

O'REILLY®

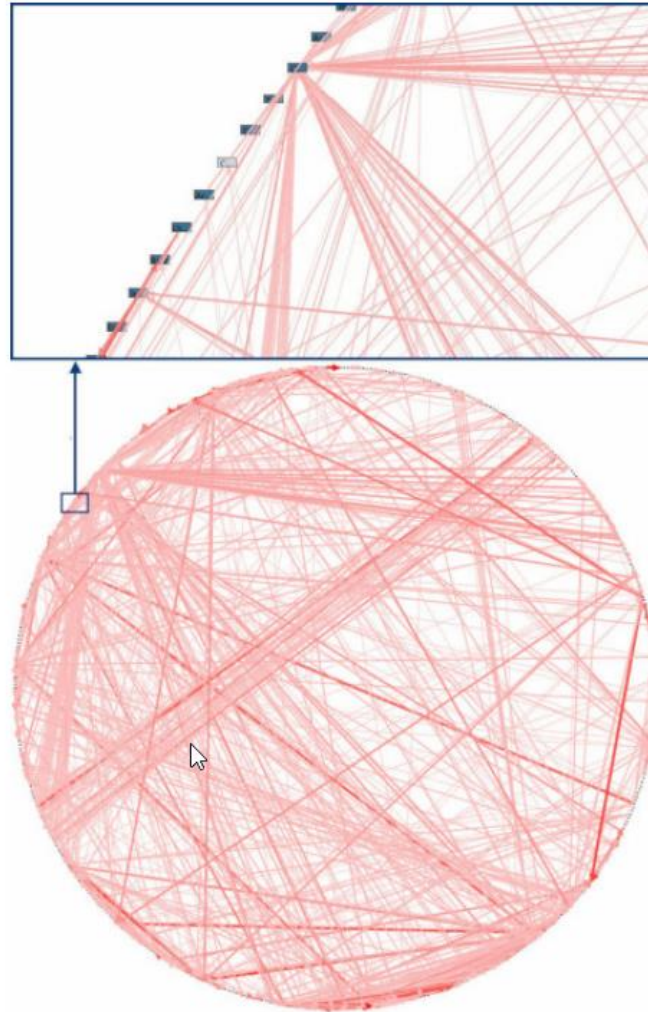
Architecture Patterns with Python

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

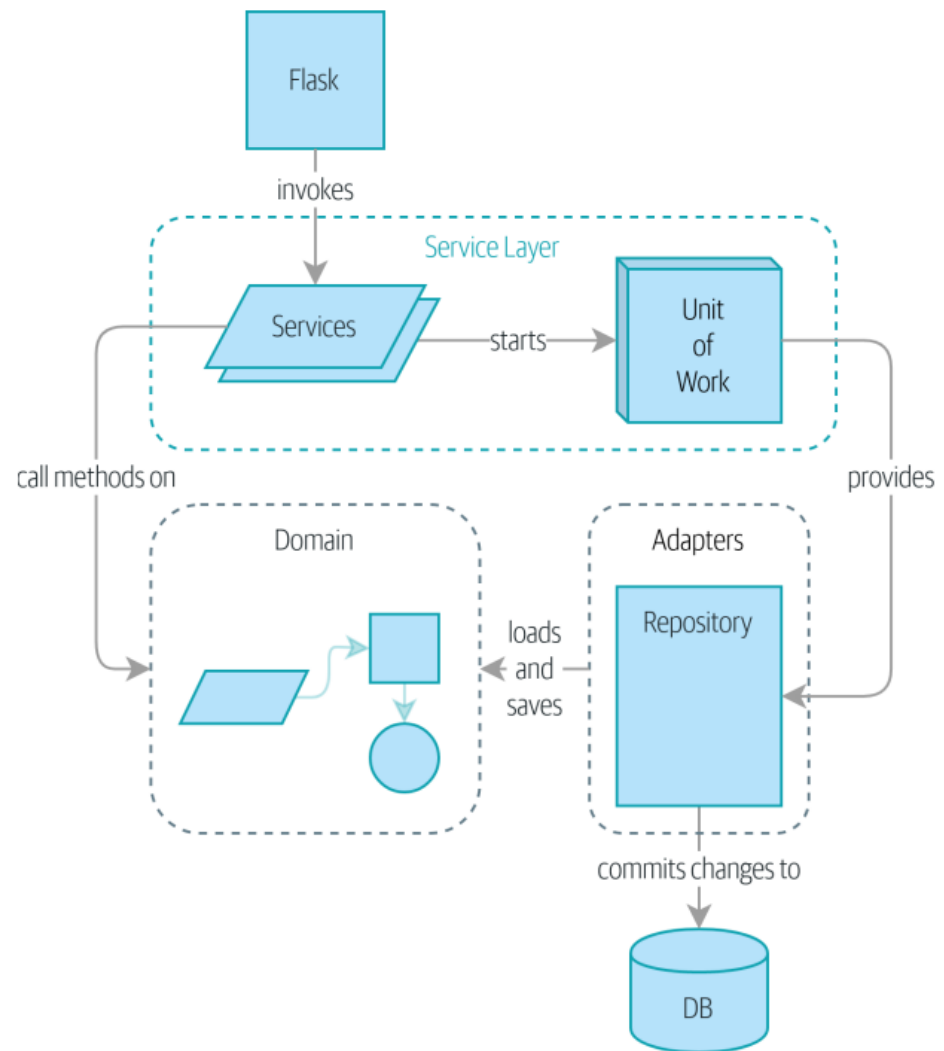


Harry J.W. Percival
& Bob Gregory

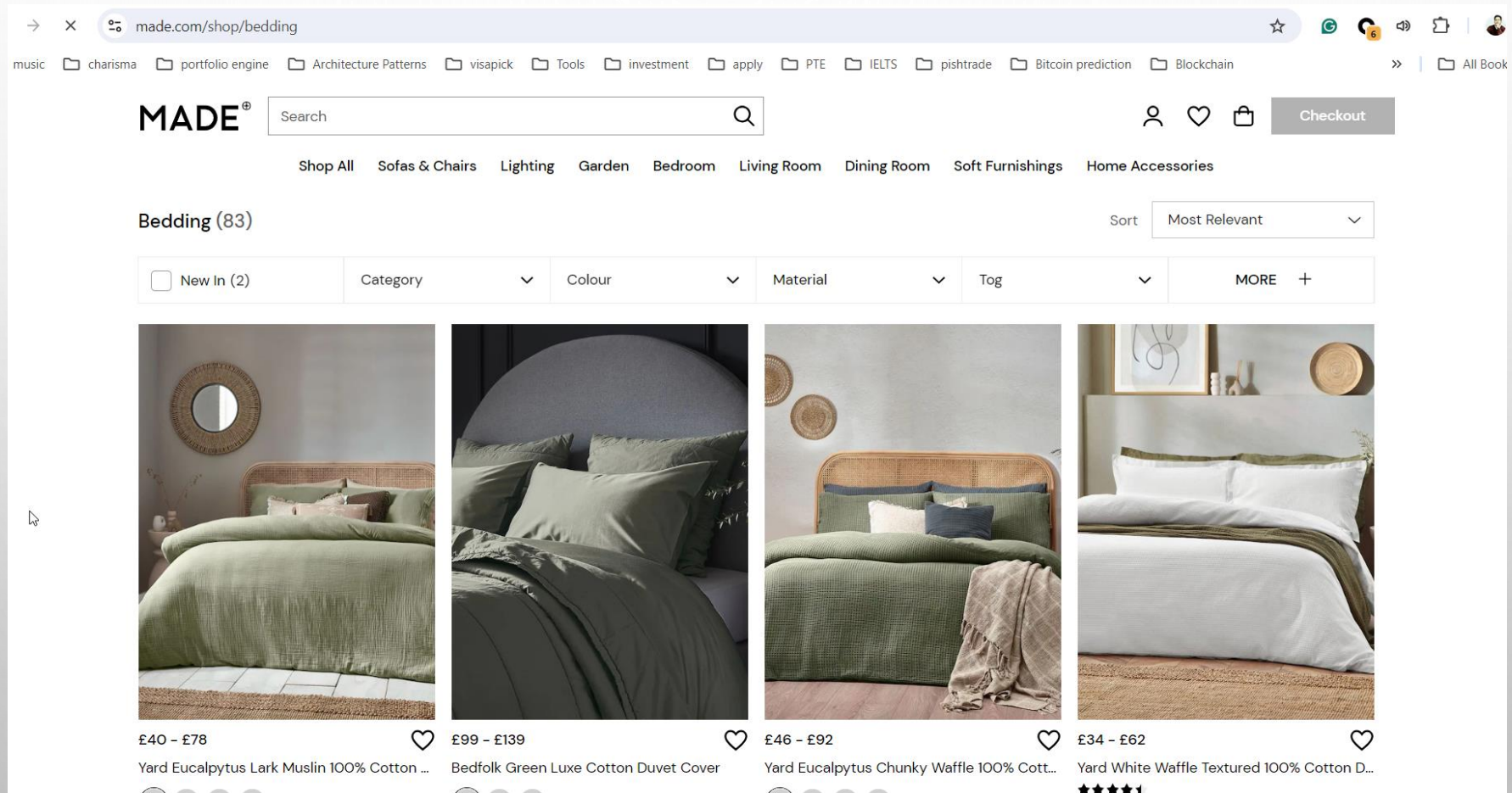
The Big Ball of Mud anti-pattern.



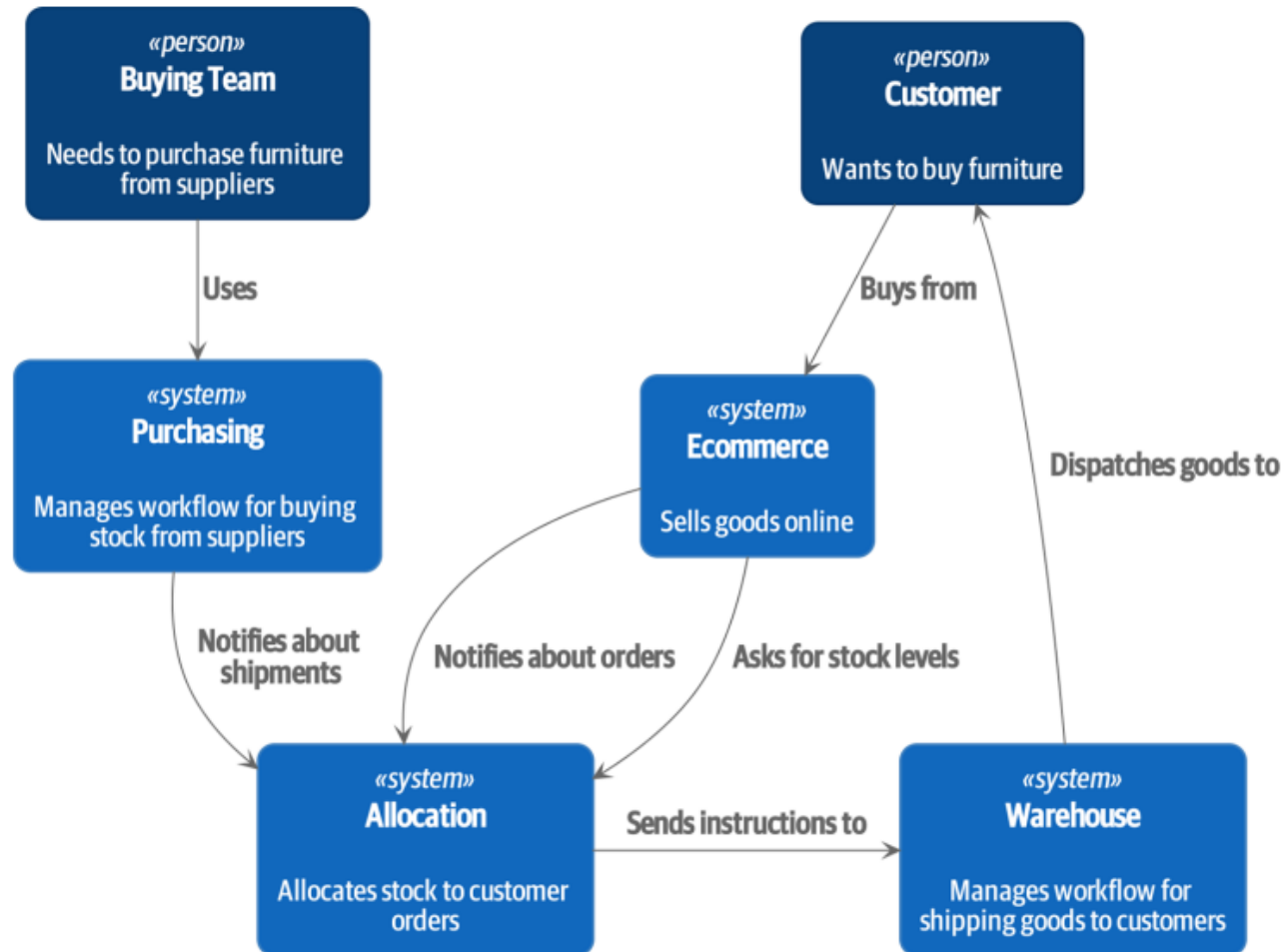
Where we're going in part one of this book



Chapter 1: Problem definition



Chapter 1: Problem definition



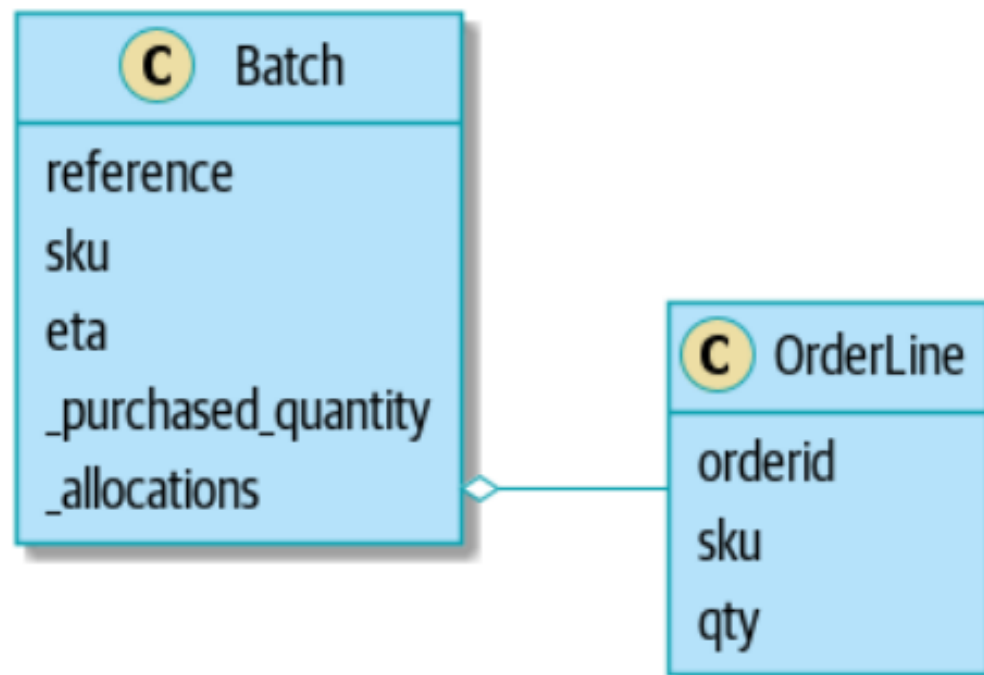
Exploring the Domain Language

- 1- A product is identified by a SKU, pronounced “skew,” which is short for stock-keeping unit. Customers place orders. An order is identified by an order reference and comprises multiple order lines, where each line has a SKU and a quantity. For example: 10 units of RED-CHAIR
- 2- The purchasing department orders small batches of stock. A batch of stock has a unique ID called a reference, a SKU, and a quantity.
- 3- 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
 - 4- We can’t allocate to a batch if the available quantity is less than the quantity of the order line
 - 5- We can’t allocate the same line twice.

Unit Testing Domain Models

```
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
```

Here is our UML and domain model:



[Link of the model in GitHub](#)

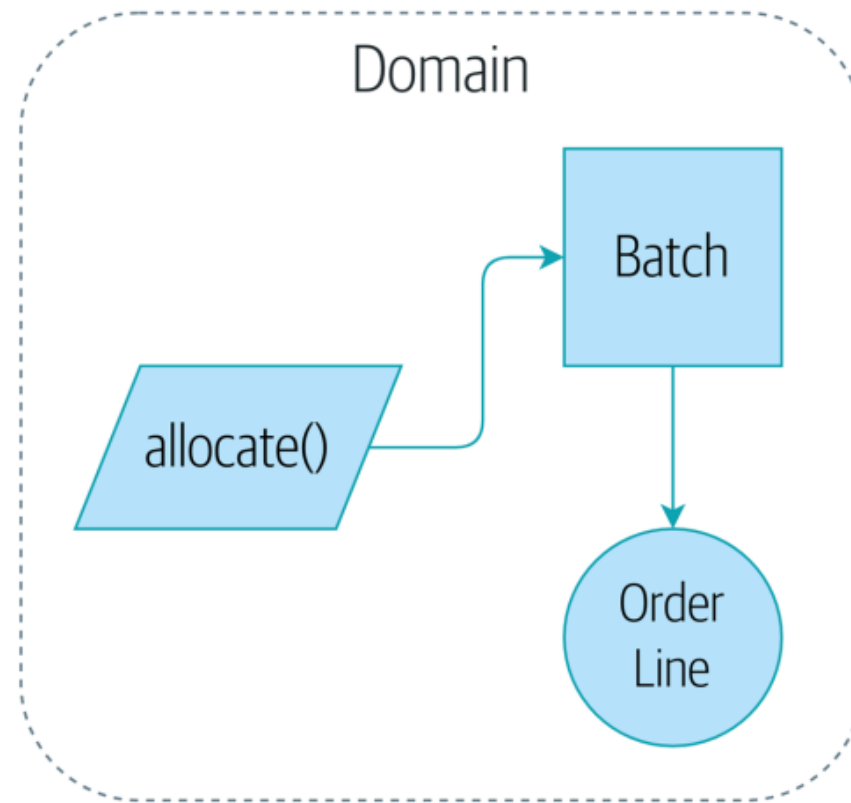
Value Object and Entity

- Whenever we have a business concept that has data but no identity, we often choose to represent it using the **Value Object** pattern. A value object is any domain object that is uniquely identified by the data it holds.
- We use the term entity to describe a domain object that has long-lived **identity**. We can change their values, and they are still recognizably the same thing. Batches, in our example, are entities.

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

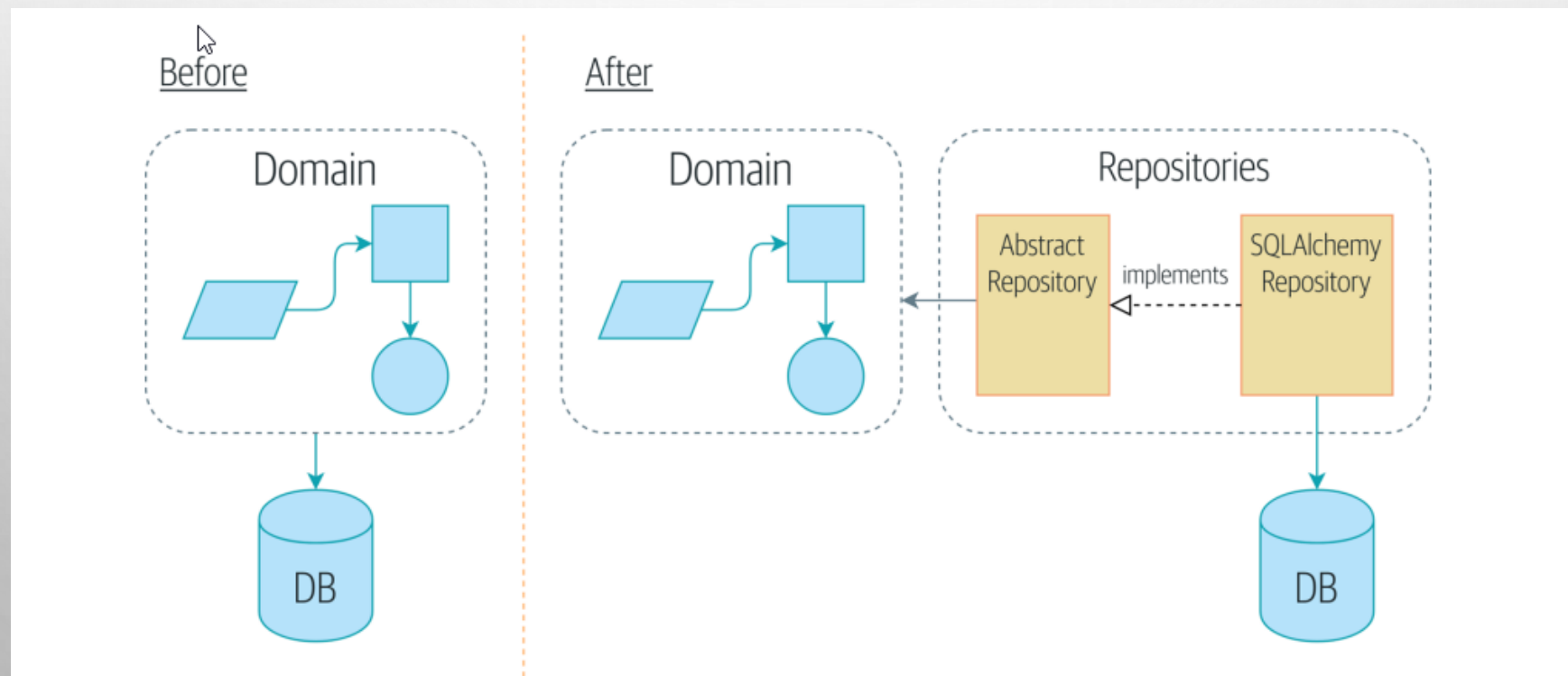
- A thing that allocates an order line, given a set of batches, sounds a lot like a function, and we can take advantage of the fact that Python is a multiparadigm language and just make it a function.
- Python is a multiparadigm language, so let the “verbs” in your code be functions.
- [Link of the allocate service](#)

Figure below is a visual representation of where we've ended up.



Chapter 2: Repository Pattern

- We'll introduce the **Repository pattern**, a simplifying abstraction over data storage, allowing us to decouple our model layer from the business layer (**Dependency Inversion**).



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

```
1 @flask.route.gubbins
2 def allocate_endpoint():
3     session = start_session()
4     # extract order line from request
5     line = OrderLine(
6         request.json['orderid'],
7         request.json['sku'],
8         request.json['qty'],
9     )
10    # load all batches from the DB
11    batches = session.query(Batch).all()
12    # call our domain service
13    allocate(line, batches)
14    # save the allocation back to the database
15    session.commit()
16    💡 return 201
```


The **Repository Pattern** is a design pattern that acts as an abstraction layer between the data access layer and the business logic of an application.

[GitHub Link](#)

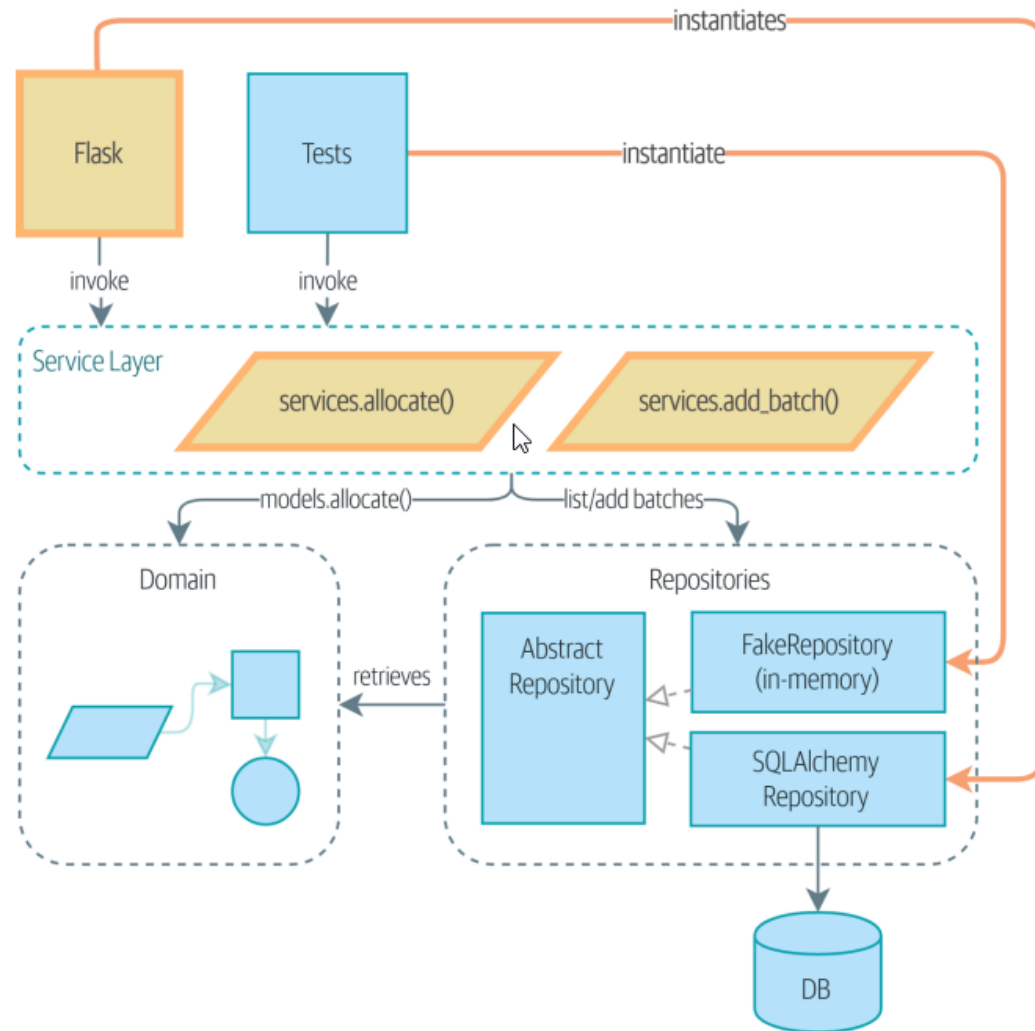
⚠ Invalid Python interpreter selected for the project

```
1  @flask.route.gubbins
2  def allocate_endpoint():
3      batches = SQLAlchemyRepository.list()
4      lines = [
5          OrderLine(l['orderid'], l['sku'], l['qty'])
6          for l in request.params...
7      ]
8      allocate(lines, batches)
9      session.commit()
10  💡 return 201
```

Chapter 3: A Brief Interlude: On Coupling and Abstractions

- Read about this chapter in the following Link: [Chapter 3](#)


Chapter 4: Our First Use Case: Flask API and Service Layer



A First End-to-End Test

```
1
2
3  @pytest.mark.usefixtures("restart_api")
4 def test_happy_path_returns_201_and_allocated_batch(add_stock):
5     sku, othersku = random_sku(), random_sku("other")
6     earlybatch = random_batchref(1)
7     laterbatch = random_batchref(2)
8     otherbatch = random_batchref(3)
9     add_stock(
10         [
11             (laterbatch, sku, 100, "2011-01-02"),
12             (earlybatch, sku, 100, "2011-01-01"),
13             (otherbatch, othersku, 100, None),
14         ]
15     )
16     data = {"orderid": random_orderid(), "sku": sku, "qty": 3}
17     url = config.get_api_url()
18
19     r = requests.post(f"{url}/allocate", json=data)
20
21     assert r.status_code == 201
22     assert r.json()["batchref"] == earlybatch
```

The Straightforward Implementation

```
2
3 @app.route("/allocate", methods=['POST'])
4 def allocate_endpoint():
5     session = get_session()
6     batches = repository.SqlAlchemyRepository(session).list()
7     line = model.OrderLine(
8         request.json['orderid'],
9         request.json['sku'],
10        request.json['qty'],
11    )
12      batchref = model.allocate(line, batches)
13     return jsonify({'batchref': batchref}), 201
```

I

But if we want to add more error handling to this API it should be changed as below:

```
3
4 def is_valid_sku(sku, batches):
5     return sku in {b.sku for b in batches}
6
7 @app.route("/allocate", methods=['POST'])
8 def allocate_endpoint():
9     session = get_session()
10    batches = repository.SqlAlchemyRepository(session).list()
11    line = model.OrderLine(
12        request.json['orderid'],
13        request.json['sku'],
14        request.json['qty'],
15    )
16    if not is_valid_sku(line.sku, batches):
17        return jsonify({'message': f'Invalid sku {line.sku}'}), 400
18    try:
19        batchref = model.allocate(line, batches)
20    except model.OutOfStock as e:
21        return jsonify({'message': str(e)}), 400
22    session.commit()
23    ⚡ return jsonify({'batchref': batchref}), 201
24
```

And finally we'll write a service function that looks something like this (**services.py**):

```
3
4 class InvalidSku(Exception):
5     pass
6
7
8 def is_valid_sku(sku, batches):
9     return sku in {b.sku for b in batches}
10
11
12 def allocate(line: OrderLine, repo: AbstractRepository, session) -> str:
13     batches = repo.list()
14     if not is_valid_sku(line.sku, batches):
15         raise InvalidSku(f"Invalid sku {line.sku}")
16     batchref = model.allocate(line, batches)
17     session.commit()
18     💡 return batchref
```

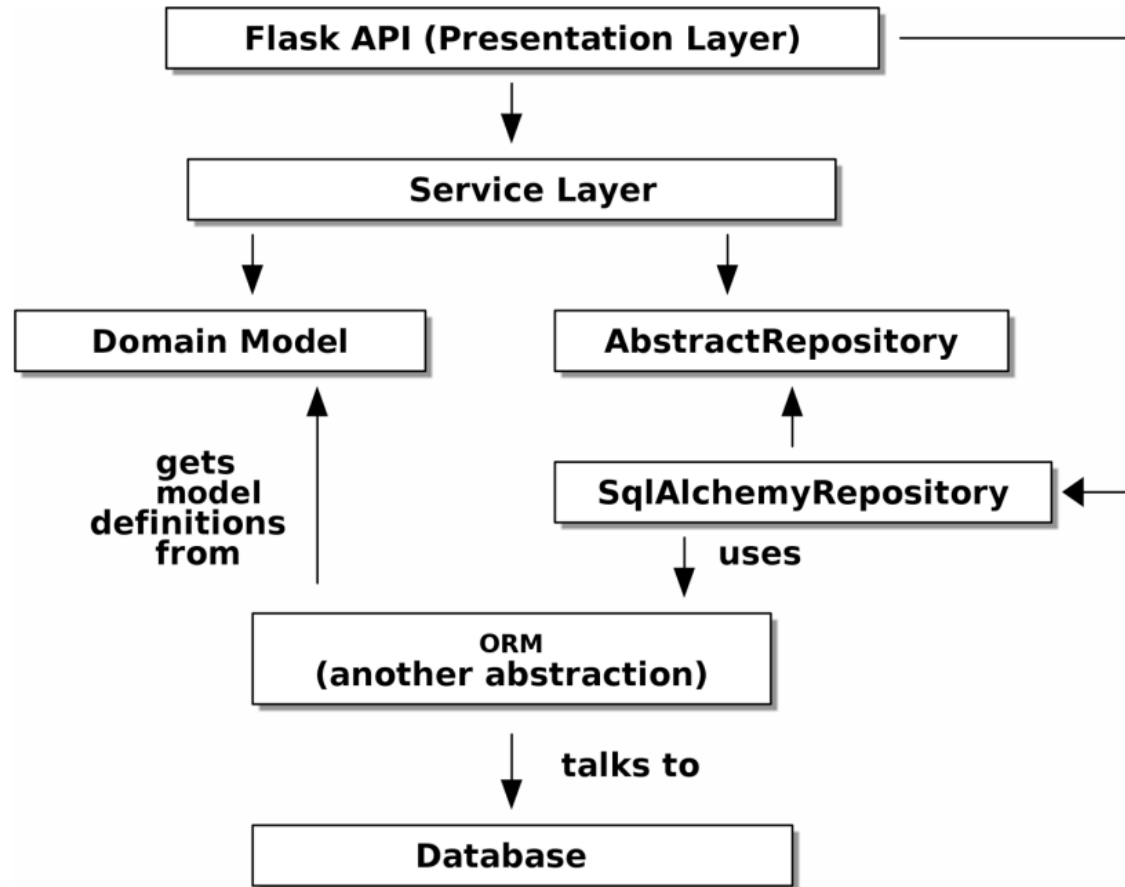
our Flask app now looks a lot cleaner:


```
3
4 @app.route("/allocate", methods=["POST"])
5 def allocate_endpoint():
6     session = get_session()
7     repo = repository.SqlAlchemyRepository(session)
8     line = model.OrderLine(
9         request.json["orderid"], request.json["sku"], request.json["qty"],
10    )
11
12     try:
13         batchref = services.allocate(line, repo, session)
14     except (model.OutOfStock, services.InvalidSku) as e:
15         return {"message": str(e)}, 400
16
17     💡 return {"batchref": batchref}, 201
```

- 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.


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


The DIP in Action [GitHub](#)





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

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

Chapter 5: TDD in High Gear and Low Gear

- Tests are supposed to help us change our system fearlessly, but often we see teams writing too many tests against their domain model. This causes problems when they come to change their codebase and find that they need to update tens or even hundreds of unit tests.
- As we get further into the book, you'll see how the service layer forms an API for our system that we can drive in multiple ways. Testing against this API reduces the amount of code that we need to change when we refactor our domain model.

On Deciding What Kind of Tests to Write

Low feedback
Low barrier to change
High system coverage



API Tests

Service-Layer Tests


High feedback
High barrier to change
Focused coverage



Domain Tests



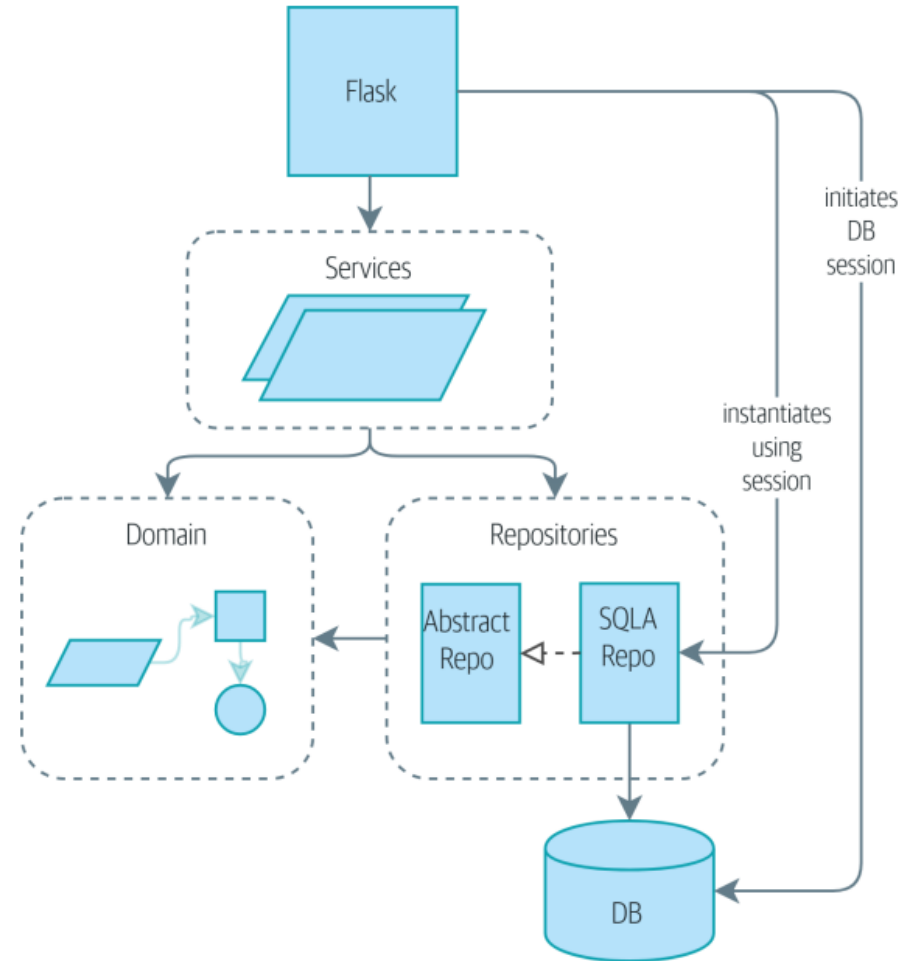
High and Low Gear

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

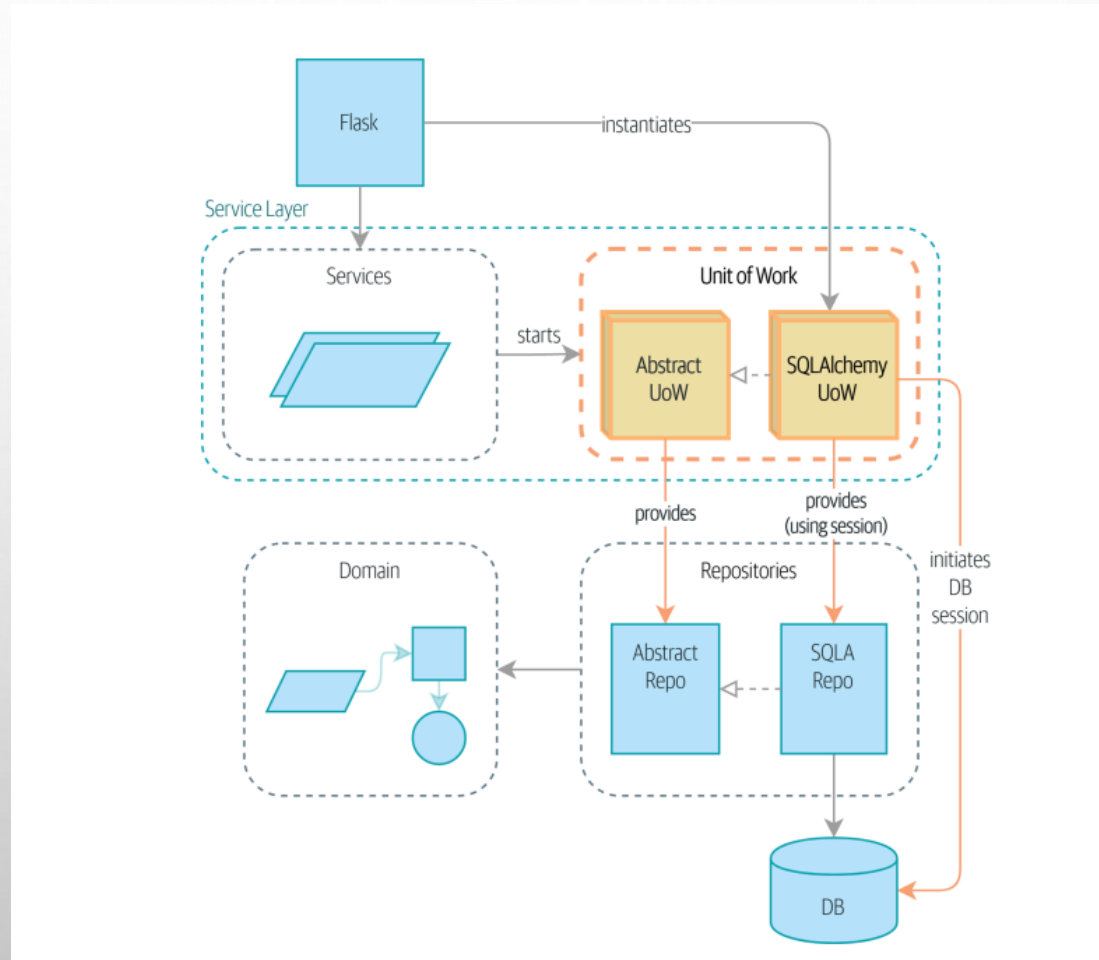
Chapter 6: Unit of Work Pattern

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

Without UoW: API talks directly to three layers



The Flask API now does only two things: it initializes a unit of work, and it invokes a service. The service collaborates with the UoW, but neither the service function itself nor Flask now needs to talk directly to the database.



```
2 class AbstractUnitOfWork(abc.ABC):
3     batches: repository.AbstractRepository
4
5     def __enter__(self) -> AbstractUnitOfWork:
6         return self
7
8     def __exit__(self, *args):
9         self.rollback()
10
11     @abc.abstractmethod
12     def commit(self):
13         raise NotImplementedError
14
15     @abc.abstractmethod
16     def rollback(self):
17         raise NotImplementedError
18
19
20 class SQLAlchemyUnitOfWork(AbstractUnitOfWork):
21     def __init__(self, session_factory=DEFAULT_SESSION_FACTORY):
22         self.session_factory = session_factory
23
24     def __enter__(self):
25         self.session = self.session_factory() # type: Session
26         self.batches = repository.SqlAlchemyRepository(self.session)
27         return super().__enter__()
28
29     def __exit__(self, *args):
30         super().__exit__(*args)
31         self.session.close()
32
33     def commit(self):
34         self.session.commit()
35
36     def rollback(self):
37         self.session.rollback()
```


Here's how the service layer will look when we're finished

```
2
3  def allocate(
4      orderid: str, sku: str, qty: int,
5      uow: unit_of_work.AbstractUnitOfWork) -> str:
6      line = OrderLine(orderid, sku, qty)
7      with uow:
8          batches = uow.batches.list()
9          if not is_valid_sku(line.sku, batches):
10             raise InvalidSku(f"Invalid sku {line.sku}")
11             batchref = model.allocate(line, batches)
12             uow.commit()
13     return batchref
14
15
```

Our service layer now has only the one dependency, once again on an **abstract UoW**.

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

Suppose we want to be able to **deallocate** and then **reallocate** orders. If **deallocate()** fails, we don't want to call **allocate()**, obviously. And, If **allocate()** fails, we probably don't want to actually commit the **deallocate()** either.

```
2
3 def reallocate(line: OrderLine, uow: AbstractUnitOfWork) -> str:
4     with uow:
5         batch = uow.batches.get(sku=line.sku)
6         if batch is None:
7             raise InvalidSku(f'Invalid sku {line.sku}')
8         batch.deallocate(line)
9          allocate(line)
10        uow.commit()
```


Example 2: Change Batch Quantity

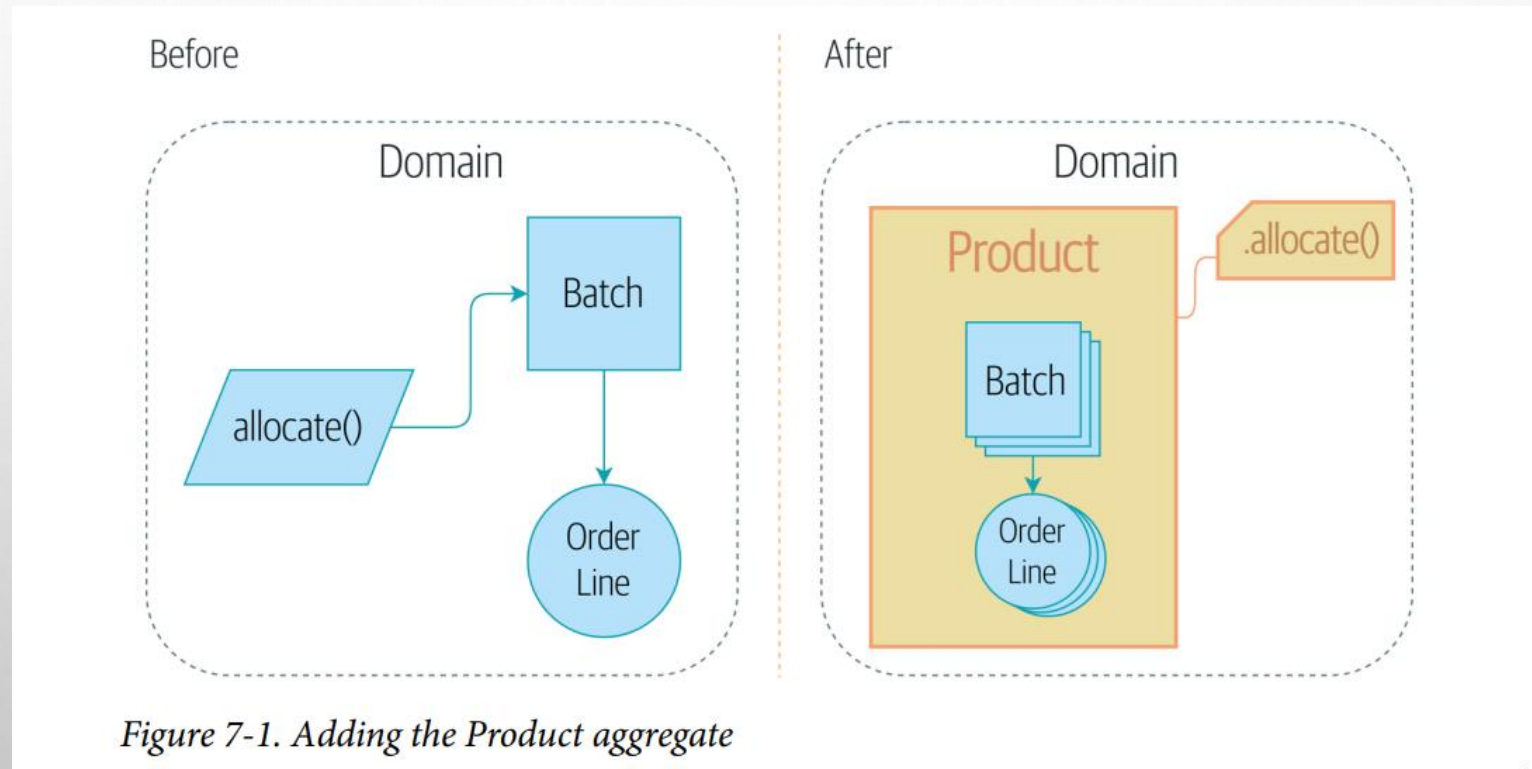
Our shipping company gives us a call to say that one of the container doors opened, and half our sofas have fallen into the Indian Ocean. Oops! Here we may need to **deallocate** any number of lines. If we get a failure at any stage, we probably want to commit none of the changes.

```
1
2
3 def change_batch_quantity(batchref: str, new_qty: int, uow: AbstractUnitOfWork):
4     with uow:
5         batch = uow.batches.get(reference=batchref)
6         batch.change_purchased_quantity(new_qty)
7         while batch.available_quantity < 0:
8             line = batch.deallocate_one()
9         uow.commit()
```

[GitHub Link UOW](#)

Chapter 7: Aggregates and Consistency Boundaries

Figure below shows a preview of where we're headed: we'll introduce a new model object called **Product** to wrap multiple batches, and we'll make the old **allocate()** domain service available as a method on **Product** instead.



Invariants, Constraints, and Consistency

- **constraint** is a rule that restricts the possible states our model can get into, while an **invariant** is defined a little more precisely as a condition that is always true.
- If we were writing a **hotel-booking system**, we might have the constraint that double bookings are not allowed. This supports the **invariant** that a room cannot have more than one booking for the same night
- 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 usually solve this problem by applying **locks** to our database tables. This prevents two operations from happening simultaneously on the same row or same table.

What Is an Aggregate?

- An aggregate is just a domain object that contains other domain objects and lets us treat the whole collection as a single unit.
- For example, if we're building a shopping site, the **Cart** might make a good aggregate: it's a collection of items that we can treat as a single unit. Importantly, we want to load the entire basket as a single blob from our data store.
- We don't want two requests to modify the basket at the same time, or we run the risk of weird concurrency errors.
- Each basket is a single consistency boundary responsible for maintaining its own invariants.
- An **AGGREGATE** is a cluster of associated objects that we treat as a unit for the purpose of data changes. our aggregate has a root entity (the Cart) that encapsulates access to items. Each item has its own identity, but other parts of the system will always refer to the Cart only as an indivisible whole.

Simple Example: Shopping Cart

- Consider a simple e-commerce system with a **ShoppingCart** aggregate. Here's how the aggregate ensures consistency:
- **ShoppingCart (Aggregate Root)**: Contains items, a total cost, and methods to add or remove items.
- **CartItem (Entity within Aggregate)**: Represents an individual item in the cart, including product details and quantity.
- **ProductID (Value Object)**: A unique identifier for a product.
- **Business Rule (Invariant)**: The total cost of the cart must always equal the sum of the costs of all items in the cart.

```
4
5 class ProductID(BaseModel):
6     id: str
7
8 class CartItem(BaseModel):
9     product_id: ProductID
10    quantity: int
11    price_per_unit: float
12
13    def total_price(self):
14        return self.quantity * self.price_per_unit
15
16 class ShoppingCart(BaseModel):
17     items: Dict[str, CartItem] # key is ProductID.id
18
19    def add_item(self, product_id: ProductID, quantity: int, price_per_unit: float):
20        if product_id.id in self.items:
21            self.items[product_id.id].quantity += quantity
22        else:
23            self.items[product_id.id] = CartItem(product_id=product_id, quantity=quantity, price_per_unit=price_per_unit)
24            self.check_invariants()
25
26    def remove_item(self, product_id: ProductID, quantity: int):
27        if product_id.id in self.items:
28            if quantity >= self.items[product_id.id].quantity:
29                del self.items[product_id.id]
30            else:
31                self.items[product_id.id].quantity -= quantity
32            self.check_invariants()
33
34    def total_cost(self):
35        return sum(item.total_price() for item in self.items.values())
36
37    def check_invariants(self):
38        if self.total_cost() < 0:
39            ⚡ raise ValueError("Total cost cannot be negative.")
40
```

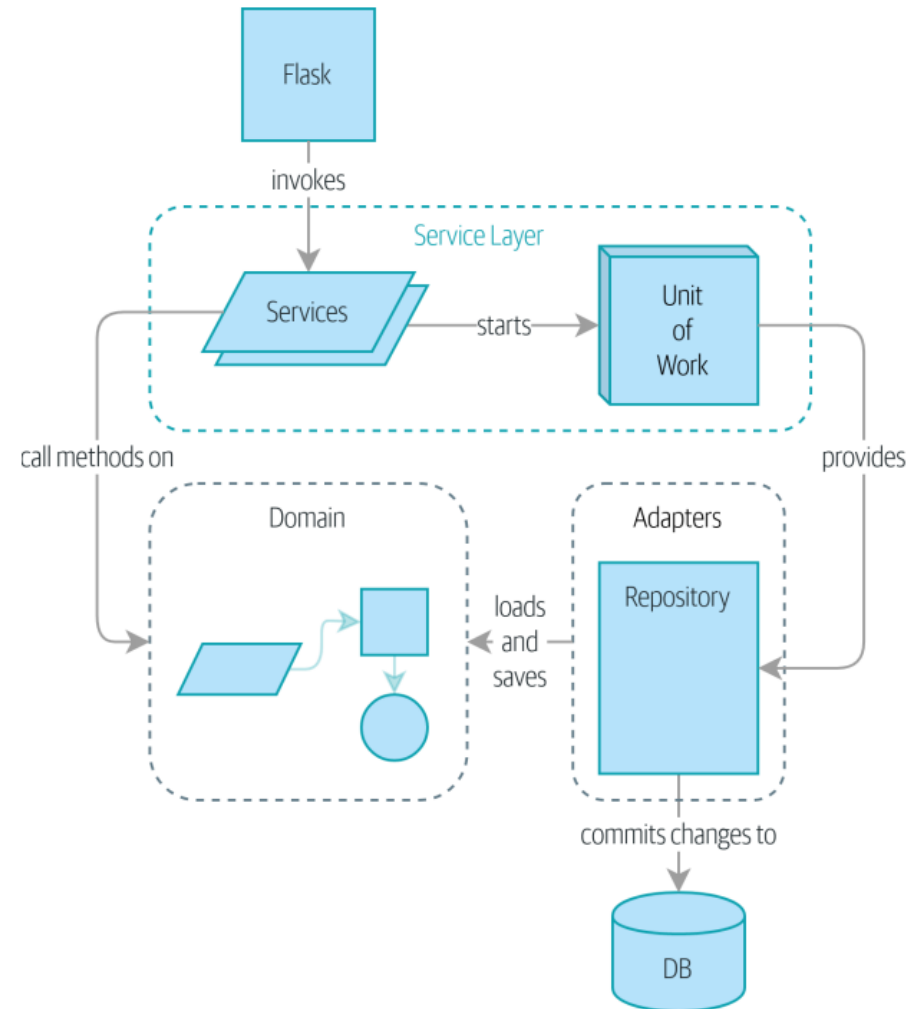

In this example:

- 1- The **ShoppingCart** class (aggregate root) manages all modifications to the cart's contents.
- 2- It ensures the **invariant** (total cost is correct) is maintained after any changes such as adding or removing items.
- 3- All interactions with **CartItem** objects go through the **ShoppingCart**, which controls and checks the consistency of its internal state. This model allows the application to maintain consistent rules about how products are added or removed, ensuring data integrity and business rule enforcement within the shopping cart system.
- (According to the previous rule) only Aggregate Root can be directly retrieved from the database by Query. In other words, a Repository is defined for each Aggregate Root, and other objects cannot be retrieved directly and must be retrieved by the corresponding Aggregate Root. Also, direct access to save or edit an entity inside an Aggregate is meaningless, and database transactions are meaningful at the level of an Aggregate.

One Aggregate = One Repository

- Once you define certain entities to be aggregates, we need to apply the rule that they are the only entities that are publicly accessible to the outside world. In other words, the only repositories we are allowed should be repositories that return aggregates.
- In our case, we'll switch from **Batchepository** to **ProductRepository**: [GitHub Link](#)
- Here is our service layer: [GitHub Link](#)

Part I Recap




Part I Recap

- We've decoupled the infrastructural parts of our system, like the database and API handlers. This helps us to keep our codebase well organized and stops us from building a big ball of mud.
- Lastly, we've talked about the idea of consistency boundaries. We don't want to lock our entire system whenever we make a change, so we have to choose which parts are consistent with one another.
- For a small system, this is everything you need to go and play with the ideas of domain-driven design. You now have the tools to build database-agnostic domain models that represent the shared language of your business experts. Hurrah!
- If your app is essentially a simple CRUD wrapper around a database and isn't likely to be anything more than that in the foreseeable future, you don't need these patterns. Go ahead and use Django, and save yourself a lot of bother.

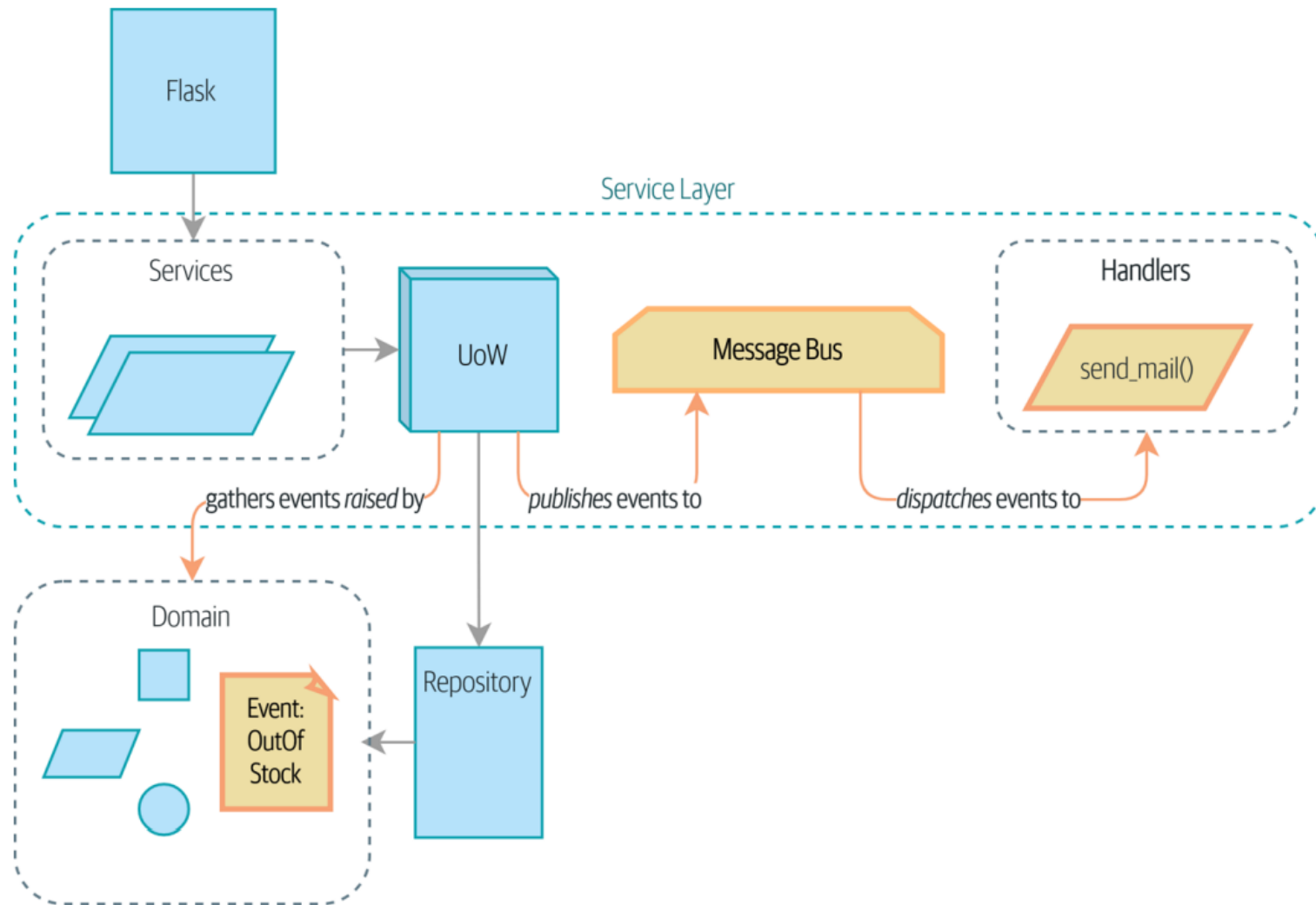


Event-Driven Architecture

- In Part II, we'll show how the techniques from Part I can be extended to distributed systems.
 - We'll zoom out to look at how we can compose a system from many small components that interact through asynchronous message passing.
 - We'll see how our Service Layer and Unit of Work patterns allow us to reconfigure our app to run as an asynchronous message processor
- 

Chapter 8: Events and the Message Bus

- Our example will be a typical notification requirement: when we can't allocate an order because we're out of stock, we should alert the buying team.
- They'll go and fix the problem by buying more stock, and all will be well.
- For a first version, our product owner says we can just send the alert by email.



First, Let's Avoid Making a Mess of Our Web Controllers

```
3
4 @app.route("/allocate", methods=['POST'])
5 def allocate_endpoint():
6     line = model.OrderLine(
7         request.json['orderid'],
8         request.json['sku'],
9         request.json['qty'],
10    )
11    try:
12        uow = unit_of_work.SqlAlchemyUnitOfWork()
13        batchref = services.allocate(line, uow)
14    except (model.OutOfStock, services.InvalidSku) as e:
15        send_mail(
16            'out of stock',
17            'stock_admin@made.com',
18            f'{line.orderid} - {line.sku}')
19        return jsonify({'message': str(e)}), 400
20    return jsonify({'batchref': batchref}), 201
21
```

But Sending email isn't the job of our HTTP layer, and we'd like to be able to unit test this new feature.

we may look at putting it right at the source, in the model

```
3
4  ✓ def allocate(self, line: OrderLine) -> str:
5      try:
6          batch = next(
7              b for b in sorted(self.batches) if b.can_allocate(line)
8          )
9          #...
10     ✓ except StopIteration:
11         ! email.send_mail('stock@made.com', f'Out of stock for {line.sku}')
12         raise OutOfStock(f'Out of stock for sku {line.sku}')
```

But that's even worse! We don't want our model to have any dependencies on infrastructure concerns like **email.send_mail**. The domain model's job is to know that we're out of stock, but the responsibility of sending an alert belongs elsewhere.

Or the Service Layer!

```
3
4 def allocate(
5     orderid: str, sku: str, qty: int,
6     uow: unit_of_work.AbstractUnitOfWork) -> str:
7     line = OrderLine(orderid, sku, qty)
8     with uow:
9         product = uow.products.get(sku=line.sku)
10        if product is None:
11            raise InvalidSku(f'Invalid sku {line.sku}')
12        try:
13            batchref = product.allocate(line)
14            uow.commit()
15            return batchref
16        except model.OutOfStock:
17            email.send_mail('stock@made.com', f'Out of stock for {line.sku}')
18            raise
```

Catching an exception and reraising it? It could be worse, but it's definitely making us unhappy. Why is it so hard to find a suitable home for this code?

Single Responsibility Principle

- **Rule of thumb:** if you can't describe what your function does without using words like “**then**” or “**and,**” you might be violating the SRP.
- We should be able to turn this feature on or off, or to switch to SMS notifications instead, without needing to change the rules of our domain model.
- We'd also like to keep the service layer free of implementation details

Domain Events and the Message Bus

- An **event** is a kind of **value object**. Events don't have any behavior, because they're pure data structures. We always name events in the language of the domain, and we think of them as part of our domain model. We could store them in `model.py`, but we may as well keep them in their own file.

```
3
4  from dataclasses import dataclass
5
6  class Event:
7      pass
8
9  @dataclass
10 class OutOfStock(Event):
11     sku: str
```


Our **aggregate** will expose a new attribute called **.events** that will contain a list of facts about what has happened, in the form of Event objects.

```
3
4 class Product:
5     def __init__(self, sku: str, batches: List[Batch], version_number: int = 0):
6         self.sku = sku
7         self.batches = batches
8         self.version_number = version_number
9         self.events = [] # type: List[events.Event]
10
11     def allocate(self, line: OrderLine) -> str:
12         try:
13             batch = next(b for b in sorted(self.batches) if b.can_allocate(line))
14             batch.allocate(line)
15             self.version_number += 1
16             return batch.reference
17         except StopIteration:
18             self.events.append(events.OutOfStock(line.sku))
19             return None
```

The Message Bus Maps Events to Handlers

```
3
4 def handle(event: events.Event):
5     for handler in HANDLERS[type(event)]:
6         handler(event)
7
8
9 def send_out_of_stock_notification(event: events.OutOfStock):
10    email.send_mail(
11        "stock@made.com",
12        f"Out of stock for {event.sku}",
13    )
14
15
16 HANDLERS = {
17     events.OutOfStock: [send_out_of_stock_notification],
18 }
```

The UoW Publishes Events to the Message Bus

```
3
4 class AbstractUnitOfWork(abc.ABC):
5     products: repository.AbstractRepository
6
7     def __enter__(self) -> AbstractUnitOfWork:
8         return self
9
10    def __exit__(self, *args):
11        self.rollback()
12
13    def commit(self):
14        self._commit()
15        self.publish_events()
16
17    def publish_events(self):
18        for product in self.products.seen:
19            while product.events:
20                event = product.events.pop(0)
21                messagebus.handle(event)
```

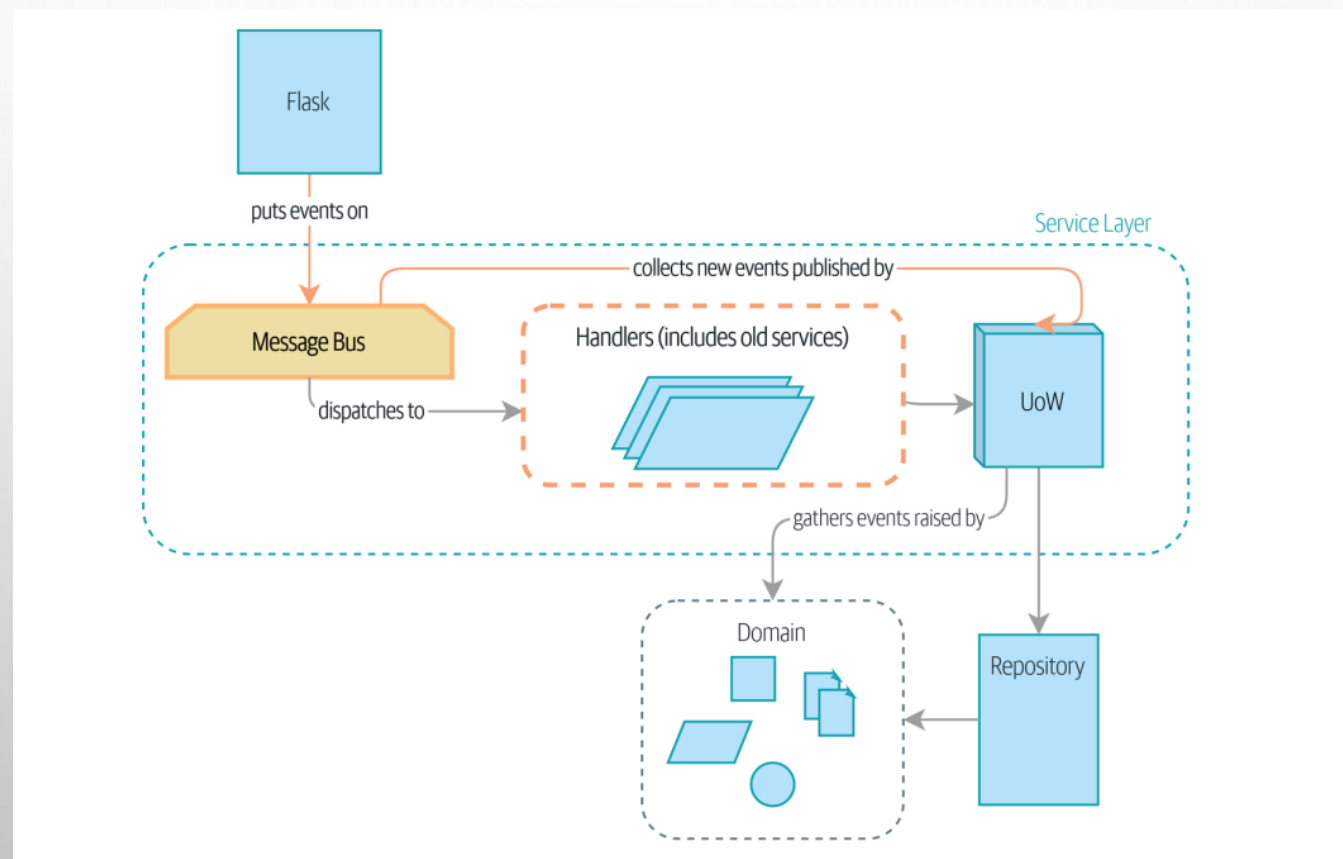
[GitHub Link](#)

Wrap-Up

- Domain events give us a way to handle workflows in our system.
- The magic words “When X, then Y” often tell us about an event that we can make concrete in our system.
- Treating events as first-class things in our model helps us make our code more testable and observable, and it helps isolate concerns.
- You can think of a message bus as a dict that maps from events to their consumers.
- we can tidy up by making our unit of work responsible for raising events that were raised by loaded objects. This is the most complex design.

Chapter 9: Going to Town on the Message Bus

We'll move from the current state, where everything goes via the message bus, and our app has been transformed fundamentally into a message processor.



A New Requirement Leads Us to a New Architecture

- Perhaps someone made a mistake on the number in the manifest, or perhaps some sofas fell off a truck. Following a conversation with the business we model the situation as in below.
- An event we'll call **BatchQuantityChanged** should lead us to change the quantity on the batch
- if the new quantity drops to less than the total already allocated, we need to deallocate those orders from that batch.
- Then each one will require a new allocation, which we can capture as an event called **AllocationRequired**

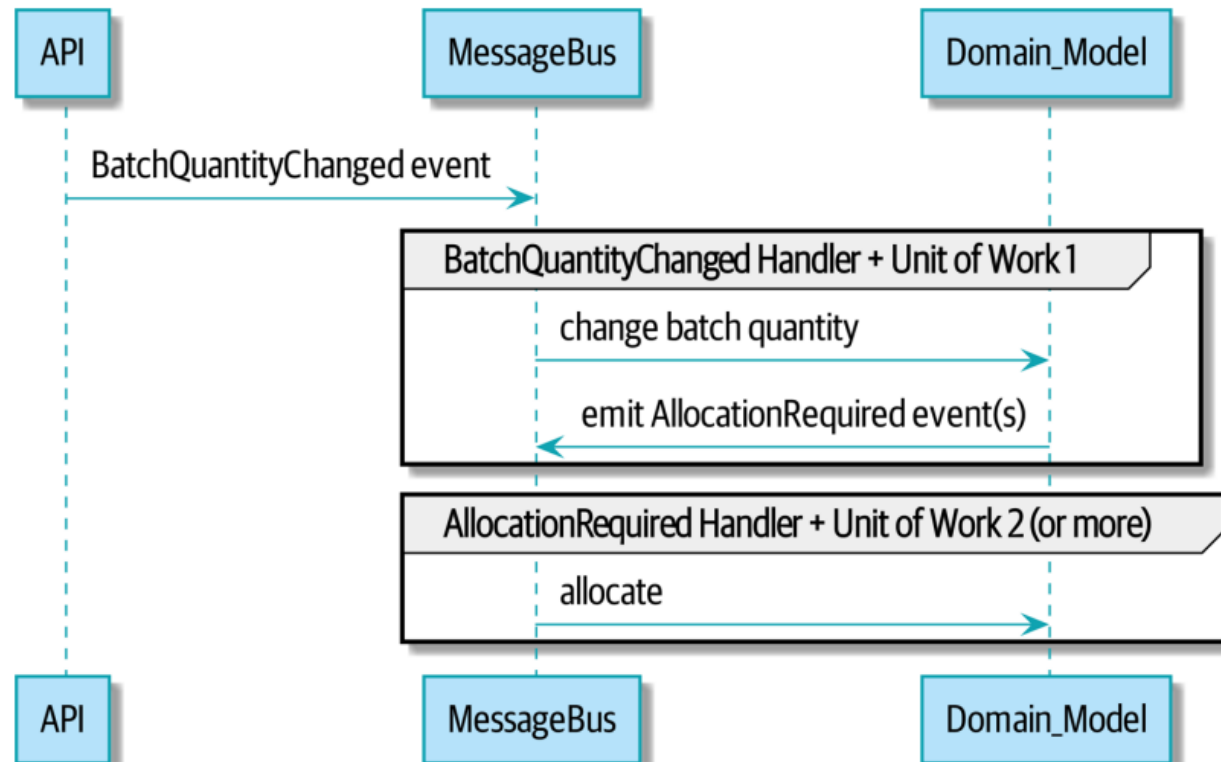
Imagining an Architecture Change: Everything Will Be an Event Handler

- 1- `services.allocate()` could be the handler for an `AllocationRequired` event and could emit `Allocated` events as its output.
- 2- `services.add_batch()` could be the handler for a `BatchCreated` event.
- 3- An event called **BatchQuantityChanged** can invoke a handler called **change_batch_quantity()**.
- 4- And the new **AllocationRequired** events that it may raise can be passed on to **services.allocate()** too

Refactoring Service Functions to Message Handlers

- We start by defining the two events that capture our current API inputs—AllocationRequired and BatchCreated. [GitHub](#)
- we change all the handlers so that they have the same inputs, an **event** and a **UoW** [GitHub](#)
- **The Message Bus Now Collects Events from the UoW** [GitHub](#)
- **Modifying Our API to Work with Events** [GitHub](#)

Implementing Our New Requirement



Implementing Our New Requirement


- The event that tells us a batch quantity has changed [GitHub](#)
- Implementation of new handler [link](#)
- A New Method on the Domain Model [Link](#)
- Message bus [link](#)

What Have We Achieved?

- Events are simple dataclasses that define the data structures for inputs and internal messages within our system.
- This is quite powerful from a DDD standpoint, since events often translate really well into business language.
- Handlers are the way we react to events.
- They can call down to our model or call out to external services.
- We can define multiple handlers for a single event if we want to.
- Handlers can also raise other events.
- This allows us to be very granular about what a handler does and really stick to the SRP.



Why Have We Achieved?

- Here we've added quite a complicated use case (change quantity, deallocate, start new transaction, reallocate, publish external notification), but architecturally, there's been no cost in terms of complexity.
 - We've added new events, new handlers, and a new external adapter (for email), all of which are existing categories of things in our architecture that we understand and know how to reason about
- 

Chapter 10: Commands and Command Handler

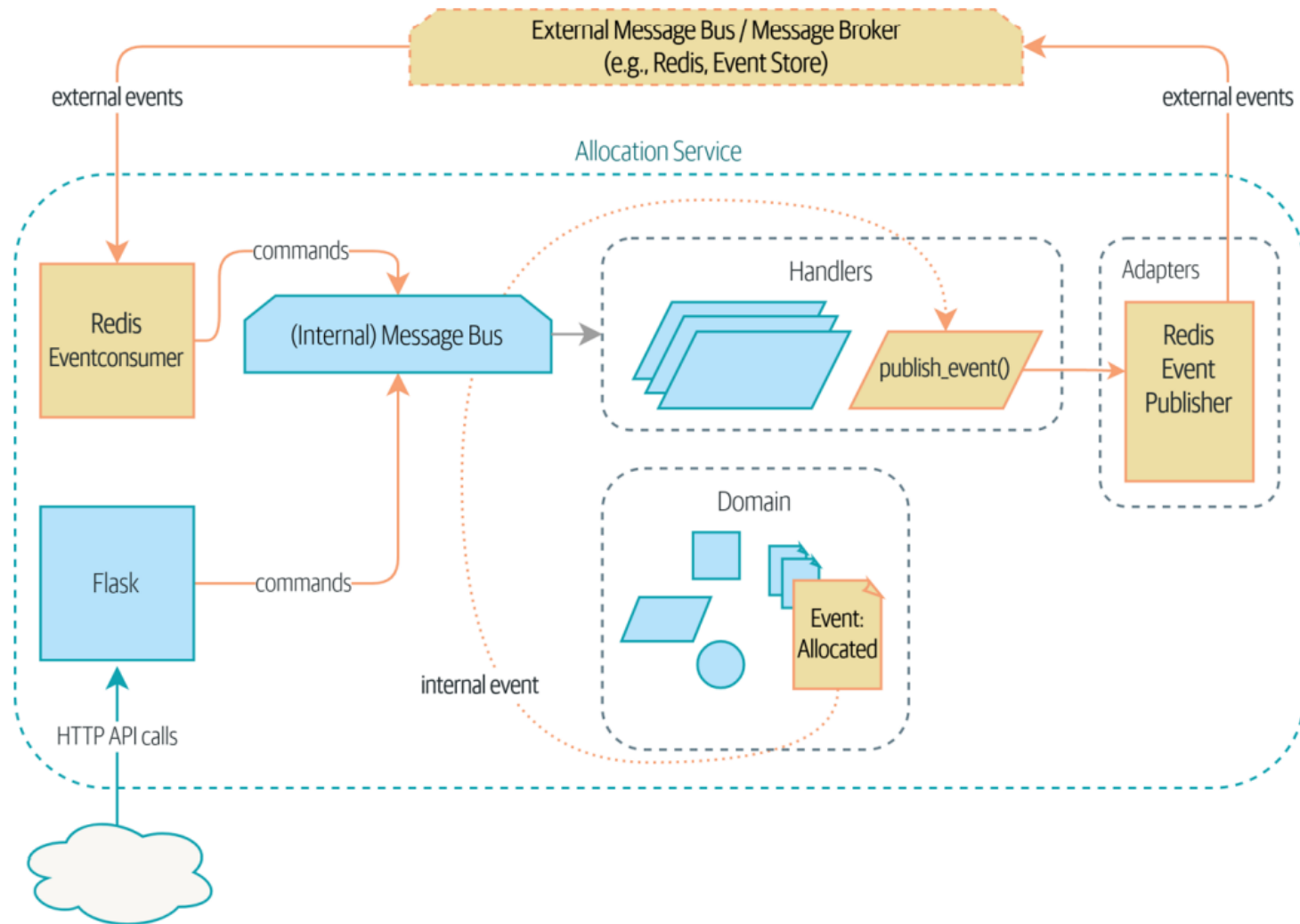
- Like events, commands are a type of message—instructions sent by one part of a system to another.
- Commands are sent by one actor to another specific actor with the expectation that a particular thing will happen as a result.
- We name commands with imperative mood verb phrases like “allocate stock” or “delay shipment.”
- They express our wish for the system to do something. As a result, when they fail, the sender needs to receive error information.

Chapter 10: Commands and Command Handler

- Events are broadcast by an actor to all interested listeners.
- When we publish `BatchQuantityChanged`, we don't know who's going to pick it up.
- We name events with past-tense verb phrases like “order allocated to stock” or “shipment delayed.”
- Events capture facts about things that happened in the past.

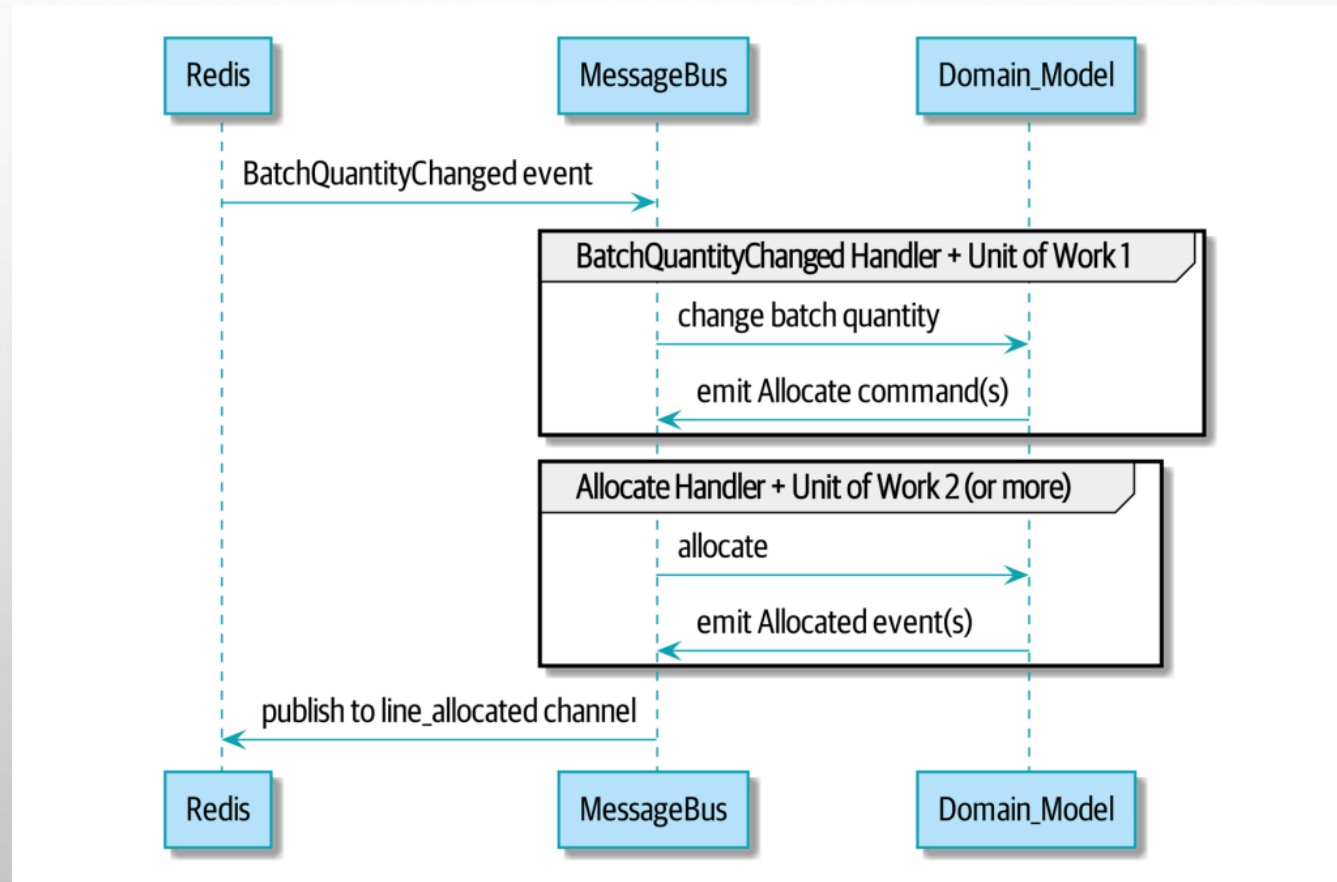
Chapter 11: Using Events to Integrate Microservices

- In the preceding chapter, we never actually spoke about how we would receive the “batch quantity changed” events, or indeed, how we might notify the outside world about reallocations.
- In this chapter, we’d like to show how the events metaphor can be extended to encompass the way that we handle incoming and outgoing messages from the system.
- our application will receive events from external sources via an external message bus (we’ll use Redis pub/sub queues as an example) and publish its outputs, in the form of events, back there as well.



Using a Redis Pub/Sub Channel for Integration

This piece of infrastructure is often called a message broker. The role of a message broker is to take messages from publishers and deliver them to subscribers.



Implementation

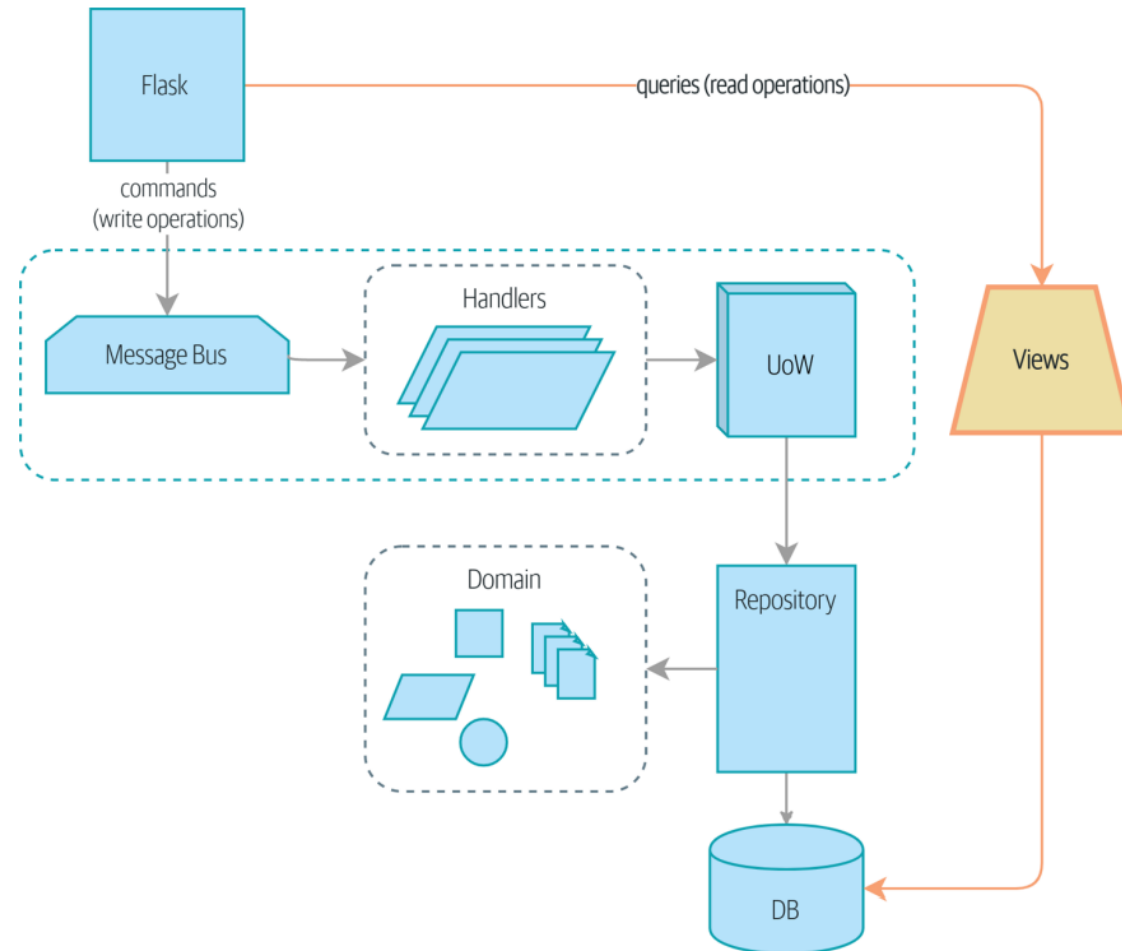
- Our Redis pub/sub listener (we call it an event consumer) [link](#)
- We also build a new downstream adapter to do the opposite job converting domain events to public events [link](#)

Wrap-Up

- Events can come from the outside, but they can also be published externally
- our publish handler converts an event to a message on a Redis
- This kind of temporal decoupling buys us a lot of flexibility in our application integrations
- but as always, it comes at a cost
- More generally, if you're moving from a model of synchronous messaging to an async one, you also open up a whole host of problems having to do with message reliability and eventual consistency.

Command-Query Responsibility Segregation (CQRS)

reads (queries) and writes (commands) are different, so they should be treated differently.



- provide a new read-only endpoint to retrieve allocation state [link](#)
- We'll still keep our view in a separate views.py module link [link](#)
- Splitting out your read-only views from your state-modifying command and event handlers is probably a good idea, even if you don't want to go to full-blown CQRS.

Chapter 13: Dependency Injection (and Bootstrapping)

- We'll also add a new component to our architecture called `bootstrap.py`; it will be in charge of dependency injection, as well as some other initialization stuff that we often need.

Figure below shows what our app looks like without a bootstrapper: the endpoints do a lot of initialization and passing around of our main dependency, the UoW.

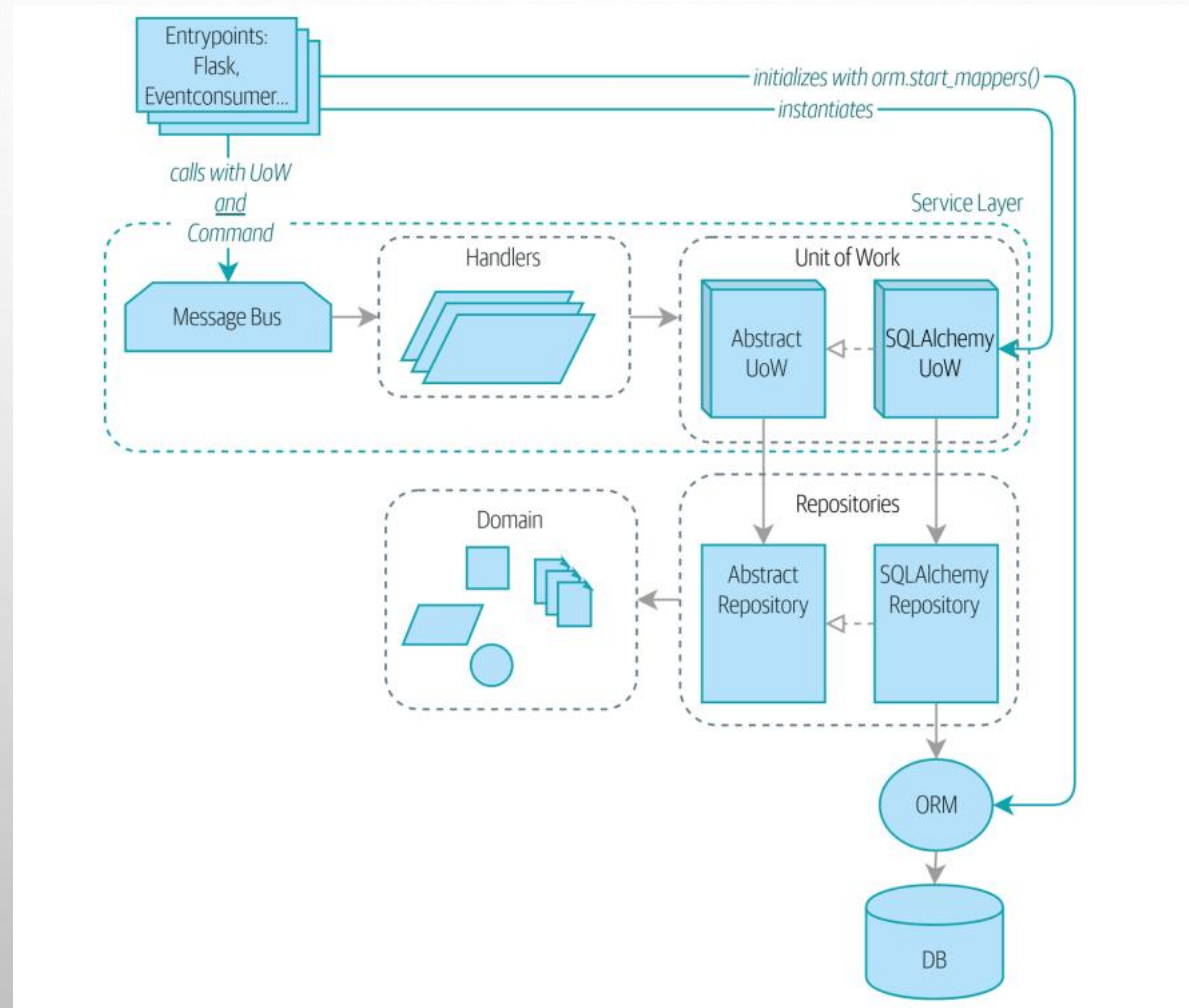
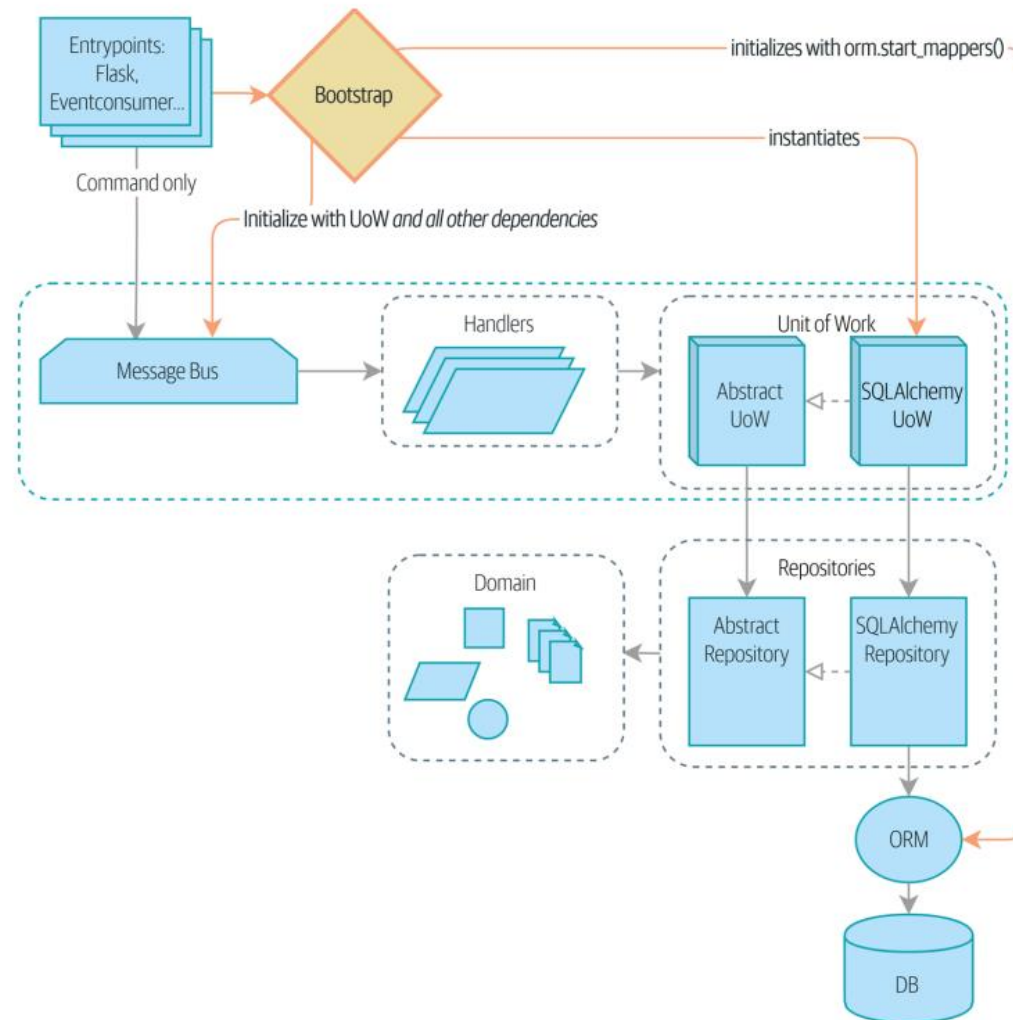


Figure below shows our bootstrapper taking over those responsibilities.



Dependency injection with Closures and Partials

```
# existing allocate function, with abstract uow dependency
def allocate(cmd: commands.Allocate, uow: unit_of_work.AbstractUnitOfWork):
    line = OrderLine(cmd.orderid, cmd.sku, cmd.qty)
    with uow:
        ...
```

Now

```
# bootstrap script prepares actual UoW
def bootstrap(..):
    uow = unit_of_work.SqlAlchemyUnitOfWork()

    # prepare a version of the allocate fn with UoW dependency captured in a closure
    allocate_composed = lambda cmd: allocate(cmd, uow)

    # or, equivalently (this gets you a nicer stack trace)
    def allocate_composed(cmd):
        return allocate(cmd, uow)

    # alternatively with a partial
    import functools
    allocate_composed = functools.partial(allocate, uow=uow)

# later at runtime, we can call the partial function, and it will have
# the UoW already bound
allocate_composed(cmd)
```

A Bootstrap Script [link](#)

- 1- Declare default dependencies but allow us to override them
- 2- Do the “init” stuff that we need to get our app started
- 3- Inject all the dependencies into our handlers
- 4- Give us back the core object for our app, the message bus

Using Bootstrap in Our Entrypoints

```
2
3 @app.route("/allocate", methods=["POST"])
4 def allocate_endpoint():
5     try:
6         cmd = commands.Allocate(
7             request.json["orderid"], request.json["sku"], request.json["qty"]
8         )
9         bus.handle(cmd)
10    except InvalidSku as e:
11        return {"message": str(e)}, 400
12    
13    return "OK", 202
```

