Python web-based framework:Flask

INTRODUCTION:

Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries.[3] It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.

What is python flask?

- **Flask** is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular **Python** web application frameworks

Purpose of flask:-

- **Flask** is a web framework. This means **flask** provides you with tools, libraries and technologies that allow you to build a web application. This web application can be some web pages, a blog, a wiki or go as big as a web-based calendar application or a commercial website. ... Werkzeug a WSGI utility library.

Is flask good for web development ?

- **Flask** is a lighter weight framework for Python. It's a tool to create sites quicker. It's not required, frameworks never are, but it makes **development** faster by offering code for all sorts of processes like database interaction or file activity.

Is flask a frontend or backend?

- **Flask** is "**Back End**" as well as Django. If you need arguments to convince him: - Every single feature of **Flask** is supposed to be running on the server context. **Front End** technologies are those who run mostly in the browser context.

Architecture:-

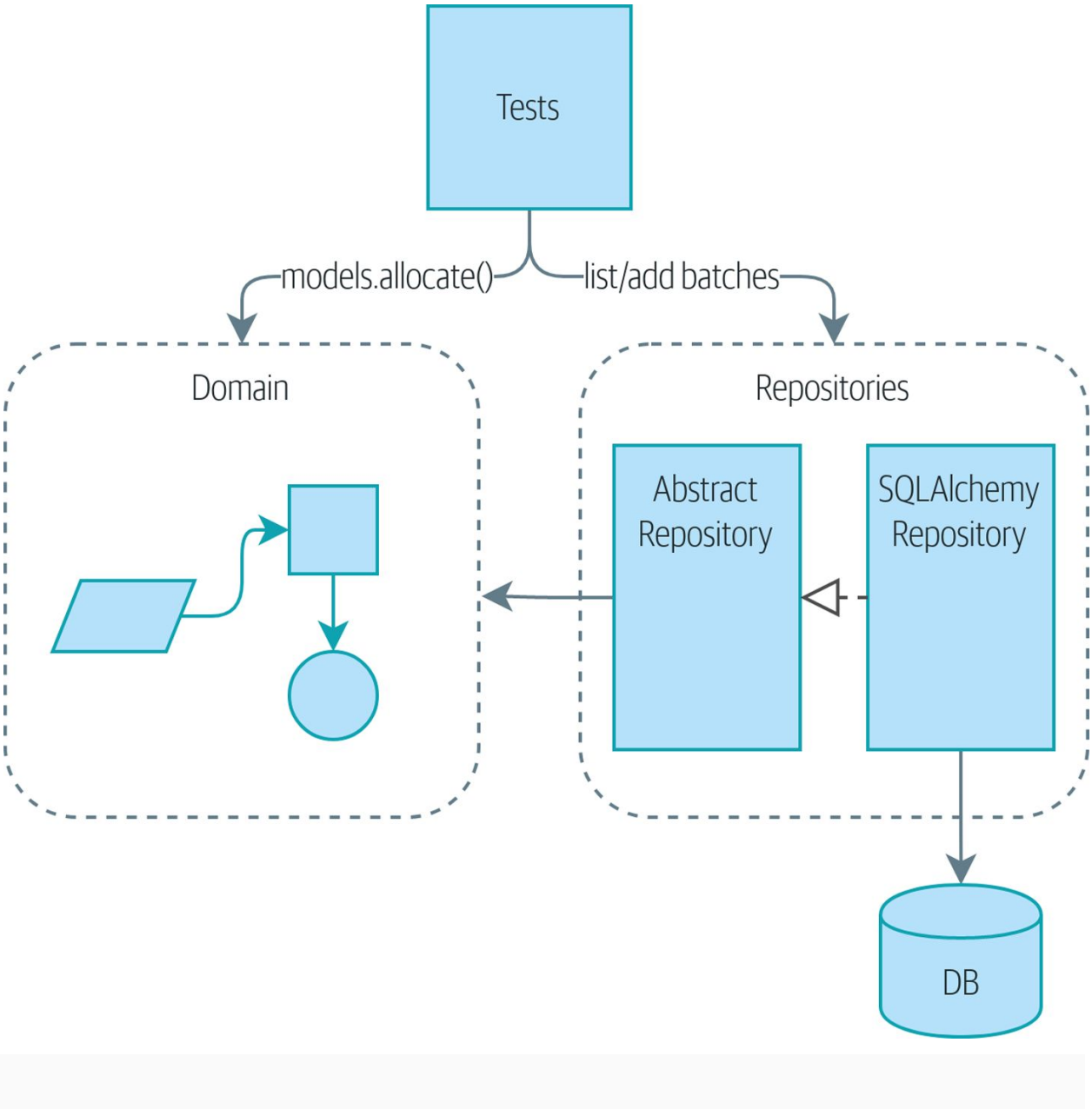

Figure 4-1. Before: we drive our app by talking to repositories and the domain model

From above figure we would be discussing the differences between orchestration logic, business logic, and interfacing code, and we introduce the *Service Layer* pattern to take care of orchestrating our workflows and defining the use cases of our system.

We'll also discuss testing: by combining the Service Layer with our repository abstraction over the database, we're able to write fast tests, not just of our domain model but of the entire workflow for a use case.

Figure 4-2 shows what we're aiming for: we're going to add a Flask API that will talk to the service layer, which will serve as the entrypoint to our domain model. Because our service layer depends on the AbstractRepository, we can unit test it by using FakeRepository but run our production code using SqlAlchemyRepository.

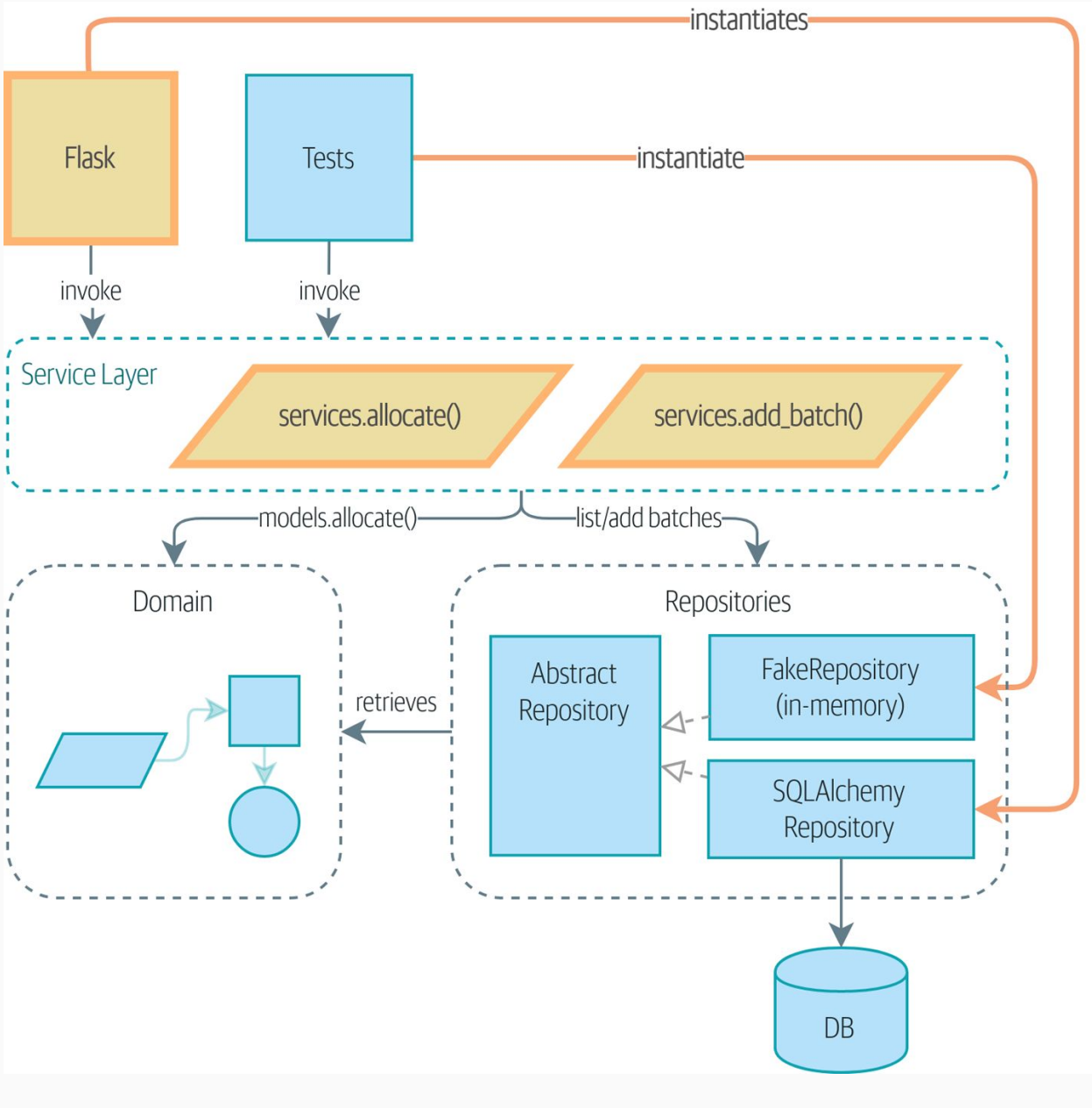

Figure 4-2. The service layer will become the main way into our app

In our diagrams, we are using the convention that new components are highlighted with bold text/lines (and yellow/orange color, if you're reading a digital version).

TIP

The code for this chapter is in the chapter_04_service_layer branch on GitHub:

```
git clone https://github.com/cosmicpython/code.git
cd code
git checkout chapter_04_service_layer
# or to code along, checkout Chapter 2:
git checkout chapter_02_repository
```

Connecting Our Application to the Real World

Like any good agile team, we're hustling to try to get an MVP out and in front of the users to start gathering feedback. We have the core of our domain model and the domain service we need to allocate orders, and we have the repository interface for permanent storage.

Let's plug all the moving parts together as quickly as we can and then refactor toward a cleaner architecture. Here's our plan:

1. Use Flask to put an API endpoint in front of our allocate domain service. Wire up the database session and our repository. Test it with an end-to-end test and some quick-and-dirty SQL to prepare test data.
2. Refactor out a service layer that can serve as an abstraction to capture the use case and that will sit between Flask and our domain model. Build some service-layer tests and show how they can use FakeRepository.
3. Experiment with different types of parameters for our service layer functions; show that using primitive data types allows the service layer's clients (our tests and our Flask API) to be decoupled from the model layer.

A First End-to-End Test

No one is interested in getting into a long terminology debate about what counts as an end-to-end (E2E) test versus a functional test versus an

acceptance test versus an integration test versus a unit test. Different projects need different combinations of tests, and we've seen perfectly successful projects just split things into "fast tests" and "slow tests."

For now, we want to write one or maybe two tests that are going to exercise a "real" API endpoint (using HTTP) and talk to a real database. Let's call them *end-to-end tests* because it's one of the most self-explanatory names.

How Flask App Works :-

```
from flask import Flask
 app = Flask(__name__)
@app.route('/')
def home():
        return " Hey There!"
if __name__ ==’__main__’:
        app.run(debug=True)
```

In **line 1** you are making available the code you need to build web apps with flask. flask is the framework here, while Flask is a Python class datatype. In other words, *Flask* is the prototype used to create instances of web application or web applications if you want to put it simple.

So, once we import *Flask*, we need to create an instance of the *Flask* class for our web app. That's what **line 3** does. __name__ is a special variable that gets as value the string "__main__" when you're executing the script. We'll go back to that in a moment.

In **lines 5-7** we are defining a function that returns the string “Hey there!”. That function is mapped to the home ‘/’ URL. That means when the user navigates to localhost:5000, the *home* function will run and it will return its output on the webpage. If the input to the route method was something else, let’s say ‘/about/’, the function output would be shown when the user visited localhost:5000/about/. How practical is that?

Then we have the mysterious **lines 9 and 10**. Something you should know is that Python assigns the name "__main__" to the script when the script is executed. If the script is imported from another script, the script keeps it given name (e.g. hello.py). In our case we are executing the script. Therefore, __name__ will be equal to "__main__". That means the if conditional statement is satisfied and the *app.run()* method will be executed. This technique allows the programmer to have control over script’s behavior.

Notice also that we are setting the debug parameter to true. That will print out possible Python errors on the web page helping us trace the errors. However, in a production environment, you would want to set it to False as to avoid any security issues.

So far, so good!

We have a running website with some content returned by the Python function. As you can imagine, returning plain strings from a function doesn’t take us anywhere far.

If you want to make something more serious and more visually appealing, you would want to incorporate HTML files along your Python file and have your Python code return HTML pages instead of plain strings.

When to use ?:

- You will want to **use Flask** for simple projects where you don't need all the power/complexity Django offers.
- Django has a template system, ORM, lots of other good stuff like authentication, caching etc.
- Don't need that or don't want to **use** their template system and ORM over Jinja and SQLAlchemy? **Use Flask**.

Advantages :

- Integrated support for unit testing
- Built-in development server and fast debugger
- RESTful request dispatching
- Jinja2 templating
- Support for secure cookies (client side sessions)
- WSGI 1.0 compliant
- Unicode based

Disadvantages:

- Full-Stack experience.
- No admin site.
- Not suitable for big applications.
- Community.
- No login or authentication.
- ORM.
- Migrations can be difficult.

COMPARISON BETWEEN FLASK AND DJANGO

Type of Web Framework

As noted earlier, Django is a full-stack Python web framework. Also, it is developed based on batteries included approach. The batteries included in Django makes it easier for Django developers to accomplish common web development tasks like user authentication, URL routing and database schema

migration. Also, Django accelerates custom web application development by providing built-in template engine, ORM system, and bootstrapping tool. On the other hand, Flask is a simple, lightweight, and minimalist web framework. It lacks some of the built-in features provided by Django. But it helps developers to keep the core of a web application simple and extensible.

Functional Admin Interface

Unlike Flask, Django makes it easier for users to handle common project administration tasks by providing a ready-to-use admin framework. It further generates the functional admin module automatically based on project models. The developers even have option to customize the admin interface to meet specific business requirements. They can even take advantage of the admin interface to simplify website content administration and user management. The functional admin user interface makes Django standout in the crowd.

Template Engine

As noted earlier, Flask is developed based on Jinja2 template engine. As a fully-featured template engine for Python, Jinja2 is also inspired by Django's template system. It enables developers to accelerate development of dynamic web applications by taking advantage of an integrated sandboxed execution environment and writing templates in an expressive language. Django comes with a built-in template engine that enables developers to define a web application's user facing layer without putting extra time and effort. It even allows developers to accelerate custom user interface development by writing templates in Django template language (DTL).

Built-in Bootstrapping Tool

Unlike Flask, Django comes with a built-in bootstrapping tool – django-admin. Django-admin enables developers to start building web applications without any external input. Django even allows developers to divide a single project into a number of applications. The developers can use django-admin to create new applications within the project. They can further use the applications to add functionality to the web applications based on varied business requirements.

Project Layout

Flask requires developers to each project as a single application. But the developers have option to add multiple models and views to the same application. On the other hand, Django allows developers to divide a project into multiple applications. Hence, it becomes easier for developers to write individual applications, and add functionality to the web application by integrating the applications into the project. The small applications further help developers to extend and maintain the web applications written in Python.

Database Support

Django allows developers to take advantage of a robust ORM system. The developers can use the ORM system to work with widely used databases like MySQL, Oracle, SQLite and PostgreSQL. Also, the ORM system enables developers to perform common database operations without writing lengthy SQL queries. Unlike Django, Flask does not provide a built-in ORM system. It requires developers to work with databases and perform database operations through SQLAlchemy. The Python developers can work with databases by using SQLAlchemy as a SQL toolkit and ORM system for Python. Also, they can perform common database queries by writing and executing SQL queries.

Flexibility

The batteries included in Django help developers to build a variety of web applications without using third-party tools and libraries. But the developers lack any option to make changes to the modules provided by Django. Hence, developers have to build web applications by availing the built-in features provided by the web framework. On the other hand, Flask is a micro but extensible web frameworks. It enables developers to build web applications more flexibly by using several widely used web development tools and libraries. Many beginners even find it easier to learn Flask than Django due to its simple and customizable architecture.

Usage and Use Cases

Both Flask and Django are currently being used by several high-traffic websites. But the usage statistics posted on various websites depict that Django is more popular than Flask. You can find some of the best websites developed using Django here. The developers can take advantage of the robust features provided by Django to build and deploy complex web applications rapidly. At the same time, they can use Flask to accelerate development of simple websites that use static content. However, the developers still have option to extend and customize Flask according to precise project requirements.

On the whole, both Flask and Django are widely used open source web frameworks for Python. Django is a full-stack web framework, whereas Flask is a micro and lightweight web framework. The features provided by Django help developers to build large and complex web applications. On the other hand, Flask accelerates development of simple web applications by providing the required functionality. Hence, the developers must keep in mind the needs of individual projects while comparing Flask and Django.

References:

1. https://flask.palletsprojects.com/en/1.1.x/api/#incoming-request-data
2. https://github.com/pallets/flask