Integration with TeamCity (legacy)

These intructions are deprecated.

Please see Integration with TeamCity for the new instructions using the new and specific plugin for TeamCity! Enjoy 🙂

~~Xray does not provide yet a plugin for TeamCity.~~ However, it is easy to setup TeamCity 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

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.

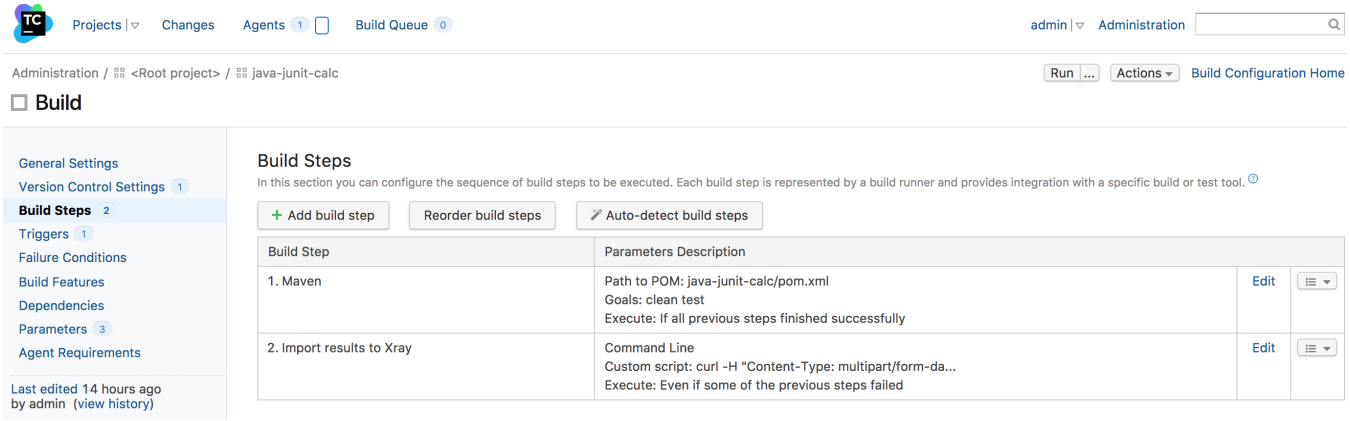
CalcTest.java

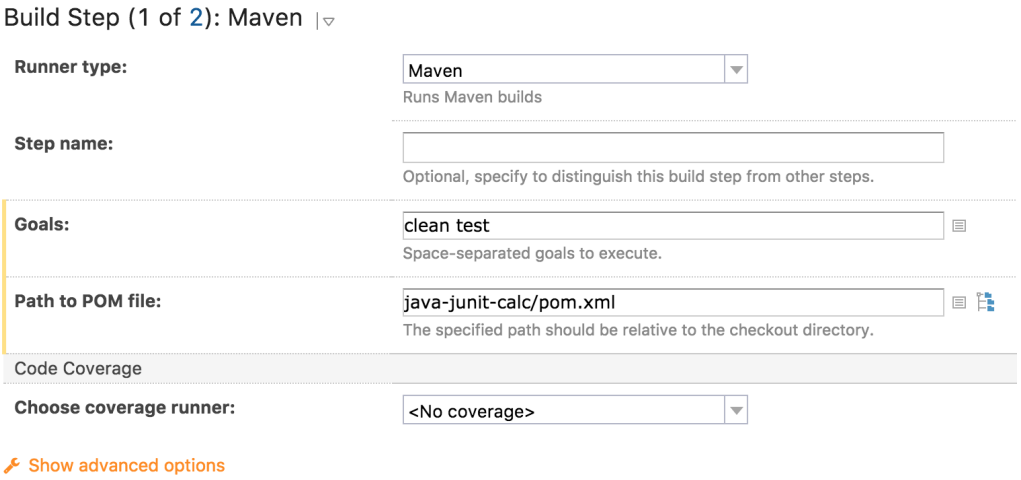
```
package com.xpand.java;

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));
    }


}
```

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

Run automated tests

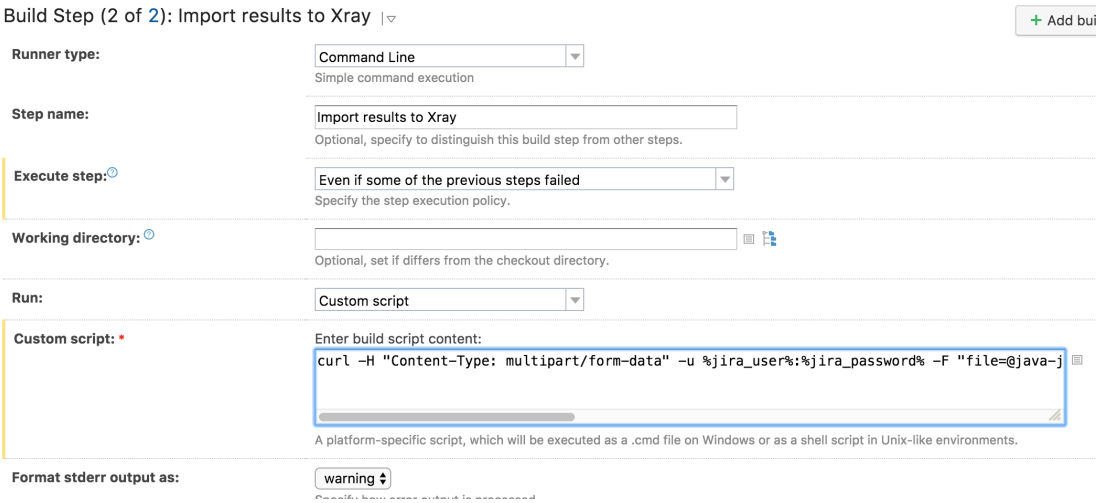
Our project is Maven based, therefore the first Build Step compiles and runs the JUnit automated tests.





Import execution results

In order to submit the results, we'll need to add a Build Step of type "Command Line", where we'll invoke the REST API, submitting the JUnit XML report generated in the previous step.



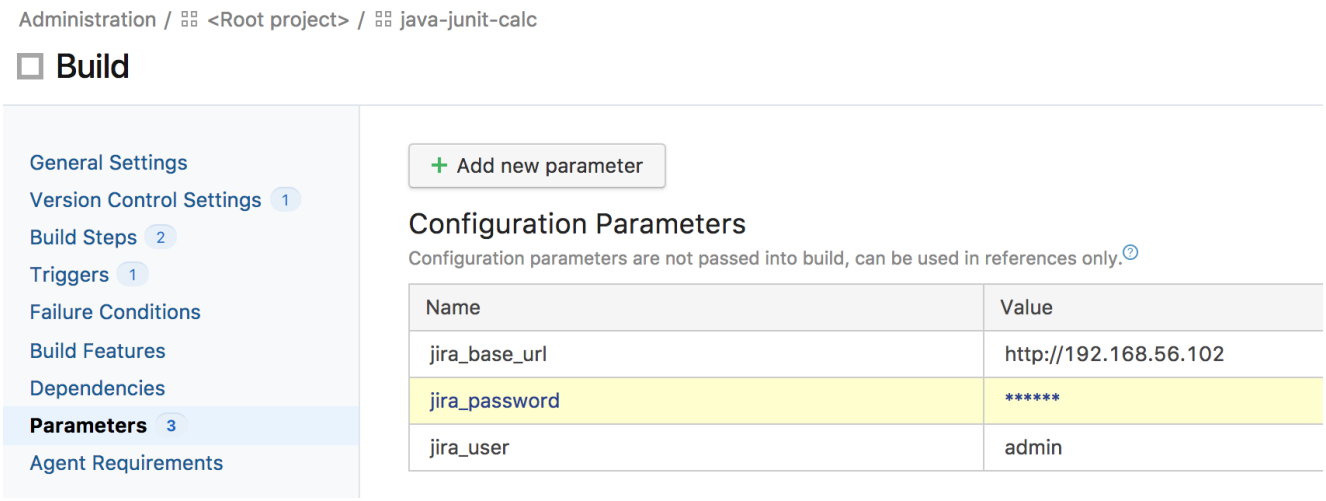
The complete script content of the "custom script" field above is:

```
curl -H "Content-Type: multipart/form-data" -u %jira_user%:%jira_password% -F "file=@java-junit-calc/target
/surefire-reports/TEST-com.xpand.java.CalcTest.xml" "%jira_base_url%/rest/raven/1.0/import/execution/junit?
projectKey=CALC&fixVersion=v3.0&revision=1234"
```

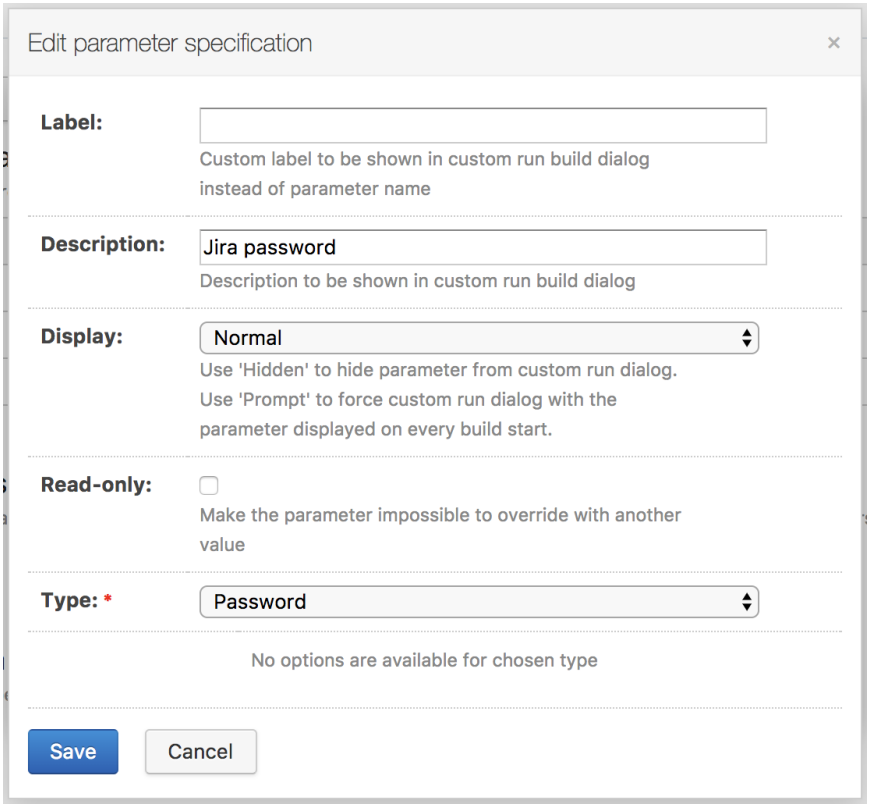
We're using "curl" utility that comes in Unix based OS'es but you can easily use another tool to make the HTTP request.

Notice that we're using some parameters for storing Jira's base URL along with the credentials to be used in the REST API.

Actually, these parameters can be defined at multiple levels; in our example we defined them at the "Build Configuration" level but they could also have been defined at the project level.



The parameters can be hidden, such as the password, if you defined them as being of type "Password".



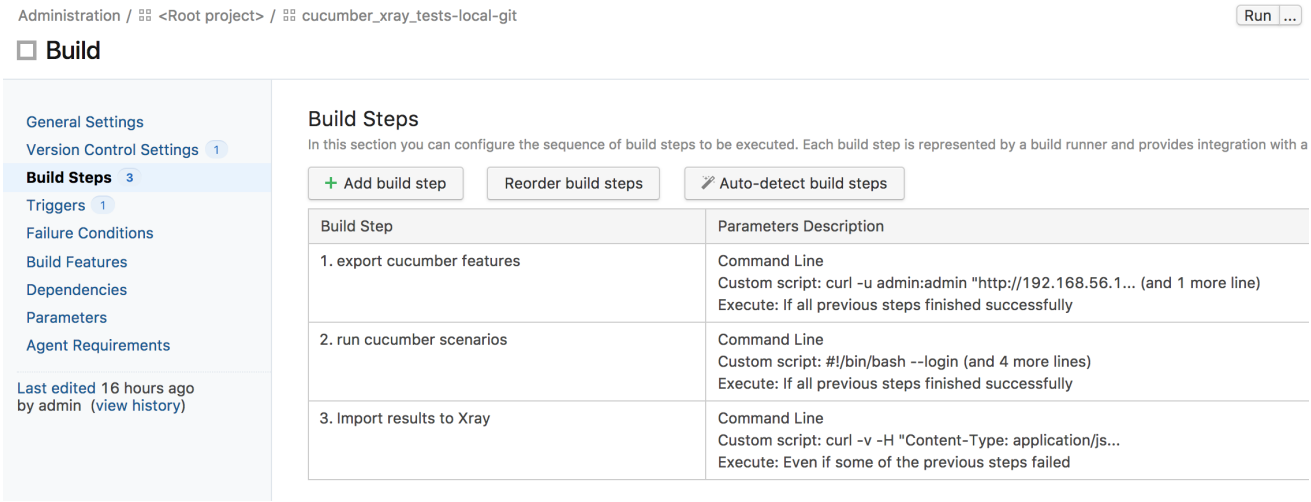
Cucumber example

In this scenario, we are managing the specification of Cucumber Scenarios/Scenario Outline(s) based tests in Jira, as detailed in the "standard workflow" mentioned in Testing with Cucumber

Then we need to extract this specification from Jira (i.e. generate related Cucumber .feature files), and run it in TeamCity 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.

Overall, our Build Configuration is composed of 3 basic steps.



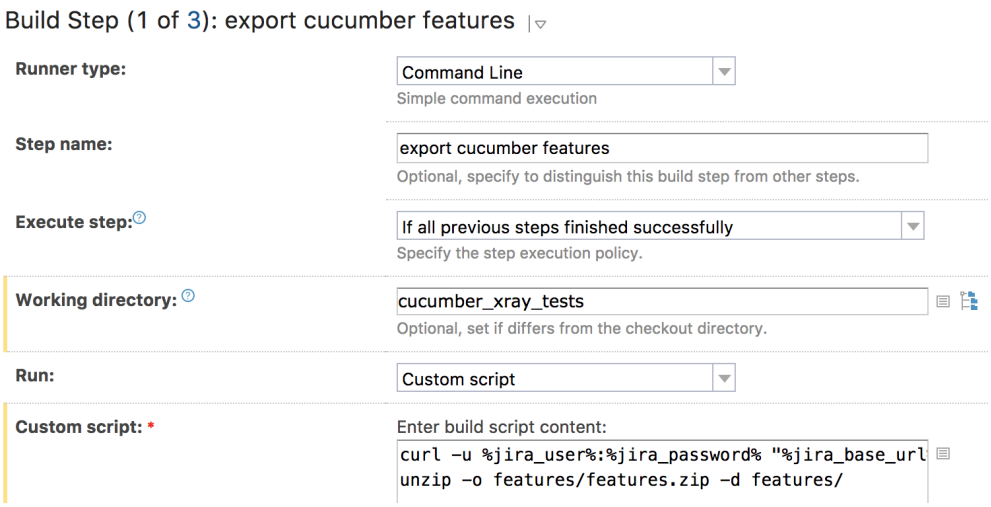
Exporting Cucumber features

We start by extracting the tests specification out of JIRA and generate the proper .feature files.

The export can take as input issue keys of requirements, Test Executions, Test Plans or a filter id, which will be the one we'll use.

For this, we'll invoke the REST API (Exporting Cucumber Tests - REST) in order to obtain a .zip file containing the .feature files.

We'll be using a Build Step of type "Command Line" for this purpose, along with "curl" utility to ease making the HTTP request.



The complete script content of the "custom script" field above is:

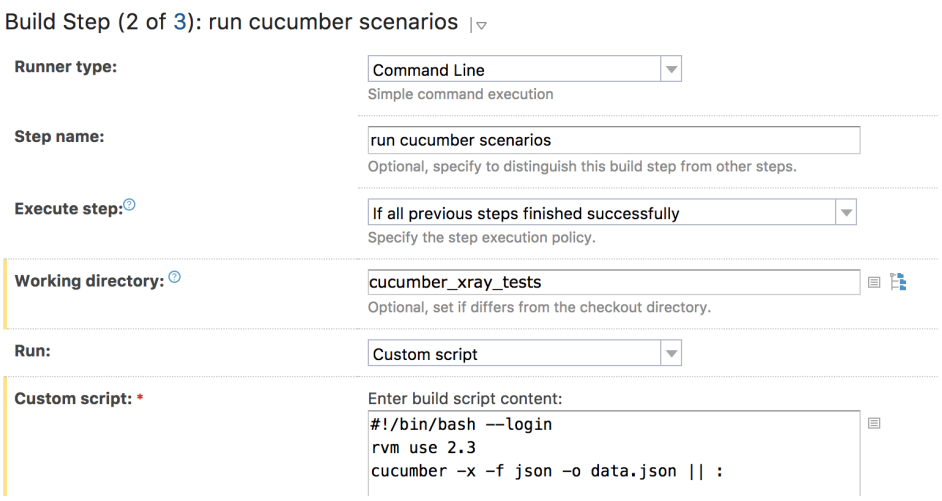
```
curl -u %jira_user%:%jira_password% "%jira_base_url%/rest/raven/1.0/export/test?filter=11400&fz=true" -o
features/features.zip
unzip -o features/features.zip -d features/
```

Notice that we're unzipping the .feature files to a local directory, so we're able to run them.

Run Cucumber scenarios

The exact syntax for running the Cucumber scenarios depends on the Cucumber implementation being used; in this case we're using Ruby's variant.

Therefore we're basically just invoking "cucumber" command with an option to generate a JSON report (e.g. "data.json").

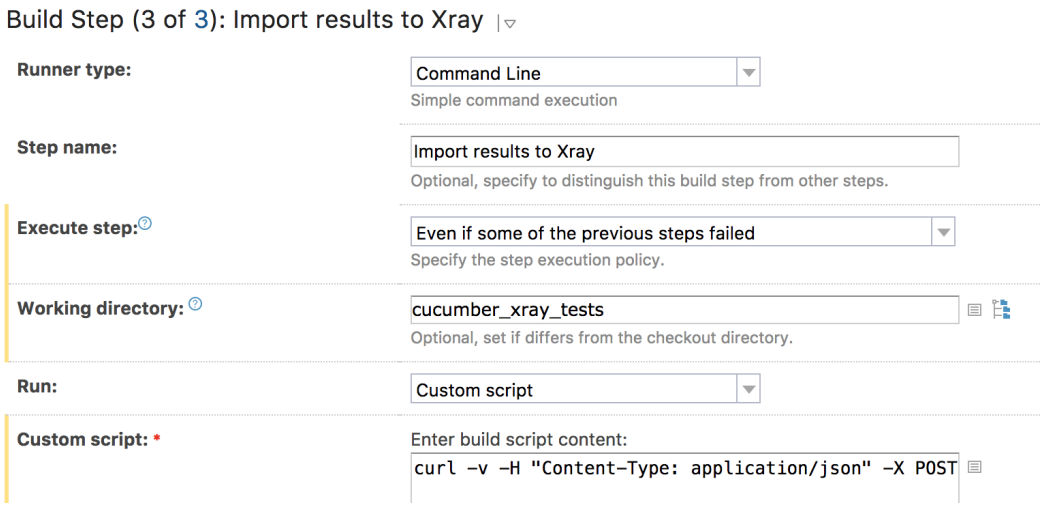


You may have noticed a trick in the cucumber line above, in the end of the command (i.e. ".... || :"). That ensures that cucumber returns with exit code 0 (i.e. success), so the build may proceed.

Import execution results

In order to submit the results, we'll need to add a Build Step of type "Command Line", where we'll invoke the REST API, submitting the Cucumber JSON report generated in the previous step.

We also make sure this step is called always.



The complete script content of the "custom script" field above is:

```
curl -v -H "Content-Type: application/json" -X POST -u %jira_user%:%jira_password% --data @data.json "%
jira_base_url%/rest/raven/1.0/import/execution/cucumber"
```

You may notice that we're using some parameters related with the Jira server, that we've configured at project level.

> **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