**CHAPTER 2 – The Foundations**

We’ll approach this task by concentrating on four basic areas:

* The two styles of syntax that can be used for creating pipelines
* The systems used to run the pipeline processes
* The basic structure of a pipeline
* The support environment (and tooling) that Jenkins provides for pipeline development and execution

We’ll start by defining and disambiguating some key concepts and terminology used with pipelines. Then we’ll survey the required DSL structures. Along the way, we’ll look at how to use the built-in editor and how to use a new tool in Jenkins to help figure out pipeline syntax

Once you know how to input your pipeline code, we’ll move on to executing a pipeline and understanding the new views that Jenkins provides. We’ll also look at how to access logs from a run. Finally, we’ll explore new functionality in Jenkins that allows us to try out changes to pipelines, without overwriting our existing versions.

Let’s get started by learning more about the different pipeline syntax styles supported in Jenkins 2.

**Syntax: Scripted Pipelines Versus Declarative Pipelines**

we now have two different styles we can use to code them: scripted syntax and declarative syntax.

Scripted syntax refers to the initial way that pipelines-as-code have been done in Jenkins. It is an imperative style, meaning it is based on defining the logic and the program flow in the pipeline script itself. It is also more dependent on the Groovy language and Groovy constructs—especially for things like error checking and dealing with exceptions

Declarative syntax is a newer option in Jenkins. Pipelines coded in the declarative style are arranged in clear sections that describe (or “declare”) the states and outcomes we want in the major areas of the pipeline, rather than focusing on the logic to accomplish it. The following code example shows a pipeline written in scripted syntax on top and a similar one written in declarative syntax underneath

*// Scripted Pipeline*

node('worker\_node1') {

stage('Source') { *// Get code*

*// get code from our Git repository*

git 'git@diyvb2:/home/git/repositories/workshop.git'

}

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

*// run Gradle to execute compile and unit testing*

sh "gradle clean compileJava test"

}

}

*// Declarative Pipeline*

pipeline {

agent {label 'worker\_node1'}

stages {

stage('Source') { *// Get code*

steps {

*// get code from our Git repository*

git 'git@diyvb2:/home/git/repositories/workshop.git'

}

}

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

steps {

*// run Gradle to execute compile and unit testing*

sh "gradle clean compileJava test"

}

}

You can think of it this way: Scripted Pipelines are more like scripts or programs written in any imperative language to execute the program flow and logic, while Declarative Pipelines are more like what was traditionally done in Jenkins if you were using the web forms—filling in key information in predefined sections that have a predefined purpose and expected behavior. Like with the traditional web forms, when you run a Declarative Pipeline the type of each section defines what happens and how, based on the data you entered.

## Choosing Between Scripted and Declarative Syntax

So what are the factors that come into play in choosing between scripted and declarative? As with most things, it’s not an exact science; in any particular situation, one model may work better than the other based on the need, the structures and flows to be implemented, and the skill and background of the person(s) implementing the pipeline.

We can best derive guidance here by looking at the advantages and disadvantages of each model and then making some general observations.

Briefly, a Scripted Pipeline has the following advantages:

* Generally fewer sections and less specification needed
* Capability to use more procedural code
* More like creating a program
* Traditional pipeline-as-code model, so more familiar and backward compatible
* More flexibility to do custom operations if needed
* Able to model more complex workflows and pipelines

A Scripted Pipeline has the following disadvantages:

* More programming required in general
* Syntax checking limited to the Groovy language and environment
* Further away from the traditional Jenkins model
* Potentially more complex for the same workflow if it can be comparably done in a Declarative Pipeline

A Declarative Pipeline has the following advantages:

* More structured—closer to the traditional sections of Jenkins web forms
* More capability to declare what is needed, so arguably more readable
* Can be generated through the Blue Ocean graphical interface
* Contains sections that map to familiar Jenkins concepts, such as notifications
* Better syntax checking and error identification
* Increased consistency across pipelines

A Declarative Pipeline has the following disadvantages:

* Less support for iterative logic (less like a program)
* Still evolving (may not support or have constructs for things you would do in traditional Jenkins)
* More rigid structure (harder to handle custom pipeline code)
* Currently not well suited for more complex pipelines or workflows

In short, the declarative model should be easier to learn and maintain for new pipeline users or those wanting more ready-made functionality like the traditional Jenkins model. This comes at the price of less flexibility to do anything not supported by the structure.

The scripted model offers more flexibility. It provides the “power-user” option, allowing users to do more things with less imposed structure.

# Systems: Masters, Nodes, Agents, and Executors

Regardless of whether we are using scripted or declarative syntax, every Jenkins pipeline has to have one or more systems to execute code on. The term *system* is used here as a generic way to describe all of the items we’re talking about. Keep in mind, though, that there can be multiple instances of Jenkins on any given system or machine.

In traditional Jenkins, there were only two categories: masters and slaves. Those are probably familiar to you. Here’s a brief description of similar terms, highlighting some of the main points for comparison.

Master

A Jenkins *master* is the primary controlling system for a Jenkins instance. It has complete access to all Jenkins configuration and options and the full list of jobs. It is the default location for executing jobs if another system is not specified.

However, it is not intended for running any heavyweight tasks. Jobs requiring any substantial processing should be run on a system other than the master.

Another reason for this is that a job running on the master has the master’s access to all data, configuration, and operations, which can pose a security risk. It is also important to note that a master system should not have potentially blocking operations executed on it, since it needs to be able to respond and manage operations continuously.

Node

*Node* is the generic term that is used in Jenkins 2 to mean any system that can run Jenkins jobs. This covers both masters and agents, and is sometimes used in place of those terms. Furthermore, a node might be a container, such as one for Docker.

A master node is always present in any Jenkins installation, but for the reasons already cited, it is not recommended to run jobs on the master node. We’ll talk more about how to define nodes in an upcoming section of this chapter.

Agent

An *agent* is the same as what earlier versions of Jenkins referred to as a *slave*. Traditionally in Jenkins, this refers to any nonmaster system. The idea is that these systems are managed by the master system and allocated as needed, or as specified, to handle processing the individual jobs. For example, we might allocate different agents to do different builds for different OS flavors, or we might allocate multiple agents to run in parallel for testing.

In order to simplify the load on these systems and reduce security concerns, typically only a lightweight Jenkins client application with limited access to resources is installed to handle running jobs.

As far as the relationship between agents and nodes goes, agents run on nodes. In a Scripted Pipeline, “node” is used as the term for a system with an agent. In a Declarative Pipeline, specifying a particular agent to use allocates a node.

##### DIRECTIVES VERSUS STEPS

There is a high-level distinction we can make between a node and an agent in terms of how they are used in the respective declarative versus scripted syntax.

node is associated with a Scripted Pipeline. It is technically a step, meaning something that can be used to cause an action to occur in a pipeline. It allocates an executor on a node with an agent and further runs code that is in its definition block. The following code excerpt shows a simple example of specifying a node step:

*// Scripted Pipeline*

node('worker') {

    stage('Source') { *// Get code*

*// Get code from our Git repository*

agent, on the other hand, is a directive in a Declarative Pipeline. Unless you use the special case agent none, it causes a node to be allocated. A simple agent declaration is shown here:

*// Declarative Pipeline*

pipeline {

    agent {label:'worker'}

    stages {

        stage('Source') { *// Get code*

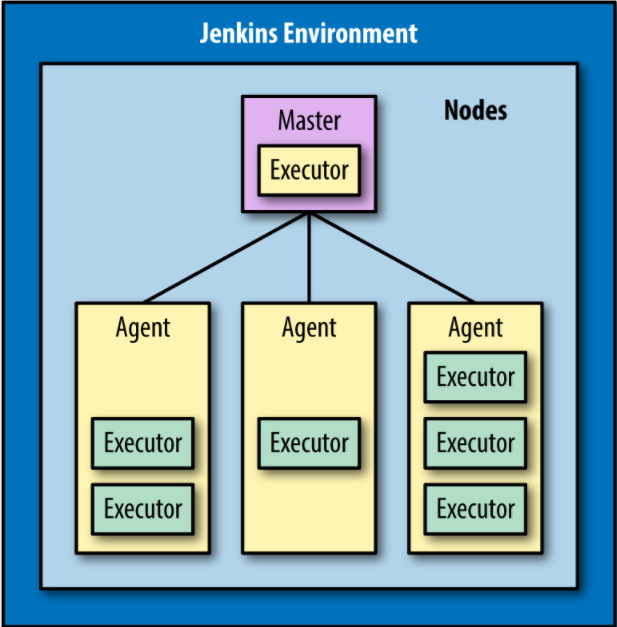
Outside of the syntax for the two different pipeline specifications, this distinction is not significant and you can think of them as the same. Just use node for Scripted Pipelines and agent for Declarative Pipelines.

## Executor

*Related to all the previous systems are executors. Let’s clarify here what Jenkins means with this term.*

*Basically, an executor is just a slot in which to run a job on a node/agent. A node can have zero or more executors. The number of executors defines how many concurrent jobs can be run on that node. When the master funnels jobs to a particular node, there must be an available executor slot in order for the job to be processed immediately. Otherwise, it will wait until an executor becomes available.*

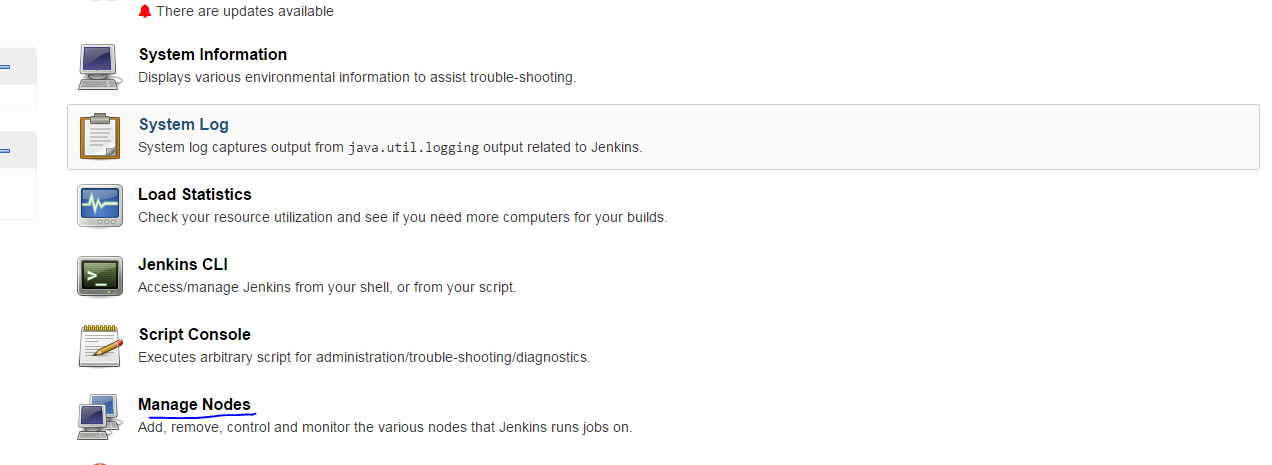
*The number of executors and other parameters can be configured when creating nodes, the subject of our next section*.



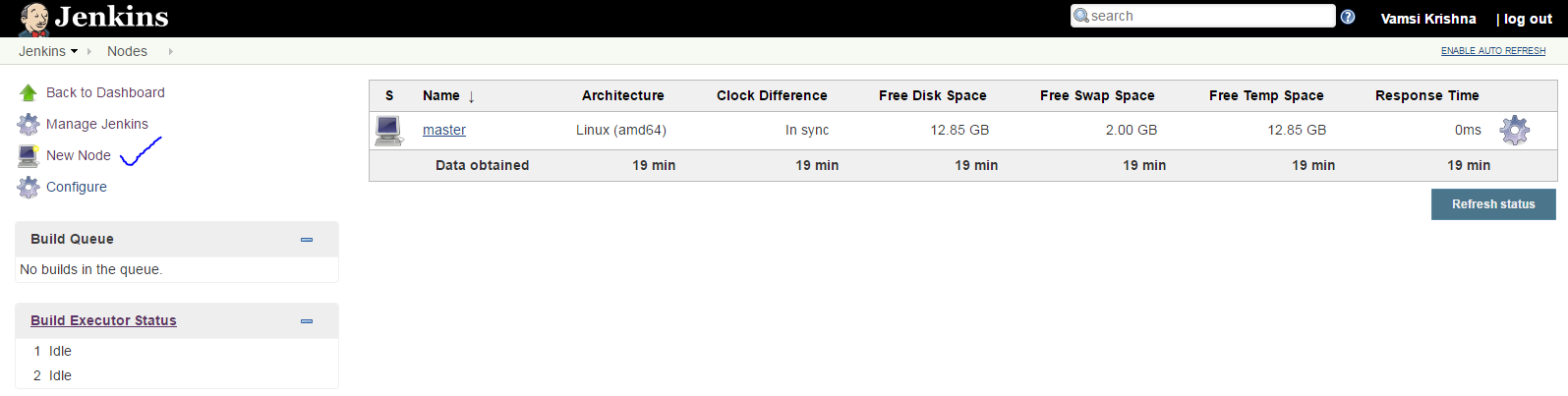
## Creating Nodes

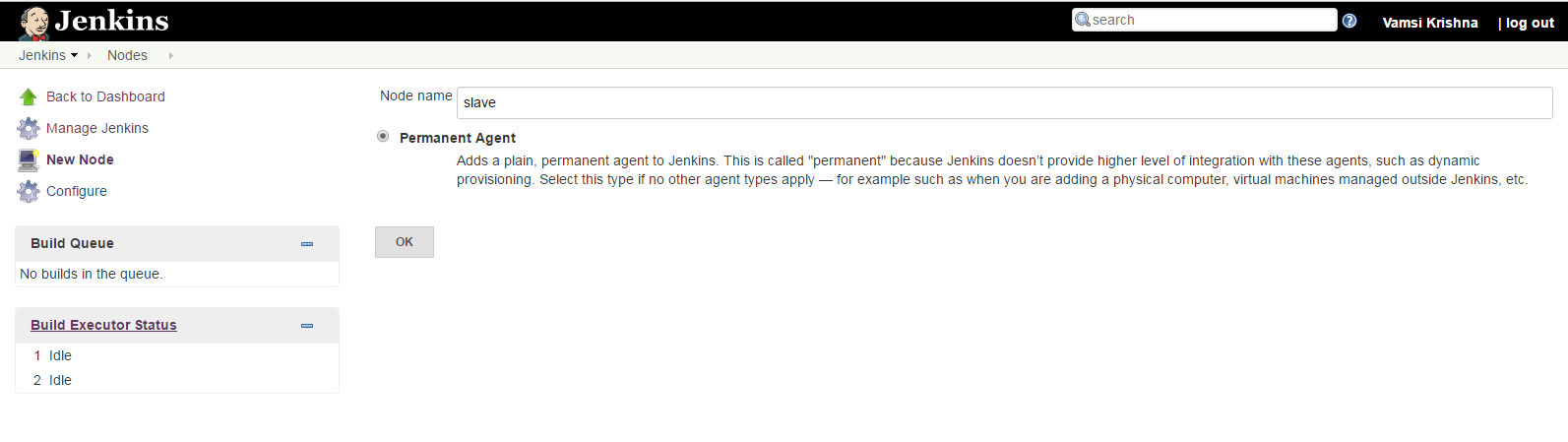
*In traditional versions of Jenkins, jobs would run either on the master instance or on slave instances. As noted previously, in Jenkins 2 terminology these kinds of instances are both referred to by the generic term “node.” We can set up new nodes just as we would have set up slaves on legacy Jenkins instances. A quick example follows*

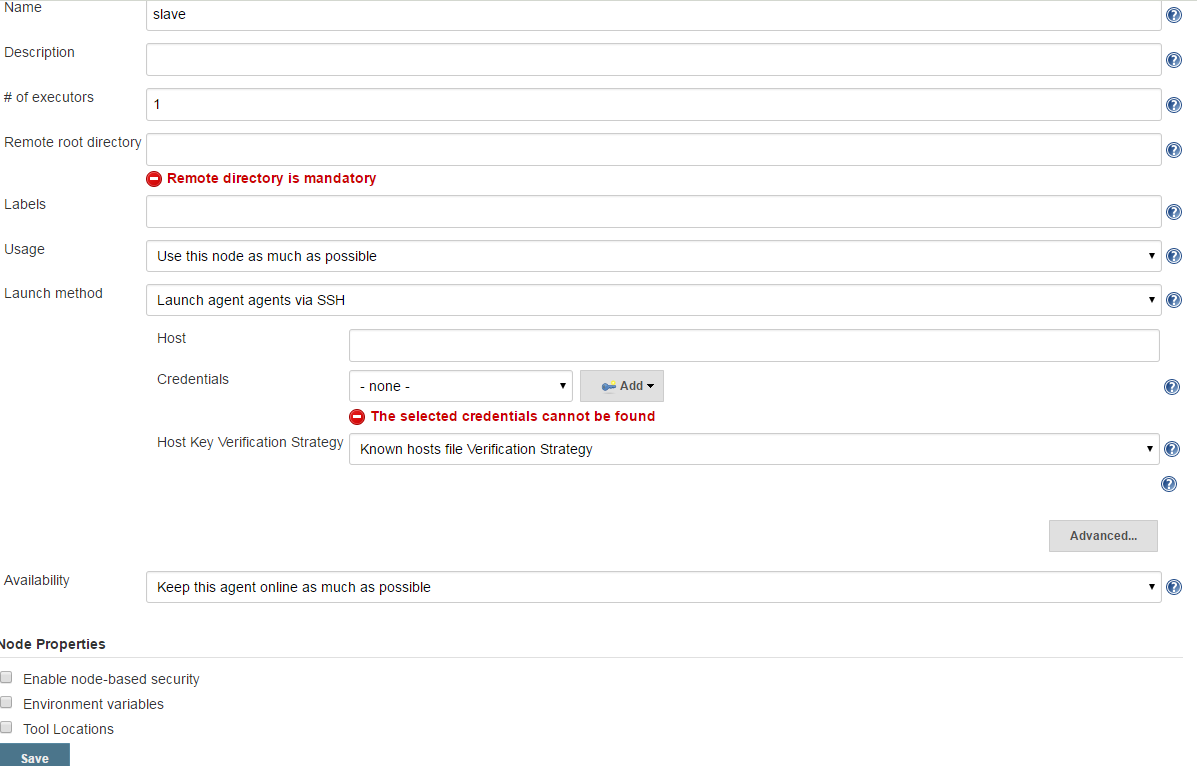
*To start with, after logging into Jenkins, go to the Manage Jenkins page and select the Manage Nodes link*



*On the Manage Nodes screen, select New Node and fill in the forms, including the number of executors*







*If you need to first set up credentials, you can find more information about that in later tutorials. Notice that you also have checkboxes near the bottom of the page for “Environment variables” and “Tool Locations.” Checking these will allow you to specify particular variables and tools for use on this node. This is only necessary if you need or want to use ones other than those set up on the master.*

*In the Labels section of the configuration, you can supply multiple labels. Spaces can be included in a label name with quotes around the label*.

### A QUICK NOTE ABOUT NODE LABELS

*Labels can be used for both system and user purposes. For example, labels can be used to:*

*Identify a specific node (via a unique label).*

*Group classes of nodes together (by giving them the same label).*

*Identify some characteristic of a node that is useful to know for processing (via a meaningful label, such as “Windows” or “West Coast”).*

*The last bullet is a recommended practice.*

*These labels can be referenced directly in the pipeline to define where to run code. An example is discussed in*[*“node”*](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch02.html#node_section)*.*

# Structure: Working with the Jenkins DSL

Here’s a very simple pipeline expressed in the Jenkins DSL

node ('worker1') {

     stage('Source') { *// for display purposes*

*// Get some code from our Git repository*

      git 'https://github.com/brentlaster/gradle-greetings.git'

      }

}

*Let’s break this down and explain what each part is doing*

## node

*First, we have the keyword node. As mentioned in*[*“Node”*](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch02.html#node_firsttime_section)*, we can think of this as the new term for a master or agent. Nodes are defined through the Manage Jenkins → Manage Nodes interface and can be set up just like slaves. Each node then has a Jenkins agent installed on it to execute jobs. (Note that in this case we are assuming we have a node already set up on the Jenkins instance labeled worker1.)*

# NODES AND AGENTS

*We previously talked about the difference between nodes and agents in Jenkins terminology. In the context here, we’re using*agent*to mean the Jenkins code running on the “nonmaster” nodes*.

*This line tells Jenkins on which node it should run this part of the pipeline. It binds the code to the particular Jenkins agent program running on that node. A particular one is specified by passing a defined name as a parameter (label). This must be a node or system that has already been defined and that your Jenkins system is aware of. You can omit supplying a label here, but if you omit a label, then you need to be aware of how this will be handled:*

* If master has been configured as the default node for execution, Jenkins will run the job on master. (master can be configured to not run any jobs.)
* Otherwise, an empty node label (or agent any in declarative syntax) will tell Jenkins to run on the first executor that becomes available on any node.

*On the other hand, using multiple names here (with logic operators) is perfectly valid and can make a lot of sense when you need to select nodes based on multiple dimensions (such as location, type, etc.). The following sidebar explains how to take advantage of this functionality.*

##### LEVERAGING MULTIPLE LABELS ON A NODE

*In the configuration for a node, you can assign multiple labels in the Labels entry box. To do this, separate them by spaces. Then, when specifying a node to execute part of your pipeline, you can specify multiple labels using standard logic operands such as || for “or” and && for “and.”*

*Why would you do this? Suppose that you had two sets of Linux systems on different coasts of the United States. Depending on the particular processing, you might want some Jenkins jobs sent to one set, and some sent to the other set.*

*So, in this case, you could add the label Linux to all of the nodes and an additional label to indicate where each is located—i.e., east or west. Once that’s done, you could specify which nodes to use by using combinations of operands and labels. For example, to direct a job to run on a Linux node on the East Coast, you could use:*

node("linux && east") {

*There are more sophisticated operands available as well, which you’ll find if you look in the help for the node step.*



## stage

*Within a node definition, a stage closure allows us to group together individual settings, DSL commands, and logic. A stage is required to have a name, which provides a mechanism for describing what the stage does. As of the time of this writing, it doesn’t actually do anything in the script but does show up in the output to identify the stage when running a pipeline.*

*How much of the pipeline’s logic goes into a particular stage is up to the developer. However, a general practice is to create stages that mimic the separate pieces of a traditional pipeline. For example, you might have a stage that handles retrieving the source code, one that handles compiling the source code, one that handles running unit tests, one that handles integration tests, and so on. We’ll use this sort of structure when we work with example pipelines in the book.*

## steps

*Inside the stage, we have the actual Jenkins DSL commands. These are referred to as*steps*in Jenkins terminology. A step is the lowest level of functionality defined by the DSL. These are not Groovy commands, but can be used with Groovy commands. In the case of our example, we have this initial step to get our source:*

git 'https://github.com/brentlaster/gradle-greetings.git'

*This is pretty straightforward to figure out. It calls Git and passes a parameter—the location from which to pull the code (using the secure HTTP protocol). This is using a shorthand format for the full step syntax*

*You will be encountering both the shorthand and full step syntax when working with the DSL in scripts, so it’s worth taking a moment to better understand the syntax model in more detail.*

### UNDERSTANDING STEP SYNTAX

*Steps in the Jenkins DSL always expect mapped (named) parameters. To illustrate this, here’s another version of the git step definition*

git branch: 'test',

url: 'https://github.com/brentlaster/gradle-greetings.git'

*Notice that we have two named parameters here, mapped to their intended* values: branch to 'test' and url to 'http://github.com/brentlaster/gradle-greetings.git'.

This syntax itself is actually a shorthand notation for a mapping syntax used by Groovy. The [named parameter: value, named parameter: value] form equates to the Groovy mapping syntax of [key: value, key: value]. The named parameters function as the keys of the map.

Groovy also allows skipping the parentheses for parameters. Without these shortcuts, the longer version of our step would be:

git([branch: 'test',

url: 'http://github.com/brentlaster/gradle-greetings.git'])

Another trick is this: if there is a single required parameter, and only one value is passed, the parameter name can be omitted. This is how we arrive at our short version of the step as:

git 'https://github.com/brentlaster/gradle-greetings.git'

The required url parameter here is the only one we needed to provide in this case

