Test Strategy Plan

End-to-End Test Engineering for an Appointment and Dynamic Pricing Service

Project: Fitness Centre REST API

Author: Mehmet Kaplan

Date: December 2025

Version: 2.0

§ 1. Introduction

This document defines an end-to-end test strategy for the **Fitness Centre Appointment and Dynamic Pricing Service**. The goal is to verify functional correctness and demonstrate advanced software test engineering techniques including unit, integration, system, performance, security, chaos, and formal verification testing.

The testing approach aims to ensure code quality through rigorous coverage metrics, validate non-functional requirements, and apply state-of-the-art testing methodologies. The system is built using **Java 21** and **Spring Boot 3**.

§ 2. Test Levels & Scope

The following comprehensive test levels will be applied to ensure end-to-end quality assurance:

- **Unit Testing:** Focuses on testing individual components using JUnit 5, Mockito, and Test-Driven Development (TDD) principles. External dependencies are mocked for isolation.
- **Integration Testing:** Verifies the interaction between modules, specifically the Controller-Service-Repository flow using H2 in-memory database.
- **System / API Testing:** Black-box testing of REST API endpoints using Postman and Newman to ensure correct HTTP responses and payload handling.
- **Advanced Test Design:** Application of Decision Tables, Control Flow Graphs (CFG), and Data Flow Analysis for systematic test case design.
- **Property-Based & Model-Based Testing:** Automated input generation using jqwik and State Machine testing for complex workflows.
- **Performance Testing:** Load and stress testing using k6 to evaluate system behavior under concurrent user requests.
- **Security Testing:** Dynamic Application Security Testing (DAST) using OWASP ZAP and threat modeling with STRIDE methodology.
- **Chaos Engineering:** Testing system resilience through Docker container failure scenarios and network disruption simulations.
- **Formal Verification:** Design-by-contract assertions and TLA+ specifications for critical business logic validation.

§ 3. Tools & Frameworks

The project will utilize the following tools and libraries:

Category	Tool/Framework	Purpose
Development	Java 21, Spring Boot 3, Maven	Core application framework
Unit Testing	JUnit 5, Mockito	Isolated component testing with TDD approach
Code Coverage	JaCoCo	Line and branch coverage measurement
Mutation Testing	PITest	Test effectiveness analysis
Property-Based Testing	jqwik	Automated input generation for invariant testing
Combinatorial Testing	ACTS (NIST)	Pairwise test case generation
API Automation	Postman & Newman	REST API functional testing
Performance Testing	k6	Load and stress testing
Security	OWASP ZAP	Vulnerability scanning (DAST) and STRIDE analysis
Formal Methods	TLA+	Specification and verification of concurrent systems
CI/CD & Containerization	GitHub Actions, Docker	Automated testing pipeline and deployment

§ 4. Coverage & Quality Goals

To ensure high maintainability and reliability, the project aims for the following strict coverage metrics:

> - **Line Coverage:** > 80%
> - **Branch Coverage:** > 70%
> - **Mutation Score:** Analysis of survived mutants will be performed to improve test quality and eliminate weak tests.

These targets will be continuously monitored through JaCoCo reports integrated into the CI/CD pipeline. Any decrease below the threshold will trigger a pipeline failure.

§ 5. Test Environment

The testing strategy employs multiple environments to isolate concerns and ensure comprehensive validation:

- **Local Environment (Unit Tests):** Tests run on the local development machine using Mockito to bypass the database and external dependencies.
- **Local Environment (Integration Tests):** Tests run using H2 In-Memory Database to verify data persistence, entity relationships, and repository methods without requiring external database setup.
- **CI/CD & System Environment:** The application runs in a Docker Container connected to a PostgreSQL container. Performance and Security scans are executed against this containerized environment to simulate production-like conditions.
- **Chaos Testing Environment:** Dockerized environment with controlled failure injection for resilience testing.

§ 6. Exit Criteria (Success Definition)

The testing phase will be considered complete and successful when the following criteria are met:

- All functional requirements (Member Management, Class Management, Reservation Management, Dynamic Pricing) are implemented and verified.
- Coverage targets (80% Line Coverage / 70% Branch Coverage) are achieved and validated by JaCoCo reports.
- All automated Postman tests pass successfully in the CI pipeline with 100% success rate.
- No critical or high-severity security vulnerabilities are detected by OWASP ZAP scanning.
- Performance testing demonstrates acceptable response times (p95 < 2000 ms) under expected load (50+ concurrent users).
- All survived mutants from PITest have been analyzed and addressed with improved test cases.
- State machine models have been validated and all state transitions tested.
- Chaos engineering tests demonstrate acceptable system resilience and recovery.