CST8915 – Full-stack Cloud-native Development

Week 10 Lab - Test-Driven Development

Introduction

In this lab, you will learn how to **build** and **test** a **production-grade** microservice powered by Python, Flask, and Docker. You will establish a development setup using Docker for constructing and launching a microservice that utilizes Python and Flask. Throughout the lab, you will implement Test-Driven Development techniques using pytest while you work on developing a RESTful API.

By the end of this lab, you will have built a RESTful API, using Test-Driven Development. The API itself will follow RESTful design principles, using the basic HTTP verbs: GET, POST, PUT, and DELETE.

Endpoint	HTTP Method	CRUD Method	Result
/users	GET	READ	get all users
/users/:id	GET	READ	get a single user
/users	POST	CREATE	add a user
/users/:id	PUT	UPDATE	update a user
/users/:id	DELETE	DELETE	delete a user

Objective

The goal of this lab activity is to familiarize students with the concepts and practices of Test-Driven Development (TDD) and its role in cloud-native development, ensuring high-quality and reliable application components.

Prerequisites:

- Basic understanding of cloud computing concepts, cloud-native development, web application development, microservices, and containerization technologies (e.g., Docker).

- Familiarity with container management tools (e.g., Docker Compose), RESTful APIs, and container orchestration platforms (e.g., Kubernetes).
- A computer with internet access and a code editor installed.

Overview of tools used in this Lab:

Along with Python and Flask, we will use Docker to quickly set up our local development environment and simplify deployment. SQLAlchemy will be used to interact with a Postgres database. We will use pytest instead of unittest for writing unit and integration tests to test the Flask API. Flask-RESTX is a community-driven fork of Flask-RESTPlus that makes it easy to build and document RESTful APIs with Flask.

Part1: Getting Started

Step 1-1: Create a new project and install Flask along with Flask-RESTX:

```
$ mkdir tdd-lab-w9 && cd tdd-lab-w9
$ mkdir src
$ python3.11 -m venv env
$ source env/bin/activate

(env)$ pip install flask==2.2.3
(env)$ pip install flask-restx==1.0.6
(env)$ pip install Werkzeug==2.2.2
```

Step 1-2: Add an *__init__.py* file to the "src" directory and configure the first endpoint:

```
# src/__init__.py


from flask import Flask, jsonify
from flask_restx import Resource, Api


# instantiate the app
app = Flask(__name__)

api = Api(app)


class Ping(Resource):
    def get(self):
        return {
            'status': 'success',
            'message': 'pong!'
        }


api.add_resource(Ping, '/ping')
```

Step 1-3: Next, let's configure the Flask CLI tool to run and manage the app from the command line.

First, add a *manage.py* file to the project root, "tdd-lab-w9":

```
# manage.py


from flask.cli import FlaskGroup

from src import app


cli = FlaskGroup(app)


if __name__ == '__main__':
    cli()
```

Here, we created a new `FlaskGroup` instance to extend the normal CLI with commands related to the Flask app.

Step 1-4: Run the server:

```
(env)$ export FLASK_APP=src/__init__.py
(env)$ python manage.py run
```

Navigate to http://127.0.0.1:5000/ping in your browser. You should see:

```
{
  "message": "pong!",
  "status": "success"
}
```

Step 1-5: Shut down the server. Then, add a new file called *config.py* to the "src" directory, where we'll define environment-specific configuration variables:

```python
# src/config.py


class BaseConfig:
    TESTING = False


class DevelopmentConfig(BaseConfig):
    pass


class TestingConfig(BaseConfig):
    TESTING = True


class ProductionConfig(BaseConfig):
    pass
```

Update *__init__.py* to pull in the development config on init:

```python
# src/__init__.py


from flask import Flask, jsonify
from flask_restx import Resource, Api


# instantiate the app
app = Flask(__name__)

api = Api(app)

# set config
app.config.from_object('src.config.DevelopmentConfig')  # new


class Ping(Resource):
    def get(self):
        return {
            'status': 'success',
            'message': 'pong!'
        }


api.add_resource(Ping, '/ping')
```

Step 1-6: Run the app again. This time, let's enable debug mode by setting the `FLASK_DEBUG` environment variable to `1`:

```
(env)$ export FLASK_APP=src/__init__.py
(env)$ export FLASK_DEBUG=1
(env)$ python manage.py run

 * Serving Flask app 'src/__init__.py'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use
a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 483-377-737
```

Now when you make changes to the code, the app will automatically reload. Try this out. You also have access to the interactive debugger. Shut down the server once done. Exit then remove the virtual environment as well. Then, add a *requirements.txt* file to the project root:

```
flask==2.2.3
flask-restx==1.0.6
Werkzeug==2.2.2
```

Finally, add a *.gitignore* to the project root:

```
__pycache__
env
```

Init a git repo and commit your code.

Part2: Docker Configuration

Step 2-1: Start by ensuring that you have Docker and Docker Compose:

```
$ docker -v
Docker version 20.10.23, build 7155243

$ docker-compose -v
Docker Compose version v2.15.1
```

Make sure to install or upgrade them if necessary.

Step 2-2: Add a *Dockerfile* to the project root, making sure to review the code comments:

```
# pull official base image
FROM python:3.11.2-slim-buster

# set working directory
WORKDIR /usr/src/app

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# add and install requirements
COPY ./requirements.txt .
RUN pip install -r requirements.txt

# add app
COPY . .

# run server
CMD python manage.py run -h 0.0.0.0
```

Here, we started with a slim-buster-based Docker image for Python 3.11.2. We then set a working directory along with two environment variables:

1. `PYTHONDONTWRITEBYTECODE`: Prevents Python from writing pyc files to disc (equivalent to `python -B` option)
2. `PYTHONUNBUFFERED`: Prevents Python from buffering stdout and stderr (equivalent to `python -u` option)

Depending on your environment, you may need to add `RUN mkdir -p /usr/src/app` just before you set the working directory:

```
# set working directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app
```

Add a *.dockerignore* file to the project root as well:

```
env
Dockerfile
Dockerfile.prod
```

Like the *.gitignore* file, the .dockerignore file lets you exclude specific files and folders from being copied over to the image.

Step 2-3: Then add a *docker-compose.yml* file to the project root:

```
version: '3.8'

services:

  api:
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - .:/usr/src/app
    ports:
      - 5004:5000
    environment:
      - FLASK_APP=src/__init__.py
      - FLASK_DEBUG=1
      - FLASK_ENV=development
```

This config will create a service called `api` from the Dockerfile.

The `volume` is used to mount the code into the container. This is a must for a development environment in order to update the container whenever a change to the source code is made. Without this, you would have to re-build the image each time you make a change to the code.

Take note of the Docker compose file version used -- `3.8`. Keep in mind that this version does *not* directly relate back to the version of Docker Compose installed; it simply specifies the file format that you want to use.

Step 2-4: Build the image:

```
$ docker-compose build
```

This will take a few minutes the first time. Subsequent builds will be much faster since Docker caches the results.

Step 2-5: Once the build is done, fire up the container in detached mode:

```
$ docker-compose up -d
```

Navigate to http://localhost:5004/ping. Make sure you see the same JSON response as before:

```
{
  "message": "pong!",
  "status": "success"
}
```

If you run into problems with the volume mounting correctly, you may want to remove it altogether by deleting the volume config from the Docker Compose file. You can still go through the lab without it; you'll just have to re-build the image after you make changes to the source code.

Windows Users: Having problems getting the volume to work properly? Review the following resources:

1. Docker on Windows—Mounting Host Directories
2. Configuring Docker for Windows Shared Drives

You also may need to add `COMPOSE_CONVERT_WINDOWS_PATHS=1` to the `environment` portion of your Docker Compose file. Review Declare default environment variables in file for more info.

Step 2-6: Next, add an environment variable to the *docker-compose.yml* file to load the app config for the development environment:

```
version: '3.8'

services:

  api:
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - .:/usr/src/app
    ports:
      - 5004:5000
    environment:
      - FLASK_APP=src/__init__.py
      - FLASK_DEBUG=1
      - FLASK_ENV=development
      - APP_SETTINGS=src.config.DevelopmentConfig  # new
```

Then update *src/__init__.py*, to pull in the environment variables:

```
# src/__init__.py


import os  # new

from flask import Flask, jsonify
from flask_restx import Resource, Api


# instantiate the app
app = Flask(__name__)

api = Api(app)

# set config
app_settings = os.getenv('APP_SETTINGS')  # new
app.config.from_object(app_settings)      # new


class Ping(Resource):
    def get(self):
        return {
            'status': 'success',
            'message': 'pong!'
        }
```

```
api.add_resource(Ping, '/ping')
```

Step 2-7: Update the container:

```
$ docker-compose up -d --build
```

Step 2-8: Want to ensure the proper config was loaded? Add a `print` statement to *__init__.py*, right before the route handler, as a quick test:

```
import sys
print(app.config, file=sys.stderr)
```

Then view the Docker logs:

```
$ docker-compose logs
```

You should see something like:

```
<Config {
    'ENV': 'development', 'DEBUG': True, 'TESTING': False,
    'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None,
    'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31),
    'USE_X_SENDFILE': False, 'SERVER_NAME': None, 'APPLICATION_ROOT': '/',
    'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_DOMAIN': None,
    'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_HTTPONLY': True,
    'SESSION_COOKIE_SECURE': False, 'SESSION_COOKIE_SAMESITE': None,
    'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None,
    'SEND_FILE_MAX_AGE_DEFAULT': None, 'TRAP_BAD_REQUEST_ERRORS': None,
    'TRAP_HTTP_EXCEPTIONS': False, 'EXPLAIN_TEMPLATE_LOADING': False,
    'PREFERRED_URL_SCHEME': 'http', 'JSON_AS_ASCII': None, 'JSON_SORT_KEYS': None,
    'JSONIFY_PRETTYPRINT_REGULAR': None, 'JSONIFY_MIMETYPE': None,
    'TEMPLATES_AUTO_RELOAD': None, 'MAX_COOKIE_SIZE': 4093,
    'RESTX_MASK_HEADER': 'X-Fields', 'RESTX_MASK_SWAGGER': True,
    'RESTX_INCLUDE_ALL_MODELS': False
}>
```

Make sure to remove the `print` statement before moving on.

Part3: Postgres Setup

In this part, we'll configure Postgres, get it up and running in another container, and link it to the `users` service.

Step 3-1: Add Flask-SQLAlchemy and psycopg2 to the *requirements.txt* file:

```
Flask-SQLAlchemy==3.0.3
psycopg2-binary==2.9.5
```

Step 3-2: Update *config.py*:

```
# src/config.py


import os  # new


class BaseConfig:
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False  # new


class DevelopmentConfig(BaseConfig):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')  # new


class TestingConfig(BaseConfig):
    TESTING = True
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_TEST_URL')  # new


class ProductionConfig(BaseConfig):
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL')  # new
```

Step 3-3: Update *__init__.py*, to create a new instance of SQLAlchemy and define the database model:

```
# src/__init__.py

```

```python
import os
from flask import Flask, jsonify
from flask_restx import Resource, Api
from flask_sqlalchemy import SQLAlchemy  # new


# instantiate the app
app = Flask(__name__)

api = Api(app)

# set config
app_settings = os.getenv('APP_SETTINGS')
app.config.from_object(app_settings)

# instantiate the db
db = SQLAlchemy(app)  # new


# model
class User(db.Model):  # new
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), nullable=False)
    email = db.Column(db.String(128), nullable=False)
    active = db.Column(db.Boolean(), default=True, nullable=False)

    def __init__(self, username, email):
        self.username = username
        self.email = email


class Ping(Resource):
    def get(self):
        return {
            'status': 'success',
            'message': 'pong!'
        }


api.add_resource(Ping, '/ping')
```

Step 3-4: Add a "db" directory to "src", and add a *create.sql* file in that new directory:

```sql
CREATE DATABASE api_dev;
CREATE DATABASE api_test;
```

Step 3-5: Next, add a *Dockerfile* to the same directory:

```
# pull official base image
FROM postgres:15

# run create.sql on init
ADD create.sql /docker-entrypoint-initdb.d
```

Here, we extended the official Postgres image by adding *create.sql* to the "docker-entrypoint-initdb.d" directory in the container. This file will execute on init.

Step 3-6: Update *docker-compose.yml*:

```
version: '3.8'

services:

  api:
    build:
      context: .
      dockerfile: Dockerfile
    volumes:
      - '.:/usr/src/app'
    ports:
      - 5004:5000
    environment:
      - FLASK_APP=src/__init__.py
      - FLASK_DEBUG=1
      - FLASK_ENV=development
      - APP_SETTINGS=src.config.DevelopmentConfig
      - DATABASE_URL=postgresql://postgres:postgres@api-db:5432/api_dev  # new
      - DATABASE_TEST_URL=postgresql://postgres:postgres@api-db:5432/api_test  # new
    depends_on:  # new
      - api-db

  api-db:  # new
    build:
      context: ./src/db
      dockerfile: Dockerfile
    expose:
      - 5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

Once spun up, Postgres will be available on port `5432` for services running in other containers. Since the `api` service is dependent not only on the container being up and running but also the actual Postgres instance being up and healthy.

Step 3-7: Let's add an *entrypoint.sh* file to the project root:

```
#!/bin/sh

echo "Waiting for postgres..."

while ! nc -z api-db 5432; do
  sleep 0.1
done

echo "PostgreSQL started"

python manage.py run -h 0.0.0.0
```

So, we referenced the Postgres container using the name of the service, `api-db`. The loop continues until something like `Connection to api-db port 5432 [tcp/postgresql] succeeded!` is returned.

Step 3-8: Update *Dockerfile* to install the appropriate packages and copy over the *entrypoint.sh* script:

```
# pull official base image
FROM python:3.11.2-slim-buster

# set working directory
WORKDIR /usr/src/app

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# new
# install system dependencies
RUN apt-get update \
  && apt-get -y install netcat gcc postgresql \
  && apt-get clean

# add and install requirements
COPY ./requirements.txt .
RUN pip install -r requirements.txt

# add app
COPY . .
```

```
# new
# add entrypoint.sh
COPY ./entrypoint.sh .
RUN chmod +x /usr/src/app/entrypoint.sh
```

Step 3-9: Add an entrypoint to the Docker Compose file:

```
version: '3.8'

services:

  api:
    build:
      context: .
      dockerfile: Dockerfile
    entrypoint: ['/usr/src/app/entrypoint.sh']  # new
    volumes:
      - '.:/usr/src/app'
    ports:
      - 5004:5000
    environment:
      - FLASK_APP=src/__init__.py
      - FLASK_DEBUG=1
      - FLASK_ENV=development
      - APP_SETTINGS=src.config.DevelopmentConfig
      - DATABASE_URL=postgresql://postgres:postgres@api-db:5432/api_dev
      - DATABASE_TEST_URL=postgresql://postgres:postgres@api-db:5432/api_test
    depends_on:
      - api-db

  api-db:
    build:
      context: ./src/db
      dockerfile: Dockerfile
    expose:
      - 5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

Sanity check:

```
$ chmod +x entrypoint.sh
$ docker-compose up -d --build
```

Step 3-10: Ensure http://localhost:5004/ping still works:

```
{
  "message": "pong!",
  "status": "success"
}
```

Depending on your environment, you may need to chmod 755 or 777 instead of +x. If you still get a "permission denied", review the docker entrypoint running bash script gets "permission denied" Stack Overflow question.

Step 3-11: Update *manage.py*:

```
# manage.py


from flask.cli import FlaskGroup

from src import app, db  # new


cli = FlaskGroup(app)


# new
@cli.command('recreate_db')
def recreate_db():
    db.drop_all()
    db.create_all()
    db.session.commit()


if __name__ == '__main__':
    cli()
```

This registers a new command, `recreate_db`, to the CLI so that we can run it from the command line, which we'll use shortly to apply the model to the database.

Part4: Pytest Setup

Step 4-1: Add a "tests" directory to the "src" directory, and then create the following files inside the newly created directory:

1. *__init__.py*
2. *conftest.py*
3. *pytest.ini*
4. *test_config.py*
5. *test_ping.py*

By default, pytest will autodiscover test files that start or end with `test` -- i.e., `test_*.py` or `*_test.py`. Test functions must begin with `test_`, and if you want to use classes they must also begin with `Test`.

Example:

```python
# if a class is used, it must begin with Test
class TestFoo:

    # test functions must begin with test_
    def test_bar(self):
        assert "foo" != "bar"
```

If this is your first time with pytest be sure to review the Installation and Getting Started.

Fixtures

Step 4-2: Define a `test_app` and `test_database` (for initializing a test database) fixture in *conftest.py*:

```python
# src/tests/conftest.py

import pytest

from src import app, db


@pytest.fixture(scope='module')
def test_app():
    app.config.from_object('src.config.TestingConfig')
    with app.app_context():
        yield app  # testing happens here


@pytest.fixture(scope='module')
def test_database():
    db.create_all()
    yield db  # testing happens here
    db.session.remove()
    db.drop_all()
```

Fixtures are reusable objects for tests. They have a scope associated with them, which indicates how often the fixture is invoked:

1. function - once per test function (default)
2. class - once per test class
3. module - once per test module
4. session - once per test session

Take note of the `test_database` fixture. In essence, all code before the `yield` statement serves as setup code while everything after serves as the teardown.

For more on this review About fixtures.

Step 4-3: Then, add pytest to the requirements file:

```
pytest==7.2.1
```

Step 4-4: We need to re-build the Docker images since requirements are installed at build time rather than run time:

```
$ docker-compose up -d --build
```

Step 4-5: With the containers up and running, run the tests:

```
$ docker-compose exec api python -m pytest "src/tests"
```

You should see:

```
======================================= test session starts
=======================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 0 items

=================================== no tests ran in 0.06 seconds
====================================
```

Tests

Step 4-6: Moving on, let's add a few tests to *test_config.py*:

```
# src/tests/test_config.py


import os


def test_development_config(test_app):
    test_app.config.from_object('src.config.DevelopmentConfig')
    assert test_app.config['SECRET_KEY'] == 'my_precious'
    assert not test_app.config['TESTING']
    assert test_app.config['SQLALCHEMY_DATABASE_URI'] ==
os.environ.get('DATABASE_URL')


def test_testing_config(test_app):
    test_app.config.from_object('src.config.TestingConfig')
    assert test_app.config['SECRET_KEY'] == 'my_precious'
    assert test_app.config['TESTING']
    assert test_app.config['SQLALCHEMY_DATABASE_URI'] ==
os.environ.get('DATABASE_TEST_URL')


def test_production_config(test_app):
    test_app.config.from_object('src.config.ProductionConfig')
    assert test_app.config['SECRET_KEY'] == 'my_precious'
    assert not test_app.config['TESTING']
    assert test_app.config['SQLALCHEMY_DATABASE_URI'] ==
os.environ.get('DATABASE_URL')
```

While unittest requires test classes, pytest just requires functions to get up and running. In other words, pytest tests are just functions that either start or end with `test`.

You can still use classes to organize tests in pytest if that's your preferred pattern.

To use the fixture, we passed it in as an argument.

Step 4-7: Run the tests again:

```
$ docker-compose exec api python -m pytest "src/tests"
```

You should see the following error:

```
>       assert test_app.config['SECRET_KEY'] == 'my_precious'
E       AssertionError: assert None == 'my_precious'
```

Step 4-8: Update the base config in *src/config.py*:

```
class BaseConfig:
    TESTING = False
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    SECRET_KEY = 'my_precious'  # new
```

Then re-test!

```
======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 3 items

project/tests/test_config.py ...
[100%]

===================================== 3 passed in 0.07 seconds
======================================
```

Step 4-9: Add a functional test to *test_ping.py*:

```
# src/tests/test_ping.py


import json
```

```
def test_ping(test_app):
    client = test_app.test_client()
    resp = client.get('/ping')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert 'pong' in data['message']
    assert 'success' in data['status']
```

Test:

```
======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 4 items

project/tests/test_config.py ...
[ 75%]
project/tests/test_ping.py .
[100%]

===================================== 4 passed in 0.12 seconds
======================================
```

Step 4-10: Did you notice that the tests in *test_config.py* are unit tests while the tests in *test_ping.py* are functional? You may want to differentiate between the two by splitting them up into separate folders, like so:

```
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── functional
    │   └── test_ping.py
    ├── pytest.ini
    └── unit
        └── test_config.py
```

With this structure you have the flexibility to run a single type of test at a time. You can also check code coverage for a single type of test.

Given-When-Then

Step 4-11: When writing tests, try to follow the Given-When-Then framework to help make the process of writing tests easier and faster. It also helps communicate the purpose of your tests better so it should be easier to read by your future self and others.

Example:

```
def test_ping(test_app):
    # Given
    client = test_app.test_client()

    # When
    resp = client.get('/ping')
    data = json.loads(resp.data.decode())

    # Then
    assert resp.status_code == 200
    assert 'pong' in data['message']
    assert 'success' in data['status']
```

Selecting Tests

You can select specific tests to run using substring matching.

Step 4-12: For example, to run all tests that have the word `config` in their names:

```
$ docker-compose exec api python -m pytest "src/tests" -k config

======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 4 items / 1 deselected / 3 selected

project/tests/test_config.py ...
[100%]

================================== 3 passed, 1 deselected in 0.10s
==================================
```

So, you can see that the three tests in *test_config.py* ran (and passed) while the single test in *test_ping.py* was skipped.

Part5: Flask Blueprints

Step 5-1: With tests in place, let's refactor the app, adding in Blueprints.

Unfamiliar with Blueprints? Check out the official Flask documentation. Essentially, they are self-contained components, used for encapsulating code, templates, and static files.

Create a new directory in "src" called "api", and add an *__init__.py* file along with *ping.py*, *users.py*, and *models.py*. Then within *ping.py* add the following:

```
# src/api/ping.py


from flask import Blueprint
from flask_restx import Resource, Api


ping_blueprint = Blueprint('ping', __name__)
api = Api(ping_blueprint)


class Ping(Resource):
    def get(self):
        return {
            'status': 'success',
            'message': 'pong!'
        }


api.add_resource(Ping, '/ping')
```

Here, we created a new instance of the `Blueprint` class and bounded the `Ping` resource to it.

Step 5-2: Then, add the following code to *models.py*:

```
# src/api/models.py


from sqlalchemy.sql import func

from src import db
```

```python
class User(db.Model):

    __tablename__ = 'users'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    username = db.Column(db.String(128), nullable=False)
    email = db.Column(db.String(128), nullable=False)
    active = db.Column(db.Boolean(), default=True, nullable=False)
    created_date = db.Column(db.DateTime, default=func.now(), nullable=False)

    def __init__(self, username, email):
        self.username = username
        self.email = email
```

Step 5-3: Update *src/__init__.py*, removing the route and model and adding the Application Factory pattern:

```python
# src/__init__.py


import os

from flask import Flask  # new
from flask_sqlalchemy import SQLAlchemy


# instantiate the db
db = SQLAlchemy()


# new
def create_app(script_info=None):

    # instantiate the app
    app = Flask(__name__)

    # set config
    app_settings = os.getenv('APP_SETTINGS')
    app.config.from_object(app_settings)

    # set up extensions
    db.init_app(app)

    # register blueprints
    from src.api.ping import ping_blueprint
    app.register_blueprint(ping_blueprint)

    # shell context for flask cli
    @app.shell_context_processor
    def ctx():
        return {'app': app, 'db': db}

    return app
```

Take note of the `shell_context_processor`. This is used to register the `app` and `db` to the shell. Now we can work with the application context and the database without having to import them directly into the shell, which you'll see shortly.

Step 5-4: Update *manage.py*:

```
# manage.py


import sys

from flask.cli import FlaskGroup

from src import create_app, db   # new
from src.api.models import User  # new


app = create_app()  # new
cli = FlaskGroup(create_app=create_app)  # new


@cli.command('recreate_db')
def recreate_db():
    db.drop_all()
    db.create_all()
    db.session.commit()


if __name__ == '__main__':
    cli()
```

Now, you can work with the app and db context directly:

```
$ docker-compose up -d
$ docker-compose exec api flask shell

Python 3.11.2 (main, Feb  9 2023, 05:22:10) [GCC 8.3.0] on linux
App: src [development]
Instance: /usr/src/app/instance

>>> app
<Flask 'src'>

>>> db
```

```
<SQLAlchemy engine=postgres://postgres:***@api-db:5432/api_dev>

>>> exit()
```

Step 5-5: Import `create_app` into *src/tests/conftest.py*, which we can use to return an instance of the app:

```
# src/tests/conftest.py


import pytest

from src import create_app, db  # updated


@pytest.fixture(scope='module')
def test_app():
    app = create_app()  # new
    app.config.from_object('src.config.TestingConfig')
    with app.app_context():
        yield app  # testing happens here


@pytest.fixture(scope='module')
def test_database():
    db.create_all()
    yield db  # testing happens here
    db.session.remove()
    db.drop_all()
```

Step 5-6: Finally, remove the `FLASK_APP` environment variable from *docker-compose.yml*:

```
environment:
  - FLASK_DEBUG=1
  - FLASK_ENV=development
  - APP_SETTINGS=src.config.DevelopmentConfig
  - DATABASE_URL=postgresql://postgres:postgres@api-db:5432/api_dev
  - DATABASE_TEST_URL=postgresql://postgres:postgres@api-db:5432/api_test
```

Step 5-7: Test!

```
$ docker-compose up -d

$ docker-compose exec api python -m pytest "src/tests"
```

Step 5-8: Apply the model to the dev database:

```
$ docker-compose exec api python manage.py recreate_db
```

Step 5-9: Did this work? Let's hop into psql:

```
$ docker-compose exec api-db psql -U postgres

psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

postgres=# \c api_dev
You are now connected to database "api_dev" as user "postgres".

api_dev=# \dt
         List of relations
 Schema | Name  | Type  |  Owner
--------+-------+-------+----------
 public | users | table | postgres
(1 row)

api_dev=# \q
```

Correct any errors and move on.

Part6: RESTful Routes

Next, let's set up three new routes, following RESTful best practices, with TDD:

Endpoint	HTTP Method	CRUD Method	Result
/users	GET	READ	get all users
/users/:id	GET	READ	get a single user
/users	POST	CREATE	add a user

For each, we'll:

1. write a test
2. run the test, to ensure it fails (**red**)
3. write just enough code to get the test to pass (**green**)
4. **refactor** (if necessary)

Step 6-1: Let's start with the POST route.

POST Route

First, add a new file called *test_users.py* to "src/tests". Add the following test:

```python
# src/tests/test_users.py


import json


def test_add_user(test_app, test_database):
    client = test_app.test_client()
    resp = client.post(
        '/users',
        data=json.dumps({
            'username': 'john',
            'email': 'john@algonquincollege.com'
        }),
        content_type='application/json',
    )
    data = json.loads(resp.data.decode())
    assert resp.status_code == 201
    assert 'john@algonquincollege.com was added!' in data['message']
```

Did you notice that we passed in both test fixtures, `test_app` and `test_database`?

Run the test to ensure it fails:

```
$ docker-compose exec api python -m pytest "src/tests"
```

Step 6-2: Then add the route handler to *src/api/users.py*:

```python
# src/api/users.py


from flask import Blueprint, request
from flask_restx import Resource, Api

from src import db
from src.api.models import User


users_blueprint = Blueprint('users', __name__)
api = Api(users_blueprint)


class UsersList(Resource):

    def post(self):
        post_data = request.get_json()
        username = post_data.get('username')
        email = post_data.get('email')

        db.session.add(User(username=username, email=email))
        db.session.commit()

        response_object = {
            'message': f'{email} was added!'
        }
        return response_object, 201


api.add_resource(UsersList, '/users')
```

Here, we linked a Flask-RESTX API to the Flask Blueprint and then defined the route handler.

It's worth noting that Flask-RESTX has a Namespace construct that's used for organizing RESTful resources and endpoints within a given API. They can be used with or without Blueprints. We'll use them later in the course. Review Scaling your project from the docs for more info.

Step 6-3: Register the Blueprint in *src/__init__.py*:

```
from src.api.users import users_blueprint
app.register_blueprint(users_blueprint)
```

Step 6-3: Run the tests. They all should pass:

```
======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 5 items

src/tests/test_config.py ...
[ 60%]
src/tests/test_ping.py .
[ 80%]
src/tests/test_users.py .
[100%]

========================================= 5 passed in 0.38s
=========================================
```

Did a warning get outputted? Pass `-p no:warnings` on the command-line in to disable them.

What about errors and exceptions? Like:

1. A payload is not sent
2. The payload is invalid -- e.g., the JSON object is empty or it contains the wrong keys
3. The user already exists in the database

Step 6-4: Add some tests:

```
def test_add_user_invalid_json(test_app, test_database):
    client = test_app.test_client()
    resp = client.post(
        '/users',
        data=json.dumps({}),
        content_type='application/json',
    )
    data = json.loads(resp.data.decode())
    assert resp.status_code == 400
    assert 'Input payload validation failed' in data['message']


def test_add_user_invalid_json_keys(test_app, test_database):
    client = test_app.test_client()
    resp = client.post(
        '/users',
        data=json.dumps({"email": "john@testdriven.io"}),
        content_type='application/json',
    )
    data = json.loads(resp.data.decode())
    assert resp.status_code == 400
    assert 'Input payload validation failed' in data['message']


def test_add_user_duplicate_email(test_app, test_database):
    client = test_app.test_client()
    client.post(
        '/users',
        data=json.dumps({
            'username': 'john',
            'email': 'john@algonquincollege.com'
        }),
        content_type='application/json',
    )
    resp = client.post(
        '/users',
        data=json.dumps({
            'username': 'john',
            'email': 'john@algonquincollege.com'
        }),
        content_type='application/json',
    )
    data = json.loads(resp.data.decode())
    assert resp.status_code == 400
    assert 'Sorry. That email already exists.' in data['message']
```

Step 6-5: Ensure the tests fail, and then update the route handler:

```
class UsersList(Resource):
    def post(self):
        post_data = request.get_json()
        username = post_data.get('username')
        email = post_data.get('email')
        response_object = {}
        user = User.query.filter_by(email=email).first()
        if user:
            response_object['message'] = 'Sorry. That email already exists.'
            return response_object, 400

        db.session.add(User(username=username, email=email))
        db.session.commit()

        response_object['message'] = f'{email} was added!'
        return response_object, 201
```

The tests should still be failing.

Validation

Step 6-6: Next, to handle the validation of the JSON payload, we can use an API model to define the shape of the object:

```
user = api.model('User', {
    'id': fields.Integer(readOnly=True),
    'username': fields.String(required=True),
    'email': fields.String(required=True),
    'created_date': fields.DateTime,
})
```

Add this to *src/api/users.py* right before the `UsersList` class.

Here, we used the `api.model()` factory pattern to instantiate and register the user model to our API. We then defined the type of each field and passed in some optional arguments.

Review the docs or source code to learn more about the model fields.

Step 6-7: Add the import for `fields`:

```
from flask_restx import Resource, Api, fields
```

With that, we can use the `@api.expect` decorator to attach the model to the `post` method in order to validate the payload:

```
class UsersList(Resource):

    @api.expect(user, validate=True)
    def post(self):
        ...
```

You should now have:

```python
# src/api/users.py


from flask import Blueprint, request
from flask_restx import Resource, Api, fields  # updated

from src import db
from src.api.models import User


users_blueprint = Blueprint('users', __name__)
api = Api(users_blueprint)

# new
user = api.model('User', {
    'id': fields.Integer(readOnly=True),
    'username': fields.String(required=True),
    'email': fields.String(required=True),
    'created_date': fields.DateTime,
})


class UsersList(Resource):

    @api.expect(user, validate=True)  # new
    def post(self):
        post_data = request.get_json()
        username = post_data.get('username')
        email = post_data.get('email')
        response_object = {}

        user = User.query.filter_by(email=email).first()
        if user:
            response_object['message'] = 'Sorry. That email already exists.'
            return response_object, 400

        db.session.add(User(username=username, email=email))
        db.session.commit()

        response_object['message'] = f'{email} was added!'
        return response_object, 201


api.add_resource(UsersList, '/users')
```

Step 6-8: Now, let's re-run just the tests that failed during the last run with the `--lf` flag:

```
$ docker-compose exec api python -m pytest "src/tests" --lf

======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 7 items / 4 deselected / 3 selected
run-last-failure: rerun previous 3 failures

src/tests/test_users.py ...
[100%]

================================== 3 passed, 4 deselected in 0.23s
==================================
```

Ensure all the tests pass and then move on to the next route.

GET single user Route

Step 6-9: Start with a test:

```python
def test_single_user(test_app, test_database):
    user = User(username='jeffrey', email='jeffrey@testdriven.io')
    db.session.add(user)
    db.session.commit()
    client = test_app.test_client()
    resp = client.get(f'/users/{user.id}')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert 'jeffrey' in data['username']
    assert 'jeffrey@testdriven.io' in data['email']
```

Add the following imports:

```python
from src import db
from src.api.models import User
```

Ensure the test fails.

Marshalling

Step 6-10: Add the following route:

```
class Users(Resource):

    @api.marshal_with(user)
    def get(self, user_id):
        return User.query.filter_by(id=user_id).first(), 200
```

We used the `@api.marshal_with()` decorator here and passed in the user model. This model is now being used as a serializer to generate a JSON object with the fields from the model.

Review the Response marshalling documentation for more info.

Step 6-11: Add the resource to the API:

```
api.add_resource(Users, '/users/<int:user_id>')
```

The tests should pass. Next, what if the `id` doesn't exist?

Add a test:

```
def test_single_user_incorrect_id(test_app, test_database):
    client = test_app.test_client()
    resp = client.get('/users/999')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 404
    assert 'User 999 does not exist' in data['message']
```

Updated view:

```
class Users(Resource):

    @api.marshal_with(user)
    def get(self, user_id):
        user = User.query.filter_by(id=user_id).first()
        if not user:
            api.abort(404, f"User {user_id} does not exist")
        return user, 200
```

GET all users Route

Step 6-12: Again, let's start with a test.

Since we'll have to add a few users first, let's add a fixture that uses the "factory as fixture" pattern to *src/tests/conftest.py*:

```
@pytest.fixture(scope='function')
def add_user():
    def _add_user(username, email):
        user = User(username=username, email=email)
        db.session.add(user)
        db.session.commit()
        return user
    return _add_user
```

Add the import:

```
from src.api.models import User
```

Now, refactor the *test_single_user* test, like so:

```
def test_single_user(test_app, test_database, add_user):
    user = add_user('jeffrey', 'jeffrey@testdriven.io')
    client = test_app.test_client()
    resp = client.get(f'/users/{user.id}')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert 'jeffrey' in data['username']
    assert 'jeffrey@testdriven.io' in data['email']
```

Remove the `db` import:

```
from src import db
```

Make sure the tests still pass.

Step 6-13: With that, let's add the new test:

```
def test_all_users(test_app, test_database, add_user):
    add_user('john', 'john@algonquincollege.com')
    add_user('fletcher', 'fletcher@notreal.com')
    client = test_app.test_client()
    resp = client.get('/users')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert len(data) == 2
    assert 'john' in data[0]['username']
    assert 'john@algonquincollege.com' in data[0]['email']
    assert 'fletcher' in data[1]['username']
    assert 'fletcher@notreal.com' in data[1]['email']
```

Make sure it fails. Then add the view to the `UsersList` resource:

```
@api.marshal_with(user, as_list=True)
def get(self):
    return User.query.all(), 200
```

The `as_list=True` argument indicates that we want to return a list of objects rather than a single object.

Does the test past?

It should pass if you run the test by itself:

```
$ docker-compose exec api python -m pytest "src/tests" --lf

======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 5 items / 4 deselected / 1 selected
run-last-failure: rerun previous 1 failure

src/tests/test_users.py .
[100%]

================================== 1 passed, 4 deselected in 0.26s
=================================
```

However, It should fail when you run all the tests:

```
>       assert len(data['data']['users']) == 2
E       AssertionError: assert 4 == 2
```

Why are there four users?

Essentially, we added a single user each in the `test_add_user` and `test_single_user` tests. Two more get added in the `test_all_users`. We could fix this by changing the test to:

```
def test_all_users(test_app, test_database, add_user):
    add_user('john', 'john@algonquincollege.com')
    add_user('fletcher', 'fletcher@notreal.com')
    client = test_app.test_client()
    resp = client.get('/users')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert len(data) == 4
    assert 'john' in data[2]['username']
    assert 'john@algonquincollege.com' in data[2]['email']
    assert 'fletcher' in data[3]['username']
    assert 'fletcher@notreal.com' in data[3]['email']
```

However, what would happen if another developer changed the number of users added in either `test_add_user` or `test_single_user`?

When writing tests, it's a good idea to run each test in isolation to ensure that each can be run mostly on their own. This makes it easier to run tests in parallel and it helps reduce flaky tests. When there's shared testing code, it's easy for other developers to change that code and have other tests break or produce false positives. Don't worry so much about keeping your tests DRY, in other words.

Step 6-14: Let's remove all the rows in the table at the start of the test:

```
def test_all_users(test_app, test_database, add_user):
    test_database.session.query(User).delete()  # new
    add_user('john', ' john@algonquincollege.com')
    add_user('fletcher', 'fletcher@notreal.com')
    client = test_app.test_client()
    resp = client.get('/users')
    data = json.loads(resp.data.decode())
    assert resp.status_code == 200
    assert len(data) == 2
    assert 'john' in data[0]['username']
    assert 'john@algonquincollege.com' in data[0]['email']
    assert 'fletcher' in data[1]['username']
    assert 'fletcher@notreal.com' in data[1]['email']
```

Do the tests pass now?

```
======================================== test session starts
========================================
platform linux -- Python 3.11.2, pytest-7.2.1, pluggy-1.0.0
rootdir: /usr/src/app/src/tests, configfile: pytest.ini
collected 11 items

src/tests/test_config.py ...
[ 27%]
src/tests/test_ping.py .
[ 36%]
src/tests/test_users.py .......
[100%]

======================================== 11 passed in 0.34s
=========================================
```

Step 6-15: Before moving on, let's test the route in the browser: http://localhost:5004/users. You should see:

```
[]
```

Add a seed command to the *manage.py* file to populate the database with some initial data:

```
@cli.command('seed_db')
def seed_db():
    db.session.add(User(username='john', email="john@algonquincollege.com"))
    db.session.add(User(username='Dave', email="dave@algonquincollege.com"))
    db.session.commit()
```

Try it out:

```
$ docker-compose exec api python manage.py seed_db
```

Step 6-16: Make sure you can view the users in the JSON response http://localhost:5004/users:

```
[
    {
        "id": 1,
        "username": "john",
        "email": "john@algonquincollege.com",
        "created_date": "2023-10-30T01:47:15.510362"
    },
    {
        "id": 2,
        "username": "dave",
        "email": "dave@algonquincollege.com",
        "created_date": "2023-10-30T01:47:15.510362"
    }
]
```

Think about how you could trim down some of the tests with shared setup code. Try not to sacrifice readability if you do decide to refactor. Remember: If your code is easy to test it usually means that it's easy to use as well.

Pytest Commands

Step 6-17: Let's review some useful pytest commands:

```
# normal run
$ docker-compose exec api python -m pytest "src/tests"

# disable warnings
$ docker-compose exec api python -m pytest "src/tests" -p no:warnings

# run only the last failed tests
$ docker-compose exec api python -m pytest "src/tests" --lf

# run only the tests with names that match the string expression
$ docker-compose exec api python -m pytest "src/tests" -k "config and not
test_development_config"

# stop the test session after the first failure
$ docker-compose exec api python -m pytest "src/tests" -x

# enter PDB after first failure then end the test session
$ docker-compose exec api python -m pytest "src/tests" -x --pdb

# stop the test run after two failures
$ docker-compose exec api python -m pytest "src/tests" --maxfail=2

# show local variables in tracebacks
$ docker-compose exec api python -m pytest "src/tests" -l

# list the 2 slowest tests
$ docker-compose exec api python -m pytest "src/tests" --durations=2
```

Lab Submission

- Complete the remaining routes.
- Write a brief summary of the app created and describe any challenges faced.
- Include a link to your GitHub repo which has the final version of your app.