

DevOps



Caltech

Center for Technology &
Management Education

Post Graduate Program in DevOps



Build Jobs and Configurations

Learning Objectives

By the end of this lesson, you will be able to:

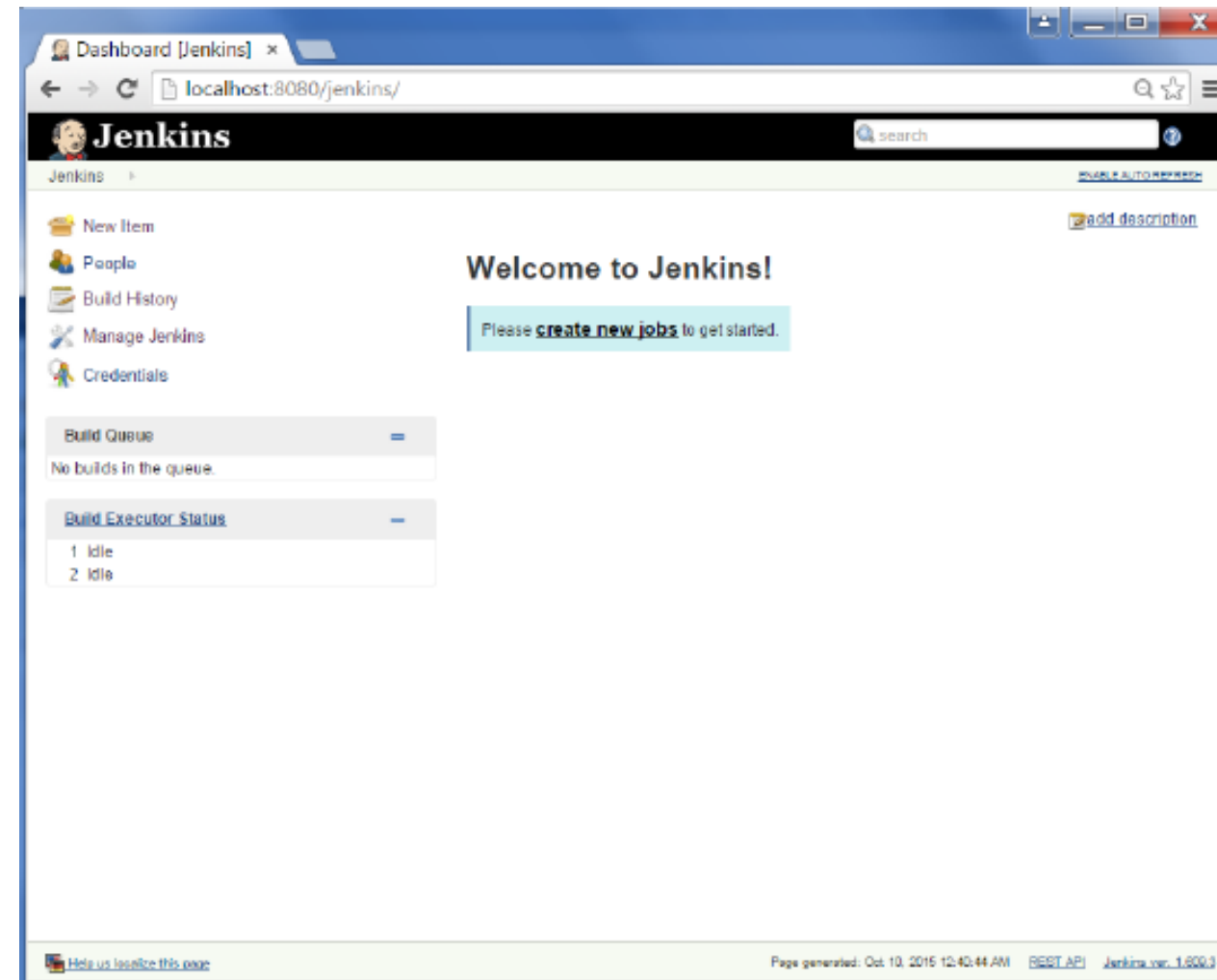
- 🕒 Explain build jobs in Jenkins
- 🕒 Trigger build jobs in Jenkins
- 🕒 Configure build notifications in Jenkins
- 🕒 Build projects in various languages
- 🕒 Create parameterized build jobs



Understanding Jenkins Build Jobs

Creation of Jenkins Build Jobs

You can create new build jobs by clicking on the *New Item* button shown in the UI below. This will give you the list of build jobs available in Jenkins to choose from.



Types of Build Jobs

Jenkins supports four types of build jobs. They are:



Freestyle
software
project

Maven
project

External job

Multi-
configuration
job

Types of Build Jobs

Freestyle software project:

- Freestyle build jobs are general-purpose build jobs.
- They provide maximum flexibility.

Maven project:

- The maven project is specially adapted to Maven projects.
- Jenkins interprets the Maven pom files and project structures to create a basic setup for your project.

Types of Build Jobs

External job:

- The *external job* option helps you monitor non-interactive processes, like cron jobs.

Multi-configuration job:

- The *Multiconfiguration job* is also known as the matrix job.
- It lets you run the same build job with different configurations.
- It is used for testing an application in different environments, with different databases, or on different build machines.

Freestyle Build Jobs

Freestyle Build Jobs

- The freestyle build job can be used for any type of project and is relatively easy to set up.
- The first section in the *create a new freestyle job* tab holds general information about the project, like the name, description, and how and where the build job has to be executed as shown below:

The screenshot shows the Jenkins 'create a new freestyle job' configuration page, specifically the 'General' tab. The tabs at the top are: General, Source Code Management, Build Triggers, Build Environment, Build, and Post-build Actions. The 'General' tab contains a 'Description' text area, a 'JIRA site' dropdown menu, and a list of checkboxes: 'Discard old builds', 'GitHub project', 'This project is parameterized', 'Throttle builds', 'Disable this project', and 'Execute concurrent builds if necessary'. Each checkbox has a help icon to its right. Below these is an 'Advanced...' button. The 'Source Code Management' section is partially visible at the bottom, showing radio buttons for 'None' (selected) and 'Git'. At the very bottom, there are 'Save' and 'Apply' buttons.

General Configurations

Discard Old Builds option:

- The *Discard Old Builds* option gives the provision to limit the number of builds stored in the build history.
- Jenkins can be configured to keep only the recent builds or a specified number of builds.
- Particular builds can be kept forever using the *Keep forever* button on the build details page.

General Configurations

Disable The Build option:

- The *Disable The Build* option allows a build to be disabled so that it will not be executed until it is enabled again.
- This option is usually used to temporarily suspend a build during maintenance or refactoring when notification of the build failures are not useful.

Advanced Configurations

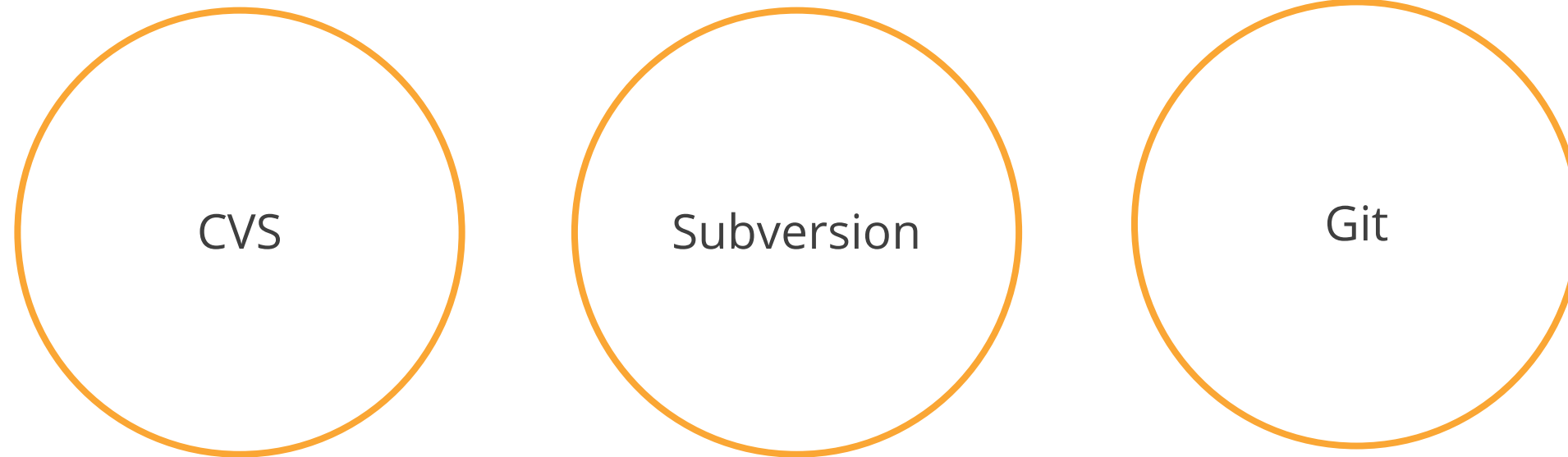
- The Advanced Project Options has less frequently used configuration options.
- These options appear when the Advanced button is clicked on.
- The *Quiet Period* option in the build job configuration overrides the system-wide quiet period defined in the Jenkins System Configuration screen.
- The *Block build when upstream project is building* option is used when several related projects are affected by a single commit, but they must be built in a specific order.

Advanced Configurations

- The *Block build when upstream project is building* option is used when several related projects are affected by a single commit, but they must be built in a specific order.
- If this option is activated, Jenkins will wait until all upstream build jobs have finished before starting the new build.
- The default directory used by Jenkins can be overridden by ticking the *Use custom workspace* option and providing the path yourself.
- The path can be either absolute or relative to Jenkins' home directory.

Configuring Source Code Management

SCM configuration options in Jenkins are identical across all types of build jobs. Jenkins supports:



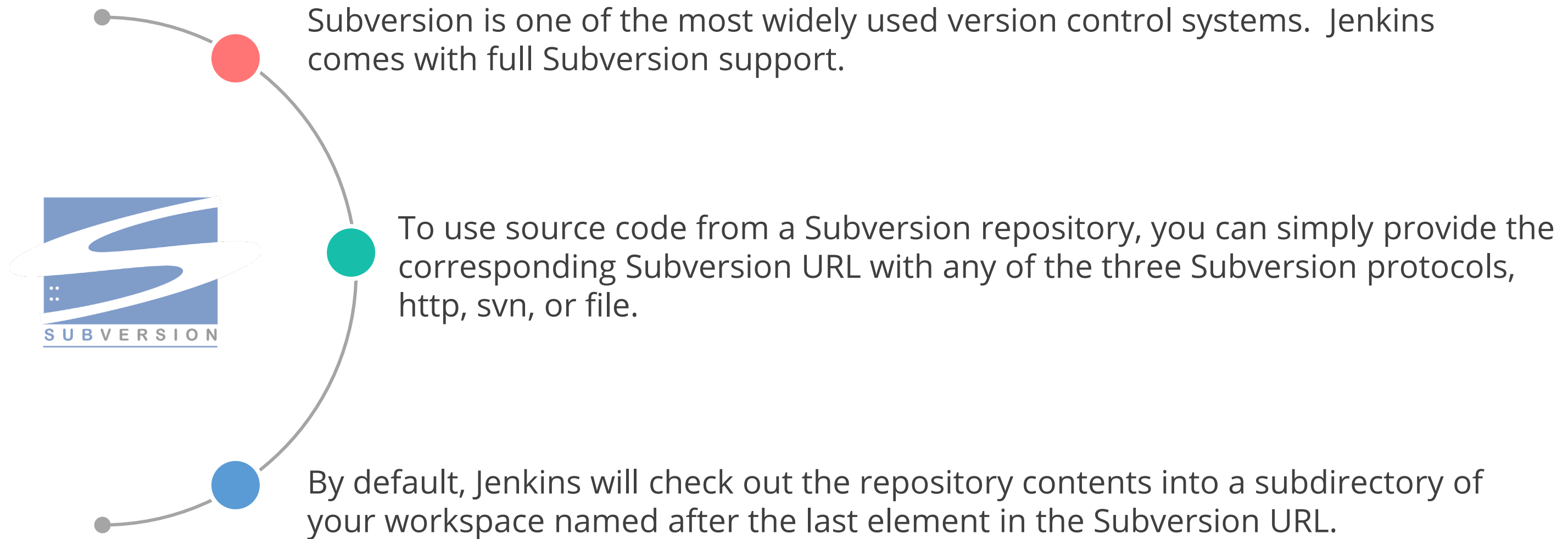
It also integrates with lots of other version control systems via plugins.

Configuring Source Code Management

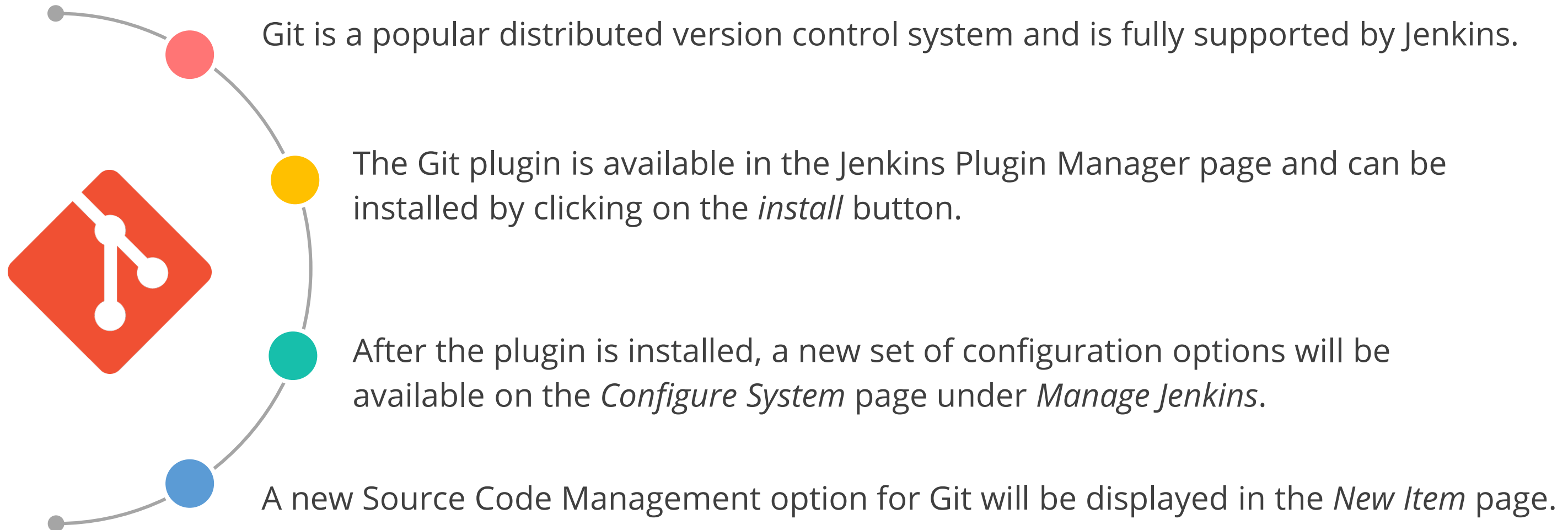
SCM plugin support includes:

- Accurev
- Bazaar
- BitKeeper
- ClearCase
- CMVC
- Dimensions
- CA Harvest
- Mercurial
- Perforce
- Microsoft Team Foundation Server

Working with Subversion



Working with Git



Assisted Practice

Create a Freestyle Build Job

Problem Statement: You have been asked to create a freestyle build job using Jenkins to compile and run a simple Java program.

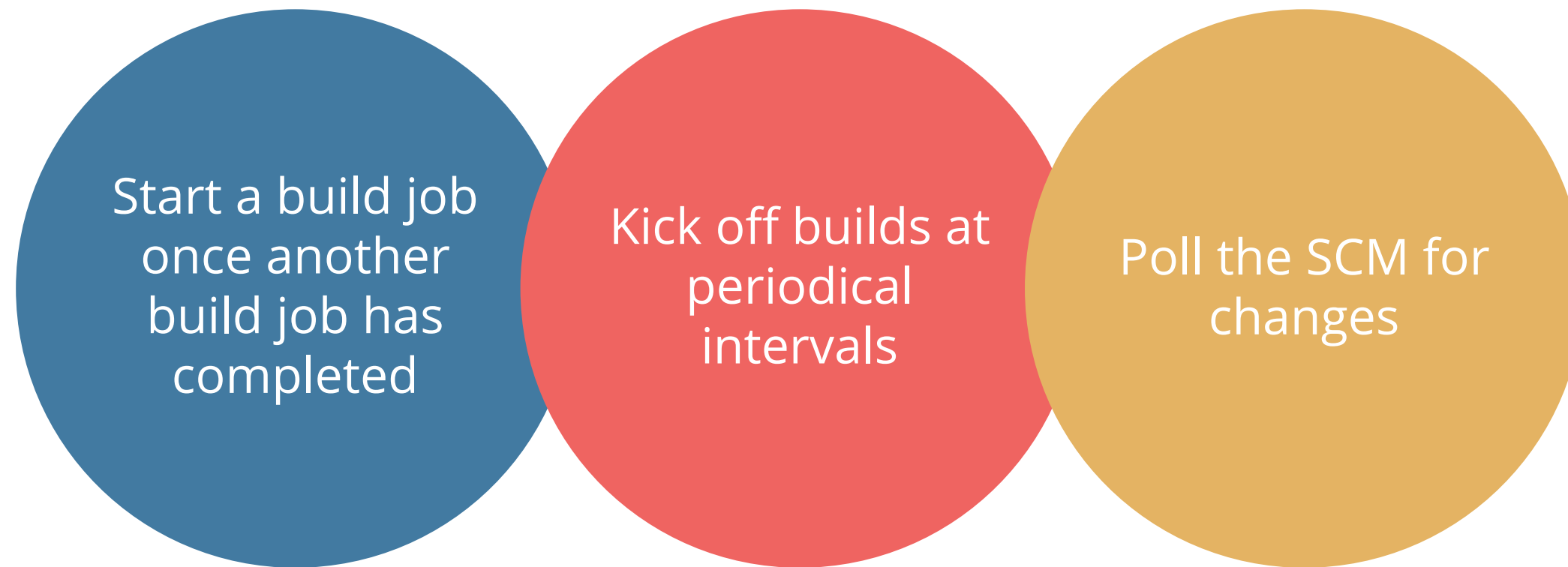
Steps to perform:

1. Create a Git repository
2. Add a Java program to the repository
3. Create a freestyle build job in Jenkins
4. Build the Java program with Jenkins

Build Triggers

Build Triggers

Once the version control system is configured, you need to tell Jenkins when to start a build. This is configured in the *Build Triggers* section. There are three ways to trigger a build job in a freestyle job:



Start a Build Job After Another

Start a build job once another build job has completed option:

- The *Start a build job once another build job has completed* option lets you set up a build that will be run when another build has finished.
- This is configured by entering the name of the preceding build job in this field.
- If the build job can be triggered by several other build jobs, a list of names, separated by commas can be used. In such cases, the build job will be triggered once any of the build jobs in the list finish.

Scheduled Build Jobs

Scheduled builds simply triggers your build job at regular intervals. This strategy can be used for regular nightly builds.

Scheduled builds are not the same as Continuous Integration and could be done with something as simple as a Unix cronjob.

Scheduled Build Jobs

Continuous Integration requires faster feedback than scheduled builds. Scheduled builds can be used for long running build jobs, where quick feedback is less critical.

For example: Intensive load and performance tests.

Scheduled Build Jobs

Jenkins uses a cron-style syntax, consisting of five fields separated by white space in the following format, for all scheduling tasks:

MINUTE HOUR DOM MONTH DOW

The meaning of these fields are as follows:

- 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

Assisted Practice

Scheduled Builds

Problem Statement: You've been asked to schedule a freestyle build job in Jenkins that runs twice a day.

Steps to perform:

1. Set up a Git repository
2. Create a freestyle build in Jenkins
3. Define a regular expression for twice a day
4. Test the scheduled build

Scheduled Build Jobs

Shortcut syntax for scheduling build jobs:

Notation	Meaning	Example
*	Represents all possible values for a field	"* * * * *" means once a minute
M-N	Defines ranges	"1-5" in the DOW field would mean Monday to Friday
/	Define skips through a range	"*/5" in the MINUTE field would mean every five minutes
Comma-separated list	Indicates a list of valid values	"15,45" in the MINUTE field would mean at 15 and 45 minutes past every hour
Shorthand values		@yearly, @annually, @monthly, @weekly, @daily, @midnight, and @hourly

Poll the SCM

Polling involves checking the version control server at regular intervals for any changes that have been committed.

- If any changes have been made to the source code in the project, Jenkins kicks off a build.
- SCM polling is easy to configure and uses the same cron syntax as scheduled builds.

Trigger Builds Remotely

- Polling as frequently as possible works well for small projects but leads to SCM server saturation as build jobs pile up in big projects.
- Getting the SCM system to trigger the Jenkins build whenever a change is committed is a more precise strategy. This strategy is referred to as **triggering builds remotely**.

You can start a Jenkins build job remotely by invoking a URL of the form:
`http://SERVER/jenkins/job/PROJECTNAME/build`

Trigger Builds Remotely

- The details of triggering builds remotely are different for each version control system.
- In Subversion, you would need to write a post-commit hook script that would trigger a build.
- A more flexible approach with Subversion is to use the Jenkins Subversion API directly. This would automatically start any Jenkins build jobs monitoring this Subversion repository.
- This is set up by activating the Trigger builds remotely option and providing a special string that can be used in the URL

Assisted Practice

Polling SCM

Problem Statement: You have been asked to configure a freestyle build job in Jenkins that polls the SCM every night.

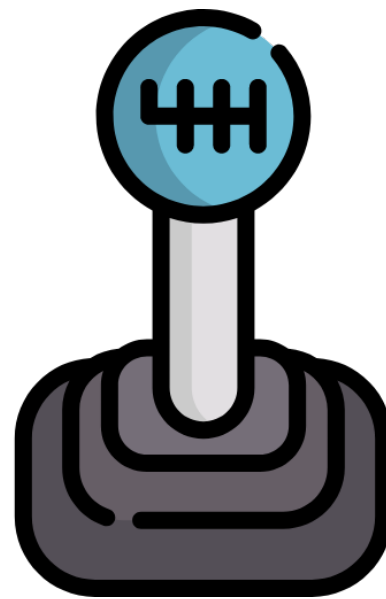
Steps to perform:

1. Create a Maven project
2. Configure a freestyle job that run Maven build steps
3. Define the regular expression for nightly builds
4. Test the scheduled build

Build Steps

Manual Build Jobs

- A build does not have to be triggered automatically.
- Some build jobs should only be started manually by human intervention.
 - For example: you may want to set up an automated deployment to a UAT environment that should only be started on the request of the QA.
- In such cases, you can simply leave the Build Triggers section empty.



Build Steps

Build steps are the basic building blocks for the Jenkins freestyle build process.

They are how you inform Jenkins exactly how you want your project built.

A build job may have one or more steps.

A build job may even occasionally have no steps.

In a basic Jenkins installation, you will be able to add steps to invoke Maven and Ant, as well as run OS-specific shell or Windows batch commands.

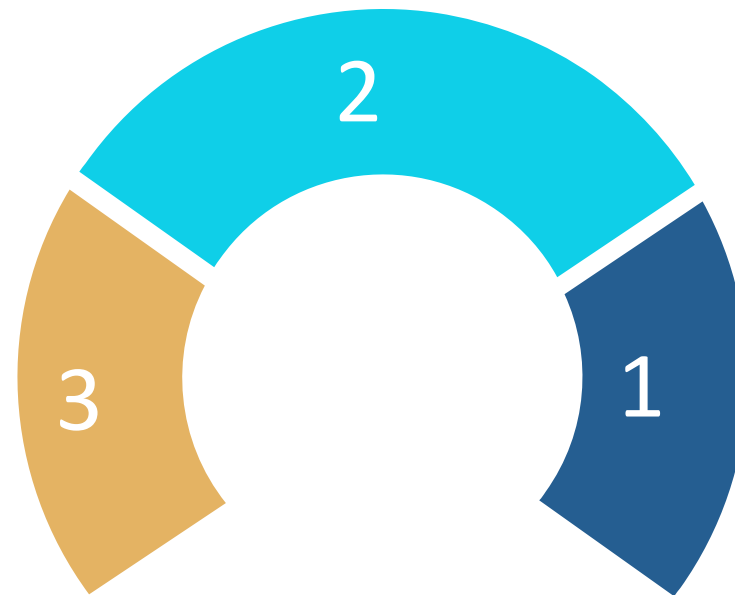
By installing additional plugins, you can also integrate other build tools, such as Groovy, Gradle, Grails, Jython, MSBuild, Phing, Python, Rake, and Ruby.

Maven Build Steps

Jenkins has excellent Maven support. Maven build steps are easy to configure and very flexible. The steps to define a Maven build are:

Pick a version of Maven to run if you have multiple versions installed. Jobs work with both Maven 2 and Maven 3.

Enter the Maven goals you want to run. You can specify as many goals as you want.



Pick the *Invoke top-level Maven targets* from the build step lists.

Maven Build Steps

Command-line options can also be provided in the Maven build steps. Some useful Maven options are:

-B or --batch-mode:

- This option informs Maven not to prompt for any input from the user and to just use the default values if required.
- If Maven does prompt for any input during the Jenkins build, the build hangs indefinitely.

-U or --update-snapshots:

- This option forces Maven to check for updated releases and snapshot dependencies on the remote repository.
- This makes sure that you are building with the latest snapshot dependencies and not any older local copies which may not be in sync with the latest version of source code.

Maven Build Steps

-Dsurefire.useFile=false

- This option forces Maven to write JUnit output to the console, rather than to text files in the target directory.
- Any test failure details are directly visible in the build job console output.
- This will still generate the XML files that Jenkins needs for its test reporting.

Maven Build Steps

The screenshot below shows adding a build step to a freestyle build job:

The screenshot displays the 'Build' configuration section of a Jenkins job. It features a list of build steps, with 'Invoke top-level Maven targets' currently selected. The configuration for this step includes a 'Maven Version' dropdown set to 'Maven 2.2.1' and a 'Goals' text field containing 'clean install -B -U -Dsurefire.useFile=false'. To the right of the goals field are buttons for 'Advanced...' and 'Delete'. Below the configuration, an 'Add build step' button is shown with a dropdown menu open, listing various build tools: 'Execute shell', 'Invoke top-level Maven targets' (highlighted), 'Execute Windows batch command', 'Invoke Ant', 'Invoke Gant script', 'Invoke Gradle script', 'Build With Grails', 'Execute Groovy script', and 'Execute system Groovy script'. On the right side of the build step list, there are three question mark icons. The bottom of the screenshot shows a partial view of a file path: '...reports/* vml'.

Ant Build Steps

Freestyle build jobs work well with Ant.

Apache Ant is a popular Java build scripting tool. A large number of Java projects rely on Ant build scripts.

Ant is not only used as a primary build scripting tool, but can also be used to do other tasks. Even if your project uses Maven, you can call Ant scripts to do more specific tasks.

Ant libraries are available for many development tools and low-level tasks, such as using SSH, or working with proprietary application servers.

Ant Build Steps

Configuring a basic Ant build step:

- Provide the version of Ant you want to use.
- Give the name of the target you want to invoke.

Advanced button:

- Advanced options include specifying a different build script, or a build script in a different directory other than the default build.xml in the root directory.
- You can also specify the properties and JVM options just as you can for Maven.

Jenkins Environment Variables

Using Jenkins Environment Variables in Your Builds

In any build step, you can obtain information from Jenkins about the current build job. When Jenkins starts a build step, the following environment variables become available to the build script:

Environment Variable	Value
BUILD_NUMBER	The current build number.
BUILD_ID	A timestamp for the current build id in the format YYYY-MM-DD_hh-mm-ss.
JOB_NAME	The name of the job.
BUILD_TAG	A way to identify the current build job in the form of Jenkins -\${JOB_NAME}-\${BUILD_NUMBER}.
EXECUTOR_NUMBER	A number identifying the executor running the build among the executors of the same machine.

Using Jenkins Environment Variables in Your Builds

Environment Variable	Value
NODE_NAME	The name of the slave if the build is running on a slave or "" if the build is running on master.
NODE_LABELS	The list of labels associated with the node that the build is running on.
JAVA_HOME	The JAVA_HOME of the specified JDK. When this variable is set, PATH is also updated to \$JAVA_HOME/bin.
WORKSPACE	The absolute path of the workspace
JENKINS_URL	The full URL of the Jenkins server.
JOB_URL	The full URL for this build job.
BUILD_URL	The full URL for this build.

Using Jenkins Environment Variables in Your Builds

These variables are easy to access. You can access them in an Ant script using the **<property>** tag as follows:

```
<target name="printinfo">  
    <property environment="env" />  
    <echo message="${env.BUILD_TAG}"/>  
</target>
```

Using Jenkins Environment Variables in Your Builds

In Maven, you can access the variables using the ***env.*** prefix or using the Jenkins environment variable. In the following pom.xml file, the project URL will point to the Jenkins build job that ran the mvn site build:

```
<project...>
...
<groupId>com.simpli.hello</groupId>
<artifactId>hello-core</artifactId>
<version>0.0.55-SNAPSHOT</version>
<name>hello-core</name>
<url>${JOB_URL}</url>
```

Post Build Actions

Post Build Actions

Once the build is completed, you might want to perform the following actions:

- Archive some of the generated artifacts.
- Report on test results.
- Notify people about the results.



Reporting on Test Results

Reporting on test results is one of the most important requirements of a build job.

A build job should report test failures as well as number of tests executed, time taken for their execution, and other details.

Jenkins provides great support for test reporting. Check the *Publish JUnit test result report* option, and provide a path to your JUnit report files.

You can use a wildcard expression to include JUnit reports from a number of different directories and Jenkins will aggregate the results into a single report.

Reporting on Test Results

The screenshot below shows the *Post-build Actions* page for reporting on test results:

Post-build Actions

☐ Publish Javadoc

☐ Archive the artifacts

☐ Aggregate downstream test results

☒ Publish JUnit test result report

Test report XMLs

/target/surefire-reports/.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



Archiving Build Results

The primary goal of a build job is generally to build something, such as, an artifact.

An artifact might be a binary executable, like a JAR or WAR file, for a Java project or some related deliverable, like documentation or source code.

- A build job can store one or many different artifacts.
- It can keep only the latest copy or every artifact ever built.
- You can configure Jenkins to store your artifacts by checking the *Archive the artifacts* checkbox in the Post-build Actions, and specifying the artifacts you want to store.

Archiving Build Results

Wildcards:

- You can also use wildcards to specify the files to archive.
- Using wildcards makes your build less dependent on version control set up.

Exclude option:

- If you are using wildcards to find your artifacts, there might be directories that you would have to exclude from the search.
- This is done by filling in the Excludes field. Enter a pattern to match the files that you don't want to archive.

Archiving Build Results

Discard option:

- Archived artifacts can take a lot of disk space, especially, if the builds are frequent.
- Checking the Discard all but the last successful/stable artifact option limits the amount of artifacts stored.
- Jenkins can then, only keep artifacts from the last stable build, the artifacts of the latest unstable build following the stable build, and the artifacts from the last failed build that happened.

Archiving Build Results

- Archived build artifacts appear on the build results page.
- The most recent build artifacts are also displayed on the build job home page.
- You can also use permanent URLs to access the most recent build artifacts.
- This lets you reuse the latest artifacts from your builds, either in other Jenkins build jobs or in external scripts.
- There are three URLs available:



Last stable
build

Last
successful
build

Last
completed
build

Archiving Build Results

The format of the artifact URLs takes the following form:

Latest stable build:

```
<server-url>/job/<build-job>/lastStableBuild/artifact/<path-to-artifact>
```

Latest successful build:

```
<server-url>/job/<build-job>/lastSuccessfulBuild/artifact/<path-to-artifact>
```

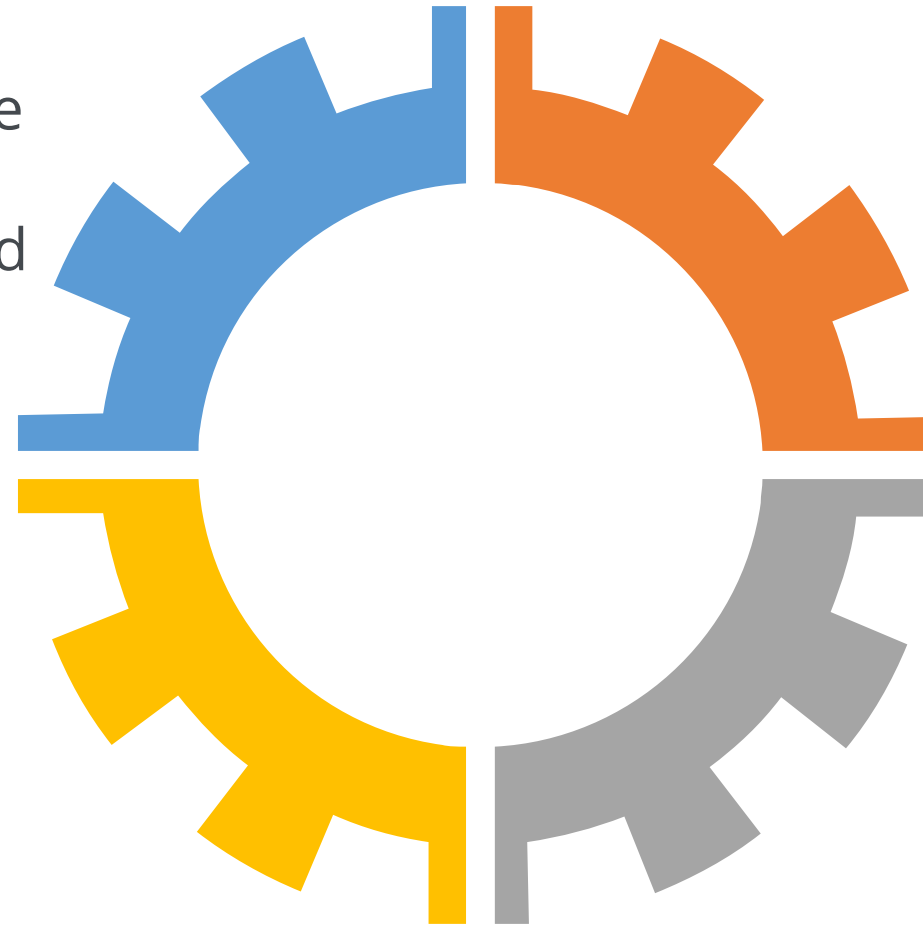
Latest completed build:

```
<server-url>/job/<build-job>/lastCompletedBuild/artifact/<path-to-artifact>
```

Stable and Successful Builds

Unit test failures, insufficient code coverage, or other code quality metrics issues, could cause a build to be marked as unstable.

A build is successful when the compilation reports no errors. A build is stable if it was built successfully, and no publisher reported it as unstable.



A stable build is always successful, but a build can also be successful without being stable.

A completed build is simply a build that has finished, no matter what its result. The archiving step will run regardless of the outcome of the build.

Notifications

- The most important use of a CI server is to notify people when a build breaks. This comes under the *Notification* page in Jenkins.
- Jenkins provides support for email notifications out of the box.
- This can be activated by checking the *E-mail Notification* checkbox in the *Post-build Actions*.
- Enter the email addresses of the team members who will need to know when the build breaks.

When the build breaks, Jenkins will send an email to the users in this list with a link to the broken build.

Building Other Projects

Other build jobs can be started in the Post-build Actions. This is done using the Build other project option.

This can be used to organize your build process in several, smaller steps, rather than one long build job.

These projects will only be triggered if the build was stable.

You also have the option to trigger another build job even if the current build is unstable. This might be useful to run a code quality metrics reporting build job after a project's main build job, even if there are test failures in the main build.

Assisted Practice

Post-Build Actions

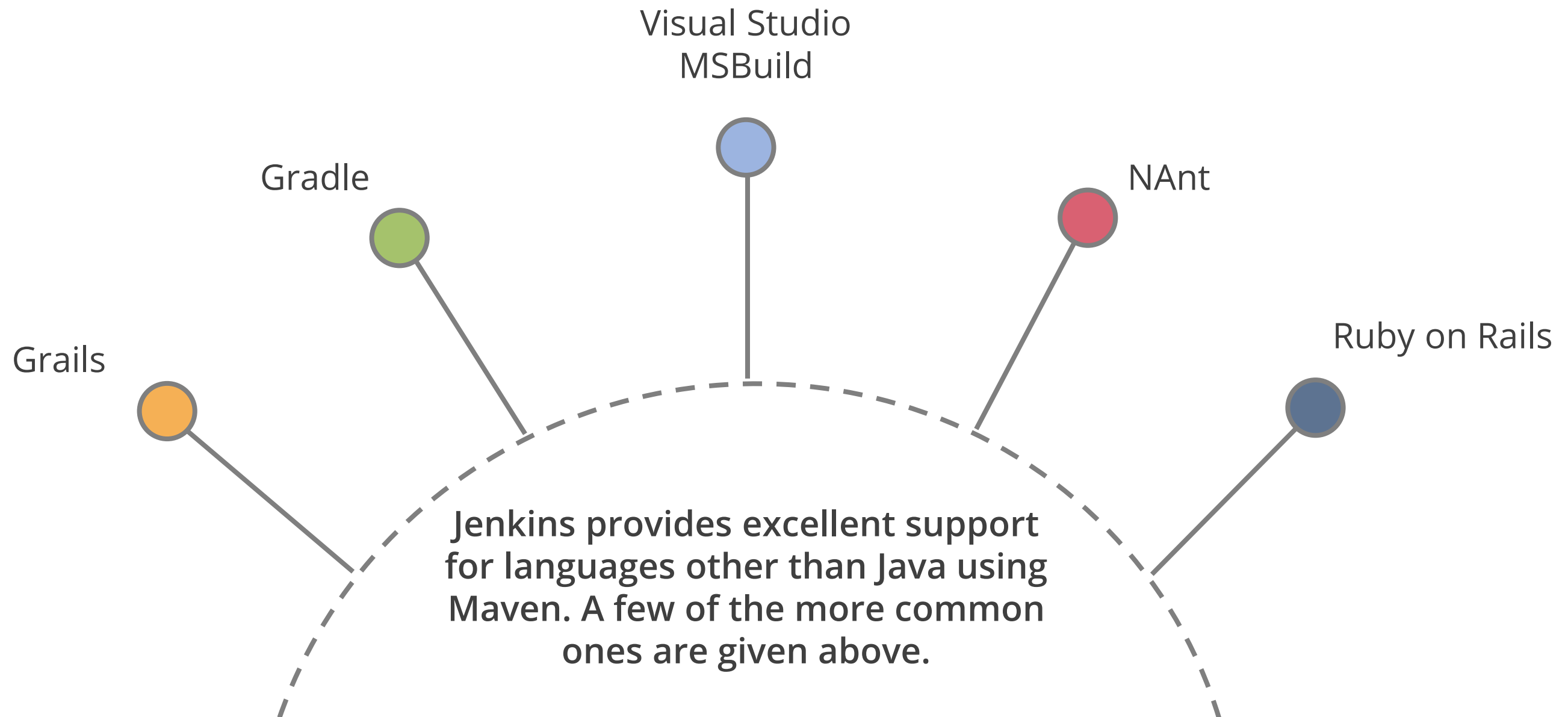
Problem Statement: You have been asked to configure a freestyle build job with post-build actions.

Steps to perform:

1. Configure SMTP settings to enable email alerts in Jenkins
2. Define a freestyle build job
3. Configure post-build actions to send email alerts if build fails
4. Test the job and the SMTP settings

Using Jenkins with Other Languages

Using Jenkins with Other Languages



Building Projects with Grails

Building Projects with Grails



Building Projects with Grails

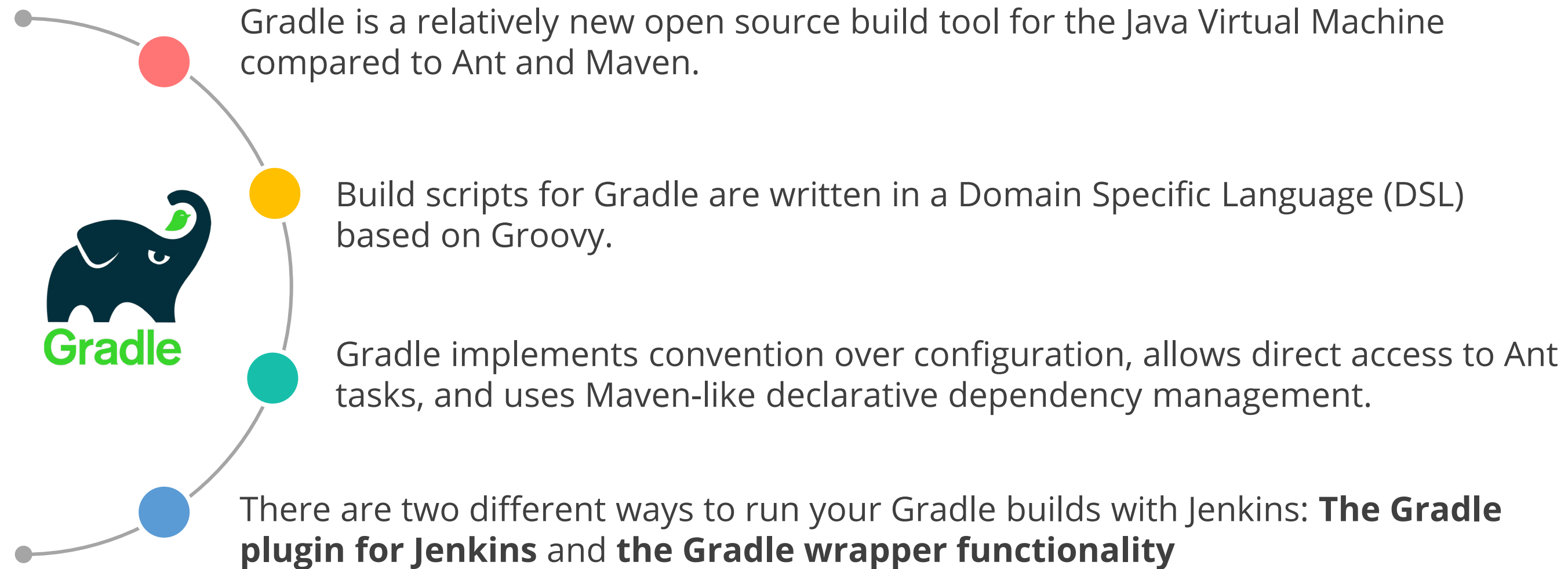
The screenshot below shows creating a freestyle build job with the Jenkins Grails plugin:

The screenshot displays the 'Build With Grails' configuration interface within a Jenkins job configuration page. The interface is organized into a sidebar on the left and a main configuration area on the right. The sidebar includes a 'Build' section with a 'Build With Grails' icon and label. The main area contains several configuration fields: 'Grails Installation' is a dropdown menu set to 'Grails 1.3.4'; 'Force Upgrade' is a checked checkbox; 'Targets' is a text field containing 'clean "test-app -unit -non-interactive"' with a help icon; 'server.port' is an empty text field; 'grails.work.dir' is an empty text field; 'grails.project.work.dir' is an empty text field; 'Project Base Directory' is an empty text field; and 'Properties' is a large empty text area. Each text field has a corresponding help icon. At the bottom, there is a label 'Additional system properties to set (optional)'.

Build	
Build With Grails	
Grails Installation	Grails 1.3.4
	Select a Grails installation to use
Force Upgrade	<input checked="" type="checkbox"/>
	Run 'grails upgrade --non-interactive' first
Targets	clean "test-app -unit -non-interactive" ?
	Specify target(s) to run separated by spaces (optional).
server.port	
	Specify a value for the server.port system property (optional)
grails.work.dir	
	Specify a value for the grails.work.dir system property (optional)
grails.project.work.dir	
	Specify a value for the grails.project.work.dir system property (optional)
Project Base Directory	
	Specify a path to the root of the Grails project (optional)
Properties	
	Additional system properties to set (optional)

Building Projects with Gradle

Building Projects with Gradle



The Gradle Plugin For Jenkins

The Gradle plugin for Jenkins can be installed from the *Manage Plugins* page.

Click *Install* and restart your Jenkins instance to use the Gradle plugin.

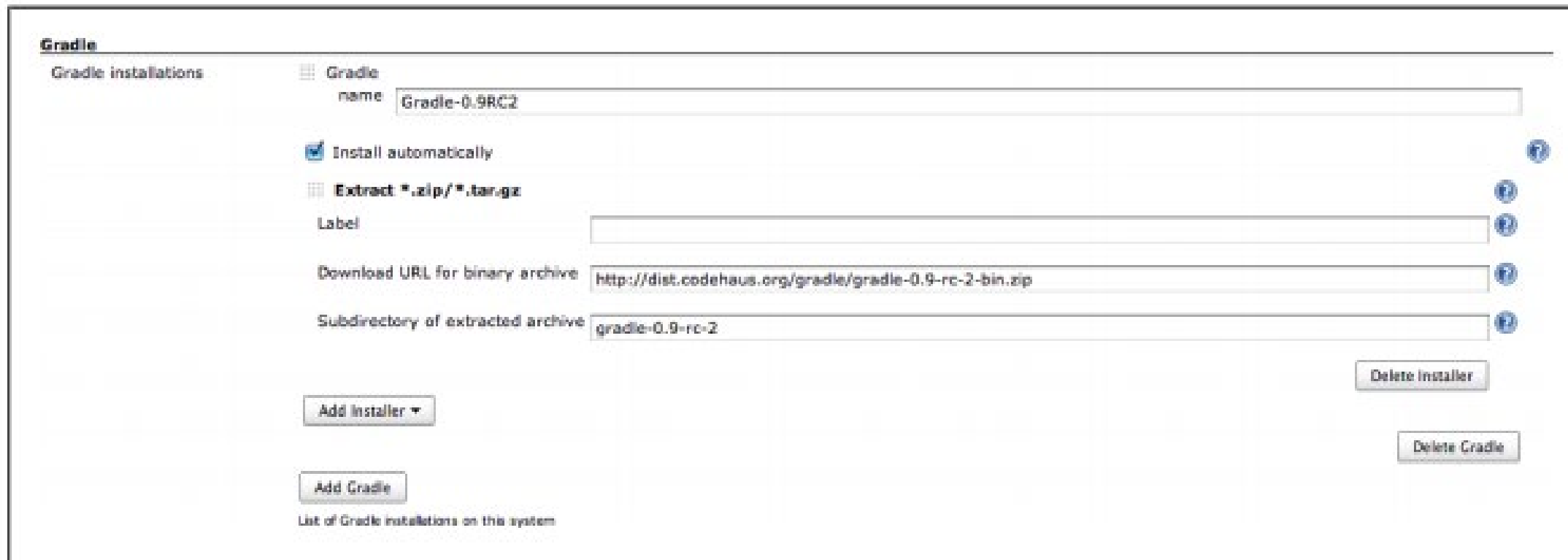
Click the *Add Gradle* button to add a new Gradle installation, and enter an appropriate name.

A new Gradle section will be added in the Configure System screen.

On adding a build step to a Freestyle build job, you will have a new option called *Invoke Gradle script*, which lets you add Gradle specific settings to your build job.

Configuring the Gradle Plugin

The screenshot below shows how to configure the Gradle plugin for Jenkins:



The screenshot displays the Jenkins configuration page for the Gradle plugin. On the left, a sidebar shows 'Gradle' as the selected section under 'Gradle installations'. The main area contains the following fields and controls:

- name:** A text field containing 'Gradle-0.9RC2'.
- Install automatically:** A checkbox that is checked.
- Extract *.zip/*.tar.gz:** A checkbox that is checked.
- Label:** An empty text field.
- Download URL for binary archive:** A text field containing 'http://dist.codehaus.org/gradle/gradle-0.9-rc-2-bin.zip'.
- Subdirectory of extracted archive:** A text field containing 'gradle-0.9-rc-2'.
- Buttons:** 'Add Installer' (with a dropdown arrow), 'Delete Installer', 'Add Gradle', and 'Delete Gradle'.
- Footer:** A link labeled 'List of Gradle installations on this system'.

Assisted Practice

Gradle Build

Problem Statement: You have been asked to set up a freestyle job that build projects with Gradle in Jenkins.

Steps to perform:

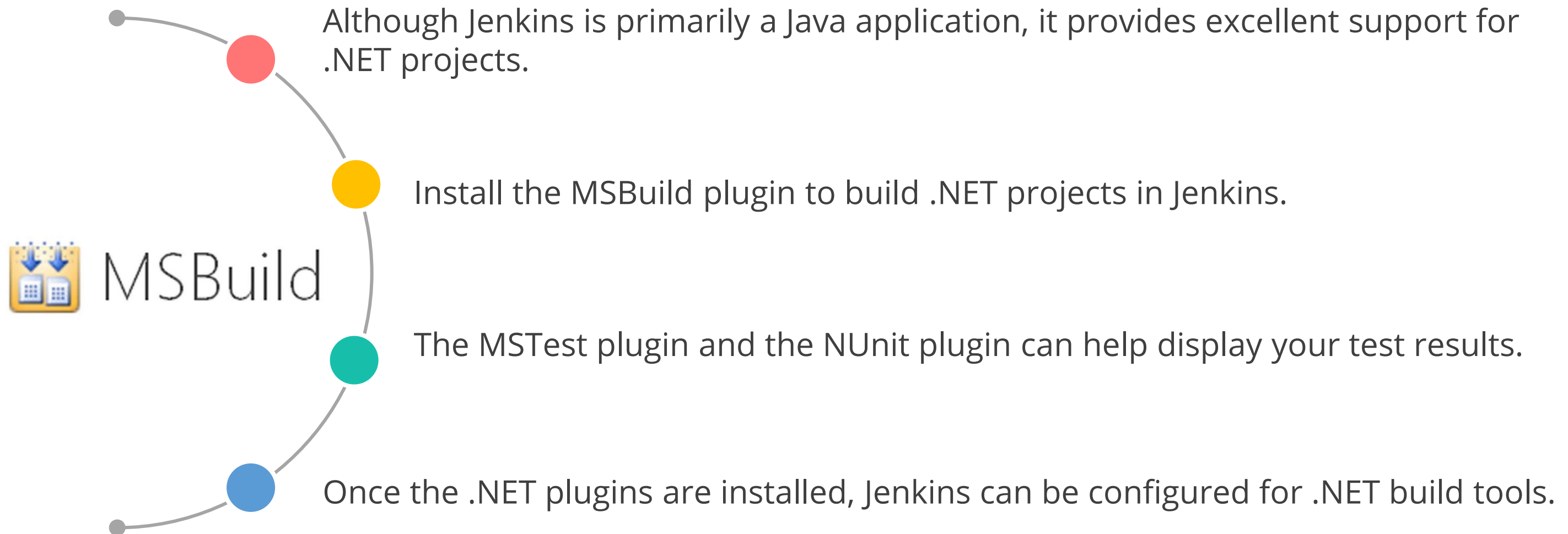
1. Configure Gradle plugin for Jenkins
2. Create a Gradle project
3. Build a project with Gradle in Jenkins

Incremental Builds

- Gradle runs its builds incrementally while running a Gradle build job with unchanged sources.
- If the output of a Gradle task is still available and the sources haven't changed since the last build, Gradle can skip the task execution. It marks the task as up-to-date.
- This incremental build feature can decrease the duration of a running build job considerably.
- If Gradle evaluates the test task as up-to-date, the execution of unit tests is also skipped. This can cause problems when running Gradle build and the Jenkins job may be marked as failed.

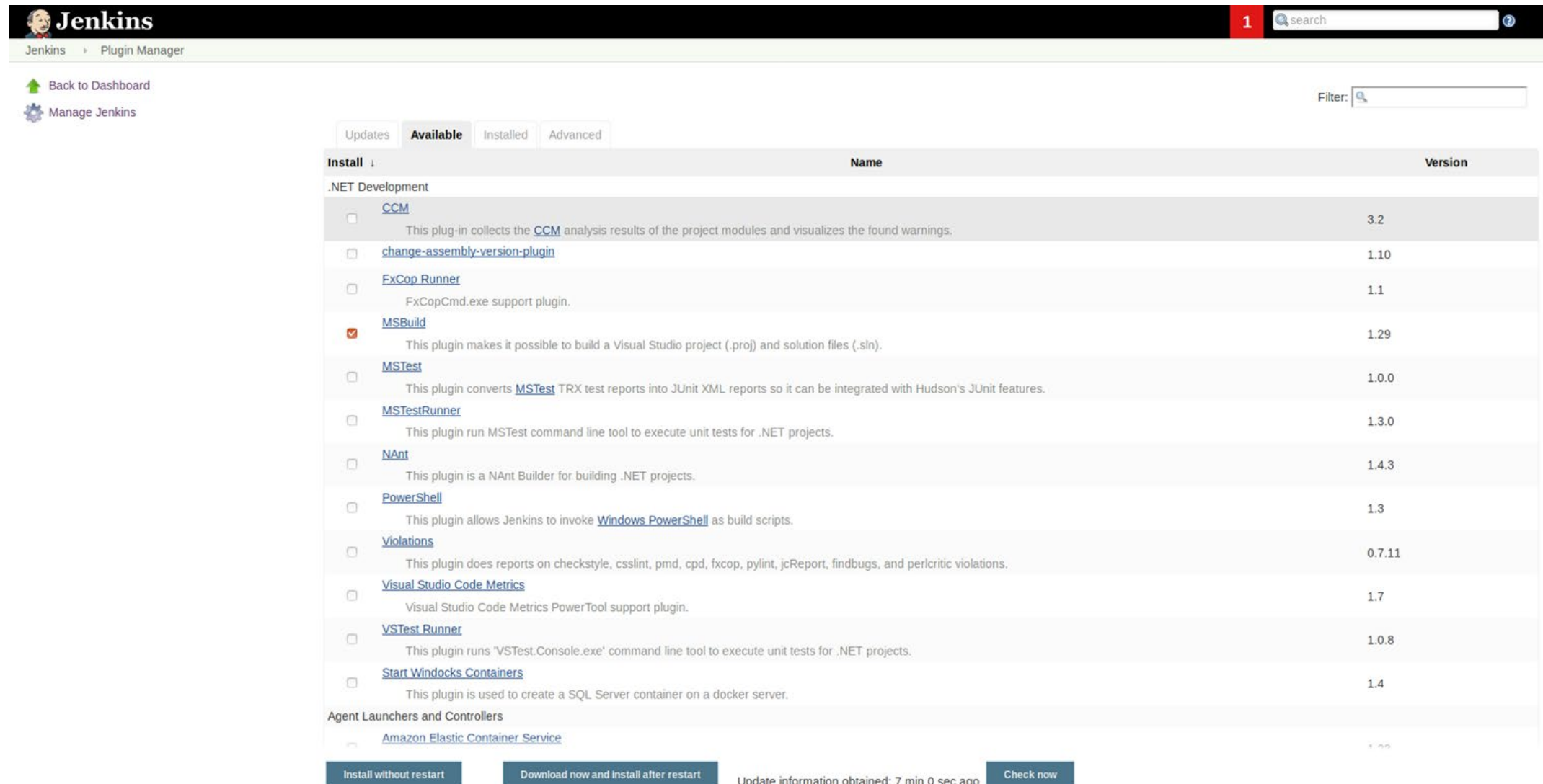
Building Projects with MSBuild

Building Projects with MSBuild



Building Projects with MSBuild

The screenshot below shows the plugins listed in the Manage Plugins screen. Select the required plugins and click on *Install*.



The screenshot shows the Jenkins Manage Plugins interface. The 'Available' tab is selected, displaying a list of plugins under the 'Available' section. The 'MSBuild' plugin is selected, indicated by a checked checkbox. The list includes various plugins for .NET Development, Agent Launchers and Controllers, and other tools.

Name	Version
CCM	3.2
change-assembly-version-plugin	1.10
FxCop Runner	1.1
MSBuild	1.29
MSTest	1.0.0
MSTestRunner	1.3.0
NAnt	1.4.3
PowerShell	1.3
Violations	0.7.11
Visual Studio Code Metrics	1.7
VSTest Runner	1.0.8
Start Windocks Containers	1.4

Buttons at the bottom: [Install without restart](#), [Download now and install after restart](#), [Check now](#). Update information obtained: 7 min 0 sec ago.

Parameterized Build Jobs


Parameterized Build Jobs




The Parameterized Build plugin lets you configure parameters for a build job that are either entered by the user when the build job is triggered, or from another build job.



For a deployment build job, you can either choose the target environment from a drop-down list or specify the version of the application.



When running a build job involving web tests, you can specify the browser to run the Selenium or WebDriver tests in. You can also upload files to be used by build jobs.



Jenkins simply provides an interface to enter values for the parameters. It is the job of the build script to analyze and process the parameter values correctly.

Creating a Parameterized Build Job

Install the *Build with Parameters* plugin via the Plugin Manager screen.

Select the *This build is parameterized option* and click on *Add Parameter* to add a new build job parameter.

Pick the parameter type in the drop-down list to add a parameter to your build job.

Configure the details of the parameter. You can choose from several different parameter types, like Strings, Booleans, and drop-down lists.

Enter the configuration details based on the type you choose. All parameter types, with the exception of the File parameter, have a name and a description, and most often a default value.

Creating a Parameterized Build Job

The screenshot below shows how to create a parameterized build in Jenkins:

The screenshot displays the Jenkins configuration interface for a new build job. The 'General' tab is selected, showing a 'Description' text area, a 'Discard old builds' checkbox, and a checked 'This project is parameterized' checkbox. An 'Add Parameter' dropdown menu is open, listing various parameter types: Boolean Parameter, Choice Parameter, Credentials Parameter, File Parameter, Multi-line String Parameter, Password Parameter, Run Parameter, and String Parameter. The 'Source Code Management' section shows 'None' selected. An 'Advanced...' button is visible on the right.

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

Description

[Plain text] [Preview](#)

☐ Discard old builds

☒ This project is parameterized

Add Parameter ▾

- Boolean Parameter
- Choice Parameter
- Credentials Parameter
- File Parameter
- Multi-line String Parameter
- Password Parameter
- Run Parameter
- String Parameter

☐ Disable this project

☐ Execute concurrent builds

Advanced...

Source Code Management

☒ None

Creating a Parameterized Build Job

The screenshot below shows adding a String parameter, *VERSION*, to a parameterized build:

✓ This project is parameterized

String Parameter

Name: VERSION

Default Value: RELEASE

Description:

[Plain text] [Preview](#)

☐ Trim the string

Add Parameter ▾

Assisted Practice

Parameterized Builds

Problem Statement: You have been asked to create and run a parameterized build in Jenkins.

Steps to perform:

1. Create a freestyle build in Jenkins
2. Define a string parameter
3. Build the project

Adapting Builds for Parameterized Build Scripts

- Once you have added a parameter, you need to configure the build scripts to use it.
- Choosing the parameter name is important as Jenkins will pass this name through as an environment variable when it runs the build job.
- It is good practice to put them all in upper case to make the environment variables portable across operating systems.
- You can also use environment variables from your build scripts.

Adapting Builds for Parameterized Build Scripts

In an Ant or Maven build, you can use the special **env** property to access the current environment variables:

```
<target name="printversion">  
  <property environment="env" />  
  <echo message="${env.VERSION}"/>  
</target>
```

Another option is to pass the parameter into the build script as a property value.

Building from a Subversion Tag



The parameterized trigger provides a special support for Subversion that allows you to build against a specific Subversion tag.

This support for Subversion is useful if you want to run a release build using a tag generated by a previous build job. For example, an upstream build job may tag a particular revision.

You can configure a Jenkins build to run against a selected tag by using the *List Subversion Tag* parameter type and providing the Subversion repository URL pointing to the tags directory of your project.

Alternatively, you can use the standard Maven release process to generate a new release.

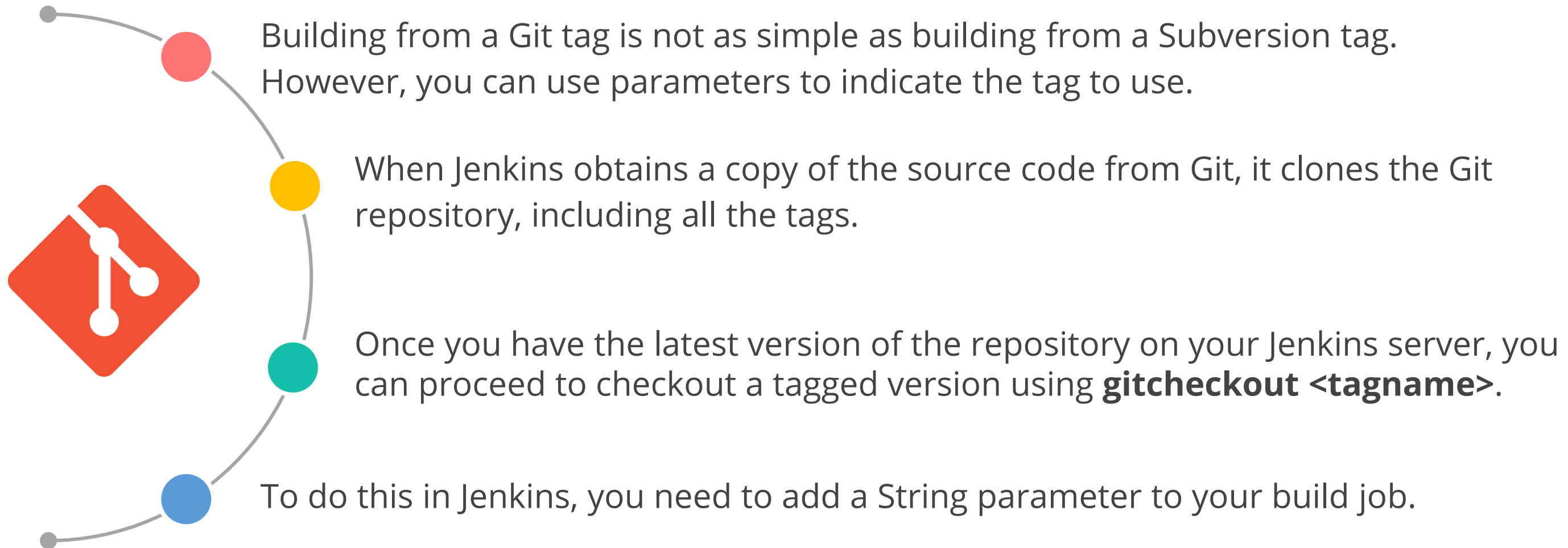
Building From A Subversion Tag

The screenshot below shows how to use parameters to build from a Subversion tag.



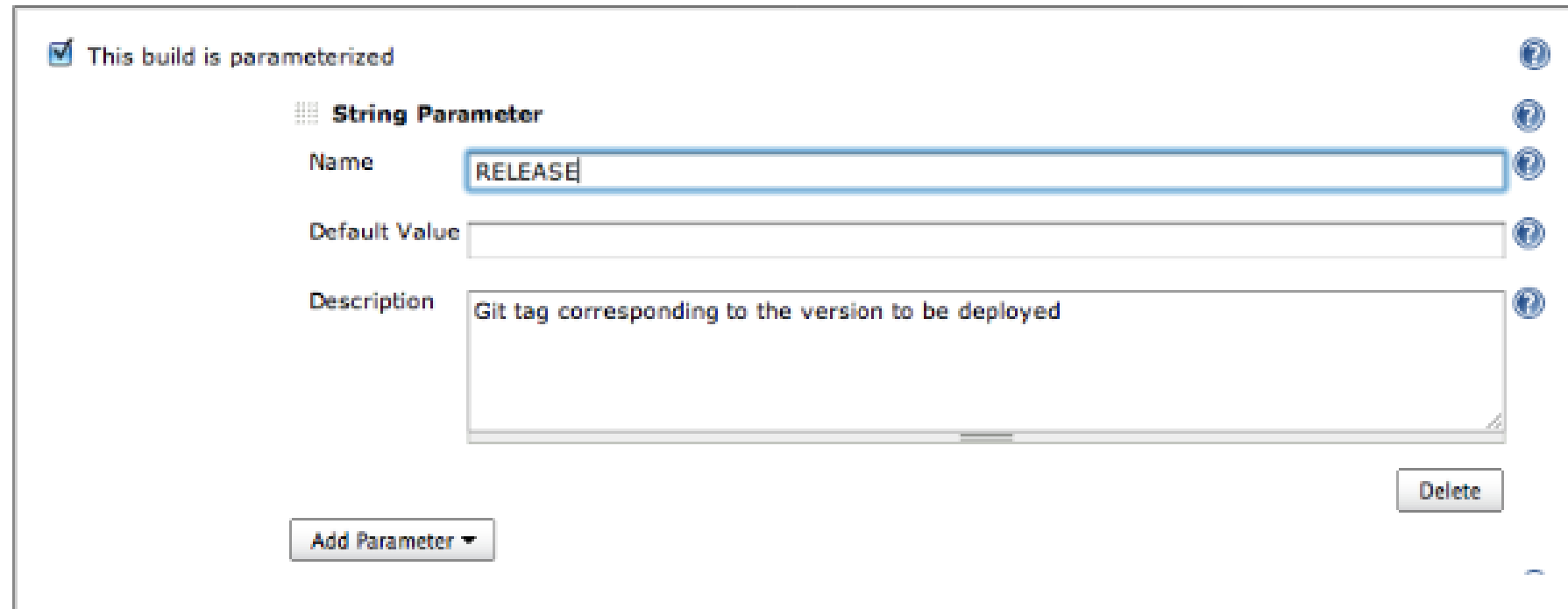
The screenshot displays the Jenkins build configuration interface for a parameterized build. At the top, a checkbox labeled "This build is parameterized" is checked. Below this, a section titled "List Subversion tags" contains two input fields: "Name" with the value "Release" and "Repository URL" with the value "svn://localhost/gameoflife/tags". To the right of the "Repository URL" field is a "Delete" button. At the bottom left of the configuration area is an "Add Parameter" button with a dropdown arrow. Three help icons (question marks) are visible on the right side of the configuration area.

Building From A Git Tag



Building From A Git Tag

The screenshot below shows how to configure the parameter for a Git tag:



The screenshot displays the Jenkins configuration interface for a build parameter. At the top, a checkbox labeled "This build is parameterized" is checked. Below this, a section titled "String Parameter" is shown. It contains three fields: "Name" with the value "RELEASE", "Default Value" which is empty, and "Description" with the text "Git tag corresponding to the version to be deployed". Each field has a help icon (question mark) to its right. At the bottom left of the parameter section is an "Add Parameter" button with a dropdown arrow, and at the bottom right is a "Delete" button.

Assisted Practice

Building From Tags

Problem Statement: You have been asked to configure a build job in Jenkins that builds from a Git tag.

Steps to perform:

1. Add the Git parameter plugin to Jenkins
2. Create a tag in a Git repository
3. Configure a freestyle build job to read from the tag
4. Test the project and build from the Git tag

Starting a Parameterized Build Job Remotely

You can start a parameterized build job remotely, by invoking the URL of the build job.

The typical form of a parameterized build job URL is:

`http://jenkins.acme.org/job/myjob/buildWithParameters?PARAMETER=Value`

- Note that the parameter names in the URL are case-sensitive, and the values need to be escaped like any HTTP parameter.
- If you are using a Run parameter, you need to provide the name of the build job as well as the run number.

Parameterized Build Job History

It is important to know what parameters were used to run a particular parameterized build. Jenkins stores these values in the build history, as shown below:



The screenshot shows the Jenkins web interface for a specific build. The top navigation bar includes the Jenkins logo, a search box, and a link to 'ENABLE AUTO REFRESH'. The breadcrumb trail indicates the path: Jenkins > gameoflife-deploy-to-uat > #3. On the left, a sidebar contains links for 'Back to Project', 'Status', 'Changes', 'Console Output', 'Configure', 'Parameters', 'Tag this build', 'Downstream build view', 'Previous Build', and 'Next Build'. The main content area displays 'Build #3 (Sep 19, 2010 2:40:27 AM)' with a blue sphere icon. Below this, it states 'Deployed version 0.0.24 to UAT'. To the right of the build title, there is a 'Delete this build' button and information indicating the build 'Started 4 mo 24 days ago' and 'Took 4.1 sec on master'. Below the build title, there is a link to 'edit description'. Further down, a 'Revision: 48' section shows 'No changes' with a notepad icon. At the bottom, it indicates the build was 'Started by user anonymous' with a person icon.

Parameterized Triggers



When triggering another build job from within a parameterized one, it is helpful to have the provision to pass the parameters of the current build job to the new one.



If you have an application that needs to be tested against several different databases, you can set up a parameterized build job that accepts the target database as a parameter.



You will need to start a series of builds, all of which will need this parameter.



You cannot do this using the conventional Build other projects option in the Post-Build Actions section. This can be done using the Jenkins Parameterized Trigger plugin.

Assisted Practice

Remote Triggering Parameterized Builds

Problem Statement: You have been asked to configure a parameterized build in Jenkins that can be triggered remotely.

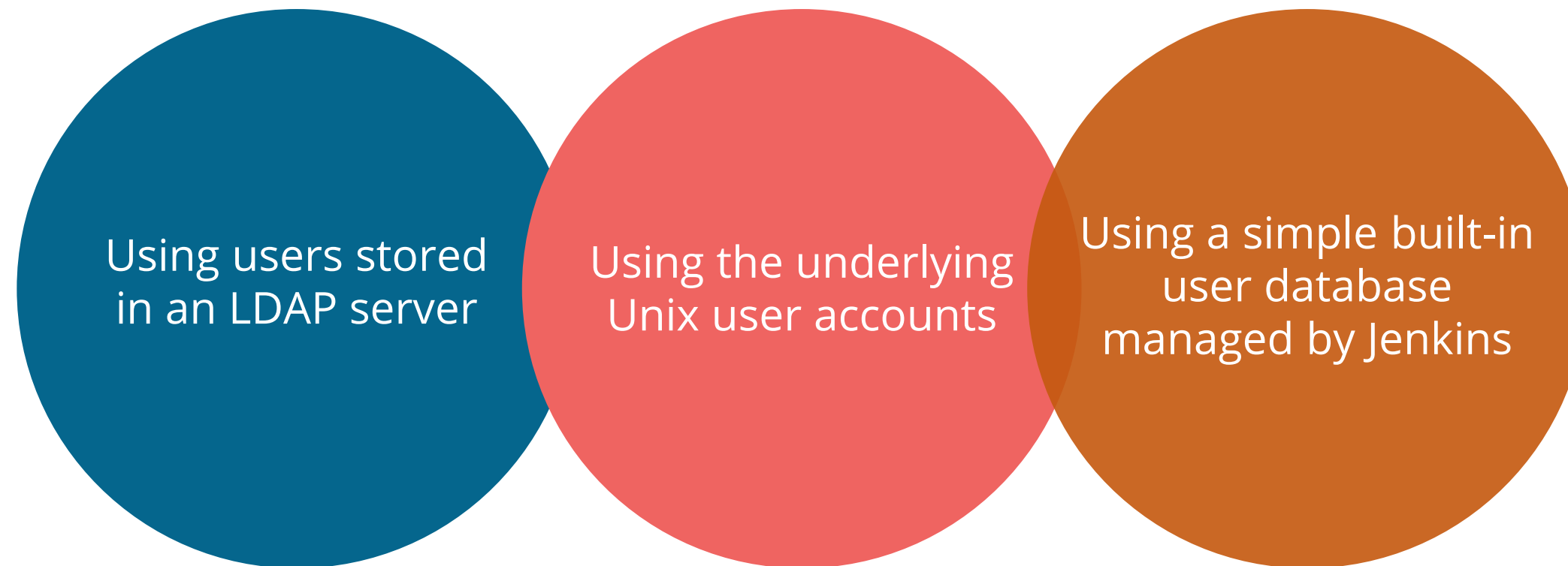
Steps to perform:

1. Create a freestyle build in Jenkins
2. Define a string parameter
3. Trigger the parameterized build remotely

Enabling Security in Jenkins

Enabling Security in Jenkins

- Setting up basic security in Jenkins is easy.
- In the main configuration page, check the *Enable security* checkbox. This will display a number of options.
- The first section, *Security Realm*, determines where Jenkins will look for users during authentication. The options include :



Enabling Security in Jenkins

The screenshot below shows how to enable security in Jenkins:



The screenshot displays the Jenkins 'Configure' page for security settings. The 'Enable security' checkbox is checked. The 'TCP port for JNLP slave agents' is set to 'Random'. The 'Markup Formatter' is set to 'Raw HTML'. The 'Access Control' section is expanded, showing the 'Security Realm' options. 'Jenkins's own user database' is selected, and the 'Allow users to sign up' checkbox is also checked. Other options like 'Delegate to servlet container', 'LDAP', and 'Unix user/group database' are unselected. Each setting has a help icon (question mark in a circle) to its right.

☒ Enable security

TCP port for JNLP slave agents: ☐ Fixed : ☒ Random ☐ Disable

Markup Formatter: Raw HTML
Treat the text as HTML and use it as is without any translation

Access Control

Security Realm

☐ Delegate to servlet container

☒ Jenkins's own user database

☒ Allow users to sign up

☐ LDAP

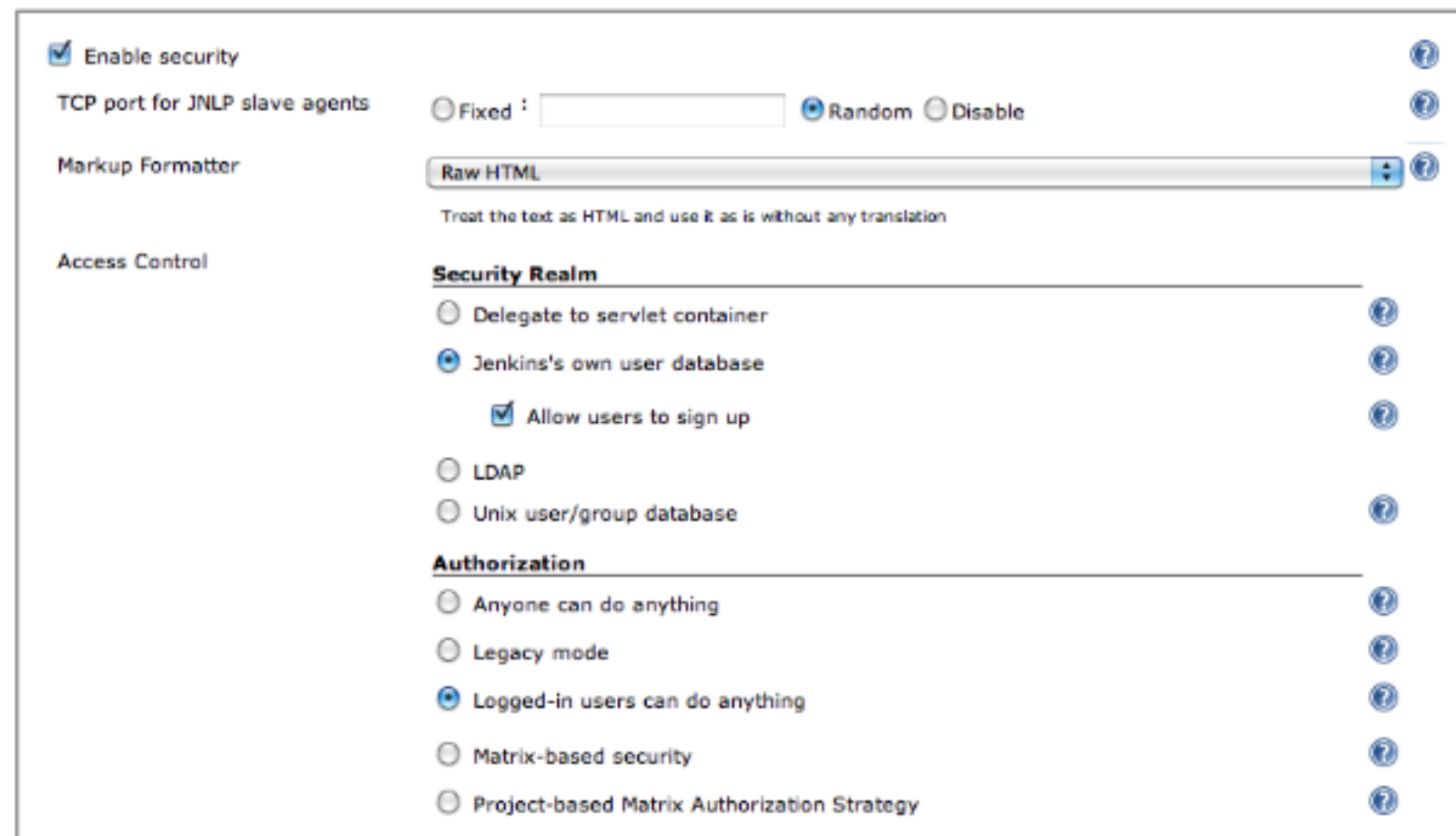
☐ Unix user/group database

Different Levels of Authentication

The second section, *Authorization*, determines what users can do once they are logged in.

- This ranges from simple options like *Anyone can do anything* or *Logged-in users can do anything*, to more sophisticated roles and project-based authorization policies.

The screenshot below shows the levels of authorization in Jenkins:



The screenshot displays the Jenkins Security Configuration page. It includes the following sections and options:

- Enable security:** ☒
- TCP port for JNLP slave agents:** ☐ Fixed : ☒ Random ☐ Disable
- Markup Formatter:**
 Treat the text as HTML and use it as is without any translation
- Access Control:**
 - Security Realm:**
 - ☐ Delegate to servlet container
 - ☒ Jenkins's own user database
 - ☒ Allow users to sign up
 - ☐ LDAP
 - ☐ Unix user/group database
 - Authorization:**
 - ☐ Anyone can do anything
 - ☐ Legacy mode
 - ☒ Logged-in users can do anything
 - ☐ Matrix-based security
 - ☐ Project-based Matrix Authorization Strategy

Matrix-based Security

- The first step to setup matrix-based security in Jenkins is to create an administrator.
- You can activate matrix-based security by selecting *Matrix-based security* in the Authorization section.
- Jenkins will display a table containing authorized users, and checkboxes corresponding to the various permissions that you can assign to these users as shown below:

Authorization

☐ Legacy mode

☐ Project-based Matrix Authorization Strategy

☐ Logged-in users can do anything

☐ Anyone can do anything

☒ Matrix-based security

User/group	Overall	Slave			Job					Run		View			SCM			
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure	Promote	Tag
Anonymous	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

User/group to add:

User Permissions

The individual permissions you can assign a Jenkins user are as follows:

Overall: This group covers basic system-wide permissions.

- Administer: Lets a user make system-wide configuration changes and other sensitive operations.
- Read: Provides read-only access to virtually all of the pages in Jenkins.

Slave: This group covers permissions about remote build nodes, or slaves.

- Configure: Create and configure new build nodes.
- Delete: Delete build nodes.

User Permissions

Job: This group covers job-related permissions.

- Create: Create a new build job.
- Delete: Delete an existing build job.
- Configure: Update the configuration of an existing build jobs.
- Read: View build jobs.
- Build: Start a build job.
- Workspace: View and download the workspace contents for a build job.
- Release: Start a Maven release for a project configured with the M2Release plugin.

User Permissions

Run: This group covers rights related to particular builds in the build history.

- Delete: Delete a build from the build history.
- Update: Update the description and other properties of a build in the build history.

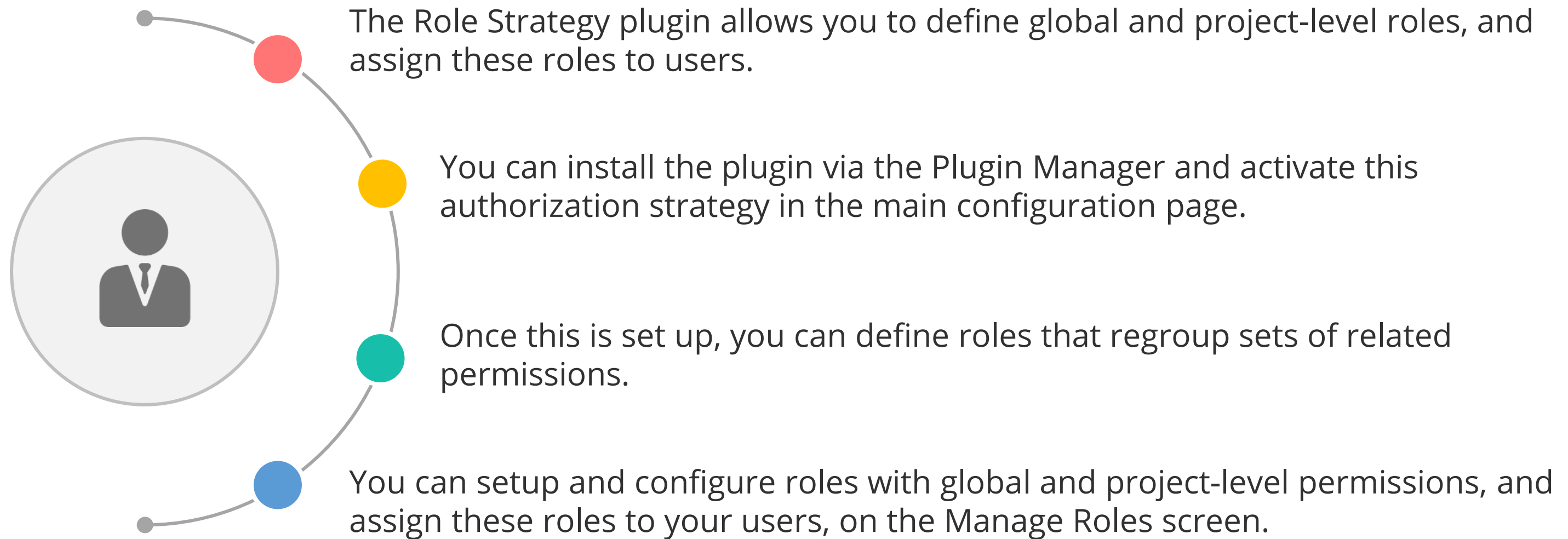
View: This group covers managing views.

- Create: Create a new view.
- Delete: Delete an existing view.
- Configure: Configure an existing view.

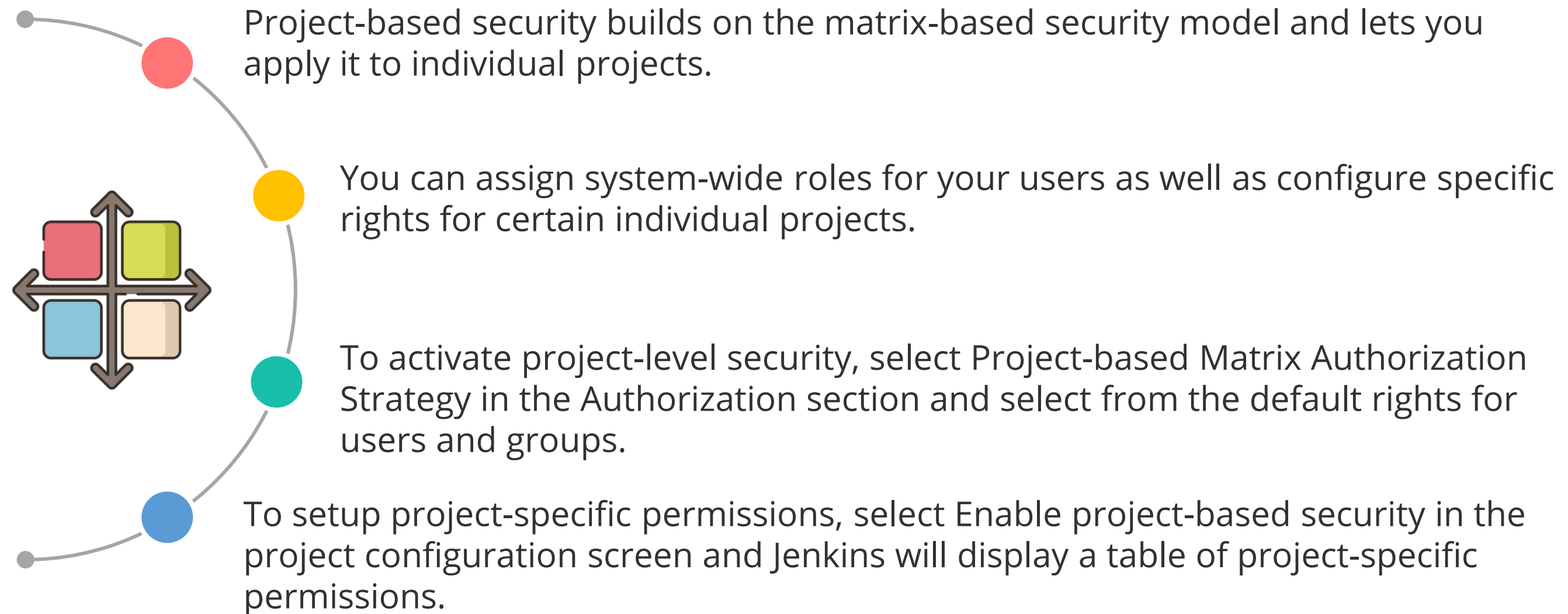
SCM: This group covers permissions related to your version control system.

- Tag: Create a new tag in the source code repository for a given build.

Role-based Security



Project-based Security



Assisted Practice

Enabling Security

Problem Statement: You have been asked to define the security policies in Jenkins.

Steps to perform:

1. Configure Security Realm to use Jenkins' own user database and allow users to sign up.
2. Configure Authorization to allow Logged-in users to do anything.
3. Set up a new user account

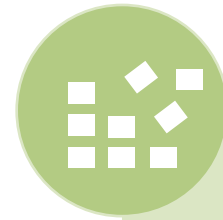
Build Pipelines and Promotions

Build Promotion

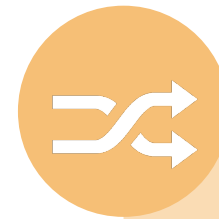
The Promoted Builds plugin lets you identify specific builds that have met additional quality criteria, and trigger actions on these builds. For example:



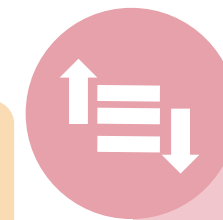
A default build job runs unit and some integration tests and produces a WAR file.



This WAR file is then reused for more extensive integration tests and then for a series of automated web tests.



If the build passes the automated web tests, you may deploy the application to a functional testing environment where it can be tested by human testers.



The deployment to this environment is implemented in the final build job. Once the testers have validated a version, it can be promoted into UAT, and then into production.

Configuring Build Promotions

The screenshot below shows how to configure a build promotion:

The screenshot displays a configuration window for a build promotion. At the top, a checkbox labeled "Promote builds when..." is checked. Below this, the "Promotion process" section contains a "Name" field with the value "promote-to-test" and an "Icon" dropdown menu showing "Gold star". The "Criteria" section has three options: "When the following downstream projects build successfully" (checked), "Only when manually approved", and "When the following upstream promotions are promoted". The "When the following downstream projects build successfully" option includes a "Job names" field with the value "phoenix-web-tests" and a checkbox for "Trigger even if the build is unstable". The "Actions" section features a "Build other projects" option, which includes a "Projects to build" field with the value "phoenix-deploy-to-test". At the bottom, there are buttons for "Add action", "Delete", and "Delete this promotion process".

☒ Promote builds when...

Promotion process

Name

Icon

Criteria

☒ When the following downstream projects build successfully

Job names

☐ Trigger even if the build is unstable

☐ Only when manually approved

☐ When the following upstream promotions are promoted

Actions

☒ Build other projects

Projects to build

Configuring Build Promotions



Build promotion can be either automatic, based on the result of a downstream build job, or manually activated by a user.

The build promotion for the build job in the screenshot will be automatically triggered when the automated web tests are successful.

You can also have certain build jobs that can only be promoted manually. Manual build promotion is used for cases where human intervention is needed to approve a build promotion. Deployment to UAT or production are common examples.

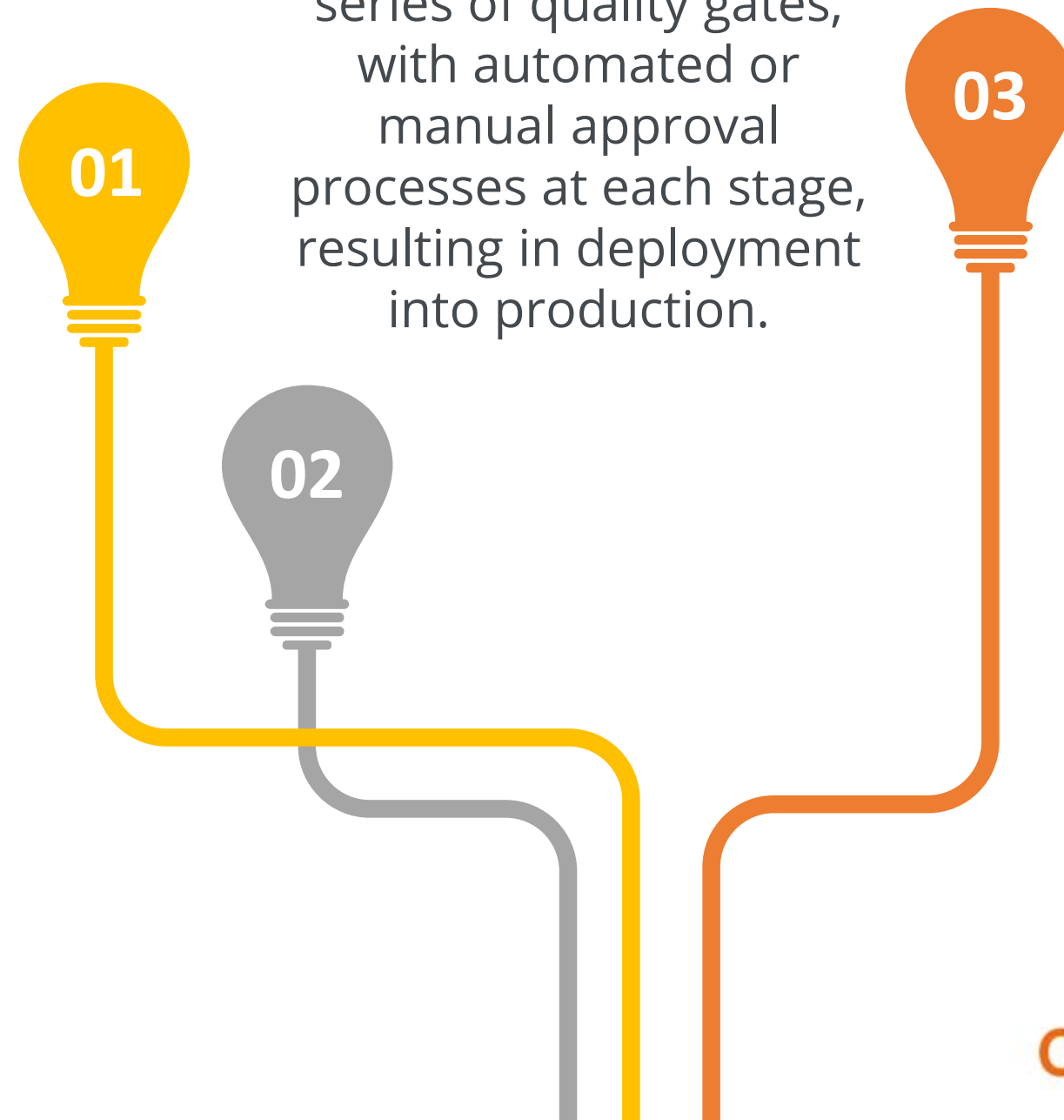
You also need to tell Jenkins what to do when the build is promoted. This is done by adding actions, making build promotions extremely flexible.

Build Pipelines

The Build Pipelines plugin takes the idea of build promotion further and helps you design and monitor deployment pipelines.

A deployment pipeline is a way of orchestrating your build through a series of quality gates, with automated or manual approval processes at each stage, resulting in deployment into production.

The Build Pipeline plugin provides an alternative way to define downstream build jobs.





Knowledge Check

Knowledge Check

1

Which of the following is NOT a type of Jenkins build job?

- A. Freestyle jobs
- B. Pipelines
- C. External jobs
- D. Maven projects



Knowledge Check

1

Which of the following is NOT a type of Jenkins build job?

- A. Freestyle jobs
- B. Pipelines
- C. External jobs
- D. Maven projects



The correct answer is **A**

While Pipelines and jobs can be used to achieve the same functionality, they're inherently different. A Pipeline is a Groovy-based definition of steps in a build.

Knowledge Check

2

Which of the following is NOT a post-build action?

- A. Archive build results
- B. Send notifications
- C. Delete workspace before build
- D. Delete workspace after build



Knowledge Check

2

Which of the following is NOT a post-build action?

- A. Archive build results
- B. Send notifications
- C. Delete workspace before build
- D. Delete workspace after build



The correct answer is **C**

'Delete workspace before build' is a *Build Environment* configuration option.

Knowledge Check

3

A multi-configuration job is also known as a _____.

- A. Hierarchical Job
- B. Recursive Job
- C. Multi-file Job
- D. Matrix Job



Knowledge Check

3

A multi-configuration job is also known as a _____.

- A. Hierarchical Job
- B. Recursive Job
- C. Multi-file Job
- D. Matrix Job



The correct answer is **D**

A multi-configuration job is also known as a matrix job.

Knowledge Check

4

How many artifacts can one build job store using the *Archive the artifacts* option?

- A. One
- B. Three
- C. Five
- D. Unlimited



Knowledge Check

4

How many artifacts can one build job store using the *Archive the artifacts* option?

- A. One
- B. Three
- C. Five
- D. Unlimited



The correct answer is **A**

You can only save one artifact per build job using the *Archive the artifacts* option. However, you can use wildcards to save any number of artifacts.

Key Takeaways

- 🕒 Jenkins supports four types of build jobs: Freestyle software project, Maven project, External job, and Multi-configuration job.
- 🕒 Once a build is completed, you can archive some of the generated artifacts, report on test results, and notify people about the results.
- 🕒 Jenkins also provides excellent support for languages like Grails, Gradle, MSBuild, NAnt, and Ruby on Rails.
- 🕒 The Parameterized Build plugin lets you configure parameters for a build job, that are either entered by the user when the build job is triggered or from another build job.



Lesson-End Project

Triggering Parameterized Builds

Problem Statement:

You're a DevOps engineer at ChefHelp, which is a web and mobile-based app company that provides users with various recipes and cooking tips. The company wants to automate categorizing recipes. You're required to set up a Jenkins job stream where a job gets triggered when a new recipe is added to the database. The job has to accept the file path as a parameter, run the program to find the category it belongs to, return the result and trigger another job to delete the recipe file from the server to avoid server space issues.

Requirements:

- The program should accept the file path as a command line argument.
- The parameterized build job should be designed to be triggered remotely.
- The delete temps job should be triggered only if the build job succeeds.

