TESTING REPORT

DELIVERABLE 4

DESING AND TESTING 2

2023-2024

LUIS JAVIER PERIÁÑEZ FRANCO

|  |  |
| --- | --- |
| Date | Version |
| 05/27/2024 | V1.0 |

|  |  |
| --- | --- |
| Group: C1.053 | |
| Members | Email |
| Sánchez Márquez, Julia | [julsanmar2@alum.us.es](mailto:javgarrod5@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/miggonort1/Acme-ANS-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 | 05/26/2025 | First revision with some mistakes |
|  |  |  |
|  |  |  |

# 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.

### **Booking**

#### List

The tests here were simple, so I achieved 100% coverage. I created .safe tests to access and view the listing, including pagination and related functionality. The .hack tests simulate access attempts by users with different roles, unauthenticated users, and users with the same role accessing their own list.

#### Show

#### I also achieved 100% coverage here. The .safe tests check access to both published and unpublished bookings. The .hack tests include attempts by users with different roles, unauthenticated users, users with the same role but not the creator, and also attempts to access a null booking. All cases were tested for both published and unpublished bookings.

#### Create

Here I reached 98% coverage. The .safe tests involve executing the operation with different parameters to validate all types of constraints. The .hack tests simulate a create attempt by a user with a different role, an unauthenticated user, and a user with the same role accessing their own create. I also tested possible client-side manipulations via F12 by tampering with flight-related data.

#### Delete

For delete, I obtained 97.9% coverage. The .safe tests verify deletion of an unpublished booking. The .hack tests cover access attempts by unauthenticated users, users with a different role, users with the same role but not the creator, and attempts to delete a published booking.

#### Update

#### Here I achieved 97.3% coverage. The .safe tests include multiple parameter combinations to trigger all validation paths. The .hack tests involve an authenticated user with a different role, an unauthenticated user, a user with the same role but not the creator, and a published booking. Some conditional branches (if statements) couldn’t be fully covered, but I believe they are necessary and justified.

#### Publish

### I obtained 97.6% coverage for the publish operation. The .safe tests focus on different parameter combinations to validate all constraints. The .hack tests include access attempts by an authenticated user with a different role, an unauthenticated user, a user with the same role but not the creator, and a published booking. As with update, there are some conditional clauses I couldn’t fully cover, but I consider them necessary for security and consistency.

In both the **Update** and **Publish** operations, it is possible to modify the price field via the browser’s developer tools (F12) and submit the form without triggering an error — although the value is not actually updated. Initially, I had designed the system to throw an *unauthorised* error in such cases, to detect and block any tampering.

However, this led to significant issues related to the flight, since the price is directly derived from it. For example, if someone hacked the form to set the flight to null, the system would always throw an *unauthorised* error because it detected a mismatch in the calculated price, causing the process to break even in legitimate scenarios.

Due to the complexity and the risk of false positives, I ultimately decided the best option was to remove that specific authorisation check.

### **Booking Passenger (intermediate entity)**

#### List

I achieved 100% coverage. The .safe tests check the listing of passengers for both published and unpublished bookings. The .hack tests simulate access attempts by unauthenticated users, users with a different role, and users with the same role but who are not the owner of the booking.

#### Show

I also reached 100% coverage here. The .safe tests verify access to individual BookingPassenger entries, particularly showing the linked passenger through the master bookingPassenger.passenger. The .hack tests cover the usual cases: unauthenticated users, users with a different role, and users with the same role but not the owner.

#### Create

Here I obtained 96.2% coverage. The .safe tests focus on adding passengers and verifying that the total price updates accordingly, as well as triggering a validation error if the total currency amount reaches the limit of 1,000,000. The .hack tests include attempts by various types of users (different role, unauthenticated, same role but different customer) and cover both published and unpublished scenarios.

There was a particular issue here: while running tests, I encountered a payload error in which the test returned OK even when passengerId=0 was submitted — regardless of what I tried to inject via F12. As a result, validation errors (e.g., "passenger cannot be null") were triggered at a later stage, making the test behavior somewhat confusing and preventing me from fully testing that part of the code.

However, I’ve attached a video named ERROR that demonstrates how the validations behave correctly when manually tested. The tested passenger IDs were:

* **ID 439**: Passenger not published
* **ID 472**: Passenger belonging to another customer
* **ID 440**: Passenger already assigned to the same booking
* **ID 445**: Passenger born after the booking date

#### Delete

Here I achieved 93.2% coverage, even though everything appears green in the coverage report, so I’m not entirely sure why it didn’t reach 100%. The .safe test deletes an existing connection, while the .hack tests attempt to perform the deletion using unauthenticated users, users with a different role, and users with the same role but not the owner.

### **Passenger**

#### List

The tests here were simple, so I achieved 100% coverage. I created .safe tests to access and view the listing, including pagination and related functionality. The .hack tests simulate access attempts by users with different roles, unauthenticated users, and users with the same role accessing their own list.

#### Show

#### I also achieved 100% coverage here. The .safe tests check access to both published and unpublished bookings. The .hack tests include attempts by users with different roles, unauthenticated users, users with the same role but not the creator, and also attempts to access a null booking. All cases were tested for both published and unpublished bookings.

#### Create

Here I reached 96.5% coverage. The .safe tests involve executing the operation with different parameters to validate all types of constraints. The .hack tests simulate a create attempt by a user with a different role, an unauthenticated user, and a user with the same role accessing their own create.

#### Delete

For delete, I obtained 96.9% coverage. The .safe tests verify deletion of an unpublished booking. The .hack tests cover access attempts by unauthenticated users, users with a different role, users with the same role but not the creator, and attempts to delete a published passenger.

#### Update

#### Here I achieved 96.9% coverage. The .safe tests include multiple parameter combinations to trigger all validation paths. The .hack tests involve an authenticated user with a different role, an unauthenticated user, a user with the same role but not the creator, and a published booking.

#### Publish

### I obtained 97.0% coverage for the publish operation. The .safe tests focus on different parameter combinations to validate all constraints. The .hack tests include access attempts by an authenticated user with a different role, an unauthenticated user, a user with the same role but not the creator, and a published booking.

## Performance Test

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

### **No Index**

Gráfico

El contenido generado por IA puede ser incorrecto.

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.

### **Index**

Gráfico

El contenido generado por IA puede ser incorrecto.

### **Before vs After**

Tabla

El contenido generado por IA puede ser incorrecto.

We observe through the mean that there is an improvement in time. We also see that the interval time for "No Index" is [0.01, 0.01] and for "Index" is [0.02, 0.02],

Aplicación

El contenido generado por IA puede ser incorrecto.

We conducted a Z-Test between both, receiving a p-value of 0.008. Therefore, we can affirm that we have improved performance.

# 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