1 Introduction

This document explains the test strategy for Assignment 1: API Testing, Coverage, and Benchmarking. In this assignment, we will build a simple REST API using Python and make sure it works correctly. We will test all its endpoints, measure test coverage, and check its performance with load testing tools.

2 Test Strategy Design

The goal of this strategy is to describe how we will test the API and make sure it is correct, reliable, and fast. We will use the following approaches:

- **Unit Testing** to check each function or class works correctly.
- **Integration Testing** to check how different parts of the API work together.
- **API Testing with HTTP Files and REST Client** to manually send requests and see responses.
- **Code Coverage Measurement** to see how much of our code is tested.
- **Basic Load Testing with JMeter** to check how the API performs with many users.

2.1 Scope

We will test all main parts of the REST API:

- Creating, reading, updating, and deleting resources (CRUD operations).
- Making sure the API returns the right status codes (200, 201, 400, 404).
- Making sure the API handles invalid inputs and errors properly.
- Measuring code coverage and improving tests to reach at least 80%.
- Measuring performance under small and medium loads.

2.2 Testing Types

Unit Testing: Checks a single function or class in isolation. *Goal:* Find problems early and keep the code easy to maintain.

Integration Testing: Checks if different modules (routes, services, database) work together. *Goal:* Make sure data flows correctly through the whole API.

API Testing Using HTTP Files and REST Client: Send real HTTP requests to the API and see if the responses match what we expect. *Goal:* Make sure endpoints work as described.

Basic Load Testing with JMeter: Send many requests at the same time to check how the API performs under load. *Goal:* Measure response time, throughput, and error rate.

3 Tools

- **Python + Pytest:** For writing and running automated tests.
- **Coverage.py:** For measuring code coverage.
- **HTTP Files / REST Client:** For manual API testing.
- **JMete:** For load and performance testing.

4 Unit and Integration Tests

Each test will answer:

1. What function or endpoint is tested?
2. What input is used?
3. What result or response is expected?
4. What is the purpose of this test?

5 Coverage Goals

We aim for at least **80% code coverage**. This means most of the code will be tested to reduce bugs.

6 Performance Testing

We will use JMeter to:

- Measure average response time.
- Measure requests per second (throughput).
- Find errors when many users use the API.

7 Exit Criteria

- All unit and integration tests pass.
- Code coverage is 80% or higher.
- No major bugs remain.
- Performance is within acceptable limits.

8 Python Implementation Checklist

8.1 Build a REST API

Implemented using **FastAPI** framework. The API is in `main.py` with CRUD endpoints for tasks.

8.2 CRUD Operations

All CRUD operations (Create, Read, Update, Delete) are available via endpoints: `/tasks` and `/tasks/{id}`.

8.3 Unit Testing (Core Logic)

Unit tests are written using `pytest` to test the `tasks` list for adding, updating, and deleting tasks. Tests are located in `test_main.py`.

8.4 Integration Testing (API Endpoints)

Integration tests use `pytest` and the `requests` library to test all API endpoints and ensure proper responses.

8.5 API Testing using HTTP Files / REST Client

A `.http` file in VS Code or Postman requests is created for testing the API endpoints interactively.

8.6 Code Coverage

The `coverage.py` library is used to measure test coverage. Commands: `coverage run -m pytest` and `coverage report -m`.

8.7 Performance / Load Testing

JMeter or k6 can be used to perform load and performance testing on `/tasks` endpoints. The Python API is compatible with these tools.

8.8 Persistence

Tasks are stored in a JSON file (`tasks.json`) to persist data between API runs.

8.9 Automatic Task IDs

The API automatically assigns unique IDs to new tasks, simplifying task creation and avoiding conflicts.

8.10 Local Testing

The API runs locally on `http://127.0.0.1:8000` and can be tested via Swagger UI, HTTP files, or Postman.

8.11 SQALE / Code Quality Principles

Python code is structured for readability and maintainability. Unit and integration tests ensure correctness and modularity.

9 Reflection on Coverage and Performance

9.1 Code Coverage

During this project, we used `pytest` and `coverage.py` to make sure my code was tested thoroughly. We started by writing unit tests for the core logic of our Task Manager application. This included functions for creating, updating, and deleting tasks, as well as checking that invalid input is handled correctly. Then, we wrote integration tests that simulated real API calls using the FastAPI test client. These tests called the actual endpoints like `/tasks`, `/tasks/{id}`, and verified the correct status codes (200, 404) and responses. After running the tests, wevgenerated a coverage report which showed 100% coverage. This gave us confidence that all parts of our code were being executed during tests.

9.2 Balance Between Unit and Integration Tests

We maintained a balance between unit and integration testing. Unit tests helped us catch small logic issues quickly, as they run very fast and focus on one function at a time. Integration tests were useful to make sure that all parts of the system work correctly together, such as the routing, JSON storage, and response formatting. Having both types of tests meant that if something broke, we could easily find out whether the problem was in a single function or in how components interact.

9.3 Importance of Code Coverage

Code coverage is very important because it shows which parts of the code are tested and which are not. If coverage is low, there is a higher chance of bugs hiding in untested parts of the code. Reaching high coverage, like 80% or more, increases confidence before deploying the API. However, we also learned that 100% coverage does not guarantee bug-free code. The quality of the tests is also important. For example, just calling a function without checking its result is not enough. We made sure our tests checked the expected output and the status codes so that they are meaningful.

9.4 Performance and Optimization

We performed basic load testing using JMeter with multiple virtual users making requests to the API at the same time. This helped us see how the API behaves under load. The performance was good for a small number of users, but we noticed that since we are storing data in a JSON file, it may become slower if many tasks are added or if many users use the system at once. In a real production system, it would be better to use a database like mysql or PostgreSQL for better scalability and faster queries.

9.5 Conclusion

Overall, we learned that unit tests and integration tests together help create a reliable API. High coverage is a good goal because it reduces the risk of missed bugs, but it should be combined with meaningful test cases. Load testing was helpful to understand the performance limits of the API and think about improvements for larger systems. This project showed us how testing and performance analysis are key parts of building high-quality software, not just optional steps.