CLOUDBEES UNIVERSITY

Continuous Delivery with the CloudBees Jenkins Platform

User training Labs

Table of Contents

Introduction

This workbook is designed to supplement your CloudBees Jenkins training. It consists of a sequence of lab exercises that will introduce continuous integration concepts and Jenkins best practices.

These lab exercises should be completed in sequence and in the presence of an instructor. The instructor will distribute the software and code necessary to complete these exercises.

The "Install Lab Environment" document will cover the setup of all the required software.

Your instructor will tell you if this has already been done on your machine, or if you need to do so yourself.

Most of the plugins required for this course are bundled with this installation, the Lab steps have additional information on which plugins are used and how to go about installing them if they were needed.

IMPORTANT Never try to upgrade plugins of the Lab's Jenkins instance !

If you encounter problems with your software installation or if you do not understand any of the instructions, please ask your instructor for help.

Lab 1: Creating a Freestyle Build Job

Goal

The aim of this lab exercise is to configure a simple freestyle build job on your Jenkins server.

Pre-Step: Configure Jenkins for Maven

Start by accessing Jenkins configuration:

- Click on the **Manage Jenkins** menu item.
- Then, click on **Configure System**.

We need to configure a Maven installation, that will be used for building the example applications:

- On the **Maven** section, click on the **Maven installations…** button.
- Then, configure the new Maven Installation with this settings:
  - **Name**: `maven-3`

- **Install automatically**: Checked
- Use **Install from Apache**, using the latest 3.3.x available

Figure 1. Maven Configuration

Finally, save this configuration by clicking on the **Save** button at the bottom of the screen.

Step 1. Create a New Build Job

Click on the **New Item** menu on the home page, and choose **Build a free-style software project**.

Name this project `gameoflife-freestyle` and click **OK**:

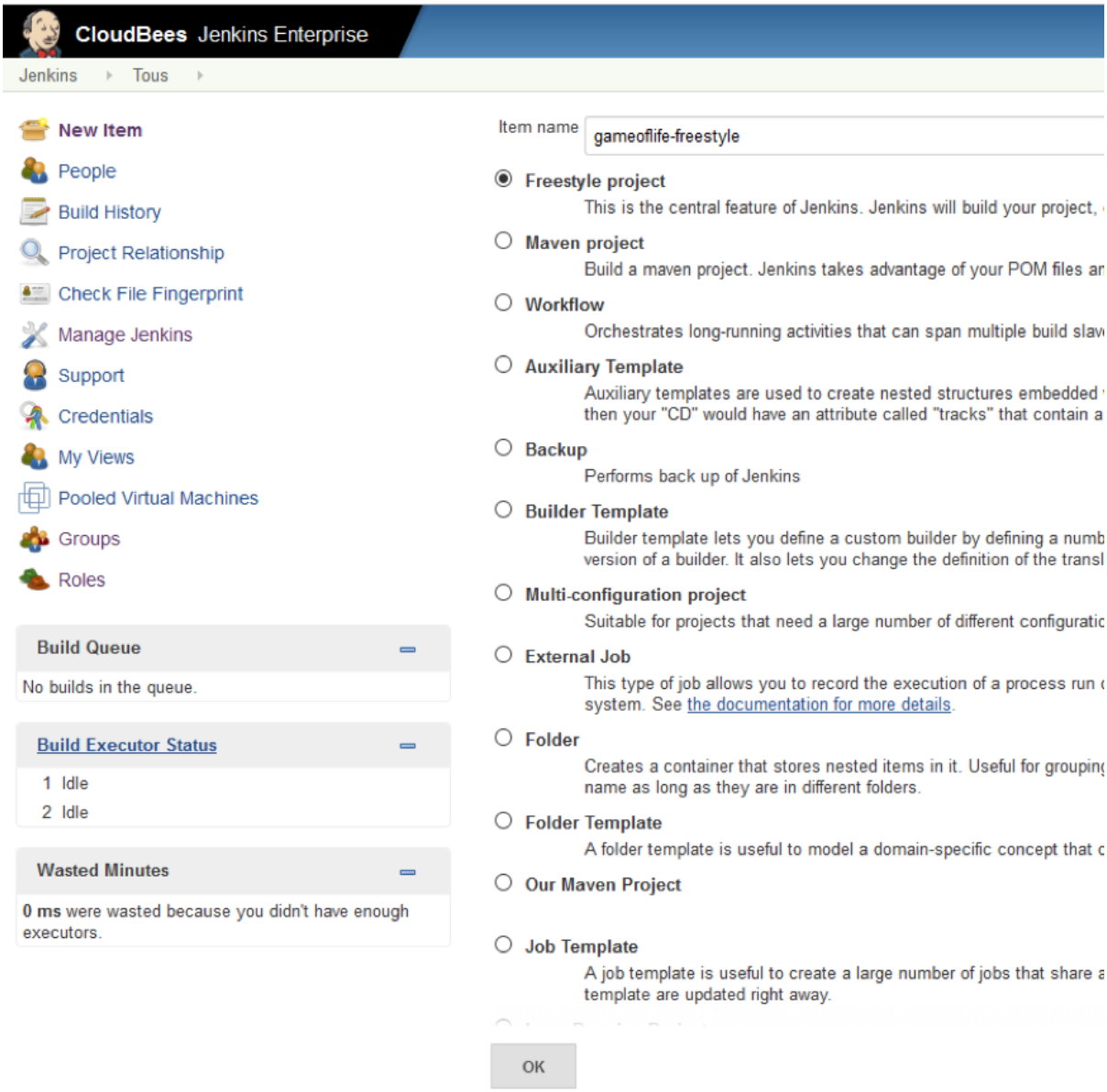


Figure 2. New Job Form of kind Freestyle

Step 2. Configure the SCM Details

For this build job, you need to configure:

- First, choose **Git** in the **Source Code Management** section.
- Then, in the **Repository URL** field, enter your GIT repository address:

```
http://localhost:5000/gitserver/harry/gameoflife.git
```

- Since this repository need authentication to be accessed, you will need to add a new credential with those properties:
  - Kind: **Username with password**

- Scope: **Global** (to allow reusing in other jobs)
- Username is `harry`
- Password is the **same** value as the username

- Then ensure your newly created credential is selected:

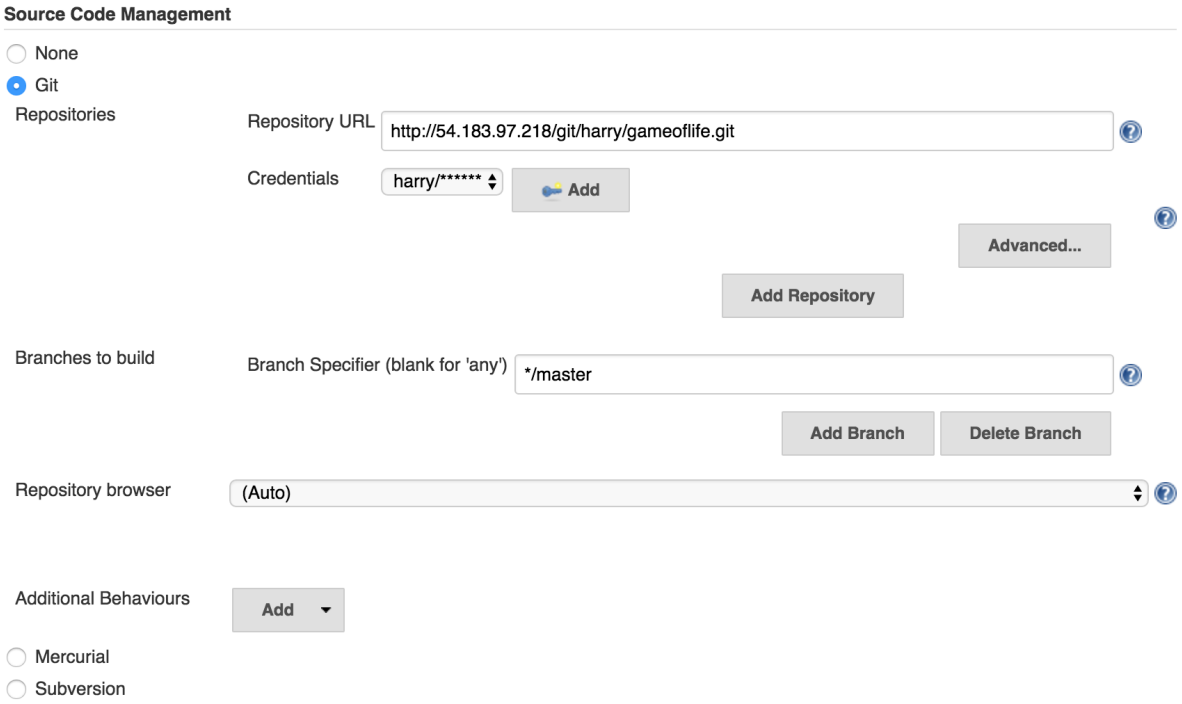


Figure 3. SCM Configuration for Job

Step 3. Configure the Build Triggers

Next, set up the build triggers. We will be polling the Git repository every minute, so in the **Poll SCM** section, enter:

```
* * * * *
```

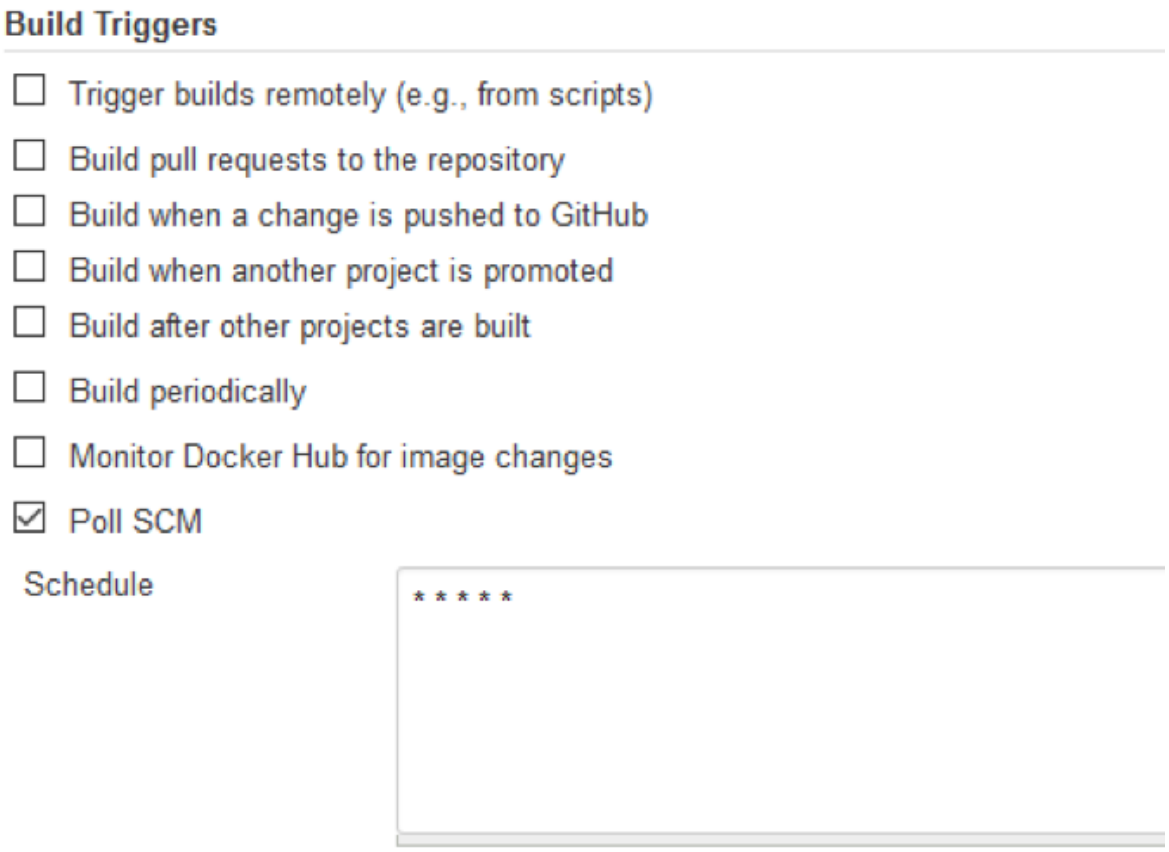



Figure 4. Build Triggers Configuration for Freestyle Job

TIP

Note that a warning (Yellow text) may appears proposing to use:

```
H * * * *
```

instead. Do **not** take it in account for now.

Step 4. Configure the Build Goals

Finally, you need to configure the Maven build goals.

First, add a new **Invoke top-level Maven targets** build step:

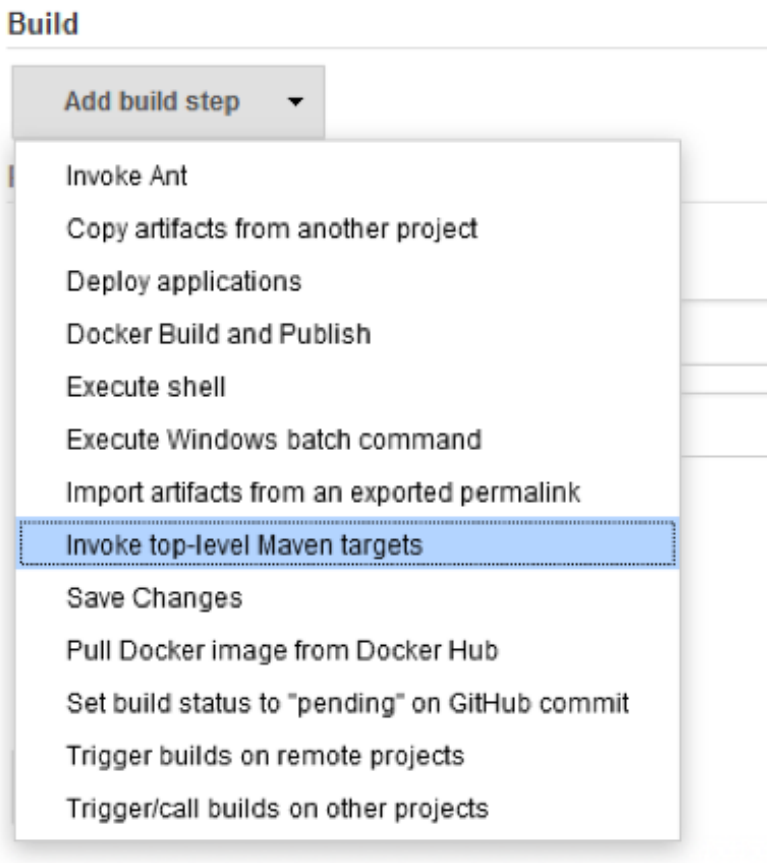


Figure 5. Adding a Maven Build Step

Then:

- Select your **maven-3** installation as **Maven Version**
- Add the following content in the **Goals** field:

```
clean install -B -U
```

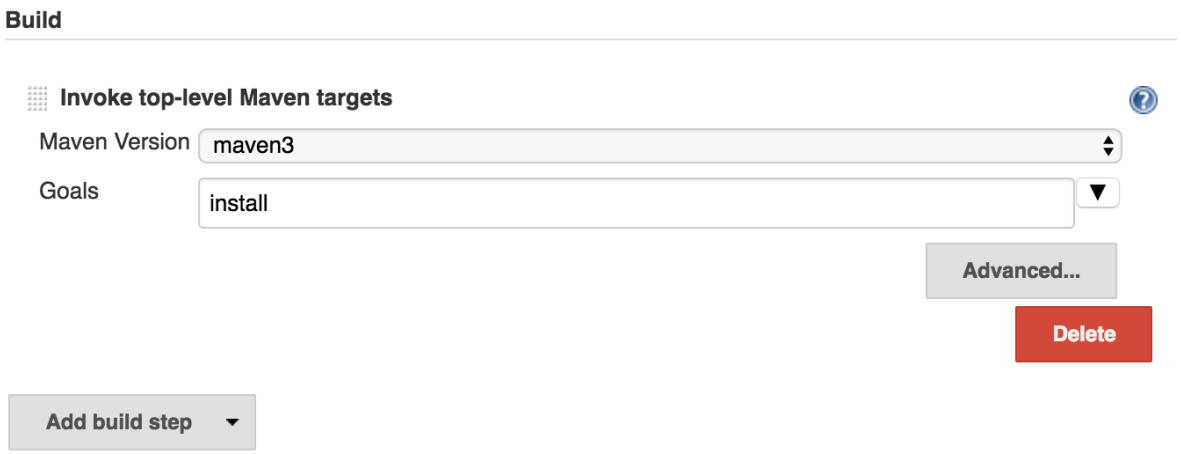


Figure 6. Maven Build Step Configuration

- Next, click on the **Advanced** button to enter some extra property values to pass to the build:


```
surefire.useFile=false
webdriver.port=9091
jetty.stop.port=9998
```

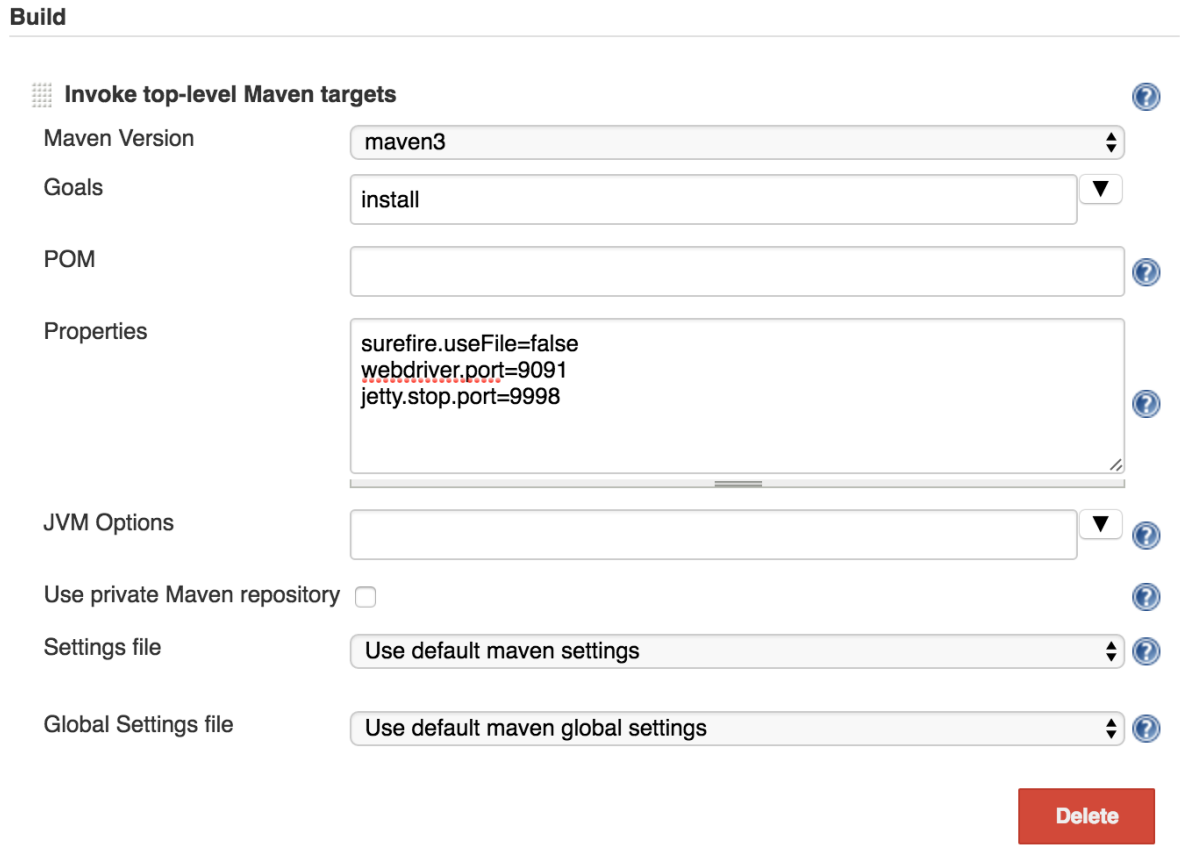


Figure 7. Maven Build Step Advanced Configuration

This will compile, test and package the application.

The `webdriver.port` and `jetty.stop.port` options are parameters used in the "Game of Life" project, that tell Maven which port to run the test application on during the web tests.

- Finally, you need to tell Jenkins where to find the test results:
  - Go to the **Post-build Actions** and tick the **Publish JUnit test result report**.
  - Enter the following in the **Test report XMLs** field:

```
**/target/surefire-reports/*.xml
```

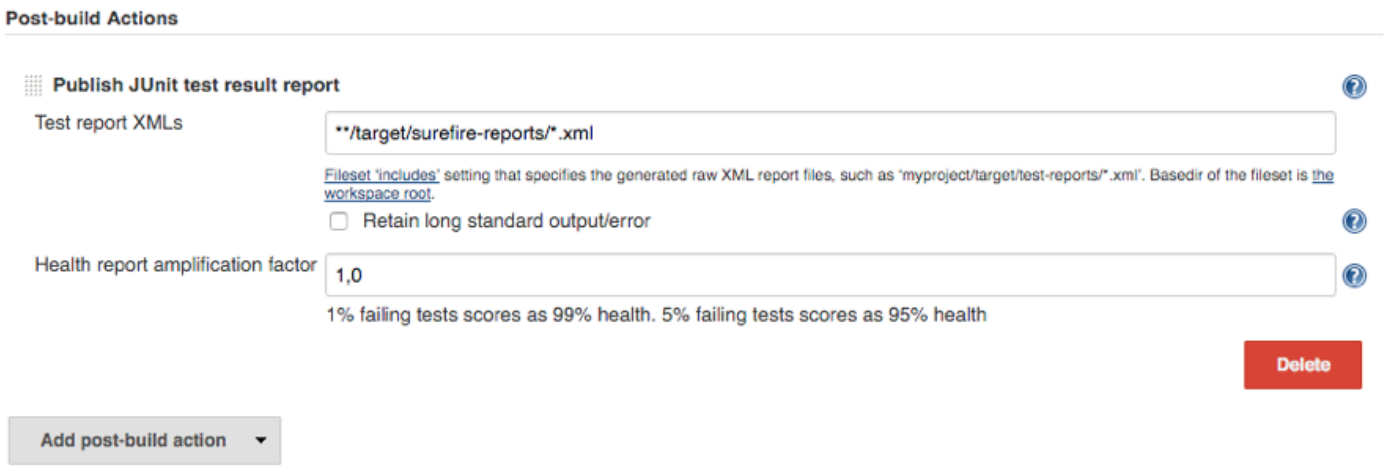



Figure 8. Post Build Step: Publishing JUnit Reports

Now save this project. A build should kick off almost immediately. Go back to the dashboard to watch the build in progress.

Go back to slides

Lab 2: Creating a Maven build job

Goal

The aim of this lab exercise is to configure a simple Maven build job on your Jenkins server.

Step 1. Create a new build job

Start by clicking on the **New Item** menu item on the home page, and choose **Maven project**.

Call this project `gameoflife-default`:

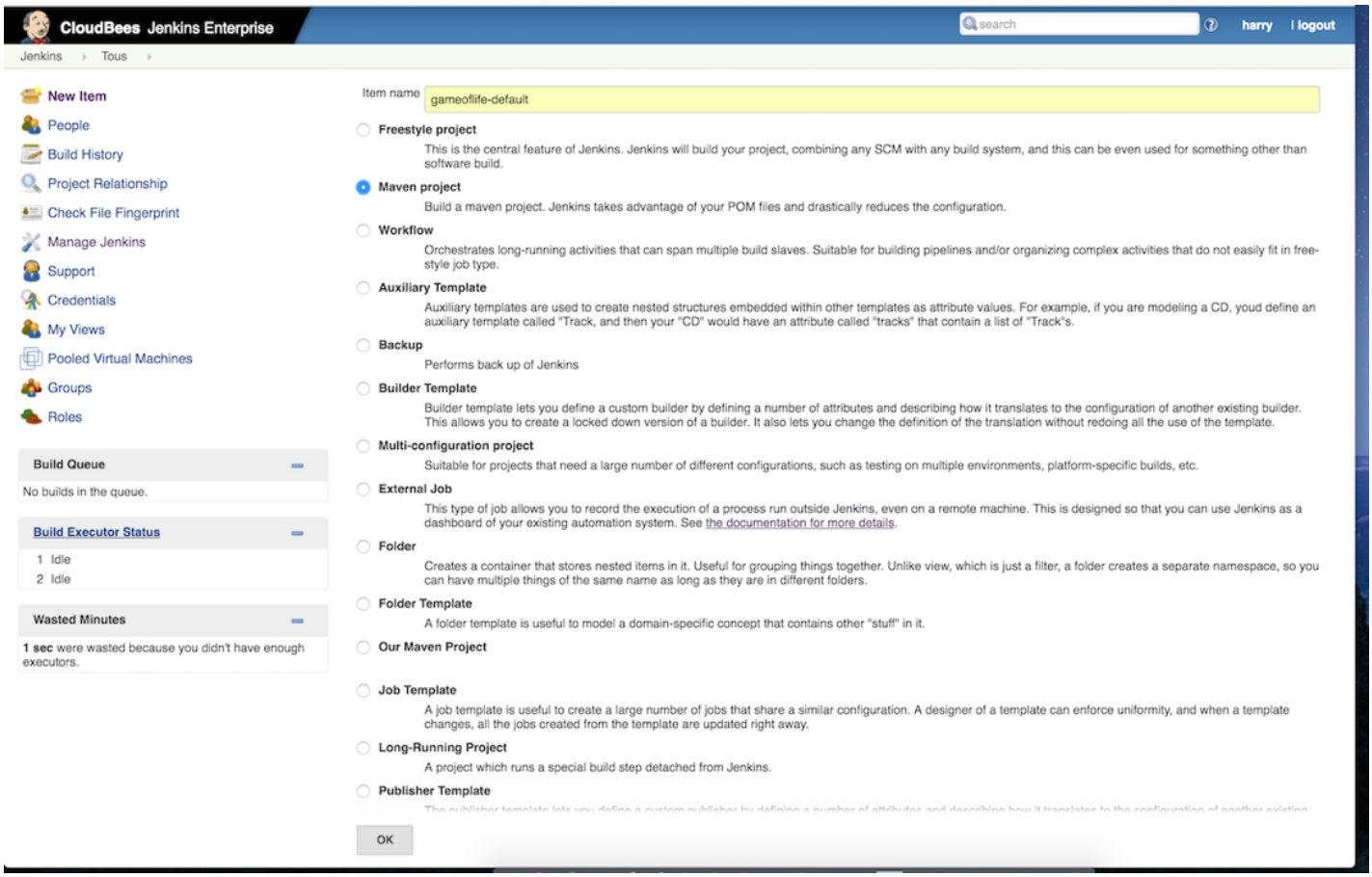


Figure 9. New Job Form of Maven kind

Step 2. Configure the SCM details

For this build job, you need to configure those SCM settings:

- As for the Lab 2, choose **Git** in the **Source Code Management** section
- In the **Repository URL** field, enter your GIT repository address:

```
http://localhost:5000/gitserver/harry/gameoflife.git
```

- Re-use the Credentials you configured in Lab 2

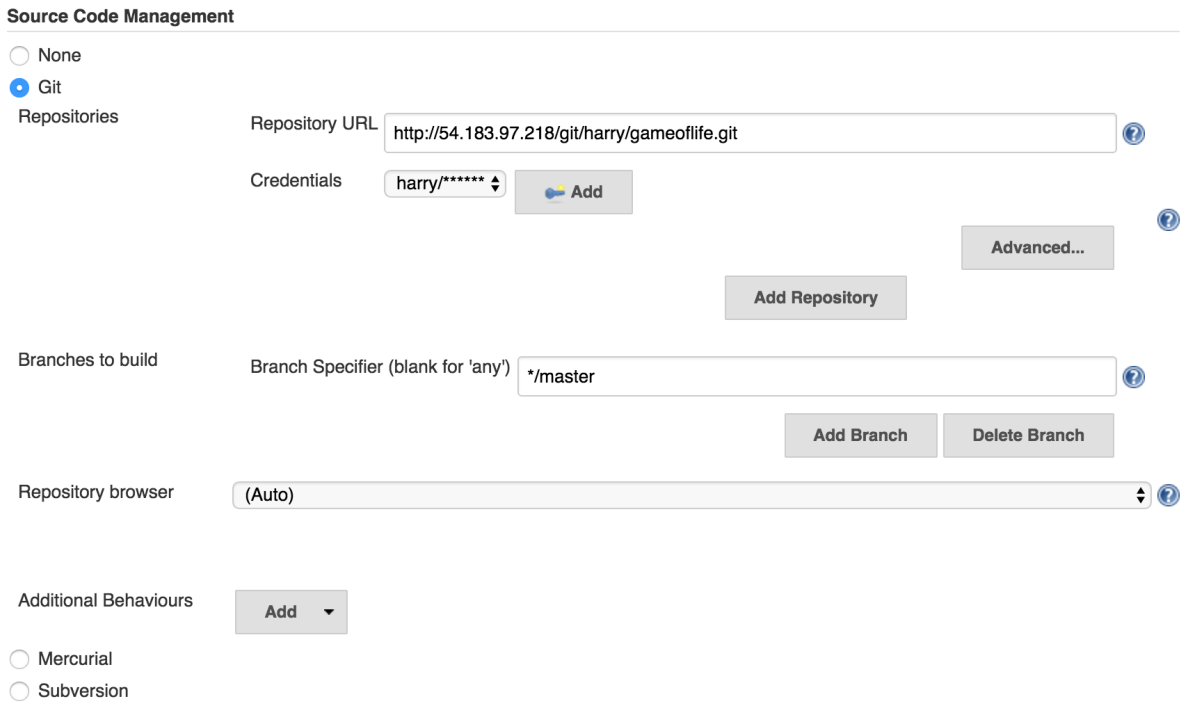


Figure 10. SCM Configuration for Job

Step 3. Configure the Build Triggers

As for the Lab 2, Build Triggers have to be configured as below:

- Polling the SCM repository every minute. In the **Poll SCM** section, enter:

```
* * * * *
```

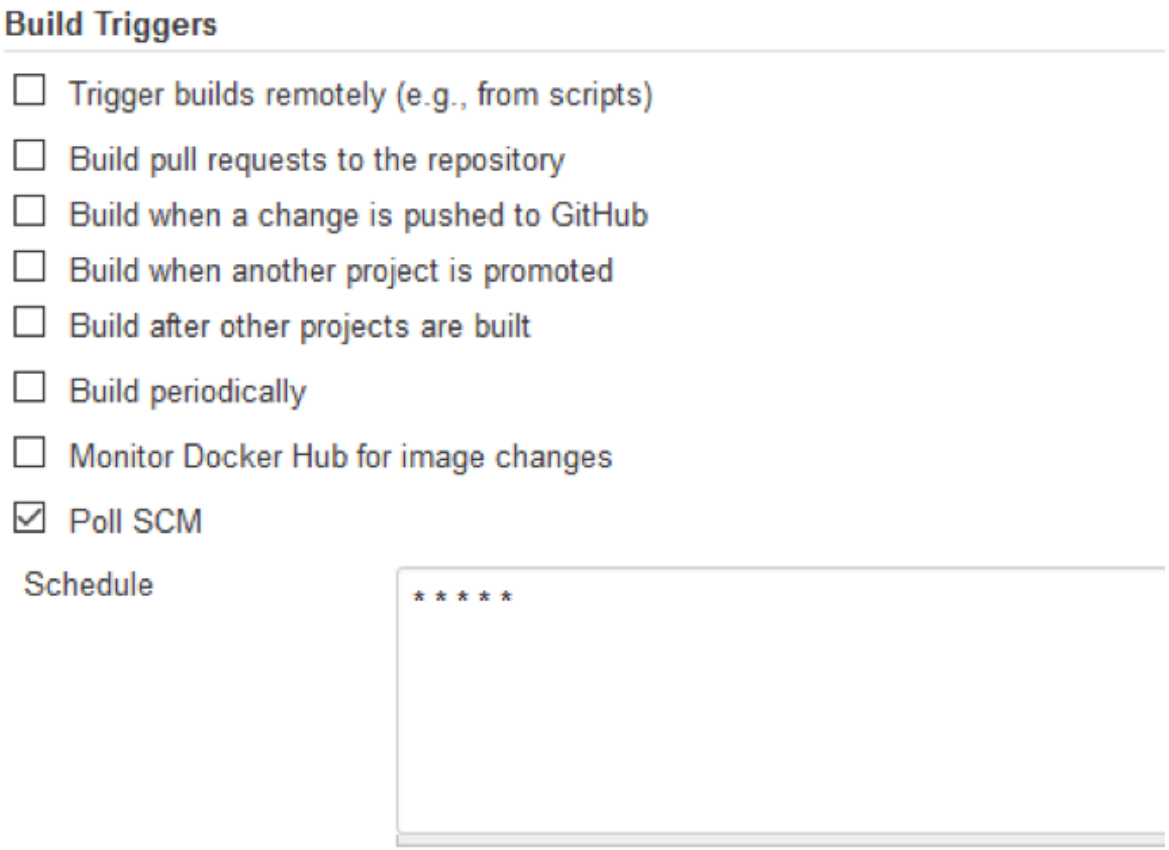



Figure 11. Build Triggers Configuration for Job

Step 4. Configure the build goals

Finally, you need to configure the Maven build goals, in the **Goals and options** field:

```
clean install -B -U -Dsurefire.useFile=false
```

Figure 12. Maven Job Build Configuration

This will compile, test and package the application, and **deploy** the latest snapshots to the local repository.

Now save this project. A build should kick off almost immediately.

Go back to the dashboard to watch the build in progress.

Go back to slides

Lab 3: Displaying test results

Goal

In this lab, you will see how Jenkins displays failing tests. You will need to modify the source code of the application.

For this, we provide you an online IDE, hosted on the machine.

Access the IDE with this URL, and authenticate as `harry`:

```
http://localhost:5000/webide
```

TIP **harry**'s password is also `harry`

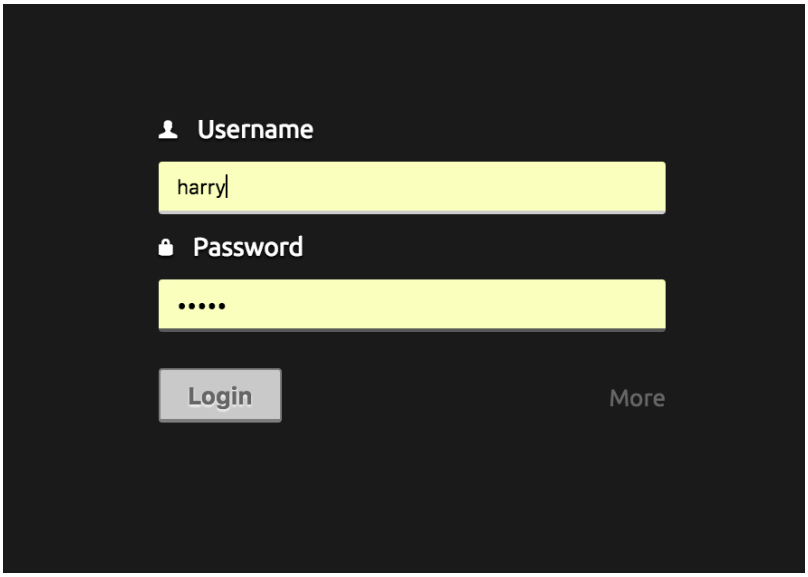


Figure 13. Authenticating on the Web IDE

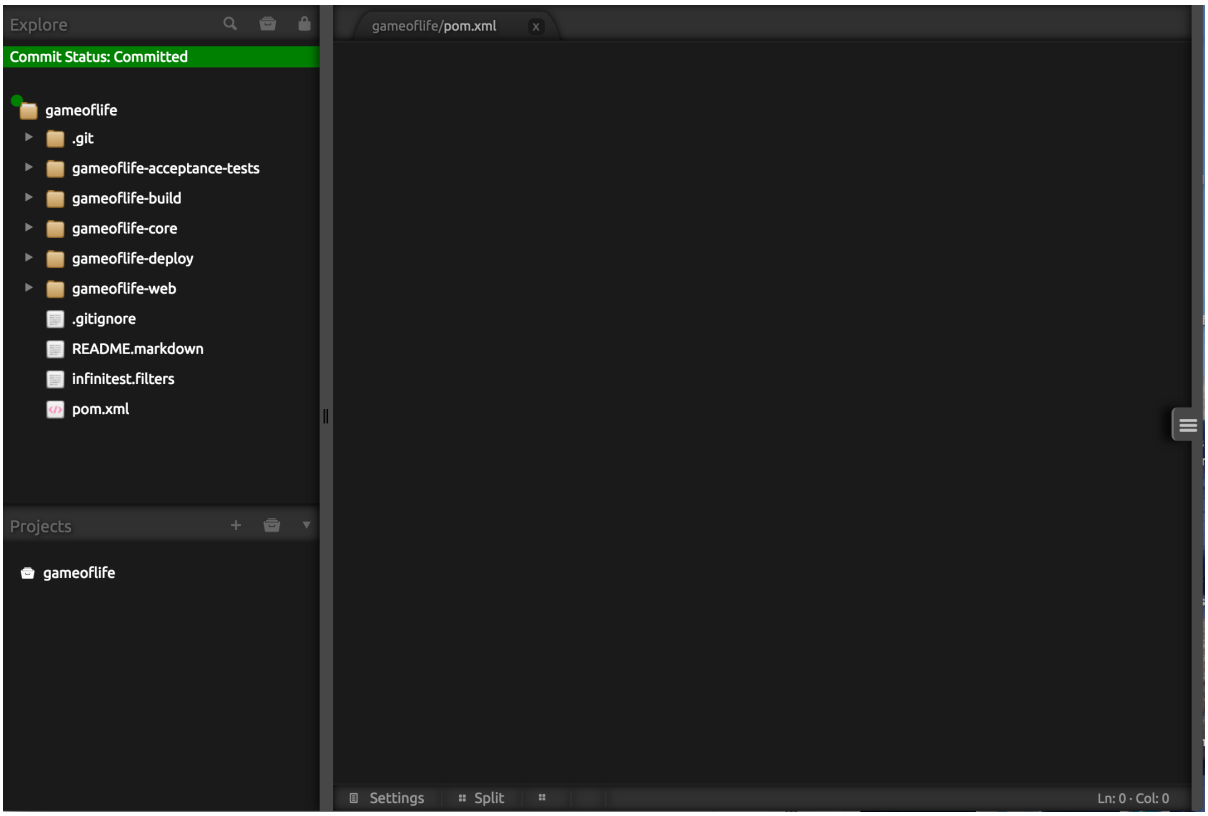


Figure 14. Main Page of the Web IDE

Step 1. Triggering a failed build

On the **Projects** section (bottom right panel), click on the **+** button to initialize our project.

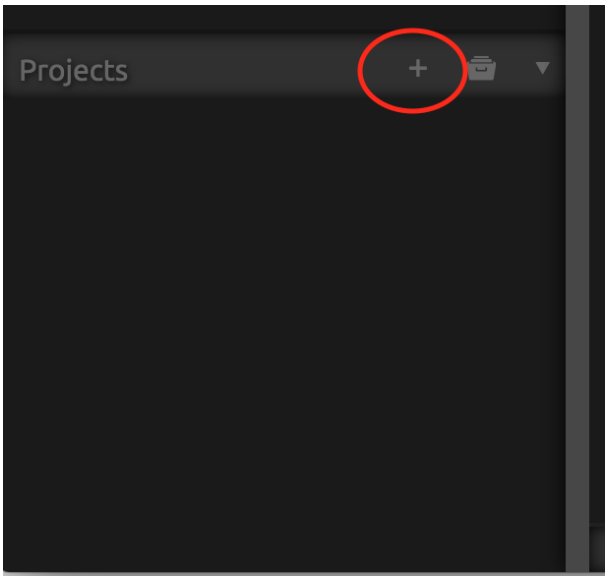


Figure 15. Create project in the Web IDE

Then extend the form by clicking the **...from Git Repo** button, and fill it with those settings:

- **Project Name**: `gameoflife`

- **Folder Name or Absolute Path**: `gameoflife`
- **Git Repository**: `http://localhost:5000/gitserver/harry/gameoflife.git`
- **Branch**: `master`

Then launch the initialization by clicking the **Create Project** button:

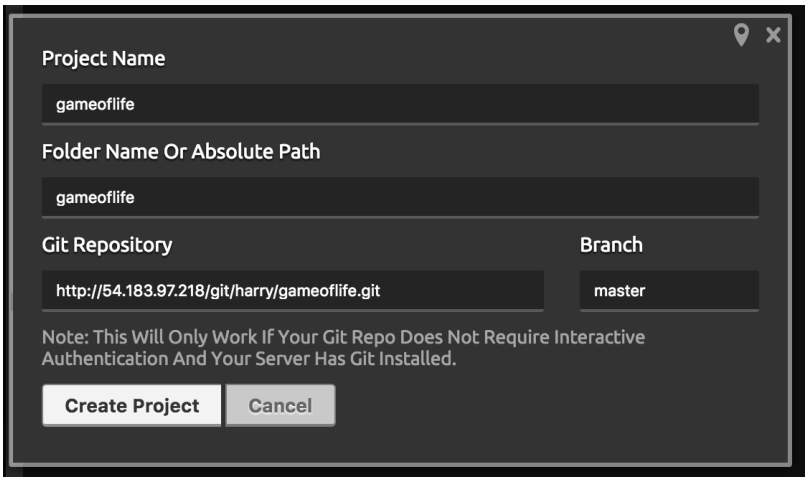


Figure 16. New project form in the Web IDE

TIP Don't forget to watch the notification in the bottom right corner !

To see how Jenkins reacts to a build failure, we are going to introduce an error into the source code.

One way to break the build is to change the symbol representing a dead cell from a period (".") to something else (say, a dash ("-")).

You can do this by modifying the `Cell.java` class, which is located in:

```
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/
```

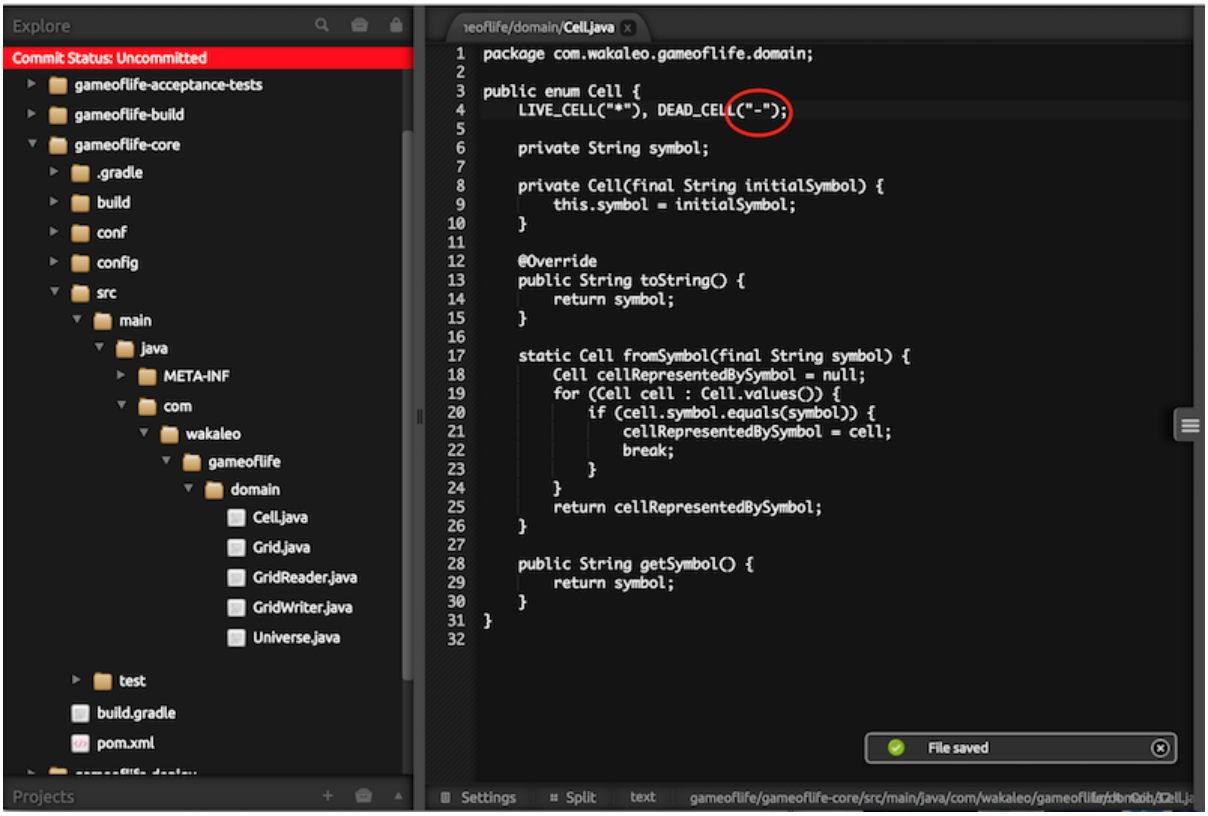



Figure 17. Editing Cell.java Class

Now we have to commit these changes to the **local** Git repository, and synchronize the change to the **remote** Git repository.

- First, navigate to the **CodeGit**, by using the right panel:

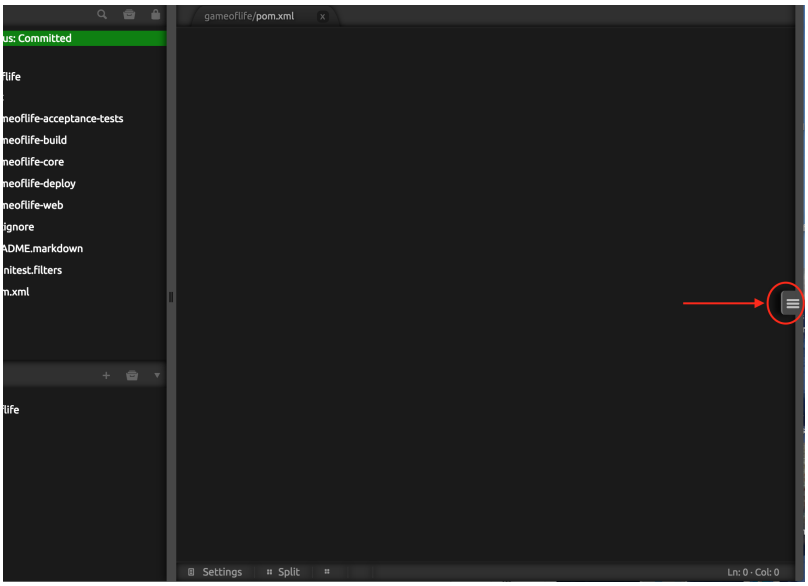


Figure 18. Accessing the right panel

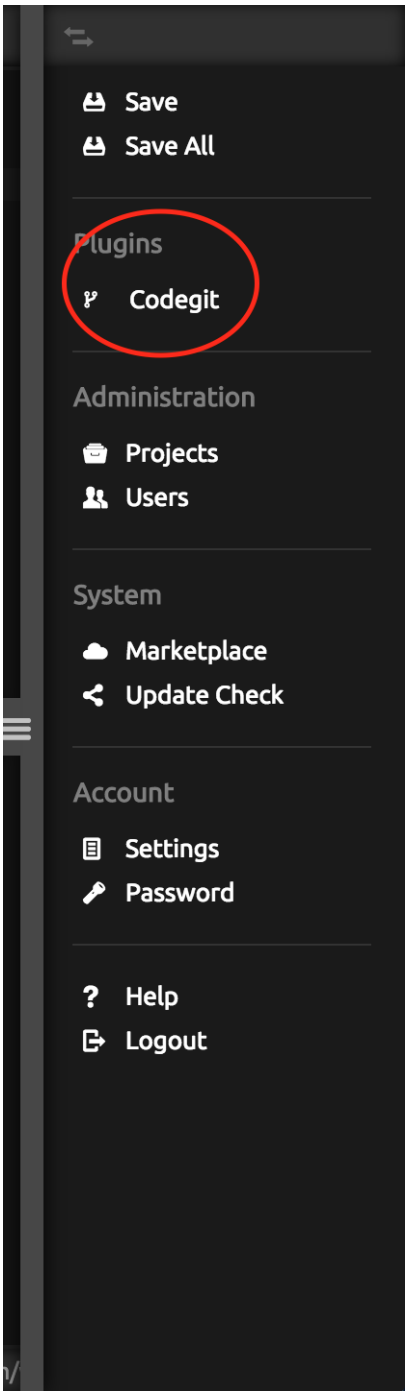


Figure 19. Accessing CodeGit

- In the GitLab view, select the change you've made on Cell class and click the **Commit** button:

Figure 20. CodeGit main view

- Then, enter a commit message like: `Changing Cell symbol` and click the **Commit** button:

Figure 21. Commit message for Cell class

TIP A green "Change committed" notification will popup on the bottom right

- We have to "push" the committed changes to the **remote** repository. For this, click the **Pull/Push** button in the main view to access the related view:

Figure 22. Acces to Push/Pull section

- On the Pull/Push view, fill the form and click to the **Push** button:
  - **Remote**: `origin`
  - **Branch**: `master`

Figure 23. Push/Pull view

- When pushing your changes, you will be prompted for a username and password. Just use `harry` for both username and password.

Figure 24. Authenticating against Remote Git Repository

Step 2. Look at the failed build in Jenkins

Now go to Jenkins and investigate how Jenkins displays the build failure.

Drill down and study the details in some of the test failures.

Also investigate the **Trend** link to see how long the tests take to run over time.

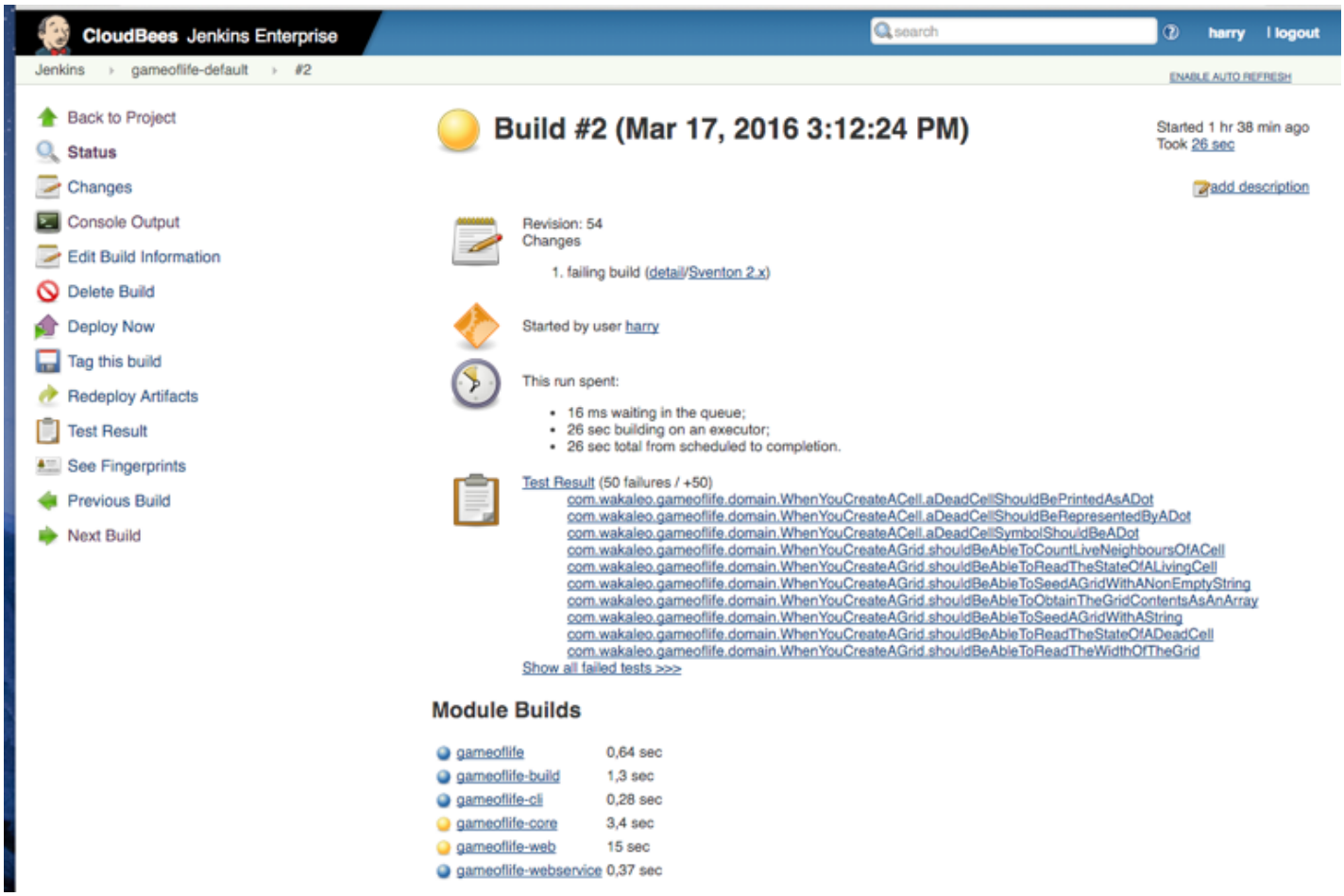


Figure 25. An unstable Build Overview

Step 3. Fix the build

Now fix the error, commit and push the changes to Git again.

Study how Jenkins displays the record of the test failures in the build history.

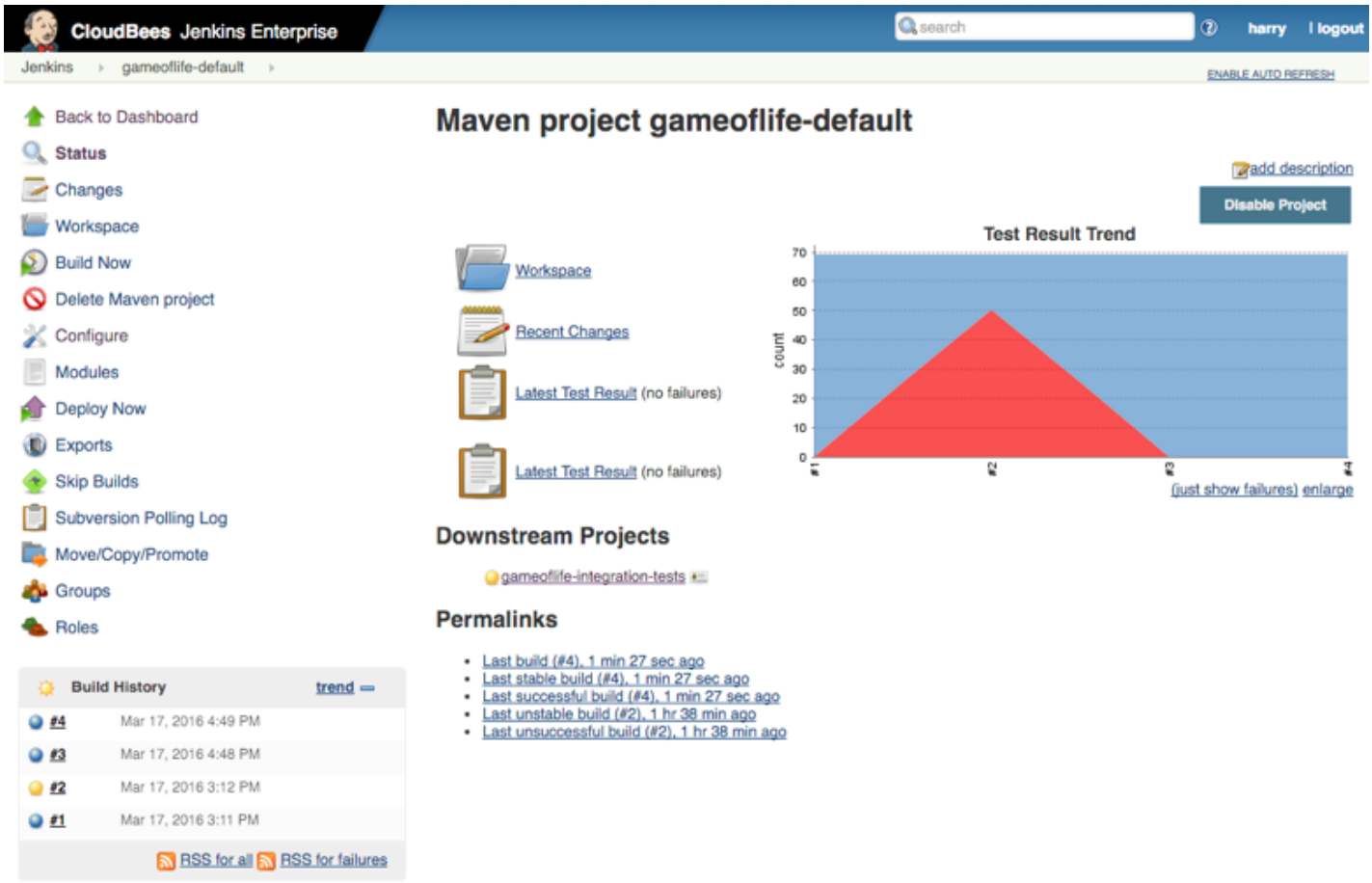


Figure 26. A fixed Build

Go back to slides

Lab 4: Creating an integration tests build

Goal

In this lab, you will learn how to create a second build job that is invoked after an initial one.

This build job will run the web tests in a Firefox browser, rather than with HTMLUnit as is done in the previous labs.

Step 1

In this lab, you will create a job called `gameoflife-integration-tests`. This build will run the `verify` Maven goal to generate a set of HTML reports about the project.

To make things simple, you will just copy your existing `gameoflife-default` project and tweak it:

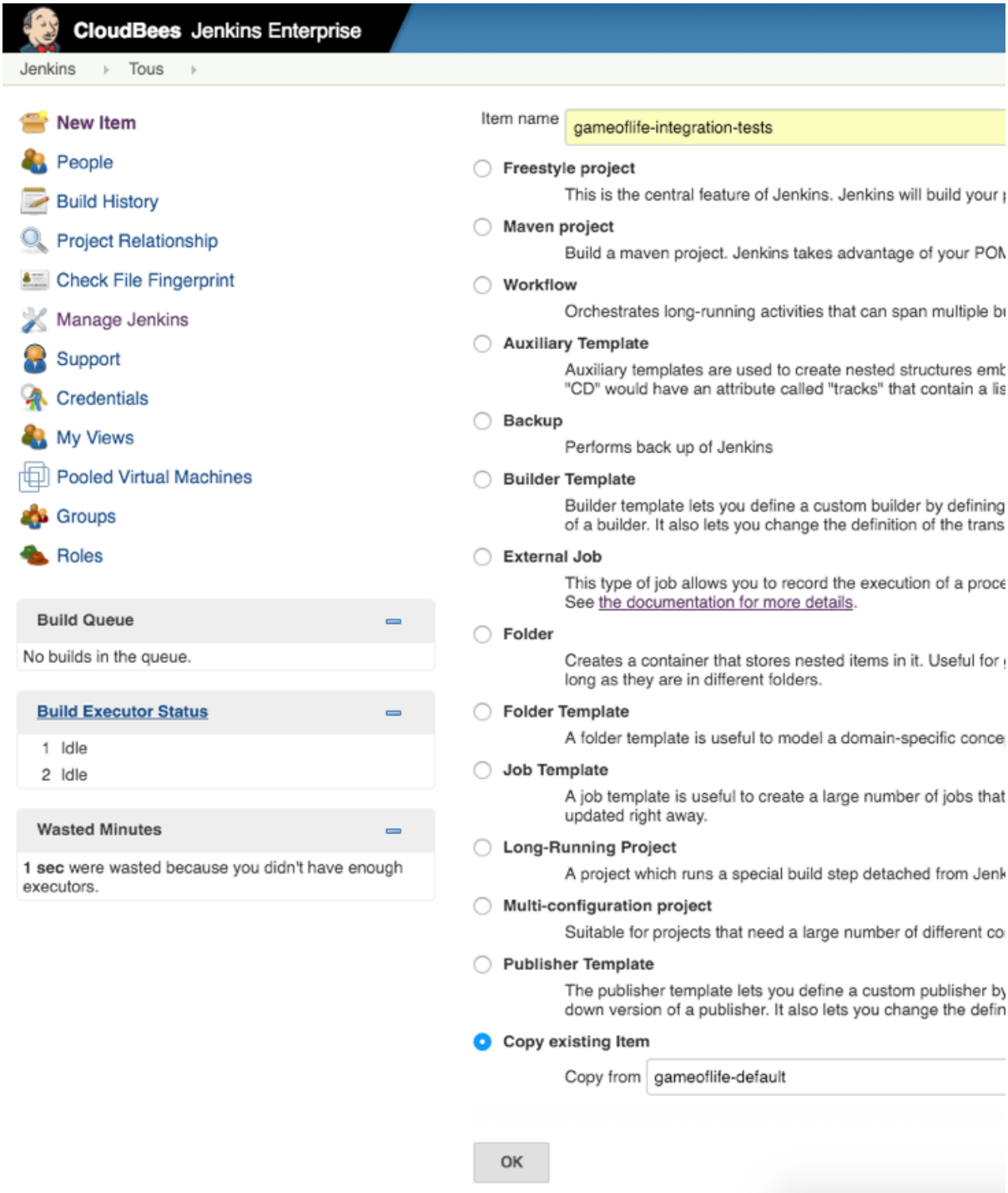


Figure 27. New Job by Copy Form

Step 2

This build will be triggered immediately after the default build job. In the Build Triggers section:

- Untick **all** the checkboxes of this section
- Select the **Build after other projects are built** checkbox and enter `gameoflife-default` in the corresponding field.

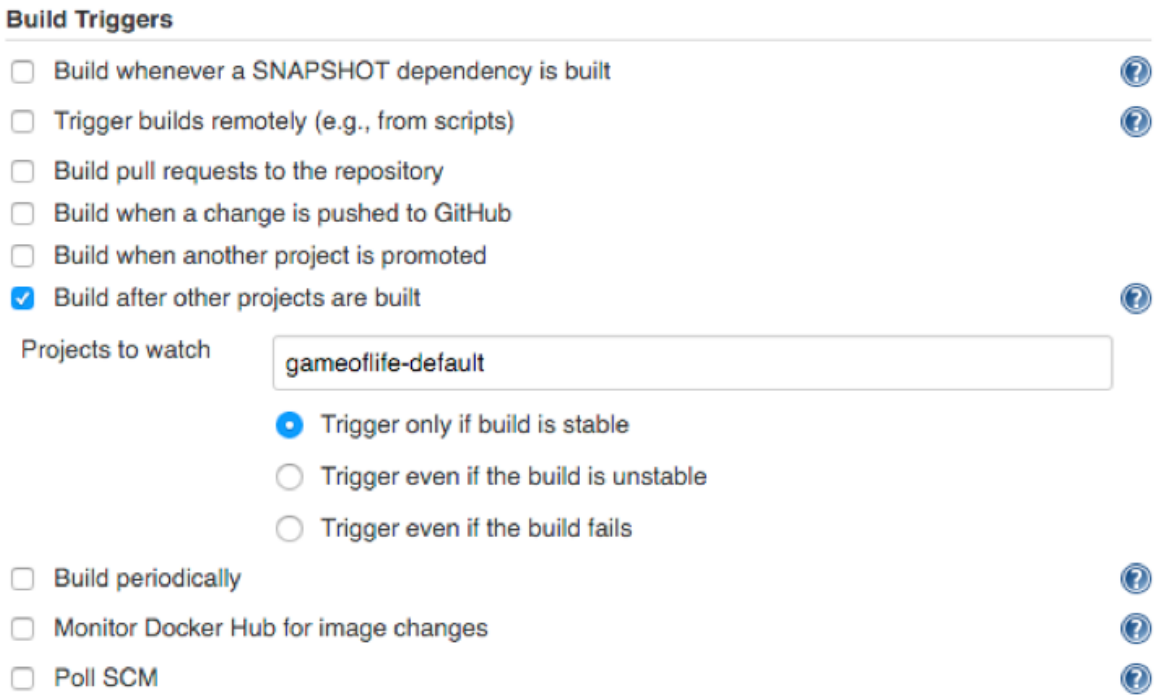


Figure 28. Trigger Build after Another Job

- Change the goals the Maven goal in the **Goals and options** field with the following values (in one line):

```
clean verify -B -U -pl gameoflife-web -Dsurefire.useFile=false
-Dwebdriver.driver=firefox -Dwebdriver.port=9092 -Djetty.stop.port=9997
```

TIP In this test, we will run the integration tests using Firefox in Jenkins.

Figure 29. Maven Configuration

Now trigger the `gameoflife-default` build manually. This should trigger this new build job, and

result in web tests being executed automatically in Firefox.

Web browser window instances will pop and close, due to the integration test build.

Go back to slides

Lab 5: Integrating with a source repository browser

Goal

In this lab, you will see how Jenkins can integrate with a source code repository browser.

Step 1. Integrate with Gogs

One of the tools installed for this workshop is the Gogs, "A painless self-hosted Git service".

It features a web GUI aimed at browsing repositories.

You can consult this server on the following URL: http://localhost:5000/gitserver

TIP You can configure Jenkins to integrate with Gogs, as well as with other source code repository browsers such like Fisheye, Trac, gitweb, Gitlab, GitHub…

In the **Source Code Management** of the configuration screen for the **gameoflife-default** project, configure the **Repository Browser list** by selecting `Go Git Service (Gogs)`.

Then configure the Repository browser with this URL:

```
http://localhost:5000/gitserver/harry/gameoflife
```

Figure 30. Configuration for Gogs Repository Browser

Step 2. Making a change

Now make a change to the source code in the `gameoflife` project and commit your change to your

local Git repository.

Open up Codiad and make one simple (and **harmless**) change. One place where you can do this is in the pom.xml file in the gameoflife directory.

Change the `<url>` element to point to your Jenkins project, i.e. `http://localhost:5000/jenkins/gameoflife-default`:

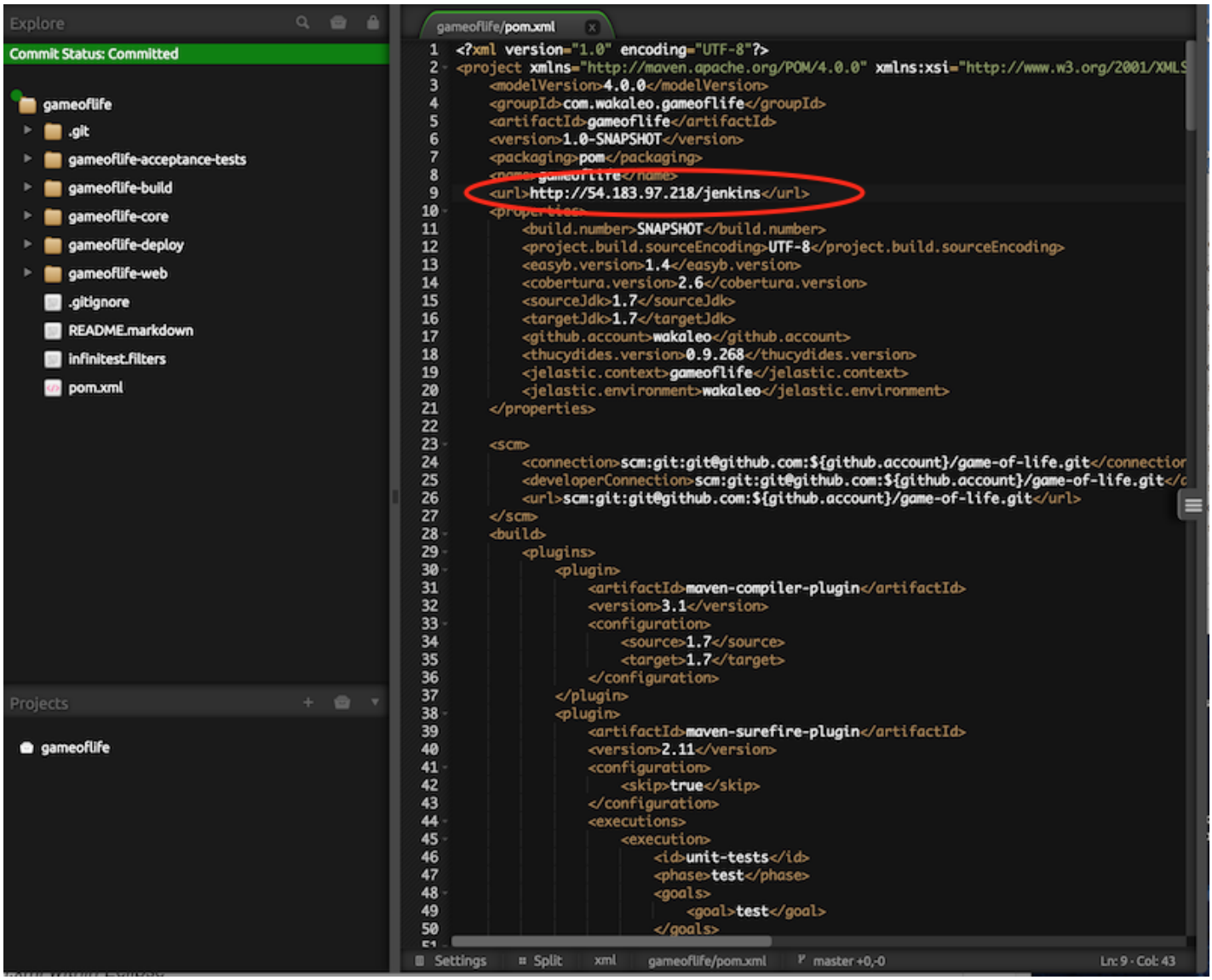


Figure 31. Editing pom.xml

Save your changes, commit and don't forget to push to the remote Git repository.

TIP If prompted, use harry as both the username and password.

Now watch the build kick off, and click on the **gameoflife-default** build job from the Jenkins home screen to go to the job's details page.

TIP Direct URL access: http://localhost:5000/jenkins/gameoflife-default

From here, click on the **Changes** link in the left-hand menu (3rd item in the menu):

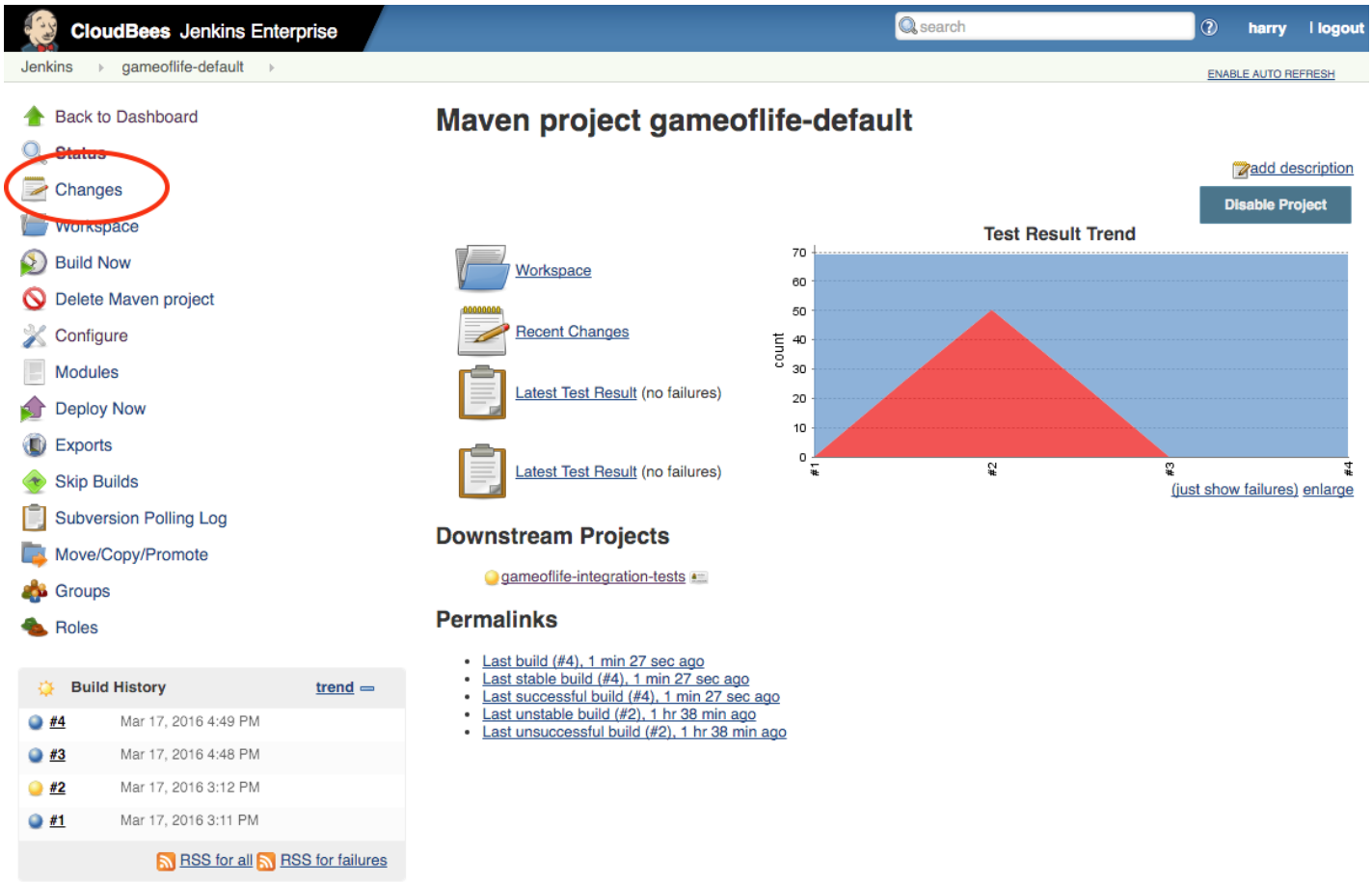


Figure 32. Link to Changes from Job Page

This will take you to a page where you can see the list of changes for each build:

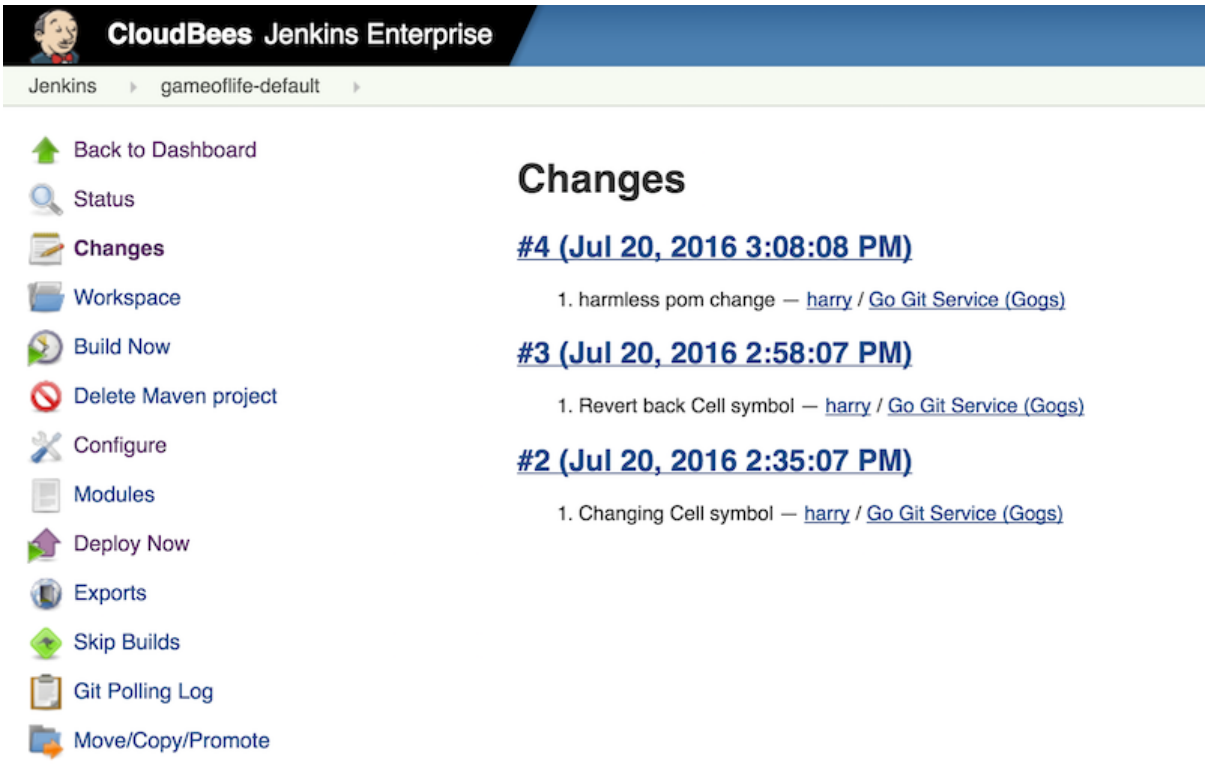


Figure 33. Changes Page

From here you can:

- Click on a build title to see the Jenkins "diff summary view":

Figure 34. Summary view

- Or access directly the diff within Gogs, using the "Repository Browser" capability, by clicking the **Go Git Service (Gogs)** or the changeset link in the summary view:

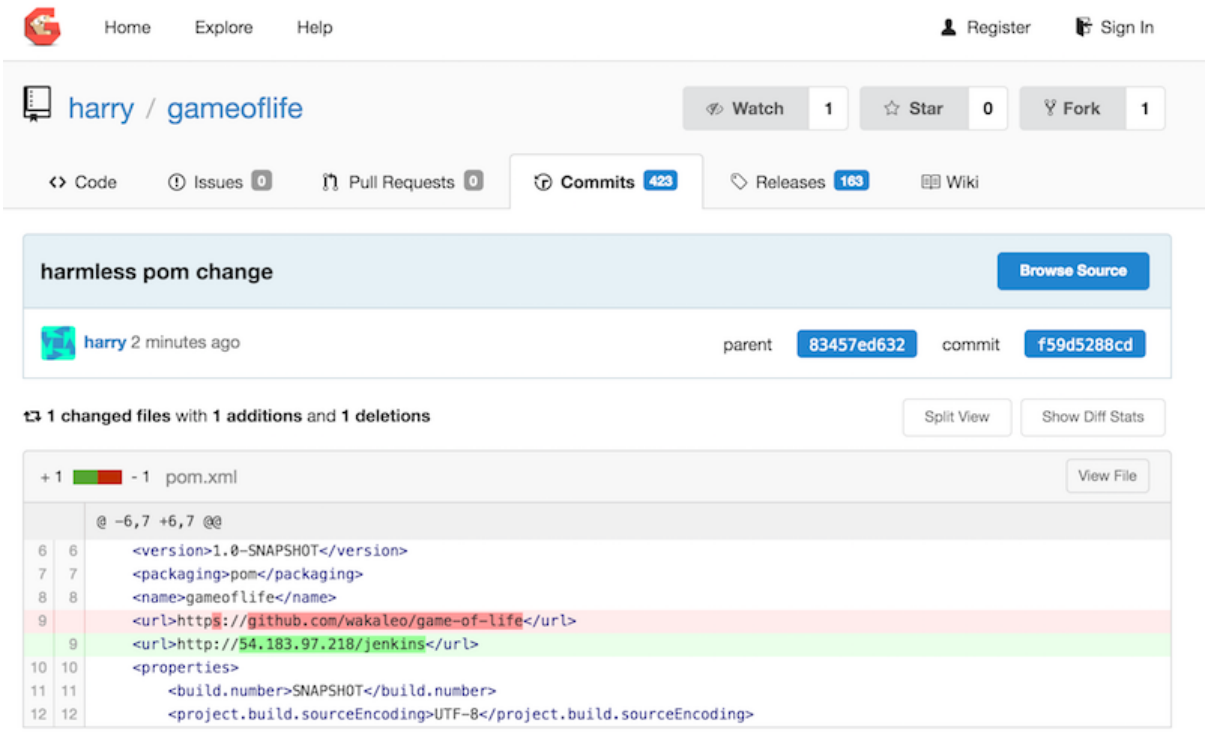


Figure 35. Diff view in Gogs

Go back to slides

Lab 6: Code quality metrics

Goal

In this lab, you will learn how to integrate code quality metrics such as Checkstyle, PMD and FindBugs into your Jenkins builds.

Step 1. Generate the code quality metrics

As we did for the integration tests, you'll have to hit the **New Item** left-hand menu item.

Name your project `gameoflife-metrics`, and select the **Copy from existing job** option. Type in **gameoflife-default** to this field:

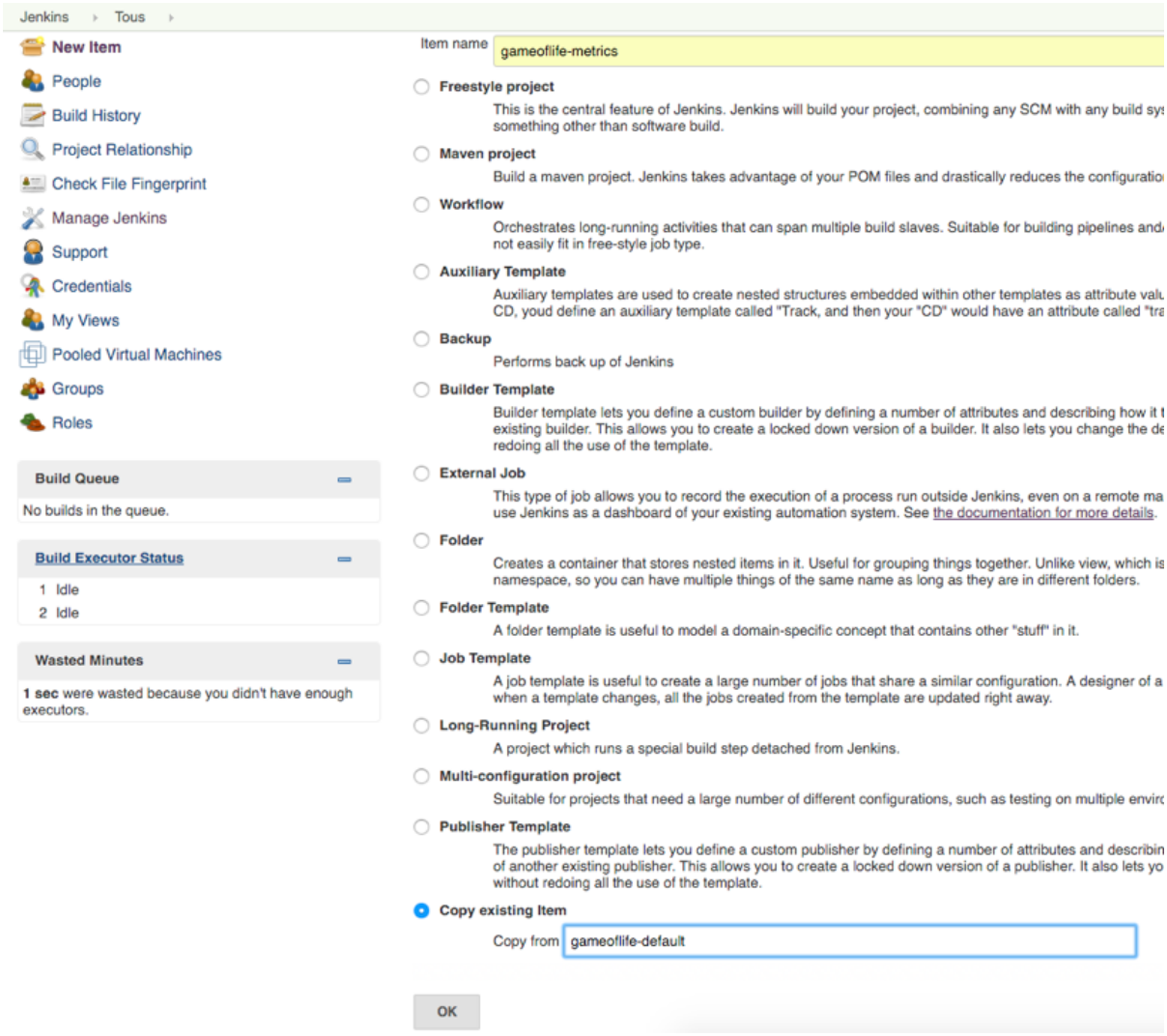


Figure 36. Create Metrics Job by Copying Another

Now we're going configure this build up to run after the integration tests by configuring the build triggers.

Since this job is copied from the **gameoflife-default** one, you'll first have to delete the **Poll SCM build trigger**.

Now, check the **Build after other projects are built** option and name **gameoflife-integration-tests** as the upstream project:

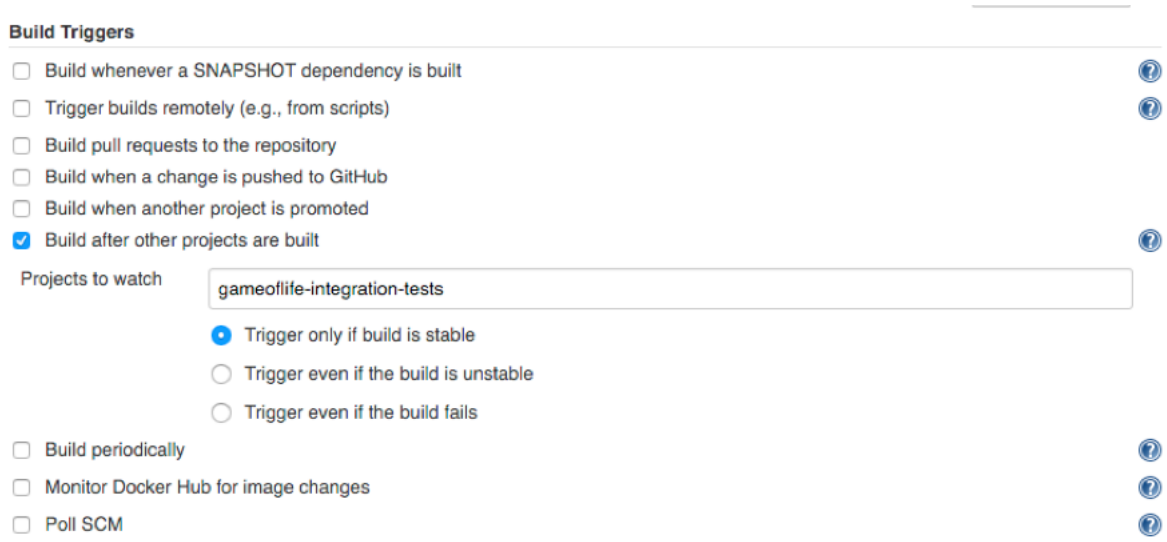


Figure 37. Job Build Trigger

We want a build job that it generates a full set of code quality metrics instead of simply the code coverage ones, so now we'll change the Maven Goal to the following (all on one line):

```
clean install site -Pmetrics -Pcoverage -B -U -Dwebdriver.port=9093
-Djetty.stop.port=9996
```

IMPORTANT Code quality metrics need plenty of memory, so click on the **Advanced** button and set `MAVEN_OPTS` field to `-Xmx256m`

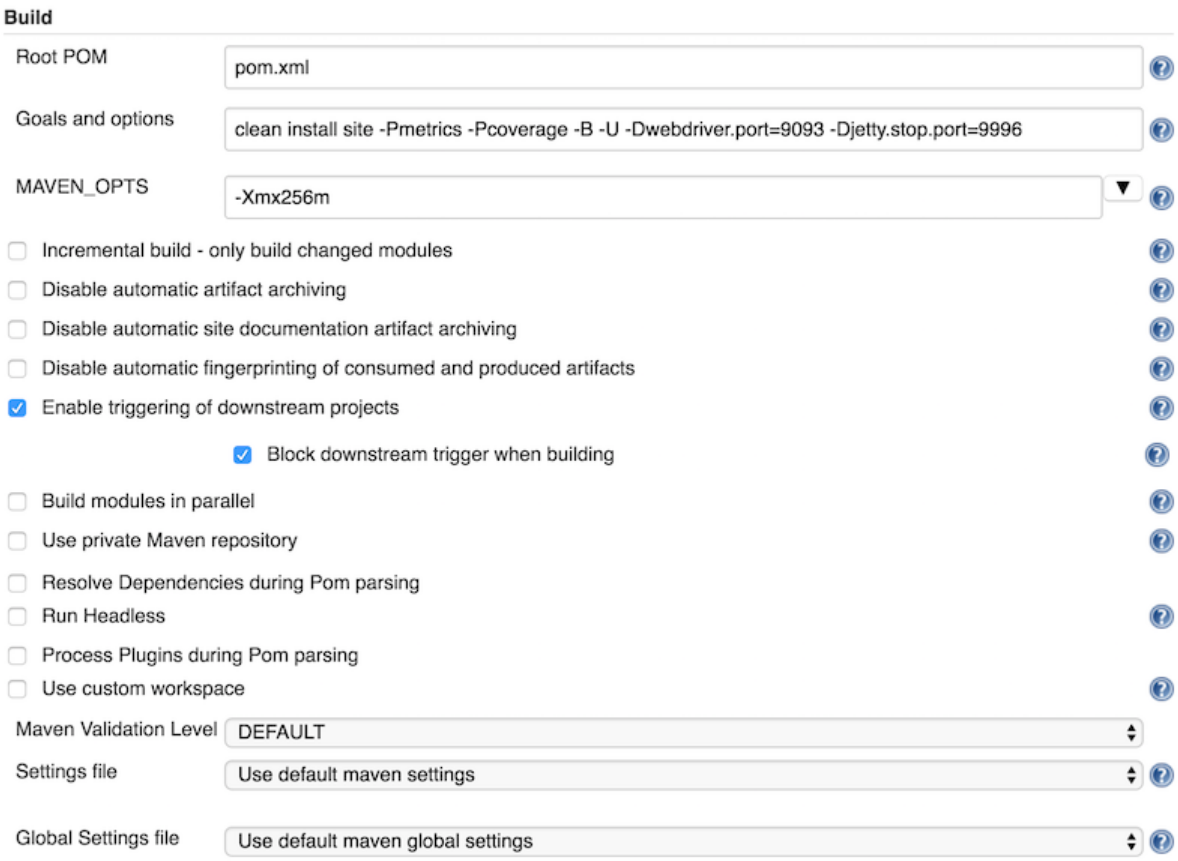


Figure 38. Job Build Configuration

Next, go to the **Build settings** section and tick the **Violations** option. For a Maven project, this plugin knows where to look to find the Checkstyle, PMD and Findbugs XML reports, so you can leave the default settings:

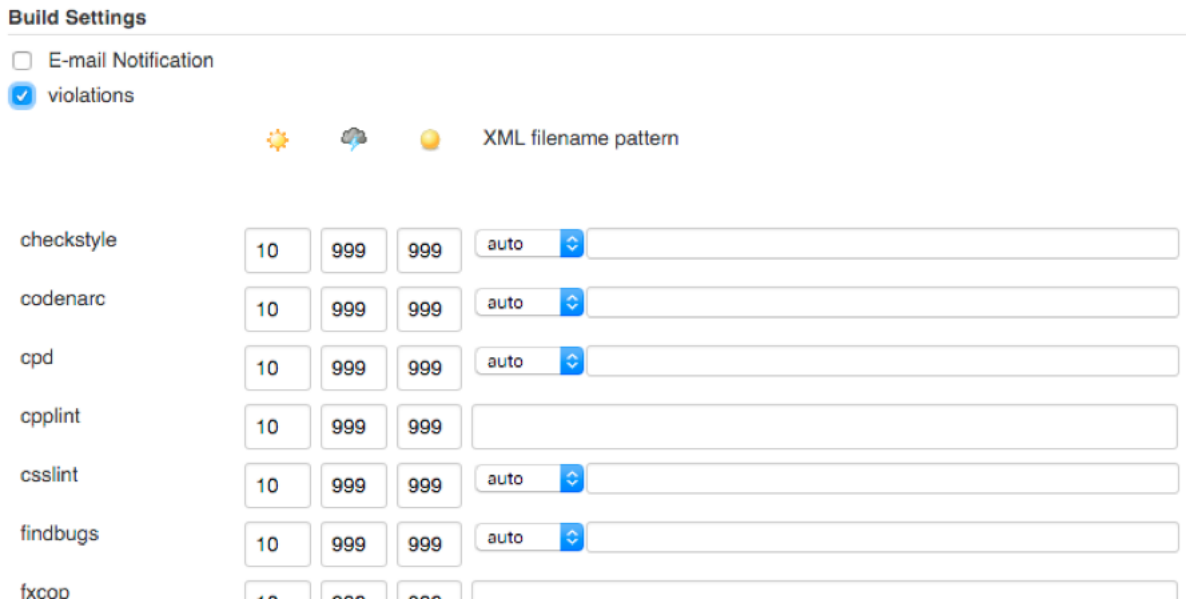


Figure 39. Violations Configuration

The configuration save should have started a first build.

TIP Since we are studying trends, a second build must have been run.

After ensuring that you have 2 builds, whether success or failed, check the project page to see global quality metrics:

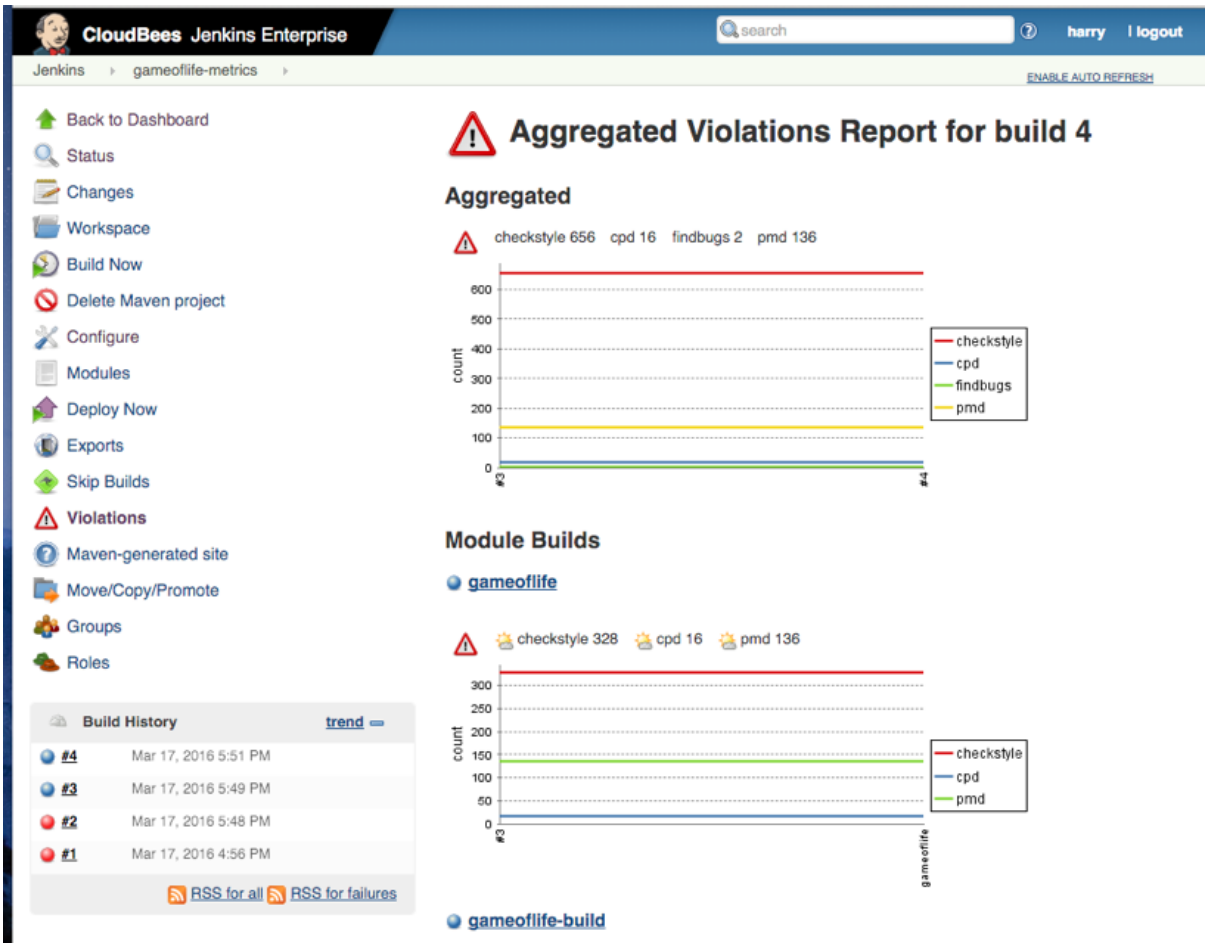


Figure 40. Violations Trend

Drill down into the modules to see how the detailed metrics are reported:

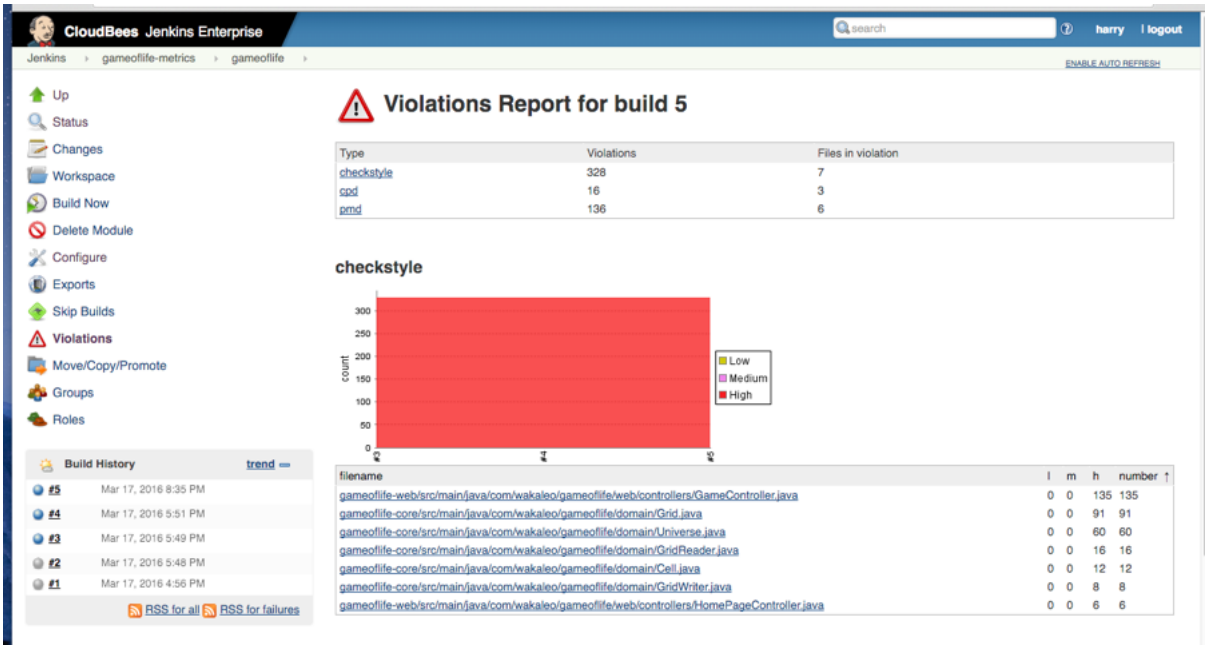


Figure 41. Details metrics

Go back to slides

Lab 7: Code coverage metrics

Goal

In this lab, you will learn how to integrate code coverage into your Jenkins builds.

Step 1. Set up code coverage reporting

We will add code coverage reporting to the **gameoflife-metrics** build job.

The project implements code coverage using JaCoCo (Java Code Coverage), a dedicated library that handles all the data generation and reporting.

IMPORTANT	The JaCoCo plugin is already installed. Do **not** try to update it since latest versions introduced breaking changes.

Triggering the export is done by the `mvn verify` goal, with just a few options, without the `install` nor the `site` goals.

TIP	We will add a post-step Maven invocation, that will clean some compiled classes and then generate the test coverage reports.

Open the build job configuration page, and:

- First, in the section **Post Steps**, add a **post-build step** step:
  - The type to use is **Invoke top-level Maven targets**.
  - Select the **maven3** Maven installation
  - Enter those targets:

```
clean verify -Pmetrics
```

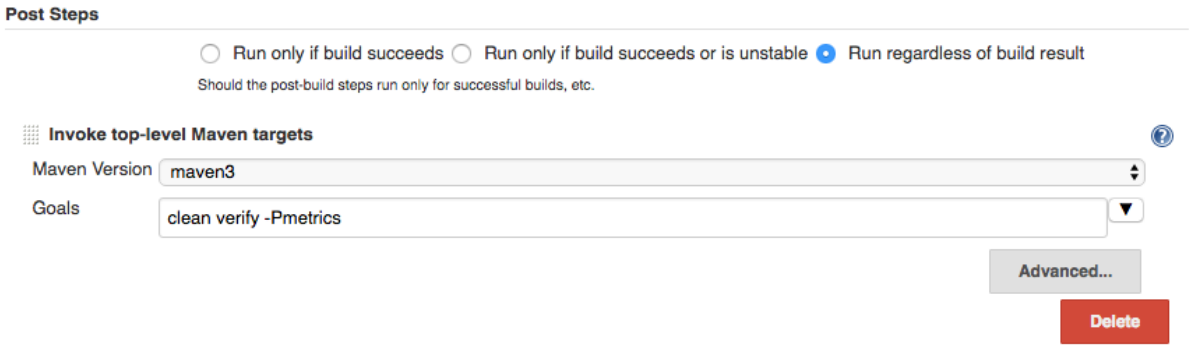


Figure 42. Post-Step Maven for Coverage

TIP Now, it is time to make Jenkins aware of the generated data to let it publish a report for each build

- Then go to the **Post-build Actions** and select the type **Record JaCoCo coverage report**, with this configuration:
  - **Path to exec files**: `**/target/*.exec`
  - **Path to class directories**: `**/target`
  - **Path to source directories**: `**/src/main/java`
  - **Inclusions**: `**/*.class`
  - **Exclusions**: `**/When*`

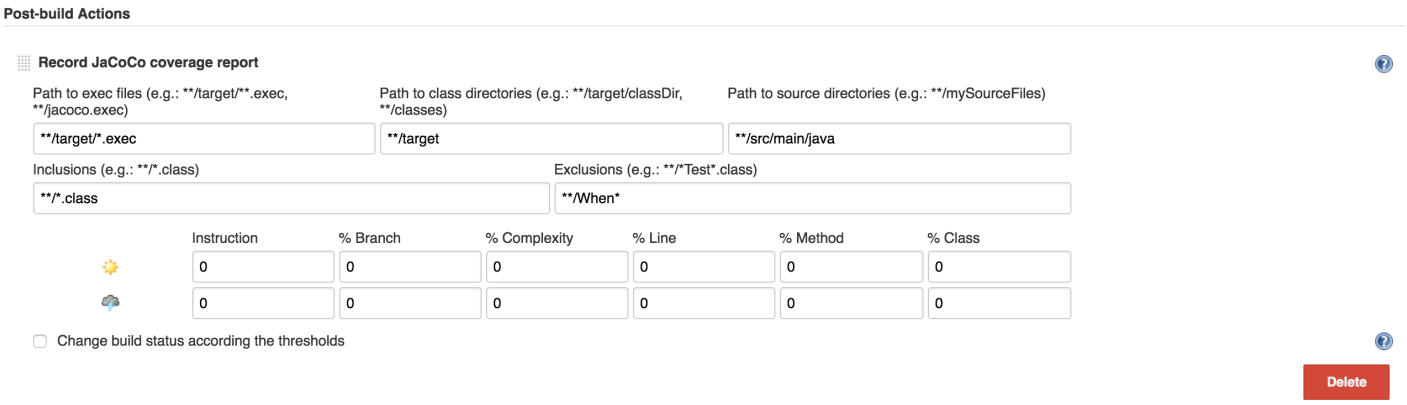


Figure 43. Post-Build Action Configuration

TIP

This is the JaCoCo settings for our projects. For more information, please read the `pom.xml` file and JaCoCo documentation

Make sure you hit the **Save** button on the bottom of the screen,

Now let the build run, and then consult the new code coverage metrics reports:

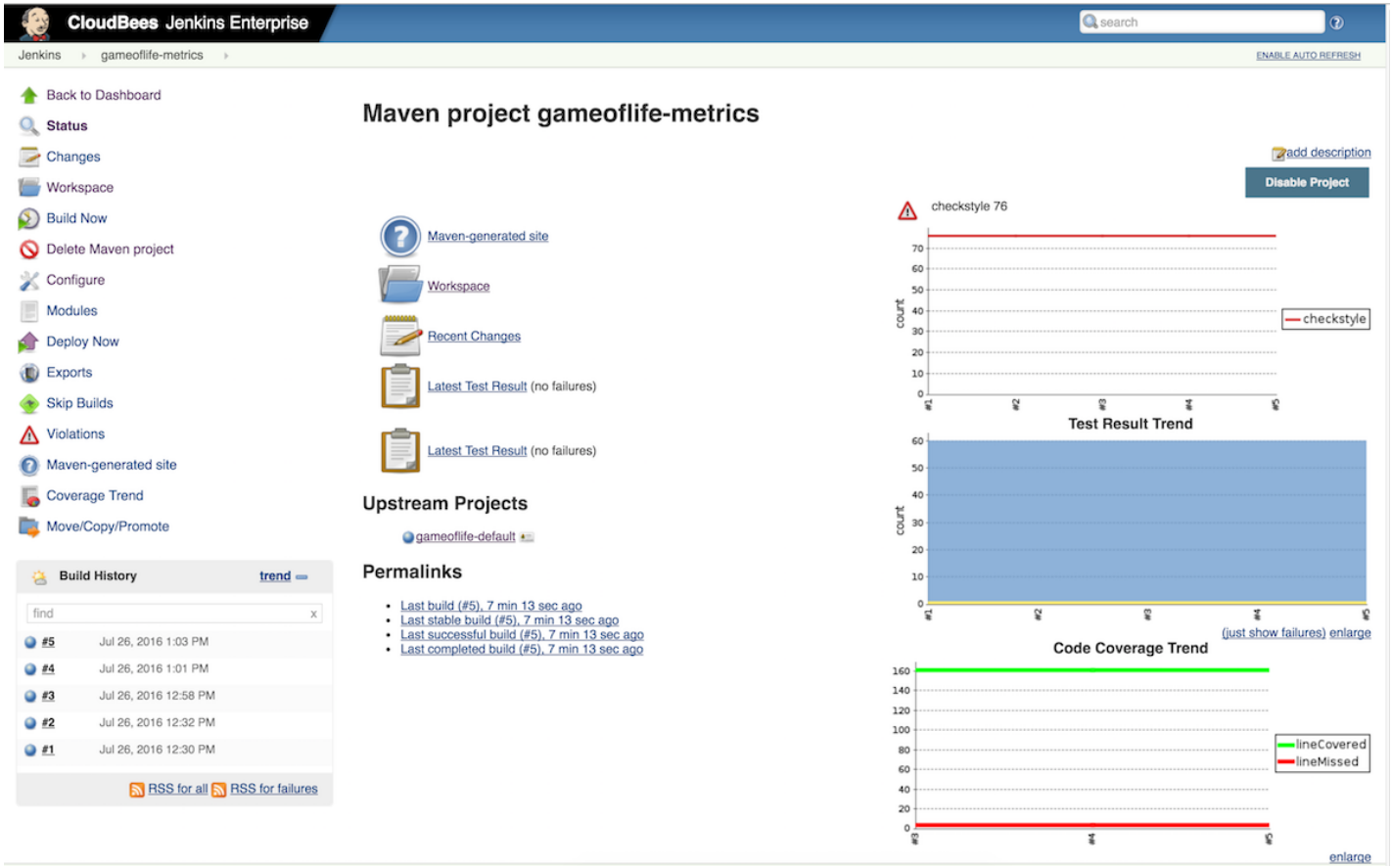
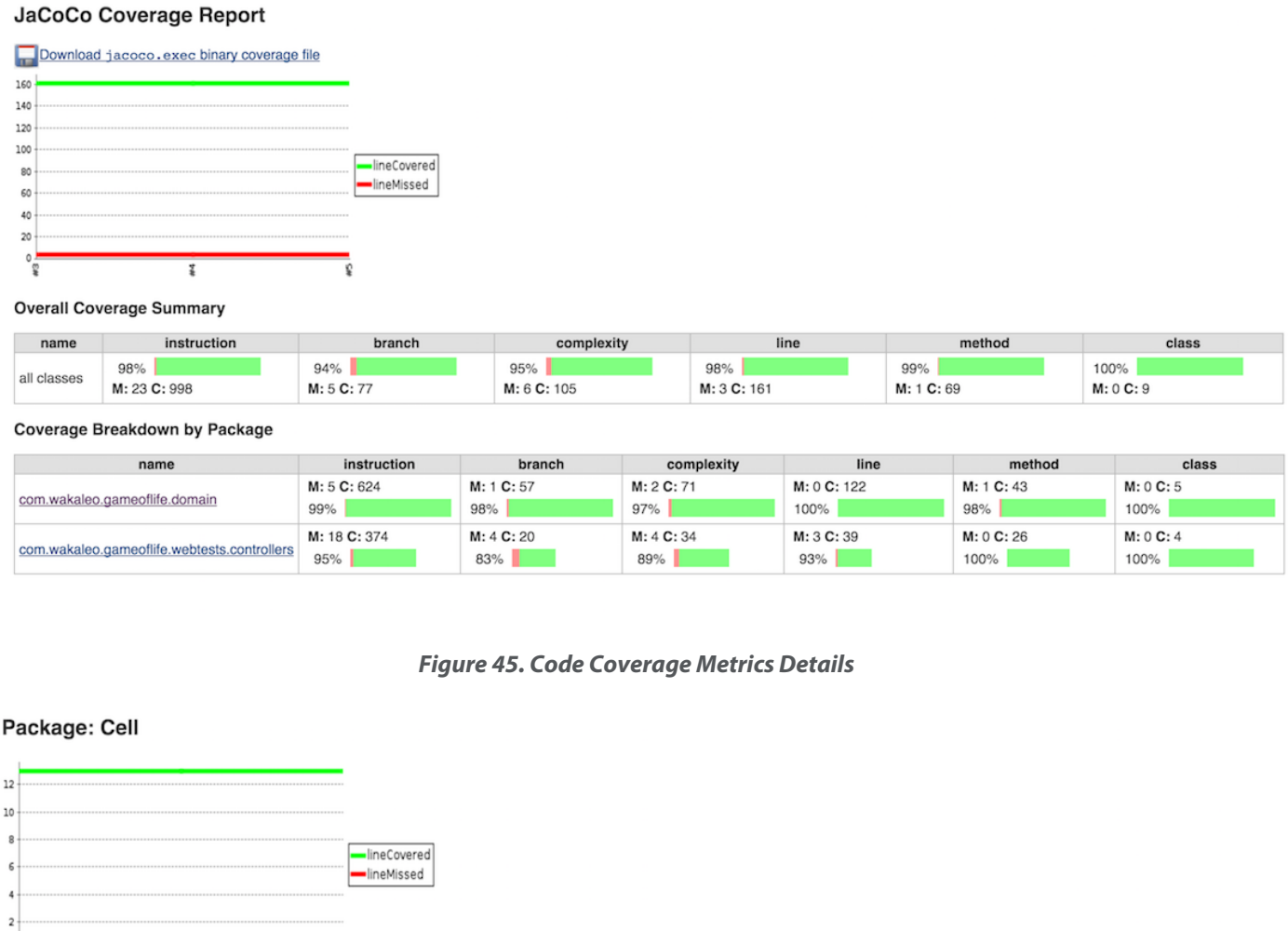


Figure 44. Code Coverage Metrics

Now click on either the graph itself or the **Coverage Report** link on the build's summary page to see the detailed breakdown:



JaCoCo Coverage Report

Download jacoco.exec binary coverage file

lineCovered
lineMissed

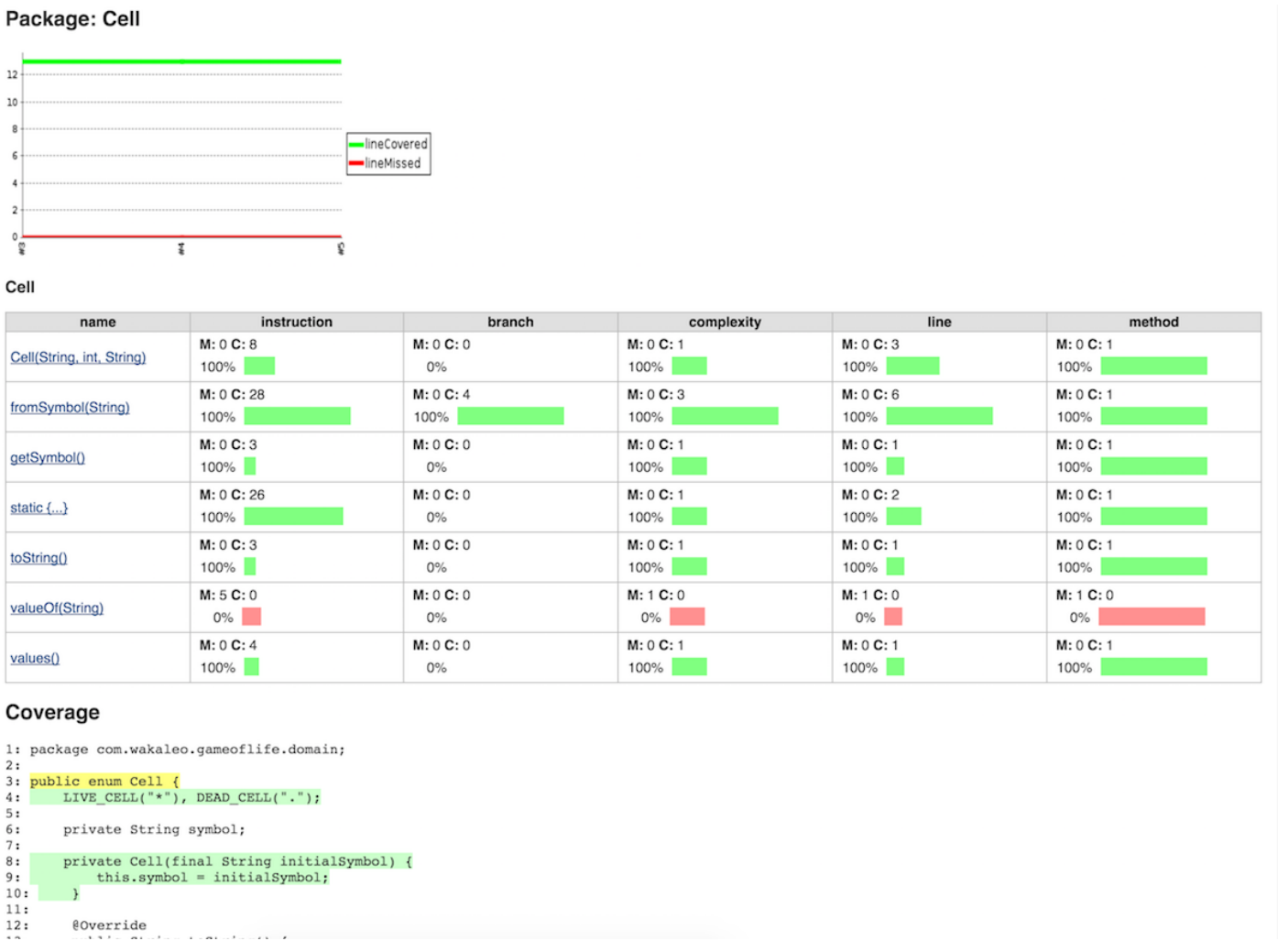
Overall Coverage Summary

name	instruction	branch	complexity	line	method	class
all classes	98% M: 23 C: 998	94% M: 5 C: 77	95% M: 6 C: 105	98% M: 3 C: 161	99% M: 1 C: 69	100% M: 0 C: 9

Coverage Breakdown by Package

name	instruction	branch	complexity	line	method	class
com.wakaleo.gameoflife.domain	M: 5 C: 624 99%	M: 1 C: 57 98%	M: 2 C: 71 97%	M: 0 C: 122 100%	M: 1 C: 43 98%	M: 0 C: 5 100%
com.wakaleo.gameoflife.webtests.controllers	M: 18 C: 374 95%	M: 4 C: 20 83%	M: 4 C: 34 89%	M: 3 C: 39 93%	M: 0 C: 26 100%	M: 0 C: 4 100%

Figure 45. Code Coverage Metrics Details



Package: Cell

lineCovered
lineMissed

Cell

name	instruction	branch	complexity	line	method
Cell(String, int, String)	M: 0 C: 8 100%	M: 0 C: 0 0%	M: 0 C: 1 100%	M: 0 C: 3 100%	M: 0 C: 1 100%
fromSymbol(String)	M: 0 C: 28 100%	M: 0 C: 4 100%	M: 0 C: 3 100%	M: 0 C: 6 100%	M: 0 C: 1 100%
getSymbol()	M: 0 C: 3 100%	M: 0 C: 0 0%	M: 0 C: 1 100%	M: 0 C: 1 100%	M: 0 C: 1 100%
static {...}	M: 0 C: 26 100%	M: 0 C: 0 0%	M: 0 C: 1 100%	M: 0 C: 2 100%	M: 0 C: 1 100%
toString()	M: 0 C: 3 100%	M: 0 C: 0 0%	M: 0 C: 1 100%	M: 0 C: 1 100%	M: 0 C: 1 100%
valueOf(String)	M: 5 C: 0 0%	M: 0 C: 0 0%	M: 1 C: 0 0%	M: 1 C: 0 0%	M: 1 C: 0 0%
values()	M: 0 C: 4 100%	M: 0 C: 0 0%	M: 0 C: 1 100%	M: 0 C: 1 100%	M: 0 C: 1 100%

Coverage

```
1: package com.wakaleo.gameoflife.domain;
2:
3: public enum Cell {
4:     LIVE_CELL("*"), DEAD_CELL(".");
5:
6:     private String symbol;
7:
8:     private Cell(final String initialSymbol) {
9:         this.symbol = initialSymbol;
10:    }
11:
12:    @Override
```

Figure 46. Code Coverage Metrics per Class

Go back to slides

Lab 8: Parameterized builds

Goal

In this lab, you will learn how to run the integration tests in a different browser, by getting Jenkins to prompt for the browser type when you trigger the build.

Step 1. Check installation state of the Parameterized Triggers plugin

For this lab, we will be using the **Parameterized Trigger** plugin.

It should be already installed, check it inside the **Installed** tabs of the **Managed plugins** section of the global administration.

If this is not the case, install this plugin (without restarting) from the **Available** tab and go back to the top page of Jenkins.

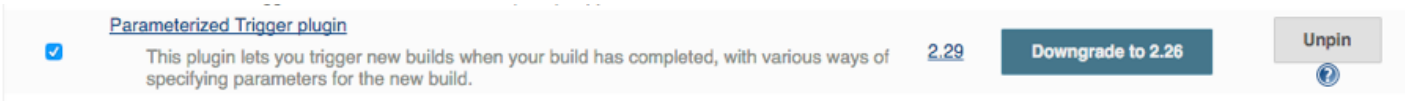


Figure 47. Parameterized Builds Plugin

Step 2. Configuring the Parameterized Triggers plugin

For this lab, we will copy the **gameoflife-integration-tests** build job, and adapt it so that it can be triggered manually. Start by copying this build job. Next, we need to create a parameter for this build job.

Tick on the **This build is parameterized**, and add a Choice parameter as shown here:

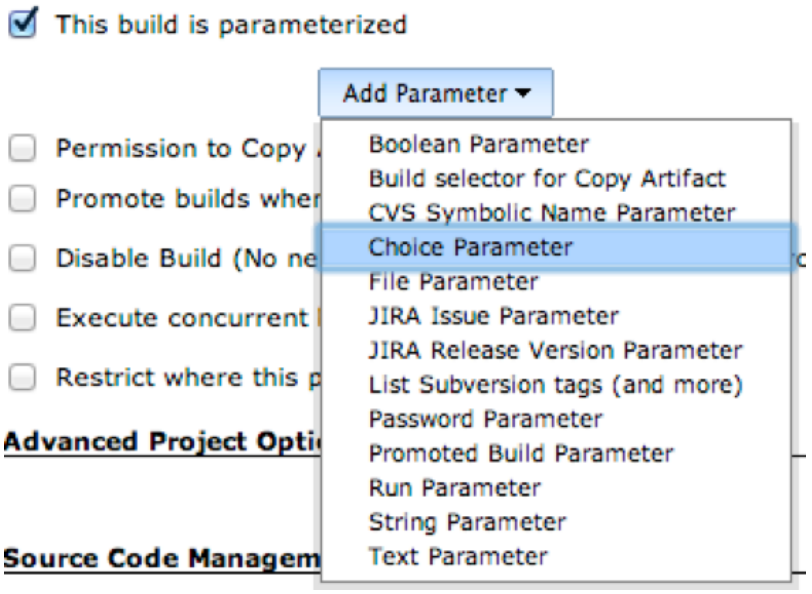


Figure 48. Adding a Choice Parameter

Name it:

```
Browser
```

And add those choices:

```
firefox
htmlunit
```

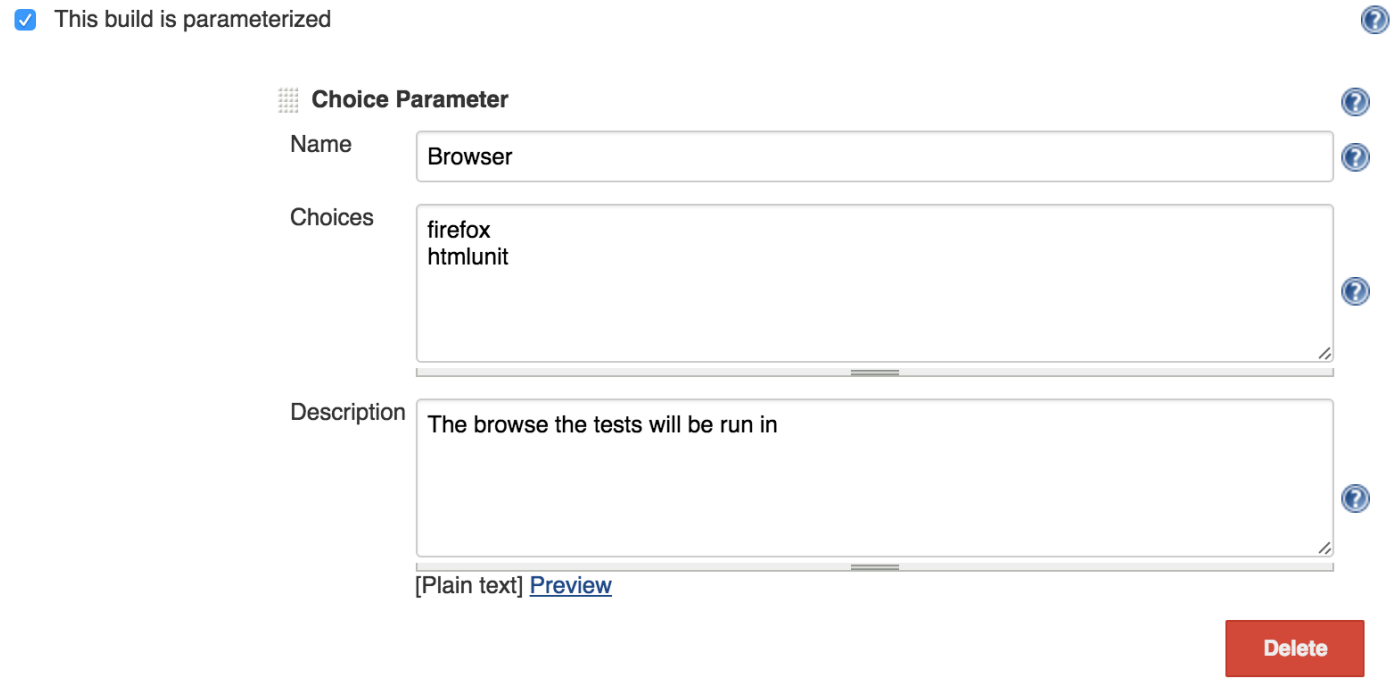


Figure 49. Configuring choices for Browser Parameter

Make it a "manual build" by deactivating all of the build triggers:

Figure 50. Manual Job Build Triggers

Now pass this parameter into the build as the `webriver.driver` property, as shown here:

```
clean verify -B -U -pl gameoflife-web -Dwebdriver.driver=${Browser}
-Dwebdriver.port=9094 -Djetty.stop.port=9996
```

Now start this build manually. Jenkins will prompt for a browser, presenting the list of supported options in a dropdown list:

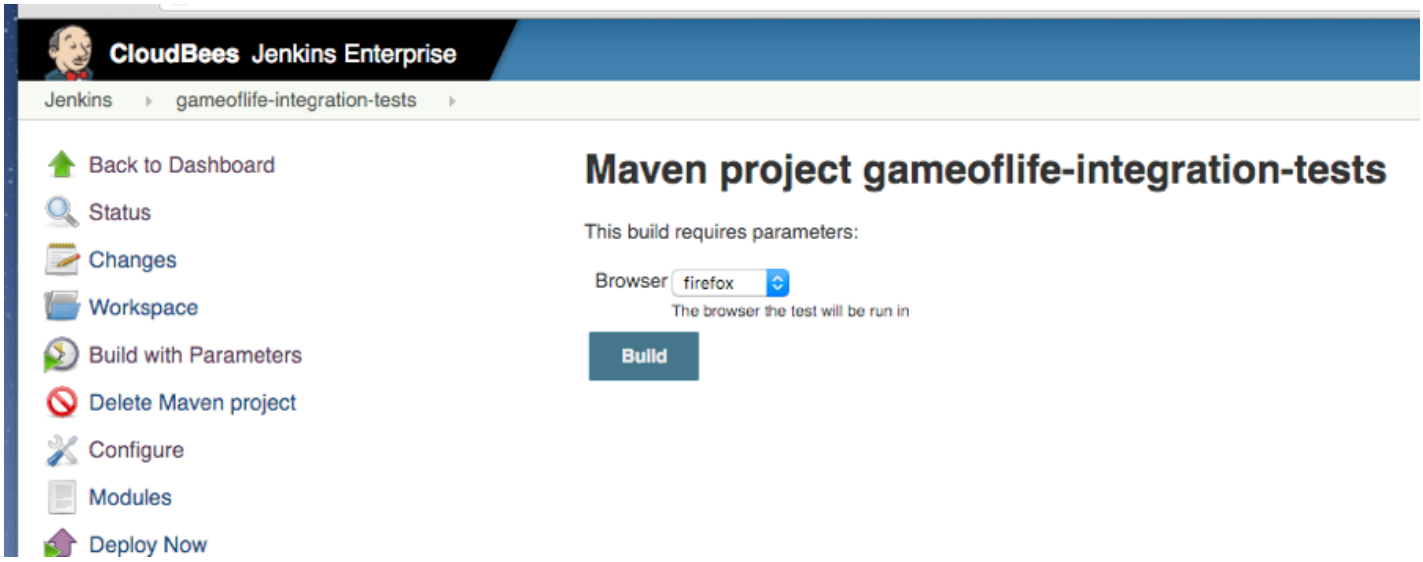


Figure 51. Launch Parameterized Job

The tests should now run on the browser you've chosen.

Go back to slides

Lab 9. Automatic deployments to Tomcat

Goal

In this lab, you will learn how to automatically deploy snapshots to Tomcat.

Step 1: Automatically deploy a SNAPSHOT web application to Tomcat

We are going to modify the **gameoflife-default** build job so that it automatically deploy the gameoflife web application to a Tomcat Application Server after each build.

There is an instance of Tomcat running at http://localhost:5000/production which we will use as a test server.

Open the configuration screen of the gameoflife-default build job and go to the **Post-build Actions** section.

Click on the **Add a post-build action** button and then select the **Deploy war/ear to a container** option:

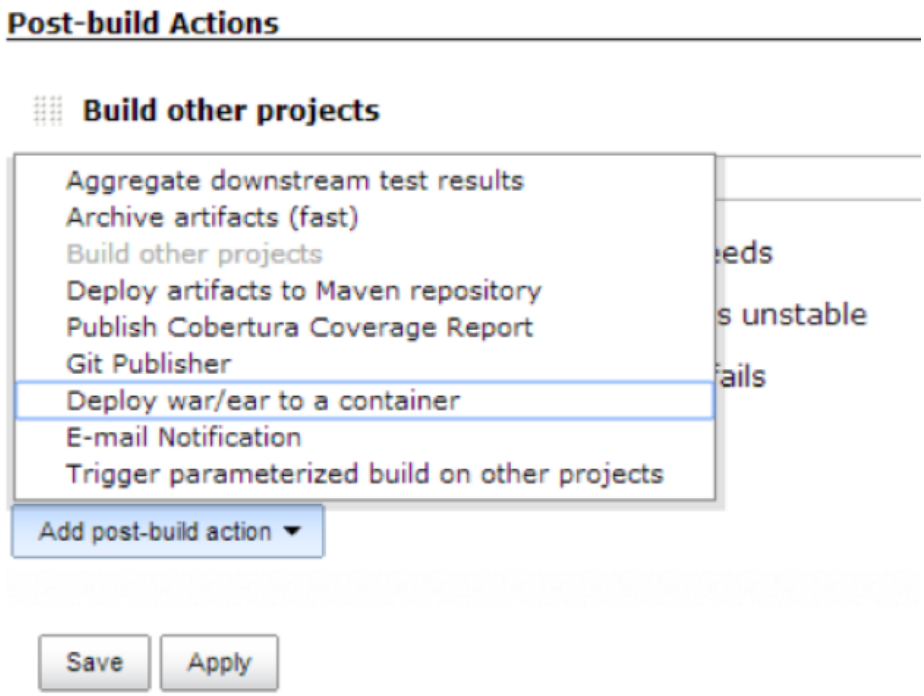


Figure 52. Adding a Post Build Step

Select **Tomcat 7.x** as your container and enter, and type in the **WAR/EAR files** field (see screenshots below):

TIP If you cannot reach the "Tomcat 7.x", add, drag and drop a 2nd post build action to you will delete afterwards. And check BlueOcean !

```
**/target/*.war
```

TIP This will deploy any WAR files produced y the builds in any "target" folder.

Now configure the Tomcat instance:

- **URL**: `http://production:8080`
  - Note this is the *private* address of the Tomcat instance manager
- **Manager username**: `admin`
- **Manager password**: `password`

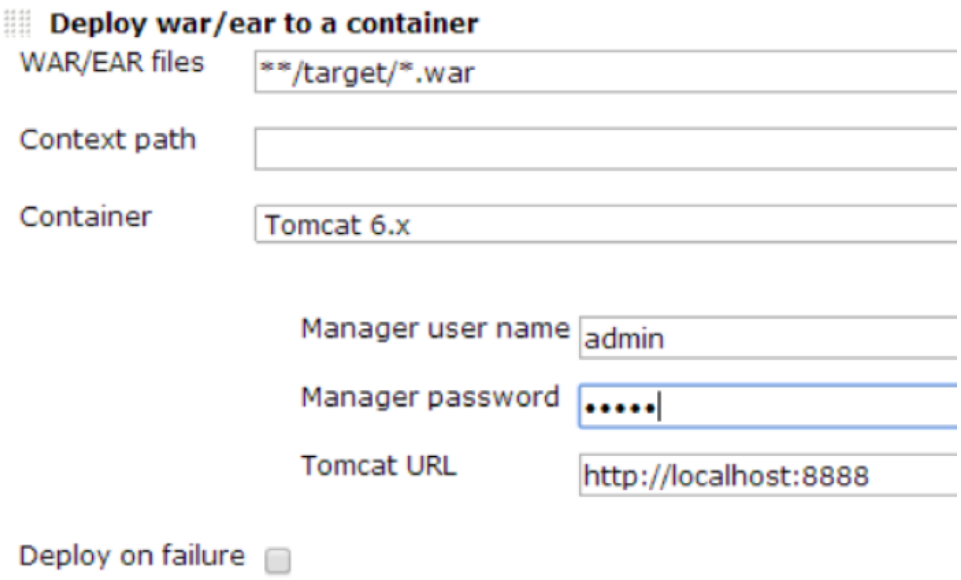


Figure 53. Deploying WAR to Tomcat - Configuration

TIP If your build fails and you receive a "401" error in your console output, this is an authentication error and was likely caused by a typo in the password or username fields.

Save the build job and trigger a manual build by hitting the **Build Now** button in the left-hand menu.

Now go to http://localhost:5000/production/gameoflife/ to see the deployed application:

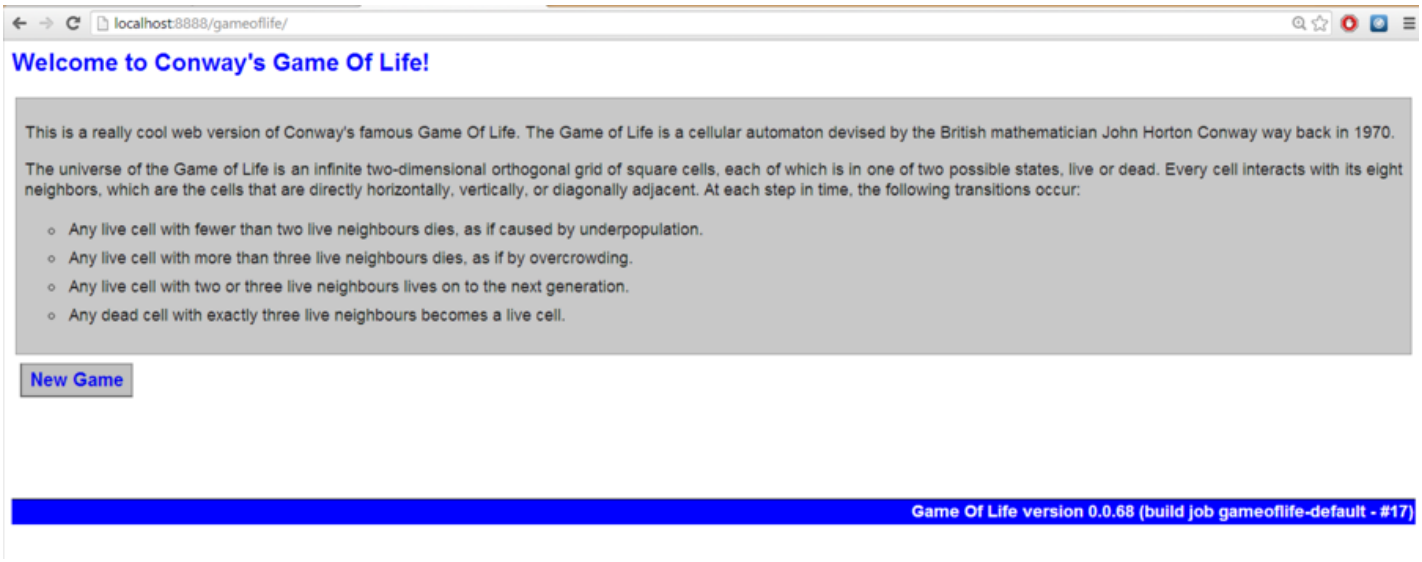


Figure 54. Application Deployed inside Tomcat

To prove that Jenkins really is deploying the latest snapshot version of the webapp, go and make some cosmetic changes to the **`home.jsp`** page, that is located within `gameoflife-web/src/main/webapp/WEB-INF/jsp/` folder:

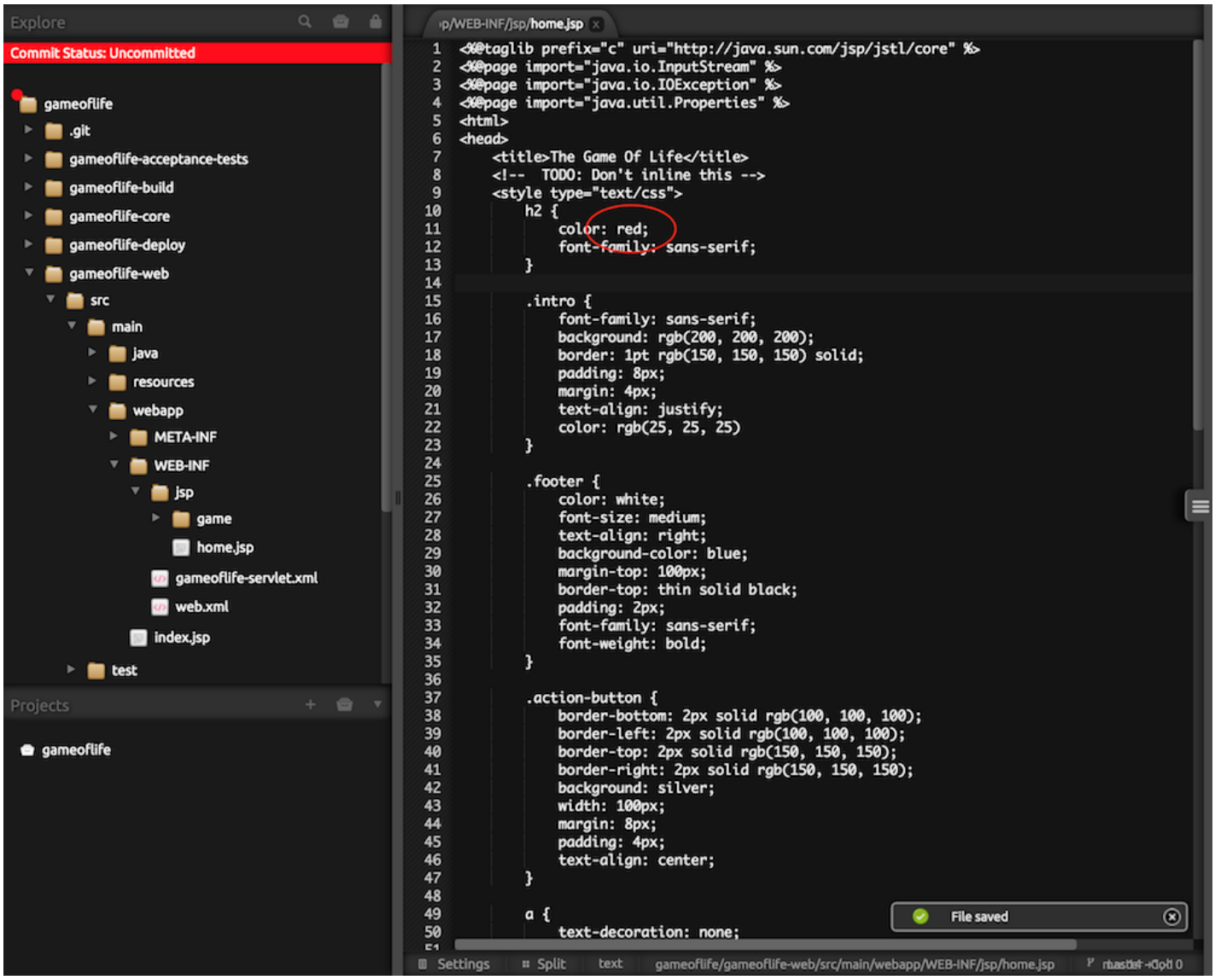


Figure 55. Changing Application Index

Save these changes and commit them to Git.

Then wait for the **gameoflife-default** build to complete and refresh the gameoflife URL.

You should now see your cosmetic changes appear.

Go back to slides

Lab 10: Job Organization and Security with Folders

Goal

This will cover how to **secure** and **organize** jobs in a Jenkins instance.

Step 1. Securing with RBAC plugin

Step 1.1 Configuring security

We will use the "Mock Security Realm" plugin for this lab. Please ensure that the plugin is installed on your CloudBees Jenkins Enterprise instance, using the "Manage Plugins" option.

TIP Direct URL access at http://localhost:5000/jenkins/pluginManager/installed

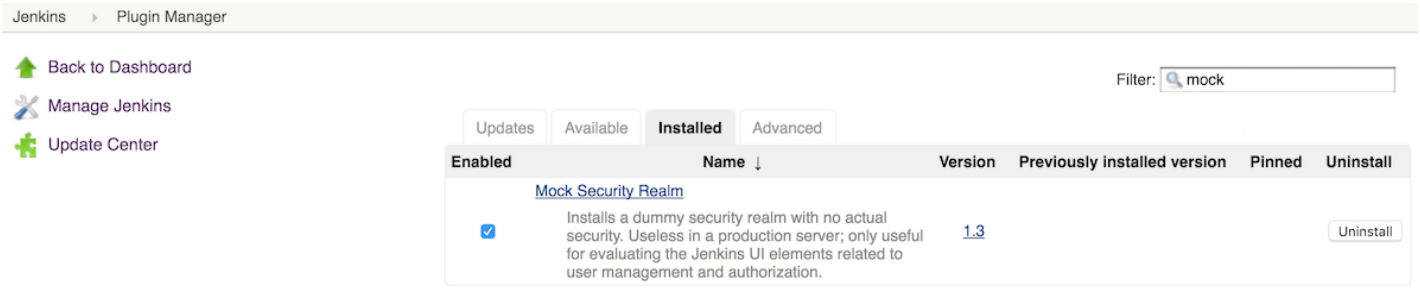


Figure 56. Plugin Mock Security Realm is installed

The first step is to enable security on the CloudBees Jenkins Enterprise master, so click on the "Manage Jenkins" link in the left-hand menu and then on the "Configure Global Security" main menu option.

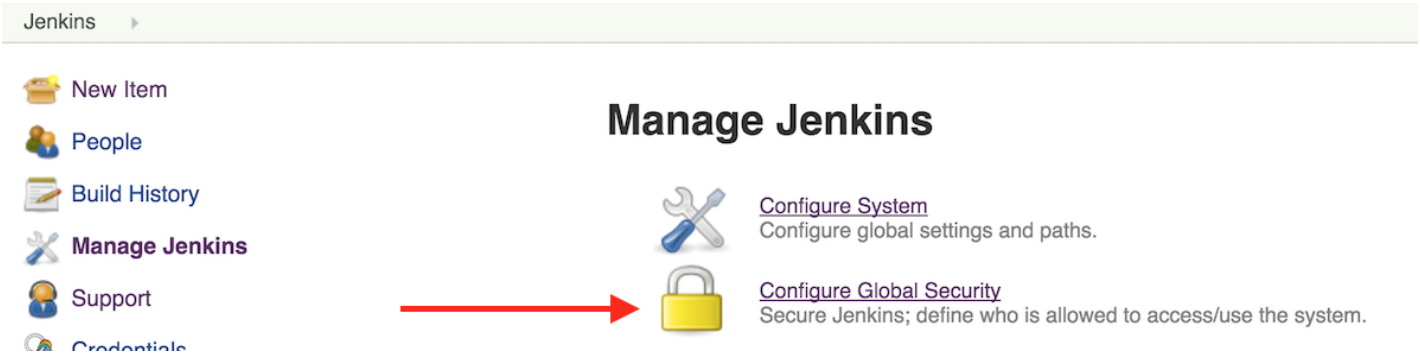


Figure 57. Access to the Configure Global Security

TIP Direct URL access at http://localhost:5000/jenkins/configureSecurity/

Configure security with those settings:

- Select the **Enable Security** checkbox
- Select the **Mock Security Realm** as the security realm.
  - Use this content for the `Users and Groups` text-field:


```
harry admin-ext
sally developer-ext
barry
```

IMPORTANT Mock Security Realm will simulate an external LDAP or Active Directory populated with this content. Left column is the username and right column is the group"s name.

TIP This will create 3 users: `harry`, `sally` and `barry`. Jenkins will also recognize `admin-ext` and `developer-ext` as "external" groups and the related users" membership

- Select **Role-based Matrix Authorization Strategy** as the authorization policy.

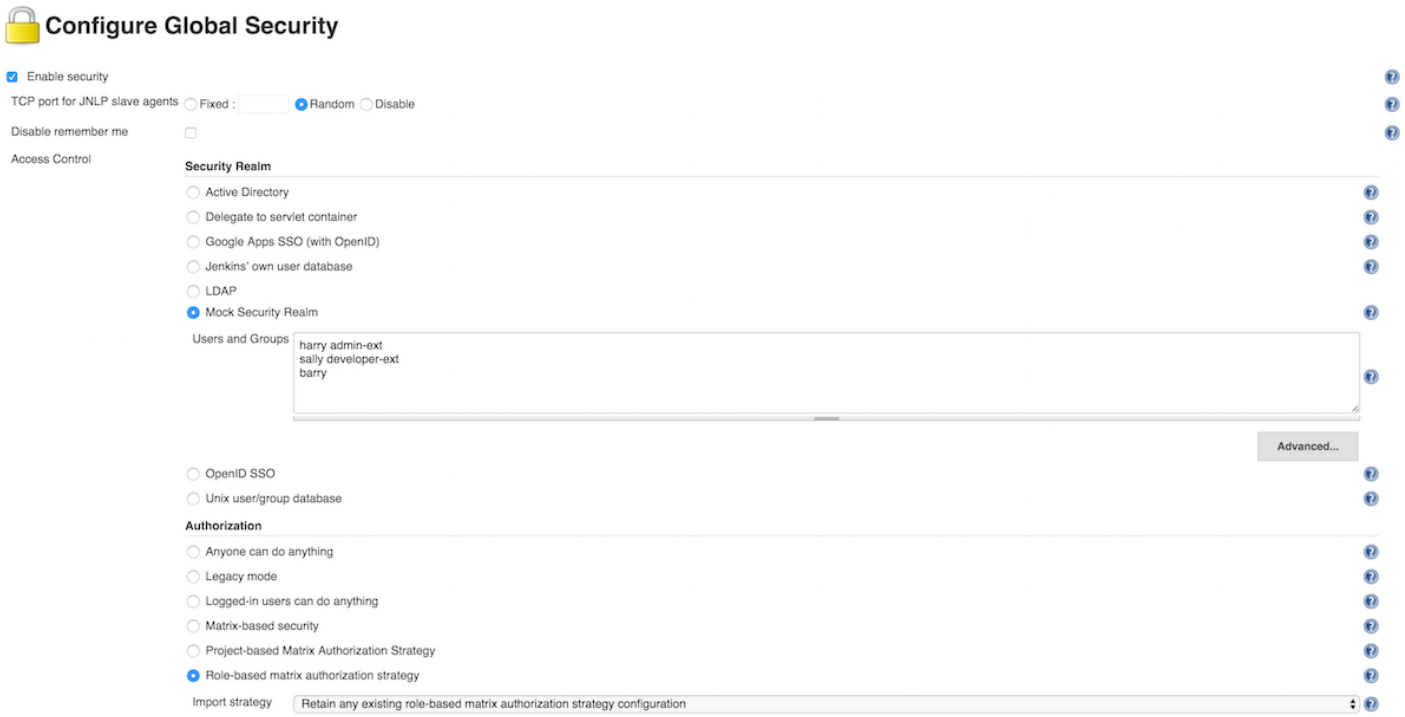


Figure 58. Global Security configuration

Now save your security settings by clicking the **Save** button and return to the top-level of CloudBees Jenkins Enterprise.

You will now see a **log in** link in the top right-hand corner, so click on that and log in as `harry`.

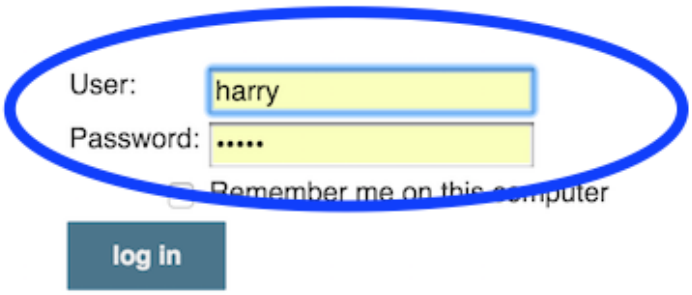


Figure 59. Jenkins Log In

TIP Password have the same value as usernames.

Step 1.2. Creating roles

Now we will need to create roles for our mock imported users. From the top-level screen, click on the **Roles** link in the left-hand menu. Now click on the **Manage** link in the left-hand menu and you will be taken to the "Manage Roles" screen.

TIP This link is now available because you selected the "Role-based Matrix Authorization Strategy" option in the last section.

TIP You have a direct-URL access: http://localhost:5000/jenkins/plugin/nectar-rbac/manage/

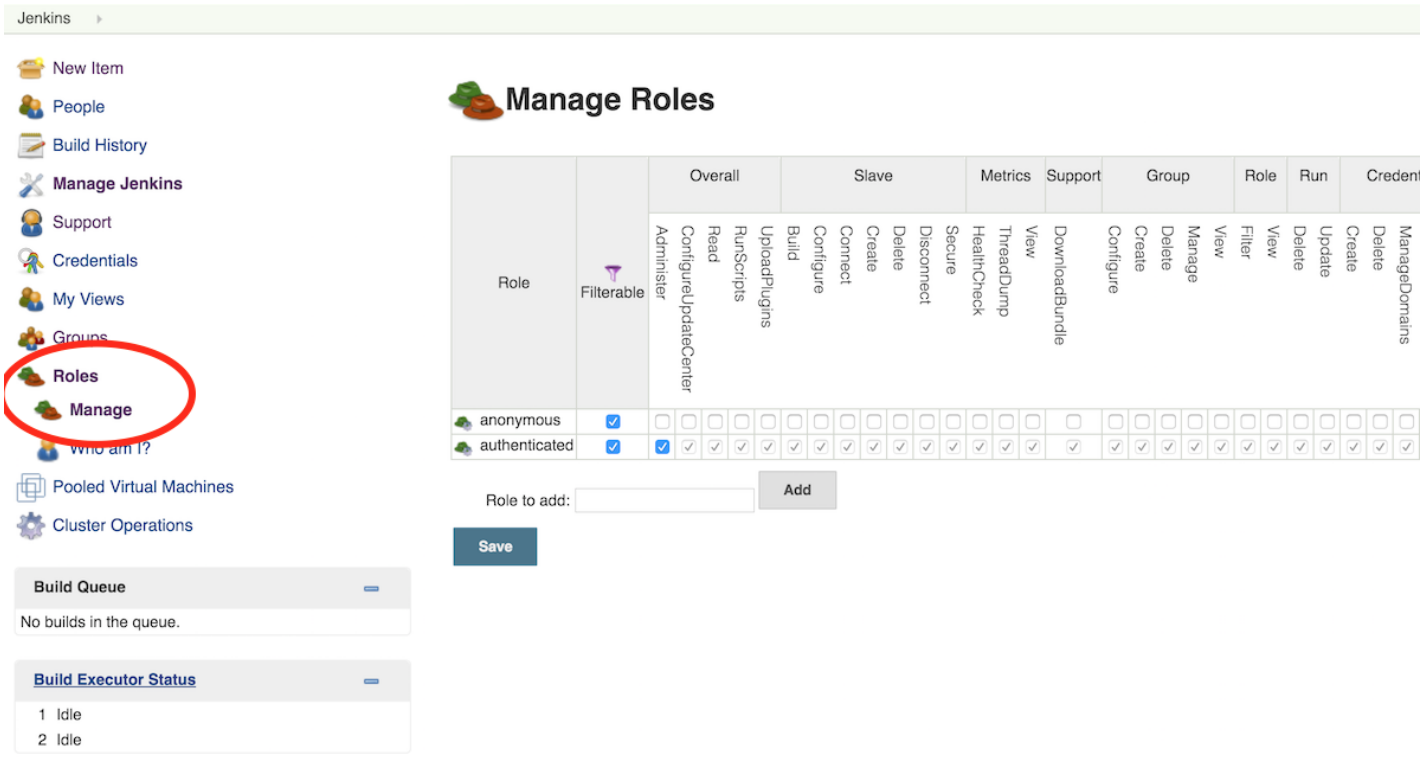


Figure 60. Accessing Manage Roles page

Our goal here is to create a security set up that has the following characteristics:

- Anonymous users have no access whatsoever, and they need to login first before even seeing the CloudBees Jenkins Enterprise top page.
- Once logged in, users have generic read access.
- Jobs can assign some users to the **reader** role that has read-only access to jobs, their test results, build results, etc.
- Jobs can assign some users to the **developer** role, who can start a new build, configure jobs, and so on.
- A few people will be in the **admin** role, which gets irrevocable full access to Jenkins.

This maps to the **two pre-defined system roles** (`anonymous` and `authenticated`) and **three additional roles we had to create** (`reader`, `developer` and `admin`).

To create those additional roles, type in the titles to the "Role to add" field on the "Manage Roles" page and then hit the add button:

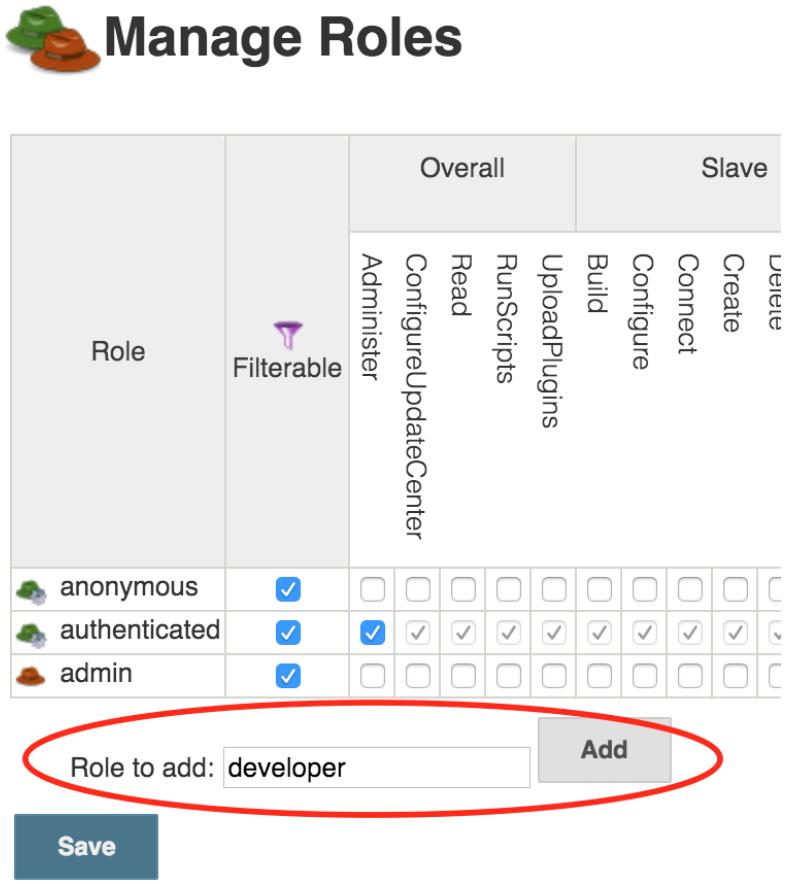


Figure 61. Quickly add a new role

By default, **filterable** property will be set to true for them all. As such, you'll now need to edit each role"s

permissions. Note that the first permissions column shows the name of the ***section*** the permission is for, while the second column shows the name of the ***checkbox*** in that section:

Table 1. Roles and Permissions table

Role	Permissions (section, checkbox)		Filterable
anonymous	None	None	Yes
authenticated	Overall	Read	Yes
reader	Overall Job	Read Read	Yes
developer	Job Job Job Job Job	Read Create Configure Build Workspace	Yes
admin	Overall	Administer	No

Once you are ready to edit the "authenticated role", note that you'll be able to uncheck all boxes at once by scrolling to the right and clicking on the  button.

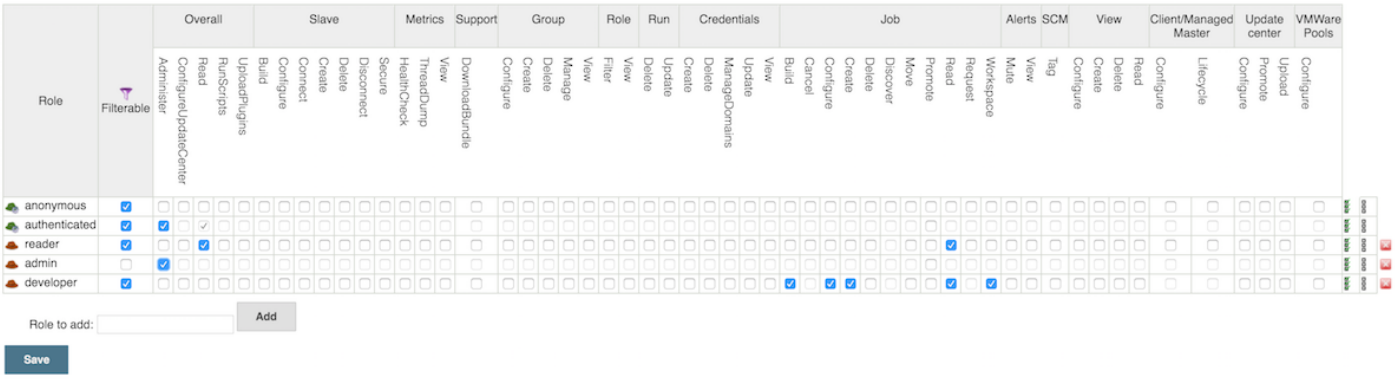
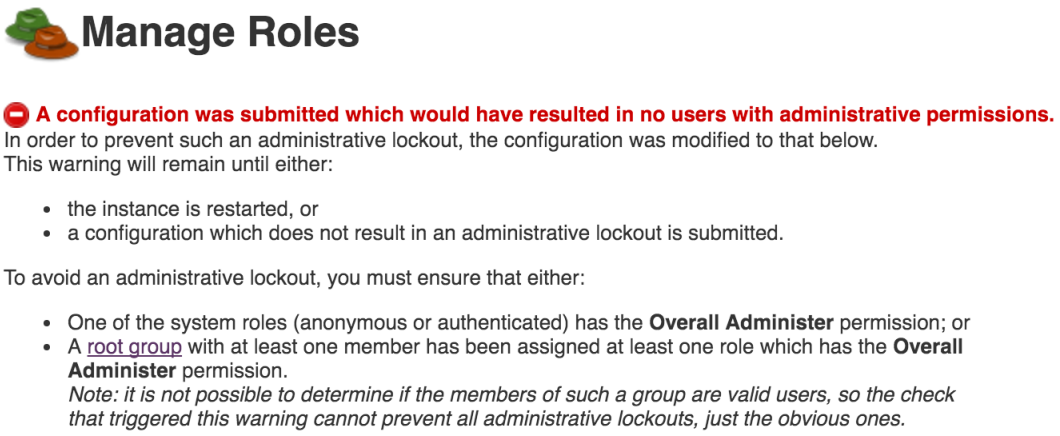


Figure 62. Resulting Permission Matrix

Try to save your configuration.

IMPORTANT

Note that the "Anti-lockout" guard will kick in and change our configuration, giving "authenticated" overall Administer permissions.



This is because the "admin" role is not yet associated with a group that contains at least one member. As such, saving the configuration as-is would have locked you out of the instance.

Now let's save these altered roles "as-it" by clicking the "Save" button at the bottom of the screen.

Step 1.3. Creating groups

The next step is to create groups that connect users to the roles, and populate these groups. We will create 3 groups:

- `Developers`
- `Administrators`
- `Browsers`

Click on the "Groups" link in the left-hand menu, which will take you to the "Groups" list. Then, click on the "add some?" link to start creating groups (see below for settings).

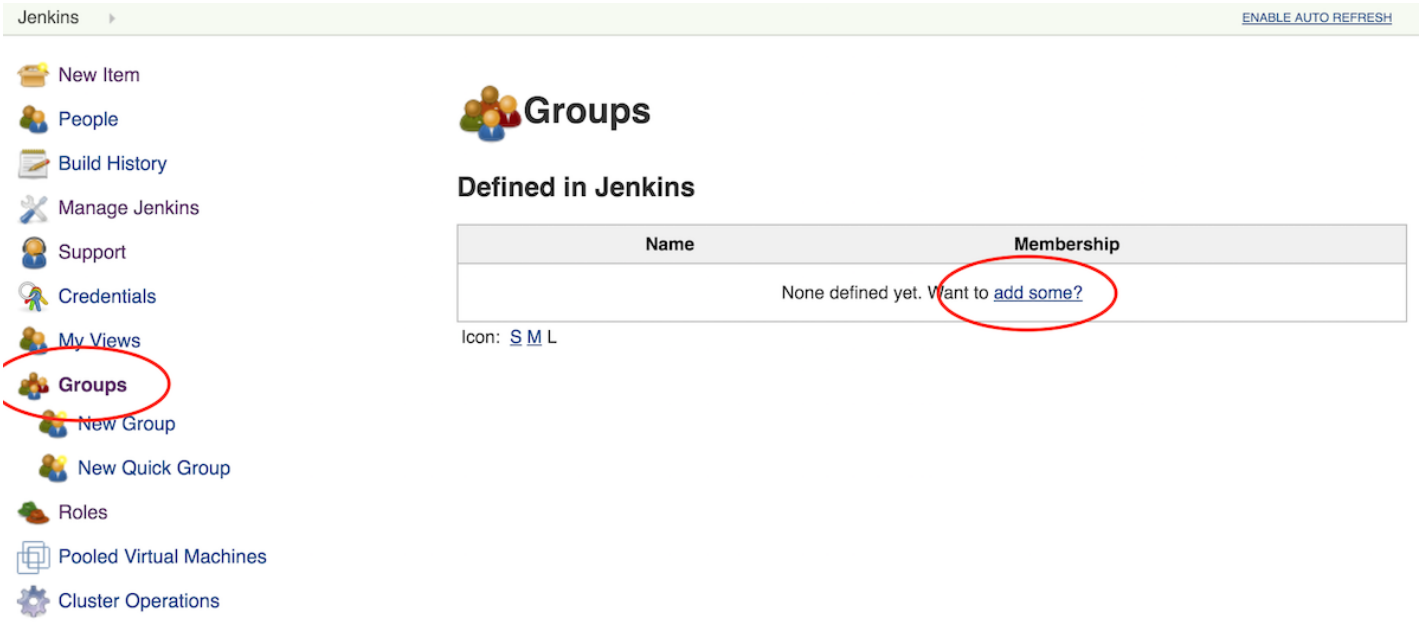


Figure 63. Access to group creation

We want to create groups to represent each role we have created, and we also want to add both our imported users *and* our imported groups as members of each group.

Table 2. Groups final configuration

Group Name	Roles	Propagated	Members
Browsers	reader	Yes	barry, harry, sally
Developers	developer	Yes	sally, developer-ext
Administrators	admin	Yes	harry, admin-ext

For example, "Developer" would be configured like this:

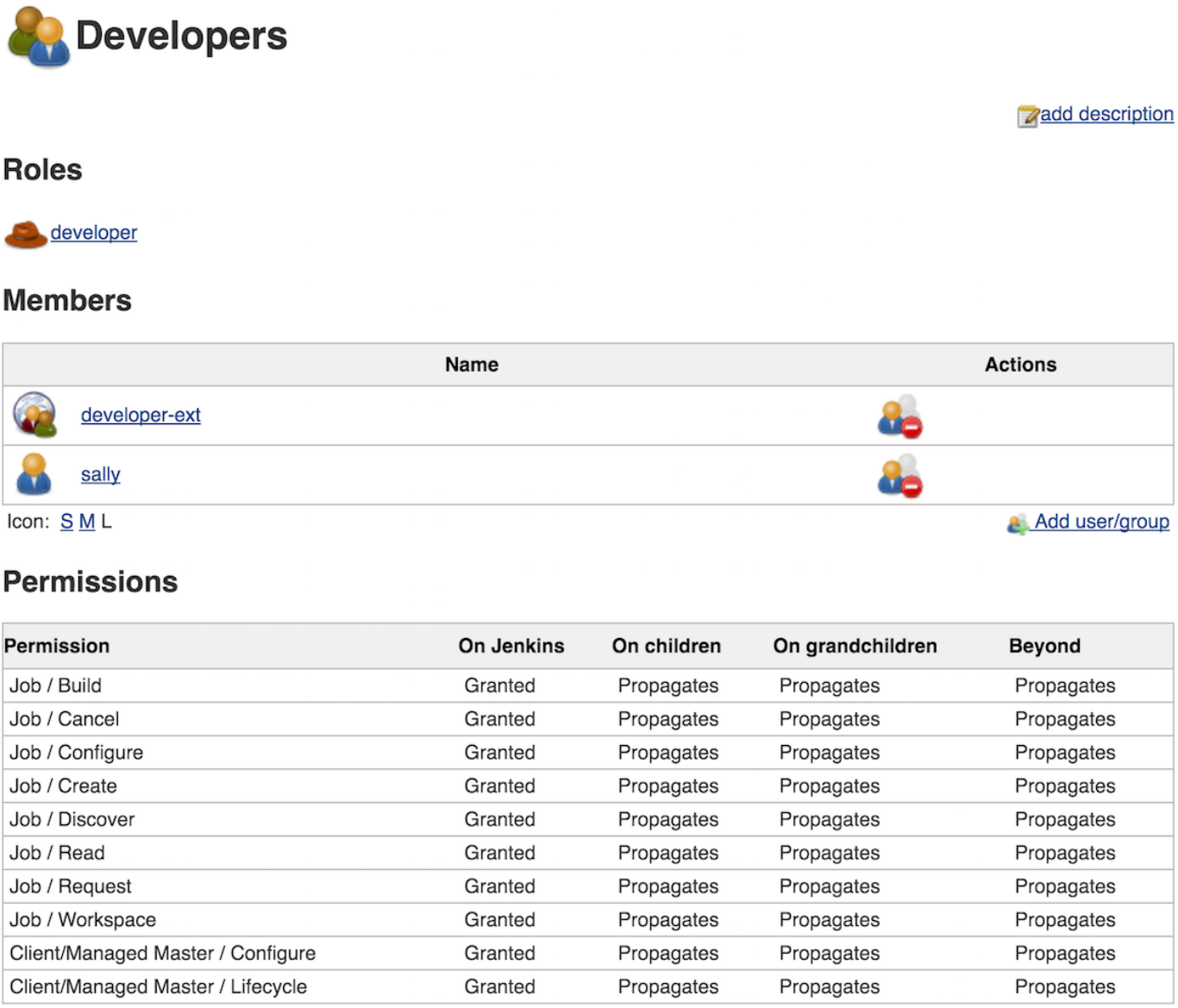


Figure 64. Developers Group configuration

TIP

Once you have created a group, use the Add user/group button to add users and sub-groups to the current group.

If you want to create another group, hit the Back to Groups button in the left-hand menu and then hit the "New Group" button

Repeat this process until all 3 groups have been created

Step 1.4. Completing configuration of the roles

Now that we have a group with the Overall/Administer permission, we can complete the configuration of the roles.

Go back to the **Manage Roles** screen (**Manage Jenkins** , **Manage Roles**). De-select the Overall/Admin permission for the **authenticated** role and ensure that the Overall/Read permission is still on selected.

Now save the configuration. Since the **admin** role is now associated with a populated group, you should not receive an **Anti-lockout** warning.

Step 1.5. Single sign-on

Ensure that the users **harry** and **sally** exist and that the current logged user is **harry**.

Step 2. Organizing with Folders Plus Plugin

Step 2.1. Creating Folders

The first step is to create a folder, which is a dashboard-level object in Jenkins.

To start, log into CloudBees Jenkins Enterprise and click on the **New Item** link in the left-hand menu.

Now select the **Folder** type for the job and name the folder `Harry's Project`:

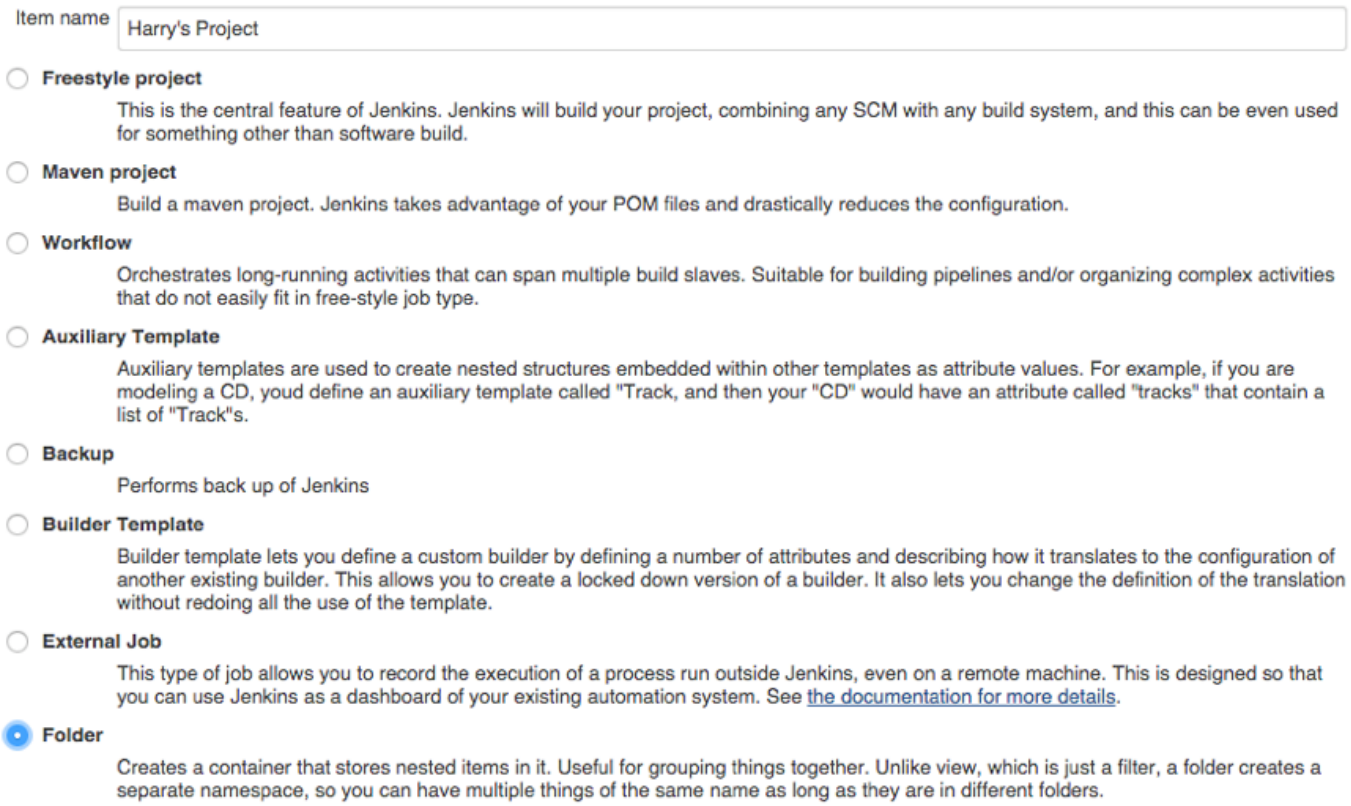


Figure 65. Creating a New Folder

Once this is done, let the default configuration fields and click **Ok** to save.

Now at the top page or dashboard-level of Jenkins, you should see the **Harry's Project** folder in the default view, along with any other configured jobs:

Figure 66. Harry's Project in Default View

To edit the configuration of this folder, either click on the folder and the **Configure** link in the left-hand menu or click on the black arrow to the right of the folder's name and the **Configure** link in that drop-down menu:

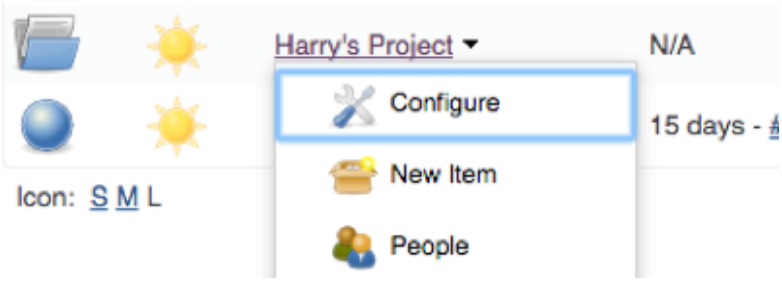


Figure 67. Configure Harry's Project from Default View

The Folders plugin allows users to organize jobs into folders, but the **CloudBees Folders Plus** plugin expands this functionality to also allow folders to:

1. control agents
2. show an overall health report for jobs in a folder
3. display custom folder icons
4. contain environment variables, which are accessible to jobs
5. restrict the content of folders to certain item types
6. inherit global roles and group permissions, or have a custom group with folder-specific permissions

Step 2.2. Create security on folders

To create a custom set of permissions only on this folder, we will need to create a new group for this folder.

To do this, click the **Groups** link in the left-hand menu of the **Harry's Project** folder configuration screen:

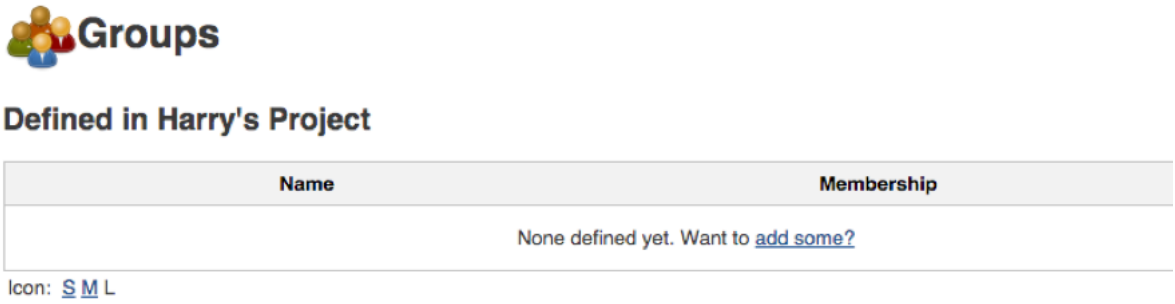


Figure 68. Harry's Project Group Configuration

As you can see, there are not yet any groups configured on this folder, but the **Folders Plus** plugin is giving us the option of creating one. In the **Inherited from Jenkins** section below, you will also see the names and members of all groups configured at the root level of Jenkins.

For now, just click either the **Add some?** link in the Groups list of the **New Group** link in the left-hand menu of this group configuration screen.

We are now creating a new group specific to **Harry's Project**, so on the next screen configure the Group name to be `ProjectDevelopers` and click **Ok**.

You'll see a screen prompting for a description and assign to the group any role configured at the root level of Jenkins. Since this is a developers group for this folder, select the **developer** role and click **save** to continue.

On the next page we can add group members, so for now add **harry** and **barry** to this group:

Figure 69. Harry and Barry are in the Project

Normally, permissions granted closer to the root gets inherited through to folders and jobs in them. When we activate a **require explicit assignment** filter, we are essentially saying that users explicitly need to be assigned to a group in this folder to be able to get a role.

Now go back to the **Harry's Project** folder and click on the **Roles** link in the left-hand menu, then click the **Filter** link and mark all roles as requiring explicit assignment:

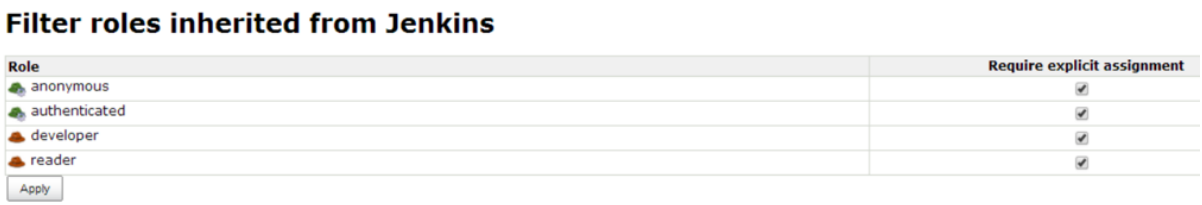


Figure 70. Filter Roles

In practice, this means that **sally**, who has the developer role to the root, will not get the developer role in the **Harry's project** folder unless she is explicitly added to the **ProjectDevelopers** group.

barry, who only has the authenticated role in the root has been given explicit permission on the **Harry's Project** folder as a developer, and so should be able to have just as many permissions on this folder as **harry**.

Logout as **harry** and login as **barry**. Verify that you can create and build jobs, but only in the **Harry's Project** folder and that any other jobs in the root level of Jenkins are hidden from view.

Logout as **barry** and login as **sally**. Verify that you cannot see the folder **Harry's Project**. Try to access the project directly using the URL, e.g. http://localhost:5000/jenkins/job/Harry's%20Project/ and confirm that you get a 404 error page.

Now logout as **sally** and log back in as *harry.

Step 2.3. Move jobs into folders

Since we just created **Harry's Project**, this folder is empty. To populate it with jobs, lets create jobs into the root of Jenkins and move them into this folder.

Go back to the top page of Jenkins, either by click on the CloudBees Jenkins Enterprise banner or the **Jenkins** breadcrumb in the navigation bar.

Now create 3 jobs of any type (freestyle, Maven, Pipeline, etc) and name them anything while leaving their configurations blank.

Once done, click on either the job's name or the black arrow to the right of the job's name.

In either menu, there will be an option called **Move** with a icon of a dolly.

Select the **Move** option and select **Jenkins** - > **Harry's Project** from the dropdown menu as the destination for your job.

Repeat this for all of the dummy jobs you just created.

Now open up the **Harry's Project** folder: all jobs that you just moved should now be listed there.

Go back to slides

Lab 11: Validated merge

Goal

In this lab, you will learn how to set up Jenkins for validated merge with Git as an administrator, then how to perform actual validated merge as a developer.

Step 1: Fork project in Gogs as "dev" user

We will use the Gogs service for this exercise. A new "user" is required: **dev**, different than `harry` This new user will fork `harry's` project and change things inside, to have a different workflow than before:

- Navigate to the Gogs service, on http://localhost:5000/gitserver
- Create the **dev** user by clicking the **Register** link in the top right, and fill the form like this:
  - **Username**: `dev`
  - **Email**: `dev@localhost.local`
  - **Password**: `dev`

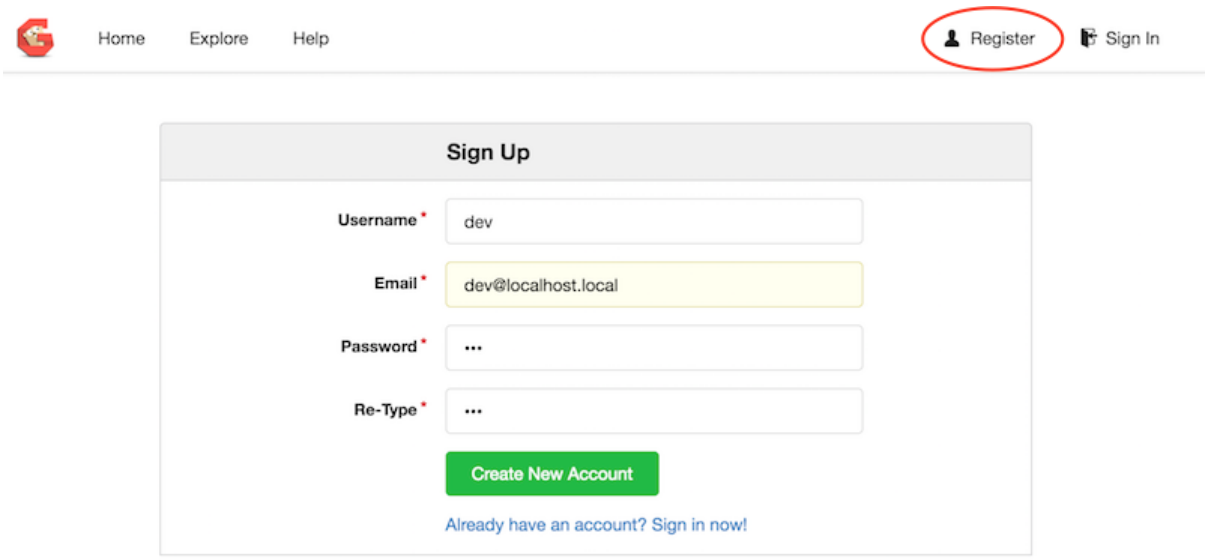


Figure 71. Register as dev

- Then, sign-in as **dev** user, to grant the right of forking the project:

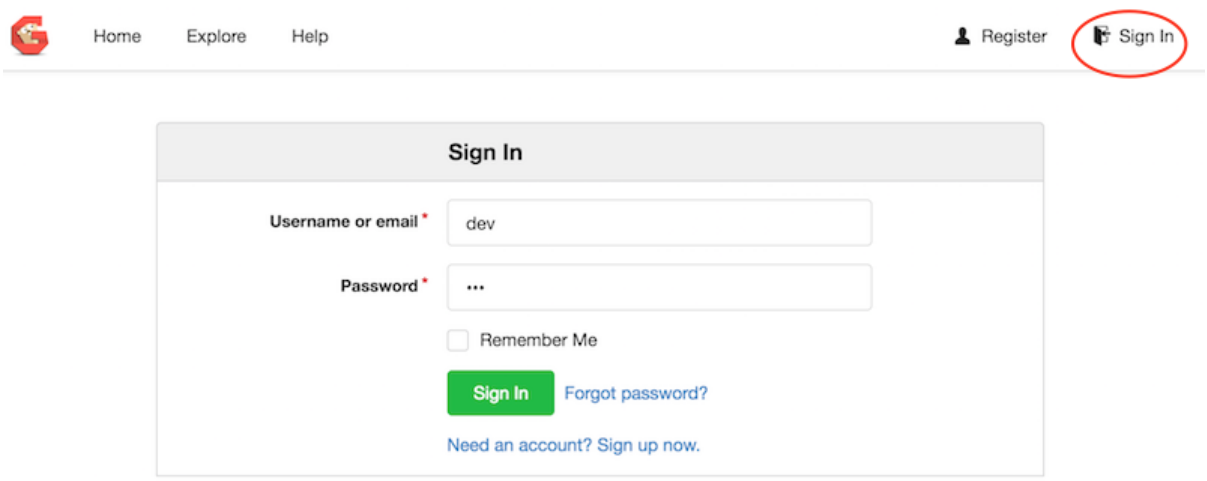


Figure 72. Signing-in as dev

- Once you are logged-in under your GitHub account, browse to the **harry**'s project: http://localhost:5000/gitserver/harry/gameoflife
- Then, create a fork of this project in your "dev" account namespace, that we will use for this Labs:

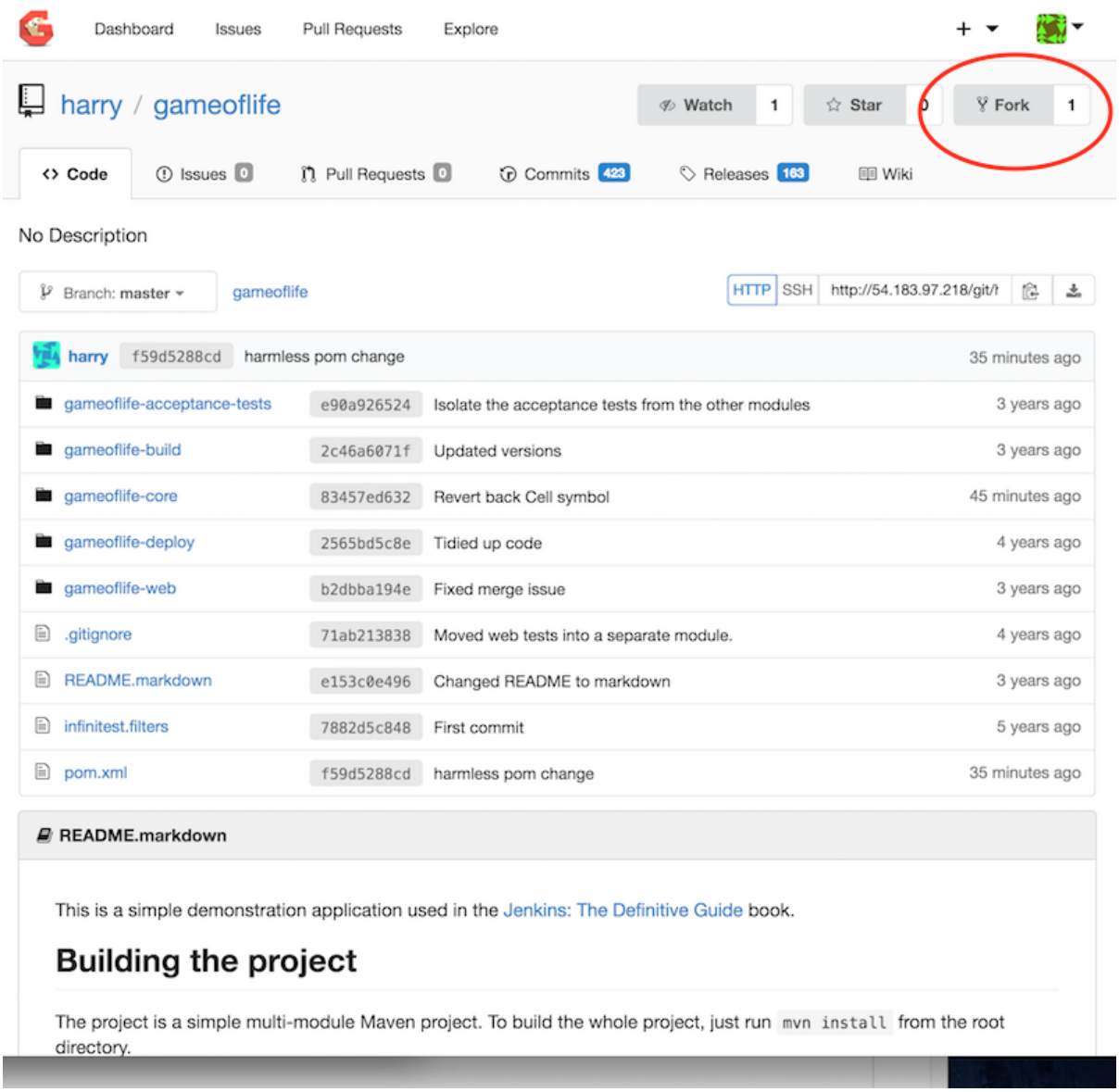


Figure 73. Fork the Project in Gogs

- When asked, leave the default options:

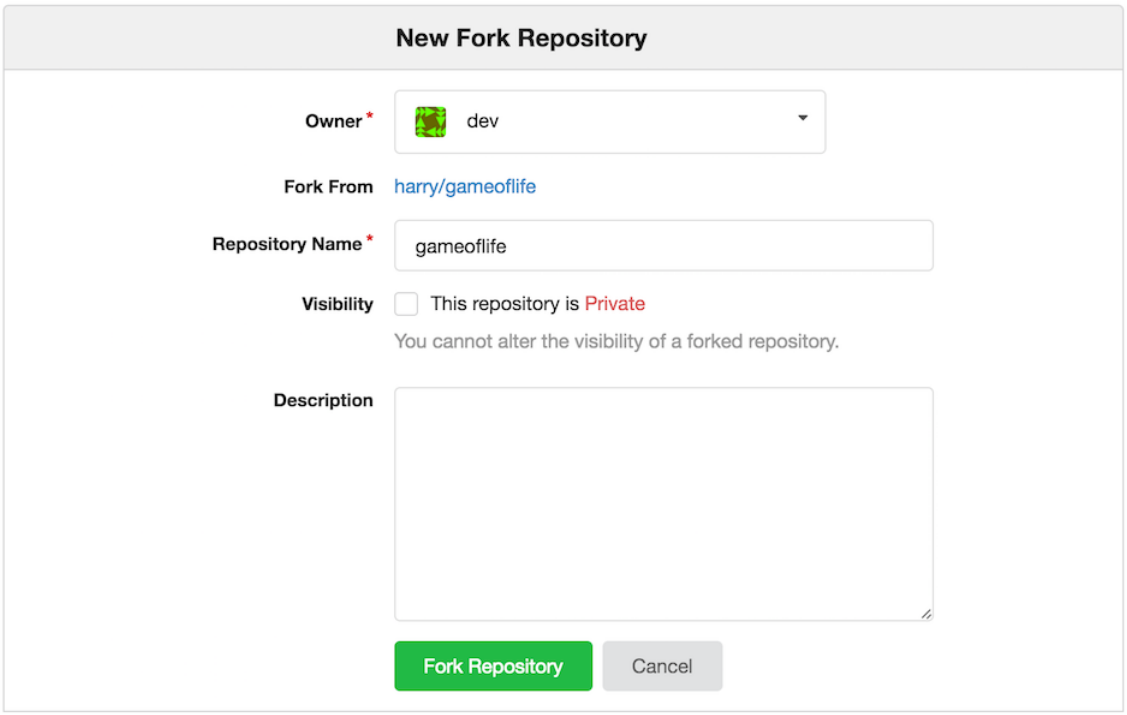


Figure 74. Fork options

- Write down the **HTTP** URL of the newly **forked repository**:

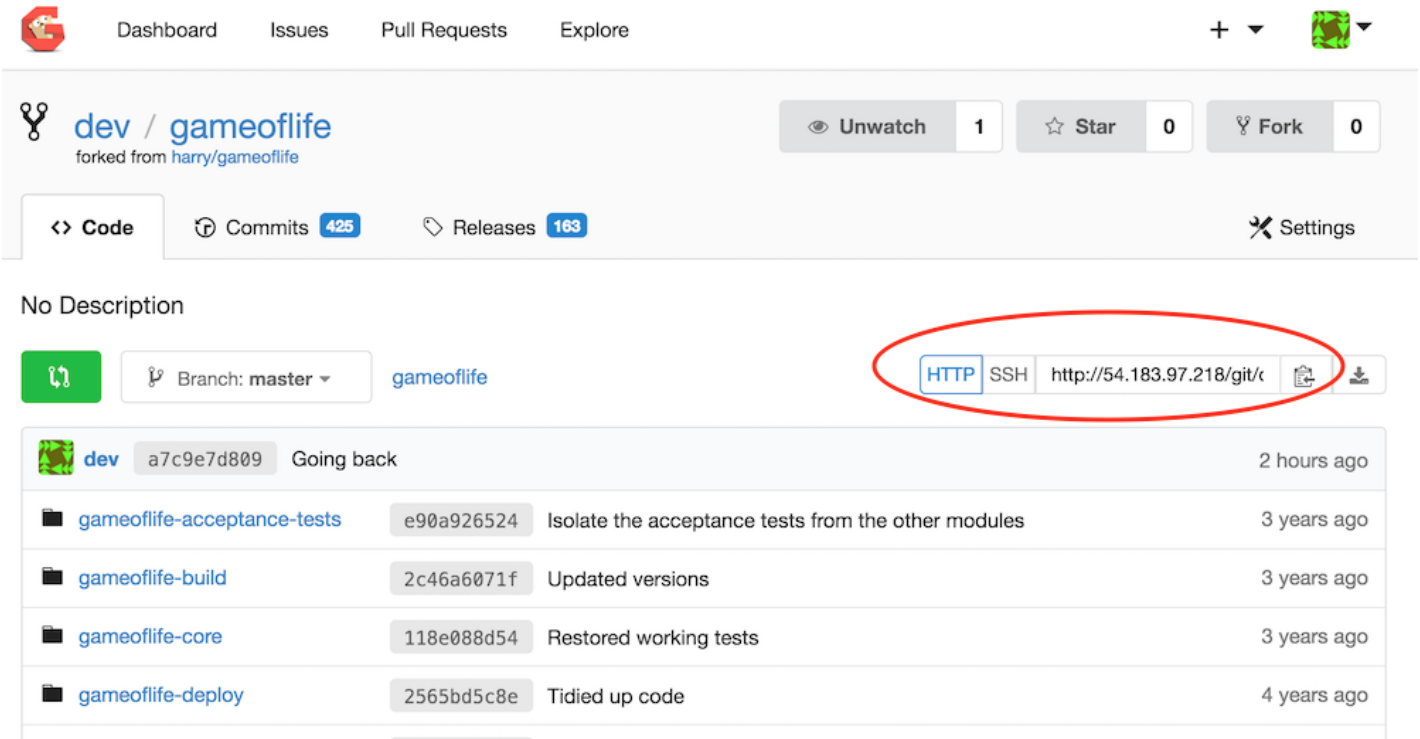


Figure 75. Get the Fork HTTP URL

Step 2: Global Jenkins Administration Setup

Navigate to the Jenkins global settings (**Manage Jenkins** - **Configure System**) and check those global

settings:

- In the **GitHub Validated Merge** section, enable the **Prefer HTTP URLs** option:

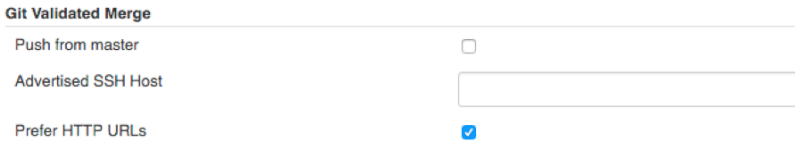


Figure 76. Global Configuration for Validated Merge

- Finally, configure the internal SSH server to use a fixed port. We propose to use **8000** :

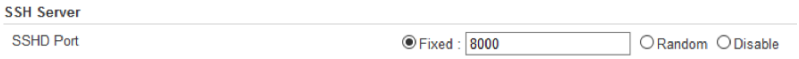


Figure 77. SSH Configuration for Validated Merge

Save those configuration, and navigate to the Jenkins global credentials (**Credentials** item in the left menu):

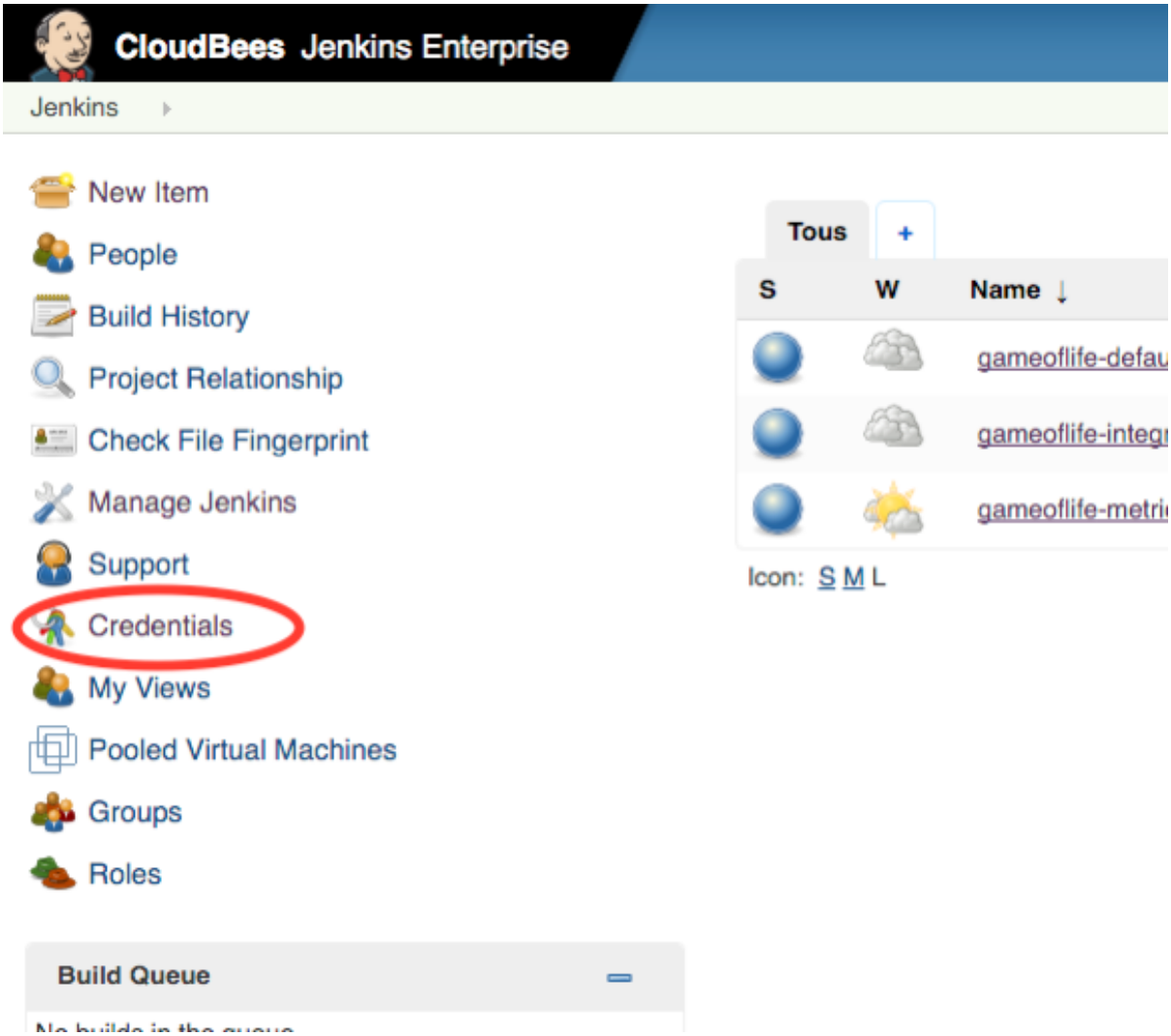


Figure 78. Access Credentials from Main Page

Browse to the **Global Credentials** domain:

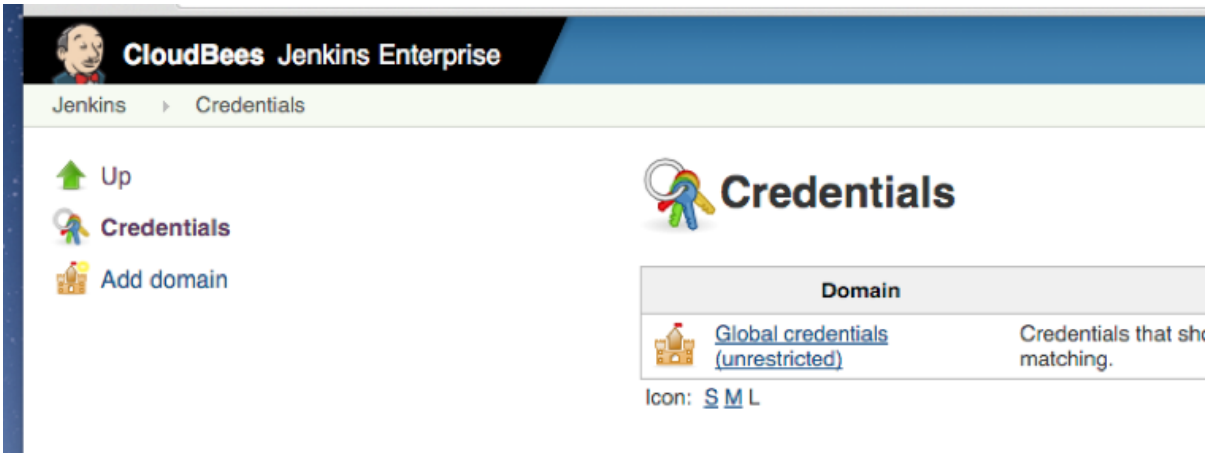


Figure 79. Global Credentials

And add a new credentials for the **dev** user.

TIP The password for `dev` user is `dev`. Simple isn't it ?

Step 3: Setup a Jenkins job for CI build & validated merge

Let's pretend that you are a team lead and set up a Jenkins job that does validated merge.

Go to the Jenkins top page, click on the **New Item** menu item, and choose "Freestyle project".

Call this project `gameoflife-personal` and click **OK**:

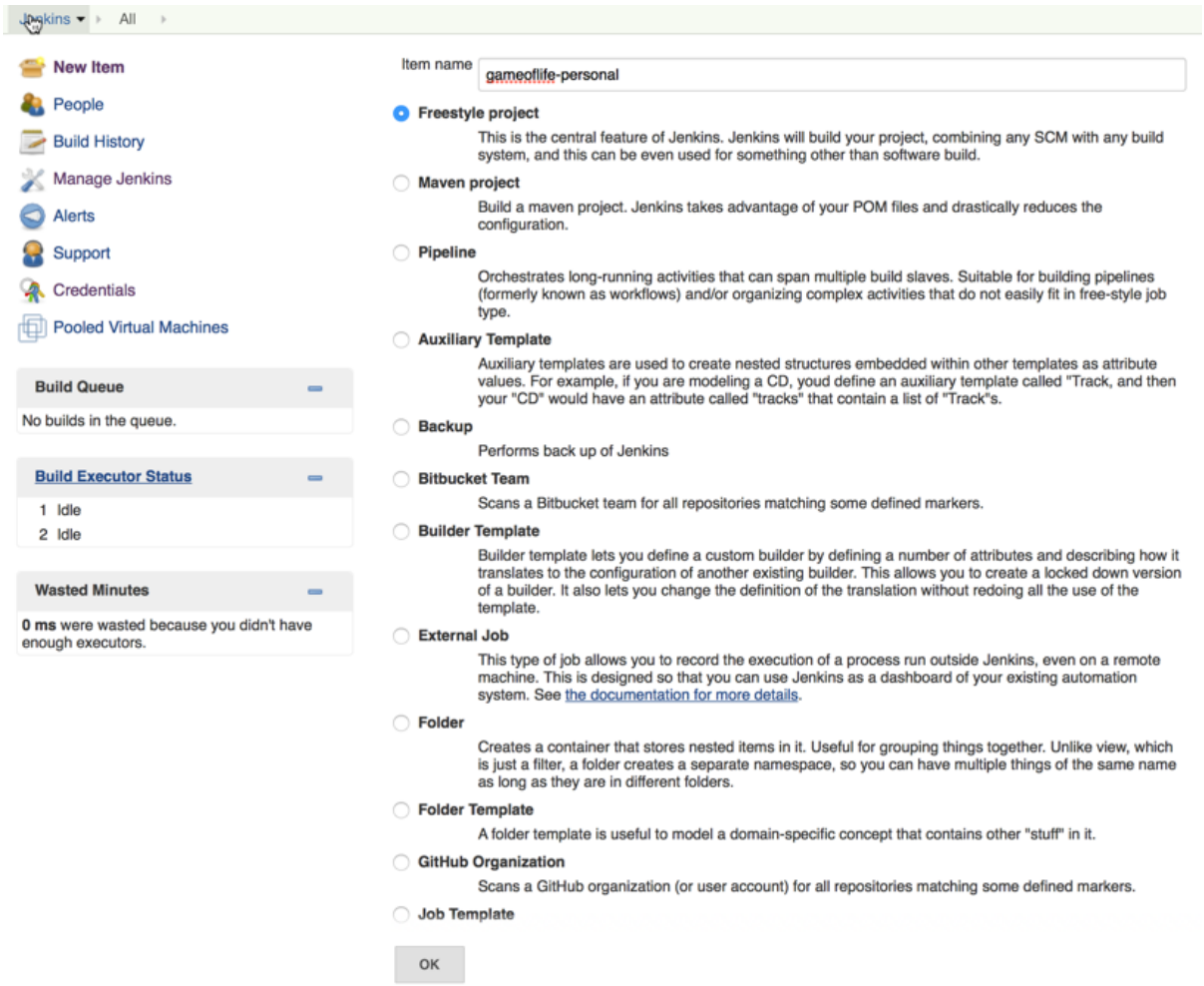


Figure 80. Create a new Freestyle Job

In the job configuration screen :

- Select **Enable Git validated merge support** to enable the feature, with those settings:
  - Select you git credentials (`dev` user) from the list
  - Specify an **Alternate Push URL**: http://gitserver:3000/dev/gameoflife.git

TIP The alternate URL use the "internal" address of the git server.

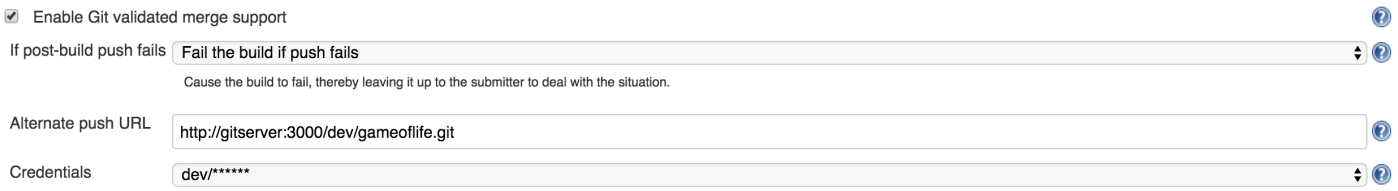


Figure 81. Job configuration for validated merge

- Specify the repository URL in the Source Code Management section as usual:
  - URL of the repo: `http://localhost:5000/gitserver/harry/gameoflife.git`

- Don't forget to use the new **dev** credential !

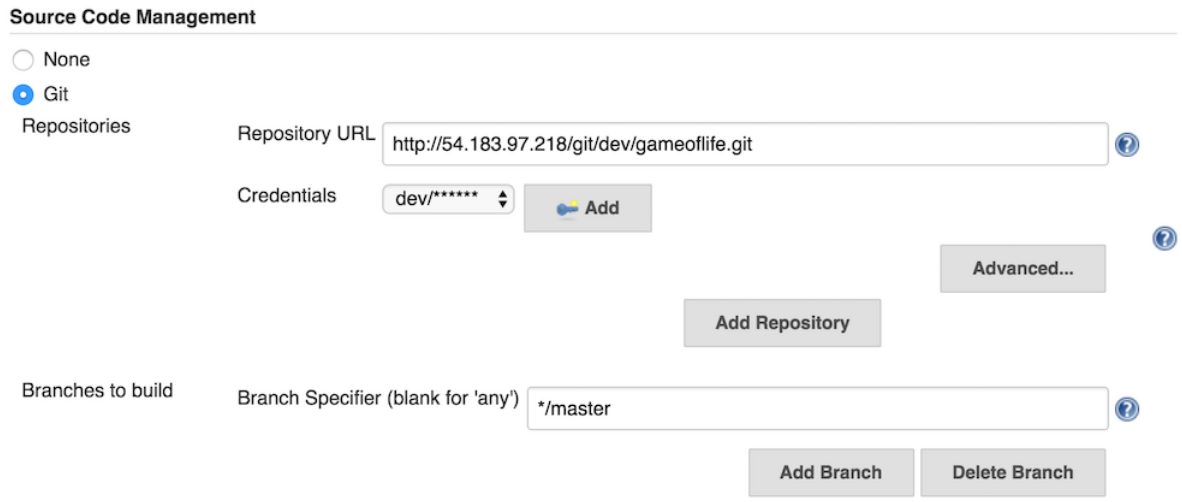


Figure 82. SCM Configuration

- In the main build section of the configuration page, add **Invoke top-level Maven targets** and then type in **install** as the Maven goal:

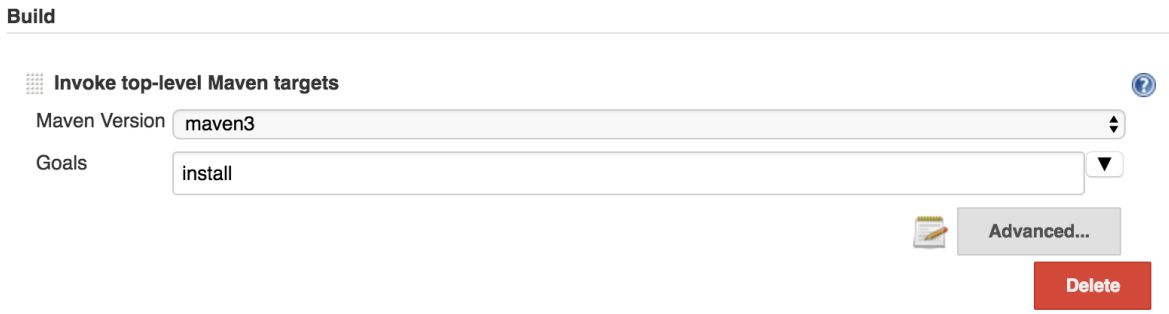


Figure 83. Maven Configuration

Click the **Save** button to apply this configuration. You can review the configuration is valid by checking that your project is now linked to the **Git Repository for Validated Merged**:

Figure 84. Validated Merge options for our Job

If you'd like, you can click **Build Now** to make sure this set up is correct. That completes the team lead's work.

Step 4: Edit the code and test the change

As a developer, let's make changes to the source code.

- First, let's login again in the WebIDE as **dev** user (instead of `harry`), using the right panel to logout:

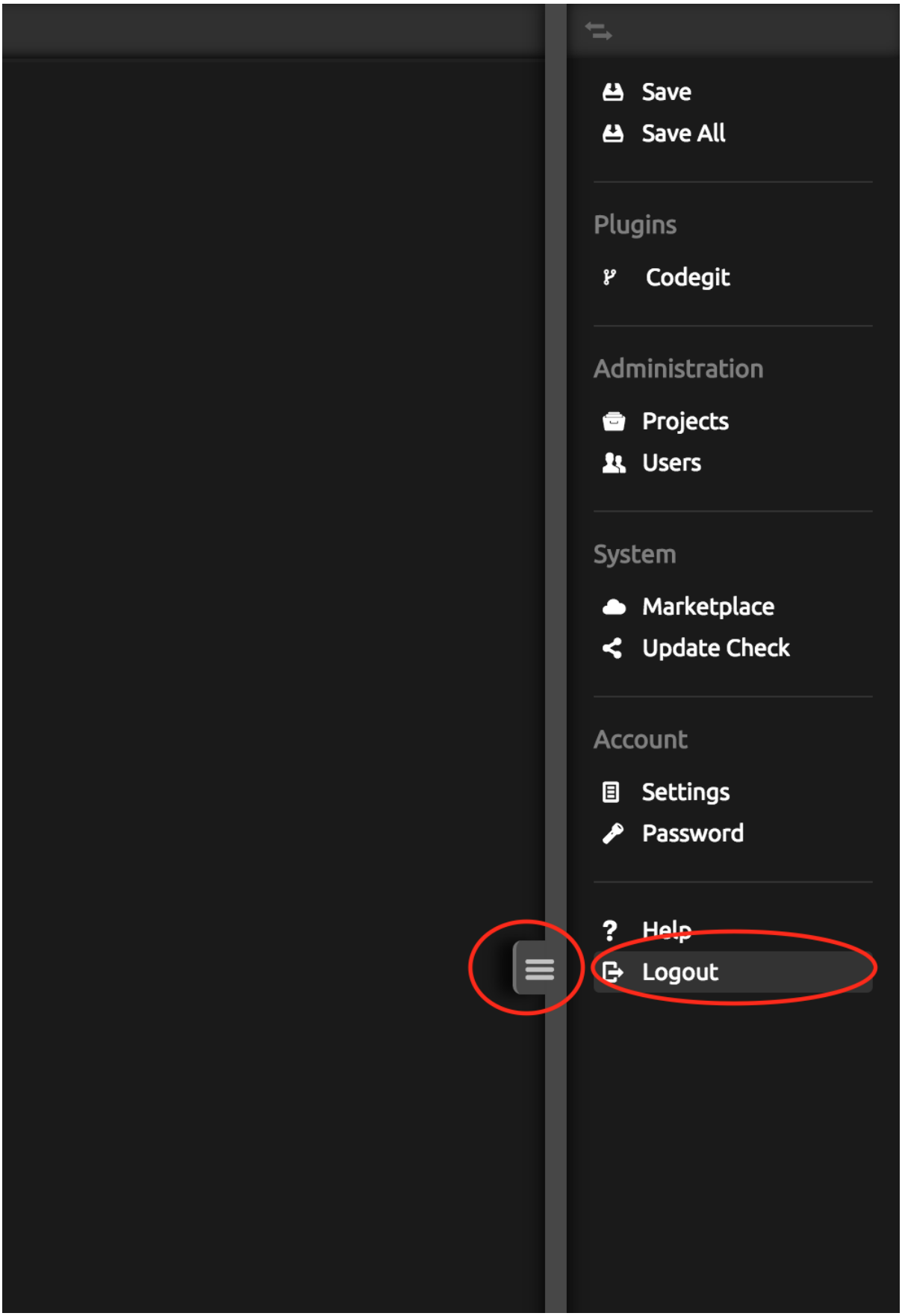


Figure 85. Logging out of the Web IDE

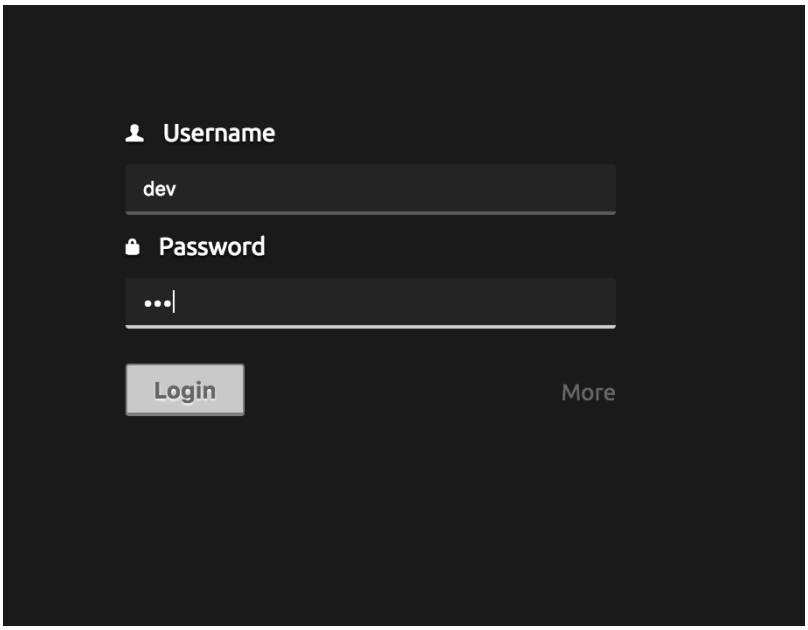


Figure 86. Logging in as dev user in the Web IDE

Then, checkout the code of the fork in the Web IDE:

TIP Use the repository URL that you copied from Gogs at the end of Step 1 It should be an URL like this: http://localhost:5000/gitserver/dev/gameoflife.git

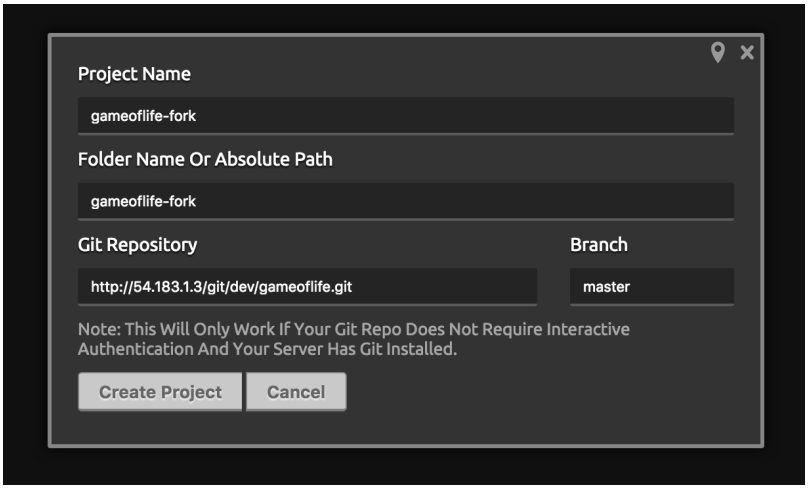


Figure 87. Checkout the code of the fork

Let's play a scenario where **dev** unknowingly breaking a build by a change. Any chance would do, but for example, edit

```
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/Cell.java
```

and make it un-compilable by adding garbage (say `!!!`) at the top of the file. Commit this change as follows.

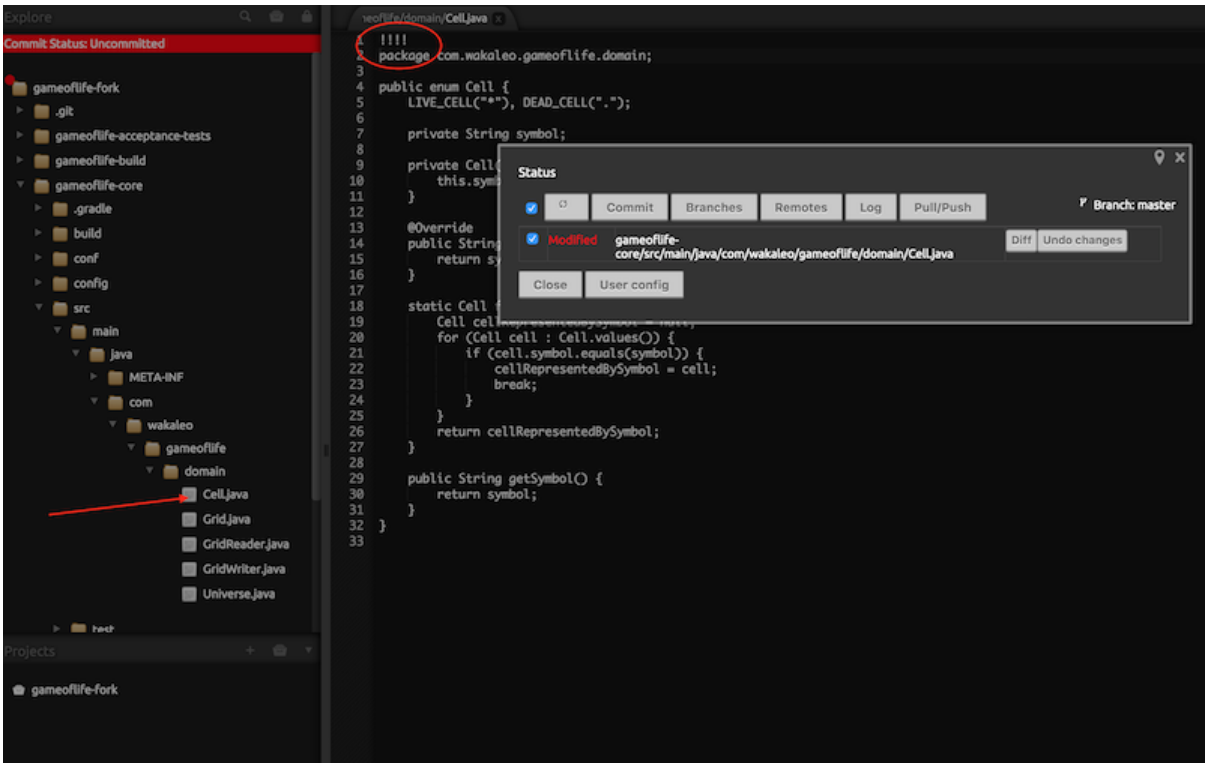


Figure 88. Breaking the Cell class

Figure 89. Committing the breaking change

IMPORTANT Because Git is distributed,at this point this change is not exposed to the team repository yet.

To have this change tested by Jenkins, click the **Git repository for validated merge** menu item from the **gameoflife-personal** project page, then click the clipboard icon on the right to copy the command line that needs to be executed:

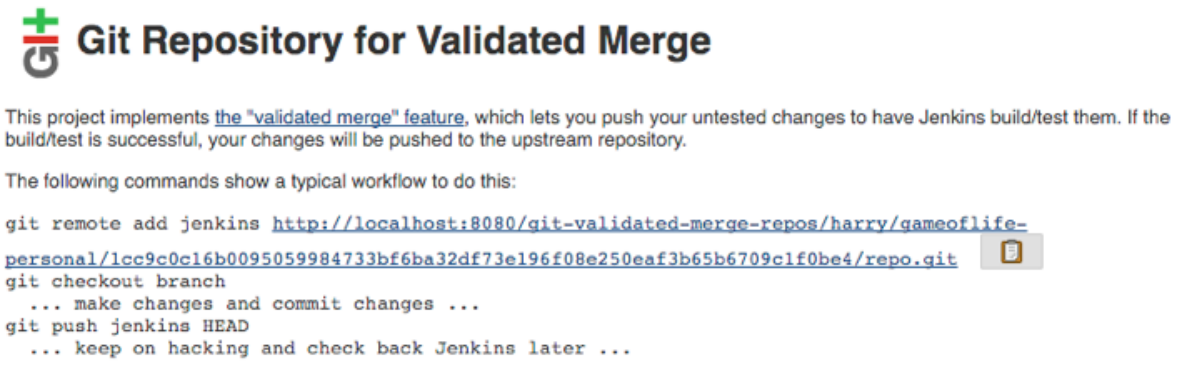


Figure 90. Validated Merge Build Log

We have to add a git "Remote" pointing to this **Jenkins-hosted** repository.

For this, use the **CodeGit** view from the right panel, and click the "Remote" button:

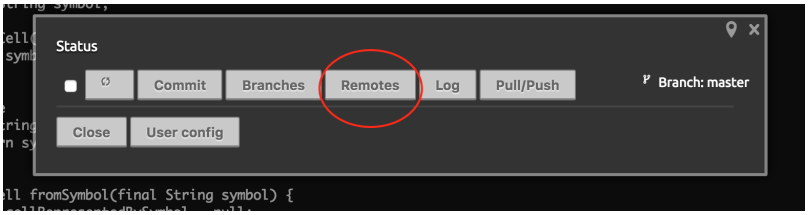


Figure 91. Accessing the CodeGit's Remote view

You see that the remote named **origin** points to your fork. Click the **New** button to add a new one:

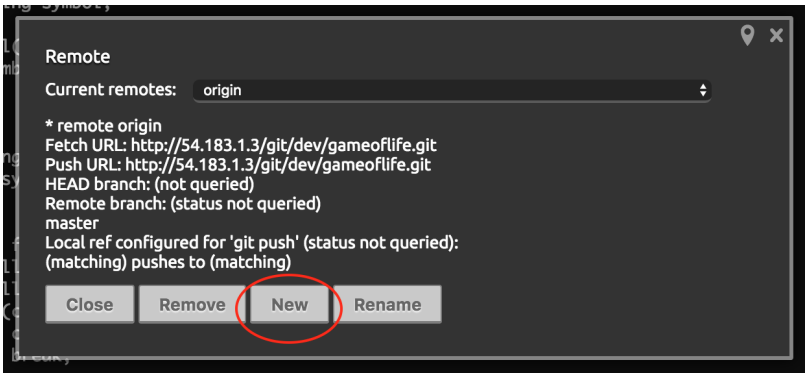


Figure 92. Accessing the new Remote view

Then fill the "New Remote" form with those settings:

- Name: **jenkins**
- Address: <The address you copied from your Job Validated Merge view>

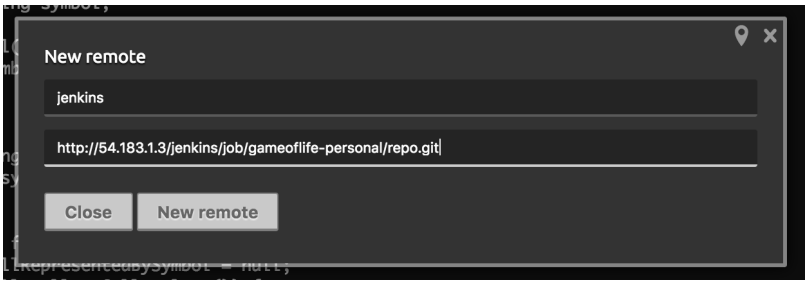


Figure 93. New Remote form pointing to Jenkins

This will register Jenkins as one of the "Remote" upstream repository.

Next, push your committed change to this remote, using the "Push/Pull" section of CodeGit view:

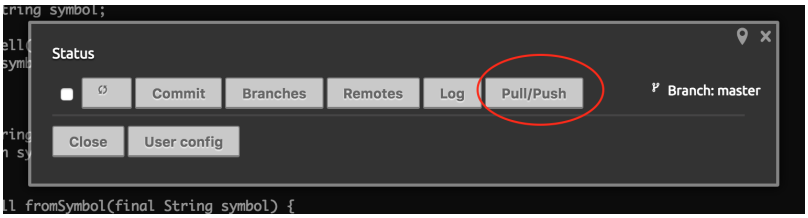


Figure 94. Accessing the CodeGit's Push/Pull view

and Push to the **jenkins** remote repository:

Figure 95. Pushing to jenkins remote

Switch back to Jenkins and wait for the build to take place. Look around in the Jenkins UI to make sure that:

- Change log correctly reports your personal change
- Build has actually picked up your change
- UI in the build top page that shows how others can retrieve this change.

Step 5: Verify there is no regression

Navigate on the Gogs repository and validate that the change has not pushed by Jenkins since the build failed.

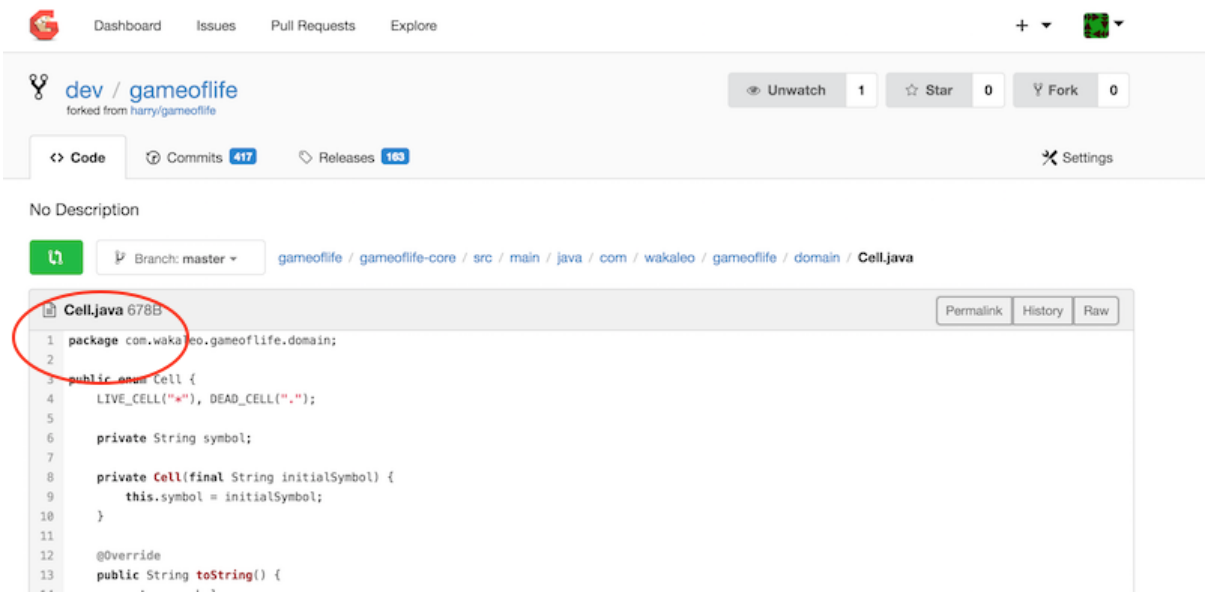


Figure 96. Fork is not changed

Step 6: Fixing the regression

Going back to the WebIDE, just make the `!!!` a comment:

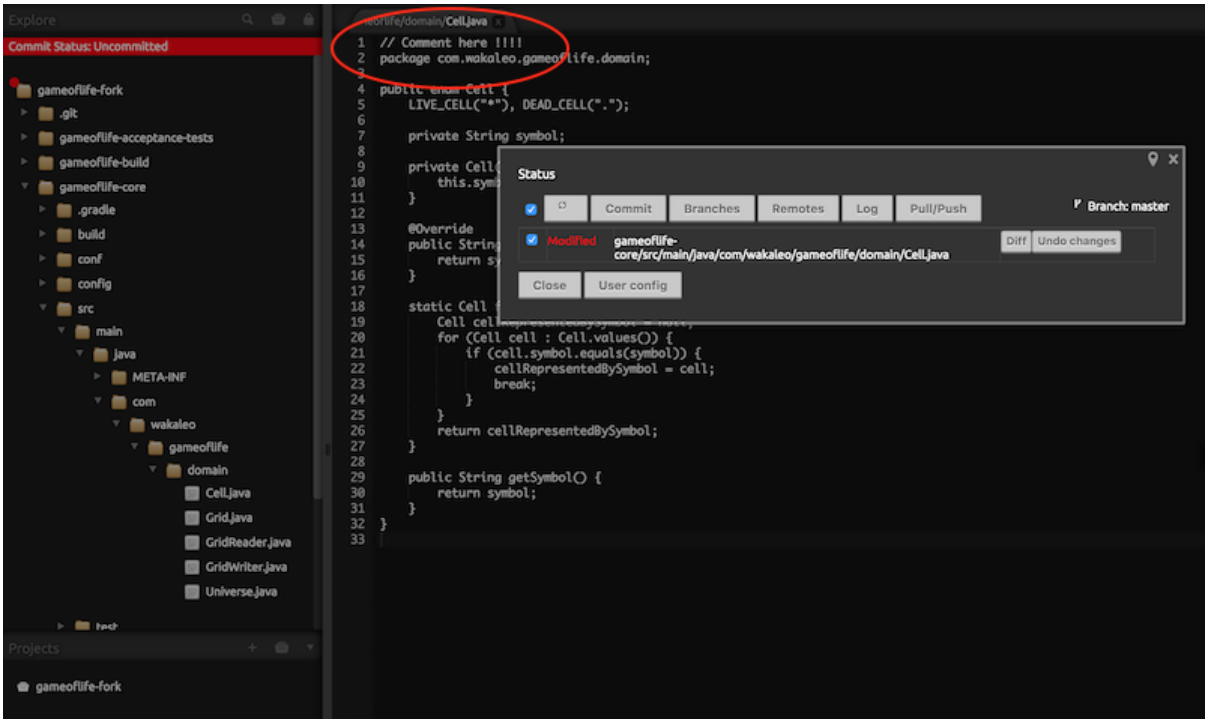


Figure 97. Commenting out the breaking change

IMPORTANT Don't forget to commit the change, and push it to the **jenkins** remote again.

Jenkins will kick-off a new build, that should be a success:

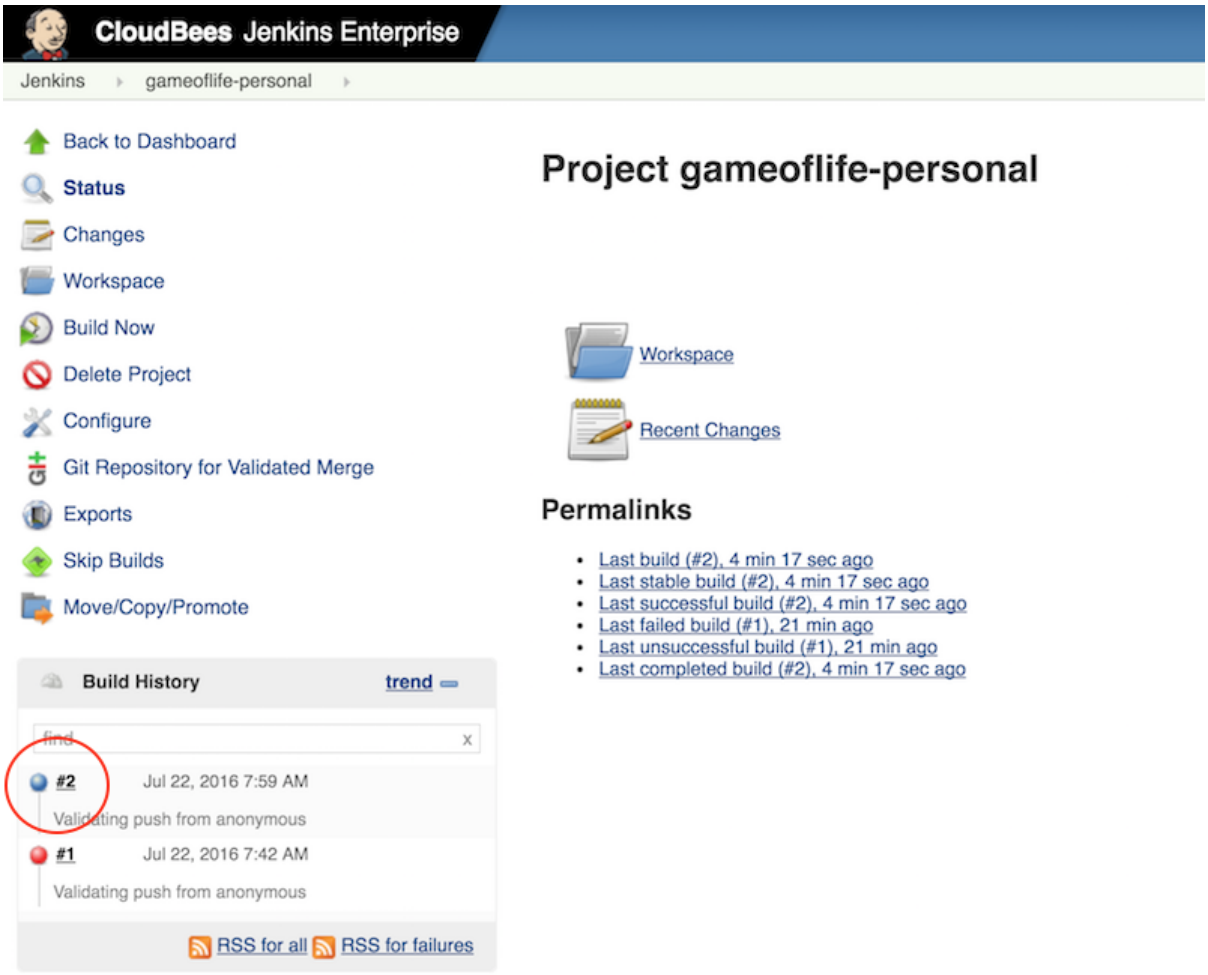


Figure 98. Jenkins jobs in success state again

Step 7: Verifying the change pushed back

Since the build has been a success, Jenkins should have pushed automatically the change to the **origin** repository.

You can check this by navigating again to the `Cell` class on the Gogs repository, to check that your commented line is here:

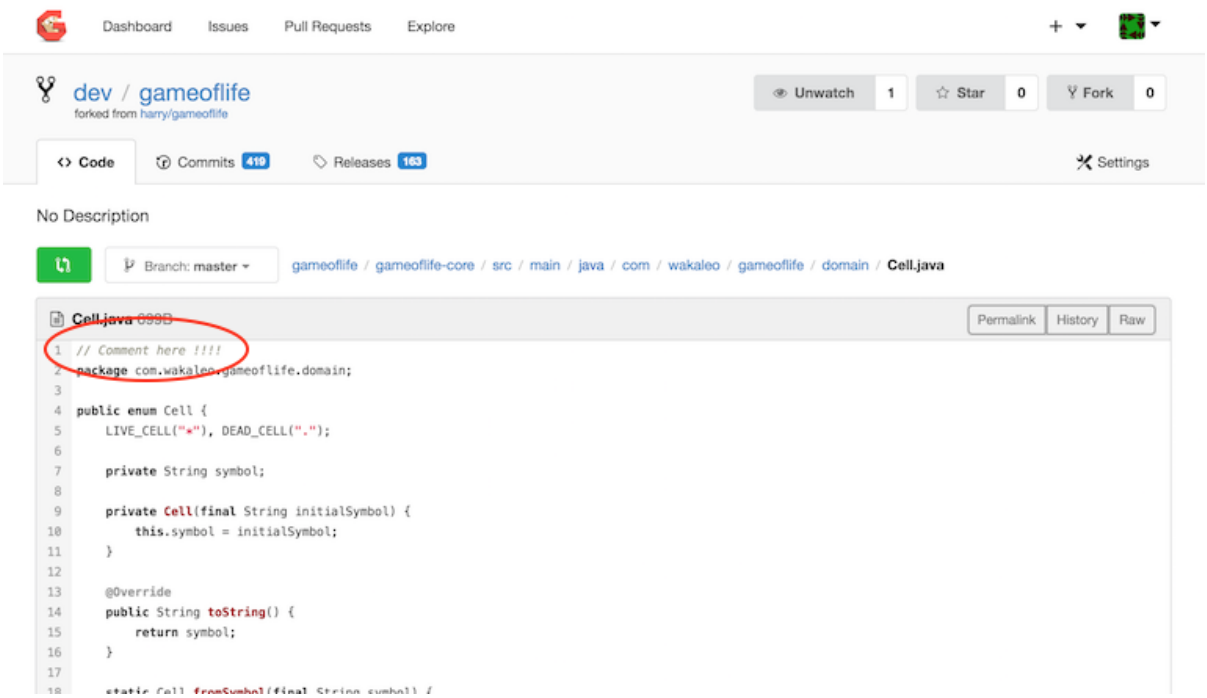


Figure 99. Verifying the change has been pushed to origin

Go back to slides

Lab 12. Pipeline

The following lab exercises complements the pipeline core training.

It consists of a sequence of lab exercises that will introduce the Pipeline plugin and its DSL syntax.

Most of them are open to experimentation, and only introduce the initial usage.

Feel free to experiment further with your own ideas and ask the training instructor for advice and/or assistance.

Experiment with DSL Syntax

The aim of this initial Lab is to get familiar with Pipeline plugin and DSL syntax basis.

Step 1. Create a Pipeline Project

With Pipeline plugin installed, a new Job type is offered creating an item on Jenkins web UI : **Pipeline**.

Create a new item of the type **Pipeline** and name it `pipeline-lab`.

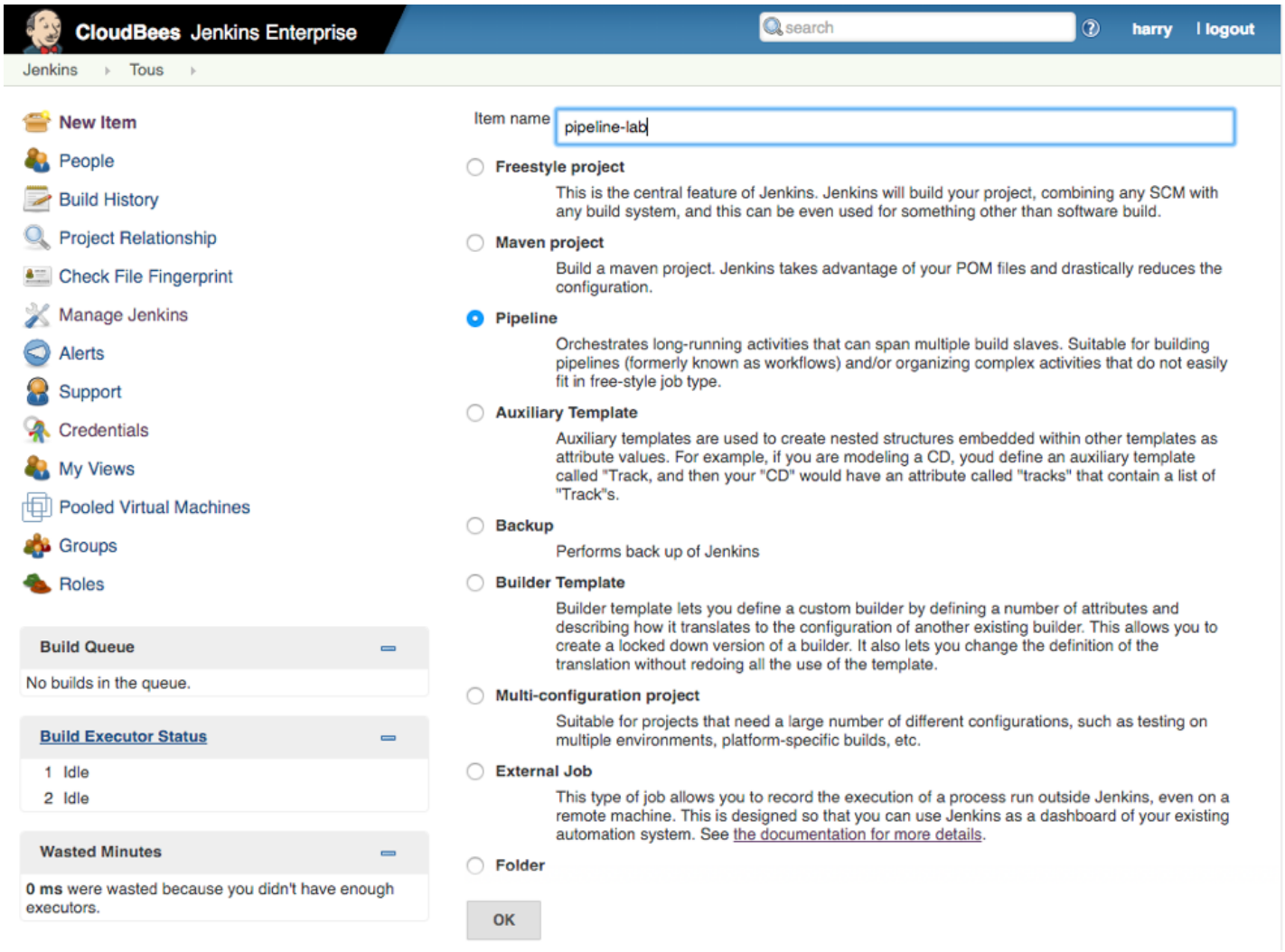


Figure 100. Creating a Pipeline Job

Compared to job types you might be used to manage, a pipeline has a very limited set of options.

You can define general :

- Job description
- Data persistence options
- Build triggers
- And pipeline script

That's it.

Everything you used to setup using form controls in UI is **replaced** in a pipeline with **DSL script**.

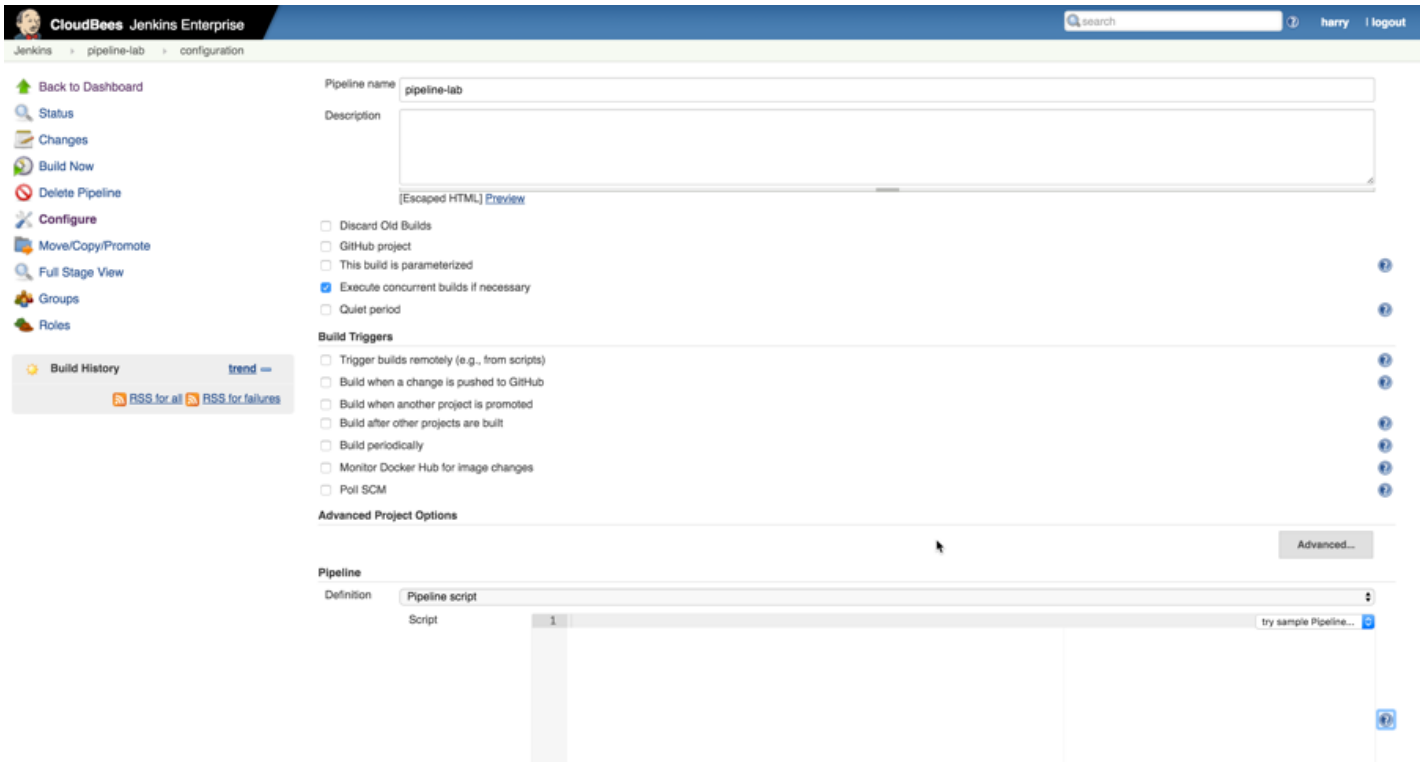


Figure 101. Overview of the Configuration

Step 2. Basic syntax

Let's start with the canonical **hello world** exercise.

TIP Pipeline DSL is based on Groovy but you actually don't need to know anything about Groovy for simplest usages. Just rely on the **Snippet generator** to learn the DSL syntax.

Select a **Print Message** step, configure attributes and let the snippet generator generate the adequate DSL script:

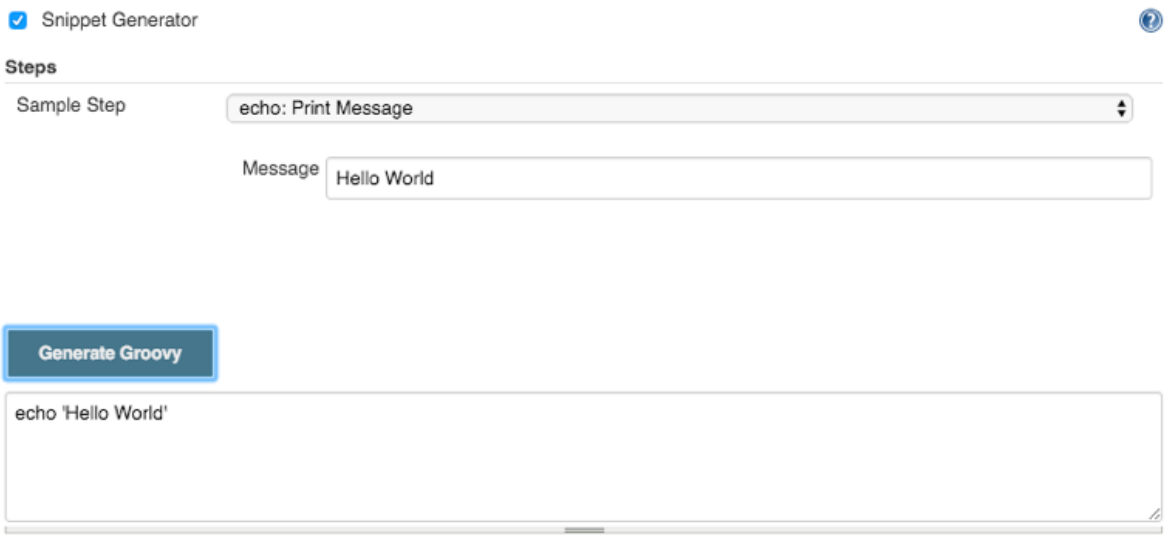


Figure 102. The Pipeline Snippet Generator

Copy-Paste to pipeline DSL script, save the job configuration (using the **Apply** button) and run it.

Pipeline is executed and you can check build console to confirm message has been printed as expected.

Step 3. Shell script

For this exercise we will introduce an execution step to run on build node, defining command to run from pipeline script.

For this purpose, we use a shell script build step:

```
node {
  git 'https://github.com/jenkinsci/docker'
  sh 'ls -l'
}
```

Such a shell script let you execute arbitrary tools within your pipeline. For non-trivial shell scripts, you will need more than just a single command line.

TIP You can use Groovy multi-line syntax for this purpose.

Groovy let you define a long, multi-line string, delimited by 3 double quotes:

```
node {
      git 'https://github.com/jenkinsci/docker'
      sh """
          ls -l
          cat README.md
      """
}
```

Pipeline Control flows

During this exercise we will experiment with pipeline DSL control flow syntax (based on Groovy).

Step 1. Control flow syntax

Let's parameterize the pipeline job with a boolean parameter.

As already done in the Lab 9 (Parameterized builds), add a new parameter to your job's config.

Our goal is to have a distinct execution path depending on this parameter value:

Figure 103. Define a Parameter

Snippet generator does not assist you in writing control flow syntax, for this one you will need to learn some basic of Groovy language.

Anyway this is the easiest part and comparable to many other popular languages.

The **Script** text area features a Groovy text editor that will assists (syntax color, autocompletion, etc.).

```
node {
  git 'https://github.com/jenkinsci/docker'
  echo "foo is $foo"
  if (foo == 'true') {
    echo "foo is true"
    sh 'ls -l'
  } else {
    echo "foo is false"
    sh 'cat docker-compose.yml'
  }
}
```

Step 2. Interact with humans

Add an **input** step to your pipeline to ask administrator for pipeline continuation. Such a manual step is useful to confirm some status that you can't automate, like checking some web UI.

TIP Can also be required when you need a set of authorized people to approve deployment.

```
input message: 'Is it ok to push to server ?',ok: 'Yes, please do'
```

Experiment with human interaction, and also check what happens when you restart Jenkins while a pipeline is waiting for human input.

Concurrent execution

This exercise aims to demonstrate concurrent execution within a single pipeline. For this purpose we will have two concurrent branches running loops

Step 1. Create a loop

Use Groovy syntax to create a loop, and pipeline's **sleep** and **echo** steps within this loop to fake some long running build activity.

TIP Use the snippet generator to discover those steps DSL syntax.

```
for (def i = 0; i < 20; i++) {
  echo "building… $i"
  sleep 1
}
```

Step 2. Create a second loop to run in parallel

Use the parallel sub-pipeline snippet generator to discover parallel execution syntax.

Copy the build loop as a parallel branch, and create a second one with distinct output:

```
parallel firstBranch: {
  for (def i = 0; i < 20; i++) {
    echo "building... $i"
    sleep 1
  }
}, secondBranch: {
  for (def i = 0; i < 10; i++) {
    echo "building in parallel ... $i"
    sleep 2
  }
}
```

Run the pipeline and check build log to confirm parallel execution.

Pipeline Stages

Goal:

The goal of this lab will be to create a complete pipeline which is composed of a set of sub-steps.

Step 1. Define stages

Let's introduce **Stages** in the pipeline, to make our process explicit:


```
stage "build"
echo "step 1"
echo "step 2"

stage "test"
echo "step 3"
echo "step 4"

stage "package"
echo "step 5"
echo "step 6"

stage "acceptance-test"
echo "step 7"
echo "step 8"

stage "deploy"
echo "step 9"
```

Run this pipeline and check the job index page for visualization

Step 2. Use stages concurrency

Configure the **acceptance-test** stage so it only support a single concurrent execution.

This will match a scenario where your test infrastructure only can run a **single** test suite, for licensing or resource consumption reasons.

```
stage concurrency: 1, name: "acceptance-test"
echo "step 7"
sleep 30
echo "step 8"
```

IMPORTANT	Do not forget to disable the Groovy sandbox since this capability (concurrency) needs low level rights.

Trigger pipeline job multiple times and check what's happening on stage execution.

Step 3. Use checkpoints

Declare a checkpoint after the **package** stage completion, and introduce a failure in acceptance-test stage:


```
stage "package"
echo "step 5"
echo "step 6"

stage "acceptance-test"
echo "step 7"
echo "step 8"
```

Figure 104. Introducing a Failure

Introduce a checkpoint just before the acceptance-test stage:

```
checkpoint 'packaged'
stage "acceptance-test"
```

Run a new build and navigate the build index page. On side-panel, click the Checkpoints link.

A **packaged** checkpoint has been recorded, and can be used to restart the pipeline from saved state:

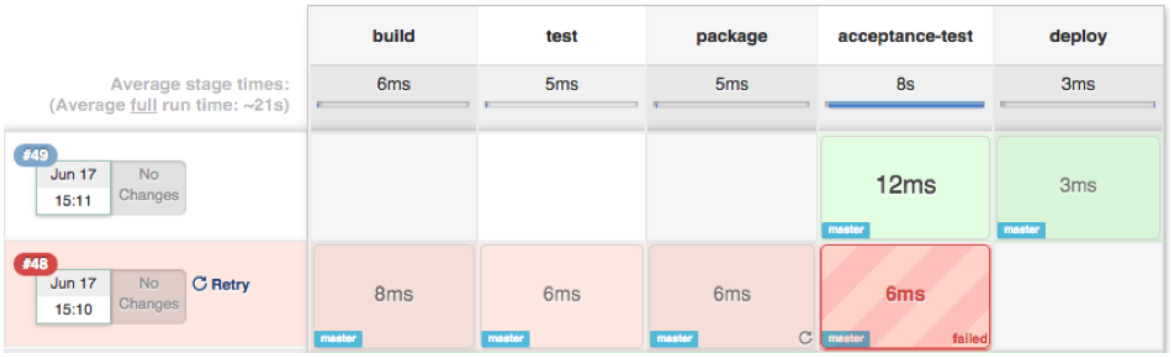


Figure 105. Checkpoint on Stage

Go back to slides

Lab 13: Templates

Builder template

Goal

In this lab, you'll learn how to create a builder template. A builder template allows you to create custom build steps that can be used many times over in different Freestyle jobs.

Step 1. Create a builder template

First, we'll create a builder template. When executed, this builder template produces a text file in the workspace that lists all the changes that went into this build.

In Jenkins, click **New Item** and select **Builder Template**. Give it the name `Save Changes`.

Click **OK** to create a new empty builder template.

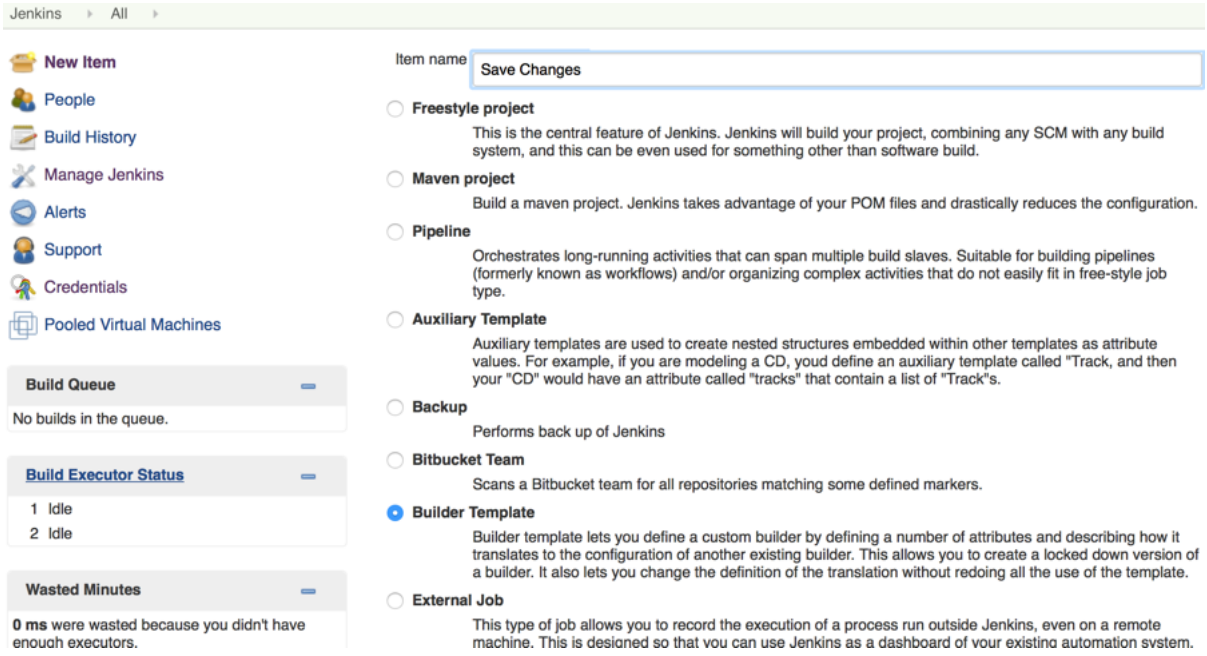


Figure 106. Creating a new Builder Template

You are now in the template configuration page.

In the **Attribute** section, click **Add Attribute**, and configure as below:

- **ID**: `changelog`
- **Display name**: `Change Log File`

- **Type**: `Text field` (should be default)
- **Inline Help**: A short description of the field, like `This is where the changelog entries are written to`

In the **Transformer** section:

- Make sure that the **Type** field is set to **Sets some variables using Groovy** (should be by default)
- **Groovy Script** field: enter the code below:

```
build.workspace.child(changelog).
  write(build.changeSet.items*.msg.join('\r\n'), null)
```

- Set the **Shell/Batch Script** field as follows, so that the build console will indicate that the file has been written:

```
echo "Wrote changelog"
```

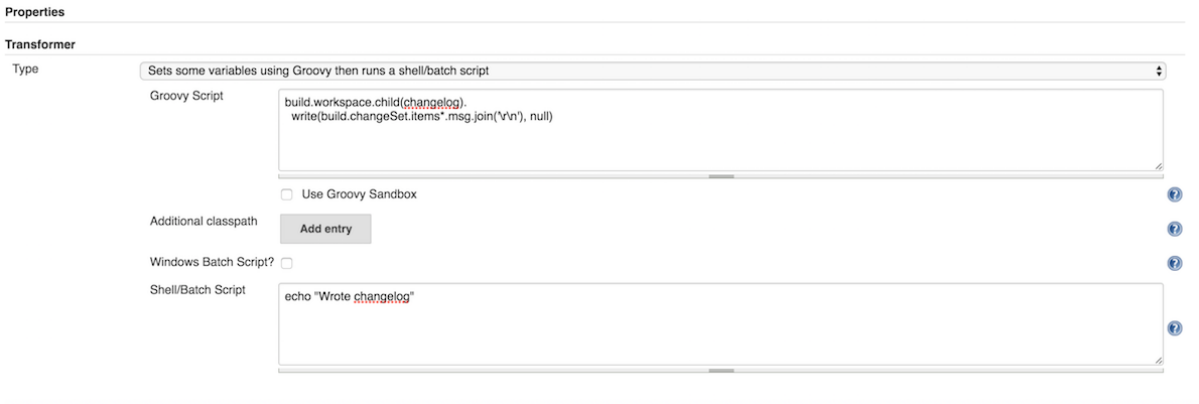


Figure 107. Configuration of Builder Template

Click **Save** to complete the setup of this template.

TIP

How did you come up with that Groovy script?
The "build" variable refers to the instance of the `AbstractBuild` class that represents the build currently in progress.
For other exposed variables, see the "Variable Bindings" section in http://jenkins-enterprise.cloudbees.com.
For methods of AbstractBuild, see http://javadoc.jenkins-ci.org/byShortName/AbstractBuild.

Step 2. Use a builder template

Open the **gameoflife-freestyle** project from Lab 2, and then click the **Configure** link from the left.

Turn off **Poll SCM** because we want to make several commits before firing a new build to test our new builder template, and polling might automatically start a new build when we don't want one.

Click **Add build step**, then select **Save Changes** to insert a new build step. Enter `changes.txt` for the **Change Log File** field.

Grab the **Save Changes** header of this build step, then drag this to the top of the **Build** section.

Click **Add post-build action**, select **Archive artifacts**, and enter `changes.txt` for the **Files to archive** field.

This lets us see the generated changelog file from Jenkins web UI:

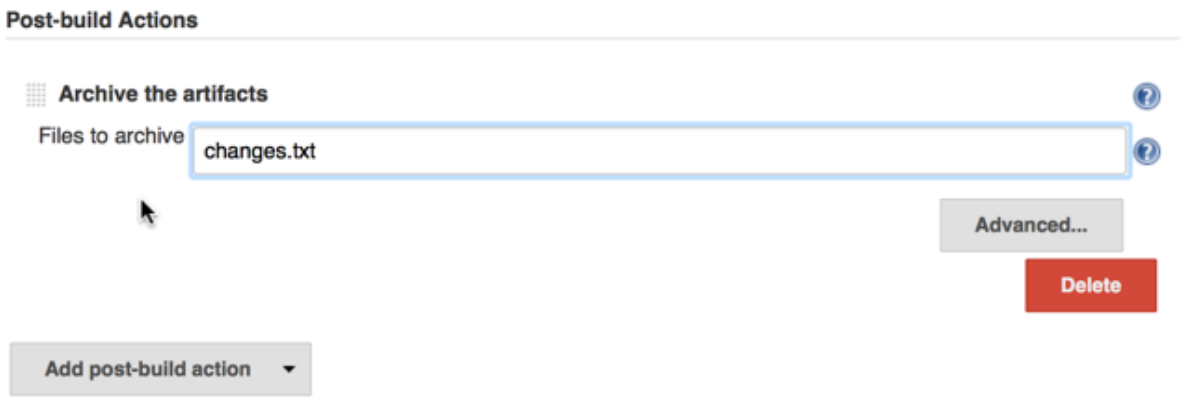


Figure 108. Post Build Step with Builder Template

Click **Save** to apply this configuration to Jenkins.

Step 3. See the builder in action

Now open Codiad, and edit any file in this project (e.g. add some spaces or add a comment to pom.xml).

Save your changes, commit and don't forget to push to the remote Git repository.

TIP If prompted, use harry as both the username and password.

Do this again to make a second commit (or more), but make sure to use different commit messages so that we can tell them apart!

Click **Build Now** in the Jenkins project to schedule a build. When it has finished, click on the newly completed build.

Build Artifacts should list `changes.txt`.

Click this link and you should see your commit messages.

TIP Use the Back button to return to Jenkins.

Job template

Goal

In this lab, you will learn how to create a job template. Job templates are useful when you would like to create multiple jobs that are very similar to each other with only a few parameters that are different between jobs. An example would be defining a Job template for a build job where the build process is the same for different projects but may only differ by the repository to be used. As with all of the templates, when changes are made to the template all child jobs using the template are also affected.

Step 1. Create a job template

Now we will turn the **gameoflife-freestyle** project into a job template.

In Jenkins, click **New Item** and select **Job Template**. Give it the name `Our Maven Project`.

Click **OK** to create a new empty job template.

Select **Groovy template transformation**.

In the **Load Prototype Job** field enter the job name **gameoflife-freestyle** and click **Load**. This will load the `config.xml` file in the **Script** field. The `config.xml` is the backend definition that Jenkins uses to read the settings of the job. The `config.xml` will define what is the 'same' between the different child jobs.

You can drag the thick bar at the bottom of the text area down to see more than a few lines of the script at once.

Step 2. Define attributes

Let's define some attributes:

We'll use an attribute to let users specify where to check out the source code:

Click **Add Attribute** and specify:

- **ID** field: the value `repoLocation`
- **Display name** field: `SCM Checkout Location`

- **Type** field: Ensure that **Text-field** is selected (should be default)

We'll use another to allow users to specify extra Maven properties:

Add another attribute and specify:

- **ID** field: `properties`
- **Display name** field: `Extra Maven Properties`
- Set the **Type** field to **Text area**.

Figure 109. Attributes for the Job Template

Next, we'll tweak the **Transformers** to use these attributes:

Look in the **Script** text area of the **Transformer** section and replace:

- `http://localhost:5000/gitserver/harry/gameoflife.git` by `${repoLocation}`
- `webdriver.port=9091<linebreak>jetty.stop.port=9998` with `${properties}`

IMPORTANT Don't delete nor remove `surefire.useFile=false`

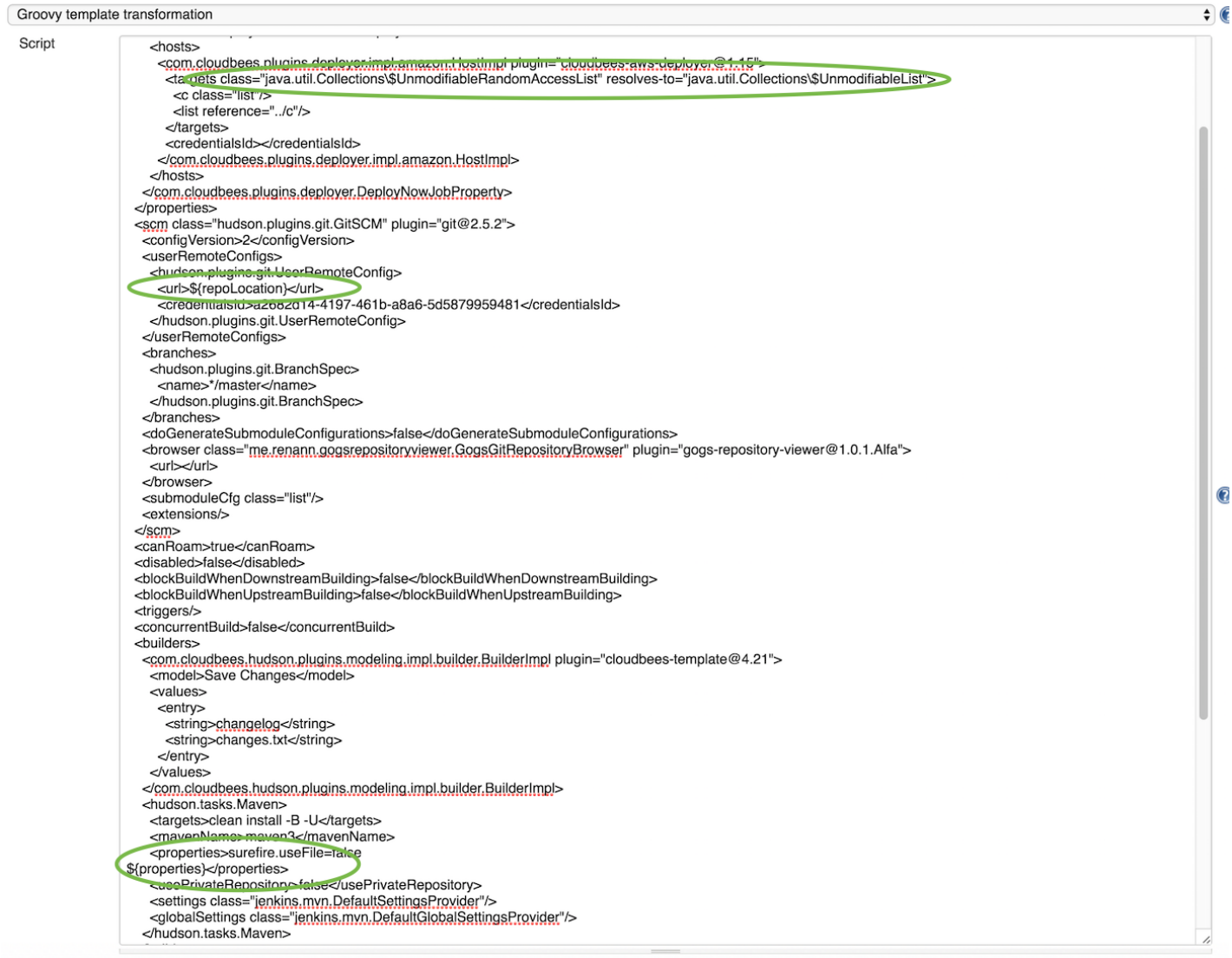


Figure 110. Final Transformer for Job Template

Finally, click **Save** to define this new job template.

Step 3. Instantiate this template

From the breadcrumb menu, select **Jenkins**, click **New Item** and select **Our Maven Project**.

Name it `gameoflife-freestyle-templatized`, then click **OK** to create a new job:

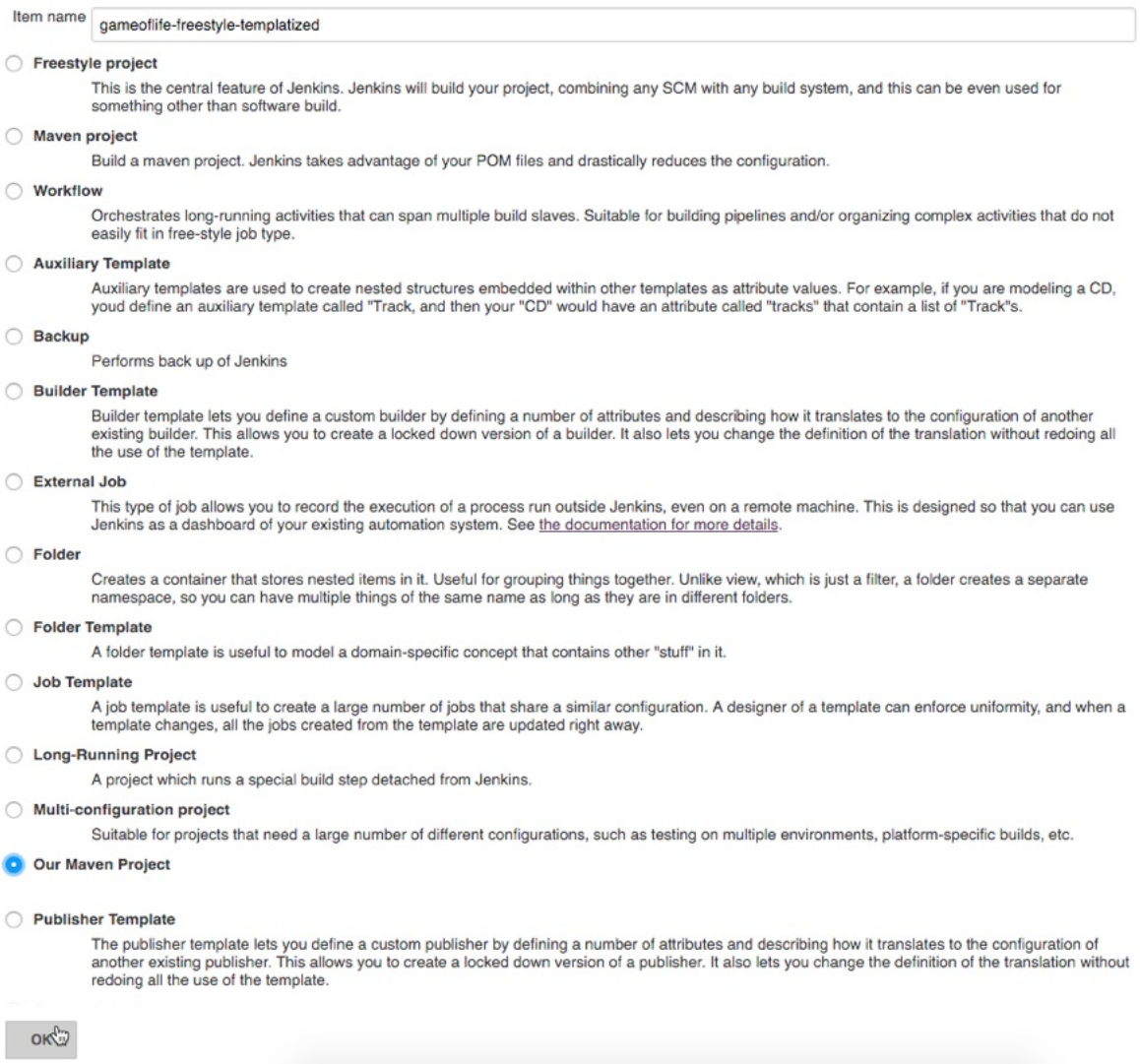


Figure 111. New item from Job Template

You should be now taken to the job configuration page. Fill the attributes as it:

- **SCM Checkout Location** : `http://localhost:5000/gitserver/harry/gameoflife.git`
- **Extra Maven Properties**, in two lines:

```
webdriver.port=9091
jetty.stop.port=9998
```

Click **Save** to complete changes.

If you now look at http://localhost:5000/jenkins/job/gameoflife-freestyle-templatized/config.xml, you should see very similar content that you saw earlier, except for a

```
<com.cloudbees.hudson.plugins.modeling.impl.jobTemplate.JobPropertyImpl>
```

with attribute properties for the instance.

Click **Build Now** and you should see similar results as when building the original job.

Step 4. Applying a template change

To verify that the template is a template and not just a wizard that creates a job, let's update the definition of this job template and make sure it gets reflected right away.

Go back to the template definition, and look for in the **Script** field, this is how we invoke Maven:

```
clean install -B -U
```

Change this, by removing the `clean` target, then save this configuration:

```
install -B -U
```

Go back to the **gameoflife-freestyle-templatized** job. Without touching the configuration, simply click the **Build Now** link to start a new build, then observe that the build runs **without** the clean goal in Maven now.

Pipeline Templates

Goal

In this lab, you will learn how to create a Pipeline template.

Pipelines allow you to define and visualize the whole application lifecycle. With Templates you can standardize pipelines, quicken the onboarding process for new projects, and make changes globally to your templatized jobs.

Step 1. Create a Pipeline template

In Jenkins, click **New Item** and select **Job Template**. Give it the name `Pipeline Template`.

Click **OK** to create a new empty job template.

Select **Groovy template for Pipeline** in the **Transformer** section.

In the **Pipeline Script** section, paste the following pipeline script:


```
node {
  stage 'build'
  git 'https://github.com/jenkinsci/docker'

  echo "jdk8 is $jdk8"

  if (jdk8 == true) {
    echo "Building in jdk8"
  } else {
    echo "Building in jdk7"
  }
}
```

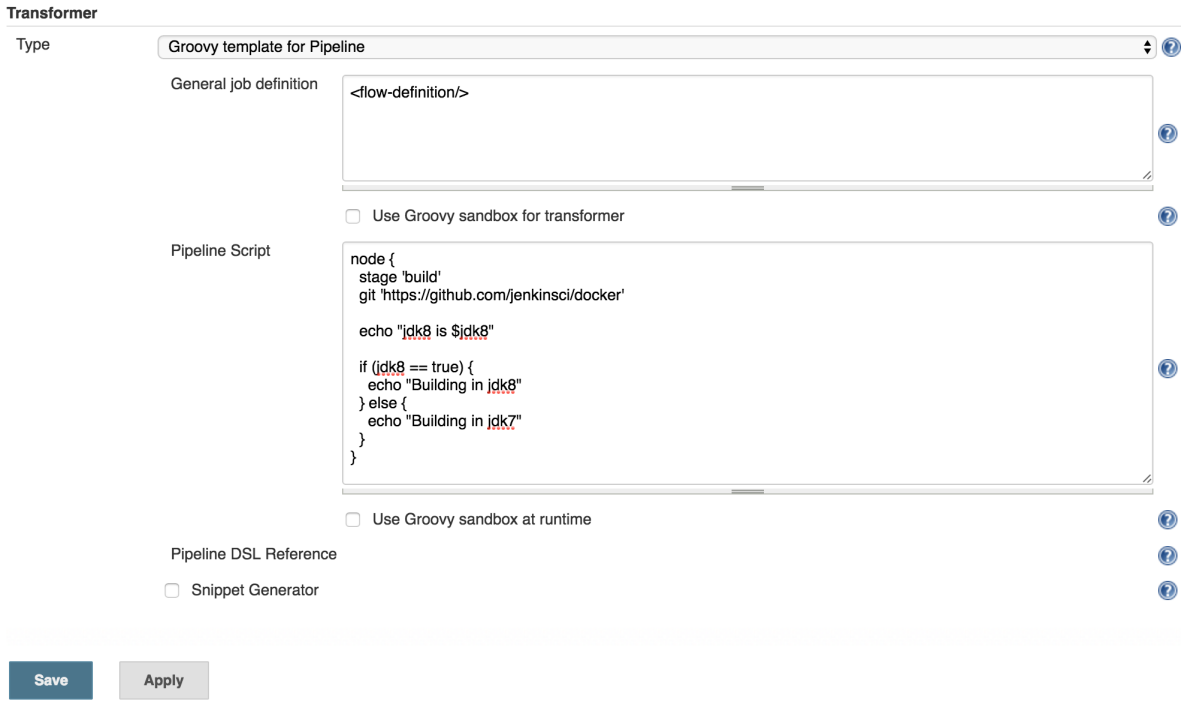


Figure 112. Pipeline Script

Step 2. Define attributes

Let's define some attributes:

We'll use an attribute to let users specify where to check out the source code:

Click **Add Attribute** and specify:

- **ID** field: the value `repoLocation`
- **Display name** field: `Repository Checkout Location`
- **Type** field: Ensure that **Text-field** is selected (should be default)

We'll use another to allow users to specify whether to build the project with jdk8 or not.

- **ID** field: `jdk8`
- **Display name** field: `Use JDK8?`
- Set the **Type** field to **Checkbox**.

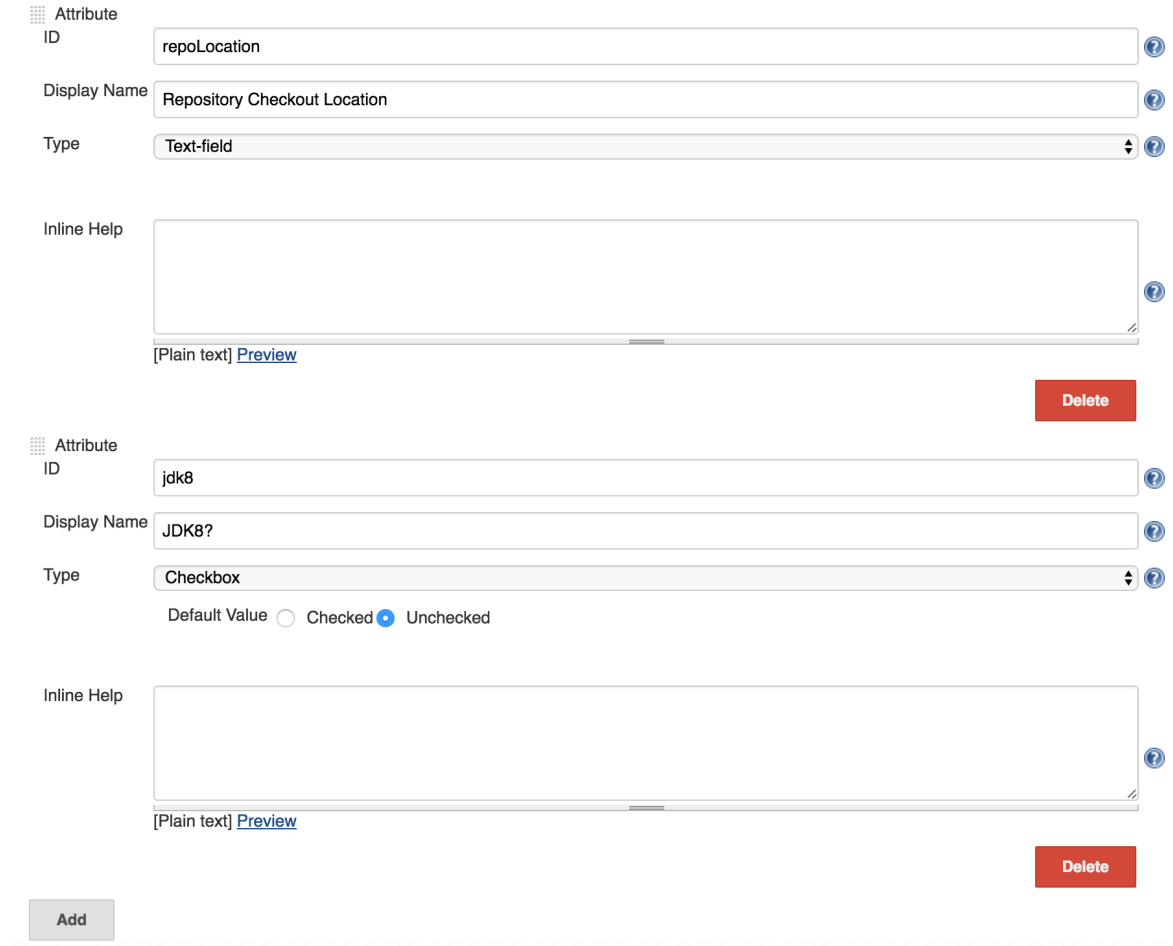


Figure 113. Pipeline Template Attributes

Next, we'll tweak the **Transformers** to use these attributes:

Look in the **Pipeline Script** text area of the **Transformer** section and replace:

- `git 'https://github.com/jenkinsci/docker'` with `git repoLocation`

The jdk8 variable is already referenced in the script.

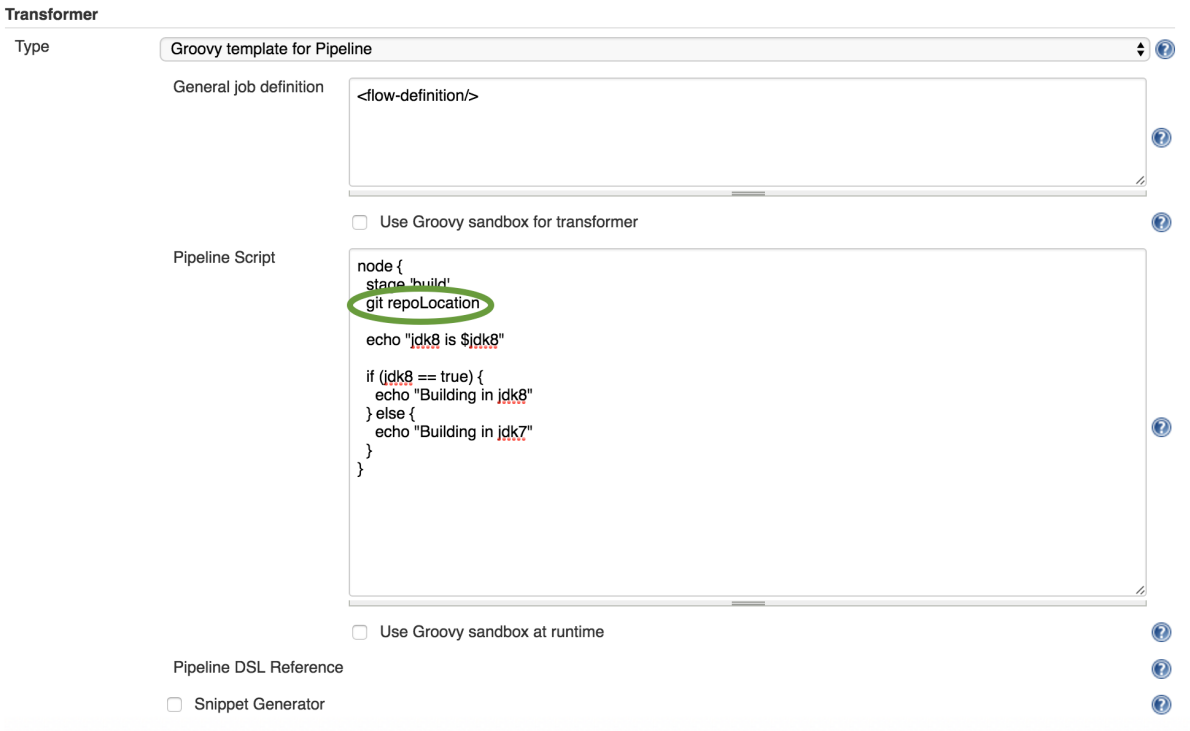


Figure 114. Attributes added to Pipeline Script

Finally, click **Save** to define this new job template.

Step 3. Instantiate this template

From the breadcrumb menu, select **Jenkins**, click **New Item** and select **Pipeline Template**.

Name it `gameoflife-pipeline-templatized`, then click **OK** to create a new job.

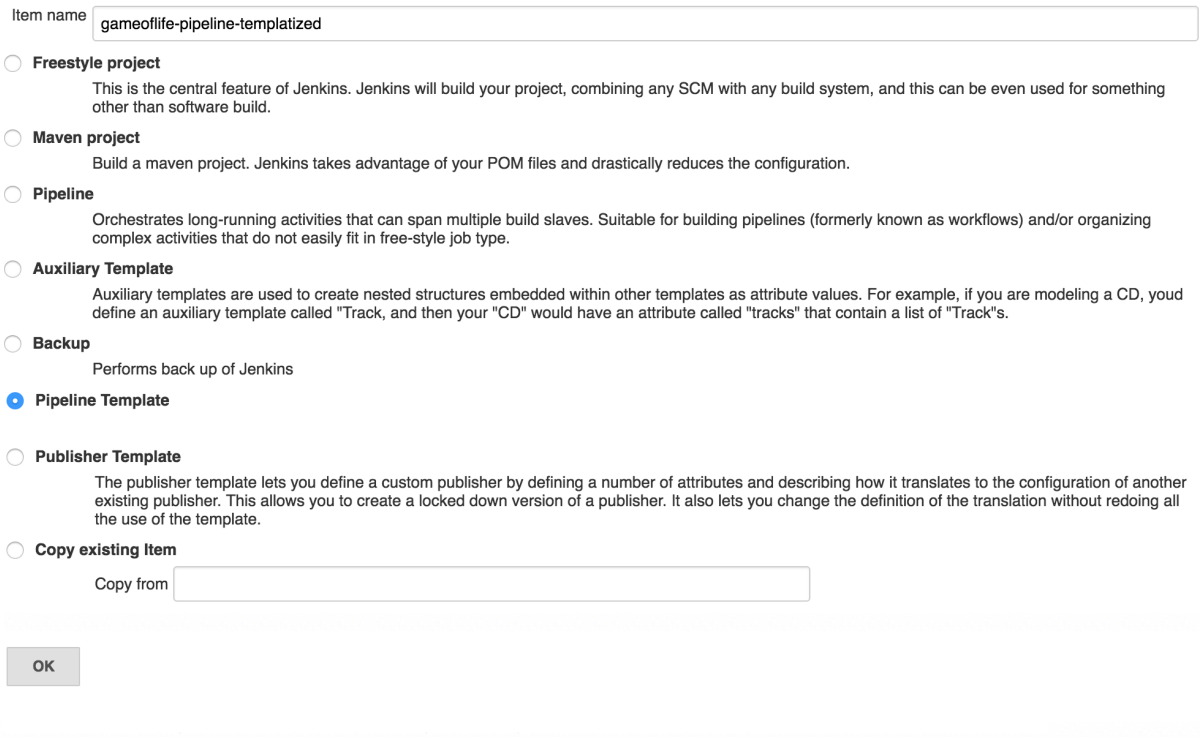


Figure 115. Create a new job from the Pipeline Template

You should be now taken to the job configuration page. Fill the attributes as specified below:

- **Repository Checkout Location** :
  `http://localhost:5000/gitserver/harry/gameoflife.git`
- **Use JDK8?** should be checked

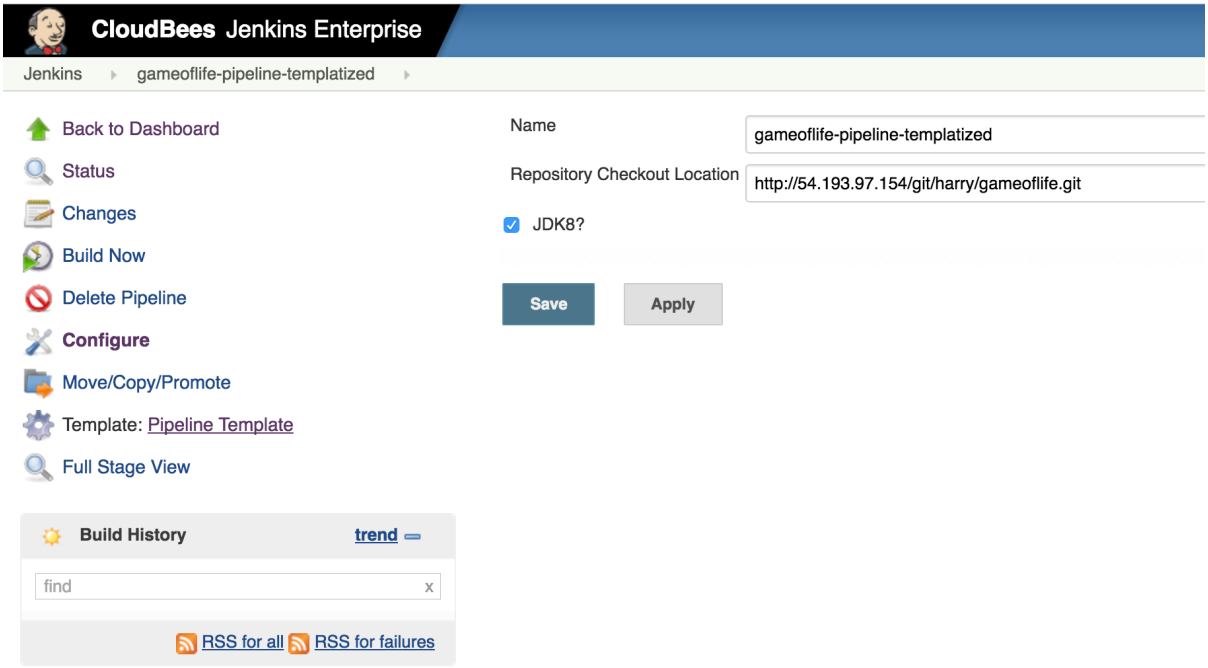


Figure 116. Templatized Job Configuration

Click **Save** to complete changes.

Click **Build Now**.

Note in the console output that the job checks out the specified directory and uses jdk8.

Figure 117. Console Output for Templatized Job

Step 4. Applying a template change

To verify that the template is a template and not just a wizard that creates a job, let's add another stage to the Pipeline template and make sure that it gets reflected in the child job.

Select the `Pipeline Template` job and click **Configure**.

Now go back to the **Pipeline Script** section, paste the following:


```
node {
  stage 'build'
  git repoLocation

  echo "jdk8 is $jdk8"

  if (jdk8 == true) {
    echo "Building in jdk8"
  } else {
    echo "Building in jdk7"
  }

  stage 'Deploy'
  input 'deploy?'

  echo "deployed application!"

}
```

Notice that it is the same script but we added a new `Deploy` stage to the template.

Click **Save**.

Go back to the `gameoflife-pipeline-templatized` job. Without touching the configuration, simply click the **Build Now** link to start a new build. Observe that the job now asks for an approval after the build completes and will subsequently deploy the application if the approval is given.

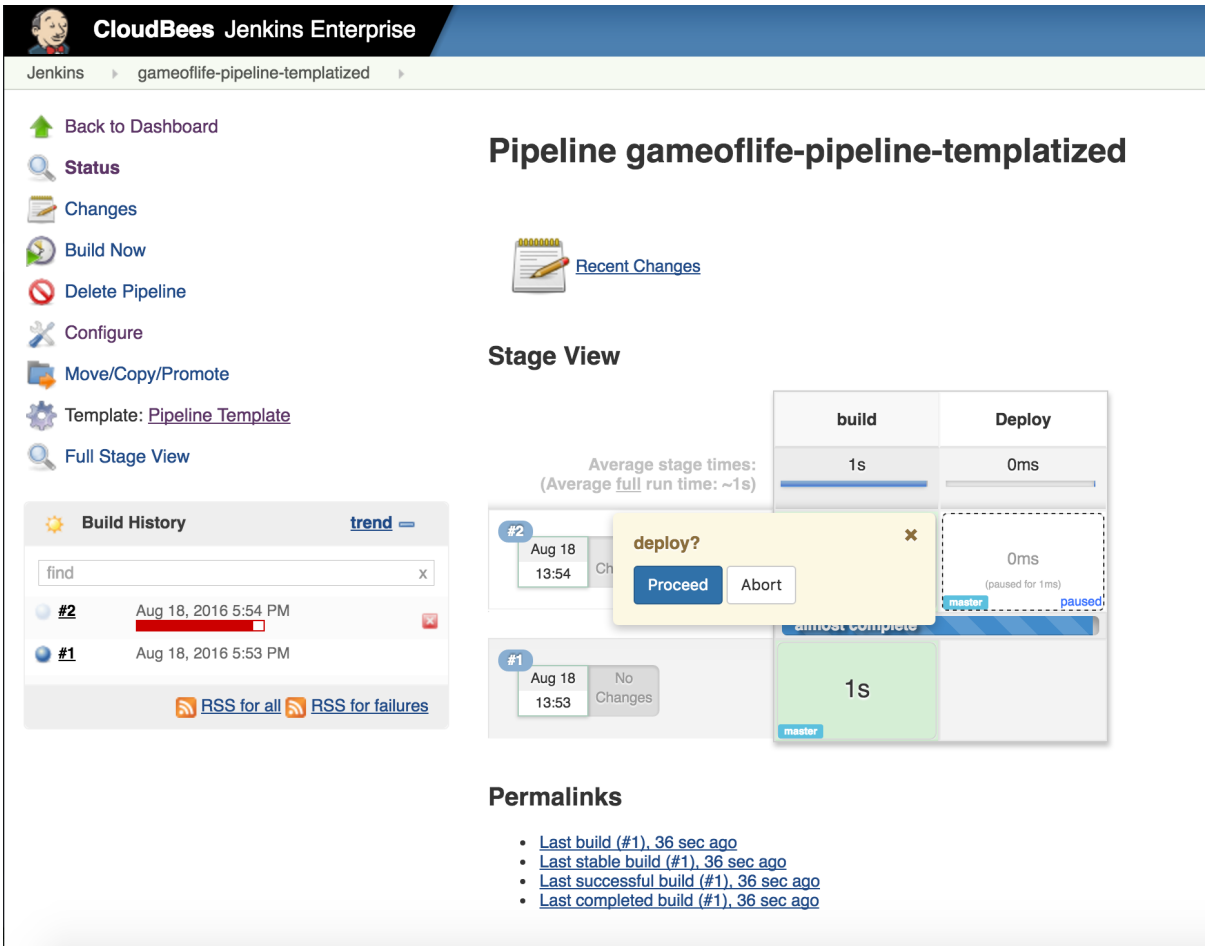


Figure 118. Deploy Stage added to Template

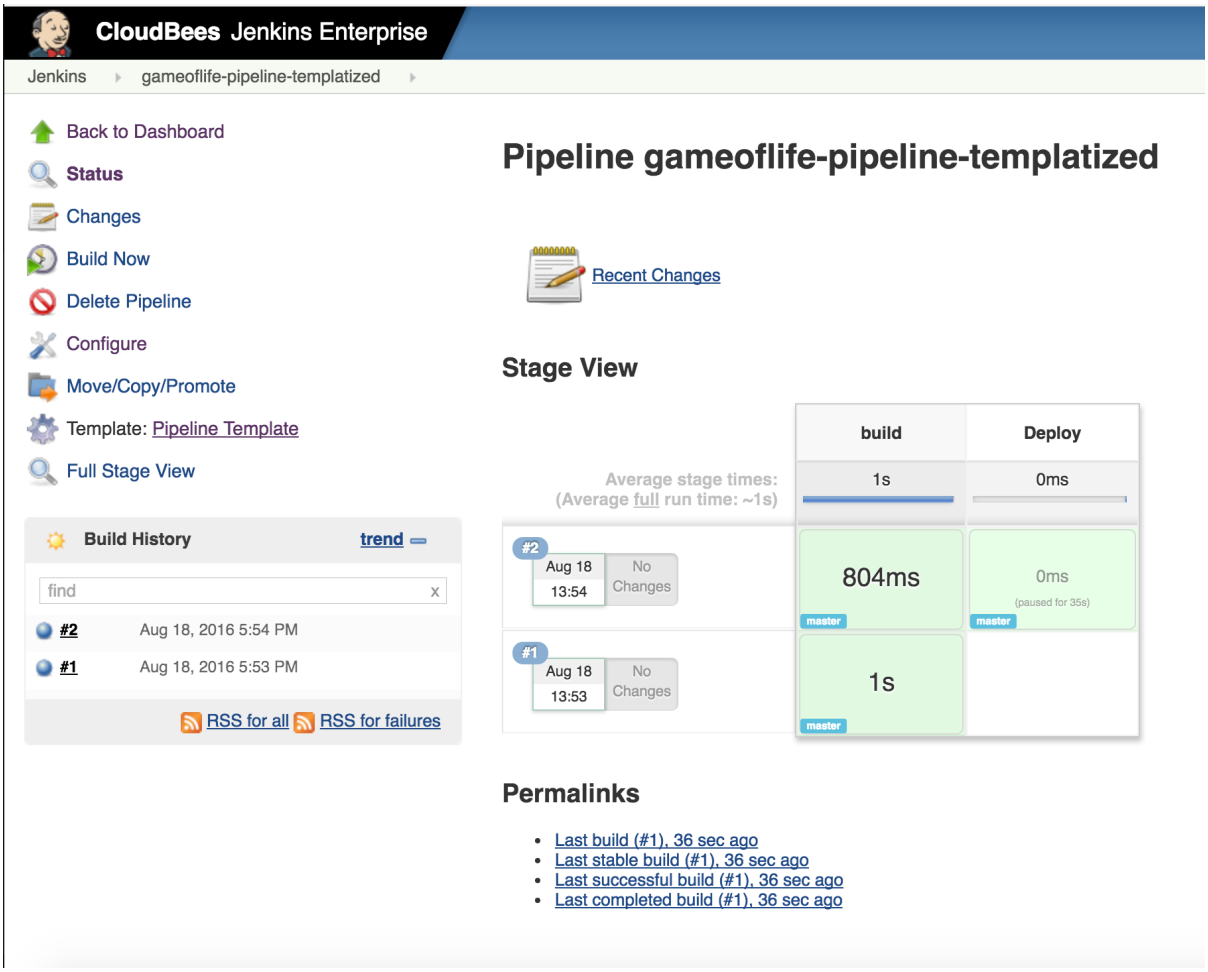


Figure 119. Deploy stage ran after approval

Go back to slides

Lab 14: CloudBees Support

Goal

This lab will teach users the basics of CloudBees support plugin. This plugin provides the ability to generate a bundle of all the commonly requested information used by CloudBees when resolving support issues.

Step 1: Verify / Install the Support plugin

Since some CloudBees customers purchase support contracts without being licensed for CloudBees Jenkins Enterprise, this plugin is also available for download and installation into any Jenkins installation.

Simply go to the related release page, pick the latest version and use the download from the **Download** section.

You can then install the downloaded plugin using **Manage Jenkins** , **Manage Plugins** , **Advanced** , **Upload Plugin**, and select the downloaded ".hpi" file.

Figure 120. Plugin installed by default in CJE

Step 2: Generate a bundle

To generate a bundle, simply go to the CloudBees Support link on the Jenkins instance and Generate the Bundle with default options checked.

TIP Direct URL access from http://localhost:5000/jenkins/support/

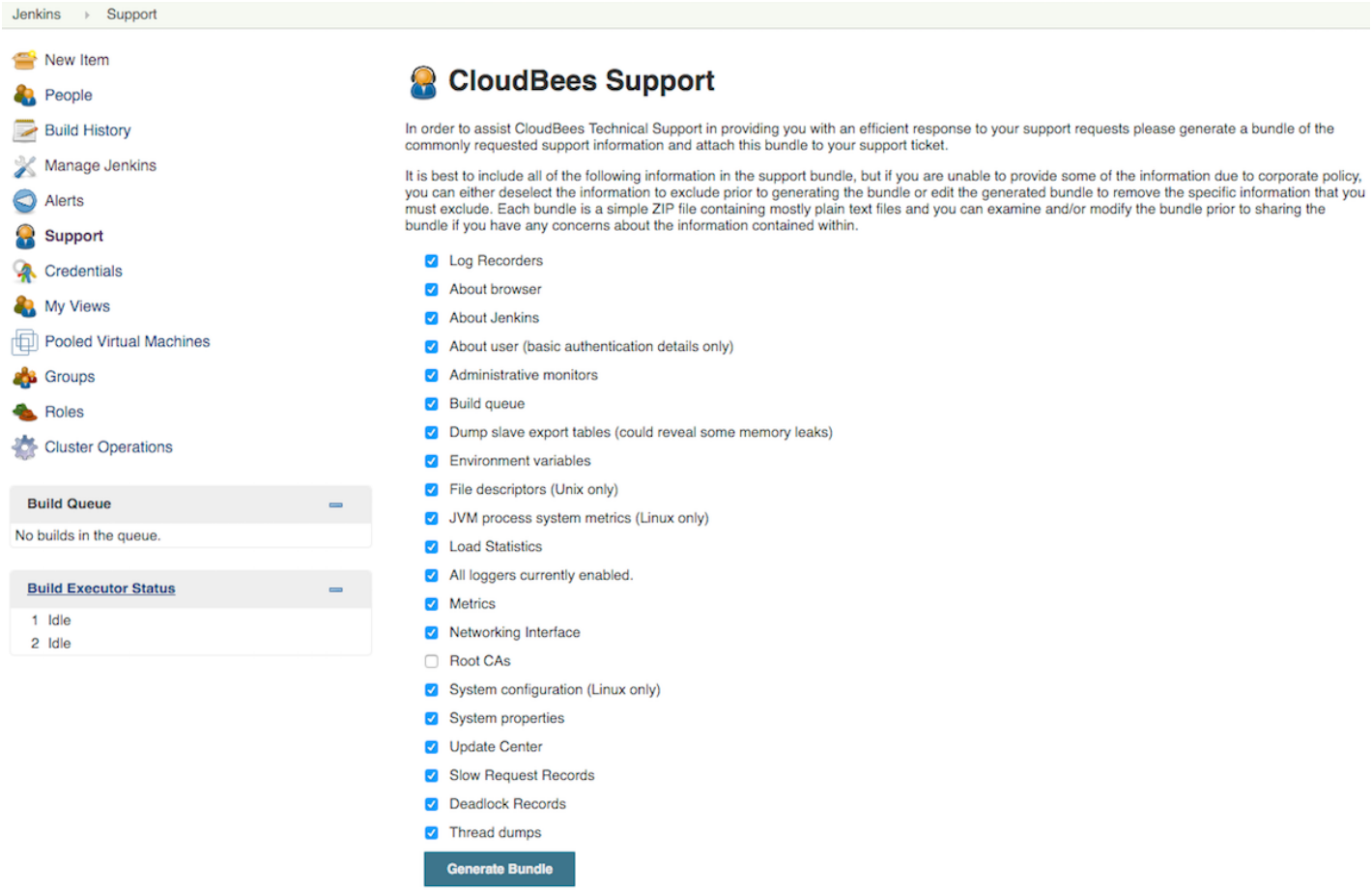


Figure 121. CloudBees Support Bundle page

The bundle will be generated and downloaded as a ZIP achive:

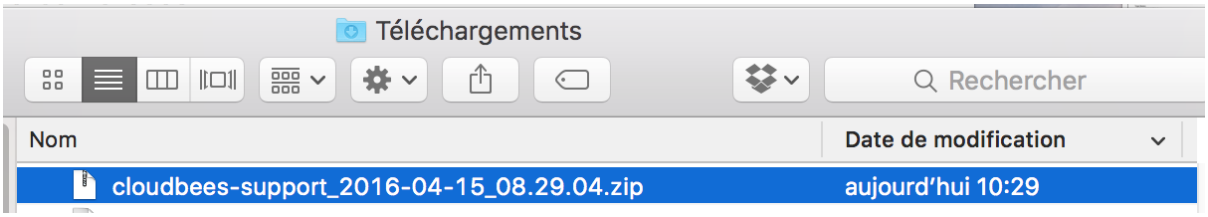


Figure 122. CloudBees Support Bundle ZIP

