Integration with GitHub

blocked URLblocked URL

GitHub is a well-known platform hosting thousands of source-code repositories.

It also provides issue tracking and basic project management capabilities.

More recently, GitHub provided the ability to automate workflows using GitHub Actions.

With GitHub Actions, it's possible to implement CI/CD directly in GitHub and reuse already available actions from GitHub Marketplace to automate steps.

An introduction to GitHub actions can be seen here.

Main concepts

In a nutshell, **workflows** are automated processes described as YAML files, stored under *.github/workflows.* These are usually triggered by events (e.g. code-commit, pull-request) or can also be scheduled.

One or more workflows can be defined. Each **workflow** is in turn composed by one or more **jobs**, that can run sequentially or in parallel. A **job** performs a set of sequential **steps** to achieve a certain goal. A **step** is an individual automation task; it can be either an **action** or simply a shell command.

An **action** abstracts some automation task; it can be named and versioned. Actions can be implemented directly in Javascript or as Docker containers. GitHub also supports composite actions built of multiple inner steps.

Actions and workflows can be stored in the local repository; actions can also be published in the GitHub Marketplace.

Each time a workflow is triggered, a **workflow run** is created; it contains a specific context. Each job in the workflow uses a fresh virtual environment (e.g. ubuntu-latest) sharing the same virtual file system.

Accessing and sharing data

A job can generate output variables that can be used by another job that depends on it; this is the preferred way to share data between jobs.

Another way of sharing data, especially between jobs, would be to produce and store artifact(s) in a job and obtain them in another job.

Environment variables can also be used to access some data and share them with care. Environment variables are available at workflow, job or step level. GitHub fills out some environment variables by default.

It's also possible to access secrets defined in GitHub project settings, as environment variables or as a step input.

Examples

Basic JUnit example

In this basic example showcasing a dummy calculator, we want to get visibility of the automated test results from some tests implemented in Java, using the JUnit framework.

> **Please note**
>
> The source code for this example is available in this GitHub repository.

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));
    }


}
```

To implement the continuous integration, we'll implement a specific *workflow* for it and store it `.github/workflows/CI-jira-onpremises-example.yaml`.

We'll use the actions/checkout action to checkout the code from our repository to the virtual environment. This action is one of the "standard" actions provided by GitHub (check full list here).

To compile the code, we need to use a JDK; we can use the action actions/setup-java which allows us to choose the specific Java version.

We use Maven to build and run the tests.

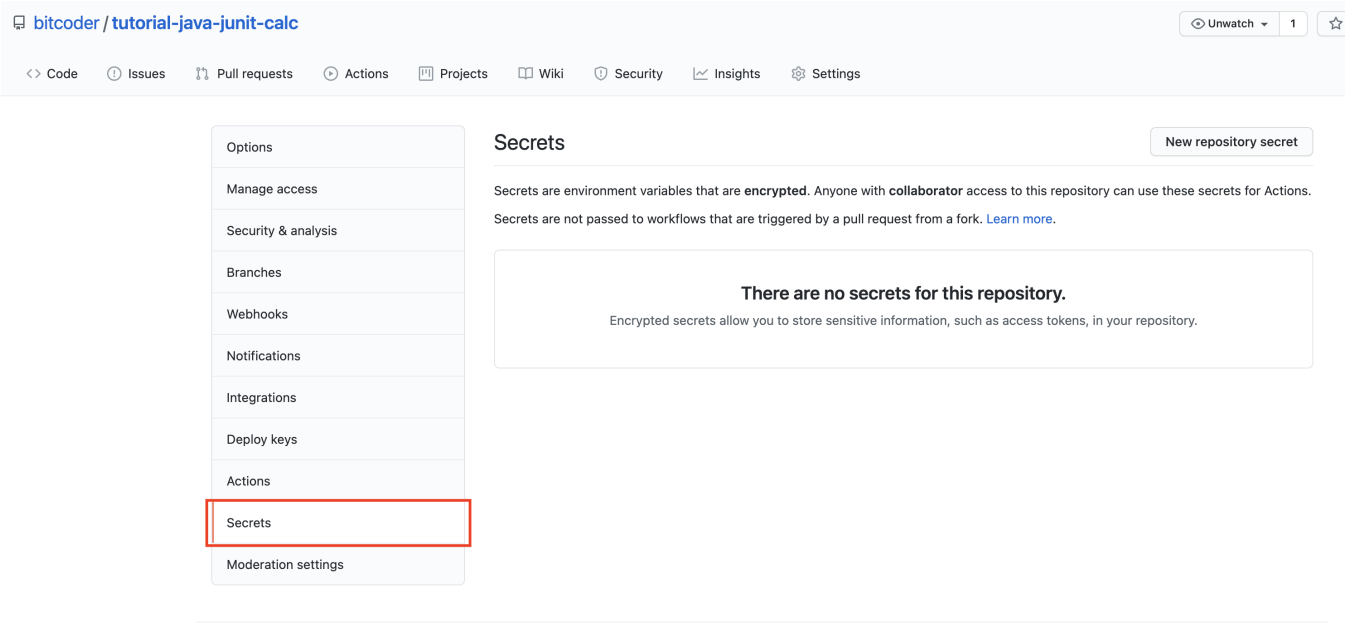
.github/workflows/CI-jira-onpremises-example.yaml

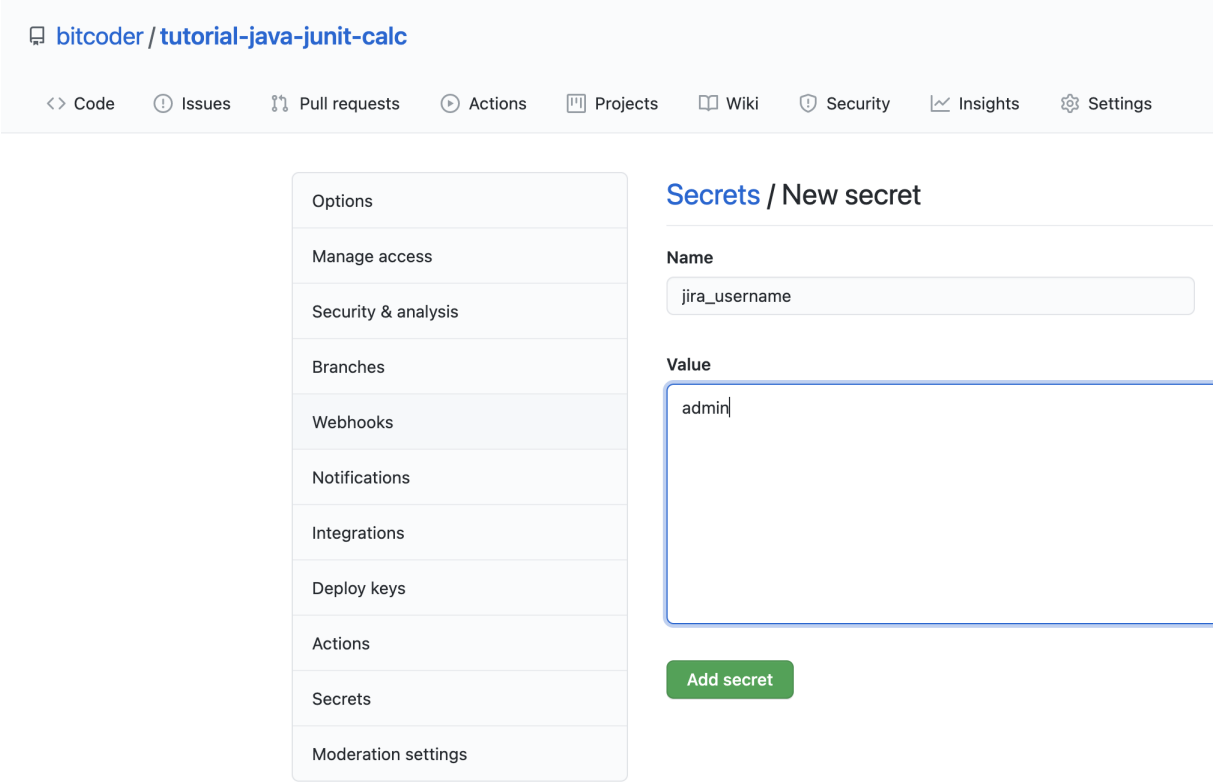
```
    name: CI (Jira on-premises example)
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v1
    - name: Set up Java
      uses: actions/setup-java@v1
      with:
        java-version: '1.8'
    - name: Cache Maven packages
      uses: actions/cache@v2
      with:
        path: ~/.m2
        key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
        restore-keys: ${{ runner.os }}-m2
    - name: Build with Maven
      run: mvn clean compile test --file pom.xml
    - name: Submit results to Xray
      env:
        JIRA_SERVER_URL: ${{ secrets.jira_server_url }}
        JIRA_USERNAME: ${{ secrets.jira_username }}
        JIRA_PASSWORD: ${{ secrets.jira_password }}
      run: 'curl -H "Content-Type: multipart/form-data" -u $JIRA_USERNAME:$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"'
```

In order to submit those results to Xray, 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 the configuration file. Therefore, we'll use some secret variables defined in GitHub project settings.

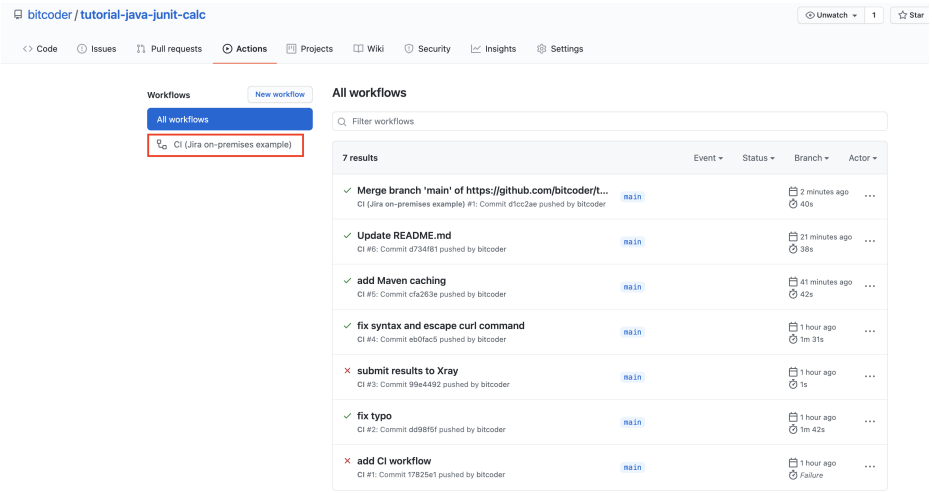


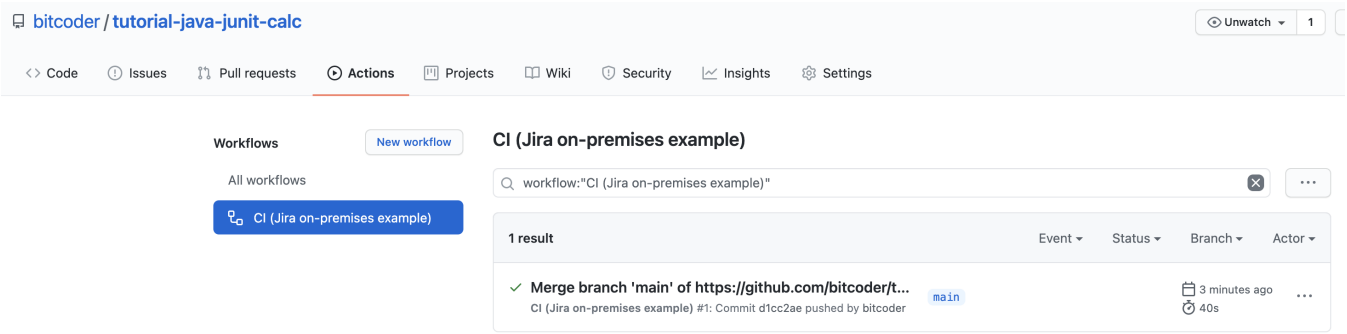


Some parameters may be hardcoded on the HTTP request used to submit the result; this is up to you to define what makes sense to be explicit on the request or what could be set, for example, using a secret variable in GitHub.

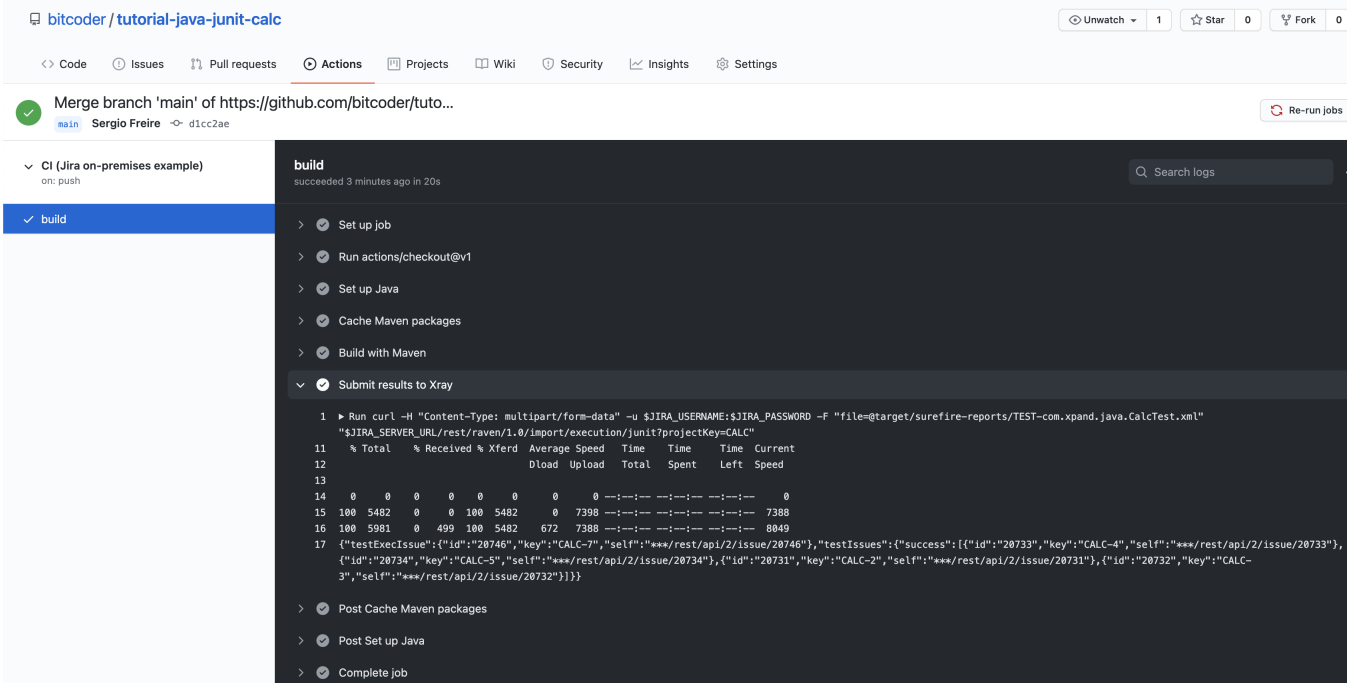
> **Please note**
>
> The Jira username must exist in the Jira instance and have permission to create Test and Test Execution Issues.

To see the runs for your workflows (i.e. workflow runs), you may access the Actions tab in your repository browser.

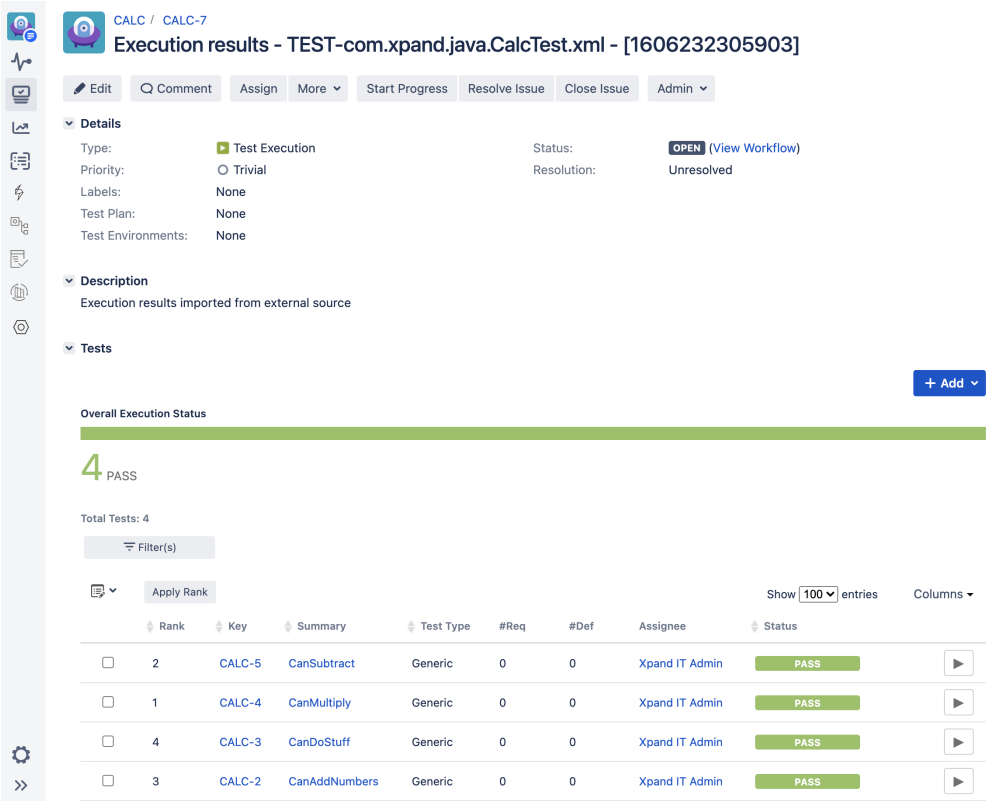




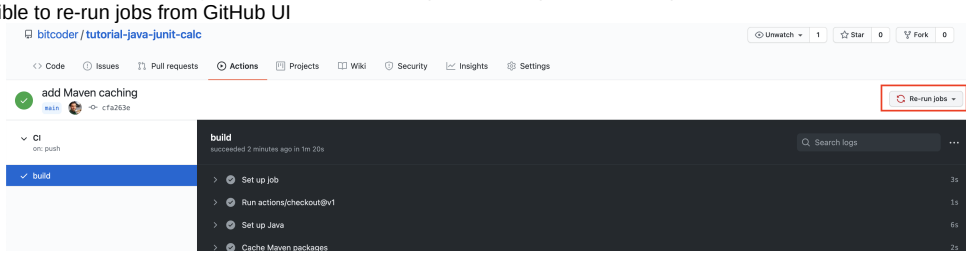
Clicking in the last event that triggered the workflow run will show the details.



In Jira, Xray now shows the results of the automated tests in a brand new Test Execution issue. Test issues corresponding to each test method will be auto-provisioned, if they don't exist yet; otherwise, results will be reported against existing Tests.



Tips

- for editing workflow YAML files, you can do it directly from GitHub UI as it provides syntax highlighting, auto-completion, and more
- in the workflow definition, configure it to cache Maven dependencies (more info here)
- it's possible to re-run jobs from GitHub UI
  - 

- instead of using `curl` command to interact with Xray REST API, you can abstract it in a GitHub Action and use input parameters to be passed to the REST call

References

- Introduction to GitHub Actions
- Building and testing Java with Maven with GitHub Actions