STAMP

Deliverable D4.4

Final public version of API and implementation of services and course-ware

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 course-ware
Dissemination Level	:	Public
Version	:	0.9
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)	:	Daniele Gagliardi, ENG Ciro Formisano, ENG Luca Andreatta, ENG Nicola Bertazzo, ENG Jesús Gorroñogoitia Cruz, ATOS Ricardo Tejada, ATOS Mael Audren De Kerdrel, AEON Caroline Landry, INRIA
Reviewer(s)	:	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 Industrialization section of D1.4 offers on overview of it, this deliverable focus on showing the internals that make STAMP works seamlessly within existing DevOps processes and developer toolboxes. The first part of the report describes the CI/CD reference scenario, with a complete 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 second part of this report 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 developed component has its own documentation to understand its usage and apply it within software life-cycle processes. Last two parts of the report present activities performed on collaborative platform, and documentation and course-ware available in the STAMP official repository, in terms of structure and content.

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	Adding PitMP section	Caroline Landry, INRIA
0.5	completed CI/CD scenario. Added contribution by ATOS (STAMP IDE) Added contribution by ActiveEon (Botsing Gradle plugin)	Daniele Gagliardi, ENG Ricardo Tejada, ATOS Mael Audren De Kerdrel, ActiveEon
0.6	completed architecture section. Added first draft for docs and courseware	Daniele Gagliardi, ENG
0.9	first draft complete.	Daniele Gagliardi, ENG Ciro Formisano, ENG

Contents

Acronyms

CD	Continuous Delivery
CI	Continuous Integration
EC	European Commission

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The main focus of integration activities during the past year was about implementing a CI/CD process, enhanced with STAMP novel features.

A reference architecture has been finalized thanks to a strict collaboration among all the partners: ENG presented several versions of this architecture, by the means of demos held during various in-person and online meetings, collecting requirements and feedback by other partners who provided real use cases possible scenarios and validated the architecture against them.

The result of this collaborative process is a complete reference architecture based on tools widely used among developers and DevOps communities: Jenkins CI, Jira Software, GitHub, etc. Each of these tools has its own proper STAMP integration component which enable test amplification features in it.

In the meanwhile also other tools not directly involved in the CI/CD scenario has been enhanced and consolidated, in order to provide developers with a toolbox containing everything to introduce test amplification aspects easily in their development process. This is really important to increase the adoption of STAMP: while DevOps is currently in a hype, it would be a mistake to exclude STAMP novel concepts from other development contexts, more traditional and more related to incremental/iterative processes. Test amplification can greatly increase the quality of software not just in DevOps contexts but also in more traditional contexts, quite common especially in large software companies.

This report presents the final release of STAMP components developed to integrate STAMP amplification services within 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 reader will be presented with a description of Continuous Integration Scenario, in which several components, orchestrated by a CI/CD server, cooperate to build software with a better quality. The description of the "Ops" phase will demonstrate how easy is for developers to use automatic reproduction features in the bug analysis process for all bugs risen in production environments: it is simply a matter of asking end users to attach production logs containing the error stack-trace to a bug opened in well known issue trackers, such as Jira Software and GitHub Issues.

As already mentioned in D4.3, this reference architecture can be expanded and varied, thanks to the fact that STAMP ecosystem is made of several services and plugins that facilitate the integration in different development contexts.

The CI/CD architecture is presented in chapter 2.

In the second part of this deliverable we provide a description of the STAMP ecosystem, made of several plugins usable within common developers toolboxes: IDE, build tools plugins,

etc. These tools are presented in chapter 3.

After the STAMP ecosystem section, another section describes the activities performed in STAMP collaborative platform during the last year (see chapter 4).

In the last part of this report we'll describe which kind of documentation can be found in STAMP official repository, how it is organized in order to find all needed information by STAMP adopters. Moreover we describe a Docker image that let STAMP adopters to experiment STAMP in a typical CI/CD environment, based on a Jenkins CI server. This last part is presented in chapter 5.

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 4 tasks of WP4.

Table 1.1: WP4 tasks reported in this deliverable

Task 4.1	This task, related to setting up and maintaining the STAMP project collaborative platform, has been extended at M36 with the last amendment. In this deliverable we present all activities performed on the platform in order to support the design and development activities of the project
Task 4.2	We have finalized the CI/CD integration architecture to support test amplification practices in a typical DevOps tool-chain
Task 4.3	We built new plugins and maintained the existing ones (keeping them aligned with new versions of STAMP tools) to extend STAMP integration in developers toolboxes
Task 4.4	We extended the documentation needed to use STAMP tools and integration, in form of wiki pages on the official STAMP GitHub repository, and we developed a Docker image, along with usage documentation, to smooth learning curve for STAMP adopters

1.2 Participants contribution to WP4

All the work done within the WP4 is the result of the collaboration of several STAMP project partners: it is a natural consequence of the fact that WP4 scope is about integrating STAMP tools. In Table 1.2 we summarize how we collaborated to achieve results presented in the current report.

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 the reader with a detailed description of the CI/CD STAMP-enhanced scenario, implemented and made available within the STAMP Demo server, a component of STAMP collaborative platform in which all STAMP partners can experiment and test STAMP tools within 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

Actually DSpot, Descartes/PitMP and CAMP intended usage in the CI/CD scenario is within unit, integration and system levels, while Botsing intended usage is within bugs life-cycle, as a tool to speed-up the investigation phase.

2.2 The reference Scenario

In section 2.3 of D4.3 we presented a general description of the CI/CD scenario, named as "CI/CD landscape". This schema has been further developed during past year, leading to the reference scenario described in following picture:

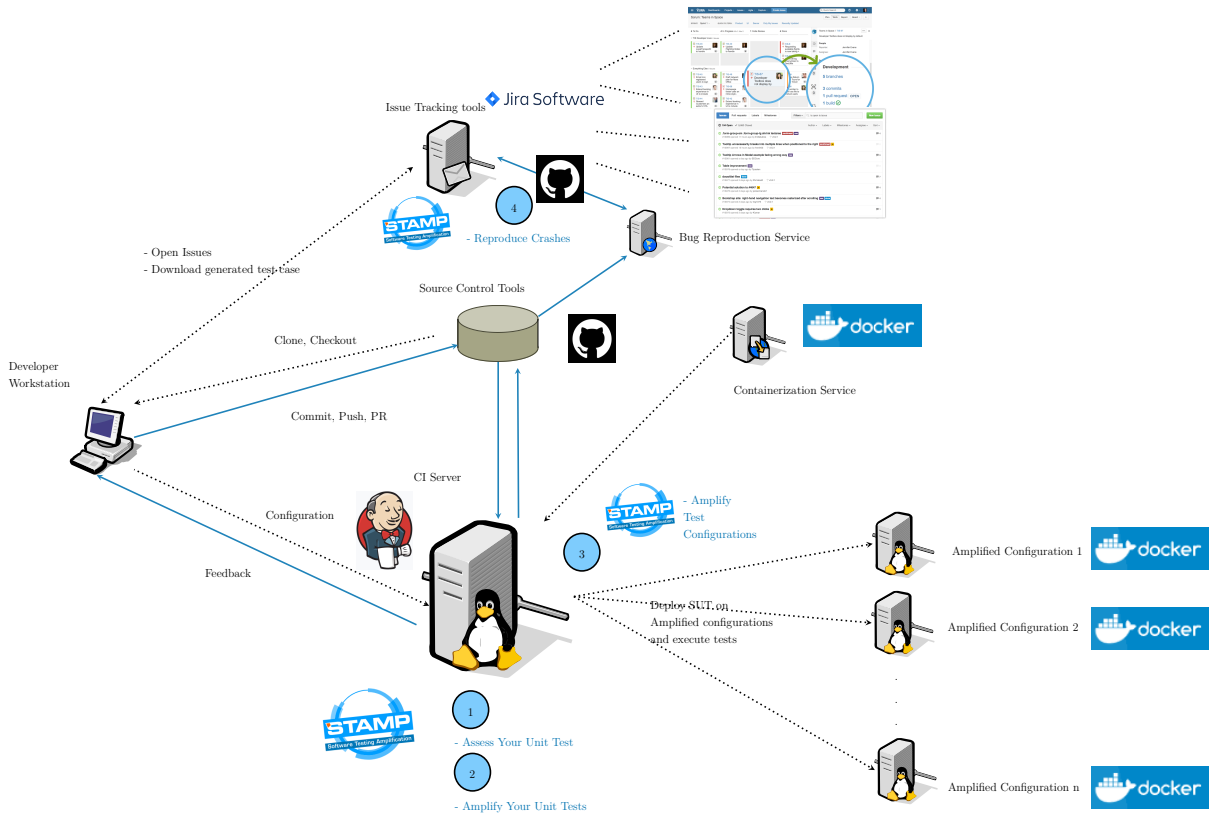


Figure 2.1: STAMP CI/CD reference scenario

Jenkins CI is the **Build/CI Server**, providing and supervising the integrated automation of all the tasks that contribute to build, verify and validate the software:

1. Developers provide Jenkins CI with specific **pipelines** that implement their automated processes. The Build/CI Server is configured to execute several jobs which run periodically or are 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

   This jobs are enriched with test amplification capabilities;
2. Developers receive **feedback** from it, in form of reports and dashboards
3. Developers also configure and integrate container-based tools to provide target systems where the software will run (**test configuration provisioning**). STAMP CI/CD scenario leverages explicitly Docker technology;
4. Developers will manage target systems with **configuration management tools**, using "infrastructure as code" practices. Actually they configure Docker files and Docker compose files to define target systems to execute integration and system level tests (usually functional and performance tests);

5. Developers will have **their own productivity tools** to write source code and tests. In this case we provided developers with STAMP IDE, a set of Eclipse plugins which enable developer workstations with STAMP capabilities;

6. Developers will store source code, test cases and Jenkins pipelines in **source code repositories** which will keep track of any change. In the reference scenario we selected GitHub as a git repository hosting service;

7. **Issue trackers** are used by developers to track all tasks and user stories that model 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. We think that these two tools cover the majority of issue tracking needs in the open source development world.

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 2.17 shows the detailed unit test amplification reference scenario:

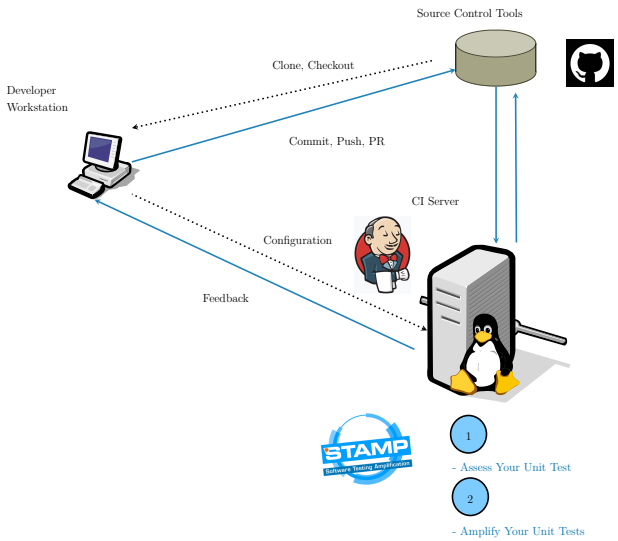


Figure 2.2: STAMP CI Unit Test Amplification reference scenario

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 pulled 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

The following pipeline (available in STAMP GitHub repository, see https://github.com/STAMP-project/joram/blob/master/Jenkinsfile) perform these tasks:

```
@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

Using the BlueOcean interface (see https://jenkins.io/projects/blueocean/), the whole execution of pipeline stages is represented as described in figure 5.2.

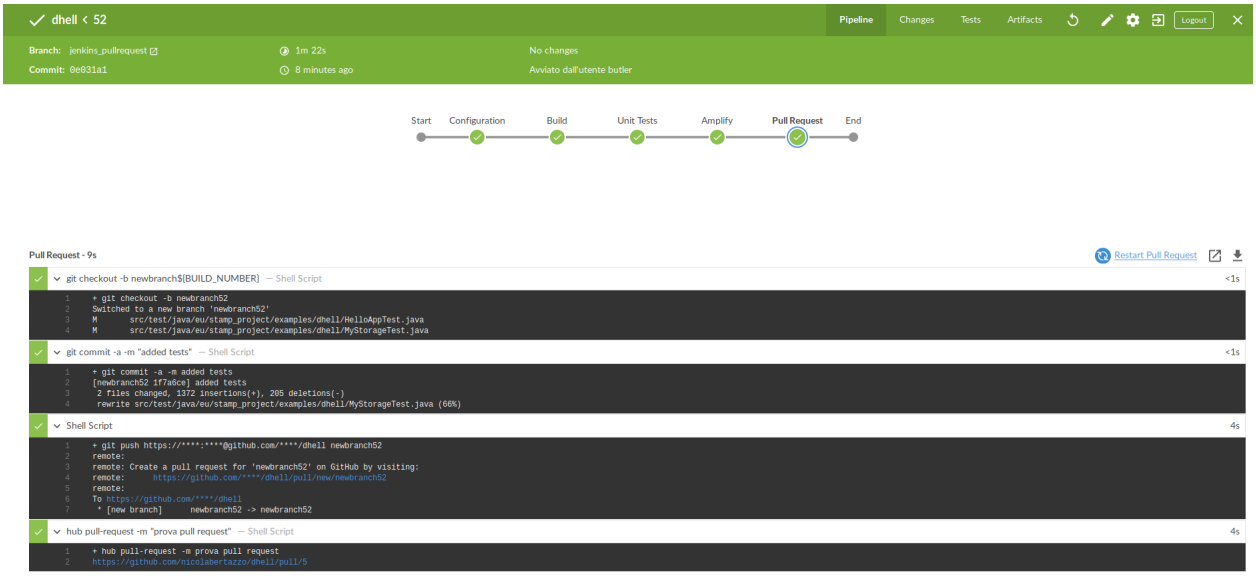


Figure 2.3: Unit test amplification pipeline execution, shown by Blue Ocean interface

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'`). Thanks to PitMP, this task is executed with this statement:

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

Maven goal `eu.stamp-project:pitmp-maven-plugin:descartes` trigger the execution of PitMP, leveraging the extreme mutation testing capabilities of Descartes, applied to the current project that happens to be 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 leverages this report format in order to have it available in 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

Thanks to this step, mutation coverage reports are available in Jenkins (figure 2.4)

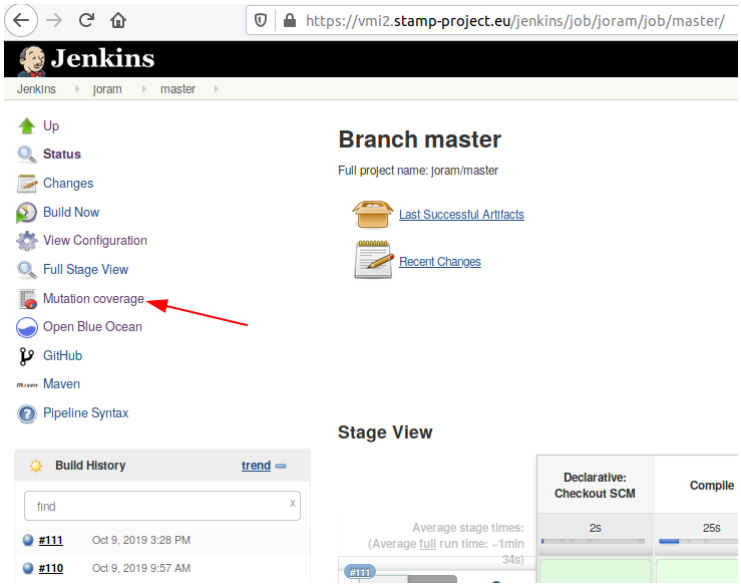


Figure 2.4: Menu item in Jenkins dashboard to access mutation coverage report

Accessing to the report, developer can inspect mutation coverage at package level (2.5) and then drilling down to the code to understand which are needed improvements in order to make his test cases stronger (figure 2.6)

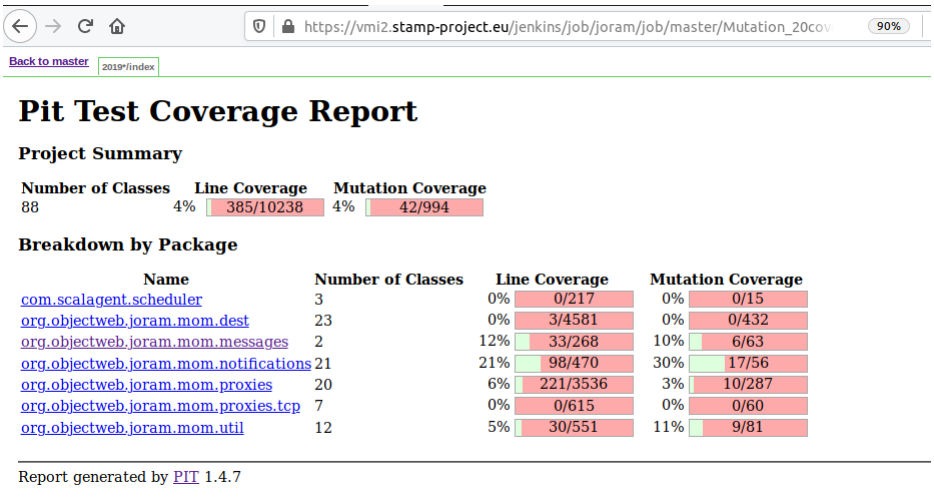


Figure 2.5: Mutation coverage report at package level.

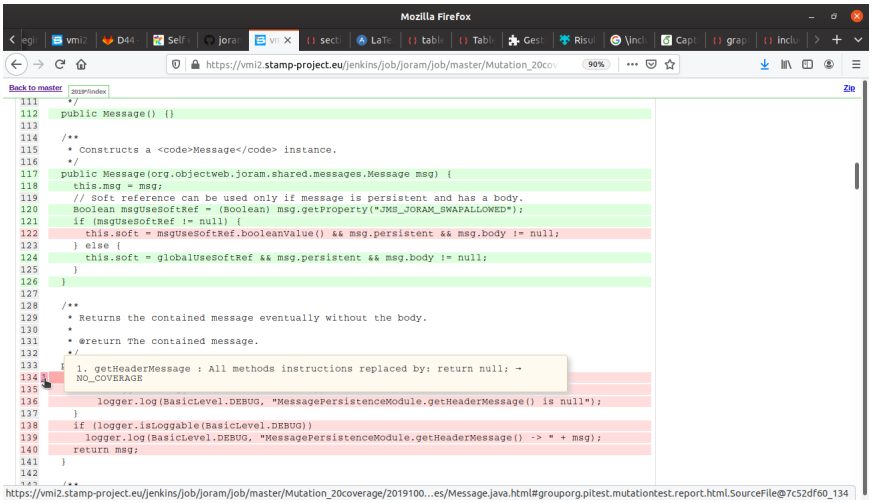


Figure 2.6: Inspecting test cases strength issues thanks to drill-down feature of mutation coverage report.

2.2.1.2 Unit test amplification - Test amplification with DSpot

The next step is the optional task 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 trigger the execution of the pipeline against this branch. Useless test amplification (and subsequent new pull request) are prevented, thanks to this condition.

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.

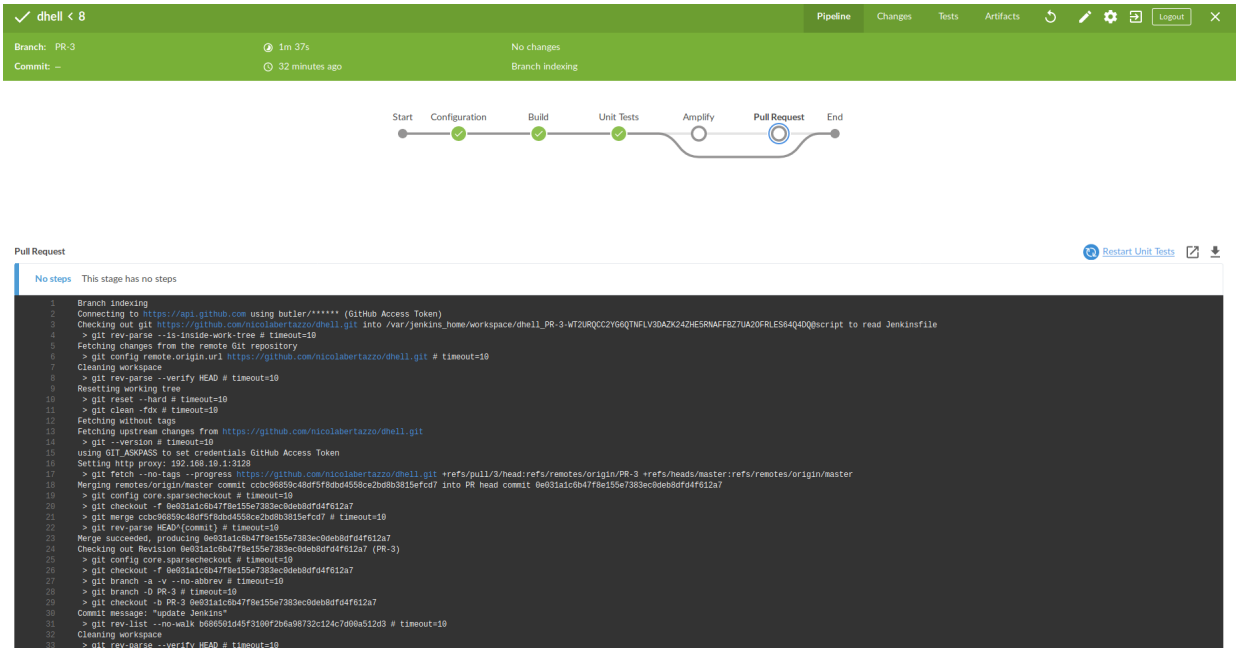


Figure 2.7: Pipeline execution skipping unit test amplification, shown by Blue Ocean interface

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

Last step makes available new amplified test cases in the repository, in a dedicated branch. Thanks to a pull request automatically opened by Jenkins, developers can inspect 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/targe` ensures that the whole step would be executed only if DSpot 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` actually copies amplified test cases in the directory containing all existing test cases.

The following three steps

```
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

commit amplified test cases, create a new branch with name `amplifybranch-`*`current branch-current build number`* (in this way the new branch has a self-explanatory name which let developer who will manage the pull request to quickly figure out in which branch and in which Jenkins build test amplification happened), and push amplified test cases in this new branch.

The last step actually open 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 functions available in STAMP pipeline library (https://github.com/STAMP-project/pipeline-library), which makes simpler complex tasks like opening pull requests using GitHub APIs, iterating over Jenkins builds to find the first successful build, cloning specific branches or commits, etc.

Usage of this library is declared at the very beginning of the pipeline, with the statement

```
@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 2.17 shows the detailed test configuration amplification reference scenario.

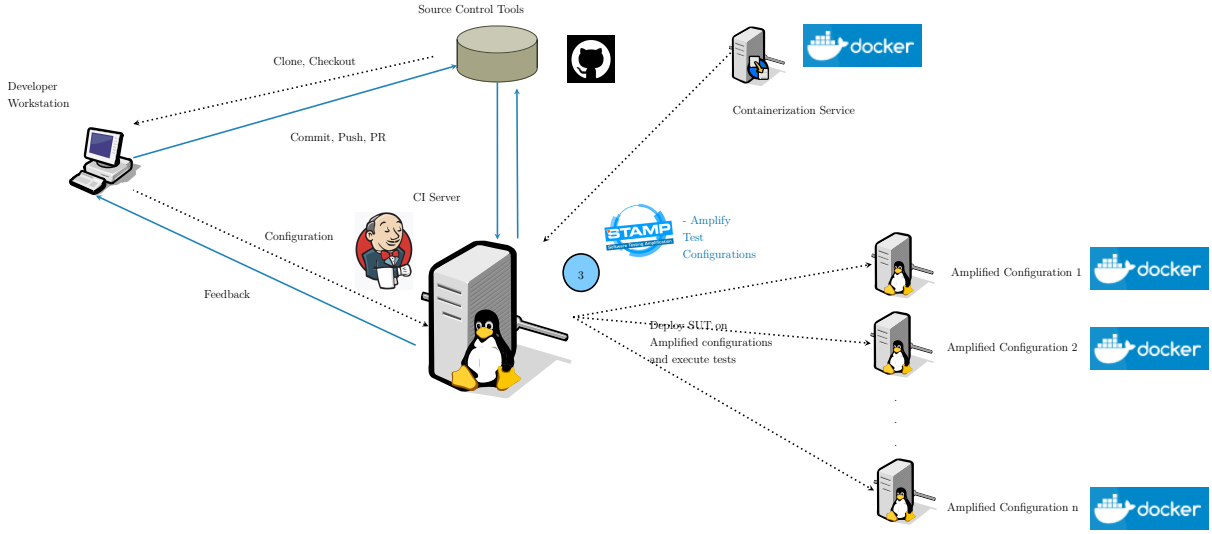


Figure 2.8: STAMP CD Test Configuration Amplification reference scenario

In this context developers leverage Docker technology to adopt IaaC practices: in particular, target environments are based on Docker files and on a Docker compose file.

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 the components.

Thanks to CAMP (see https://github.com/STAMP-project/camp) developers can automate the process of generating more and more configurations and executing tests against them.

Two possible sub-scenarios are described in next sub-sections.

2.2.2.1 Test Configuration amplification - Functional tests

Figure 2.9 shows a typical project layout supporting IaaC and test configuration amplification CAMP-based.

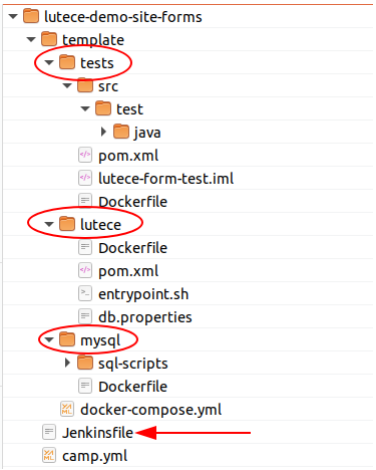


Figure 2.9: Typical project layout supporting IaaC and test configuration amplification

This project, for instance, has three components which make the whole project infrastructure:

1. `tests`: contains system functional tests (in this case they are implemented with Selenium. Source code is in the sub-folder `src/`). The `Dockerfile` scripts building and execution of the test client;
2. `lutece`: contains the SUT (in this case an instance of Lutece configured to offer a demo site). The Dockerfile contains all the commands to have a Lutece instance up & running;
3. `mysql`: Lutece needs a database to persists data. This database is a separated component which can vary among different database products (MySQL, MariaDB, etc);

The `docker-compose.yml` file defines all the 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 in the existing IaaC scripts is pulled in the repository, after usual build and unit test execution, trigger an execution of CAMP 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, publish test execution reports within Jenkins dashboard
3. make generated configurations available within Jenkins dashboard

Referring 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).

Now lets analyze 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 BlueOcean 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.

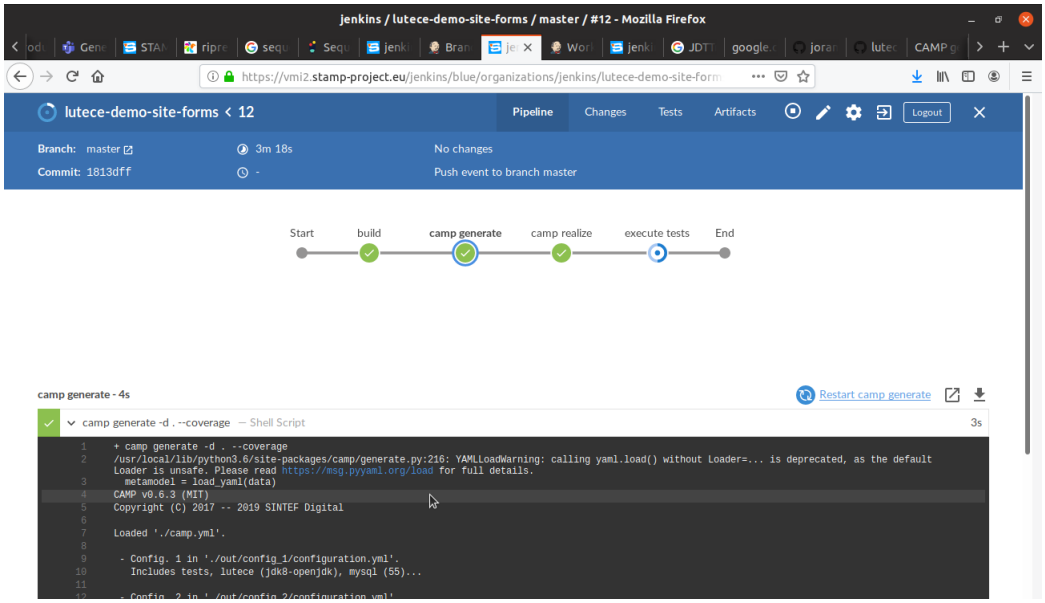


Figure 2.10: Test Configuration amplification pipeline execution, shown by Blue Ocean interface

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 having 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 would generate only the subset that covers all possible variations;

2. `camp realize -d .` stage builds the docker images and the docker-compose file that we need to run the configurations that we generated 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 are accessible in section `Artifacts` of current build:

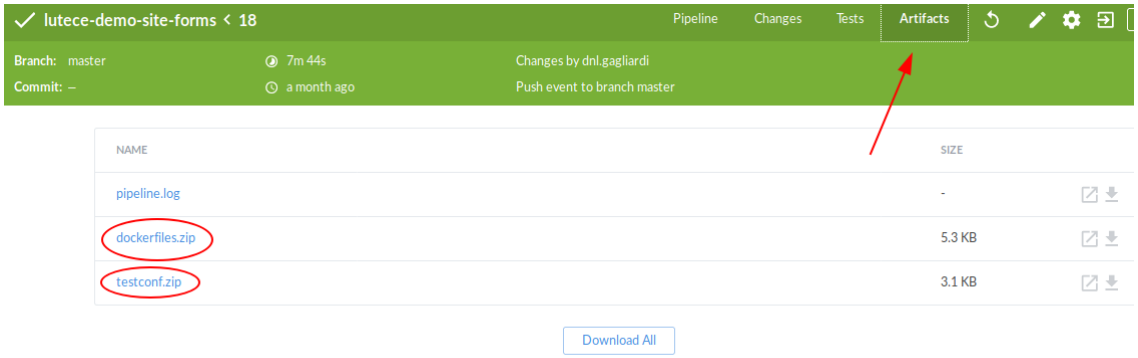


Figure 2.11: Amplified test configurations available as current build artifacts, within Jenkins Blue Ocean interface

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

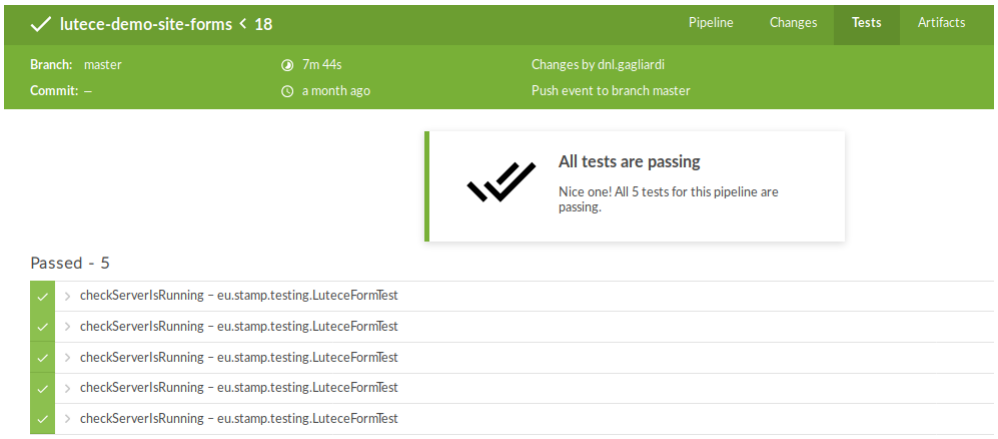


Figure 2.12: Results of Integration/System tests executed on amplified test configurations, shown within Jenkins Blue Ocean interface

2.2.2.2 Test Configuration amplification - Performance tests

Figure 2.13 shows another project layout supporting IaaC and test configuration amplification CAMP-based.

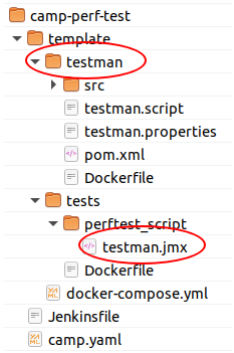


Figure 2.13: A sample project layout supporting IaaC and test configuration amplification, applied to performance testing

This project has two components which make the whole project infrastructure:

1. `tests`: contains system performance tests (in this case they are implemented with Apache JMeter script. The `Dockerfile` scripts the execution of Apache JMeter, fed by `testman.jmx`script;
2. `testman`: contains the SUT (in this case a web application named Testman, a simple test management system to store test cases and test executions (for further details, see https://stamp-project.github.io/camp/pages/execute.html - section **Example: Performance Test A Java WebApp** The Dockerfile contains all the steps to build and have a Testman instance up & running;

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

Putting everything in a CI/CD scenario, developers can automate the whole process of generating more test configurations and executing performance tests against them, inspecting HTML reports generated by JMeter, and made available within Jenkins.

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

The whole 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 BlueOcean 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.

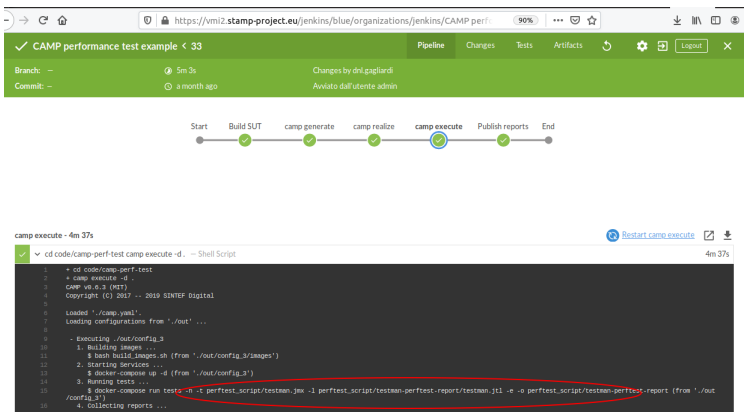


Figure 2.14: Test Configuration amplification pipeline execution for performance tests, shown by Blue Ocean interface

Now lets analyze 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 the steps are similar to what we've already seen in previous section (see 2.2.2.1): SUT is built and launched, CAMP generates and realizes more configurations, then executes 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 **??**).

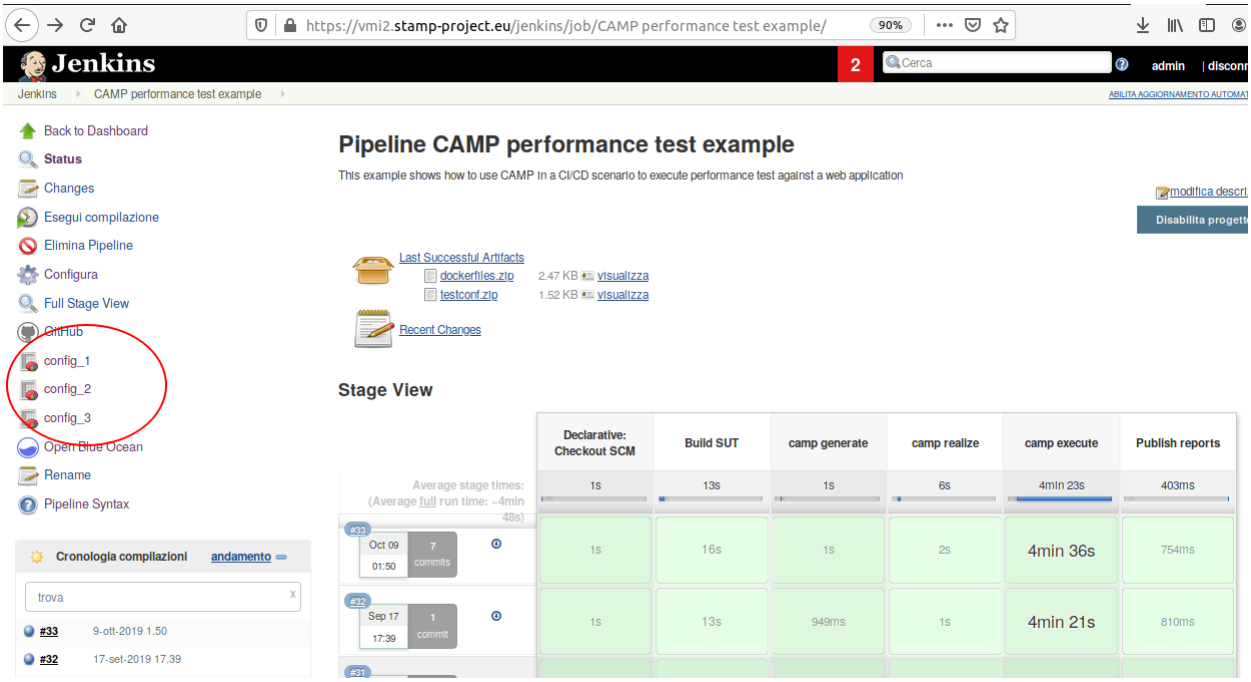


Figure 2.15: JMeter HTML reports available in Jenkins dashboard

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

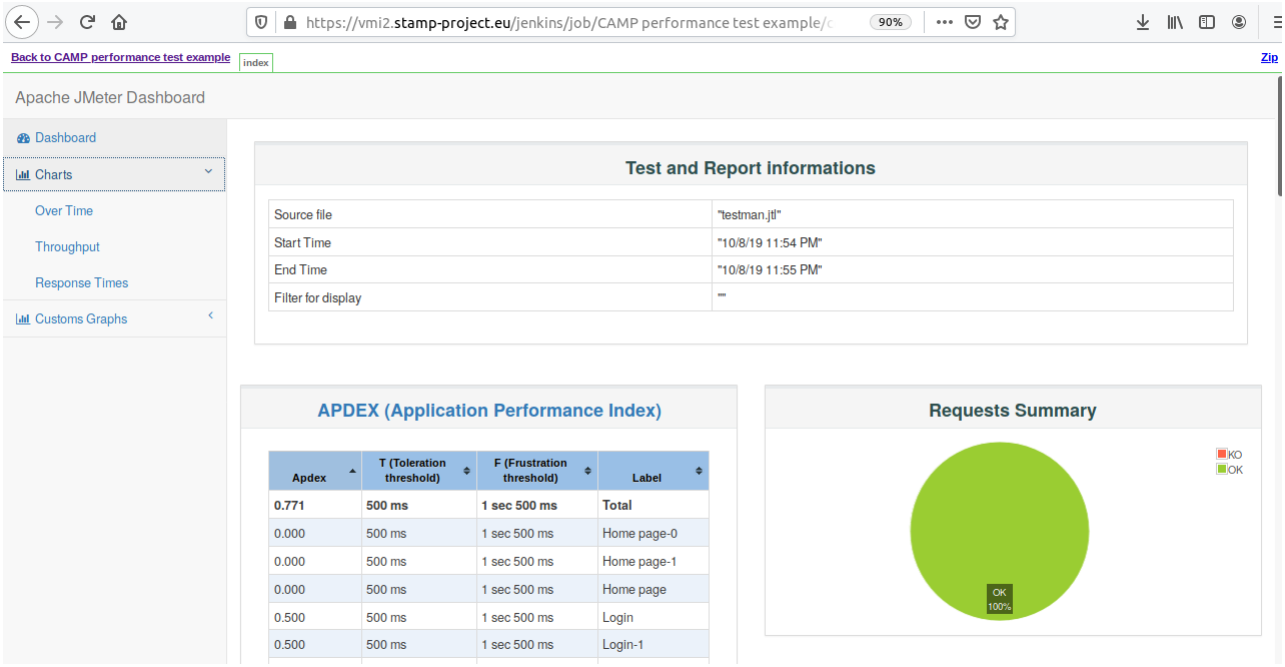


Figure 2.16: Detail pf a single JMeter HTML report

2.2.3 Online Test amplification

This section shows which are the STAMP online test amplification features that developers can adopt within their DevOps processes to speed-up the bug investigation phase. Figure 2.17 shows the detailed online test amplification reference scenario.

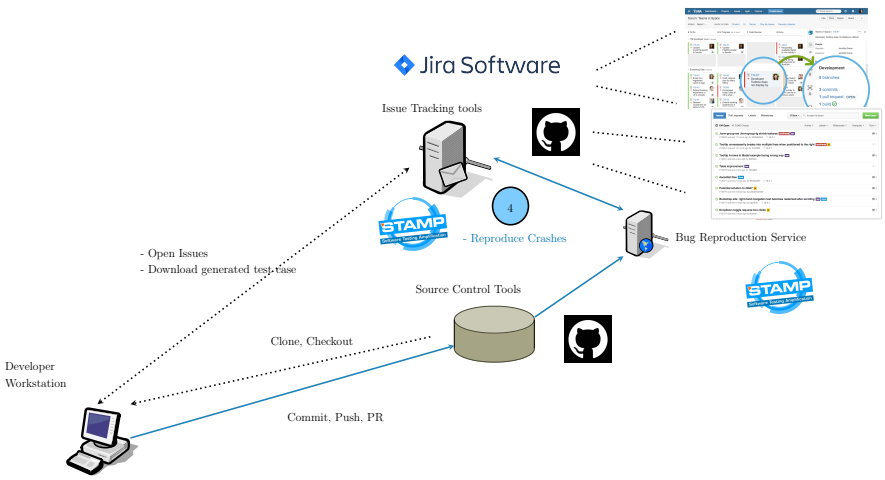


Figure 2.17: STAMP CI Online Test Amplification reference scenario

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 integrated in them the automatic crash reproduction feature.

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

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 by the means of Botsing Maven plugin - see https://github.com/STAMP-project/botsing/tree/master/botsing-maven) to reproduce the stack-trace: at first it cleans the stack-trace, thanks to Botsing pre-processor (see https://github.com/STAMP-project/botsing/tree/master/botsing-preprocessing, then it executes Botsing, starting from the highest frame, and iterating on all the frames until it finds a test case. 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 closed-source 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 thanks to a Jira plugin (Botsing Jira plugin - see https://github.com/

STAMP-project/botsing-jira-plugin), which adds this feature within ordinary Jira tasks: in fact, 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 2.18).

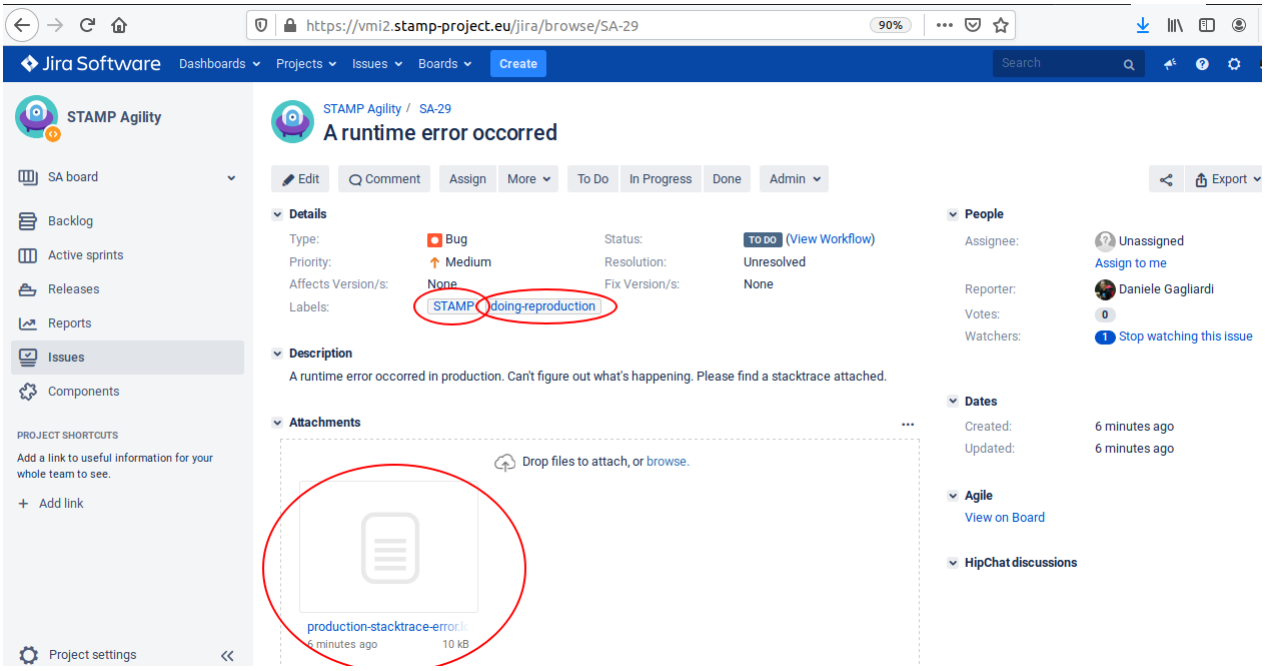


Figure 2.18: Triggering automatic crash reproduction with Jira

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 (2.19).

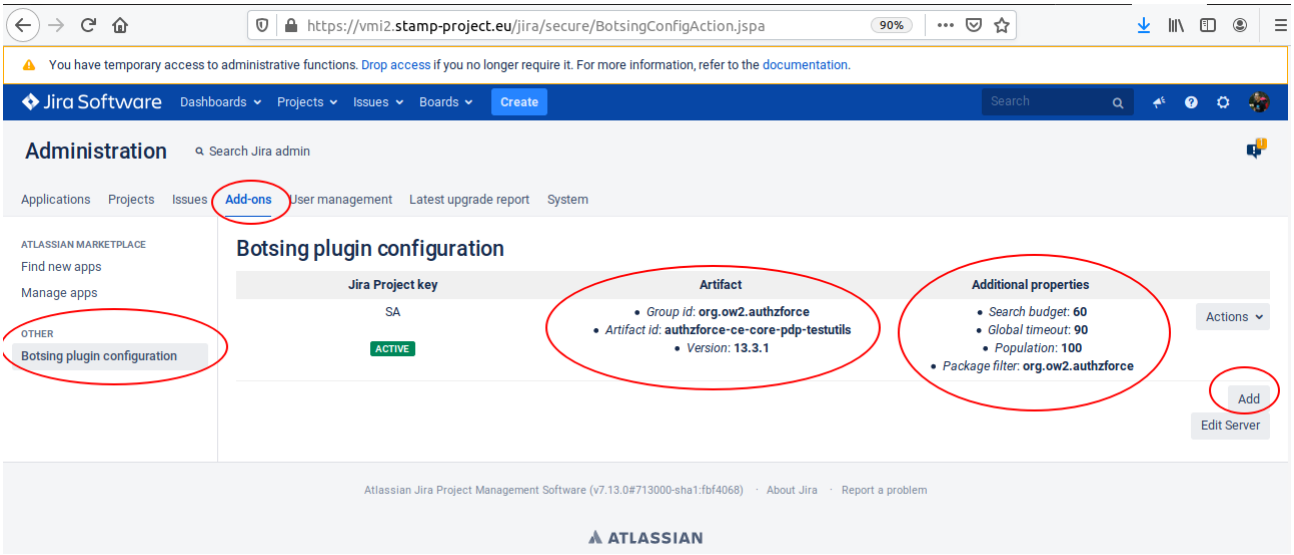


Figure 2.19: Software GAV configuration to let Botsing Server download dependencies for project classpath parameter needed by Botsing

The **Add** button lets developers to configure automatic crash reproductions for other Jira projects.

In the same section, the developer can configure the address of the remote Botsing Server (2.20).

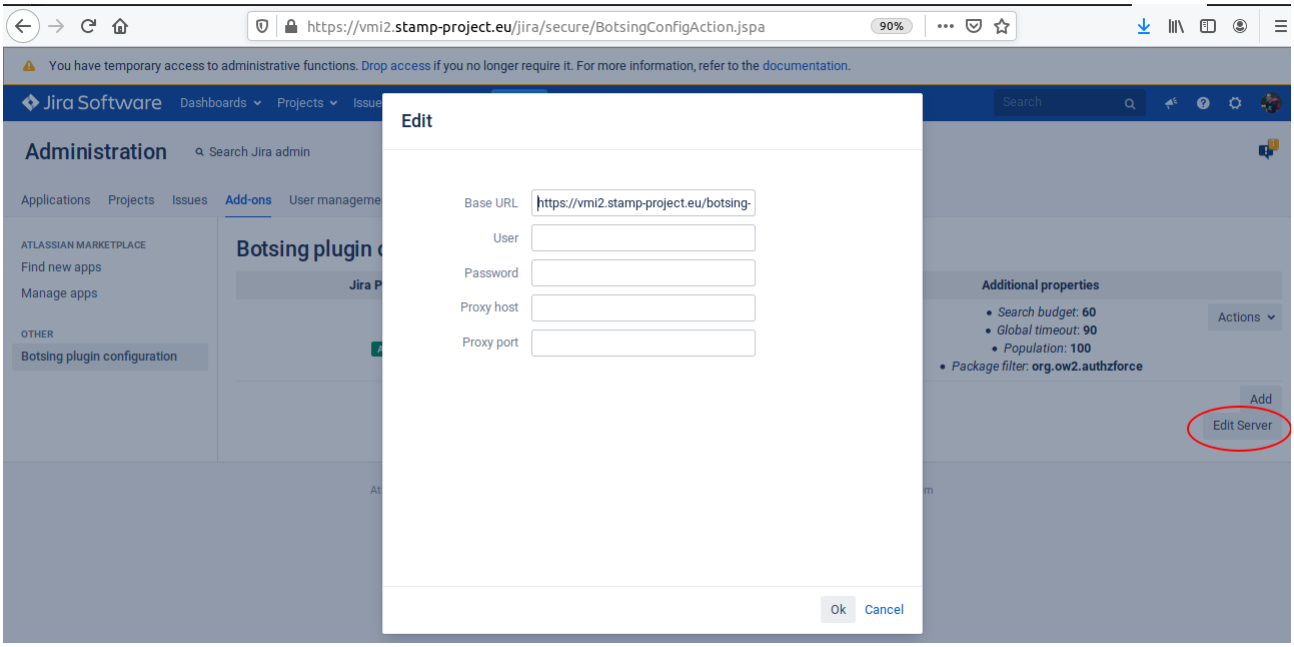


Figure 2.20: Botsing Server configuration in Jira

When Botsing server completes the generation of the new test case, it is forwarded to Jira as attachment and includes a comment with the reproduction result (figure 2.21).

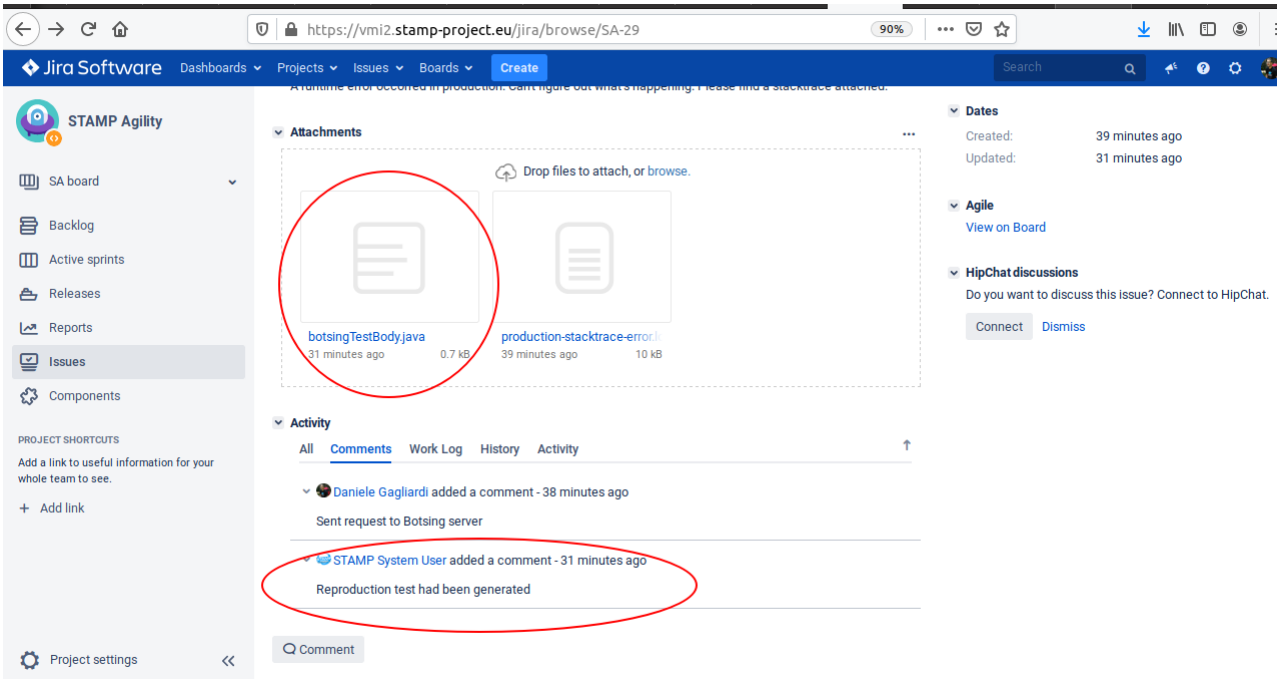


Figure 2.21: Botsing-generated test case attached to Jira issue

The picture below shows the test case content:

Figure 2.22: 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 2.23).

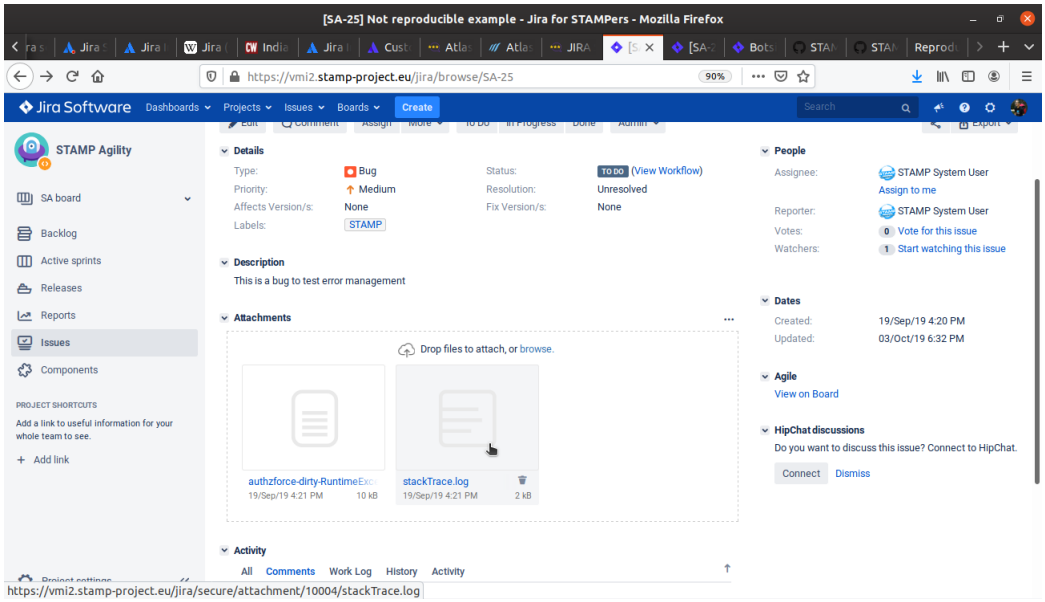


Figure 2.23: Automatic crash reproduction failure

Botsing server attaches also a log error, to let developers investigate what went wrong (figure 2.24).

Figure 2.24: 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 2.25).

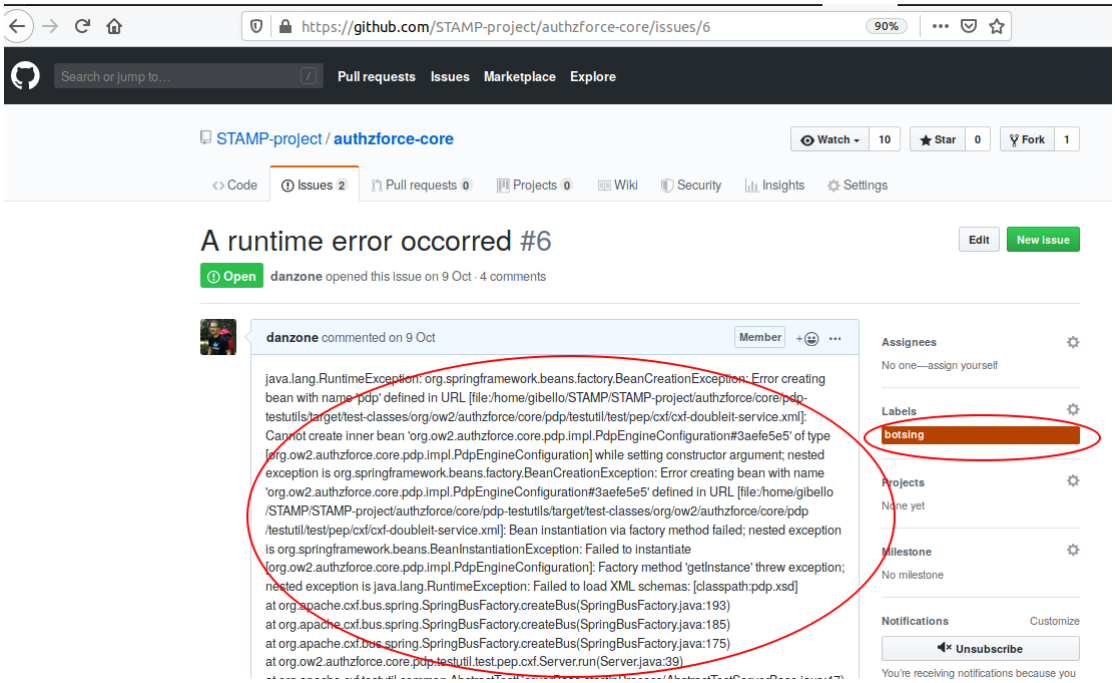


Figure 2.25: Triggering automatic crash reproduction with GitHub Issue

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 2.26).

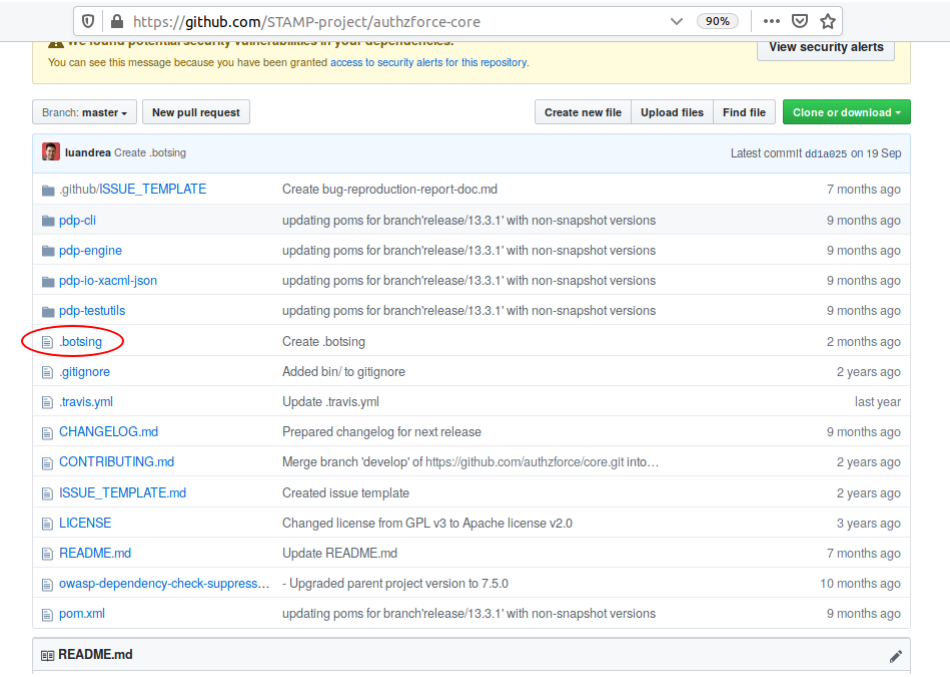


Figure 2.26: Botsing GitHub configuration

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 2.27).

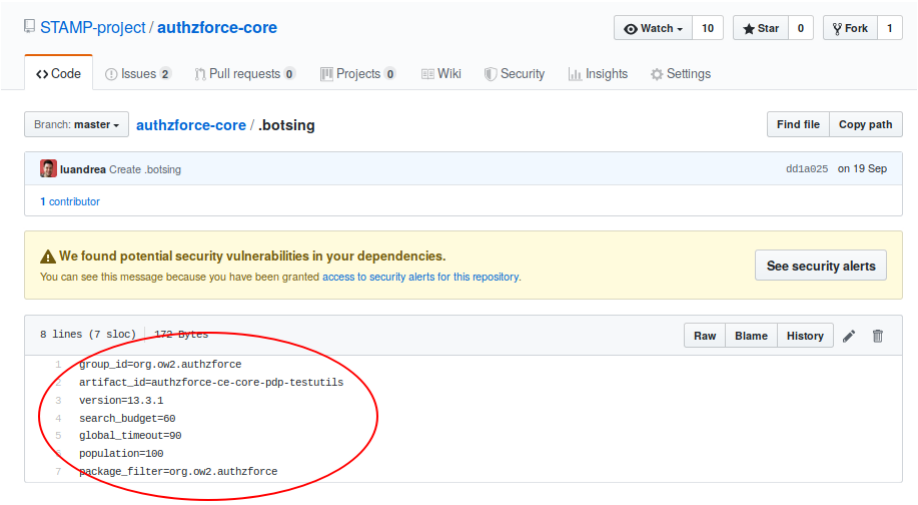


Figure 2.27: GitHub Botsing configuration detail, containing GAV and other Botsing parameters

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 **??**)

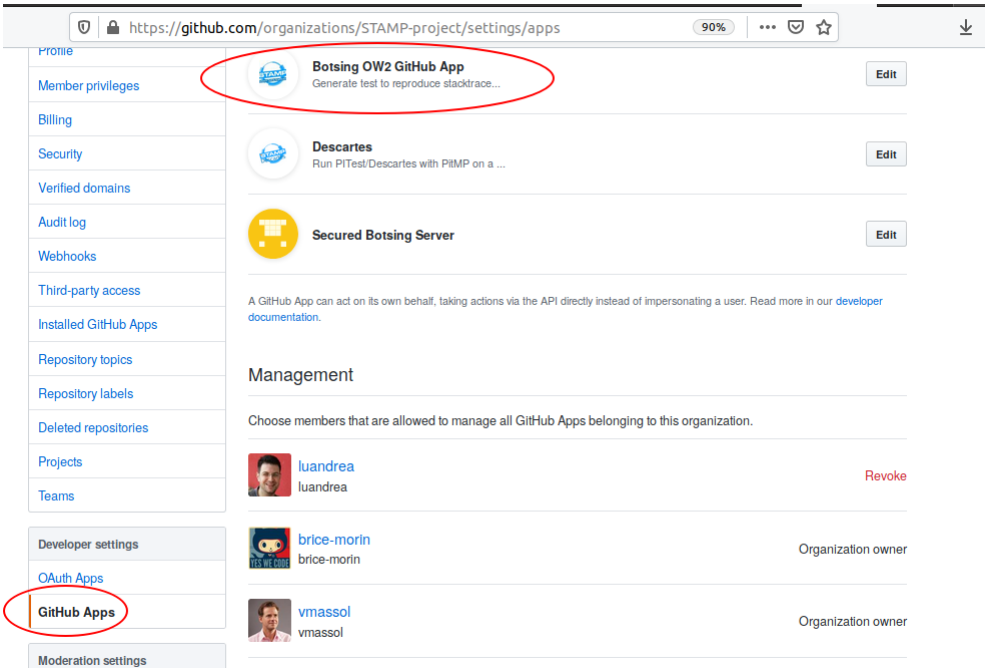


Figure 2.28: Botsing GitHub App configuration

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

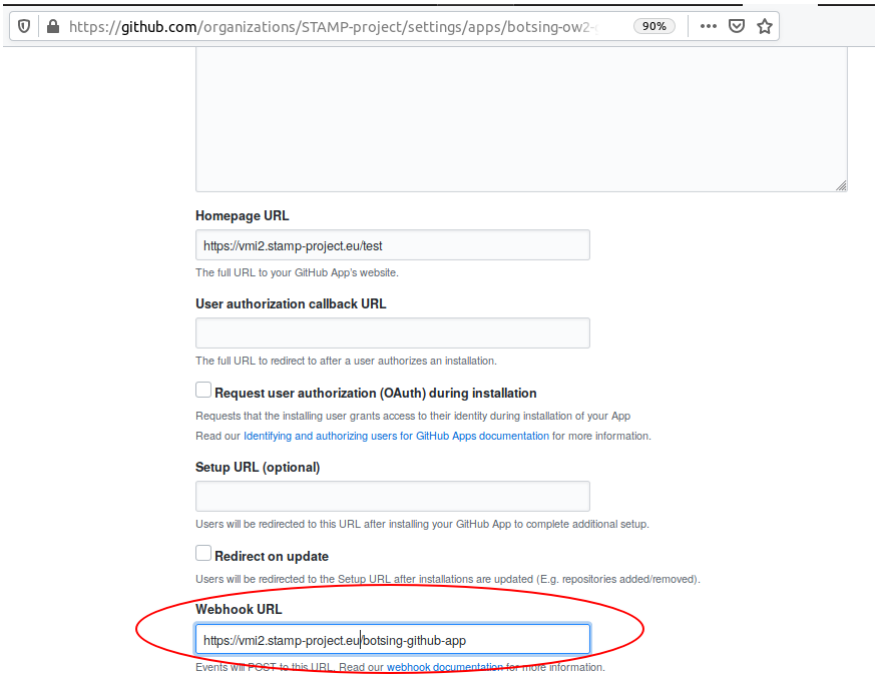


Figure 2.29: Botsing GitHub App configuration detail, containing Web-hook used by GitHub on Issue creation/edit events

2. once configured, the GitHub App should be installed, specifying the associated projects (figure 2.30).

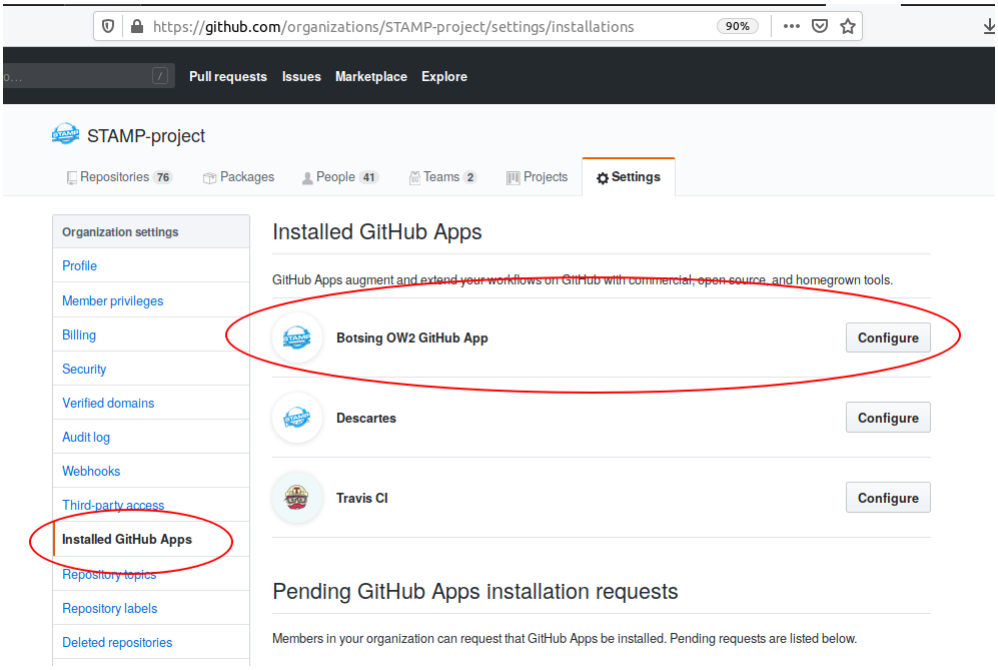


Figure 2.30: Botsing GitHub App activation

Figure 2.31 contains the configuration details and the needed authorization (performing operations on issues and metadata) for the GitHub App and the associated projects.

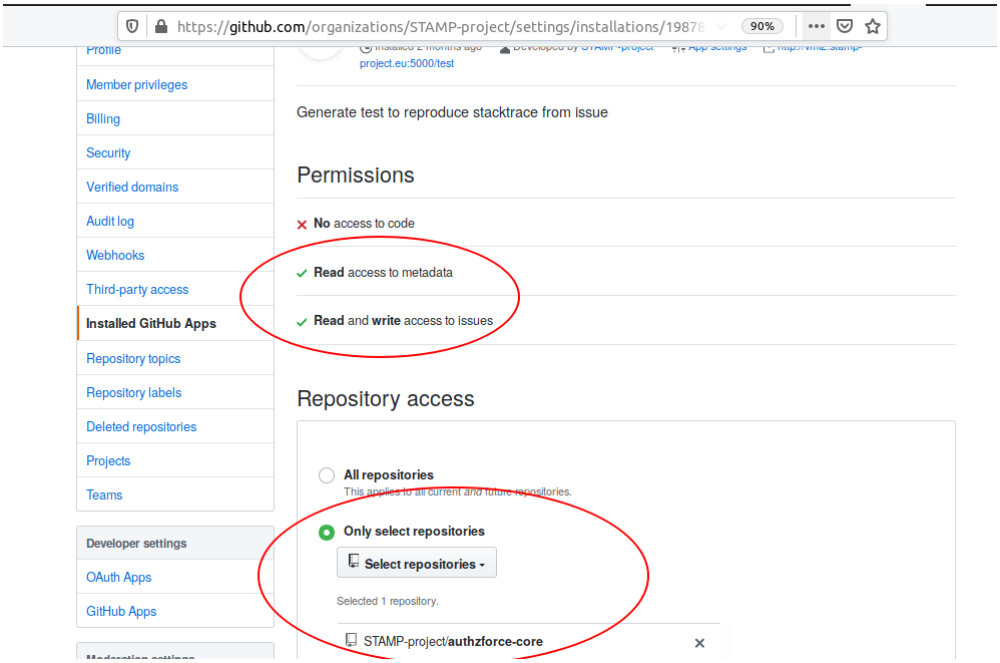


Figure 2.31: Botsing GitHub App activation detail, containing permissions and projects on which it will be activated

When Botsing Server completes the generation of the test case, it sends back to GitHub Issue the result as a comment (figure 2.32).

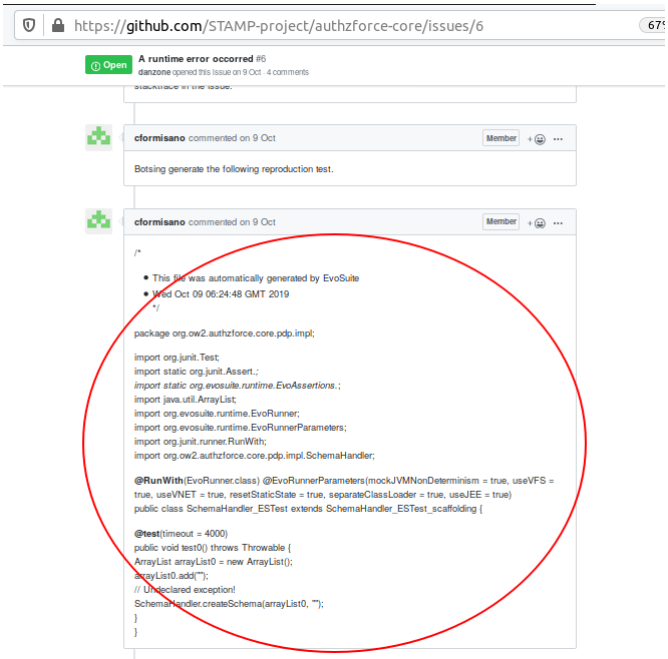


Figure 2.32: Botsing-generated test case added as a comment to GitHub issue

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 (2.33).

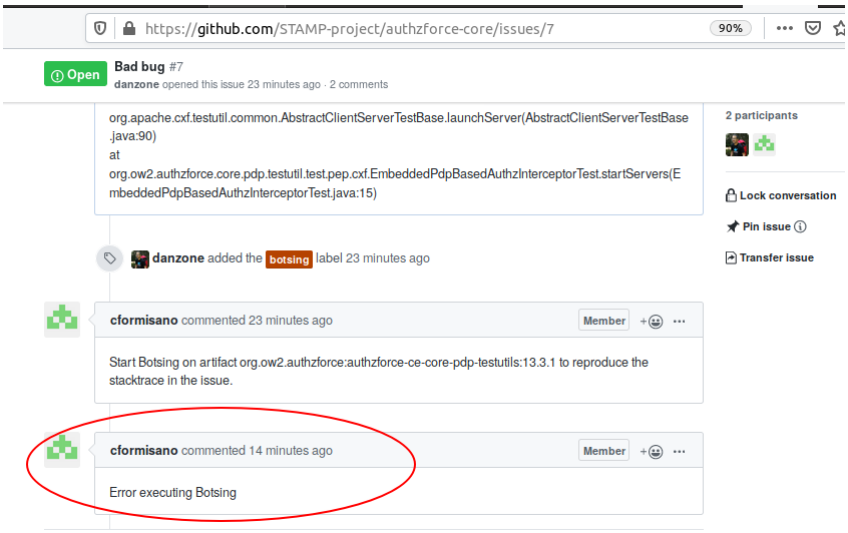


Figure 2.33: Automatic crash reproduction failure

2.3 STAMP CI/CD Architecture

In this section we'll describe briefly the STAMP CI/CD Architecture, showing which are main components that constitute the STAMP CI/CD scenario, and how they interact each other. The diagram in figure 2.34 shows STAMP CI/CD architecture.

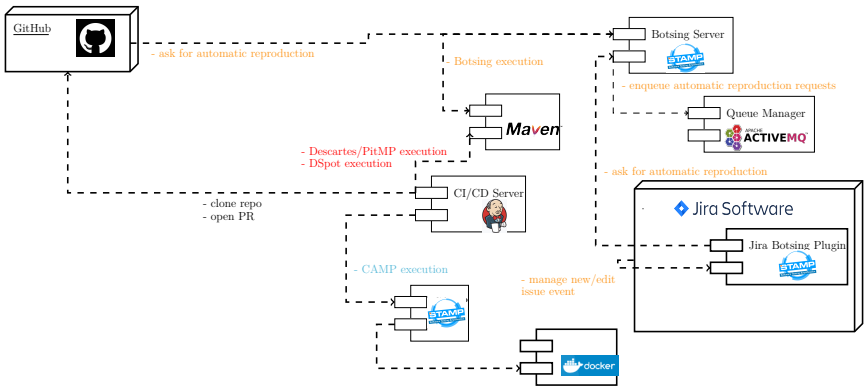


Figure 2.34: STAMP CI/CD architecture

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. CAMP modules to execute functional tests and to execute performance tests with JMeter are part of CAMP official distribution, so, except for Python3, Docker, Docker compose and Z3 solver (requirements for CAMP), anything else 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

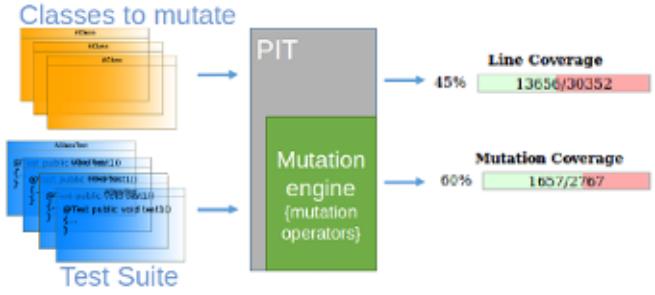
During past year we kept all Maven and Gradle plugin updated with latest STAMP tools versions, as reported also in Deliverable 1.4, section 5.3, so they won't be described here (with the exception of Botsing Gradle plugin whose development has been completed during the past year). Maven plugins opened the way to introduce STAMP features within ordinary CI/CD processes, described by Jenkins pipelines.

3.1.1 PitMP - PIT for Multi-module Project

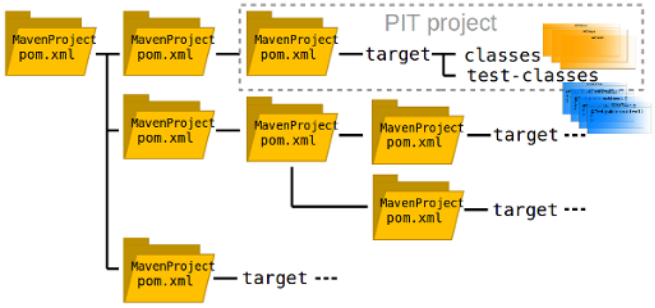
PitMP (PITest for Multi-module Project) is a Maven plugin able to run PITest on multi-module projects. PITest is a mutation testing system for Java applications, which allows you to evaluate the quality of your test suites. Since Descartes is a PITest plugin, PitMP also provide a full support to run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

3.1.2 How does PitMP work ?

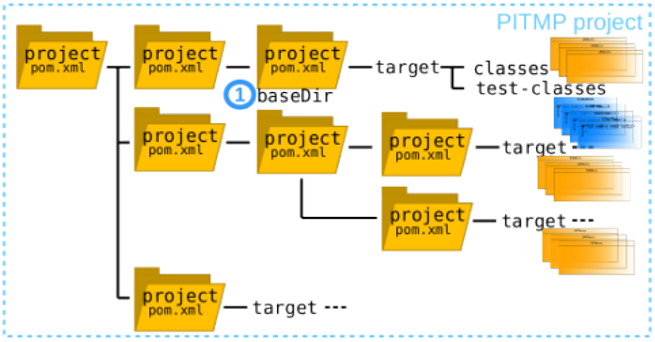
PITest takes a test suite, a set of classes to be mutated and a set of mutation operators and computes a line coverage and a mutation coverage:



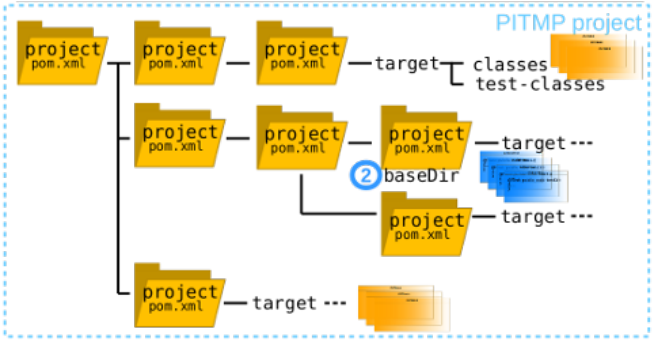
PITest mutates only the classes defined in the same module than the test suite:



PitMP runs PITest on every test suite, mutating classes of all dependencies of modules located in the same project tree:



then:



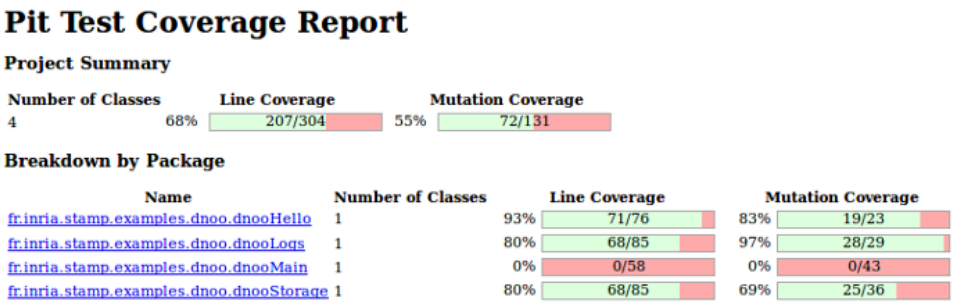
etc...

PitMP just extends PITest, it doesn't rewrite any feature, so all PITest's properties can be used. PitMP runs test suite as PITest does, just extending the list of classes to be mutated to the whole project tree, instead of mutating only the classes of the test suite module.

3.1.3 PitMP Output

PITest produces a report that includes:

- a summary of line coverage and mutation coverage scores:



Pit Test Coverage Report

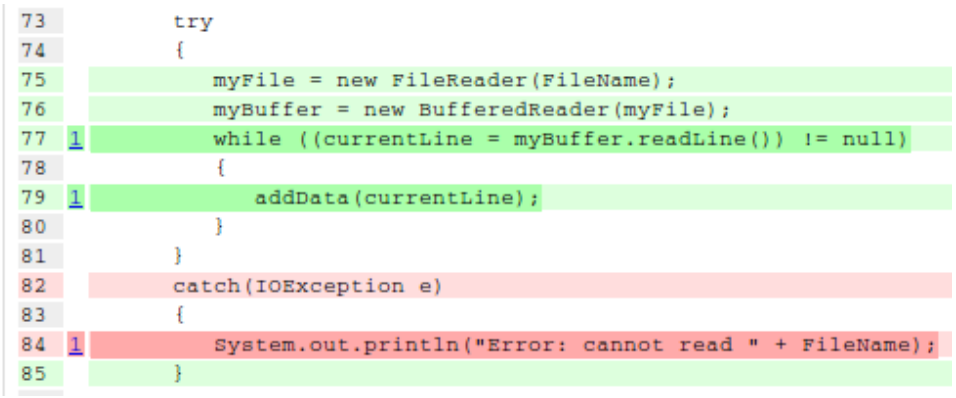
Project Summary

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.

To be completed with Descartes specific reports

3.1.4 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`

3.1.5 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.

3.1.6 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>
```

3.1.7 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>
```

3.1.8 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

The tests executed by the use cases with Botsing 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, 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 https://github.com/STAMP-project/botsing-gradle-plug 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 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 our experiments, we highlight that it was difficult to find the correct target frame. As a consequence, we also decided 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.

3.1.9 DSpot and Descartes Jenkins integration

As already reported in Deliverable 1.4, section 5.3.3, we completed the development of two Jenkins plugins able to visualize DSpot and Descartes specific reports. These plugins can also be combined with declarative Jenkins pipelines, adding an optional step to visualize the report within Jenkins dashboard. Descartes Jenkins plugin was developed during 2018, and in subsequent months we performed several maintenance activities (bug fixing, updating with new versions of Descartes). DSpot Jenkins plugin development started at the end of 2018, and, as Descartes plugin, it provides a dashboard to inspect and possibly download 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

We focused on the goal of optimizing execution time of DSpot in the CI/CD (for every commit on a branch), executing 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, we evaluated two possible approaches, 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 3.1: 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-selec -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=tes`

```
-Dverbose -Dtest-criterion=ChangeDetectorSelector -Dpath-to-properties=src/main/resource
-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 3.2: 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 3.3: 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 3.4: 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
    - 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 the maximum number of tests amplified or the budgetizer to be used, you can also select the specific test cases to amplify (the selected cases must be part of the selected test classes).
    - Page four : Additional configuration, in this page you can select test classes, cases or even packages to be excluded from the amplification process, in the bottom of the page you will find the 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).

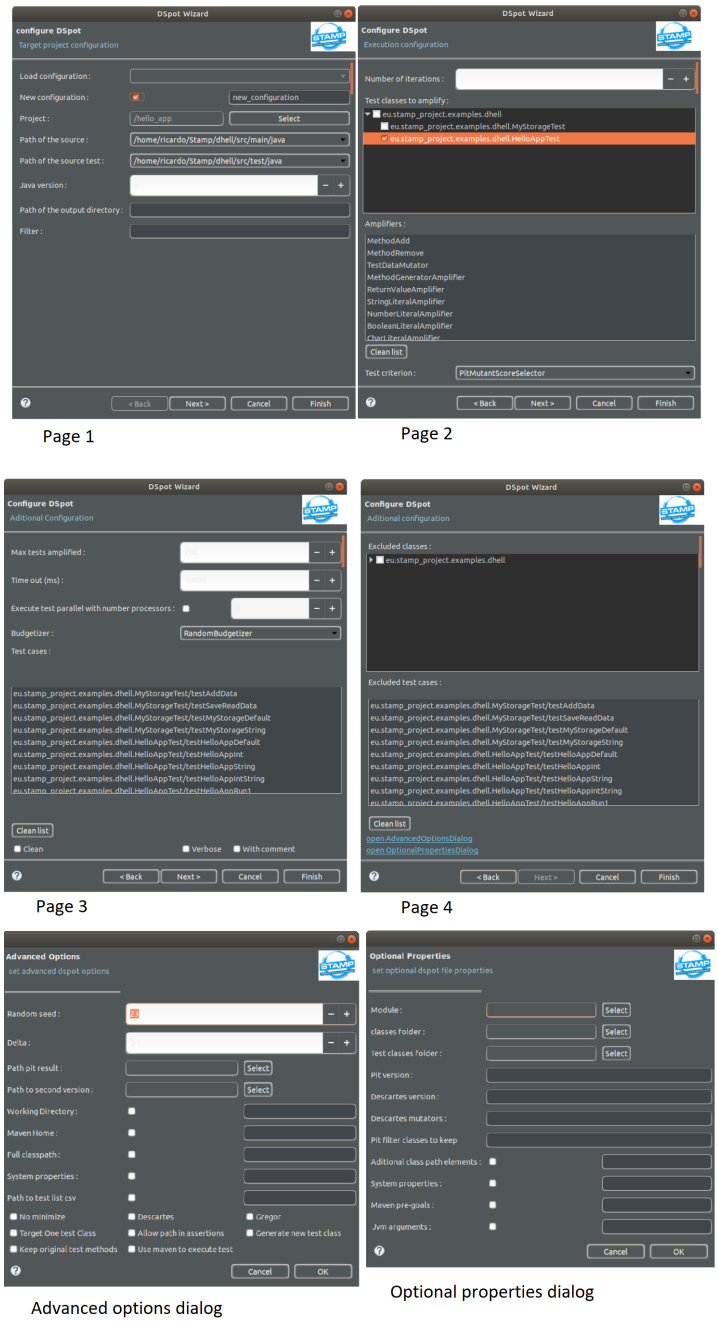


Figure 3.1: DSpot Wizard

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, now we explain some new features and changes.

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

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

in the Eclipse preferences facility in a page called "Descartes Jira preferences".

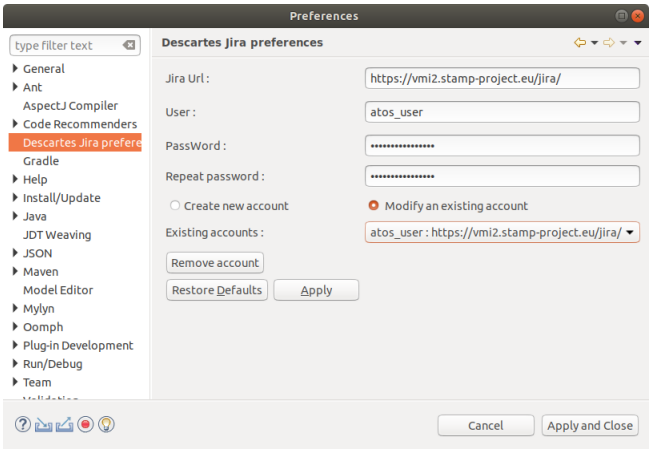


Figure 3.2: Jira preferences page implemented by Descartes plugin

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.

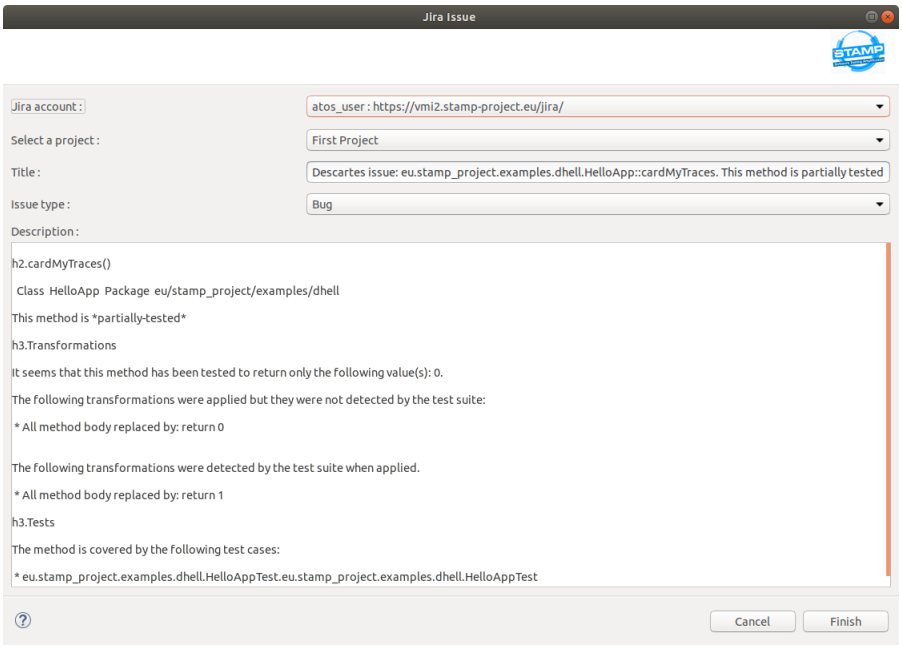


Figure 3.3: Jira Issue creation wizard

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, now we describe some new features.

- Support for the 1.0.7 Botsing version
- Now you 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 Evosuite Eclipse plugin

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

The wizard is called from a menu entry in the Eclipse main bar.

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).

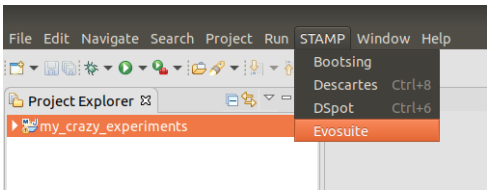


Figure 3.4: menu entry for opening the Evosuite wizard

The configurable options in the wizard main page 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 Evosuite documentation).
4. Seed clone, a real number between 0 and 1.

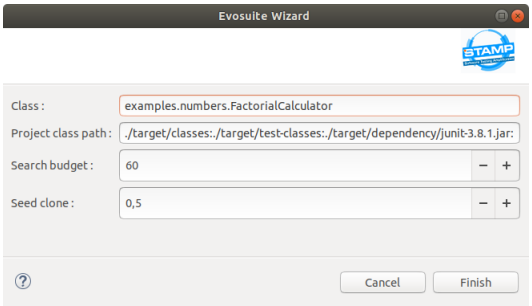


Figure 3.5: Evosuite wizard main page

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

Evosuite is executed as an Eclipse job, so the Evosuite 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 Evosuite work, the field to set the model path can be activated with a check button.

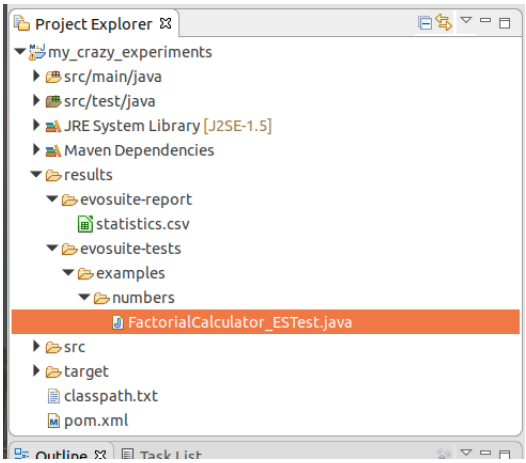


Figure 3.6: Evosuite output

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 Evosuite plugins, the wizard can be accessed from the Botsing or Evosuite wizards, in this case the model generation job will be launched before the Botsing/Evosuite 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 (we preferred to keep all discussions around features or project activities within integrated issue trackers available both in Gitlab and GitHub).

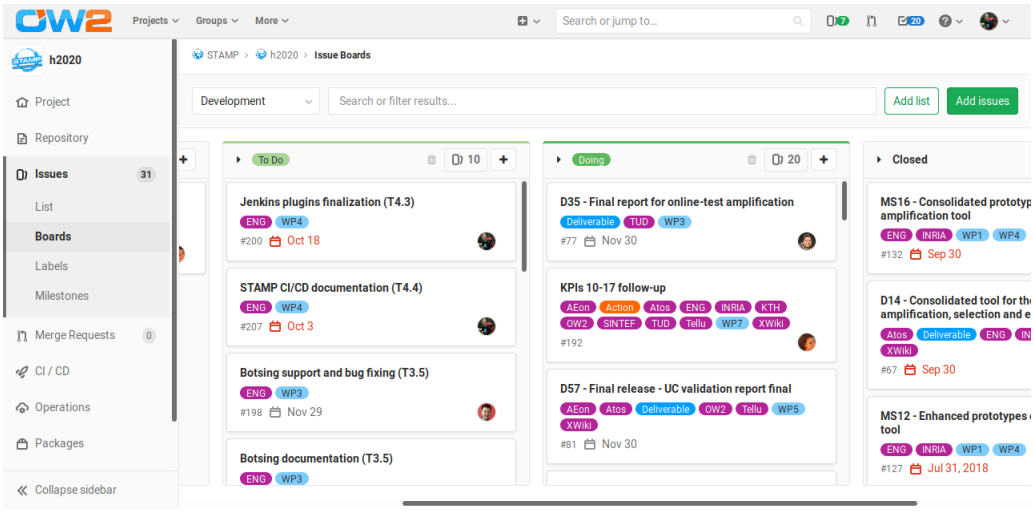


Figure 4.1: STAMP discussions through issues

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.

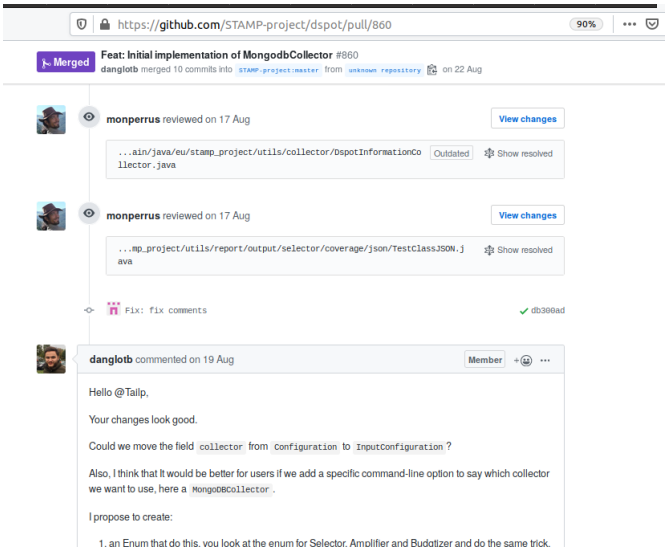


Figure 4.2: STAMP code reviews through PR/MR

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: 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 we'll provide 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).

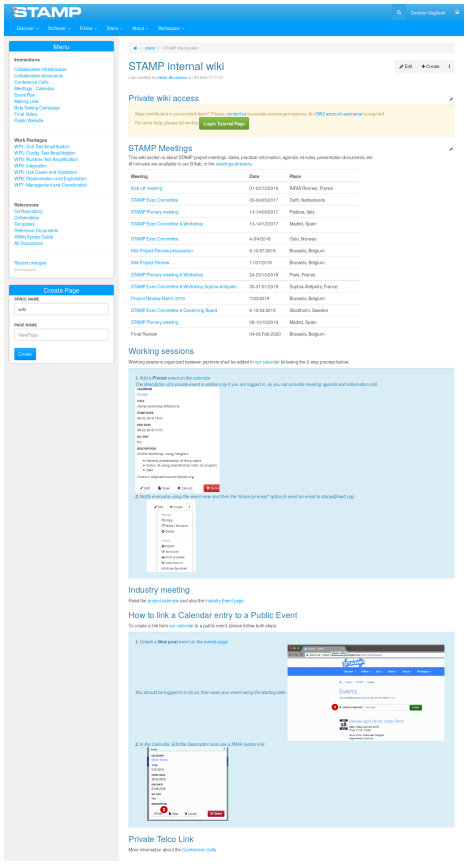


Figure 4.4: STAMP private portal

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

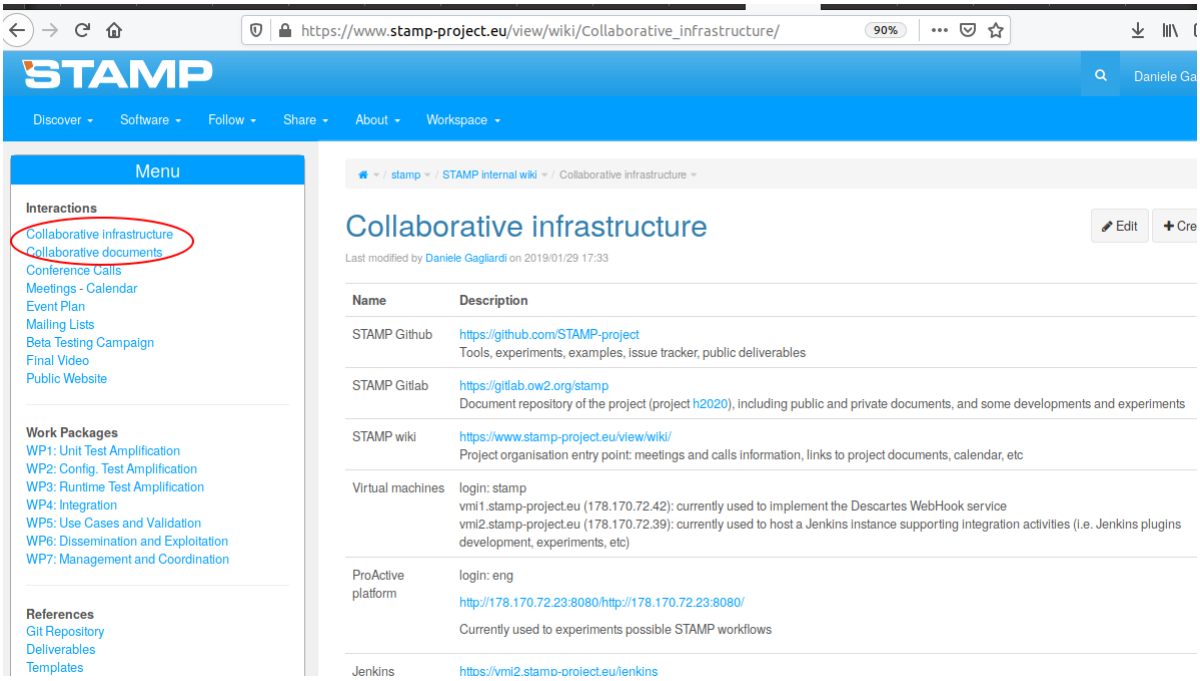


Figure 4.5: reference to STAMP dev,test,demo environments

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

4.2 STAMP Demo server

In order to validate the CI/CD reference architecture described in 2.3, we setup, in vmi2.stamp-project.eu server, a complete environment 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:

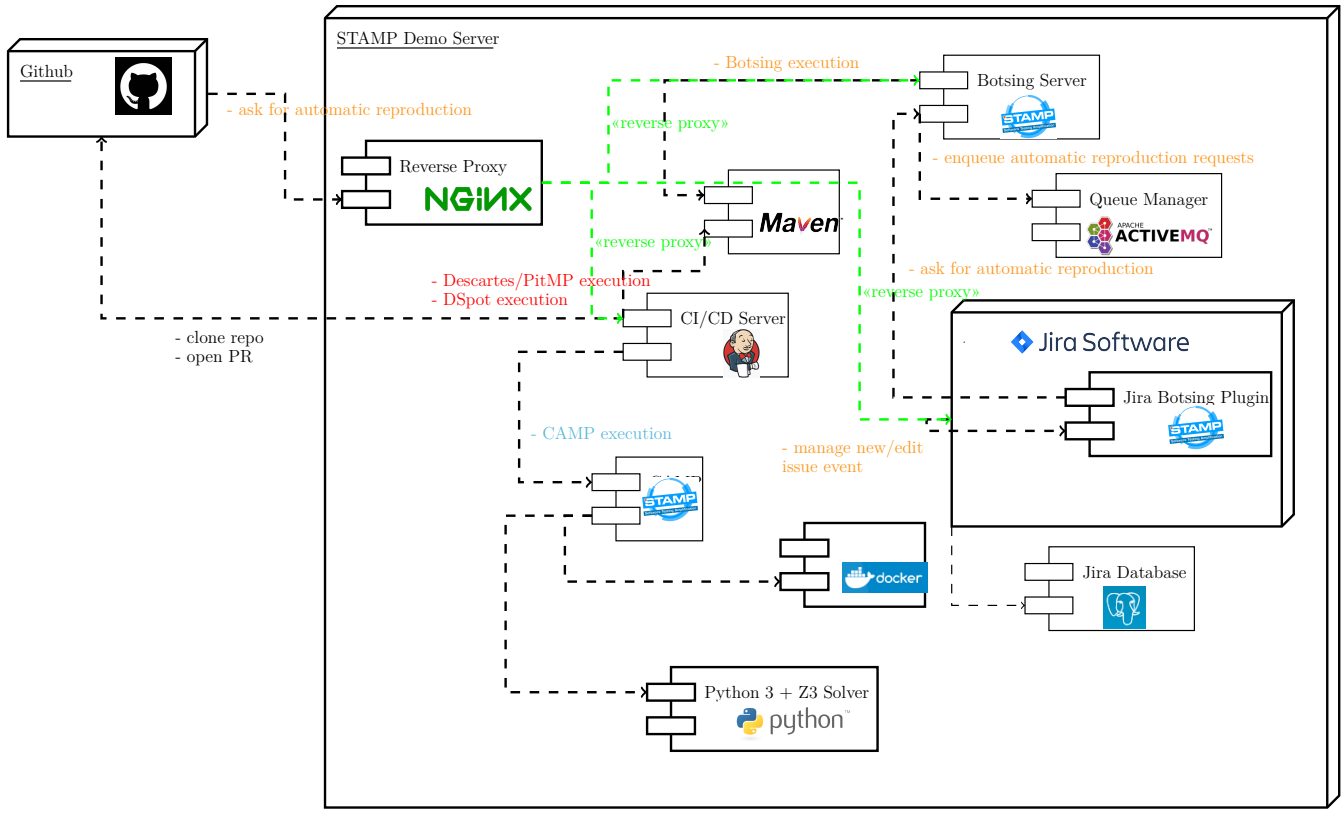


Figure 4.6: STAMP CI/CD architecture reference implementation

Below a short description of main components:

- Jenkins CI: we installed an instance of Jenkins CI, and installed 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 5.2 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 (we replace the embedded database in order to have a real world Jira installation, more reliable and stable). We then activated a perpetual license for ten users, in order to have a fully-functional environment, with no time constraints (due to temporary licenses). We then installed Botsing plugin in Jira;
- Docker and Docker container: we installed the latest available versions for Debian
- Python 3: we replaced default Python installation (Python 2.7) with 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 about reverse proxy configuration for Botsing Server, Jira 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;
  ignore_invalid_headers off; #pass through headers from
     Jenkins which are considered invalid by Nginx server.

  location ~ "^/static/[0-9a-fA-F]{8}\/(.*)$" {
    #rewrite all static files into requests to the root
    #E.g /static/12345678/css/something.css will become /css/
       something.css
    rewrite "^/static/[0-9a-fA-F]{8}\/(.*)" /$1 last;
  }

  location /userContent {
    #have nginx handle all the static requests to the
       userContent folder files
    #note : This is the $JENKINS_HOME dir
        root /var/lib/jenkins/;
    if (!-f $request_filename){
      #this file does not exist, might be a directory or a
         /**view** url
      rewrite (.*) /$1 last;
          break;
    }
        sendfile on;
  }

  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 https;
        #proxy_set_header   X-Forwarded-Proto $scheme;
        proxy_max_temp_file_size 0;

        #this is the maximum upload size
        client_max_body_size       10m;
        client_body_buffer_size    128k;

        proxy_connect_timeout      90;
        proxy_send_timeout         90;
        proxy_read_timeout         90;
        proxy_buffering            off;
        # proxy_request_buffering    off; # Required for HTTP
           CLI commands in Jenkins > 2.54
        proxy_set_header Connection ""; # Clear for keepalive
    }
    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 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 has 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 paramters, 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:

  - Evosuite model seeding tutorial, available at https://github.com/STAMP-project/evosuite-model-seeding-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 usgin STAMP in DevOps (5.1).

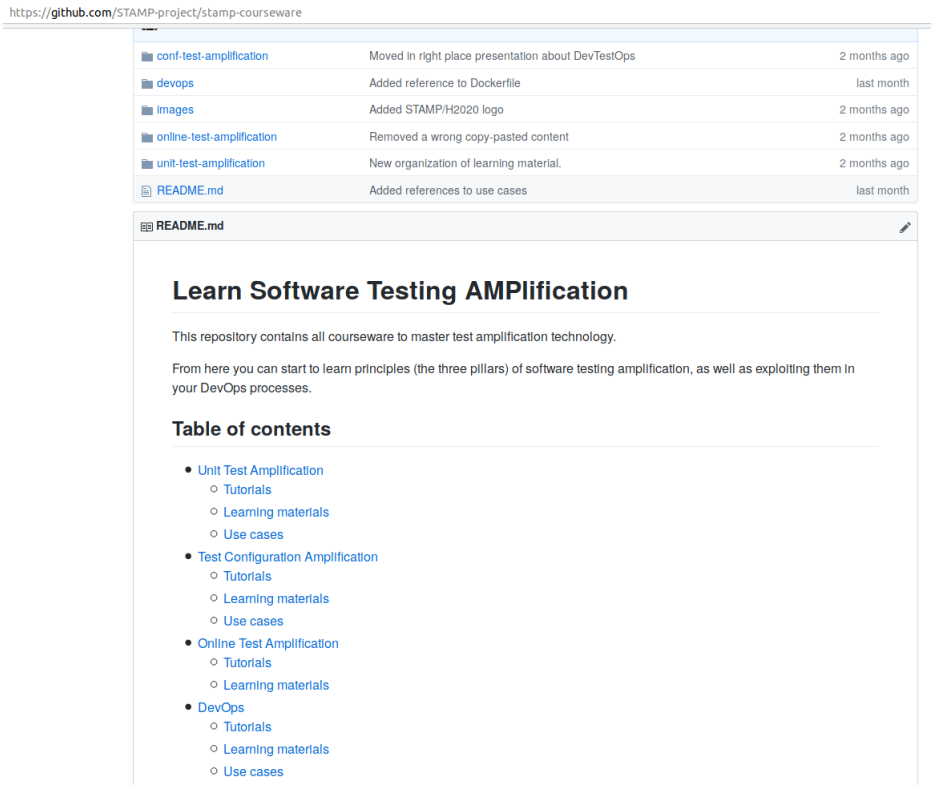


Figure 5.1: STAMP courseware path

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 09:52:04.821+0000 [id=1] INFO  winstone.Logger#
   logInternal: Beginning extraction from war file
2019-11-12 09:52:05.840+0000 [id=1] WARNING o.e.j.s.handler.
   ContextHandler#setContextPath: Empty contextPath
2019-11-12 09:52:05.892+0000 [id=1] INFO  org.eclipse.jetty.
   server.Server#doStart: jetty-9.4.z-SNAPSHOT; built: 2019-05-02
   T00:04:53.875Z; git: e1bc35120a6617ee3df052294e433f3a25ce7097;
    jvm 1.8.0_222-b10
2019-11-12 09:52:06.156+0000 [id=1] INFO  o.e.j.w.
   StandardDescriptorProcessor#visitServlet: NO JSP Support for
   /, did not find org.eclipse.jetty.jsp.JettyJspServlet
```

```
2019-11-12 09:52:06.203+0000 [id=1] INFO  o.e.j.s.s.
   DefaultSessionIdManager#doStart: DefaultSessionIdManager
   workerName=node0
2019-11-12 09:52:06.203+0000 [id=1] INFO  o.e.j.s.s.
   DefaultSessionIdManager#doStart: No SessionScavenger set,
   using defaults
2019-11-12 09:52:06.207+0000 [id=1] INFO  o.e.j.server.session.
   HouseKeeper#startScavenging: node0 Scavenging every 660000ms
Jenkins home directory: /var/jenkins_home found at: EnvVars.
   masterEnvVars.get("JENKINS_HOME")
2019-11-12 09:52:06.605+0000 [id=1] INFO  o.e.j.s.handler.
   ContextHandler#doStart: Started w.@53499d85{Jenkins v2
   .190.1,/,file:///var/jenkins_home/war/,AVAILABLE}{/var/
   jenkins_home/war}
2019-11-12 09:52:06.619+0000 [id=1] INFO  o.e.j.server.
   AbstractConnector#doStart: Started ServerConnector@302c971f{
   HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
2019-11-12 09:52:06.619+0000 [id=1] INFO  org.eclipse.jetty.
   server.Server#doStart: Started @2391ms
2019-11-12 09:52:06.620+0000 [id=22]  INFO  winstone.Logger#
   logInternal: Winstone Servlet Engine v4.0 running: controlPort
   =disabled
2019-11-12 09:52:08.405+0000 [id=29]  INFO  jenkins.
   InitReactorRunner$1#onAttained: Started initialization
2019-11-12 09:52:08.433+0000 [id=33]  INFO  jenkins.
   InitReactorRunner$1#onAttained: Listed all plugins
2019-11-12 09:52:09.375+0000 [id=37]  INFO  jenkins.
   InitReactorRunner$1#onAttained: Prepared all plugins
2019-11-12 09:52:09.380+0000 [id=39]  INFO  jenkins.
   InitReactorRunner$1#onAttained: Started all plugins
2019-11-12 09:52:09.387+0000 [id=36]  INFO  jenkins.
   InitReactorRunner$1#onAttained: Augmented all extensions
2019-11-12 09:52:09.989+0000 [id=42]  INFO  jenkins.
   InitReactorRunner$1#onAttained: Loaded all jobs
2019-11-12 09:52:10.005+0000 [id=55]  INFO  hudson.model.
   AsyncPeriodicWork$1#run: Started Download metadata
2019-11-12 09:52:10.013+0000 [id=55]  INFO  hudson.util.Retrier#
   start: Attempt #1 to do the action check updates server
2019-11-12 09:52:10.704+0000 [id=30]  INFO  o.s.c.s.
   AbstractApplicationContext#prepareRefresh: Refreshing org.
   springframework.web.context.support.
   StaticWebApplicationContext@8aa4843: display name [Root
   WebApplicationContext]; startup date [Tue Nov 12 09:52:10 UTC
   2019]; root of context hierarchy
2019-11-12 09:52:10.704+0000 [id=30]  INFO  o.s.c.s.
   AbstractApplicationContext#obtainFreshBeanFactory: Bean
   factory for application context [org.springframework.web.
   context.support.StaticWebApplicationContext@8aa4843]: org.
   springframework.beans.factory.support.
   DefaultListableBeanFactory@78e4012e
2019-11-12 09:52:10.714+0000 [id=30]  INFO  o.s.b.f.s.
```

```
   DefaultListableBeanFactory#preInstantiateSingletons: Pre-
   instantiating singletons in org.springframework.beans.factory.
   support.DefaultListableBeanFactory@78e4012e: defining beans [
   authenticationManager]; root of factory hierarchy
2019-11-12 09:52:10.890+0000 [id=30]  INFO  o.s.c.s.
   AbstractApplicationContext#prepareRefresh: Refreshing org.
   springframework.web.context.support.
   StaticWebApplicationContext@680ab0be: display name [Root
   WebApplicationContext]; startup date [Tue Nov 12 09:52:10 UTC
   2019]; root of context hierarchy
2019-11-12 09:52:10.891+0000 [id=30]  INFO  o.s.c.s.
   AbstractApplicationContext#obtainFreshBeanFactory: Bean
   factory for application context [org.springframework.web.
   context.support.StaticWebApplicationContext@680ab0be]: org.
   springframework.beans.factory.support.
   DefaultListableBeanFactory@39a315d6
2019-11-12 09:52:10.891+0000 [id=30]  INFO  o.s.b.f.s.
   DefaultListableBeanFactory#preInstantiateSingletons: Pre-
   instantiating singletons in org.springframework.beans.factory.
   support.DefaultListableBeanFactory@39a315d6: defining beans [
   filter,legacy]; root of factory hierarchy
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 09:52:14.434+0000 [id=55]  INFO  hudson.model.
   UpdateSite#updateData: Obtained the latest update center data
   file for UpdateSource default
2019-11-12 09:52:14.977+0000 [id=55]  INFO  h.m.
   DownloadService$Downloadable#load: Obtained the updated data
   file for hudson.tasks.Maven.MavenInstaller
2019-11-12 09:52:14.979+0000 [id=55]  INFO  hudson.util.Retrier#
   start: Performed the action check updates server successfully
   at the attempt #1
2019-11-12 09:52:14.985+0000 [id=55]  INFO  hudson.model.
```

```
   AsyncPeriodicWork$1#run: Finished Download metadata. 4,979 ms
2019-11-12 09:52:16.751+0000 [id=30]  INFO  hudson.model.
   UpdateSite#updateData: Obtained the latest update center data
   file for UpdateSource default
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
```

Listing 5.5: STAMP for DevOps Dockerfile:running the container

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

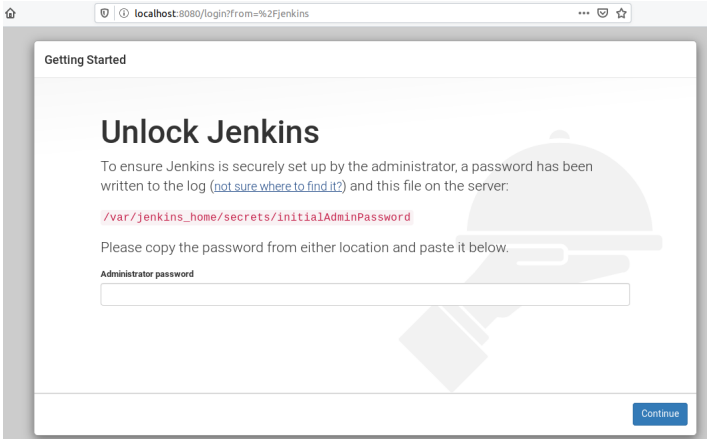


Figure 5.2: STAMP CI/CD Docker image: start-up configuration

The following schema shows the internals of STAMP Docker CI image:

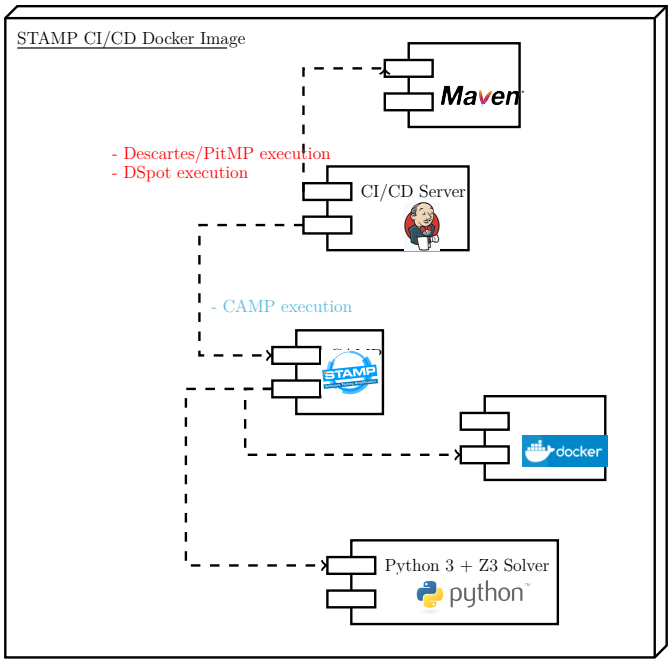


Figure 5.3: STAMP CI Docker image internals

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 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.

In T4.3 we developed all the needed components 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. Moreover we enhanced existing STAMP components 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.

In T4.4 we collected all the documentation produces by all STAMP partners, organizing it in rational structure easy to navigate. Moreover we wrote some documentation providing 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.