Apache Commons-email

Luigi Allocca, Simone Della Porta, Rocco Iuliano

ACM Reference Format:
Luigi Allocca, Simone Della Porta, Rocco Iuliano. 2023. Apache Commons-email. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The aim of this project is to analyze and improve an existing project by Apache foundation. We will improve the availability (i), reliability (ii), safety (iii), security (iv) and resilience (v) of the project in order to improve the dependability. We will focus on improve security by improving the coverage of the test cases and creating new ones, fix the bugs related to security and reduce security hotspot issues which refer to a specific area or component of a software system that has a higher risk of security vulnerabilities or breaches. These issues are often identified through a security analysis or review, and they can pose a significant threat to the overall security of the system. By removing these security hotspots, the overall security of the project will be improved, reducing the risk of security breaches and protecting sensitive user data. This can lead to increased trust from users and stakeholders, and can help to ensure that the project is compliant with relevant security regulations and standards. Overall the general goals are:

- fix as many project bugs as possible, prioritizing the crucial bugs;
- improve the coverage of project testing by developing new test cases and improving the existing ones;
- minimize the number of code smells;
- reduce security hotspot issues;
- verify the project performance.

2 PROJECT PRE REQUIREMENTS

We choose a project with the following characteristics in order to improve the dependability of the software with CI/CD paradigm using the tools introduced in the course:

- Git Actions to check code quality, coverage, Java CI and security.
- The project should use Maven to manage the project build, so it should have the pom.xml file.

3 CONTEXT OF THE PROJECT

Apache Commons-Email is an open source project and it is released in 01/08/2017 and it is available on GitHub to the following link: https://github.com/apache/commons-email. It aims to provide a API for sending email. Apache Commons-Email is built on top of the Java Mail API, which it aims to simplify. This project is composed by twentyfour contributors and has got 78 fork, 5 branches and 1132 commits of which the last commit was made in 18/03/2023 (id commit: f41dd76). Moreover, Apache Commons-Email has got three type of actions:

(1) **CodeQL** that allows us to verify the quality of the project;
(2) **Coverage** that allows us to verify the per cent of coverage of project test cases;
(3) **Java CI** that allows us to apply all the test cases to each commit that we make, so we are sure that the changes that we applied to the software doesn't introduce a bug.

Apache Commons-Email has got a user guide and a JavaDoc API documentation that are available to the following link:

- User guide: https://commons.apache.org/proper/commons-email/userguide.html
- JavaDoc: https://commons.apache.org/proper/commons-email/javadocs/api-release/index.html

Some popular Java-based applications that use Apache Commons Email are Apache Jenkins, Apache OFBiz, and Apache Syncope. Additionally, many Java-based web applications and enterprise systems use Apache Commons Email to handle email sending functionality.
To apply the changes to the project we must follow this rules:

(1) No tabs, instead use spaces for indentation.
(2) Respect the code style.
(3) Create minimal diffs - disable on save actions like reformat source code or organize imports. If you feel the source code should be reformatted create a separate PR for this change.
(4) Provide JUnit tests for your changes and make sure your changes don't break any existing tests by running mvn.

4 PRELIMINAR ANALISYS

After selecting the project Commons-email, we have created a fork of the repository, cloned the repository and built the project in order to run all test cases which result passed successfully. Then we performed a test push to verify the project actions and check the reults.

5 METHODOLOGICAL STEPS CONDUCTED TO ADDRESS THE GOALS

Then we have conducted a preliminar analisys of the project by using *sonarcloud* [1] and *codecov* [2] and we obtained this results:

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

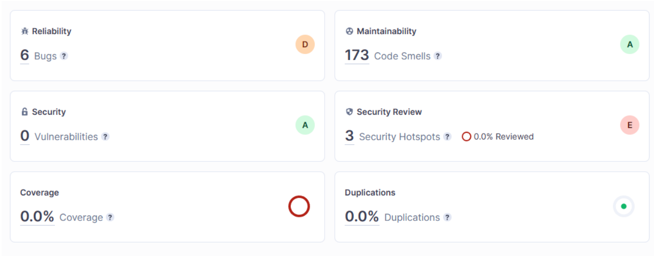


Figure 1: Sonarcloud analysis

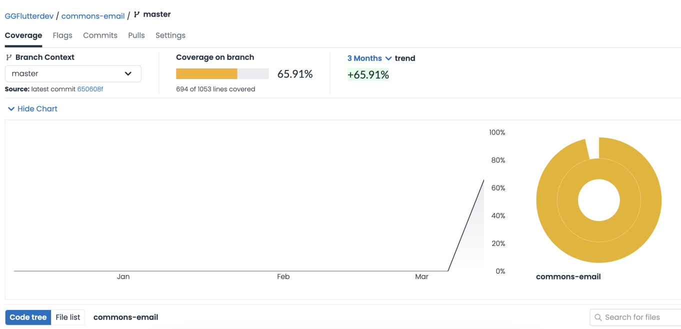


Figure 2: Codecov analysis

The result of the analysis are:
The coverage is 65,91%;
6 bugs, of which:

Kind of problem	Number	Severity	Topic
Bug	2	Major	Synchronization
Bug	2	Major	NullPointerException
Bug	2	Major	bad formatted try/ catch

Table 1: Bugs report

3 security hotspots of which:

Kind of problem	Number	Severity	Topic
Security hotspot	2	-	DoS attack
Security hotspot	1	-	Weak cryptography

Table 2: Security hotspots report

173 code smells, of witch:

Kind of problem	Number	Severity	Topic
Code Smell	11	Bloker	Adding at least one assertion in test cases
Code Smell	3	Critical	Try-catch implementation
Code Smell	5	Critical	Cognitive Complexity allowed
Code Smell	7	Critical	Duplicate literals
Code Smell	7	Major	Commented out code
Code Smell	42	Major	Assert Equals and Assert Not Equals
Code Smell	4	Major	Expressions that are always evaluated to `true`
Code Smell	4	Major	Lambda exceptions
Code Smell	7	Major	Use of a generic exeption instead of create and use a dedicated exception
Code Smell	3	Major	Testing
Code Smell	3	Major	Swap arguments
Code Smell	2	Major	Visibility of one constructor
Code Smell	2	Major	Try-catch implementation
Code Smell	1	Major	Number of assertions in a method
Code Smell	1	Major	Nested block of code
Code Smell	1	Major	Usage of a method instead of the usage of another method
Code Smell	1	Major	Usage of the utility of System class instead of the introdiction of a logger
Code Smell	14	Minor	Deprecated code
Code Smell	2	Minor	Chatset name argument
Code Smell	2	Minor	Hard-coded path-delimiter
Code Smell	2	Minor	Try-catch implementation
Code Smell	1	Minor	Return statement
Code Smell	48	Info	Deprecated code

Table 3: Code smells report

Then for having a continuous analisys of the project we decided to integrate these tools in the our project in order to analyze the code after every push or pull request.

5.1 Microservices and Docker

A **microservices architecture** improves the dependability of our project by providing greater fault tollerance, scalability, flexibility and error isolation. In order to achieve this, we have used **Docker** which is a platform that lets us to create, deploy, and run applications using containers. A container is used to run images, which we have created with a Dockerfile in the root directory of our project.

The Dockerfile code:

```
#We specified the maven version and jdk version that we
    want in our image
FROM maven:3.8.4-jdk-11-slim AS build

#We created a directory
WORKDIR /app

#We copied the project in the image
COPY src ./src

#We copied the pom file in the image
COPY pom.xml ./

#We copied the conf directory in the image
COPY conf ./conf

RUN mvn clean package
```

In order to share between the members of our team the image of the project we have used **Docker Hub** which is a web based service that allowed us to store, manage, and share docker images.

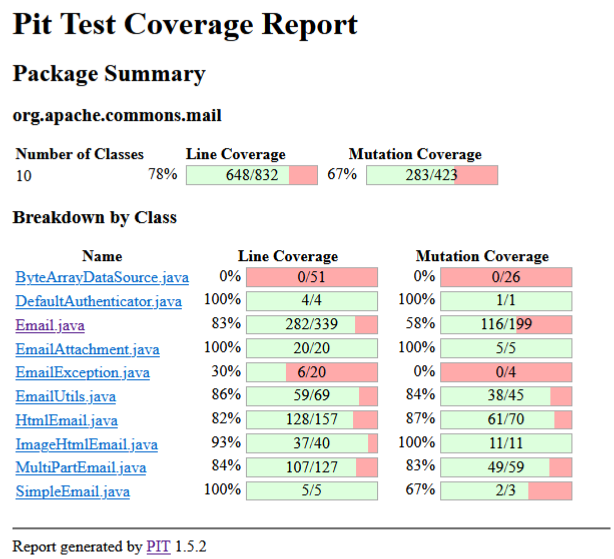
Then we have integrated Docker with **CI/CD**, through GitHub actions. Infact when we make a commit and the project builds successfully, the image will be updated on Docker Hub, and we all have a working and updated image of the project with the command *docker pull –platform linux/x86_64 ggflutter/commons-email* on linux x86_64 or *docker pull ggflutter/commons-email* on other platforms.

5.2 Test quality

For evaluating the test quality of commons-email, we used "Pitest". Pitest is a mutation testing tool that can be used to test the effectiveness of a suite of unit tests in detecting faults in code. It works by making small changes to the code and running the unit tests to verify if any of the changes cause the tests to fail. This tool helps us to identify gaps in unit test coverage, by highlighting areas of the code that are not being tested effectively. This can help to improve the overall quality of the codebase by identifying potential issues early in the development process. We used the default mutation criteria for creating mutants (these criteria are available at this link: https://pitest.org/quickstart/mutators/). We chose the Pitest version based on the JUnit, Jupiter and Java versions that our project use. The versions of these tools are:

- The JUnit version is 4.131.2;
- The Jupiter version is 5.9.1;
- The Java version is 11.0.12

For this reason, we used the Pitest version 1.5.2. The results that we obtained after we executed Pitest are shown in Figure 3.



Pit Test Coverage Report

Package Summary

org.apache.commons.mail

Number of Classes	Line Coverage		Mutation Coverage	
10	78%	648/832	67%	283/423

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ByteArrayDataSource.java	0%	0/51	0%	0/26
DefaultAuthenticator.java	100%	4/4	100%	1/1
Email.java	83%	282/339	58%	116/199
EmailAttachment.java	100%	20/20	100%	5/5
EmailException.java	30%	6/20	0%	0/4
EmailUtils.java	86%	59/69	84%	38/45
HtmlEmail.java	82%	128/157	87%	61/70
ImageHtmlEmail.java	93%	37/40	100%	11/11
MultiPartEmail.java	84%	107/127	83%	49/59
SimpleEmail.java	100%	5/5	67%	2/3

Report generated by PIT 1.5.2

Figure 3: Pitest report

5.3 Energy consumption analysis

Energy consumption analysis is an important activity to ensure the energy efficiency of the application and improve performance. In addition, energy consumption analysis can help ensure that the application is sustainable and has a reduced environmental impact. Also, users complain about the power consumption of their apps and power consumption affects user ratings on app stores.

5.3.1 Our analysis with EcoCode. Performing an energy consumption analysis of a Java application can help identify the parts of the code that consume the most energy. These parts of the code can then be optimized to reduce the overall consumption of the application. We performed our Analysis using SonarQube, and in particular, the plugin **ecoCode**. **ecoCode** is a collective project aiming to reduce the environmental footprint of software at the code level. The goal of the project is to provide a list of static code analyzers to highlight code structures that may have a negative ecological impact. The analysis gave us these results:

Topic	Number	Severity
Global variable	89	Minor
Unused variable	6	Minor
Use switch instead multiple if	70	Minor
Use AutoClosable interface	3	Minor
Use List instead Array in foreach	4	Minor
Use ++i	6	Minor
Initialize StringBuilder or StringBuffer size	3	Minor
Do not concatenate String in loop (with +)	1	Minor
Do not call a function in for declaration	1	Minor
Return the expression instead of assign it to a local variable	1	Minor

Table 4: Code smells founded by EcoCode

Test class name	Time (second)
DefaultAuthenticatorTest	0.025
EmailAttachmentTest	0.066
EmailLiveTest	0.268
EmailTest	0.269
EmailUtilsTest	0.01
HtmlEmailTest	1.55
ImageHtmlEmailTest	8.124
InvalidAddressTest	0.006
InvalidInternetAddressTest	0
MultiPartEmailTest	1.028
DataSourceClassPathResolverTest	0.024
DataSourceCompositeResolverTest	0.559
DataSourceFileResolverTest	0.01
DataSourceUrlResolverTest	0.755
SendWithAttachmentsTest	0.066
SimpleEmailTest	0.015
IDNEmailAddressConverterTest	0.007
MimeMessageParserTest	0.063

Table 5: Execution time needs for each class during the test.

All the elements of the "Use switch instead multiple if" class are false positives because the clauses of the series if regarding different variables or the check is not the equal operator. Moreover, the "Unused variable" class elements are also false positives because these code-smells regarding the caught exceptions that are necessary for the correct execution.

5.4 Benchmark testing

Benchmark testing helps measure and compare the performance of the application under different conditions. Furthermore, benchmark testing can help ensure that the application meets performance requirements and can handle expected levels of user traffic or data processing. By simulating different usage scenarios, benchmark testing can help identify potential scalability issues and ensure that the application can handle increasing levels of load. We made this by using **JMH** (the Java Microbenchmark Harness), adding the dependency in the *pom.xml* file.

5.4.1 Our test. For finding the classes with high computational cost we execute the following command: "*mvn package*", in order to visualize the classes that spend more time for testing. The results that we obtained are shown in Table 5. Based on these results, we applied the benchmark testing to these three classes: *HtmlEmail, ImageHtmlEmail and MultiPartEmail*. For each class we analyzed the same methods that JUnit tests have tested. The obtained results are shown in Table 6, Table 7 and Table 8. Depending on these results, we can affirm that the methods of the classes under testing took a short execution time therefore we did not apply any change.

6 TEST CASE GENERATION

In order to improve the test coverage in our project, we have used tools to automatically generate test cases on classes with lower coverage. We choose the classes based on the plot that is shown in Figure 4. Therefore the classes that we selected are: *MultiPartEmail, MimeMessageUtils, DataSourceFileResolver, DataSourceUrlResolver and HtmlEmail.* Automated tests for software are important because they verify software functionality precisely and efficiently

Method name	Average Time	Min Time	Max Time
embed(final URL url, final String name)	0.001	0.001	0.001
embed(final File file)	0.002	0.002	0.002
embed(final DataSource dataSource, final String name)	0.001	0.001	0.001
embed(final File file, final String cid)	0.002	0.002	0.002
send() inherit by Email.java	0.028	0.027	0.028
buildMimeMessage()	0.003	0.003	0.004

Table 6: Benchmark testing of HtmlEmail methods

Method name	Average Time	Min Time	Max Time
send() using simple text	1.302	1.227	1.391
send() using text with URL	3.030	1.749	3.721
send() using more images	3.330	1.656	4.387
buildMimeMessage()	0.042	0.037	0.051

Table 7: Benchmark testing of ImageHtmlEmail methods

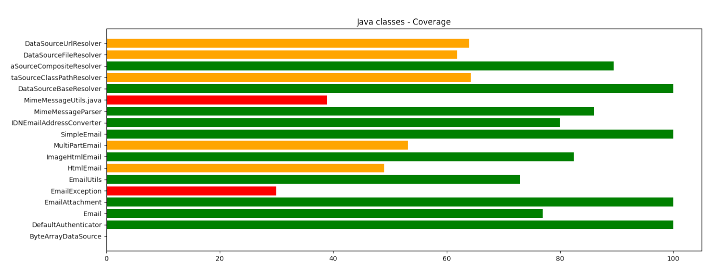


Figure 4: Class Coverage

Method name	Average Time	Min Time	Max Time
setMsg(final String msg)	$\approx 10^{-6}$	-	-
setMsg(final String msg) with charset	$\approx 10^{-6}$	-	-
send()	0.017	0.012	0.023
attach(final EmailAttachment attachment) with path	$\approx 10^{-4}$	-	-
attach(final EmailAttachment attachment) with url	$\approx 10^{-3}$	-	-
attach(final File file)	$\approx 10^{-4}$	-	-
addPart(final String s1, final String s2)	0.340	0.304	0.403
addPart(final MimeMultiPart)	0.371	0.291	0.416

Table 8: Benchmark testing of MultiPartEmail methods

and ensure greater test coverage than manual testing methods. In particular, we have used **EvoSuite** which generates test cases using a technique called search-based software testing. During the search process, Evosuite tracks the code coverage achieved by each test case and prioritizes the generation of new test cases that exercise previously untested code paths. The EvoSuite goal is to maximise the coverage of the class under testing then the goal function that we used is based on line coverage and branch coverage. The other EvoSuite parameters were set with default values. The GA process continues until a stop criterion, such as a maximum number of iterations or a desired coverage threshold, is met. In our case, we used the default stop criterion which is 60 seconds. Then Evosuite generates Java code for the test suite, which can be executed using a unit testing framework, in our case JUnit. After adding the test cases generated, we noticed an improvement in Total coverage by —% in CodeCov.

7 RESULTS AND FINDINGS

8 CONCLUSIONS