**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

03/12/2025

**Table of Contents**

1. **Executive Summary**
2. **Revision Table**
3. **Introduction**
4. **Contents**
   1. **Functional testing**
   2. **Performance testing**
5. **Conclusions**
6. **Bibliography**

**Executive Summary**

Testing is a cornerstone of design and maintenance of software projects, aiding the members of the development team to more easily and reliably assert the quality and robustness of the system they are working on while ensuring new changes don’t introduce unknown vulnerabilities or defects. In this document, student #1 will produce a report on the testing done for his requirements #8 and #9

**Revision Table**

|  |  |  |
| --- | --- | --- |
| **Revision Number** | **Date** | **Description** |
| 1.0 | 26/05/2025 | Initial report for D02 |

**Introduction**

In the first section about functional testing, personal requirements will be tested formally, grouped by feature.

In the second section, about performance testing, said performance will be analyzed using statistical analysis and graphing tools looking for statistical significance and a confidence interval of 95% for our expected delay when serving requests to the user; alongside a contrast of hypothesis resulting of said analysis.

**Contents**

* 1. **Functional testing:**

In the following paragraphs, I’ll summarize the implemented testcases, grouped by entity and feature, and what each file in the testsuite has been used on.

**Flight**

* **Create.safe:** We log in as manager1 and attempt to create as many invalid flights as possible (empty fields, outside of range, invalid format, duplicate flightNumber, etc), and with valid fields (middle of range, edge of range). We finish with a valid create form to also get coverage and validate the perform method
* **Delete.safe**: We log in as manager1 and attempt to delete all unpublished flights. Some of these contain published legs (these flights are mocking a flight-leg agglomeration that’s in the process of being published by a manager right as the testing happens) so we also test the unbind function and the validate. Aside from that, delete doesn’t hold much validation to test since it only needs a legal form and a flight with no published legs to be fully valid
* **List.safe:** We log in as manager1 and attempt to access our flights. Then we do the same thing as manager3 (empty list). Not much more testing can be done on this feature
* **Publish.safe:** We log in as manager1 and attempt to publish unpublished flights with valid data but only unpublished legs, valid data but no legs associated, and valid data and published legs but also unpublished legs associated. Then we also attempt to publish as many invalid flights as possible (similar to create.safe, empty fields, outside of range, invalid format, duplicate flightNumber), and with valid fields (middle of range, edge of range). We finish with valid publishes to test the proper functionality of the perform function
* **Publish2.safe:** In improving the authorization restrictions, some positive tests (mainly combinations of empty fields that get checked during the authorise function) were left uncovered. This second publish file only checks said new branches
* **Show.safe:** We log in as manager1 and access many of our flights, both published and unpublished. We log in as manager2 and do the same. Not much more testing can be done on this functionality
* **Update.safe:** We log in as manager1 and access one of our flights, we then attempt to update with as many invalid forms as possible, similar to publish and create files. We finish with valid updates to test the positive path and the perform function
* **Create.hack:** We log in as manager1 and access the create form. We then use the inspect tool to assign a non-zero id to the hidden field (overwriting an existing flight), with an id with no flight associated, and invalid formats (“asdf”,”-5”,”as12345”,””). We also erase the entire field of the form. All attempts return non authorized exceptions. We don’t check for specific draft/non-draft owned/not-owned existing flights as the only valid id we accept during the authorise is 0. We also attempt to access this form as an administrator and as an unlogged user. We also try to add invalid values that can’t be obtained through the UI to different fields (same tests as with the id, but with airline, and also checking if the indication field has been tampered with (not “true” or “false”)
* **Delete.hack:** We log in as manager1 and access one of our unpublished flights. We then use the inspection tool to set the hidden id field to an id with no flight associated, invalid formats, the id of a published flight we own, the id of an unpublished flight from another user, and the id of a published flight of another user. We also tamper with the same fields as in create.hack. As this endpoint can only be accessed via other airline-manager features, we don’t test attempts at accessing it from wrong realms or unlogged users (these will be caught in the show feature, and by the framework)
* **List.hack:** We log in as administrator and try to access the endpoint. We do the same as an unlogged user. Both are caught by the framework.
* **Publish.hack**: We log in as manager1 and access one of our unpublished flights. We then follow the same steps as the delete.hack file as they share all authorization checks.
* **Show.hack:** We log in as manager1, and access one of our flights. We then change the id in the url to that of a non-existing flight, that of a flight that belongs to another user, and invalid formats (“asdf”,”-123”,””, deleting the entire field). We also do this as an administrator and as an unlogged user
* **Update.hack:** We log in as manager1 and access one of our unpublished flights. We then follow the same steps as the delete.hack and publish.hack file as they share all authorization checks.

**Leg**

* **Create.safe:** We log in as manager1 and attempt to create as many invalid legs as possible (empty fields, outside of range, invalid format, same arrival airport as departure airport, etc), and with valid fields (middle of range, edge of range). We finish with a valid create form to also get coverage and validate the perform method
* **Delete.safe:** We log in as manager1 and attempt to delete as many unpublished legs as we can. Aside from that, delete doesn’t hold much validation to test since it only needs a form that won’t flag the framework’s bind function. Said form can be obtained to get coverage on the unbind function by introducing invalid dates (“this is not a valid date”, for example) in the departure and arrival date fields, which will be flagged by the frameworks and trigger an unbind.
* **Empty-delete.safe:** In introducing more authorization checks, some happy paths were not checked properly (empty fields, mostly, as those can’t trigger any validation in the validate/bind functions). This file logs in as manager1 and deletes a completely emptied out leg to cover these conditionals.
* **List.safe:** We log in as manager1 and access our listed legs. We do the same as manager3 (empty list). No other testing can be performed.
* **List-flight.safe:** We log in as manager1 and access a few of our flights, then click on the List legs button for all of them (published flights, unpublished flights both with and without legs). No more testing is needed to cover all valid accesses to this feature.
* **Publish.safe:** We log in as manager1 and attempt to publish as many invalid legs as possible. Some entities have been prepared for this (Aircraft in Use, InvalidDraft1, InvalidDraft2), which trigger all validations that can’t be triggered within a single publish form. We then try empty fields, invalid data (out of range, invalid formats, empty fields). The testing of invalid fields is especially thorough as the validate checks for most of the fields before doing the more complex validations. We also alter the dates to trigger validations for publishing legs in the past, legs that finish before they start, legs that overlap with published legs of the same flight, loops in airports against published legs of the associated flights (arriving at an airport we departed from in the past, or departing from an airport we’ll arrive at in the future) and trying to use an aircraft that’s already scheduled to be in the sky during our leg. We also do some valid publishes to cover happy paths.
* **Show.safe:** We log in as manager1 and try to access most of our legs. No further testing is needed for this feature.
* **Update.safe:** We follow a similar behaviour as within the publish.safe, except for checking the more complex validations (airport loops within multiple legs, aircraft in use, overlapping with published legs of the same flight) as these are more restrictive on the edition of legs and will still be properly handled by the publish.
* **Create.hack:** We log in as manager1 and access the create form, we then try to tamper with all SelectChoices fields (by assigning flights that don’t belong to us, that don’t exist, that are already published and don’t belong to us, that are already published and belong to us, that are invalid numbers for ids (“asdf”,”-123”,””), and erasing the entire field altogether with the inspect tool. We also test by changing the hidden id field of the leg to a non-zero value and previously mentioned invalid numbers. We also try to access this form from the administrator account and an unlogged user, the framework handles these correctly.
* **Create2.hack:** Some of the illegal tamperings were missed during the first recorded create.hack, those are covered here.
* **Delete.hack**: We log in as manager1 and access one of our unpublished legs. Then with the inspect tool we attempt the same tamperings as in the create.hack, but also try to send delete requests with ids that belong to published legs that are ours and not ours, and to unpublished legs that are not ours.
* **List.hack:** We try to access the list functionality both as administrator and as an unlogged user, the framework handles it.
* **List-flight.hack:** We log in as manager1 and access the list of legs by one of our flights. We then attempt to edit the url to a flight id that doesn’t belong to us, both published and unpublished, to one that doesn’t exist, and to invalid id integers (“asdf”,”-123”,””). We also delete the flightId property from the url. All are properly handled. We then repeat these actions as administrator and an unlogged user, which are handled correctly by the framework.
* **Publish.hack:** We follow the exact same procedure as the delete.hack, since they share authorizations.
* **Show.hack:** We log in as manager1 and access one of our legs, we then try to change the url to unowned legs, both published and unpublished, invalid id integers, delete the field, and flights that don’t exists. We repeat these actions as administrator and as an unlogged user.
* **Update.hack:** We follow the exact same procedure as the delete.hack and publish.hack, since they share authorizations.
  1. **Performance testing:**

Testing before adding indexes

**A graph with orange and white text

Description automatically generated**

**A screenshot of a table

Description automatically generated**

**Testing after adding indexes**

**A graph with orange and black text

Description automatically generated**

**A screenshot of a table

Description automatically generated**

As we can see from the graphs, the slowest functions are the publish and update of the legs. This stands to reason as those are the ones that undergo most validations relying on database queries (alongside the create for legs, which is the third slowest feature although by a much smaller margin)

Also, that before adding indexes, requests took a bit longer to be answered by the server, on the order of 2 milliseconds or 10% faster after indexing.

The confidence intervals with 95% confidence were 22,44-25-74 milliseconds before adding indexes, down to 19,96-23,02 after adding them.

Contrast hypothesis

A table with numbers and a few words

Description automatically generated

With a two-tail p-value of 0,021 and a confidence level of .95, we are below the .05 p-value threshold for statistical significance, meaning we can make the claim that adding indexes did indeed make an improvement in our average response time compared to not having said indexes

**Conclusions**

This report holds all testing related information collected by student #1, which have served to spot many small yet important bugs in authorization efforts and to amend a badly implemented restriction that had previously gone unnoticed. It has also aided us in measuring the performance of the app before and after refactoring to introduce indexes, which has helped us demonstrate that said refactorization has indeed made an improvement in our response time.

**Bibliography**

Intentionally blank.