[CS304] Lab8. Testing with JUnit and JaCoCo

> Author: Yida Tao

In this tutorial, we'll learn about the basics of JUnit testing. We'll also use `Teedy` to demonstrate common testing practices using `maven`, `JUnit`, and test coverage tools.

Getting Started with JUnit

JUnit is essentially a dependency to your project, which could be downloaded and managed using Maven.

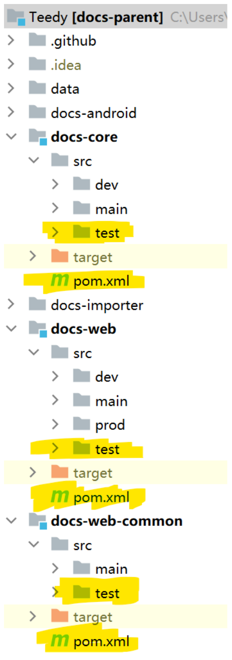
You may refer to this official guide of IntelliJ IDEA to create a Maven project and add JUnit dependency in `pom.xml`.

```
<dependencies>
      <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.7.1</version>
      </dependency>
</dependencies>
```

Follow this official guide to create application code, generate tests, and execute the tests to observe the results.

Examining Tests in Teedy

Teedy has 3 modules `docs-core`, `docs-web-common`, and `docs-web`, each can be built and tested independently. You may observe the JUnit dependency in `pom.xml` of any module, and observe the test cases written for any of the modules.



Running Teedy Tests

In previous labs, we skipped tests when building Teedy using `mvn clean -DskipTests install`. You could simply remove the `-DskipTests` option if you want to run tests in building.

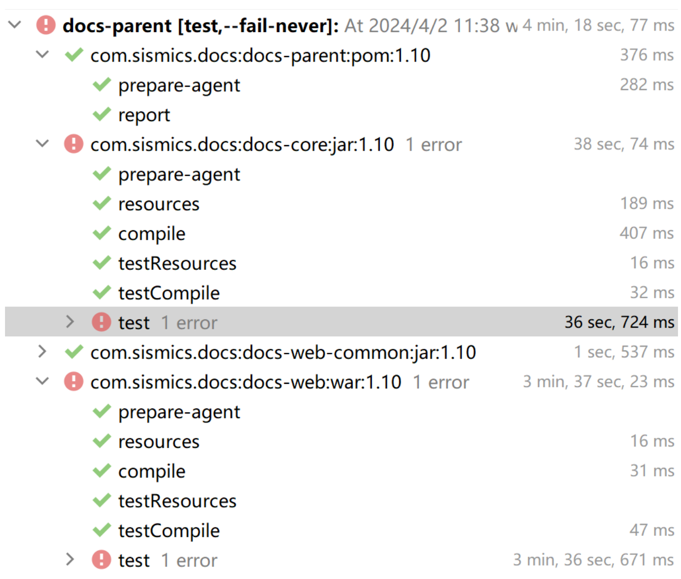
Alternatively, you could run `mvn test` to execute all unit tests in the project. Here, the `test` phase uses the `Surefire Plugin` to execute tests, which by default automatically include all test classes with the following wildcard patterns:

- `**/Test*.java`
- `**/*Test.java`
- `**/*Tests.java`
- `**/*TestCase.java`

If the test classes do not follow the default wildcard patterns, then override them by configuring the Surefire Plugin and specify the tests you want to include (or exclude) or another patterns.

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0</version>
        <configuration>
          <includes>
            <include>Sample.java</include>
          </includes>
          <excludes>
            <exclude>**/TestCircle.java</exclude>
            <exclude>**/TestSquare.java</exclude>
          </excludes>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Use `mvn test --fail-never` so that the testing continues even if certain test cases fail. (Depending on your version of Teedy, the test result may be different from below):



```
docs-parent [test,--fail-never]: At 2024/4/2 11:38 w 4 min, 18 sec, 77 ms
  com.sismics.docs:docs-parent:pom:1.10                       376 ms
    prepare-agent                                             282 ms
    report
  com.sismics.docs:docs-core:jar:1.10  1 error                38 sec, 74 ms
    prepare-agent
    resources                                                 189 ms
    compile                                                   407 ms
    testResources                                             16 ms
    testCompile                                               32 ms
    test  1 error                                             36 sec, 724 ms
  com.sismics.docs:docs-web-common:jar:1.10                   1 sec, 537 ms
  com.sismics.docs:docs-web:war:1.10  1 error                 3 min, 37 sec, 23 ms
    prepare-agent
    resources                                                 16 ms
    compile                                                   31 ms
    testResources
    testCompile                                               47 ms
    test  1 error                                             3 min, 36 sec, 671 ms
```

Running all tests may take a long time. Sometimes you may want to run only a few interesting test classes or test methods. In that case, you could run:

`mvn -Dtest=TestCss test`

`mvn -Dtest=TestCss,TestImageUtil test`

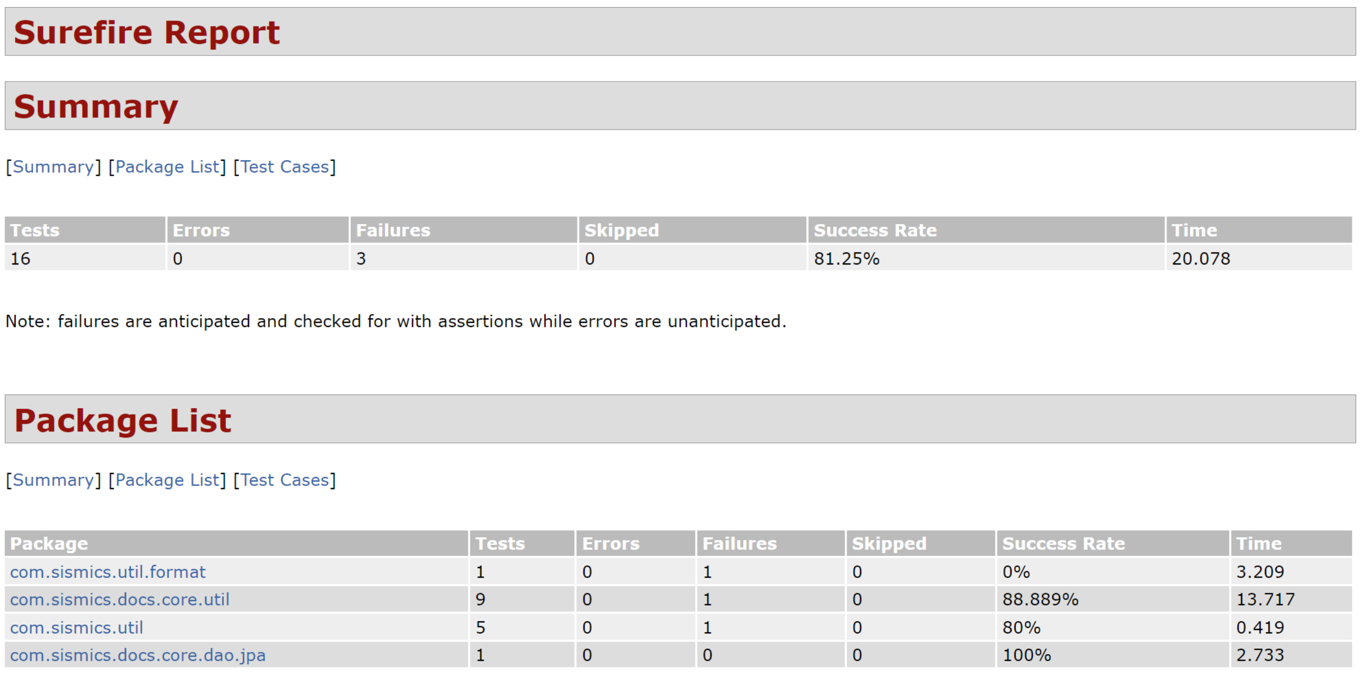
`mvn -Dtest=TestEncryptUtil#encryptStreamTest+decryptStreamTest test`

See here for detailed syntax on running single test.

Checking Test Report

If you want to get easy access to test report, run `mvn surefire-report:report` which generates report in html format in `target/site/surefire-report.html`. Note that **you have to execute tests first before you could generate report**.

You could open the report in a browser for examination.



Surefire Report

Summary

[Summary] [Package List] [Test Cases]

Tests	Errors	Failures	Skipped	Success Rate	Time
16	0	3	0	81.25%	20.078

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary] [Package List] [Test Cases]

Package	Tests	Errors	Failures	Skipped	Success Rate	Time
com.sismics.util.format	1	0	1	0	0%	3.209
com.sismics.docs.core.util	9	0	1	0	88.889%	13.717
com.sismics.util	5	0	1	0	80%	0.419
com.sismics.docs.core.dao.jpa	1	0	0	0	100%	2.733

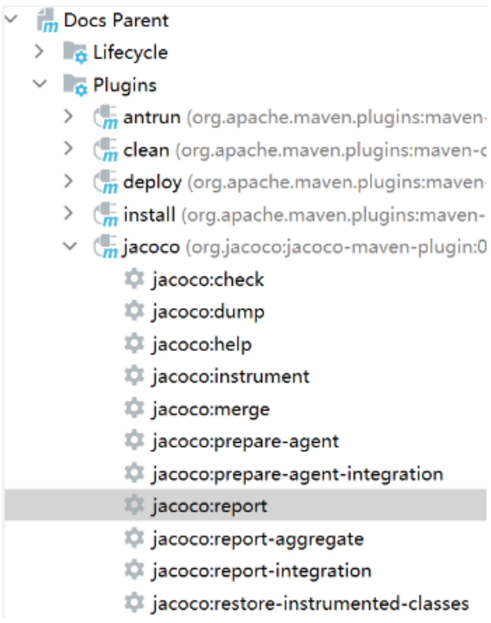
Test Coverage

Code coverage is a software metric used to measure how many parts of our code are executed during automated tests. In this tutorial, we'll use JaCoCo, a free code coverage reports generator for Java projects, to check the test coverage of Teedy.

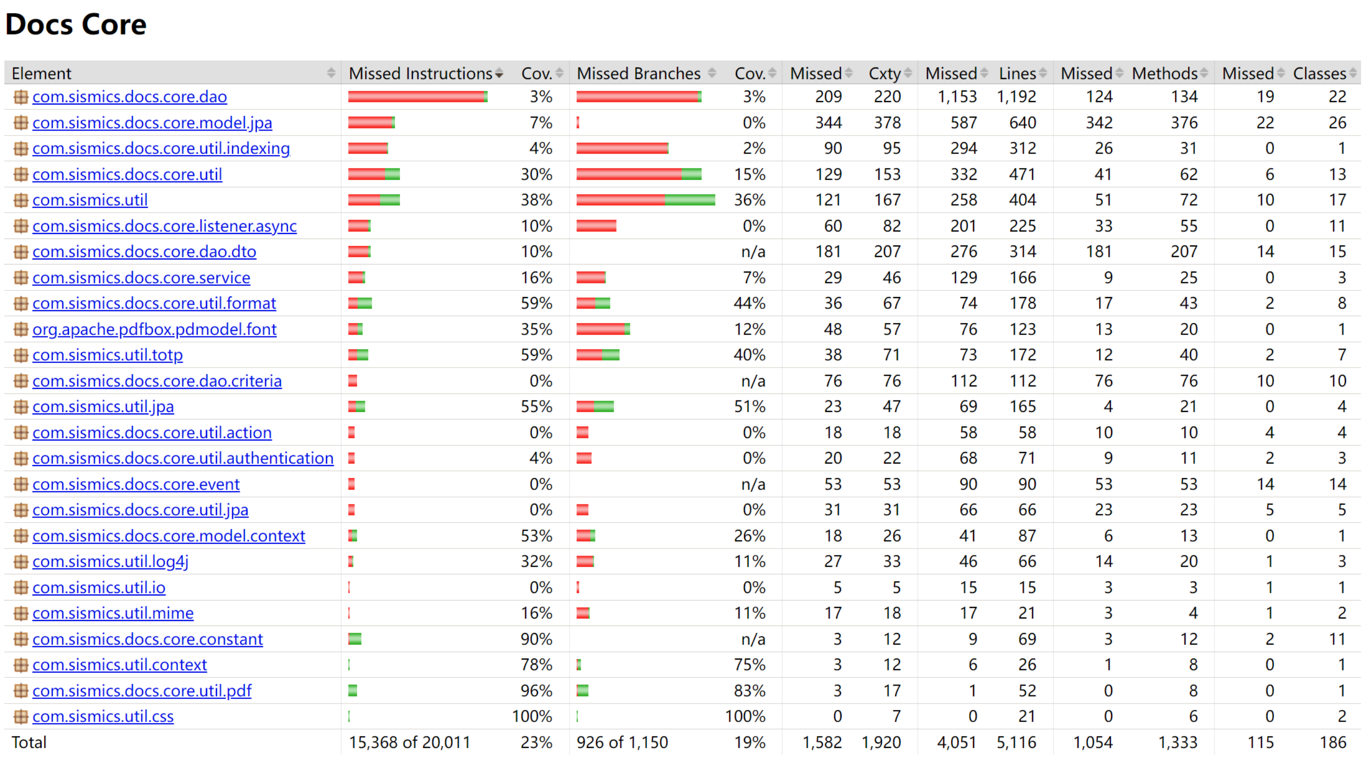
First, add the following into the `pom.xml` of Teedy (you might want to manually reload the project to reflect the change):

```
        <plugin>
          <groupId>org.jacoco</groupId>
          <artifactId>jacoco-maven-plugin</artifactId>
          <version>0.8.9</version>
          <executions>
            <execution>
              <goals>
                <goal>prepare-agent</goal>
              </goals>
            </execution>
            <!-- attached to Maven test phase -->
            <execution>
              <id>report</id>
              <phase>test</phase>
              <goals>
                <goal>report</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
```

Then, run `jacoco:report`.


Docs Parent
Lifecycle
Plugins
antrun (org.apache.maven.plugins:maven-
clean (org.apache.maven.plugins:maven-c
deploy (org.apache.maven.plugins:maven
install (org.apache.maven.plugins:maven-
jacoco (org.jacoco:jacoco-maven-plugin:0
jacoco:check
jacoco:dump
jacoco:help
jacoco:instrument
jacoco:merge
jacoco:prepare-agent
jacoco:prepare-agent-integration
jacoco:report
jacoco:report-aggregate
jacoco:report-integration
jacoco:restore-instrumented-classes

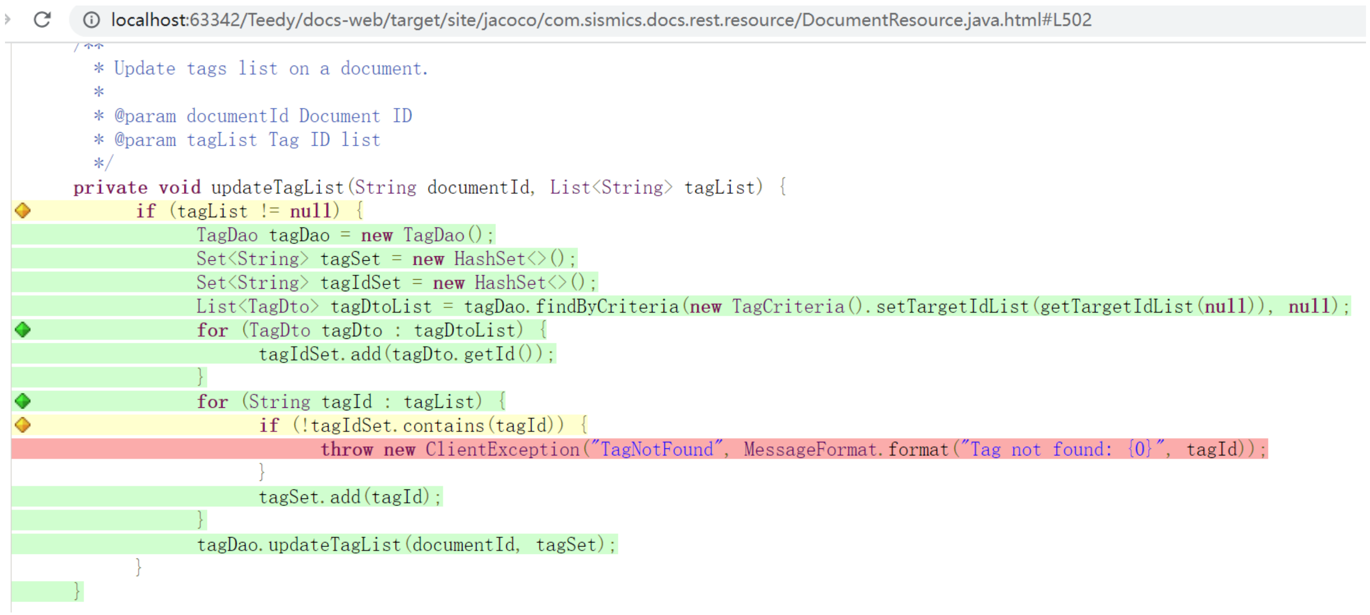
This will generate a coverage report at `target/site/jacoco/index.html` within each module. Open the report in a browser to navigate and observe the results.



Docs Core

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.sismics.docs.core.dao		3%		3%	209	220	1,153	1,192	124	134	19	22
com.sismics.docs.core.model.jpa		7%		0%	344	378	587	640	342	376	22	26
com.sismics.docs.core.util.indexing		4%		2%	90	95	294	312	26	31	0	1
com.sismics.docs.core.util		30%		15%	129	153	332	471	41	62	6	13
com.sismics.util		38%		36%	121	167	258	404	51	72	10	17
com.sismics.docs.core.listener.async		10%		0%	60	82	201	225	33	55	0	11
com.sismics.docs.core.dao.dto		10%		n/a	181	207	276	314	181	207	14	15
com.sismics.docs.core.service		16%		7%	29	46	129	166	9	25	0	3
com.sismics.docs.core.util.format		59%		44%	36	67	74	178	17	43	2	8
org.apache.pdfbox.pdmodel.font		35%		12%	48	57	76	123	13	20	0	1
com.sismics.util.totp		59%		40%	38	71	73	172	12	40	2	7
com.sismics.docs.core.dao.criteria		0%		n/a	76	76	112	112	76	76	10	10
com.sismics.util.jpa		55%		51%	23	47	69	165	4	21	0	4
com.sismics.docs.core.util.action		0%		0%	18	18	58	58	10	10	4	4
com.sismics.docs.core.util.authentication		4%		0%	20	22	68	71	9	11	2	3
com.sismics.docs.core.event		0%		n/a	53	53	90	90	53	53	14	14
com.sismics.docs.core.util.jpa		0%		0%	31	31	66	66	23	23	5	5
com.sismics.docs.core.model.context		53%		26%	18	26	41	87	6	13	0	1
com.sismics.util.log4j		32%		11%	27	33	46	66	14	20	1	3
com.sismics.util.io		0%		0%	5	5	15	15	3	3	1	1
com.sismics.util.mime		16%		11%	17	18	17	21	3	4	1	2
com.sismics.docs.core.constant		90%		n/a	3	12	9	69	3	12	2	11
com.sismics.util.context		78%		75%	3	12	6	26	1	8	0	1
com.sismics.docs.core.util.pdf		96%		83%	3	17	1	52	0	8	0	1
com.sismics.util.css		100%		100%	0	7	0	21	0	6	0	2
Total	15,368 of 20,011	23%	926 of 1,150	19%	1,582	1,920	4,051	5,116	1,054	1,333	115	186

Click any element to observe detailed code coverage.



localhost:63342/Teedy/docs-web/target/site/jacoco/com.sismics.docs.rest.resource/DocumentResource.java.html#L502

```
     * Update tags list on a document.
     *
     * @param documentId Document ID
     * @param tagList Tag ID list
     */
    private void updateTagList(String documentId, List<String> tagList) {
        if (tagList != null) {
            TagDao tagDao = new TagDao();
            Set<String> tagSet = new HashSet<>();
            Set<String> tagIdSet = new HashSet<>();
            List<TagDto> tagDtoList = tagDao.findByCriteria(new TagCriteria().setTargetIdList(getTargetIdList(null)), null);
            for (TagDto tagDto : tagDtoList) {
                tagIdSet.add(tagDto.getId());
            }
            for (String tagId : tagList) {
                if (!tagIdSet.contains(tagId)) {
                    throw new ClientException("TagNotFound", MessageFormat.format("Tag not found: {0}", tagId));
                }
                tagSet.add(tagId);
            }
            tagDao.updateTagList(documentId, tagSet);
        }
    }
```

JaCoCo mainly provides three important metrics:

- *Lines coverage* reflects the amount of code that has been exercised based on the number of Java byte code instructions called by the tests.
- *Branches coverage* shows the percent of exercised branches in the code, typically related to if/else and switch statements.
- *Cyclomatic complexity* reflects the complexity of code by giving the number of paths needed to cover all the possible paths in a code through linear combination. This includes not only the conditional branches but also other control structures like loops and try-catch blocks.

JaCoCo reports help us visually analyze code coverage by using diamonds with colors for branches, and background colors for lines:

- Red diamond means that no branches have been exercised during the test phase.
- Yellow diamond shows that the code is partially covered – some branches have not been exercised.
- Green diamond means that all branches have been exercised during the test.

The same color code applies to the background color, but for lines coverage.

References

- Maven Surefire documentation
- JaCoCo tutorial