

Project (First Part)

Individual report

QSOFT

Master in Informatics Engineering - 2024/2025

Porto, November 8, 2024

Pedro Miguel Santos Coelho

Version 7, 2024-11-08

Revision History

| Revision | Date | Author(s) | Description |
|----------|------------|--------------|---|
| 1 | 2024-10-23 | Pedro Coelho | Initial version |
| 2 | 2024-11-05 | Pedro Coelho | Performance |
| 3 | 2024-11-06 | Pedro Coelho | Funcional Correctness |
| 4 | 2024-11-06 | Pedro Coelho | Maintainability |
| 5 | 2024-11-07 | Pedro Coelho | Security |
| 6 | 2024-11-08 | Pedro Coelho | Update Performance |
| 7 | 2024-11-08 | Pedro Coelho | Architectural Compliance and Conclusion |

Contents

| | |
|--|------------|
| List of Figures | vii |
| 1 Introduction | 1 |
| 2 Functional correctness | 3 |
| 2.1 Project Code Coverage | 3 |
| 2.2 PetType Code Coverage | 4 |
| 2.2.1 Mapper | 5 |
| 2.2.2 Rest Controller | 5 |
| 2.2.3 Model | 6 |
| 2.3 Flaky Tests | 6 |
| 3 Maintainability | 9 |
| 3.1 Coupling and Structural Erosion | 9 |
| 3.2 Size and Complexity | 10 |
| 3.3 Cyclomatic Complexity to three methods | 10 |
| 3.4 Instability | 12 |
| 3.5 Maintainability of test code | 13 |
| 4 Performance | 15 |
| 4.1 Requirements | 15 |
| 4.2 Testing Environment | 16 |
| 4.3 Load Test | 16 |
| 4.3.1 Scenario GET | 16 |
| 4.3.2 Jmeter | 17 |
| 4.3.3 K6 | 18 |
| 4.3.4 Scenario POST | 19 |
| 4.3.5 Jmeter | 19 |
| 4.3.6 K6 | 21 |
| 4.4 Stress Test | 22 |
| 4.4.1 Scenario GET | 23 |

Contents

| | | |
|----------|---|-----------|
| 4.4.2 | Jmeter | 23 |
| 4.4.3 | K6 | 24 |
| 4.4.4 | Scenario POST | 25 |
| 4.4.5 | Jmeter | 25 |
| 4.4.6 | K6 | 26 |
| 4.5 | Soak Test | 27 |
| 4.5.1 | Scenario GET | 28 |
| 4.5.2 | Jmeter | 28 |
| 4.5.3 | K6 | 29 |
| 4.5.4 | Scenario POST | 30 |
| 4.5.5 | Jmeter | 30 |
| 4.5.6 | K6 | 32 |
| 4.6 | Conclusion | 33 |
| 5 | Security | 35 |
| 5.1 | Dependency Checker | 35 |
| 5.2 | Spring Core Dependency | 35 |
| 5.2.1 | Identified Vulnerabilities | 36 |
| 5.2.2 | CVSS3 Scale | 36 |
| 6 | Architectural compliance | 37 |
| 6.1 | Package Dependency | 37 |
| 6.1.1 | Code | 37 |
| 6.1.2 | Result | 38 |
| 6.2 | Class Dependency | 38 |
| 6.2.1 | Code | 38 |
| 6.2.2 | Result | 38 |
| 6.3 | Class and Package Containment | 39 |
| 6.3.1 | Code | 39 |
| 6.3.2 | Result | 40 |
| 6.4 | Inheritance | 40 |
| 6.4.1 | Code | 40 |
| 6.4.2 | Result | 40 |
| 6.5 | Annotation | 41 |
| 6.5.1 | Code | 41 |
| 6.5.2 | Result | 41 |

| | | |
|----------|----------------------|-----------|
| 6.6 | Layer | 42 |
| 6.6.1 | Code | 42 |
| 6.6.2 | Result | 42 |
| 6.7 | Cycle | 43 |
| 6.7.1 | Code | 43 |
| 6.7.2 | Result | 43 |
| 6.8 | Conclusion | 43 |
| 7 | Conclusions | 45 |
| | References | 47 |

List of Figures

| | | |
|------|--|----|
| 2.1 | Overall Coverage | 4 |
| 2.2 | PetTypeMapperImpl Coverage | 5 |
| 2.3 | PetTypeModel Coverage | 5 |
| 2.4 | PetTypeModel Coverage | 6 |
| 2.5 | Flaky Tests Report | 7 |
| | | |
| 3.1 | ACD Metric | 9 |
| 3.2 | Number Of Component and Sources Metric | 10 |
| 3.3 | getPetType Method | 11 |
| 3.4 | updatePetType Method | 11 |
| 3.5 | deletePetType Method | 11 |
| 3.6 | Dependencies PetType | 12 |
| 3.7 | TestSmells PetType | 13 |
| | | |
| 4.1 | System Details | 16 |
| 4.2 | Jmeter Load Configuration | 17 |
| 4.3 | Duration GET Load Configuration | 17 |
| 4.4 | GET Load Test Aggregate Report | 17 |
| 4.5 | GET Load Test Responses Over Time | 18 |
| 4.6 | K6 Load Configuration | 18 |
| 4.7 | K6 GET LOAD | 19 |
| 4.8 | LoadCSV Jmeter | 19 |
| 4.9 | LoadCSV Jmeter2 | 20 |
| 4.10 | Jmeter Load Configuration | 20 |
| 4.11 | Duration POST Load Configuration | 20 |
| 4.12 | POST Load Test Aggregate Report | 21 |
| 4.13 | GET Post Test Responses Over Time | 21 |
| 4.14 | LoadCSV K6 | 21 |
| 4.15 | K6 Load Configuration | 22 |
| 4.16 | K6 POST LOAD | 22 |

List of Figures

| | | |
|------|---|----|
| 4.17 | Jmeter Stress Configuration | 23 |
| 4.18 | Duration GET Stress Configuration | 23 |
| 4.19 | GET Stress Test Aggregate Report | 23 |
| 4.20 | GET Stress Test Responses Over Time | 24 |
| 4.21 | K6 Stress Configuration | 24 |
| 4.22 | K6 GET STRESS | 25 |
| 4.23 | Jmeter Stress Configuration | 25 |
| 4.24 | Duration POST Stress Configuration | 26 |
| 4.25 | POST Stress Test Aggregate Report | 26 |
| 4.26 | POST Stress Test Responses Over Time | 26 |
| 4.27 | K6 Stress Configuration | 27 |
| 4.28 | K6 POST STRESS | 27 |
| 4.29 | Jmeter SOAK Configuration | 28 |
| 4.30 | Duration GET Soak Configuration | 28 |
| 4.31 | GET Soak Test Aggregate Report | 29 |
| 4.32 | GET Soak Test Responses Over Time | 29 |
| 4.33 | K6 Soak Configuration | 29 |
| 4.34 | K6 GET SOAK | 30 |
| 4.35 | Jmeter Soak Configuration | 31 |
| 4.36 | Duration POST Soak Configuration | 31 |
| 4.37 | POST Soak Test Aggregate Report | 31 |
| 4.38 | POST Soak Test Responses Over Time | 31 |
| 4.39 | K6 Soak Configuration | 32 |
| 4.40 | K6 POST SOAK | 32 |
| 6.1 | Package Dependency Code | 37 |
| 6.2 | Package Dependency Test Result | 38 |
| 6.3 | Class Dependency Code | 38 |
| 6.4 | Class Dependency Test Result | 38 |
| 6.5 | Class and Package Containment Code | 39 |
| 6.6 | Class and Package Containment Test Result | 40 |
| 6.7 | Inheritance Code | 40 |
| 6.8 | Inheritance Test Result 1 | 40 |
| 6.9 | Inheritance Test Result 1 | 41 |
| 6.10 | Annotation Code | 41 |
| 6.11 | Annotation Test Result | 41 |

| | |
|----------------------------------|----|
| 6.12 Layer Code | 42 |
| 6.13 Layer Test Result | 42 |
| 6.14 Cycle | 43 |
| 6.15 Cycle Test Result | 43 |

1 Introduction

This document outlines the contributions of Pedro Coelho for the "QSOF" UC of the Master's program. The primary aim was to examine and assess the quality of an application named "Pet Clinic," which functions as the backend for an online Pet Clinic management. The application is built in Java, utilizing the Spring Boot, with a Domain-Driven Design (DDD) architecture. While the main emphasis is placed on evaluating the "petType" aggregate, the report occasionally considers the application as a whole. The evaluation addresses several key areas: maintainability, performance, security vulnerabilities, adherence to architectural principles, and adequacy of testing.

2 Functional correctness

Functional correctness measures the extent to which software performs its specified functions accurately. To assess this, unit and integration tests are crucial, as they validate that each component works correctly in isolation and in combination with others. Code coverage is a key metric here, as it indicates how much of the code is exercised during testing. High coverage suggests well-tested code, while low coverage may point to areas needing further validation to minimize potential risks in reuse. In this analysis, both line and branch coverage are evaluated to ensure that critical parts of the software have been thoroughly tested.

2.1 Project Code Coverage

Code coverage is a critical metric in evaluating the thoroughness of software testing, indicating how much of the codebase is exercised by tests. Coverage metrics, such as line and branch coverage, reveal the extent to which code paths and statements are executed during testing, which directly impacts functional correctness and robustness.

To measure code coverage in this project, we will use JaCoCo, a popular open-source tool for Java projects. JaCoCo generates comprehensive reports that highlight covered and uncovered sections, allowing the team to pinpoint specific areas of their model that might need more testing. This targeted approach ensures that the project meets current quality standards and supports informed decisions on software reuse.

2 Functional correctness

The figure below (Figure 2.1) showcases the JaCoCo coverage results for this project, illustrating the current level of code coverage of all the application.

spring-petclinic-rest

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes | |
|--|------------------------|--------------|------------------------|------------|--------|------|--------|-------|--------|---------|--------|---------|----|
| org.springframework.samples.petclinic.mapper | <div><div></div></div> | 80% | <div><div></div></div> | 57% | 55 | 123 | 79 | 342 | 5 | 55 | 0 | 7 | |
| org.springframework.samples.petclinic.rest.controller | <div><div></div></div> | 88% | <div><div></div></div> | 76% | 21 | 95 | 33 | 285 | 5 | 57 | 0 | 10 | |
| org.springframework.samples.petclinic.repository.jdbc | <div><div></div></div> | 92% | <div><div></div></div> | 85% | 8 | 88 | 28 | 389 | 2 | 64 | 0 | 14 | |
| org.springframework.samples.petclinic.util | <div><div></div></div> | 19% | <div><div></div></div> | 40% | 12 | 14 | 26 | 30 | 8 | 9 | 1 | 2 | |
| org.springframework.samples.petclinic.model | <div><div></div></div> | 84% | <div><div></div></div> | 55% | 9 | 83 | 21 | 139 | 4 | 74 | 0 | 11 | |
| org.springframework.samples.petclinic.repository.jpa | <div><div></div></div> | 92% | <div><div></div></div> | 76% | 8 | 52 | 4 | 93 | 1 | 37 | 0 | 7 | |
| org.springframework.samples.petclinic.rest.advice | <div><div></div></div> | 51% | <div><div></div></div> | 50% | 4 | 7 | 7 | 14 | 3 | 6 | 0 | 2 | |
| org.springframework.samples.petclinic.security | <div><div></div></div> | 69% | <div><div></div></div> | n/a | 3 | 8 | 6 | 23 | 3 | 8 | 1 | 3 | |
| org.springframework.samples.petclinic.repository.springdatajpa | <div><div></div></div> | 95% | <div><div></div></div> | 66% | 4 | 14 | 0 | 32 | 0 | 8 | 0 | 4 | |
| org.springframework.samples.petclinic.service | <div><div></div></div> | 98% | <div><div></div></div> | 60% | 4 | 45 | 1 | 63 | 0 | 40 | 0 | 2 | |
| org.springframework.samples.petclinic | <div><div></div></div> | 37% | <div><div></div></div> | n/a | 1 | 2 | 2 | 3 | 1 | 2 | 0 | 1 | |
| org.springframework.samples.petclinic.config | <div><div></div></div> | 100% | <div><div></div></div> | n/a | 0 | 4 | 0 | 16 | 0 | 4 | 0 | 1 | |
| Total | | 794 of 5 875 | 86% | 113 of 342 | 66% | 129 | 535 | 207 | 1 429 | 32 | 364 | 2 | 64 |

Figure 2.1: Overall Coverage

The overall code coverage results indicate that 86% of the project's instructions are covered by tests, reflecting a solid testing foundation. The config package achieves perfect coverage at 100%, while the util package has the lowest at 19%. For branch coverage, the project scores 66% overall, with the repository.jdbc package leading at 85%. However, the model package shows lower branch coverage, suggesting untested logical paths within those classes. The services package, which handles complex operations, demonstrates strong test coverage, emphasizing its importance in the project.

2.2 PetType Code Coverage

In this section, I will verify and analyze the code coverage of the PetType class. In order to do that, I will focus on the packages that contain the functionalities of the Pet Clinic and the Pets. Those package are:

- Mapper
- Rest Controller
- Model

2.2.1 Mapper

PetTypeMapperImpl

Source file "org.springframework.samples.petclinic/mapper/PetTypeMapperImpl.java" was not found during generation of report.












| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|-----------------------------|---|------|---|------|--------|------|--------|-------|--------|---------|
| toPetType(PetTypeDto) |  | 0% |  | 0% | 2 | 2 | 6 | 6 | 1 | 1 |
| toPetTypeDtos(Collection) |  | 93% |  | 75% | 1 | 3 | 1 | 7 | 0 | 1 |
| toPetTypeDto(PetType) |  | 88% |  | 50% | 1 | 2 | 1 | 6 | 0 | 1 |
| toPetType(PetTypeFieldsDto) |  | 85% |  | 50% | 1 | 2 | 1 | 5 | 0 | 1 |
| toPetTypeFieldsDto(PetType) |  | 85% |  | 50% | 1 | 2 | 1 | 5 | 0 | 1 |
| PetTypeMapperImpl() |  | 100% | n/a | n/a | 0 | 1 | 0 | 1 | 0 | 1 |
| Total | 26 of 96 | 72% | 6 of 12 | 50% | 6 | 12 | 10 | 30 | 1 | 6 |

Figure 2.2: PetTypeMapperImpl Coverage

After examining the JaCoCo analysis of PetTypeMapperImpl, Figure 2.2, we see the 72% coverage and 50% branch coverage, indicating that, even though the majority of the code is being tested, there is still a significant portion of code that remains untested, which could potentially affect the reliability of the application. The methods toPetTypeDtos(Collection), toPetTypeDto(PetType), toPetType(PetTypeFieldsDto), and toPetTypeFieldsDto(PetType) have code coverages of 93%, 88%, 85%, and 85%, respectively, indicating that these methods are generally well-tested, though there may still be some areas for improvement to ensure comprehensive test coverage. Notably, the method toPetType(PetTypeDto) has 0% coverage, highlighting an area for improvement to enhance application reliability.

2.2.2 Rest Controller

PetTypeRestController












| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods |
|---|---|------|---|------|--------|------|--------|-------|--------|---------|
| updatePetType(Integer, PetTypeDto) |  | 83% |  | 50% | 1 | 2 | 1 | 6 | 0 | 1 |
| addPetType(PetTypeFieldsDto) |  | 100% |  | n/a | 0 | 1 | 0 | 5 | 0 | 1 |
| listPetTypes() |  | 100% |  | 100% | 0 | 2 | 0 | 4 | 0 | 1 |
| getPetType(Integer) |  | 100% |  | 100% | 0 | 2 | 0 | 4 | 0 | 1 |
| deletePetType(Integer) |  | 100% |  | 100% | 0 | 2 | 0 | 5 | 0 | 1 |
| PetTypeRestController(ClinicService, PetTypeMapper) |  | 100% | n/a | n/a | 0 | 1 | 0 | 4 | 0 | 1 |
| Total | 5 of 144 | 96% | 1 of 8 | 87% | 1 | 10 | 1 | 28 | 0 | 6 |

Figure 2.3: PetTypeModel Coverage

After examining the JaCoCo analysis of PetTypeRestController, Figure 2.3, we can see that the coverage is 96% and the branch coverage is 87%, indicating a strong testing foundation for the class. However, the branch coverage suggests that there is still one untested branch, which could lead to potential issues in specific scenarios.

2.2.3 Model

PetType

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Meth |
|-----------------------------|------------------------|------|-----------------|------|--------|------|--------|-------|--------|------|
| • PetType() | <div><div></div></div> | 100% | | n/a | 0 | 1 | 0 | 1 | 0 | |
| Total | 0 of 3 | 100% | 0 of 0 | n/a | 0 | 1 | 0 | 1 | 0 | |

Created with [JaCoCo](#) 0.8.12.202403310830

Figure 2.4: PetTypeModel Coverage

After examining the JaCoCo analysis of PetTypeModel, Figure 2.4, we can see that coverage analysis of the PetType model reveals an impressive 100% coverage rate, indicating that all instructions are tested. However, it is important to note that the model is relatively small, consisting of only three instructions. While the high coverage is good, the limited size of the model suggests that the coverage metric should be interpreted with caution.

2.3 Flaky Tests

Flaky tests are tests that sometimes pass and sometimes fail without any changes to the codebase, test logic, or environment. This inconsistency makes them unreliable and difficult to trust, as they can obscure real issues or cause false alarms. Therefore, the presence of flaky tests indicates a problem in the way the tests are constructed, making it essential to identify and resolve the root cause.

To help detect flaky tests, we will use the flaky-test-extractor-maven-plugin, which has been added to the pom.xml.

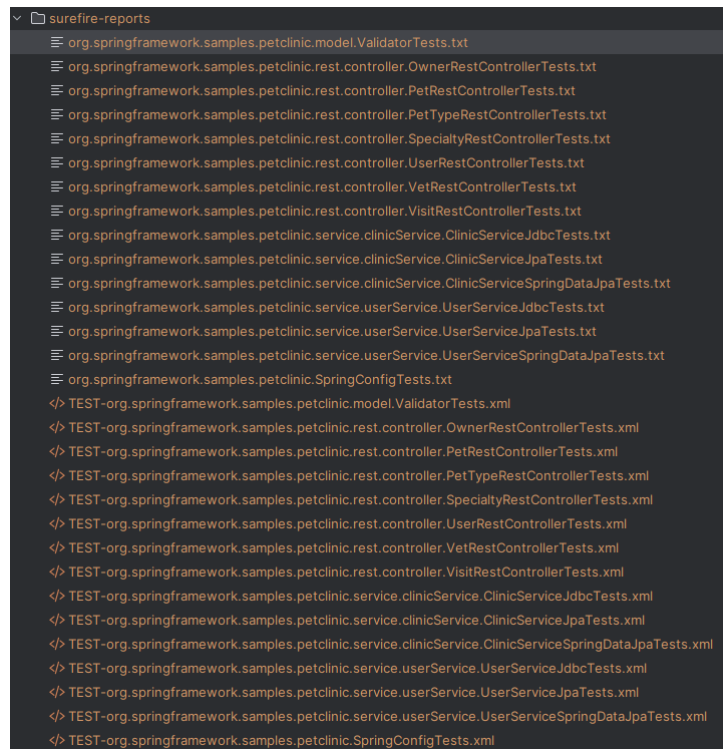


Figure 2.5: Flaky Tests Report

Based on the report in Figure 2.5 we can see that no file has the suffix -FLAKY, indicating that there are no flaky tests in this project.

3 Maintainability

Maintainability is the measure of how easily a software system can be altered after its initial deployment. Good maintainability affects how effortlessly a system can be updated, improved, or fixed to address new requirements, correct issues, or enhance performance. In this analysis, Sonargraph Explorer will be used to assess the relevant metrics in the subsequent sections.

3.1 Coupling and Structural Erosion

Metric: Average Component Dependency (ACD)

Category: Cohesion/Coupling (John Lakos)

Value: 7.66

The ACD value of 7.66 indicates that, on average, each component in the system depends on approximately 7-8 other components, both directly and indirectly. This level of dependency suggests a moderate degree of coupling among components. While the architecture is modular enough to allow individual components to function, this level of coupling may require careful attention when refactoring or updating components to prevent unintended interactions.

A high ACD value could signal challenges in maintaining and evolving the code, as each component’s dependencies may limit its flexibility and increase the likelihood of cascading changes. Reducing dependencies where possible can improve modularity and ease of maintenance.

| Element [1] | ACD |
|---|------|
|  PetClinic | 7,66 |

Figure 3.1: ACD Metric

3.2 Size and Complexity

Metric: Number of Components/Sources

Category: Size

Value: 131

The project consists of 131 components or source files, which provides a structural overview of its size and organization. In the context of a codebase with close 10000 lines of code, the distribution of 131 components suggests an average of approximately 74 lines per component. This number points to a relatively well-organized project structure, where code is compartmentalized across manageable components, aiding readability, traceability, and modularity. This organizational structure can support ongoing maintainability by making individual components easier to understand, modify, and test.

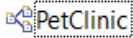
| Element [1] | Number of Components/Sources |
|---|------------------------------|
|  | 131 |

Figure 3.2: Number Of Component and Sources Metric

3.3 Cyclomatic Complexity to three methods

The cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program.

To calculate the cyclomatic complexity we use:

- $CC = D + 1$
- D = the number loop statement or conditional statement (Decision Points)

Now I will calculate the Cyclomatic Complexity of the following methods

getPetType

In the method 3.3, there is one decision point (the conditional statement if (petType == null)). This conditional statement divides the flow into two paths, and therefore, the Cyclomatic Complexity is:

$$CC = D + 1 = 1 + 1 = 2$$

```

@Override
public ResponseEntity<PetTypeDto> getPetType(Integer petTypeId) {
    PetType petType = this.clinicService.findPetTypeById(petTypeId);
    if (petType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    return new ResponseEntity<>(petTypeMapper.toPetTypeDto(petType), HttpStatus.OK);
}

```

Figure 3.3: getPetType Method

updatePetType

In the method 3.4, there is one decision point (the conditional statement if (currentPetType == null)). This conditional statement splits the flow into two possible paths. Therefore, the Cyclomatic Complexity is: $CC = D + 1 = 1 + 1 = 2$

```

@Override
public ResponseEntity<PetTypeDto> updatePetType(Integer petTypeId, PetTypeDto petTypeDto) {
    PetType currentPetType = this.clinicService.findPetTypeById(petTypeId);
    if (currentPetType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    currentPetType.setName(petTypeDto.getName());
    this.clinicService.savePetType(currentPetType);
    return new ResponseEntity<>(petTypeMapper.toPetTypeDto(currentPetType), HttpStatus.NO_CONTENT);
}

```

Figure 3.4: updatePetType Method

deletePetType

In the method ??, there is one decision point (the conditional statement if (currentPetType == null)). This conditional statement splits the flow into two possible paths. Therefore, the Cyclomatic Complexity is: $CC = D + 1 = 1 + 1 = 2$

```

@Transactional
@Override
public ResponseEntity<PetTypeDto> deletePetType(Integer petTypeId) {
    PetType petType = this.clinicService.findPetTypeById(petTypeId);
    if (petType == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    this.clinicService.deletePetType(petType);
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}

```

Figure 3.5: deletePetType Method

Based on the results, we can see that all three methods have a Cyclomatic Complexity

of 2, indicating a low level of complexity with only one decision point. This suggests that the methods are straightforward and require two test cases to fully cover both execution paths.

3.4 Instability

Instability measures a class's dependency on other classes and how much other classes depend on it. A high instability value indicates a class with many dependencies, while a low value suggests a more independent and stable class.

In order to calculate the Instability of the class we will use the following formula

- $\text{Instability} = E / (I + E)$
- E is the number of outgoing dependencies (Efferent Coupling).
- I is the number of incoming dependencies (Afferent Coupling).

We will calculate the Instability of the module PetType.java

| Incoming - To [23 elements] | Dependency Type | From | From scope | Number of depend.. |
|------------------------------|-----------------------|-----------------------------------|--------------------|--------------------|
| > PetType.java | Aggregated | AbstractClinicServiceTests.java | Main | 26 |
| > PetType.java | Aggregated | ClinicService.java | Main | 5 |
| > PetType.java | Aggregated | ClinicServiceImpl.java | Main | 6 |
| > PetType.java | Aggregated | JdbcOwnerRepositoryImpl.java | Main | 6 |
| > PetType.java | Aggregated | JdbcPetRepositoryImpl.java | Main | 9 |
| > PetType.java | Aggregated | JdbcPetTypeRepositoryImpl.java | Main | 20 |
| > PetType.java | Aggregated | JdbcVisitRepositoryImpl.java | Main | 5 |
| > PetType.java | Type Argument [In ... | JpaPetRepositoryImpl.java | Main | 1 |
| > PetType.java | Aggregated | JpaPetTypeRepositoryImpl.java | Main | 12 |
| > PetType.java | Aggregated | Pet.java | Main | 3 |
| > PetType.java | Aggregated | PetMapper.java | Main | 3 |
| > PetType.java | Aggregated | PetMapperImpl.java | Main | 12 |
| > PetType.java | Type Argument [In ... | PetRepository.java | Main | 1 |
| > PetType.java | Aggregated | PetTypeMapper.java | Main | 5 |
| > PetType.java | Aggregated | PetTypeMapperImpl.java | Main | 19 |
| > PetType.java | Aggregated | PetTypeRepository.java | Main | 5 |
| > PetType.java | Parameter | PetTypeRepositoryOverride.java | Main | 1 |
| > PetType.java | Aggregated | PetTypeRestController.java | Main | 7 |
| > PetType.java | Aggregated | PetTypeRestControllerTests.java | Main | 39 |
| > PetType.java | Type Argument [In ... | SpringDataPetRepository.java | Main | 1 |
| > PetType.java | Type Argument [In ... | SpringDataPetTypeRepository.ja... | Main | 1 |
| > PetType.java | Aggregated | SpringDataPetTypeRepositoryIm... | Main | 2 |
| > PetType.java | Aggregated | VisitRestControllerTests.java | Main | 5 |
| Internal - From [0 elements] | Dependency Type | To | Number of depend.. | |
| Outgoing - From [3 elements] | Dependency Type | To | To scope | Number of depend.. |
| > PetType.java | Has Annotation [On... | Entity | External [Java] | 1 |
| > PetType.java | Has Annotation [On... | Table | External [Java] | 1 |
| > PetType.java | Extends | NamedEntity.java | Main | 1 |

Figure 3.6: Dependencies PetType

Efferent Coupling (E) and Afferent Coupling(I)

As seen in the figure 3.6 the module PetType has a total of 3 outgoing dependencies. On the other hand we have a significant amount of Incoming dependencies with 23 elements.

Using the formula that we expesified earlier we get

$$\text{Instability} = 3 / (3 + 23) = 3 / 26 = 0.11$$

An instability value of 0.11 indicates that the class is relatively stable, with few dependencies on other classes and minimal impact when changes occur in external components. This suggests that the class is well-contained and less prone to breakage during maintenance, which is generally a positive characteristic for maintainability.

3.5 Maintainability of test code

Now I will explore and examine testing smells in this project. Test smells refer to possible violations of best practices in testing code, that can compromise the reliability and effectiveness of tests. Eventhough they are sometimes ignored, testing smells can lead to false positives or negatives and make tests harder to maintain or comprehend

| App | TestFileName | LOC | numberMi | Assertion | Eager Test | Mystery Gi | Sleepy Test | Unknown | Redundant | Dependen | Magic Nui | Condition | Empty Test | General Fi | IgnoredTe | Sensitive T | Verbose T | Default Te | Resource | Duplicate | Exception | Construct | Print State | Lazy Test |
|-----------|----------------------------|-----|----------|-----------|------------|------------|-------------|---------|-----------|----------|-----------|-----------|------------|------------|-----------|-------------|-----------|------------|----------|-----------|-----------|-----------|-------------|-----------|
| qsotfaste | PetTypeRestControllerTests | 253 | 18 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 3.7: TestSmells PetType

As shown in Figure 3.7, we analyzed 18 test methods. Out of these, 12 were initially labeled as "Unknown tests" and later classified as "IgnoredTests." This means that only six tests passed the jNose test smells criteria, indicating a deficiency in the project's testing quality that should be addressed and improved.

4 Performance

In software development, the performance of an application is crucial for its success and integration within a larger system. The efficiency and responsiveness of an application greatly influence the user experience and overall user satisfaction. This evaluation focuses on analyzing the performance aspects of the current project. Performance analysis is essential as it reveals the application's ability to handle different workloads, respond quickly to user interactions, and utilize resources optimally.

To conduct a thorough performance assessment, I will use JMeter and K6, both powerful testing tools renowned for their ability to evaluate the performance of software applications. JMeter will simulate various scenarios that mimic real-world usage conditions, while K6 will be used to perform the same exact performance tests, collecting valuable data to assess the project's performance characteristics.

In the following sections, we will explore specific performance metrics, analyze the results obtained from JMeter and K6 tests, and draw conclusions about the project's suitability for integration into a larger system based on its performance attributes.

4.1 Requirements

In order to have a standard, we consider that, for normal conditions, the application must:

In normal conditions:

- Respond to any type of request in less than 3 seconds 99% of the time;
- Process more than 250 requests per second.

In heavy conditions:

- Supports heavy user access of up to 200 users;
- Process more than 150 requests per second.

4.2 Testing Environment

All tests conducted for this evaluation were performed on a single machine/computer, with the specifications outlined in the image below (4.1). It's important to note that the computer remained consistently connected to the internet via an Ethernet cable throughout the testing process. This ensures a consistent and standardized testing environment, allowing for accurate and reliable performance assessments under the specified configurations. The use of a wired connection guarantees stable network conditions, contributing to the precision of the results obtained during the evaluation.



Figure 4.1: System Details

4.3 Load Test

Load testing evaluates a system's behavior under typical conditions by simulating realistic user activity. This process enables the early identification of performance issues, such as response time delays and resource overuse, even at normal load levels. Additionally, it provides insights into the system's capacity, supporting scalability planning to ensure it can handle anticipated user growth effectively. In this section we will evaluate the Loads tests to the endpoints GET and POST of petType

4.3.1 Scenario GET

- 50 users of the web application;
- Requesting all existing PetTypes
- The system returns a list with all PetTypes
- Less than 3 seconds 99% of the times;
- Normal Load.

4.3.2 Jmeter

Test Configuration

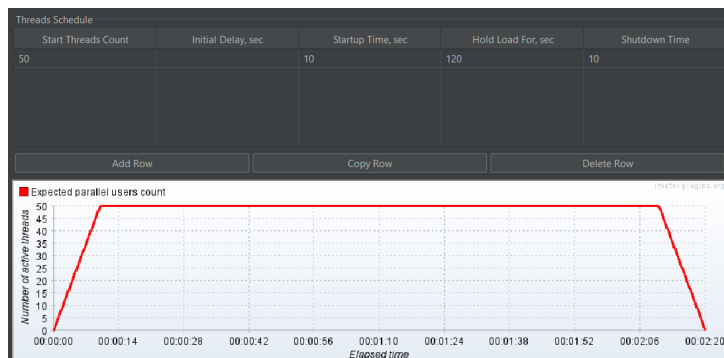


Figure 4.2: Jmeter Load Configuration

The screenshot shows the 'Duration Assertion' configuration in Jmeter. The fields are as follows:

- Name: Duration Assertion
- Comments:
- Apply to: ☐ Main sample and sub-samples ☒ Main sample only ☐ Sub-samples only
- Duration to Assert: 3000

Figure 4.3: Duration GET Load Configuration

Based on the scenario we designed earlier I created the Jmeter Load Configuration that holds for 120 seconds 50 Threads (Users), as seen in the Figure 4.2. In the Figure 4.3 I've set the maximum duration to 3000 milliseconds.

Results

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received KB/sec | Sent KB/sec |
|---------------------|-----------|---------|--------|----------|----------|----------|-----|---------|---------|------------|-----------------|-------------|
| LoadTestGetPetTypes | 9600 | 676 | 677 | 959 | 1059 | 1230 | 27 | 1622 | 0.00% | 68.0/sec | 38.63 | 71.45 |

Figure 4.4: GET Load Test Aggregate Report

4 Performance

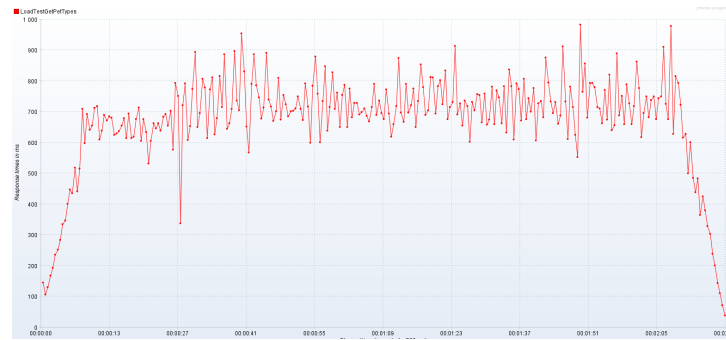


Figure 4.5: GET Load Test Responses Over Time

The results, seen in Figure 4.4, indicate that the API performs incredibly well under this scenario, with 0% errors and an average response time of 676 ms. The highest response time was just 1622 ms, which equates to half of the maximum defined. In Figure 4.5, we can see the response times over time, which remained stable throughout the entire process

4.3.3 K6

Test Configuration

```
// Test configuration
export let options = {stages: [(duration: string, target: number, thresholds: (...)) = { no usages
  stages: [
    { duration: '10s', target: 50 }, // Ramp-up to 50 users over 10 seconds
    { duration: '120s', target: 50 }, // Sustain 50 users for 120 seconds
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(99)<3000'], // 99% of response times should be below 3 seconds
  },
},
];
```

Figure 4.6: K6 Load Configuration

Similar to the JMeter configuration, in the k6 test, I configured the test to hold 50 users for 120 seconds, as shown in Figure 4.6. I've also set the Duration to be less than 3000 milliseconds.

Results

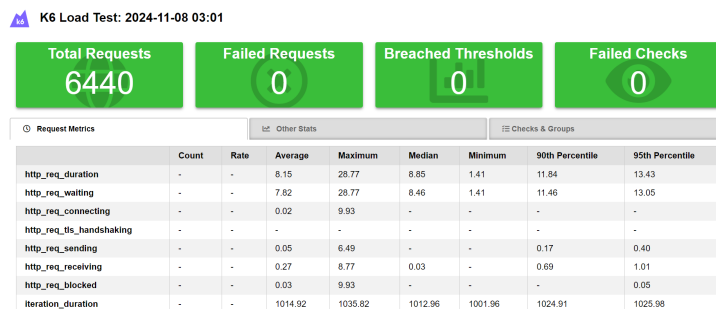


Figure 4.7: K6 GET LOAD

The results, seen in Figure 4.7, indicate that the API performs incredibly well under this scenario, with 0 failed request and an average response time of just 1014 ms. The highest response time was just 1035 ms, which is lower than the maximum defined.

4.3.4 Scenario POST

- 50 users of the web application;
- Posting a new PetTypes
- The system returns 201
- Less than 3 seconds 99% of the times;
- Normal Load.

4.3.5 Jmeter

Load CSV

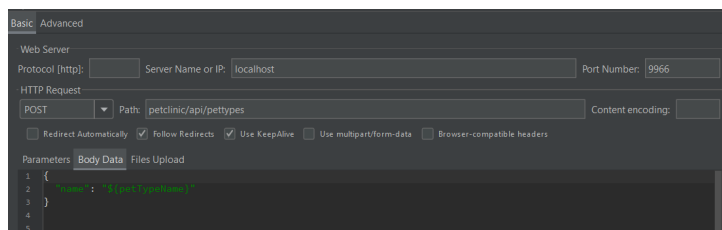
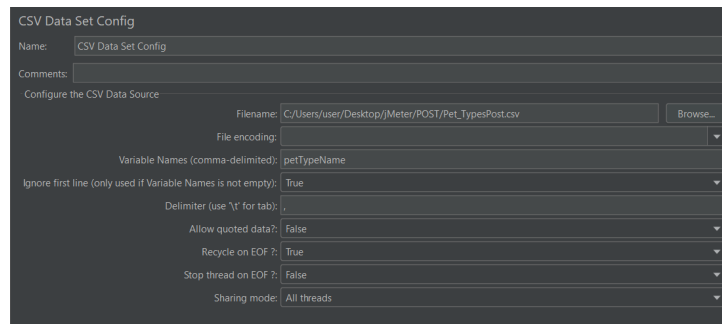


Figure 4.8: LoadCSV Jmeter

4 Performance



CSV Data Set Config

Name: CSV Data Set Config

Comments:

Configure the CSV Data Source

Filename: C:/Users/user/Desktop/JMeter/POST/Pet_TypesPost.csv

File encoding: UTF-8

Variable Names (comma-delimited): petTypeName

Ignore first line (only used if Variable Names is not empty): True

Delimiter (use '\t' for tab): ,

Allow quoted data?: False

Recycle on EOF?: True

Stop thread on EOF?: False

Sharing mode: All threads

Figure 4.9: LoadCSV Jmeter2

In the figure 4.9 we can see the configuration needed in jmeter in order to use an CSV Data Set for the tests. In the figure 4.8 we can see the body Data in order to use the csv data.

Test Configuration

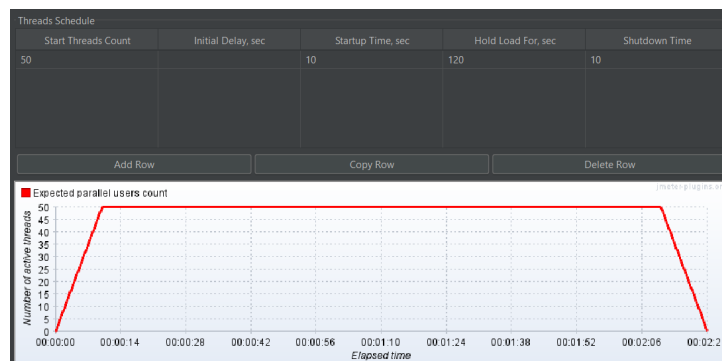
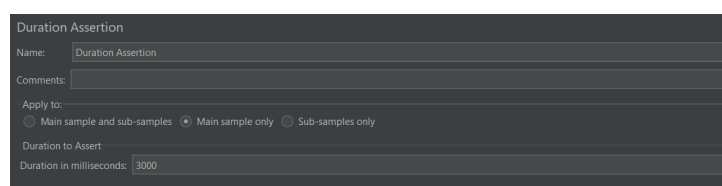


Figure 4.10: Jmeter Load Configuration



Duration Assertion

Name: Duration Assertion

Comments:

Apply to:

☐ Main sample and sub-samples ☒ Main sample only ☐ Sub-samples only

Duration to Assert:

Duration in milliseconds: 3000

Figure 4.11: Duration POST Load Configuration

Based on the scenario we designed earlier I created the Jmeter Load Configuration that holds for 120 seconds 50 Threads (Users), as seen in the Figure 4.10. In the Figure 4.11 I've set the maximum duration to 3000 milliseconds.

Results

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received KB/sec | Sent KB/sec |
|----------------------|-----------|---------|--------|----------|----------|----------|-----|---------|---------|------------|-----------------|-------------|
| LoadTestPostPetTy... | 19402 | 668 | 675 | 948 | 1038 | 1172 | 23 | 1714 | 0.00% | 138.5/sec | 66.42 | 29.66 |

Figure 4.12: POST Load Test Aggregate Report

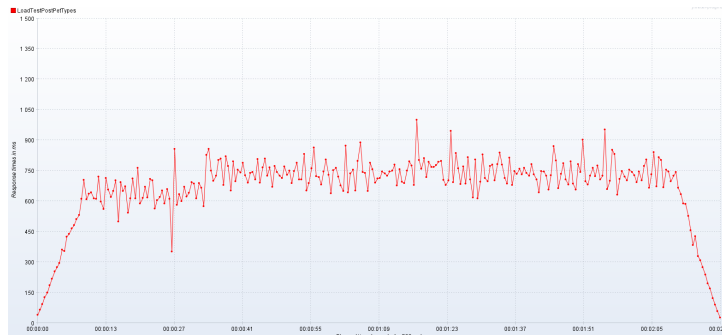


Figure 4.13: GET Post Test Responses Over Time

The results, seen in Figure 4.12, indicate that the API performs incredibly well under this scenario, with 0% errors and an average response time of 668 ms. The highest response time was just 1714 ms, which is a little bit over the half of the maximum defined. In Figure 4.13, we can see the response times over time, which remained stable throughout the entire process

4.3.6 K6

Load CSV

```
// Load data from CSV
const csvData = new SharedArray("Pet Types", function() {
  return papaparse.parse(open('C:/Users/user/Desktop/Tests/k6/POST/Pet_Types.csv'), { header: true }).data;
});
```

Figure 4.14: LoadCSV K6

In the figure 4.14 we can see the configuration needed in K6 in order to use an CSV Data Set for the tests.

Test Configuration

4 Performance

```
// Test configuration
export let options = {stages: [{duration: string, target: number, thresholds: {}}] = { no usages
  stages: [
    { duration: '10s', target: 50 }, // Ramp-up to 50 users over 10 seconds
    { duration: '120s', target: 50 }, // Sustain 50 users for 120 seconds
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(99)<3000'], // 99% of response times should be below 3 seconds
  },
}
```

Figure 4.15: K6 Load Configuration

Similar to the JMeter configuration, in the k6 test, I configured the test to hold 50 users for 120 seconds, as shown in Figure 4.15. I've also set the Duration to be less than 3000 milliseconds.

Results

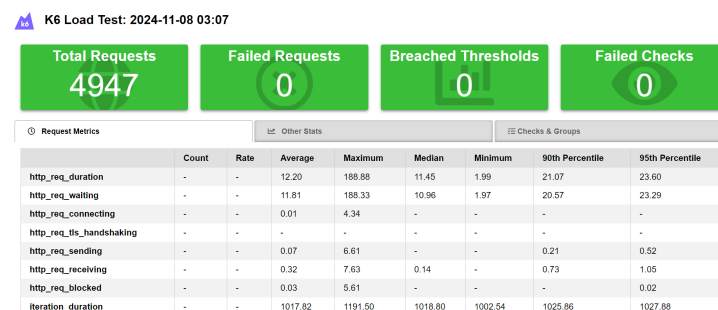


Figure 4.16: K6 POST LOAD

The results, seen in Figure 4.16, indicate that the API performs incredibly well under this scenario, with 0 failed request and an average response time of just 1017 ms. The highest response time was just 1191 ms, which is lower than the maximum defined.

4.4 Stress Test

Stress testing involves subjecting the system to conditions that exceed its standard operational limits to assess its behavior under severe strain. This type of testing facilitates crucial insights, including resilience evaluation, which examines the system's capacity to recover from high-stress situations and whether it maintains functionality or fails catastrophically. Furthermore, it assists in identifying the failure threshold, providing an understanding of the system's limits and its response under extreme conditions.

4.4.1 Scenario GET

- 200 users of the web application;
- Requesting all existing PetTypes
- The system returns a list with all PetTypes
- Less than 3 seconds 99% of the times;
- Heavy Load.

4.4.2 Jmeter

Test Configuration

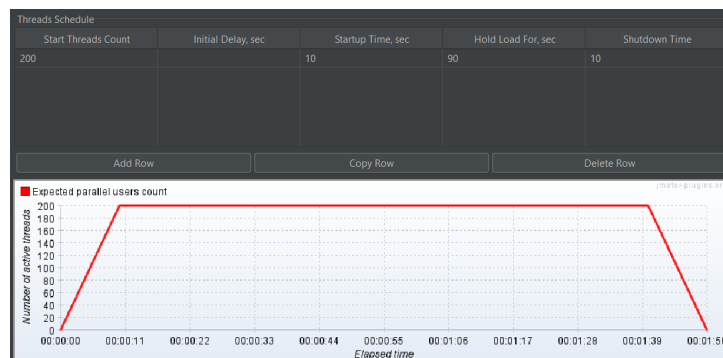


Figure 4.17: Jmeter Stress Configuration

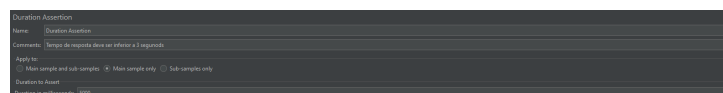


Figure 4.18: Duration GET Stress Configuration

Based on the scenario we designed earlier I created the Jmeter Stress Configuration that holds for 90 seconds 200 Threads (Users), as seen in the Figure 4.17. In the Figure 4.18 I've set the maximum duration to 5000 milliseconds.

Results

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received KB/sec | Sent KB/sec |
|--------------------|-----------|---------|--------|----------|----------|----------|-----|---------|---------|------------|-----------------|-------------|
| StressTestGetPetTy | 8193 | 2440 | 2335 | 3580 | 3890 | 4500 | 24 | 3124 | 0.12% | 74.6/sec | 41.90 | 12.46 |

Figure 4.19: GET Stress Test Aggregate Report

4 Performance

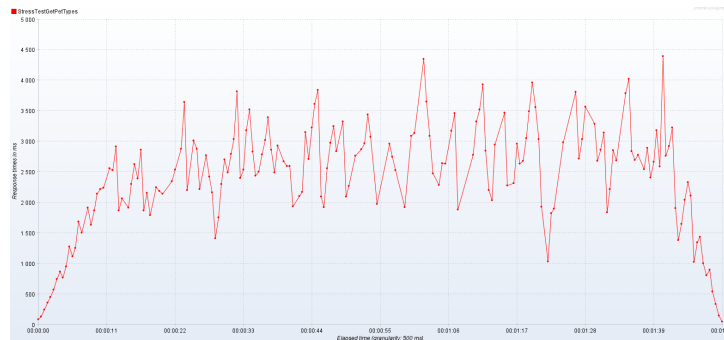


Figure 4.20: GET Stress Test Responses Over Time

The results, seen in Figure 4.19, indicate that the API performs well under this scenario, with just 0.12% errors and an average response time of 2448 ms. The highest response time was just 5124 ms, which is a little over the maximum defined, and therefore was counted as an error. In Figure 4.20, we can see the response times over time, which, unlike the LoadTest, did not remain stable throughout the entire process

4.4.3 K6

Test Configuration

```
// Test configuration
export let options = {stages: [{duration: string, target: number, thresholds: (...)}] = { no usages
  stages: [
    { duration: '10s', target: 200 }, // Ramp-up to 200 users over 10 seconds
    { duration: '90s', target: 50 }, // Sustain 200 users for 90 seconds
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(95)<5000'], // 95% of response times should be below 5 seconds
  },
};
```

Figure 4.21: K6 Stress Configuration

Similar to the JMeter configuration, in the k6 test, I configured the test to hold 200 users for 90 seconds, as shown in Figure 4.21. I've also set the Duration to be less than 5000 milliseconds.

Results

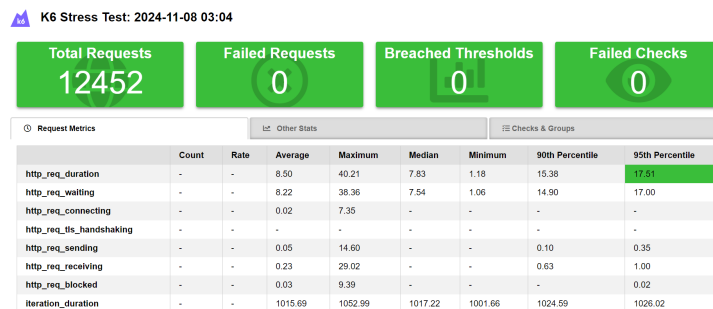


Figure 4.22: K6 GET STRESS

The results, seen in Figure 4.22, indicate that the API performs incredibly well under this scenario, with 0 failed request and an average response time of just 1015 ms. The highest response time was just 1052 ms, which is lower than the maximum defined.

4.4.4 Scenario POST

- 200 users of the web application;
- Posting a new PetTypes
- The system returns 201
- Less than 5 seconds 95% of the times;
- Heavy Load.

4.4.5 Jmeter

Test Configuration

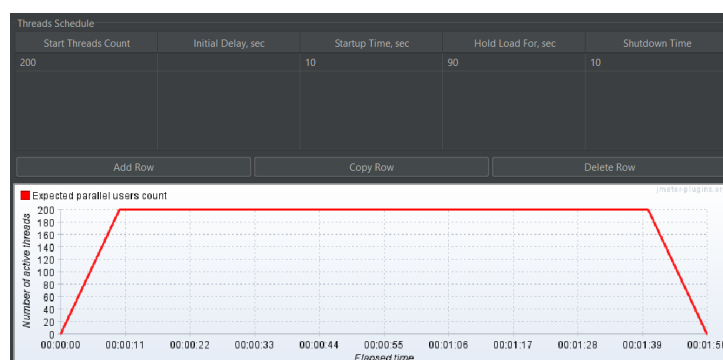


Figure 4.23: Jmeter Stress Configuration

4 Performance

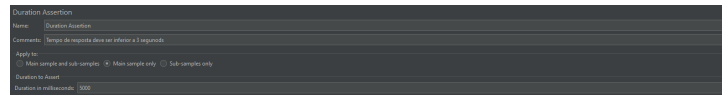


Figure 4.24: Duration POST Stress Configuration

Based on the scenario we designed earlier I created the Jmeter Stress Configuration that holds for 90 seconds 200 Threads (Users), as seen in the Figure 4.23. In the Figure 4.24 I've set the maximum duration to 5000 milliseconds.

Results

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received KB/sec | Sent KB/sec |
|------------------------|-----------|---------|--------|----------|----------|----------|-----|---------|---------|------------|-----------------|-------------|
| StressTestPostPetTy... | 19829 | 2634 | 2736 | 3846 | 4160 | 4760 | 28 | 5864 | 33.81% | 141.8/sec | 67.99 | 35.13 |

Figure 4.25: POST Stress Test Aggregate Report

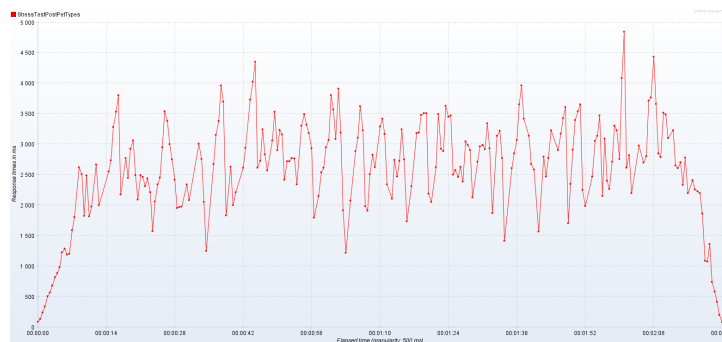


Figure 4.26: POST Stress Test Responses Over Time

The results, seen in Figure 4.25, indicate that the API performs below the expected under this scenario, with 33.81% errors and an average response time of 2634 ms. The highest response time was just 5864 ms, which is almost over a second higher than the maximum defined, and therefore was counted as an error. In Figure 4.26, we can see the response times over time, which, unlike the LoadTest, did not remain stable throughout the entire process

4.4.6 K6

Test Configuration

```
// Test configuration
export let options: {stages: [{duration: string, target: number, thresholds: (...)}] = { no usages
  stages: [
    { duration: '10s', target: 200 }, // Ramp-up to 200 users over 10 seconds
    { duration: '90s', target: 50 }, // Sustain 200 users for 90 seconds
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(95)<5000'], // 95% of response times should be below 5 seconds
  },
};
```

Figure 4.27: K6 Stress Configuration

Similar to the JMeter configuration, in the k6 test, I configured the test to hold 200 users for 90 seconds, as shown in Figure 4.27. I've also set the Duration to be less than 5000 milliseconds.

Results

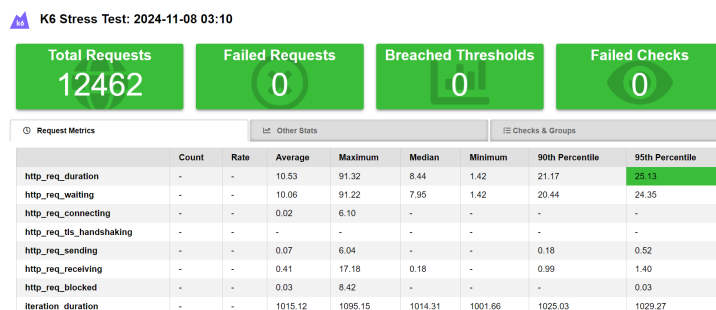


Figure 4.28: K6 POST STRESS

The results, seen in Figure 4.28, indicate that the API performs incredibly well under this scenario, with 0 failed request and an average response time of just 1015 ms. The highest response time was just 1095 ms, which is lower than the maximum defined.

4.5 Soak Test

Soak testing involves running a system at normal load levels for an extended period to identify performance issues that may not be apparent during shorter testing phases. This type of testing helps uncover problems such as memory leaks, resource depletion, and degradation in response times over time. By maintaining a steady load, soak tests provide insights into the system's long-term reliability and stability, ensuring it can handle sustained usage without performance deterioration. In this section, we will evaluate the soak tests for the endpoints GET and POST of petType.

4 Performance

4.5.1 Scenario GET

- 50 users of the web application;
- Sending the petType information to the application;
- The application gives a success or error response to the user;
- Less than 5 seconds 95% of the times;
- Normal Load.

4.5.2 Jmeter

Test Configuration

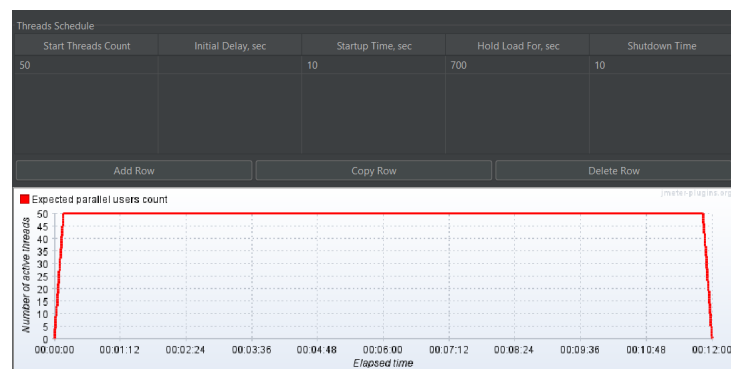


Figure 4.29: Jmeter SOAK Configuration

The screenshot shows the 'Duration Assertion' configuration in Jmeter. The 'Name' field is set to 'Duration Assertion'. The 'Apply to' section has 'Main sample only' selected. The 'Duration to Assert' field is set to '3000' milliseconds.

Figure 4.30: Duration GET Soak Configuration

Based on the scenario we designed earlier I created the Jmeter Soak Configuration that holds for 700 seconds 50 Threads (Users), as seen in the Figure 4.29. In the Figure 4.30 I've set the maximum duration to 3000 milliseconds.

Results

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received KB/sec | Sent KB/sec |
|---------------------|-----------|---------|--------|----------|----------|----------|-----|---------|---------|------------|-----------------|-------------|
| SoakTestGetPetTypes | 50877 | 677 | 659 | 941 | 1014 | 1156 | 21 | 1788 | 0.00% | 70.7/sec | 39.69 | 11.85 |

Figure 4.31: GET Soak Test Aggregate Report

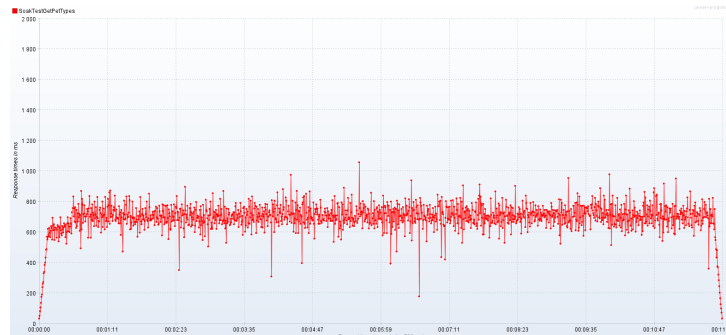


Figure 4.32: GET Soak Test Responses Over Time

The results, seen in Figure 4.31, indicate that the API performs incredibly well under this scenario, with 0% errors and an average response time of 679 ms. The highest response time was just 1788 ms, which equates a little over half of the maximum defined. In Figure 4.32, we can see the response times over time, which remained stable throughout the entire process.

4.5.3 K6

Test Configuration

```
export let options = {
  stages: [
    { duration: '10s', target: 50 }, // Ramp-up to 50 users over 10 seconds
    { duration: '700s', target: 50 }, // Sustain 50 users for 700 seconds
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(99)<3000'], // 99% of response times should be below 3 seconds
  },
};
```

Figure 4.33: K6 Soak Configuration

Similar to the JMeter configuration, in the k6 test, I configured the test to hold 50 users for 700 seconds, as shown in Figure 4.33. I've also set the Duration to be less than 3000 milliseconds.

Results

4 Performance

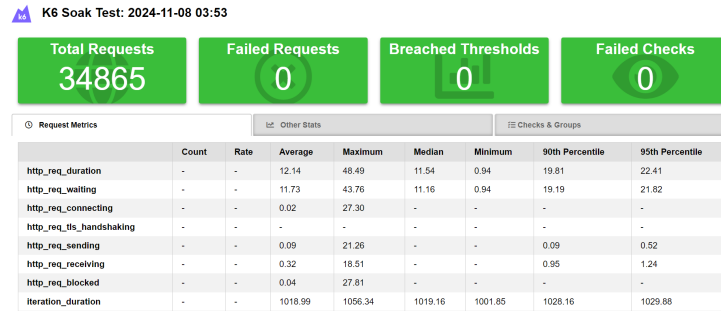


Figure 4.34: K6 GET SOAK

The results, seen in Figure 4.34, indicate that the API performs incredibly well under this scenario, with 0 failed request and an average response time of just 1018 ms. The highest response time was just 1056 ms, which is lower than the maximum defined.

4.5.4 Scenario POST

- 200 users of the web application;
- Posting a new PetTypes
- The system returns 201
- Less than 5 seconds 95% of the times;
- Heavy Load.

4.5.5 Jmeter

Test Configuration

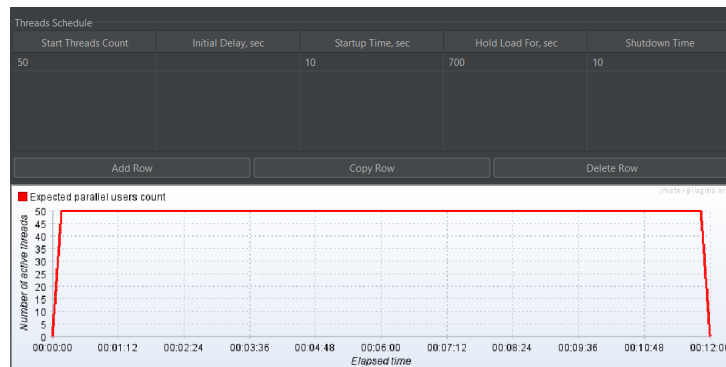


Figure 4.35: Jmeter Soak Configuration

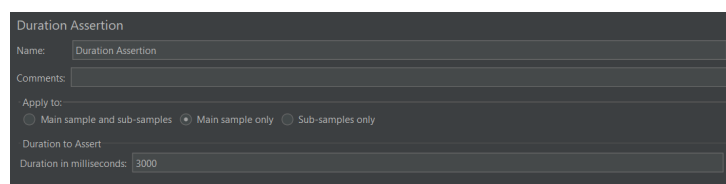


Figure 4.36: Duration POST Soak Configuration

Based on the scenario we designed earlier I created the Jmeter Soak Configuration that holds for 700 seconds 50 Threads (Users), as seen in the Figure 4.35. In the Figure 4.36 I've set the maximum duration to 3000 milliseconds.

Results

| Label | # Samples | Average | Median | 90% Line | 95% Line | 99% Line | Min | Maximum | Error % | Throughput | Received KB/sec | Sent KB/sec |
|----------------------|-----------|---------|--------|----------|----------|----------|-----|---------|---------|------------|-----------------|-------------|
| SoakTestPostPetTy... | 48200 | 735 | 733 | 883 | 1060 | 1212 | 22 | 2200 | 0.00% | 67.0/sec | 32.08 | 14.26 |

Figure 4.37: POST Soak Test Aggregate Report

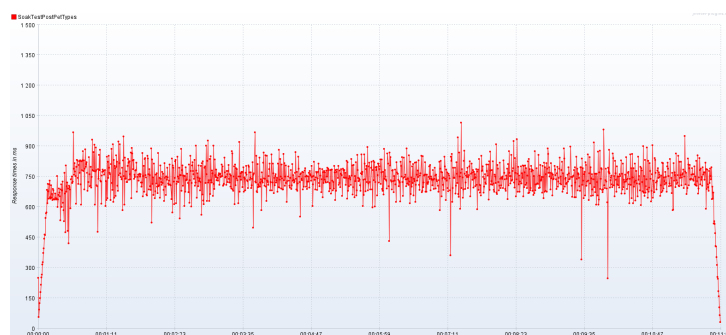


Figure 4.38: POST Soak Test Responses Over Time

4 Performance

The results, seen in Figure 4.37, indicate that the API performs incredibly well under this scenario, with 0% errors and an average response time of 735 ms. The highest response time was just 2200 ms, which is lower than the maximum defined. In Figure 4.38, we can see the response times over time, which remained stable throughout the entire process.

4.5.6 K6

Test Configuration

```
export let options = {stages: [{duration: string, target: nu..., thresholds: {...}] = { no usages
  stages: [
    { duration: '10s', target: 50 }, // Ramp-up to 50 users over 10 seconds
    { duration: '700s', target: 50 }, // Sustain 50 users for 700 seconds
    { duration: '10s', target: 0 },
  ],
  thresholds: {
    'http_req_duration': ['p(99)<3000'], // 99% of response times should be below 3 seconds
  },
};
```

Figure 4.39: K6 Soak Configuration

Similar to the JMeter configuration, in the k6 test, I configured the test to hold 50 users for 700 seconds, as shown in Figure 4.39. I've also set the Duration to be less than 3000 milliseconds.

Results

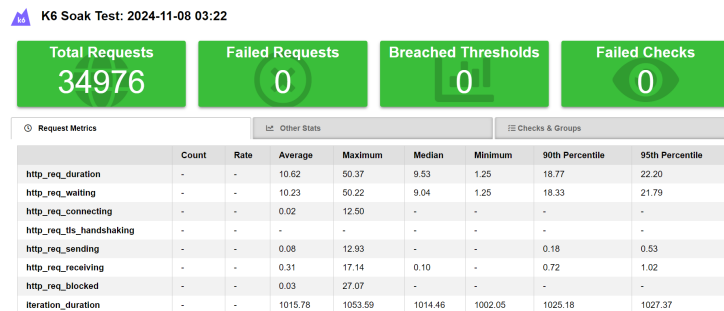


Figure 4.40: K6 POST SOAK

The results, seen in Figure 4.40, indicate that the API performs incredibly well under this scenario, with 0 failed request and an average response time of just 1015 ms. The highest response time was just 1053 ms, which is lower than the maximum defined.

4.6 Conclusion

In conclusion, the application's performance was extremely good across most scenarios, with stable response times and minimal errors observed. The API consistently demonstrated low average response times and stayed well within the maximum limits in both load and soak tests. However, under the POST stress test, performance declined, showing a 33.81% error rate and response times that exceeded the expected maximum threshold. This indicates that while the application is robust in general load conditions, improvements may be necessary to handle high-stress scenarios in POST requests.

5 Security

In today's digital landscape, security has emerged as a paramount concern for both individuals and organizations, particularly as applications become more sophisticated and deeply embedded within critical infrastructure. The growing complexity of software systems increases the risk of security breaches, particularly through vulnerabilities in third-party dependencies that attackers may exploit to compromise data, disrupt operations, or gain unauthorized access.

5.1 Dependency Checker

Based on the report generated by the OWASP Dependency Check plugin, the following dependencies were detected:

- jackson-databind-2.15.3.jar
- mysql-connector-j-8.1.0.jar
- postgresql-42.6.0.jar
- spring-core-6.1.2.jar
- spring-security-core-6.2.1.jar
- spring-security-web-6.2.1.jar
- spring-webmvc-6.1.2.jar

Now, I will focus my analysis on the spring-core-6.1.2.jar dependency.

5.2 Spring Core Dependency

The spring-core-6.1.2 version had 2 Common Vulnerabilities and Exposure (CVE). The highest severity of those CVE is HIGH, which means that the problem

5.2.1 Identified Vulnerabilities

CVE-2024-22233

- **Description:** This vulnerability allows a user to send specially crafted HTTP requests that may cause a denial-of-service (DoS) condition.
- **Score:** 7.5 **High** on the CVSSv3 scale
- **Solution:** The recommended solution is to update spring-core to version 6.1.14, where this vulnerability is patched, and configure the application to mitigate potential DoS issues.

CVE-2024-38820

- **Description:** The fix for CVE-2022-22968 made disallowedFields patterns in DataBinder case insensitive. However, String.toLowerCase() has some Locale dependent exceptions that could potentially result in fields not protected as expected.
- **Impact Score:** 5.3 **Medium** on the CVSSv3 scale
- **Solution:** An update to a version beyond 6.1.14 addresses this issue. Additionally, manual reviews of binding configurations can ensure safe data handling.

5.2.2 CVSS3 Scale

In order to better understand the Impact Scores I decided to show how the labels are attributed

- **Critical:** 9.0 - 10
- **High:** 7.0 - 8.9
- **Medium:** 4.0 - 6.9
- **Low:** 0.1 - 3.9

6 Architectural compliance

This section focuses on assessing the architectural compliance of the application, ensuring it aligns with the defined guidelines. Using ArchUnit, we create fitness functions to detect architectural issues like package dependencies, class dependencies, inheritance misuse, and layer violations. We will specifically examine the petType Aggregate to verify adherence to these standards. The goal is to identify and correct any deviations, ensuring the application remains scalable, maintainable, and efficient.

6.1 Package Dependency

6.1.1 Code

```
//Package Dependency
@Test
public void domainPackageShouldNotDependOnAnyPackage() {
    noClasses()
        .that().haveNameMatching(MAINPACKAGE + ".model.PetType")
        .should().dependOnClassesThat()
        .resideInAnyPackage(
            MAINPACKAGE + ".controller..",
            MAINPACKAGE + ".repository..",
            MAINPACKAGE + ".dto..")
        .check(importedClasses);
}
```

Figure 6.1: Package Dependency Code

This test ensures that the PetType class in the model package does not depend on classes from the controller, repository, or dto packages.

6 Architectural compliance

6.1.2 Result

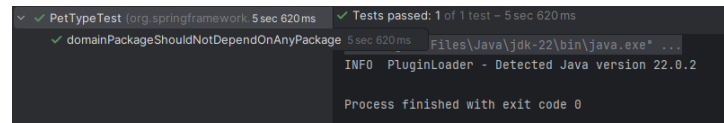


Figure 6.2: Package Dependency Test Result

As shown in the Figure 6.2, the test was successful.

6.2 Class Dependency

6.2.1 Code

```
//Class Dependency
@Test
public void petTypeClassesShouldOnlyHaveDependenciesOnItsClasses() {

    classes()
        .that().haveNameMatching(".*PetType*")
        .should().onlyHaveDependentClassesThat()
        .haveSimpleNameContaining("PetType")
        .check(importedClasses);
}
```

Figure 6.3: Class Dependency Code

This test ensures that classes with "PetType" in their name can only have dependencies on other classes with "PetType" in their name.

6.2.2 Result

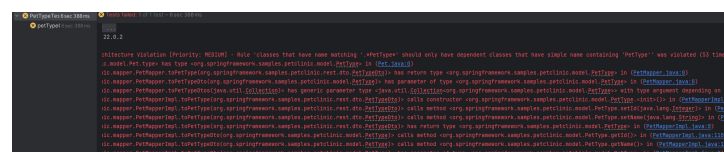


Figure 6.4: Class Dependency Test Result

As shown in the Figure 6.4, the test has failed. It was violated 53 times.

6.3 Class and Package Containment

6.3.1 Code

```
// Class and Package Containment
@Test
public void classAndPackageContainment() {
    classes()
        .that()
        .haveSimpleName("PetTypeRestController")
        .should()
        .resideInAnyPackage(MAINPACKAGE + ".rest.controller..")
        .check(importedClasses);

    classes()
        .that()
        .haveSimpleName("PetTypeRepository")
        .should()
        .resideInAnyPackage(MAINPACKAGE + ".repository..")
        .check(importedClasses);

    classes()
        .that()
        .haveSimpleName("JpaPetTypeRepositoryImpl")
        .should()
        .resideInAnyPackage(MAINPACKAGE + ".repository.jpa..")
        .check(importedClasses);

    classes()
        .that()
        .haveSimpleName("PetTypeDto")
        .should()
        .resideInAnyPackage(MAINPACKAGE + ".rest.dto..")
        .check(importedClasses);
}
```

Figure 6.5: Class and Package Containment Code

This test checks if specific classes like `PetTypeRestController`, `PetTypeRepository`, and others reside in their correct packages (`rest.controller`, `repository`, `repository.jpa`, `rest.dto`).

6.3.2 Result

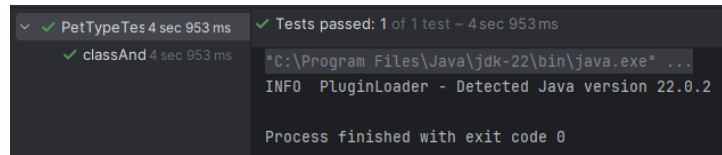


Figure 6.6: Class and Package Containment Test Result

As shown in the Figure 6.6, the test was successful.

6.4 Inheritance

6.4.1 Code

```
// Inheritance
@Test
public void checkPetTypeRepositoryClassInheritance() {

    classes()
        .that().resideInAnyPackage(MAINPACKAGE + ".repository.jpap.")
        .and().haveSimpleNameContaining("JpaPetTypeRepository")
        .should().beAssignableTo(PetTypeRepository.class)
        .check(importedClasses);
}

@Test
public void petTypeClassesShouldInheritFromBaseClass() {
    classes()
        .that().haveSimpleName("PetType")
        .should().beAssignableTo(PetType.class).check(importedClasses);
}
```

Figure 6.7: Inheritance Code

The first test ensures that JpaPetTypeRepository implements PetTypeRepository. The second test ensures that PetType inherits from itself.

6.4.2 Result

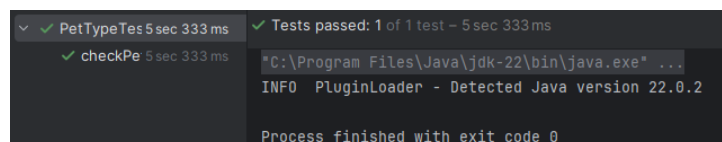


Figure 6.8: Inheritance Test Result 1

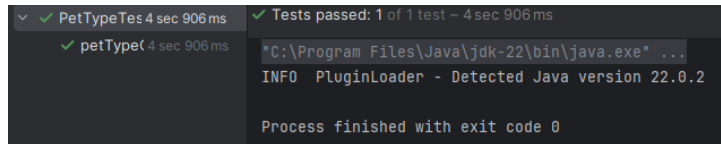


Figure 6.9: Inheritance Test Result 1

As shown in the Figure 6.8 and Figure ??, both tests were successful.

6.5 Annotation

6.5.1 Code

```
// Annotation
@Test
public void checkPetTypeRestControllerAnnotation() {
    classes()
        .that().haveSimpleName("PetTypeRestController")
        .should().beAnnotatedWith(RestController.class)
        .check(importedClasses);
}
```

Figure 6.10: Annotation Code

This test ensures that the PetTypeRestController class is annotated with RestController.

6.5.2 Result

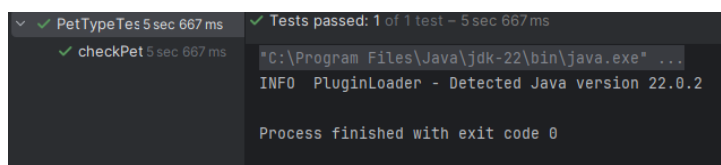


Figure 6.11: Annotation Test Result

As shown in the Figure 6.11, the test was successful.

6.6 Layer

6.6.1 Code

```
// Annotation
@Test
public void checkPetTypeRestControllerAnnotation() {
    classes()
        .that().haveSimpleName("PetTypeRestController")
        .should().beAnnotatedWith(RestController.class)
        .check(importedClasses);
}
```

Figure 6.12: Layer Code

This test verifies that the architecture follows specific rules, where layers like "Controller" can only access "DTO", "Repository", and "Model", but not "Model" can access any layer, ensuring clean separation.

6.6.2 Result

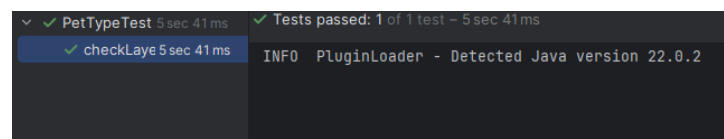


Figure 6.13: Layer Test Result

As shown in the Figure 6.13, the test was successful.

6.7 Cycle

6.7.1 Code

```
// Cycle
@Test
public void checkCycles() {
    slices().matching(MAINPACKAGE + ".*(..)")
        .should().beFreeOfCycles()
        .check(importedClasses);
}
```

Figure 6.14: Cycle

This test ensures that there are no cyclic dependencies between the classes in the project.

6.7.2 Result

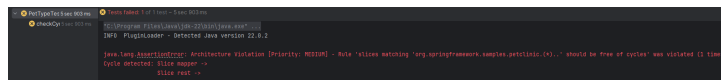


Figure 6.15: Cycle Test Result

As shown in the Figure 6.15, the test was successful.

6.8 Conclusion

In conclusion, the architectural compliance analysis confirms that the application largely adheres to the defined standards. Most tests on package dependencies, class containment, inheritance, annotations, and layer separation were successful, ensuring a well-structured and modular design. However, the Class Dependency test revealed some violations, suggesting areas for improvement in maintaining clear dependency boundaries. Addressing these issues will further enhance the system's scalability and maintainability.

7 Conclusions

This project was focused on evaluating key aspects of the application, mostly the Pet-Type aggregate. By addressing topics like Funcional Correctness, Maintainability, Performance, Security and Architectural Compliance, and based on the results, I can confidently conclude that the aggregate performed to the expected levels in nearly every aspect tested, indicating a well-functioning and robust design.

The development of this project allowed me to use and test different tools, which led to multiple challenges but undoubtedly provided a good opportunity to learn and improve my skills in software analysis and testing.

References