**CHAPTER 1 – Introducing Jenkins 2 :**

**What is Jenkins 2:**

Jenkins 2 is used a bit loosely. In the specific context it is way to refer the new version of Jenkins that directly incorporate support for pipeline-as-code and other new features such as “jenkinsfile”.

Instead of filling in web forms to define jobs for Jenkins, users can now write programs using the Jenkins DSL and Groovy to define their pipelines and do other tasks.

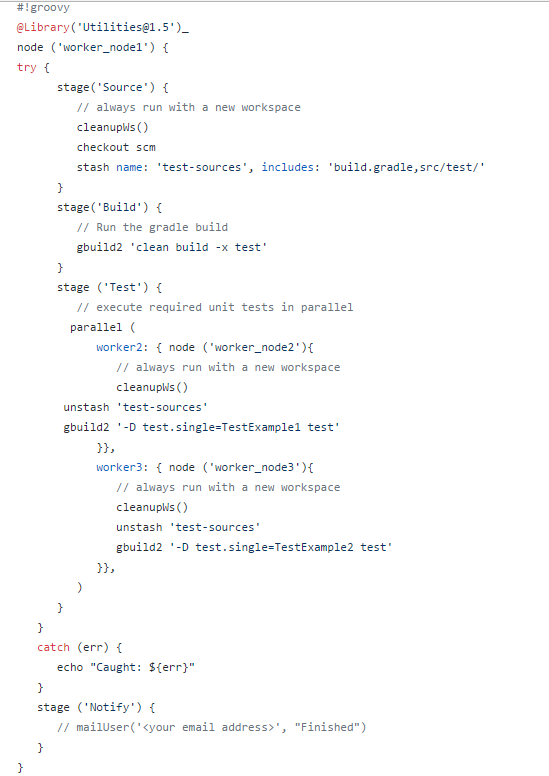
*DSL* here refers to *Domain-Specific Language*, the “programming language” for Jenkins.The DSL is Groovy-based and it includes terms and constructs that encapsulate Jenkins-specific functionality. An example is the node keyword that tells Jenkins that you will be programmatically selecting a node (formerly “master” or “slave”) that you want to execute this part of your program on.

**The Jenkins file:**

In Jenkins 2, your pipeline definition can now be separate from Jenkins itself. In past versions of Jenkins, your job definitions were stored in configuration files in the Jenkins home directory. This meant they required Jenkins itself to be able to see, understand, and modify the definitions (unless you wanted to work with the XML directly, which was challenging). In Jenkins 2, you can write your pipeline definition as a DSL script within a text area in the web interface. However, you can also take the DSL code and save it externally as a text file with your source code. This allows you to manage your Jenkins jobs using a file containing code like any other source code, including tracking history, seeing differences, etc.

The filename that Jenkins 2 expects your job definitions/pipelines to be stored as is *Jenkinsfile*. You can have many Jenkinsfiles, each differentiated from the others by the project and branch it is stored with. You can have all of your code in the Jenkinsfile, or you can call out/pull in other external code via shared libraries.

The Jenkinsfile can also serve as a marker file, meaning that if Jenkins sees a Jenkinsfile as part of your project’s source code, it understands that this is a project/branch that Jenkins can run. It also understands implicitly which source control management (SCM) project and branch it needs to work with. It can then load and execute the code in the Jenkinsfile. If you are familiar with the build tool Gradle, this is similar to the idea of the *build.gradle* file used by that application.



Sample Jenkins file

**Declarative Pipelines :**

In the previous incarnations of pipelines-as-code in Jenkins, the code was primarily a Groovy script with Jenkins-specific DSL steps inserted. There was very little imposed structure, and the program flow was managed by Groovy constructs. Error reporting and checking were based on the Groovy program execution rather than what you were attempting to do with Jenkins.

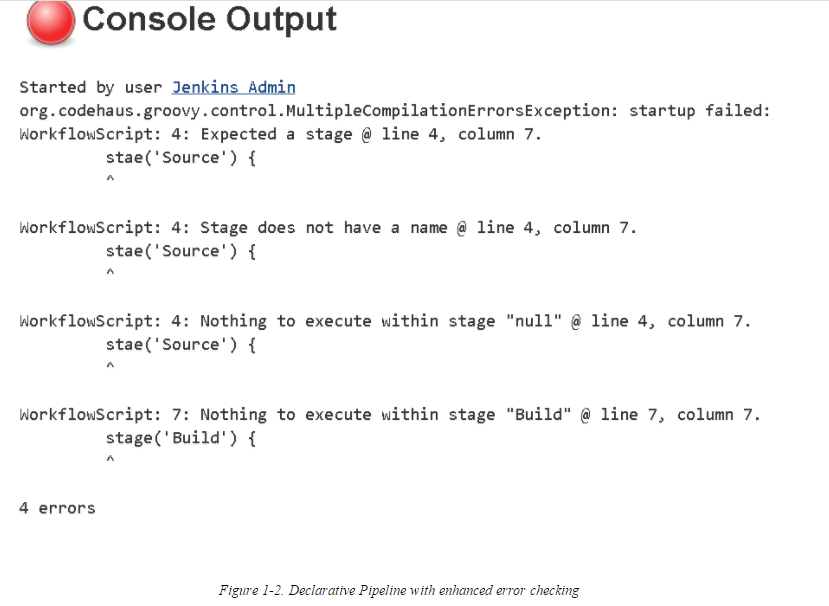
In Scripted Pipelines, the DSL supported a large number of different steps to do tasks, but was missing some of the key metafeatures of Jenkins-oriented tasks, such as post-build processing, error checking for pipeline structures, and the ability to easily send notifications based on different states. Much of this could be emulated via Groovy programming mechanisms such as try-catch-finally blocks. But that required more Groovy programming skills in addition to the Jenkins-oriented programming. The Jenkinsfile shown in [Figure 1-1](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch01.html#fig_jenkinsfile_exsc) is an example of a Scripted Pipeline with try-catch notification handling.

CloudBees, the enterprise company that is the majority contributor to the Jenkins project, introduced an enhanced programming syntax for pipelines-as-code called *Declarative Pipelines*. This syntax adds a clear, expected structure to pipelines as well as enhanced DSL elements and constructs. The result more closely resembles the workflow of constructing a pipeline in the web interface (with Freestyle projects).

An example here is post-build processing, with notifications based on build statuses, which can now be easily defined via a built-in DSL mechanism. This reduces the need to supplement a pipeline definition with Groovy code to emulate traditional features of Jenkins.

The more formal structure of Declarative Pipelines allows for cleaner error checking. So, instead of having to scan through Groovy tracebacks when an error occurs, the user is presented with a succinct, directed error message—in most cases pointing directly to the problem.





**Blue Ocean Interface**

The structure that comes with Declarative Pipelines also serves as the foundation for another innovation in Jenkins 2—Blue Ocean, the new Jenkins visual interface. Blue Ocean adds a graphical representation for each stage of a pipeline showing indicators of success/failure and progress, and allowing point-and-click access to logs for each individual piece. Blue Ocean also provides a basic visual editor.

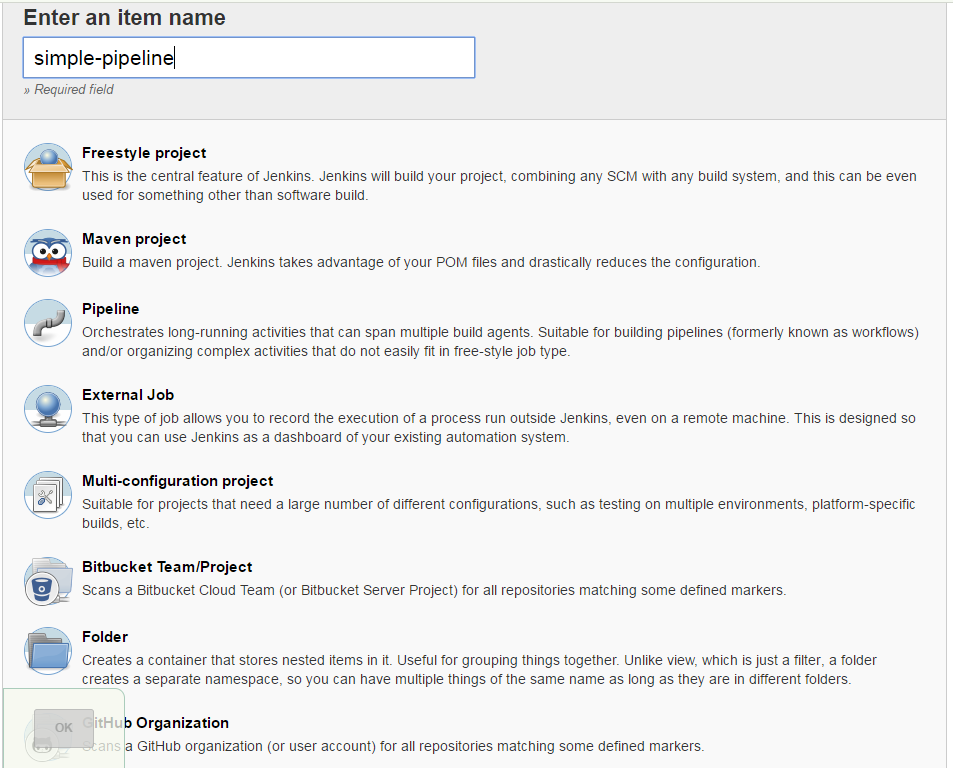


**New Job Types in Jenkins 2:**

Jenkins 2 comes with a number of new job types, mostly designed around taking advantage of key functionalities such as pipelines-as-code and Jenkinsfiles. These types make it easier than ever to automate job and pipeline creation and organize your projects. Creation of each new job/item/project starts the same way.

Once Jenkins 2 is installed and you have logged in, you can create new jobs just as before. As [below](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch01.html#Jenkins_welcome_screen_fig) fig  shows, the blurb under the “Welcome to Jenkins!” banner suggests users “create new jobs,” but the menu item for this is actually labeled “New Item.” Most of these items are ultimately a kind of project as well. For our purposes, I’ll use the terms “job,” “item,” and “project” interchangeably throughout the book.

When you choose to create a new item in Jenkins 2, you’re presented with the screen to select the type of new job You’ll notice some familiar types, such as the Freestyle project, but also some that you may not have seen before.



PIPELINE

As the name implies, the Pipeline type of project is intended for creating pipelines. This is done by writing the code in the Jenkins DSL. This is the main type of project we’ll be talking about throughout the book.

As already noted, pipelines can either be written in a “scripted” syntax style or a “declarative” syntax style. Pipelines created in this type of project can also be made easily into Jenkinsfiles.

FOLDER

This is a way to group projects together rather than a type of project itself. Note that this is not like the traditional “View” tabs on the Jenkins dashboard that allow you to filter the list of projects. Rather, it is like a directory folder in an operating system. The folder name becomes part of the path of the project.

ORGANIZATION

Certain source control platforms provide a mechanism for grouping repositories into “organizations.” Jenkins integrations allow you to store Jenkins pipeline scripts as Jenkinsfiles in the repositories within an organization and execute based on those. Currently GitHub and Bitbucket organizations are supported, with others planned for the future. For simplicity in this book, we’ll talk mainly about GitHub Organization projects as our example.

Assuming sufficient access, Jenkins can automatically set up an organization *webhook* (a notification from the website) on the hosting side that will notify your Jenkins instance when any changes are made in the repository. When Jenkins is notified, it detects the Jenkinsfile as a marker in the repository and executes the commands in the Jenkinsfile to run the pipeline.

MULTIBRANCH PIPELINE

In this type of project, Jenkins again uses the Jenkinsfile as a marker. If a new branch is created in the project with a Jenkinsfile in it, Jenkins will automatically create a new project in Jenkins just for that branch. This project can be applied to any Git or Subversion repository.

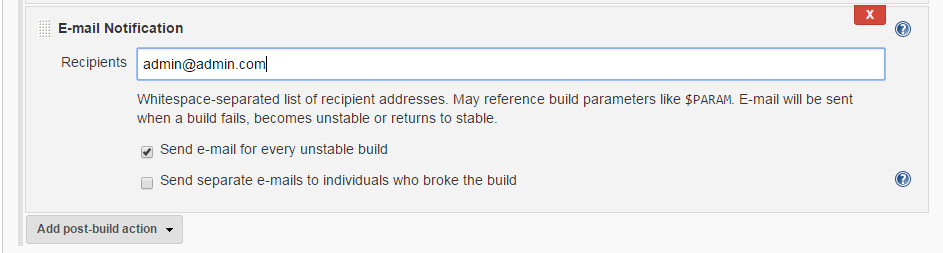
However, it is also worth noting that Jenkins still supports the traditional workhorse of jobs—Freestyle projects. You can still create jobs using web-based forms there and execute them as you have before. But certainly the emphasis in Jenkins 2 is on Pipeline jobs.

It’s easy to see that Jenkins 2 represents a major shift from the traditional Jenkins model. As such, it’s worth spending a few minutes to discuss the reasons for the change.

**Compatibility:**

As noted, Jenkins 2 now supports two styles of pipelines—scripted and declarative—each with their own syntax and structure. We will delve more into both types in the next few chapters, but for now let’s look at one specific example: post-build notification in a traditional Freestyle structure and corresponding functionality in Scripted and Declarative Pipelines.

Below shows a traditional Freestyle project’s post-build configuration for a typical operation, sending email notifications. In a Freestyle project, there’s a specific web page element for this with fields to fill in to do the configuration.



In the syntax for a Scripted Pipeline, we don’t have a built-in way to do such post-build actions. We are limited to the DSL steps plus whatever can be done with Groovy coding. So, to always send an email after a build, we need to resort to coding as shown here:

node {

**try** {

*// do some work*

        }

**catch**(e) {

            currentBuild.result = "FAILED"

**throw** e

        }

**finally** {

            mail to:"buildAdmin@mycompany.com",

            subject:"STATUS FOR PROJECT: ${currentBuild.fullDisplayName}",

            body: "RESULT: ${currentBuild.result}"

        }

}

This highlights compatibility exceptions in the case of some Jenkins functions such as post-build processing. DSL constructs can be missing for cases like this. In those instances, you may have to resort to using Groovy constructs that can mimic the processing that Jenkins would do

If you choose to use the Declarative Pipeline structure, then chances are good that you will have constructs available to handle most of the common Jenkins functions. For example, in the Declarative Pipeline syntax, there is a post section that can be defined to handle post-processing steps along the lines of the traditional post-build processing and notifications

pipeline {

    agent any

    stages {

        stage ("dowork") {

           steps {

*// do some work*

           }

        }

    }

    post {

       always {

          mail to:"buildAdmin@mycompany.com",

          subject:"STATUS FOR PROJECT: ${currentBuild.fullDisplayName}",

          body: "RESULT: ${currentBuild.result}"

       }

    }

}

Plugin Compatibility

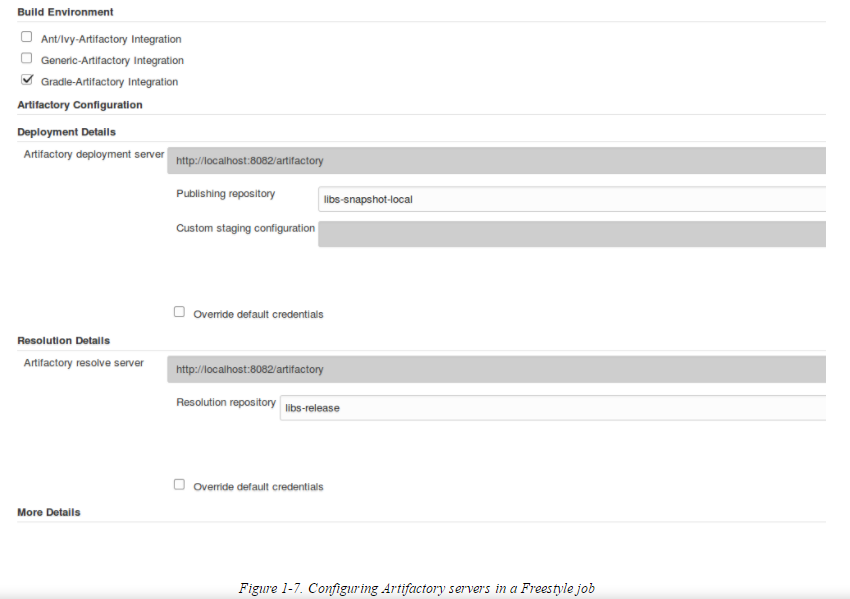
As with legacy Jenkins, the majority of functionality for Jenkins 2 is provided through integration with plugins. With the advent of Jenkins 2, new requirements were created for plugins to be compatible. We can broadly categorize the requirements into two categories: they must survive restarts and provide advanced APIs that can be used in pipeline scripts.

SURVIVING RESTARTS

One of the features/requirements of Jenkins 2 pipelines is that they must be able to survive restarts of a node. In order to support this, the main criterion is that stateful objects in plugins be *serializable*—that is, able to have their state recorded. This is not a given for many of the constructs in Java and Groovy, so plugins may have to be substantially changed to meet this requirement.

To be compatible with writing pipeline scripts, steps that were formerly done by filling in the Jenkins web forms now have to be expressible as pipeline steps with compatible Groovy syntax. In many cases, the terms and concepts may be close to what was used in the forms. Where Foo was a label for a text entry box in the form-based version of the plugin, there may now be a DSL call with Foo as a named parameter with a value passed in.

As an example, we’ll use configuration and operations for Artifactory, a binary artifact manager. Below shows how we might configure the build environment for a Freestyle Jenkins job to be able to access Artifactory repositories.



And here’s how we could do the similar configuration in a pipeline script:

*// Define new Artifactory server based on our configuration*

**def** server = Artifactory.server "LocalArtifactory"

*// Create a new Artifactory for Gradle object*

**def** artifactoryGradle = Artifactory.newGradleBuild()

artifactoryGradle.tool = "gradle4" *// Tool name from Jenkins configuration*

artifactoryGradle.deployer repo:'libs-snapshot-local', server:server

artifactoryGradle.resolver repo:'remote-repos', server:server

Beyond configuration, we have the actual operations that need to be done. In the Freestyle jobs, we have checkboxes and web forms again to tell Jenkins what to do.



And, again, in the context of a pipeline script, if the plugin is pipeline-compatible we will likely have similar DSL statements to make the API calls to provide the same functionality. The following shows a corresponding pipeline script example for the preceding Artifactory Freestyle example:

*// buildinfo configuration*

**def** buildInfo = Artifactory.newBuildInfo()

buildInfo.env.capture = **true**

*// Deploy Maven descriptors to Artifactory*

artifactoryGradle.deployer.deployMavenDescriptors = **true**

*// extra gradle configurations*

artifactoryGradle.deployer.artifactDeploymentPatterns.addExclude("\*.jar")

artifactoryGradle.usesPlugin = **false**

*// run the Gradle piece to deploy*

artifactoryGradle.run buildFile: 'build.gradle'

tasks: 'cleanartifactoryPublish'

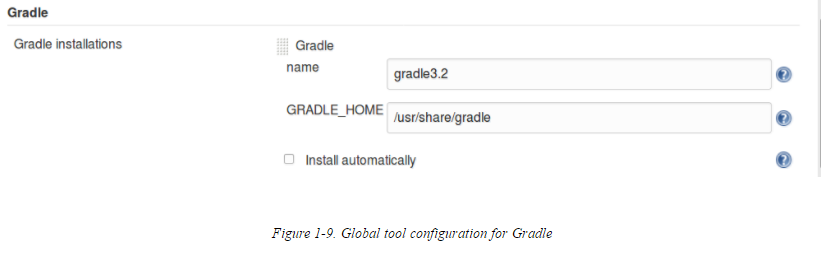
buildInfo: buildInfo

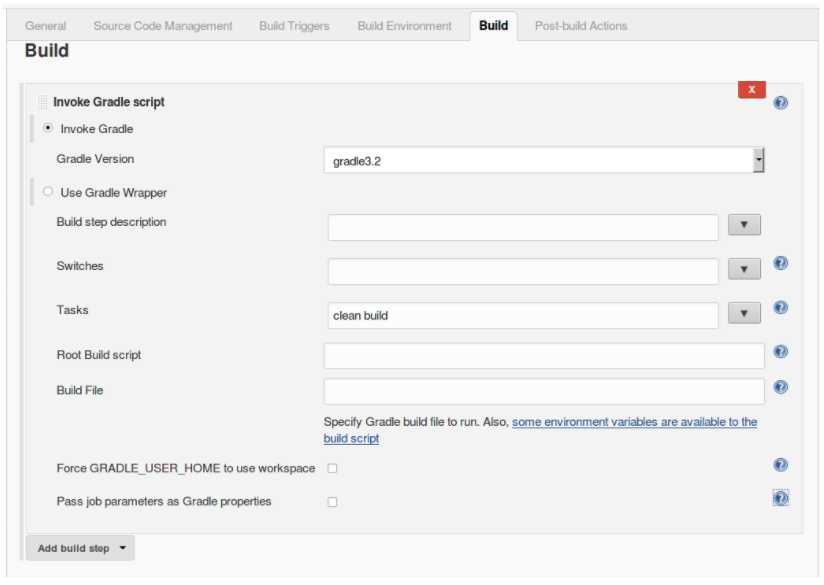
*// publish build info*

server.publishBuildInfo buildInfo

In some cases, pipeline scripts may also take advantage of items already configured in the traditional Jenkins interface, such as global tools. An example with the use of Gradle is shown nex

In the first figure ([Figure 1-9](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch01.html#fig_glob_tool_config_gradle)), we see the global tool setup for our Gradle instance. Then we see it used in a Freestyle project ([Figure 1-10](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch01.html#fig_using_global_tool_gradle_version_fsp)), and finally we see it used in a pipeline project via a special DSL step called toolthat allows us to refer back to the global configuration based on the supplied name argument.





stage('Compile') { *// Compile and do unit testing*

*// Run gradle to execute compile*

sh "${tool 'gradle3.2'}/bin/gradle clean build"

}

As we have seen, providing APIs (and thus plugin pipeline compatibility) is central to being able to execute traditional functionality in pipelines. Eventually all plugins will need to be Pipeline-compatible, but at this point, there are still plugins that are not compatible, or not completely compatible. There are places a user can go to check for compatibility, though