TESTING REPORT

Imagen en blanco y negro

Descripción generada automáticamente con confianza media

DELIVERABLE 4

DESING AND TESTING 2

2023-2024

LUIS JAVIER PERIÁÑEZ FRANCO

|  |  |
| --- | --- |
| Date | Version |
| 07/07/2024 | V1.0 |

|  |  |
| --- | --- |
| Group: 21 | |
| Members | Email |
| Fernández Rodríguez, Jesús | [jesferrod1@alum.us.es](mailto:jesferrod1@alum.us.es) |
| García Rodríguez, Javier | [javgarrod5@alum.us.es](mailto:javgarrod5@alum.us.es) |
| González Ortiz, Miguel | [miggonort1@alum.us.es](mailto:miggonort1@alum.us.es) |
| Palomo García, Miguel | [migpalgar1@alum.us.es](mailto:migpalgar1@alum.us.es) |
| Periáñez Franco, Luis Javier | [luiperfra1@alum.us.es](mailto:luiperfra1@alum.us.es) |

GitHub repository: <https://github.com/JesusFern/Acme-SF-D04>

TABLE OF CONTENTS

[Executive Summary 3](#_Toc167735698)

[Revision Table 3](#_Toc167735699)

[Introduction 4](#_Toc167735700)

[Contents 4](#_Toc167735701)

[ Functional Test 4](#_Toc167735702)

[**Contract** 4](#_Toc167735703)

[List-Mine 4](#_Toc167735704)

[List-All 4](#_Toc167735705)

[Show 5](#_Toc167735706)

[Create 5](#_Toc167735707)

[Delete 5](#_Toc167735708)

[Update 5](#_Toc167735709)

[Publish 5](#_Toc167735710)

[**ProgressLog** 6](#_Toc167735711)

[ListMine 6](#_Toc167735712)

[Show 6](#_Toc167735713)

[Create 6](#_Toc167735714)

[Delete 6](#_Toc167735715)

[Update 7](#_Toc167735716)

[ Performance Test 7](#_Toc167735717)

[**Before** 7](#_Toc167735718)

[**After** 8](#_Toc167735719)

[**Before vs After** 8](#_Toc167735720)

[Conclusion 9](#_Toc167735721)

[Bibliography 9](#_Toc167735722)

# Executive Summary

This report provides detailed information obtained through the execution of functional and performance tests for deliverable D04 of the project. In this way, we can thoroughly understand the methodology to follow for conducting these tests and the conclusions we can draw from them.

# Revision Table

|  |  |  |
| --- | --- | --- |
| Revision Number | Date | Description |
| V1r0 | 07/07/2024 | Final revision |
|  |  |  |

# Introduction

The report consists of two main sections. The first section focuses on functional tests, verifying that the system's functionalities meet the specified requirements. The second section focuses on performance tests, ensuring that the system operates within the established performance parameters.

# Contents

## Functional Test

The development of these tests has been carried out following the methodology proposed in the course slides. The highest possible coverage has been achieved, discarding those cases where our natural intelligence indicated that it did not make sense to attempt to cover certain code instructions.

The only setback during development was the appearance of FAILED messages in the console due to the banner and its changing ID, but this did not pose any impediment to the correct execution of the tests. Below are the details of how the tests for each functionality were carried out, as well as the proposed hacking attempts.

### **Contract**

#### List-Mine

The tests for this functionality are straightforward. The procedure involved listing the contracts for a registered client. In terms of hacking, attempts were made to access the associated endpoints without the required permissions, but this was unsuccessful due to the URL structure preventing access. For this test, we achieved a code coverage of 92.6%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### List-All

The tests for this functionality are simple. The approach taken was to enumerate the published contracts. Regarding hacking, the only thing we had to check was that of an unauthenticated user. For this test, we attained a code coverage of 91.7%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Show

Similar to the listing operations, the testing process for this functionality has been quite straightforward, ensuring that the various cases of the object's state to be displayed were covered. Regarding hacking, in this case, the initial attempt was to display a client without being logged into the system. Then, once logged in as a client, an attempt was made to access contracts that did not belong to them. We also checked different users with different roles, unauthenticated users, and attempted to access a non-existent contract. The coverage in this case was 97.1%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Create

In this case, following the procedure outlined in the slides, we have covered all possible combinations in the creation form to verify that our validation methods were indeed correct. Regarding hacking, we conducted tests with different roles and unauthenticated users. We achieved a coverage of 93.9%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Delete

The procedure followed was to delete contracts from different clients, with the associated cascade deletion of their ProgressLog entries. As for hacking, attempts were made to perform a delete by modifying the form ID from the page inspector, testing with different roles, unauthenticated users, attempting to delete already published contracts, and from the same role but with a different contract creator. The coverage was 90.8%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Update

On this occasion, we proceeded in a similar manner to the create process, testing multiple combinations of invalid and valid forms. For hacking, we attempted, similar to the delete process, to submit a form by modifying the ID from the page inspector, testing with different roles, unauthenticated users, attempting to delete already published contracts, and from the same role but with a different contract creator. The coverage was 94.5%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Publish

I had a problem, I didn't understand how to compare a double type representing hours with a Money type. Therefore, I decided to create a converter from hours to Money, establishing that 150 EUR is equivalent to one hour (I found this information through research, noting that in a software project, this was the approximate rate). Then, the Money type is converted again using constant values to compare between different currencies (only USD, EUR, and GBP are accepted). I understand there might be better approaches, but I discussed with colleagues from other groups and sought advice (many suggested using a double type and leaving it at that, but I considered that budget, as I asked the client, was a Money type). I saw that a student asked this same question on the forum, I didn't fully understand the client's response, so I chose to use this method. The coverage is 96.2%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

### **ProgressLog**

#### ListMine

The tests for this functionality are straightforward. The procedure involved listing the progress log of a contract for a registered client. Regarding hacking, we tested with different types of users: anonymous, another role, and another user of the same role, and we found it was impossible to access the list-mine of another user. For this test, we achieved a code coverage of 92.6%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### List-All

The procedure we followed was to list the published contracts. This has two sections: authenticated (which this is not), where all authenticated users can access, and this one, which is only for clients, so users of other roles or anonymous users cannot access this section. Although this section is exactly the same as the authenticated one, the coverage is 92.1%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Show

Similar to the listing operations, the testing process for this functionality has been quite straightforward, ensuring that the various cases of the object's state to be displayed were covered. Regarding hacking, in this case, the initial attempt was to display a client without being logged into the system. Then, once logged in as a client, an attempt was made to access progress logs that did not belong to them. We also checked different users with different roles, unauthenticated users, attempted to access a non-existent progress log, and tried to access a progress log of a different client. The coverage in this case was 96.8%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Create

In this case, following the procedure outlined in the slides, we have covered all possible combinations in the creation form to verify that our validation methods were indeed correct. Regarding hacking, we conducted tests with different roles, unauthenticated users, and tried to access a progress log of a different client. We achieved a coverage of 91.8%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Delete

The procedure followed was to delete progress logs from different clients. As for hacking, attempts were made to perform a delete by modifying the form ID from the page inspector, testing with different roles, unauthenticated users, attempting to delete already published contracts, and from the same role but with a different contract creator. The coverage was 89.7%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Update

On this occasion, we proceeded in a similar manner to the create process, testing multiple combinations of invalid and valid forms. For hacking, we attempted, similar to the delete process, to submit a form by modifying the ID from the page inspector, testing with different roles, unauthenticated users, attempting to delete already published contracts, and from the same role but with a different contract creator. The coverage was 91.6%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

#### Publish

We proceeded in a manner similar to an update, testing possible cases since we realized that if you changed a value before publishing, it would be published with the changed value, so we had to use the update validation. The hacking test was the same as the update, with different users, different roles, no roles, or users with the same role but not the creators of the progress log. The coverage is 91.4%. The remaining coverage is due to the assert object != null and non-testable lines of code such as comments or variable instantiation.

## Performance Test

The performance has been analyzed before and after the application of the indices, where we see a considerable improvement in performance.

### **Before**

Gráfico, Gráfico en cascada

Descripción generada automáticamente

The graph allows us to intuitively observe that the average response time of the different instructions is more or less homogeneous, highlighting those operations that need to write to the database and therefore have a considerably longer time.

### **After**

Gráfico

Descripción generada automáticamente

### **Before vs After**

Interfaz de usuario gráfica, Aplicación, Tabla, Excel

Descripción generada automáticamente

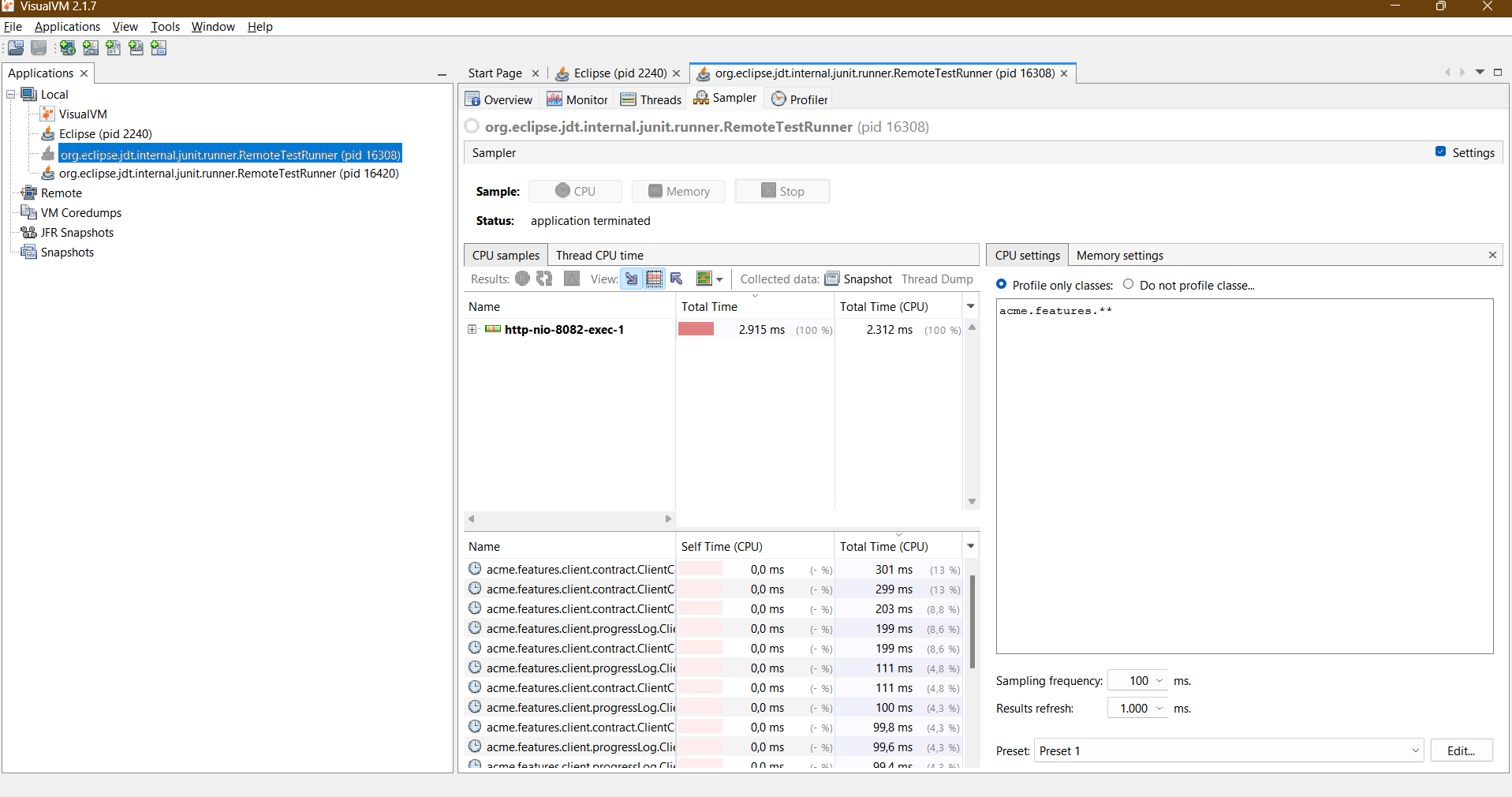
We observe through the mean that there is an improvement in time. We also see that the interval time for "Before" is [0.00957301, 0.01118953], and for "After" is [0.00839763, 0.00995823]

Tabla

Descripción generada automáticamente

We conducted a Z-Test between both, receiving a p-value of 0.027674302. Therefore, we can affirm that we have improved performance.

## Profiling Software



We see that, for example, the first one, which is createService unbind(), is consuming a lot. But in Self time, we see that all the data is at 0.0ms, which means that it’s not the method that is consuming too much time, but the methods that it invokes

## Profiling Hardware

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

The performance monitor screenshot indicates that while running the tests, the processor, disk, and network resources are being utilized efficiently without any major bottlenecks or resource contention. The occasional spikes in processor and network activity are typical during testing and do not suggest any underlying performance issues. The overall resource usage remains within acceptable limits, suggesting that the system is capable of handling the test load effectively

# Conclusion

In summary, we have successfully evaluated our code rigorously and gained detailed insights into its functioning, potential dead code, and bugs. Furthermore, thanks to the performance tests, we have been able to identify which functionalities we may need to improve if a client's requirement for response time arises.

# Bibliography

Intentionally blank