

JENKINS DASHBOARD

<https://plugins.jenkins.io/git/#issues>

<https://medium.com/@knoldus/jenkins-build-triggers-8b7acdb05de1#:~:text=Build%20after%20other%20projects%20are%20built,-If%20your%20project&text=Whenever%20the%20build%20of%20the,then%20this%20project%20build%20invokes.>

what is mean by discard the old projects in Jenkins?

- 1) By default, you can enable the "Discard Old Builds" in each project/job's configuration page. Cleanup per project/job is performed after that job runs. "Discard Old Builds" will perform basic cleanup, using functionality found in Jenkins core.

2) this project is parameterized in Jenkins

Now you have to configure your Jenkins job. First under General section check "This project is parameterized" option and then select String Parameter option by clicking the "Add Parameter" button. Enter Your parameter name (In my case BROWSER) and default value (In my case Firefox) and click on "Apply" button.

4. Setting Parameter Values

So far, we've seen how to define parameters and use them inside our Jenkins jobs. The final step is to pass values for our parameters when we execute jobs.

4.1. Jenkins UI

Starting a job with Jenkins UI is the easiest way to pass build parameters.

All we do is log in, navigate to our job, and click the *Build with Parameters* link:



This will take us to a screen that asks for inputs for each parameter. Based on the type of parameter, the way we input its value will be different.

For example, *String* parameters will show up as a plain text field. *Boolean* parameters will be displayed as a checkbox. And *Choice* parameters are displayed as a dropdown list:

Project parameterized-example

This build requires parameters:

packageType

Type of artifact to build

jdkVersion

Version of Java compiler to use

☐ debug

Whether to enable debug logging

Once we provide a value for each parameter, all we have to do is click the *Build* button, and Jenkins begins executing the job.

Remote Execution

Jenkins jobs can also be executed with a remote API call. We do this by calling a special URL for the job on our Jenkins server:

```
http://<JENKINS_URL>/job/<JOB_NAME>/buildWithParameters/packageT  
ype=war&jdkVersion=11&debug=true
```

Plugin Information. View Throttle Concurrent Builds on the plugin site for more information. This plugin allows for throttling the number of concurrent builds of a project running per node or globally. To set an unlimited value of concurrent builds for the project or node, use

Jenkins allows for parallel execution of builds for a Job. Job configuration page has a check box, "Execute concurrent builds if necessary". Also, in the master node configuration set the "# of executors" field to more than 1. Once these two are done, parallel job execution is enabled.

BUILD TRIGGER REMOTELY IN DASHBOARD

I believe the answer lies within the security settings. The purpose of the Authentication Token is to allow unauthorized users (developers) to trigger a build without having login access to Jenkins (see <https://wiki.jenkins-ci.org/display/JENKINS/Authenticating+scripted+clients> and <https://wiki.jenkins-ci.org/display/JENKINS/Quick+and+Simple+Security>).

These are the most common Jenkins build triggers:

- Trigger builds remotely
- Build after other projects are built
- Build periodically
- GitHub hook trigger for GITScm polling
- Poll SCM

1. Trigger builds remotely :

If you want to trigger your project built from anywhere anytime then you should select Trigger builds remotely option from the build triggers.

You'll need to provide an authorization token in the form of a string so that only those who know it would be able to remotely trigger this project's builds. This provides the predefined URL to invoke this trigger remotely.

predefined URL to trigger build remotely:

JENKINS_URL/job/JobName/build?token=TOKEN_NAME

JENKINS_URL: the IP and PORT which the Jenkins server is running

TOKEN_NAME: You have provided while selecting this build trigger.//Example:

<http://e330c73d.ngrok.io/job/test/build?token=12345>

Whenever you will hit this URL from anywhere you project build will start.

2. Build after other projects are built

If your project depends on another project build then you should select Build after other projects are built option from the build triggers.

In this, you must specify the project(Job) names in the Projects to watch field section and select one of the following options:

1. Trigger only if the build is stable

Note: A build is stable if it was built successfully and no publisher reports it as unstable 2. Trigger even if the build is unstable

Note: A build is unstable if it was built successfully and one or more publishers report it unstable 3. Trigger even if the build fails

After that, It starts watching the specified projects in the Projects to watch section.

Whenever the build of the specified project completes (either is stable, unstable or failed according to your selected option) then this project build invokes.

Build periodically:

If you want to schedule your project build periodically then you should select the Build periodically option from the build triggers.

You must specify the periodical duration of the project build in the scheduler field section

This field follows the syntax of cron (with minor differences). Specifically, each line consists of 5 fields separated by TAB or whitespace:

```
MINUTE HOUR DOM MONTH DOW
```

MINUTE Minutes within the hour (0–59) HOUR The hour of the day (0–23)
DOM The day of the month (1–31) MONTH The month (1–12) DOW The day of the week (0–7) where 0 and 7 are Sunday.

To specify multiple values for one field, the following operators are available. In the order of precedence,

- * specifies all valid values
- M-N specifies a range of values
- M-N/X or */X steps by intervals of X through the specified range or whole valid range
- A,B,...,Z enumerates multiple values

Examples:

```
# every fifteen minutes (perhaps at :07, :22, :37, :52)
```

```
H/15 * * * *
```

```
# every ten minutes in the first half of every hour (three times, perhaps at :04,
```

:14, :24)

H(0-29)/10 * * * *

once every two hours at 45 minutes past the hour starting at 9:45 AM and finishing at 3:45 PM every weekday.

45 9-16/2 * * 1-5

once in every two hours slot between 9 AM and 5 PM every weekday
(perhaps at 10:38 AM, 12:38 PM, 2:38 PM, 4:38 PM)

H H(9-16)/2 * * 1-5

once a day on the 1st and 15th of every month except December

H H 1,15 1-11 *

After successfully scheduled the project build then the scheduler will invoke the build periodically according to your specified duration.

GitHub webhook trigger for GITScm polling:

A webhook is an HTTP callback, an HTTP POST that occurs when something happens through a simple event-notification via HTTP POST.

GitHub webhooks in Jenkins are used to trigger the build whenever a developer commits something to the branch.

Let's see how to add build a webhook in GitHub and then add this webhook in Jenkins.

1. Go to your project repository.

2. Go to "settings" in the right corner.

3. Click on "webhooks."

4. Click "Add webhooks."

5. Write the Payload URL as

```
http://e330c73d.ngrok.io/github-webhook
```

```
//This URL is a public URL where the Jenkins server is running
```

Here <https://e330c73d.ngrok.io/> is the IP and port where my Jenkins is running.

If you are running Jenkins on localhost then writing <https://localhost:8080/github-webhook/> will not work because Webhooks can only work with the public IP.

So if you want to make your localhost:8080 expose public then we can use some tools.

In this example, we used ngrok tool to expose my local address to the public.

To know more on how to add webhook in Jenkins pipeline, visit: <https://blog.knoldus.com/opsinit-adding-a-github-webhook-in-jenkins-pipeline/>

Poll SCM:

Poll SCM periodically polls the SCM to check whether changes were made (i.e. new commits) and builds the project if new commits were pushed since the last build.

You must schedule the polling duration in the scheduler field. Like we explained above in the Build periodically section. You can see the Build periodically section to know how to schedule.

After successfully scheduled, the scheduler polls the SCM according to your specified duration in scheduler field and builds the project if new commits were pushed since the last build.

This will scan all the jobs that:

- Have Build Triggers > Poll SCM enabled. No polling schedule is required.
- Are configured to build the repository at the specified URL

For jobs that meet these conditions, polling will be triggered. If polling finds a change worthy of a build, a build will be triggered.

This allows a notify script to remain the same for all Jenkins jobs. Or if you have multiple repositories under a single repository host application (such as Gitosis), you can share a single post-receive hook script with all the repositories. Finally, this URL doesn't require authentication even for secured Jenkins, because the server doesn't directly use anything that the client is sending. It polls to verify that there is a change before it actually starts a build.

When notifyCommit is successful, the list of triggered projects is returned.

Enabling JGit

See [the git client plugin documentation](#) for instructions to enable JGit. JGit becomes available throughout Jenkins once it has been enabled.

Global Configuration

Dashboard > configuration

Git plugin

Global Config user.name Value ?

Global Config user.email Value ?

☒ Create new accounts based on author/committer's email ?
| ☒ Use existing account with same email if found ?
|

☒ Show the entire commit summary in changes ?
☒ Hide credential usage in job output ?
☒ Disable performance enhancements ?
☒ Preserve second fetch during checkout ?
☒ Add git tag action to jobs ?

In the **Configure System** page, the Git Plugin provides the following options:

Global Config user.name Value

Defines the default git user name that will be assigned when git commits a change from Jenkins. For example, Janice Examplesperson. This can be overridden by individual projects with the [Custom user name/e-mail address](#) extension.

Global Config user.email Value

Defines the default git user e-mail that will be assigned when git commits a change from Jenkins. For example, janice.examplesperson@example.com. This can be overridden by individual projects with the [Custom user name/e-mail address](#) extension.

Create new accounts based on author/committer's email

New user accounts are created in Jenkins for committers and authors identified in changelogs. The new user accounts are added to the internal Jenkins database. The e-mail address is used as the id of the account.

Show the entire commit summary in changes

The changes page for each job would truncate the change summary prior to git plugin 4.0. With the release of git plugin 4.0, the default was changed to show the complete change summary. Administrators that want to restore the old behavior may disable this setting.

Hide credential usage in job output

If checked, the console log will not show the credential identifier used to clone a repository.

Disable performance enhancements

If JGit and command line git are both enabled on an agent, the git plugin uses a "git tool chooser" to choose a preferred git implementation. The preferred git implementation depends on the

size of the repository and the git plugin features requested by the job. If the repository size is less than the JGit repository size threshold and the git features of the job are all implemented in JGit, then JGit is used. If the repository size is greater than the JGit repository size threshold or the job requires git features that are not implemented in JGit, then command line git is used.

Preserve second fetch during initial checkout

If checked, the initial checkout step will not avoid the second fetch. Git plugin versions prior to git plugin 4.4 would perform two fetch operations during the initial repository checkout. Git plugin 4.4 removes the second fetch operation in most cases. Enabling this option will restore the second fetch operation. This setting is only needed if there is a bug in the redundant fetch removal logic. If you enable this setting, please report a git plugin issue that describes why you needed to enable it.

Add git tag action to jobs

If checked, the git tag action will be added to any builds that happen after the box is checked. Prior to git plugin 4.5.0, the git tag action was always added. Git plugin 4.5.0 and later will not add the git tag action to new builds unless the administrator enables it.

The git tag action allows a user to apply a tag to the git repository in the workspace based on the git commit used in the build applying the tag. The git plugin does not push the applied tag to any other location. If the workspace is removed, the tag that was applied is lost.

Repository Browser

The screenshot shows the Jenkins configuration page for Source Code Management. The 'Repository browser' dropdown menu is open, displaying a list of options: (Auto), AssemblaWeb, FishEye, Kiln, Microsoft Team Foundation Server/Visual Studio Team Services, bitbucketweb, cgit, gitblit, and githubweb. The 'Build Triggers' section on the left includes checkboxes for 'Build after other projects', 'Build periodically', 'Build when another project is built', and 'Poll SCM'. The 'Additional Behaviours' section includes a radio button for 'Multiple SCMs'.

A Repository Browser adds links in "changes" views within Jenkins to an external system for browsing the details of those changes. The "Auto" selection attempts to infer the repository browser from the "Repository URL" and can detect cloud versions of GitHub, Bitbucket and GitLab.

Repository browsers include:

AssemblaWeb

The screenshot shows the Jenkins configuration page for Source Code Management with 'AssemblaWeb' selected as the 'Repository browser'. Below this, the 'Assembla Git URL' field is populated with the value 'https://app.assembla.com/spaces/git-plugin/git/source'.

Repository browser for git repositories hosted by [Assembla](#) Options include:

Assembla Git URL

Root URL serving this Assembla repository. For example, <https://app.assembla.com/spaces/git-plugin/git/source>

FishEye



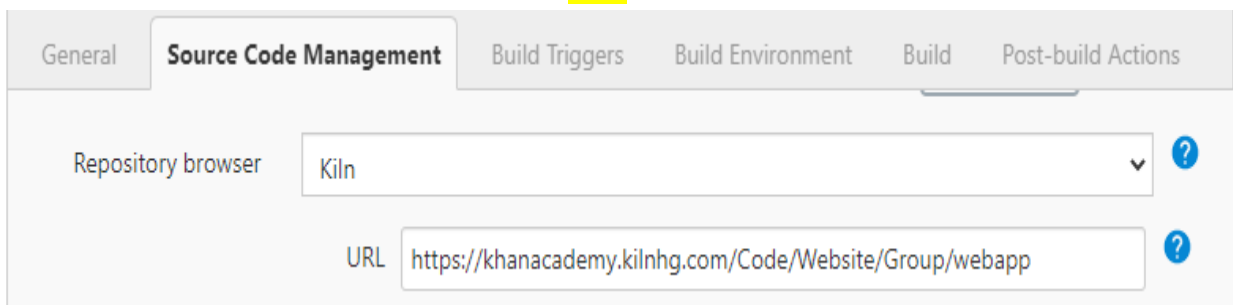
The screenshot shows the Jenkins configuration page for Source Code Management. The 'Source Code Management' tab is selected. Under 'Repository browser', a dropdown menu is set to 'FishEye'. Below it, the 'URL' field contains the text 'https://source.jboss.org/browse/Forge'. Both fields have a blue question mark icon to their right.

Repository browser for [git repositories hosted by Atlassian Fisheye](#)
Options include:

URL

Root URL serving this FishEye repository. For example, <https://fisheye.example.com/browser/my-project>

Kiln



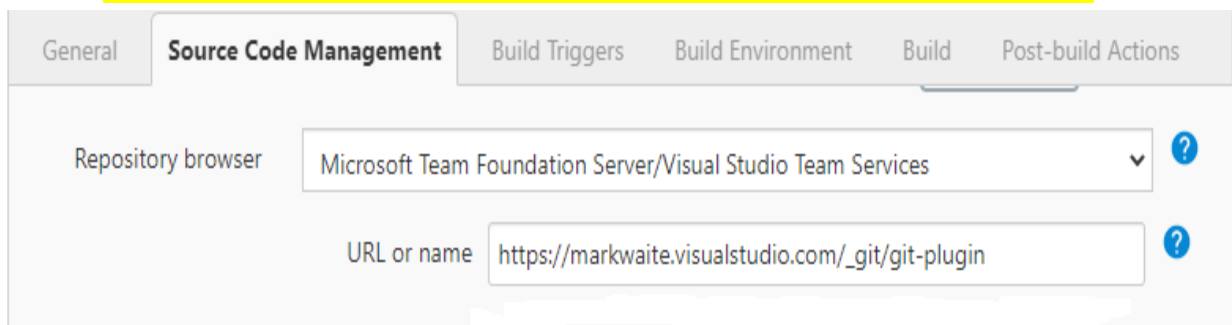
The screenshot shows the Jenkins configuration page for Source Code Management. The 'Source Code Management' tab is selected. Under 'Repository browser', a dropdown menu is set to 'Kiln'. Below it, the 'URL' field contains the text 'https://khanacademy.kilnhg.com/Code/Website/Group/webapp'. Both fields have a blue question mark icon to their right.

Repository browser for git repositories hosted by [Kiln](#). Options include:

URL

Root URL serving this Kiln repository. For example, <https://kiln.example.com/username/my-project>

Microsoft Team Foundation Server/Visual Studio Team Services



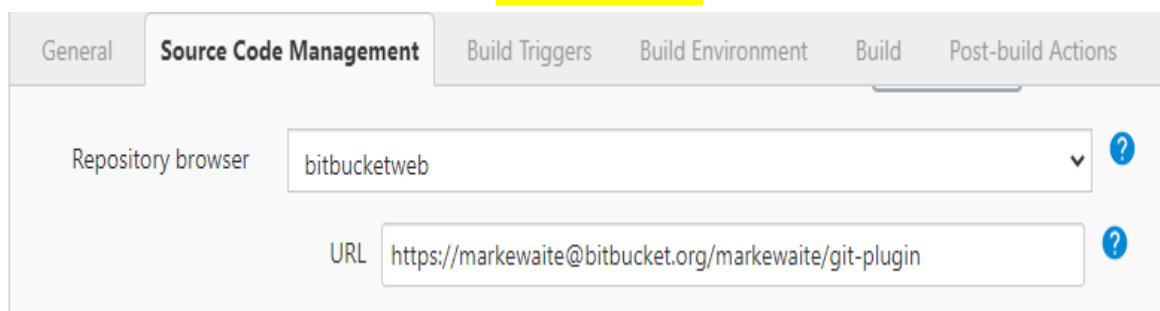
The screenshot shows the 'Source Code Management' tab in a settings interface. The 'Repository browser' dropdown is set to 'Microsoft Team Foundation Server/Visual Studio Team Services'. The 'URL or name' text box contains the URL 'https://markwaite.visualstudio.com/_git/git-plugin'. There are blue question mark icons next to both fields.

Repository browser for git repositories hosted by [Azure DevOps](#). Options include:

URL or name

Root URL serving this Azure DevOps repository. For example, https://example.visualstudio.com/_git/my-project.

bitbucketweb



The screenshot shows the 'Source Code Management' tab in a settings interface. The 'Repository browser' dropdown is set to 'bitbucketweb'. The 'URL' text box contains the URL 'https://markewaite@bitbucket.org/markewaite/git-plugin'. There are blue question mark icons next to both fields.

Repository browser for git repositories hosted by [Bitbucket](#). Options include:

URL

Root URL serving this Bitbucket repository. For example, <https://bitbucket.example.com/username/my-project>

bitbucketserver

The screenshot shows the Jenkins configuration page for Source Code Management. The 'Repository browser' dropdown is set to 'bitbucketserver'. The 'URL' field contains 'http://bitbucket:7990/'. There are help icons (question marks) next to both fields.

Repository browser for git repositories hosted by an on-premises Bitbucket Server installation. Options include:

URL

Root URL serving this Bitbucket repository. For example, <https://bitbucket-server:7990/username/my-project>

cgit

The screenshot shows the Jenkins configuration page for Source Code Management. The 'Repository browser' dropdown is set to 'cgit'. The 'URL' field contains 'https://git.zx2c4.com/cgit/'. There is a help icon (question mark) next to the 'Repository browser' field.

Repository browser for git repositories hosted by [cgit](#). Options include:

URL

Root URL serving this cgit repository. For example, <https://git.zx2c4.com/cgit/>

gitblit

The screenshot shows the Jenkins configuration page for the 'Source Code Management' section. The 'Repository browser' dropdown is set to 'gitblit'. Below it, the 'GitBlit root url' is set to 'https://gitblit.example.com/' and the 'Project name in GitBlit' is set to 'git-plugin'. Each field has a blue question mark icon to its right.

GitBlit root url

Root URL serving this GitBlit repository. For example, <https://gitblit.example.com/>

Project name in GitBlit

Name of the GitBlit project. For example, [my-project](#)

githubweb

The screenshot shows the Jenkins configuration page for the 'Source Code Management' section. The 'Repository browser' dropdown is set to 'githubweb'. Below it, the 'URL' is set to 'https://github.com/jenkinsci/git-plugin'. Each field has a blue question mark icon to its right.

Repository browser for git repositories hosted by [GitHub](#). Options include:

URL

Root URL serving this GitHub repository. For example, <https://github.example.com/username/my-project>

gitiles

General	Source Code Management	Build Triggers	Build Environment	Build	Post-build Actions
Repository browser		<div>gitiles</div>			
URL		<div>https://gerrit.googlesource.com/gitiles/</div>			

Repository browser for git repositories hosted by [Gitiles](#). Options include:

gitiles root url

Root URL serving this Gitiles repository. For example, <https://gerrit.googlesource.com/gitiles/>

gitlab

General	Source Code Management	Build Triggers	Build Environment	Build	Post-build Actions
Repository browser		<div>gitlab</div>			
URL		<div>https://gitlab.com/MarkEwaite/git-client-plugin</div>			
Version		<div></div>			

Repository browser for git repositories hosted by [GitLab](#). Options include:

URL

Root URL serving this GitLab repository. For example, <https://gitlab.example.com/username/my-project>

Version

Major and minor version of GitLab you use, such as 12.6. If you don't specify a version, a modern version of GitLab (≥ 8.0) is assumed. For example, [12.6](#)

gitlist

The screenshot shows the Jenkins configuration page for Source Code Management. The 'Source Code Management' tab is selected. Under 'Repository browser', the dropdown menu is set to 'gitlist'. Below it, the 'URL' field contains the text 'https://gitlist.example.com/username/my-project'. There are blue question mark icons to the right of both the dropdown and the URL field.

Repository browser for git repositories hosted by [GitList](#). Options include:

URL

Root URL serving this GitList repository. For example, <https://gitlist.example.com/username/my-project>

gitoriousweb

Gitorious was acquired in 2015. This browser is deprecated.

URL

Root URL serving this Gitorious repository. For example, <https://gitorious.org/username/my-project>

gitweb

The screenshot shows the Jenkins configuration page for Source Code Management. The 'Source Code Management' tab is selected. Under 'Repository browser', the dropdown menu is set to 'gitweb'. Below it, the 'URL' field contains the text 'https://gitweb.example.com/username/my-project'. There are blue question mark icons to the right of both the dropdown and the URL field.

Repository browser for git repositories hosted by [GitWeb](#). Options include:

URL

Root URL serving this GitWeb repository. For example, <https://gitweb.example.com/username/my-project>

gogs

General	Source Code Management	Build Triggers	Build Environment	Build	Post-build Actions
<div>Repository browser <input type="text" value="gogs"/></div> <div>URL <input type="text" value="https://gogs.example.com/username/my-project"/></div>					

Repository browser for git repositories hosted by [Gogs](#). Options include:

URL

Root URL serving this Gogs repository. For example, <https://gogs.example.com/username/my-project>

phabricator

General	Source Code Management	Build Triggers	Build Environment	Build	Post-build Actions
<div>Repository browser <input type="text" value="phabricator"/></div> <div>URL <input type="text" value="https://phabricator.example.com/"/></div> <div>Repository name in Phab <input type="text" value="git-plugin"/></div>					

Repository browser for git repositories hosted by [Phacility Phabricator](#). Options include:

URL

Root URL serving this Phabricator repository. For example, <https://phabricator.example.com/>

Repository name in Phab

Name of the Phabricator repository. For example, [my-project](#)

redmineweb

The screenshot shows the Jenkins configuration interface for the 'Source Code Management' tab. The 'Repository browser' dropdown is set to 'redmineweb'. Below it, the 'URL' field contains the text 'https://redmine.example.com/username/projects/my-project/repository'. There are blue question mark icons to the right of both the dropdown and the URL field.

Repository browser for git repositories hosted by [Redmine](#). Options include:

URL

Root URL serving this Redmine repository. For example, <https://redmine.example.com/username/projects/my-project/repository>

rhodecode

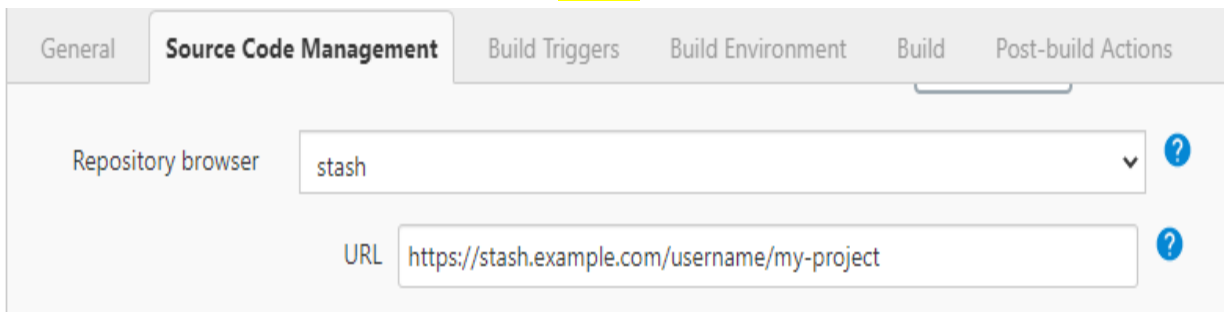
The screenshot shows the Jenkins configuration interface for the 'Source Code Management' tab. The 'Repository browser' dropdown is set to 'rhodecode'. Below it, the 'URL' field contains the text 'https://rhodecode.example.com/username/my-project'. There are blue question mark icons to the right of both the dropdown and the URL field.

Repository browser for git repositories hosted by [RhodeCode](#). Options include:

URL

Root URL serving this RhodeCode repository. For example, <https://rhodecode.example.com/username/my-project>

stash



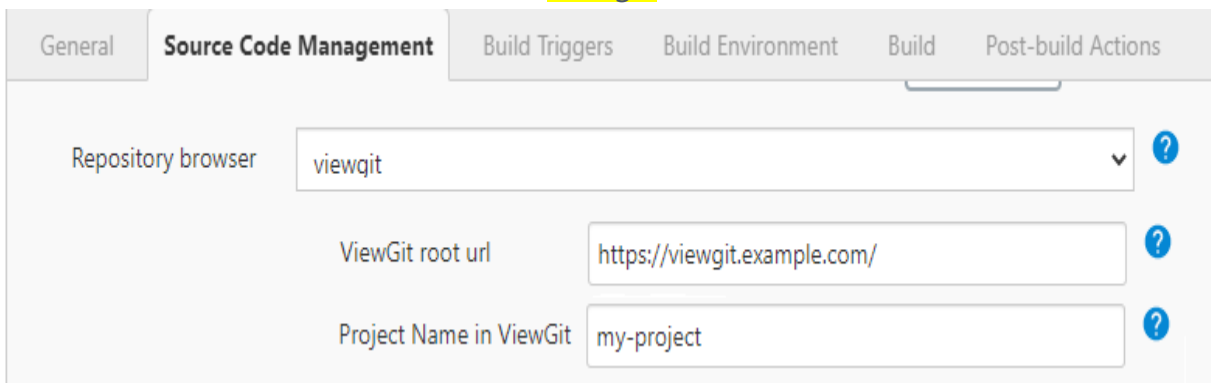
The screenshot shows the Jenkins configuration page for Source Code Management. The 'Source Code Management' tab is selected. Under 'Repository browser', a dropdown menu is set to 'stash'. Below it, the 'URL' field contains 'https://stash.example.com/username/my-project'. Both fields have a blue question mark icon to their right.

Stash is now called BitBucket Server. Repository browser for git repositories hosted by [BitBucket Server](#). Options include:

URL

Root URL serving this Stash repository. For example, <https://stash.example.com/username/my-project>

viewgit



The screenshot shows the Jenkins configuration page for Source Code Management. The 'Source Code Management' tab is selected. Under 'Repository browser', a dropdown menu is set to 'viewgit'. Below it, the 'ViewGit root url' field contains 'https://viewgit.example.com/' and the 'Project Name in ViewGit' field contains 'my-project'. All three fields have a blue question mark icon to their right.

Repository browser for git repositories hosted by [viewgit](#). Options include:

ViewGit root url

Root URL serving this ViewGit repository. For example, <https://viewgit.example.com/>

Project Name in ViewGit

ViewGit project name. For example, [my-project](#)

Extensions

Extensions add new behavior or modify existing plugin behavior for different uses. Extensions help users more precisely tune the plugin to meet their needs.

Extensions include:

- [Clone Extensions](#)
- [Checkout Extensions](#)
- [Changelog Extensions](#)
- [Tagging Extensions](#)
- [Build Initiation Extensions](#)
- [Merge Extensions](#)
- [Deprecated Extensions](#)

Clone Extensions

Clone extensions modify the git operations that retrieve remote changes into the agent workspace. The extensions can adjust the amount of history retrieved, how long the retrieval is allowed to run, and other retrieval details.

Advanced clone behaviours

General **Source Code Management** Build Triggers Build Environment Build Post-build Actions

Additional Behaviours **Advanced clone behaviours**

☒ Fetch tags ?

☐ Honor refspec ?
on initial clone

Path of the reference repo to use during clone /cache/repo.git ?

Timeout (in minutes) for clone and fetch operations 37 ?

☒ Shallow clone ?

Shallow clone depth 1 ?

Advanced clone behaviors modify the git clone and git fetch commands.
They control:

- breadth of history retrieval (refspecs)
- depth of history retrieval (shallow clone)
- disc space use (reference repositories)
- duration of the command (timeout)
- tag retrieval

Advanced clone behaviors include:

Honor refspec on initial clone

Perform initial clone using the refspec defined for the repository. This can save time, data transfer and disk space when you only need to access the references specified by the refspec. If this is not enabled, then the plugin default refspec includes all remote branches.

Shallow clone

Perform a shallow clone by requesting a limited number of commits from the tip of the requested branch(es). Git will not download the complete history of the project. This can save time and disk space when you just want to access the latest version of a repository.

Shallow clone depth

Set shallow clone depth to the specified number of commits. Git will only download depth commits from the remote repository, saving time and disk space.

Path of the reference repo to use during clone

Specify a folder containing a repository that will be used by git as a reference during clone operations. This option will be ignored if the folder is not available on the agent.

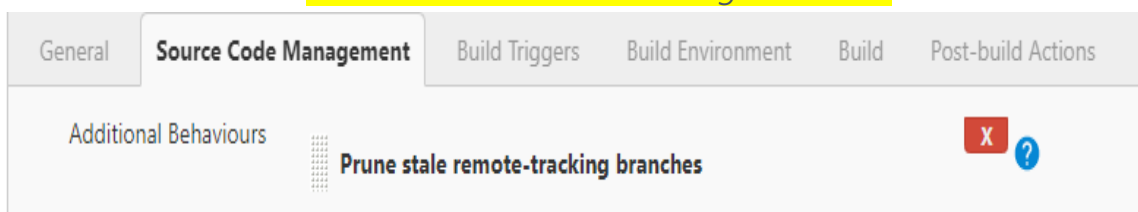
Timeout (in minutes) for clone and fetch operations

Specify a timeout (in minutes) for clone and fetch operations.

Fetch tags

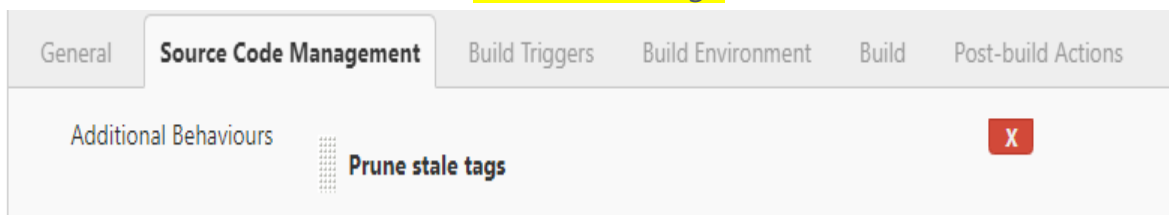
Deselect this to perform a clone without tags, saving time and disk space when you want to access only what is specified by the refspec, without considering any repository tags.

Prune stale remote tracking branches



Removes remote tracking branches from the local workspace if they no longer exist on the remote. See [git remote prune](#) and [git fetch --prune](#) for more details.

Prune stale tags

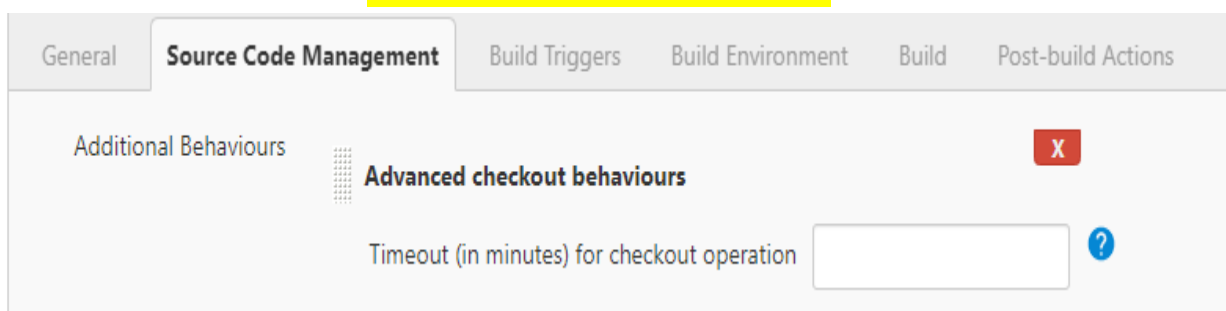


Removes tags from the local workspace before fetch if they no longer exist on the remote. If stale tags are not pruned, deletion of a remote tag will not remove the local tag in the workspace. If the local tag already exists in the workspace, git correctly refuses to create the tag again. Pruning stale tags allows the local workspace to create a tag with the same name as a tag which was removed from the remote.

Checkout Extensions

Checkout extensions modify the git operations that place files in the workspace from the git repository on the agent. The extensions can adjust the maximum duration of the checkout operation, the use and behavior of git submodules, the location of the workspace on the disc, and more.

Advanced checkout behaviors



Advanced checkout behaviors modify the [git checkout](#) command. Advanced checkout behaviors include

Timeout (in minutes) for checkout operation

Specify a timeout (in minutes) for checkout. The checkout is stopped if the timeout is exceeded. Checkout timeout is usually only required with slow file systems or large repositories.

Advanced sub-modules behaviours

General **Source Code Management** Build Triggers Build Environment Build Post-build Actions

Additional Behaviours **Advanced sub-modules behaviours**

☐ Disable submodules processing

☒ Recursively update submodules

☐ Update tracking submodules to tip of branch

☒ Use credentials from default remote of parent repository

Path of the reference repo to use during submodule update

Timeout (in minutes) for submodules operations

Number of threads to use when updating submodules

☒ Shallow clone

Shallow clone depth

Advanced sub-modules behaviors modify the [git submodule](#) commands.

They control:

- depth of history retrieval (shallow clone)
- disc space use (reference repositories)
- credential use
- duration of the command (timeout)
- concurrent threads used to fetch submodules

Advanced sub-modules include:

Disable submodules processing

Ignore submodules in the repository.

Recursively update submodules

Retrieve all submodules recursively. Without this option, submodules which contain other submodules will ignore the contained submodules.

Update tracking submodules to tip of branch

Retrieve the tip of the configured branch in .gitmodules.

Use credentials from default remote of parent repository

Use credentials from the default remote of the parent project. Submodule updates do not use credentials by default. Enabling this extension will provide the parent repository credentials to each of the submodule repositories. Submodule credentials require that the submodule repository must accept the same credentials as the parent project. If the parent project is cloned with https, then the authenticated submodule references must use https as well. If the parent project is cloned with ssh, then the authenticated submodule references must use ssh as well.

Path of the reference repo to use during submodule update

Folder containing a repository that will be used by git as a reference during submodule clone operations. This option will be ignored if the folder is not available on the agent running the build. A reference repository may contain multiple subprojects. See the combining repositories section for more details.

Timeout (in minutes) for submodule operations

Specify a timeout (in minutes) for submodules operations. This option overrides the default timeout.

Number of threads to use when updating submodules

Number of parallel processes to be used when updating submodules. Default is to use a single thread for submodule updates

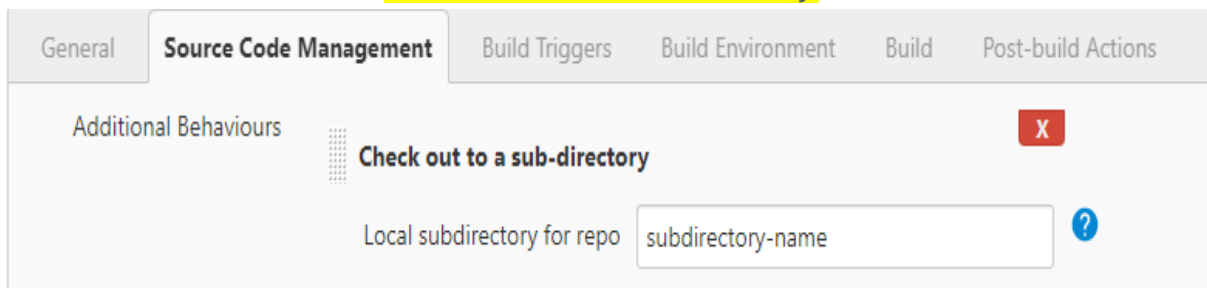
Shallow clone

Perform shallow clone of submodules. Git will not download the complete history of the project, saving time and disk space.

Shallow clone depth

Set shallow clone depth for submodules. Git will only download recent history of the project, saving time and disk space.

Checkout to a sub-directory



The screenshot shows the Jenkins configuration interface for Source Code Management. The 'General' tab is selected. Under 'Additional Behaviours', the 'Check out to a sub-directory' option is checked, indicated by a red 'X' icon. Below this, there is a text input field labeled 'Local subdirectory for repo' with the value 'subdirectory-name' entered. A blue question mark icon is visible next to the input field.

Checkout to a subdirectory of the workspace instead of using the workspace root.

This extension should not be used in Jenkins Pipeline (either declarative or scripted). Jenkins Pipeline already provides standard techniques for checkout to a subdirectory. Use `ws` and `dir` in Jenkins Pipeline rather than this extension.

Local subdirectory for repo

Name of the local directory (relative to the workspace root) for the git repository checkout. If left empty, the workspace root itself will be used.

Checkout to specific local branch

The screenshot shows the Jenkins configuration interface for the 'Source Code Management' plugin. The 'Additional Behaviours' section is active, displaying the 'Check out to specific local branch' option. Below this, there is a text input field labeled 'Branch name' containing the value 'develop/new-feature'. The interface includes tabs for 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'. A red 'X' icon and a blue question mark are visible in the top right corner of the configuration area.

Branch name

If given, checkout the revision to build as HEAD on the named branch. If value is an empty string or "***", then the branch name is computed from the remote branch without the origin. In that case, a remote branch 'origin/master' will be checked out to a local branch named 'master', and a remote branch 'origin/develop/new-feature' will be checked out to a local branch named 'develop/new-feature'. If a specific revision and not branch HEAD is checked out, then 'detached' will be used as the local branch name.

Wipe out repository and force clone

The screenshot shows the Jenkins configuration interface for the 'Source Code Management' plugin. The 'Additional Behaviours' section is active, displaying the 'Wipe out repository & force clone' option. The interface includes tabs for 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'. A red 'X' icon and a blue question mark are visible in the top right corner of the configuration area.

Delete the contents of the workspace before build and before checkout. Deletes the git repository inside the workspace and will force a full clone.

Clean after checkout

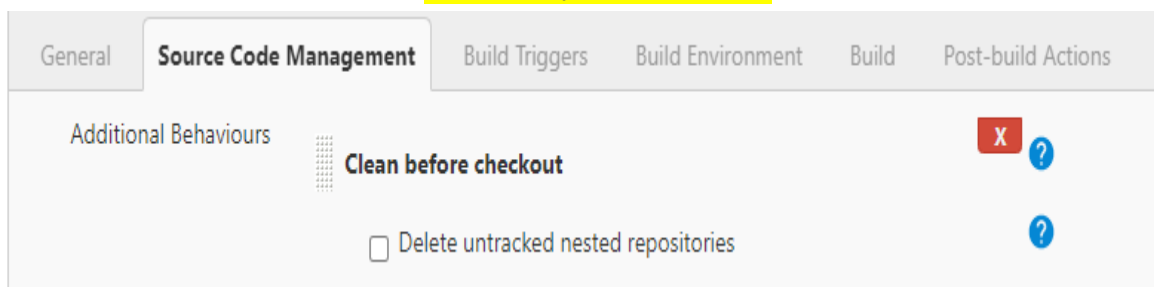
The screenshot shows the Jenkins configuration interface for the 'Source Code Management' plugin. The 'Additional Behaviours' section is active, displaying the 'Clean after checkout' option. Below this, there is a checkbox labeled 'Delete untracked nested repositories'. The interface includes tabs for 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'. A red 'X' icon and a blue question mark are visible in the top right corner of the configuration area.

Clean the workspace after every checkout by deleting all untracked files and directories, including those which are specified in `.gitignore`. Resets all tracked files to their versioned state. Ensures that the workspace is in the same state as if clone and checkout were performed in a new workspace. Reduces the risk that current build will be affected by files generated by prior builds. Does not remove files outside the workspace (like temporary files or cache files). Does not remove files in the `.git` repository of the workspace.

Delete untracked nested repositories

Remove subdirectories which contain `.git` subdirectories if this option is enabled. This is implemented in command line git as `git clean -xffd`. Refer to the [git clean manual page](#) for more information.

Clean before checkout

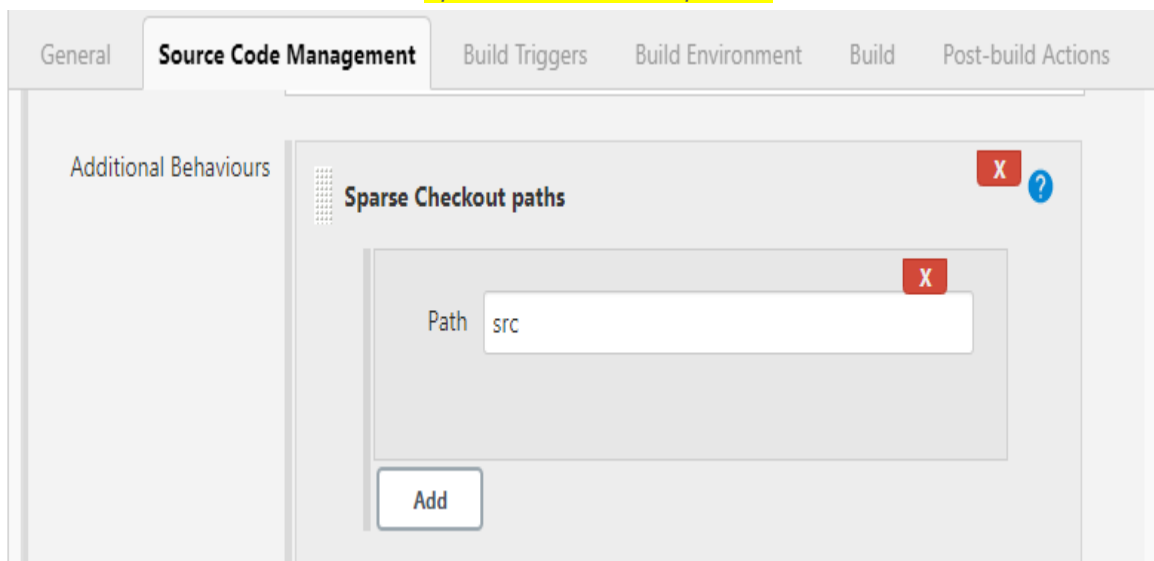


Clean the workspace before every checkout by deleting all untracked files and directories, including those which are specified in `.gitignore`. Resets all tracked files to their versioned state. Ensures that the workspace is in the same state as if cloned and checkout were performed in a new workspace. Reduces the risk that current build will be affected by files generated by prior builds.

Delete untracked nested repositories

Remove subdirectories which contain `.git` subdirectories if this option is enabled. This is implemented in command line git as `git clean -xffd`. Refer to the [git clean manual page](#) for more information.

Sparse checkout paths



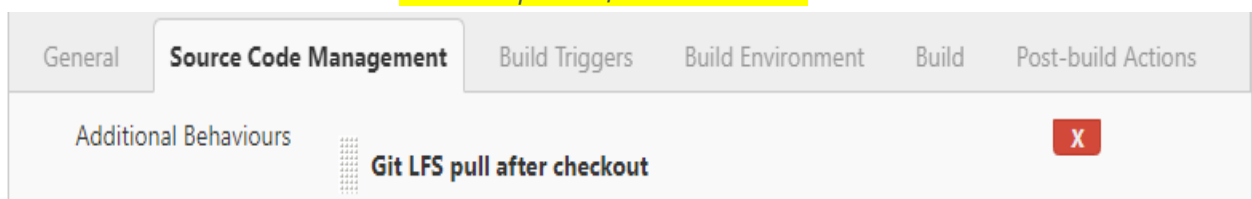
Specify the paths that you'd like to sparse checkout. This may be used for saving space (Think about a reference repository). Be sure to use a recent version of Git, at least above 1.7.10.

Multiple sparse checkout path values can be added to a single job.

Path

File or directory to be included in the checkout

Git LFS pull after checkout



Enable git large file support for the workspace by pulling large files after the checkout completes. Requires that the controller and each agent performing an LFS checkout have installed git lfs.

Changelog Extensions

The plugin can calculate the source code differences between two builds. Changelog extensions adapt the changelog calculations for different cases.

Calculate changelog against a specific branch

The screenshot shows the Jenkins configuration interface for 'Source Code Management'. The 'Additional Behaviours' section is expanded, showing the option 'Calculate changelog against a specific branch'. Below this, there are two input fields: 'Name of repository' with the value 'origin' and 'Name of branch' with the value 'master'. Each input field has a help icon (question mark) to its right. The top navigation bar includes tabs for 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'.

'Calculate changelog against a specific branch' uses the specified branch to compute the changelog instead of computing it based on the previous build. This extension can be useful for computing changes related to a known base branch, especially in environments which do not have the concept of a "pull request".

Name of repository

Name of the repository, such as 'origin', that contains the branch.

Name of branch

Name of the branch used for the changelog calculation within the named repository.

Use commit author in changelog

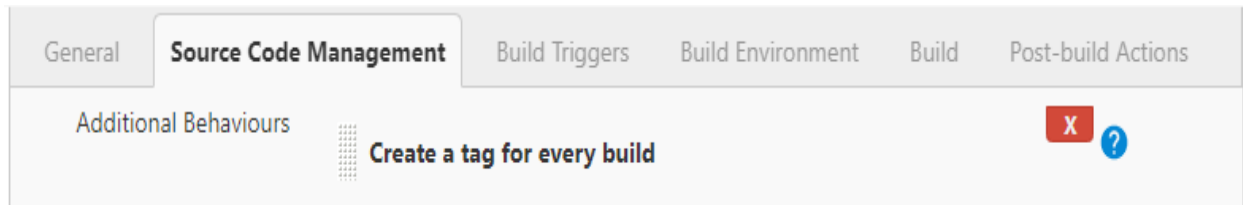
The screenshot shows the Jenkins configuration interface for 'Source Code Management'. The 'Additional Behaviours' section is expanded, showing the option 'Use commit author in changelog'. The top navigation bar includes tabs for 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'.

The default behavior is to use the Git commit's "Committer" value in build changesets. If this option is selected, the git commit's "Author" value is used instead.

Tagging Extensions

Tagging extensions allow the plugin to apply tags in the current workspace

Create a tag for every build

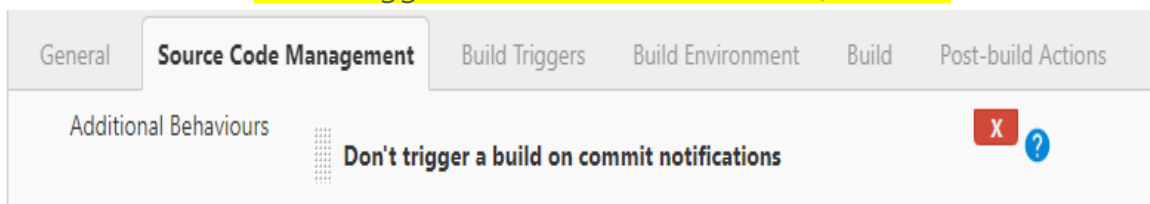


Create a tag in the workspace for every build to unambiguously mark the commit that was built. You can combine this with Git publisher to push the tags to the remote repository.

Build Initiation Extensions

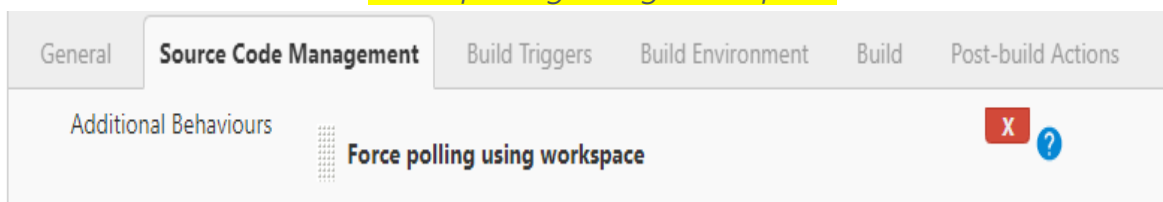
The git plugin can start builds based on many different conditions. The build initiation extensions control the conditions that start a build. They can ignore notifications of a change or force a deeper evaluation of the commits when polling

Don't trigger a build on commit notifications



If checked, this repository will be ignored when the notifyCommit URL is accessed whether the repository matches or not.

Force polling using workspace

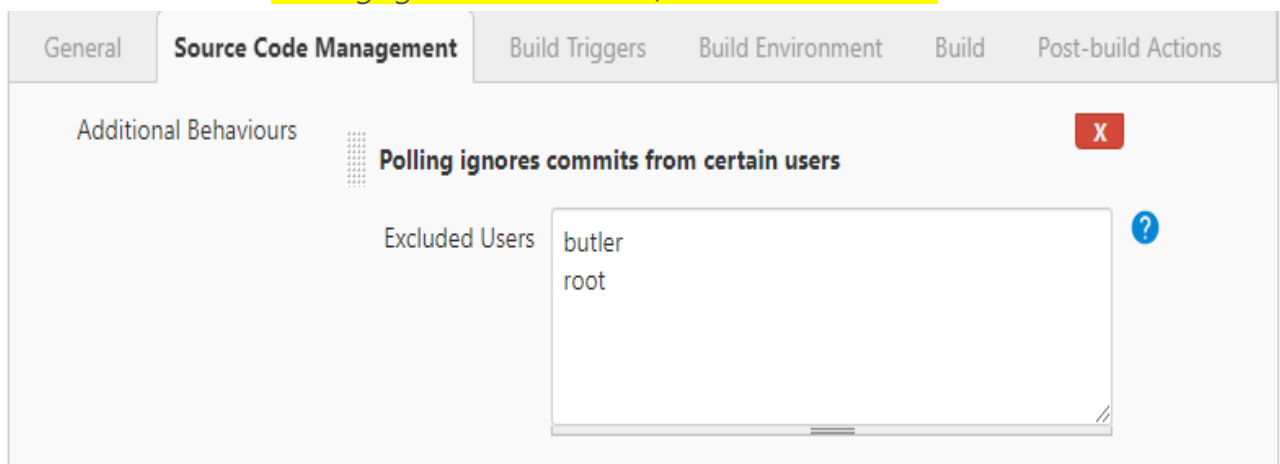


The git plugin polls remotely using `ls-remote` when configured with a single branch (no wildcards!). When this extension is enabled, the polling is performed from a cloned copy of the workspace instead of using `ls-remote`.

If this option is selected, polling will use a workspace instead of using `ls-remote`.

By default, the plugin polls by executing a polling process or thread on the Jenkins controller. If the Jenkins controller does not have a git installation, the administrator may [enable JGit](#) to use a pure Java git implementation for polling. In addition, the administrator may need to [disable command line git](#) to prevent use of command line git on the Jenkins controller.

Polling ignores commits from certain users



The screenshot shows the Jenkins configuration interface for Source Code Management. The 'Additional Behaviours' section is expanded, showing the option 'Polling ignores commits from certain users' which is currently checked. Below this, there is a text area labeled 'Excluded Users' containing the text 'butler' and 'root'. A red 'X' icon is visible in the top right corner of the configuration area, and a blue question mark icon is next to the text area.

These options allow you to perform a merge to a particular branch before building. For example, you could specify an integration branch to be built, and to merge to master. In this scenario, on every change of integration, Jenkins will perform a merge with the master branch, and try to perform a build if the merge is successful. It then may push the merge back to the remote repository if the Git Push post-build action is selected.

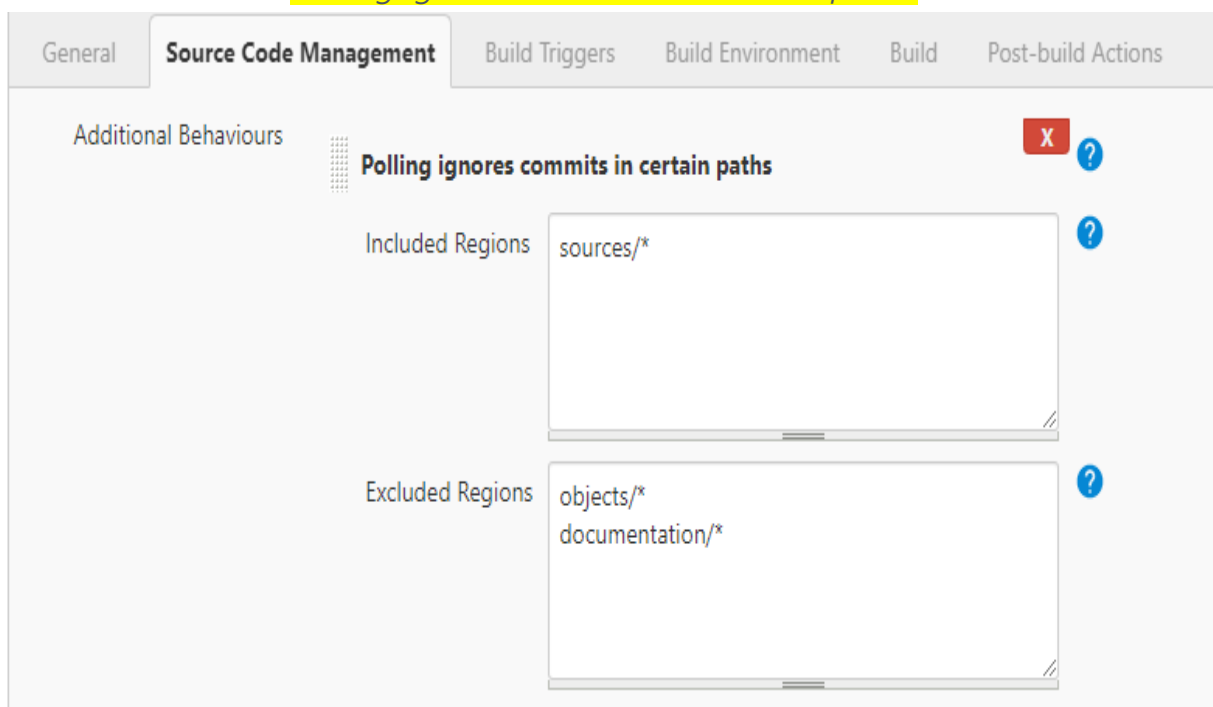
Excluded Users

If set and Jenkins is configured to poll for changes, Jenkins will ignore any revisions committed by users in this list when determining if a build should be triggered. This can be used to exclude commits done by the build itself from triggering another build, assuming the build server commits the change with a distinct SCM user. Using this behavior prevents the faster git ls-remote polling mechanism. It forces polling to require a workspace, as if you had selected the [Force polling using workspace](#) extension.

Each exclusion uses exact string comparison and must be separated by a new line.

User names are only excluded if they exactly match one of the names in this list.

Polling ignores commits in certain paths



The screenshot shows the Jenkins configuration interface for 'Source Code Management'. The 'Additional Behaviours' section is expanded, showing the 'Polling ignores commits in certain paths' option, which is currently disabled (indicated by a red 'X' icon). Below this, there are two text input fields: 'Included Regions' containing 'sources/*' and 'Excluded Regions' containing 'objects/*' and 'documentation/*'. Each input field has a blue question mark icon to its right.

If set and Jenkins is configured to poll for changes, Jenkins will pay attention to included and/or excluded files and/or folders when determining if a build needs to be triggered.

Using this behavior will preclude the faster remote polling mechanism, forcing polling to require a workspace thus sometimes triggering

unwanted builds, as if you had selected the [Force polling using workspace](#) extension as well. This can be used to exclude commits done by the build itself from triggering another build, assuming the build server commits the change with a distinct SCM user

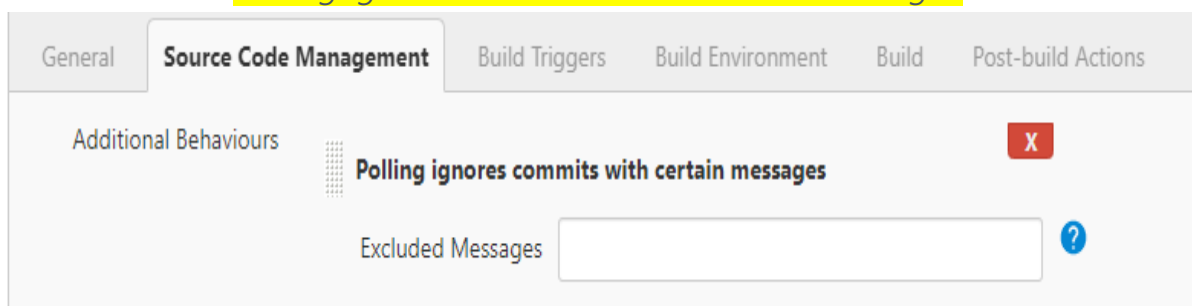
Included Regions

Each inclusion uses java regular expression pattern matching, and must be separated by a new line. An empty list implies that everything is included.

Excluded Regions

Each exclusion uses java regular expression pattern matching, and must be separated by a new line. An empty list excludes nothing.

Polling ignores commits with certain messages



The screenshot shows the Jenkins configuration interface for 'Source Code Management'. The 'General' tab is selected. Under 'Additional Behaviours', the option 'Polling ignores commits with certain messages' is checked, indicated by a red 'X' icon. Below this option is a text input field labeled 'Excluded Messages' with a blue question mark icon to its right.

Excluded Messages

If set and Jenkins is set to poll for changes, Jenkins will ignore any revisions committed with message matched to the regular expression pattern when determining if a build needs to be triggered. This can be used to exclude commits done by the build itself from triggering another build, assuming the build server commits the change with a distinct message. You can create more complex patterns using embedded flag expressions.

Strategy for choosing what to build

The screenshot shows the Jenkins configuration interface for the 'Strategy for choosing what to build' extension. The top navigation bar includes tabs for 'General', 'Source Code Management' (which is active), 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'. Below the tabs, the page title is 'Strategy for choosing what to build'. The main configuration area contains three settings: 1. 'Choosing strategy' is a dropdown menu currently set to 'Ancestry'. 2. 'Maximum Age of Commit' is a text input field containing the value '7'. 3. 'Commit in Ancestry' is a text input field containing the commit hash '580f57853880192353c9e33adc01ee2f7224bac3'. To the right of the 'Maximum Age of Commit' field, there is explanatory text: 'The maximum age of a commit (in days) for it to be built. This uses the GIT_COMMITTER_DATE, not GIT_AUTHOR_DATE.' To the right of the 'Commit in Ancestry' field, there is explanatory text: 'If an ancestor commit (sha1) is provided, only branches with this commit in their history will be built.'

When you are interested in using a job to build multiple branches, you can choose how Jenkins chooses the branches to build and the order they should be built.

This extension point in Jenkins is used by many other plugins to control the job as it builds specific commits. When you activate those plugins, you may see them installing a custom build strategy.

Ancestry

Maximum Age of Commit

The maximum age of a commit (in days) for it to be built. This uses the GIT_COMMITTER_DATE, not GIT_AUTHOR_DATE

Commit in Ancestry

If an ancestor commit (SHA-1) is provided, only branches with this commit in their history will be built.

Default

Build all the branches that match the branch name pattern.

Inverse

Build all branches except for those which match the branch specifiers configure above. This is useful, for example, when you have jobs building your master and various release branches and you want a second job which builds all new feature branches. For example, branches which do not match these patterns without redundantly building master and the release branches again each time they change.

Merge Extensions

The git plugin can optionally merge changes from other branches into the current branch of the agent workspace. Merge extensions control the source branch for the merge and the options applied to the merge.

Merge before build

General	Source Code Management	Build Triggers	Build Environment	Build	Post-build Actions
Additional Behaviours					
<div><div></div><div>Merge before build</div><div><div>X</div><div>?</div></div></div>					
Name of repository		origin		?	
Branch to merge to		master		?	
Merge strategy		default		?	
Fast-forward mode		--ff		?	

These options allow you to perform a merge to a particular branch before building. For example, you could specify an integration branch to be built, and to merge to master. It then may push the merge back to the remote repository if the [Git Publisher post-build action](#) is selected.

Name of repository

Name of the repository, such as origin, that contains the branch. If left blank, it'll default to the name of the first repository configured.

Branch to merge to

The name of the branch within the named repository to merge to, such as master.

Merge strategy

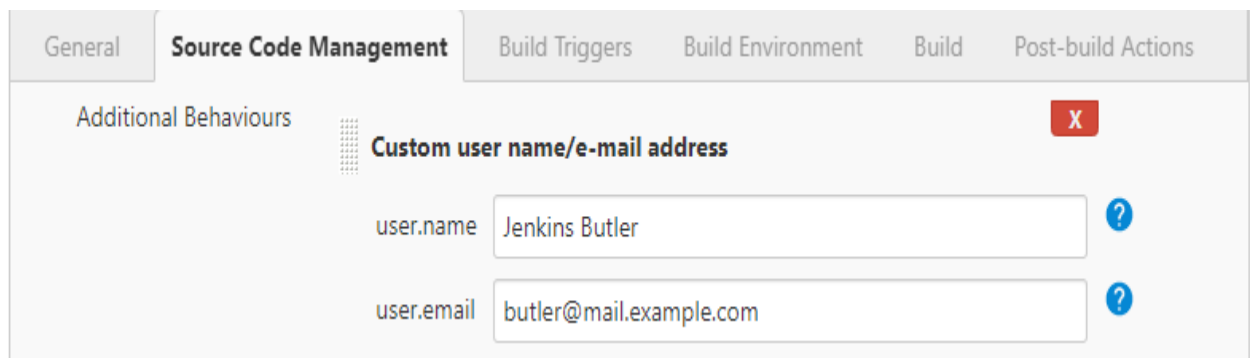
Merge strategy selection. Choices include:

- default
- resolve
- recursive
- octopus
- ours
- subtree
- recursive_theirs

Fast-forward mode

- `--ff`: fast-forward which gracefully falls back to a merge commit when required
- `-ff-only`: fast-forward without any fallback
- `--no-ff`: merge commit always, even if a fast-forward would have been allowed

Custom user name/e-mail address



The screenshot shows the Jenkins configuration interface for a job. The 'Source Code Management' tab is selected. Under 'Additional Behaviours', the 'Custom user name/e-mail address' option is expanded. It contains two input fields: 'user.name' with the value 'Jenkins Butler' and 'user.email' with the value 'butler@mail.example.com'. Both fields have a blue question mark icon to their right. A red 'X' icon is visible in the top right corner of the 'Additional Behaviours' section.

user.name

Defines the user name value which git will assign to new commits made in the workspace. If given, the environment variables `GIT_COMMITTER_NAME` and `GIT_AUTHOR_NAME` are set for builds and override values from the global settings.

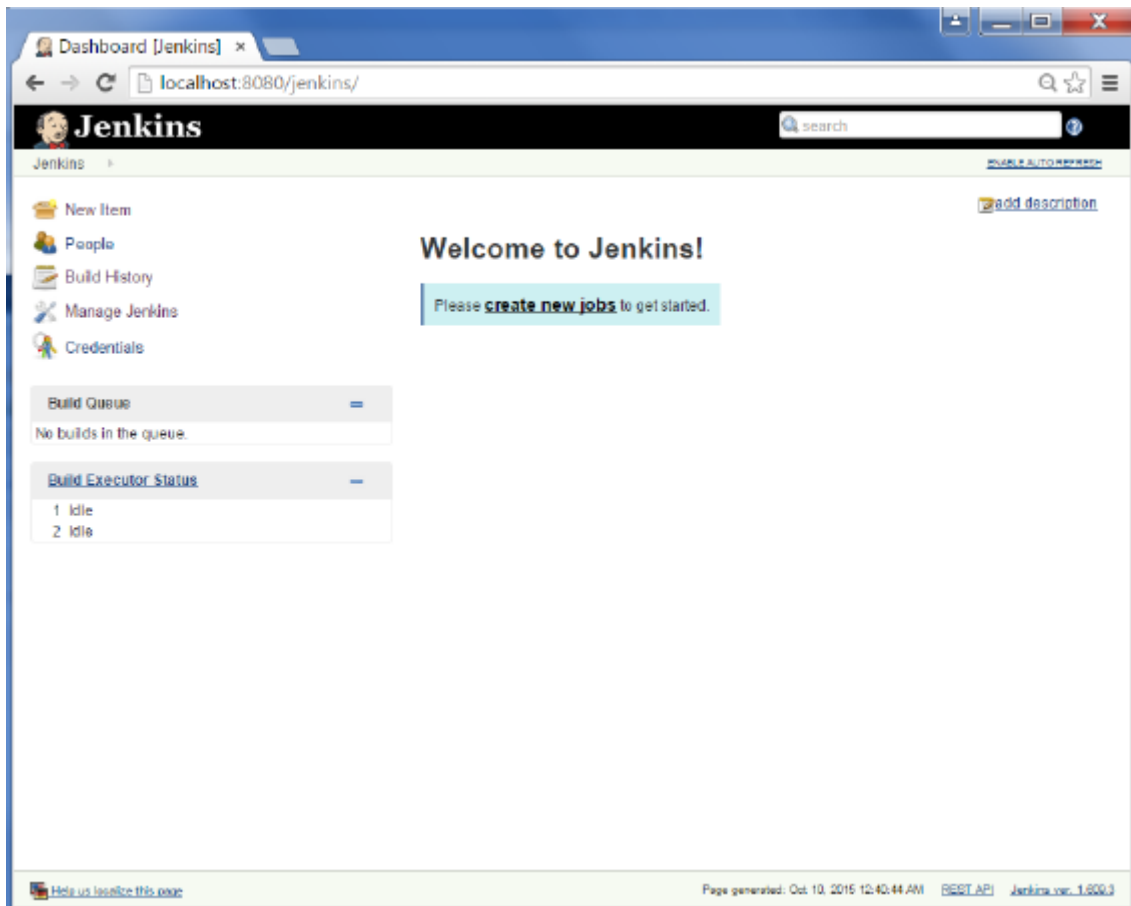
user.email

Defines the user email value which git will assign to new commits made in the workspace. If given, the environment variables `GIT_COMMITTER_EMAIL` and `GIT_AUTHOR_EMAIL` are set for builds and override values from the global setting

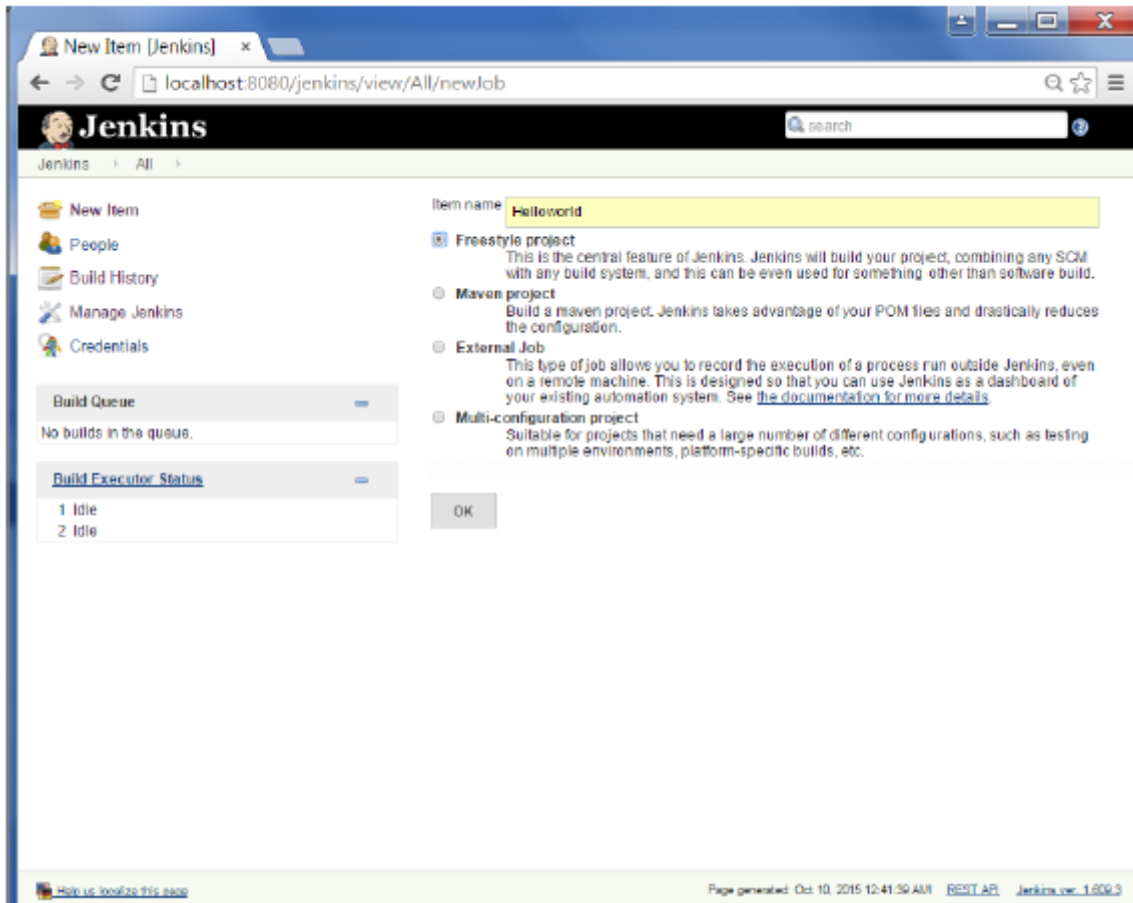
ADD BUILD STEP

For this exercise, we will create a job in Jenkins which picks up a simple HelloWorld application, builds and runs the java program.

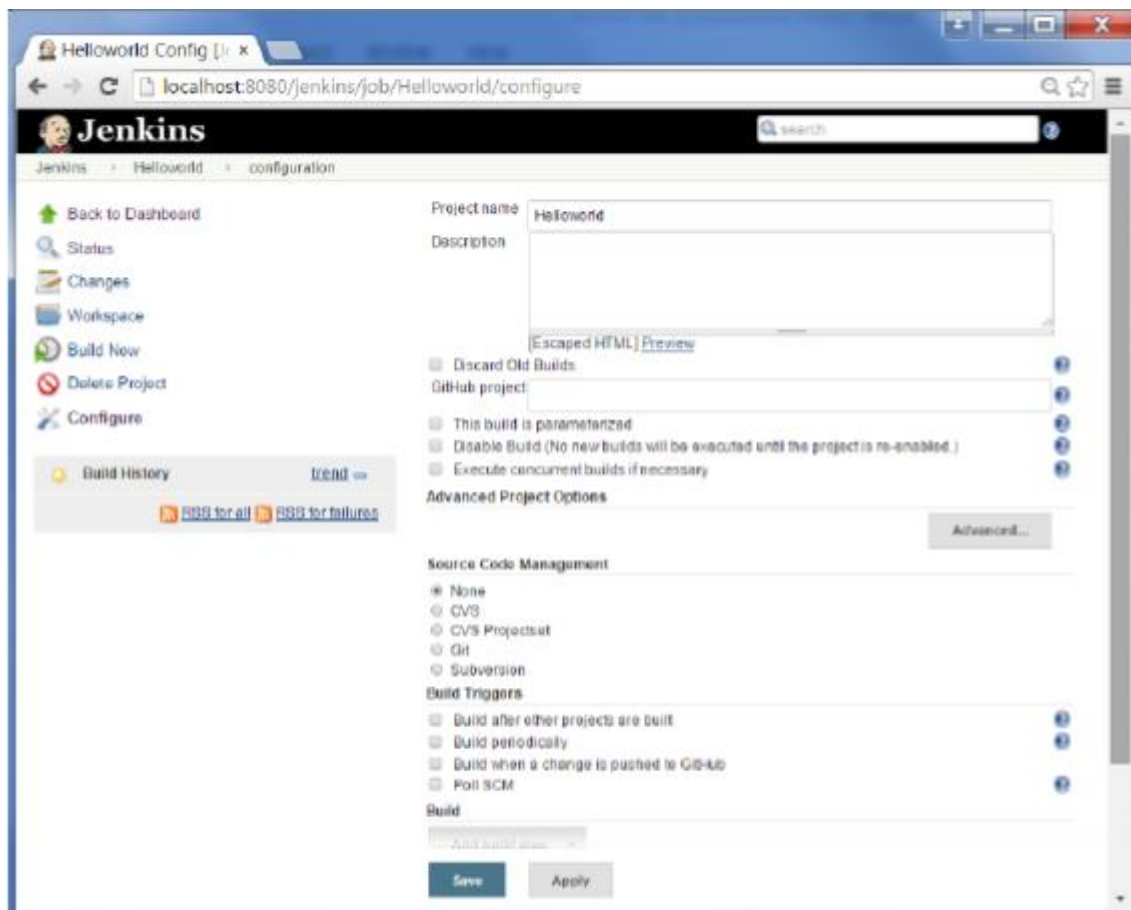
Step 1 – Go to the Jenkins dashboard and Click on New Item



Step 2 – In the next screen, enter the Item name, in this case we have named it Helloworld. Choose the 'Freestyle project option'

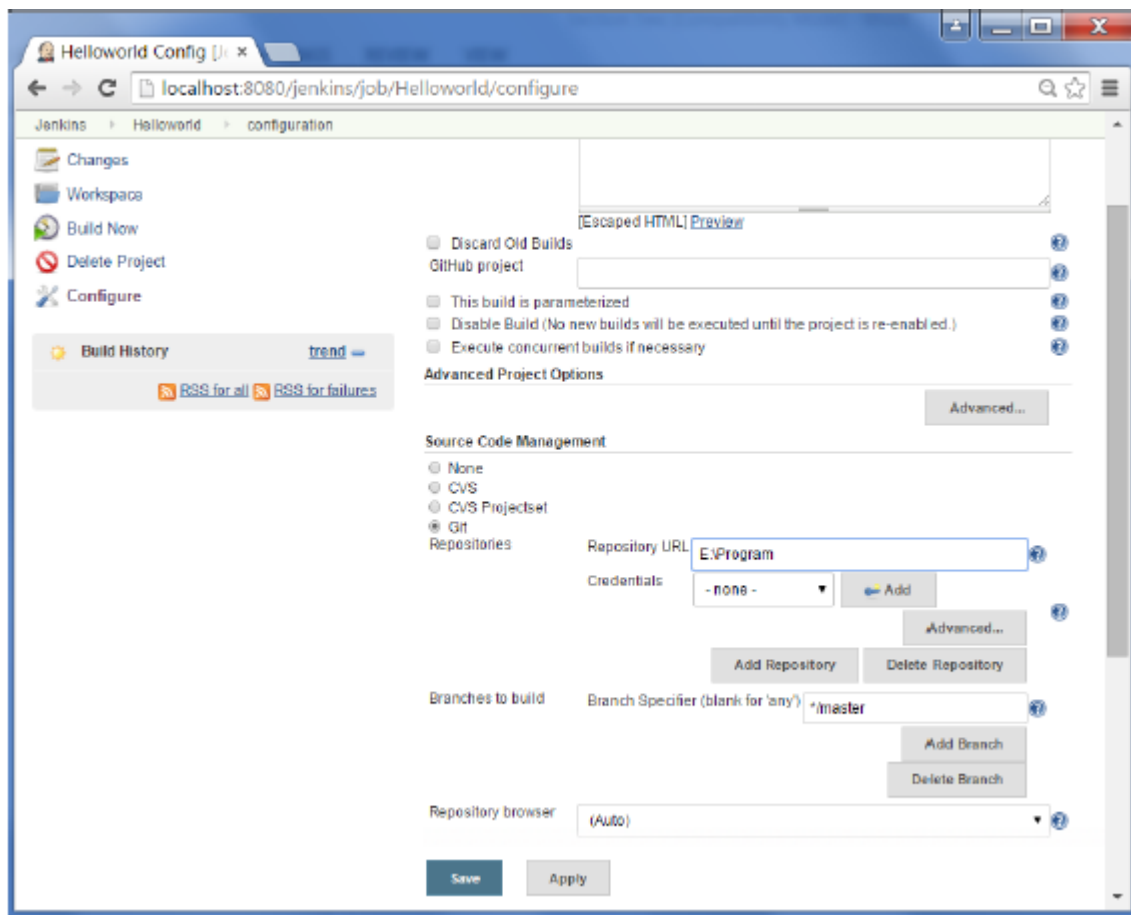


Step 3 – The following screen will come up in which you can specify the details of the job.

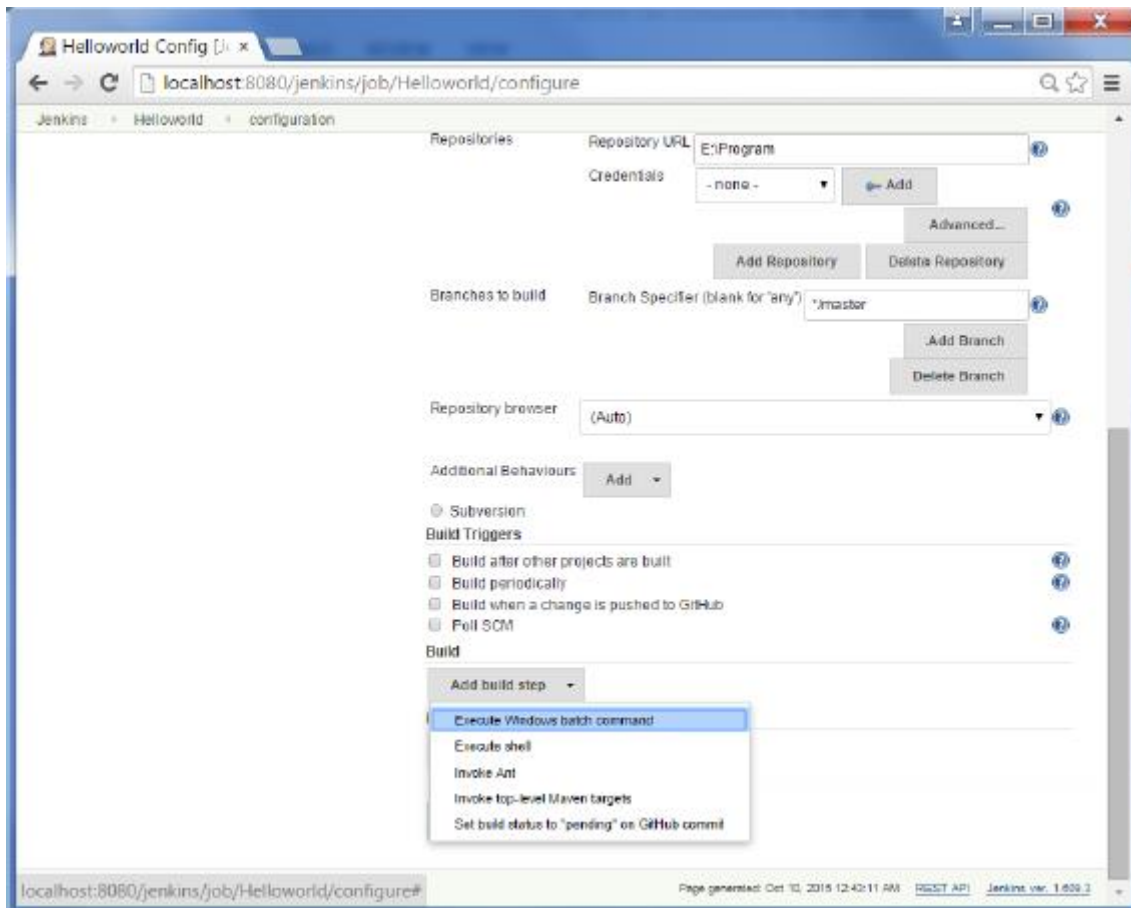


Step 4 – We need to specify the location of files which need to be built. In this example, we will assume that a local git repository(E:\Program) has been setup which contains a 'HelloWorld.java' file. Hence scroll down and click on the Git option and enter the URL of the local git repository.

Note – If you repository is hosted on Github, you can also enter the url of that repository here. In addition to this, you would need to click on the Add button for the credentials to add a user name and password to the github repository so that the code can be picked up from the remote repository.



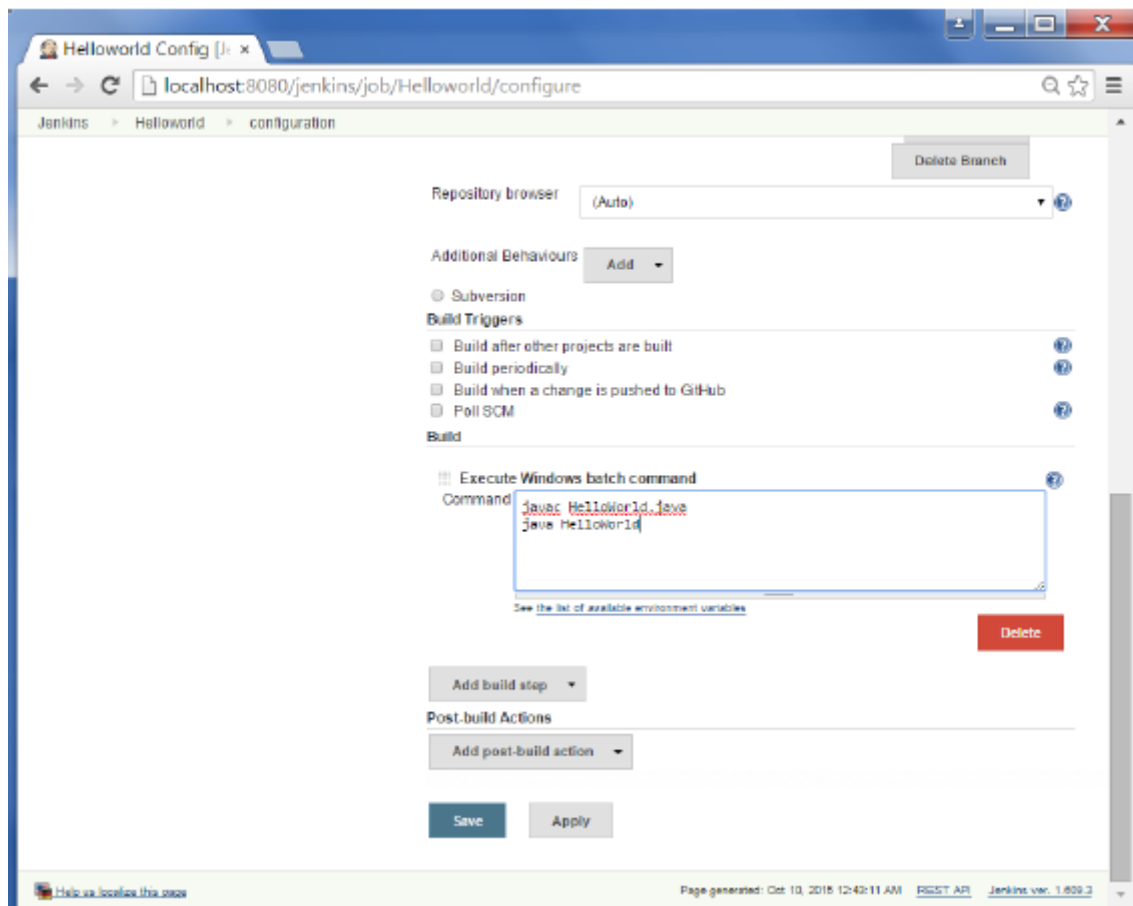
Step 5 – Now go to the Build section and click on Add build step → Execute Windows batch command



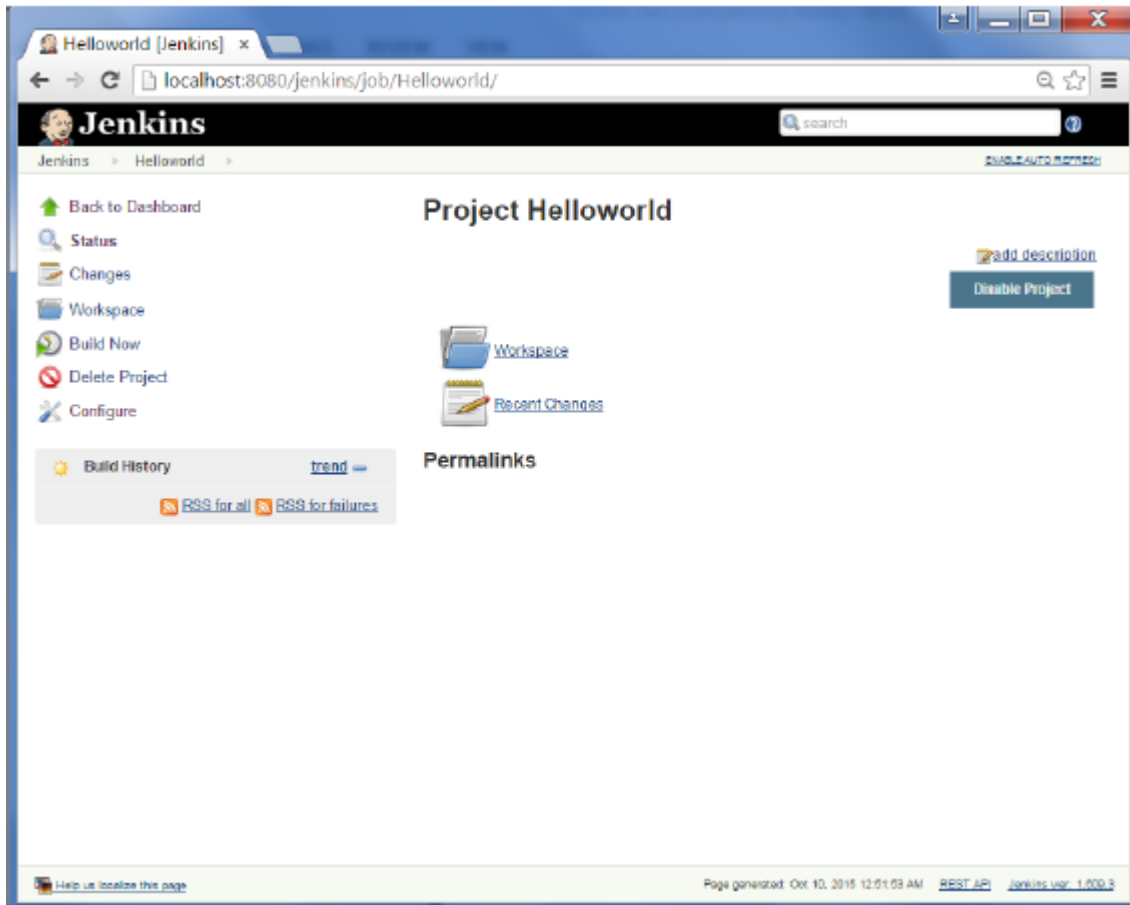
Step 6 – In the command window, enter the following commands and then click on the Save button.

```
Javac HelloWorld.java
```

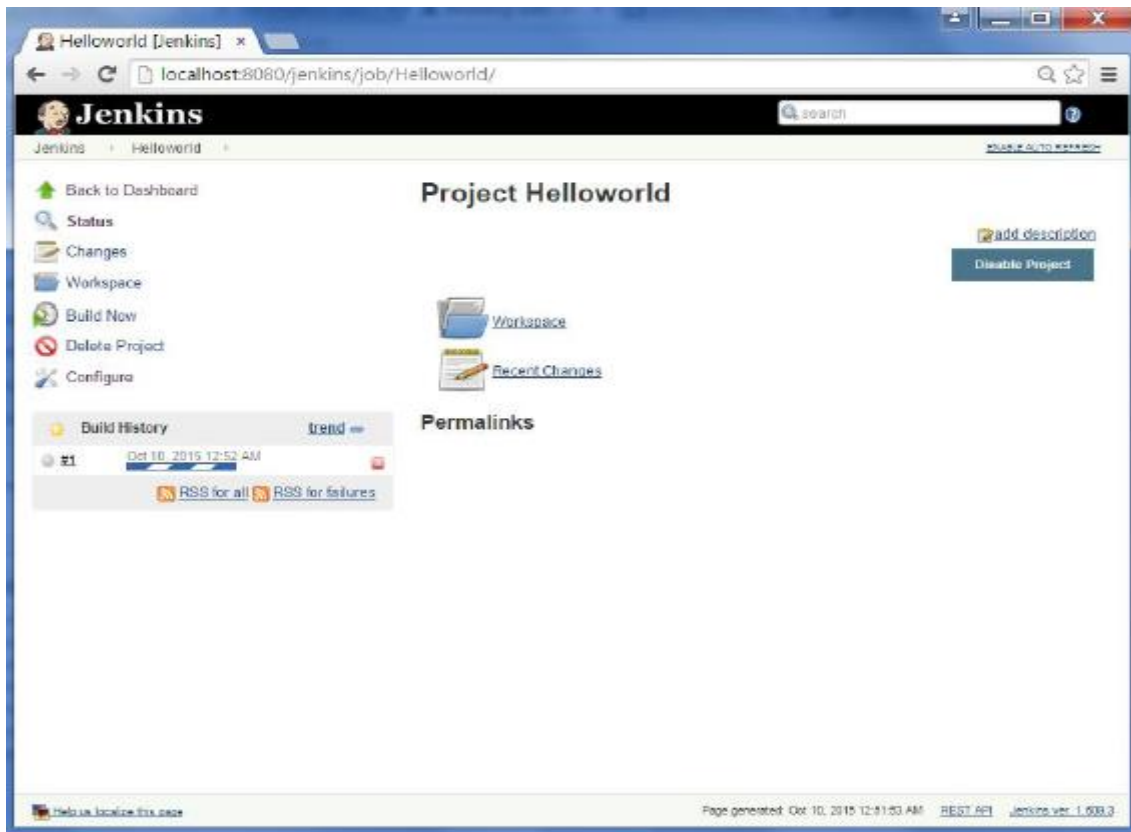
```
Java HelloWorld
```



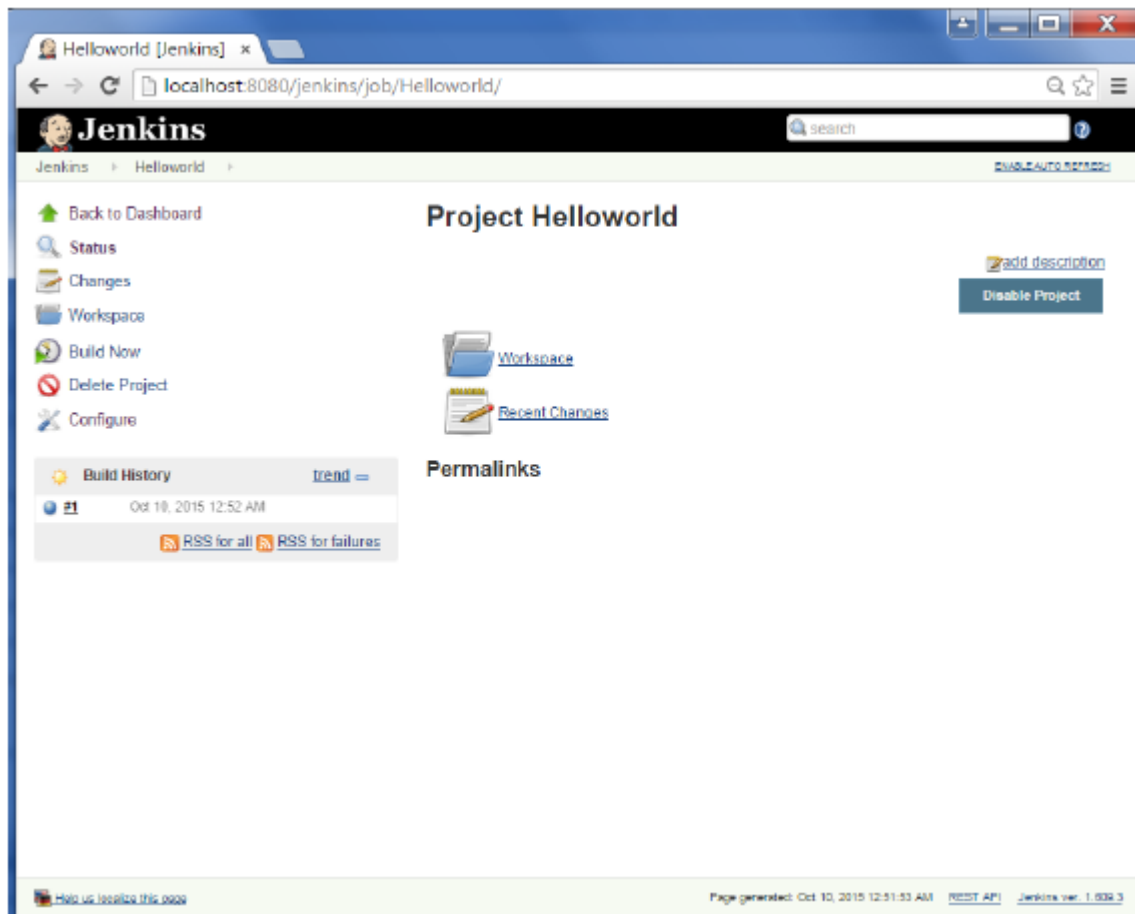
Step 7 – Once saved, you can click on the Build Now option to see if you have successfully defined the job.



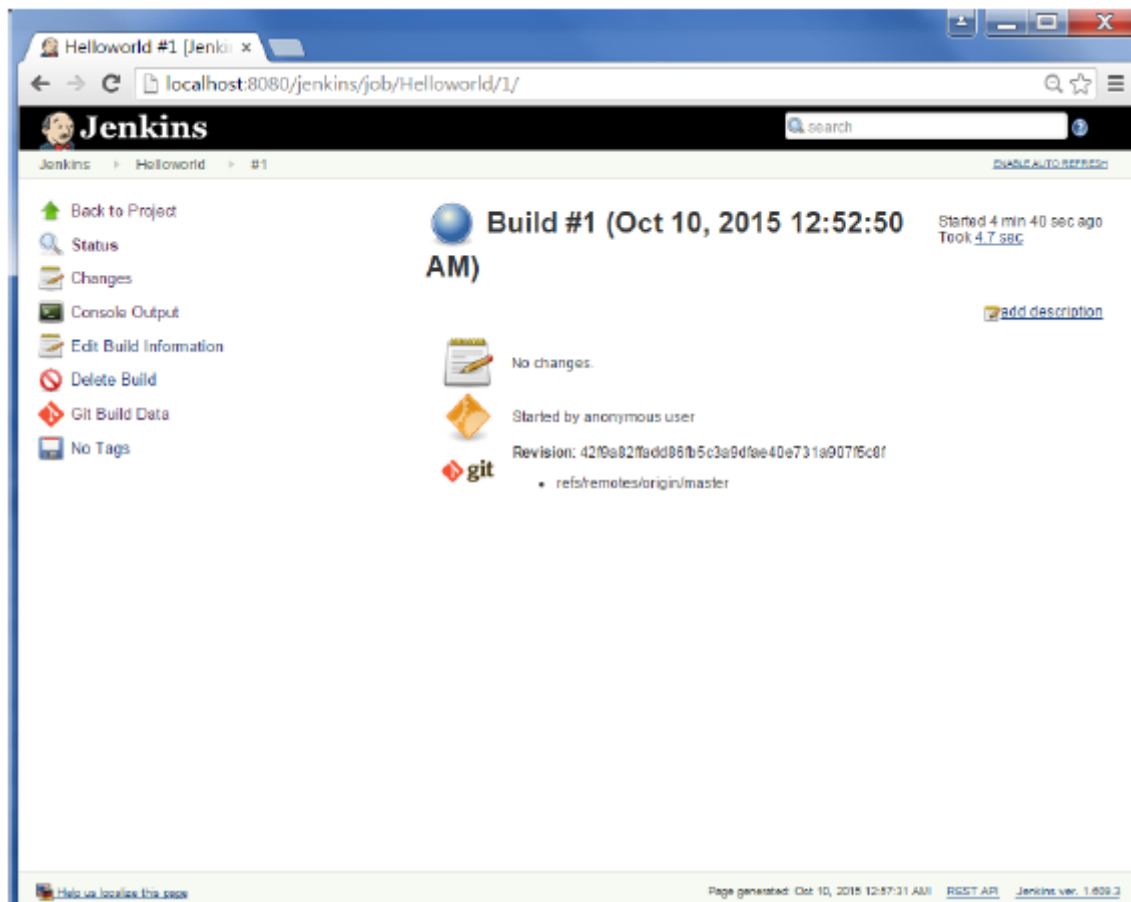
Step 8 – Once the build is scheduled, it will run. The following Build history section shows that a build is in progress.



Step 9 – Once the build is completed, a status of the build will show if the build was successful or not. In our case, the following build has been executed successfully. Click on the #1 in the Build history to bring up the details of the build.



Step 10 – Click on the Console Output link to see the details of the build



Apart from the steps shown above there are just so many ways to create a build job, the options available are many, which makes Jenkins such a fantastic continuous deployment tool.

Bindings :

Secret file:


you can see, the properties of the secret file are not exposed anywhere during the build process and also in the application. The contents of the secret file can also only be accessed only during the first upload of the file.

This plugin gives you an easy way to package up all a job's secret files and passwords and access them using a single environment variable during the build. To use, first go to the Credentials link and add items of type Secret file and/or Secret text.

Steps:


1. After installing the plugin, you will now have a new option under "Build Environment" which is called "Use secret text(s) or file(s)", check it and a new box will appear named "Bindings". Under it, the drop down called "Add " will be visible.

Build Environment

- ☐ Start Xvfb before the build, and shut it down after.
- ☐ Delete workspace before build starts
- ☒ Use secret text(s) or file(s) 

Bindings

Add ▾

- ☐ Provide Configuration files 
- ☐ Abort the build if it's stuck
- ☐ Add timestamps to the Console Output

2. Click on Add and choose "Username and password (separated)"

Bindings

Add ▾

- ☐ Certificate
- ☐ Docker client certificate
- ☐ SSH User Private Key
- ☐ Secret ZIP file
- ☐ Secret file
- ☐ Secret text
- ☐ Username and password (conjoined)
- ☒ Username and password (separated)

- ☐ With Ant

3. We have to add the credentials needed for the project. Select 'Add' → 'Jenkins' in order to add global credentials.

To maximize security, credentials configured in Jenkins are stored in an encrypted form.

General Office 365 Connector Source Code Management Build Triggers **Build Environment** Bindings Build

Post-build Actions

☒ Use secret text(s) or file(s) ?

Bindings

Username and password (separated) ?

Username Variable

Password Variable

Credentials

☒ Specific credentials ☐ Parameter expression

/***** v

Add

Jenkins

Add

4. Set 'Global Credentials' under domain and 'Username and Password' under kind. The scope can be set according to how the credentials should be available for usage.


- Global Scope — This is the default scope. Global scope credentials are available to all jobs within Jenkins, so that all Jenkins jobs are allowed to use global-scoped credentials.
- System Scope — These credentials are only exposed to the Jenkins system and background tasks where the Jenkins instance itself is using the credentials, and will not be available to jobs within Jenkins. Unlike the global scope, this significantly restricts where the credential can be used, thereby providing a higher degree of confidentiality to the credential

General Office 365 Connector Source Code Management Build Triggers **Build Environment** Bindings Build


Post-build Actions

☐ Start VMS before the build, and shut it down after.

☐ Delete workspace before build starts


☒ Use secret text(s) or file(s) 

Bindings

Username and password (separated) 


Username Variable

USER



Password Variable


PASS






Credentials

☒ Specific credentials ☐ Parameter expression

tqeuser/*****



 Add 



5. After adding and selecting the global credentials, you can set the variables to separately identify the username and the password of the selected credentials.

6. The variables can now be used for shell/ batch scripts as needed.

Shell example:

```
1 echo $USER
```

```
2 echo $PASS
```


Windows batch example:

```
1 mvn clean test -Pscenario -Dsuite="TestSuite" -Duser="%USER%" -  
Dpassword="%PASS%"
```

The username and password in the console output will be masked and shown as follows:



Using credentials

There are numerous 3rd-party sites and applications that can interact with Jenkins, for example, artifact repositories, cloud-based storage systems and services, and so on

A systems administrator of such an application can configure credentials in the application for dedicated use by Jenkins. This would typically be done to "lock down" areas of the application's functionality available to Jenkins, usually by applying access controls to these credentials. Once a Jenkins manager (i.e. a Jenkins user who administers a Jenkins site) adds/configures these credentials in Jenkins, the credentials can be used by Pipeline projects to interact with these 3rd party applications.

Note: The Jenkins credentials functionality described on this and related pages is provided by the [Credentials Binding plugin](#).

Credentials stored in Jenkins can be used:

- anywhere applicable throughout Jenkins (i.e. global credentials),
- by a specific Pipeline project/item (read more about this in the [Handling credentials](#) section of [Using a Jenkinsfile](#)),
- by a specific Jenkins user (as is the case for [Pipeline projects created in Blue Ocean](#)).

Jenkins can store the following types of credentials:

- Secret text - a token such as an API token (e.g. a GitHub personal access token),
- Username and password - which could be handled as separate components or as a colon separated string in the format `username:password` (read more about this in [Handling credentials](#)),
- Secret file - which is essentially secret content in a file,

- SSH Username with private key - an [SSH public/private key pair](#),
- Certificate - a [PKCS#12 certificate file](#) and optional password, or
- Docker Host Certificate Authentication credentials.

Credential security

To maximize security, credentials configured in Jenkins are stored in an encrypted form on the controller Jenkins instance (encrypted by the Jenkins instance ID) and are only handled in Pipeline projects via their credential IDs.

This minimizes the chances of exposing the actual credentials themselves to Jenkins users and hinders the ability to copy functional credentials from one Jenkins instance to another.

Configuring credentials

This section describes procedures for configuring credentials in Jenkins.

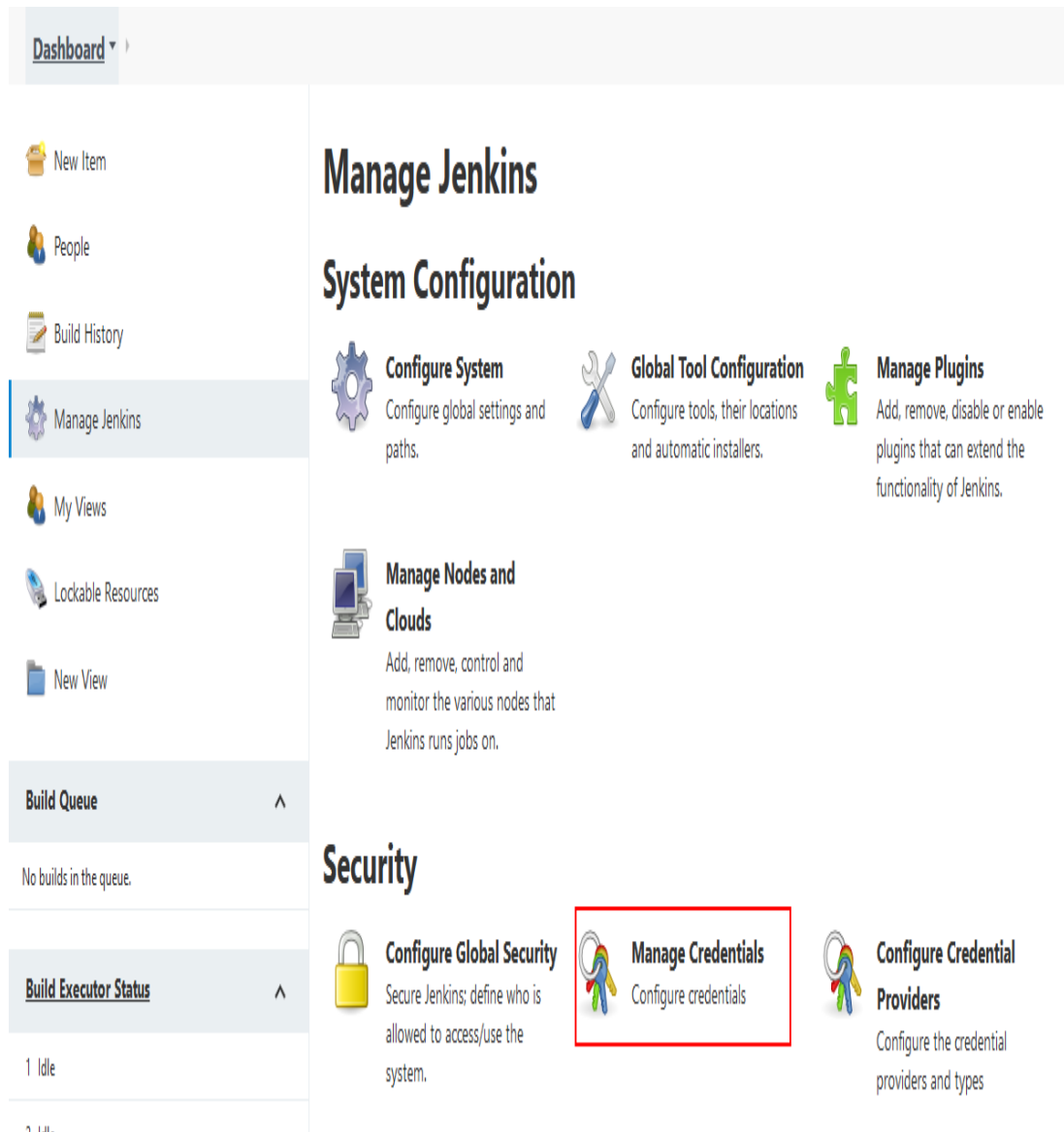
Credentials can be added to Jenkins by any Jenkins user who has the **Credentials > Create** permission (set through Matrix-based security). These permissions can be configured by a Jenkins user with the Administer permission. [Read more about this in the Authorization section of Managing Security.](#)

Otherwise, any Jenkins user can add and configure credentials if the Authorization settings of your Jenkins instance's Configure Global Security settings page is set to the default Logged-in users can do anything setting or Anyone can do anything setting

Adding new global credentials

To add **new global credentials** to your Jenkins instance:

1. If required, **ensure you are logged in to Jenkins** (as a user with the Credentials > Create permission).
2. From the **Jenkins home page (i.e. the Dashboard of the Jenkins classic UI)**, click **Manage Jenkins > Manage Credentials**.




3. Under **Stores scoped to Jenkins** on the right, click on Jenkins.

Stores scoped to Jenkins

P	Store ↓	Domains
	Jenkins	 (global)

4. Under **System**, click the **Global credentials** (unrestricted) link to access this default domain.

System

Domain	Description
 Global credentials (unrestricted)	Credentials that should be available irrespective of domain specification to requirements matching.

Icon: S M L

5. Click **Add Credentials** on the left.

Note: If there are no credentials in this **default domain**, you could also click the **add some credentials** link (which is the same as clicking the **Add Credentials** link).

6. From the Kind field, choose the **type of credentials** to add.

7. From the **Scope** field, choose either:

- Global - if the credential/s to be added is/are for a Pipeline project/item. Choosing this option applies the scope of the credential/s to the Pipeline project/item "object" and all its descendent objects.
- System - if the credential/s to be added is/are for the Jenkins instance itself to interact with system administration functions, such as email authentication, agent connection, etc. Choosing

this option applies the scope of the credential/s to a single object only.

8. Add the credentials themselves into the appropriate fields for your chosen credential type:

- Secret text - copy the secret text and paste it into the Secret field.
- Username and password - specify the credential's Username and Password in their respective fields.
- Secret file - click the Choose file button next to the File field to select the secret file to upload to Jenkins.
- SSH Username with private key - specify the credentials Username, Private Key and optional Passphrase into their respective fields.
Note: Choosing Enter directly allows you to copy the private key's text and paste it into the resulting Key text box.
- Certificate - specify the Certificate and optional Password. Choosing Upload PKCS#12 certificate allows you to upload the certificate as a file via the resulting Upload certificate button.
- Docker Host Certificate Authentication - copy and paste the appropriate details into the Client Key, Client Certificate and Server CA Certificate fields.

9. In the ID field, specify a meaningful credential ID value - for example, `jenkins-user-for-xyz-artifact-repository`. You can use upper- or lower-case letters for the credential ID, as well as any valid separator character. However, for the benefit of all users on your

Jenkins instance, it is best to use a single and consistent convention for specifying credential IDs.

Note: This field is optional. If you do not specify its value, Jenkins assigns a globally unique ID (GUID) value for the credential ID. Bear in mind that once a credential ID is set, it can no longer be changed.

10. Specify an optional Description for the credential/s.

11. Click OK to save the credentials.

Configure a job to Record Fingerprint(s) of a file or set of files

Go to your project, click **Configure** in the left navigation bar, then scroll down to the **Post-build Actions** section of the job

Click on the button to add a **Post-build** action.

Select **Record fingerprints of files to track usage**.

The post-build action configuration fields provide you with a pattern option to match the files you want to fingerprint as well as a couple checkbox selections to do your file fingerprinting.

Maven job type does this automatically for its dependencies and artifacts.

How does it work?

The fingerprint of a file is simply an MD5 checksum. Jenkins maintains a database of MD5 checksums, and for each MD5 checksum, Jenkins records which builds of which projects used it. This database is updated every time a build runs and files are fingerprinted.

To avoid excessive disk usage, Jenkins does not store the actual file. Instead, it just stores MD5 checksums and their usages.

`$JENKINS_HOME/fingerprints`

Plugins can store additional information in these records. For example, [Deployment Notification Plugin](#) tracks files deployed on servers via chef/puppet through fingerprints.

How can I use it?

Here are a few typical scenarios that benefit from this feature:

You develop the BOTTOM project and you want to know who is using BOTTOM #13 in which builds

1. Go to BOTTOM #13 build page.
2. Click the "fingerprint" icon of `bottom.jar` in the build artifacts
3. You'll see all the projects and builds that use it.

You develop the TOP project and you want to know which build of `bottom.jar` and `middle.jar` you are using in TOP #10.

1. Go to TOP #10 build page.
2. Click "see fingerprints"
3. You'll see all the files fingerprinted in TOP #10, along with where they came from.

You have the TOP project that builds a jar. You also have the TOP-TEST project that runs after the TOP project and does extensive integration tests on the latest TOP bits. You want to know the test results of TOP #7.

1. Go to TOP #7 build page.
2. Click the "fingerprint" icon of `top.jar` in the build artifacts
3. You'll see all the TOP-TEST builds that used it.
4. Click it and you'll be taken to the appropriate TOP-TEST build page, which will show you test reports.
5. If there's no TOP-TEST builds displayed, then that means TOP-TEST build didn't run against TOP #7. Maybe it skipped TOP #7 (this can happen if there are a lot of TOP builds in a short period of time), or maybe a new TOP-TEST build is in progress.

Generating an SSH key pair

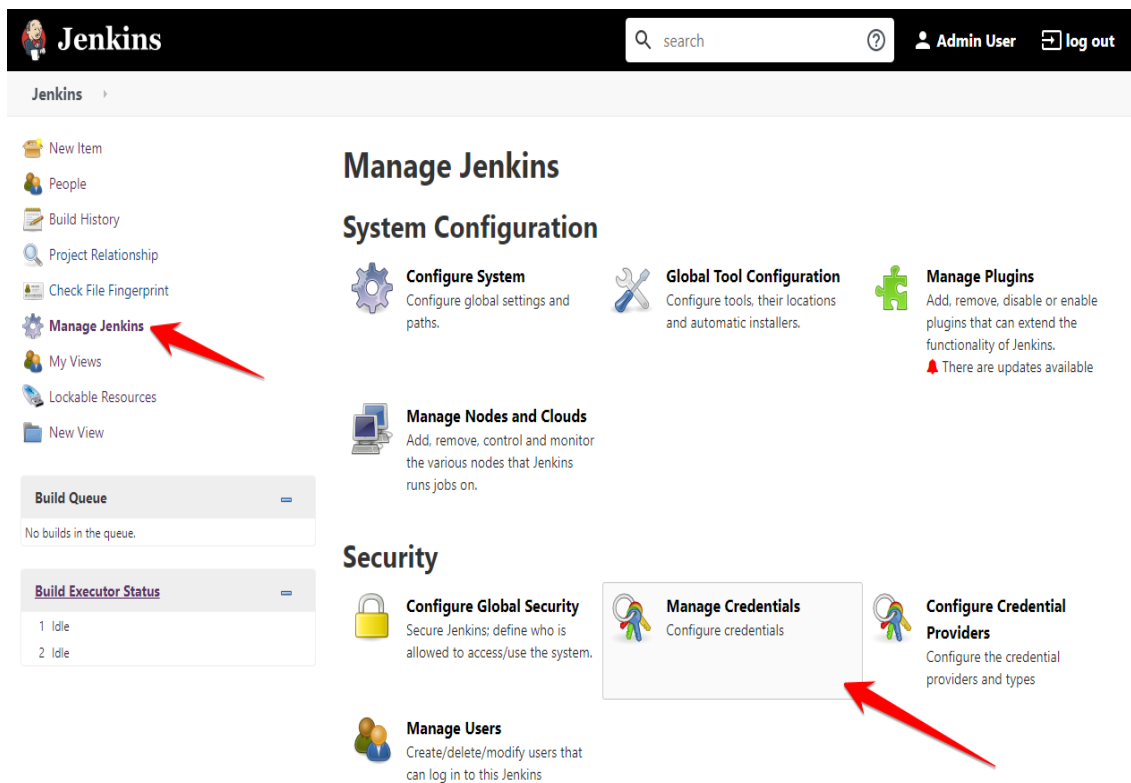
1. In a terminal window run the command: `ssh-keygen -f ~/.ssh/jenkins_agent_key`
2. Provide a passphrase to use with the key
3. Confirm the output looks something like this:
4. `ubuntu@desktop:~$ ssh-keygen -f ~/.ssh/jenkins_agent_key`
5. Generating public/private rsa key pair.
6. Enter passphrase (empty for no passphrase):
7. Enter same passphrase again:
8. Your identification has been saved in
/home/ubuntu/.ssh/jenkins_agent_key
9. Your public key has been saved in
/home/ubuntu/.ssh/jenkins_agent_key.pub
10. The key fingerprint is:
11. SHA256:XqxxjqSLlvDD0ZHm9Y2iR7zC6lbsUIMEHo3ffY8TzGs

12. The key's randomart image is:

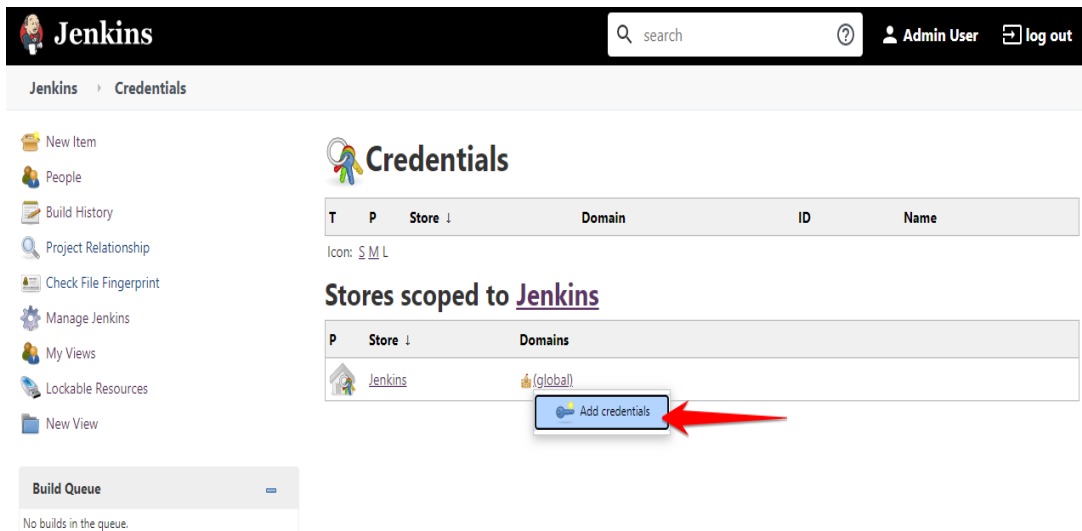
```
13. +---[RSA 3072]-----+
14. |  o+                  |
15. |  ...o   .            |
16. |  .o  .+ .           |
17. |      o+.+ o o       |
18. |  ... o.S o* .        |
19. |  o+ = +.X=          |
20. |  o oO + *.+.        |
21. | . oo.o o .E .        |
22. | o... oo.. o         |
    +---[SHA256]-----+
```

Create a Jenkins SSH credential

1. Go to your Jenkins dashboard;
2. Go to **Manage Jenkins** option in main menu and click on credentials button;



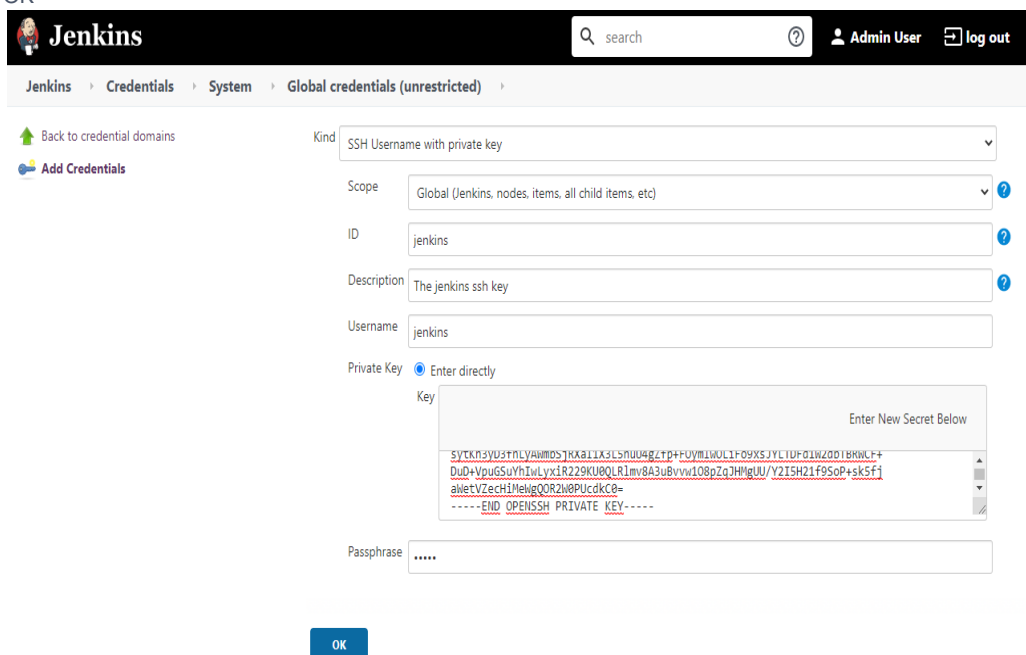
3. select the drop option **Add Credentials** from the global item;



The screenshot shows the Jenkins web interface. At the top, there's a header with the Jenkins logo, a search bar, and user information (Admin User, log out). Below the header, the breadcrumb navigation shows 'Jenkins > Credentials'. On the left sidebar, there are links for 'New Item', 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', 'My Views', 'Lockable Resources', and 'New View'. The main content area is titled 'Credentials' and features a table with columns 'T', 'P', 'Store', 'Domain', 'ID', and 'Name'. Below the table, there's a section 'Stores scoped to Jenkins' with a table showing 'Jenkins' and '(global)'. A red arrow points to the 'Add credentials' button. At the bottom left, there's a 'Build Queue' section indicating 'No builds in the queue.'

4. Fill the form:

- Kind: SSH Username with private key;
- id: jenkins
- description: The jenkins ssh key
- username: jenkins
- Private Key: select **Enter directly** and press the Add button to insert your private key from `~/.ssh/jenkins_agent_key`
- Passphrase: fill your passphrase used to generate the SSH key pair and then press OK



The screenshot shows the 'Add Credentials' form in Jenkins. The breadcrumb navigation is 'Jenkins > Credentials > System > Global credentials (unrestricted)'. On the left, there are links for 'Back to credential domains' and 'Add Credentials'. The form fields are: 'Kind' (SSH Username with private key), 'Scope' (Global (Jenkins, nodes, items, all child items, etc)), 'ID' (jenkins), 'Description' (The jenkins ssh key), 'Username' (jenkins), 'Private Key' (radio button selected for 'Enter directly'), 'Key' (a text area containing a sample SSH private key), and 'Passphrase' (a text field). An 'OK' button is at the bottom.

Job Chaining and Jenkins Continuous Delivery Build Pipelines

Job chaining in Jenkins is the process of automatically starting other job(s) after the execution of a job. This approach lets you build multi-step Jenkins build pipelines or trigger the rebuild of a project if one of its dependencies is updated. These chained jobs form the core of Jenkins continuous delivery pipelines.

- [Out of the Box Solution](#)
- [Jenkins Build Pipeline Plugin](#)
- [Parameterized Trigger Plugin](#)
- [Downstream Buildview Plugin](#)
- [Conclusions](#)

Out of the Box Solution - Using Jenkins Build Other Projects

Jenkins has a built-in feature to build other projects. It is in the Post-build Actions section. You can specify the projects that you want to build after this project is built (you can trigger more than one). So whenever project A is built you will trigger the building of project B. You can also specify the conditions when the other jobs are built. Most often you are interested in continuing with the build pipeline only if the job is successful but your mileage might vary.

☒ Build other projects



Trigger builds of the other projects once a build is successfully completed. Multiple projects can be specified by using comma, like "abc, def".

Other than the obvious use case where you'd like to build other projects that have a dependency on the current project, this can also be useful to split a long build process in to multiple stages (such as the build portion and the test portion).

Projects to build

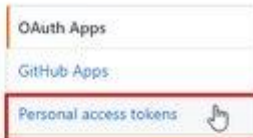
performance-tests

- ☒ Trigger only if build succeeds
- ☐ Trigger even if the build is unstable
- ☐ Trigger even if the build fails

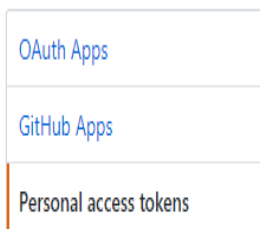
Configure Github



2



3



Personal access tokens

[Generate new token](#) [Revoke all](#)

Tokens you have generated that can be used to access the [GitHub API](#).

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

4 Token description

Jenkins

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

- | | |
|---|--------------------------------------|
| <input type="checkbox"/> repo | Full control of private repositories |
| 5 <input checked="" type="checkbox"/> repo:status | Access commit status |
| <input type="checkbox"/> repo_deployment | Access deployment status |
| <input type="checkbox"/> public_repo | Access public repositories |
| <input type="checkbox"/> repo:invite | Access repository invitations |

6

Generate token

Cancel

Personal access tokens

Generate new token

Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your new personal access token now. You won't be able to see it again!



ghp_1234567890abcdefghijklmnopqrstuvwxyz0123456789



Delete

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Configure Jenkins


 **Jenkins**

Jenkins >

 New Item

 People

 Build History

 Project Relationship

 Check File Fingerprint **8**

 **Manage Jenkins**

 My Views

 Credentials

 New View

Manage Jenkins

 **Configure System**
Configure global settings and paths. **9**


 **Configure Global Security**
Secure Jenkins; define who is allowed to access/use the system.

 **Configure Credentials**
Configure the credential providers and types

GitHub

GitHub Servers

10 **Add GitHub Server** ▼

GitHub Server 

GitHub

GitHub Servers

GitHub Server

Name

Personal_Access_Token_USER

API URL

https://api.github.com

Credentials

- none -

Add

Jenkins

Test connection

Advanced...

Delete

Manage hooks

Jenkins Credentials Provider

Add GitHub Server

Advanced...



Jenkins Credentials Provider: Jenkins

Add Credentials

Domain	Global credentials (unrestricted) ▼
Kind	Secret text ▼
Scope	Global (Jenkins, nodes, items, all child items, etc) ▼ ?
Secret	Personal Access Token (Step 7)
ID	?
Description	?

15

Add

Cancel

GitHub

GitHub Servers

GitHub Server

Name

Personal_Access_Token_USER

API URL

https://api.github.com

Credentials

Secret text



Credentials verified for user rate limit: 4998

Manage hooks

Add GitHub Server

Test connection

Advanced...

Delete

Advanced...

Save

Apply

Configure Jenkins Job

Jenkins

Jenkins > Github_Build_Status_Tutorial >

New Item

People

Build History

Project Relationship

Check File Fingerprint

Manage Jenkins

My Views

Credentials

New View

All +

S W Name ↓

Icon: S M L

Back to Dashboard

Status

Changes

Workspace

Build Now

Delete Project

Configure

GitHub Hook Log

Rename

Post-build Actions

Add post-build action ▼

Save Apply

Aggregate downstream test results

Archive the artifacts

Build other projects

Publish JUnit test result report

Record fingerprints of files to track usage

Git Publisher

E-mail Notification

Editable Email Notification

Set GitHub commit status (universal)

Set build status on GitHub commit [deprecated]

Delete workspace when build is done

Post-build Actions

Set GitHub commit status (universal)

Where:

Commit SHA: Latest build revision

Repositories: Any defined in job repository

What:

Commit context: From GitHub property with fallback to job name

Status result: One of default messages and statuses

Status backref: Backref to the build

Advanced...

Add post-build action ▼

Save

Apply