Tutorial

The ultimate Jenkins setup

Version 0.1

Revision History

Version 0.1

Per-Gustaf Stenberg, dt08ps5@student.lth.se
Joakim Hembrink, dt08jh8@student.lth.se

March 5, 2013

Contents

1 Introduction

Jenkins is an open source tool that is forked from the Hudson project to facilitate continuous integration. This means that Jenkins for instance, can build projects, test and publish them.

Setting up a good Jenkins environment can be hard and tricky, this document will hopefully help you to get some basic knowledge about how Jenkins works and how to configure it to work with your Java-project.

This tutorial will focus on Jenkins configured with *SVN*, *Ant* and some specific plugins; for instance, *Emma* and *FindBugs*.

2 Setting up Jenkins

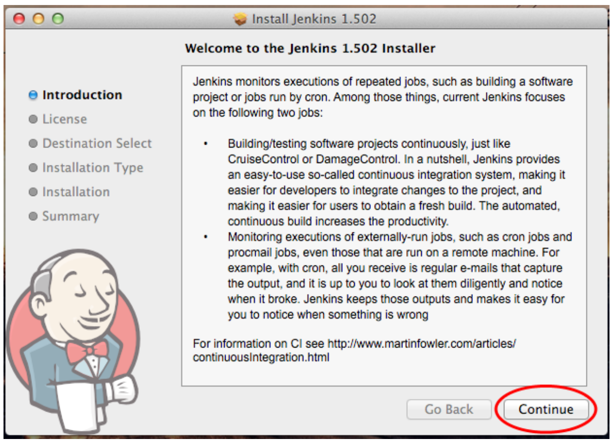
Jenkins can be downloaded from: `http://jenkins-ci.org`.

There are a couple of native packages available, and the download links can be located directly on the index page to the right. There you can see the different operating systems that are supported; download your OS of choice. This tutorial will show how to install Jenkins on the *Ubuntu/Debian* and *Mac OS X*.

2.1 Installing Jenkins

2.1.1 Mac OS X

When your Jenkins is downloaded (the Mac OS X version in this case) **double click** on downloaded .pkg file. This will give you an installation wizard like in figure 1, press **Continue** and follow the steps presented by the wizard.



Welcome to the Jenkins 1.502 Installer

Jenkins monitors executions of repeated jobs, such as building a software project or jobs run by cron. Among those things, current Jenkins focuses on the following two jobs:

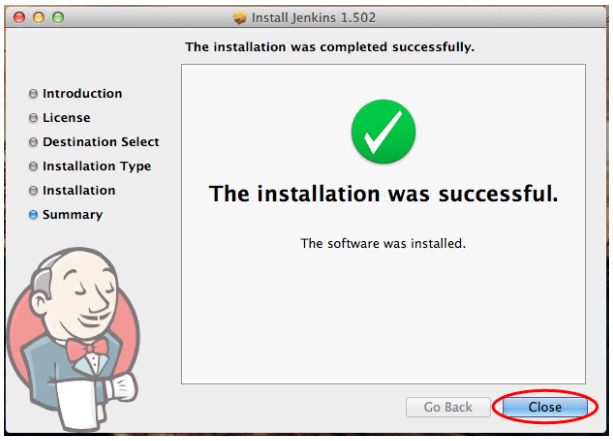
- Building/testing software projects continuously, just like CruiseControl or DamageControl. In a nutshell, Jenkins provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases the productivity.
- Monitoring executions of externally-run jobs, such as cron jobs and procmail jobs, even those that are run on a remote machine. For example, with cron, all you receive is regular e-mails that capture the output, and it is up to you to look at them diligently and notice when it broke. Jenkins keeps those outputs and makes it easy for you to notice when something is wrong

For information on CI see http://www.martinfowler.com/articles/continuousIntegration.html

Figure 1:

Eventually, when the installation is finished you will get to the last page of the wizard, as shown in figure 2. The web browser will open a pop up window with Jenkins' index page, which will be indicating the preparation for start up.

Jenkins uses port 8080 as its default port, which mean you can reach the Jenkins-server by typing:



The installation was completed successfully.

The installation was successful.

The software was installed.

Figure 2:

Jenkins host

```
http://localhost:8080
```

When the installation is finished the system will have created a dedicated user called "Jenkins" that will run a daemon with Jenkins in the background. To control the start and stop of this daemon you type the following commands in the terminal:

To start Jenkins daemon

```
sudo launchctl load /Library/LaunchDaemons/org.jenkins-ci.plist
```

To stop Jenkins daemon

```
sudo launchctl unload /Library/LaunchDaemons/org.jenkins-ci.plist
```

Jenkins places itself by default in a specific Jenkins home folder in the users folder, which can be found:

Jenkins default home directory

```
/Users/Shared/Jenkins/Home
```

Congratulations, you have now got a fresh installation of Jenkins and are ready to start experimenting!

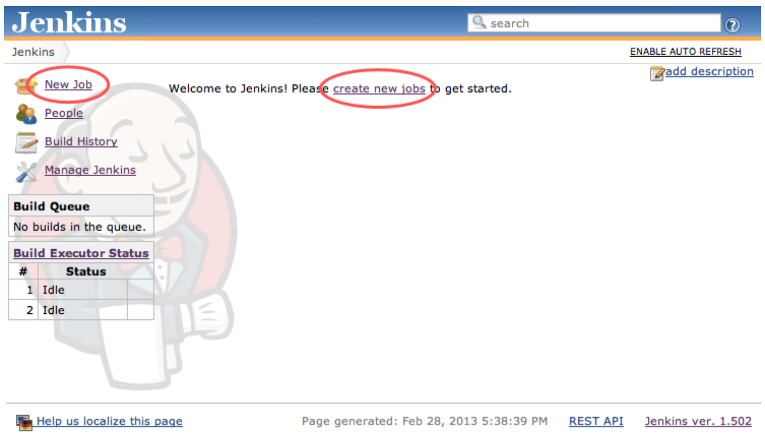
2.1.2 Ubuntu/Debian

To be written in some other release.

2.2 Configurate a new project

When you first visit Jenkins you will see the Jenkins start page: here you can add new projects, manage plugins and general Jenkins settings. To create your first Jenkins project click **create new job** or the **New Job** button, as shown in figure 3.

This will take you to another page. Here you will give your new project a name and select what project type you are going to set up. There are four different project types to choose from:


Jenkins

search

Jenkins

ENABLE AUTO REFRESH

add description

New Job

People

Build History

Manage Jenkins

Welcome to Jenkins! Please create new jobs to get started.

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Help us localize this page

Page generated: Feb 28, 2013 5:38:39 PM   REST API   Jenkins ver. 1.502

Figure 3:

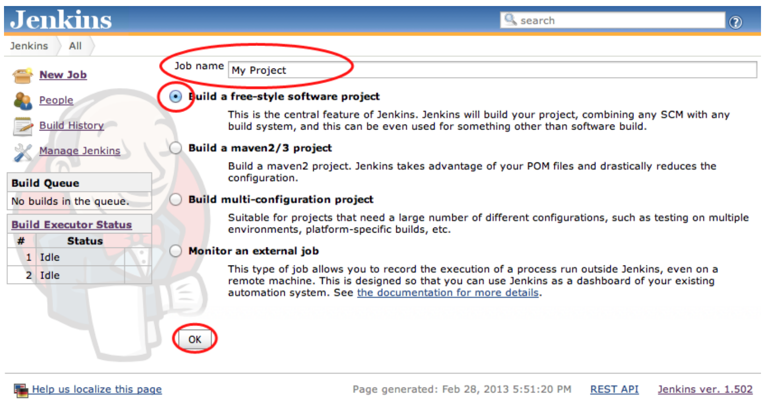
Build a free-style software project This is the default and also the most common project setup. This option gives you the most flexibility to adapt your setup for your chosen revision tool and building strategy.

Build a maven2/3 project This setup uses mavens simplicity and configures your project depending on maven.

Build multi-configuration project This setup is most suitable for bigger project with different configurations.

Monitor an external job This makes the possibility to have Jenkins as a dashboard that monitors existing automated systems.

In this tutorial we will use the option "Build a free-style software project" to make it possible to configurate our system with SVN and Ant. Name your project and select **Build a free-style software project** and press **OK** as shown in figure 4.


Jenkins

search

Jenkins > All

New Job

People

Build History

Manage Jenkins

Build Queue

No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Job name: My Project

[✓ **Build a free-style software project**]

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Build a maven2/3 project

Build a maven2 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

Build multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

Monitor an external job

This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See the documentation for more details.

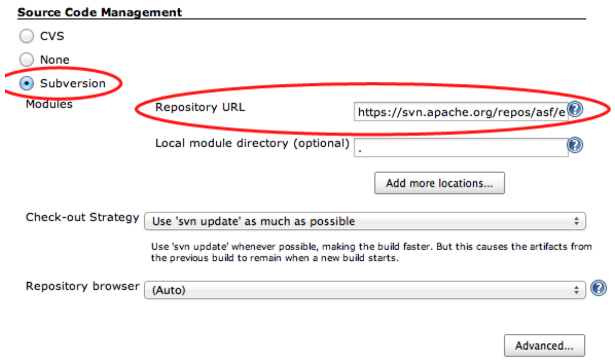
OK

Help us localize this page

Page generated: Feb 28, 2013 5:51:20 PM   REST API   Jenkins ver. 1.502

Figure 4:

After you clicked "OK" you will be redirected to the project configuration page. This is where all your future configurations that affects your project will be handled. Scroll down until you get to the **Source Code Management**-section in the project configuration page. This is where you will setup your connection to your repository, in this tutorial we will be using SVN so select **subversion** and enter your subversion-URL like in figure 5.



Source Code Management

CVS
None
[✓ Subversion]

Modules
Repository URL: https://svn.apache.org/repos/asf/e
Local module directory (optional): .
Add more locations...

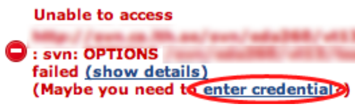
Check-out Strategy: Use 'svn update' as much as possible
Use 'svn update' whenever possible, making the build faster. But this causes the artifacts from the previous build to remain when a new build starts.

Repository browser: (Auto)

Advanced...

Figure 5:

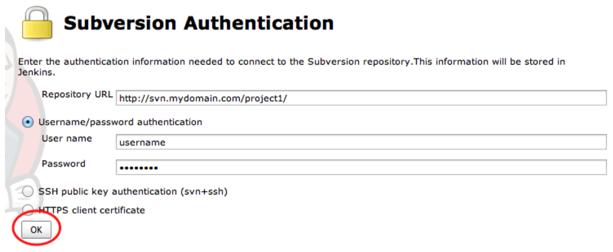
If your repository is not a public repository and you have got some kind of security on it you will get a "Unable to access" message as exemplified in figure 6 under the URL-input field. Looking at the figure above, press **enter credential** which will redirect your to another page to validate your security credential.



Unable to access [redacted] : svn: OPTIONS [redacted] failed (show details)
(Maybe you need to enter credential)

Figure 6:

Enter your security credential like in figure 7 and press **OK**, hopefully this will give you access to your repository.



Subversion Authentication

Enter the authentication information needed to connect to the Subversion repository.This information will be stored in Jenkins.

Repository URL: http://svn.mydomain.com/project1/

[✓ Username/password authentication]
User name: username
Password: ••••••••

SSH public key authentication (svn+ssh)
HTTPS client certificate

OK

Figure 7:

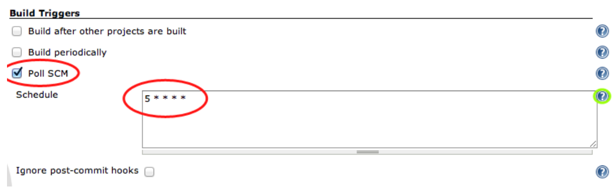
You will now have complete access to your repository with Jenkins and you are ready to setup your build triggers. Scroll down to the section "Build Triggers", this is the section were you will configured your build triggers. There are three different options to trigger your build:

Build after other projects are built Trigger your build after another project has been built. This is useful when you have got a lot of projects that are dependent on each other.

Build periodically Trigger your build according to a fixed schedule.

Poll SCM Look in the repository for changes and trigger the build when a new change has occurred.

We will use the option “Poll SCM”, click the **Poll SCM** like in figure 8 and enter **5 * * * *** in the schedule text field. “5 * * * *” means that Jenkins should check for changes every 5:th minute. The schedule input syntax is values separated by a whitespace and follows the **Cron** specification with some exceptions. For more information about how to setup a schedule click the question mark beside the input-field that is marked with green in figure 8 or read more about Cron at this URL `https://en.wikipedia.org/wiki/Cron`.



Build Triggers

- Build after other projects are built: ☐
- Build periodically: ☐
- Poll SCM: ☑
- Schedule: 5 * * * *
- Ignore post-commit hooks: ☐

Figure 8:

Congratulations, you now got a running Jenkins project that triggers build depending on the repository!

3 Writing your building script

When you got a working build trigger and a connection with the repository your are ready to set up your building script. The building script is executed when a build is triggered and can for instance compile java code and generate reports.

In this tutorial we will be using Ant as our building script, we will not get into details so you can read more about how Ant works here `http://ant.apache.org/` and get familiar with it. When you know the basics of Ant your are ready to set up your first building script.

We are going to set up four targets in our first building script; a clean-, init-, compile-, and jar-target. Start by creating your building-script file, make a **new file** and name it **main.xml**. Open the file with your favourite XML editor. First of all we need to add our project tag into our XML file, by typing this into the top of the file:

```
<project name="MyProjectName" basedir=".">
```

The property “basedir” is the directory where your Ant-project should operate in, “.” means were the script is executed. The attribute “name” is your project name.

We are going of to add some properties that we are going to use in our targets in order to make a fine structure in our building script; add these lines into your XML file:

```
<property name="project.name" value="MyProjectName"/>

<property name="lib.dir" value="lib"/>
<property name="src.dir" value="src"/>
<property name="build.dir" value="build"/>
<property name="classes.dir" value="${build.dir}/classes"/>
<property name="jar.dir" value="${build.dir}/jar"/>
```

These properties is setting up your basic file structure for your project.

Now we are going to make our first target the “init” target. Add these into your XML file:

```
<target name="init">
    <mkdir dir="${build.dir}"/>
</target>
```

This target creates your build directory which are used for compiling your source and store your jar files.

We also need a “clean” target, for cleaning up the mess our script makes when executed. Add this to your XML file as well:

```
<target name="clean">
    <delete dir="${build.dir}"/>
</target>
```

When you got your “init”- and “clean” target you are ready to make your first “compiling” target, but before this we need to set up our class-path in order to compile our source with the depending libraries. In order to set our class-path target we need to add this into our script file:

```
<path id="classpath">
        <pathelement location="${lib.dir}/junit-4.0.jar" />
</path>
```

This is were you are going to add all your needed libraries, in this case we are just in need of the jUnit library. If your project needs more libraries just add them into the class-path tag like the jUnit library.

Now you got your classpath and are ready to add the “compiling” target by add this to your script file:

```
<target name="compile" depends="init">
    <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}">
        <classpath refid="classpath" />
        </javac>
</target>
```

As you can see this target is dependant on the “init” target, which sets up the directory for compiling your classes.

The final step is to make the executable jar file which can be used to make new releases for your project. In order to make this possible add a new “jar” target, like this, into your file:

```
<target name="jar" depends="compile">
    <mkdir dir="${jar.dir}"/>
        <jar destfile="${jar.dir}/${project.name}.myjar.jar" basedir="${classes.dir}">
                <manifest>
                        <attribute name="Main-Class" value="MyMainClass.Main"/>
                </manifest>
    </jar>
</target>
```

The tricky part here is to get your “manifest” right. The most important attribute in the manifest is to specify your “Main-Class” which we, in this case, named “MyMainClass.Main”.

In order to execute all the targets in a final sequence we need to make a target that executes the targets in the right order. This can be done by adding this last thing into your script file:

```
<target name="main" depends="clean, jar"/>
```

As you can see, we specified a new target called “main”, which just calls two of our targets “clean” and “jar”; however, these targets are dependent on the other targets which means that they will also be executed in the right order.

When your are finished you should have a script file that looks something like this:

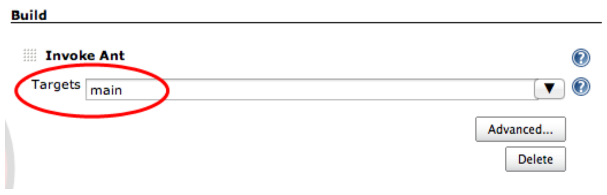
```
<project name="MyProjectName" basedir=".">

<property name="project.name" value="MyProjectName"/>

<property name="lib.dir" value="lib"/>
<property name="src.dir" value="src"/>
<property name="build.dir" value="build"/>
<property name="classes.dir" value="${build.dir}/classes"/>
<property name="jar.dir" value="${build.dir}/jar"/>

<path id="classpath">
       <pathelement location="${lib.dir}/junit-4.0.jar" />
</path>


<target name="init">
    <mkdir dir="${build.dir}"/>
</target>

<target name="clean">
    <delete dir="${build.dir}"/>
</target>

<target name="compile" depends="init">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${src.dir}" destdir="${classes.dir}">
        <classpath refid="classpath" />
        </javac>
</target>

<target name="jar" depends="compile">
    <mkdir dir="${jar.dir}"/>
    <jar destfile="${jar.dir}/${project.name}.myjar.jar" basedir="${classes.dir}">
                <manifest>
                        <attribute name="Main-Class" value="MyMainClass.Main"/>
                </manifest>
    </jar>
</target>

<target name="main" depends="clean, jar"/>
```

You can try executing the script by typing this into your console when your are located in the folder were the script file is located:

Executing buildingscript

```
ant main
```

When you got a running building-script your are ready to configure it with Jenkins. In order to let Jenkins find your script file you need to **commit** your script file into your repository root directory. Then **re-open** Jenkins and go into your project's configuration page again. Scroll down until you find **Build** and press the **Add build step** button and pick the **Invoke Ant** option. This will add a new Ant-build into your project. Type **Main** into the **Target** textfield like in figure 9.



Targets: main

Figure 9:

Congratulations, you should now have a very basic Jenkins environment. Try your first build by pressing **Build Now** in the menu. If your script works and no errors occurs you should get your first blue ball into the build history, which means that you got your first successfully build.

4 Setting up your Jenkins plugins

What makes Jenkins so powerful is the customization possibilities you have got with all the thousands of different plugins. In this part we will show you how to implement the ones that we find most important.

4.1 Javadoc

In order to be able to publish *Javadoc* with Jenkins we need to generate them. To generate them we are going to use Ant once again. Start by **re-open** your **building-script file** and add a new target by adding this to your script-file:

```
<target name="doc" depends="compile">
    <mkdir dir="${doc.dir}"/>
                <javadoc
                destdir="${doc.dir}"
                verbose="true"
                        author="true"
                        version="true">
                <classpath refid="classpath" />
                    <packageset dir="${src.dir}" defaultexcludes="yes">
                        <include name="**" />
                        <exclude name="**/tests/" />
                    </packageset>
                </javadoc>
</target>
```

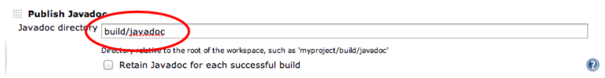
This script generates a new Javadoc into your "doc.dir" from the source code. The tag "packageset" is used to include or exclude files from the generation. As you can see in this case we include all source code except the "test" package.

You can try your new target by typing:

Executing doc target

```
ant clean doc
```

Now you are ready to configured Jenkins to be able to publish your generated Javadoc. **Re-Open** Jenkins and browse to your projects **configuration page**. Scroll down to **Post-build Actions** section and press the **Add post-build action** button and select the **Publish Javadoc** option. **Input** *build/javadoc* into the "Javadoc directory" textfield like in figure 12.



Publish Javadoc

Javadoc directory: build/javadoc

Directory relative to the root of the workspace, such as 'myproject/build/javadoc'

☐ Retain Javadoc for each successful build

Figure 10:

In order to browse your published Javadoc press the **Javadoc** button on the project menu like in figure 11.

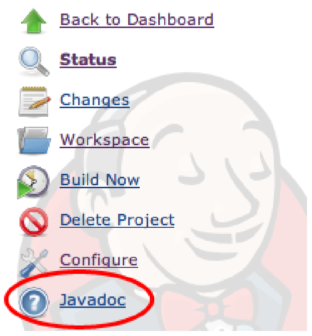


Figure 11:

4.2 Test-reports

Like Javadoc described in section 4.1 we need to generate test-reports before publishing them with Jenkins. To do this we are going to use our famous building-script again. **Re-open** your **building-script** with your favorite XML editor and add a new test-report target to your file like this:

```
<target name="report-test" depends="compile">
    <mkdir dir="${test-reports.dir}"/>
        <mkdir dir="${test-reports-xml.dir}"/>
        <mkdir dir="${test-reports-html.dir}"/>
</target>
```

The newly added target "report-test" is a little more complicated than our last added target which we handled in section 4.1. In order to generate our report we need to run our tests. We are doing this by adding the "junit" task into our target, like this:

```
    <junit printsummary="on" haltonfailure="false">
```

```
                <classpath>
        <path refid="project.classpath" />
        </classpath>

                <formatter type="xml"/>
                <batchtest todir="${test-reports-xml.dir}">
                        <fileset dir="${src.dir}">
                                <include name="**/*Test.java"/>
                        </fileset>
                </batchtest>
        </junit>
```

The first thing to notice is that we are using a new class-path called "project.classpath", this class-path needs to be added into our script file as well in order to get the target running. Do that by adding this to your script-file:

```
<path id="project.classpath">
    <pathelement path="${classes.dir}"/>
    <path refid="classpath" />
</path>
```

One more important thing to note here is the "fileset" tag that specifies which files that contains the test. In this case we are running all source files in all packages that ends with "Test.java". You can add exclude-tags as well in order to exclude files from your testing task.

In order to make the reports we are looking for, we need to add the new task "junitreport" into our target, by adding this to your target as well:

```
    <junitreport todir="${test-reports-xml.dir}">
        <fileset dir="${test-reports-xml.dir}">
                        <include name="TEST-*.xml"/>
                </fileset>
                <report format="frames" todir="${test-reports-html.dir}"/>
        </junitreport>
        <delete file="${test-reports-xml.dir}/TESTS-TestSuites.xml"/>
```

This task creates reports-files of your liking, we are generating XML files and HTML files. Jenkins is depending on the XML-files in order to be able to publish the test reports.

When you are finished you should have a new target that looks something like this:

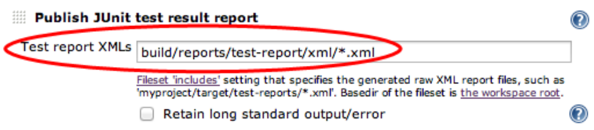
```
<target name="report-test" depends="compile">
    <mkdir dir="${test-reports.dir}"/>
    <mkdir dir="${test-reports-xml.dir}"/>
        <mkdir dir="${test-reports-html.dir}"/>

    <junit printsummary="on" haltonfailure="false">

        <classpath>
        <path refid="project.classpath" />
        </classpath>

                <formatter type="xml"/>
                <batchtest todir="${test-reports-xml.dir}">
                        <fileset dir="${src.dir}">
                                <include name="**/*Test.java"/>
                        </fileset>
                </batchtest>
        </junit>

    <junitreport todir="${test-reports-xml.dir}">
        <fileset dir="${test-reports-xml.dir}">
                        <include name="TEST-*.xml"/>
                </fileset>
                <report format="frames" todir="${test-reports-html.dir}"/>
        </junitreport>
        <delete file="${test-reports-xml.dir}/TESTS-TestSuites.xml"/>
</target>
```

To test your new target you can run this terminal command: **Executing report-test target**

```
ant clean report-test
```

When you have gotten your "report-test" target running you are ready to configure your Jenkins server once again. **Re-open** Jenkins and go to your project **configuration page** and scroll down to **Post-build Actions** and press the **Add post-build action** button and select the option **Publish JUnit test result report**.

Put in **build/reports/test-report/xml/*.xml** into the **Test report XMLs** text field, like in figure 12. This is were your XML reports will be generated from your building-script. Press **Save** and try out your new configuration by pressing **Build Now**.



Publish JUnit test result report

Test report XMLs: build/reports/test-report/xml/*.xml

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is the workspace root.

☐ Retain long standard output/error

Figure 12:

You will now be able to see some "Test Result Trend" for your project.

4.3 Test-Coverage

To be able to get a test-coverage, Ant needs to use an external library for generating test-coverage. There are a couple of different libraries out there, we are going to use *Emma*, which is available here `http://emma.sourceforge.net`. **Download** Emma lib files and **extract** the content of the zip file. The package should contain a "emma_ant.jar" file and a "emma.jar" file, these are your library files that are used by Java and Ant. **Create** a new **folder** in your **lib** folder and name it **emma** and **copy** these two files into your **lib/emma** folder.

Now you are ready to modify your build-script once again to be able to publish your reports with Jenkins. **Open** your **building-script** with your favorite XML editor.

Before you start making your new targets you need to add some properties used by Emma by adding these properties to your script file:

```
<property name="emma.dir" value="${build.dir}/emma"/>
<property name="emma-coverage.file" value="${emma.dir}/coverage.emma"/>
<property name="instrument.dir" value="${emma.dir}/instrument"/>
```

The first thing we have to do is to make Emma available for Ant, this is done by adding a new "task definition" for Emma, by adding this to your script:

```
<taskdef resource="emma_ant.properties" classpathref="emma.classpath" />
```

As you can see this task definition is using a new class-path called "emma.classpath", this is were your newly added emma library files are located. Add the new class-path by adding this to your script:

```
<path id="emma.classpath" >
    <pathelement location="${lib.dir}/emma/emma.jar" />
    <pathelement location="${lib.dir}/emma/emma_ant.jar" />
</path>
```

You are now ready to start making your targets. The first target we need is the "instrument" target, this specially compiled source code which is able to report test-coverage on your source code. Add the "instrument" target by adding this to your script-file:

```
<target name="instrument" depends="compile">
    <mkdir dir="${instrument.dir}"/>
    <emma enabled="true" >
      <instr instrpathref="project.classpath"
             destdir="${instrument.dir}"
             metadatafile="${emma-coverage.file}"
             merge="true"
      >
      <filter excludes="org.*, junit.*, *.tests.*" />
      </instr>
    </emma>
</target>
```

The important part to note here is the filter used by the emma task. This is were you can exclude files from your test-coverage reports. You need to exclude the standard libraries used in the "project.classpath" like the "org" and "junit" library, we also excluded all the test files.

Now we need to modify our "report-test" target to be able to calculate the coverage when running the tests. Start by adding this tag into your target that creates a new temporary emma folder:

```
<mkdir dir="${emma.dir}"/>
```

To be able to "fork" our jUnit task we need to add these attributes, change your jUnit task tag to this:

```
<junit fork="yes" printsummary="on" haltonfailure="false" forkmode="once">
```

"forkmode=once" are only usable with the later jUnit version, this means that the forking will only be done once which save us a lot of time when running the script.

To make the jUnit task able to use emma and the compiled instruments we need to modify the jUnit's task "classpath" tag, change this tag to look like this instead:

```
<classpath>
    <pathelement location="${instrument.dir}" />
    <path refid="project.classpath" />
    <path refid="emma.classpath" />
</classpath>
```

Now we need to modify in our jUnit task is to add the “jvmarg” tags that indicated that we should do the coverage when running the task; do this by adding this lines to your jUnit task:

```
<classpath>
    <jvmarg value="-Demma.coverage.out.file=${emma-coverage.file}" />
     <jvmarg value="-Demma.coverage.out.merge=true" />
</classpath>
```

The last thing is to modify our “report-test” target to be dependent on the “instrument” target, this is done by changing the target tag to this:

```
<target name="report-test" depends="instrument">
```

Your modified “report-test” target should now look like this:

```
<target name="report-test" depends="instrument">
    <mkdir dir="${test-reports.dir}"/>
        <mkdir dir="${test-reports-xml.dir}"/>
        <mkdir dir="${test-reports-html.dir}"/>
        <mkdir dir="${emma.dir}"/>

        <junit fork="yes" printsummary="on" haltonfailure="false" forkmode="once">

                <classpath>
        <pathelement location="${instrument.dir}" />
        <path refid="project.classpath" />
        <path refid="emma.classpath" />
        </classpath>

                <formatter type="xml"/>
                <batchtest todir="${test-reports-xml.dir}">
                        <fileset dir="${src.dir}">
                                <include name="**/*Test.java"/>
                        </fileset>
                </batchtest>
                <jvmarg value="-Demma.coverage.out.file=${emma-coverage.file}" />
      <jvmarg value="-Demma.coverage.out.merge=true" />
        </junit>

        <junitreport todir="${test-reports-xml.dir}">
                <fileset dir="${test-reports-xml.dir}">
                        <include name="TEST-*.xml"/>
                </fileset>
                <report format="frames" todir="${test-reports-html.dir}"/>
        </junitreport>
        <delete file="${test-reports-xml.dir}/TESTS-TestSuites.xml"/>
</target>
```

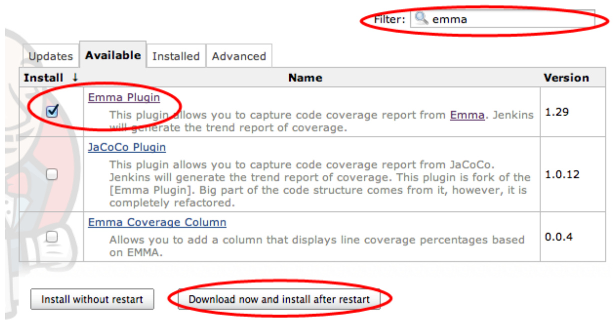
Try your new modification of the build-script by typing this command into your terminal:

Executing coverage-reporting

```
ant clean report-test
```

When you got the script running you are ready to configure Jenkins to publish your Emma reports, but before we can configure our project to publish the Emma reports we need to install the Emma plugin.

To do this go to the Jenkins **index page** and click on **Manage Jenkins** in the main menu, then click on **Manage Plugins**. This is your page for managing all your Jenkins plugins, here you can update, manage and install new plugins. Click on the **available** column and write **emma** into the **filter** text field. This will filter out all the plugins to ease the finding of the emma plugin. You should now be able to **check** the box to the **Emma Plugin** and click on **Download now and install after restart** button like in figure 13.



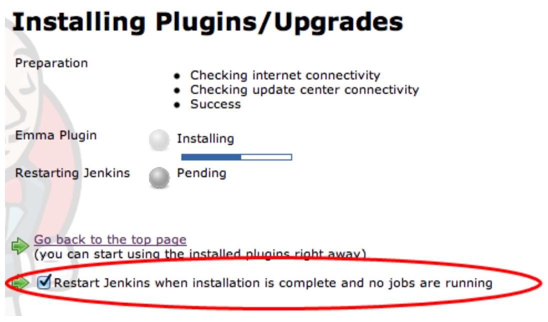
Filter: emma

Updates | Available | Installed | Advanced

Install ↓	Name	Version
✓	Emma Plugin This plugin allows you to capture code coverage report from Emma. Jenkins will generate the trend report of coverage.	1.29
☐	JaCoCo Plugin This plugin allows you to capture code coverage report from JaCoCo. Jenkins will generate the trend report of coverage. This plugin is fork of the [Emma Plugin]. Big part of the code structure comes from it, however, it is completely refactored.	1.0.12
☐	Emma Coverage Column Allows you to add a column that displays line coverage percentages based on EMMA.	0.0.4

Figure 13:

This will take you to the "Installing Plugins/Upgrades" page like in figure 14. **Check** the box **Restart Jenkins when installation is complete and no jobs are running**, and then relax and let Jenkins do all the work.



Installing Plugins/Upgrades

Preparation
- Checking internet connectivity
- Checking update center connectivity
- Success

Emma Plugin Installing

Restarting Jenkins Pending

Go back to the top page
(you can start using the installed plugins right away)

✓ Restart Jenkins when installation is complete and no jobs are running

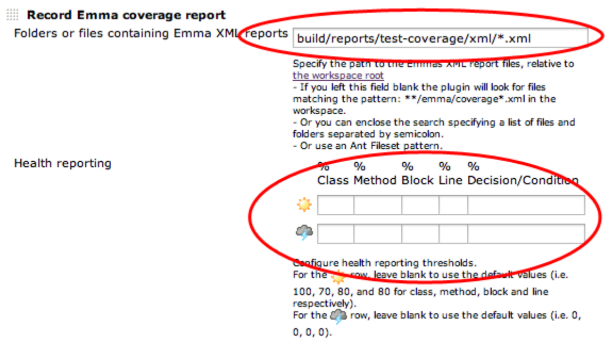
Figure 14:

If you are experiencing problems with the auto-restart, try to start and stop Jenkins in the same way as explained in section 2.1.1.

When your plugin has successfully been installed and your Jenkins have been restarted you are ready to configure your project again. Navigate to the project **configuration page** and scroll down to **Post-build Actions** and press the **Add post-build action** button and select the option **Record Emma coverage Report**.

Start by specified were your generated XML reports are located by typing **build/reports/test-coverage/xml/*.xml** into the **Folders or files containing Emma XML reports** text-field like in figure 15.

One important part to note here is that you can set threshold points for the coverage. By leaving the field blank like in figure 15 you set the default values (100, 70, 80, and 80 for class, method, block and line respectively).



Record Emma coverage report

Folders or files containing Emma XML reports: build/reports/test-coverage/xml/*.xml

Specify the path to the Emmas XML report files, relative to the workspace root
- If you left this field blank the plugin will look for files matching the pattern: **/emma/coverage*.xml in the workspace.
- Or you can enclose the search specifying a list of files and folders separated by semicolon.
- Or use an Ant Fileset pattern.

Health reporting

	% Class	% Method	% Block	% Line	% Decision/Condition
☀					
⛈					

Configure health reporting thresholds.
For the ☀ row, leave blank to use the default values (i.e. 100, 70, 80, and 80 for class, method, block and line respectively).
For the ⛈ row, leave blank to use the default values (i.e. 0, 0, 0, 0).

Figure 15:

You should by now have complete support of test-reporting and test-coverage, the most essential parts of your Jenkins environment. You can see your test and coverage trend in the index page for your project, and for each build.

4.4 FindBugs

Another great plugin to use with your Jenkins environment is the *FindBugs* plugin. FindBugs is a aid for finding defects in your source-code. FindBugs are available here `http://findbugs.sourceforge.net`.

To make Ant able to run FindBugs we need to move the lib files into our lib folder. **Create** a new **folder** in our lib folder and name it **FindBugs** and copy the zip-file content into this folder.

Now we are ready to modify our building-script to generate our FindBugs-reports. Start by **open** the **script-file** once again with your favorite XML editor.

Before we start making our targets we need to add a properties that describes were FindBugs is located by adding this:

```
<property name="findbugs.dir" value="${lib.dir}/findbugs"/>
```

Then we need to add a new "task definition" to our Ant script by adding this:

```
<taskdef name="findbugs" classname="edu.umd.cs.findbugs.anttask.FindBugsTask" classpathref="fin
```

As you can see there is another "classpathref" that needs to be added, called "findbugs.classpath", do this by adding this:

```
<path id="findbugs.classpath" >
        <pathelement location="${findbugs.dir}/lib/ant_findbugs.jar" />
         <pathelement location="${findbugs.dir}/lib/findbugs.jar" />
</path>
```

Now you should be ready to start writing your target that will generate our reports from FindBugs. FindBugs will use the compiled jar files and the source-code and is therefore not needed to be forked into some other task. Add your new "findbugs" target by adding this to your building-script:

```
<target name="findbugs" depends="jar">
    <mkdir dir="${bug-reports.dir}"/>
```

```
            <findbugs home="${findbugs.dir}"
                      output="xml"
                      outputFile="${bug-reports.dir}/findbugs.xml" >
                <auxClasspath refid="classpath" />
              <sourcePath path="${src.dir}" />
                <class location="${jar.dir}/MyJarFile.jar" />
            </findbugs>
</target>
```

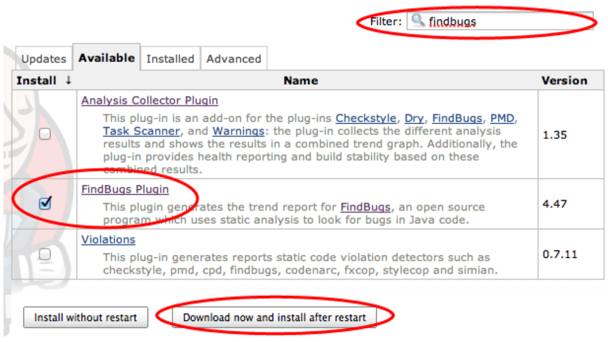
As you can see in the target-tag the "findbugs"-target is dependent on the "jar"-target. But the important thing to note here is the class tag which indicates which jar file that should be executed to be able to find new bugs, therefore you need to change "MyJarFile.jar" into your specific jar file.

Now try to run your new "findbugs" target by typing this command into your terminal: **Executing coverage-reporting**

```
ant clean findbugs
```

When you get your new building-script to work you are ready to configure your Jenkins server to be able to publishing your reports.

First we need to install the FindBugs plugin; follow the procedure done in section 4.3, which the exception to **filter** for **FindBugs** instead like in figure 4.5.



Filter: findbugs

Updates | **Available** | Installed | Advanced

Install ↓	Name	Version
☐	Analysis Collector Plugin This plug-in is an add-on for the plug-ins Checkstyle, Dry, FindBugs, PMD, Task Scanner, and Warnings: the plug-in collects the different analysis results and shows the results in a combined trend graph. Additionally, the plug-in provides health reporting and build stability based on these combined results.	1.35
☑	FindBugs Plugin This plugin generates the trend report for FindBugs, an open source program which uses static analysis to look for bugs in Java code.	4.47
☐	Violations This plug-in generates reports static code violation detectors such as checkstyle, pmd, cpd, findbugs, codenarc, fxcop, stylecop and simian.	0.7.11

Figure 16:

When you have installed the new plugin and restarted Jenkins you are ready to configure your project. **Navigate** to your project's **configuration page** and scroll down to **Post-build Actions** and press the **Add post-build action** button and select the option **Publish FindBugs analysis results**. Then write **build/reports/bug-report/findbugs.xml** into your **FindBugs results** textfield, this is were FindBugs places the generated XML file that need to be published with Jenkins. When you are done click the **save** button, **Build Now** and enjoy the newly generated FindBugs reports.

One thing to take into consideration when you use FindBugs is that this tool is huge and will consume a lot of time depending on how big the projects are. Therefore it is recommended to do some kind of cost estimation before implementing FindBugs.

4.5 Other plugins

Jenkins offers thousands of plugins and we have just tested a few of them. We tried those which we found the most interesting and most relevant for our project. This is a list of which plugins that we found useful in our environment beyond the use of test-reports,test-coverage and FindBugs.

Status Monitor Plugin This plugin allows you to have a dedicated monitor that indicates what state your projects are in, which are very useful for a team whose members are all located in the same room.

Green Balls This plugin changes the blue balls and the blue successful color to green, which we found more logic. However this only changes the Jenkins system and the plugins that uses the standard color. If you use a plugin that has got separate configuration for the color you need to change this as well, like the Status Monitor Plugin that needs some modification in the HTML files.

Jenkins Sound Plugin This is very useful for giving sound feedback to the team. This requires of course that the team is located in the same room.

Edgewall Trac Plugin This lets you add a link to your projects Trac to the menu, which are useful for project that uses Trac like we did.

ChuckNorris Plugin For god sakes, don't forget the Chuck Norris Plugin!