```
            publishHTML (target: [
                allowMissing: false,
                alwaysLinkToLastBuild: false,
                keepAll: true,
                reportDir: "$PWD/coverage/marketplace",
                reportFiles: "index.html",
                reportName: "Coverage Report"
            ])
        }
        stage('Static Code Analysis'){
            withSonarQubeEnv('sonarqube') {
                sh 'sonar-scanner'
            }
        }
        stage("Quality Gate"){
            timeout(time: 5, unit: 'MINUTES') {
                def qg = waitForQualityGate()
                if (qg.status != 'OK') {
                    error "Pipeline aborted due to
quality gate failure: ${qg.status}"
                }
            }
        }
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
    }
}
```

Consumes the coverage report with the Jenkins Publish HTML plugin

Runs code-quality inspection with SonarQube

Interrupts SonarQube inspection if it takes more than 5 minutes

Add a `Build` stage to create a Docker container to install the npm dependencies and copy the dependencies folder as well as the generated static web application files to the current workspace, as shown in the following listing. Note the use of the `--build-arg` argument to inject the API Gateway URL at the build time.

Listing 12.15 Building the Angular application

```
stage('Build'){
            sh """
                docker build -t ${imageName} --build-arg ENVIRONMENT=sandbox .
                containerName=\$(docker run -d ${imageName})
                docker cp \$containerName:/app/dist dist
                docker rm -f \$containerName
            """
 }
```

Then, in the following listing, use the AWS CLI to push the generated static web application to the S3 bucket where website hosting is enabled.

Listing 12.16 Storing the Angular static application to S3

```
stage('Push'){
    sh "aws s3 cp --recursive dist/ s3://${bucket}/"
}
```

Recursively copies local files to S3

Push the changes to the develop branch. A new pipeline should be triggered, and the stages in figure 12.21 will be executed successfully.

Figure 12.21 Marketplace CI/CD workflow

You can verify that the files have been successfully stored from the Amazon S3 bucket dashboard, or by running the `aws s3 ls` command in your terminal session. Figure 12.22 shows the content of the marketplace S3 bucket.

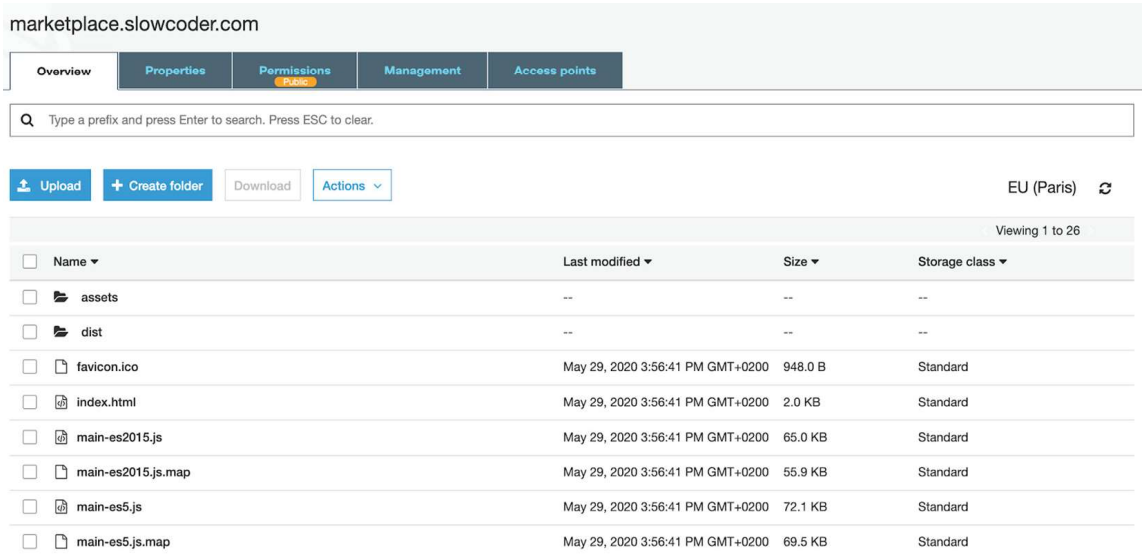

Figure 12.22 Marketplace S3 bucket content

If you head to the S3 website URL (http://BUCKET.s3-website-REGION.amazonaws.com/), it should display the marketplace UI, shown in figure 12.23.

That's great! However, when you're building your serverless application, you must separate your deployment environments to test new changes without impacting your production. Therefore, having multiple environments makes sense while building serverless applications.

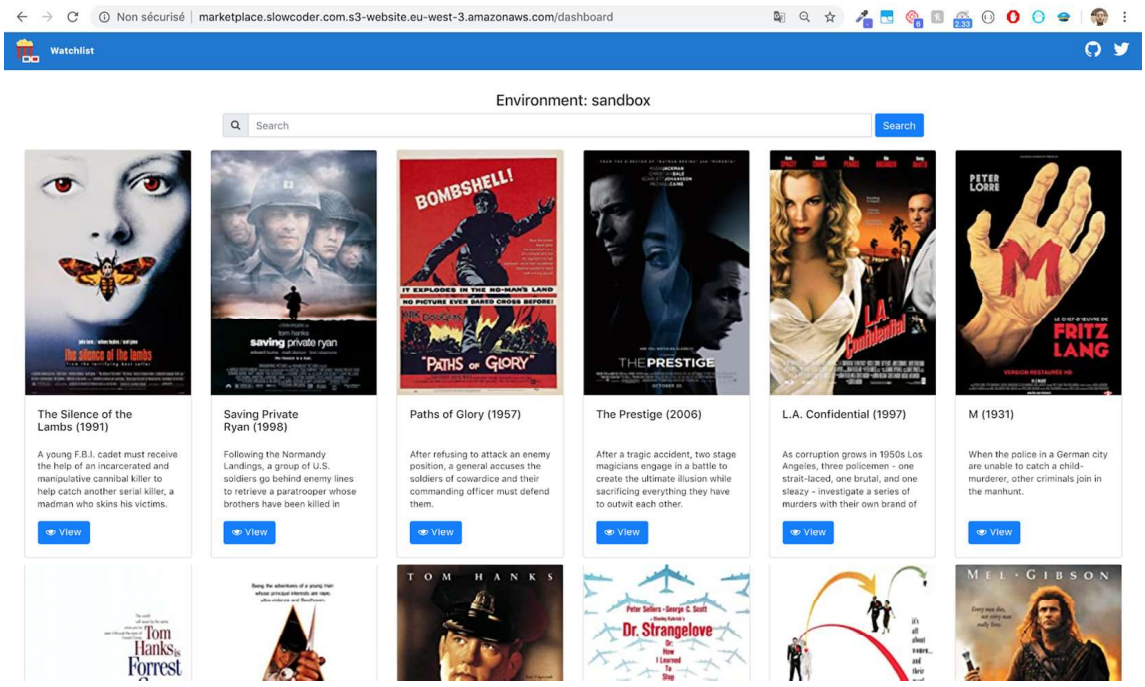

Figure 12.23 Marketplace dashboard running in the sandbox environment

12.5 Maintaining multiple Lambda environments

AWS Lambda allows you to publish a version, which represents the state of the function's code and configuration in time. By default, each Lambda function has the `$LATEST` version pointing to the latest changes deployed to the function.

To publish a new version from the `$LATEST` version, update the Jenkinsfile (chapter12/functions/movies-loader/Jenkinsfile) to add a new stage to publish a new Lambda function's version, as shown in the following listing.

Listing 12.17 Publishing a new Lambda version

```
stage('Deploy'){
    sh "aws lambda update-function-code --function-name ${functionName}
            --s3-bucket ${bucket} --s3-key ${functionName}/${commitID()}.zip
            --region ${region}"

    sh "aws lambda publish-version --function-name ${functionName}
            --description ${commitID()} --region ${region}"
}
```

When you publish a new version of your Lambda function, you should give it a meaningful version name that allows you to track different changes made to your function through its development cycle. In listing 12.17, we're using the Git commit ID as a version scheme. However, you can use an advanced version mechanism like semantic versioning (https://semver.org/).

When the pipeline is executed, at the `Deploy` stage the preceding commands will be executed. Figure 12.24 shows their execution logs.

Figure 12.24 Update and Publish commands executed within the deploy stage

NOTE Versions are immutable: once they're created, you cannot update their code or settings (memory, execution time, VPC config, and so forth).

On the MoviesLoader Lambda dashboard, a new version will be published based on the develop branch source code, as shown in figure 12.25.

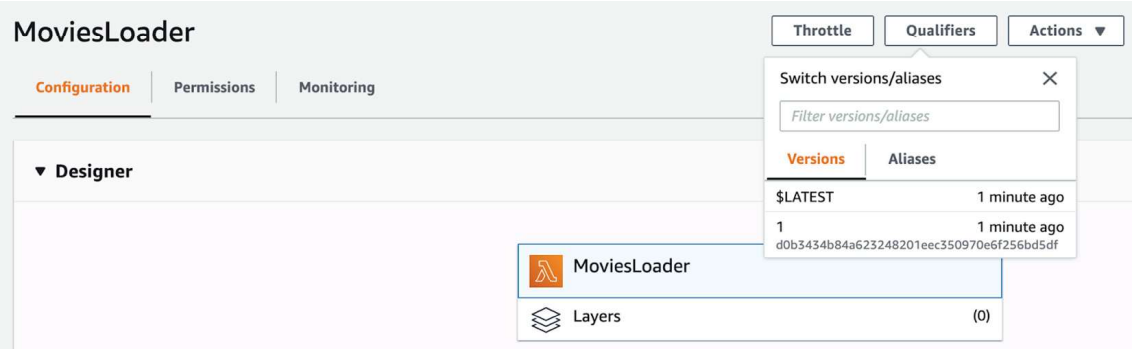

Figure 12.25 MoviesLoader Lambda new published version

The publication of Lambda versions for the MoviesStore API will be done in parallel to reduce the execution time of the pipeline; see figure 12.26.

As a result, you can work with different variations of your Lambda function in your development workflow.

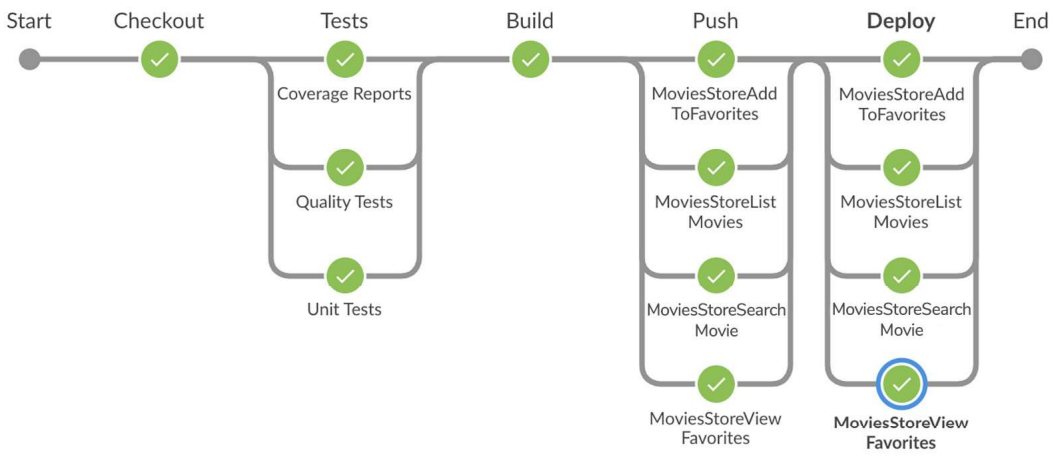

Figure 12.26 Running the publish command in parallel

For now, API Gateway triggers the `MoviesStore` Lambda functions based on the `$LATEST` version, so every time a new version is published, we need to update API Gateway to point to the newest version (figure 12.27)—a tedious and not handy task.

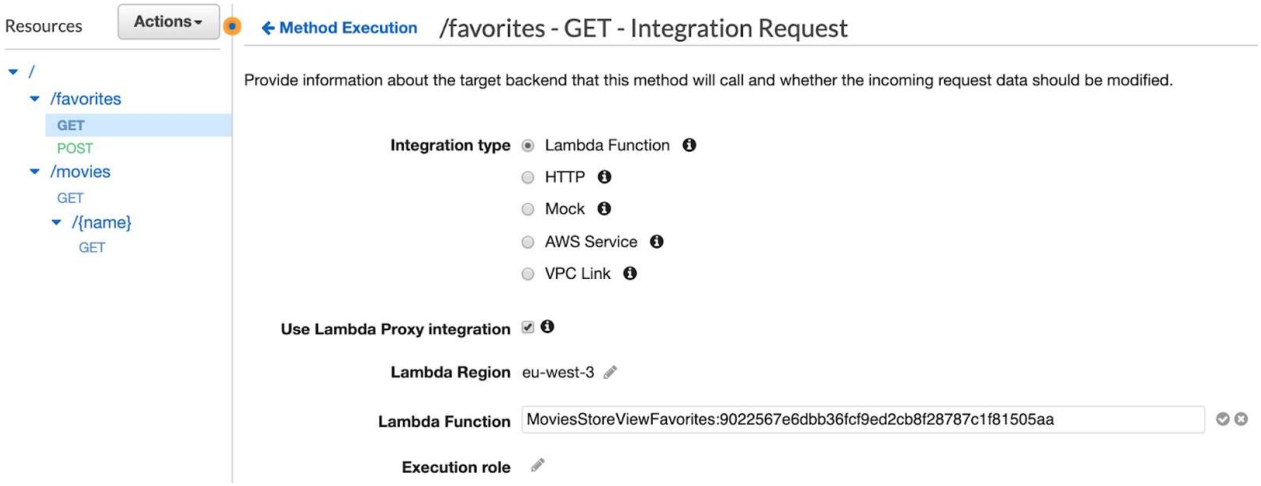

Figure 12.27 GET /favorites integration request

Fortunately, there's the concept of a *Lambda alias*. The alias, a pointer to a specific version, allows you to promote a function from one environment to another (such as staging to production). Aliases are mutable, unlike immutable versions. Now, instead of directly assigning a Lambda function version in an API Gateway integration request, you can assign Lambda alias, where the alias is a variable. The variable will be resolved from a value during runtime.

That being said, create an alias for the sandbox, staging, and production environments that points to the latest version published by using the AWS command line:

```
aws lambda create-alias --function-name MoviesStoreViewFavorites --name
     sandbox    --version 1
```

Once created, the new aliases should be added to the list of Aliases under the Qualifiers drop-down list (figure 12.28).

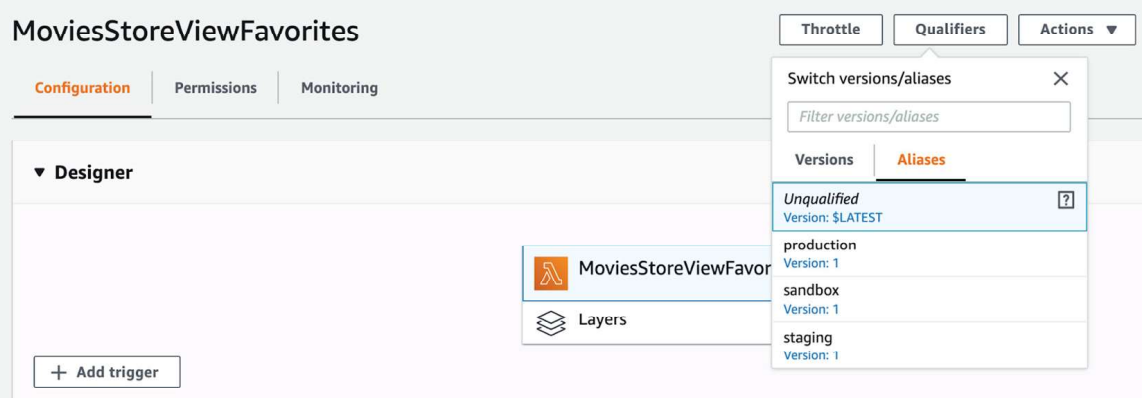

Figure 12.28 Using multiple aliases to reference different environments

We can update the Jenkinsfile to update the alias directly. Update the `Deploy` stage with the code in the next listing. It updates the Lambda function code, publishes a new version, and then points the alias corresponding to the current Git branch (master branch = production alias, preprod branch = staging alias, develop branch = sandbox alias) to the newly deployed version.

Listing 12.18 Updating the Lambda alias to point to the newest version

```
sh "aws lambda update-function-code --function-name ${it}
        --s3-bucket ${bucket} --s3-key ${it}/${fileName}.zip
        --region ${region}"

def version = sh(
    script: "aws lambda publish-version --function-name ${it}
                --description ${fileName}
--region ${region} | jq -r '.Version'",
    returnStdout: true
).trim()

if (env.BRANCH_NAME in ['master','preprod','develop']){
    sh "aws lambda update-alias  --function-name ${it}
            --name ${environments[env.BRANCH_NAME]}
--function-version ${version}
            --region ${region}"
}
```

The `publish-version` operation returns JSON output with the deployed version number as an attribute. The `jq` command is used to parse the `Version` attribute and store its value in a `version` variable. Then, based on the current Git branch, the corresponding alias will point to the published version number.

Push the changes to the develop branch. The function code will be updated, a new version will be created, and the sandbox alias will point to the newest published version, as you can see in figure 12.29.

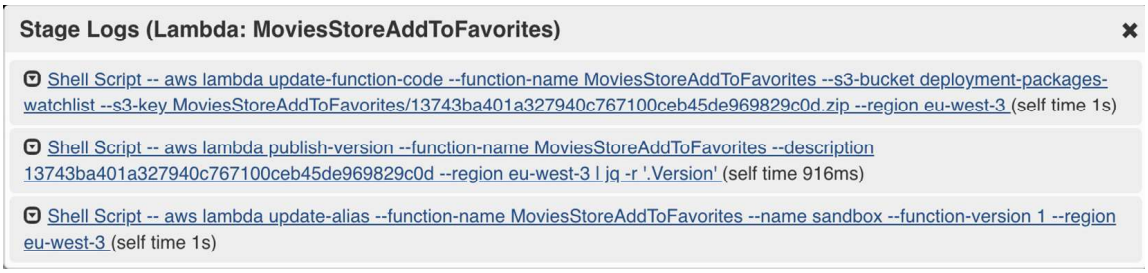

Figure 12.29 Updating the Lambda alias to the deployed version

On the `MoviesStoreListMovies` Lambda, for instance, the sandbox alias should point to the version with the develop branch source code, as shown in figure 12.30.

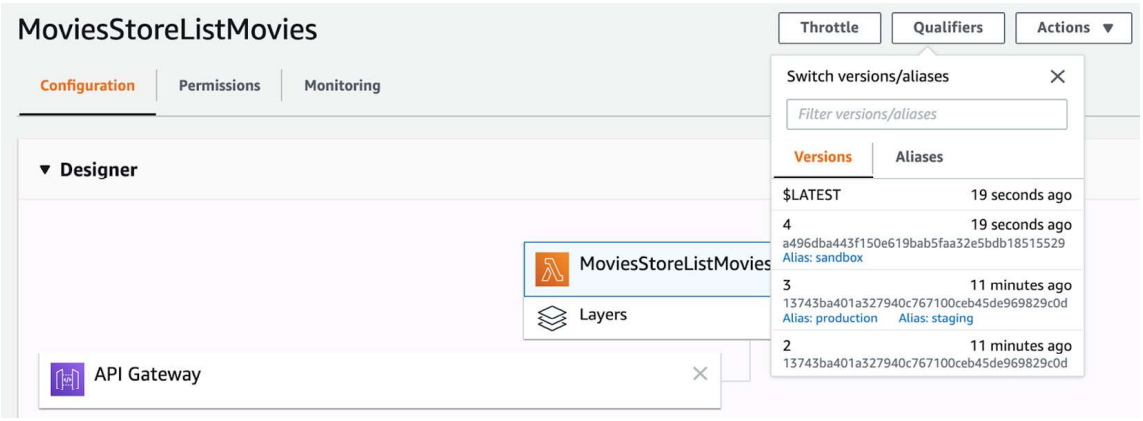

Figure 12.30 Sandbox alias pointing to the new Lambda version

Now that you have seen how to create aliases and switch their values within a Jenkins pipeline, let's configure the API Gateway to use these aliases with stage variables.

Stage variables are environment variables that can be used to change the behavior at runtime of the API Gateway methods for each deployment stage.

On the API Gateway Console, navigate to the Movies API, click the GET method for the instance, and update the target Lambda function to use a stage variable instead of a hardcoded Lambda function version, as shown in figure 12.31.

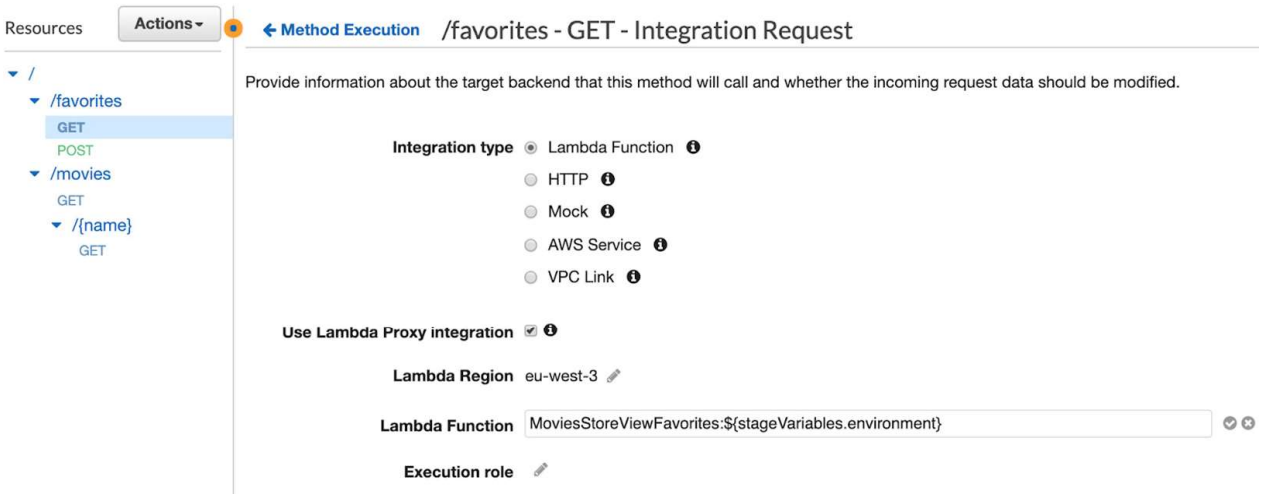

Figure 12.31 Using a stage variable while configuring the API integration request

In the Lambda Function field, the `${stageVariables.environment}` tells API Gateway to read the value for this field from a stage variable at runtime.

When you save the configuration, a new prompt will ask you to grant the permissions to API Gateway to call your Lambda function aliases. At this point, we need to deploy our API to make it publicly available.

From the Actions drop-down, select Deploy API. Choose the New Deployment Stage option, enter `sandbox` as a stage name, and deploy it. Or use the Terraform code in listing 12.19. The sandbox stage will set the environment stage variable to `sandbox`. As a result, if a user invokes an HTTP request on any endpoint of the sandbox deployment, the corresponding Lambda function with the sandbox alias will be triggered.

Listing 12.19 API Deployment with an alias stage variable

```
resource "aws_api_gateway_deployment" "sandbox" {
   depends_on = [
     module.GetMovies,
     module.GetOneMovie,
     module.GetFavorites,
     module.PostFavorites
   ]

   variables = {
    "environment" = "sandbox"
  }
```

```
   rest_api_id = aws_api_gateway_rest_api.api.id
   stage_name  = "sandbox"
}
```

Create additional deployment stages for staging and production environments. On completion of the `terraform apply` command, the three deployment stage URLs will be displayed, as shown in figure 12.32.

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

api = https://rth65vizrb.execute-api.eu-west-3.amazonaws.com/test
marketplace = marketplace.slowcoder.com.s3-website.eu-west-3.amazonaws.com
production = https://rth65vizrb.execute-api.eu-west-3.amazonaws.com/production
sandbox = https://rth65vizrb.execute-api.eu-west-3.amazonaws.com/sandbox
staging = https://rth65vizrb.execute-api.eu-west-3.amazonaws.com/staging
```

Figure 12.32 API Gateway deployment URLs

If you open the API at https://id.execute-api.region.amazonaws.com/sandbox/movies, you will get the response from Lambda `MoviesStoreListMovies` with the alias `sandbox`.

To deploy the serverless application to the staging environment, create a pull request to merge the develop branch to the preprod branch. Jenkins will post the build status of the develop job on the PR (figure 12.33). Then, merge develop to preprod.

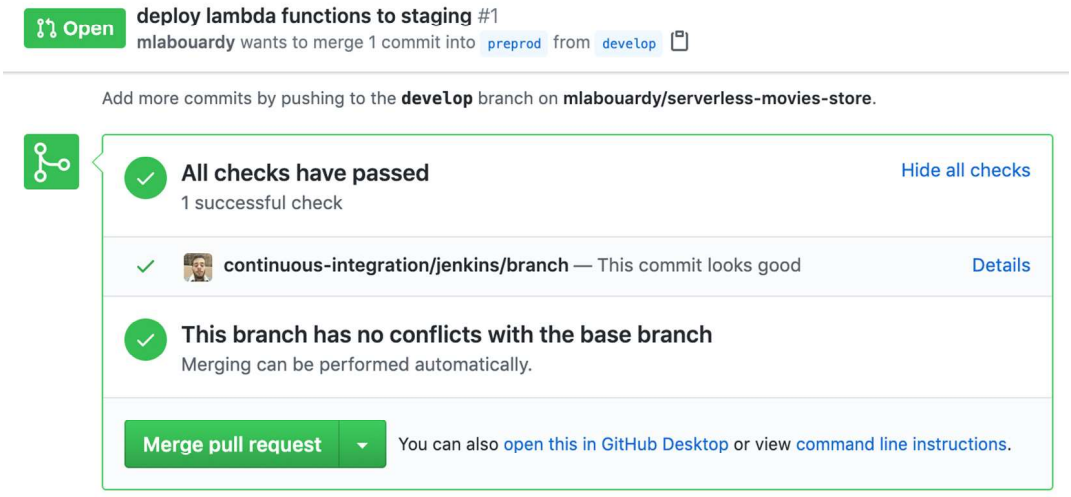

Figure 12.33 Jenkins post build status on GitHub PR

Once the PR is merged, a new build will be triggered on the preprod branch. At the end of the CI/CD pipeline, the staging alias will point to the newly deployed version, as you can see in figure 12.34.

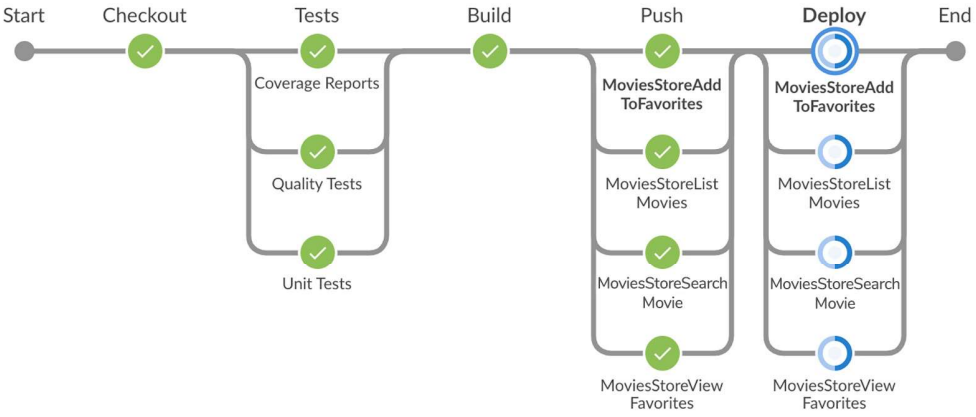

Figure 12.34 Deploying Lambda functions to staging

Now, to deploy the marketplace on multiple environments, we will inject the environment name based on the current branch name; see the following listing.

Listing 12.20 Injecting the environment name during the build

```
stage('Build'){
  sh """
     docker build -t ${imageName}
--build-arg ENVIRONMENT=${environments[env.BRANCH_NAME]} .
     containerName=\$(docker run -d ${imageName})
     docker cp \$containerName:/app/dist dist
     docker rm -f \$containerName
  """
}
```

Then, in listing 12.21, we update the `aws s3 cp` instruction to push the static files to a folder named as the environment name under the S3 bucket. You can also create an S3 bucket per environment, but for simplicity, we use a single S3 to store different environments of the marketplace.

Listing 12.21 Pushing static files to an S3 bucket

```
if (env.BRANCH_NAME in ['master','preprod','develop']){
  stage('Push'){
   sh "aws s3 cp --recursive dist/ s3://${bucket}/
     ${environments[env.BRANCH_NAME]}/"
  }
}
```

Push these changes to a feature branch. Then raise a pull request to merge to the develop branch. When the merge occurs, the new pipeline in figure 12.35 will be executed.

Figure 12.35 Marketplace new CI/CD pipeline

Merge the changes to preprod to deploy the application to staging. Then, merge from preprod to master branch for production deployment. As a result, the S3 bucket should contain three folders. Each folder holds a different runtime environment of the marketplace, as you can see in figure 12.36.

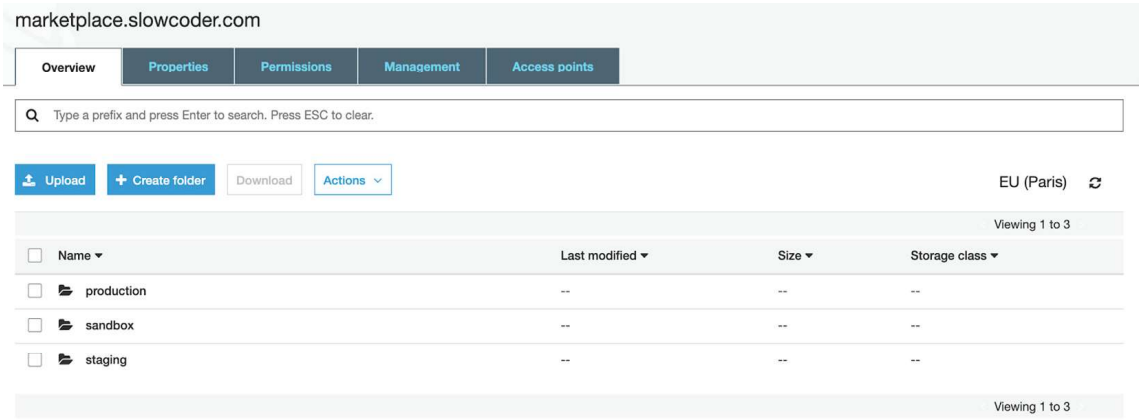

Figure 12.36 S3 bucket with multiple environments

If you point to the S3 bucket website URL and add the /staging endpoint, it should serve the staging environment of the marketplace, as shown in figure 12.37.

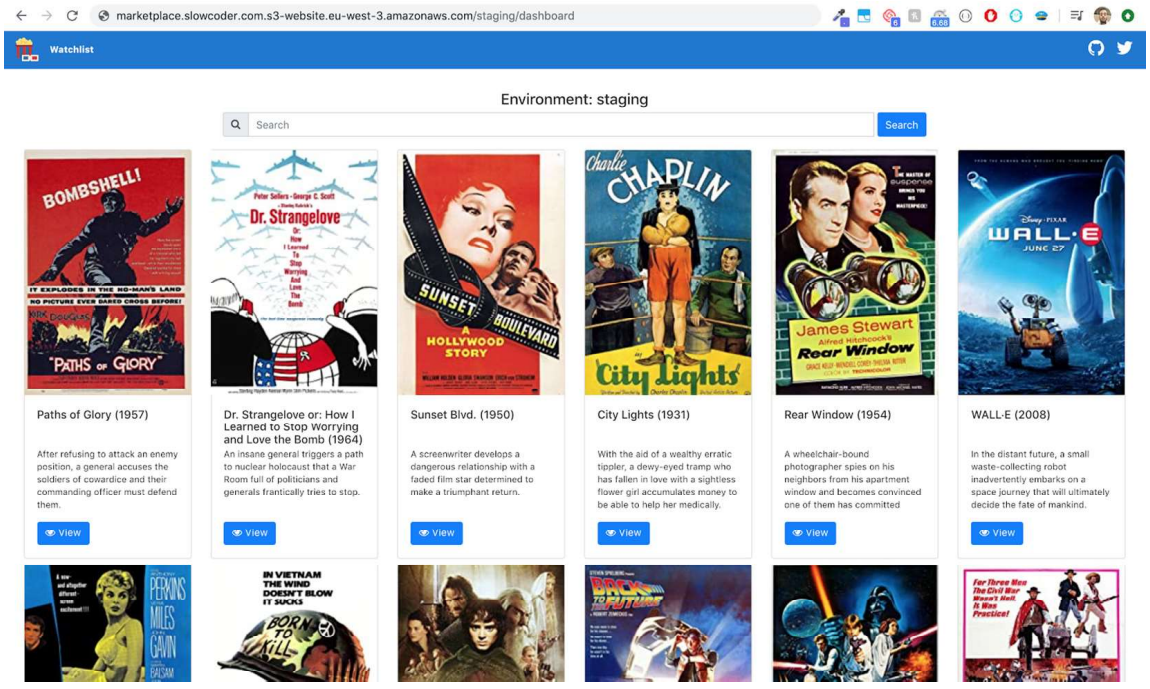

Figure 12.37 Marketplace staging environment

Now, to deploy the Lambda functions to production, merge the preprod branch to the master branch by raising a pull request, as shown in figure 12.38.

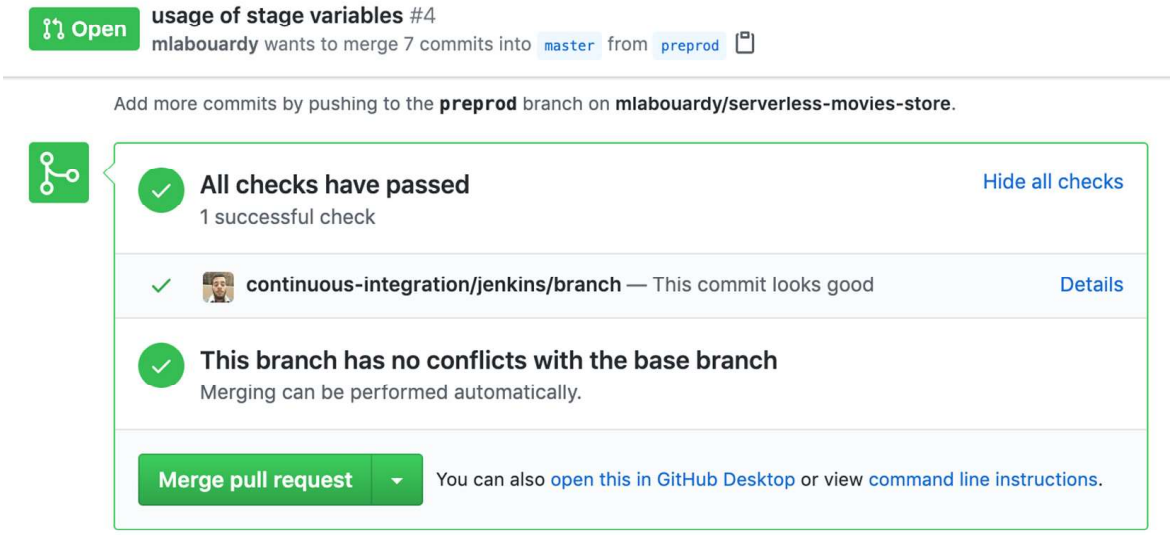

Figure 12.38 Merging the movies-store Lambda functions' preprod branch to master

When the merge occurs, the pipeline will be triggered on the master branch; see figure 12.39.

Figure 12.39 Deploying Lambda functions to production

The movies-store functions will be updated, a new version will be created, and the production alias will point to the newly deployed version.

You can take this further and ask for developer authorization before actual deployment to production by using the Jenkins Input Step plugin; see the following listing. When the `Deploy` stage is reached, an input dialog will pop up for deployment confirmation.

Listing 12.22 Asking for user approval before production deployment

```
if (env.BRANCH_NAME == 'preprod' || env.BRANCH_NAME == 'develop'){
   sh "aws lambda update-alias  --function-name ${it}
            --name ${environments[env.BRANCH_NAME]}
--function-version ${version}
            --region ${region}"
}

if(env.BRANCH_NAME == 'master'){
   timeout(time: 2, unit: "HOURS") {
        input message: "Deploy to production?", ok: "Yes"
   }
   sh "aws lambda update-alias  --function-name ${it}
            --name ${environments[env.BRANCH_NAME]}
--function-version ${version}
            --region ${region}"
}
```

The interactive input will ask whether we approve the deployment. If we click Yes, the pipeline will be resumed, and the production alias will point to the newly deployed version, as shown in figure 12.40.

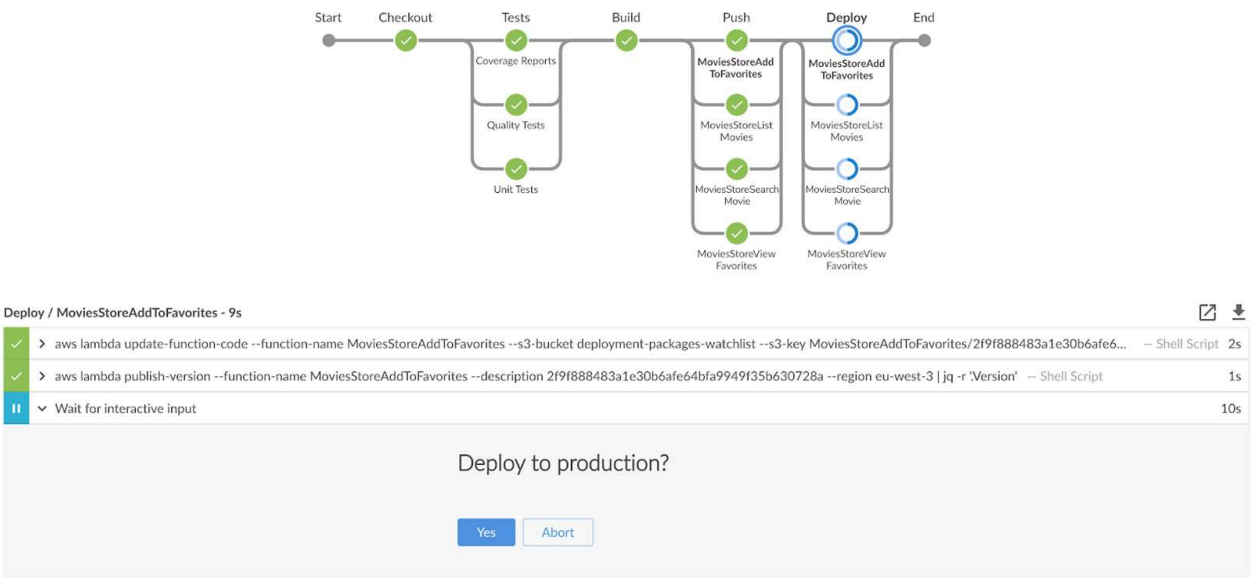

Figure 12.40 Production deployment confirmation within the Jenkins pipeline

So now if we make any change to our serverless application, CI/CD pipelines will be triggered, and the newly published Lambda function code will be promoted to production. A Slack notification will also be sent with the deployment job status, as shown in figure 12.41.

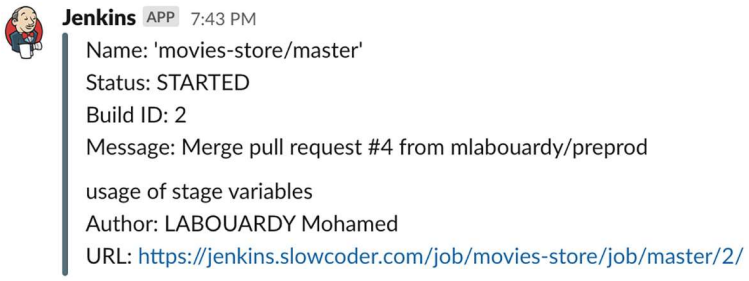

Figure 12.41 Production deployment Slack notification

Sending notifications on pipeline triggering and progress helps to communicate the work among team members. So far, we have used it to send start, completed, and failure notifications. but Slack can also be used to take actions or execute commands from the chat window to confirm the production deployment, for instance, or trigger the build of a Jenkins job.

Another way of raising awareness of job build status and reporting testing results is through email notifications.

12.6 Configuring email notification in Jenkins

Email notification within Jenkins can be done with the help of an Email Extension plugin (https://plugins.jenkins.io/email-ext/). This plugin comes with a list of essentials plugins installed on Jenkins.

To enable email notification, you need to configure an SMTP server. Go to Manage Jenkins, then Configure System. Scroll down to the Extended E-mail Notification section. Enter your SMTP credentials, if you're using Gmail, and then type `smtp.gmail.com` for the SMTP server and enter your Gmail username and password. Select the use of SSL and enter the port number as 465.

To be able to send an email, you need to configure a list of recipient addresses. Next, click the Apply and Save buttons, as shown in figure 12.42.

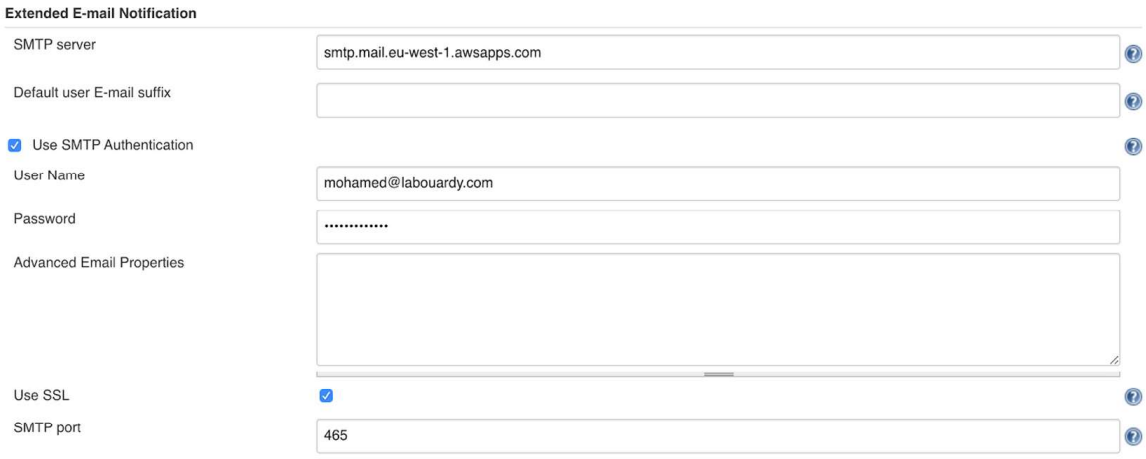

Figure 12.42 Extended email notification configuration

You can test configurations by entering the recipient email address and clicking Test Configuration. If all is good, you will see the message `Email sent successfully`.

Now that the plugin is configured, type the following listing in your Jenkinsfile to define a function responsible for sending an email with customizable attributes based on the job build status.

Listing 12.23 Sending email to report job build status

```
def sendEmail(String buildStatus){
    buildStatus =  buildStatus ?: 'SUCCESSFUL'
    emailext body: "More info at: ${env.BUILD_URL}",
             subject: "Name: '${env.JOB_NAME}' Status: ${buildStatus}",
             to: '$DEFAULT_RECIPIENTS'
}
```

Finally, you can invoke the function upon the completion of the CI/CD pipeline by calling the `sendEmail()` method on the `finally` block. In the following listing, an email notification is sent only if a build is running on the master branch to avoid spamming developers.

Listing 12.24 Sending email when a production deployment is happening

```
node('workers'){
    try {
        stage('Checkout'){...}
        stage('Tests'){...}
        stage('Build'){...}
        stage('Push'){...}
        stage('Deploy'){...}
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)

        if (env.BRANCH_NAME == 'master'){
            sendEmail(currentBuild.result)
        }
    }
}
```

Push the new Jenkinsfile to GitHub. When a build is occurring on the master branch, an email will be sent. Once the pipeline is finished, you should be able to see an email like the one in figure 12.43.

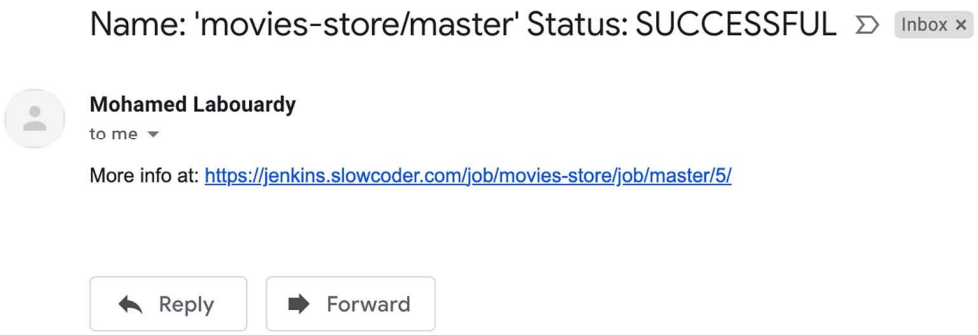

Figure 12.43 Email notification reporting job build status

The email's subject contains the name of the Jenkins job as well as its build status. The email's body has a link to the job output.

The declarative approach of writing Jenkinsfiles provides a `post` section, which can be used to place post-execution scripts. You can invoke the `sendEmail()` method by placing it in the `post` build section, as shown in the following listing.

Listing 12.25 Post steps in Jenkins declarative pipeline

```
pipeline {
    agent{
        label 'workers'
    }
    stages {
        stage('Checkout'){...}
        stage('Unit Tests'){...}
        stage('Build'){...}
        stage('Push'){...}
    }
    post {
        always {
            if (env.BRANCH_NAME == 'master'){
                sendEmail(currentBuild.currentResult)
            }
        }
    }
}
```

You can also attach the job build logs by enabling the `attachLog` attribute with the following listing.

Listing 12.26 Attaching log files in a notification mail

```
def sendEmail(String buildStatus){
    buildStatus =  buildStatus ?: 'SUCCESSFUL'
    emailext body: "More info at: ${env.BUILD_URL}",
             subject: "Name: '${env.JOB_NAME}' Status: ${buildStatus}",
             to: '$DEFAULT_RECIPIENTS',
             attachLog: true
}
```

As a result, email sent by Jenkins will now contain the job status as well the full console output as an attachment, as shown in figure 12.44.

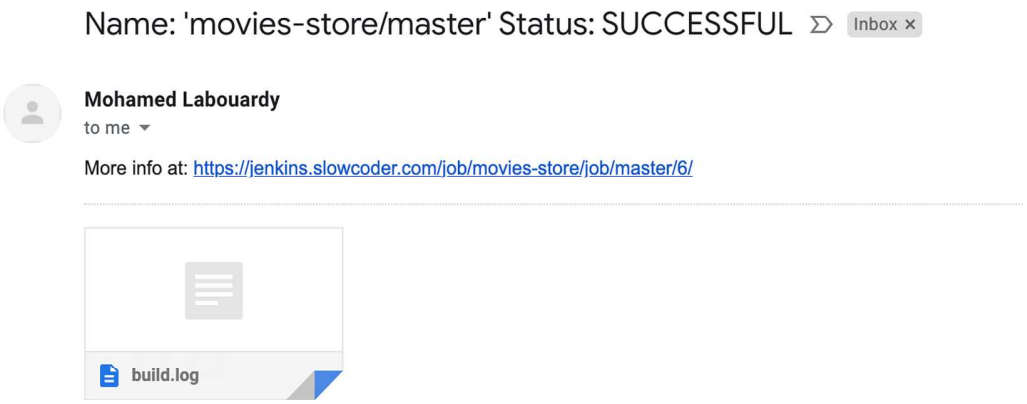

Figure 12.44 Sending job logs as an email notification attachment

Summary

- Terraform modules allow you to better organize your infrastructure configuration code and make the resources reusable.
- When building a serverless application as a collection of Lambda functions, you need to decide whether you're going to push each function individually to its own Git repository, or bundle them all together as a single repo.
- AWS Lambda supports aliases, which are named pointers to a particular version. This makes it easy to use a single Lambda function for sandbox, staging, and production environments.
- The API Gateway stage variable feature enables you to dynamically access different Lambda function environments.
- The Email Extension plugin allows you to configure every aspect of email notifications. You can customize when an email is sent, who should receive it, and what the email says.

Part 4

Managing, scaling, and monitoring Jenkins

This final part is about combining and coalescing everything you've learned and moving even further. You'll learn how to monitor and troubleshoot a running Jenkins cluster. We'll start by exposing Jenkins metrics with Prometheus and build an interactive dashboard with Grafana. Next, I will demonstrate how to stream Jenkins logs to a centralized logging platform based on the Elastic-Search, Logstash, and Kibana (ELK) stack. Finally, I will share tips and best practices to secure and maintain Jenkins.

13 Collecting continuous delivery metrics

This chapter covers

- Monitoring Jenkins and its jobs effectively
- Forwarding Jenkins build logs to a centralized logging platform
- Parsing Jenkins logs into something structured and queryable
- Exposing Jenkins internal metrics with Prometheus
- Building interactive dashboards with Grafana
- Creating metric-based alerts for Jenkins

In the previous chapters, you learned to design, build, and deploy a Jenkins cluster from scratch by using automation tools; you also learned to set up a fully working CI/CD pipeline for several cloud-native applications. In this chapter, we will dive into advanced Jenkins topics: monitoring a running Jenkins server and detecting anomalies and resource starvation. Along the way, we will cover how to build a centralized logging platform for Jenkins logs.

13.1 Monitoring Jenkins cluster health

The cluster we built in chapter 5 consists of a Jenkins master and workers, with each node running inside an EC2 instance. Figure 13.1 shows a typical Jenkins node configuration.

S	Name ↓	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	ip-10-0-0-29.eu-west-3.compute.internal	Linux (amd64)	In sync	27.82 GB	0 B	27.82 GB	23ms
	ip-10-0-2-216.eu-west-3.compute.internal	Linux (amd64)	In sync	27.82 GB	0 B	27.82 GB	34ms
	master	Linux (amd64)	In sync	27.46 GB	0 B	27.46 GB	0ms
	Data obtained	30 ms	31 ms	32 ms	30 ms	24 ms	26 ms

Figure 13.1 Jenkins distributed build architecture

So far, the Jenkins cluster is working as expected. However, you should never take your IT infrastructure for granted. Your Jenkins master or workers one day will break and will need to be replaced. So, how do you know if your Jenkins cluster is working effectively if you aren't monitoring it?

Monitoring Jenkins should become a crucial part of your IT management. Monitoring helps you look for abnormalities and spot issues on instances running the cluster, saves you money as it minimizes the network downtime, and enhances efficiency.

In AWS, you can monitor Jenkins instances by using Amazon CloudWatch (https://aws.amazon.com/cloudwatch). The platform consumes data coming from all AWS services and allows the user to visualize, query, and take action on the data. By default, Amazon EC2 sends metrics data to CloudWatch.

> **NOTE** You can use Azure Monitor (http://mng.bz/wQYQ) or Google Cloud's operations (https://cloud.google.com/monitoring/quickstart-lamp) if you want to monitor the overall health and performance of Jenkins instances running in Azure or GCP environments.

Navigate to the Amazon CloudWatch console and jump to the All Metrics tab. Then, under EC2, look for instances running the cluster by typing their instance ID on the search bar, as shown in figure 13.2.

Figure 13.2 Key metrics for EC2 monitoring

You will see a pretty long list of reported metrics for your Jenkins EC2 instances. You can scroll and select one or more metrics to display (for example, EC2 instance CPU utilization) and create a graph widget to display them, as shown in figure 13.3.

Figure 13.3 The percentage of allocated EC2 compute units currently in use on the Jenkins instances

By default, EC2 reports metrics to CloudWatch in 5-minute intervals. However, if your Jenkins cluster is being extensively used (for example, hosting multiple jobs and scheduling many CI/CD pipelines), you can enable the enhanced monitoring feature (http://mng.bz/GOZR) on each instance to get metrics in 1-minute intervals (though an additional cost applies).

CloudWatch also offers dashboards, which provide a quick view of how your instances are performing as well as tremendous flexibility in terms of data visualization—for example, zooming in or rescaling.

You can customize the dashboard and add additional graphs showing, for example, the number of bytes received and sent out on all network interfaces, or disk usage (bytes written and read from all instance store volumes), as demonstrated in figure 13.4.

Now you know how to monitor Jenkins instances using CloudWatch. However, it can be error-prone and tedious to set up CloudWatch monitoring for all your Jenkins instances (and remembering to do it for Jenkins workers created for scaling events). Additionally, some metrics are unavailable through CloudWatch (such as memory usage). Hence, we will use an advanced monitoring stack.

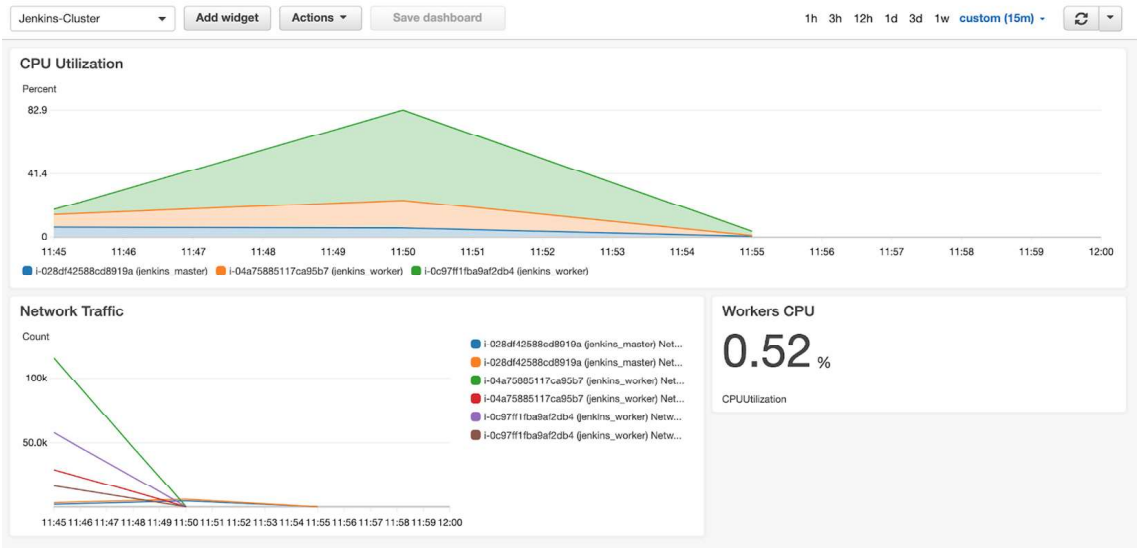

Figure 13.4 Building the CloudWatch dashboard to monitor Jenkins instances

> **NOTE** The Amazon CloudWatch agent can be installed on EC2 instances to report additional and useful metrics. This feature is seldom used, but it is good to know it exists. Refer to the official guide at http://mng.bz/q5J2 for instructions.

Many tools, from open source to a commercial level, can help you monitor your infrastructure and notify you of any failure. (Section 13.3 covers how to set up alerts that will notify you in near real-time.) The good thing is that a powerful open source monitoring solution is available, thanks to the open source community that maintains it. Figure 13.5 summarizes the open source solution we're going to implement.

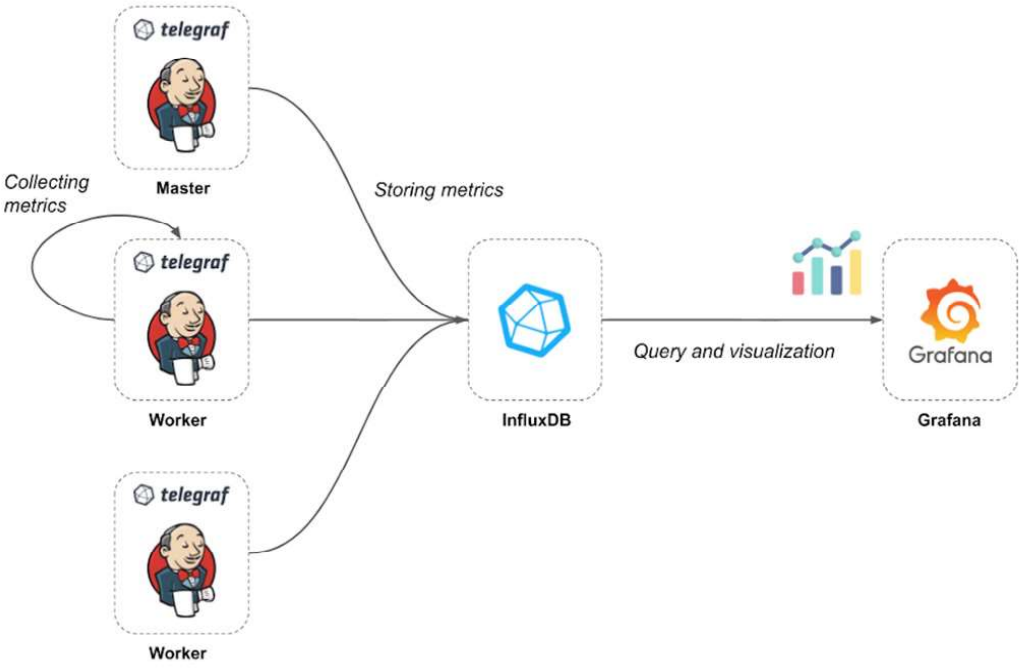

Figure 13.5 Telegraf will collect metrics, store them in InfluxDB, and from there we can visualize them in Grafana.

This monitoring solution can be split into three parts:

- *Telegraf*—A metric collector agent, installed on each Jenkins instance. It collects the internal metrics and ships them to a time-series database.
- *InfluxDB*—An open source time-series database (TSDB), optimized for fast, high-availability storage. It consumes the telemetry coming from Telegraf agents.
- *Grafana*—An open source visualization platform, used to build dynamic and interactive dashboards based on data stored in InfluxDB.

Now that the architecture is clear, we need to deploy an InfluxDB server on an EC2 instance. Check out the InfluxDB official documentation at http://mng.bz/7lJy for a step-by-step guide on how to install and configure InfluxDB.

Once the instance is up and running, SSH to the InfluxDB instance and type the `influx` command on the terminal. The `influx` CLI, which is included in all InfluxDB packages, is a lightweight and simple way to interact with the database. We need to create two databases:

- *instances*—To store metrics about resource usage, such as CPU utilization, memory, network traffic, disk usage, and so forth.
- *containers*—To store metrics about containers running in the Jenkins workers. The containers are basically build jobs scheduled for Jenkins workers.

Create the databases by using the `CREATE DATABASE` Influx Query Language (InfluxQL) statement:

```
CREATE DATABASE containers;
CREATE DATABASE instances;
```

The databases can also be created by making raw HTTP requests to an InfluxDB API over port 8086 (see http://mng.bz/m1z2).

Now that we have databases, InfluxDB is ready to accept queries and writes. To collect Jenkins instance metrics, we need to install a Telegraf agent on each server. One way to do this is to install Telegraf on the existing instances, but this solution won't scale, as we need to install and configure a Telegraf agent each time a new Jenkins worker is deployed. Therefore, the best way is to ship Telegraf within the Jenkins AMI. Once again, we will use Packer to bake the Jenkins master and worker AMIs with a preinstalled and configured Telegraf agent.

Add the code in the next listing to the setup.sh (chapter13/telegraf/setup.sh) script provided in chapter 4, listings 4.4 and 4.5. This code will install the latest stable version of Telegraf (at the time of writing this book, version 1.19.0 is available).

Listing 13.1 Installing the Telegraf agent with the Yum utility

```
wget https://dl.influxdata.com/telegraf/releases/telegraf-1.19.0-1.x86_64.rpm
yum localinstall telegraf-1.19.0-1.x86_64.rpm
systemctl enable telegraf
systemctl restart telegraf
```

Next, we tell Telegraf what metrics to collect, by creating a configuration file at /etc/telegraf/telegraf.conf. The config file consists of *inputs* (where the metrics come from) and *outputs* (where the metrics go). The following listing specifies three inputs (CPU memory usage, and Docker), and specifies InfluxDB as the output. The Docker input reads metrics about the Docker daemon and then outputs this data to InfluxDB.

Listing 13.2 Telegraf configuration file with various inputs

```
[global_tags]
hostname="Jenkins"

[[inputs.cpu]]
  percpu = false
```

Overrides default hostname; if empty, use os.Hostname()

Gathers metrics on the system CPUs

```
  totalcpu = true
  fieldpass = [ "usage*" ]
  name_suffix = "_vm"

[[inputs.disk]]
  fielddrop = [ "inodes*" ]
  Mount_points = ["/"]
  name_suffix = "_vm"

[[inputs.mem]]
  name_suffix = "_vm"

[[inputs.swap]]
  name_suffix = "_vm"

[[inputs.system]]
  name_suffix = "_vm"

[[inputs.docker]]
  endpoint = "unix:///var/run/docker.sock"
  container_names = []
  name_suffix = "_docker"

[[outputs.influxdb]]
  database = "instances"
  urls = ["http://INFLUXDB_IP:8086"]
  namepass = ["*_vm"]

[[outputs.influxdb]]
  database = "containers"
  urls = ["http://INFLUXDB_IP:8086"]
  namepass = ["*_docker"]
```

Gathers metrics about disk usage. By default, stats are gathered for all mount points, and setting Mount_points will restrict the stats to the root volume.

Collects system memory metrics

Collects system swap metrics

Gathers general stats on system load, uptime, and number of users logged in. It is similar to the Unix uptime command.

Uses the Docker Engine API to gather metrics on running Docker containers

Writes system metrics to the InfluxDB instance database

Writes Docker metrics to the InfluxDB container database

Make sure to replace the `INFLUXDB_IP` variable with the IP address of the instance where the InfluxDB server is running.

Bake a new Jenkins AMI and redeploy a Jenkins cluster with the newly built image by following steps described in section 5.3. Once the new Jenkins cluster is up and running, Telegraf will start collecting metrics and streaming them to InfluxDB for storage and indexing.

To explore the metrics, we will use Grafana. You can install Grafana from a Yum repository or by running a Docker image. (Check out the Grafana official documentation at http://mng.bz/5ZY1 for more details.) Once Grafana is installed, head your browser to HOST_IP:3000. On the login page, enter `admin` for the username and password.

Before we create a dashboard to monitor the overall health of the Jenkins instances, we need to link the InfluxDB databases to Grafana. To do so, we need to create a data source for each InfluxDB database.

In the side panel, click the cog icon and then click Configuration > Data Sources. Click the Add Data Source button, shown in figure 13.6. Then fill the settings page with the following values:

- *Name*—The data source name. (This is how you'll refer to the data source in queries.)

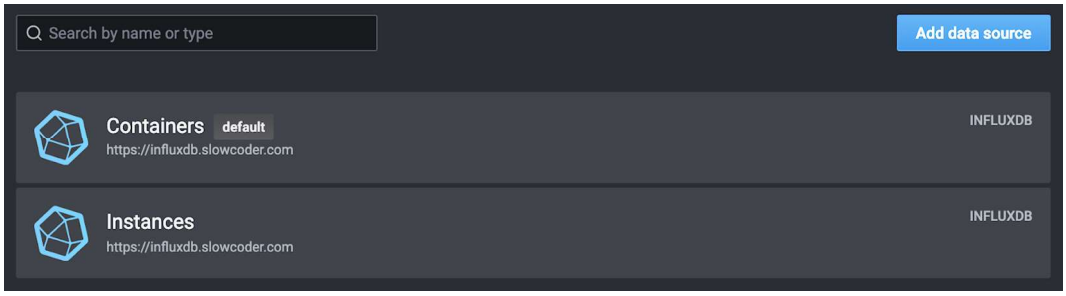

Figure 13.6 Configuring InfluxDB-based data sources in Grafana

- *URL*—The HTTP, IP address, and port of your InfluxDB API. (By default, the InfluxDB API port is 8086.)
- *Database*—Name of the InfluxDB database (*instances* or *containers* database).

With your InfluxDB connection configured, use Grafana and InfluxQL to query and visualize time-series data stored in InfluxDB. From the left panel, click Dashboards. From the top menu, click Home to get a list of dashboards. Click the Create New button at the bottom to create a new dashboard. To add a graph, just click the graph button in the panel filter. In the Query section, type the following InfluxQL statement:

```
SELECT mean("used_percent") FROM "mem_vm"
WHERE $timeFilter
GROUP BY time($__interval), "host" fill(null)
```

This query selects the memory usage from the `mem_vm` measurement and groups the results by Jenkins node. The query results in the graph in figure 13.7.

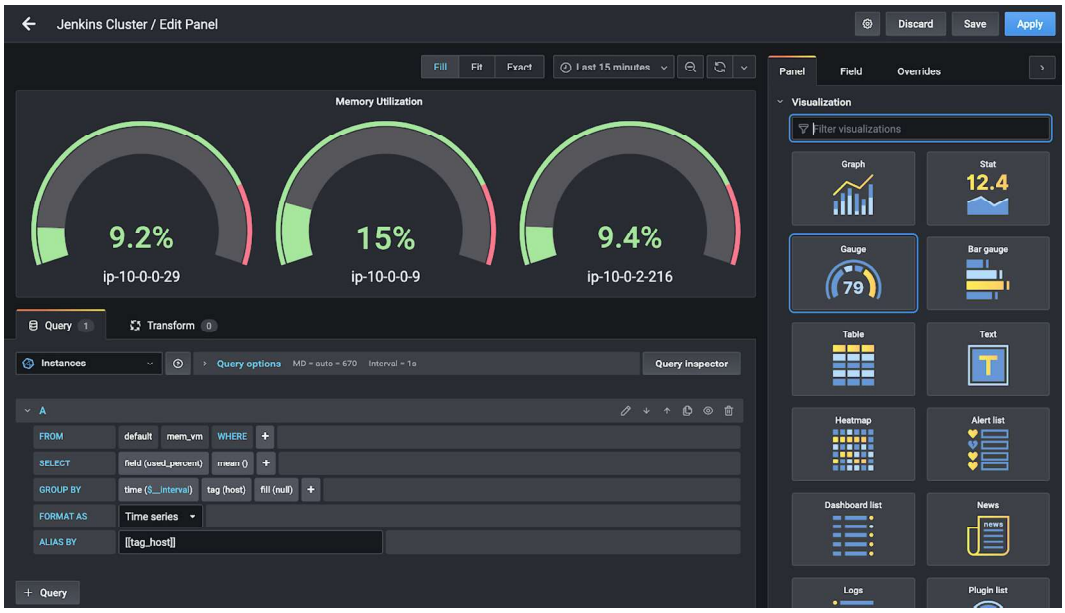

Figure 13.7 Building a memory utilization gauge chart

To monitor the Jenkins jobs build time, you can use the following statement:

```
SELECT mean("uptime_ns") FROM "docker_container_status_docker"
WHERE ("hostname" = 'Jenkins') AND $timeFilter
GROUP BY time($__interval), "container_name" fill(null)
```

This selects the uptime value (the amount of time the container is online and operational) from the `docker_container_status_docker` measurement and groups the results by the container name (figure 13.8).

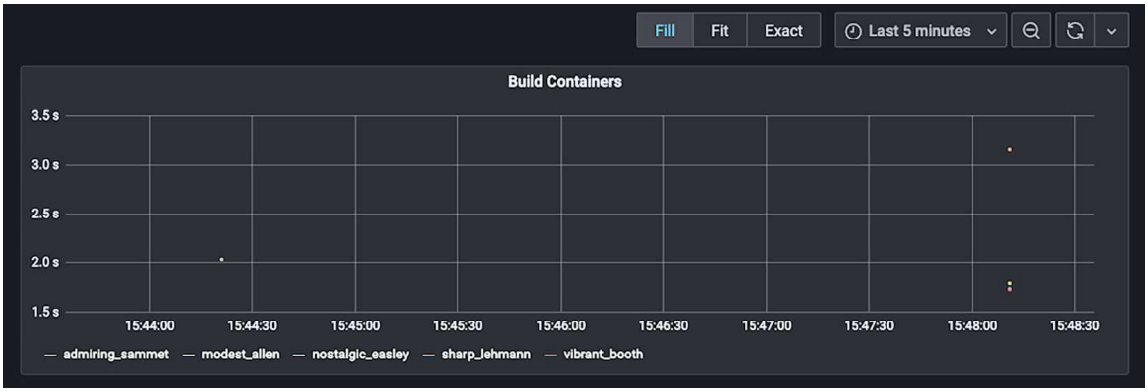

Figure 13.8 Monitoring containers built within CI/CD pipelines

Back to Grafana, you can create multiple graphs to monitor various metrics of the Jenkins cluster:

- CPU usage of Jenkins nodes (master and worker instances)
- Network traffic (in and out bytes)
- Memory utilization of each Jenkins node
- Number of running build jobs
- Overall health and number of workers

Figure 13.9 shows host-level details for the Jenkins cluster. The complete dashboard can be imported from the JSON file (chapter13/grafana/dashboard/influxdb.json). Refer to http://mng.bz/6mGD for instructions.

As mentioned earlier, monitoring the state of your instances is imperative to keeping your Jenkins cluster healthy, and by using the preceding metrics (and the many others) provided by Telegraf, you can achieve this with relative ease.

So far, you have seen how to monitor the Jenkins instances (server side). Let's explore monitoring the Jenkins server itself (application side). As you may have already guessed, a monitoring plugin for Jenkins can provide a lot of data about what's going on within Jenkins and about the tasks being performed by Jenkins. For example, the Metrics plugin (https://plugins.jenkins.io/metrics/) provides health

Figure 13.9 Jenkins host metrics

checks by exposing an API on the Jenkins server at the $JENKINS_URL/metrics endpoint. The API provides information on the following:

- HTTP sessions and current HTTP requests
- Detailed statistics of the build times and the build steps by period
- Threads, process list of OS, and heap dumps

For instance, the API call in figure 13.10 returns statistics about the number of executors available to Jenkins.

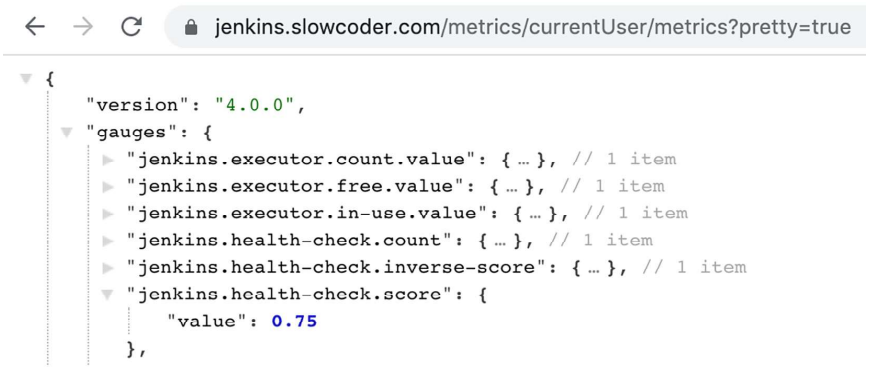

Figure 13.10 Metrics API with health-check endpoints

To create a dashboard based on those metrics, we can write a custom script to save those values regularly to InfluxDB, or use a Prometheus metric plugin (https://plugins.jenkins.io/prometheus/) to expose an endpoint (the default is /prometheus) with metrics that a Prometheus server can scrape.

Prometheus (https://prometheus.io/) is an open source monitoring system with a dimensional data model, flexible query language, efficient time-series database, and modern alerting approach.

NOTE The Packer template file and Terraform HCL files for baking and deploying a Prometheus server are available in the chapter13/prometheus folder.

First, install the Prometheus Metrics plugin (https://plugins.jenkins.io/prometheus/) from the Manage Plugins section. Once it's installed, you can see the plugin's output through JENKINS _URL/prometheus (figure 13.11).

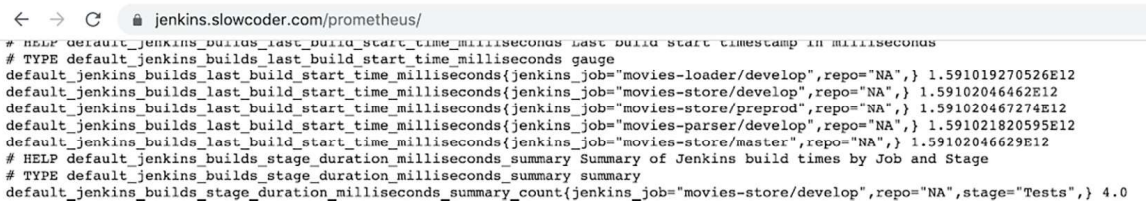

Figure 13.11 Prometheus endpoint serves a list of metrics

Then, you need to configure a Prometheus server to scrape metrics from Jenkins. Edit the configuration file at /etc/prometheus/prometheus.yml (listing 13.3). In the `scrape_configs` section, add a job for the Jenkins server. The format for writing this config file can be found at http://mng.bz/o8Vr.

Listing 13.3 Configuring Prometheus to scrape metrics from Jenkins

```
global:
  scrape_interval: 10s

scrape_configs:
  - job_name: 'prometheus_master'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
  - job_name: 'jenkins'
    metrics_path: '/prometheus/'
    scheme: https
    static_configs:
       - targets: ['JENKINS_URL']
```

On the Prometheus dashboard (the default port is 9090), you can explore the metrics collected from Jenkins. You will be greeted will the screen in figure 13.12.

Collected metrics are not very useful unless they are visualized. Connect Prometheus with Grafana by creating a new data source. To create a Prometheus data source in Grafana, follow these steps:

1. Click the cogwheel icon in the side panel to open the Configuration menu.
2. Click Data Sources.

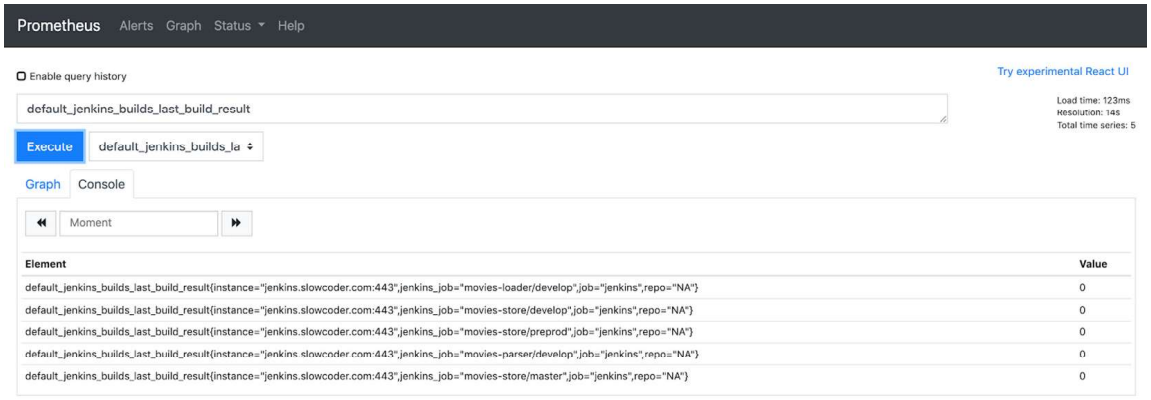

Figure 13.12 Exploring Jenkins metrics from the Prometheus dashboard

3. Click Add Data Source.
4. Select Prometheus as the type.
5. Set the appropriate Prometheus server URL to http://prometheus:9090.
6. Click Save & Test to save the new data source.

Then, create a dashboard based on the available metrics. The dashboard features application-level metrics (which track the total number of jobs in a queue, how many are pending, and how many are stuck or otherwise delayed), followed by internal operation metrics (JVM), and finally system-level metrics (disk I/O, network, memory, and so forth). Figure 13.13 shows a part of the dashboard.

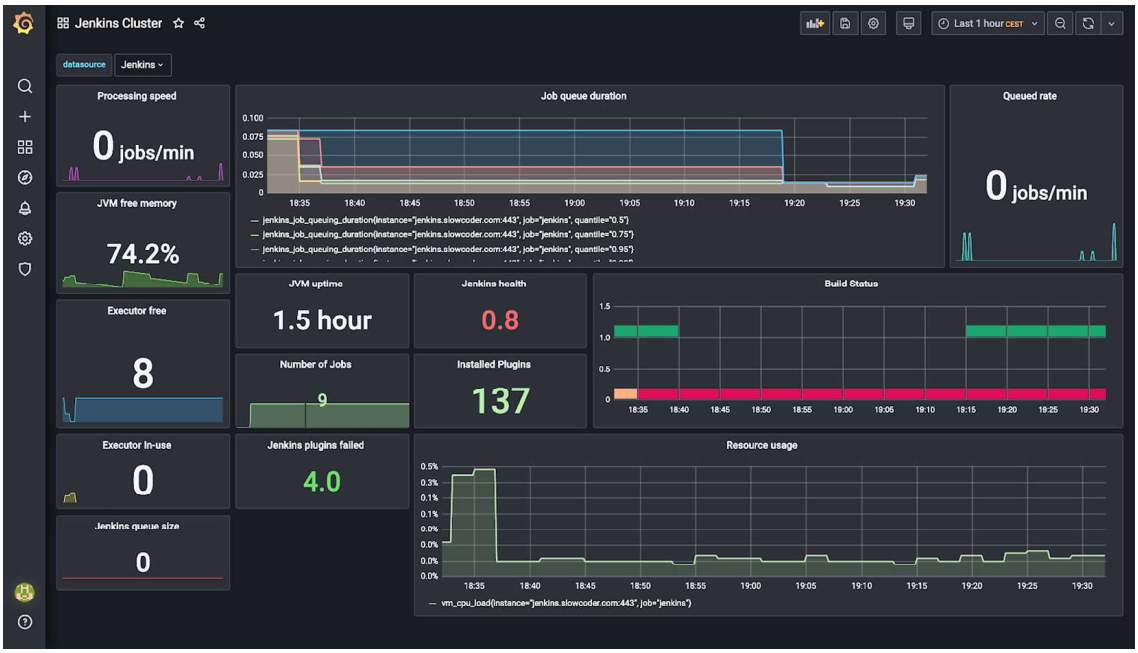

Figure 13.13 Comprehensive Jenkins monitoring summary of jobs and builds

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

Another popular solution for monitoring Jenkins is the Monitoring plugin (previously called JavaMelody). This plugin produces comprehensive HTML reports about the state of Jenkins, including CPU and system load, average response time, and memory usage; see https://plugins.jenkins.io/monitoring/ for more details. Moreover, the reports are served from the Jenkins dashboard, as shown in figure 13.14.

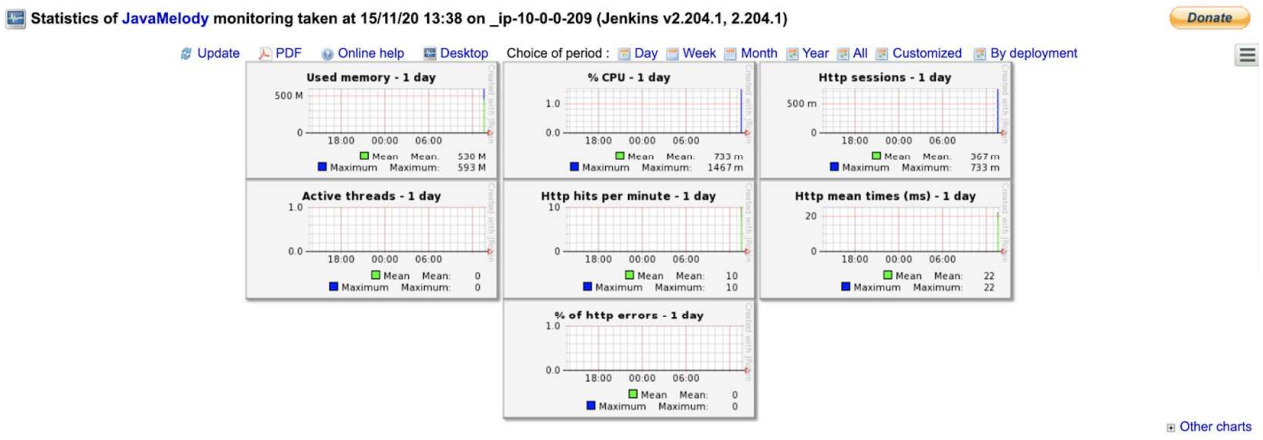

Figure 13.14 Statistics of JavaMelody monitoring

Great! You should now be able to monitor a Jenkins cluster running in production. To provide even further visibility into your Jenkins environment, you can collect and analyze Jenkins logs of real-time system and security events and correlate them with performance and server metrics to identify and resolve issues.

13.2 Centralized logging for Jenkins logs with ELK

By default, Jenkins logs are located at /var/log/jenkins/jenkins.log. To view those logs, SSH to the Jenkins master instance with the bastion host, and then issue the following command:

```
tail -f -n 100 /var/log/jenkins/jenkins.log
```

Figure 13.15 shows the command output.

```
[root@ip-10-0-0-130 ec2-user]# tail -f /var/log/jenkins/jenkins.log
        at hudson.remoting.SingleLaneExecutorService$1.run(SingleLaneExecutorService.java:131)
        at jenkins.util.ContextResettingExecutorService$1.run(ContextResettingExecutorService.java:28)
        at jenkins.security.ImpersonatingExecutorService$1.run(ImpersonatingExecutorService.java:59)
        at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:511)
        at java.util.concurrent.FutureTask.run(FutureTask.java:266)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)
2020-06-02 11:00:36.938+0000 [id=167]   INFO    c.s.o.i.Platform$JdkWithJettyBootPlatform#getSelectedProtocol:
2020-06-02 11:02:33.304+0000 [id=418]   INFO    c.s.o.i.Platform$JdkWithJettyBootPlatform#getSelectedProtocol:
```

Figure 13.15 Viewing Jenkins logs at /var/log/jenkins/jenkins.log

You can also view those logs from the web dashboard (figure 13.16). Head to the Jenkins dashboard and select System Log from the Manage Jenkins page.

Figure 13.16 Viewing Jenkins logs from the Jenkins dashboard

By default, Jenkins records every `INFO` log to stdout, but you can configure Jenkins to record logs of a specific Jenkins plugin by creating a custom log recorder. From the System Log page, click the Add New Log Recorder button and choose a name that makes sense to you. The example in figure 13.17 creates a log recorder for the Slack plugin (the Java package is located at jenkins.plugins.slack).

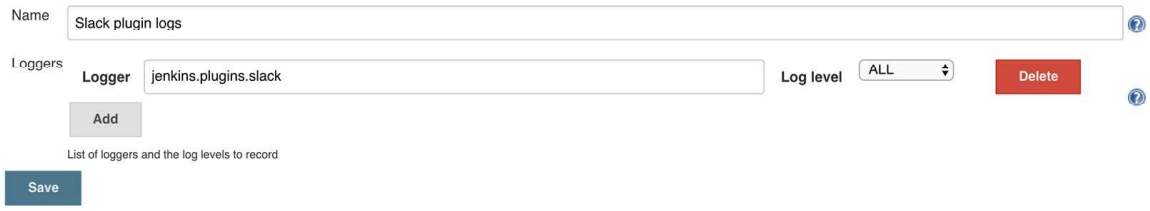

Figure 13.17 Capturing the Slack plugin's login with a custom log recorder

Now, if any Slack notification is sent from a Jenkins pipeline, a log should be captured as shown in figure 13.18.

Figure 13.18 Display of Slack plugin's logs

You can also view the build logs for a particular job by navigating to the job item from the dashboard and clicking Console Output, or by viewing the content of the logfile at $JENKINS_HOME/jobs/$JOB_NAME/builds/$BUILD_NUMBER/log.

Depending on a log rotation configuration, the logs could be saved for *X* number of builds (or days, and so forth), meaning the old job logs might be lost. That's why you need to persist the logs in a centralized logging platform for auditing and potential troubleshooting.

> **NOTE** You can enable the Discard Old Build plugin (https://plugins.jenkins.io/discard-old-build/) in each project or job configuration page to configure the interval to keep old builds (for example, once a month, once in 10 builds, and so forth).

Additionally, analyzing Jenkins logs can provide a lot of information that helps with troubleshooting the root cause of pipeline job failure. Build logs contain a full set of records such as build name, number, execution time, and other things. However, to analyze those logs, you need to ship them to an external logging platform. That's where a platform like the ELK stack (Elasticsearch, Logstash, and Kibana) comes into play.

13.2.1 Streaming logs with Filebeat

Filebeat (www.elastic.co/beats/filebeat), a lightweight agent that will be installed on the Jenkins master instance, will ship the logs to Logstash (www.elastic.co/logstash) for processing and aggregation. From there, the logs will be stored in Elasticsearch (www.elastic.co/elasticsearch) and visualized in Kibana (www.elastic.co/kibana) through interactive dashboards. Figure 13.19 summarizes the entire workflow.

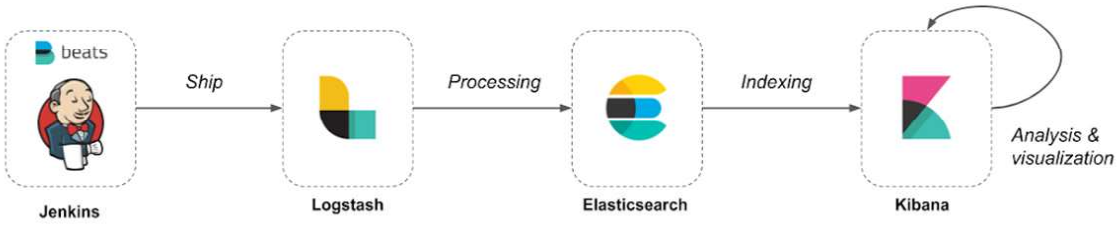

Figure 13.19 Shipping Jenkins logs to the ELK platform with Filebeat

To deploy this architecture, we need to create a machine image for each component. You can use Packer to bake the AMIs (figure 13.20). The Packer templates are available in the GitHub repository at chapter13/COMPONENT_NAME/packer/template.json.

Once the AMIs are created, you can use Terraform to deploy the ELK stack. The template resources are available in the GitHub repository at chapter13/COMPONENT_NAME/terraform/*.tf.

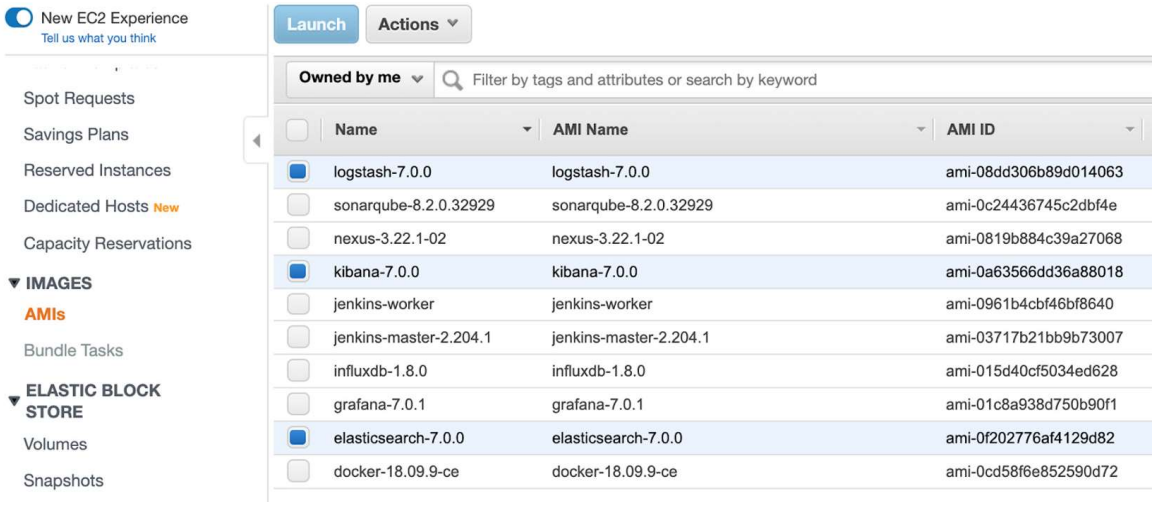

Figure 13.20 Logstash, Kibana, and Elasticsearch AMIs built with Packer

By the end of the provisioning process, three EC2 instances should be created, as shown in figure 13.21.

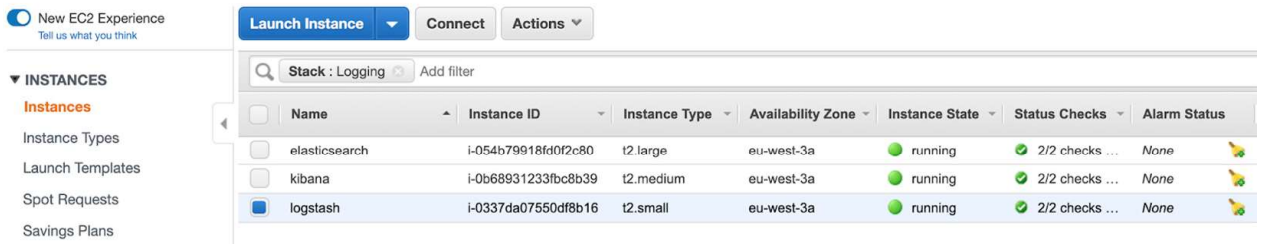

Figure 13.21 Deployed ELK stack on AWS

With the logging platform ready to consume incoming Jenkins logs, we need to install Filebeat on the Jenkins master instance. SSH to the Jenkins server, and run the commands in the following listing to install the latest stable version of Filebeat (at the time of writing this book, version 7.13.2 is available).

Listing 13.4 Installing the Filebeat agent on the Jenkins server

```
curl -L -O https://artifacts.elastic.co/downloads/beats/
filebeat/filebeat-7.13.2-x86_64.rpm
sudo rpm -vi filebeat-7.13.2-x86_64.rpm
```

Next, we need to set the path of the log files that we want to forward to ELK. Here we want to forward logs to /var/log/jenkins/jenkins.log. Go to the configuration directory of Filebeat under the location /etc/filebeat, and update filebeat.yml with the following listing.

Listing 13.5 Filebeat input configuration

```
filebeat.inputs:
- type: log
  enabled: true
  paths:
    - /var/log/jenkins/jenkins.log
  fields:
    type: jenkins
  multiline.pattern: '[0-9]{4}-[0-9]{2}-[0-9]{2}'
  multiline.negate: true
  multiline.match: after
output.logstash:
  hosts: ["LOGSTASH_HOST"]

processors:
 - add_host_metadata: ~
 - add_cloud_metadata: ~
 - add_docker_metadata: ~
 - add_kubernetes_metadata: ~
```

Harvests lines from the /var/log/jenkins/jenkins.log file

Adds a field called type to the output, so we can easily identify logs coming from Jenkins

Configures Filebeat to handle a multiline message

Sends logs directly to Logstash

Annotates each log event with relevant metadata from the host machine

Multiline messages are common in Jenkins logs, especially for log messages containing Java stack traces. Here's an example of a Java stack trace:

```
2020-10-22 20:06:58.217+0000[id=124635] FATAL: Ping failed.
                        java.util.concurrent.TimeoutException:
                        at
     hudson.remoting.PingThread.ping(PingThread.java:134)
                        at hudson.remoting.PingThread.run(PingThread.java:90)
```

To correctly handle these multiline messages, we use the `multiline` settings to specify which lines are part of a single log message.

Replace the `LOGSTASH_HOST` variable, with the IP address of the Logstash server. Then restart the Filebeat agent with the following command:

```
systemctl restart filebeat
```

Head to the Kibana dashboard (at `KIBANA_IP:5601`), jump to the Management tab, and to Index Patterns. We have to create a new index pattern. Creating an index pattern means mapping Kibana with an Elasticsearch index. Since Logstash stores incoming Jenkins logs to a series of indices in the format *jenkins-YYYY.MM.DD*, we will create an index pattern `jenkins-*` to explore all the logs, as shown in figure 13.22.

Click the Next Step option. From the Time Filter Field Name drop-down, select @timestamp. Then click the Create Index Pattern button.

Now, to view logs, go to the Discover page. You can see your index data coming in (figure 13.23).

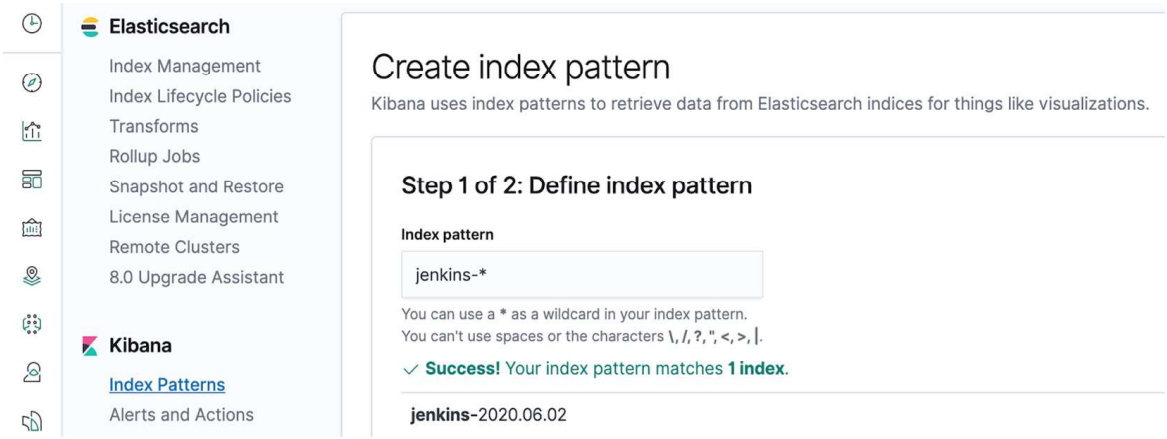

Figure 13.22 Connecting an Elasticsearch index to Kibana

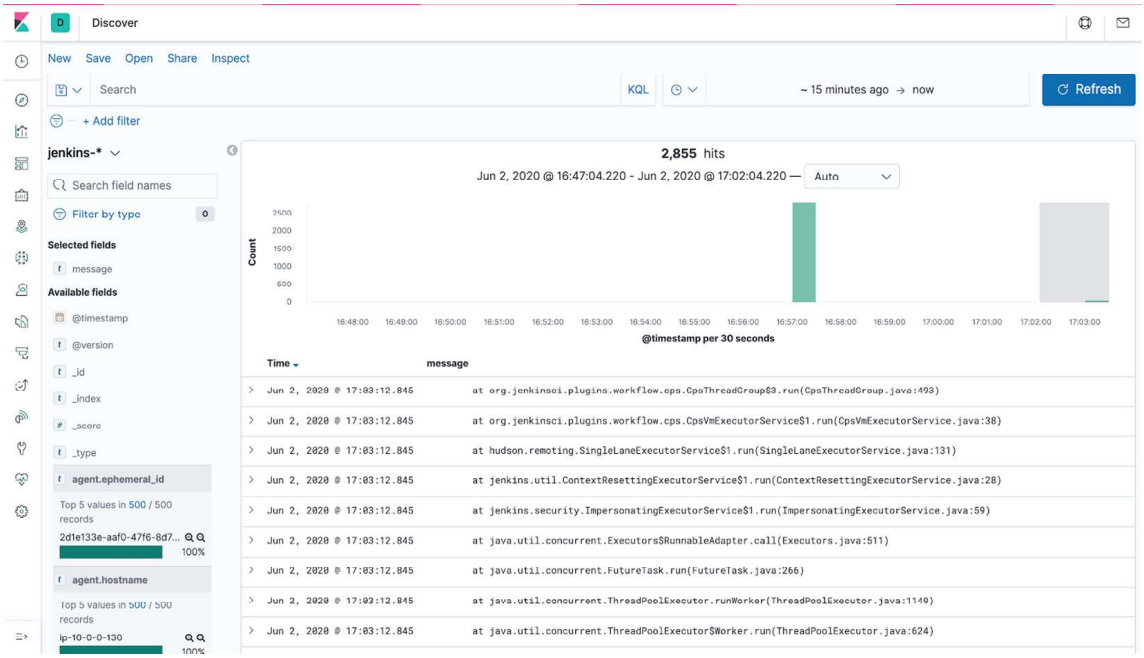

Figure 13.23 Visualizing Jenkins logs from Kibana

Now you have a working pipeline that reads Jenkins logs. However, you'll notice that the format of the log messages is not ideal. You want to parse the log messages to create specific, named fields from the logs. Let's take, as an example, the following Jenkins log:

```
2020-06-02 15:21:56.990+0000 INFO  o.j.p.workflow.job.WorkflowRun#finish:
     movies-loader/develop #7 completed: SUCCESS
```

The timestamp at the beginning of the line is easy to define as the level of the log (`INFO`, `WARNING`, `DEBUG`, etc.). To parse the line, we can write a Grok expression.

Grok works by parsing text patterns, using regular expressions, and assigning them to an identifier. The syntax is `%{PATTERN:IDENTIFIER}`. We can write a sequence of Grok patterns and assign various pieces of the preceding log message to various identifiers, as you can see in the following listing.

Listing 13.6 Grok expression to parse Jenkins log message

```
%{TIMESTAMP_ISO8601:createdAt}
     %{LOGLEVEL:level}%{SPACE}%{JAVACLASS:class}%{DATA:state}:%{SPACE}%{JOBNA
     ME:project} #%{NUMBER:buildNumber} %{DATA:execution}: %{WORD:status}
```

Grok comes with its own dictionary of patterns that you can use out of the box. But you can always define your own custom pattern, as shown in the following listing.

Listing 13.7 Grok custom patterns definition

```
JAVACLASS (?:[a-zA-Z0-9-]+\.)+[A-Za-z0-9$]+
JOBNAME [a-zA-Z0-9\-\/]+
```

You can use the Kibana Grok Debugger console to debug the expression. This feature, which is automatically enabled in Kibana, is located on the DevTools tab.

Enter the log message in the Sample Data field, and the Grok expression in the Grok Pattern field. Then click Simulate. You will see the simulated event that results from applying the Grok pattern (figure 13.24).

Note that the Grok pattern references the `JAVACLASS` and `JOBNAME` custom patterns. They are defined in the Custom Patterns section. Each pattern definition is specified on its own line.

NOTE If an error occurs, you can continue iterating over the custom pattern until the output matches the event that you expect.

Figure 13.24 Simulating Grok parsing with Grok Debugger tool

The Grok expression is working, but we want the parsing mechanism to be done before storing logs to Elasticsearch. That's why we will update the Logstash config (chapter13/logstash/packer/jenkins.conf) to parse incoming logs from Filebeat. The `filter` section will attempt to match messages coming from Jenkins with the Grok expression defined earlier, as shown in the following listing.

Listing 13.8 Parsing Jenkins logs at the Logstash level

```
filter {
  if [type] == "jenkins" {
    grok {
      patterns_dir => ["/etc/logstash/patterns"]
      match => {
        "message" =>
     "%{TIMESTAMP_ISO8601:createdAt}%{SPACE}\[id=%{INT:buildId}\]
%{SPACE}%{LOGLEVEL:level}%{SPACE}%{JAVACLASS:class}
%{DATA:state}:%{SPACE}%{JOBNAME:project}
#%{NUMBER:buildNumber} %{DATA:execution}: %{WORD:status}"
      }
    }
  }
}
```

This code takes the Jenkins logs collected by Filebeat, parses them into fields, and sends the fields to Elasticsearch. The `pattern_dir` setting tells Logstash where your custom patterns directory is. You can customize the parsing mechanism by adding more processing, such as dropping unused fields or renaming fields. See the Mutate Filter plugin at http://mng.bz/J6Av for more information.

Restart Logstash to reload the configuration. Your Jenkins logs will be gathered and structured into fields (figure 13.25). Right now, not much is in there because you are gathering only Jenkins logs. Here, you can search and browse through your logs.

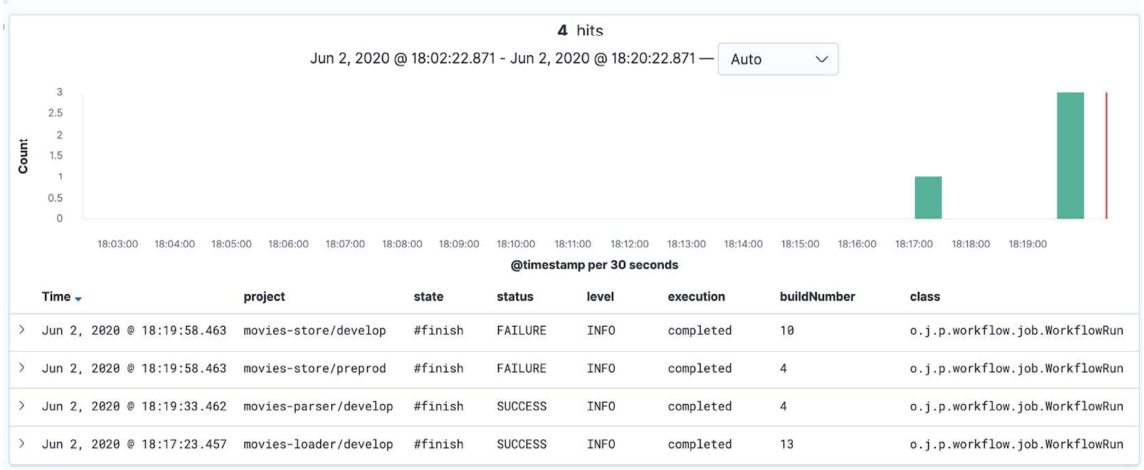

Figure 13.25 Structuring Jenkins logs into separated queryable fields

Each log message coming from Jenkins will match and result in the fields listed in table 13.1.

Table 13.1 Jenkins index fields in Elasticsearch

Field	Description
`time`	The data and time of the message in UTC format
`level`	The log message level (INFO, WARNING, DEBUG, FATAL, ERROR)
`project`	The Jenkins job's build name
`buildNumber`	The build number of the job, which identifies how many times Jenkins runs this build process
`status`	The status of the build (FAILURE or SUCCESS)
`execution`	The current state of the build (running, pending, terminated, or completed)

You can create a stacked bar chart showing the number of failed versus successful builds based on the `status` field over a period of time; see figure 13.26.

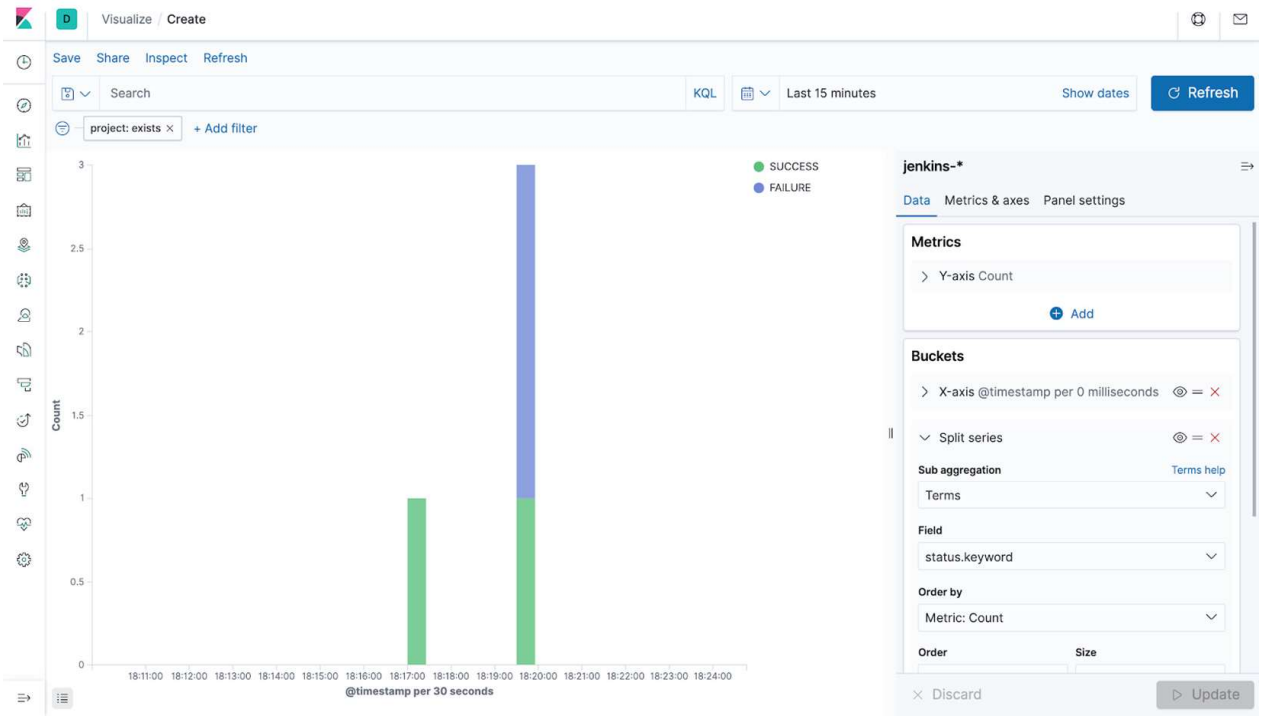

Figure 13.26 Building interactive widgets based on Jenkins structured fields

You can save the bar chart as a widget and import it to a dashboard. With a dashboard, you can combine multiple visualizations onto a single page, and then filter them by providing a search query or by selecting filters by clicking elements in the visualization.