Integration with GitLab

GitLab is a well-known CI/CD tool available on-premises and as SaaS.

Xray does not provide yet a plugin for GitLab. However, it is easy to setup GitLab in order to integrate it with Xray Cloud.

Since Xray provides a full REST API, you may interact with Xray, for submitting results for example.

Integration scenarios

JUnit example

In this scenario, we want to get visibility of the automated test results from some tests implemented in Java, using the JUnit framework.

This recipe could also be applied for other frameworks such as NUnit or Robot (if supported).

We need to setup a Git repository containing the code along with the configuration for GitLab build process.

The tests are implemented in a JUnit class as follows.

CalcTest.java

```
package com.xpand.java;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import static org.hamcrest.CoreMatchers.is;
import static org.junit.Assert.assertThat;

public class CalcTest {

    @Before
    public void setUp() throws Exception {

    }

    @After
    public void tearDown() throws Exception {

    }

        @Test
    public void CanAddNumbers()
    {
        assertThat(Calculator.Add(1, 1), is(2));
        assertThat(Calculator.Add(-1, 1), is(0));
    }


    @Test
    public void CanSubtract()
    {
        assertThat(Calculator.Subtract(1, 1), is(0));
        assertThat(Calculator.Subtract(-1, -1), is(0));
        assertThat(Calculator.Subtract(100, 5), is(95));
    }


    @Test
    public void CanMultiply()
    {
        assertThat(Calculator.Multiply(1, 1), is(1));
        assertThat(Calculator.Multiply(-1, -1), is(1));
        assertThat(Calculator.Multiply(100, 5), is(500));
    }


    public void CanDivide()
    {
        assertThat(Calculator.Divide(1, 1), is(1));
        assertThat(Calculator.Divide(-1, -1), is(1));
        assertThat(Calculator.Divide(100, 5), is(20));
    }


    @Test
    public void CanDoStuff()
    {
        assertThat(true, is(true));
    }


}
```

The GitLab configuration file `.gitlab-ci.yml` contains the definition of the build steps, including running the automated tests and submitting the results.

.gitlab-ci.yml

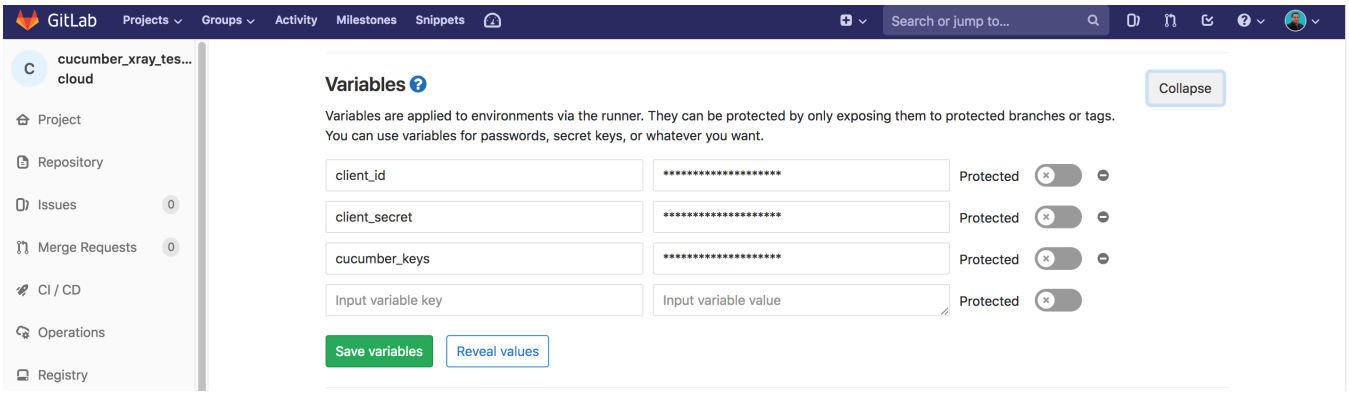
```
# Use Maven 3.5 and JDK8
image: maven:3.5-jdk-8

variables:
  # This will supress any download for dependencies and plugins or upload messages which would clutter the
console log.
  # `showDateTime` will show the passed time in milliseconds. You need to specify `--batch-mode` to make this
work.
  MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.
Slf4jMavenTransferListener=WARN -Dorg.slf4j.simpleLogger.showDateTime=true -Djava.awt.headless=true"
  # As of Maven 3.3.0 instead of this you may define these options in `.mvn/maven.config` so the same config is
used
  # when running from the command line.
  # `installAtEnd` and `deployAtEnd`are only effective with recent version of the corresponding plugins.
  MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end --show-version -DinstallAtEnd=true -DdeployAtEnd=true"


# Cache downloaded dependencies and plugins between builds.
# To keep cache across branches add 'key: "$CI_JOB_REF_NAME"'
cache:
  paths:
    - .m2/repository


maven_build:
  script:
    - |
        echo "building my amazing repo..."
        mvn test
        export token=$(curl -H "Content-Type: application/json" -X POST --data "{ \"client_id\": \"$client_id\",
\"client_secret\": \"$client_secret\" }" https://xray.cloud.getxray.app/api/v2/authenticate| tr -d '"')
        echo $token
        curl -H "Content-Type: text/xml" -H "Authorization: Bearer $token" --data @target/surefire-reports/TEST-
com.xpand.java.CalcTest.xml  "https://xray.cloud.getxray.app/api/v2/import/execution/junit?projectKey=CALC"
        echo "done"
```

In order to submit those results, we'll just need to invoke the REST API (as detailed in Import Execution Results - REST).

However, we do not want to have the Xray API credentials hardcoded in GitLab's configuration file. Therefore, we'll use some environment variables defined in project settings, including:

- **client_id**: the client_id associated with the API key created in the Xray cloud instance
- **client_secret**: the client_secret associated with the API key created in the Xray cloud instance

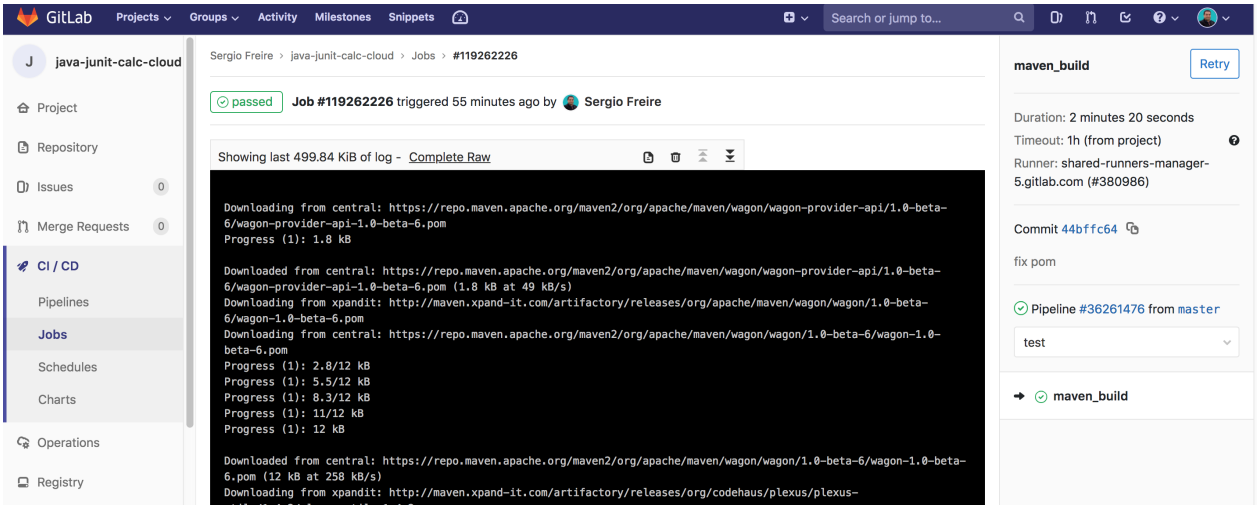
> **Please note**
>
> The user associated with Xray's API key must have permission to Create Test and Test Execution Issues.



In `.gitlab-ci.yml` a "step" must be included that will use "curl" in order to first obtain a token and then finally submit the results to the REST API, using that token.

```
export token=$(curl -H "Content-Type: application/json" -X POST --data "{ \"client_id\": \"$client_id\",\"
client_secret\": \"$client_secret\" }" https://xray.cloud.getxray.app/api/v2/authenticate| tr -d '"')

curl -H "Content-Type: text/xml" -H "Authorization: Bearer $token" --data @target/surefire-reports/TEST-com.xpand.
java.CalcTest.xml  "https://xray.cloud.getxray.app/api/v2/import/execution/junit?projectKey=SP"
```

We're using "curl" utility that comes in Unix based OS'es but you can easily use another tool to make the HTTP request; however, "curl" is provided in the container used by GitLab.



> **Triggering automation from Xray**
>
> If you aim to trigger automation from Xray/Jira side, please have a look at Taking advantage of Jira Cloud built-in automation capabilities page where you can see an example of triggering a GitLab pipeline from a Test Plan and reporting results back to it.

Robot Framework example

In this scenario, we want to get visibility of the automated test results from some UI tests implemented in Robot Framework (Python) together with Selenium (using the "robotframework-seleniumlibrary"), and using Chrome for testing.

We need to set up a Git repository containing the code along with the configuration for GitLab build process.

The tests are implemented in Robot Framework .robot files as follows.

valid_login.robot

```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.robot

*** Test Cases ***
Valid Login
    [Tags]  UI
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser
```

The GitLab configuration file `.gitlab-ci.yml` contains the definition of the build steps, including running the automated tests and submitting the results, as two different stages.

.gitlab-ci.yml

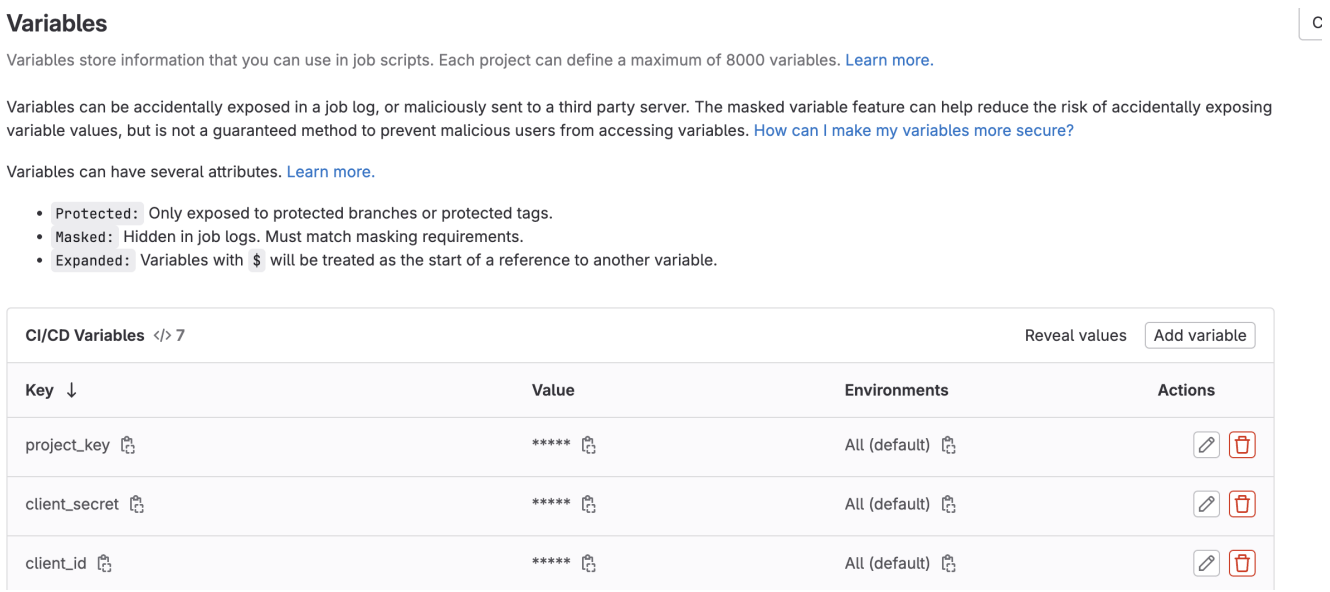
```
# Official language image. Look for the different tagged releases at:
# https://hub.docker.com/r/library/python/tags/
image: python:3.12.2

# Change pip's cache directory to be inside the project directory since we can
# only cache local items.
variables:
  PIP_CACHE_DIR: "$CI_PROJECT_DIR/.cache/pip"

# https://pip.pypa.io/en/stable/topics/caching/
cache:
  paths:
    - .cache/pip

stages:
  - execute_automated_tests
  - upload_test_results

before_script:
  - python --version ; pip --version  # For debugging
  - pip install virtualenv
  - virtualenv venv
  - source venv/bin/activate
  - pip install -r requirements.txt
  - apt-get update


test:
  stage: execute_automated_tests
  before_script: |
    set -e
    apt-get install -yqq unzip curl
    # Install Chrome & chromedriver
    curl -sS -o - https://dl.google.com/linux/linux_signing_key.pub | apt-key add -
    echo "deb https://dl.google.com/linux/chrome/deb/ stable main" >> /etc/apt/sources.list.d/google.list
    apt update && apt install google-chrome-stable -y
    wget -O /tmp/chromedriver.zip https://storage.googleapis.com/chrome-for-testing-public/121.0.6167.85/linux64
/chromedriver-linux64.zip
    ls -la /tmp/chromedriver.zip
    unzip -j /tmp/chromedriver.zip chromedriver-linux64/chromedriver -d /usr/local/bin/
    nohup python demoapp/server.py &
  script: |
    chromedriver -v && \
    pip install -r requirements.txt && \
    robot -x junit.xml -o output.xml login_tests || true
  allow_failure: true
  artifacts:
    paths:
      - output.xml
    when: always


upload_results_to_xray:
  stage: upload_test_results
  script:
    - |
      echo "uploading results to Xray..."
      export token=$(curl -H "Content-Type: application/json" -X POST --data "{ \"client_id\": \"$client_id\",\"
client_secret\": \"$client_secret\" }" https://xray.cloud.getxray.app/api/v2/authenticate| tr -d '"')
      curl -H "Content-Type: text/xml" -H "Authorization: Bearer $token" --data @"output.xml" "https://xray.
cloud.getxray.app/api/v2/import/execution/robot?projectKey=$project_key"
  dependencies:
  - test
```

In order to submit those results, we'll just need to invoke the REST API (as detailed in Import Execution Results - REST).

However, we do not want to have the Xray API credentials hardcoded in the GitLab's configuration file. Therefore, we'll use environment variables defined in the project settings, including:

- **client_id**: the client_id associated with the API key created in the Xray cloud instance
- **client_secret**: the client_secret associated with the API key created in the Xray cloud instance
- **project_key**: the Jira project key

> **Please note**
>
> The user associated with the Xray's API key must have permissions to Create Test and Test Execution Issues.

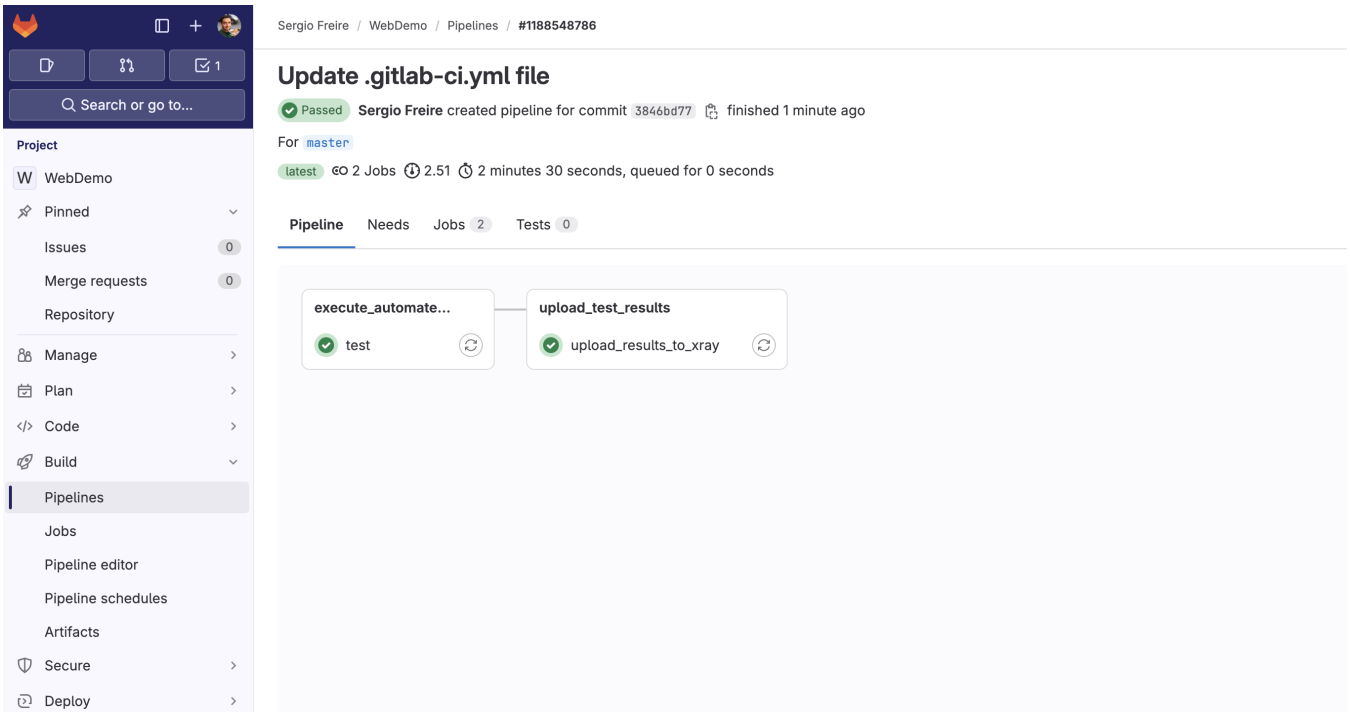


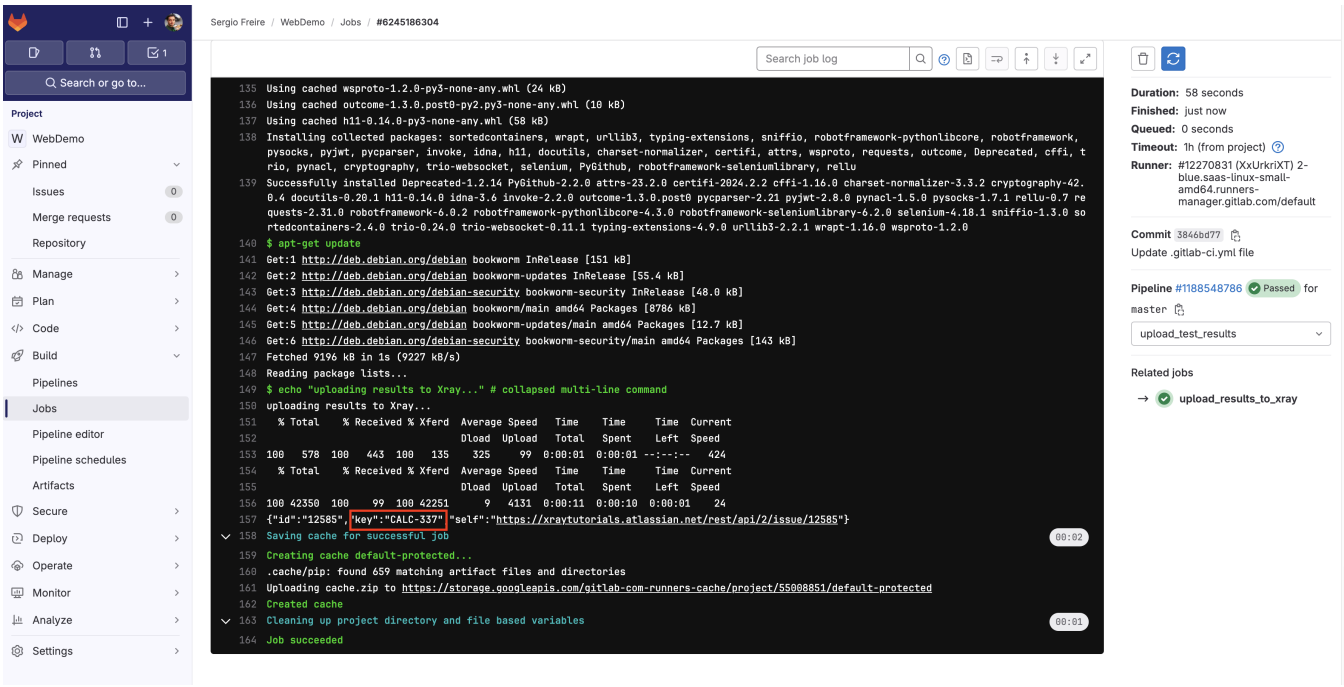
In `.gitlab-ci.yml` a "step" must be included that will use "curl" in order to first obtain a token and then finally submit the results to the REST API, using that token.

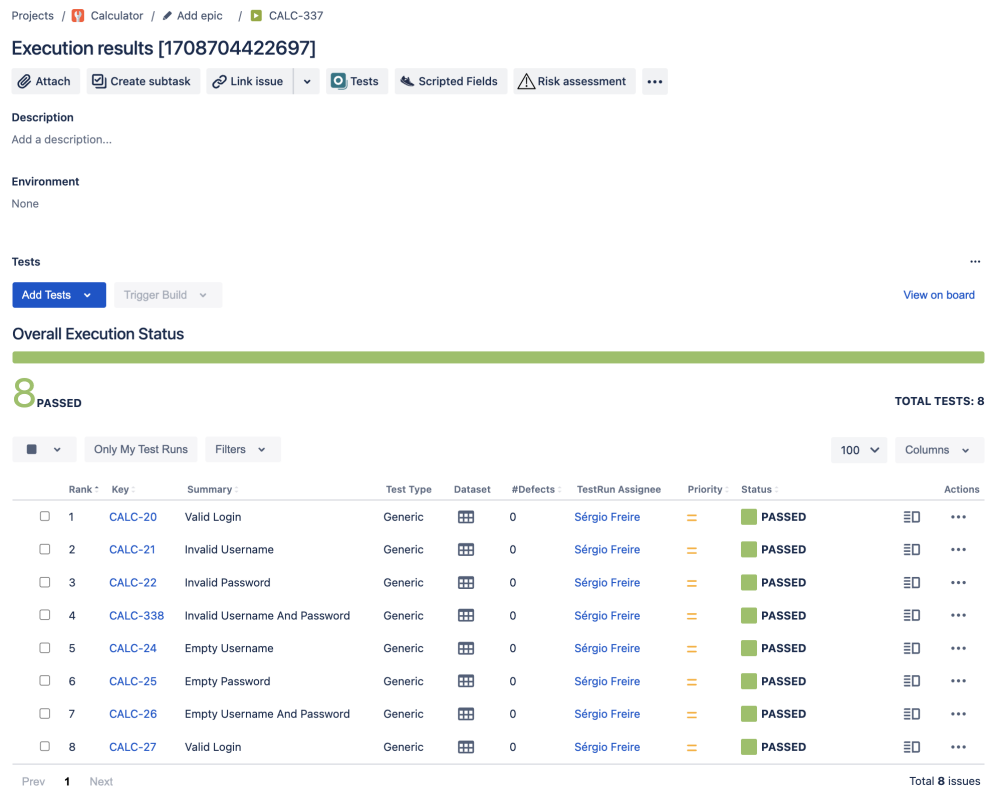
```
export token=$(curl -H "Content-Type: application/json" -X POST --data "{ \"client_id\": \"$client_id\",\"
client_secret\": \"$client_secret\" }" https://xray.cloud.getxray.app/api/v2/authenticate| tr -d '"')
```

```
curl -H "Content-Type: text/xml" -H "Authorization: Bearer $token" --data @"output.xml" "https://xray.cloud.
getxray.app/api/v2/import/execution/robot?projectKey=$project_key"
```

We're using "curl" utility that comes in Unix based OS'es but you can easily use another tool to make the HTTP request; however, "curl" is provided in the container used by GitLab.


Sergio Freire / WebDemo / Pipelines / #1188548786
Search or go to...
Project
WebDemo
Pinned
Issues 0
Merge requests 0
Repository
Manage
Plan
Code
Build
Pipelines
Jobs
Pipeline editor
Pipeline schedules
Artifacts
Secure
Deploy
Update .gitlab-ci.yml file
Passed Sergio Freire created pipeline for commit 3846bd77 finished 1 minute ago
For master
latest 2 Jobs 2.51 2 minutes 30 seconds, queued for 0 seconds
Pipeline Needs Jobs 2 Tests 0
execute_automate...
test
upload_test_results
upload_results_to_xray





> **Triggering automation from Xray**
>
> If you aim to trigger automation from the Xray/Jira side, please have a look at Taking advantage of Jira Cloud built-in automation capabilities page where you can see an example of triggering a GitLab pipeline from a Test Plan and reporting results back to it.

Cucumber example

Standard workflow (Xray as master)

In this scenario, we are managing the specification of Cucumber Scenarios/Scenario Outline(s) based tests in Jira, using Xray, as detailed in the "standard workflow" mentioned in Testing in BDD with Gherkin based frameworks (e.g. Cucumber).

Then we need to extract this specification from Jira (i.e. generate related Cucumber .feature files), and run it in GitLab against the code that actually implements each step that are part of those scenarios.

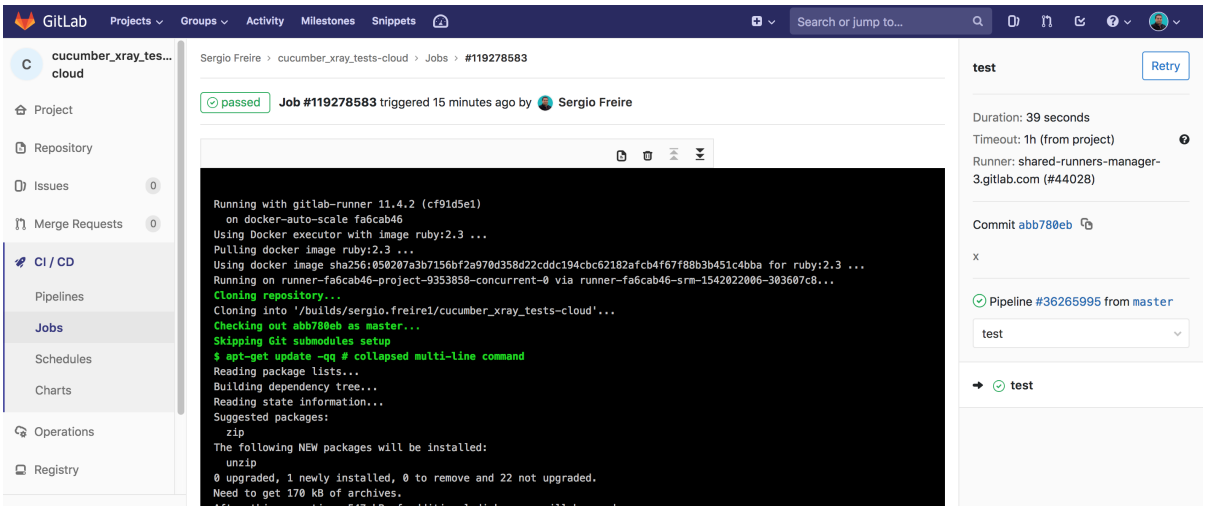
Finally, we can then submit the results back to JIRA and they'll be reflected on the related entities.

The GitLab configuration file `.gitlab-ci.yml` contains the definition of the build steps, including extracting the cucumber specification from Xray, running the automated tests and submitting back the results.

.gitlab-ci.yml

```
image: "ruby:2.6"

test:
  script:
    - |
        apt-get update -qq
        apt-get install unzip
        gem install cucumber
        gem install rspec-expectations
        export token=$(curl -H "Content-Type: application/json" -X POST --data "{ \"client_id\": \"$client_id\",
\"client_secret\": \"$client_secret\" }" https://xray.cloud.getxray.app/api/v2/authenticate| tr -d '"')
        curl -H "Content-Type: application/json" --output features/features.zip -X GET -H "Authorization:
Bearer ${token}"  "https://xray.cloud.getxray.app/api/v2/export/cucumber?keys=$cucumber_keys"
        mkdir -p features
        rm -f features/*.feature
        unzip -o features/features.zip -d features/
        cucumber -x -f json -o data.json
        curl -H "Content-Type: application/json" -X POST -H "Authorization: Bearer ${token}" --data @data.json
https://xray.cloud.getxray.app/api/v2/import/execution/cucumber
        echo "done"
```

In this example, we're using a variable **cucumber_keys** defined in the CI/CD project-level settings in GitLab. This variable contains one or more keys of the issues that will be used as source data for generating the Cucumber .feature files; it can be the key(s) of Test Plan(s), Test Execution(s), Test(s), requirement(s). For more info, please see: Exporting Cucumber Tests - REST.



VCS workflow (Git as master)

In this scenario, we are managing (i.e. editing) the specification of Cucumber Scenarios/Scenario Outline(s) based tests outside Jira, as detailed in the "VCS workflow" mentioned in Testing in BDD with Gherkin based frameworks (e.g. Cucumber).

The GitLab configuration file `.gitlab-ci.yml` contains the definition of the build steps, including synchronizing the Scenarios/Backgrounds to Xray, extracting the cucumber specification from Xray, running the automated tests and submitting back the results.
.gitlab-ci.yml

```
image: "ruby:2.6"

test:
  script:

    - |
        apt-get update -qq
        apt-get -y install zip unzip
        gem install cucumber
        gem install rspec-expectations
        export token=$(curl -H "Content-Type: application/json" -X POST --data "{ \"client_id\": \"$client_id\",
\"client_secret\": \"$client_secret\" }" https://xray.cloud.getxray.app/api/v2/authenticate| tr -d '"')

        cd features; zip -R features.zip "*.feature"; cd ..; curl -H "Content-Type: multipart/form-data" -H
"Authorization: Bearer ${token}" -F "file=@features/features.zip"

"https://xray.cloud.getxray.app/api/v2/import/feature?projectKey=CALC"
        mkdir -p features

        rm -f features/*.feature

        curl -H "Content-Type: application/json" --output features/features.zip -X GET -H "Authorization:
Bearer ${token}"  "https://xray.cloud.getxray.app/api/v2/export/cucumber?filter=$filter_id"
        unzip -o features/features.zip -d features/
        cucumber -x -f json -o data.json || true
        curl -H "Content-Type: application/json" -X POST -H "Authorization: Bearer ${token}" --data @data.json
https://xray.cloud.getxray.app/api/v2/import/execution/cucumber
        echo "done"
```

In this example, we're using a variable **filter_id** defined in the CI/CD project level settings in GitLab. This variable contains the *id* of the Jira issues based filter that will be used as source data for generating the Cucumber .feature files; it can be the key(s) of Test Plan(s), Test Execution(s), Test(s), requirement(s). For more info, please see: Exporting Cucumber Tests - REST.

Triggering automation from Xray side

Please have a look at Integration with Automation for Jira to see some examples of how automation can be triggered from Xray side.