**7 Service implementation patterns for microservices**

This chapter covers

* How hexagonal architecture helps us design loosely coupled services
* Implementing the business layer for a microservice and implementing database models using SQLAlchemy
* Using the repository pattern to decouple the data layer from the business layer
* Using the unit of work pattern to ensure the atomicity of all transactions and using the dependency inversion principle to build software that is resilient to changes
* Using the inversion of control principle and the dependency injection pattern to decouple components that are dependent on each other

In this chapter, we’ll learn how to implement the business layer of a microservice. In previous chapters, we learned how to design and implement REST APIs. In those implementations, we used an in-memory representation of the resources managed by the service. We took that approach to keep the implementation simple and allow ourselves to focus on the API layer of the service.

In this chapter, we’ll complete our implementation of the orders service by adding a business layer and a data layer. The business layer will implement the capabilities of the orders service, such as taking orders, processing their payments, or scheduling them for production. For some of these tasks, the orders service requires the collaboration of other services, and we’ll learn useful patterns to handle those integrations.

The data layer will implement the data management capabilities of the service. The orders service owns and manages data about orders, so we’ll implement a persistent storage solution and an interface to it. However, as a gateway to users regarding the life cycle of an order, the orders service also needs to fetch data from other services—for example, to keep track of the order during production and delivery. We’ll also learn useful patterns to handle access to those services.

To articulate the implementation patterns of the service, we’ll also cover elements of the architectural layout required to keep all pieces of our microservices loosely coupled. Loose coupling will help us ensure that we can change the implementation of a specific component without having to make changes to other components that rely on it. It’ll also make our codebase generally more readable, maintainable, and testable. The code for this chapter is available in the ch07 directory in the repository provided with this book.

**7.1 Hexagonal architectures for microservices**

This chapter introduces the concept of hexagonal architecture and how we’ll apply it to the design of the orders service. In chapter 2, we introduced the three-tier architecture pattern to help us organize the components of our application in a modular and loosely coupled way. In this section, we’ll take this idea further by applying the concept of hexagonal architecture to our design.

In 2005, Alistair Cockburn introduced the concept of *hexagonal architecture*, also called the architecture of *ports and adapters*, as a way to help software developers structure their code into loosely coupled components.[**1**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1092720) As you can see in figure 7.1, the idea behind the hexagonal or ports-and-adapters architecture is that, in any application, there’s a core piece of logic that implements the capabilities of a service, and around that core we “attach” *adapters* that help the core communicate with external components. For example, a web API is an adapter that helps the core communicate with web clients over the internet. The same goes for a database, which is simply an external component that helps a service persist data. We should be able to swap the database if we want, and the service would still be the same. Therefore, the database is also an adapter.

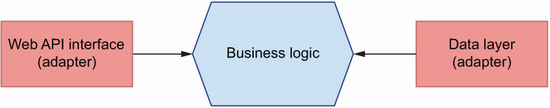


Figure 7.1 In hexagonal architecture, we distinguish a core layer in our application, the business layer, which implements the service’s capabilities. Other components, such as a web API interface or a database, are considered adapters that depend on the business layer.

How does this help us build loosely coupled services? Hexagonal architecture requires that we keep the core logic of the service and the logic for the adapters strictly separated. In other words, the logic that implements our web API layer shouldn’t interfere with the implementation of the core business logic. And the same goes for the database: regardless of the technology we choose, and its design and idiosyncrasies, it shouldn’t interfere with the core business logic. How do we achieve that? By building ports between the core business layer and the adapters. *Ports* are technology-agnostic interfaces that connect the business layer with the adapters. Later in this chapter, we’ll learn some design patterns that will help us design those ports or interfaces.

When working out the relationship between the core business logic and the adapters, we apply the dependency inversion principle, which states that (see figure 7.2 for clarification)

* High-level modules shouldn’t depend on low-level details. Instead, both should depend on abstractions, such as interfaces. For example, when saving data, we want to do it through an interface that doesn’t require understanding of the specific implementation details of the database. Whether it’s an SQL or a NoSQL database or a cache store, the interface should be the same.
* Abstractions shouldn’t depend on details. Instead, details should depend on abstractions.[**2**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1092733) For example, when designing the interface between the business layer and the data layer, we want to make sure that the interface doesn’t change based on the implementation details of the database. Instead, we make changes to the data layer to make it work with the interface. In other words, the data layer depends on the interface, not the other way around.

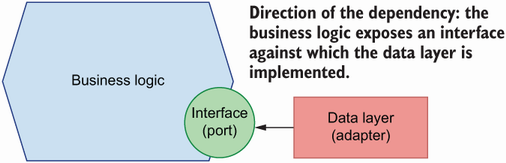


Figure 7.2 We apply the dependency inversion principle to determine which components drive the changes. In hexagonal architecture, this means that our adapters will depend on the interface exposed by the core business layer.

**DEFINITION** The *dependency inversion principle* encourages us to design our software against interfaces and to make sure we don’t create dependencies between the low-level details of our components.

The concept of dependency inversion often appears with the concepts of inversion of control and dependency injection. These are related but different concepts. As we’ll see in section 7.5, the inversion of control principle consists of supplying code dependencies through the execution context (also called the inversion of control container). To supply such dependencies, we can use the dependency injection pattern, which we’ll describe in section 7.5.

What does this mean in practice? It means we should make the adapters depend on the interface exposed by the core business logic. That is, it’s okay for our API layer to know about the core business logic’s interface, but it’s not okay for our business logic to know specific details of our API layer or low-level details of the HTTP protocol. The same goes for the database: our data layer should know how the application works and how to accommodate the application’s needs to our choice of storage technology, but the core business layer should know nothing specific about the database. Our business layer will expose an interface, and all other components will be implemented against it.

What exactly are we inverting with the dependency inversion principle? This principle inverts the way we think about software. Instead of the more conventional approach of building the low-level details of our software first, and then building interfaces on top of them, the dependency inversion principle encourages us to think of the interfaces first and then build the low-level details against them.[**3**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1092747)

As you can see in figure 7.3, when it comes to the orders service, we’ll have a core package that implements the capabilities of the service. This includes the ability to process an order and its payment, to schedule its production, or to keep track of its progress. The core service package will expose interfaces for other components of the application. Another package implements the web API layer, and our API modules will use functions and classes from the business layer interface to serve the requests of our users. Another package implements the data layer, which knows how to interact with the database and return business objects for the core business layer.

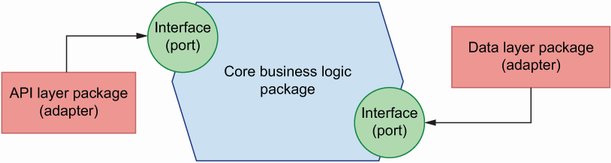


Figure 7.3 The orders service consist of three packages: the core business logic, which implements the capabilities of the service; an API layer, which allows clients to interact with the service over HTTP; and a data layer, which allows the service to interact with the database. The core business logic exposes interfaces against which the API layer and the data layer are implemented.

Now that we know how we are going to structure the application, it’s time to start implementing it! In the next section, we’ll set up the environment to start working on the service.

**7.2 Setting up the environment and the project structure**

In this section, we set up the environment to work on the orders service and lay out the high-level structure of the project. As in previous chapters, we’ll use Pipenv to manage our dependencies. Run the following commands to set up a Pipenv environment and activate it:

$ pipenv --three

$ pipenv shell

We’ll install our dependencies as we need in the following sections. Or if you prefer, copy the Pipfile and Pipfile.lock files from the GitHub repository under the ch07 folder and run pipenv install.

Our service implementation will live under a folder named orders, so go ahead and create it. To reinforce the separation of concerns between the core business layer and the API and database adapters, we’ll implement each of them in different directories, as shown in figure 7.4. The business layer will live under orders/orders\_service. Since the API layer is a web component, it will live under orders/web, which contains web adapters for the orders service. In this case, we are only including one type of web adapter, namely, a REST API, but nothing prevents you from adding a web adapter that returns dynamically rendered content from the server, as you would in a more traditional Django application.

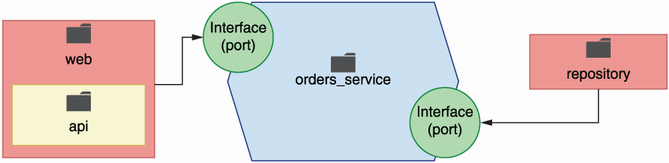


Figure 7.4 To reinforce the separation of concerns, we implement each layer of the application in different directories: orders\_service for the core business layer; repository for the data layer; and web/api for the API layer.

The data layer will live under orders/repository. “Repository” might look like an unlikely name for our data layer, but we’re choosing this name because we’ll implement the repository pattern to interface with our data. This concept will become clearer in section 7.4. In chapters 2 and 6 we covered the implementation of the API layer, so go ahead and copy over the files from the GitHub repository under ch07/order/web into your local directory. Notice that the API implementation has been adapted for this chapter.

Listing 7.1 High-level structure of the orders service

├── Pipfile ①

├── Pipfile.lock

└── orders ②

├── orders\_service ③

├── repository ④

└── web ⑤

├── api ⑥

│ ├── api.py

│ └── schemas.py

└── app.py ⑦

① Pipfile contains the list of dependencies.

② The full implementation of the orders service

③ The business layer

④ The data layer

⑤ Web adapters

⑥ REST API implementation

⑦ This file contains the instance of our web server object.

Since the folder structure has changed, the path to our FastAPI application object has also changed location, and therefore the command to run the API server is now

$ uvicorn orders.web.app:app --reload

Due to the new folder structure, a few import paths and file locations have also changed. For the full list of changes, please refer to the ch07 folder under the GitHub repository for this book.

Now that our project is set up and ready to go, it’s time to get on with the implementation. Move on to the next section to learn how to add database models to the service!

**7.3 Implementing the database models**

In the previous section, we learned how we’ll structure our project into three different layers: the core business layer, the API layer, and the data layer. This structure reinforces the separation of concerns among each layer, as recommended by the hexagonal architecture pattern that we learned in section 7.1. Now that we know how we’ll structure our code, it’s time to focus on the implementation. In this section, we’ll define the database models for the orders service; that is, we’ll design the database tables and their fields. We start our implementation from the database since it will facilitate the rest of the discussion in this chapter. In a real-world context, you might start with the business layer, mocking the data layer and iterating back and forth between each layer until you’re done with the implementation. Just bear in mind that the linear approach we take in this chapter is not meant to reflect the actual development process, but is instead intended to illustrate concepts that we want to explain.

To keep things simple in this chapter, we’ll use SQLite as our database engine. SQLite is a file-based relational database system. To use it, we don’t need to set up and run a server, as we would with PostgreSQL or MySQL, and there’s no configuration needed to start using it. Python’s core library has built-in support for interfacing with SQLite, which makes it a suitable choice for quick prototyping and experimentation before we are ready to move on to a production-ready database system.

We won’t manage our connection to the database and our queries manually. That is, we won’t be writing our own SQL statements to interact with the database. Instead, we’ll use SQLAlchemy—by far the most popular ORM (object relational mapper) in the Python ecosystem. An ORM is a framework that implements the data mapper pattern, which allows us to map the tables in our database to objects.

**DEFINITION** A *data mapper* is an object wrapper around database tables and rows. It encapsulates database operations in the form of class methods, and it allows us to access data fields through class attributes.[**4**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1092805)

As you can see in figure 7.5, using an ORM makes it easier to manage our data since it gives us a class interface to the tables in the database. This allows us to leverage the benefits of object-oriented programming, including the ability to add custom methods and properties to our database models that enhance their functionality and encapsulate their behavior.

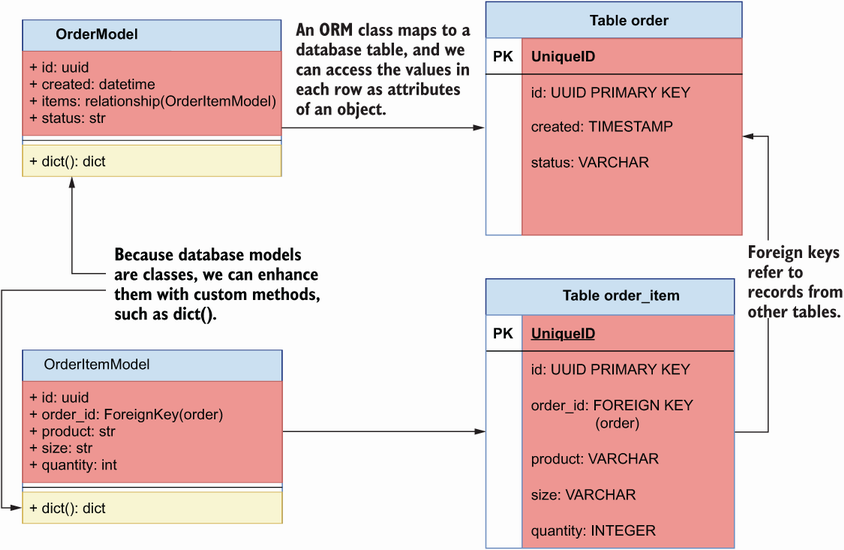


Figure 7.5 Using an ORM, we can implement our data models as classes that map to database tables. Since the models are classes, we can enhance them with custom methods to add new functionality.

Over time, our database models will change, and we need to be able to keep track of those changes. Changing the schema of our database is called a *migration*. As our database evolves, we’ll accumulate more and more migrations. We need to keep track of our migrations, since they allow us to reliably replicate the database schema in different environments and to roll out database changes to production with confidence. To manage this complex task, we’ll use Alembic. Alembic is a schema migration library that integrates seamlessly with SQLAlchemy.

Let’s start by installing both libraries by running the following command:

$ pipenv install sqlalchemy alembic

Before we start working on our database models, let’s set up Alembic. (For additional help, please check out my video tutorial about setting up Alembic with SQLAlchemy at <https://youtu.be/nt5sSr1A_qw>.) Run the following command to create a migrations folder, which will contain the history of all migrations in our database:

$ alembic init migrations

This creates a folder called migrations, which comes with a configuration file called env.py and a versions/ directory. The versions/ directory will contain the migration files. The setup command also creates a configuration file called alembic.ini. To make Alembic work with an SQLite database, open alembic.ini, find a line that contains a declaration for the sqlalchemy.url variable, and replace it with the following content:

sqlalchemy.url = sqlite:///orders.db

**COMMIT THE FILES GENERATED BY ALEMBIC** The migrations folder contains all the information required to manage our database schema changes, so you should commit this folder, as well as alembic.ini. This will allow you to replicate the database setup in new environments.

In addition, open migrations/env.py and find the lines with this content:[**5**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1092825)

# from myapp import mymodel

# target\_metadata = mymodel.Base.metadata

target\_metadata = None

Replace them with the following content:

from orders.repository.models import Base

target\_metadata = Base.metadata

By setting target\_metadata to our Base model’s metadata, we make it possible for Alembic to load our SQLAlchemy models and generate database tables from them. Next, we’ll implement our database models. Before we jump into the implementation, let’s pause for a moment to think about how many models we’ll need and the properties we should expect each model to have. The core object of the orders service is the order. Users place, pay, update, or cancel orders. Orders have a life cycle, and we’ll keep track of it through a status property. We’ll use the following list of properties to define our order model:

* *ID*—Unique ID for the order. We’ll give it the format of a Universally Unique Identifier (UUID). Using UUIDs instead of incremental integers is quite common these days. UUIDs work well in distributed systems, and they help to hide information about the number of orders that exist in the database from our users.
* *Creation date*—When the order was placed.
* *Items*—The list of items included in the order and the amount of each product. Since an order can have any number of items linked to it, we’ll use a different model for items, and we’ll create a one-to-many relationship between the order and the items.
* *Status*—The status of the order throughout the system. An order can have the following statuses:
  + *Created*—The order has been placed.
  + *Paid*—The order has been successfully paid.
  + *Progress*—The order is being produced in the kitchen.
  + *Cancelled*—The order has been cancelled.
  + *Dispatched*—The order is being delivered to the user.
  + *Delivered*—The order has been delivered to the user.
* *Schedule ID*—The ID of the order in the kitchen service. This ID is created by the kitchen service after scheduling the order for production, and we’ll use it to keep track of its progress in the kitchen.
* *Delivery ID*—The ID of the order in the delivery service. This ID is created by the delivery service after scheduling it for dispatch, and we’ll use it to keep track of its progress during delivery.

When users place an order, they add any number of items to the order. Each item contains information about the product selected by the user, the size of the product, and the amount of it that the user wishes to purchase. There’s a one-to-many relationship between orders and items, and therefore we’ll implement a model for items and link them with a foreign key relationship. The item model will have the following list of attributes:

* *ID*—A unique identifier for the item in UUID format.
* *Order ID*—A foreign key representing the ID of the order the item belongs to. This is what allows us to connect items and orders that belong together.
* *Product*—The product selected by the user.
* *Size*—The size of the product.
* *Quantity*—The amount of the product that the user wishes to purchase.

Our SQLAlchemy models will live under the orders/repository folder, which we created to encapsulate our data layer, in a file called orders/repository/models.py. We’ll use these classes to interface with the database and rely on SQLAlchemy to translate these models into their corresponding database tables behind the scenes. Listing 7.2 shows the definition of the database models for the orders service. First, we create a declarative base model by using SQLALchemy’s declarative\_base() function. The declarative base model is a class that can map ORM classes to database tables and columns, and therefore all our database models must inherit from it. We map class attributes to specific database columns by setting them to instances of SQLAlchemy’s Column class.

To map an attribute to another model, we use SQLAlchemy’s relationship() function. In listing 7.2, we use relationship() to create a one-to-many relationship between OrderModel’s items attribute and the OrderItemModel model. This means that we can access the list of items in an order through OrderModel’s items attribute. Each item also maps to the order it belongs to through the order\_id property, which is defined as a foreign key column. Furthermore, relationship()’s backref argument allows us to access the full order object from an item directly through a property called order.

Since we want our IDs to be in UUID format, we create a function that SQLAlchemy can use to generate the value. If we later switch to a database engine with built-in support for generating UUID values, we’ll leave it to the database to generate the IDs. Each database model is enhanced with a dict() method, which allows us to output the properties of a record in dictionary format. Since we’ll use this method to translate database models to business objects, the dict() method only returns the properties relevant to the business layer.

Listing 7.2 SQLAlchemy models for the orders service

# file: orders/repository/models.py

import uuid

from datetime import datetime

from sqlalchemy import Column, Integer, String, ForeignKey, DateTime

from sqlalchemy.ext.declarative import declarative\_base

from sqlalchemy.orm import relationship

Base = declarative\_base() ①

def generate\_uuid(): ②

return str(uuid.uuid4())

class OrderModel(Base): ③

\_\_tablename\_\_ = 'order' ④

id = Column(String, primary\_key=True, default=generate\_uuid) ⑤

items = relationship('OrderItemModel', backref='order') ⑥

status = Column(String, nullable=False, default='created')

created = Column(DateTime, default=datetime.utcnow)

schedule\_id = Column(String)

delivery\_id = Column(String)

def dict(self): ⑦

return {

'id': self.id,

'items': [item.dict() for item in self.items], ⑧

'status': self.status,

'created': self.created,

'schedule\_id': self.schedule\_id,

'delivery\_id': self.delivery\_id,

}

class OrderItemModel(Base):

\_\_tablename\_\_ = 'order\_item'

id = Column(String, primary\_key=True, default=generate\_uuid)

order\_id = Column(Integer, ForeignKey('order.id'))

product = Column(String, nullable=False)

size = Column(String, nullable=False)

quantity = Column(Integer, nullable=False)

def dict(self):

return {

'id': self.id,

'product': self.product,

'size': self.size,

'quantity': self.quantity

}

① We create our declarative base model.

② Custom function to create random UUIDs for our models

③ All our models must inherit from Base.

④ Name of the table that maps to this model

⑤ Every class property maps to a database column by using the Column class.

⑥ We use relationship() to create a one-to-many relationship with the OrderItemModel model.

⑦ Custom method to render our objects as Python dictionaries

⑧ We call dict() on each item to get its dictionary representation.

To apply the models to the database, run the following command from the ch07 directory:

$ PYTHONPATH=`pwd` alembic revision --autogenerate -m "Initial migration"

This will create a migration file under migrations/versions. We set the PYTHONPATH environment variable to the current directory using the pwd command so that Python looks for our models relative to this directory. You should commit your migration files and keep them in your version control system (e.g., a Git repository) since they’ll allow you to re-create your database for different environments. You can look in those files to understand the database operations that SQLAlchemy will perform to apply the migrations. To apply the migrations and create the schemas for these models in the database, run the following command:

$ PYTHONPATH=`pwd` alembic upgrade heads

This will create the desired schemas in our database. Now that our database models are implemented and our database contains the desired schemas, it’s time to move on to the next step. Go to the next section to learn about the repository pattern!

**7.4 Implementing the repository pattern for data access**

In the previous section, we learned to design the database models for the orders service and to manage changes to the database schema through migrations. With our database models ready, we can interact with the database to create orders and manage them. Now we have to decide how we make the data accessible to the business layer. In this section, we’ll first discuss different strategies to connect the business layer with the data layer, and we’ll learn what the repository pattern is and how we can use it to create an interface between the business layer and the database. Then we’ll move on to implementing it.

**7.4.1 The case for the repository pattern: What is it, and why is it useful?**

In this section, we discuss different strategies for interfacing with the database from the business layer, and we introduce the repository pattern as a strategy that helps us decouple the business layer from the implementation details of the database.

As shown in figure 7.6, a common strategy to enable interactions between the business layer and the database is to use the database models directly within the business layer. Our database models already contain data about the orders, so we could enhance them with methods that implement business capabilities. This is called the *active record pattern*, which represents objects that carry both data and domain logic.[**6**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1092953) This pattern is useful when we have one-to-one mapping between service capabilities and database operations, or when we don’t need the collaboration of multiple domains.

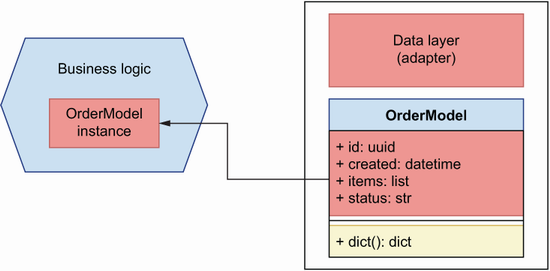


Figure 7.6 A common approach to enable interactions between the data layer and the business layer is by using the database models directly in the business layer.

This approach works for simple cases; however, it couples the implementation of the business layer to the database and to the ORM framework of choice. What happens if we want to change the ORM framework later on, or if we want to switch to a different data storage technology that doesn’t involve SQL? In those cases, we’d have to make changes to our business layer. This breaks the principles we introduced in section 7.1. Remember, the database is an adapter that the orders service uses to persist data, and the implementation details of the database should not leak into the business logic. Instead, data access will be encapsulated by our data access layer.

To decouple the business layer from the data layer, we’ll use the repository pattern. This pattern gives us an in-memory list interface of our data. This means that we can get, add, or delete orders from the list, and the repository will take care of translating these operations into database-specific commands. Using the repository pattern means the data layer exposes a consistent interface to the business layer to interact with the database, regardless of the database technology we use to store our data. Whether we use an SQL database such as PostgreSQL, a NoSQL database like MongoDB, or an in-memory cache such as Redis, the repository pattern’s interface will remain the same and will encapsulate whichever specific operations are required to interact with the database. Figure 7.7 illustrates how the repository pattern helps us invert the dependency between the data layer and the business layer.

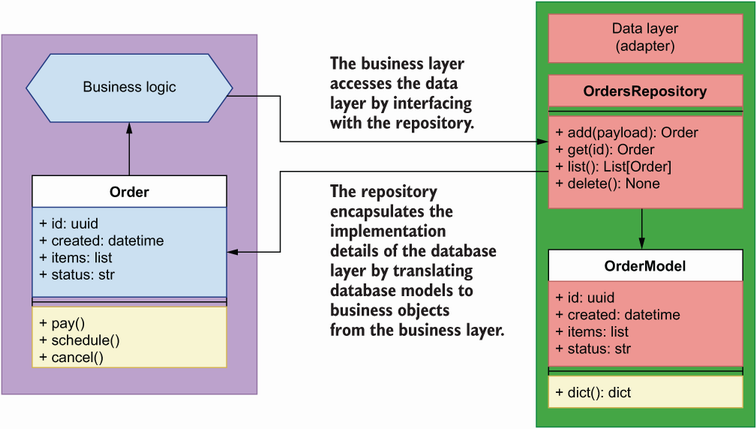


Figure 7.7 The repository pattern encapsulates the implementation details of the data layer by exposing an in-memory list interface to the business layer, and it translates database models to business objects.

**DEFINITION** The *repository pattern* is a software development pattern that provides an in-memory list interface to our data store. This helps us decouple our components from the low-level implementation details of the database. The repository takes care of managing interactions with the database and provides a consistent interface to our components, regardless of the database technology used. This allows us to change the database system without having to change our core business logic.

Now that we know how we can use the repository pattern to allow the business layer to interface with the database while decoupling its implementation from low-level details of the database, we’ll learn to implement the repository pattern.

**7.4.2 Implementing the repository pattern**

How do we implement the repository pattern? We can use different approaches to this as long as we meet the following constraint: none of the operations carried out by the repository can be committed by the repository. What does this mean? It means that when we add an order object to the repository, the repository will add the order to a database session, but it will not commit the changes. Instead, it will be the responsibility of the consumer of OrdersService (i.e., the API layer) to commit the changes. Figure 7.8 illustrates this process.

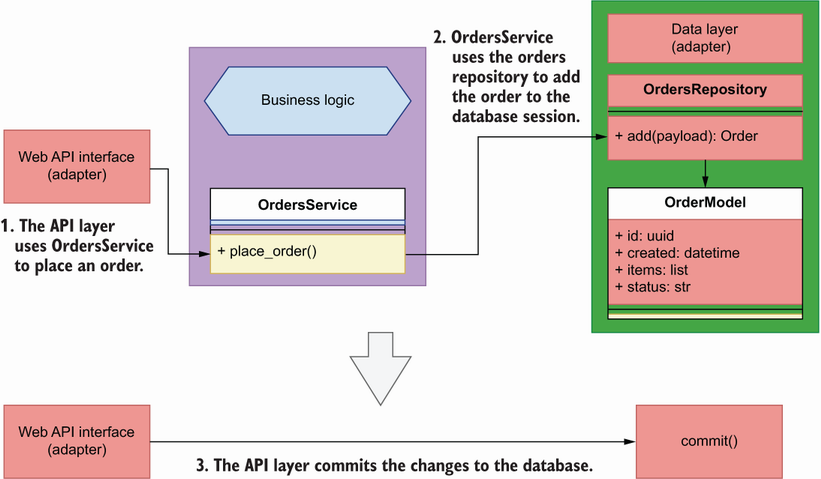


Figure 7.8 Using the repository pattern, the API layer uses the place\_order() capability of OrdersService to place an order. To place the order, OrdersService interfaces with the orders repository to add the order to the database. Finally, the API layer must commit the changes to persist them in the database.

Why can’t we commit database changes within the repository? First, because the repository acts just like an in-memory list representation of our data, and as such it doesn’t have a concept of database sessions and transactions; second, because the repository is not the right place to execute a database transaction. Instead, the context in which the repository is invoked provides the right context for executing database transactions. In many cases, our applications will execute multiple operations that involve one or more repositories and also call to other services. For example, figure 7.9 shows the number of operations involved in processing a payment:

1. The API layer receives the request from the user and uses the OrdersService’s pay\_order() method to process the request.
2. OrdersService talks to the payments service to process the payment.
3. If the payment is successful, OrdersService schedules the order with the kitchen service.
4. OrdersService updates the state of the order in the database using the orders repository.
5. If all the previous operations were successful, the API layer commits the transaction in the database; otherwise, it rolls back the changes.

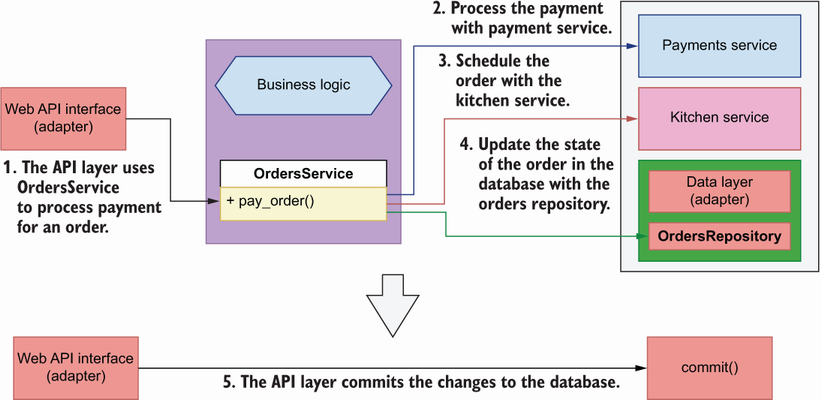


Figure 7.9 In some situations, OrdersService has to interface with multiple repositories or services to perform an operation. In this example, OrdersService interfaces with the payments service to process a payment, then with the kitchen service to schedule the order for production, and finally updates the status of the order through the orders repository. All these operations must succeed or fail together, and it’s the responsibility of the API layer to commit or rollback accordingly.

These steps can be taken synchronously, one after the other, or asynchronously, in no specific order, but regardless of the approach, all steps must succeed or fail all together. As the unit of execution context, it’s the responsibility of the API layer to ensure that all changes are committed or rolled back as required. In section 7.6, we’ll learn how exactly the API layer controls the database session and commits the transactions.

At a minimum, a repository pattern implementation consists of a class that exposes a get() and an add() method, respectively, to be able to retrieve and add objects to the repository. For our purposes, we’ll also implement the following methods: update(), delete(), and list(). This will simplify the CRUD interface of the repository.

The following question bears some consideration in this context: when we fetch data through the repository, what kind of object should the repository return? In many implementations, you’ll see repositories returning instances of the database models (i.e., the classes defined in orders/repository/models.py). We won’t do that in this chapter. Instead, we’ll return objects that represent orders from the business layer domain. Why is it a bad idea to return instances of the database models through the repository? Because it defeats the purpose of the repository, which is to decouple the business layer from the data layer. Remember, we may want to change our persistence storage technology or our ORM framework. If that happens, the database classes we implemented in section 7.2 will no longer exist, and there’s no guarantee that a new framework would allow us to return objects with the same interfaces. For this reason, we don’t want to couple our business layer with them. Figure 7.10 illustrates the relationship between the business layer and the orders repository.

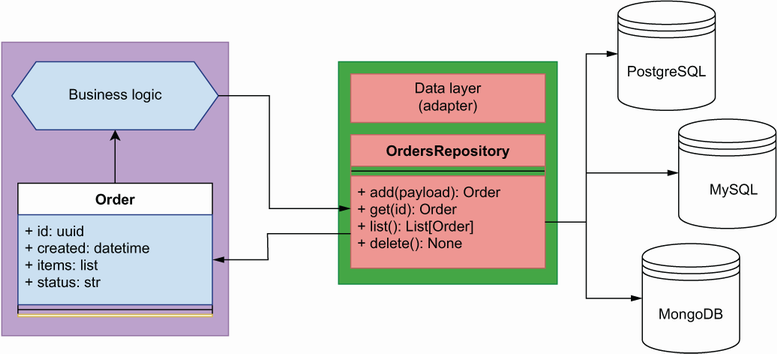


Figure 7.10 The repository pattern encapsulates the implementation details of the persistent storage technology used to manage our data. Our business layer only ever deals with the repository, and therefore we are free to change our persistent storage solution to a different technology without affecting our core application implementation.

Our orders repository implementation will live under orders/repository/orders\_ repository.py. Listing 7.3 shows the implementation of the orders repository. It takes one required argument that represents the database session. Objects are added and deleted from the database session. The add() and update() methods take payloads that represent orders in the form of a Python dictionary. Our payloads are fairly simple, so a dictionary is sufficient here, but if we have more complex payloads, we should consider using objects instead.

With the exception of the delete() method, all methods of the repository return Order objects from the business layer (see section 7.5 for Order’s implementation details). To create instances of Order, we pass dictionary representations of the SQLAlchemy models using our custom dict() method from listing 7.2. In the add() method, we also include a pointer to the actual SQLAlchemy model through Order’s order\_ parameter. As we’ll see in section 7.5, this pointer will help us access the order’s ID after committing the database transaction.

OrdersRepository’s get(), update(), and delete() methods use the same logic to pull a record before returning, updating, or deleting it, so we define a common \_get() method that knows how to obtain a record given an ID and optional filters. We fetch the record using the first() method of SQLAlchemy’s query object. first() returns an instance of the record if it exists, and otherwise it returns None. Alternatively, it’s also possible to use the one() method, which raises an error if the record doesn’t exist. \_get() returns a database record, so it’s not meant to be used by the service layer, and we signal that by prefixing the method’s name with an underscore.

The list() method accepts a limit parameter and optional filters. We build our query dynamically using SQLAlchemy’s query object. We also leverage SQLAlchemy’s filter\_by() method to include additional filters in the query as keyword arguments, and we limit the query results by adding the limit parameter. Finally, we transform the database records into Order objects for consumption by the business layer by using the dict() method we implemented in listing 7.2.

The repository implementation is tightly coupled to the methods of SQLAlchemy’s Session object, but it also encapsulates these details, and to the business layer the repository appears as an interface to which we submit IDs and payloads, and we get Order objects in return. This is the point of the repository: to encapsulate and hide the implementation details of the data layer from the business layer. This means that if we switch to a different ORM framework, or to a different database system, we only need to make changes to the repository.

Listing 7.3 Orders repository

# file: orders/repository/orders\_repository.py

from orders.orders\_service.orders import Order

from orders.repository.models import OrderModel, OrderItemModel

class OrdersRepository:

def \_\_init\_\_(self, session): ①

self.session = session

def add(self, items):

record = OrderModel(

items=[OrderItemModel(\*\*item) for item in items]

) ②

self.session.add(record) ③

return Order(\*\*record.dict(), order\_=record) ④

def \_get(self, id\_): ⑤

return (

self.session.query(OrderModel)

.filter(OrderModel.id == str(id\_))

.filter\_by(\*\*filters)

.first()

) ⑥

def get(self, id\_):

order = self.\_get(id\_) ⑦

if order is not None: ⑧

return Order(\*\*order.dict())

def list(self, limit=None, \*\*filters): ⑨

query = self.session.query(OrderModel) ⑩

if 'cancelled' in filters: ⑪

cancelled = filters.pop('cancelled')

if cancelled:

query = query.filter(OrderModel.status == 'cancelled')

else:

query = query.filter(OrderModel.status != 'cancelled')

records = query.filter\_by(\*\*filters).limit(limit).all()

return [Order(\*\*record.dict()) for record in records] ⑫

def update(self, id\_, \*\*payload):

record = self.\_get(id\_)

if 'items' in payload: ⑬

for item in record.items:

self.session.delete(item)

record.items = [

OrderItemModel(\*\*item) for item in payload.pop('items')

]

for key, value in payload.items(): ⑭

setattr(record, key, value)

return Order(\*\*record.dict())

def delete(self, id\_):

self.session.delete(self.\_get(id\_)) ⑮

① The repository’s initializer method requires a session object.

② When creating a record for an order, we also create a record for each item in the order.

③ We add the record to the session object.

④ We return an instance of the Order class.

⑤ Generic method to retrieve a record by ID

⑥ We fetch the record using SQLAlchemy’s first() method.

⑦ We retrieve a record using \_get().

⑧ If the order exists, we return an Order object.

⑨ list() accepts a limit parameter and other optional filters.

⑩ We build our query dynamically.

⑪ We filter by whether an order is cancelled using the SQLAlchemy’s filter() method.

⑫ We return a list of Order objects.

⑬ To update an order, we first delete the items linked to the order and then create new items from the supplied payload.

⑭ We dynamically update the database object using the setattr() function.

⑮ To delete a record, we call SQLAlchemy’s delete() method.

This completes the implementation of our data layer. We have implemented a persistent storage solution with the help of SQLAlchemy, and we have encapsulated the details of this solution with the help of the repository pattern. It’s now time to work on the business layer and see how it will interact with the repository!

**7.5 Implementing the business layer**

We’ve done a lot of work designing the database models for the orders service and using the repository pattern to build the interface to the data. It’s now time to focus on the business layer! In this section, we’ll implement the business layer of the orders service. That’s the core of the hexagon we introduced in section 7.1 and illustrated in figure 7.1, which is reproduced here as figure 7.11 for your convenience. The business layer implements the service’s capabilities. What are the business capabilities of the orders service? From the analysis in chapter 3 (section 3.4.2), we know that the orders service allows users of the platform to place their orders and manage them.

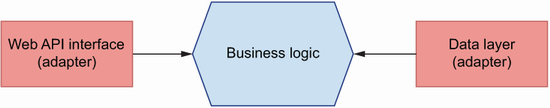


Figure 7.11 In hexagonal architecture, we distinguish a core layer in our application, the business layer, which implements the service’s capabilities. Other components, such as a web API interface or a database, are considered adapters that depend on the business layer.

As illustrated in figure 7.12, the orders service manages the life cycle of an order through integrations with other services. The following list describes the capabilities of the orders service and highlights integrations with other services (refer to figure 7.9 for further clarification):

* *Place orders*—Creates a record of an order in the system. The order won’t be scheduled in the kitchen until the user pays for it.
* *Process payments*—Processes payment for an order with the help of the payments service. If the payments service confirms the payment is successful, the orders service schedules the order for production with the kitchen service.
* *Update orders*—Users can update their order any time to add or remove items from it. To confirm a change, a new payment must be made and processed with the help of the payments service.
* *Cancel orders*—Users can cancel their orders anytime. Depending on the status of the order, the orders service will communicate with the kitchen or the delivery service to cancel the order.
* *Schedule order for production in the kitchen*—After payment, the orders service schedules the order for production in the kitchen with the help of the kitchen service.
* *Keep track of orders’ progress*—Users can keep track of their orders’ status through the orders service. Depending on the status of the order, the orders service checks with the kitchen or the delivery service to get updated information about the state of the order.

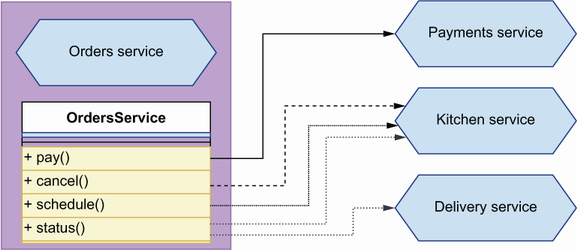


Figure 7.12 In order to perform some of its functions, the orders service needs to interact with orders services. For example, to process payments, it must interact with the payments service, and to schedule an order for production, it must interact with the kitchen service.

What’s the best way to model these actions in our business layer? We can use different approaches, but to make it easy for other components to interact with the business layer, we’ll expose a single unified interface through a class called OrdersService. We’ll define this class under orders/orders\_service/orders\_service.py. To fulfill its duties, OrdersService uses the orders repository to interface with the database. We could let OrdersService import and initialize the orders repository as in the following code:

from repository.orders\_repository import OrdersRepository

class OrdersService:

def \_\_init\_\_(self):

self.repository = OrdersRepository()

However, doing this would place too much responsibility on the orders service since it would need to know how to configure the orders repository. It would also tightly couple the implementation of the orders repository and the orders service, and we wouldn’t be able to use different repositories if we needed to. As you can see in figures 7.13 and 7.14, a better approach is to use dependency injection in combination with the inversion of control principle.

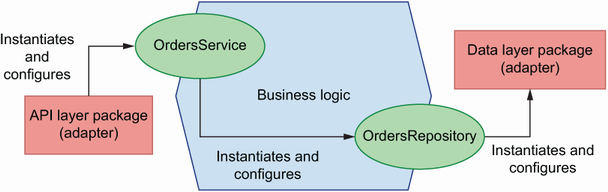


Figure 7.13 In conventional software design, dependencies follow a linear relationship, and each component is responsible for instantiating and configuring its own dependencies. In many cases, this couples our components to low-level implementation details in their dependencies.

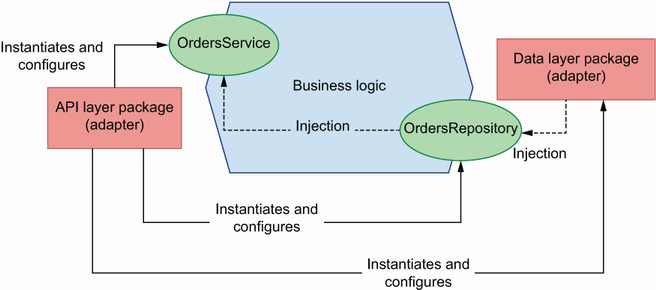


Figure 7.14 With inversion of control, we decouple components from their dependencies by supplying them at runtime using methods such as dependency injection. In this approach, it’s the responsibility of the context to provide correctly configured instances of the dependencies. The solid lines show relationships of dependency, while the dotted lines show how dependencies are injected.

**DEFINITION** *Inversion of control* is a software development principle that encourages us to decouple our components from their dependencies by supplying them at runtime. This allows us to control how the dependencies are supplied. One popular pattern to accomplish this is dependency injection. The context in which the dependencies are instantiated and supplied is called an *inversion of control container*. In the orders service, a suitable inversion of control container is the request object since most operations are specific to the context of a request.

The inversion of control principle states that we should decouple the dependencies in our code by letting the execution context supply those dependencies at runtime. This means that, instead of letting the orders service import and instantiate the orders repository, we should supply the repository at runtime. How do we do that? We can use different patterns to supply dependencies to our code, but one of the most popular, due to its simplicity and effectiveness, is dependency injection.

**DEFINITION** *Dependency injection* is a software development pattern whereby we supply code dependencies at runtime. This helps us decouple our components from the specific implementation details of the code they depend on, since they don’t need to know how to configure and instantiate their dependencies.

To make the orders repository injectable into the orders service, we parameterize it:

class OrdersService:

def \_\_init\_\_(self, orders\_repository):

self.orders\_repository = orders\_repository

It’s now the responsibility of the caller to instantiate and configure the orders repository correctly. As you can see in figure 7.11, this has a very desirable outcome: depending on the context, we can supply different implementations of the repository or add different configurations. This makes the orders service easier to use in different contexts.[**7**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1093145)

Listing 7.4 shows the interface exposed by OrdersService. The class initializer takes an instance of the orders repository as a parameter to make it injectable. As per the inversion of control principle, when we integrate OrdersService with the API layer, it will be the responsibility of the API to get a valid instance of the orders repository and pass it to OrdersService. This approach is convenient, since it allows us to swap repositories at will when necessary, and it’ll make it very easy to write our tests in the next chapter.

Listing 7.4 Interface of the OrdersService class

# file: orders/orders\_service/orders\_service.py

class OrdersService:

def \_\_init\_\_(self, orders\_repository):

self.orders\_repository = orders\_repository

def place\_order(self, items):

pass

def get\_order(self, order\_id):

pass

def update\_order(self, order\_id, items):

pass

def list\_orders(self, \*\*filters):

pass

def pay\_order(self, order\_id):

pass

def cancel\_order(self, order\_id):

pass

Some of the actions listed under OrdersService, such as payment or scheduling, take place at the level of individual orders. Since orders contain data, it will be useful to have a class that represents orders and has methods to perform tasks related to an order. Within the context of the orders service, an order is a core object of the orders domain. In domain-driven design (DDD), we call these objects *domain objects*. These are the objects returned by the orders repository. We’ll implement our Order class under orders/orders\_service/orders.py. Listing 7.5 shows a preliminary implementation of the Order class.

In addition to the Order class, listing 7.5 also provides an OrderItem class that represents each of the items in an order. We’ll use the Order class to represent orders before and after saving them to the database. Some of the properties of an order, such as the creation time or its ID, are set by the data layer and can be known only after the changes to the database have been committed. As we explained in section 7.4, committing changes is out of the scope of a repository, which means that when we add an order to the repository, the returned object won’t have those properties. The order’s ID and its creation time become available through the order’s database record after committing the transaction. For this reason, Order’s initializer binds the order’s ID, creation time, and status as private properties with a leading underscore (like in self.\_id), and we use the order\_ parameter in the Order class to hold a pointer to the order’s database record. If we retrieve the details of an order already saved to the database, \_id, \_created, and \_status will have their corresponding values in the initializer; otherwise, they’ll be None and we’ll pull their values from order\_. That’s why we define Order’s id, created, and status properties using the property() decorator, since it allows us to resolve their value depending on the state of the object. This is the only degree of coupling we’ll allow between the business layer and the data layer. And to make sure this dependency can be easily removed if we ever have to, we’re setting order\_ to None by default.

Listing 7.5 Implementation of the Order business object class

# file: orders/orders\_service/orders.py

class OrderItem: ①

def \_\_init\_\_(self, id, product, quantity, size): ②

self.id = id

self.product = product

self.quantity = quantity

self.size = size

class Order:

def \_\_init\_\_(self, id, created, items, status, schedule\_id=None,

delivery\_id=None, order\_=None): ③

self.\_id = id ④

self.\_created = created

self.items = [OrderItem(\*\*item) for item in items] ⑤

self.\_status = status

self.schedule\_id = schedule\_id

self.delivery\_id = delivery\_id

@property

def id(self): ⑥

return self.\_id or self.\_order.id

@property

def created(self):

return self.\_created or self.\_order.created

@property

def status(self):

return self.\_status or self.\_order.status

① Business object that represents an order item

② We declare the parameters of OrderItem’s initializer method.

③ The order\_ parameter represents a database model instance.

④ Since we resolve the ID dynamically, we store the provided ID as a private property.

⑤ We build an OrderItem object for each order item.

⑥ We resolve the ID dynamically using the property() decorator.

In addition to holding data about an order, the Order class also needs to handle tasks such as cancelling, paying, and scheduling an order. To fulfill those tasks, we must interface with external dependencies, such as the kitchen and payments services. As we explained in section 7.1, the goal of hexagonal architecture is to encapsulate access to external dependencies through adapters. However, to keep things simple in this chapter, we’ll implement the external API calls within the Order class. A good adapter pattern for encapsulating external API calls is the *facade pattern*.[**8**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1093223) Before we proceed with the implementation, we should know what those API calls look like.

To build the integration between the orders service and the kitchen and payments services, we’d want to run the kitchen and payments services and see how they work. However, we don’t need to run the actual services. The folder for this chapter in the GitHub repository for this book contains three OpenAPI files: one for the orders API (ch07/oas.yaml), one for the kitchen API (ch07/kitchen.yaml), and one for the payments API (ch07/payments.yaml). kitchen.yaml and payments.yaml tell us how the kitchen and payments APIs work, and that’s all the information we need to build our integration. Make sure to pull the kitchen.yaml and payments.yaml files from GitHub to be able to work with the following examples.

As it turns out, we can also use the kitchen and payments API specifications to simulate their behavior using mock servers. API mock servers replicate the server behind the APIs, validating our requests and returning valid responses. We’ll use Prism CLI (<https://github.com/stoplightio/prism>), a library built and maintained by Stoplight, to mock the API server for the kitchen and payments services. Prism is a Node.js library, but don’t worry, it’s just a CLI tool; you don’t need to know any JavaScript to use it. To install the library, run the following command:

$ yarn add @stoplight/prism-cli

**DEALING WITH ERRORS RUNNING PRISM** You may run into errors when running Prism. A common error is not having a compatible version of Node.js. I recommend you install nvm to manage your Node versions and use the latest stable version of Node to run Prism. Also, make sure the port you select to run Prism is available.

This command will create a node\_modules/ folder within your application folder, where Prism and all its dependencies will be installed. You don’t want to commit this folder, so make sure you add it to your .gitignore file. You’ll also see a new file called package.json, and another one called yarn.lock within your application directory. These are the files you want to commit since they’ll allow you to re-create the same node\_modules/ directory in any other environment.

To see Prism in action with the kitchen API, run the following command:

$ ./node\_modules/.bin/prism mock kitchen.yaml --port 3000

This will start a server on port 3000 that runs a mock service for the kitchen API. To get a taste of what we can do with it, run the following command to hit the GET /kitchen/schedules endpoint, which returns a list of schedules:

$ curl http://localhost:3000/kitchen/schedules

**DISPLAY JSON IN THE TERMINAL LIKE A PRO WITH JQ** When outputting JSON to the terminal, either using cURL to interact with an API or catting a JSON file, I recommend you use JQ—a command-line utility that parses the JSON and produces a beautiful display. You can use JQ like this: curl http://localhost: 3000/kitchen/schedules | jq.

You’ll see that the mock server started by Prism is able to return a perfectly valid payload representing a list of schedules. Impressive, to say the least! Now that we know how to run mock servers for the kitchen and payments APIs, let’s analyze the requirements of the API integrations with them:

* *Kitchen service* *(kitchen.yaml)*—To schedule an order with the kitchen service, we must call the POST /kitchen/schedules endpoint with a payload containing the list of items in the order. In the response to this call, we’ll find the schedule\_id, which we can use to keep track of the state of the order.
* *Payments service* *(payments.yaml)*—To process the payment for an order, we must call the POST /payments endpoint with a payload containing the ID of the order. This is a mock endpoint for integration testing purposes.

Before we can cancel an order, we need to check its status. If the order is scheduled for production, we must hit the POST /kitchen/schedules/{schedule\_id}/cancel endpoint to cancel the schedule. If the order is out for delivery, we won’t allow users to cancel the order, and therefore we raise an exception.

To implement the API integrations, we’ll use the popular Python requests library. Run the following command to install the library with pipenv:

$ pipenv install requests

Listing 7.6 extends the implementation of the Order class by adding methods that implement API calls to the kitchen and payment services. For testing purposes, we’re expecting the kitchen API to run on port 3001 and the payments service to run on port 3000. You can accomplish this by running the following commands:

$ ./node\_modules/.bin/prism mock kitchen.yaml --port 3000

$ ./node\_modules/.bin/prism mock payments.yaml --port 3001

In each API call, we check that the response contains the expected status code, and if it doesn’t, we raise a custom APIIntegrationError exception. Also, if a user tries to perform an invalid action, such as cancelling an order when it’s already out for delivery, we raise an InvalidActionError exception.

Listing 7.6 Encapsulating per-order capabilities within the Order class

# file: orders/orders\_service/orders.py

import requests

from orders.orders\_service.exceptions import (

APIIntegrationError, InvalidActionError

)

...

class Order:

...

**def cancel(self):**

**if self.status == 'progress':**  ①

**kitchen\_base\_url = "http://localhost:3000/kitchen"**

**response = requests.post(**

**f"{kitchen\_base\_url}/schedules/{self.schedule\_id}/cancel",**

**json={"order": [item.dict() for item in self.items]},**

**)**

**if response.status\_code == 200:**  ②

**return**

**raise APIIntegrationError(**  ③

**f'Could not cancel order with id {self.id}'**

**)**

**if self.status == 'delivery':**  ④

**raise InvalidActionError(**

**f'Cannot cancel order with id {self.id}'**

**)**

**def pay(self):**

**response = requests.post(**  ⑤

**'http://localhost:3001/payments', json={'order\_id': self.id}**

**)**

**if response.status\_code == 201:**

**return**

**raise APIIntegrationError(**

**f'Could not process payment for order with id {self.id}'**

**)**

**def schedule(self):**

**response = requests.post(**  ⑥

**'http://localhost:3000/kitchen/schedules',**

**json={'order': [item.dict() for item in self.items]}**

**)**

**if response.status\_code == 201:**  ⑦

**return response.json()['id']**

**raise APIIntegrationError(**

**f'Could not schedule order with id {self.id}'**

**)**

① If an order is in progress, we cancel its schedule by calling the kitchen API.

② If the response from the kitchen service is successful, we return.

③ Otherwise, we raise an APIIntegrationError.

④ We don’t allow orders that are being delivered to be cancelled.

⑤ We process a payment by calling the payments API.

⑥ We schedule an order for production by calling the kitchen API.

⑦ If the response from the kitchen service is successful, we return the schedule ID.

Listing 7.7 contains the implementation of the custom exceptions we use in the order service to signal that something has gone wrong. We’ll use OrderNotFoundError in the OrdersService class when a user tries to fetch the details of an order that doesn’t exist.

Listing 7.7 Orders service custom exceptions

# file: orders/orders\_service/exceptions.py

class OrderNotFoundError(Exception): ①

pass

class APIIntegrationError(Exception): ②

pass

class InvalidActionError(Exception): ③

pass

① Exception to signal that an order doesn’t exist

② Exception to signal that an API integration error has taken place

③ Exception to signal that the action being performed is invalid

As we mentioned earlier, the API module won’t use the Order class directly. Instead, it will use a unified interface to all our adapters through the OrdersService class, whose interface we showed in listing 7.4. OrdersService encapsulates the capabilities of the orders domain, and it takes care of using the orders repository to get orders objects and perform actions on them. Listing 7.8 shows the implementation of the OrdersService class.

To instantiate the OrdersService class, we require an orders repository object that we can use to add or delete orders from our records. To place an order, we create a database record using the orders repository, and to retrieve the details of an order, we fetch the corresponding record from the database. If the requested order isn’t found, we raise an OrderNotFoundError exception. The list\_orders() method accepts filters in the form of a dictionary. To get a list of orders, the orders repository forces us to pass a specific value for the limit argument, and therefore we extract its value from the filters dictionary by using the pop() method, which allows us to set a default value and also removes the key from the dictionary. In the pay\_order() method, we process the payment using the payments API, and if the payment is successful, we schedule the order by calling the kitchen API. After scheduling the order, we update the order record by setting its schedule\_id attribute to the schedule ID returned by the kitchen API.

Listing 7.8 Implementation of the OrdersService

# file: orders/orders\_service/orders\_service.py

from orders.orders\_service.exceptions import OrderNotFoundError

class OrdersService:

def \_\_init\_\_(self, orders\_repository): ①

self.orders\_repository = orders\_repository

def place\_order(self, items):

return self.orders\_repository.add(items) ②

def get\_order(self, order\_id):

order = self.orders\_repository.get(order\_id) ③

if order is not None: ④

return order

raise OrderNotFoundError(f'Order with id {order\_id} not found')

def update\_order(self, order\_id, items):

order = self.orders\_repository.get(order\_id)

if order is None:

raise OrderNotFoundError(f'Order with id {order\_id} not found')

return self.orders\_repository.update(order\_id, {'items': items})

def list\_orders(self, \*\*filters):

limit = filters.pop('limit', None) ⑤

return self.orders\_repository.list(limit, \*\*filters)

def pay\_order(self, order\_id):

order = self.orders\_repository.get(order\_id)

if order is None:

raise OrderNotFoundError(f'Order with id {order\_id} not found')

order.pay()

schedule\_id = order.schedule() ⑥

return self.orders\_repository.update(

order\_id, {'status': 'scheduled', 'schedule\_id': schedule\_id}

)

def cancel\_order(self, order\_id):

order = self.orders\_repository.get(order\_id)

if order is None:

raise OrderNotFoundError(f'Order with id {order\_id} not found')

order.cancel()

return self.orders\_repository.update(order\_id, status="cancelled")

① To instantiate the OrdersService class, we require an instance of the orders repository.

② We place an order by creating a database record.

③ We get the details of an order using the orders repository and passing in the requested ID.

④ If the order doesn’t exist, we raise an OrderNotFoundError exception.

⑤ We capture filters as a dictionary by using keyword arguments.

⑥ After scheduling the order, we update its schedule\_id attribute.

The orders service is ready to be used in our API module. However, before we continue with this integration, there’s one more piece in this puzzle that we need to solve. As we mentioned in section 7.4, the orders repository doesn’t commit any actions to the database. It’s the responsibility of the API, as the consumer of the OrdersService, to ensure that everything is committed at the end of an operation. How exactly does that work? Move on to section 7.6 to learn how!

**7.6 Implementing the unit of work pattern**

In this section, we’ll learn to handle database commits and rollbacks when interacting with the OrdersService. As you can see in figure 7.15, when we use the OrdersService class to access any of its capabilities, we must inject an instance of the OrdersRepository class. We must also open an SQLAlchemy session before we perform any actions, and we must commit any changes to our data to persist them in the database.

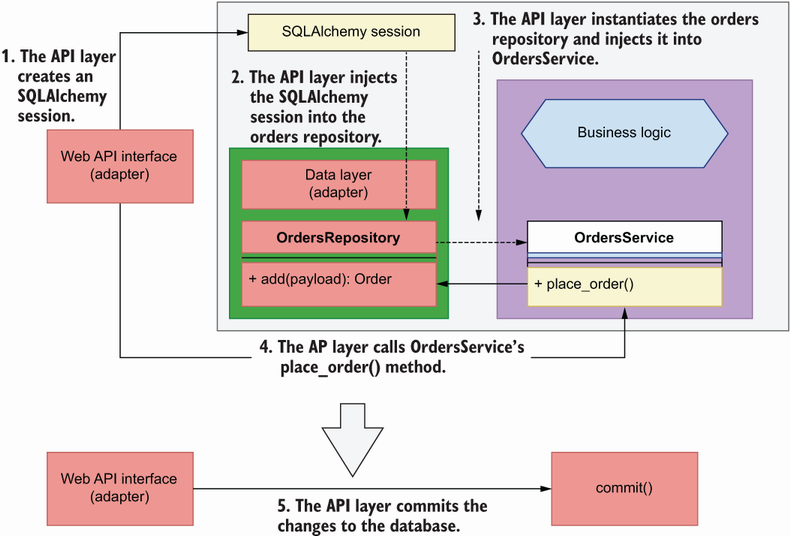


Figure 7.15 To persist our changes to the database, we could simply make the API layer use the SQLAlchemy session object to commit the transaction. In this figure, the solid lines represent calls, while the dashed lines represent injections of dependencies.

What’s the best way to orchestrate these operations? We can use different approaches for this implementation. We could simply use SQLAlchemy session objects to wrap our calls to OrdersService, and once our operations succeed, use the session to commit, or roll back otherwise. This would work if OrdersService only ever had to deal with a single SQL database. However, what if we had to interact with a different type of database at the same time? We’d need to open a new session for it as well. What if we also had to handle integrations with other microservices within the same operation, and ensure we make the right API calls at the end of the transaction in case we had to roll back? Again, we could just add special clauses and guards to our code. The same code would have to be repeated in every API function that interacts with the OrdersService, so wouldn’t it be nice if there was pattern that can help us put it all together in a single place? Enter the unit of work pattern.

**DEFINITION** The *unit of work* is a design pattern that guarantees the atomicity of our business transactions, ensuring that all transactions are committed at once, or rolled back if any of them fails.

The unit of work is a pattern that ensures that all objects of a business transaction are changed together, and if something fails, it ensures none of them changes.[**9**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1093398) The notion comes from the world of databases, where database transactions are implemented as units of work which ensure that every transaction is

* *Atomic*—The whole transaction either succeeds or fails.
* *Consistent*—It conforms to the constrains of the database.
* *Isolated*—It doesn’t interfere with other transactions.
* *Durable*—It’s written to persistent storage.

These properties are known as the *ACID principles* in the world of databases (<https://en.wikipedia.org/wiki/Database_transaction>). When it comes to services, the unit of work pattern helps us apply these principles in our operations. SQLAlchemy’s Session object already implements the unit of work pattern for database transactions (<http://mng.bz/jA5z>). This means that we can add as many changes as we need to the same session and commit them all together. If something goes wrong, we can call the rollback method to undo any changes. In Python, we can orchestrate these steps with context managers.

As you can see in figure 7.16, a context manager is a pattern that allows us to lock a resource during an operation, ensure that any necessary cleanup jobs are undertaken in case anything goes wrong, and finally release the lock once the operation is finished. The key syntactical feature of a context manager is the use of the with statement, as illustrated in figure 7.16. As you can see in the illustration, context managers can return objects, which we can capture by using Python’s as clause. This is useful if the context manager is creating access to a resource, such as a file, on which we want to operate.

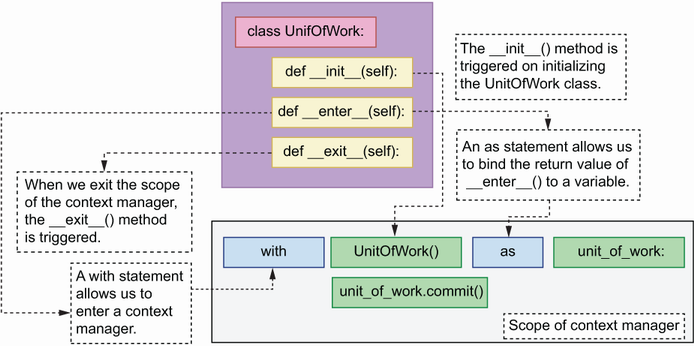


Figure 7.16 A class-based context manager has an \_\_init\_\_(), an \_\_enter\_\_(), and an \_\_exit\_\_() method. \_\_init\_\_() is triggered when we initialize the context manager. The \_\_enter\_\_ method allows us to enter the context, and it’s called when we use the with statement. Using an as statement within the same line allows us to bind the return value of the \_\_enter\_\_() method to a variable (unit\_of\_work in this case). Finally, when we exit the context manager, the \_\_exit\_\_() method is triggered.

In Python, we can implement context managers in multiple ways, including as a class or using the contextmanager() decorator from the contextlib module.[**10**](https://learning.oreilly.com/library/view/microservice-apis/9781617298417/OEBPS/Text/07.htm#pgfId-1093426) In this section, we’ll implement our unit of work context manager as a class. A context manager class must implement at least the two following special methods:

* \_\_enter\_\_()—Defines the operations that must be undertaken upon entering the context, such as creating a session or opening a file. If we need to perform actions on any of the objects created within the \_\_enter\_\_() method, we can return the object and capture its value through an as clause, as illustrated in figure 7.16.
* \_\_exit\_\_()—Defines the operations that must be undertaken upon exiting the context, for example, closing a file or a session. The \_\_exit\_\_() method captures any exceptions raised during the execution of the context through three parameters in its method signature:
  + exc\_type—Captures the type of exception raised
  + exc\_value—Captures the value bound to the exception, typically the error message
  + traceback—A traceback object that can be used to pinpoint the exact place where the exception took place

If no exceptions are raised, the value of these three parameters will be None.

Listing 7.9 shows the implementation of the unit of work pattern as a context manager for the orders service. In the initializer method, we obtain a session factory object using SQLAlchemy’s sessionmaker() function, which requires a connection object that we produce with the help of SQLAlchemy’s create\_engine() function. To keep the example simple, we’re hardcoding the database connection string to point to our local SQLite database. In chapter 13, you’ll learn to parameterize this value and pull it from the environment.

When we enter the context, we create a new database session, and we bind it to the UnitOfWork instance so that we can access it in other methods. We also return the context manager object itself so that the caller can access any of its attributes, such as the session object or the commit() method. On exiting the context, we check whether any exceptions were raised while adding or removing objects to the session, and if that’s the case, we roll back the changes to avoid leaving the database in an inconsistent state. We have access to the exception’s type (exc\_type) and value (exc\_val), and the traceback (traceback) context, which we can use to log the details of the error. If no exception took place, all three parameters will be set to None. Finally, we close the database session to release database resources and to end the scope of the transaction. We also add wrappers around SQLAlchemy’s commit() and rollback() methods to avoid exposing database internals to the business layer.

Listing 7.9 Unit of work pattern as a context manager

# file: orders/repository/unit\_of\_work.py

from sqlalchemy import create\_engine

from sqlalchemy.orm import sessionmaker

class UnitOfWork:

def \_\_init\_\_(self):

self.session\_maker = sessionmaker( ①

bind=create\_engine('sqlite:///orders.db')

)

def \_\_enter\_\_(self):

self.session = self.session\_maker() ②

return self ③

def \_\_exit\_\_(self, exc\_type, exc\_val, traceback): ④

if exc\_type is not None: ⑤

self.rollback() ⑥

self.session.close() ⑦

self.session.close()

def commit(self):

self.session.commit() ⑧

def rollback(self):

self.session.rollback() ⑨

① We obtain a session factory object.

② We open a new database session.

③ We return an instance of the unit of work object.

④ On existing the context, we have access to any exceptions raised during the context’s execution.

⑤ We check whether an exception took place.

⑥ If an exception took place, roll back the transaction.

⑦ We close the database session.

⑧ Wrapper around SQLAlchemy’s commit() method

⑨ Wrapper around SQLAlchemy’s rollback() method

This is all very good, but how exactly are we supposed to use the UnitOfWork in combination with the orders repository and the OrdersService? In the next section, we’ll delve more into the details of this, but before we do that, listing 7.10 gives you a template for how to use all these components together. We enter the unit of work context with Python’s syntax for context managers using a with statement. We also use an as statement to bind the return value of UnitOfWork’s \_\_enter\_\_() method to the unit\_of\_work variable. Then we get an instance of the orders repository passing in the UnitOfWork’s database session object, and an instance of the OrdersService class passing in the orders repository object. Then we use the orders service object to place an order, and we commit the transaction using UnitOfWork’s commit() method.

Listing 7.10 Template pattern for using the unit of work and the repository

with UnitOfWork() as unit\_of\_work: ①

repo = OrdersRepository(unit\_of\_work.session) ②

orders\_service = OrdersService(repo) ③

orders\_service.place\_order(order\_details) ④

unit\_of\_work.commit() ⑤

① We enter the unit of work context.

② We get an instance of the orders repository passing in the UnitOfWork’s session.

③ We get an instance of the OrdersService class passing in the orders repository object.

④ We place an order.

⑤ We commit the transaction.

Now that we have a unit of work that we can use to commit our transactions, let’s see how we put this all together by integrating the API layer with the service layer! Move on to section 7.7 to learn how we do that.

**7.7 Integrating the API layer and the service layer**

In this section, we put everything we have learned in this chapter together to integrate the service layer with the API layer. We’ll make use of the template pattern we showed in listing 7.10 to use the UnitOfWork class in combination with OrdersRepository and OrdersService. When a user tries to perform an action on an order, we make sure we have checks in place to verify that the order exists in the first place; otherwise, we return a 404 (Not Found) error response.

Listing 7.11 shows the new version of the orders/web/api/api.py module. The first thing we do in every function is enter the context of UnitOfWork, making sure we bind the context object to a variable, unit\_of\_work. Then we create an instance of OrdersRepository using the session object from the UnitOfWork context object. Once we have an instance of the repository, we inject it into OrdersService as we create an instance of the service. Then we use the service to perform the operations required in each endpoint. In endpoints that perform actions on a specific order, we guard against the possibility of an OrderNotFoundError being raised by OrdersService if the requested order doesn’t exist.

In the create\_order() function, we retrieve the dictionary representation of the order using order.dict() before we exit the UnitOfWork context so that we can access properties generated by the database during the commit process, such as the order’s ID. Remember that the order ID doesn’t exist until the changes are committed to the database, and therefore it’s only accessible within the scope of the database session. In our implementation, that means that we must access the ID before we exit the UnitOfWork context, since the database session closes right before exiting the context. Figure 7.17 illustrates this process.

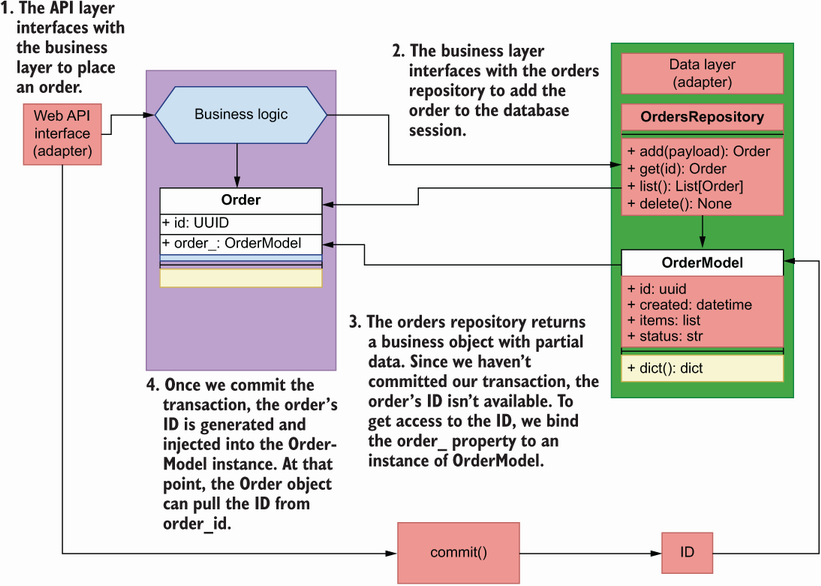


Figure 7.17 When we place an order, the object returned by the orders repository doesn’t contain an ID. The ID will be available once we commit the database transaction through the OrderModel instance. Therefore, we bind an instance of the model to the Order object so that it can pull the ID from the model after the commit.

Listing 7.11 Integration between API layer and service layer

# file: orders/web/api/api.py

from http import HTTPStatus

from typing import List, Optional

from uuid import UUID

from fastapi import HTTPException

from starlette import status

from starlette.responses import Response

**from orders.orders\_service.exceptions import OrderNotFoundError**

**from orders.orders\_service.orders\_service import OrdersService**

**from orders.repository.orders\_repository import OrdersRepository**

**from orders.repository.unit\_of\_work import UnitOfWork**

**from orders.web.app import app**

**from orders.web.api.schemas import (**

**GetOrderSchema,**

**CreateOrderSchema,**

**GetOrdersSchema,**

**)**

@app.get('/orders', response\_model=GetOrdersSchema)

def get\_orders(

cancelled: Optional[bool] = None,

limit: Optional[int] = None,

):

**with UnitOfWork() as unit\_of\_work:** ①

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**results = orders\_service.list\_orders(**

**limit=limit, cancelled=cancelled**

**)**

**return {'orders': [result.dict() for result in results]}**

@app.post(

'/orders',

status\_code=status.HTTP\_201\_CREATED,

response\_model=GetOrderSchema,

)

def create\_order(payload: CreateOrderSchema):

**with UnitOfWork() as unit\_of\_work:**

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**order = orders\_service.place\_order(payload.dict()['order'])**

**order = payload.dict()['order']**

**for item in order:**

**item['size'] = item['size'].value**

**order = orders\_service.place\_order(order)** ②

**unit\_of\_work.commit()**

**return\_payload = order.dict()** ③

**return return\_payload**

@app.get('/orders/{order\_id}', response\_model=GetOrderSchema)

def get\_order(order\_id: UUID):

**try:** ④

**with UnitOfWork() as unit\_of\_work:**

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**order = orders\_service.get\_order(order\_id=order\_id)**

**return order.dict()**

**except OrderNotFoundError:**

raise HTTPException(

status\_code=404, detail=f'Order with ID {order\_id} not found'

)

@app.put('/orders/{order\_id}', response\_model=GetOrderSchema)

def update\_order(order\_id: UUID, order\_details: CreateOrderSchema):

**try:**

**with UnitOfWork() as unit\_of\_work:**

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**order = order\_details.dict()['order']**

**for item in order:**

**item['size'] = item['size'].value**

**order = orders\_service.update\_order(**

**order\_id=order\_id, items=order**

**)**

**unit\_of\_work.commit()**

**return order.dict()**

**except OrderNotFoundError:**

raise HTTPException(

status\_code=404, detail=f'Order with ID {order\_id} not found'

)

@app.delete(

"/orders/{order\_id}",

status\_code=status.HTTP\_204\_NO\_CONTENT,

response\_class=Response,

)

def delete\_order(order\_id: UUID):

**try:**

**with UnitOfWork() as unit\_of\_work:**

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**orders\_service.delete\_order(order\_id=order\_id)**

**unit\_of\_work.commit()**

**return**

**except OrderNotFoundError:**

raise HTTPException(

status\_code=404, detail=f'Order with ID {order\_id} not found'

)

@app.post('/orders/{order\_id}/cancel', response\_model=GetOrderSchema)

def cancel\_order(order\_id: UUID):

**try:**

**with UnitOfWork() as unit\_of\_work:**

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**order = orders\_service.cancel\_order(order\_id=order\_id)**

**unit\_of\_work.commit()**

**return order.dict()**

**except OrderNotFoundError:**

raise HTTPException(

status\_code=404, detail=f'Order with ID {order\_id} not found'

)

@app.post('/orders/{order\_id}/pay', response\_model=GetOrderSchema)

def pay\_order(order\_id: UUID):

**try:**

**with UnitOfWork() as unit\_of\_work:**

**repo = OrdersRepository(unit\_of\_work.session)**

**orders\_service = OrdersService(repo)**

**order = orders\_service.pay\_order(order\_id=order\_id)**

**unit\_of\_work.commit()**

**return order.dict()**

**except OrderNotFoundError:**

raise HTTPException(

status\_code=404, detail=f'Order with ID {order\_id} not found'

)

① We enter the unit of work context.

② We place an order.

③ We access the order’s dictionary representation before exiting the unit of work context.

④ We use a try/except block to catch the OrderNotFoundError exception.

This concludes our journey through the implementation of the service layer for the orders service. The patterns we learned in this chapter are not only applicable to the world of APIs and microservices, but to all application models generally. In particular, the repository pattern will always help you ensure that you keep your data access layer fully decoupled from the business layer, and the unit of work pattern will help you ensure that all transactions of a business operation are handled atomically and consistently.

**Summary**

* Hexagonal architecture, or architecture of ports and adapters, is a software architectural pattern that encourages us to decouple the business layer from the implementation details of the database and the application interface.
* The dependency inversion principle teaches us that the implementation details of our application components should depend on interfaces. This helps us decouple our components from the implementation details of their dependencies.
* To interface with the database, you can use an ORM library such as SQLAlchemy, which can translate database tables and rows into classes and objects. This provides the possibility of enhancing our database models with useful functionality for our application needs.
* Repository is a software development pattern that helps to decouple the data layer from the business layer by adding an abstraction layer, which exposes an in-memory list interface of the data. Regardless of the database engine we use, the business layer will always receive the same objects from the repository.
* The unit of work pattern helps ensure that all the business transactions that are part of an application operation succeed or fail together. If one of the transactions fails, the unit of work pattern ensures that all changes are rolled back. This mechanism ensures that data is never left in an inconsistent state.

**1** Alistair Cockburn, “Hexagonal Architecture,” <https://alistair.cockburn.us/hexagonal-architecture/>. You may be wondering why hexagonal and not pentagonal or heptagonal. As Alistair points out, it “is not a hexagon because the number six is important,” but because it helps to visually highlight the idea of a core application communicating with external components through ports (the sides of the hexagon), and it allows us to represent the two main sides of an application: the public-facing side (web components, APIs, etc.) and the internal side (databases, third-party integrations, etc.).

**2** Robert C. Martin, *Agile Software Development, Principles, Patterns, and Practices* (Prentice Hall, 2003), pp. 127–131.

**3** For an excellent introduction to the dependency inversion principle, see Eric Freeman, Elizabeth Robson, Kathy Sierra, and Bert Bates, *Head First Design Patterns* (O’Reilly, 2014), pp. 141–143.

**4** Martin Fowler, *Patterns of Enterprise Architecture* (Addison-Wesley, 2003), pp. 165–181.

**5** The shape and format of this file may change over time, but for reference, at the time of this writing, those lines are 18–20.

**6** Fowler, *Patterns of Enterprise Architecture*, pp. 160–164.

**7** For more details on the inversion of control principle and the dependency injection pattern, see Martin Fowler, “Inversion of Control Containers and the Dependency Injection pattern,”