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.

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

- JUnit example
- Cucumber example
    - Standard workflow (Xray as master)
    - VCS workflow (Git as master)

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.

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

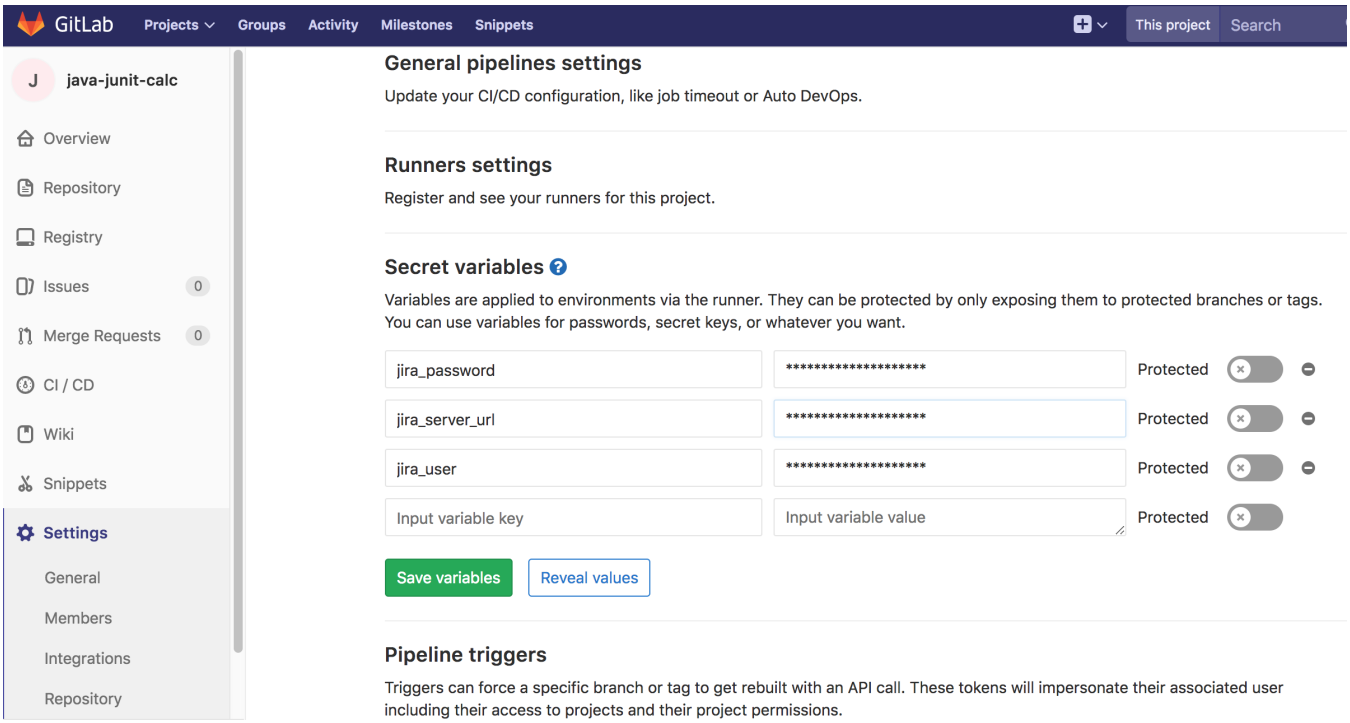
```
# Use Maven 3.5 and JDK8
image: maven:3.5-jdk-8

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
    - 'curl -H "Content-Type: multipart/form-data" -u $jira_user:$jira_password -F "file=@target/surefire-
reports/TEST-com.xpand.java.CalcTest.xml" "$jira_server_url/rest/raven/1.0/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 JIRA credentials hardcoded in GitLab's configuration file. Therefore, we'll use some secret variables defined in GitLab project settings.

Please note

The user present in the configuration below must exist in the JIRA instance and have permission to Create Test and Test Execution Issues



In `.gitlab-ci.yml` a "step" must be included in the maven_build section, that will use "curl" in order to submit the results to the REST API.

```
curl -H "Content-Type: multipart/form-data" -u $jira_user:$jira_password -F "file=@target/surefire-reports/TEST-com.xpand.java.CalcTest.xml" "$jira_server_url/rest/raven/1.0/import/execution/junit?projectKey=CALC"
```

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.

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:2.3"

test:
  script:
  - apt-get update -qq
  - apt-get install unzip
  - gem install cucumber
  - gem install rspec-expectations
  - 'curl -u $jira_user:$jira_password "$jira_server_url/rest/raven/1.0/export/test?keys=$cucumber_keys" -o
features/features.zip'
  - 'rm -f features/*.feature'
  - unzip -o features/features.zip -d features/
  - cucumber -x -f json -o data.json
  - 'curl -H "Content-Type: application/json" -u $jira_user:$jira_password --data @data.json "$jira_server_url
/rest/raven/1.0/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:2.3"

test:
  script:
  - apt-get update -qq
  - apt-get -y install unzip zip
  - gem install cucumber
  - gem install rspec-expectations
  - 'cd features; zip -R features.zip "*.feature"; cd ..; curl -H "Content-Type: multipart/form-data" -u
$jira_user:$jira_password -F "file=@features/features.zip" "$jira_server_url/rest/raven/1.0/import/feature?
projectKey=CALC" '
  - 'rm -f features/*.feature'
  - 'curl -u $jira_user:$jira_password "$jira_server_url/rest/raven/1.0/export/test?filter=$filter_id" -o
features/features.zip'
  - unzip -o features/features.zip -d features/
  - cucumber -x -f json -o data.json || true
  - 'curl -H "Content-Type: application/json" -u $jira_user:$jira_password --data @data.json "$jira_server_url
/rest/raven/1.0/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 filterhat 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.