Dependability analysis of a Java open-source project: Apache Commons Email

Software Dependability Course,
Università degli Studi di Salerno
Francesco Di Lauro
f.dilauro5@studenti.unisa.it | Matricola: 0522501700

1 Introduction

Apache Commons Email is an open-source Java library that simplifies the creation and sending of emails, building on the JavaMail API to provide a more user-friendly interface. It allows developers to compose text and HTML emails, attach files, embed images, and handle mixed content with minimal effort. The library also supports secure email delivery through SSL and TLS, making it suitable for integrating reliable and secure email functionality into Java applications. Its design emphasizes simplicity and flexibility, catering to a wide range of use cases, from basic notifications to complex, multi-part messages.

The original repository of the open-source project is currently a monorepo, in which different distributions of the Apache Commons Email project can be found:

- commons-email2-javax: a distribution of the project for Java EE
- commons-email2-jakarta: a distribution for the newer Jakarta EE.
- commons-email2-core: an utility library shared between the two main distributions, containing exceptions, constants and utilities.

The shift from javax to jakarta occurred due to a trademark issue following the transfer of Java EE from Oracle to the Eclipse Foundation. Java EE was rebranded as Jakarta EE, with all APIs under the *javax.** namespace transitioning to *jakarta.**

While javax is tied to older versions of Java EE, jakarta supports the latest updates and innovations within the Jakarta EE platform.
The repository of this Software Dependability project is available at the following URL: GitHub repo

For this project, I chose to focus initially the analysis on the Core and Jakarta components.

CI/CD pipelines were already configured with GitHub Actions and I decided to restrict the Java version used for the builds in CI/CD to Java 17. For almost all the steps of the project, I created a different branch. At the end I merged all the branches (except for the mutation-coverage branch as its execution is too time-consuming) into main, in order to build the project in CI/CD.

2 Code Quality Inspection

In the first part of this project, a complete analysis of the system was conducted with **SonarCloud** in order to identify and resolve issues that could lead to bugs, vulnerabilities, or decreased development velocity. Initially, I attempted to set up the SonarQube Cloud's CI-based analysis by creating a new GitHub workflow, but problems occurred during the build, possibly due to the project's monorepo structure. Consequently, I was forced to use SonarCloud automatic analysis. The Analysis and Refactoring were performed by setting up an exclusion of the javax component from the scope settings of SonarCloud.

2.1 Analysis

The set of rules applied during the project's analysis was the default "*Sonar way*" profile provided by SonarCloud. SonarCloud categorizes issues into three main software quality metrics: Security, Reliability, and Maintainability. For each impacted metric, SonarCloud assigns a severity level ranging across five levels: blocker, high, medium, low and info. The analysis performed by SonarCloud on the project identified a total of **539 issues**:

- **0 Security issues**
- **6 Reliability issues** that can be categorized based on their severity level:
  - **4 High severity issues** related to some test cases with multiple method invocations throwing the same checked exception.
  - **2 Medium severity issues** related to synchronization on a method parameter instead of a final class field.
- **533 Maintainability issues** categorized by severity:
  - **24 Blocker severity issues** due to missing assertions in some test cases.
  - **19 High severity issues** involving excessive cognitive complexity in methods and duplications of literals instead of defining constants.
  - **51 Medium severity issues** due to the usage of standard output instead of a more robust type of logging, public visibility of abstract classes instead of protected, identical implementations across methods, catching Throwable instead of Exception, nested Try-catch blocks, presence of commented-out lines of code, excessive assertion in some test cases, similar methods that could be grouped into a single parameterized one, multiple runtime exception invocations in a single lambda expression

within test cases, usage of @Disabled annotation on test cases without providing explanation, incorrect order of expected value and actual value in assertion, and local variable shadowing class fields.

  - **28 Low severity issues** related to the usage of non-specialized functional interfaces, the usage of the Boolean wrapper class instead of a primitive boolean expression in conditional statements, incompleted TODO tasks, hard-coded path delimiters in URIs, declared exceptions not thrown by method bodies, and field names not compliant with Java naming conventions.
  - **411 info issues**, all related to functional test cases with public modifiers instead of default package visibility.

SonarCloud also found **8 Security Hotspots**:

- **4 issues** concerning the usage of regular expressions that, due to their backtracking property, could lead to denial-of-service in case of polynomial runtime during the evaluation of strings. These can be mitigated by restricting the input size and setting a timeout for string evaluation.
- **1 issue** concerning the usage of a non-cryptographic pseudorandomic number generator.
- **3 issues** concerning the usage of the HTTP protocol instead of HTTPS to retrieve an image from the web in some test cases.

2.2 Refactoring

After the analysis and identification of the issues, I decided to refactor the most significant issues (based on SonarCloud severities, Blocker and High) to enhance the software's overall quality and code readability. To facilitate this process, I created a separated branch named *refactoring*, in order to distinguish the original code from the modified version and effectively compare the results of the analyses (as in the microbenchmark section).
All of the issues with **Blocker** severity, were due to missing assertion in some test cases (the goal of those tests was to ensure that no exceptions are thrown), and I skipped all of them.

The **Higher** severity issues found concerned the excessive Cognitive Complexity and duplications of literals instead of defining constants, problems that affect the readability of the code. The Cognitive Complexity problem was due to nested if statements in the code and several conditions inside if statements. I had to understand the logic behind the methods that included this problem, and I extracted the complex conditions into new methods, in order to break down large functions. To fix the duplicated literals problem, I introduced constants.

3 Containerization

After analyzing the project with SonarCloud, the next step was to prepare a Docker image, available on **Docker Hub**, that could be executed in a container. For this, an executable JAR file with a main method as the entrypoint for the application was

required. Since Apache Commons Email is a library, I created a web application using the **Spring Boot** (v3.3.6) framework. Additionally, I used the **Vaadin**(v24.5.5) framework, which allowed me to create a simple UI in Java with server-side rendering.
I started by creating a new branch, *springboot-web-app*, and a corresponding springboot module in the root directory of the project. I included both *commons-email2-core* and *commons-email2-jakarta* as dependencies in this new module. A key concept for this step was the `mvn install` command which I used to install the locally generated Commons Email dependencies into the local Maven repository (usually *∼/.m2/repository*). This ensured that the Spring Boot project's build process, which relies on the apache-commons-email2 dependencies declared in the pom.xml, used the local changes instead of resolving them from the remote Maven Central repository.

Another important aspect was building Vaadin in production mode in order to avoid runtime errors caused by missing resource files, which are only generated in production builds.

To emphasize these steps, I integrated them into the Docker image creation process. The Dockerfile added to the root directory of the project includes the following instructions:

- `FROM maven:3.8.3-openjdk-17` : This command specifies the base image as maven:3.8.3-openjdk-17, a base Docker image that includes the JDK 17 and Maven already installed.
- `WORKDIR /app` : This command creates and sets the working directory of the application inside the container to */app*.
- `COPY pom.xml .` : This command copies the *pom.xml* file from the project directory of the host machine to the */app* directory inside the container. This step is necessary because the *pom.xml* file of the root project is actually the parent one. Without it, trying to build the components like jakarta or core would result in a failure.
- `COPY commons-email2-core ./commons-email2-core`
- `COPY commons-email2-jakarta ./commons-email2-jakarta`
- `COPY springboot-email ./springboot-email` : These commands perform a copy of the modules directories from the project of the host machine to the */app* directory inside the container.
- `RUN mvn clean install -f ./commons-email2-core/pom.xml`
- `RUN mvn clean install -f ./commons-email2-jakarta/pom.xml` : These commands runs the Maven command *mvn clean install* inside the container which, as mentioned earlier, builds the project and creates a JAR file in the */target* directories of each module, also installing the dependencies on the local Maven repository of the image.
- `RUN mvn clean package -Pproduction -f ./springboot-email/pom.xml`: This command runs the Maven command *mvn clean package* inside the container, which builds the Spring Boot project in production mode and creates a JAR file in the */springboot-email/target* directory.

- `RUN cp ./springboot-email/target/ springboot-email-0.0.1-SNAPSHOT.jar app.jar`: This command copies the freshly generated jar of the springboot application inside the container, to the */app* folder of the image, with the name *app.jar*
- `EXPOSE 8080`: This command specifies that the container will listen on port 8080 for requests when it is running.
- `ENTRYPOINT ["java", "-jar", "app.jar"]`: Sets the command to be executed when the container starts. In this case, it runs the JAR file located in the */app* directory of the image using the *java -jar* command.

To build the Docker image, I navigated to the root directory of the project where the Dockerfile is located, and ran the following command in the terminal: `docker build -t commons-email2-demo .`
This command creates an image named commons-email2-demo in approximately 2 minutes. Next, to generate and run the container, I executed:
`docker run -td -p 8080:8080 -name SwDProject commons-email2-demo`
This created and started a container named SwDProject, mapping port 8080 on both the host machine and the container for handling requests.

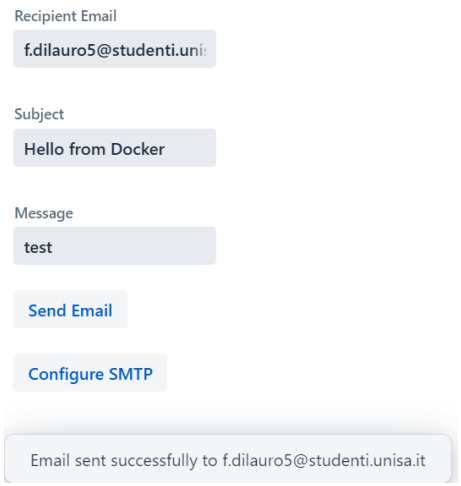


Figure 1: Interface of the Spring Boot web application running inside Docker.

In the final step, to deploy the image on **Docker Hub**, I configured a GitHub Actions workflow. Using GitHub secrets to store the Docker Hub username and password, I created a *docker.yml* file inside *.github/workflows* directory with the following commands:

```
docker login -u $username -p $password
docker build -t ${secrets.DOCKER_USERNAME}/commons-email2-demo:latest .
docker push ${secrets.DOCKER_USERNAME}/commons-email2-demo:latest
```

This workflow builds a new Docker image whenever a push is made to the springboot-web-app branch and automatically uploads it to the Docker Hub repository.

4 Code Coverage

From now on, the only project analyzed will be commons-email2-jakarta. Code Coverage for the Apache Commons Email project is computed by default with **JaCoCo**, which is embedded in the Github Actions workflow. Each module has a *src/site/resources/profile.jacoco* file that triggers the JaCoCo profile from the parent POM. The coverage configuration for each module is specified in its *pom.xml* file.

To upload the code coverage data generated by JaCoCo to Codecov, I created a new GitHub workflow and configured the Jakarta component to generate the report in the */target/site/jacoco directory*. The overall Code Coverage calculated is approximately 78.99% . As shown in Figure 2, the least-covered classes are inside the resolver package.

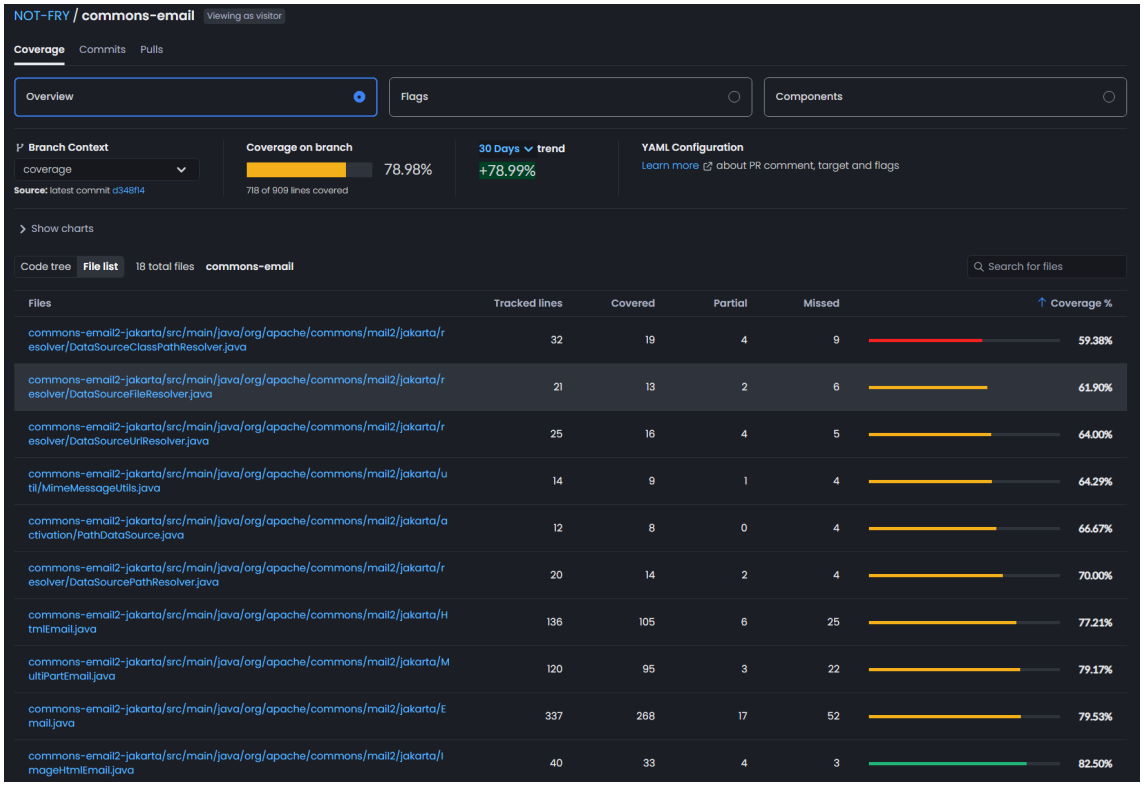


Figure 2: Codecov report.

4.1 Mutation testing

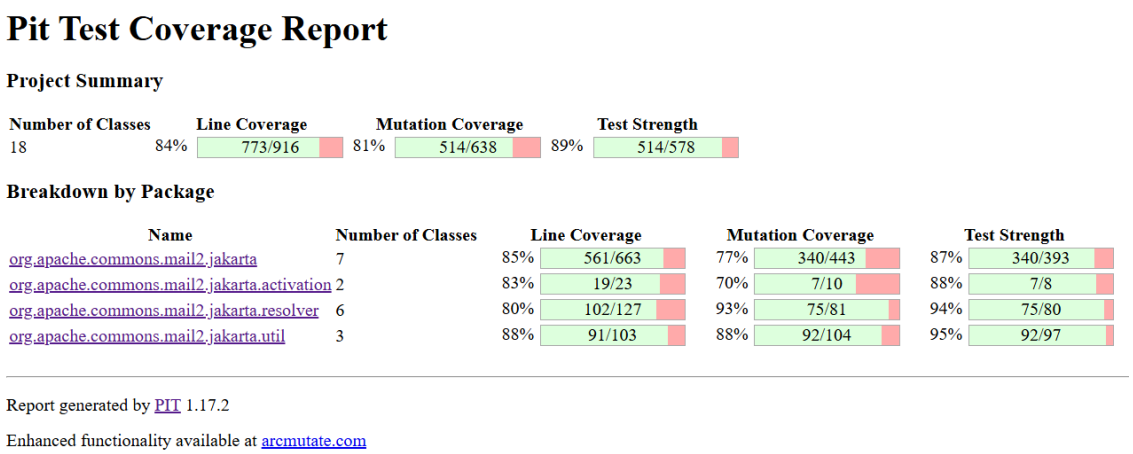
For the Mutation Testing campaign, I used the **PiTest** (v1.17.2)Maven plugin, integrating it in the project alongside the Junit 5 plugin for PiTest. The PiTest mutators used were part of the "*Stronger*" *group*, including:

- Conditional boundary mutators
- Increment mutators
- Invert negatives mutators
- Arithmetic operations mutators
- Negate conditionals mutators
- Return values mutators
- Remove conditionals mutators
- Switch mutators

The *pom.xml* file of the Jakarta module was modified to ensure that the mutation test are executed during the test phase of the Maven build lifecycle. As a result, running the `mvn test` command triggers the mutation coverage performed by Pitest after all standard defined test cases have been executed.
PiTest required approximately 9 hours to generate and execute the mutations on 13 threads, highlighting the performance overhead of mutation testing for larger projects.

In Figure 3, the results of the analysis are presented, showing three key metrics calculated for the entire jakarta module and for each subpackage.

- **Line Coverage** represents the measure of how many code lines are indeed covered by tests. The project achieved a score of 84% for mutated classes.
- **Mutation Coverage** evaluates the percentage of generated mutants that were successfully eliminated. With 81% of mutants "killed", the results suggest that a significant portion of mutations triggered test failures.
- **Test Strength** focuses on the percentage of mutants eliminated among those covered by tests. Since surviving mutants due to insufficient coverage are excluded from this metric, in this case the mutations with no coverage resulted to be 60 and the project attained a result of 89%.



Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
18	84%	773/916	81%	514/638	89%	514/578

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
org.apache.commons.mail2.jakarta	7	85%	561/663	77%	340/443	87%	340/393
org.apache.commons.mail2.jakarta.activation	2	83%	19/23	70%	7/10	88%	7/8
org.apache.commons.mail2.jakarta.resolver	6	80%	102/127	93%	75/81	94%	75/80
org.apache.commons.mail2.jakarta.util	3	88%	91/103	88%	92/104	95%	92/97

Report generated by PIT 1.17.2

Enhanced functionality available at arcmutate.com

Figure 3: PiTest report.

5 Automated Test Case Generation

In this section I will describe the automated generation of test cases I performed to cover poorly tested code components. I decided to apply the automatic generation of tests to the poorly tested classes identified in section 4: *DataSourceClassPathResolver, DataSourceFileResolver, DataSourceUrlResolver, DataSourceCompositeResolver, DataSourcePathResolver*. I used **Randoop** (v4.3.2) to automatically generate unit tests for the mentioned class. I ran the following command:

```
java -cp "./randoop-all-4.3.2.jar:./target/classes:
    ./jakarta.activation-api-2.1.3.jar" randoop.main.Main gentests
    --classlist=classesToTest.txt --time-limit=20
    --junit-output-dir=randoop-tests
```

I had to add the *jakarta.activation* jar to the Java classpath because there were dependencies on this package. I set the time limit to 20. Randoop generated 72 test cases in the randoop-tests directory.
Next, I looked up for dependencies used by the project named junit, junit-jupiter, hamcrest, or hamcrest-core in order to compile the generated tests :

```
mvn dependency:copy-dependencies -DincludeArtifactIds=junit,
    junit-jupiter,hamcrest,hamcrest-core
```

I compiled the generated tests, by including the JUnit (v5.11.2) JAR, which was copied by the previous command to the /target/dependencies directory, in the classpath:

```
javac $(find randoop-tests -name "*.java")
    -cp ./target/classes:./randoop-all-4.3.2.jar:
    ./junit-jupiter5.11.2.jar:./jakarta.activation-api-2.1.3.jar
```

Finally, I executed the generated tests using JUnit (v1.9.2) launcher:

```
java -cp randoop-tests:./target/classes:./randoop-all-4.3.2.jar:
    ./jakarta.activation-api-2.1.3.jar:
    ./junit-platform-console-standalone-1.9.2.jar
    org.junit.platform.console.ConsoleLauncher -scan-class-path
```

All generated Tests ran successfully. I added these test cases generated by Randoop into the default Test directory and calculated the new code coverage with Codecov, which reached approximately **81.07%**.

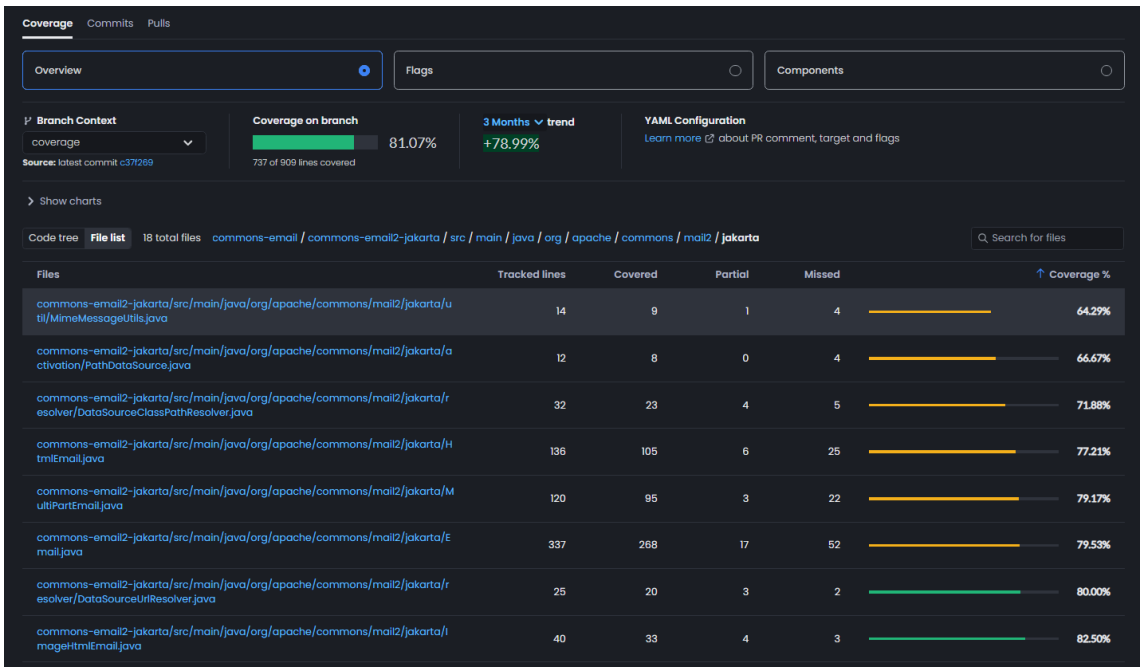


Figure 4: Code Coverage after adding Randoop's generated test cases.

I also used **GitHub Copilot** to generate test cases for the classes *Email and HtmlEmail*, further increasing the Code Coverage to **81.51%**.

6 Performance testing

Performance testing is a critical aspect of software development, aimed at evaluating the speed, scalability, and stability of an application under different conditions. Microbenchmarks are particularly useful for assessing the performance of specific, isolated operations or functions. These tests focus on extremely small workloads, typically measured in microseconds or nanoseconds, providing precise insights into the performance of critical code paths.

One of the critical aspects of microbenchmarking is the execution environment: operating system, background processes, memory capabilities, and others factors. This implies that different tests on the same component, must be performed under identical conditions. Below are the specifications of the execution environment on which the microbenchmarks tests were conducted:

- *OS: Windows 11*
- *CPU: Intel i5-13600k*
- *RAM: 32GB*
- *JDK version : 17*
- *JMH version: 1.37*

In this section, I used the **JMH** (Java Microbenchmark Harness) framework. I included the *jmh-core* and *jmh-generator-annprocess* dependencies inside the *pom.xml* file, along with the Maven Shade plugin, which creates an executable JAR.

Then, I created benchmark methods *inside the src/main/java* folder and defined the configurations for the microbenchmarks. I chose to run the benchmarks on a single Thread to emulate a system that build and send one email at a time. I also specified that each benchmark should be executed with 5 warmup iterations and 1 fork. For the tests I measured 10 iterations. I did not include any *@Setup* method in the benchmark because I wanted to include the instatiation cost in the test. I executed the tests in *AverageTime* mode, in this way JMH calculates the average time it takes for the benchmark method to be executed (e.g. the time it takes to prepare an email). I set the JMH output time unit to be expressed in Milliseconds (even though for the *addRecipientsBenchmark* the time could be expressed in Nanoseconds). I avoided sending real emails in the tests, to eliminate the bias caused by the delay of a real SMTP server. I noticed in fact that tests performed in the *EmailLiveTest* class made use of real recipients and real mail servers. I attempted to use a dummy SMTP server (like Wiser or GreenMail) to emulate the email-sending process (as in some project's test cases), but dependency conflicts arose, so I opted to call the *buildMimeMessage()* method at the end of the benchmarks to closely emulate sending of an email.

I created 4 benchmarks:

- *addRecipientsBenchmark*: A simple benchmark that tests for the time it takes to add different type of recipient during email building. This test took an average of **937 ns** to be executed.
- *simpleEmailBenchmark*: A simple benchmark that tests for the time it takes to build an email as a whole(SMTP server, Subject, From and Recipient Fields) and at the end calls the *buildMimeMessage()* method to emulate as close as possible the sending of an email. This test took on average **0.223 ms** to be executed.
- *addAttachmentBenchmark*: Similar to *simpleEmailBenchmark* but with an image attachment. This test took on average **0.252 ms** to be executed.
- *htmlEmailBenchmark*: A benchmark of an email with an embedded HTML message. This test took on average **0.227 ms** to be executed.

To build the project, I had to lower the JaCoCo code coverage limits, because I included the benchmark methods inside the *src/main/java* source folder. Then I run the following commands:

`mvn clean install`: to build the project inside the *target* directory

`java -jar .\target\commons-email2-jakarta-2.0.0-M2-SNAPSHOT.jar -rf json`: to run the benchmarks and generate a JSON report.

I also performed the microbenchmarks on the refactored code (section 2.2), to detect any performance losses, the results were comparable to those obtained with the original code.

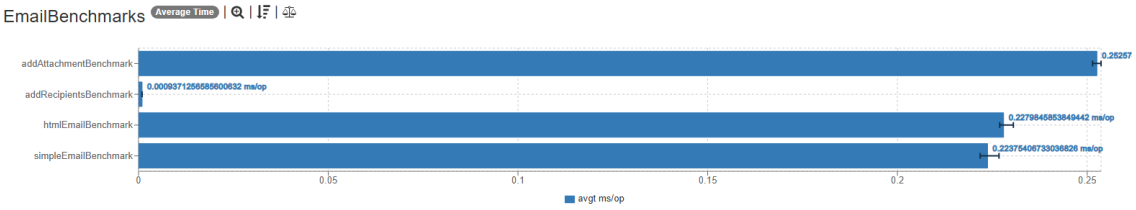


Figure 5: Microbenchmark results.

Benchmark	Mode	Cnt	Score	Error	Units
EmailBenchmarks.addAttachmentBenchmark	avgt	10	0,250 ±	0,002	ms/op
EmailBenchmarks.addRecipientsBenchmark	avgt	10	0,001 ±	0,001	ms/op
EmailBenchmarks.htmlEmailBenchmark	avgt	10	0,226 ±	0,002	ms/op
EmailBenchmarks.simpleEmailBenchmark	avgt	10	0,220 ±	0,001	ms/op

Figure 6: Microbenchmark results on refactored code.

7 Security

In this last section, I'm going to analyze the security vulnerabilities of this project with the help of two tools: **OWASP FindSecBugs** to identify security bugs and potential vulnerabilities and **OWASP DC** to detect dependency versions with known vulnerabilities.

7.1 OWASP FindSecBugs

I used the tool find-sec-bugs (v1.12.0) in order to finds potential vulnerabilities, by running the following command:

```
.\findsecbugs.bat -progress -html -output report.html -effort:max
    -exclude myexclude.xml
    C:\...\commons-email\commons-email2-jakarta\target\classes
```

I excluded from the analysis all the test cases by adding a filter inside myexclude.xml. By setting the effort parameter to max, find-sec-bugs applies a more precise and time consuming analysis, in order to reduce the rate of false positives.

The analysis reported a total of 9 medium-priority security warnings, with a density of 6.91 defects per 1000 lines of code. FindSecBugs categorizes these warning under different codes:

- **SECLDAPI** (1 warning) related to a vulnerability to LDAP injection inside the Email.setMailSessionFromJNDI method. An LDAP (Lightweight Directory Access Protocol) query is vulnerable, just like SQL, to LDAP injection. However, for this protocol, prepared statements doesn't exist, so a strong input validation is needed before including any untrusted data inside the query.

- **SECPTI** (6 warnings), potential vulnerabilities related to the presence of APIs (e.g. java.io.File) that read files whose location might be specified by user input: if the parameter is not filtered, files from an arbitrary filesystem location could be read. However, in many cases, the constructed file path cannot be controlled by the user making these instances a false positive.

- **SECSSSRFUC** (2 warnings), a vulnerability related to Server-Side Request Forgery, that occurs when a web server execute a request to a user-supplied destination parameter that is not validated. This could allow an attacker to access internal services.

Security Warnings

Code	Warning
SECLDAPI	This use of javax/naming/Context.lookup(Ljava/lang/String;)Ljava/lang/Object; can be vulnerable to LDAP injection
SECPTI	This API (java/io/File.<init>(Ljava/lang/String;)V) reads a file whose location might be specified by user input
SECPTI	This API (java/io/File.<init>(Ljava/lang/String;)V) reads a file whose location might be specified by user input
SECPTI	This API (java/io/File.<init>(Ljava/io/File;Ljava/lang/String;)V) reads a file whose location might be specified by user input
SECPTI	This API (java/io/File.<init>(Ljava/lang/String;)V) reads a file whose location might be specified by user input
SECPTI	This API (java/nio/file/Paths.get(Ljava/lang/String;[Ljava/lang/String;)Ljava/nio/file/Path;) reads a file whose location might be specified by user input
SECPTI	This API (java/nio/file/Paths.get(Ljava/lang/String;[Ljava/lang/String;)Ljava/nio/file/Path;) reads a file whose location might be specified by user input
SECSSSRFUC	This web server request could be used by an attacker to expose internal services and filesystem.
SECSSSRFUC	This web server request could be used by an attacker to expose internal services and filesystem.

Figure 7: FindSecBugs warnings.

7.2 OWASP Dependency Check

I used OWASP DC (v8.2.1) to analyze the dependency and potential vulnerabilities of the apache-commons-email project by running the following command:

```
.\dependency-check.bat -s C:\...\commons-email\commons-email2-jakarta
    \target --exclude "**\*-sources.jar" --exclude "**\*-tests.jar"
    --exclude "**\junit*.jar" -exclude "**\hamcrest*.jar"
```

No dependency vulnerabilities were found in the project.