**Chapter 14. Integrating Containers**

Docker forms a key component in many pipelines these days. The ease, flexibility, and isolation provided by containers allows us to create custom, specific environments for processing with exact repeatability. In this chapter, we’ll look at the different ways that Docker can be used with Jenkins 2.

For Jenkins 2, there are essentially four options for incorporating Docker into your pipeline:

* Configured as a “cloud,” as a standalone Jenkins agent
* As an agent created on the fly for a Declarative Pipeline
* Via the special DSL docker global variable and its associated methods
* Directly in the script via the DSL shell call (sh)

Let’s take a closer look at each of these.

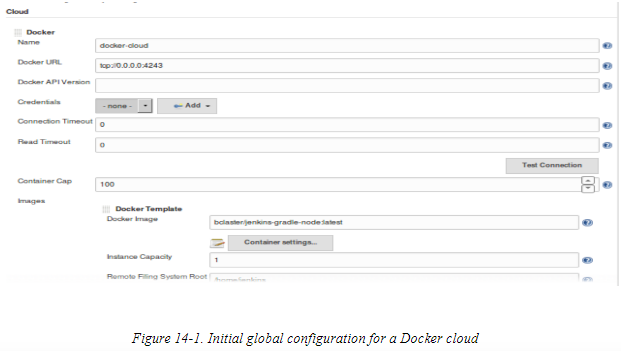
# Configured as a Cloud

The idea here is that you are defining one or more Docker images that Jenkins can use as agents. This is the “cloud” environment from which to start up agents. When your pipeline runs, it can reference the cloud setup and start up instances of the images as agents. The agents can then be used to run the various stages and steps. After the pipeline is done, Jenkins will stop and remove the containers running those images, thus removing the agents.

In order for this option to be available, the [Docker plugin](http://bit.ly/2J7OLR8) has to be installed. (Note that this is different from the Docker Pipeline plugin that we will talk about later in this chapter.) The other requirement is that any Docker image you supply here has to be able to function as a “standalone agent”—meaning it is set up like a node. We’ll talk more about the requirements for that in a moment. But first, as with all major functionality in Jenkins, we have some global configuration to do.

## Global Configuration

When you install the Docker plugin (or other cloud plugins, such as Amazon EC2), a new Cloud section is added to the Configure System screen. After you click the “Add a new cloud” button, you are given the option to select Docker. Then a new configuration section is presented. [Figure 14-1](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch14.html#fig_init_glob_conf_docker_cloud)shows an example of this section with some completed fields.



Let’s look at some of these fields in more detail. The Name field is simply a name to refer to this cloud. The Docker URL field refers to a way to access the Docker Remote API. By default, this is probably not enabled, and you will need to enable it so that Jenkins can access it.

There is a lot of information on the web on getting the remote API to work with Docker in Jenkins—much of it confusing. In the simple case, which hopefully will work for most readers, here’s what you need to do:

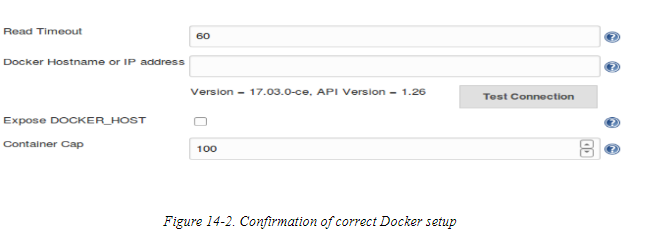
1. Look at the arguments you would supply for the -H option for Docker (the “host list” option). Most commonly, these would be of the form tcp://<ip-addr>:<docker-port> and unix:///var/run/docker.sock.
2. Add these arguments into your Docker startup file. If you’re running on a Linux system, your first thought might be to add these in */etc/init/docker.conf*—but when you look for the startup options, in that file, you’ll typically see a line of the form # modify these in /etc/default/$UPSTART\_JOB (/etc/default/docker).
3. Assuming that last statement is true, add a line in */etc/default/docker* like the following one (here, for the sake of simplicity, we are running Docker on our local system and so can use the 0.0.0.0 IP address: if that is not the case, you would use the IP address of the remote system where you host Docker):
4. DOCKER\_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/docker.sock'
5. After updating the file, you’ll need to restart the Docker service and, depending on your system, possibly reload the daemon.

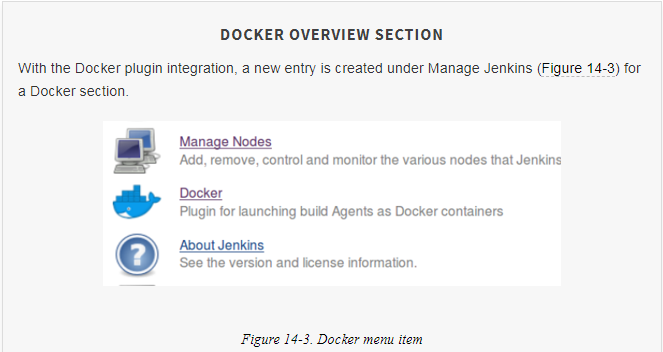
With the Remote API enabled, you are ready to configure the connection to it in the global Jenkins Docker cloud configuration. For this, you’ll want to fill in the Docker URL field with the same tcp... value you supplied to Docker; i.e., tcp://#.#.#.#:4243. Optionally, you may be able to use the unix:///var/run/docker.sock setting.

There are some related fields under the URL field. For the Docker API Version, you only need to supply a value if you want a version other than the default one. Supply a set of credentials in the Credentials field if needed, and, optionally, provide values for the read and connection timeouts.

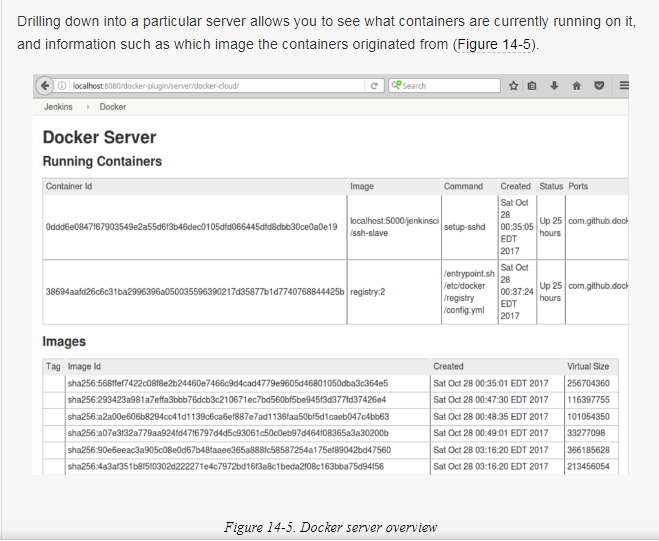
The Container Cap field is there if you want to limit the number of containers the Docker system can run. Note that this also includes containers not started by Jenkins. It has a default of 100.

With your API connection set up, it is advisable to test the connection by clicking the Test Connection button. If everything is working, you should see text with the Docker version and API version displayed inline

After you have the basic Docker configuration working, you’re ready to specify images that the cloud can use to run as agents. This is done by clicking the Add Docker Template button and selecting Docker Template. We discuss more setup details for this in the next section.







## Using Docker Images as Agents

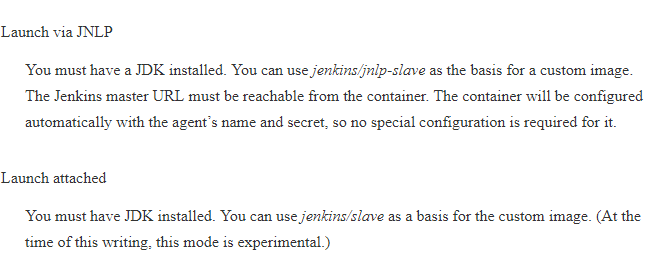
The basic requirement for a Docker image to be used as an agent is that it needs to be able to run like a standalone agent. Typically this will mean that it has basic applications installed on it, such as Java and SSH. As discussed on the Docker plugin page, depending on how the agent is to be launched, there are different base images that may be appropriate.

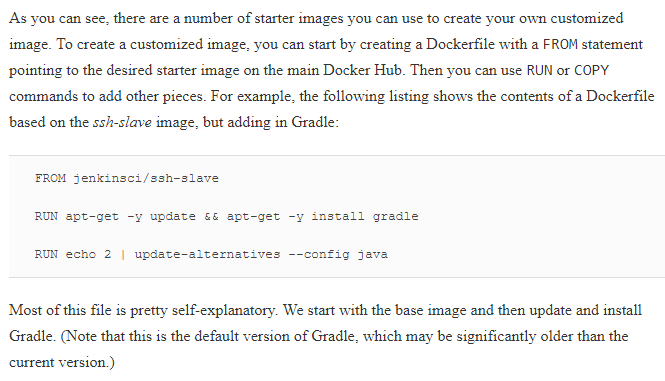
The Jenkins wiki [outlines the prerequisites for the Docker image](http://bit.ly/2qLYJA6) to be used as follows, depending on the launch method selected:

Launch via SSH

You must have an *sshd* server and a JDK installed. You can use *jenkins/ssh-slave* as the basis for a custom image. An SSH key based on the unique Jenkins master instance’s identity can be injected in the container on startup, so you don’t need a credential set as long as you use the standard *openssl sshd*.

For backward compatibility or if you have a nonstandard *sshd* packaged in your Docker image, you also have the option to provide manually configured SSH credentials.





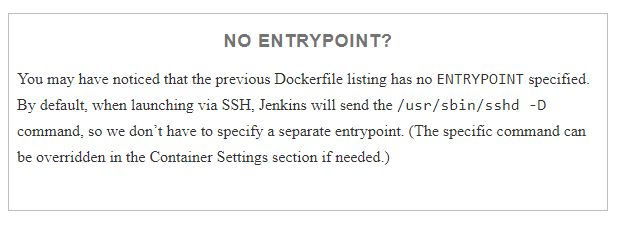
However, the last line deserves a bit of explanation. For this type of SSH-launched node, the connection to the Docker container relies on the SSH Slaves plugin. One of the things this plugin does is connect to the container and check the version of Java on it to make sure it is compatible with the Jenkins JARs that need to be used. If it can’t find a compatible version, it attempts to install one.

The base image has a number of Java versions installed. Unfortunately, the default is an old level for most versions of Jenkins (as of this writing). So, when Jenkins checks, it detects the older version and tries to install a new version from Oracle. Unfortunately, the Oracle installation wants a username and password (which aren’t available), so the startup fails.

The base image contains a newer JDK that is compatible. The last line in our file selects that version. Granted, this isn’t particularly elegant, and there are other ways to handle this, but it serves for our example purposes.

The idea here is that we are creating images for the cloud to instantiate as agents for running jobs. If you are using a modified image (with your own Dockerfile), then you would build the image and push it out to a Docker registry that you have access to. For the examples here, we’ll assume our images are in the public Docker registry.

The next step in this process is defining the “template” for the cloud to be able to use our configuration.



### SETTING UP A DOCKER CLOUD TEMPLATE—BASIC OPTIONS

With Jenkins and Docker configured to talk to each other via Docker’s REST API, and the image(s) set up for our cloud, we can move on. We next need to define the section of the global configuration for the cloud that tells it what image to use, and provides any needed options and an access method. This configuration is done by adding a template.

In the Cloud section of the Configure System screen, click the Add Docker Template button, then click on the “Docker Template” pop-up. Next, you’re presented with a set of options to fill in for the template. We’ll cover what’s needed for our SSH image example. Setup for other types can be interpolated from this one.

The first field to fill in is the Docker Image field. This should be the image that you want to have spun up as the agent. If you have created a custom Docker image and pushed it to the Docker Hub, you would enter the name of that image.

We’ll come back to the “Container settings...” in a moment.

For pipeline usage, you need to put some text in the Labels field. This text will be what you include in the pipeline’s agent definition. This will allow your pipeline to select a container based on the image defined in this template section. For example, if you put “docker-cloud-gradle” in the Labels field, you could use this label to select a Docker agent created from the image (assuming declarative syntax):

pipeline {

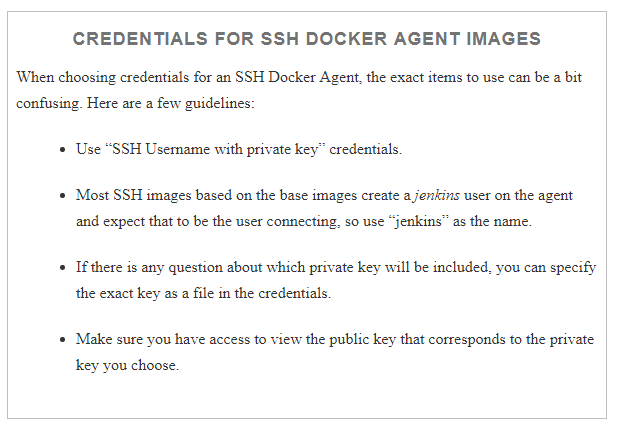
    agent {

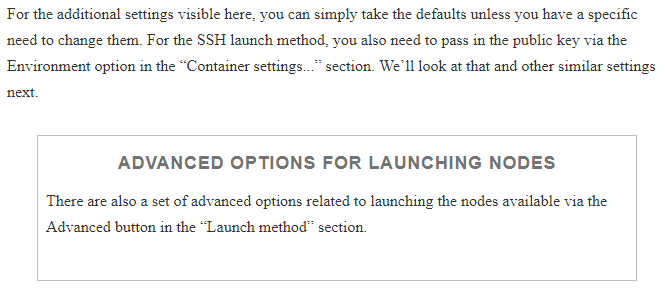
label 'docker-cloud-gradle'

    }

    stages {

Next, you’ll want to make sure that the launch method is set (only the SSH option is production-ready at the time of this writing) and that the appropriate credentials are selected and in place. The credentials should be an “SSH Username with private key” as explained in the following note. We’ll discuss where the public key goes in a bit.





### CONTAINER SETTINGS

Near the top of the template section is a “Container settings...” button. Clicking on that brings up additional fields for container-specific options. Here are details on a few of the common ones:

Docker Command

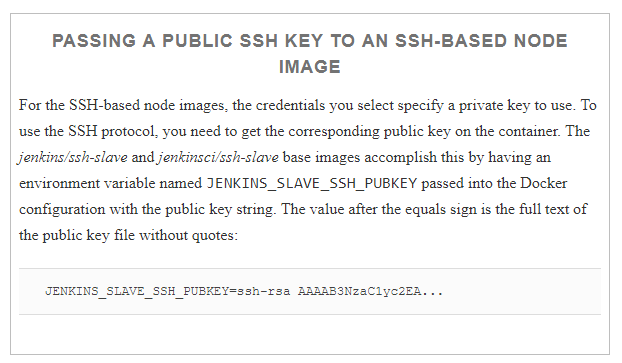
This is the command to have Jenkins run on the image. Typically, you would just leave this as the default that starts the SSH daemon (/usr/sbin/sshd -D).

Volumes

A list of volume mounts, such as /host/path:/container/path:mode. If multiple entries are listed, they should be separated by newlines. The idea of /host/path:/container/path:mode is that this will mount the path on the host to a path in the container with the specified mode—either ro for read-only or rw for read-write. The mode is optional and defaults to read-write.

Environment

Environment variable values to pass into the container. For an example, see the following note.



Port Bindings

The specifications of the form <host-port>:<container-port> to bind a port between the host and container. This is the same as the -p option on the Docker command line.

Instance Capacity

The maximum number of instances to run of this image. Note that if this is not set, the default is unlimited. It’s important to set this to a low value (unless you have a good reason to do otherwise) to prevent having a large number of instances running if something doesn’t go right.

With the cloud configured and templates defined, we are ready to move on to using the images in our pipeline

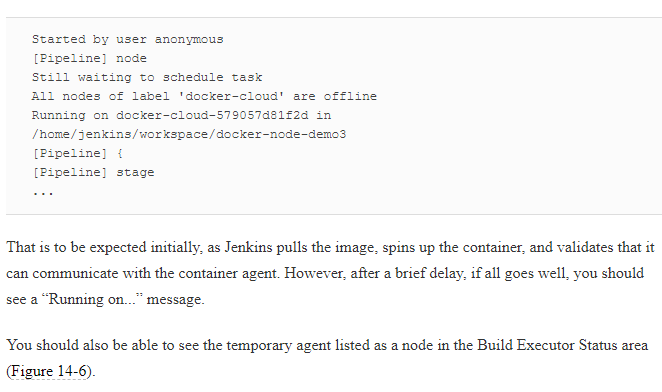
## Using Cloud Images in a Pipeline

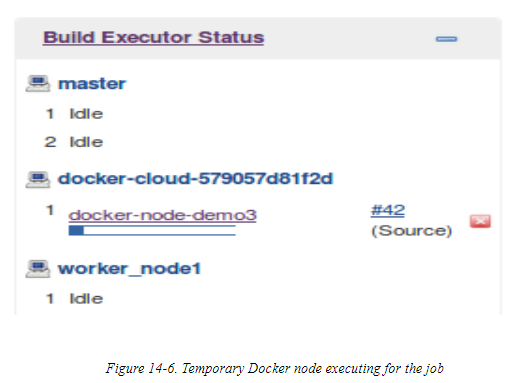
The following listing shows a simple pipeline script in declarative syntax that makes use of the cloud we’ve defined so far:



Note again the use of the label that we set in the template area to select the image and options associated with that template. In this case, we are executing both stages on the Docker node, but you could also use agent directives within the individual stages if desired.

Once you start the build of the pipeline, if you look at the console output of the job, you will probably see a message either that indicates that the node is offline or that all nodes corresponding to the label you supplied are offline:

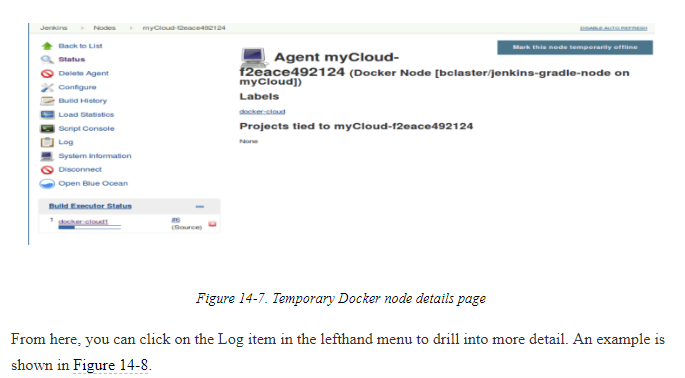


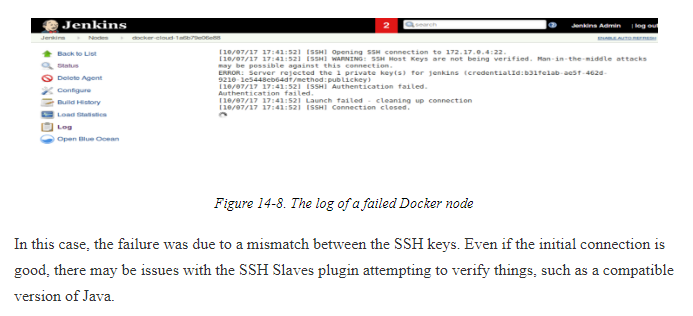


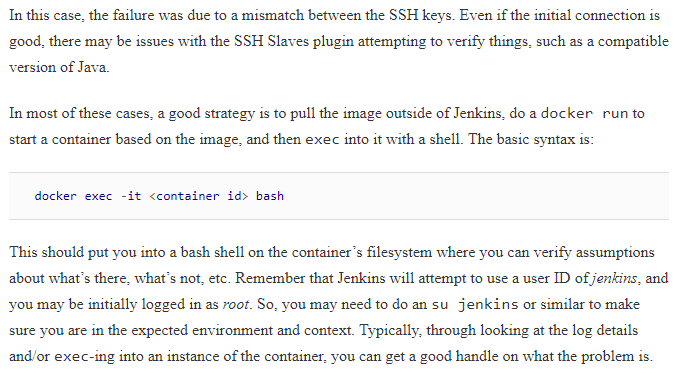
If your Docker container has the environment and tooling set up as needed for the stages of your pipeline, the pipeline should run to completion. At that point, Jenkins will remove the agent/node and its corresponding running Docker container.

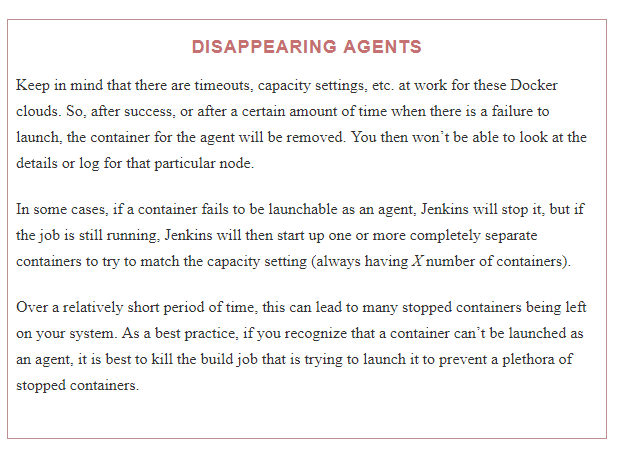
### TROUBLESHOOTING

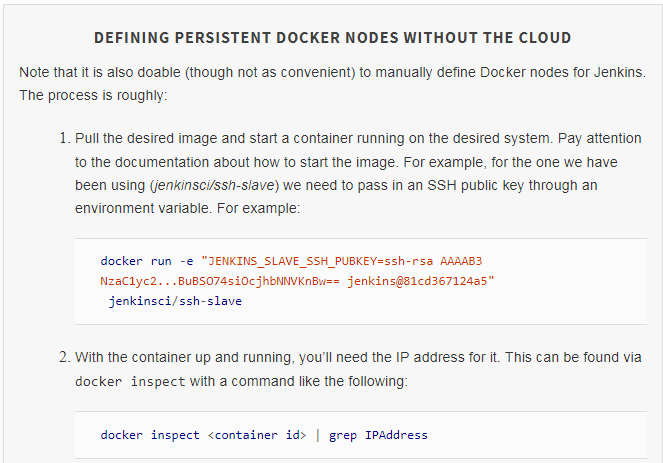
If you do not get the “Running on...” message in the console output and/or you see an indication in the Build Executor Status area that the node is still offline, Jenkins may be having trouble starting up, or communicating with, the Docker agent. If this is the case, you can click on the node and go to the corresponding node details page for more information ([Figure 14-7](https://www.safaribooksonline.com/library/view/jenkins-2-up/9781491979587/ch14.html#fig_temp_dock_node_det_page)). (You can also get there through the Manage Nodes menu item under Manage Jenkins.)

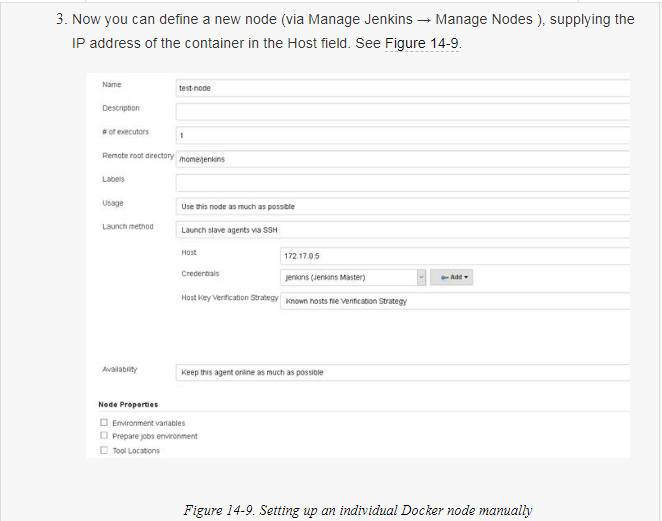












# Agent Created on the Fly for a Declarative Pipeline

The Declarative Pipeline syntax includes special functionality for creating agents dynamically at the time they are needed. This is done by pointing the agent directive to a Dockerfile, from which it can run a container that uses a Docker image, set up to function as an agent. Most of these are just variations of the syntax for declaring an agent, as described here:

agent { docker '<image>' }

This short syntax tells Jenkins to pull the given image from Docker Hub and run the pipeline or stage in a container based on the image, on a dynamically provisioned node.

agent docker { <elements> }

This longer syntax allows for defining more specifics about the Docker agent. Three additional elements can be in the declaration (within {} block):

image '<image>'

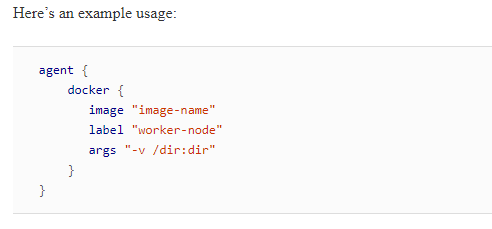
Tells Jenkins to pull the given image and use it to run the pipeline code.

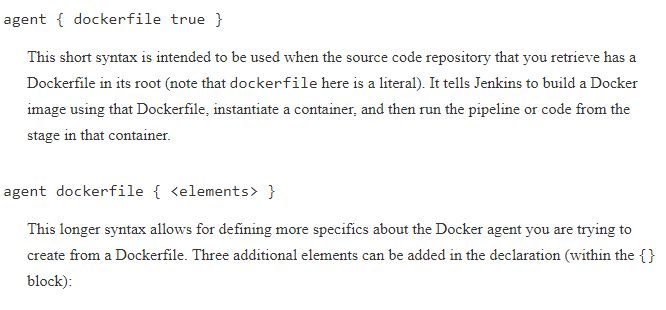
label '<label>'

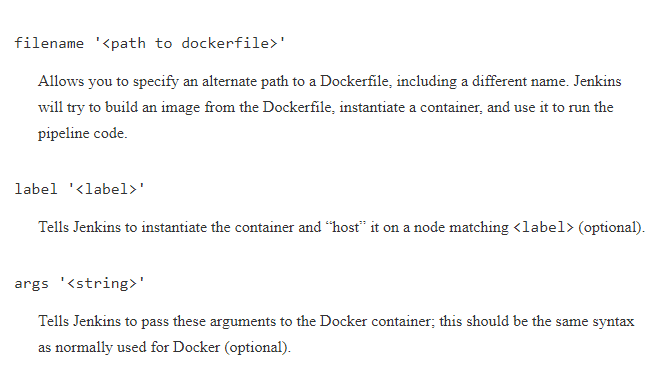
Tells Jenkins to instantiate the container and “host” it on a node matching <label> (optional).

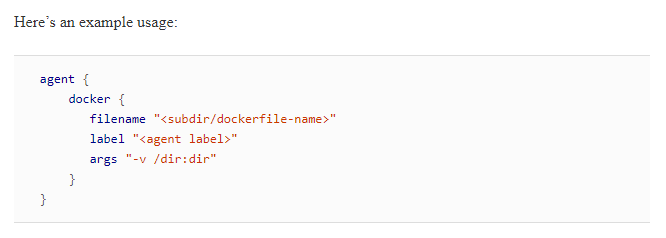
args '<string>'

Tells Jenkins to pass these arguments to the Docker container; uses the same Docker syntax as you would normally use (optional).









reuseNode

This tells Jenkins to reuse the same node and workspace that were defined for the original pipeline agent to “host” the resulting Docker container.

This last one requires a bit of explanation. Remember that even though we are running a Docker container for our agent in these cases, we still have to have a system where Docker is actually hosted and running. That’s what the label argument in these calls is specifying: which system is hosting Docker.

If we start our pipeline running on a particular node, then it may do operations that leave code or other input on the node (such as cloning source out of source control). If we later want to use a Docker container to do something in the pipeline (such as build the source), then it makes things simpler if we can just run/host the Docker container on the same underlying node. Since the code is already there and the Docker commands can mount the workspace as a path inside them, that simplifies this kind of setup. That’s what the reuseNode option is for—running an upcoming Docker container on the same node that we started with.

