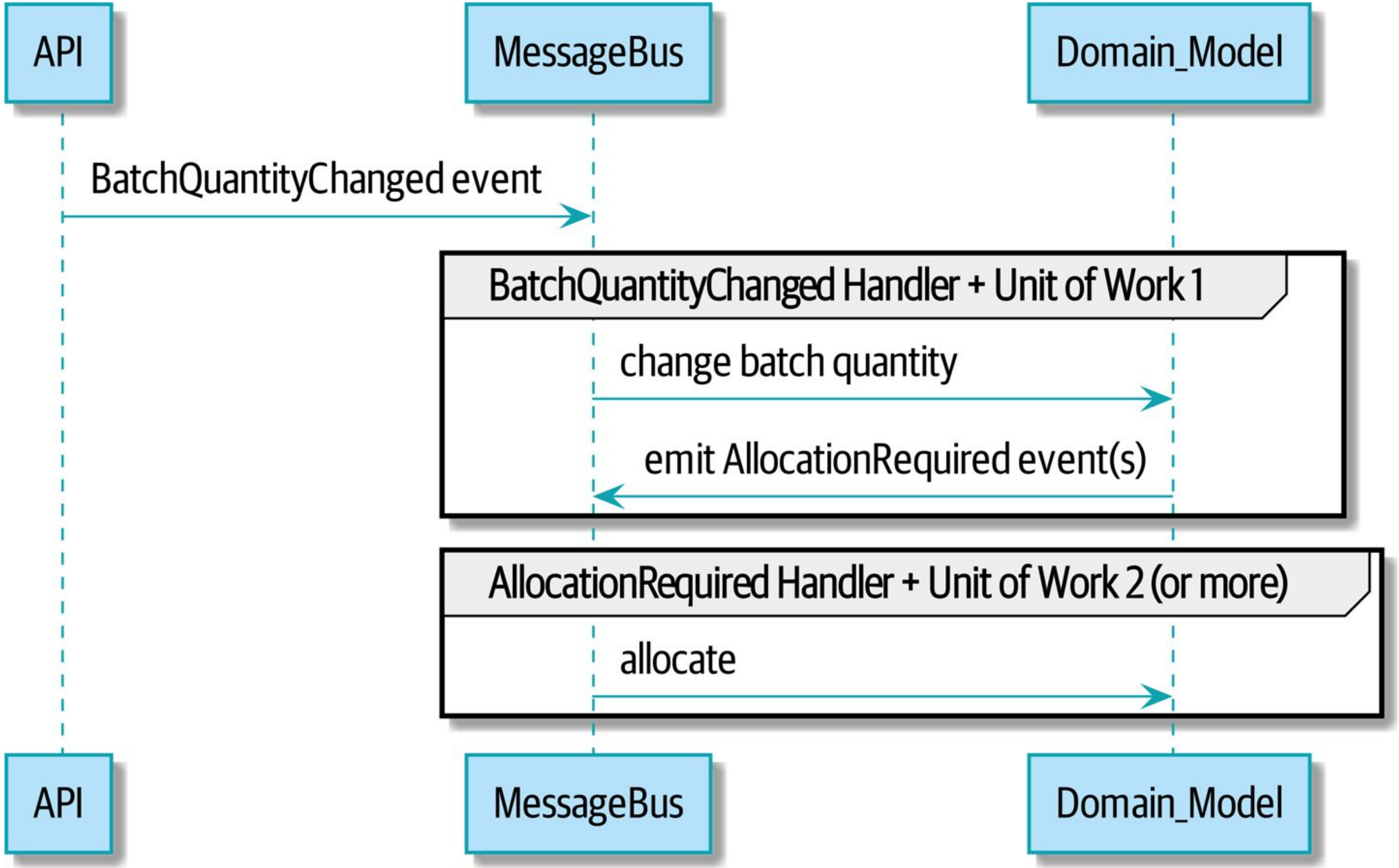
We need to “re-enter”

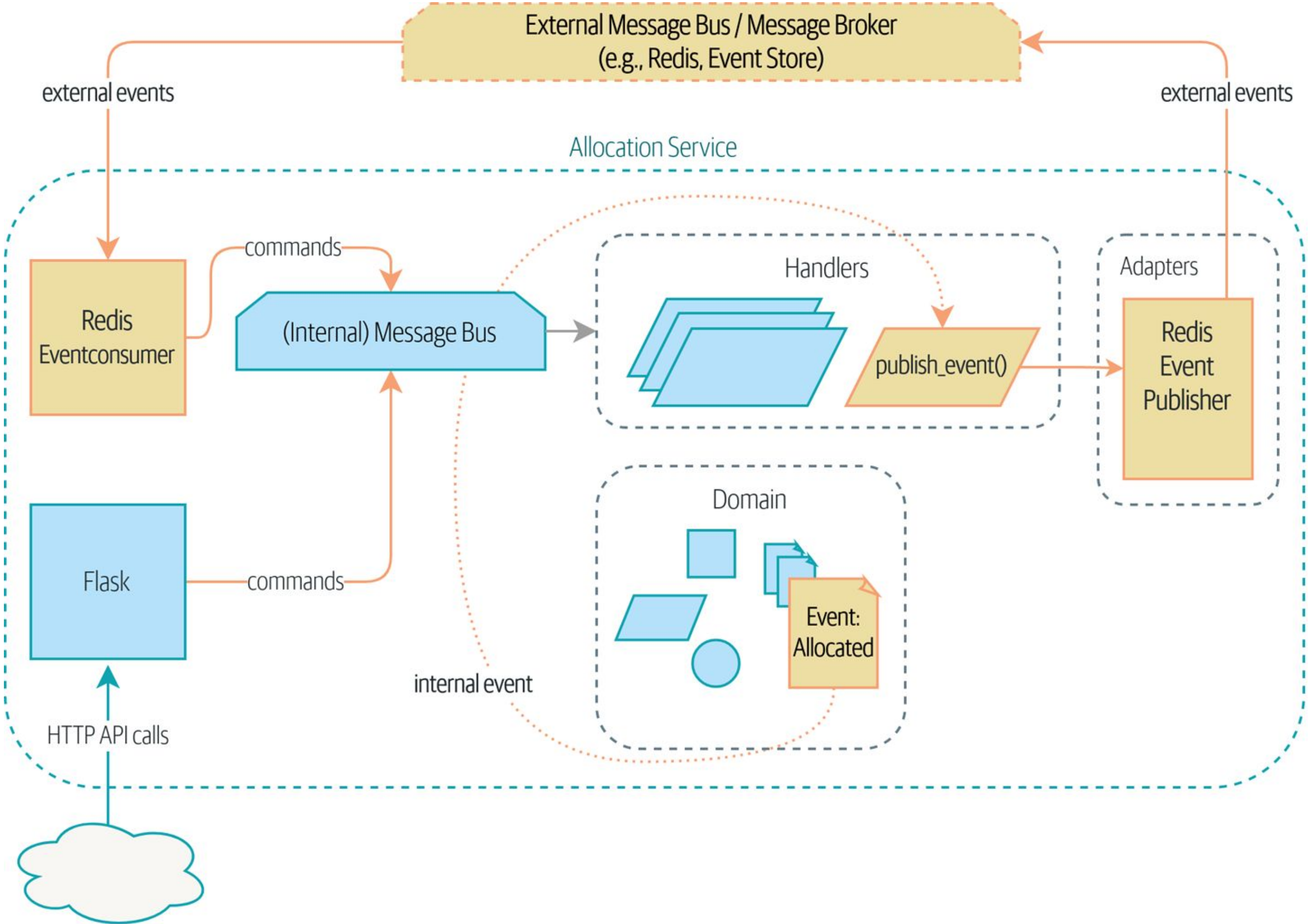
Message Bus

Maps Events to Handlers.
Collects new Events from Handlers.

Command Events

`AllocationRequired` is an event.
The service function `allocate`
becomes a handler.


API
MessageBus
Domain_Model
BatchQuantityChanged event
BatchQuantityChanged Handler + Unit of Work 1
change batch quantity
emit AllocationRequired event(s)
AllocationRequired Handler + Unit of Work 2 (or more)
allocate
API
MessageBus
Domain_Model


External Message Bus / Message Broker
(e.g., Redis, Event Store)
external events
external events
Allocation Service
commands
Redis
Eventconsumer
(Internal) Message Bus
Handlers
publish_event()
Adapters
Redis
Event
Publisher
Domain
Flask
commands
Event:
Allocated
internal event
HTTP API calls

Questions?

We did not talk about:
transactions, CQRS, and a lot more