DP2 2023-2024

Testing Report

Acme Software Factory

Repository: https://github.com/rafcasceb/Acme-SF-D04

Student #5:

Vento Conesa, Adriana	adrvencon@alum.us.es

Other members:

Castillo Cebolla, Rafael	rafcasceb@alum.us.es
Flores de Francisco, Daniel	danflode@alum.us.es
Heras Pérez, Raúl	rauherper@alum.us.es
Mellado Díaz, Luis	luimeldia@alum.us.es

GROUP C1.049

Version 1.0

19-05-24

Content Table

Abstract

This report presents a comprehensive overview of the performance and functional testing conducted for deliverable D04 of the project. The report aims to provide clarity and insights into the testing process, including detailed descriptions of the testing procedures and outcomes, offering a transparent view of the project's testing phase.

Revision Table

Date	Version	Description of the changes	Deliverable
19/05/2024	V1	• Abstract. • Introduction. • Functional testing.	4
20/05/2024	V1	• Performance testing. • Conclusion.	4

Introduction

This document serves as a comprehensive analysis of the testing procedures and outcomes pertaining to the following operations within the scope of the obligatory tasks in this delivery:

- Operations by Auditors on Code Audits (Corresponding to requirement #6.)
- Operations by Auditors on Audit Records (Corresponding to requirement #7.)

The document is structured into two distinct chapters, each focusing on different aspects of the testing process:

1. **Functional Testing:** This chapter entails a detailed listing of implemented test cases, organized by the respective operations. Each test case is accompanied by a short description and an evaluation of its effectiveness in bug detection.

2. **Performance Testing:** In this section, relevant charts are provided to illustrate performance metrics. Additionally, a 95% confidence interval for the wall time taken by the project to handle requests in functional tests, along with a 95%-confidence hypothesis contrast, is presented to offer insights into performance reliability and efficiency.

Functional Testing

Operations by Auditors on Code Audits

1. Test case: list-mine.

For this case, our approach was straightforward: we initiated the process by listing all code audits belonging to an auditor across three different auditor accounts: two standard ones, and an account with no code audits.

Regarding security testing, we probed for vulnerabilities by attempting unauthorized access via manipulation of the URL. However, attempting access with valid credentials but incorrect user details was deemed impractical and thus not explored further, as the URL necessary for accessing these code audits will be the same one for all auditors.

We obtained a coverage rate of 94.4%, encompassing all instructions except for a default assertion. Notably, this test case revealed no detectable bugs.

2. Test case: show.

For this case, our approach was also straightforward: we conducted multiple iterations of listing code audits associated with an auditor, focusing on varied scenarios. Specifically, we performed this operation across different states of code audits, including both published and non-published audits. These tests were conducted using the account “auditor1” for consistency.

In terms of security testing, as well as performing the unauthorized access via manipulation of the URL, we expanded our evaluation by attempting unauthorized access with valid credentials but incorrect user details. Notably, this included attempting access to code audits from “auditor1” using the credentials of “auditor2”, simulating potential security breaches.

We obtained a coverage rate of 95.9%, encompassing all instructions except for default assertions. Notably, this test case revealed no detectable bugs.

3. Test case: create.

In executing the "create" command, our strategy centered on creating a new code audit with the peculiarity of testing each attribute with both valid and invalid data according to the restrictions defined by the client. This data included but was not limited to codes, dates, links, etc. An appropriate variability was taken into account when conducting the tests.

For security assessment, it's important to clarify that we were unable to register the POST requests for creating new code audits as part of our hacking test due to current framework limitations.

We obtained a coverage rate of 94.2%, encompassing all instructions except for default assertions. Notably, this test case revealed no detectable bugs.

4. Test case: update.

In executing the "update" command, our strategy centered on updating an already existing code audit by changing each attribute to both valid and invalid data according to the restrictions defined by the client in a systematic way. As such, the procedure was very similar to that of the “create” command, as the same data was used.

For security assessment, we performed two different tests: the first one concerning the modification of the read-only field “modeMark” which, if changed through the web inspection feature, and we subsequently try to update it, it shouldn’t perform the change. As it does not throw a panic, it was included in the “safe” files. Regardless, it passed the test. The second one pertains to updating an already published code audit. It should throw a panic instead of allowing it. It did, just as expected. However, we found that the panic thrown was not the expected "Access Unauthorized” one, but a “WorkArea” panic. This happened in the working computer of student five alone. Nevertheless, it was tested on other computers, and it did not allow us to perform the update as well, hence we can conclude that the test does pass.

We obtained a coverage rate of 93.8%, encompassing all instructions except for default assertions, as well as an assert for null code audits (which we cannot check through the browser). Notably, this test case revealed no detectable bugs, but dead code regarding the attribute publish was detected and consequently eliminated (that is, it is dead code as the implement authorized method will not permit an update on a published code audit before the validation can alert us of this same error).

5. Test case: delete.

For this case, our approach was also straightforward: we conducted multiple iterations of deleting code audits associated with “auditor1”. This scenario does not have any relevant restrictions; thus, many code audits were simply eliminated. Their corresponding audit records were deleted in cascade.

For security assessment, we could only perform one test: deleting an already published code audit. It should throw a panic instead of allowing it. The situation was just as the previous section detailed; hence, the test was passed.

We obtained a coverage rate of 84.6%, encompassing all instructions except for default assertions, as well as an assert for null code audits (which we cannot check through the browser). Notably, this test case revealed no detectable bugs, but dead code regarding the attribute publish and other(s) were detected and consequently eliminated.

6. Test case: publish.

In executing the "publish" command, our strategy was extremely like that of updating an already existing code audit, as the nature of both commands are the same, but “publish” sets the corresponding attribute to true. Therefore, the methodology was like the “update” test case, adding the following cases: trying to publish one with less than a C in “modeMark”, trying to publish with a C, trying to publish with no audit records, trying to publish one with one audit record, and trying to publish with many audit records. Other general valid publish operations were also performed to gather more data.

For security assessment, we performed two different tests: the first one concerning the modification of the read-only field “modeMark” which, if changed through the web inspection feature, and we subsequently try to update it, it shouldn’t perform the change. It passed the test. The second one pertains to publishing an already published code audit. The situation was just as the previous section(s) detailed; hence, the test was passed.

We obtained a coverage rate of 95.9%, encompassing all instructions except for default assertions, as well as an assert for null code audits (which we cannot check through the browser). There was also a line of code in which one of its branches was not covered, line 83, in which one of the branches is missing. This is simply because “modeMark” is an attribute that is not stored in the database; therefore, it is not present in the entity. As such, no errors will ever be evaluated in the buffer. For structure, it was left as is, as it is not a bug nor a potential bug. Notably, this test case revealed no detectable bugs, but dead code regarding the attribute publish was detected and consequently eliminated.

Operations by Auditors on Audit Records

1. Test case: list-mine.

For this case, our approach was straightforward: we initiated the process by listing all audit records belonging to an auditor across three different auditor accounts: two standard ones, and an account with no audit records.

Regarding security testing, we probed for vulnerabilities by attempting unauthorized access via manipulation of the URL. However, attempting access with valid credentials but incorrect user details was deemed impractical and thus not explored further, as the URL necessary for accessing these audit records will be the same one for all auditors.

We obtained a coverage rate of 92.6%, encompassing all instructions except for a default assertion. Notably, this test case revealed no detectable bugs.

2. Test case: list-for-code-audits.

This case, unlike the rest of the cases we will be analyzing in this section, is a new command. Nevertheless, our approach was similar to the previous one: we initiated the process by listing all code audits belonging to “auditor 1”. Then, we performed a show command to several code audits and then listed the audit records belonging to said code audit. This procedure was done several times, for both published and unpublished code audits (and therefore, for both published and unpublished audit records).

In terms of security testing, as well as performing the unauthorized access via manipulation of the URL, we expanded our evaluation by attempting unauthorized access with valid credentials but incorrect user details. Notably, this included attempting access to audit records from code audits from “auditor1” using the credentials of “auditor2”, simulating potential security breaches.

We obtained a coverage rate of 93.3%, encompassing all instructions except for a default assertion, as well as an assert for null code audits. This test case did in fact reveal a bug. We encountered that, when clicking on an already published code audit to show its audit records, the button did not appear, unlike when the code audit was unpublished. This could have also been detected in test case two for the previous feature, but it was noticed now that the button was needed to carry out the test. The error was simple to fix, as it only a looser restriction for said button in the “jsp” file of the form.

3. Test case: show.

For this case, our approach was also straightforward: we conducted multiple iterations of listing audit records associated with “auditor1”, focusing on varied scenarios, including both published and non-published records.

As in the previous case, we did both unauthorized access via URL manipulation and by attempting unauthorized access with valid credentials but incorrect user details. Notably, this included attempting access to code audits from “auditor1” using the credentials of “auditor2”, as in previous cases.

We obtained a coverage rate of 95.7%, encompassing all instructions except for default assertions, as well as an assert for null code audits. Notably, this test case revealed no detectable bugs.

4. Test case: create.

In executing the "create" command, our strategy centered on creating a new audit record focusing on the same aspect as in the other create operation: we tested each attribute with both valid and invalid data according to the restrictions defined by the client. This data included but was not limited to codes, dates, links, etc. An appropriate variability was considered when conducting the tests. It is also important to note that for this entity two aspects were taken into account: there existed two linked dates, which had to be tested together and not as separate attributes (although it was done to some extent) and the link of the audit record to a code audit, which would not be performed if the code audit was already published.

For security assessment, it's important to clarify that we were unable to register the POST requests for creating new audit records as part of our hacking test due to current framework limitations.

We obtained a coverage rate of 95.2%, encompassing all instructions except for default assertions. In the validate, some lines were not covered fully due to testing limitations. Notably, this test case revealed no detectable bugs, only dead code which was consequently eliminated (again, regarding the attribute published).

5. Test case: update.

In executing the "update" command, we updated an already existing code audit by changing each attribute to both valid and invalid data according to the restrictions defined by the client in a systematic way. As such, the procedure was very similar to that of the “create” command, as the same data was used.

For security assessment, we could only perform one test: updating an already published audit record. It should throw a panic instead of allowing it. The situation was just as the previous section(s) detailed; hence, the test was passed.

We obtained a coverage rate of 94.9%, encompassing all instructions except for default assertions, as well as an assert for null code audits (which we cannot check through the browser). In the validate, some lines were not covered fully due to testing limitations. Notably, this test case revealed no detectable bugs, but dead code regarding the attribute publish was detected and consequently eliminated (as explained in previous test cases).

6. Test case: delete.

For this case, our approach was also straightforward: we conducted multiple iterations of deleting audit records associated with “auditor1”. This scenario does not have any relevant restrictions; thus, many audit records were simply eliminated.

For security assessment, we could only perform one test: deleting an already published audit record. It should throw a panic instead of allowing it. The situation was just as the previous section(s) detailed; hence, the test was passed.

We obtained a coverage rate of 83.6%, encompassing all instructions except for default assertions, as well as an assert for null code audits (which we cannot check through the browser). Notably, this test case revealed no detectable bugs, but dead code regarding the attribute publish and other(s) were detected and consequently eliminated.

7. Test case: publish.

In executing the "publish" command, our strategy was extremely like that of updating an already existing audit record, as the nature of both commands are the same, but “publish” sets the corresponding attribute to true. Therefore, the methodology was like the “update” test case, and as there are no restrictions to publish for audit records, no extra cases were needed.

For security assessment, we could only perform one test: publishing an already published audit record. It should throw a panic instead of allowing it. The situation was just as the previous section(s) detailed; hence, the test was passed.

We obtained a coverage rate of 95.0%, encompassing all instructions except for default assertions, as well as an assert for null code audits (which we cannot check through the browser. In the validate, some lines were not covered fully due to testing limitations. No bugs were detected, only dead code (as explained in previous test cases).

Performance Testing

Let's start by analyzing the performance data from the functional testing. First, we'll analyze the results from the test replayer, and subsequently, we'll do a hypothesis contrast.

Performance Data

The tests were performed in a laptop with an 12th Gen Intel(R) Core(TM) i7-1260P 2.10 GHz CPU and 16GB of RAM.

After performing the corresponding tests, we obtain the following figure. It represents the average response time per request path, that is, the features outlined in the previous test cases.

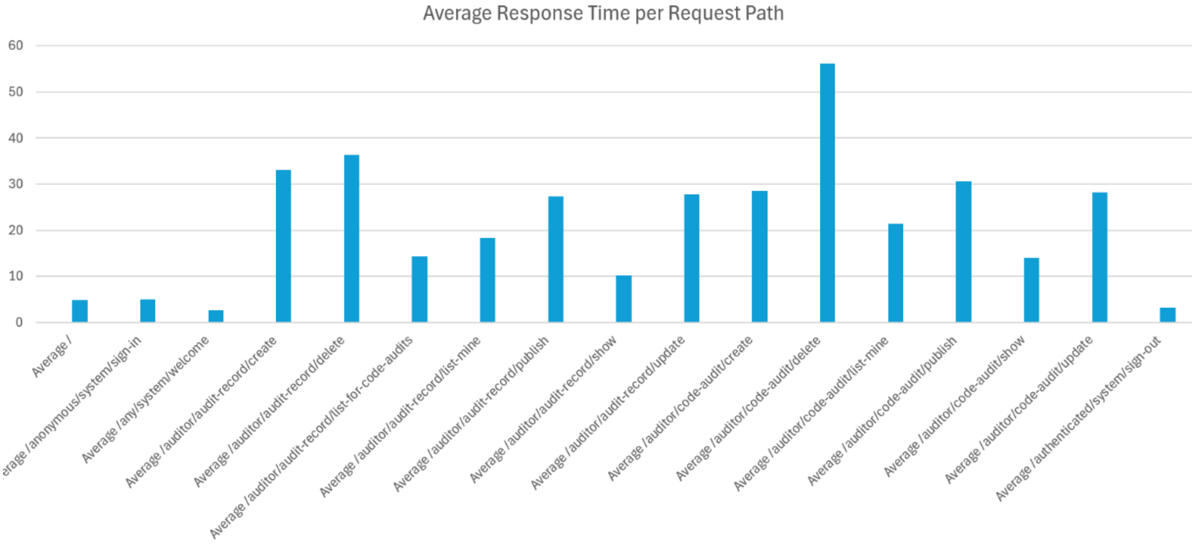


Figure 1. Average response time per request path.

As evident from our observations, the average response time across the request paths remains relatively consistent. The primary distinction lies in the uptick in response time for requests other than listings or shows. This variance is logical given the simplicity of these operations, which don't require data binding. It is also notable the increase in time for the operation “code-audit/delete”. After further inspection, we determined that this is due to the deleting in cascade that some of the tests required, as deleting an unpublished code audit will delete the audit records associated to said entity. Consequently, since the variance is readily explicable, there's no cause for concern in this graph.

Data Analysis	
Average	19,72620772
Standard error	0,460578182
Median	18,44035
Mode	27,9418
Standard deviation	13,36470592
Sample variance	178,6153644

Kurtosis	6,498873352
Asymmetry coefficient	1,498093366
Range	115,6903
Minimum	1,5438
Maximum	117,2341
Sum	16609,4669
Count	842
Confidence level (95.0%)	0,904017676

Figure 2. Data analysis computations.

After computing the confidence interval, we obtain the following range: [18.82, 20.63] in milliseconds. Transformed into seconds, we obtain: [0.018, 0.020]. These ranges were obtained by computing the lower and upper limit by summing and subtracting the confidence level to the average.

We do not have any performance expectations, but the computed ranges seem acceptable, as it is below one second.

Hypothesis Contrast

We will be increasing by 10% the data obtained in Figure 2 to simulate a hypothesis contrast. Therefore, we obtain:

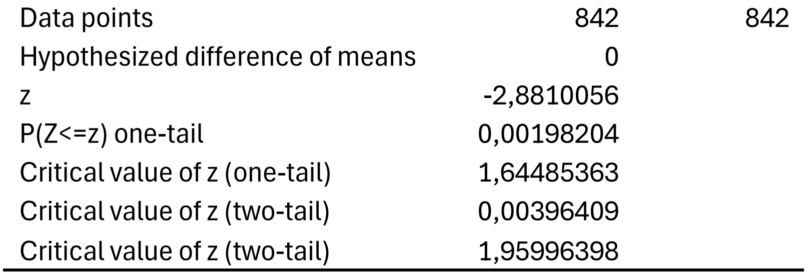
Data Analysis with 10% Increase	
Average	21,69882849
Standard error	0,506636
Median	20,284385
Mode	30,73598
Standard deviation	14,70117651
Sample variance	216,1245909
Kurtosis	6,498873352
Asymmetry coefficient	1,498093366
Range	127,25933
Minimum	1,69818
Maximum	128,95751
Sum	18270,41359
Count	842
Confidence level (95.0%)	0,994419444

Figure 3. Data analysis after 10% increase.

We obtain a new interval of: [20.70, 22.69] in milliseconds, and [0.020, 0.022] seconds.

We will now use a Z-Test to compare the results.

	time	*after*
Average	19,7262077	21,6988285
Variance (known)	178,615364	216,124591



Data points	842	842
Hypothesized difference of means	0	
z	-2,8810056	
P(Z<=z) one-tail	0,00198204	
Critical value of z (one-tail)	1,64485363	
Critical value of z (two-tail)	0,00396409	
Critical value of z (two-tail)	1,95996398	

Figure 4. Z-test for two sample means.

In a Z-Test, we use a significance level called alpha, which is calculated as 1 minus the confidence level percentage. So, if the confidence level is 95%, alpha would be 0.05.

To interpret the Z-Test result, we look at the p-value. This p-value is calculated assuming a two-tailed test. Now, we compare this two-tailed p-value to alpha. If the p-value is less than alpha, it means that the observed difference in averages is unlikely to have occurred by random chance alone, and we proceed to compare the averages themselves to see if the new average has decreased. Indeed, in our scenario, the critical value of z (two-tail) is approximately 0.003, which is significantly lower than our chosen alpha level. This indicates that the observed difference in averages is statistically significant.

Upon comparing the averages, we notice that the "time" (representing our original data) exhibits a notably better average than the "after" data. This outcome is quite expected since we implemented a 10% increase in time from the initial measurement. Consequently, it's evident that the performance was not improved.

Conclusions

In summary, the performance data obtained from the functional testing indicate that the system operates well within expected parameters, with response times largely consistent and fluctuations easily explained. Our systematic testing approach enabled us to detect and analyze potential issues, ensuring thorough scrutiny for bugs. While minor increases in response times were observed, particularly in simpler operations, these were anticipated and do not warrant concern.

Bibliography

Intentionally blank.