

Testing Report

Group: C1.027

Repository: <https://github.com/DP2-C1-027/AirNav-Logistics.git>

Group Members:

1. **Garcia de Tejada Delgado, Jose**
 - Role: Developer
 - Email: josgardel8@alum.us.es
2. **Peñaloza Friqui, Nora**
 - Role: Developer
 - Email: norpennfri@alum.us.es
3. **Niza Cobo, Manuel Jesús**
 - Role: Manager, Developer
 - Email: mannizcob@alum.us.es
4. **Gomez Claraco, Nicolas**
 - Role: Developer
 - Email: nicgomcla@alum.us.es
5. **Campos Diez, Lucia**
 - Role: Tester, Developer
 - Email: luccamdie@alum.us.es

05/25/2025

Table of Contents

1. Executive Summary
2. Review Table
3. Introduction
4. Content
 - a. Funtional Testing
 - b. Performance Testing
5. Conclusion
6. Bibliography

1. Executive Summary





This document aims to present a comprehensive report on the tests carried out in project G1.027, focusing on the individual requirements of student #2. Through the chapters on functional testing and performance testing, as well as a coverage analysis, the report details how these tests have contributed both to improving the code implementation and to detecting and correcting previously unknown errors. Testing represents an essential component in the development and maintenance of software, as it allows verification of the system’s correct behavior and ensures its quality.

2. Review Table

Version	Description	Date
V1.0	Initial version	05/25/2025
V2.0	Final version	05/26/2025

3. Introduction

This testing report is structured into two main sections: functional tests and performance tests. The first section details the test cases implemented for the entities Booking, BookingRecord, and Passenger, following a formal methodology for functional validation. The tests are grouped by action type — list, show, create, delete, update, and publish — and are classified into files with the .safe extension, which include both positive and negative (invalid data) cases, and .hack files, which are designed to evaluate the system’s resilience against cyberattacks or malicious behavior. Despite time constraints, an exhaustive process was followed according to the principles of integrity, quality, and security as taught in class, achieving a 100% success rate in the execution of the tests, all of which returned the expected results. A screenshot of the test coverage for the Booking, BookingRecord, and Passenger entities is attached.

>  acme.features.customers.passenger	100,0 %	1.498	0	1.498
>  acme.features.customers.dashboard	2,2 %	12	535	547
>  acme.features.customers.bookingRecord	100,0 %	933	0	933
>  acme.features.customers.booking	100,0 %	1.666	0	1.666

▼  acme.features.customers.passenger	100,0 %	1.498	0	1.498
>  CustomersPassengersUpdateService.java	100,0 %	210	0	210
>  CustomersPassengersShowService.java	100,0 %	164	0	164
>  CustomersPassengersListService.java	100,0 %	45	0	45
>  CustomersPassengersController.java	100,0 %	47	0	47
>  CustomersPassengerPublishService.java	100,0 %	213	0	213
>  CustomersPassengerDeleteService.java	100,0 %	213	0	213
>  CustomersPassengerCreateService2.java	100,0 %	289	0	289
>  CustomersPassengerCreateService.java	100,0 %	188	0	188
>  CustomersBookingPassengerListService.java	100,0 %	129	0	129

▼ acme.features.customers.booking	100,0 %	1.666	0	1.666
> CustomersBookingUpdateService.java	100,0 %	281	0	281
> CustomersBookingShowService.java	100,0 %	223	0	223
> CustomersBookingPublishService.java	100,0 %	334	0	334
> CustomersBookingListService.java	100,0 %	112	0	112
> CustomersBookingDeleteService.java	100,0 %	264	0	264
> CustomersBookingCreateService.java	100,0 %	417	0	417
> CustomersBookingController.java	100,0 %	35	0	35

The second section analyzes the system's performance before and after an index-based refactor. It includes graphs and statistical analysis of response times, with 95% confidence intervals and a hypothesis test used to objectively assess performance improvements. These tests not only verify the current functioning of the system but also provide a solid foundation for validating correct behavior when future modifications or new functionalities are introduced.

4. Content

a. Functional testing:

Below is a summary of the test cases implemented. In every instance where a test failed due to unexpected behavior or an implementation error, the issue was corrected and the test was repeated to ensure the system functioned as intended. It is worth noting that any instance of a 500 error corresponded to the message "Access is not authorised", indicating unauthorized access.

The test cases were organized into two main categories: .safe and .hack. The .safe files contain both positive and negative functional tests. Positive cases evaluated whether the functionality under test executed correctly under various conditions, including data variations, different date ranges, and string inputs of varying lengths, to ensure system stability and robustness. Negative cases focused on verifying that validations triggered appropriately in response to invalid input, extreme or erroneous values, incomplete or incorrectly formatted forms, and scenarios where the system was expected to block the action.

Meanwhile, the .hack files focused on security testing, specifically targeting the system's integrity and confidentiality. For each functional action, three main security scenarios were tested: one with an anonymous (unauthenticated) user, another with an authenticated user with an unauthorized role (as administrator), and a third with an authenticated user of the correct role. This final scenario evaluated the system's response to attempts to access data that did not belong to the logged-in user. These tests used data from other users within the same realm, including both published and unpublished resources, as well as data owned by the user executing the action, allowing for comprehensive comparison across access levels.

Additionally, for listings related to Booking and Passenger, tests were conducted using users from a different realm (e.g., from the administrator role) to verify that roles other than customer could not access this information or its associated resources. These tests

helped uncover potential vulnerabilities in data separation among users who may share roles but operate within different usage contexts.

It is important to highlight that prior to conducting formal functional testing, a thorough and meticulous round of informal testing was performed. During this phase, the development team focused on identifying and resolving as many bugs and issues as possible through manual exploration. This proactive approach allowed the application to reach a relatively stable state before entering the functional testing phase, thereby ensuring that the core functionalities could be tested more efficiently and with fewer interruptions due to critical errors.

Overall, this testing process not only validated the functional correctness and security of the system, but also identified and resolved key issues that could have compromised its integrity in a real-world environment.

- **Booking:**
 - **List-mine.safe:** This test ensures that a customer can list their own bookings. We log in as customer1 and confirm that all of their bookings can be successfully listed
 - **Show.safe:** This test verifies that a customer can view the full details of their own bookings listed in the system. We log in as customer1, list their bookings, and access the detail view of each one.
 - **Create.safe:** It is verified that a customer can create a booking as long as valid data is entered. We log in as customer1 and access the form, leave the entire form empty, and confirm that it requires us to complete the appropriate fields with valid values in order to create the booking. With the *LocatorCode* attribute, we tested all possible cases: we entered a repeated code, used lowercase letters, a string shorter than the required length, a string longer than allowed, tried adding characters that are neither numbers nor uppercase letters... We attempted to leave the *travelClass* unselected. The *purchaseMoment* date is a read-only attribute, as is the price, so we did not modify them. Similarly to *travelClass*, we left the flight unselected. For the *lastNibble* attribute, which is optional, we tested entering letters instead of numbers and exceeding the 4-character limit. After performing all types of invalid input attempts, we made a valid attempt meeting the validation rules, and the booking was successfully created.
 - **Update.safe:** It is verified that a customer can update a booking as long as the data is valid. To do this, we performed a series of attempts such as entering a *LocatorCode* that was already in use, leaving *travelClass* unselected, entering letters in the *lastNibble* field, and exceeding its length. We also tested a variety of *LocatorCode* strings, such as lowercase strings and characters that were not uppercase letters or numbers. After performing these attempts, we made a valid one, and the booking was successfully updated.

- **Publish.safe:** It is verified that a customer can only publish a booking if the data is correct and if there are *Passenger* entities associated with that booking. These passengers must be published, and the *lastNibble* value must also be present. We attempted to publish a booking whose *Passengers* were not published and confirmed that a message appeared indicating that there were unpublished passengers. We published the *Passengers* and tried again. Before doing so, we had deleted the *lastNibble* value; the system did not allow us to publish it because the attribute was missing. We tested a *lastNibble* with letters, more than 4 characters... Finally, we completed the form with valid data and published the booking. We went to another booking without *Passengers* and attempted to publish it, but a message appeared indicating that there were no *Passengers*. We associated a *Passenger* with the booking and tried again. Finally, we published the *Passenger* and then published the booking.
- **Delete.safe:** We attempted to verify that a customer can delete any of their bookings as long as they are not published. We logged in as customer1, selected an unpublished booking, and deleted it. We then selected a published one and verified that the delete button was not visible, so we could not delete it
- **List-mine.hack:** We aimed to verify that a user who should not have access would try to list bookings. We logged in as customer1, listed their bookings, and copied the URL. We then logged out of customer1 and logged in as an administrator. We pasted the booking list URL and received a *500 Access is not authorised* response.
- **Show.hack:** The intention is to verify whether a user can display information about bookings that they should not be able to access, and to ensure that such access cannot be hacked. While logged in as customer1, we selected a booking URL and made a series of modifications to it. First, we tested by changing the booking ID in the URL to one belonging to customer2 (we tested with both unpublished and published bookings). We also modified the ID to incorrect values: we used strings, negative numbers, and left it as null. We confirmed that these hacking attempts are properly handled, as each time the system returned a *500 Access is not authorised* error.
- **List-mine-other.hack:** We aimed to verify that another customer, when accessing the booking list URL, only sees their own bookings and not those of another customer. To do this, we logged in as customer1, copied the URL, logged out, then logged in as customer2. We pasted the URL and verified that the data displayed belonged to customer2. We also confirmed that access to the list is not allowed if the user is not authenticated as a customer.
- **Create.hack:** We verified that the creation of a booking cannot be hacked. Using the browser inspection tool, we made some changes to check that it's not possible to submit invalid data that isn't visible

in the user interface (we focused particularly on the select choices). First, we tested by changing the value of the select choices, and the framework handled it properly by throwing an invalid value exception, so no further action was needed. Then, regarding the select choice for the flight selection, we verified that it's not possible to create a booking with flights that are not in the future or are not published. To test this, we changed the selected flight ID to one corresponding to a flight with a past date (relative to the current creation date) and also to flights that were unpublished. In both cases, the system returned a *500 Access is not authorised* error. Additionally, we verified that the flight ID was valid — meaning it was not a string, an ID of a non-existent flight, or null. Finally, we confirmed that when creating a booking, it does not overwrite any existing booking in the database. To ensure this, we checked that when the submit button is pressed, the booking ID is 0. We verified that if the ID is different from 0, the system returns an *Access is not authorised* error.

- **Update.hack:** We aimed to ensure that one customer cannot modify the bookings of other customers and, in addition, that the update operation is secure against hacking attempts. We logged in as customer1 and, by inspecting the update button of a booking, we attempted to update: one of our own bookings that was already published, an unpublished booking belonging to customer2, and a published booking from customer2. In all three cases, the error *Access is not authorised* was triggered, as these are illegal actions. Additionally, we verified that the received IDs are valid. We tested by setting the ID to null, a string, and negative numbers. In all cases, it was confirmed that the update action does not authorize any of these invalid ID changes.
- **Publish.hack:** We aimed to ensure that one customer cannot publish the bookings of other customers and, in addition, that the publish operation is secure against hacking attempts. We logged in as customer1 and, by inspecting the publish button of a booking, we attempted to publish: one of our own bookings that was already published, an unpublished booking belonging to customer2, and a published booking from customer2. In all three cases, the error *Access is not authorised* was triggered, as these are illegal actions. Additionally, we verified that the received IDs are valid. We tested by setting the ID to null, a string, and negative numbers. In all cases, it was confirmed that the publish action does not authorize any of these invalid ID changes.
- **Delete.hack:** We aimed to ensure that one customer cannot delete the bookings of other customers and, in addition, that the delete operation is secure against hacking attempts. We logged in as customer1 and, by inspecting the delete button of a booking, we attempted to delete: one of our own bookings that was already published, an unpublished booking belonging to customer2, and a

published booking from customer2. In all three cases, the error *Access is not authorised* was triggered, as these are illegal actions. Additionally, we verified that the received IDs are valid. We tested by setting the ID to null, a string, and negative numbers. In all cases, it was confirmed that the delete action does not authorize any of these invalid ID changes.

- **readOnly.hack:** We aimed to verify that the form attributes marked as readOnly during the create and update processes cannot be modified in the database. To do this, we changed their values using the browser's inspect tool and observed that when clicking create or update, the values reverted to their original state. In this way, it was confirmed that readOnly data cannot be externally altered.
- **Show-other.hack:** The intention is to verify that a user cannot view booking information they are not authorized to access. Without authenticating, we tried to access the URL to display a customer given its ID (`customers/booking/show?id=345`), and we received a 500 ERROR.
- **CUDP-other.hack:** The intention is to verify that a user cannot delete, create, update, publish booking they are not authorized to access. Without authenticating, we tried to access the following URL: `customers/booking/create`, `customers/booking/delete`, `customers/booking/update`, `customers/booking/publish`, and we received a 500 ERROR.

- **Passenger:**

- **Create.safe:** It must be verified that a customer can create a passenger as long as the data is valid. We logged in as customer1 and accessed the passenger creation form, attempting to submit with all types of invalid data—from a completely empty form to a seemingly correct form with individually invalid fields. We verified that the date of birth cannot be in the future, and checked the character limits for the specialNeeds and fullName fields and invalid emails too. Finally, we create a passenger with valid data.
- **Create.hack** We verified that the creation of a passenger cannot be hacked and that the IDs sent to the backend are correct, ensuring that existing IDs cannot be overwritten. To do this, we opened the inspection window and tested various ID values before submitting; all values except `id=0` resulted in a 500 Access is not authorised error. For this test, we changed the ID to that of any passenger in the dataset. We tried using a string, a null value, and a negative number.
- **createBooking.safe:** We confirmed that a passenger can be directly created and associated with a booking using valid values. To do this, we logged in as customer1, accessed the passenger creation form, and attempted submissions with all kinds of invalid data—from completely empty forms to individually invalid attributes. We verified

that the date of birth cannot be in the future and that the character limits for specialNeeds and fullName are enforced.

- **createBooking.hack:** We confirmed that the creation of a passenger associated with a booking cannot be hacked and that valid IDs are received by the backend, preventing the overwriting of existing IDs. We used the inspect window to test various ID values before submission; all values except id=0 triggered a 500 Access is not authorised error. Additionally, we verified that the booking to which the passenger is being added exists, is not published, and does not belong to another customer. Also, we tested that the bookingID is a valid id and is not a string or an empty value.
- **delete.safe:** We attempted to verify that a customer can delete any of their passengers as long as they are not published. We logged in as customer1, selected an unpublished passenger, and deleted it. Then, we selected a published one and confirmed that the delete button was not visible—hence, it could not be deleted.
- **delete.hack:** We aimed to ensure that one customer cannot delete another customer's passenger and that the delete operation is secure against hacking attempts. We logged in as customer1 and, using the inspect tool on the delete button of a passenger, we attempted to delete: one of our own passengers that was already published, an unpublished passenger belonging to customer2, and a published passenger from customer2. In all three cases, the system triggered the Access is not authorised error, as the actions were illegal. Additionally, we verified that the IDs received were valid by testing values such as null, strings, and negative numbers. In every case, it was confirmed that the delete operation did not authorize any of these invalid ID modifications.
- **list-mine.hack:** We aimed to verify that a user who should not have access would try to list passengers. We logged in as customer1, listed their passenger, and copied the URL. We then logged out of customer1 and logged in as an administrator. We pasted the passenger list URL and received a *500 Access is not authorised* response.
- **list-mine.safe:** This test ensures that a customer can list their own passengers. We log in as customer1 and confirm that all of their passengers can be successfully listed
- **passengerListBooking.hack:** The goal is to verify that the list of passengers for a booking is not displayed if the logged-in user is not the owner of that booking. This is tested by using booking IDs that belong to customer2 (both published and unpublished bookings). Additionally, it is verified that a 500 error is triggered if the booking ID in the URL is replaced with an invalid value such as null, letters, or non-existent IDs.
- **passengerListBooking.safe:** This test ensures that a customer can list their own passengers in their booking. We log in as customer1 and confirm that all of their passengers can be successfully listed

- **publish.safe:** It is verified that a passenger can only be published if it has not already been published. We log in as customer1 and confirm that we can only publish a passenger who is not already published
- **publish.hack:** We ensure that it is not possible to publish a passenger through illegal means. We logged in as customer1 and, using the browser's inspect tool on the publish button of a passenger, attempted to publish one of our own passenger that was already published, an unpublished passenger belonging to customer2 and a published. In all three cases, the system triggered a 500 Access is not authorised error, as these actions are illegal. Additionally, we tested the handling of invalid IDs by setting them to null, a string, or negative numbers. In all cases, it was confirmed that the publish action does not authorize any of these invalid ID values.
- **show.safe:** this test verifies that a customer can view the full details of their own passengers listed in the system. We log in as customer1, list their passengers, and access the detail view of each one.
- **show.hack:** The intention is to verify whether a user can display information about passengers they should not have access to, and ensure that such unauthorized access cannot be achieved through hacking. While logged in as customer1, we took a valid passenger URL and tried several modifications. We replaced the passenger ID with one belonging to customer2 (both published and unpublished), we tested invalid values such as strings, negative numbers, and null. Each time, the system responded with a 500 Access is not authorised error, confirming that these hacking attempts are blocked.
- **update.hack:** We aimed to ensure that one customer cannot modify the passenger of other customers and, in addition, that the update operation is secure against hacking attempts. We logged in as customer1 and, by inspecting the update button of a passenger, we attempted to update: one of our own passenger that was already published, an unpublished passenger belonging to customer2, and a published passenger from customer2. In all three cases, the error *Access is not authorised* was triggered, as these are illegal actions. Additionally, we verified that the received IDs are valid. We tested by setting the ID to null, a string, and negative numbers. In all cases, it was confirmed that the update action does not authorize any of these invalid ID changes.
- **update.safe:** We verify that a customer can update a passenger as long as the data is valid and the passenger is not published. We logged in as customer1, accessed an unpublished passenger, and tested various invalid inputs just like in the creation tests. Once confirmed, we successfully performed a valid update.
- **other.hack:** The intention is to verify that a user cannot delete, create, update, publish passenger they are not authorized to access. Without authenticating, we tried to access the following URL: customers/passenger/create, customers/passenger/delete, customers/passenger/update, customers/passenger/publish,

customers/passenger/list, customers/passenger/show?id=391, customers/passenger/createBooking?bookingId=345, customers/passenger/passengerList?bookingId=345 and we received a 500 ERROR.

- **publishInvalid.safe:** It is verified that a passenger cannot be published with invalid data. First, the form is submitted empty, and then different test cases are applied to each individual attribute: invalid values, incorrect formats, disallowed lengths, etc. In all cases, the system prevents the passenger from being published until all data meets the required validations.
- **BookingRecord:**
 - **Create.hack:** We verified that the creation of a bookingRecord cannot be hacked. Using the browser inspection tool, we made some changes to check that it's not possible to submit invalid data that isn't visible in the user interface (we focused particularly on the select choices). We verified that it's not possible to create a bookingRecord with passenger that are from other customer. To test this, we changed the selected passenger ID to one corresponding to a passenger from customer2 (we did it with passenger unpublished and published) In both cases, the system returned a *500 Access is not authorised* error. Additionally, we verified that the passenger ID was valid — meaning it was not a string, an ID of a non-existent passenger, or null. Finally, we confirmed that when creating a bookingRecord it does not overwrite any existing bookingRecord in the database. To ensure this, we checked that when the submit button is pressed, the ID is 0. We verified that if the ID is different from 0, the system returns an *Access is not authorised* error.
 - **Create.safe:** It must be verified that a customer can create a passenger as long as the data is valid. We left the passenger unselected and also tried selecting a passenger that was already associated with the same booking. In both cases, we confirmed that the system did not allow the creation due to invalid data. Finally, we selected a valid option and successfully created the booking record.
 - **Delete.hack:** We aimed to ensure that one customer cannot delete another customer's bookingRecord and that the delete operation is secure against hacking attempts. We logged in as customer1 and, using the inspect tool on the delete button of a bookingRecord, we attempted to delete: one of our own that was already published, an unpublished belonging to customer2, and a published from customer2. In all three cases, the system triggered the *Access is not authorised* error, as the actions were illegal. Additionally, we verified that the IDs received were valid by testing values such as null, strings, and negative numbers. In every case, it was confirmed that the delete operation did not authorize any of these invalid ID modifications.
 - **Delete.safe:** We attempted to verify that a customer can delete any of their bookingRecord as long as they are not published. We logged

in as customer1, selected an unpublished bookingRecord, and deleted it. Then, we selected a published one and confirmed that the delete button was not visible—hence, it could not be deleted

- **List-mine.safe:** This test ensures that a customer can list their own bookingRecord. We log in as customer1 and confirm that all of their bookingRecord in their passenger can be successfully listed
- **List-mine.hack:** The goal is to verify that the list of bookingRecord for a passenger is not displayed if the logged-in user is not the owner of that passenger. This is tested by using passenger IDs that belong to customer2 (both published and unpublished). Additionally, it is verified that a 500 error is triggered if the booking ID in the URL is replaced with an invalid value such as null, letters, or non-existent IDs.
- **Show.safe:** this test verifies that a customer can view the full details of their own bookingRecord listed in the system. We log in as customer1, list their bookingRecord, and access the detail view of each one.
- **Show.hack:** The intention is to verify whether a user can display information about bookingRecord they should not have access to, and ensure that such unauthorized access cannot be achieved through hacking. While logged in as customer1, we took a valid bookingRecord URL and tried several modifications. We replaced the passenger ID with one belonging to customer2 (both published and unpublished), we tested invalid values such as strings, negative numbers, and null.

Each time, the system responded with a 500 Access is not authorised error, confirming that these hacking attempts are blocked.

- **Other.hack:** The intention is to verify that a user cannot delete, create, show bookingRecord they are not authorized to access.

Without authenticating, we tried to access the following URL: customers/ booking-record/create?bookingId=345, customers/ booking-record/delete, customers/ booking-record/list?passengerId=391, customers/ booking-record/show?id=450 and we received a 500 ERROR.

- **passengerCreate.hack:** When creating a BookingRecord, it is essential to ensure that the passenger_id provided is valid and consistent with the application's data integrity rules. Specifically, the system must verify that the passenger exists in the database and is associated with the same customer who is initiating the booking. Under no circumstances should a passenger belonging to a different customer be linked to the booking. So we tested using id from customer2 (published and no published passenger). To ensure the robustness of this validation, the system is tested with various invalid inputs, including a null value, a string, or an arbitrary numeric. These test cases help confirm that the application correctly prevents the

creation of booking records with invalid or unauthorized passenger references

Features								
Verb	Feature	Request counter	Request entropy	Request Simpson	Response counter	Response entropy	Response Simpson	
GET	anonymous/system/sign-in	89	0	0	89	0.173	0.168	
GET	any/system/welcome	203	0	0	203	0.842	0.82	
GET	authenticated/system/sign...	45	0	0	45	0	0	
GET	customers/booking-record...	32	0.468	0.406	32	0.233	0.224	
GET	customers/booking-record...	9	0	1	9	0.234	0.222	
GET	customers/booking-record...	29	0.683	0.68	47	0.591	0.457	
GET	customers/booking-record...	20	0	0	20	0.71	0.487	
GET	customers/booking/create	16	0	0	16	0.236	0.221	
GET	customers/booking/delete	2	0	0	2	0.125	1	
GET	customers/booking/list	87	0	0	2205	0.87	0.844	
GET	customers/booking/publish	1	0	0	1	0	1	
GET	customers/booking/show	113	0	0	113	0.33	0.224	
GET	customers/booking/update	1	0	0	1	0	1	
GET	customers/passenger/create	8	0	0	8	0.31	0.298	
GET	customers/passenger/crea...	18	0.805	0.794	18	0.318	0.301	
GET	customers/passenger/delete	2	0	0.333	2	0.036	0.29	
GET	customers/passenger/list	68	0	0	1190	0.977	0.973	
GET	customers/passenger/pas...	23	0.764	0.745	41	0.962	0.915	
GET	customers/passenger/publ...	1	0	0	1	0	1	
GET	customers/passenger/show	61	0	0	61	0.765	0.606	
GET	customers/passenger/upd...	1	0	0	1	0	1	
POST	anonymous/system/sign-in	89	0.453	0.345	89	0	0	
POST	customers/booking-record...	24	0.312	0.309	24	0.395	0.583	
POST	customers/booking-record...	8	0.544	0.25	8	0.969	0.893	
POST	customers/booking/create	29	0.484	0.384	29	0.507	0.379	
POST	customers/booking/delete	9	0	0	9	0.946	0.861	
POST	customers/booking/publish	17	0.704	0.441	17	0.72	0.623	
POST	customers/booking/update	20	0.557	0.432	20	0.365	0.394	
POST	customers/passenger/create	26	0.676	0.557	26	0.647	0.489	
POST	customers/passenger/crea...	19	0.579	0.454	19	0.652	0.575	
POST	customers/passenger/delete	9	0	0	9	0.918	0.778	
POST	customers/passenger/publ...	17	0.747	0.644	17	0.758	0.707	
POST	customers/passenger/upd...	21	0.781	0.669	21	0.593	0.516	

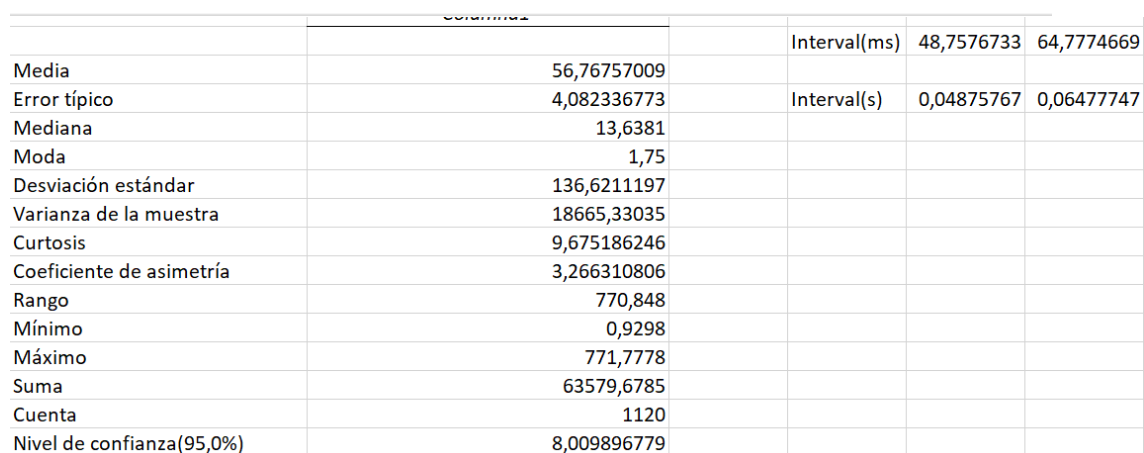
Search

Features								
Verb	Feature	Request counter	Request entropy	Request Simpson	Response counter	Response entropy	Response Simpson	
GET	anonymous/system/sign-in	38	0	0	38	0.187	0.176	
GET	any/system/welcome	76	0	0	76	0.814	0.813	
GET	authenticated/system/sign...	19	0	0	19	0	0	
GET	customers/booking-record...	3	0	0	3	0.115	0.083	
GET	customers/booking-record...	6	0.959	0.867	9	0.657	0.7	
GET	customers/booking-record...	3	0	0	3	0.575	0.5	
GET	customers/booking/create	1	0	0	1	0	0.143	
GET	customers/booking/list	20	0	0	520	0.928	0.814	
GET	customers/booking/show	16	0	0	16	0.694	0.492	
GET	customers/passenger/create	1	0	0	1	0	0.25	
GET	customers/passenger/crea...	1	0	1	1	0	0.25	
GET	customers/passenger/list	17	0	0	306	0.961	0.949	
GET	customers/passenger/pas...	5	0.96	0.8	15	0.954	0.933	
GET	customers/passenger/show	13	0	0	13	0.821	0.636	
POST	anonymous/system/sign-in	38	0.667	0.342	38	0	0	
POST	customers/booking-record...	5	0.32	0.267	5	0.286	0.429	
POST	customers/booking-record...	1	0	0.5	1	0	1	
POST	customers/booking/create	13	0.473	0.34	13	0.422	0.415	
POST	customers/booking/delete	1	0	0.167	1	0	1	
POST	customers/booking/publish	9	0.851	0.693	9	0.6	0.431	
POST	customers/booking/update	10	0.62	0.437	10	0.368	0.397	
POST	customers/passenger/create	21	0.667	0.538	21	0.648	0.49	
POST	customers/passenger/crea...	13	0.597	0.457	13	0.654	0.576	
POST	customers/passenger/delete	1	0	0.2	1	0	1	
POST	customers/passenger/publ...	9	0.916	0.833	9	0.765	0.707	
POST	customers/passenger/upd...	13	0.636	0.467	13	0.595	0.519	

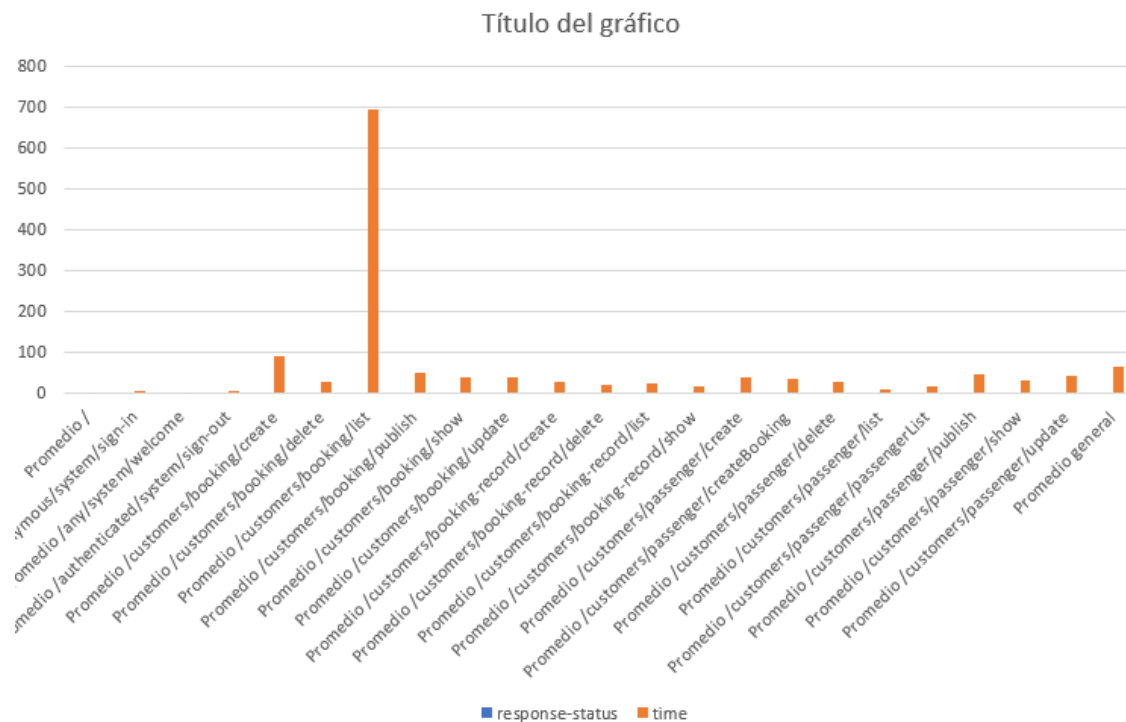
Features								
Verb	Feature	Request counter	Request entropy	Request Simpson	Response counter	Response entropy	Response Simpson	
GET	anonymous/system/sign-in	51	0	0	51	0.176	0.164	
GET	any/system/welcome	127	0	0	127	0.852	0.828	
GET	authenticated/system/sign...	26	0	0	26	0	0	
GET	customers/booking-record...	29	0	0	29	0.237	0.227	
GET	customers/booking-record...	9	0	1	9	0.234	0.222	
GET	customers/booking-record...	23	0.642	0.606	38	0.535	0.373	
GET	customers/booking-record...	17	0	0	17	0.599	0.408	
GET	customers/booking/create	15	0	0	15	0.239	0.224	
GET	customers/booking/delete	2	0	0	2	0.125	1	
GET	customers/booking/list	67	0	0	1685	0.881	0.844	
GET	customers/booking/publish	1	0	0	1	0	1	
GET	customers/booking/show	97	0	0	97	0.246	0.158	
GET	customers/booking/update	1	0	0	1	0	1	
GET	customers/passenger/create	7	0	0	7	0.325	0.317	
GET	customers/passenger/crea...	17	0	0	17	0.322	0.304	
GET	customers/passenger/delete	2	0	0.333	2	0.036	0.25	
GET	customers/passenger/list	51	0	0	884	1	0.973	
GET	customers/passenger/pas...	18	0.764	0.728	26	0.973	0.903	
GET	customers/passenger/publ...	1	0	0	1	0	1	
GET	customers/passenger/show	48	0	0	48	0.864	0.589	
GET	customers/passenger/upd...	1	0	0	1	0	1	
POST	anonymous/system/sign-in	51	0.471	0.353	51	0	0	
POST	customers/booking-record...	19	0.311	0.305	19	0.97	0.924	
POST	customers/booking-record...	7	0	0	7	0.963	0.905	
POST	customers/booking/create	16	0.311	0.204	16	0.272	0.219	
POST	customers/booking/delete	8	0	0	8	0.928	0.857	
POST	customers/booking/publish	8	0	0	8	1	1	
POST	customers/booking/update	10	0.234	0.1	10	0.935	0.844	
POST	customers/passenger/create	5	0	0	5	0.961	0.9	
POST	customers/passenger/crea...	6	0	0	6	0.97	0.933	
POST	customers/passenger/delete	8	0	0	8	0.953	0.821	
POST	customers/passenger/publ...	8	0	0	8	0.969	0.893	
POST	customers/passenger/upd...	8	0	0	8	0.969	0.893	

Search

b. Performance Testing
Pre-refactoring tests with indexes



Refactoring tests with indexes



		Interval(ms)	64,0275593	85,8007585
Media	74,91415889			
Error típico	5,548476335	Interval(s)	0,06402756	0,08580076
Mediana	14,6463			
Moda	2,964			
Desviación estándar	185,6046196			
Varianza de la muestra	34449,07481			
Curtosis	8,498086079			
Coefficiente de asimetría	3,176489501			
Rango	1010,9718			
Mínimo	0,7731			
Máximo	1011,7449			
Suma	83828,9438			
Cuenta	1119			
Nivel de confianza(95,0	10,88659957			

By looking at both charts, it can be seen that the requests that take the most time are the booking listings. It can be observed that, after the refactoring process involving the addition of indexes, the average execution time per feature has experienced a slight increase. This outcome, although seemingly counterintuitive, is consistent with the nature of indexing mechanisms in database systems.

Indexes are designed to optimize query performance, particularly when operating on large datasets. Their benefits become more evident as the volume of data grows, enabling faster access and retrieval by reducing the amount of data that needs to be scanned during query execution. However, in environments where the dataset is relatively small—as is the case in this scenario—the overhead introduced by maintaining and accessing indexes may outweigh the potential performance gains. As a result, the expected improvement in execution time is not yet reflected in the current metrics.

Therefore, the slight increase in execution time does not indicate an ineffective refactoring but rather highlights the importance of data volume when evaluating the impact of indexing strategies. It is anticipated that, as the dataset scales, the advantages of indexing will become more pronounced and lead to significant performance improvements.

The 95% confidence intervals are (48.75, 64.77) before the refactoring and (64.02, 85.80) after the refactoring. It can be observed that the average execution time is slightly higher following the refactoring process.

This increase in the confidence interval range indicates a modest upward shift in the mean execution time. Although the intervals do not overlap significantly, the difference is not large enough to suggest a drastic change in performance. Instead, it reflects a consistent but small increase in average values.

It is important to note that such variations are expected, especially in scenarios where the changes made—such as the introduction of indexes—do not provide immediate performance benefits due to limited data volume. In this context, the slight increase in execution time is not necessarily indicative of inefficiency, but rather a temporary effect that may be reversed or significantly improved as the database scales and the indexes begin to demonstrate their full potential.

Analysis: Hypothesis Testing:

	<i>before</i>	<i>after</i>
Media	50,27401937	66,21499059
Varianza (conocida)	16705,82184	30781,49407
Observaciones	1275	1275
Diferencia hipotética de las medias	0	
z	-2,612047429	
P(Z<=z) una cola	0,004500088	
Valor crítico de z (una cola)	1,644853627	
Valor crítico de z (dos colas)	0,009000177	
Valor crítico de z (dos colas)	1,959963985	

In this case, we observe that the two-sample z-test for means yields a p-value of 0.004, which is below the significance level $\alpha = 0.05$. According to hypothesis testing principles, when the p-value falls within the interval $[0.00, \alpha)$, we reject the null hypothesis and accept that there is a statistically significant difference between the two sample means. To interpret the direction of this difference, we compare the average execution times before and after the refactoring. Since the average execution time after the changes is higher, we conclude that the refactoring did not improve performance under the current conditions. Instead, it led to a slight increase in execution time, which, as previously discussed, may be due to the relatively small dataset where indexing does not yet yield measurable benefits.

5. Conclusion

The report includes all the tests carried out by Student 4, which have been useful in identifying some errors in the code. The informal testing process has been essential for detecting bugs prior to formal testing, thus helping to achieve a stable and bug-free version of the application.

6. Bibliography

Intentionally blank.