STAMP

Deliverable D4.4

Final public version of API and implementation of services and courseware

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D4.4
Title of Deliverable	:	Final public version of API and implementation of services and courseware
Dissemination Level	:	Public
Version	:	1.7
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d44_final_api_public_version_services_courseware.pdf
Contractual Delivery Date	:	M36 November, 30 2019
Contributing WPs	:	WP 4
Editor(s)	:	Daniele Gagliardi, ENG
Author(s)	:	Mael Audren De Kerdrel, AEON Luca Andreatta, ENG Nicola Bertazzo, ENG Ciro Formisano, ENG Daniele Gagliardi, ENG JesÃžs GorroÃśogoitia Cruz, ATOS Caroline Landry, INRIA Ricardo Tejada, ATOS
Reviewer(s)	:	Benoit Baudry, KTH Caroline Landry, INRIA Vincent Massol, XWiki

Abstract

This Deliverable reports on the final release of STAMP components developed to integrate STAMP amplification services within developers' and DevOps tool-chains. While the Section on Industrialization of D1.4 offers on overview of it, this Deliverable specifies the mechanisms that enable STAMP to work seamlessly within existing DevOps processes and developer toolboxes. The Chapter 2 describes the CI/CD reference scenario, with a working example of all STAMP tools integrated in a Continuous Integration/Continuous Delivery server. The CI/CD scenario references also the usage of the other three STAMP tools (Descartes, CAMP and Botsing) within it. The Chapter 3 contains a description of the current state-of-the-art of STAMP ecosystem, a set of independent components that let STAMP adopters to use its novel features within their toolboxes. Each component includes technical documentation describing its usage and its integration in software life-cycle processes. The Chapters 4 and 5 in turn describe the collaborative platform, and the documentation available in the STAMP official repository, in terms of structure and contents.

Keyword List

Continuous Integration, Continuous Delivery, CI/CD Server, Issue Tracker, Pull Request, Issue tracker, microservice, pipeline

Revision History

Version	Type of Change	Author(s)
0.1	First draft	Daniele Gagliardi, ENG
0.2	Section about PitMP included	Caroline Landry, INRIA
0.5	CI/CD scenario completed Contribution on STAMP IDE (ATOS) included Contribution on Botsing Gradle plugin (ActiveEon) included	Daniele Gagliardi, ENG Ricardo Tejada, ATOS Mael Audren De Kerdrel, ActiveEon
0.6	Section on the Architecture completed First draft on the docuentation and courseware produced	Daniele Gagliardi, ENG
0.9	First draft completed.	Daniele Gagliardi, ENG Ciro Formisano, ENG
1.0	First release after internal review by ENG	Daniele Gagliardi, ENG
1.1	PitMP section consolidated	Caroline Landry, INRIA
1.2	Second release after the review by XWiki and KTH	Daniele Gagliardi, ENG
1.3	Fine tuned chapter 4. Fixed wrong references to RAMP.	Daniele Gagliardi, ENG
1.4	Fixing typos and references	Caroline Landry, INRIA
1.5	Fixing remaining missing figure references	Daniele Gagliardi, ENG
1.6	Final release	Daniele Gagliardi, ENG Ciro Formisano, ENG
1.7	Final review: fixing references	Caroline Landry, INRIA

Contents

Acronyms

CD	Continuous Delivery
CI	Continuous Integration
EC	European Commission

List of Figures

List of Tables

Listings

Chapter 1

Introduction

During the past year of the project, the integration activities were mainly focused on the implementation of a CI/CD process, enhanced with STAMP novel features.

The reference architecture was finalized thanks to a strict collaboration among all the partners: ENG presented four different versions of this architecture, by the means of demos held during 3 in-person meetings (Sophia-Antipolis, France, Jan. 30-31 2019 ; Stockholm, Sweden, Apr. 9-10 2019; Madrid, Spain, Oct. 8-10 2019) and 9 online meetings, collecting requirements and feedback by other partners who provided real use cases possible scenarios and validated the architecture against them.

Specifically, the reference architecture is based on tools widely used among developers and DevOps communities: Jenkins CI, Jira Software, GitHub, GitHub Issues, Apache Maven, Gradle, Docker, Docker Compose, Eclipse. Each of these tools is associated with a STAMP integration component enabling test amplification features.

Other tools have been enhanced and consolidated, in order to provide developers with a complete toolbox to easily introduce test amplification aspects, regardless the specific development process. These multiple options should help to increase the adoption of STAMP: infact, even if DevOps is currently very popular, other legacy development methodologies are still widespread, and the itroduction of STAMP novel concepts will add a big value.

This deliverable presents the final release STAMP amplification services integrated in developers‘ and DevOps tool-chains, along with documentation and course-ware available to understand their usage and apply them within software life-cycle processes.

The CI/CD architecture is described in chapter 2. Specifically, the Scenario consists in several components, orchestrated by a CI/CD server, which cooperate to improve the quality of the produced software. The description of the *"Ops"* phase shows the usage of the automatic reproduction features to analyse bugs that occurrs in production. In particular, for Jira and GitHub, the error stack-trace can be included in a *issue* defined through the tool: STAMP plugins recognize the specific issue and trigger the proper tool.

As already mentioned in D4.3, this reference architecture is composed by several services and plugins that facilitate the integration in different development contexts and make it extensible and flexible.

The second part of this deliverable (chapter 3) provides a description of the STAMP ecosystem, composed by several plugins that enable the integration in the most common toolboxes, such as IDE and build tools.

STAMP collaborative platform is described in chapter 4: particular emphasis will be given to the improvements introduced during the last year of the project.

The last part of the deliverable (chapter 5) described the documentation provided in STAMP official repository: specifically it can be seen as a guide for the adopter to find all needed

information. Moreover a Docker image, based on a Jenkins CI server, enabling STAMP adopters to test STAMP in a typical CI/CD environment is provided and described.

1.1 Relation to WP4 tasks

In Table 1.1 we summarize how developed software assets and results reported in this deliverable relate to the tasks of WP4.

Table 1.1: WP4 tasks reported in this deliverable

Task 4.1	This task, about setting up and maintaining the STAMP project collaborative platform, has been extended to M36. This deliverable describes the new features, the improvements and the activities performed to maintain the platform supporting the project (chapter 4)
Task 4.2	The CI/CD integration architecture was finalized to support test amplification practices in a typical DevOps tool-chain (chapter 2)
Task 4.3	New plugins were built and the existing ones were maintained by keeping them aligned with new versions of STAMP tools. STAMP integration in developers toolboxes was extended (chapter 3)
Task 4.4	The documentation needed to use STAMP tools and integration was extended. It is available in form of wiki pages on the official STAMP GitHub repository. A Docker image was produced to integrate the usage documentation in order to smooth learning curve for STAMP adopters (chapter 5)

1.2 Participants contribution to WP4

Since the scope of WP4 is the integration of produced tools, all the work is the result of strong cooperation among technical and use cases partners. In Table 1.2 more details on the aspects related to cooperation are presented.

Table 1.2: WP4 participants and their activities

ENG	refined and implemented the CI/CD scenario, developing needed components and instantiating it on STAMP server (part of STAMP collaborative platform) (see chapter 2). ENG also led the evolution of STAMP ecosystem developing new components and keeping the existing ones aligned with new versions of STAMP tools (see chapter 3). ENG collaborated with OW2 on maintaining collaborative platform (see chapter 4). ENG eventually finalized STAMP documentation, collecting documents and tutorials developed by other partners, writing documentation about CI/CD integration and developing course-ware based on Docker images (see chapter 5).
INRIA, KTH	validated the the CI/CD scenario implemented by ENG (see chapter 2).
INRIA	finalized the development of some STAMP integration components (i.e. PitMP; see section 3.1)
XWIKI	provided solutions based on Testcontainers as an initial reference implementation to execute functional tests against CAMP-amplified test configurations (see chapter 2)
ATOS	completed implementation of STAMP IDE (see section 3.3) and collaborated with ENG in defining Jenkins pipelines to use CAMP for performance testing on amplified configurations
ActiveEon	provided feedback about STAMP integration components and finalized Gradle integration for Botsing (see section 3.1)
OW2	led STAMP collaborative platform maintenance (see chapter 4)
INRIA, SINTEF, TUD	provided support to understand how to embed respectively DSpot & Descartes, CAMP, Botsing in integration targets: build systems, IDEs, CI/CD servers, etc (see chapter 2 and chapter 3)

Chapter 2

STAMP in CI/CD

2.1 Introduction

This chapter provides a detailed description of the CI/CD STAMP-enhanced scenario, implemented and made available within the STAMP Demo server, a component of the collaborative platform in which STAMP tools can be tested in a typical DevOps tool-chain.

When referring to STAMP tools in this document, we refer to the following technical components for test amplification developed in Work-packages 1, 2, 3:

- DSpot: a tool that generates missing assertions in JUnit tests.
  https://github.com/STAMP-project/dspot
- Descartes: evaluates the capability of a suite to detect bugs using extreme mutation testing.
  https://github.com/STAMP-project/pitest-descartes
- PitMP: enables PIT https://pitest.org/ mutation testing and Descartes extreme mutation testing to be executed on Maven multi-module projects
- CAMP: automatically generates and deploy a number of diverse testing configurations.
  https://github.com/STAMP-project/camp
- Botsing: a Java framework to automatically generate crash reproducing test cases.
  https://github.com/STAMP-project/botsing

Specifically in the CI/CD scenario DSpot, Descartes/PitMP and CAMP support unit, integration and system level testing, while Botsing aims at speeding-up the investigation phase of production bug.

2.2 Reference Scenario

The section 2.3 of D4.3 presents a general description of the CI/CD scenario, named "*CI/CD landscape*". This schema has been further developed during Y3, leading to the reference scenario described in figure 2.1.

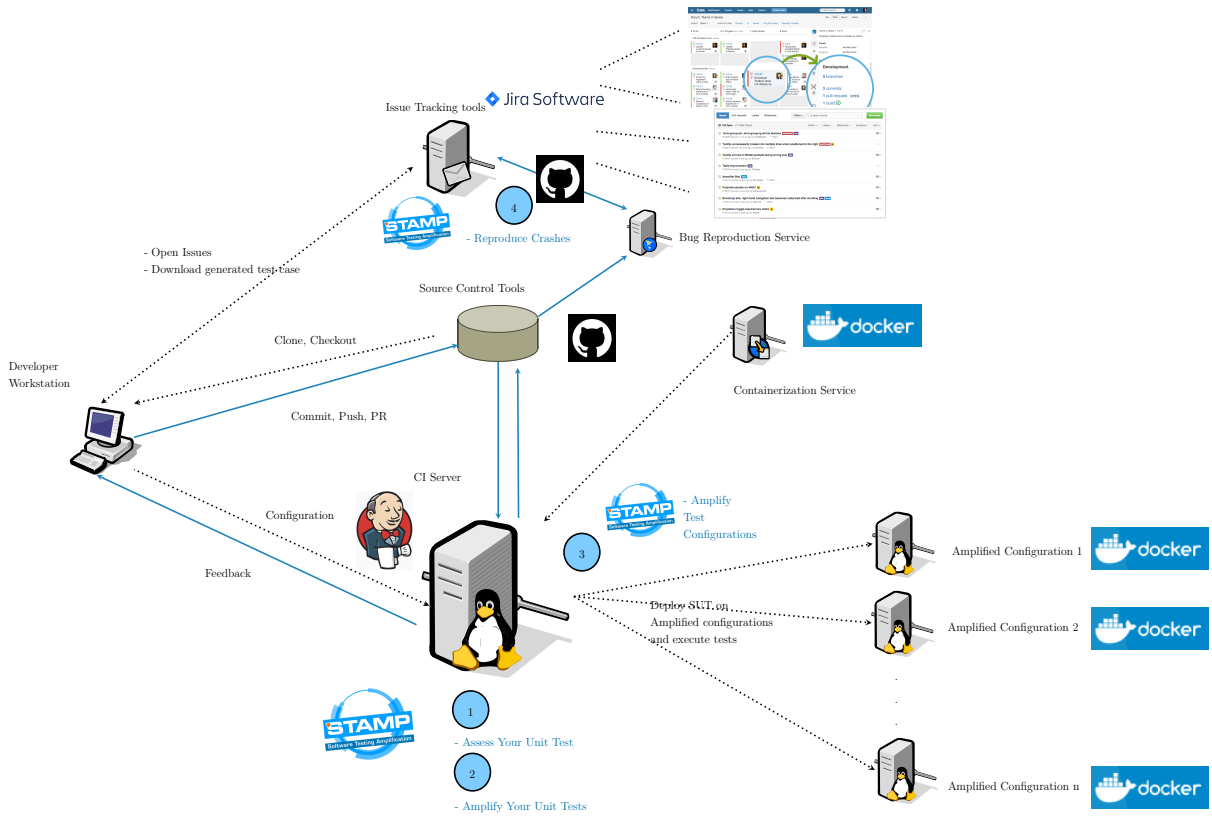


Figure 2.1: STAMP CI/CD reference scenario

Jenkins CI is the **Build/CI Server** that provides and supervises the integrated automation of all tasks aimed at building, verifying and validating the software. In order to use STAMP testing features integrated into Jenkins, the developer should perform the following actions:

1. provide Jenkins CI with specific **pipelines** implementing automated processes. The Build/CI Server is configured to execute several jobs running periodically or triggered by specific events (commits, push, pull requests, etc). Typical jobs are:
   (a) build the software
   (b) execute unit and integration tests
   (c) publish artifacts
   (d) instantiate target environments (QA, Test, Prod), deploy software on them and execute system tests

   These jobs are enriched by test amplification capabilities
2. get **feedback** in form of reports and dashboards
3. configure and integrate container-based tools to prepare the target systems where the software will run (**test configuration provisioning**). STAMP CI/CD scenario leverages Docker technology
4. manage target systems with **configuration management tools**, using "infrastructure as code" practices. Specifically Docker files and Docker compose files should be configured

in order to define target systems to execute integration and system level tests (usually functional and performance tests)

5. write source coce and tests on the developer's **preferred productivity tools**. STAMP IDE is a set of Eclipse plugins that can be used on the workstation of the developer providing all tools and functionalities of STAMP
6. store source code, test cases and Jenkins pipelines in **source code repositories** which will keep track of any change. In the reference scenario GitHub is used as a GIT repository hosting service;
7. track all tasks and user stories on **Issue trackers** in order to register the system requirements, as well as bugs that arise from testing and from production. STAMP reference scenario supports two issue trackers: **Jira Software** and **GitHub issue tracker**. These two tools are currentlt very popular and should cover several issue tracking needs in the world of open source development.

2.2.1 Unit test amplification

In this section the STAMP unit test amplification features which developers can adopt within their CI processes are described. Figure 2.2 shows the detailed unit test amplification reference scenario:

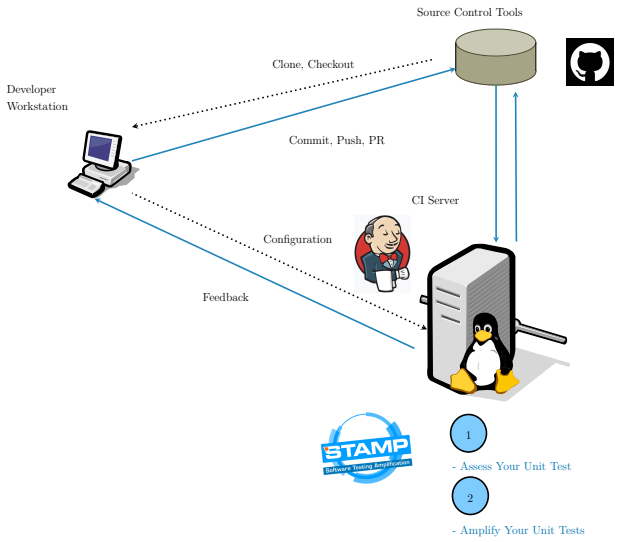


Figure 2.2: STAMP CI Unit Test Amplification reference scenario

Developers can configure the CI server with one or more pipelines enriched with test assessment and test amplification tasks:

1. when a new change in the code is pushed in the repository, after usual build and unit test execution, trigger an execution of Descartes (or PitMP for multi-module projects) on existing test cases
2. publish Pit reports with mutation coverage within Jenkins dashboard
3. trigger an execution of DSpot in order to amplify existing test cases

4. create a new branch for amplified test cases and open a pull request to make them accepted in the code base.

DSpot generated test cases may require some manual cleanup: for this reason it seems reasonable to have amplified test cases in a separate directory visible to Maven. This can be obtained by editing Maven `pom.xml` file, so that the pull request stage eventually would refer this additional test sources folder, instead of the standard src/test folder. An example of this edited pom.xml file is provided by XWiki: https://github.com/xwiki/xwiki-commons/blob/8355bbbddd9cf7158e832f6612ff445c2fd687f6/pom.xml#L2002-L2021 The generate-test-source phase is configured to have an additional test source code for inclusion in compilation (more information about Maven build lifecycle are at https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html). However, for the moment, to simplify, let's suppose that amplified test cases are stored in same folder of existing test cases. The following pipeline (available in STAMP GitHub repository, see https://github.com/STAMP-project/joram/blob/master/Jenkinsfile) performs the amplification tasks outlined before:

```
@Library('stamp') _

pipeline {
  agent any
  stages {
    stage('Compile') {
      steps {
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml clean compile"
        }
      }
    }

   stage('Unit Test') {
      steps {
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml test"
        }
      }
    }

    stage ('Test your tests'){
      steps {
        sh "echo 'Test case change detected, start to assess them
            with PitMP/Descartes...'"
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml eu.stamp-project:pitmp-maven-
              plugin:descartes -DoutputFormats=HTML"
        }
        sh "echo 'Test case change detected, assessment with Pit
           finished, publishing HTML report...'"
        publishHTML (target: [
            allowMissing: false,
            alwaysLinkToLastBuild: false,
```

```
            keepAll: true,
            reportDir: 'joram/joram/mom/core/target/pit-reports',
            reportFiles: '2019*/index.html',
            reportName: "Mutation coverage"
        ])
      }
    }

    stage('Amplify') {
     when { not {branch "amplifybranch*"}
          changeset "joram/joram/mom/core/src/main/**" }
      steps {
        sh "echo 'Code change detected, start to amplify test
            cases with DSpot...'"

         withMaven(maven: 'maven3', jdk: 'JDK8') {
            dir ("joram/joram/mom/core") {
            sh "mvn eu.stamp-project:dspot-maven:amplify-unit-
               tests -Dverbose -Diteration=4"
          }
        }
      }
    }

    stage('Pull Request') {
        when { not {branch "amplifybranch*"}
            expression { fileExists("joram/joram/mom/core/target/
               dspot/output/org")} }
      steps {
        sh 'cp -rf joram/joram/mom/core/target/dspot/output/org/
           joram/joram/mom/core/src/test/java'
        sh 'git checkout -b amplifybranch-${GIT_BRANCH}-${
           BUILD_NUMBER}'
        sh 'git commit -a -m "added tests"'
        // CREDENTIALID
        withCredentials([usernamePassword(credentialsId: 'github-
           token', passwordVariable: 'GITHUB_PASSWORD',
           usernameVariable: 'GITHUB_USER')]) {
          // REPOSITORY URL
          sh('git push https://${GITHUB_USER}:${GITHUB_PASSWORD}
             @${GIT_URL} amplifybranch-${GIT_BRANCH}-${
             BUILD_NUMBER}')

          script {
            stamp.pullRequest("${GITHUB_PASSWORD}", "joram", "
               STAMP-project", "amplify Test", "amplify Test
               Build Number ${GIT_BRANCH}-${BUILD_NUMBER}", "
               amplifybranch-${GIT_BRANCH}-${BUILD_NUMBER}", "${
               GIT_BRANCH}")
          }
        }
```

```
        }
      }
    }
     environment {
      GIT_URL = sh (script: 'git config remote.origin.url',
         returnStdout: true).trim().replaceAll('https://','')
    }
}
```

Listing 2.1: Jenkins pipeline with test assessment and test amplification tasks

This pipeline is customized for Joram project (see references to actual Joram source code in the various stages), but can be easily adapted to every Maven project.

Using the Blue Ocean interface (see https://jenkins.io/projects/blueocean/), the whole execution of pipeline stages is represented as described in figure 2.3.

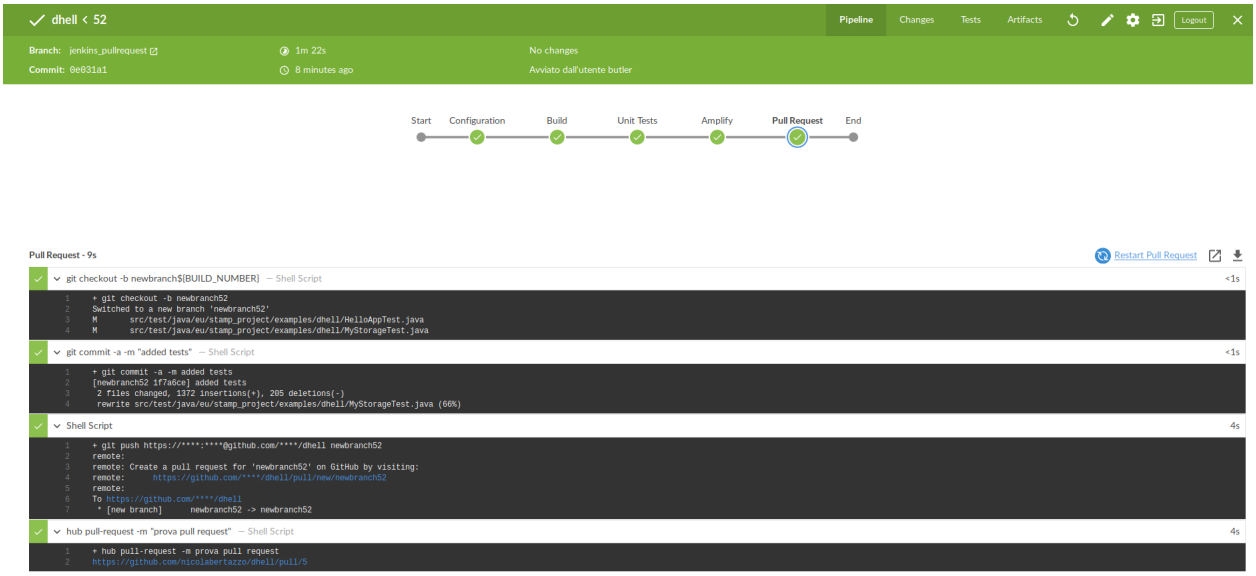


Figure 2.3: Unit test amplification pipeline execution, shown by Blue Ocean interface

2.2.1.1 Unit test amplification - Test assessment with PitMP/Descartes

After ordinary `Compile` and `Unit Test` tasks, a test assessment task is executed against existing test cases (task "`Test your tests`"). This task is executed through PitMP by using the following statement:

`mvn -f joram/pom.xml eu.stamp-project:pitmp-maven-plugin:descartes -DoutputFormats=HTML`

Maven goal `eu.stamp-project:pitmp-maven-plugin:descartes` triggers the execution of PitMP, leveraging the `extreme mutation testing` function of Descartes, applied to the current project that is a multi-module Maven project. The parameter `-DoutputFormats=HTML` instructs PitMP to provide mutation testing output in HTML format. The next step in the pipeline handles the format of the output to make it suitable to Jenkins dashboard:

```
publishHTML (target: [
            allowMissing: false,
            alwaysLinkToLastBuild: false,
```

```
            keepAll: true,
            reportDir: 'joram/joram/mom/core/target/pit-reports',
            reportFiles: '2019*/index.html',
            reportName: "Mutation coverage"
        ])
```

Listing 2.2: Jenkins pipeline snippet to publish mutation coverage report within Jenkins dashboard

Figure 2.4 shows the mutation coverage reports available in Jenkins.

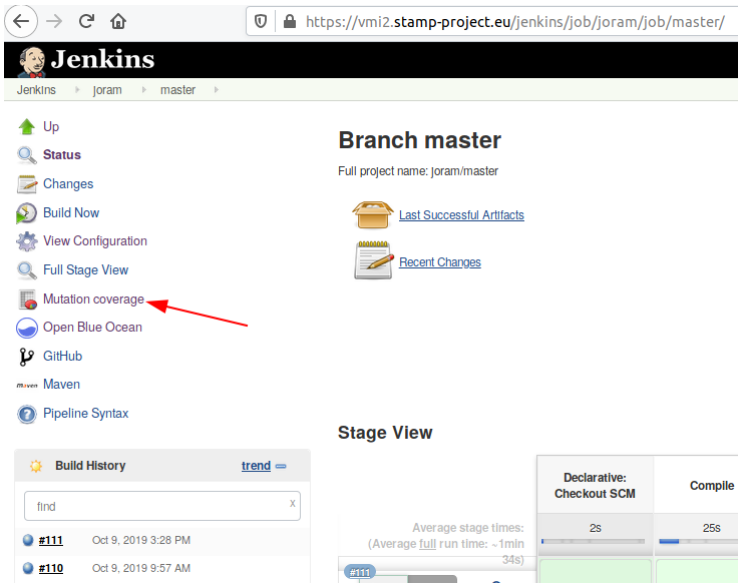


Figure 2.4: Menu item in Jenkins dashboard to access mutation coverage report

The report contains the *mutation coverage* at the package level (2.5) and, at code level, the tool proposes improvements to the test cases (figure 2.6).

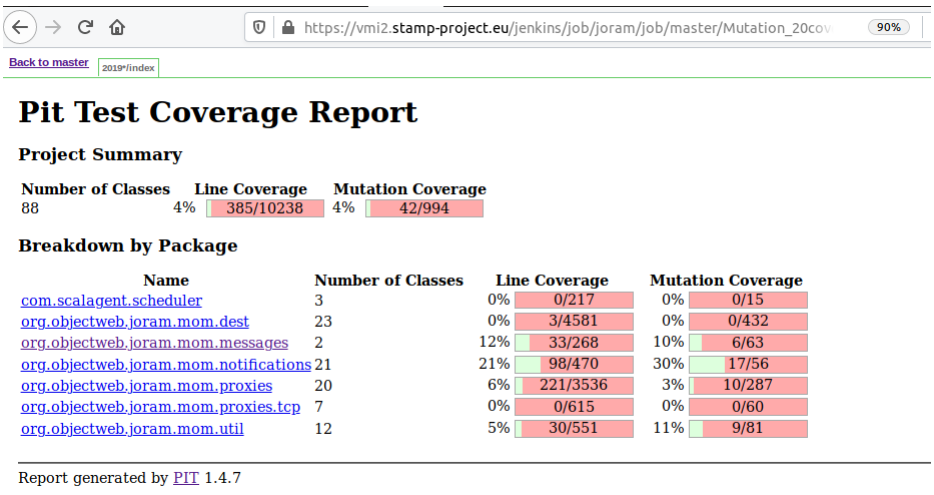


Figure 2.5: Mutation coverage report at package level.

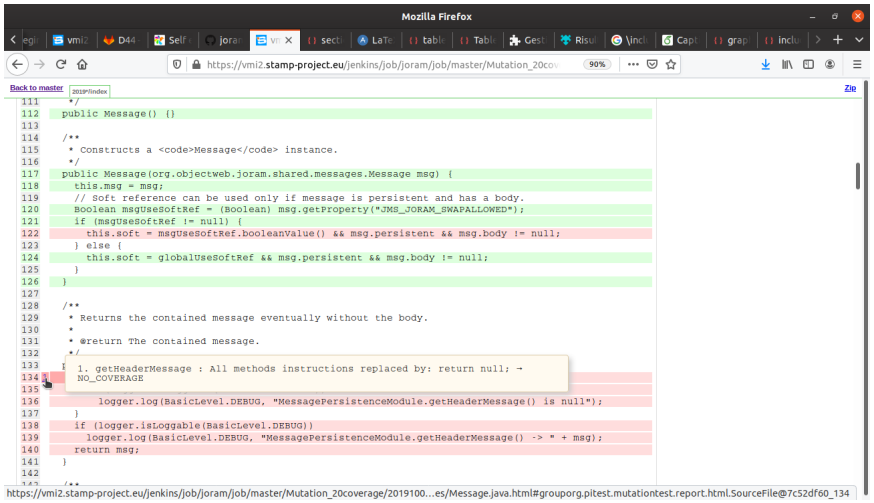


Figure 2.6: Inspecting test cases strength issues through the drill-down feature of mutation coverage report.

2.2.1.2 Unit test amplification - Test amplification with DSpot

The next step is the optional stage called `Amplify`. This task is optimized to be executed only in case of code changes, in order to avoid useless amplification sequences. In particular, specific checks are performed on the branch to detect code changes and trigger test amplification: these checks also distinguish *useful changes* (e.g. code changes) from changes related to other items (such as markdown pages, images, test cases, etc) that should not trigger any test:

```
when { not {branch "amplifybranch*"}
       changeset "joram/joram/mom/core/src/main/**" }
```

Listing 2.3: Jenkins pipeline snippet to show how to trigger test amplification only on code changes and in specific branches

The condition `not branch "amplifybranch*"` prevents a potential *infinite loop* of test amplifications. Specifically, amplified test cases are pushed in a dedicated branch and Jenkins is notified by the `push` event and triggers the execution of the pipeline against this branch. This condition prevents useless test amplification (and subsequent new pull requests).

The condition `changeset "joram/joram/mom/core/src/main/**"` ensures that test amplification will be triggered only by code changes. Changes on other items under version control in the source code repository do not trigger test amplification. Test cases changes are excluded as well (otherwise a new infinite loop of test amplification would happen).

This conditional execution is represented by Blue Ocean interface as shown in figure 2.7.

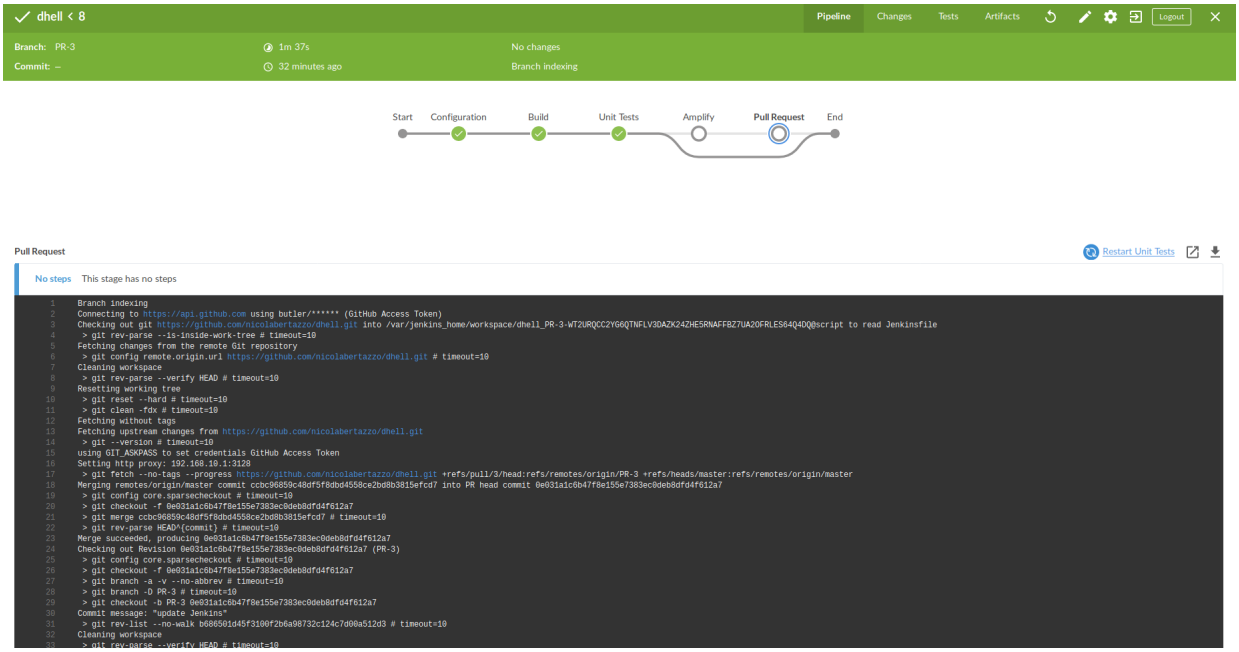


Figure 2.7: Pipeline execution skipping unit test amplification, shown by Blue Ocean interface

2.2.1.3 Unit test amplification - Push amplified test cases in the repository

The last step makes the new amplified test cases available in the repository, in a dedicated branch. Jenkins automatically openes a new pull request that can be inspected to examinate the new amplified test cases and decide whether include them in the code base or not:

```
    stage('Pull Request') {
        when { not {branch "amplifybranch*"}
            expression { fileExists("joram/joram/mom/core/target/
               dspot/output/org")} }
      steps {
        sh 'cp -rf joram/joram/mom/core/target/dspot/output/org/
           joram/joram/mom/core/src/test/java'
        sh 'git checkout -b amplifybranch-${GIT_BRANCH}-${
           BUILD_NUMBER}'
        sh 'git commit -a -m "added tests"'
        // CREDENTIALID
        withCredentials([usernamePassword(credentialsId: 'github-
           token', passwordVariable: 'GITHUB_PASSWORD',
           usernameVariable: 'GITHUB_USER')]) {
          // REPOSITORY URL
          sh('git push https://${GITHUB_USER}:${GITHUB_PASSWORD}
             @${GIT_URL} amplifybranch-${GIT_BRANCH}-${
             BUILD_NUMBER}')

          script {
            stamp.pullRequest("${GITHUB_PASSWORD}", "joram", "
               STAMP-project", "amplify Test", "amplify Test
               Build Number ${GIT_BRANCH}-${BUILD_NUMBER}", "
```

```
                amplifybranch-${GIT_BRANCH}-${BUILD_NUMBER}", "${
                GIT_BRANCH}")
            }
        }
    }
}
```

Listing 2.4: Jenkins pipeline snippet to show how to push amplified test cases in the code base

The condition `not branch "amplifybranch*"` prevents an infinite loop of creation of new branches, opening pull requests, etc. The condition `expression fileExists("joram/joram/mom/core/target/dspot/output/org")` ensures that the whole step is executed only if DSpot has previously generated new test cases: the sub-path `target/dspot/output/org` is the DSpot configured output folder to store amplified test cases.

Step `sh 'cp -rf joram/joram/mom/core/target/dspot/output/org/joram/joram/mom/core/src/test/java'` actually copies amplified test cases in the directory containing all existing test cases.

```
sh 'git checkout -b amplifybranch-${GIT_BRANCH}-${
    BUILD_NUMBER}'
    sh 'git commit -a -m "added tests"'
    // CREDENTIALID
    withCredentials([usernamePassword(credentialsId: 'github-
        token', passwordVariable: 'GITHUB_PASSWORD',
        usernameVariable: 'GITHUB_USER')]) {
      // REPOSITORY URL
      sh('git push https://${GITHUB_USER}:${GITHUB_PASSWORD}
          @${GIT_URL} amplifybranch-${GIT_BRANCH}-${
          BUILD_NUMBER}')
```

Listing 2.5: Jenkins pipeline snippet to show how to create a new branch for amplified test cases

The three steps in listing 2.5 illustrate how to commit amplified test cases, create a new branch with name `amplifybranch-`*`current branch-current build number`* (self-explanatory name containing branch name and build number), and push amplified test cases in this new branch.

The final step opens a new pull request:

```
        script {
          stamp.pullRequest("${GITHUB_PASSWORD}", "joram", "
              STAMP-project", "amplify Test", "amplify Test
              Build Number ${GIT_BRANCH}-${BUILD_NUMBER}", "
              amplifybranch-${GIT_BRANCH}-${BUILD_NUMBER}", "${
              GIT_BRANCH}")
        }
```

Listing 2.6: Jenkins pipeline snippet to show how to open a pull request for amplified test cases, using STAMP pipeline library

The `stamp.pullRequest()` function is one of custom steps available in STAMP pipeline library (https://github.com/STAMP-project/pipeline-library), which simplifies complex tasks such as opening pull requests using GitHub APIs, iterating over Jenkins builds to find the first successful build, cloning specific branches or commits, etc.

The following statement, at the very beginning of the pipeline, declares that the aforementioned library will be used:

```
@Library('stamp') _

pipeline {
...
```

Listing 2.7: Jenkins pipeline snippet to show how to declare STAMP pipeline library usage

2.2.2 Test configuration amplification

This section describes the STAMP test configuration amplification features that developers can adopt within their CD processes. Figure 2.8 shows the detailed test configuration amplification reference scenario.

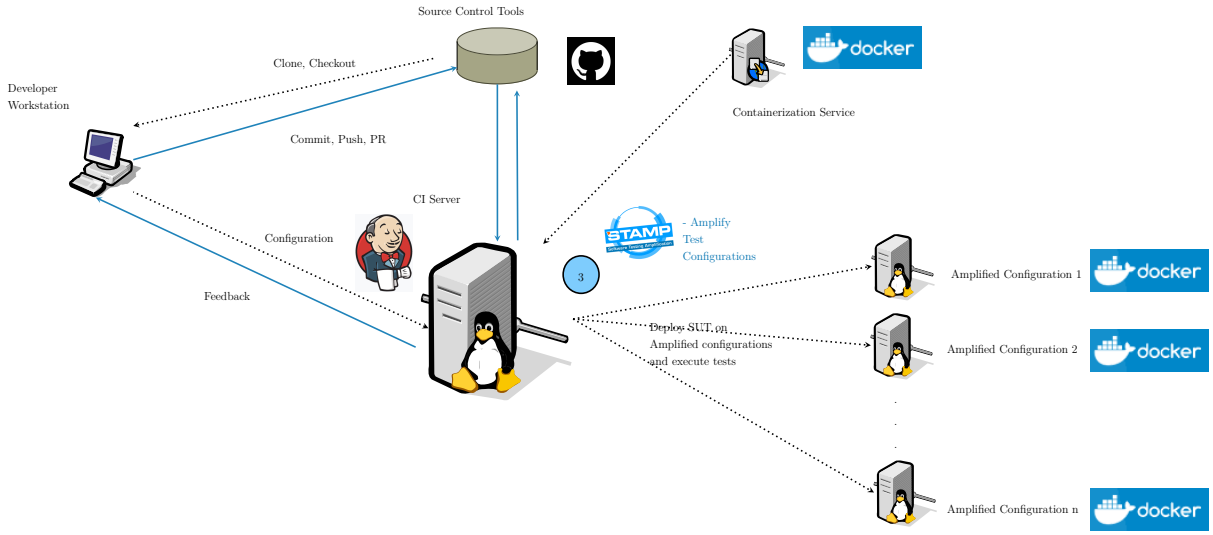


Figure 2.8: STAMP CD Test Configuration Amplification reference scenario

In this context developers leverage Docker technology to quickly setup and provide automatically test environments. Docker technology enables developers to produce their test configurations by editing them on text files (Docker and Docker Compose files) containing commands to setup the SUT (*System Under Test*), the dependencies and the test framework. Specifically, each Docker file represents a component of the infrastructure (database, web server, application server, test client, etc.), while the Docker compose file is used to define dependencies among all components. In this way developers can setup different configurations in order to execute integration tests (for instance to certify a specific rDBMS version) or performance tests to identify the best configuration in terms of performances. However, as test configuration number increases, to manually setup different Docker and Docker compose files may become an error-prone activity, and the risk to ignore significant test configurations may grow.

CAMP (see https://github.com/STAMP-project/camp) operates on these aspects, by automating the generation of several different configurations by executing tests against them.

CAMP can be used in two ways:

1. generation of new test configurations and execution of integration/system tests against them.
2. only generation of new test configurations

The second mode of use implies to delegate test execution to external frameworks. The following sections detail both the mentioned ways.

2.2.2.1 Test execution performed by CAMP

2.2.2.1.1 Test Configuration amplification - Functional tests

Figure 2.9 shows a typical project layout supporting test configuration amplification performed by CAMP. CAMP also takes care of executing test cases against generated test configurations.

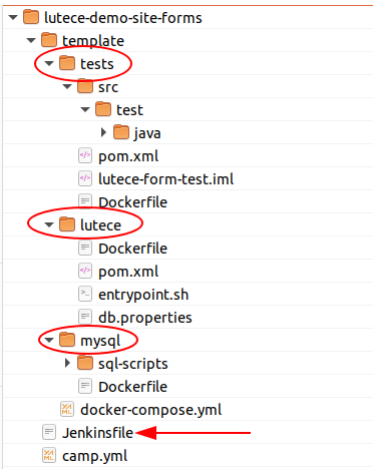


Figure 2.9: Typical project layout supporting IaaC and test configuration amplification

The layout of this project under test are organized in the following folders:

1. `tests` folder: contains system functional tests (in this case based on Selenium: source code is in the sub-folder `src/`); the `Dockerfile` provides the configuration to build and execute the test client
2. `lutece` folder: contains the SUT (in this case an instance of Lutece configured to offer a demo site); the Dockerfile contains all commands to get a Lutece instance up & running
3. `mysql` folder: Lutece needs a database to persists data. Several database products can be used, such as MySQL, MariaDB, etc

The `docker-compose.yml` file defines all dependencies among components.
Putting everything in a CI/CD scenario boosts automation at a higher level. Developers can configure the CI/CD server with one or more pipelines enriched with test configuration amplification tasks:

1. when a new change on the existing IaaC scripts is pulled in the repository, after usual build and unit tests, CAMP is executed on existing test configurations:
   (a) `generate` new configuration variants
   (b) `realize` them as actual Docker and Docker compose files
   (c) `execute` tests (functional and/or performance) at integration and system level
2. for each amplified test configuration, test execution reports are published on Jenkins dashboard

3. generated configurations are made available on Jenkins dashboard

Referring figure 2.9, the `Jenkinsfile` contains the definition of the test configuration amplification pipeline.

The whole project is available in STAMP GitHub repository, at https://github.com/STAMP-project/lutece-demo-site-forms).

The following listing contains the pipeline tasks in detail:

```
pipeline {

  agent any
  stages {

    stage('build') {
      steps {
       sh '''cd template/lutece
          mvn lutece:site-assembly'''
      }
    }

    stage('camp generate') {
      steps {
        sh 'camp generate -d .
      }
    }
    stage('camp realize') {
      steps {
        sh 'camp realize -d .'
        zip zipFile: 'testconf.zip', archive: true, dir: 'out',
           glob: '**/*.yml'
        zip zipFile: 'dockerfiles.zip', archive: true, dir: 'out
           ', glob: '**/Dockerfile'
      }
    }
    stage('execute tests') {
      steps {
        sh 'camp execute'
        junit 'out/**/TEST*.xml'
      }
    }
  }

}
```

Listing 2.8: Jenkins pipeline with test configuration amplification tasks

Using the Blue Ocean interface (see https://jenkins.io/projects/blueocean/), the whole execution of pipeline stages is represented as in figure 2.10, in Jenkins current build dashboard.

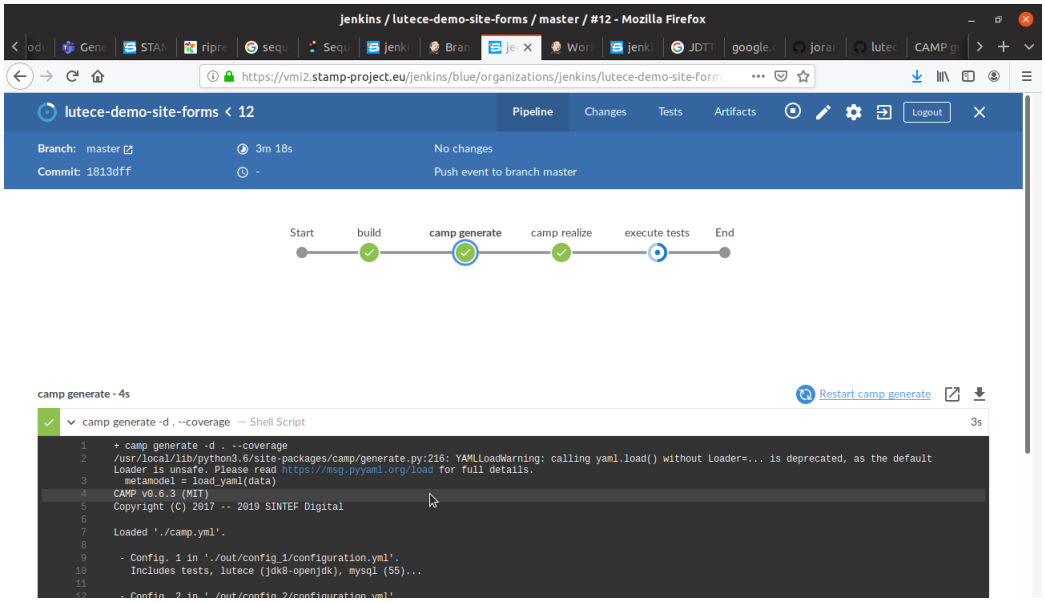


Figure 2.10: Test Configuration amplification pipeline execution, shown by Blue Ocean interface

The first step builds the SUT (System Under Test): according to official Lutece documentation (see https://fr.lutece.paris.fr/fr/wiki/maven.html), the command `mvn lutece:site-assembly` "*generates the binary distribution for a Lutece site project*".

After built the SUT, CAMP actions are performed:

1. `camp generate -d .` stage generates new configurations given a description of what can be varied. Adding the option `-coverage`, CAMP generates only the subset that covers all possible variations

2. `camp realize -d .` stage builds the docker images and the docker-compose file needed to run the configurations generated by using `camp generate -d .`. Other two steps are executed in this stage, in order to make generated configurations available in Jenkins interface:

```
zip zipFile: 'testconf.zip', archive: true, dir: 'out
    ', glob: '**/*.yml'
zip zipFile: 'dockerfiles.zip', archive: true, dir: '
    out', glob: '**/Dockerfile'
```

Listing 2.9: Jenkins pipeline with test configuration amplification tasks

`testconf.zip` file contains both CAMP configuration file (`camp.yml`) and Docker compose file (`docker-compose.yml`). `dockerfiles.zip` file contains all generated Docker files. These files can be accessed in the section `Artifacts` of current build (figure 2.11):

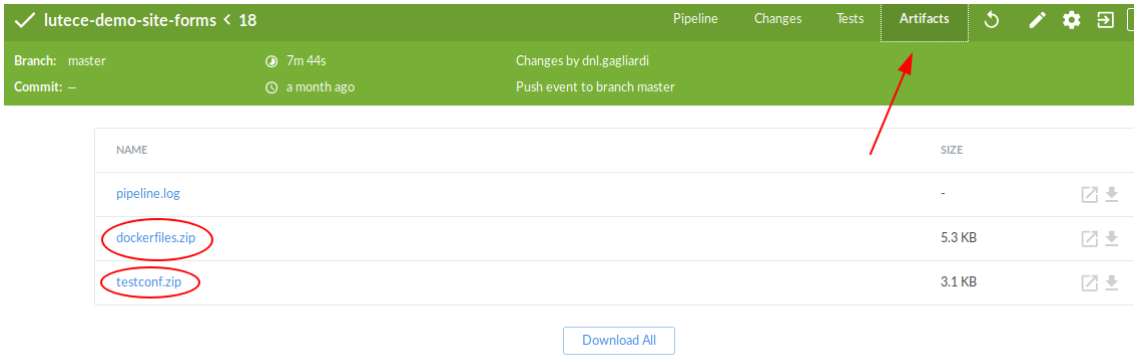


Figure 2.11: Amplified test configurations available as current build artifacts, within Jenkins Blue Ocean interface

3. `camp execute -d .` stage executes all integration/system tests on all generated configurations. When CAMP execution terminates, another step is executed by Jenkins to make test execution results available: `junit 'out/**/TEST*.xml'`. Through this step, developer can inspect test execution results in Jenkins dashboard (figure 2.12).

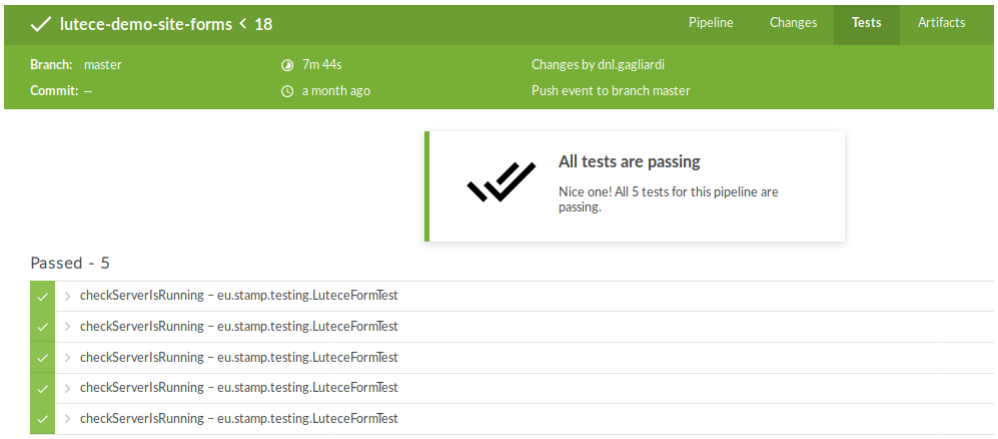


Figure 2.12: Results of Integration/System tests executed on amplified test configurations, shown within Jenkins Blue Ocean interface

2.2.2.1.2 Test Configuration amplification - Performance tests

Figure 2.13 shows another project layout supporting IaaC and test configuration amplification performed by CAMP.

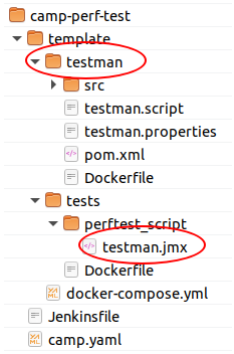


Figure 2.13: A sample project layout supporting IaaC and test configuration amplification, applied to performance testing

This project layout is organized in the following folders:

1. `tests` folder: contains system performance tests (based on Apache JMeter script); the `Dockerfile` provides the configuration to execute Apache JMeter, fed by `testman.jmx` script
2. `testman` folder: contains the SUT (in this case a web application named Testman, a simple test management system to store test cases and test executions - https://stamp-project.github.io/camp/pages/execute.html - section **Example: Performance Test A Java WebApp**); the Dockerfile contains all steps to build and have a Testman instance up & running.

As usual, `docker-compose.yml` file defines all dependencies among all components.

The whole process (generating test configurations, executing performance tests against them, inspecting HTML reports) can be automated by including everything in a CI/CD scenario.

Referring 2.13, the `Jenkinsfile` contains the definition of test configuration amplification pipeline.

The complete example is available in STAMP GitHub repository, at https://github.com/STAMP-project/e2e-STAMP-CI-demo/tree/master/code/camp-perf-test).

Using the Blue Ocean interface (see https://jenkins.io/projects/blueocean/), the whole execution of pipeline stages is represented in Jenkins current build dashboard as shown in figure 2.14.

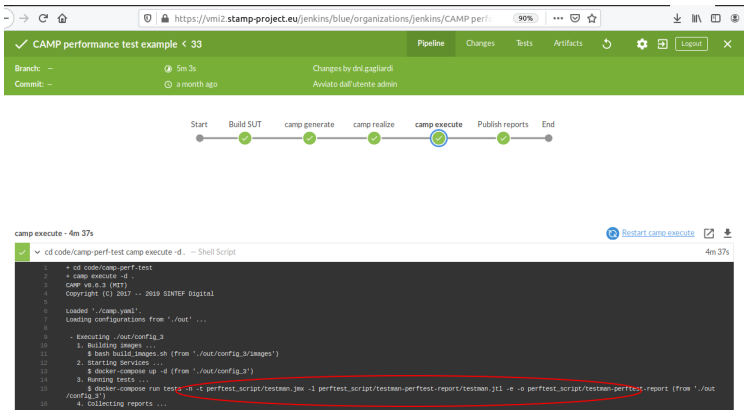


Figure 2.14: Test Configuration amplification pipeline execution for performance tests, shown by Blue Ocean interface

The following listing contains the pipeline tasks in detail:

```
pipeline {

  agent any
  stages {

    stage('Build SUT') {
      steps {
        withMaven(maven: 'MVN3', jdk: 'JDK8') {
          sh '''cd code/camp-perf-test/template/testman
          mvn clean package'''
        }
      }
    }

    stage('camp generate') {
      steps {
        sh ''' cd code/camp-perf-test
        camp generate -d .'''
      }
    }
    stage('camp realize') {
      steps {
        sh ''' cd code/camp-perf-test
        camp realize -d .'''
        zip zipFile: 'testconf.zip', archive: true, dir: 'code/
           camp-perf-test/out', glob: '**/*.yml'
        zip zipFile: 'dockerfiles.zip', archive: true, dir: 'code
           /camp-perf-test/out', glob: '**/Dockerfile'
      }
    }
    stage('camp execute') {
      steps {
        sh ''' cd code/camp-perf-test
        camp execute -d .'''
      }
    }
    stage('Publish reports') {
      steps {
        script {
          sh 'ls -d code/camp-perf-test/out/*/ > reportDirNames.
             txt'
          def reportDirectories = readFile('reportDirNames.txt').
             split("\\r?\\n")
          sh 'rm -f reportDirNames.txt'
          for (i = 0; i < reportDirectories.size(); i++) {
            publishHTML (target: [
              allowMissing: false,
              alwaysLinkToLastBuild: false,
              keepAll: true,
              reportDir: reportDirectories[i] + '/test-reports',
```

```
                reportFiles: 'index.html',
                reportName: reportDirectories[i].tokenize('/').last
                    ()
              ])
            }
          }
        }
      }
    }
  }
}
```

Listing 2.10: Jenkins pipeline with test configuration amplification tasks, applied to performance testing with JMeter

All steps are similar to what has been described in previous section (see 2.2.2.1.1): SUT is built and launched, CAMP generates extra configurations, then executes the tests (performance tests in this case) against them.

Anyway it is worth mentioning that after the execution stage, there is a further stage to publish all HTML reports generated by JMeter for each configuration:

```
    stage('Publish reports') {
      steps {
        script {
          sh 'ls -d code/camp-perf-test/out/*/ > reportDirNames.
              txt'
          def reportDirectories = readFile('reportDirNames.txt').
              split("\\r?\\n")
          sh 'rm -f reportDirNames.txt'
          for (i = 0; i < reportDirectories.size(); i++) {
            publishHTML (target: [
              allowMissing: false,
              alwaysLinkToLastBuild: false,
              keepAll: true,
              reportDir: reportDirectories[i] + '/test-reports',
              reportFiles: 'index.html',
              reportName: reportDirectories[i].tokenize('/').last
                  ()
            ])
          }
        }
      }
    }
```

Listing 2.11: Jenkins pipeline with JMeter HTML reports publishing task

This pipeline fragment iterates over amplified test configurations, retrieves paths containing JMeter HTML reports, and publish them within Jenkins dashboard (figure 2.15).

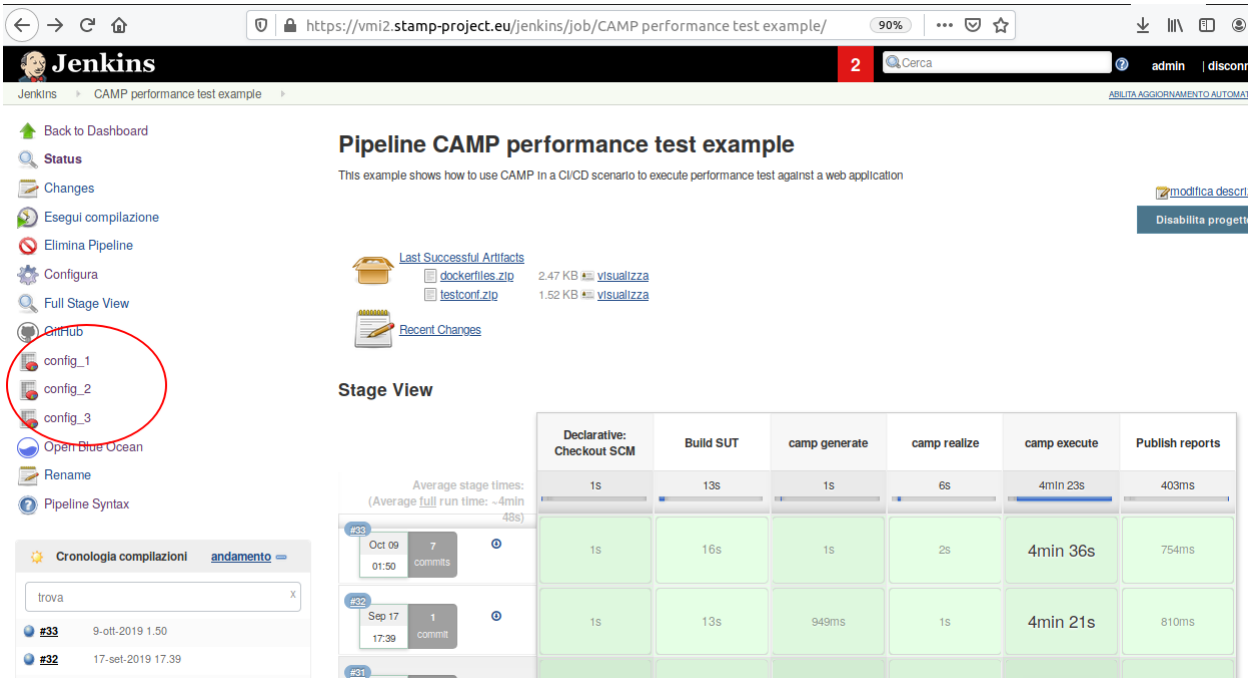


Figure 2.15: JMeter HTML reports available in Jenkins dashboard

Developers can then inspect performance test results, drilling down in each report (figure 2.16).

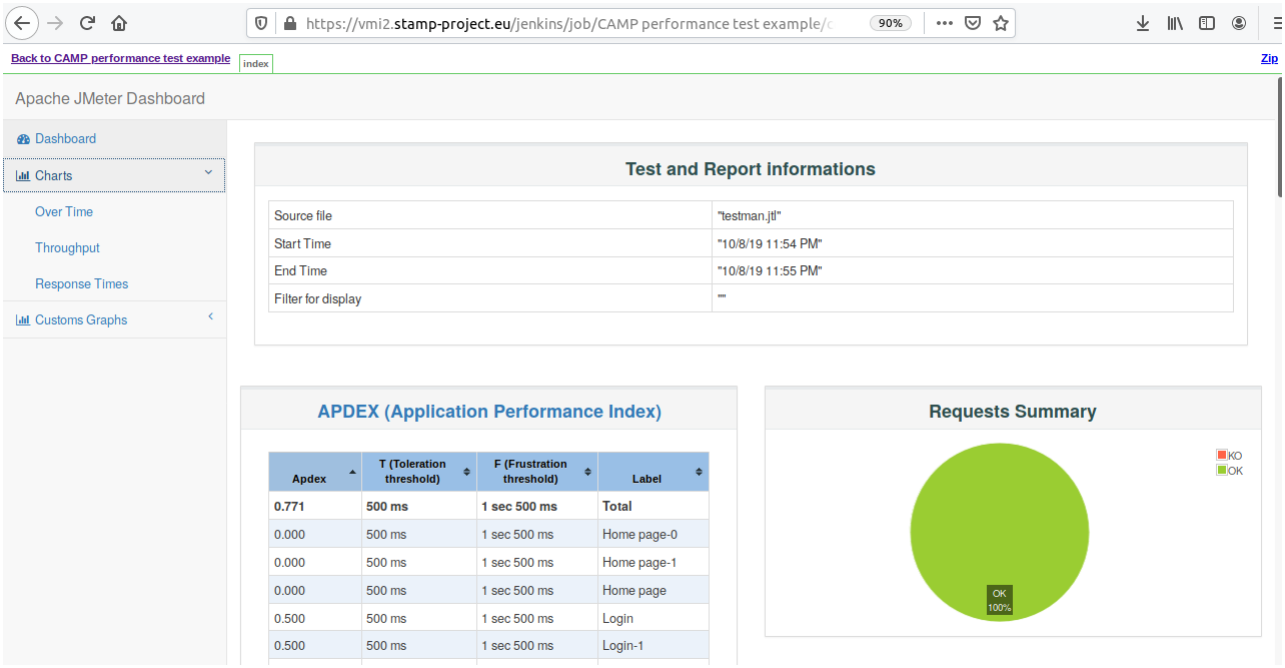


Figure 2.16: Detail pf a single JMeter HTML report

2.2.2.2 Test execution delegated to external test frameworks

In this scenario, CAMP is used only to generate extra test configurations, while an external test framework is needed to execute system tests against them. As described in D2.4 (see section 8.2), a possible approach leverages *Testcontainers framework* (https://www.testcontainers.org/), a library which makes easier to use Docker to perform functional test, by easily allowing to start and stop containers. It performs several background tasks such as waiting for the containers to be started, cleaning up dangling Docker containers, taking screenshots and videos of test executions, supporting Docker in Docker, allowing creation of Docker images on the fly, and more. The following pipeline (available at https://github.com/STAMP-project/lutece-demo-site-forms/blob/camp_output_stored/Jenkinsfile) shows how to automate the whole process:

```
pipeline {

  parameters {
      booleanParam(name: 'config_changed', defaultValue: false )
  }
  agent any
  stages {
  stage('check changelog') {
      steps {
      script {
          env.config_changed = false;
          echo "config_changed value =  ${env.config_changed}"
          def changeLogSets = currentBuild.changeSets
          for (int i = 0; i < changeLogSets.size(); i++) {
            def entries = changeLogSets[i].items
            for (int j = 0; j < entries.length; j++) {
              def entry = entries[j]
              def files = new ArrayList(entry.affectedFiles)
              for (int k = 0; k < files.size(); k++) {
                def file = files[k]
                if (file.path == "camp.yml" || file.path.
                    startsWith("template")){
                  env.config_changed = true;
                }
              }
            }
          }
          echo "config_changed value =  ${env.config_changed}"
        }
      }
    }

    stage('camp generate') {
      when { expression {env.config_changed == 'true'}}
      steps {
        script{
          if (fileExists('out')) {
            sh 'git rm -r out'
          }
```

```
        }
        sh 'camp generate -d . --all'
      }
    }
    stage('camp realize') {
      when { expression {env.config_changed == 'true'}}
      steps {
        sh 'camp realize -d .'
      }
    }
    stage('execute tests') {
      when { expression { env.config_changed == 'false'}}
      steps {
        withMaven(maven: 'MVN3', jdk: 'JDK8') {
          sh '''cd lutece-form-test
          mvn clean test -DcampOutPath="${WORKSPACE}/out"'''
          junit 'lutece-form-test/target/surefire-reports/*.xml'
        }
      }
    }
  }
  environment {
    GIT_URL = sh (script: 'git config remote.origin.url',
       returnStdout: true).trim().replaceAll('https://','')
  }
}
```

Listing 2.12: Jenkins pipeline with test configuration amplification tasks and test execution delegated to Testcontainer framework

As a prerequisite, CAMP needs to be installed in Jenkins (or in dedicated Jenkins agent).

The first stage (**check changelog**) checks whether anything changed in test configurations (using the Jenkins `Changeset` function, and iterating over changed files). The CAMP `generate` and `realize` stages generate new test configurations. The stage `execute tests` executes the tests through the maven action `mvn clean test`. All the execution logic is coded within test source code, which uses TestContainers framework to instantiate each generated test configuration, and Junit 4 Parameterized Tests feature (see https://github.com/junit-team/junit4/wiki/Parameterized-tests) to iterate execution of test case against all generated test configuration. The listing below shows the content of the test case (some `import` related to Java language and JUnit framework have been removed to improve readability):

```
package eu.stamp.testing;

...

import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import org.testcontainers.containers.DockerComposeContainer;
import org.testcontainers.containers.wait.strategy.Wait;

@RunWith(Parameterized.class)
public class LuteceFormTest {
```

```
    private static final String CAMP_OUT_PATH_SYSTEM_PROPERTY = "
       campOutPath";
    private static final String DOCKER_COMPOSE_YML = "docker-
       compose.yml";
    private static final String LUTECE_SITE_FORMS_URL = "http://
       localhost:8081/site-forms-demo/";

    private DockerComposeContainer environment;

    /**
     * get list of directories generated from CAMP
     *
     * @return
     */
    @Parameters(name = "{0}")
    public static Collection<File> data() {
        String campOutPath = System.getProperty(
           CAMP_OUT_PATH_SYSTEM_PROPERTY);
        File[] directories = new File(campOutPath).listFiles(File
           ::isDirectory);
        return Arrays.asList(directories);
    }

    public LuteceFormTest(File configFolder) {
        environment = new DockerComposeContainer(new File(
           configFolder, DOCKER_COMPOSE_YML)).withLocalCompose(
           true);
    }

    @Test
    public void checkServerIsRunning() throws IOException {
        try {
            URL url = new URL(LUTECE_SITE_FORMS_URL);
            HttpURLConnection http = (HttpURLConnection) url.
               openConnection();
            int statusCode = http.getResponseCode();
            assertEquals(200, statusCode);
        } catch (IOException e) {
            fail();
        }

    }

    @Before
    public void startDockerCompose() throws InterruptedException
       {
        environment.start();
        environment.waitingFor("lutece", Wait.forLogMessage("*
           site-forms-demo has finished*", 1));
        Thread.sleep(20000);
```

```
    }

    @After
    public void stopDockerCompose() {
        environment.stop();
    }
}
```

Listing 2.13: Java test case written using JUnit Parameterized Tests and Testcontainers framework

The annotation `@RunWith(Parameterized.class)` marks current test case as a parameterized test case. The method `Collection<File> data()` extracts all folders containing each a different configuration generated by CAMP. It is used by JUnit to instantiate several instances of current test case, one for each configuration, which is in turn passed through test case Constructor `LuteceFormTest(File configFolder)`.

The `configFolder` parameter is used to instantiate the test environment by using the statement `environment = new DockerComposeContainer(new File(configFolder, DOCKER_COMPOSE_YML)).withLocalCompose(true);`. The method annotated with `@Before`, `startDockerCompose ()` starts up the test configuration. This test case check whether the application is up & running or not, calling the home page and verifying that the application returns a 200 code: in general test cases are much more complex than this one.

At the end of the execution, the method `stopDockerCompose()` stops all containers composing the current test configuration: if other test configurations are available, JUnit instantiates a new test case instance by using the next one.

The Test Results section can be accessed by inspecting Jenkins build, as shown in figure 2.17:

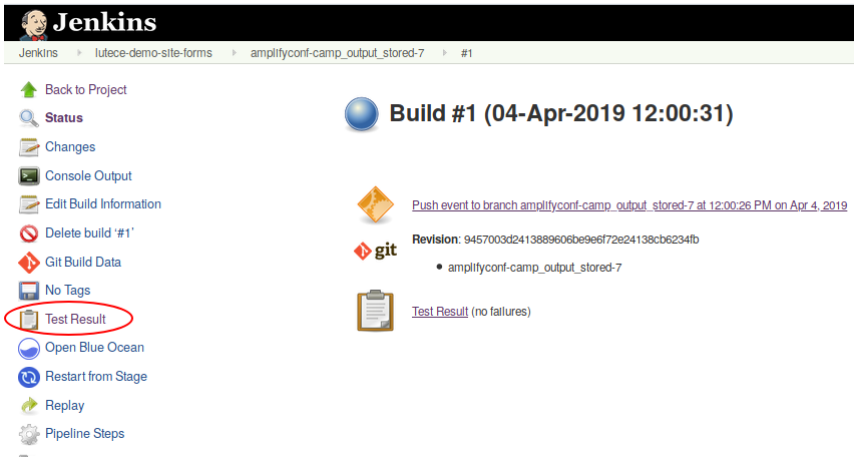


Figure 2.17: Accessing test results in Jenkins

Jenkins provides an overview of all test cases run as in figure 2.18. It is possible to drill down to single test executions, parameterized for each test configuration as in figure 2.19.

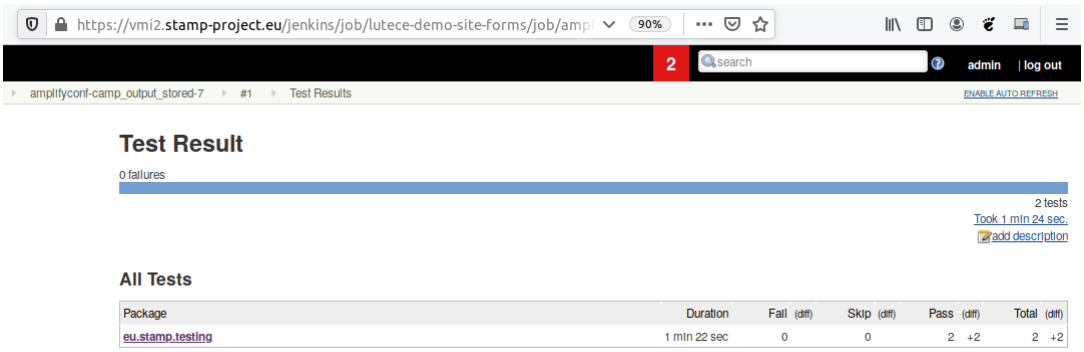


Figure 2.18: Drilling-down in test results in Jenkins

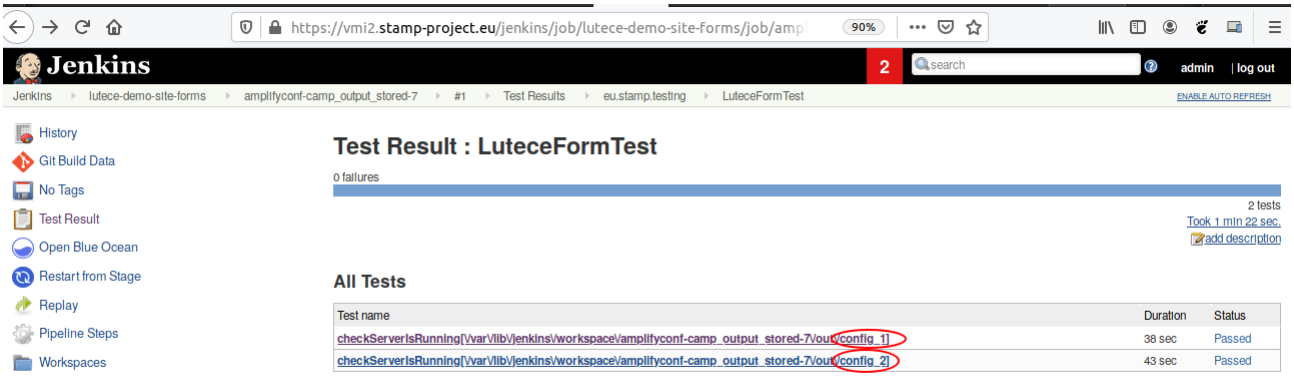


Figure 2.19: Accessing test results for each test configuration in Jenkins

The name of test configurations generated by CAMP will appear beside each test method (in this case `config_1` and `config_2`).

2.2.3 Online Test amplification

This section shows the STAMP online test amplification features which can be adopted in DevOps processes to speed-up the bug investigation phase. Figure 2.20 shows the detailed online test amplification reference scenario.

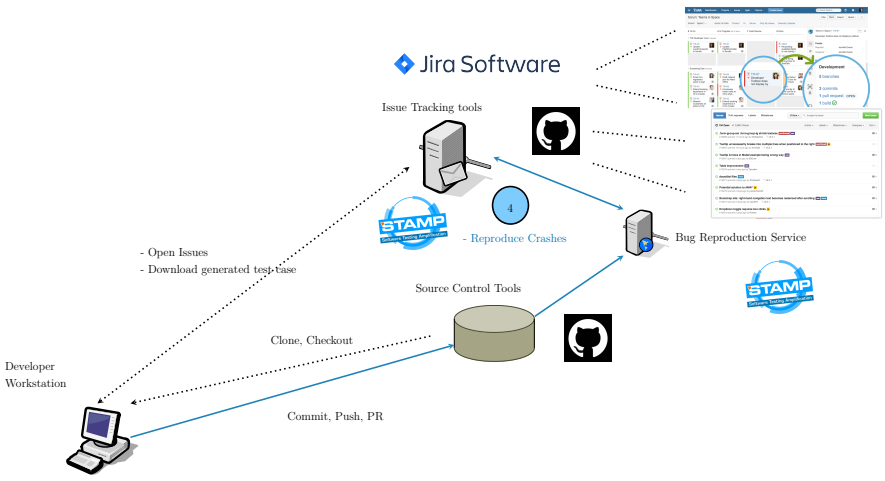


Figure 2.20: STAMP CI Online Test Amplification reference scenario

Botsing is available as a micro-service (in form of a Spring Boot App), and exposes a rest API which can be used by supported issue trackers to send production logs attached to issues by software users.

Developers configure issue trackers to communicate with Botsing server in order to integrate in them the automatic crash reproduction feature.

Once a user opens an issue in the issue tracker and attaches a log containing the bug stack-trace, the STAMP plugin for issue tracker automatically sends the stack-trace to the Botsing server .

Every issue tracker request is stored by Botsing Server in a queue messaging system (based on Apache ActiveMQ). A Botsing-based consumer micro-service takes the stack-trace, download from source code repository the dependencies configuration (software binaries and libraries), and executes Botsing through Botsing Maven plugin (https://github.com/STAMP-project/botsing/tree/master/botsing-maven) to reproduce the stack-trace. Specifically the performed steps are the following:

1. the stack-trace is cleaned through Botsing pre-processor (https://github.com/STAMP-project/botsing/tree/master/botsing-preprocessing)
2. Botsing is executed, starting from the highest frame, and iterating on all frames until it finds a test case
3. generated test case is then sent back to issue tracker in form of attachment (or comment, in case the issue tracker doesn't exposes API to manage attachments).

Currently STAMP supports Jira Software and Github Issues: these two tools have been chosen for their wide adoption (both in open-source communities and companies). Jira Software is a leader in the market of ALM tools, and, thanks to its licensing model which gives free licenses to open source projects, is widely adopted in open source communities as well. GitHub is a full-stack development platform, offering also issue tracker features, and it is probably the most popular development platform in the world.

2.2.3.1 Online Test amplification with Jira

Software projects using Jira Software as issue tracker can leverage STAMP automatic crash reproduction features through Botsing Jira plugin (https://github.com/STAMP-project/botsing-jira-plugin), which adds this feature to ordinary Jira tasks. Specifically, it's simply

a matter of adding an attachment containing the stack-trace, and... wait. The person who opens the bug attach the stack trace and marks the issue with "STAMP" label (figure 2.21).

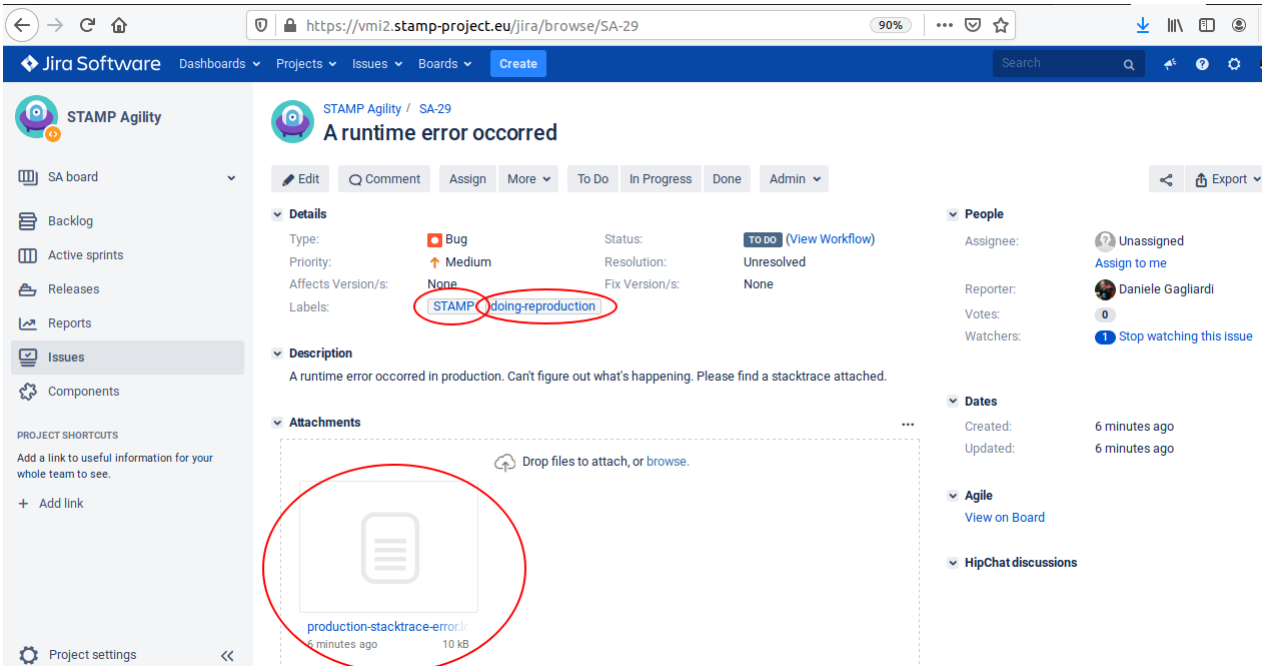


Figure 2.21: Triggering automatic crash reproduction with Jira

Jira Botsing plugin is a listener for issue creation and edit events: when a new issue is created, containing an attachment, and labeled as "STAMP", Jira Botsing plugin execution is triggered, and it sends to the remote Botsing Server an attachment with all needed information to enable Botsing to automatically reproduce the crash with a proper test case. The following pieces of information are needed to start the process:

- software classpath (binaries and libraries)
- target frame.

Moreover, other parameters can be specified in order to fine-tune the search for a test case able to reproduce the crash (see https://stamp-project.github.io/botsing/pages/crashreproduction.html for further details). In order to make the process as simple as possible, software classpath and Botsing parameters are configured as Jira add-ons (**Add-ons -> Other -> Botsing plugin configuration**); in the same section the developer can configure GAV (GroupId-ArtifactId-VersionId) for his software project, in order to enable Botsing Server to download all needed dependencies from remote Maven repositories (figure 2.22).

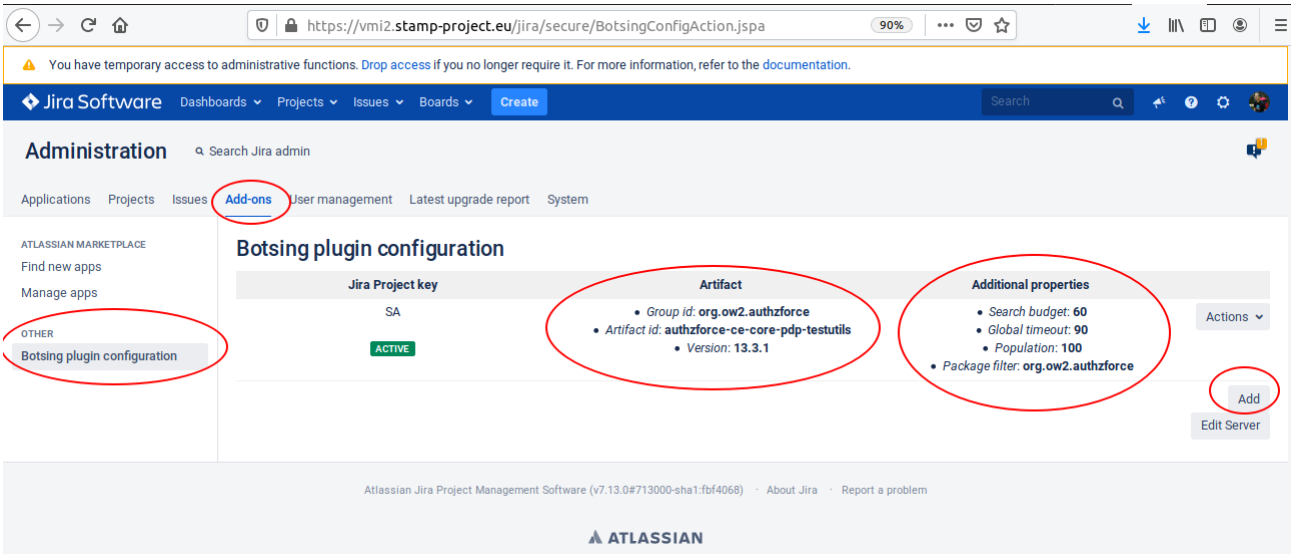


Figure 2.22: Software GAV configuration to let Botsing Server download dependencies for project classpath parameter needed by Botsing

The **Add** enables to configure automatic crash reproductions for other Jira projects.

In the same section, it is possible to configure the address of the remote Botsing Server (figure 2.23).

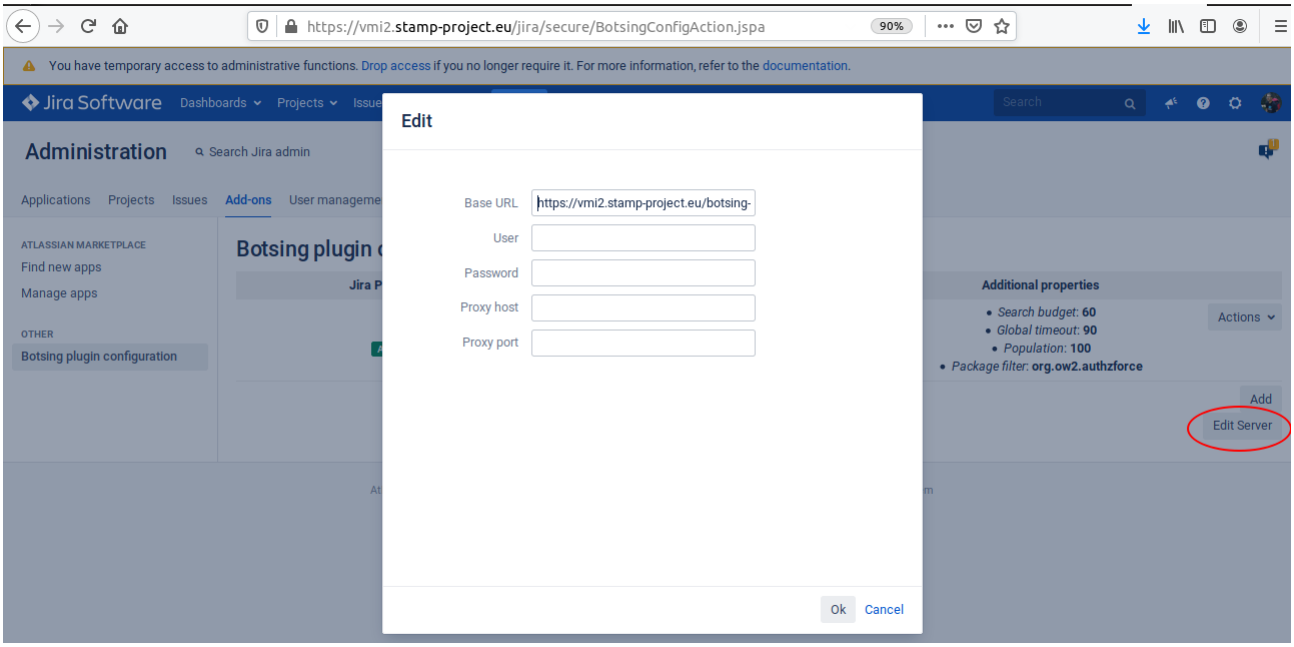


Figure 2.23: Botsing Server configuration in Jira

When Botsing server completes the generation of the new test case, it is forwarded to Jira as an attachment along with a comment containing the reproduction result (figure 2.24).

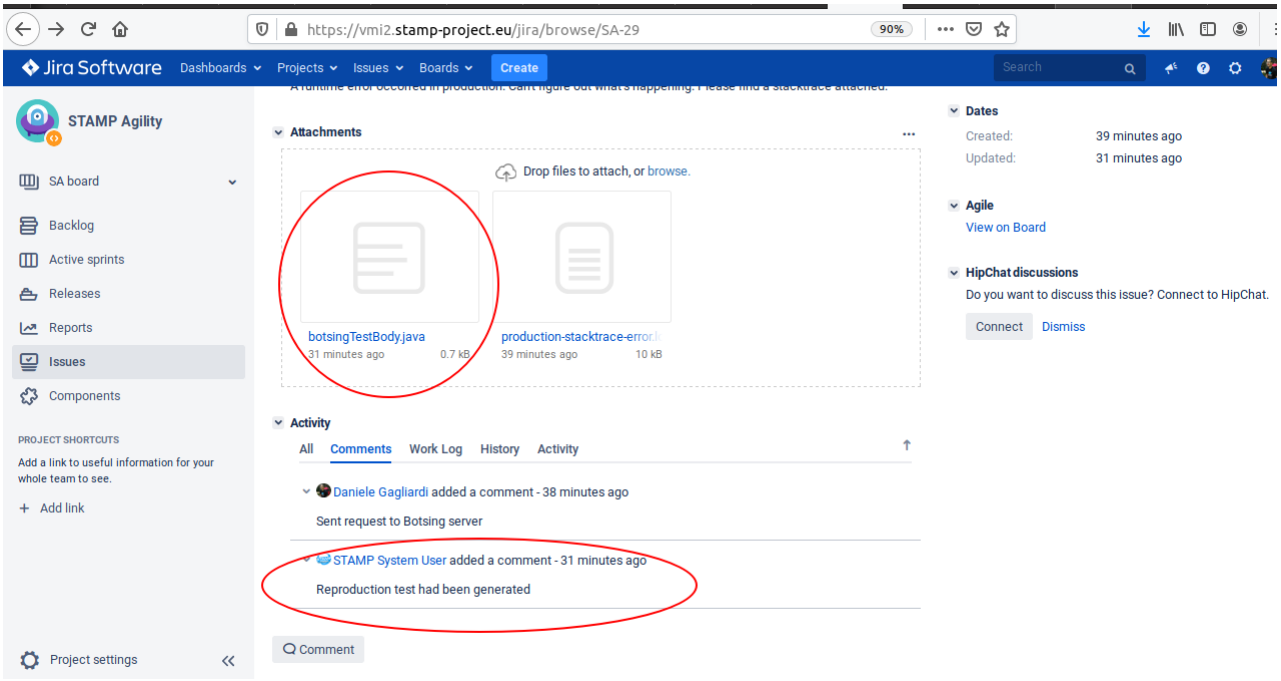


Figure 2.24: Botsing-generated test case attached to Jira issue

The figure 2.25 shows the test case content:

Figure 2.25: Botsing-generated test case content

In case automatic crash reproduction fails, Botsing server includes a comment to Jira issue to notify about the failure (figure 2.26).

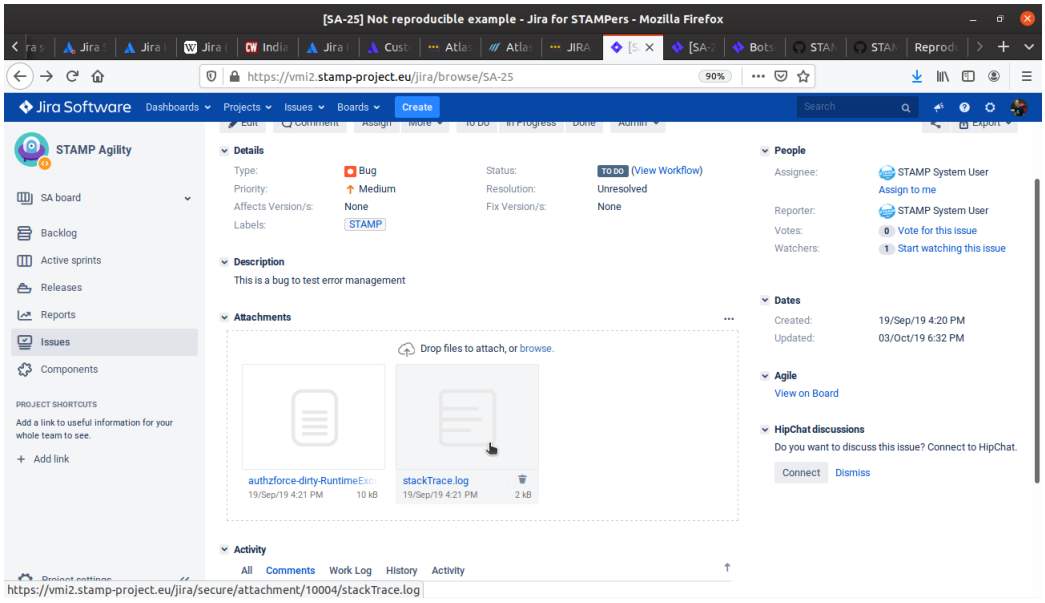


Figure 2.26: Automatic crash reproduction failure

Botsing server attaches also a log error, to let developers investigate what went wrong (figure 2.27).

Figure 2.27: Automatic crash reproduction failure log

2.2.3.2 Online Test amplification with GitHub Issues

Software projects using GitHub can leverage GitHub Issues feature to track enhancements, new features and bugs, of course. STAMP automatic crash reproduction feature can be activated in GitHub by using Botsing Server, configuring it as a GitHub App. In fact it exposes the interface required by GitHub Apps specifications (for details look at https://developer.github.com/apps/about-apps/#about-github-apps). As per Jira Software case, end users and developers can use automatic bug reproduction feature within their ordinary daily tasks. In details, the process consist in opening an ordinary issue having in its body the stack-trace (figure 2.28).

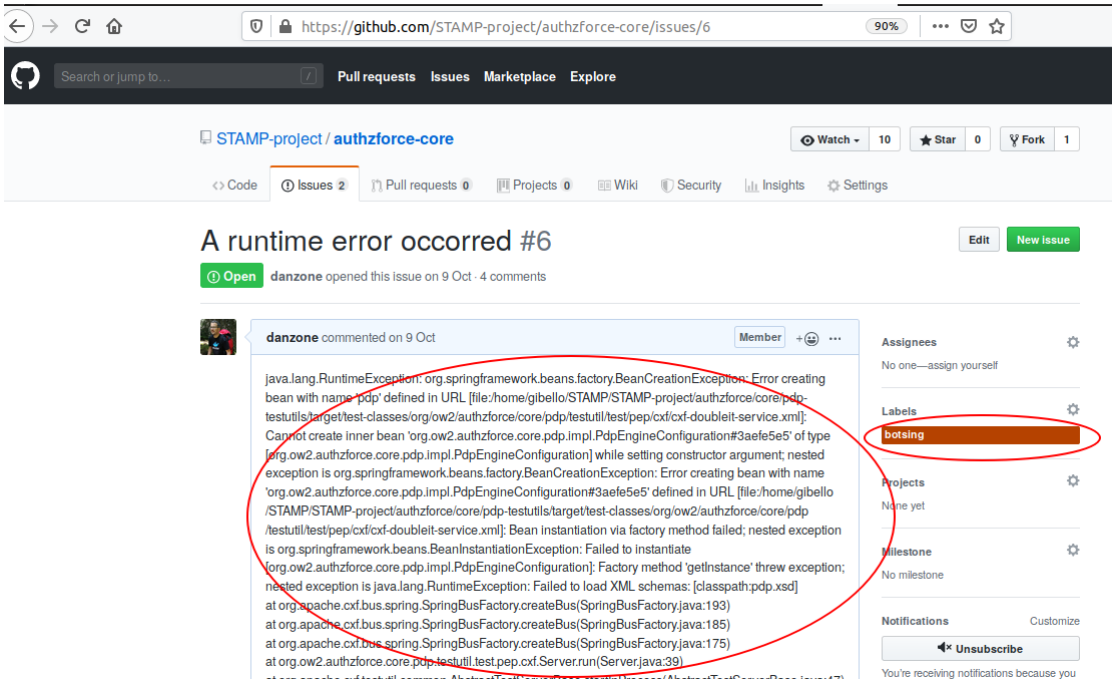


Figure 2.28: Triggering automatic crash reproduction with GitHub Issue

The new issue containing the stack-trace should also be *labeled* as "*Botsing*" in order to be processed by Botsing Github App. Specifically, if the issue meets both the requirements Github triggers the execution of Botsing Github App which, in turn, sends to the remote Botsing Server a JSON object containing the stack trace.

Botsing Server also needs other information to be retrieved on the source code repository (Github) in a text file named `.botsing` (figure 2.29).

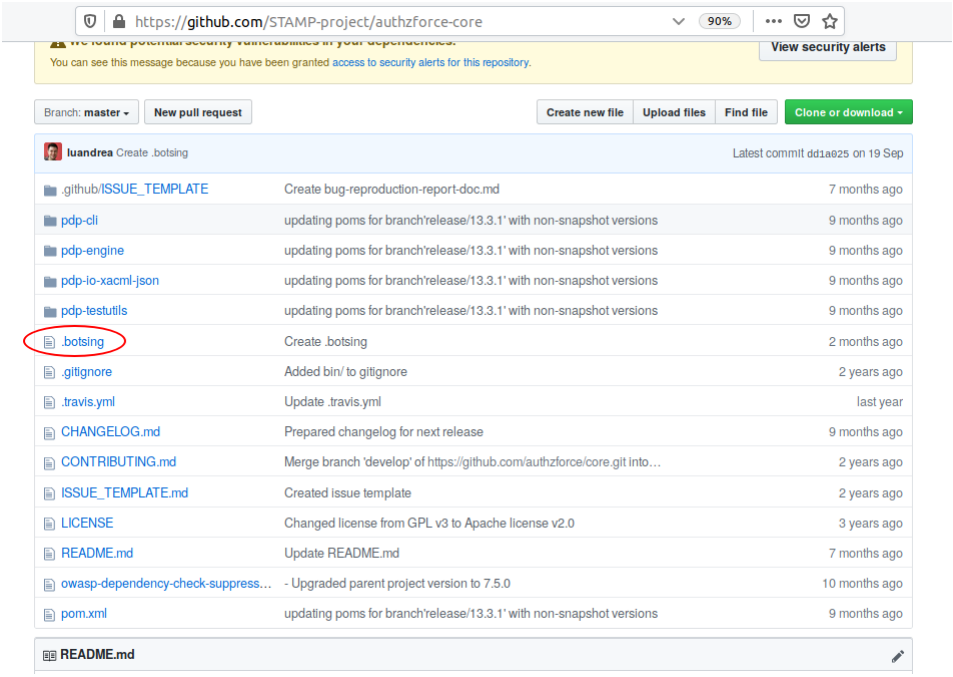


Figure 2.29: Botsing GitHub configuration

This file contains GAV (GroupId-ArtifactId-VersionId) needed by Botsing Maven plugin to retrieve binaries and dependencies from remote Maven repositories, and Botsing specific execution parameters (figure 2.30).

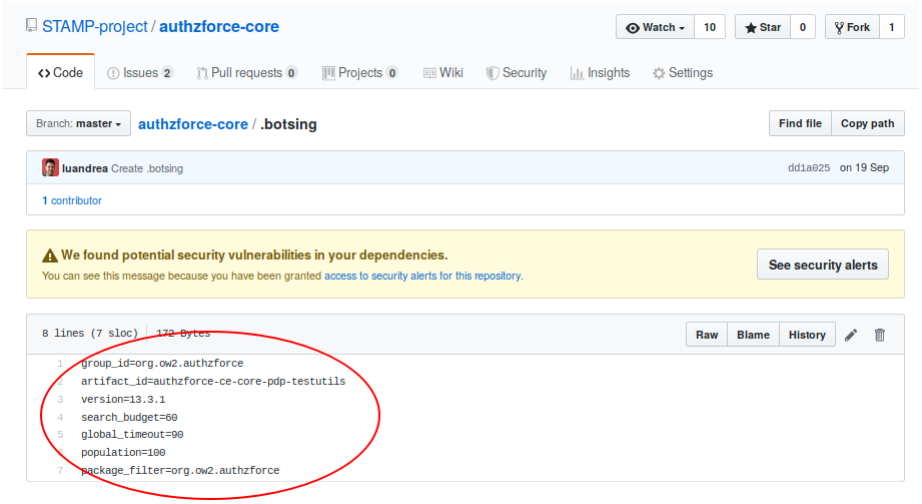


Figure 2.30: GitHub Botsing configuration detail, containing GAV and other Botsing parameters

Configuration of Botsing GitHub App is performed as usual in GitHub Platform:

1. register Botsing GitHub App among the Developer settings of GitHub space (figure 2.31)

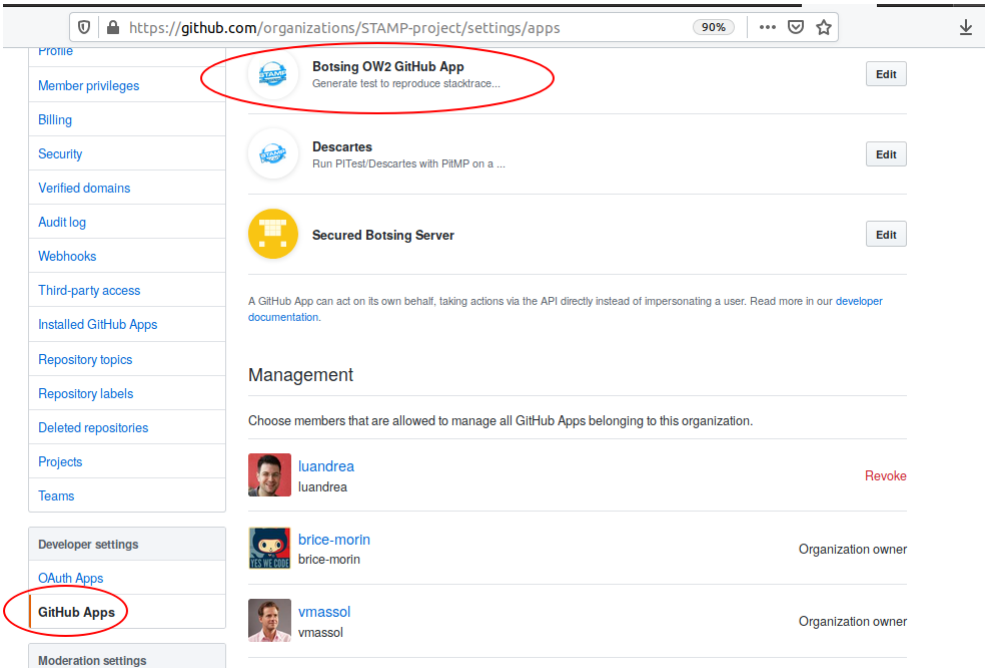


Figure 2.31: Botsing GitHub App configuration

On figure 2.32 configuration details with the web-hook invoked by GitHub once a new issue containing a stack-trace would be created:

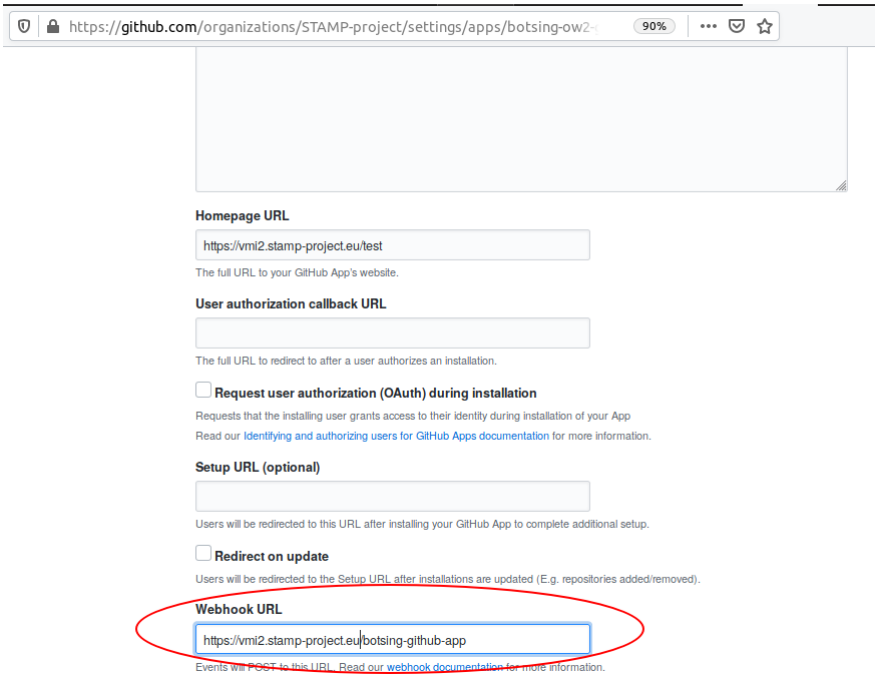


Figure 2.32: Botsing GitHub App configuration detail, containing Web-hook used by GitHub on Issue creation/edit events

2. once configured, the GitHub App should be installed, specifying the associated projects (figure 2.33).

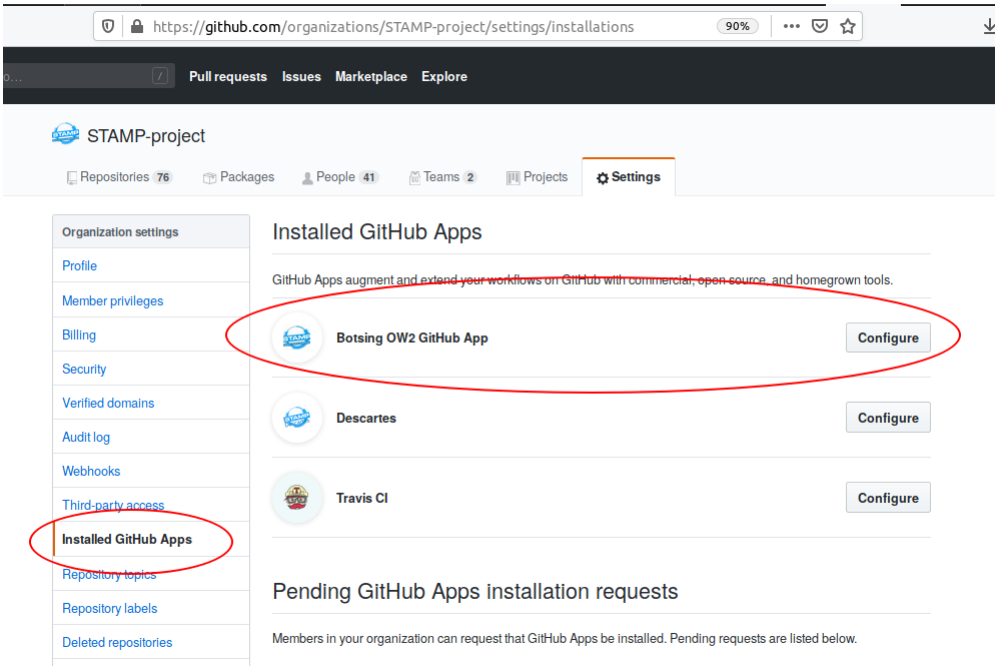


Figure 2.33: Botsing GitHub App activation

Figure 2.34 contains the configuration details and the needed access grants (to execute operations on issues and metadata) for the GitHub App and the associated projects.

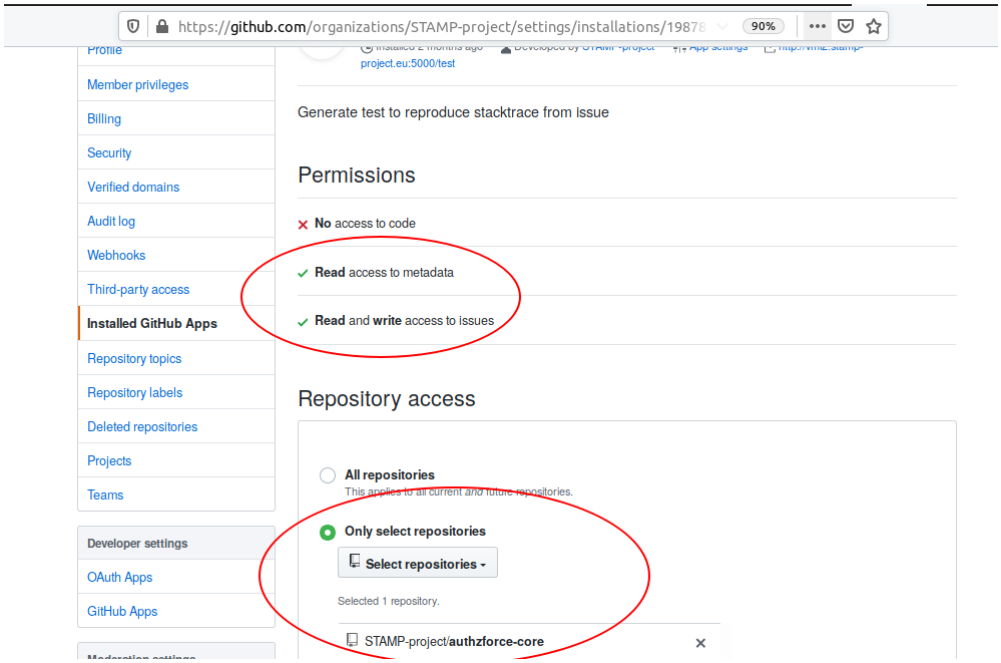


Figure 2.34: Botsing GitHub App activation detail, containing permissions and projects on which it will be activated

When Botsing Server completes the generation of the test case, it sends back to GitHub Issue the result as a comment (figure 2.35).

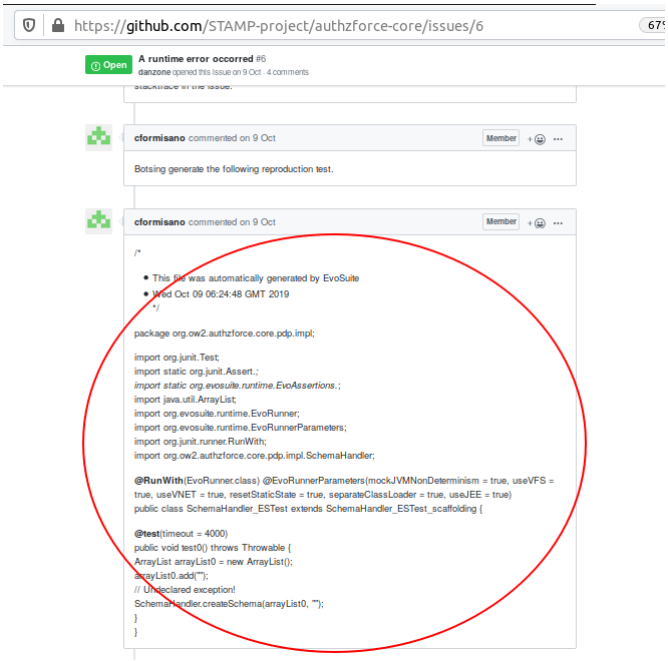


Figure 2.35: Botsing-generated test case added as a comment to GitHub issue

In this example Botsing server has been provided with credentials owned by GitHub account `cformisano`. It is possible to configure Botsing Server with a valid GitHub account.

As per Jira case, if automatic crash reproduction fails, Botsing server will produce a comment to Github issue to notify about the failure (figure 2.36).

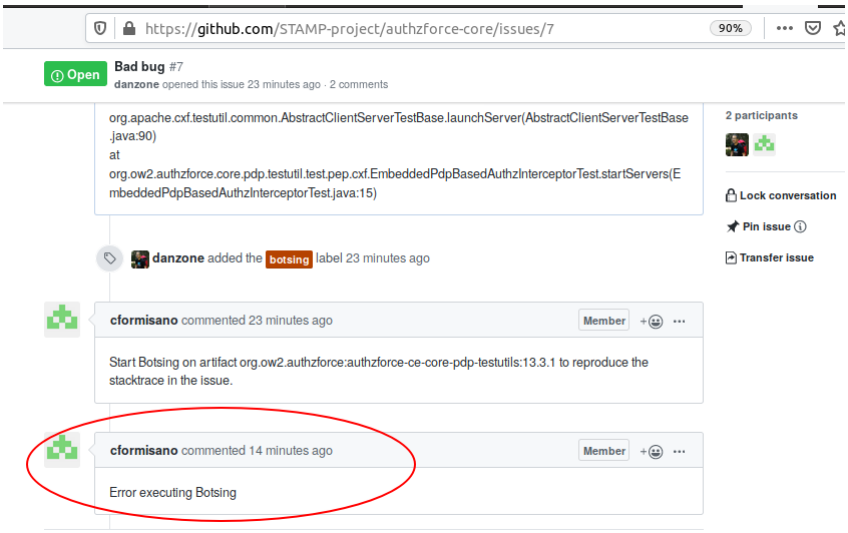


Figure 2.36: Automatic crash reproduction failure

2.3 STAMP CI/CD Architecture

This section provides a brief description of STAMP CI/CD Architecture. In particular, the main components that constitute the STAMP CI/CD scenario, are described along with their interactions. The diagram in figure 2.37 shows STAMP CI/CD architecture.

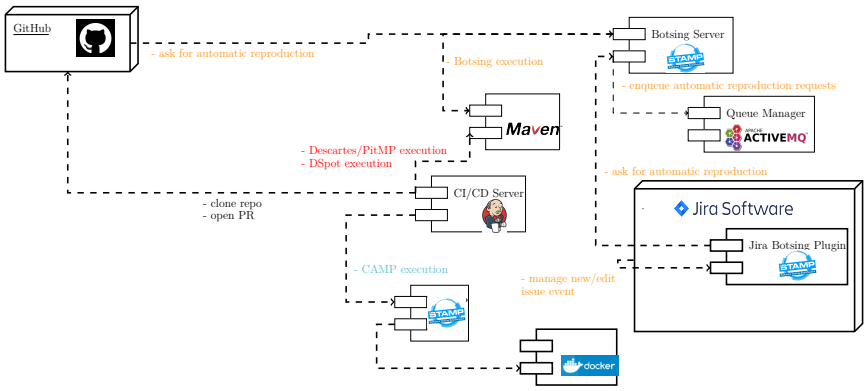


Figure 2.37: STAMP CI/CD architecture

Jenkins CI interacts with several components:

- **(STAMP) Maven plugins** are used within Jenkins pipelines to provide CI processes with Descartes/PitMP and DSpot features. Maven needs to be installed within Jenkins CI (or within Jenkins agents dedicated to test amplification tasks)

- **CAMP** is used within Jenkins pipelines to provide CD processes with test configuration amplification features. It needs to be installed within Jenkins CI, or within Jenkins agents dedicated to test configuration amplification tasks. The CAMP modules devoted to executing functional tests and performance tests with JMeter are part of CAMP official distribution. For this reason, except for Python3, Docker, Docker compose and Z3 solver (requirements for CAMP), every other CAMP pre-requirement is needed
- **Docker (and Docker compose)**, needed by CAMP. It needs to be installed within Jenkins CI, or within Jenkins agents dedicated to test configuration amplification tasks
- **Botsing Server** is a standalone app (a Spring Boot App), including a queue management system (*ActiveMQ*) and a micro-service which executes automatic crash reproduction: it depends in turns on Botsing Maven plugin (which comprises both Botsing pre-processing and Botsing crash reproduction features)
- **Jira Software**, issue tracker equipped with Botsing Jira plugin, to provide Jira users with automatic crash reproduction features integrated in Jira itself
- **GitHub**, source code repository and issue tracker integrated with Botsing Server as a GitHub App, to provide GitHub users with automatic crash reproduction features integrated in GitHub itself.

Chapter 3

STAMP ecosystem

3.1 Plugins

During past year all Maven and Gradle plugins were updated to be compliant with the latest versions of STAMP tools, as reported also in Deliverable 1.4, section 5.3. Maven plugins opened the way to introduce STAMP features within ordinary CI/CD processes, modeled by Jenkins pipelines.

3.1.1 DSpot Maven Plugin

DSpot Maven Plugin plugin lets developer integrate unit test amplification in the Maven build process. Unit test amplification can be obtained executing Maven and specifying the goal `amplify-unit-tests`.

Executing it as follows:

`mvn eu.stamp-project:dspot-maven:amplify-unit-tests`

it will amplify existing JUnit tests to kill detected mutants with Descartes.

An alternate approach consists in amplifying existing JUnit test cases to improve coverage. In this case it is necessary to pass the code coverage test criterion parameter:

`mvn eu.stamp-project:dspot-maven:amplify-unit-tests -Dtest-criterion=JacocoCoverageSelector`

All DSpot options described at https://github.com/STAMP-project/dspot#command-line-options can be passed through command line by prefixing the desired option with -D. For example:

`mvn eu.stamp-project:dspot-maven:amplify-unit-tests -Dtest=my.package.TestClass -Dcases=testMethod`

Alternatively, all options can be specified within the `pom.xml` file, in the `plugins` section of the build:

```
<plugin>
    <groupId>eu.stamp-project</groupId>
    <artifactId>dspot-maven</artifactId>
    <version>LATEST</version>
    <configuration>
        <!-- your configuration -->
    </configuration>
</plugin>
```

Listing 3.1: Maven configuration with DSpot parameters specified in pom.xml file

replacing LATEST with the latest DSpot version number available at Maven central.

DSpot Maven plugin can be used also with Maven multi-module projects. In this case the preferred way to use it is to configure DSpot options in the highest `pom.xml` and use the dedicated command-line option `-DtargetModule=<Module_to_amplify>` in order to perform test amplification only in the desired module.

After setting up your `pom.xml` with desired DSpot configuration, developer needs simply to run following command to amplify existing test cases according to given configuration:

```
mvn dspot:amplify-unit-tests
```

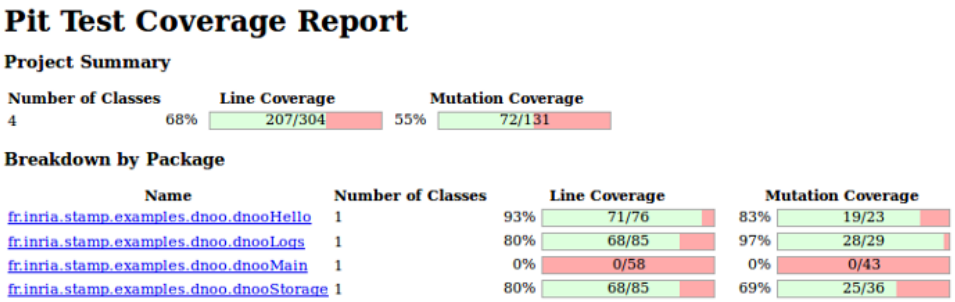
3.1.2 PitMP - PIT for Multi-module Project

PitMP (PITest for Multi-module Project) is a Maven plugin able to run PITest on multi-module projects. PITest is a mutation testing system for Java applications, which allows you to evaluate the quality of your test suites. Since Descartes is a PITest plugin, PitMP also provide a full support to run Descartes and its specific goals.

3.1.2.1 PitMP Output

PITest produces a report that includes:

- a summary of line coverage and mutation coverage scores:



Pit Test Coverage Report

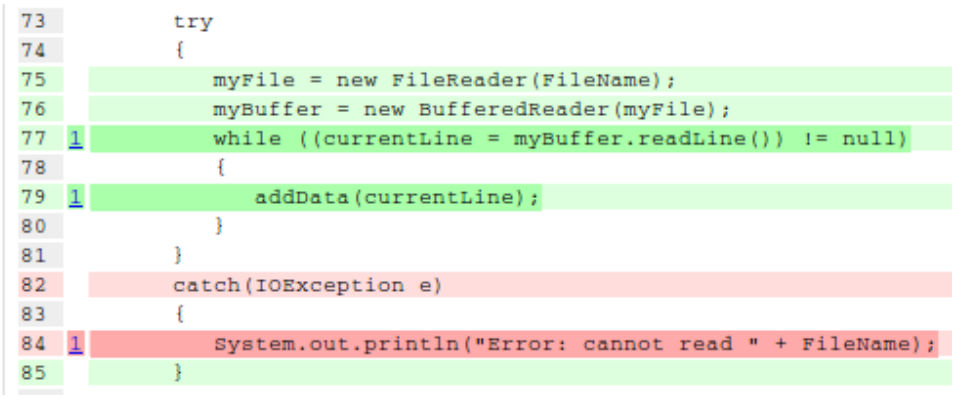
Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
4	68%	207/304	55%	72/131

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
fr.inria.stamp.examples.dnoo.dnooHello	1	93%	71/76	83%	19/23
fr.inria.stamp.examples.dnoo.dnooLogs	1	80%	68/85	97%	28/29
fr.inria.stamp.examples.dnoo.dnooMain	1	0%	0/58	0%	0/43
fr.inria.stamp.examples.dnoo.dnooStorage	1	80%	68/85	69%	25/36

- a detail report for each class combining line coverage and mutation coverage information:



```
73          try
74          {
75              myFile = new FileReader(FileName);
76              myBuffer = new BufferedReader(myFile);
77  1           while ((currentLine = myBuffer.readLine()) != null)
78              {
79  1               addData(currentLine);
80              }
81          }
82          catch(IOException e)
83          {
84  1           System.out.println("Error: cannot read " + FileName);
85          }
```

Light green shows line coverage, dark green shows mutation coverage.

Light pink show lack of line coverage, dark pink shows lack of mutation coverage.

3.1.2.2 Running PitMP on your project

PitMP is available in Maven central, so it allows you to run PITest or Descartes using a command line, without modifying your pom.xml file.

- Go to the project on which you want to apply PITest
- Compile your project
  `mvn install`
- Run PITest on your multi module project :-)
  `mvn eu.stamp-project:pitmp-maven-plugin:run`

3.1.2.3 Running Descartes

If you want to run Descartes:
`mvn eu.stamp-project:pitmp-maven-plugin:descartes`
If you want to configure Descartes, add to your root project pom.xml, in the section <plugins>:

```
<plugin>
  <groupId>eu.stamp</groupId>
  <artifactId>pitmp-maven-plugin</artifactId>
  <version>release.you.want</version>
  <!-- list all the packages of the project that contain
     classes you want to be mutated    -->
  <configuration>
    <targetClasses>
      <param>a.package.of.classes*</param>
      <param>another.package.of.classes*</param>
    </targetClasses>
    <mutationEngine>descartes</mutationEngine>
  </configuration>
</plugin>
```

For complete instructions about Descartes see the Descartes GitHub.

3.1.2.4 Configure PitMP

You can configure your project in the root pom.xml, in the section <plugins>:

```
<plugin>
  <groupId>eu.stamp</groupId>
  <artifactId>pitmp-maven-plugin</artifactId>
  <version>release.you.want</version>
  <!-- List all the packages of the project that contain
     classes you want
       to be mutated.
       All \pit's properties can be used.
  -->
  <configuration>
    <targetClasses>
      <param>a.package.of.classes*</param>
      <param>another.package.of.classes*</param>
    </targetClasses>
  </configuration>
</plugin>
```

3.1.2.5 PitMP properties

- **targetModules**: to run PITest only on specified modules, this attribute filters directories where to run PITest, not classes to be mutated. You can use the property **targetModules** in the pom.xml:

```
<targetModules>
  <param>yourFirstModule</param>
  <param>anotherModule</param>
</targetModules>
```

  or on the command line, use:

```
mvn "-DtargetModules=yourFirstModule,anotherModule" pitmp:run
```

  Running PitMP from a module directory will NOT work.

- **skippedModules**: to skip specified modules when running PITest, this attribute filters directories where to run PITest, not classes to be mutated. You can use the property **skippedModules** in the pom.xml:

```
<skippedModules>
  <param>aModuleToSkip</param>
  <param>anotherModuleToSkip</param>
</skippedModules>
```

  or on the command line, use:

```
mvn "-DtargetModules=aModuleToSkip,anotherModuleToSkip" pitmp:run
```

- **targetDependencies**: take only into account classes of targetDependencies, i.e. only code in targetDependencies will be mutated; it impacts PITest's targetClasses Note that only targetDependencies shall contains only modules of the project

- **ignoredDependencies**: ignore classes of ignoredDependencies, i.e. code in targetDependencies will not be mutated; it impacts PITest's targetClasses If a module is both in targetDependencies and ignoredDependencies, it will be ignored.

- **continueFromModule**: to run PITest starting from a given project (because continuing an aborted execution with Maven -rf is not working)

```
<continueFromModule>aModule</continueFromModule>
```

- **pseudoTestedThreshold** and **partiallyTestedThreshold**: If you want to check the number of Pseudo Tested Methods and/or Partially Tested Methods, you can add specific thresholds **pseudoTestedThreshold** and/or **partiallyTestedThreshold** in the configuration:

```
<plugin>
  <groupId>eu.stamp</groupId>
  <artifactId>pitmp-maven-plugin</artifactId>
  <version>release.you.want</version>
  <configuration>
    <!-- Check Pseudo/Partially Tested Methods -->
    <pseudoTestedThreshold>1</pseudoTestedThreshold>
    <partiallyTestedThreshold>1</partiallyTestedThreshold
       >
    <targetClasses>
      <param>a.package.of.classes*</param>
      <param>another.package.of.classes*</param>
    </targetClasses>
    <mutationEngine>descartes</mutationEngine>
  </configuration>
</plugin>
```

- **mutationThreshold** and **coverageThreshold**: The plugin can break the build, when the mutation score and/or line coverage is considered too poor. The example below breaks the build if mutation score is less than 40 or line coverage less than 60 , according to **mutationThreshold** and **coverageThreshold** options:

```
<plugin>
  <groupId>eu.stamp-project</groupId>
  <artifactId>pitmp-maven-plugin</artifactId>
  <version>release.you.want</version>
  <configuration>
    <mutationEngine>descartes</mutationEngine>
    <skip>false</skip>
    <failWhenNoMutations>false</failWhenNoMutations>
    <mutationThreshold>40</mutationThreshold>
    <coverageThreshold>60</coverageThreshold>
  </configuration>
</plugin>
```

3.1.2.6 PitMP contributor's guide

PitMP is a Maven plugin able to run PITest on multi-module projects. The main difference is that PitMP take into account the dependencies between modules of the same project. PitMP runs the test suite as PITest does. The tool just extends the list of paths containing the classes to mutate. If a module declares a set of test cases, then PitMP will group all the classes from the dependencies of this module and will instruct PITest to create all possible mutants from them. From a given set of test cases the scope of PITest will be restricted to the module, while the scope of PitMP will expand to the entire project.

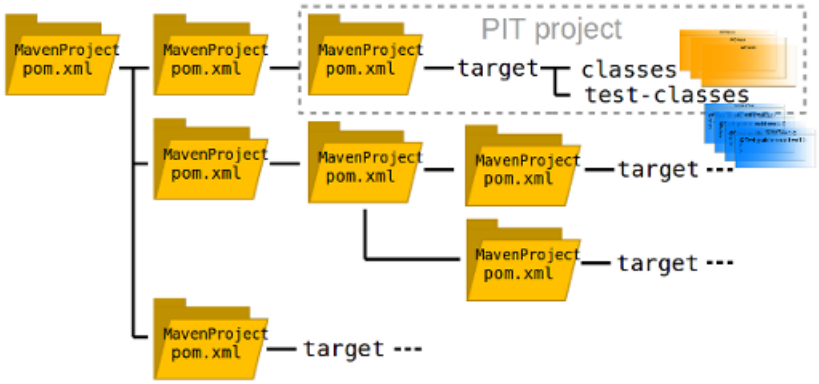


Figure 3.1: PITest project vs PitMP project (test classes are blue and classes to be mutated are orange)

Figure 3.1 shows the PITest scope in grey dotted line and PitMP scope in blue dotted line for a project. PitMP extends PITest, it does not rewrite any feature, so all the properties of PITest can be used. PitMP has only some specific properties to handle test suites to be ran and classes to be mutated. This choice of implementation allows to use all PITest features, including future improvements.

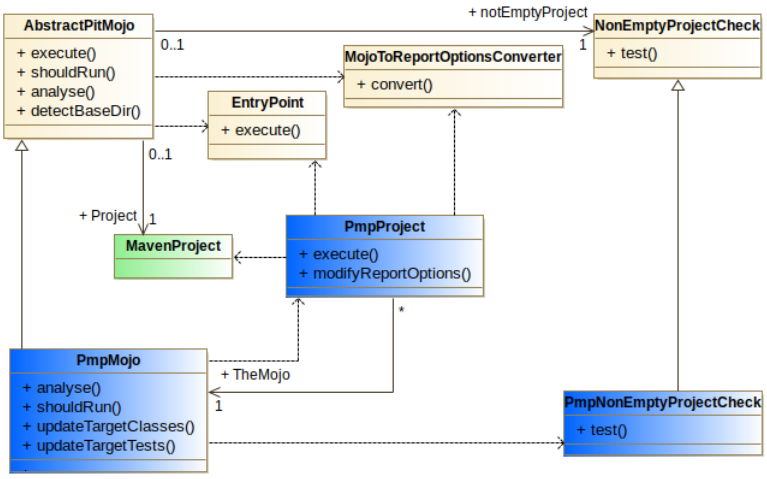


Figure 3.2: pitmp-maven-plugin classes

As shown in figure 3.2 PitMP classes (blue) use the Maven API (green) and inherits from the PITest Mojo (yellow). A Mojo is a Java class that implements a new goal or task in Maven.

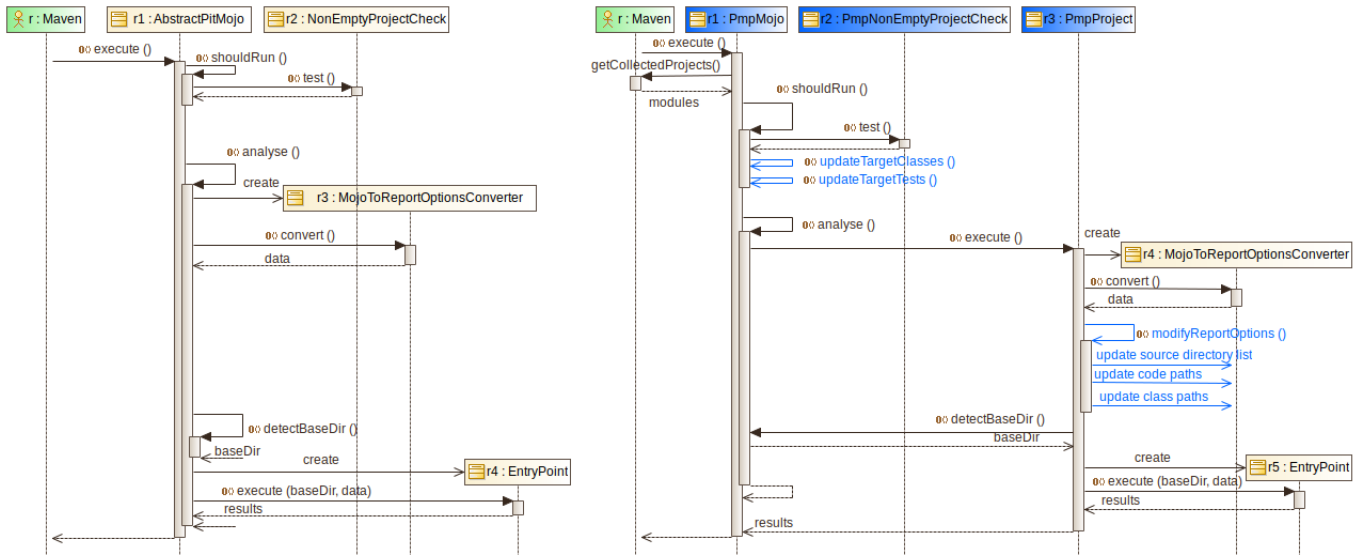


Figure 3.3: PITest Maven plugin vs PitMP execution

Both executions use the same PITest operations, but PitMP modifies the properties that handle the classes to be mutated and the tests to be executed. The operations shown in blue in Figure 3.3 are the ones redefined by PitMP. The concrete properties are:

- **target classes** are the classes to be mutated, if you don't specify target classes, all classes found in code paths can be mutated
- **target tests** are the tests to be executed, if you don't specify target tests, all tests of the baseDir will be run
- **source directories** are used to generate reports
- **code paths** is used to look for code of classes to be mutated
- **class path** is the regular Java *classpath*.

3.1.3 Botsing Maven plugin

Botsing Maven plugin lets developer run Botsing as a Maven specific goal. Botsing Maven plugin retrieves the project classpath from the `pom.xml` file, and the developer is only required to provide the log containing the error stacktrace and the target frame (the parameter to tell Botsing how many lines of the stacktrace to replicate):

```
mvn eu.stamp-project:botsing-maven:botsing -Dcrash_log=ACC-474/ACC-474.log
-Dtarget_frame=2
```

Botsing parameters are passed as ordinary Java parameters, with the prefix `-D`.

To have more information on available parameters, reader can refer to Botsing companion website at https://stamp-project.github.io/botsing/pages/crashreproduction.html.

Dependencies are fundamental to run Botsing so it can reproduce the stacktrace. Botsing Maven plugin has three ways to specify where to find dependencies:

1. from `pom.xml`
2. specifying a Maven artifact
3. from a local folder

3.1.3.1 Dependencies from `pom.xml`

Botsing Maven plugin will read the `pom.xml` in the current folder and find the dependencies specified in it. This will be the default behavior if none else has been specified.

3.1.3.2 Dependencies from artifact

Botsing Maven plugin will search in the Maven repository for current artifact for which crash reproduction is required, and all needed dependencies.

Below an example of Botsing Maven execution with artifact information:

```
mvn eu.stamp-project:botsing-maven:botsing -Dcrash_log=ACC-474.log
-Dmax_target_frame=2 -Dgroup_id=org.apache.commons
-Dartifact_id=commons-collections4 -Dversion=4.0
```

3.1.3.3 Dependencies from folder

Botsing Maven plugin will search in the specified folder and gets all the libraries inside it.

Below an example of Botsing Maven execution with project classpath available through binaries stored in a local folder (`lib` folder):

```
mvn eu.stamp-project:botsing-maven:botsing -Dcrash_log=ACC-474/ACC-474.log
-Dtarget_frame=2 -Dproject_cp=lib
```

3.1.3.4 Target frame options

`target_frame` lets developer specify how many rows of the stacktrace Botsing has to reproduce. Botsing Maven plugin has three ways to specify it:

1. directly using `target_frame` (e.g. `-Dtarget_frame=2`)
2. specifying the maximum value (e.g. `-Dmax_target_frame=2`), in this case it will start from the maximum value provided and decrease it until a reproduction test have been found
3. reading it from the maximum rows of the stacktrace, no parameter for the target frame should be provided

3.1.3.5 Common behavior goal

During past year Common Behavior feature has been introduced in Botsing Maven plugin, adding the specific goal `common-behavior`. Thanks to this Maven goal, it is now possible to generate automatically new test cases observing the behavior of the running software. For example, following command:

```
mvn eu.stamp-project:botsing-maven:common-behavior -Dproject_-
cp=target/classes -Dproject_prefix=org.ow2.authzforce -Dout_-
dir=results/authzforce
-Dclass=org.ow2.authzforce.core.pdp.impl.PdpBean -Dsearch_budget=50
```

will generate new test cases after having observed the behavior of software.

This new feature eventually supports STAMP tool RAMP (Runtime AMPlifier, see https://github.com/STAMP-project/evosuite-ramp) usage.

For more information about this, reader can check documentation at https://github.com/STAMP-project/botsing/tree/master/botsing-maven#how-to-run-common-behavior-maven-plugin, and then follow the tutorial available at RAMP Model Seeding tutorial.

3.1.3.6 Help goal

To view a list of all the parameters that can be used, the goal `help` has been developed. It provides developer with all available goals and information about usage:

```
mvn eu.stamp-project:botsing-maven:help -Ddetail=true -Dgoal=botsing
```

3.1.4 Botsing Gradle plugin

Botsing runner has three mandatory parameters:

- The stack trace to reproduce through the generated test
- The target frame
- The project and its dependencies binaries path

Tests executed with Botsing by use cases showed that it is too much time consuming to gather the good mandatory parameters for each Botsing run. For example, the ActiveEon scheduling project has 545 dependencies to be listed. This cannot be handled in a manual way in production, and it has to be automated.

Projects in production use build tools to handle automatically the dependencies. The first solution to gather the project dependencies was to write a custom task to store the dependencies in a file that will be used by Botsing as a parameter. The drawback of this solution was that the dependencies list must be regenerated each time a dependency is changed. The first aim of the Botsing plugin is to use directly the build tool to run Botsing, providing directly the required parameters.

The first version of Botsing Gradle plugin (see Botsing Gradle repository in STAMP GitHub official space) required to provide the stack trace path and the target frame as the mandatory parameters. The dependencies list was automatically generated from the information provided in `build.gradle`.

After testing the fist Botsing Gradle plugin version, the feedback was that it would be better to download the dependencies list and the binaries directly from maven instead of using the local binaries. Downloading them directly from Maven repository would help to test several software versions. In the first version, testing Botsing on several versions required to use a version controller to update the local source code to the another version. Moreover, thanks to ActiveEon experiments, it has been highlighted that it was difficult to find the correct target frame. As a consequence, a decision has been taken to implement a mechanism to decrease the target frame until Botsing execution works with the selected target frame.

The second version of Botsing Gradle plugin enables to execute Botsing from local binaries or from Maven binaries. It decreases the target frame after each run until it succeeded to find a correct target frame. Selecting the Botsing version was also implemented in order to ease the upgrade of Botsing. Providing Botsing version enables to download a specific version of Botsing and to run it against the provided configuration. Three Botsing optional parameters are also supported:

- The `output` parameter enables to choose where the output will be generated.
- The `searchBudget` parameter enables to specify an additional parameter in format.
- The `population` parameter indicates the number of random unit tests that will be generated.

3.1.5 DSpot and Descartes Jenkins integration

As already reported in Deliverable 1.4, section 5.3.3, the development of two Jenkins plugins able to visualize DSpot and Descartes specific reports has been completed and finalized.

3.1.5.1 STAMP Descartes Jenkins plugin

Descartes Jenkins plugin was developed during 2018. Past year has been dedicated to bugfixing, updating with new versions of Descartes and supporting partners in plugin configuration and usage.

3.1.5.2 STAMP DSpot Jenkins plugin

DSpot Jenkins plugin development started at the end of 2018 with the goal to provide developers with a test amplification dashboard to inspect and possibly download amplified test cases, supporting both freestyle jobs and pipelines.

Using it within a Jenkins freestyle job simply implies to add a build action to run DSpot, as shown in figure 3.4.

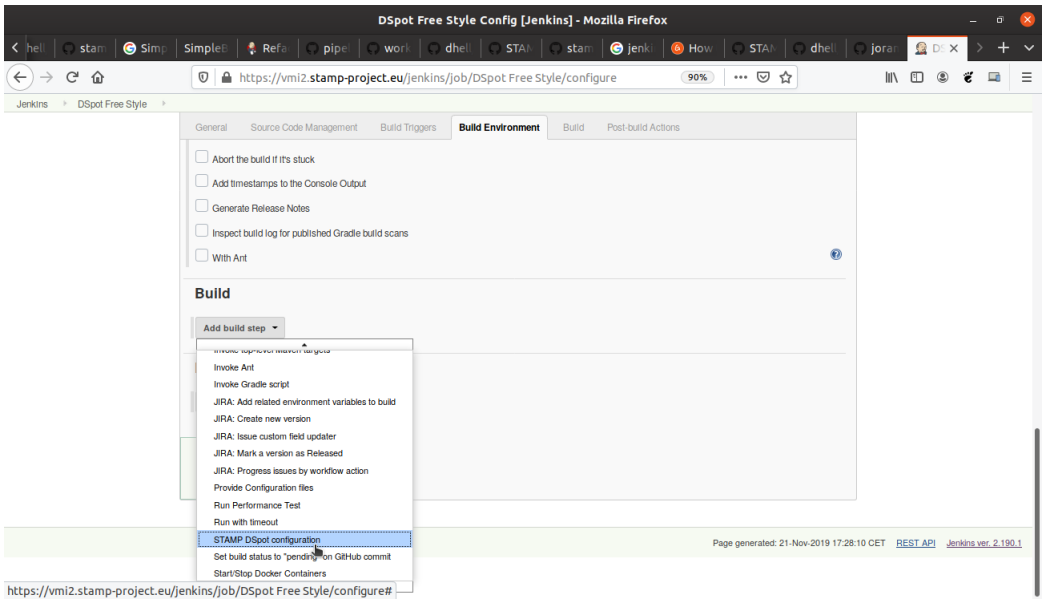


Figure 3.4: Configuring DSpot Jenkins plugin in a freestyle job

DSpot build action named "STAMP DSpot configuration" has several options to properly configure DSpot, as shown in figure 3.5.

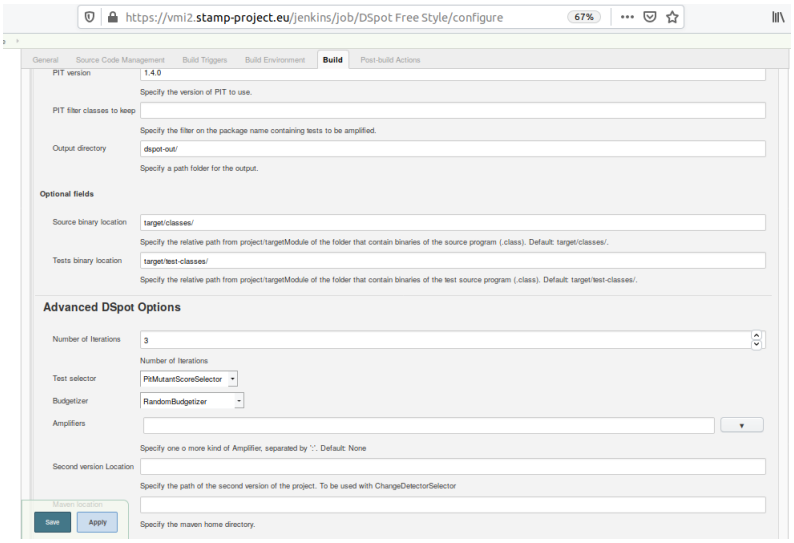


Figure 3.5: Configuring DSpot Jenkins plugin in a freestyle job: advanced configuration

Then it is possible to add a post-build action to show DSpot reports, as shown in figure 3.6.

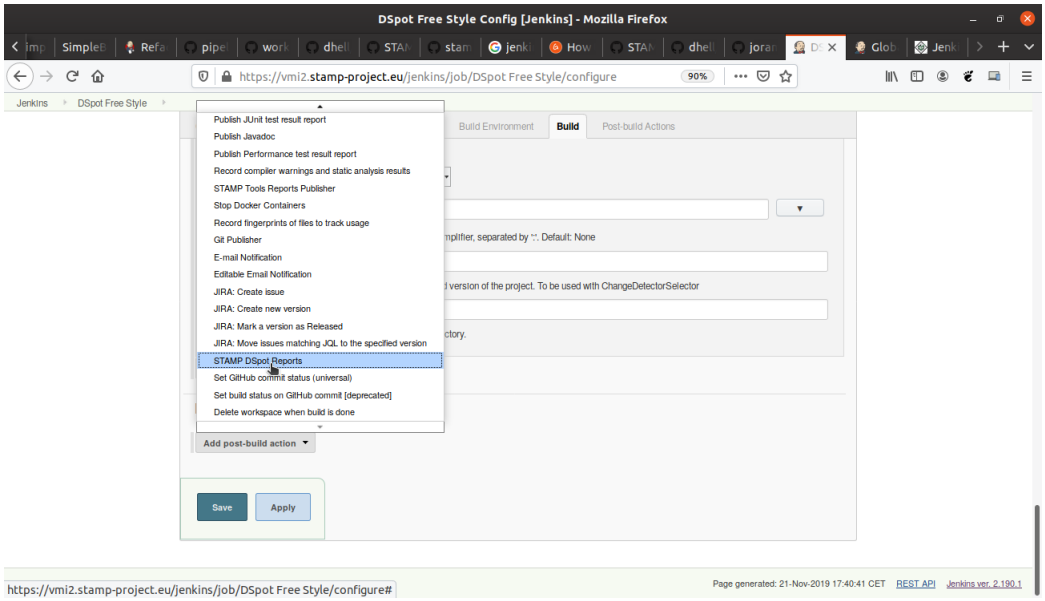


Figure 3.6: Configuring DSpot Jenkins plugin in a freestyle job: post-build action to show DSpot execution reports

This plugin can be used also with modern Jenkins pipelines, simply adding a `dspot` step in the build stage: `dspot variable1: value1, ..., variable1: valueN`. Reader can check table 3.1 for parameters available for configuration. All variables are optional and default to values in the table 3.1:

Table 3.1: DSpot Jenkins Plugin parameters

Option	**pipeline variable**	**Usage**	**Default**
Run on changes	onlyChanges	Runs only on new or changed tests since the last build	false
Show reports	showReports	shows the DSpot reports in a visual format in the Jenkins UI	false
Project Location	projectPath	path to the target project root from the folder where dspot is executed.	Defaults to Workspace
Source location	srcCode	path to the source code folder	src/main/java/
Tests location	testCode	path to the test source folder	src/test/java/
Source binary location	srcClasses	path to the compiled code folder. (.class files)	target/classes/
Tests binary location	testClasses	path to the compiled tests folder. (.class files)	target/test-classes/
Filter	testFilter	filter on the package name containing tests to be amplified	all tests
Output directory	outputDir	path to the output folder	dspot-out

For an exhaustive list of all available parameters, reader can refer to official guide available at https://github.com/STAMP-project/stamp-ci/tree/master/dspot-jenkins-plugin

For each build, a menu item is created to show the detailed Report of the DSpot run, as shown in figure 3.7.

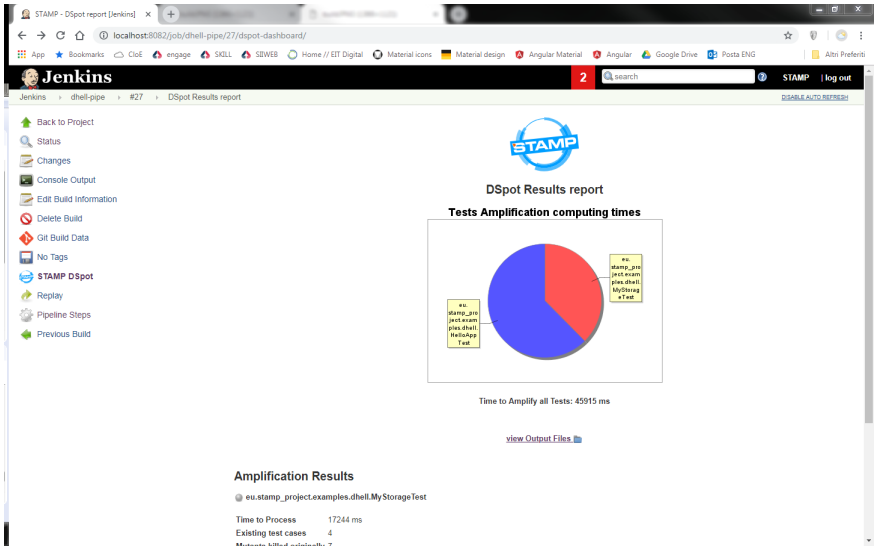


Figure 3.7: DSpot execution report

If test cases are successfully amplified, the developer can navigate the details of the test class, getting more information on the amplified test cases. Reports adapt based on the used Selector, as shown in figure 3.8.

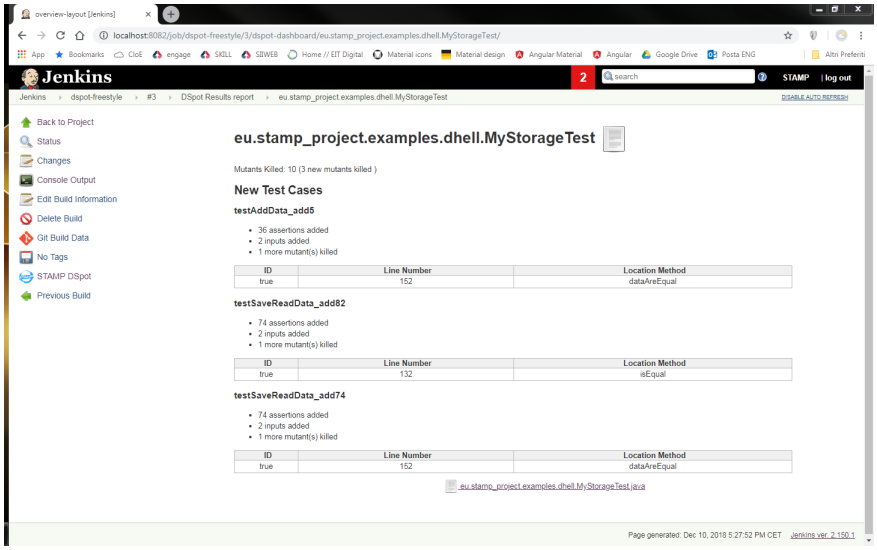


Figure 3.8: DSpot execution report: drill-down to amplified test cases

3.2 CI/CD pipeline assets

3.2.1 STAMP Pipeline library

A pipeline library of useful functions for STAMP CI/CD processes has been developed and made available in a dedicated repository (see https://github.com/STAMP-project/pipeline-library). This library, written in Groovy language, offers these functions:

- `cloneLastStableVersion(String folderName)`: this function let to have clone, within the current build, another repository version, whose build terminated with success. It is used for CI STAMP processes which make use of DSpot Diff selector (which needs in turn to have two different versions of code repository to select test cases affected by code change)
- `getLastStableCommitVersion(`: used to identify the commit identifier bound to last stable build. Needed by `cloneLastStableVersion(String folderName)`;
- `cloneCommitVersion(String commitVersion, String foldername)`: used to clone a specific version of code repository, identified by a commit. The repository is cloned in the specified folder, within the current build workspace;
- `pullRequest(String token, String repositoryName, String repositoryOwner, String pullRequestTitle, String pullRequestBody, String branchSource, String branchDestination, String proxyHost=null, Integer proxyPort=null)`: create a pull request in GitHub, using GitHub APIs. It is a better approach compared to usage of `hub` client (see https://github.com/github/hub), because, while `hub` client requires an additional installation in Jenkins (and in Jenkins slaves), using GitHub APIs doesn't require installation of any further component. This function is widely used in STAMP CI/CD processes to automatically open pull requests containing amplified test cases.

3.2.2 DSpot execution optimization in CI

One of the goals about executing DSpot within CI/CD processes was on optimizing execution time (with execution triggered on every commit on a branch), limiting execution of DSpot just on existing test cases related to code changes, and on new test cases, by the means of specific pipelines. As reported in Deliverable 1.4, section 5.3.4, two possible approaches have been evaluated, one based on DSpot Diff test selector and the other one based on Jenkins `changeset` functionality.

Below the description of the reference scenario:

1. From developer workstation:
   (a) Push an initial version (v1) into the repository
   (b) Push the new version (v2) into the repository
2. In CI/CD Server (on second push event):
   (a) checkout the new version (v2)
   (b) find and download the last stable version that passed the build in the past (v1)
   (c) Compare the two versions of the project (v2 and v1) to find tests to amplify with DSpot:
      i. the old tests in v1 that are impacted from the modifications present in production code in v2
      ii. new tests introduced in v2

As documented in the repository, the main goal of DSpot Diff tool is: "Diff-Test-Selection aims at selecting the subset of test classes and methods that execute the changed code between two versions of the same program". Dspot-diff-test-selector is intended to be used to detect regressions: given a new version of the repository, dspot-diff is able to detect which test cases are related to source code modification. This can be used as a system to evaluate quality of a pull-request before merging it into the master:

- Create a pull request (v2) on the repository
- checkout the code of the master branch (v1)
- checkout the code of the pull request branch (v2)
- from the master branch find only the old test (v1) that are impacted from the changes of v2 (and not the new tests)
- Amplify the old version (v1) to find whether the amplified tests, impacted from the changes of v2, have a regression (pass on v1 and fail in v2) or not

The following pipeline has been developed to setup the DSpot Diff-based process in Jenkins:

```
@Library('stamp') _

pipeline {
  agent any
  stages {
    stage('Compile') {
      steps {
        script {
          stamp.cloneLastStableVersion('oldversion')
        }
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml clean compile"
        }
      }
    }

   stage('Unit Test') {
      steps {
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml test"
        }
      }
    }

    stage('Amplify') {

        withMaven(maven: 'maven3', jdk: 'JDK8') {
          dir ("joram/joram/mom/core") {
          sh "mvn clean eu.stamp-project:dspot-diff-test-
              selection:list -Dpath-dir-second-version=${WORKSPACE
              }/oldVersion/joram/joram/mom/core"
          sh "mvn eu.stamp-project:dspot-maven:amplify-unit-tests
               -Dpath-to-test-list-csv=testsThatExecuteTheChange.
              csv -Dverbose -Dtest-criterion=
              ChangeDetectorSelector  -Dpath-to-properties=src/
              main/resources/tavern.properties -Damplifiers=
              NumberLiteralAmplifier -Diteration=2"
        }
      }
    }
```

```
  }

  }
   environment {
    GIT_URL = sh (script: 'git config remote.origin.url',
       returnStdout: true).trim().replaceAll('https://','')
  }
}
```

Listing 3.2: Jenkins pipeline to evaluate DSpot Diff usage to optimize DSpot execution in CI

In the **Compile** stage, the pipeline uses STAMP pipeline library (see https://github.com/STAMP-project/pipeline-library) to clone the last stable previous build, with the `stamp.cloneLastStableVersion('oldversion')` statement. In this way, in the current build workspace, we have two repository versions: the current one and the last stable one. In the **Amplify** stage two steps are performed:

1. detect test cases involved in the code change: `sh "mvn clean eu.stamp-project:dspot-diff-test-selection:list -Dpath-dir-second-version=${WORKSPACE}/oldVersion/joram /joram/mom/core"`
2. feed DSpot with these tests in order to execute test amplification only on this subset of tests: `sh "mvn eu.stamp-project:dspot-maven:amplify-unit-tests -Dpath-to-test-list-csv=testsThatExecuteTheChange.csv -Dverbose -Dtest-criterion=ChangeDetectorSelector -Dpath-to-properties=src/main/resources/tavern.properties -Damplifiers=NumberLiteralAmplifier -Diteration=2"`

The main issue with this approach is the need for cloning two repositories: the current one (bound to Jenkins build), performed by Jenkins to execute pipeline tasks (the pipeline is in the repository) and the previous one, as shown in the previous pipeline. This operation can be very time-consuming for large repositories.

The approach based on Jenkins `changeset` doesn't suffer of the double repository cloning issue, and can be used to select a subset of test to be amplified. The pipeline below is an example of this:

```
  agent any
  stages {
    stage('Compile') {
      steps {
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml clean compile"
        }
      }
    }

   stage('Unit Test') {
      steps {
        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/pom.xml test"
        }
      }
    }
```

```
    stage('Amplify') {
      when { changeset "joram/joram/mom/core/src/test/**" }
      steps {
      script {
          dspot_test_param = "";
          def changeLogSets = currentBuild.changeSets
          for (int i = 0; i < changeLogSets.size(); i++) {
            def entries = changeLogSets[i].items
            for (int j = 0; j < entries.length; j++) {
              def entry = entries[j]
              def files = new ArrayList(entry.affectedFiles)
              for (int k = 0; k < files.size(); k++) {
                def file = files[k]
                if (file.path.endsWith("Test.java") && file.path.
                   startsWith("joram/joram/mom/core/src/test/java
                   ")){
                  dspot_test_param += file.path.replace("joram/
                     joram/mom/core/src/test/java/","").replace("
                     /",".").replace(".java","") + ",";
                }
              }
            }
          }
          dspot_test_param = "-Dtest=" + dspot_test_param.
             substring(0, dspot_test_param.length() - 1)
        }

        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/joram/mom/core/pom.xml eu.stamp-
             project:dspot-maven:amplify-unit-tests -e ${
             dspot_test_param}"
        }
      }
    }

  }
   environment {
    GIT_URL = sh (script: 'git config remote.origin.url',
       returnStdout: true).trim().replaceAll('https://','')
  }
}
```

Listing 3.3: Jenkins pipeline to evaluate Jenkins `changeset` usage to optimize DSpot execution in CI

In this pipeline, `Changeset` feature is used to detect changed test cases; then this subset of test cases is arranged in the format required by DSpot (a list of fully-qualified Java class names), and is passed through the -e parameter: `-e $dspot_test_param`.

But this approach can't detect test cases related to code changes: if code changes, but its related test cases don't, `Changeset function` won't work. The problem can be solved adopting a naming convention for test cases: all test cases, stored in `src/test/java`, must have the same package of bound classes they actually unit test, and have the same name, with suffix `Test`:

it is a very common naming convention (for instance all IDEs provide developers with wizard to quickly generate unit tests with this convention). With this approach, the previous pipeline can be used with a small change:

```
pipeline {
  agent any
  ...

    stage('Amplify') {
      when { changeset "joram/joram/mom/core/src/\textbf{main}/**
         " }
      steps {
      script {
          dspot_test_param = "";
          def changeLogSets = currentBuild.changeSets
          for (int i = 0; i < changeLogSets.size(); i++) {
            def entries = changeLogSets[i].items
            for (int j = 0; j < entries.length; j++) {
              def entry = entries[j]
              def files = new ArrayList(entry.affectedFiles)
              for (int k = 0; k < files.size(); k++) {
                def file = files[k]
                if (file.path.endsWith("Test.java") && file.path.
                   startsWith("joram/joram/mom/core/src/test/java
                   ")){
                  dspot_test_param += file.path.replace("joram/
                     joram/mom/core/src/test/java/","").replace("
                     /",".").replace(".java","") + ",";
                }
              }
            }
          }
          dspot_test_param = "-Dtest=" + dspot_test_param.
             substring(0, dspot_test_param.length() - 1)
        }

        withMaven(maven: 'maven3', jdk: 'JDK8') {
          sh "mvn -f joram/joram/mom/core/pom.xml eu.stamp-
             project:dspot-maven:amplify-unit-tests -e ${
             dspot_test_param}"
        }
      }
    }

  }
   environment {
    GIT_URL = sh (script: 'git config remote.origin.url',
       returnStdout: true).trim().replaceAll('https://','')
  }
}
```

Listing 3.4: Jenkins pipeline to evaluate Jenkins `changeset` usage to optimize DSpot execution in CI

In this way Jenkins selects only changed classes: `when changeset "joram/joram/mom/-core/src/main/**"` . And with this code snippet:

```
if (file.path.endsWith("Test.java") && file.path.startsWith("
    joram/joram/mom/core/src/test/java")){
                    dspot_test_param += file.path.replace("joram/
                        joram/mom/core/src/test/java/","").replace("
                        /",".").replace(".java","") + ",";
                }
```

Listing 3.5: Jenkins pipeline to evaluate Jenkins `changeset` usage to optimize DSpot execution in CI

only test classes related to changed classes are selected for test amplification.

3.3 STAMP IDE

3.3.1 DSpot Eclipse plugin

The Stamp IDE plugin for DSpot is described in D4.3, 3.1.2 Developer productivity tools, Stamp IDE DSpot plugin. In this document we explain some changes and new features in the plugin.

1. Support for the latest DSpot version (DSpot is called using it's Maven plugin)
2. Support for DSpot options completed
3. The wizard structure has changed, now it is composed of four pages and two dialog forms, as shown in figure 3.9.
    - Page one: target project configuration. This page includes the fields to load and create configurations and the information about the project and Java version used.
    - Page two: Execution configuration. In this page you can select the tests to amplify and the amplifiers and criterion used.
    - Page three: Additional configuration. This page contains some extra configuration, for example maximum number of tests amplified or budgetizer to use. You can also select specific test cases to amplify (selected cases must be part of selected test classes).
    - Page four: Additional configuration. In this page you can select test classes, cases or even packages to exclude from amplification process. At bottom of the page you will find links for opening the dialog forms.
    - Advanced options dialog: using this dialog you can set some DSpot expert parameters, for example the random seed used for the amplification or configure your environment with options like setting the working directory or editing the system variables.
    - Optional properties dialog: this dialog includes some properties to write in the DSpot `.properties` file of your project, for example the Pit version or Maven pre goals (see DSpot documentation at https://github.com/STAMP-project/dspot#command-line-options).

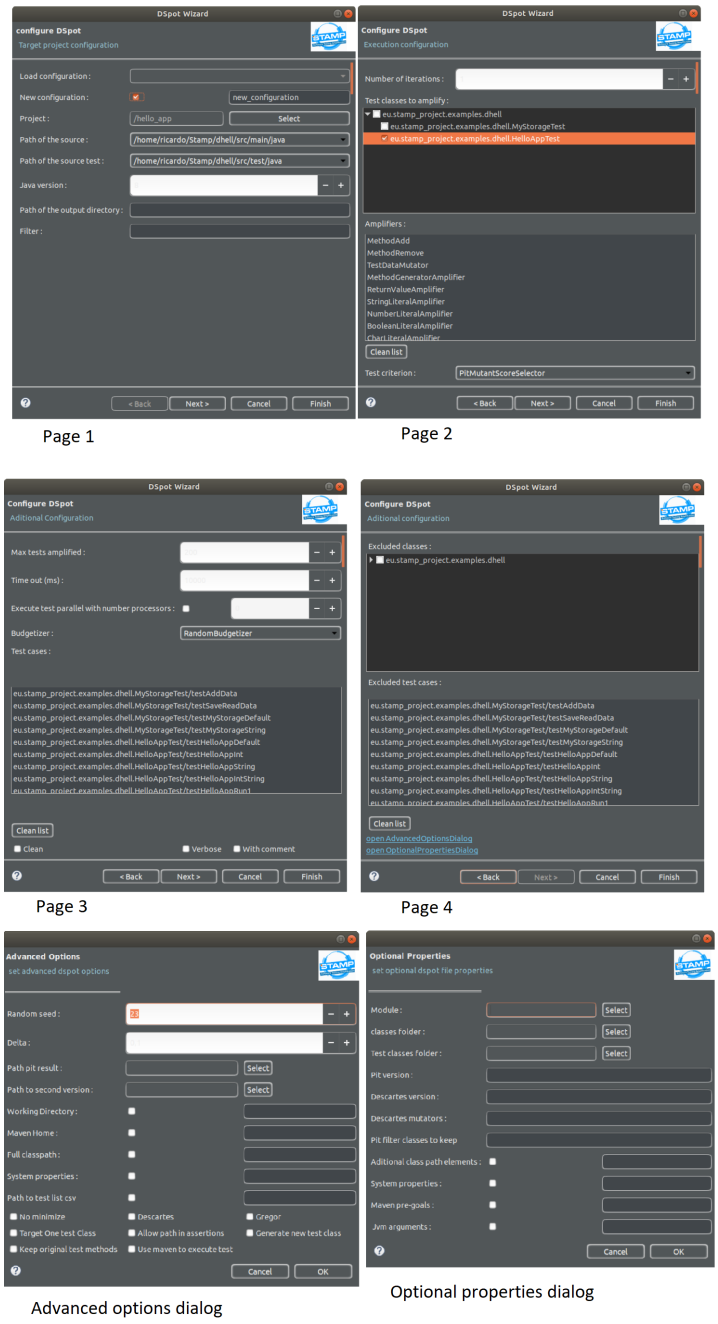


Figure 3.9: DSpot Wizard

3.3.2 Descartes Eclipse plugin

The Eclipse plugin for Descartes is described in D4.3, 3.2.2 Developer productivity tools, Stamp IDE: Descartes plugin. Here we explain some new features and changes.

Now, the Descartes Eclipse plugin allows the automatic creation of Jira tickets from issues summary. This support includes following features:

1. Definition of Jira accounts to be saved in the Eclipse secure store. Accounts can be set

in the Eclipse preferences facility within a page called "Descartes Jira preferences" (figure 3.10).

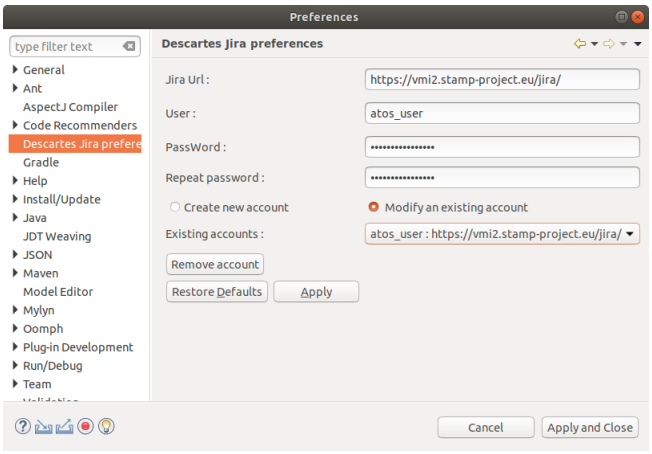


Figure 3.10: Jira preferences page implemented by Descartes plugin

2. The automatic Jira ticket creation for a issue is accessed from a link "Open Jira ticket" in the right top corner of the issues view.
3. A Jira ticket creation wizard allows to choose a Jira account between the stored accounts, select the project in Jira where the ticket will be created and the type of issue, furthermore the wizard allows to see and edit the generated title and summary, before creating the ticket (figure 3.11).

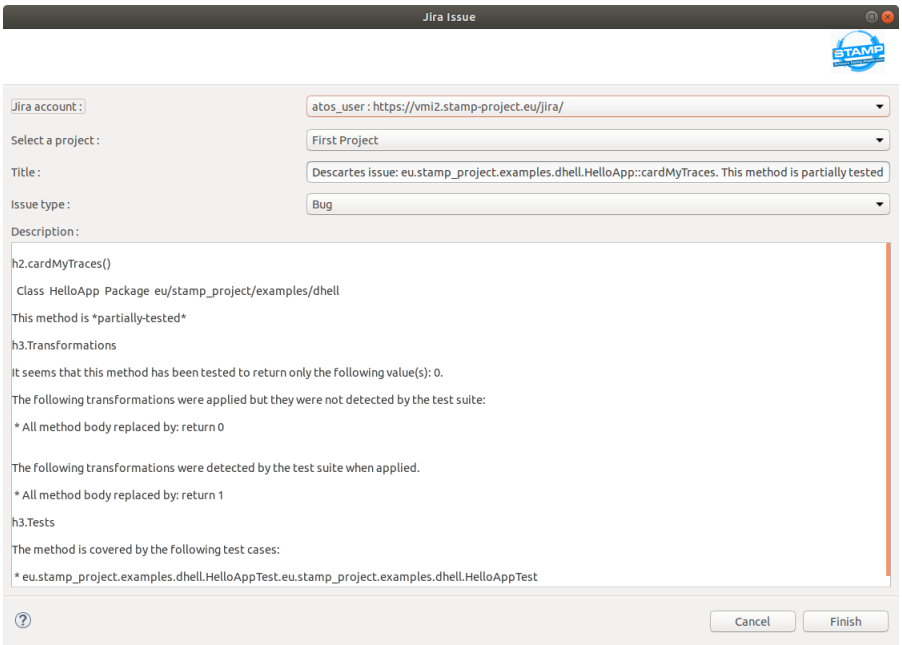


Figure 3.11: Jira Issue creation wizard

4. After ticket creation, a dialog will provide some information about the created ticket.

3.3.3 Botsing Eclipse plugin

The Stamp IDE plugin for Botsing is described in D4.3, 3.4 Botsing, 3.4.2 Developer Productivity Tools: STAMP IDE Botsing plugin. Below some new features are reported.

- Support for the 1.0.7 Botsing version
- Now developer can generate Botsing models (see Botsing model generation plugin in this section) access the model generation wizard clicking on the model generation link in Botsing wizard, (see Botsing model generation at the end of this section).
- Support for Botsing model and Object pool options, model option allows to use the Botsing models to improve the model generation process, this field points to a directory with the models, Object pool option is related with models, this to options can be enabled or disabled with a check button called "Use models".

3.3.4 RAMP Eclipse plugin

Stamp IDE includes a plugin for RAMP that offers wizard-based support.

The wizard is called from a menu entry in the Eclipse main bar (figure 3.12).

Before opening the wizard, the plugin will create a `classpath.txt` file for the selected project, the file will be placed at the project root folder (this feature works only on maven projects, for non-maven projects the classpath must be introduced manually).

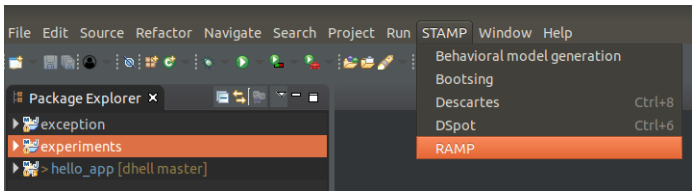


Figure 3.12: menu entry for opening the RAMP wizard

The configurable options in the wizard main page (figure 3.13) are from top to bottom:

1. Class, the full qualified name of the class for which tests will be generated.
2. Project classpath.
3. Search Budget, an integer number (see RAMP documentation at https://github.com/STAMP-project/evosuite-ramp).
4. Seed clone, a real number between 0 and 1.

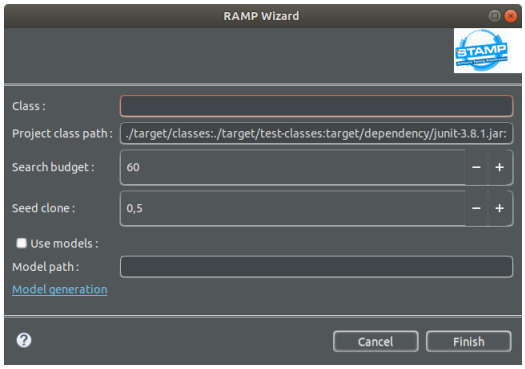


Figure 3.13: RAMP wizard main page

The RAMP plugin includes a configuration properties file with several default options, for example the output folder.

RAMP is executed as an Eclipse job, so the RAMP logs are shown in the Eclipse console.

Other important feature is the support for Botsing model generation (see Botsing model generation plugin at the end of this section), the Botsing model generation Wizard is opened by the link "Model Generation".

Furthermore, the wizard allows to use these models to improve the RAMP work, the field to set the model path can be activated with a check button as in figure 3.14.

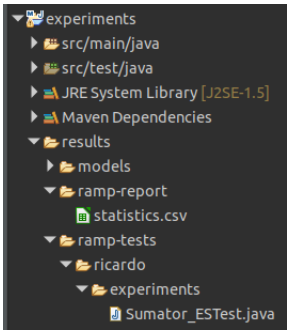


Figure 3.14: RAMP output

3.3.5 Botsing model generation Eclipse plugin

Stamp IDE includes a wizard support plugin for Botsing model generation. This plugin is used as a dependency of the Botsing and RAMP plugins. The wizard can be accessed from Botsing or RAMP wizards or a menu entry in the main bar at STAMP -> Botsing model generation: if the model generation is set in conjunction with the Botsing reproduction or RAMP jobs, the model generation job will be launched before the Botsing/RAMP jobs after pressing finish.

This wizard has only one page with three fields to define the three parameters for the models generation :

- Class path
- Project prefix
- Output directory

Chapter 4

STAMP collaborative platform

STAMP's collaborative platform has been designed and set up during the first months of the project. It consists of several integrated tools aimed at supporting the collaboration among partners. This platform addresses several needs:

- **documentation, knowledge base and information sharing**: document collaborative editing, versioning and management, calendar and scheduling, etc.
- **development**: SLM (Software Lifecycle Management), source code management, test environments provisioning, development task automation
- **communication**: keeping potential STAMP adopters informed with updates, news, use cases, mailing lists, interviews

Some tools activated at the beginning of the project, have been decommissioned because no longer needed: among them, the Slack channel (all partners preferred to keep all discussions around features or project activities within integrated issue trackers available both in Gitlab and GitHub - see as an example figure 4.1).

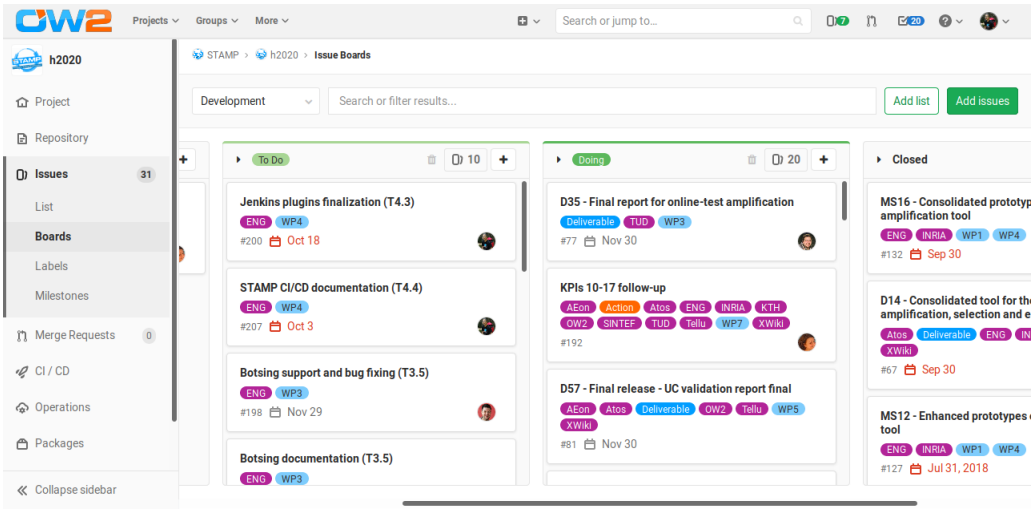


Figure 4.1: STAMP discussions through issues

Advantage of this approach is to keep track of general discussions by the issues and control

the development process by using GitHub pull requests and Gitlab merge requests as shown in figure 4.2.

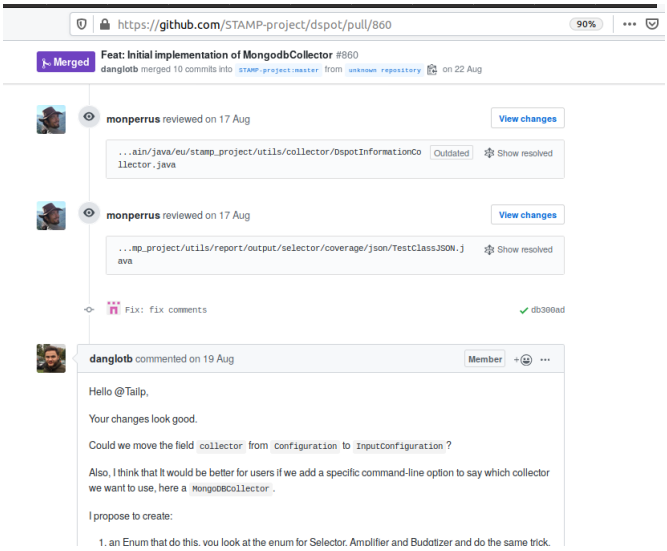


Figure 4.2: STAMP code reviews through PR/MR

SwaggerHub space has been decommissioned as well, since it has been no more needed to define the CI/CD integration architecture.

Figure 4.3: STAMP public home

The XWiki dedicated instance is structured to be the access portal to STAMP world both for STAMP adopters and STAMP consortium (see figure 4.3): adopters access all relevant material to know and understand better STAMP technology from the public home (https://www.stamp-project.eu), while STAMP consortium has its own dedicated internal space (https://www.stamp-project.eu/view/wiki/) to manage all project information and documents and to all tools available to support the development of STAMP project. Document management (review, versioning and publishing) is managed with a dedicated Gitlab space

https://gitlab.ow2.org/stamp, while source code management is delegated to a GitHub space (https://github.com/STAMP-project). Several servers have been made available by OW2 to STAMP partners as development, test and demo environments, supporting partners on all sysadmin tasks. In next sections reader will be provided with a short description about activities done on the platform during the past year.

4.1 Evolution and maintenance

After the first-half of the project, the platform has been fine-tuned by adding features requested by partners, or removing features no more needed.

Every documentation and information relevant for internal collaboration within the consortium has been published in the internal Wiki (i.e. the calendar with all monthly meeting, in-person meeting, partner to partner meetings, along with meetings minutes, Work Packages information - see figure 4.4).

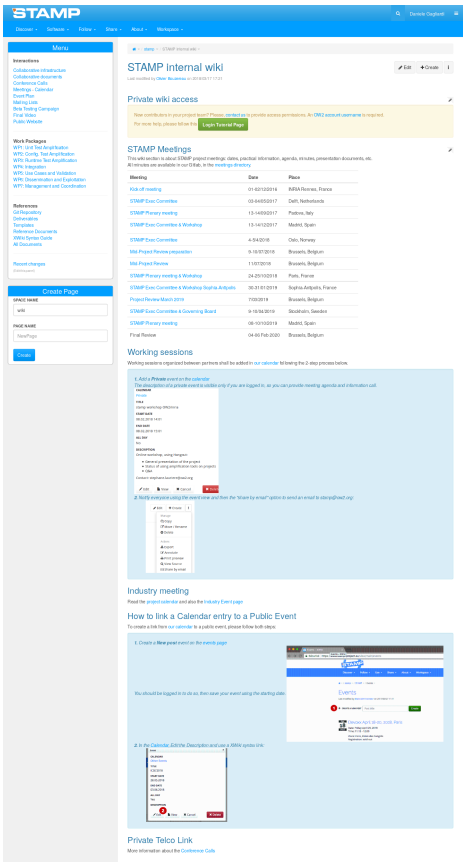


Figure 4.4: STAMP private portal

Access to tools and environment has been described in dedicated pages, along with addresses and credentials (figure 4.5).

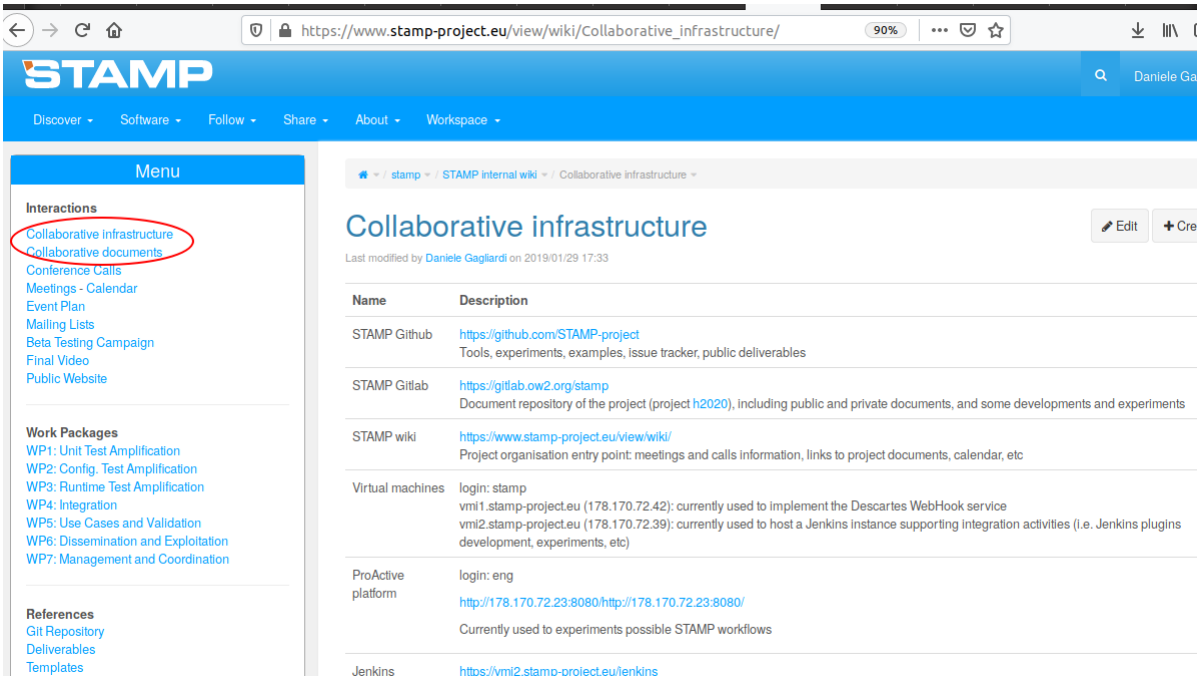


Figure 4.5: reference to STAMP dev,test,demo environments

The pictures above shows environments, set up in dedicated virtual machines and made available to STAMP partners for their experiments, development activities and demo, for example:

- ProActive installation to setup test amplification workflows based on STAMP tools;
- a server hosting Descartes GitHub App
- a server hosting the STAMP CI/CD reference architecture (described in more detail in next section)

So along with ordinary system maintenance activities (OS updates, tools updates, etc) and publishing activities (public content in STAMP official web site, private contents in STAMP private site), several environment provisioning activities has been performed by OW2 and ENG:

- OS updates on STAMP VM instances (vmi1.stamp-project.eu and vmi2.stamp-project.eu)
- re-installation of vmi1.stamp-project.eu OS to meet StackStorm (https://stackstorm.com/) requirements (replaced existing Debian 8 OS with Ubuntu 16.04). StackStorm has been used to implement Descartes as a service;
- hardware upgrades for vmi2.stamp-project.eu to support STAMP CI/CD scenario
- installation and configuration of tools needed to support STAMP CI/CD scenario: Jenkins CI and needed plugins (Blue Ocean, GitHub plugin, Pipeline plugin, HTML report plugin, Docker plugin), Jira Software (with related PostgreSQL rDBMS), Apache Maven, CAMP (with related Python 3 installation), Botsing Server, NGINX server, Let's Encrypt service integration (to provide STAMP Demo server with valid SSL certificates - needed to enhance STAMP server security and solve integration issues related to SSL handshake between client and server components)

- periodic clean-up activities to optimize disk space usage (system and application clean-up and rotation, clean up of older Jenkins build workspaces, obsolete OS update components)
- management of application and system accounts to grant access to project partners to STAMP collaborative platform and STAMP dedicated servers

4.2 STAMP Demo server

In order to validate the CI/CD reference architecture described in 2.3, a complete environment has been setup in vmi2.stamp-project.eu server with all needed components, in order to test and validate all STAMP plugins, all Jenkins pipelines and all the integration developed within the project. The OS is a Debian Jessie distribution (8.11), equipped with 8 GB of memory, and 4 vCPUs. This environment can be considered a STAMP CI/CD reference implementation (figure 4.6):

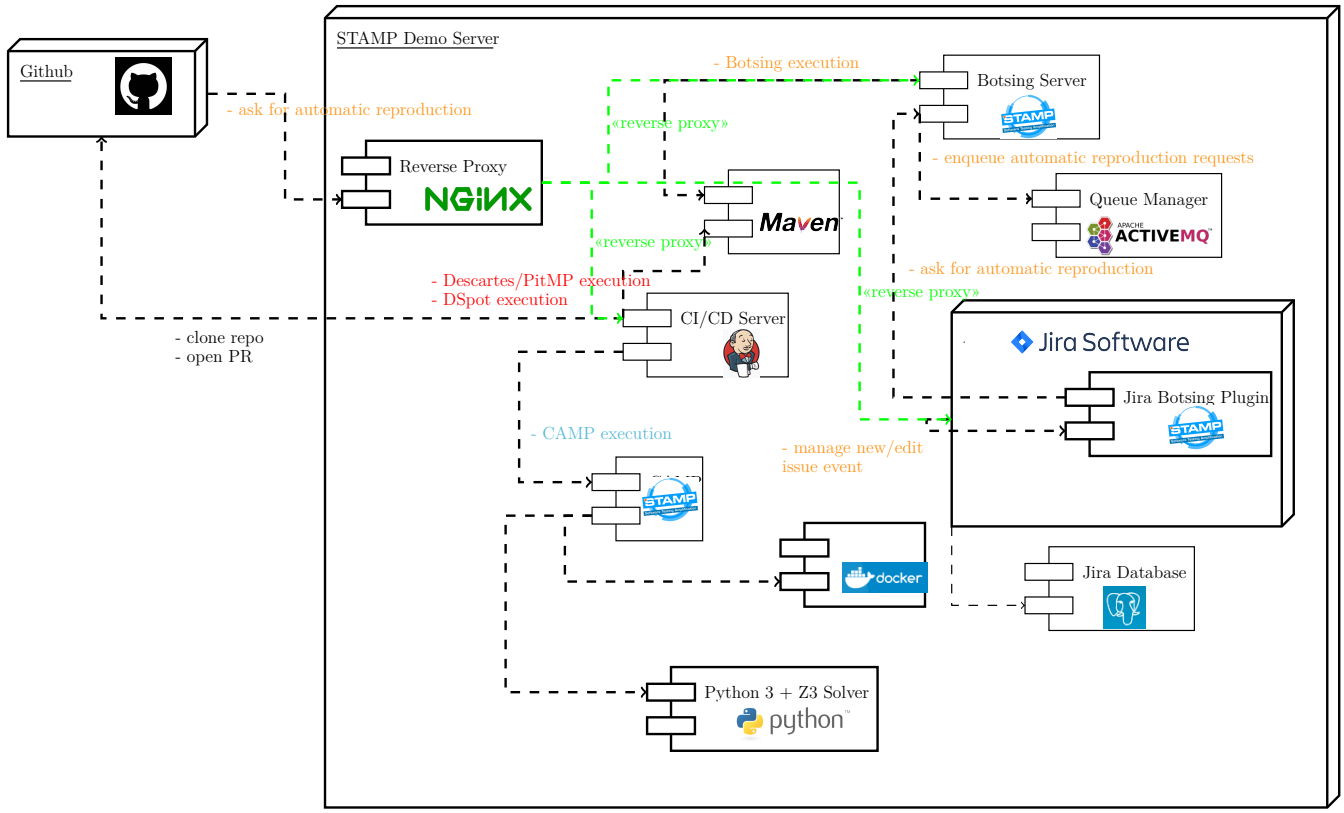


Figure 4.6: STAMP CI/CD architecture reference implementation

Below a short description of main components:

- Jenkins CI: an instance of Jenkins CI has been installed, along with several plugins such as:
  - Blue Ocean: advanced tool to edit, visualize, diagnose Jenkins pipelines. Used for all STAMP Ci/CD developed pipelines (see as an example 2.3 or 2.10)
  - HTML Published plugin, used for instance to publish mutation coverage reports and JMeter reports within Jenkins

  - GitHub plugin: used to integrated Jenkins with GitHub and its APIs;
  - Descartes Jenkins plugin: plugin developed during the project to show mutation coverage trends across each build;
  - DSpot Jenkins plugin: plugin developed during the project to configure Test Amplification with DSpot in freestyle jobs;

- Jira Software: an instance of Jira Software with a database based on PostgreSQL (the embedded database has been replaced in order to have a real world Jira installation, more reliable and stable). Then a perpetual license for ten users has been activated , in order to have a fully-functional environment, with no time constraints (due to temporary licenses). Botsing plugin has been installed within Jira itself;
- Docker and Docker container: latest available versions for Debian have been installed
- Python 3: default Python installation (Python 2.7) has been replaced by Python 3, needed by CAMP
- web frontend, based on NGINX: to expose Jira Software, Jenkins CI, Botsing Server on ordinary 443 SSL port. Below a snippet of NGINX configuration which highlights reverse proxy settings for Botsing Server, Jira Software and Jenkins:

```
upstream jenkins {
  keepalive 32; # keepalive connections
  server 127.0.0.1:8080; # jenkins ip and port
}

server {       # Listen on port 80 for IPv4 requests

  server_tokens off;
  server_name      vmi2.stamp-project.eu;

  #this is the jenkins web root directory (mentioned in the /
     etc/default/jenkins file)
  root             /var/cache/jenkins/war;

  access_log       /var/log/nginx/jenkins/access.log;
  error_log        /var/log/nginx/jenkins/error.log;

  ...

  }

  location @jenkins {
      sendfile off;
      proxy_pass          http://localhost:8080;
      proxy_redirect      http://localhost:8080  https://vmi2.
         stamp-project.eu;
      #proxy_redirect      default;
      proxy_http_version 1.1;

      proxy_set_header    Host                $host;
      proxy_set_header    X-Real-IP           $remote_addr;
```

```
        proxy_set_header   X-Forwarded-For
           $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
        proxy_max_temp_file_size 0;

    ...

    }

    location /jira {
         proxy_set_header X-Forwarded-Host $host;
         proxy_set_header X-Forwarded-Server $host;
         proxy_set_header X-Forwarded-For
            $proxy_add_x_forwarded_for;
         proxy_pass http://127.0.0.1:8180;
         client_max_body_size 10M;
    }

    location /botsing-server {
         proxy_pass http://localhost:5000;
         proxy_redirect http://localhost:5000 https://vmi2.
            stamp-project.eu/botsing-server;
         proxy_http_version 1.1;
         proxy_set_header X-Forwarded-For
            $proxy_add_x_forwarded_for;
         proxy_set_header X-Forwarded-Proto $scheme;
         proxy_set_header X-Forwarded-Port $server_port;
    }
```

Listing 4.1: STAMP Demo Server: reverse proxy configuration for Jenkins, Jira Software and Botsing Server

- Botsing Server and ActiveMQ, needed for automatic crash reproduction scenario

Other sysadmin activities performed are:

- ordinary system upgrades
- automating Let's Encrypt renewal SSL certificate
- ordinary and extraordinary system maintenance (removing old logs, freeing up disk space - sometimes Jenkins builds consume a lot of space, investigating about some brute force attack - STAMP demo server is exposed on the web and to its threats)

Chapter 5

Documentation and Courseware

One of the main objective of STAMP project is to allow the adopter to approach STAMP technology in a way as effective as possible. For this reason the documentation and training material is one of the most important outcomes of the project. During the project all the partners produced technical documentation, tutorials, examples, presentations: these material has been made consistent and published.

5.1 Documentation

The GitHub repository containing the source code of every tool, also includes a section on the documentation. The most relevant documentation assets provided by the consortium are the following:

- **Test assessment**: both Descartes and PitMP have several pages describing all the parameters needed to configure the execution of the tools. Starting from the main page of Descartes (https://github.com/STAMP-project/pitest-descartes), one can navigate through configuration parameters, tutorials ("Descartes for dummies") and report examples. The main page of PitMP (https://github.com/STAMP-project/pitmp-maven-plugin/) offers detailed information about how to use it with multi-module Maven projects;
- **Unit test amplification with DSpot**: DSpot project comprises several components, which extends DSpot features:
  - DSpot Maven plugin (to expose DSpot features as ordinary Maven goals)
  - DSpot Diff selector (to detect regressions between two different commits)
  - DSpot Web (a web application exposing DSpot features)
  - DSpot prettifier (to post-process DSpot results in order to make them more readable

  Each of these components has its own documentation, to let STAMP adopters install and use them within their tool-chains.
- Test configuration amplification: CAMP has its own website, built with GitHub pages (see https://pages.github.com/), available at https://stamp-project.github.io/camp
- Automatic crash reproduction has its own website, built with GitHub pages as well, available at https://stamp-project.github.io/botsing/

- Runtime amplification has several documentation pages:
  - RAMP model seeding tutorial, available at https://github.com/STAMP-project/evosuite-ramp-tutorial
  - Model seeding empirical evaluation, available at https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation
- STAMP in DevOps: for this topic, project partners developed several components and pieces of code, sharing them within STAMP GitHub repository, available at https://github.com/STAMP-project/stamp-ci. In this repository for instance STAMP users can find how to introduce DSpot Diff selector within their CI pipelines, following description available at https://github.com/STAMP-project/stamp-ci/tree/master/stamp-jenkins-cookbooks#amplify-on-code-changes. Moreover several components and libraries are available at https://github.com/STAMP-project/stamp-ci/tree/master/stamp-cicd-utils

5.2 Courseware

In order to enable STAMP adopters to follow a logical path within STAMP documentation, a dedicated project, made of several pages in markup language, has been created. This logical path is structured (see https://github.com/STAMP-project/stamp-courseware) in three sections dedicated each to an amplification service, with an additional section dedicated to using STAMP in DevOps (figure 5.1).

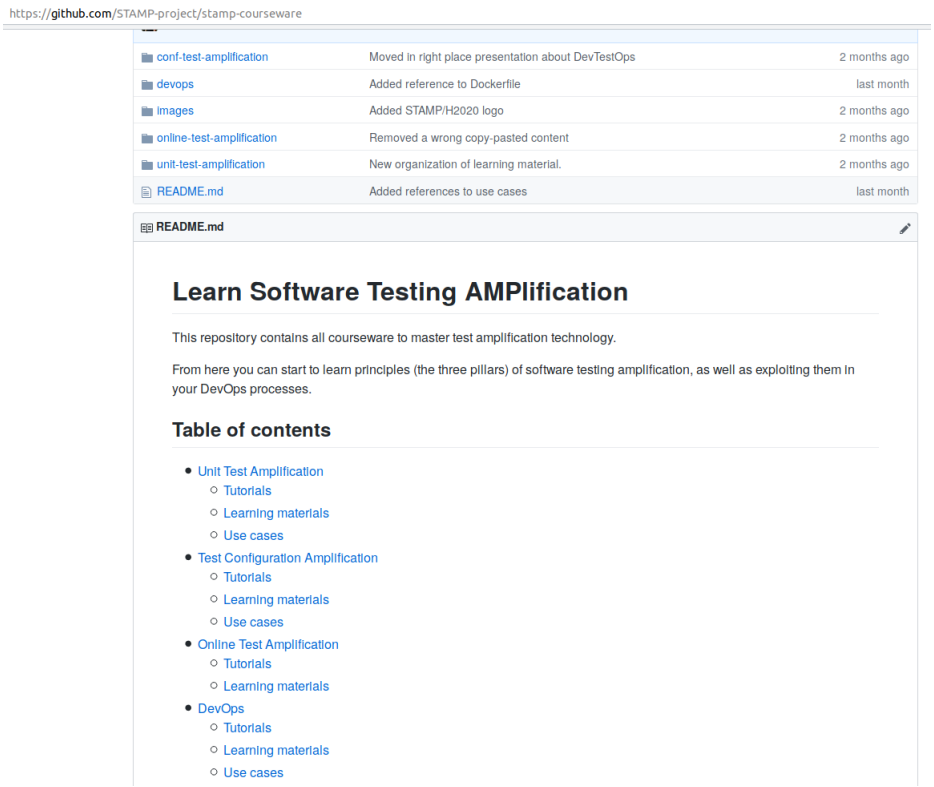


Figure 5.1: STAMP courseware path

From this repository STAMP adopters can navigate through all public STAMP documentation developed during the project by all partners and find all relevant information to understand how to introduce STAMP technology within their tool-chains. Moreover, a Docker image, containing all needed components to setup the CI/CD scenario, has been developed and published in DockerHub (see https://hub.docker.com/repository/docker/danzone/stamp-devops). In GitHub STAMP courseware repository there is also a section containing the Dockerfile (https://github.com/STAMP-project/stamp-courseware/blob/master/devops/assets/docker/Dockerfile), which will be briefly commented below:

```
FROM jenkins/jenkins:lts

MAINTAINER Daniele Gagliardi <daniele.gagliardi@eng.it>

# Update Python to Python 3
USER root
RUN apt-get update && apt-get install -y python3 python3-pip
   libgomp1 maven && apt-get clean && \
  rm -f /usr/bin/python && ln -s /usr/bin/python3 /usr/bin/python
      && \
  ln -s /usr/bin/pip3 /usr/bin/pip

# Update pip and install dependencies needed to install CAMP
RUN pip install --upgrade pip
RUN pip install setuptools

# Install CAMP from setup script. Docker, Docker Compose and last
    version of Z3 Solver are installed as well
RUN curl -L https://github.com/STAMP-project/camp/raw/master/
   install.sh | bash -s -- --install-z3 --z3-version '4.7.1' --z3
   -python-bindings /usr/lib/python3.5 --install-docker && pip
   install setuptools

# drop back to the regular jenkins user
USER jenkins
```

Listing 5.1: STAMP for DevOps Dockerfile

The parent image is based on the official Jenkins LTS (Long Term Stable) release (see https://hub.docker.com/r/jenkins/jenkins): `FROM jenkins/jenkins:lts`.

The first step is to update Python version, from 2.7 (default in `jenkins/jenkins:lts` to 3 (as required by CAMP):

```
# Update Python to Python 3
USER root
RUN apt-get update && apt-get install -y python3 python3-pip
   libgomp1 maven && apt-get clean && \
  rm -f /usr/bin/python && ln -s /usr/bin/python3 /usr/bin/python
      && \
  ln -s /usr/bin/pip3 /usr/bin/pip
```

Listing 5.2: STAMP for DevOps Dockerfile: update to Python 3

It switches to `root` user, perform OS updates and then install Python (and Pip, the Python package manager) version 3, replacing symbolic links to python.

After the installation, an upgrade of pip is performed , and then `setuptools` are installed: this is required to complete CAMP installation:

```
# Update pip and install dependencies needed to install CAMP
RUN pip install --upgrade pip
RUN pip install setuptools
```

Listing 5.3: STAMP for DevOps Dockerfile: update Pip and install `setuptools`

Then CAMP is installed with CAMP official installation script:

```
# Install CAMP from setup script. Docker, Docker Compose and last
    version of Z3 Solver are installed as well
RUN curl -L https://github.com/STAMP-project/camp/raw/master/
   install.sh | bash -s -- --install-z3 --z3-version '4.7.1' --z3
   -python-bindings /usr/lib/python3.5 --install-docker && pip
   install setuptools
```

Listing 5.4: STAMP for DevOps Dockerfile: CAMP installation

The script is used to install CAMP and all needed dependencies: Z3 solver, Docker and Docker compose.

In the last step, the Docker file switches to `jenkins` user (which is a good practice).

The user can execute this image, simply launching the command:

`docker run -p 8080:8080 -p 50000:50000 -v /var/run/-docker.sock:/var/run/docker.sock danzone/stamp-devops:0.0.1`

This command runs a new container based on STAMP DevOps image: it maps 8080 Jenkins container port to host 8080 port, and maps the the port 50000 used to connect slave agents to Jenkins. The option `-v /var/run/docker.sock:/var/run/docker.sock` is needed by CAMP to invoke Docker and create "siblings" containers (for details see https://stamp-project.github.io/camp/pages/setup.html#using-docker).

The output will be something similar to:

```
$ docker run -p 8080:8080 -p 50000:50000 -v /var/run/docker.sock
   :/var/run/docker.sock  danzone/stamp-devops:0.0.1
Running from: /usr/share/jenkins/jenkins.war
webroot: EnvVars.masterEnvVars.get("JENKINS_HOME")
2019-11-12 09:52:04.707+0000 [id=1] INFO  org.eclipse.jetty.util.
   log.Log#initialized: Logging initialized @477ms to org.eclipse
   .jetty.util.log.JavaUtilLog
...

2019-11-12 09:52:11.123+0000 [id=30]  INFO  jenkins.install.
   SetupWizard#init:

*************************************************************
*************************************************************
*************************************************************

Jenkins initial setup is required. An admin user has been created
    and a password generated.
```

```
Please use the following password to proceed to installation:

4fc9903e6de04c93859df7e16b6d0c9f

This may also be found at: /var/jenkins_home/secrets/
    initialAdminPassword

*************************************************************
*************************************************************
*************************************************************

...

2019-11-12 09:52:16.950+0000 [id=30]  INFO  jenkins.
    InitReactorRunner$1#onAttained: Completed initialization
2019-11-12 09:52:16.960+0000 [id=21]  INFO  hudson.WebAppMain$3#
    run: Jenkins is fully up and running

\label{lst:stamp-devops-image-startup-log}
```

Listing 5.5: STAMP for DevOps Dockerfile: running the container

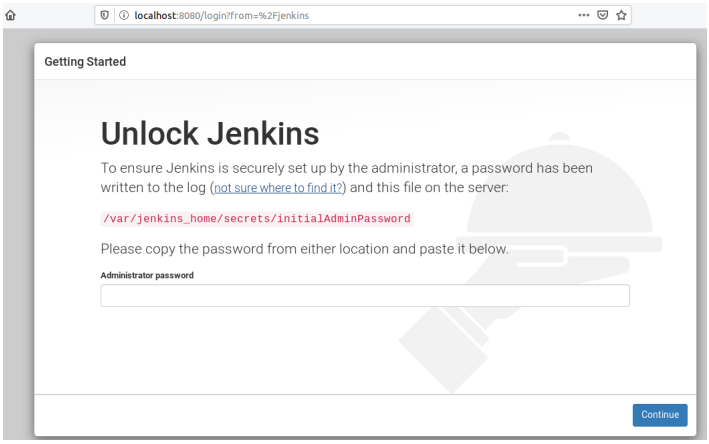


Figure 5.2: STAMP CI/CD Docker image: start-up configuration

When user starts STAMP DevOps container, first time Jenkins generates an admin password to access Jenkins administrative interface, showing it in the log as shown in figure 5.2.

The user can open a browser at `http://localhost:8080/`, follow the ordinary Jenkins start-up configuration and follow the official Getting-started guides as https://jenkins.io/doc/pipeline/tour/getting-started/:

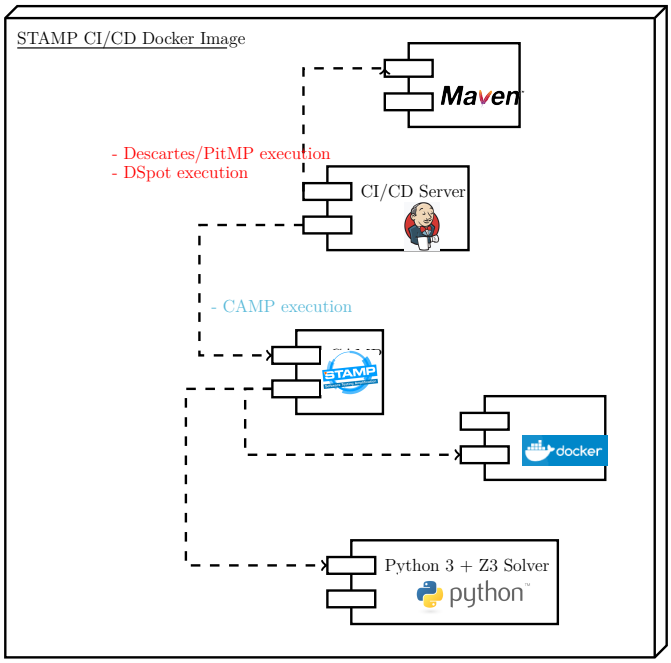


Figure 5.3: STAMP CI Docker image internals

In figure 5.3 are shown main components, which implement core features of STAMP CI/CD reference architecture, described in section 2.2:

- The CI/CD Server
- Apache Maven, needed for
  - ordinary Maven lifecycle steps (build, unit and integration tests, etc)
  - test amplification stages:
    - test assessment with Descartes/PitMP
    - unit test amplification with DSpot
    - common behavior test amplification by the means of Botsing and RAMP (see https://github.com/STAMP-project/evosuite-ramp)
- CAMP and its dependencies:
  - Docker and Docker compose, needed by CAMP `execute` steps
  - Python 3
  - Z3 solver

Chapter 6

Conclusion

During the last year, the work in WP4 spanned tasks 4.2 to 4.4, and required several maintenance activities (both ordinary and extraordinary) on the collaborative platform as per task 4.1. The activities are described in chapter 4.

The development of a complete CI/CD scenario has been the main activity in T4.2, to define a complete DevOps architecture with test amplification concepts seamlessly integrated in it, as described in chapter 2.

Within T4.3 all the needed components have been developed to make "test amplification in DevOps" happen, in order to integrate the CI server, issue trackers and to develop pipelines implementing DevOps processes with embedded test amplification stages, as described in chapter 2. Moreover existing STAMP components have been enhanced to enrich developers toolboxes with new STAMP features: for instance a new version of STAMP IDE supporting all STAMP tools, new Maven plugins offering STAMP features as ordinary Maven goals, etc. For a detailed description, please refer to chapter 3

Within T4.4 all the documentation produced by all STAMP partners has been collected and organized it in rational structure easy to navigate. Moreover some additional documentation has beed developed to provide STAMP adopters with hints to solve typical issues in a real world environment (i.e. executing STAMP behind a corporate proxy). Virtual machines developed as STAMP course-ware during the first half of the project (containing all STAMP tools ready-to-use) have been replaced by a Docker image containing all needed tools to quickly activate a CI/CD scenario STAMP-enhanced within a typical corporate DevOps infrastructure. A Docker image is smaller than an ordinary virtual machine, and requires less hardware resources. Moreover, since Docker is also supported by recent Windows platforms, this solution can be considered as a real cross-platform STAMP CI/CD solution. This image has been made available in official DockerHub repository, in order to ease a wide adoption of STAMP itself in DevOps. For a detailed description of documentation and courseware, please refer to chapter 5.