Dashboards are useful when you want to get an overview of your Jenkins logs and make correlations among various visualizations and logs; see figure 13.27.

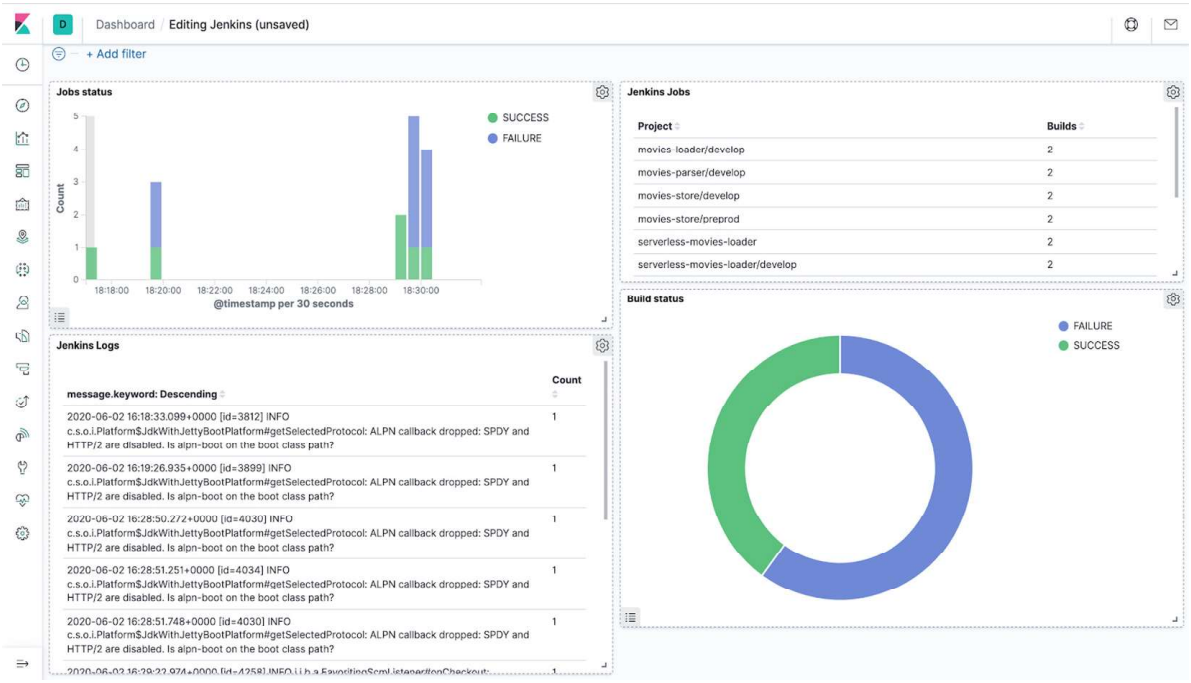

Figure 13.27 Analyzing Jenkins logs from a Kibana dashboard

The complete dashboard can be imported from the following JSON file: chapter13/kibana/dashboard/jenkins.json.

That's it! You've successfully created a pipeline that uses Filebeat to take Jenkins logs as input, forwards those logs to Logstash for parsing, and writes the parsed data to an Elasticsearch server.

13.2.2 Streaming logs with the Logstash plugin

You can skip the Filebeat and Logstash configurations by shipping Jenkins logs directly to an Elasticsearch instance via the Logstash plugin (https://plugins.jenkins.io/logstash/) on Jenkins. This solution is ideal if you're not already using external Logstash agents to stream your infrastructure or application logs to Elasticsearch, and if you don't need to enrich the parsing mechanism of logs with custom Grok expressions. Plus, the Logstash plugin can stream the log data from a Jenkins instance to any indexer solution (including Redis, RabbitMQ, and Elasticsearch). In the current scenario, we will use Elasticsearch.

After successfully installing the Logstash plugin in the global configuration of the Jenkins dashboard, we need to configure the plugin with the target indexer. Configure the URI, where the Elasticsearch server is running, as shown in figure 13.28.

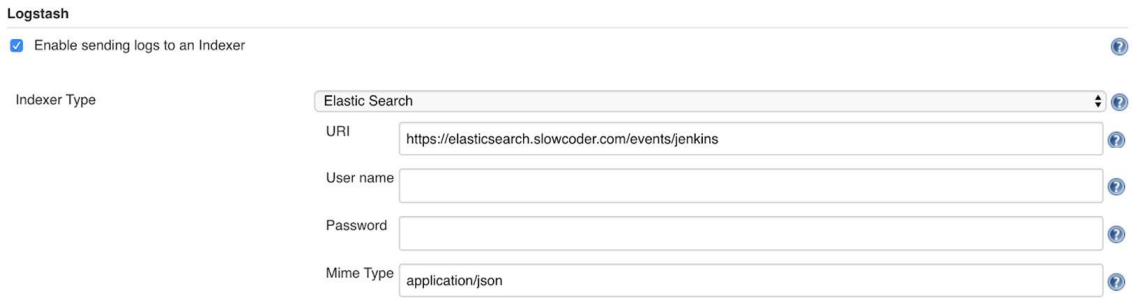

Figure 13.28 Configuring the Logstash plugin to stream logs to the Elasticsearch server

After configuring the Elasticsearch endpoint in the Logstash configuration, you can add the following block to your pipelines. That way, all the logs produced within the `logstash` step will be streamed into Elasticsearch:

```
logstash {
     echo "Job:${env.JOB_NAME}"
}
```

You can view the streamed logs by accessing the Kibana dashboard, shown in figure 13.29.

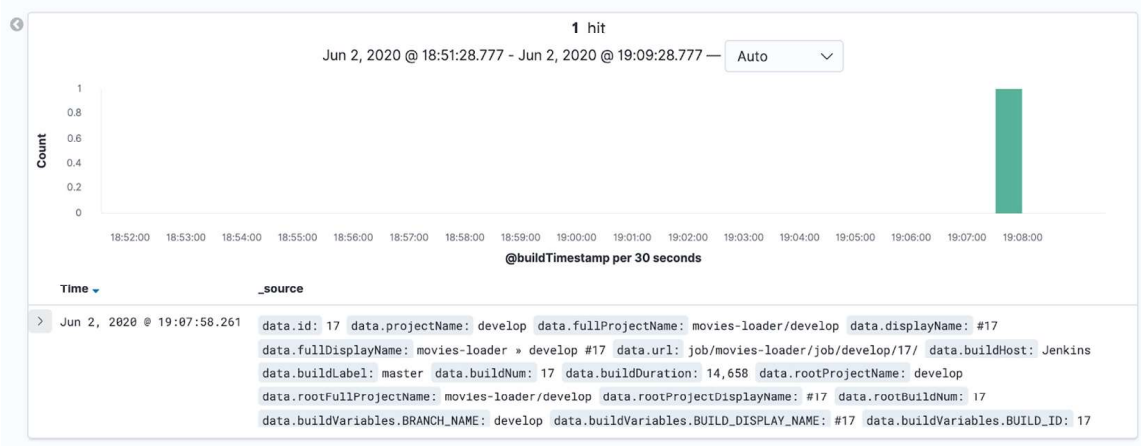

Figure 13.29 Example of a log message sent to Elasticsearch

Now we are able to stream the log data from the Jenkins instance to Elasticsearch and finally to Kibana.

13.3 Creating alerts based on metrics

We can take the logging and monitoring solutions further and set up alerts. One of the most common use cases is DevOps teams getting notifications of events, such as when the failure build rate is significantly higher than usual. Needless to say, this issue

can have a significant impact on the release of new features, hence having an impact on business and user experience.

You can use Kibana to define a meaningful alert on a specified condition; see figure 13.30. For instance, you can define an alert to periodically check the failure build rate. For the notification channel, you can use Slack, OpsGenie, or a simple email notification.

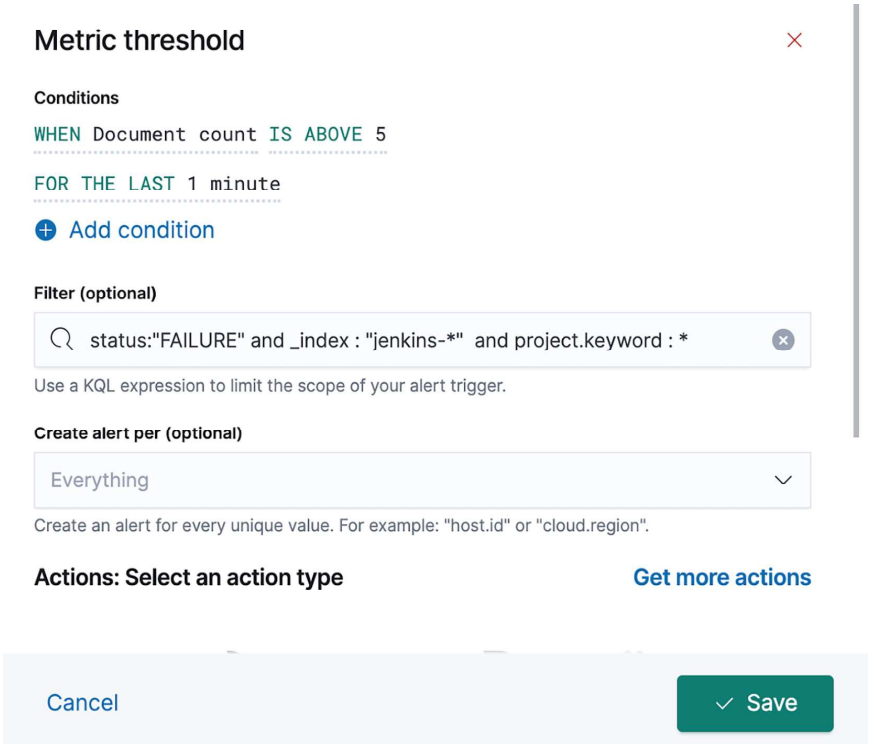

Figure 13.30 Configuring an alert on Kibana

You can also create alerts based on metrics collected by Prometheus or Telegraf, by using the Grafana alerting feature.

> **NOTE** While it's easy to set up and use Grafana alerting, it's more limited in terms of the alert rules you can apply to your metrics queries. If you're looking for an advanced solution, go with Prometheus Alertmanager (https://prometheus.io/docs/alerting/latest/alertmanager/).

Before creating monitoring alerts, we need to add the notification channel through which we will be notified. Here, we will be adding Slack as the notification channel.

To set up Slack, you need to configure an incoming Slack webhook URL. Create a Slack application by going to https://api.slack.com/apps/new. After creating the application, you'll be redirected to the Settings page of the new app (figure 13.31). From there, enable the Incoming Webhook feature by switching the radio button to On.

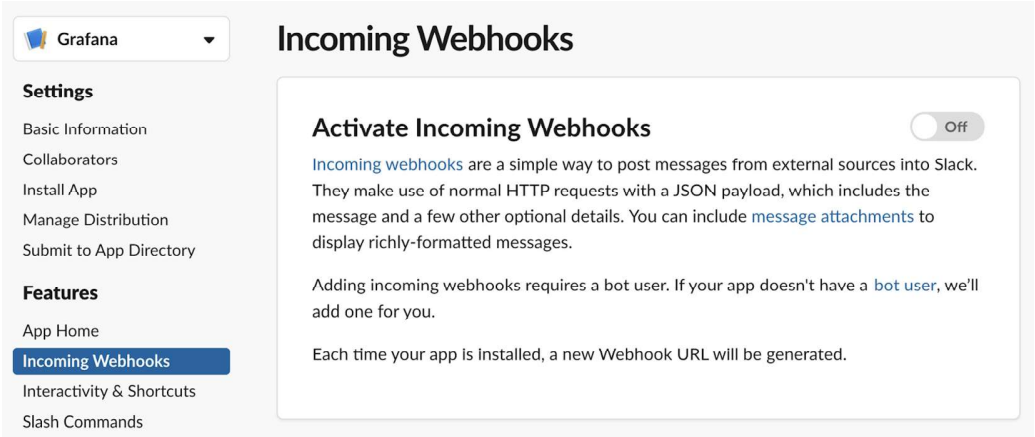

Figure 13.31 Enabling the incoming webhook on a Slack application

Now that incoming webhooks are enabled, the Settings page should refresh, and some extra options will appear. One of those options will be a really helpful button marked Add New Webhook to Workspace, and you should click it.

Go ahead and pick a Slack channel that Grafana will post to, and then click Authorize Your App. You'll be sent back to your app settings, where you should now see a new entry under the webhook URLs for the Your Workspace section, with a webhook URL. Copy it.

After creating the webhook URL, you need to create a notification channel in Grafana. In the Grafana sidebar, hover your cursor over the Alerting icon and then click Notification Channels, as shown in figure 13.32. Create a Slack notification channel as follows:

1. Input the name of the channel.
2. Change Type to Slack and input a webhook URL that you have created.

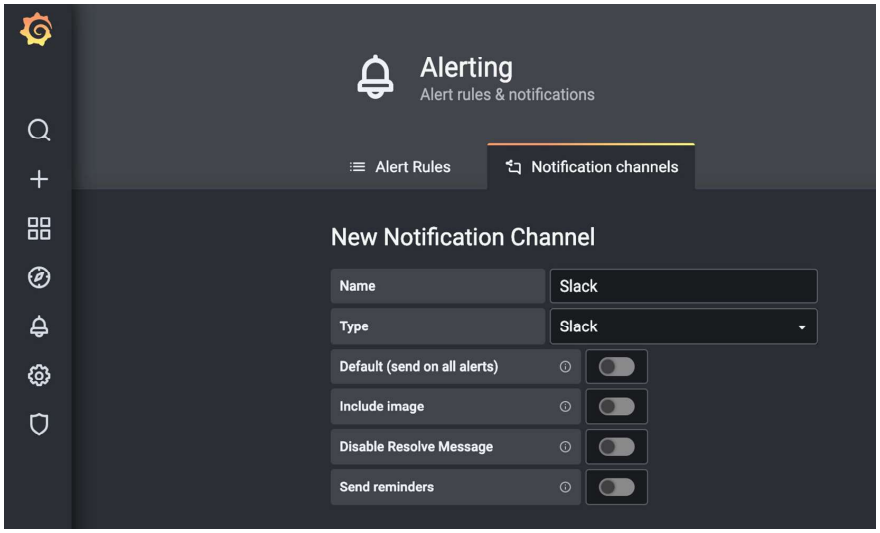

Figure 13.32 Configuring a new Slack notification channel

You can test the setup by clicking the Send Test button at the bottom. After setting up all the fields, just click the Save button.

Now let's create the alert. Select the panel where you want to create an alert. For instance, we can create an alert on the memory usage metric. Click the Alert tab and then click Create Alert. This will open a form for configuring the alert, where you can set the following options:

- *Evaluate Every*—The time interval on which you want the alert rule to be evaluated. For this example, we can set the option to Evaluate Every 1m for 1m. It means that Grafana will evaluate the rule every minute. If the metrics violate the rule, Grafana will wait for 1 minute. If, after 1 minute, the metrics are not recovered, Grafana will trigger an alert.
- *Conditions*—We can use the `avg()` function as we want to validate our rule against the average memory utilization.

This alert will be triggered when the average memory utilization is above 90%, as shown in figure 13.33.

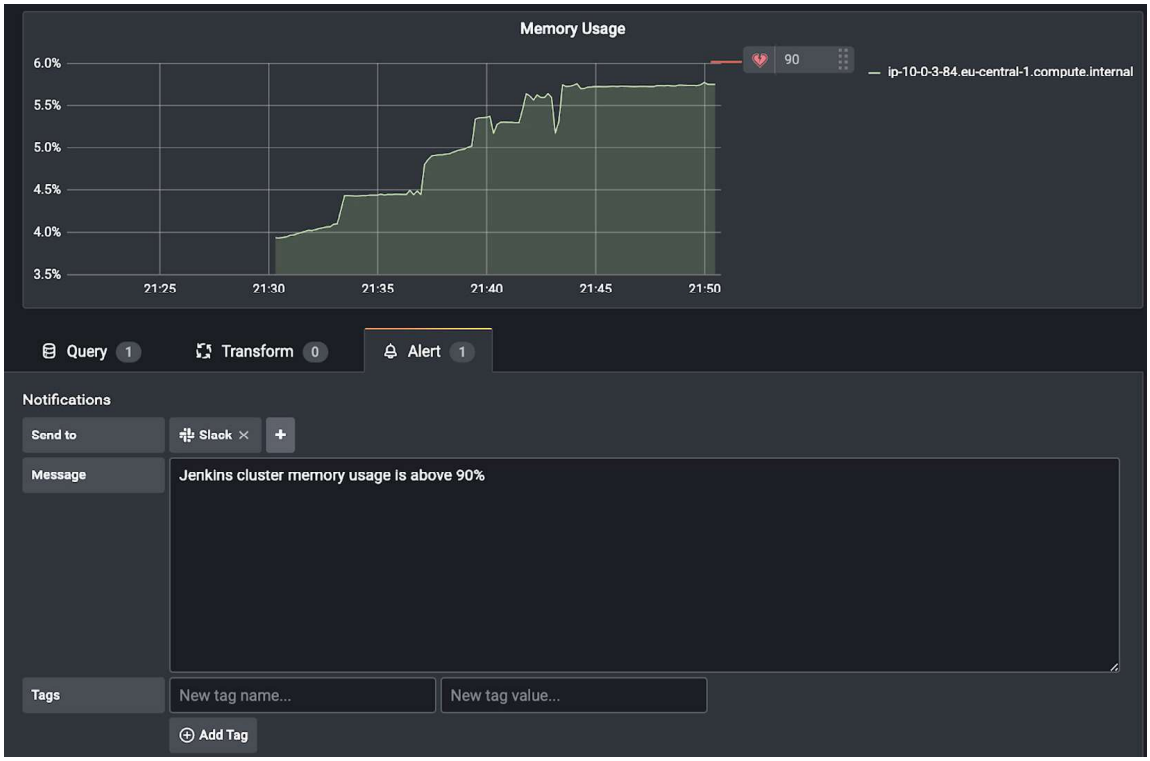

Figure 13.33 Defining an alert rule for memory usage

Additionally, we need to add the notification channel where the alert needs to be sent, as well as the alert message. If the alert is triggered, you will see the message in figure 13.34 on your Slack channel.

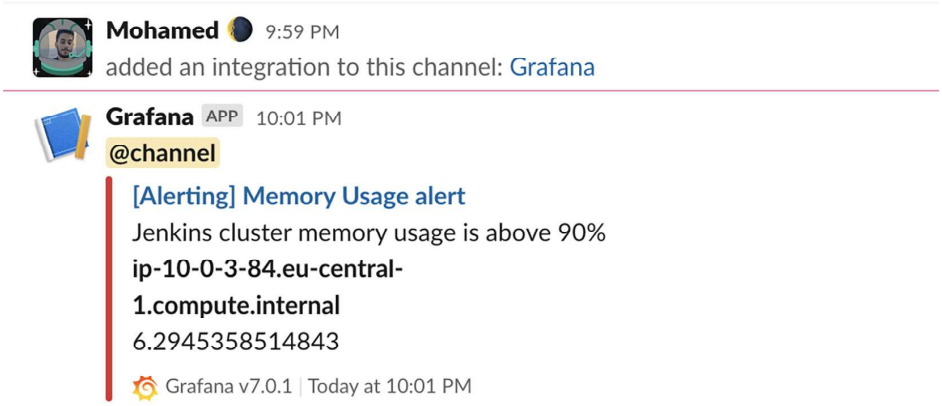

Figure 13.34 Slack notification upon memory threshold exceeded

Creating an alert to a messaging application like Slack is very beneficial. This ensures that you and your teammates get notifications immediately if something wrong happens. You can mention your team Slack group or use `@here` or `@channel` to make sure your team gets the message.

Summary

- You can build a monitoring stack with Telegraf, InfluxDB, and Grafana to collect, store, and visualize Jenkins instance metrics.
- You can collect and parse Jenkins logs into structured fields by writing Grok expressions.
- The Prometheus plugin can be used to expose internal and client-side metrics in Jenkins.
- The Logstash plugin is an easy way to integrate Jenkins logs with the ELK stack.
- Filebeat can be installed as an agent on your Jenkins master instance to ship logs to Logstash for parsing. From there, logs will be stored in Elasticsearch and analyzed from Kibana within an interactive dashboard.

14 Jenkins administration and best practices

This chapter covers

- Sharing common code and steps across CI/CD pipelines
- Granting job permissions for a user
- Using GitHub for authentication information to secure a Jenkins instance
- Backing up and restoring Jenkins plugins and jobs
- Using Jenkins as a scheduler for cron jobs
- Migrating build jobs to a new Jenkins instance

Chapter 13 covered how to monitor a Jenkins cluster, and how to configure alerts and correlate Jenkins logs and metrics to identify issues and avoid downtime. In this chapter, you will learn how to enforce security on Jenkins by setting up granular access with role-based access control (RBAC) for logged-in users and how to add an extra security layer by using the GitHub authentication mechanism.

We also will discuss a few tips and tricks that you might find useful when maintaining a Jenkins instance. We will look at things like how to back up, restore, and archive build jobs or migrate them from one server to another.

14.1 Exploring Jenkins security and RBAC authorization

The current configuration of Jenkins allows not-logged users to have read access, and logged users to access almost everything. To override this default behavior, head to the Configure Global Security section from Manage Jenkins (figure 14.1).

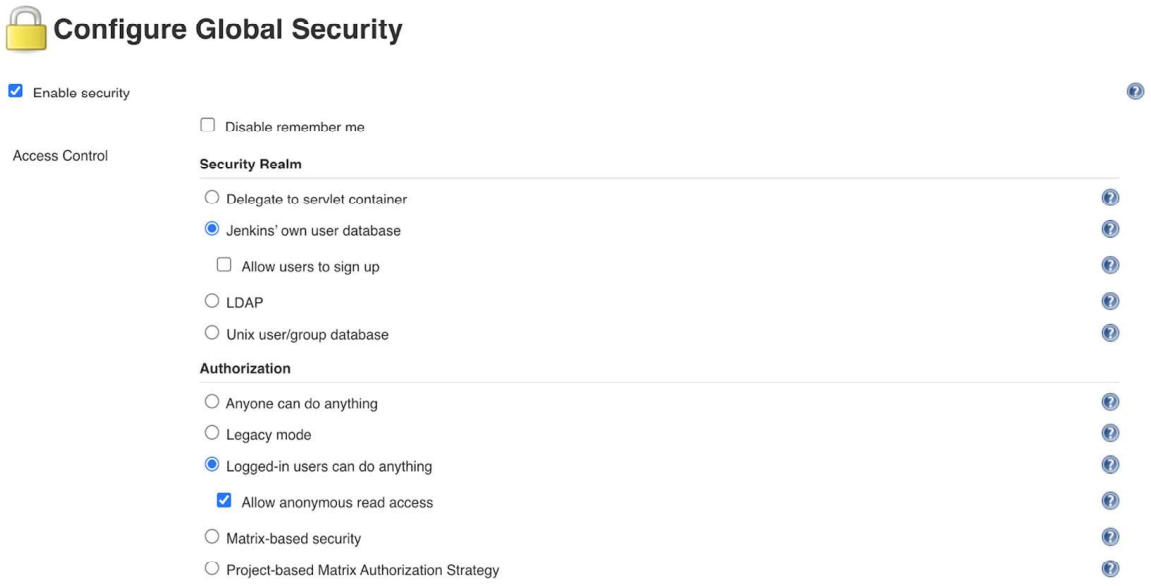

Figure 14.1 Enabling security in Jenkins

Disable Allow Anonymous Read Access and enable Allow Users to Sign Up, and you will be redirected to the sign-in page. This option allows users to create accounts by themselves via the Create an Account link, shown in figure 14.2.

Figure 14.2 Jenkins sign-in page

Click the Create an Account link. You will be prompted to add a new user. In figure 14.3, we are setting up a developer account.

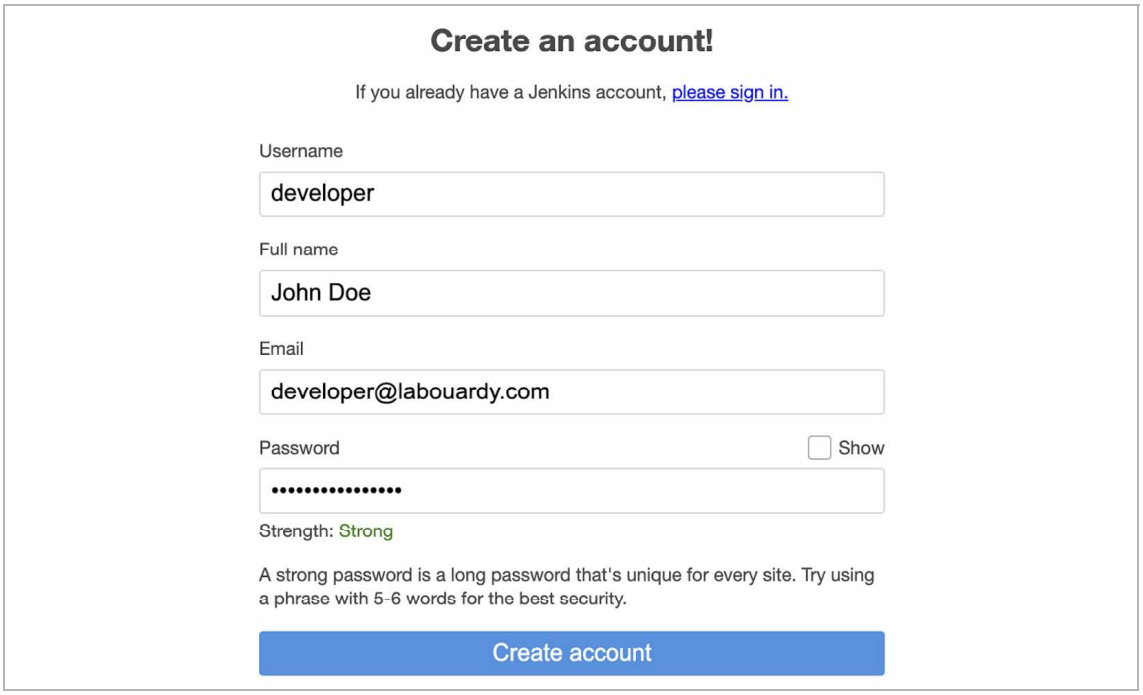

Figure 14.3 Setting up a developer account

Once the new account is created, sign in. You'll notice that it has full control of Jenkins. Letting signed-in users do anything is certainly flexible, and maybe all you need for a small team. For larger or multiple teams, or when Jenkins is being used outside the development environment, a more secure approach is generally required.

> **NOTE** By default, Jenkins does not use CAPTCHA verification if the user creates an account. If you'd like to enable CAPTCHA verification, install a support plugin such as the Jenkins JCaptcha plugin (https://plugins.jenkins.io/jcaptcha-plugin/).

14.1.1 Matrix authorization strategy

To set up granular access for logged-in users, we can use the Jenkins Matrix Authorization Strategy plugin (https://plugins.jenkins.io/matrix-auth/). This plugin allows you to control job permission on each project with specific users who can do something on that job.

Once the Matrix Authorization Strategy plugin is installed, head to Configure Global Security. In the Authorization section, enable Project-Based Matrix Authorization Strategy. Jenkins will display a table containing authorized users, and check

boxes corresponding to the various permissions that you can assign to these users (figure 14.4).

Figure 14.4 Matrix-based security configuration

The permissions are organized into several groups, such as these:

- *Overall*—Covers basic system-wide permissions.
- *Credentials*—Covers managing Jenkins credentials.
- *Agent*—Covers permissions about build nodes or workers (adding or removing Jenkins nodes).
- *Job*—Covers job-related permissions (creating a new build job, updating or deleting an existing build job).
- *Run*—Covers rights related to particular builds in the build history.
- *View*—Covers managing views. Views in Jenkins allow us to organize jobs and content into tabbed categories.
- *SCM*—Covers permissions related to a version-control system (such as Git or SVN).

The matrix controls what users can do (read jobs, execute builds, install plugins, and so forth). We have a couple of built-in authorizations to consider:

- *Anonymous*—Anyone who has not logged in
- *Authenticated*—Anyone who has logged in

You can configure permissions for a specific user by clicking Add User or Group. Add two users: one administrator (say, `mlabouardy/admin`) and a regular user (say, `developer`).

All the check boxes next to users are for setting global permissions. Select all check boxes to give admin full permissions. For Developer (aka John Doe), we are selecting read permissions under Job. With this, Developer would now have read permission to view all jobs that we created in the previous chapters; see figure 14.5.

Click Save, and the login page opens if you log in using developer credentials. In this mode, the developer account has only read permissions, as shown in figure 14.6 (for example, the developer can't trigger a build or configure job settings).

Figure 14.5 Fine-tuning user permissions

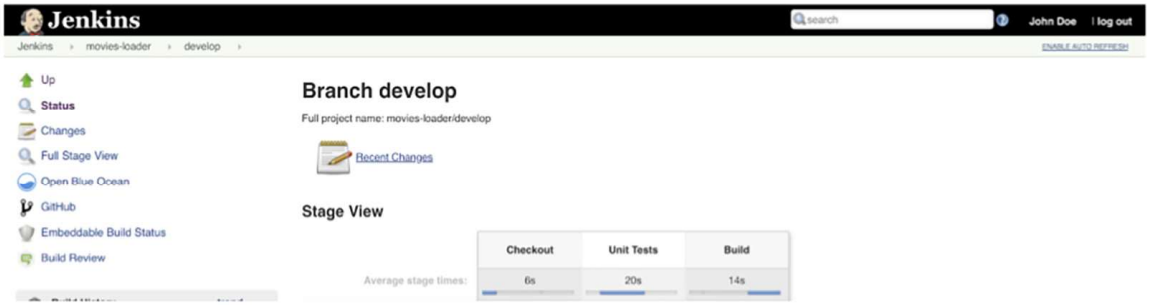

Figure 14.6 Jenkins read-only access

So far, you have seen how to create and manage Jenkins users as well as how to give granular access to these users. However, in a large organization, assigning granular permissions to multiple users can be tedious. Luckily, you can create different roles with the appropriate permissions and assign them to different users in Jenkins.

14.1.2 Role-based authorization strategy

To manage different roles, install the Role-Based Authorization Strategy plugin (https://plugins.jenkins.io/role-strategy/) from the Plugin Manager page. Then activate the Role-Based Strategy option from the Manage Global Security page, as shown in figure 14.7.

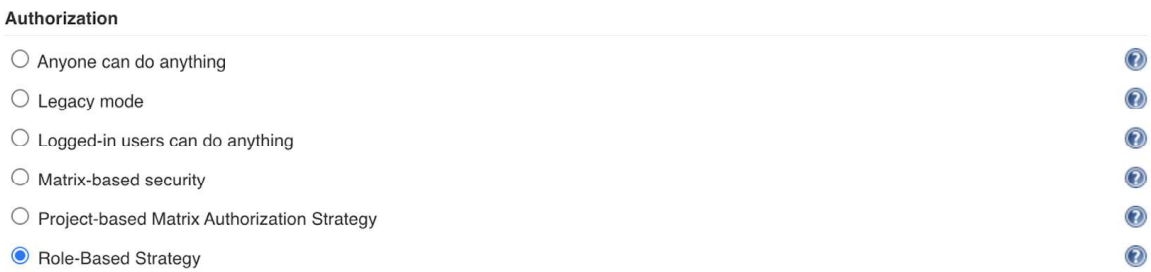

Figure 14.7 Enabling the Role-Based Authorization Strategy plugin

Then you can define global roles on the Manage Jenkins page by selecting the Manage and Assign Roles option (figure 14.8). Note that Manage and Assign Roles will be visible only if you have installed the plugin correctly.

Figure 14.8 Defining custom roles

Click the Manage Roles option to add new roles. Create three custom roles with the appropriate permissions:

- *Admin*—Will be assigned to Jenkins administrators for full access to Jenkins
- *Developer*—Will be assigned to developers for permissions to build jobs and view their logs and status
- *QA*—Will be assigned to software quality assurance engineer for permissions to view jobs status/health

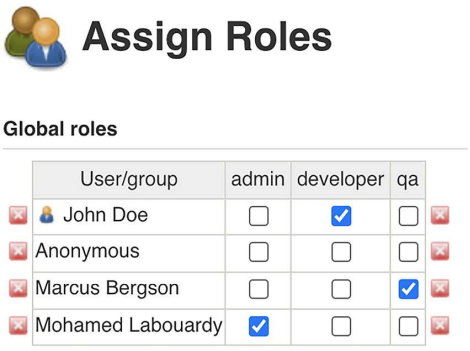

Figure 14.9 Managing and assigning roles

Then, assign these roles to specific users from the Assign Roles screen (figure 14.9). In these settings, we assign the admin's role to the administrator account, the developer's role to a member of the development team, and QA's role to a software QA.

If you're using Jenkins within an organization, creating and managing users' access might be a tedious task. You can use GitHub as an authentication mechanism.

NOTE You can configure many OAuth2 authentication services with Jenkins, including GitLab, Google, and OpenID.

14.2 Configuring GitHub OAuth for Jenkins

Jenkins supports several authentication plugins, in addition to built-in username and password authentication. If you're using GitHub as your version-control system within your organization, you can also use the GitHub OAuth service for user authentication and privileges management.

On Jenkins, install the GitHub Authentication plugin (https://plugins.jenkins.io/github-oauth/) from Manage Plugins. Once it's installed, head to your GitHub account and create a new application (https://github.com/settings/applications/new) called `Jenkins` with the settings in figure 14.10.

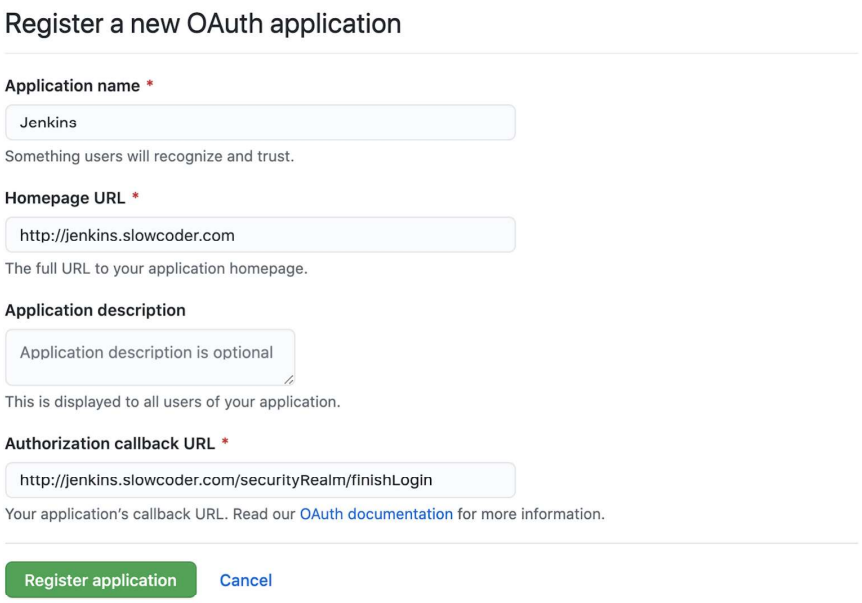

Figure 14.10 Configuring the GitHub OAuth application

The authorization callback URL must be JENKINS_URL/securityRealm/finishLogin. Click the Register Application button. A Client ID and secret will be generated, as shown in figure 14.11. Keep the page open to the application registration, so this information can be copied into your Jenkins configuration.

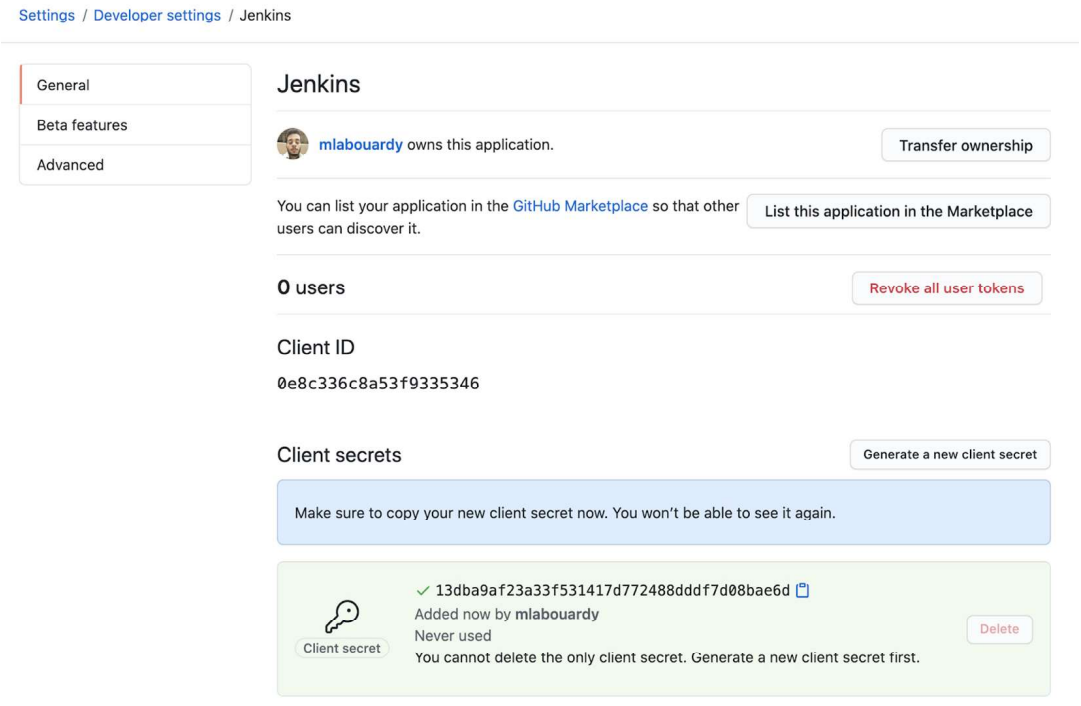

Figure 14.11 Application client ID and client secret

Head back to Jenkins, and in the Global Security configuration, set the Security Realm option to GitHub Authentication Plugin. Then set the Client ID, Client Secret, and OAuth scopes as shown in figure 14.12.

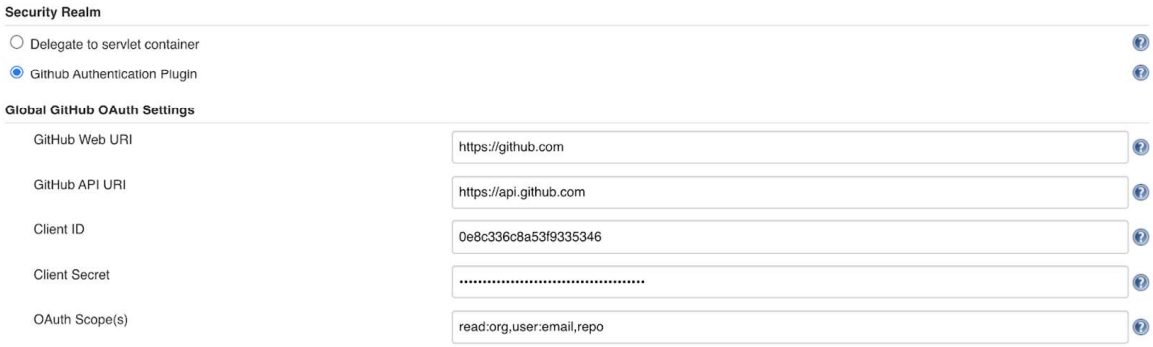

Figure 14.12 Configuring the Jenkins client settings for OAuth

Click the Save and Apply buttons to reload the configuration. You can now sign in with your GitHub account, as shown in figure 14.13.

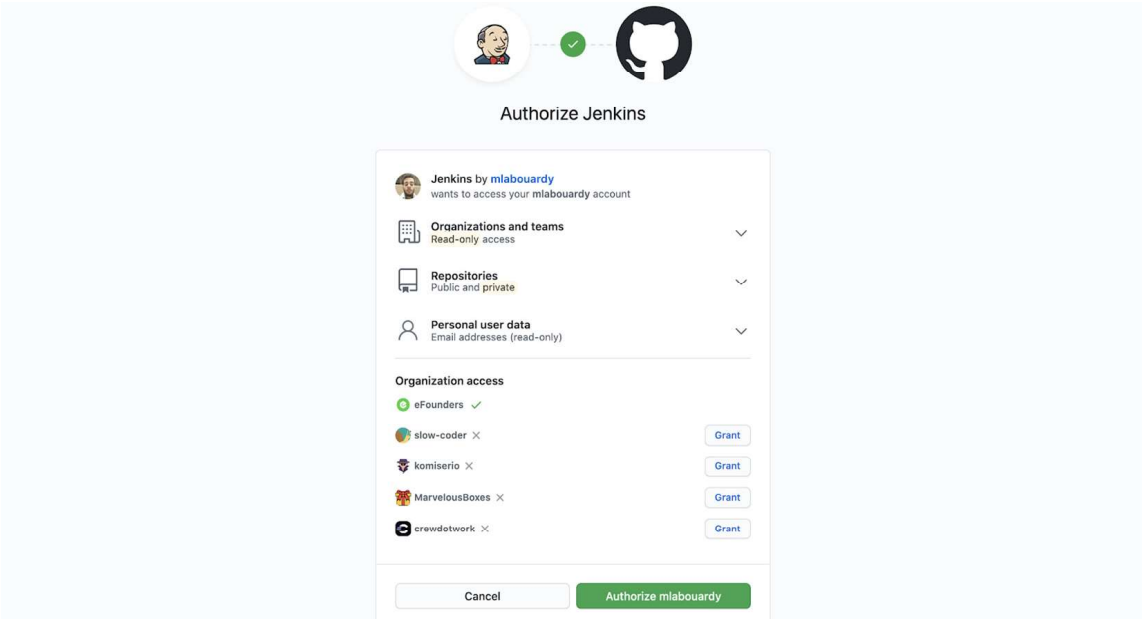

Figure 14.13 Authorizing Jenkins to access your GitHub account

Similar to classic username and password authentication, you can use a project-based matrix authorization strategy to determine Jenkins permissions for each GitHub account.

Another option is to use the GitHub Committer Authorization strategy. If you check this option, you can use GitHub repository permissions to determine permissions for each Jenkins project. If the GitHub repository of the project is public, all authenticated users will have read-only access, while project collaborators can build, edit, configure, cancel, or delete the Jenkins job. However, if the GitHub repository of the project is private, only collaborators can manage the Jenkins job.

To determine Jenkins access based on GitHub access, head to the Configure Global Security section from Manage Jenkins (figure 14.14).

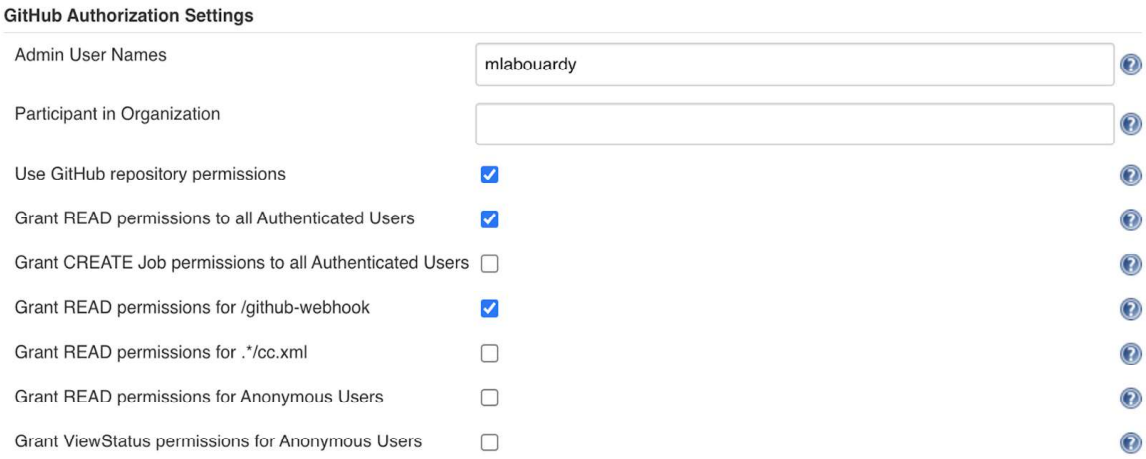

Figure 14.14 Configuring GitHub Authorization settings

NOTE We have authorized the use of the /github-webhook callback URL to receive post-commit hooks from GitHub.

14.3 Keeping track of Jenkins users' actions

In addition to configuring user accounts and access rights, keeping track of individual user actions can also be useful: in other words, who did what to your Jenkins configuration. This sort of audit trail facility is even required in many organizations for security compliance.

The Audit Trail plugin (https://plugins.jenkins.io/audit-trail/) keeps track of the main user actions in a set of rolling log files. To set this up, go to the Plugin Manager page and select the Audit Trail plugin in the list of available plugins. Then, as usual, click Install and Restart Jenkins after the plugin has been downloaded.

To enable audit logging, configure the plugin from the main Jenkins configuration page. Select Logfile as a Logger; that way, the plugin will produce a system-style log file. Then, set the log location (the directory in which the log files are to be written), as shown in figure 14.15. Of course, you need to ensure that the user running your Jenkins instance is allowed to write to this directory.

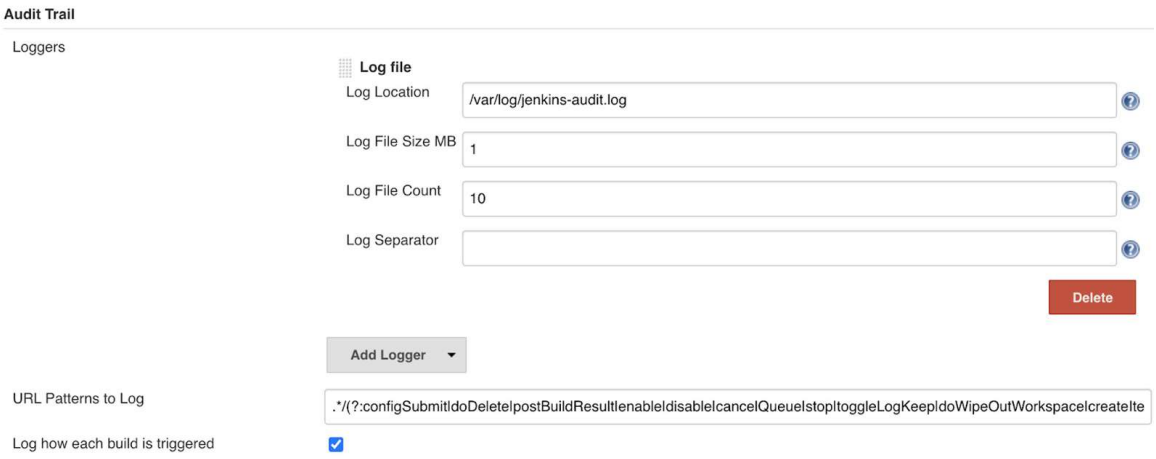

Figure 14.15 Configuring the Audit Trail plugin

By default, the details recorded in the audit logs are fairly sparse—they effectively record key actions performed, such as creating, modifying, or deleting job configurations or views, and the user who performed the actions. The log also shows how individual build jobs started. Figure 14.16 shows an extract of the default log.

```
[root@ip-10-0-0-61 jenkins]# tail -f jenkins-audit.log.0
Nov 15, 2020 5:14:19,421 PM/configSubmit by mlabouardy
Nov 15, 2020 5:14:28,449 PMjob/movies-marketplace/job/develop/ #3 Started by user mlabouardy, Parameters:[]
Nov 15, 2020 5:14:49,331 PMjob/movies-store/job/develop/ #3 Started by user John Doe, Parameters:[]
Nov 15, 2020 5:14:50,821 PMmovies-store » develop #3 Started by user John Doe, Parameters:[] on node #unknown# started
Nov 15, 2020 5:15:01,971 PMmovies-marketplace » develop #3 Started by user mlabouardy, Parameters:[] on node #unknown#
```

Figure 14.16 Viewing audit logs for the authorized user activity

You can also configure the number of log files to be maintained and the maximum size of each file. In the previous configuration, we have the Log File Count set to 10; in this case, Jenkins will write to log files with names like jenkins-audit.log.0, jenkins-audit.log.1 . . . jenkins-audit.log.9. Now, you can access the configuration history for the whole server, including system configuration updates, as well as the changes made to the configuration of each project.

> **NOTE** You can take the preceding configuration further and stream those log files to a centralized ELK platform and set up alerts on unauthorized user activities. For a step-by-step guide, head back to chapter 13.

14.4 Extending Jenkins with shared libraries

Throughout this book, you have learned how to write a CI/CD pipeline for multiple applications, and while implementing those pipeline steps, we have invoked multiple

custom functions. Those functions, shown in the following listing, were duplicated in multiple Jenkinsfiles.

Listing 14.1 Helper functions for Git and Slack

```
def commitAuthor(){
    sh 'git show -s --pretty=%an > .git/commitAuthor'
    def commitAuthor = readFile('.git/commitAuthor').trim()
    sh 'rm .git/commitAuthor'
    commitAuthor
}

def commitID() {}
def commitMessage() {}
def notifySlack(String buildStatus){}
```

Therefore, we had some common code across different pipelines. To avoid copying and pasting the same code into different pipelines, and to reduce redundancies, we can centralize the common code in a shared library within Jenkins. That way, we can reference the same code in all of the pipelines.

A *shared library* is a collection of independent Groovy scripts stored in a Git repository. This means you can version, tag, and do all the stuff you're used to with Git. Before writing our first shared library in Jenkins, we need to create a GitHub repository where Groovy scripts will be stored.

Inside the repository, create a vars folder and write a Groovy script per function. For example, create a file named commitAuthor.groovy and define a function called `call`. The body of the function is what will be executed when the `commitAuthor` instruction is invoked, as shown in the following listing.

Listing 14.2 Defining a global variable in the shared library

```
#!/usr/bin/env groovy

def call() {
 sh 'git show -s --pretty=%an > .git/commitAuthor'
 def commitAuthor = readFile('.git/commitAuthor').trim()
 sh 'rm .git/commitAuthor'
 commitAuthor
}
```

Searches your path looking for Groovy to execute the script

Allows the global variable to be invoked in a manner similar to a step

Prints the Git commit author

Notice that the Groovy script must implement the `call` method. Write your custom code within the braces `{}`. You can also add parameters to your method. Do the same for other functions and push the changes to the remote repository. Eventually, your repository should look like figure 14.17.

Figure 14.17 Shared library custom global variables

Now that you've created your library with custom steps, you need to tell Jenkins about it. To add a shared library, head to a job configuration. Under Pipeline Libraries, add a library with the following settings:

- *Name*—A short identifier that will be used in pipeline scripts
- *Default version*—Could be anything understood by Git—for example, branches, tags, or commit ID hashes

Next, load the library from the GitHub repository at the master branch, as shown in figure 14.18.

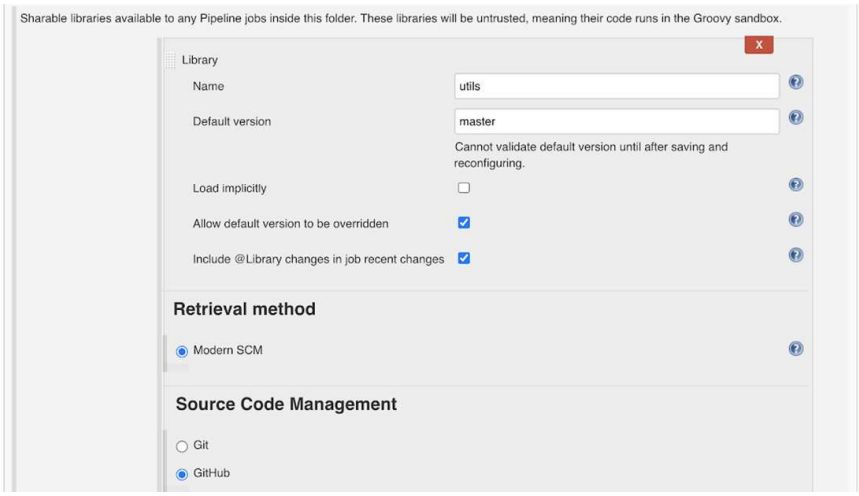

Figure 14.18 Loading a shared library from GitHub

NOTE You can also define a shared library globally, from Manage Jenkins > Configure System > Global Pipeline Libraries. That way, all pipelines can use functionality implemented in this library.

To load the shared library in a pipeline, you need to import it with the `@Library` annotation at the top of your pipeline definition. Then call the target function by its name, as shown in the following listing.

Listing 14.3 Importing the shared library in the scripted pipeline

```
@Library('utils')_

node('workers'){
 stage('Checkout'){
    checkout scm
    notifySlack 'STARTED'
 }
}
```

The underscore is required if the line immediately after the @Library annotation is not an import statement.

The underscore is not a typo or mistake; you need this if the line immediately after the `@Library` annotation is not an import statement. You can override the default version defined for the library with the `@Library('id@version')` annotation.

If you're using a declarative pipeline, you need to wrap the library name inside a libraries section, as shown in the following listing.

Listing 14.4 Importing shared library in the declarative pipeline

```
libraries {
     lib('utils')
 }
 pipeline {
     // Your pipeline would go here....
 }
```

When using a library, you may also specify a version with the following format:

```
 libraries {
     lib('utils@VERSION')
 }
```

Run the previous pipeline, and the output should look something like figure 14.19.

```
Examining mlabouardy/pipeline-library
Attempting to resolve master as a branch
Resolved master as branch master at revision e8dadc3faa6d861591acee8dd2bcbdbe3c9bd587
using credential github
 > git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url https://github.com/mlabouardy/pipeline-library.git # timeout=10
Fetching without tags
Fetching upstream changes from https://github.com/mlabouardy/pipeline-library.git
 > git --version # timeout=10
using GIT_ASKPASS to set credentials github
 > git fetch --no-tags --progress -- https://github.com/mlabouardy/pipeline-library.git +refs/heads/master:refs/remotes/origin/master
# timeout=10
Checking out Revision e8dadc3faa6d861591acee8dd2bcbdbe3c9bd587 (master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f e8dadc3faa6d861591acee8dd2bcbdbe3c9bd587 # timeout=10
Commit message: "shared library"
 > git rev-list --no-walk e8dadc3faa6d861591acee8dd2bcbdbe3c9bd587 # timeout=10
Replacing contents of vars/notifySlack.groovy
Replacing contents of vars/commitID.groovy
Replacing contents of vars/commitMessage.groovy
Replacing contents of vars/commitAuthor.groovy
[Pipeline] Start of Pipeline
```

Figure 14.19 Loading the shared library from Git within a pipeline

Another way to write a library is to define the functions within a Groovy class. Create the `Git.groovy` class in src/com/labouardy/utils, as shown in the following listing.

Listing 14.5 Writing a shared library

```
#!/usr/bin/env groovy
package com.labouardy.utils

class Git {
   Git(){}

   def commitAuthor() {
       sh 'git show -s --pretty=%an > .git/commitAuthor'
       def commitAuthor = readFile('.git/commitAuthor').trim()
       sh 'rm .git/commitAuthor'
       commitAuthor
   }

   def commitID() {
       sh 'git rev-parse HEAD > .git/commitID'
       def commitID = readFile('.git/commitID').trim()
       sh 'rm .git/commitID'
       commitID
   }

   def commitMessage() {
       sh 'git log --format=%B -n 1 HEAD > .git/commitMessage'
       def commitMessage = readFile('.git/commitMessage').trim()
       sh 'rm .git/commitMessage'
       commitMessage
   }
}
```

You can load classes defined in the library by selecting their fully qualified name:

```
@Library('utils') import com.labouardy.utils.Git
this.commitAuthor()
```

Or you can create an object constructor function and then call the method from the object:

```
def gitUtils = new Git(this)
gitUtils.commitAuthor
```

> **NOTE** It is possible to use third-party Java libraries, typically found in Maven Central (https://search.maven.org/), from trusted library code by using the `@Grab` annotation. Refer to the Grape documentation for details (http://mng.bz/nrxg).

14.5 Backing up and restoring Jenkins

Backing up your data is a universally recommended practice, and your Jenkins server should be no exception. Fortunately, backing up Jenkins is relatively easy. In this section, we will look at a few ways to do this.

In Jenkins, all the settings, build logs, and archives of the artifacts are stored under the $JENKINS_HOME directory. You can back up the directory manually, or by using a plugin like ThinBackup (https://plugins.jenkins.io/thinBackup/). The plugin provides a simple user interface that you can use to back up and restore your Jenkins configurations and data.

Once you install the plugin, you need to configure the backup directory, as shown in figure 14.20. Specify the backup directory to be /var/lib/backups. Be sure Jenkins has write rights!

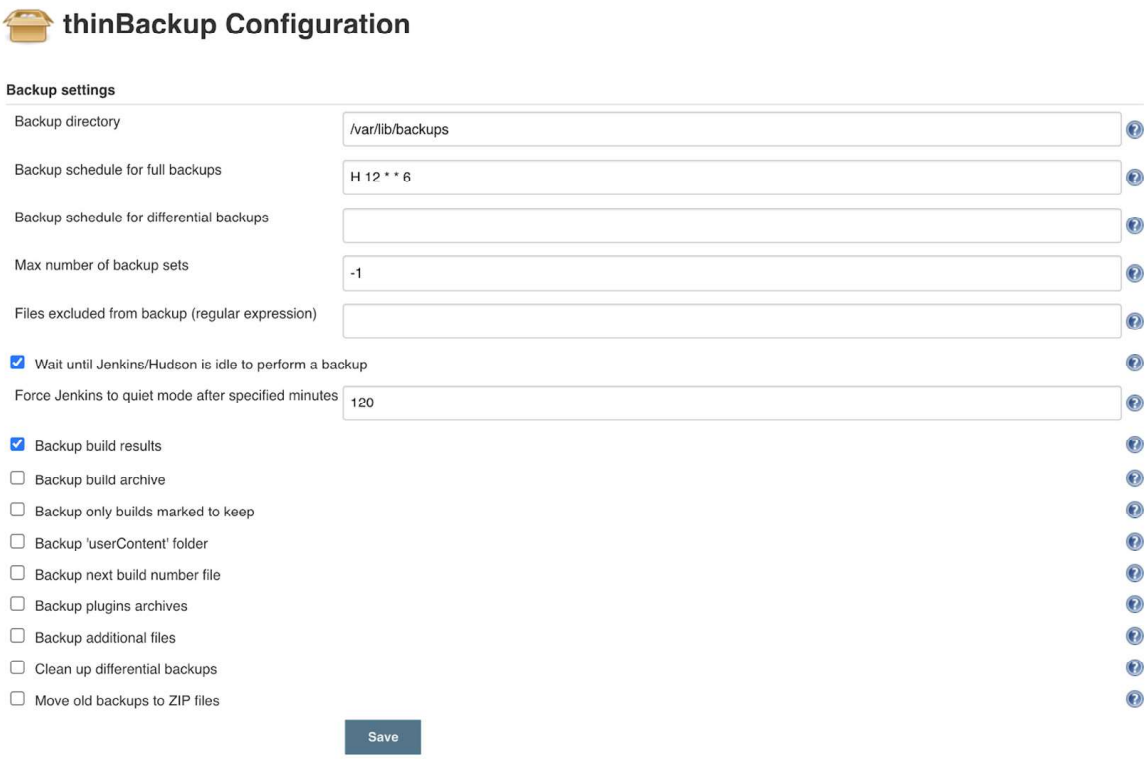

Figure 14.20 Configuring the ThinBackup plugin

Now, you can test whether the backup is working by clicking the Backup Now option. It will create a backup of Jenkins data in the backup directory you specified in the settings:

```
[root@ip-10-0-0-116 backups]# pwd
/var/lib/backups
[root@ip-10-0-0-116 backups]# ls
FULL-2020-11-14_19-11  FULL-2020-11-14_19-12
[root@ip-10-0-0-116 backups]#
```

To restore a previous configuration, just go to the Restore page and choose the date of the configuration you wish to reinstate, as shown in figure 14.21. Once the

configuration has been restored to the previous state, you need to reload the Jenkins configuration from disk or restart Jenkins.

Figure 14.21 Restoring a previous configuration

As a result of the backup, you can restore Jenkins from an earlier point in time in case of data corruption or a human-caused event.

> **NOTE** The ThinBackup plugin stores the backup locally for production usage. It's highly recommended to store your backups on a remote server or mount an external data storage.

If you're not a fan of plugins, you can set up a cron job (see the next section for more details) on Jenkins to schedule regular backups. It will back up everything located at /var/lib/jenkins to a remote repository such as S3 bucket, as shown in the following listing.

Listing 14.6 Backing up the $JENKINS_HOME folder to an S3 bucket

```
cd $JENKINS_HOME
BACKUP_TIME=$(date +'%m.%d.%Y')
zip -r backup-${BACKUP_TIME} .
aws s3 cp  backup-${BACKUP_TIME} s3://BUCKET/
```

Sometimes you need to move or copy Jenkins build jobs from one Jenkins instance to another, without copying the entire Jenkins configuration. For example, you might be migrating your build jobs to a Jenkins server on a brand-new instance.

You can copy or move build jobs between instances of projects simply by copying or moving the build job directories to the new Jenkins instance. I have built an open source CLI called Butler (https://github.com/mlabouardy/butler) to import/export Jenkins jobs and plugins easily.

To get started, find the appropriate package for your system and download it. Here's the command for Linux:

```
wget https://s3.us-east-1.amazonaws.com/butlercli/1.0.0/linux/butler
chmod +x butler
cp butler /usr/local/bin/
```

Verify that the installation worked by opening a new terminal session and checking whether Butler is available. To export Jenkins plugins, you need to provide the Jenkins URL:

```
butler jobs export --server JENKINS_URL --username USERNAME --password
     PASSWORD
```

A new jobs/ directory will be created with every job in Jenkins. Each job will have its own configuration file, config.xml.

To import the plugins, issue the `butler plugins export` command. Butler will dump a list of plugins installed to stdout, and a new file, plugins.txt, will be generated, with a list of installed Jenkins plugins with name and version pairs, as shown in figure 14.22.

```
mlabouardy@Mohameds-MBP-001 github % butler jobs export --server jenkins.slowcoder.com  --username mlabouardy --password mlabouardy
Exporting job: movies-loader
Exporting job: movies-marketplace
Exporting job: movies-parser
Exporting job: movies-store
mlabouardy@Mohameds-MBP-001 github % butler plugins export --server jenkins.slowcoder.com  --username mlabouardy --password mlabouardy
+-----------------------------------+------------+---------------------------------+
|               NAME                |  VERSION   |          DESCRIPTION            |
+-----------------------------------+------------+---------------------------------+
| blueocean-personalization         | 1.21.0     | Personalization for Blue Ocean  |
| subversion                        | 2.13.0     | Jenkins Subversion Plug-in      |
| trilead-api                       | 1.0.5      | Trilead API Plugin              |
| mapdb-api                         | 1.0.9.0    | MapDB API Plugin                |
| structs                           |       1.20 | Structs Plugin                  |
| blueocean-dashboard               | 1.21.0     | Dashboard for Blue Ocean        |
| managed-scripts                   |        1.4 | Managed Scripts                 |
| token-macro                       |       2.10 | Token Macro Plugin              |
| favorite                          | 2.3.2      | Favorite                        |
| amazon-ecr                        |        1.6 | Amazon ECR plugin               |
| workflow-api                      |       2.38 | Pipeline: API                   |
| blueocean-bitbucket-pipeline      | 1.21.0     | Bitbucket Pipeline for Blue     |
|                                   |            | Ocean                           |
| workflow-job                      |       2.36 | Pipeline: Job                   |
```

Figure 14.22 Listing of installed Jenkins plugins

You can import exported jobs and plugins with the `butler plugins/jobs import` commands. Butler will use the exported files to issue API calls to the target Jenkins instance to import plugins and jobs.

So, all in all, migrating build jobs between Jenkins instances isn't all that hard—you just need to know a couple of tricks for the corner cases, and if you know where to look, Jenkins provides some nice tools to make the process smoother.

If you want $JENKINS_HOME content to be persisted on disk even if the Jenkins master instance has been restarted or shut down, you can mount a remote filesystem on the $JENKINS_HOME folder.

If you're running Jenkins on AWS, you can use an AWS service called Amazon Elastic File System, or EFS (https://aws.amazon.com/efs/). Create a filesystem on EFS by clicking the Create File System button (figure 14.23).

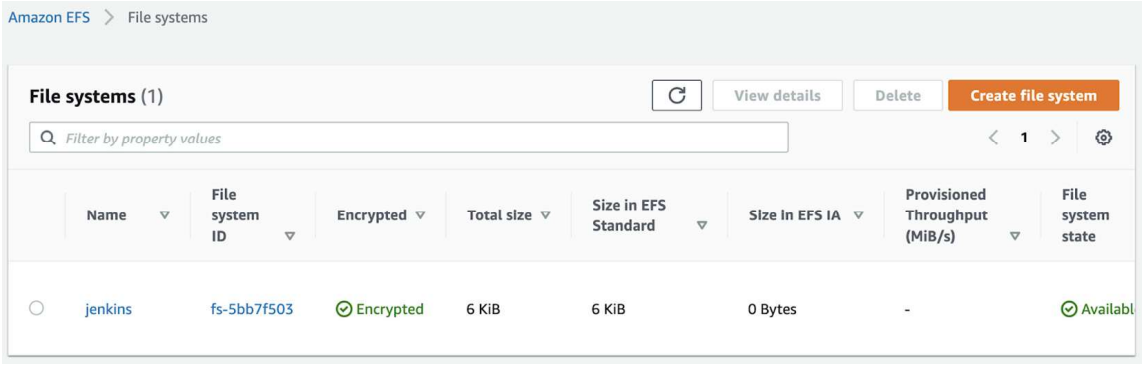

Figure 14.23 Creating an Amazon EFS filesystem

Once the filesystem is created and its state is Available, mount the EFS filesystem in the /var/lib/jenkins directory, so all the configuration will be saved in EFS:

```
sudo mount -t nfs4
-o nfsvers=4.1,rsize=1048576,wsize=1048576,
hard,timeo=600,retrans=2,noresvport
EFS_ID.efs.REGION.amazonaws.com:/ /var/lib/jenkins/
```

If you want to test it, terminate your EC2 instance and a new one will be launched automatically with the same configuration (make sure to add the mount commands to the Packer template while baking the Jenkins master AMI).

14.6 Setting up cron jobs with Jenkins

Jenkins provides a cron-like feature to periodically build a project. This feature is primarily used to run scheduled builds, like nightly/weekly builds or running tests. For example, you might want to run performance tests or integration tests for Android or iOS releases at night, when users do not access the backend under test.

To configure a scheduled nightly build that runs at a certain day and time, head over to Jenkins dashboard. Create a new job and select Freestyle Project. Configure the job accordingly by adding the job details shown in figure 14.24.

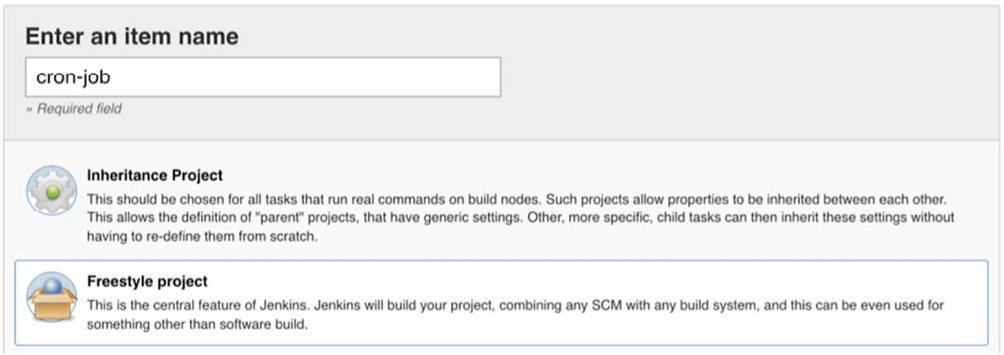

Figure 14.24 Creating a Freestyle project

Schedule your build from the Build Triggers tab by writing the cron syntax shown in figure 14.25, and then select the Build Periodically option. Fill in a cron-like value for the time you wish to trigger the pipeline execution.

Figure 14.25 Defining a cron job expression

Jenkins uses a cron expression, with fields as follows:

- MINUTES—Minutes in one hour (0–59)
- HOURS—Hours in one day (0–23)
- DAYMONTH—A day in a month (1–31)
- MONTH—Month in a year (1–12)
- DAYWEEK—Day of the week (0–7), where 0 and 7 are Sunday

For example (figure 14.26), to trigger a build at midnight on Sunday, the cron value `H 12 * * 7` will do the job.

> **NOTE** You should be aware that the time zone is relative to the location where your Jenkins virtual machine is running. This example uses Coordinated Universal Time (UTC).

Figure 14.26 Shell script to back up the $JENKINS_HOME folder

Build your job to test that everything is working as you've expected. Your build results should look like figure 14.27.

Figure 14.27 Triggering a cron job manually

Next time, your job will automatically execute at 12:00 A.M. since you have scheduled it to run at this time using cron syntax.

Jenkins jobs could be run programmatically, using API calls or the Jenkins CLI. That opens up the opportunity to implement complex schedule builds by integrating an external service like AWS Lambda to invoke a Jenkins build job based on different events; see figure 14.28.

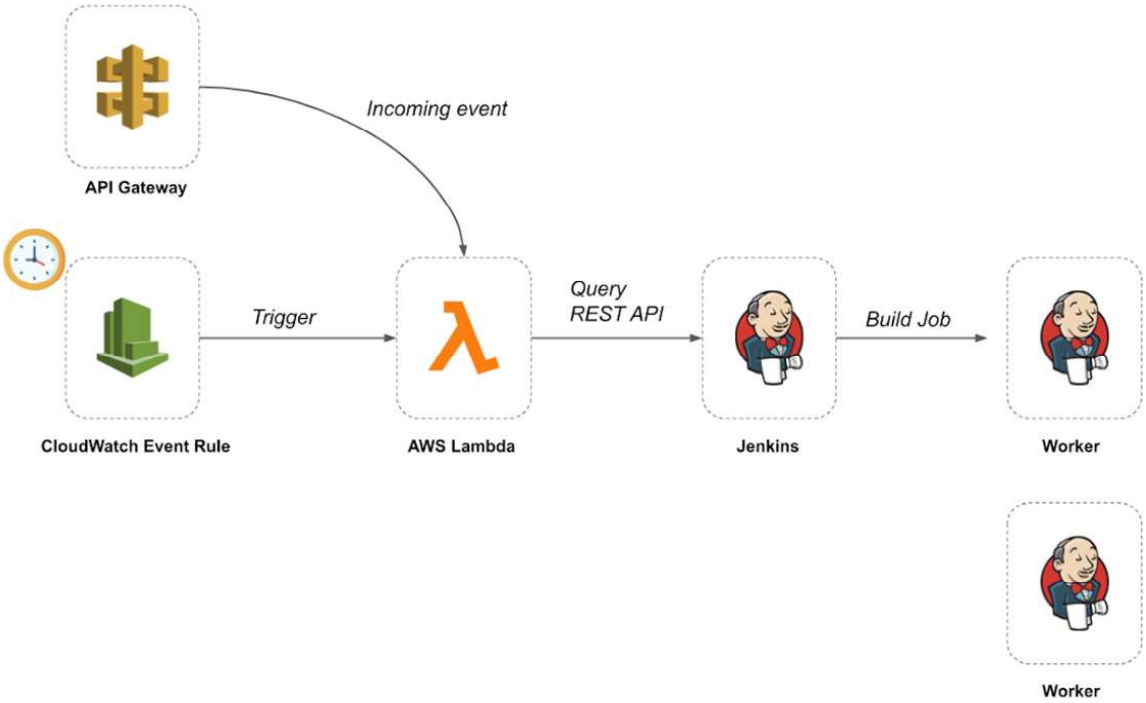

Figure 14.28 Triggering a Jenkins job from a Lambda function

This diagram covers how to trigger a Jenkins build job from a Lambda function through the Jenkins RESTful API. The Lambda function is invoked on the upcoming CloudWatch event rule (cloud-managed cron job) or HTTPS requests from API Gateway.

14.7 Running Jenkins locally as a Docker container

If you need to debug Jenkins or test a new plugin, you can deploy Jenkins locally on your machine and run it as a Docker container. That way, you can easily create and destroy a Jenkins server.

You can use the official Jenkins Docker image from the DockerHub repository (https://hub.docker.com/_/jenkins). The image contains the current LTS release of Jenkins (v2.60.3 at the time of this writing).

To get started, on your terminal, create a bridge network in Docker with the following command:

```
docker network create jenkins
```

We will need the Docker daemon to be able to provision Jenkins workers dynamically. That's why we will deploy a Docker container based on the Docker image:

```
docker run -d --name docker --privileged
--network jenkins --network-alias docker
--env DOCKER_TLS_CERTDIR=/certs
--volume jenkins-docker-certs:/certs/client
--volume jenkins-data:/var/jenkins_home
--publish 2376:2376 docker:dind
```

To avoid exposing the Docker daemon (/var/run.docker.sock) running in the host machine, we will run a Docker container providing a self-service and ephemeral Docker Engine, which Jenkins will use instead of the worker machine's Docker engine. This pattern is referred to as *Docker in Docker*, or *nested containerization*.

We will override the Jenkins official image to install the Docker CLI and needed plugins for Jenkins. Create a Dockerfile with the content in the following listing.

Listing 14.7 Dockerfile to build custom Jenkins image

```
FROM jenkins/jenkins:lts
MAINTAINER mlabouardy <mohamed@labouardy.com>

USER root
RUN apt-get update && apt-get install -y apt-transport-https \
      ca-certificates curl gnupg2 \
      software-properties-common
RUN curl -fsSL https://download.docker.com/linux/debian/gpg | apt-key add -
RUN apt-key fingerprint 0EBFCD88
RUN add-apt-repository \
      "deb [arch=amd64] https://download.docker.com/linux/debian \
      $(lsb_release -cs) stable"
RUN apt-get update && apt-get install -y docker-ce-cli
USER jenkins
RUN jenkins-plugin-cli
--plugins blueocean:1.24.3 workflow-aggregator:2.6
github:1.32.0 docker-plugin:1.2.1
```

Installs Docker community edition (CE) client

Switches to Jenkins user to avoid running the container by default in privileged mode

Installs the Jenkins plugins

This Dockerfile does the following:

- Installs the Docker Community Edition CLI
- Installs Jenkins plugins, including the following:
  - *Blue Ocean*—Sophisticated visualizations of CD pipelines for fast and intuitive comprehension of software pipeline status
  - *Workflow*—A suite of plugins that lets you write pipelines as code (Jenkinsfiles)
  - *GitHub*—GitHub API integration and support of Git operations
- *Docker*—Lets you provision Jenkins workers on Docker containers

Build a new Docker image from this Dockerfile and assign the image a meaningful name:

```
docker build -t jenkins-custom:lts .
```

Then, deploy a container based on the built image with the following `docker run` command:

```
docker run -d --name jenkins --network jenkins
--env DOCKER_HOST=tcp://docker:2376
--env DOCKER_CERT_PATH=/certs/client
--env DOCKER_TLS_VERIFY=1
--publish 8080:8080 --publish 50000:50000
--volume jenkins-data:/var/jenkins_home
--volume jenkins-docker-certs:/certs/client:ro
jenkins-custom:lts
```

This command will map a Docker volume to the /var/jenkins_home folder. In case you need to restart or recover your Jenkins instance, all of the state is stored inside the Docker volume.

You can also build and deploy all the services by writing a docker-compose.yml file, as shown in the following listing.

Listing 14.8 Grok custom patterns definition

```
version: "3.8"

services:
 docker:
   image: docker:dind
   ports:
     - "2376:2376"
   networks:
     jenkins:
         aliases:
           - docker
   environment:
     - DOCKER_TLS_CERTDIR=/certs
   volumes:
     - jenkins-docker-certs:/certs/client
```

```
      - jenkins-data:/var/jenkins_home
    privileged: true

 jenkins:
    build: .
    ports:
      - "8080:8080"
      - "50000:50000"
    networks:
      - jenkins
    environment:
      - DOCKER_HOST=tcp://docker:2376
      - DOCKER_CERT_PATH=/certs/client
      - DOCKER_TLS_VERIFY=1
    volumes:
      - jenkins-data:/var/jenkins_home
      - jenkins-docker-certs:/certs/client:ro

volumes:
 jenkins-docker-certs: {}
 jenkins-data: {}

networks:
 jenkins:
```

Run `docker-compose up`, and Docker Compose starts and runs Jenkins.

Visit localhost:8080; you should see the login page. As a part of the Jenkins setup, we need to view the password inside the container instance; use the container ID (or the name) and run the `docker exec` command:

```
docker container exec ID sh -c "cat /var/jenkins_home/secrets/
     initialAdminPassword"
```

After running the command, you should see the code. Copy and paste it on the dashboard to unlock Jenkins; see figure 14.29.

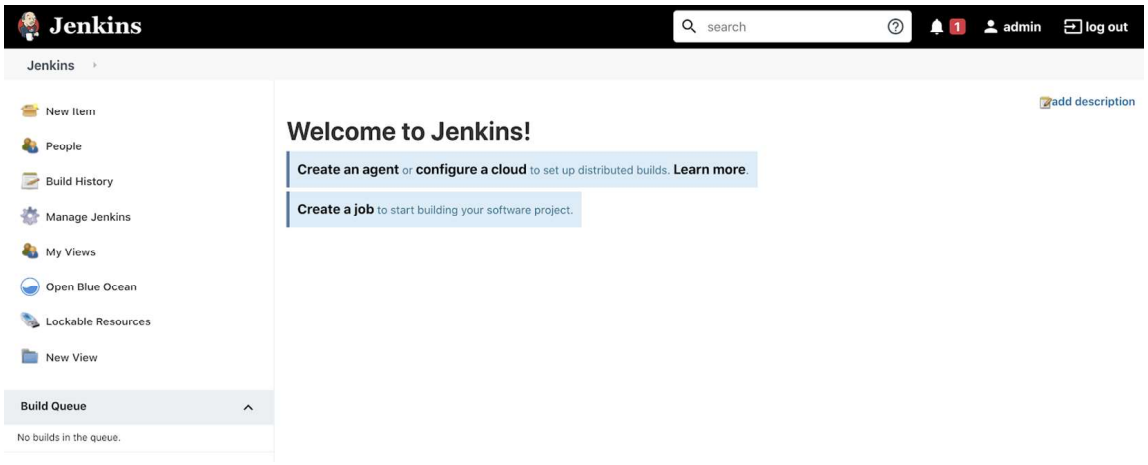

Figure 14.29 Jenkins server running inside a Docker container

To set up workers, choose Manage Jenkins and System Configuration. Then click the Configure tab in the Cloud section. The Docker option will be available. Set the Docker URI to `tcp://docker:2376`, as shown in figure 14.30. Click the Test button to check the connection.

Figure 14.30 Configuring Docker remote API on Jenkins

The Docker API should return an error: `server gave HTTP response to HTTPS client`. You need to configure the client TLS certificates to connect with the Docker daemon. The certificates can be found at the /certs/client folder within the Jenkins container.

Create a new Jenkins credential of type Certificate with the following settings:

- *Client Key*—/certs/client/key.pem content
- *Client Certificate*—/certs/client/cert.pem content
- *Server CA Certificate*—/certs/client/ca.pem content

The credential settings should look similar to those in figure 14.31.

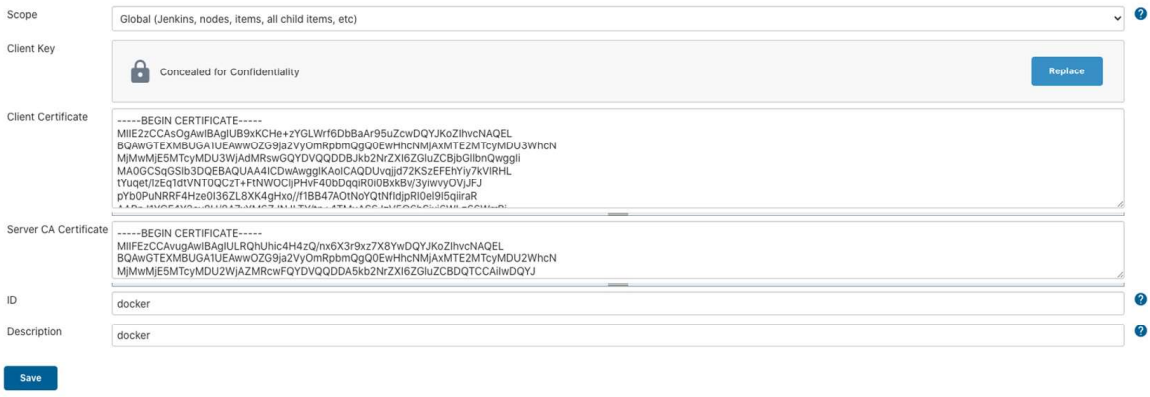

Figure 14.31 Jenkins server deployed locally inside a Docker container

Then, we need to define an agent template, as shown in figure 14.32; this template is the blueprint used to spin up Jenkins workers. You need a Docker image that can be

used to run the Jenkins agent runtime. You can use the jenkins/ssh-agent (https://hub.docker.com/r/jenkins/ssh-agent) as a base for Jenkins workers. The image has SSHD installed (this listens for an incoming connection when you attempt to connect via SSH).

Figure 14.32 Configuring a new Docker agent template

You can also build a custom Docker agent image with all dependencies and packages needed to build your projects. To test it out, create a new Jenkins pipeline with the content shown in figure 14.33.

Figure 14.33 New inline pipeline

Trigger the pipeline by clicking the Build Now link from the left navigation menu; the job will launch a container and execute the pipeline (figure 14.34).

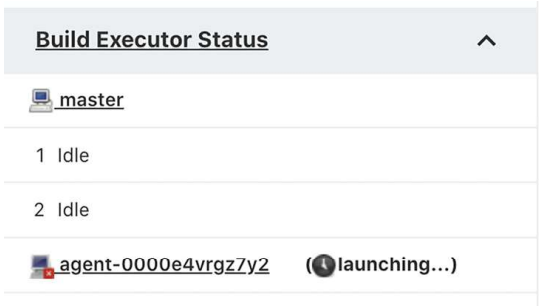

Figure 14.34 Spinning up the Jenkins agent based on a Docker container

The agents are provisioned dynamically and stopped after each build.

Summary

- You can share common code and steps across multiple pipelines by writing a Jenkins shared library.
- You can define fine-grained control over user/group permissions per project with the Matrix Authorization Strategy plugin.
- You can also create a custom role with a list of permissions and assign the role to users instead of assigning appropriate permissions to each user with the Role Strategy plugin.
- Use GitHub's own authentication scheme for implementing authentication in your Jenkins instance.
- The Docker plugin will run dynamic Jenkins agents inside Docker containers.

Wrapping up

We're at the end of our journey in this book. You learned about Jenkins and the pipeline-as-code approach. You discovered several CI/CD implementations for cloud-native applications, such as containerized applications in Kubernetes and serverless applications. You designed and deployed a Jenkins cluster on the cloud for scale and mastered monitoring and troubleshooting Jenkins.

Technology changes quickly, so it's great to have a few resources to go to for recent news and information. The weekly newsletter DevOps Bulletin (https://devopsbulletin.com) features a great collection of posts regarding PaC and the latest wonders in the DevOps space. I also recommend keeping an eye on DevOps World (www.devopsworld.com), where you can be inspired by experts and your peers and gain the tools you need to shape the future of software delivery at your organization and at large.

I hope you've enjoyed the book and learned something from it. PaC is still new, but awareness is growing rapidly. Over the next few years, you'll see many organizations, small and large, embrace PaC to release faster and reduce the feedback loop.