|  |
| --- |
| José Coronil Álvarez (joscoralv@alum.us.es)  Miguel García Vizcaíno (miggarviz@alum.us.es)  Javier Ignacio Milá de la Roca Dos Santos (javmildos@alum.us.es)  Emilio Manuel Vázquez Cruz (emivazcru@alum.us.es)  Group C1.028  5-26-2025 |

|  |
| --- |
| Acme AirNav Solutions, Inc. |
| **Testing Report** |
| https://github.com/Emilio-115/DP2-Acme-ANS |



**Contents**

[Executive Summary 2](#_Toc199179705)

[Revision Table 2](#_Toc199179706)

[Introduction 3](#_Toc199179707)

[Content 3](#_Toc199179708)

[Recorded Tests 3](#_Toc199179709)

[Test Coverage 5](#_Toc199179710)

[Performance Analysis 7](#_Toc199179711)

[Conclusions 10](#_Toc199179712)

[Bibliography 10](#_Toc199179713)

# Executive Summary

This report documents the tests implemented for Student 2’s tasks. All required test cases were created and executed, covering legal and illegal actions. Most features functioned as expected, a few bugs were found when testing with other dates and not useful lines were deleted. Most reachable lines of code were covered by the tests. The performance was measured in two different computers.

# Revision Table

|  |  |  |
| --- | --- | --- |
| Revision number | Date | Description |
| 1 | 5-26-2025 | Final version |

# Introduction

This document contains the analysis of the tests recorded to ensure the proper function of the requirements developed by Student 2 in the Acme-ANS project. It includes a summary of the test cases developed for the features implemented, covering users use cases and hacking attempts. It also reports on the achieved test coverage and a basic performance comparison between two computers.

# Content

According to the requirements, legal use cases and illegal hacking cases have been tested for Booking and Passengers functionality. Both features have 100% coverage, all the instructions and conditional branches have been executed, the validators do not reach full coverage but this will be explained next.

## Recorded Tests

This is a list with all the tests that have been recorded for Student 2 features, with a brief explanation. The “.safe” are legal use cases tests, “.hack” tests try to hack the application.

*Booking*

* **create.safe**: Legal test for create feature, lots of possible combinations such as empty values, invalid values, minimum values, maximum values and more.
* **update.safe**: Legal test for create feature, a few of possible combinations such as empty values, invalid values, minimum values, maximum values and more are tested.
* **show.safe, show-future.safe**: Booking information is correctly shown, also tested the right function with future dates.
* **publish.safe, delete.safe, delete-show.safe, future.safe**: Tests that check that a booking is published after publishing it, the same is done with the delete feature. Both features have been tested in future dates.
* **delete.hack, publish.hack**: This tests try different ways to hack their features, the tests includes the hacking of modifying the id of the entity of the form to a published one or the id of another customer booking and also the html has been modified to add both buttons.
* **create.hack, update.hack:** The create test checks that the entity id is not different from zero, the update test tries to modify the entity id to a published booking id and to other customer booking id, also modify read only information. Both tests modify the flight id to an invalid flight.
* **show.hack**: A customer try to access to a booking that is not owned by that customer.

*BookingRecord*

* **create.safe**: Legal test for adding passengers to a booking, all combinations tested, both empty and only one selected.
* **show.safe, list.safe**: They check that BookingRecords are listed and the information is correctly shown.
* **delete.safe, delete-show.safe**: Delete feature is tested, also the GET method to check the information is shown.
* **show.hack**: A customer tries to access to another customer BookingRecord.
* **delete.hack**: The test checks that cannot delete a published BookingRecord modifying the html and cannot delete other customer BookingRecord changing the id of the entity before deleting.
* **create.hack**: This test verifies that, you cannot modify a BookingRecord changing the entity id of the form, you cannot select a published booking and you cannot choose a draftMode passenger.

*Passenger*

* **Show.safe, list.safe**: The tests check that the passengers are correctly listed and the right information is shown.
* **delete.safe, publish.safe, delete-show.safe, publish-show.safe**: They verify that the publish and delete features work as expected. The GET methods are also tested in the last two.
* **create.safe, update.safe**: They ensure, trying lots of combinations that the forms for creating and updating function is the expected. Empty, invalid, extreme and valid values are tested, and combinations of them.
* **show.hack, list.hack**: Both tests check that a customer cannot access to lists of passenger and passengers information of which they are not owners.
* **publish.hack, delete.hack**: The tests try to delete or publish published passengers throught two ways, adding the html button and modifying the id of the entity of the form.
* **create.hack,update.hack:** With these tests is verified that a customer cannot modify a published passenger or another customer passenger by modifying the id field of the form.

The tests were recorded when the features had been informally tested a lot, so most of the bugs were solved before the recording, one important bug found thanks to the tests was in the Booking show when the date is changed to future, the application threw an exception.

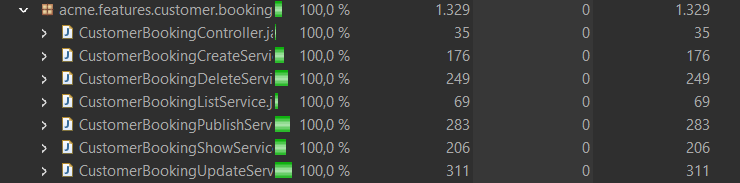
## Test Coverage

A description of the test coverage reached and explanations of the parts not fully covered.

*Booking*

**Features**

As can be seen in the picture, the recorded tests for the features have achieved 100% coverage.



**BookingValidator (86%)**

Part of the missing coverage is the result of validation paths being inaccessible through user operations even though they theoretically could be due to null checks.

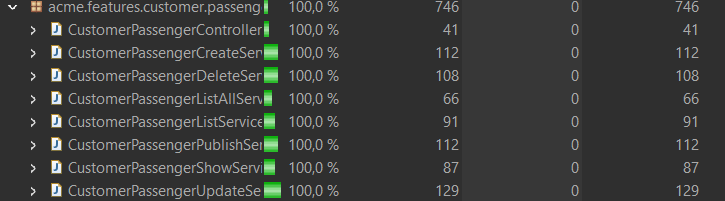
Furthermore, the necessary passenger’s validation and the credit card last nibble validation will never test all branches, because it is validated by the validate method. These checks cannot be deleted because if someone would like to add elements to the database, without these constraints they could add published bookings without passengers associated or without credit card last nibble.



*Passenger*

**Features**

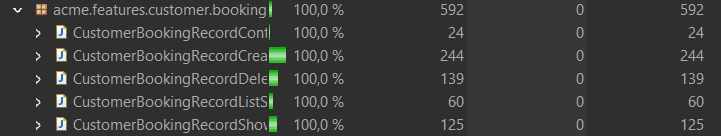
As can be seen in the picture, the recorded tests for the features have achieved 100% coverage.



*BookingRecord*

**Features**

As can be seen in the picture, the recorded tests for the features have achieved 100% coverage.



**BookingRecordValidator (87,9%)**

Part of the missing coverage is the result of validation paths being inaccessible through user operations even though they theoretically could be due to null checks.

The missing conditional branch cannot be tested, the authorize method for create and update verify that the selected passenger is published, returning a panic view if not published due to the hacking attempt. This check cannot be deleted for the reason described before; it is necessary to control manually insertions to the database with de “.csv” files and populations.

## Performance Analysis

The performance of the application was analyzed by comparing the results of two computers.

This analysis exclusively keeps in mind the customer tests, the remaining tests were not executed. Both computers executed the same tests.

*Computer 1*

The following chart shows the requests which require more time with the computer 1.

Next, this table shows some interesting metrics about the tests results.

|  |  |  |
| --- | --- | --- |
| *PC1* | |  |
|  |  |  |
| Mean | 8,074746455 |  |
| Standar Error | 0,406149223 |  |
| Median | 5,3668 |  |
| Mode | 1,8649 |  |
| Standar Deviation | 10,86022062 |  |
| Sample Variance | 117,9443919 |  |
| Kurtosis | 37,62774907 |  |
| Skewness | 5,121721063 |  |
| Range | 116,377799 |  |
| Minimum | 1,5877 |  |
| Maximum | 117,965499 |  |
| Sum | 5773,443715 |  |
| Count | 715 |  |
| Confidence Level(95,0%) | 0,797389536 |  |
|  |  |  |
| Interval(ms) | 7,277356919 | 8,87213599 |
| Interval(s) | 0,007277357 | 0,00887214 |
|  |  |  |

We are going to focus in the Cofidence Level and de Confidence Interval.

The Confidance Level is 0.797, which is a bit far from 95%. That means that 79.7% of the requests will be resolved in times between the Confindence Interval, concretely between 7.28 ms and 8.87 ms.

*Computer 2*

The following chart shows the requests which require more time with the computer 2. Times in general are a bit higher that with computer 1.

Next, the table shows some interesting metrics about the tests results.

|  |  |  |
| --- | --- | --- |
| *PC2* | |  |
|  |  |  |
| Mean | 9,88915958 |  |
| Standar Error | 0,49063799 |  |
| Median | 5,5851 |  |
| Mode | #N/D |  |
| Standar Deviation | 13,1194066 |  |
| Sample Variance | 172,118829 |  |
| Kurtosis | 12,101263 |  |
| Skewness | 3,13133766 |  |
| Range | 98,0529 |  |
| Minimum | 1,3873 |  |
| Maximum | 99,4402 |  |
| Sum | 7070,7491 |  |
| Count | 715 |  |
| Confidence Level (95,0%) | 0,96326566 |  |
|  |  |  |
| Interval(ms) | 8,92589393 | 10,8524252 |
| Interval(s) | 0,00892589 | 0,01085243 |

We are going to focus again in three, Confidence Level and Confidence Interval,

The Confidance Level is 0.96, which is over 95%. That means that 96% of the requests will be resolved in times between the Confindence Interval, concretely between 8.93 ms and 10.85 ms.

*Comparison*

The two sample z-test resulted in a P(Z<=z) two-tail value of 0,00439, which is below significantly below the alpha value for the 95% confidence level (0.05). This means that the differences amongst the times are significantly and the performance can be compared. On the one hand, the first computer is a bit faster on the average, the mean value is lower so it can respond to requests faster, also the bar chart has lower bars. On the second hand, the second computer has a Confidence Level of 96%, which is pretty good, most of the request will resolve at the expected time. The computer 1 has a Confidence Level of 79.7%, so it is not much probable that the requests will be resolved in the expected time.

In conclusion, the first computer is faster but less reliable than the second.

The results of the Z-test can be checked in the following table.

|  |  |  |
| --- | --- | --- |
|  | *PC1* | *PC2* |
| Mean | 8,07474645 | 9,88915958 |
| Known Variance | 117,944392 | 172,118829 |
| Observations | 715 | 715 |
| Hypothesized Mean Difference | 0 |  |
| z | 2,84867476 |  |
| P(Z<=z) one-tail | 0,00219509 |  |
| z Critical one-tail | 1,64485363 |  |
| P(Z<=z) two-tail | 0,00439017 |  |
| z Critical two-tail | 1,95996398 |  |

# Conclusions

The tests were recorded successfully verifying multiple use cases and GET and POST hacking cases. They were useful to find the last bugs and useless instructions. Test coverage was reached to 100% in all the features so the code has low probability to hide critical bugs.

The performance comparison was helpful to notice about how different was the impact of the computer who is executing the tests.

# Bibliography

Intentionally blank.