The setup.sh script will install SonarQube from the official release page. For this example, SonarQube 8.2.0 will be installed. SonarQube supports PostgreSQL, MySQL, Microsoft SQL Server (MSSQL), and Oracle as a backend. I opted to go with PostgreSQL to store configurations and report results. Then, the script creates a directory named sonar, sets permissions, and configures SonarQube to start automatically; see the following listing.

Listing 8.22 Installing SonarQube LTS

```
wget https://binaries.sonarsource.com/
Distribution/sonarqube/$SONAR_VERSION.zip -P /tmp
unzip /tmp/$SONAR_VERSION.zip
mv $SONAR_VERSION sonarqube
mv sonarqube /opt/

apt-get install -y unzip curl
sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/
 `lsb_release -cs`-pgdg main" >> /etc/apt/sources.list.d/pgdg.list'
wget -q https://www.postgresql.org/media/keys/ACCC4CF8.asc
-O - | sudo apt-key add -
apt-get install -y postgresql postgresql-contrib
systemctl start postgresql
systemctl enable postgresql
cat > /tmp/db.sql <<EOF
CREATE USER $SONAR_DB_USER WITH ENCRYPTED PASSWORD '$SONAR_DB_PASS';
CREATE DATABASE $SONAR_DB_NAME OWNER $SONAR_DB_USER;
EOF
sudo -u postgres psql postgres < /tmp/db.sql

mv /tmp/sonar.properties /opt/sonarqube/conf/sonar.properties
sed -i 's/#RUN_AS_USER=/RUN_AS_USER=sonar/' sonar.sh
sysctl -w vm.max_map_count=262144
groupadd sonar
useradd -c "Sonar System User" -d /opt/sonarqube -g sonar -s /bin/bash sonar
chown -R sonar:sonar /opt/sonarqube
ln -sf /opt/sonarqube/bin/linux-x86-64/sonar.sh /usr/bin/sonar
cp /tmp/sonar.init.d /etc/init.d/sonar
chmod 755 /etc/init.d/sonar
update-rc.d sonar defaults
service sonar start
```

NOTE The full shell script is available on the GitHub repository along with a step-by-step guide. Also, make sure you have at least 4 GB of memory to run the 64-bit version of SonarQube.

Once you define the needed Packer variables, issue a packer build command to start the provisioning process. Once the AMI is baked, it should be available on the EC2 dashboard in the Images section, as shown in figure 8.31.

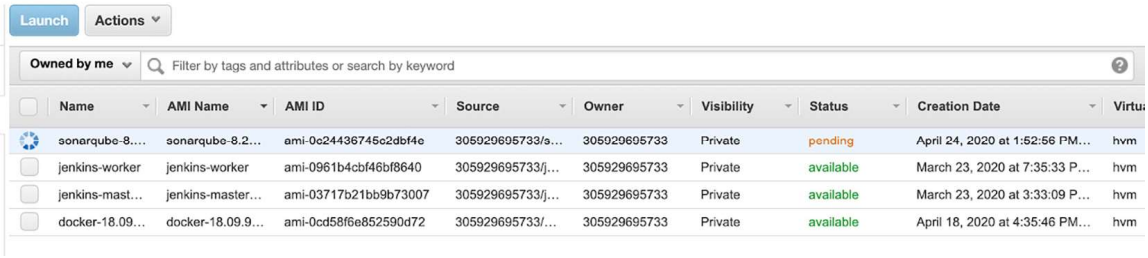

Figure 8.31 SonarQube machine image

From there, use Terraform to deploy a private EC2 instance based on the SonarQube AMI, as shown in the following listing.

Listing 8.23 SonarQube EC2 instance resource with Terraform

```
resource "aws_instance" "sonarqube" {
  ami                    = data.aws_ami.sonarqube.id
  instance_type          = var.sonarqube_instance_type
  key_name               = var.key_name
  vpc_security_group_ids = [aws_security_group.sonarqube_sg.id]
  subnet_id              = element(var.private_subnets, 0)

  root_block_device {
    volume_type           = "gp2"
    volume_size           = 30
    delete_on_termination = false
  }

  tags = {
    Name   = "sonarqube"
    Author = var.author
  }
}
```

Then, define a public load balancer to forward incoming HTTP and HTTPS (optional) traffic to the instance on port 9000 (the port to which the SonarQube dashboard is exposed). Also, create an A record in Route 53 pointing to the load balancer FQDN.

Issue the `terraform apply` command to provision the instance and other resources. The instance should be deployed in a few seconds, as shown in figure 8.32.

Figure 8.32 SonarQube private EC2 instance

On the terminal, you should have the URL of the public load balancer in the Outputs section, as shown in figure 8.33.

```
aws_security_group.elb_sonarqube_sg: Creating...
aws_security_group.elb_sonarqube_sg: Creation complete after 1s [id=sg-082e62dc156bf43cd]
aws_security_group.sonarqube_sg: Creating...
aws_security_group.sonarqube_sg: Creation complete after 2s [id=sg-0eec135401e424f0f]
aws_instance.sonarqube: Creating...
aws_instance.sonarqube: Still creating... [10s elapsed]
aws_instance.sonarqube: Still creating... [20s elapsed]
aws_instance.sonarqube: Creation complete after 22s [id=i-0ee0c0253678e09b4]
aws_elb.sonarqube_elb: Creating...
aws_elb.sonarqube_elb: Creation complete after 2s [id=tf-lb-20200424121504563600000001]
aws_route53_record.sonarqube: Creating...
aws_route53_record.sonarqube: Still creating... [10s elapsed]
aws_route53_record.sonarqube: Still creating... [20s elapsed]
aws_route53_record.sonarqube: Still creating... [30s elapsed]
aws_route53_record.sonarqube: Creation complete after 39s [id=Z2TR95QTU3UIUT_sonarqube.slowcoder.com_A]

Apply complete! Resources: 5 added, 0 changed, 0 destroyed.

Outputs:

sonarqube = https://sonarqube.slowcoder.com
```

Figure 8.33 SonarQube DNS URL

Head over to the URL and log in with the default credentials (figure 8.34). Right now, no user accounts are configured in SonarQube. However, by default, an admin account exists with the username `admin` and the password `admin`.

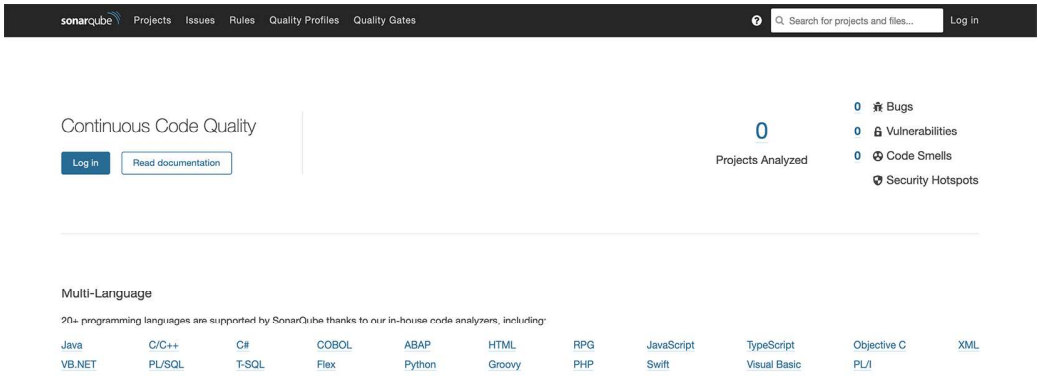

Figure 8.34 SonarQube web dashboard

Next, make sure the TypeScript analyzer is enabled from the SonarQube Plugins section, as shown in figure 8.35.

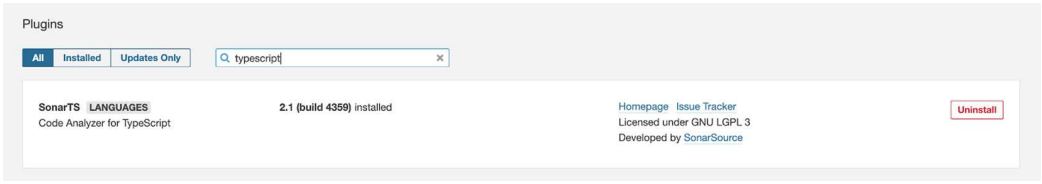

Figure 8.35 SonarQube TypeScript analyzer plugin

Then, generate a new token for Jenkins to avoid using SonarQube admin credentials for security purposes. Go to Administration and navigate to Security. On the same page under the Tokens section is an option to generate a token; click the Generate button, shown in figure 8.36.

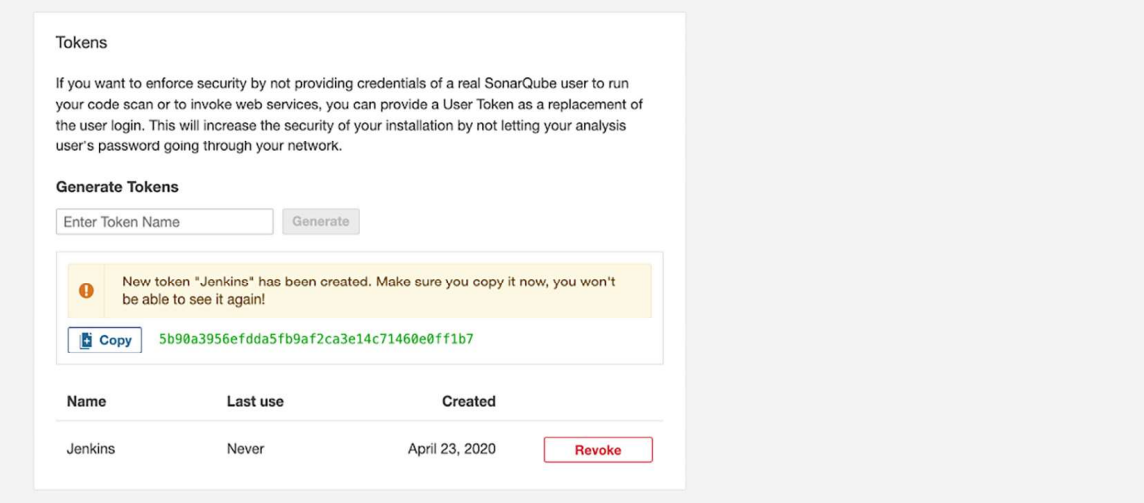

Figure 8.36 SonarQube Jenkins dedicated token

The server authentication token should be created as a `Secret text` credential from Jenkins, as shown in figure 8.37.

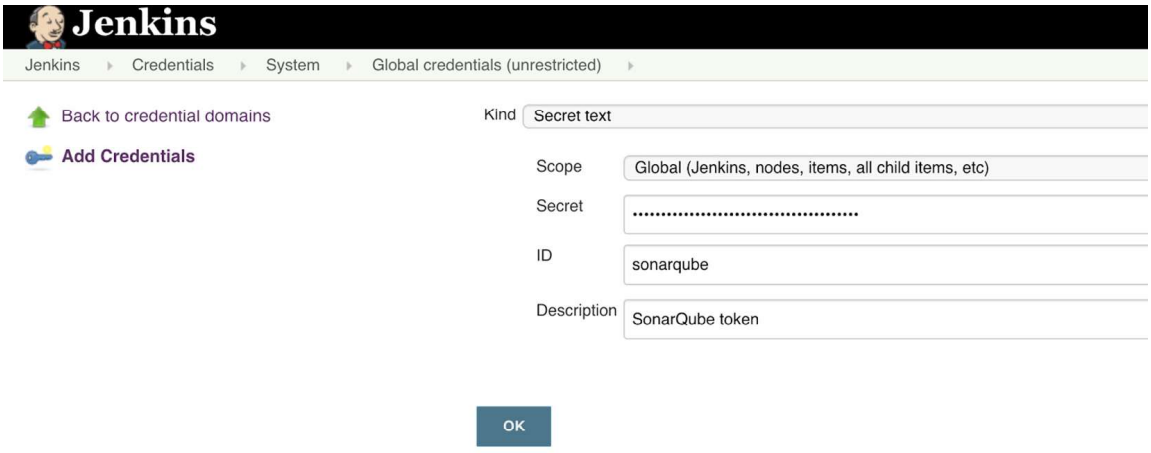

Figure 8.37 SonarQube secret text credentials

To trigger the scanning from the CI pipeline, we need to install SonarQube Scanner. You can choose to either install it automatically or provide the installation path for this tool on Jenkins workers. It can be installed by choosing Manage Jenkins > Global

Tool Configuration. Or you can bake a new Jenkins worker image with SonarQube Scanner with the commands shown in the following listing.

Listing 8.24 SonarQube Scanner installation

```
wget https://binaries.sonarsource.com/
Distribution/sonar-scanner-cli/sonar-scanner-cli-2.0.1873-linux.zip -P /tmp
unzip /tmp/sonar-scanner-cli-4.2.0.1873-linux.zip
mv sonar-scanner-4.2.0.1873-linux sonar-scanner
ln -sf /home/ec2-user/sonar-scanner/bin/sonar-scanner /usr/bin/sonar-scanner
```

> **NOTE** The launch configuration of the Jenkins workers is immutable. You will need to clone the launch configuration, update it with newly built AMI, and attach it to the Jenkins workers' Auto Scaling group to create new workers with the Sonar Scanner tool.

Lastly, make Jenkins aware of the SonarQube server installation from the Configure menu in Manage Jenkins, as shown in figure 8.38.

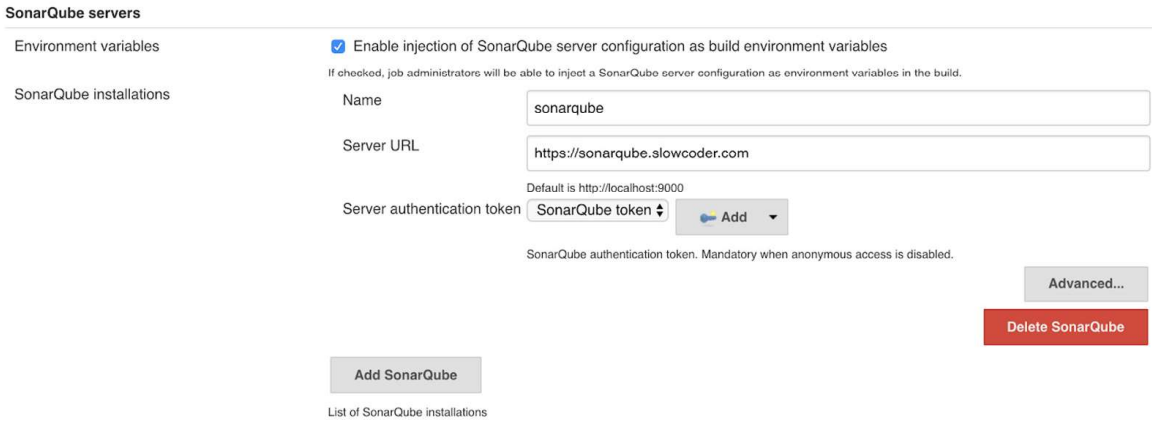

Figure 8.38 SonarQube server settings

Then, create a sonar-project.properties file in the movies-marketplace root folder to publish the coverage report to the SonarQube server. This file contains certain sonar properties, such as which folder to scan and exclude, and the name of the project; see the following listing.

Listing 8.25 SonarQube project configuration

```
sonar.projectKey=angular:movies-marketplace
sonar.projectName=movies-marketplace
sonar.projectVersion=1.0.0
sonar.sourceEncoding=UTF-8
sonar.sources=src
sonar.exclusions=**/node_modules/**,**/*.spec.ts
```

```
sonar.tests=src/app
sonar.test.inclusions=**/*.spec.ts
sonar.ts.tslint.configPath=tslint.json
sonar.javascript.lcov.reportPaths=/home/ec2-user/coverage/marketplace/
     lcov.info
```

Next, update the Jenkinsfile to create a new `Static Code Analysis` stage.

Then inject a SonarQube global configuration (secret token and SonarQube server URL values) with the `withSonarQubeEnv` block and invoke the `sonar-scanner` command to start the analysis process, as shown in the following listing.

Listing 8.26 Triggering SonarQube analysis

```
stage('Static Code Analysis'){
        withSonarQubeEnv('sonarqube') {
            sh 'sonar-scanner'
        }
}
```

You can override property values by using the `-D` flag:

```
sh 'sonar-scanner -Dsonar.projectVersion=$BUILD_NUMBER'
```

This option allows us to attach the Jenkins build number with every analysis that we perform and publish to SonarQube.

After a successful build, the logs will show you the files and folders SonarQube has scanned. After scanning, the analysis report is posted to the SonarQube server we have integrated. This analysis is based on rules defined by SonarQube. If the code passes the error threshold, it's allowed to move to the next step in its life cycle. But if it crosses the error threshold, it's dropped:

```
INFO: Sensor SonarTS [typescript] (done) | time=0ms
INFO: ------------- Run sensors on project
INFO: Sensor Zero Coverage Sensor
INFO: Sensor Zero Coverage Sensor (done) | time=3ms
INFO: CPD Executor Calculating CPD for 5 files
INFO: CPD Executor CPD calculation finished (done) | time=27ms
INFO: Analysis report generated in 103ms, dir size=121 KB
INFO: Analysis report compressed in 42ms, zip size=42 KB
INFO: Analysis report uploaded in 51ms
INFO: ANALYSIS SUCCESSFUL, you can browse https://sonarqube.slowcoder.com/dashboard?id=angular%3Amovies-marketplace
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted analysis report
INFO: More about the report processing at https://sonarqube.slowcoder.com/api/ce/task?id=AXGrwc7YDDGq9I6LnDBw
INFO: Analysis total time: 6.103 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 9.014s
INFO: Final Memory: 13M/44M
INFO: ------------------------------------------------------------------------
```

You can define your custom thresholds by creating Quality Profiles, which are a set of rules that will make the pipeline fail if an issue is raised in your codebase.

NOTE Refer to this official documentation for a step-by-step guide on how to create SonarQube custom rules with Quality Profiles: http://mng.bz/l9vy.

Finally, on visiting the SonarQube server, the project details should be visible with all the metrics captured from the code coverage report, as you can see in figure 8.39.

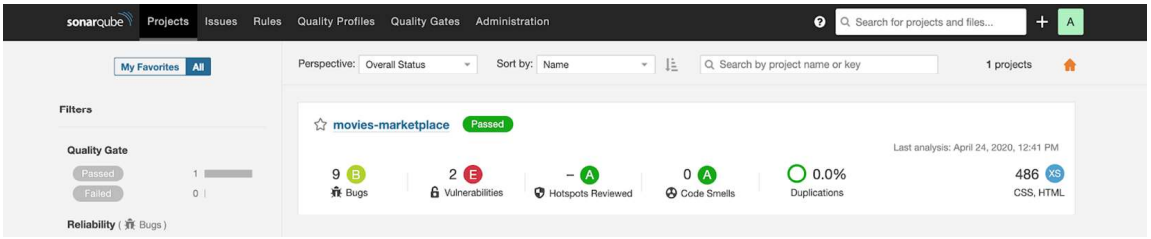

Figure 8.39 SonarQube project metrics

Now you can go inside the movies-marketplace project and discover issues, bugs, code smells, coverage, or duplication. The dashboard (figure 8.40) shows where you stand in terms of quality in the glimpse of an eye.

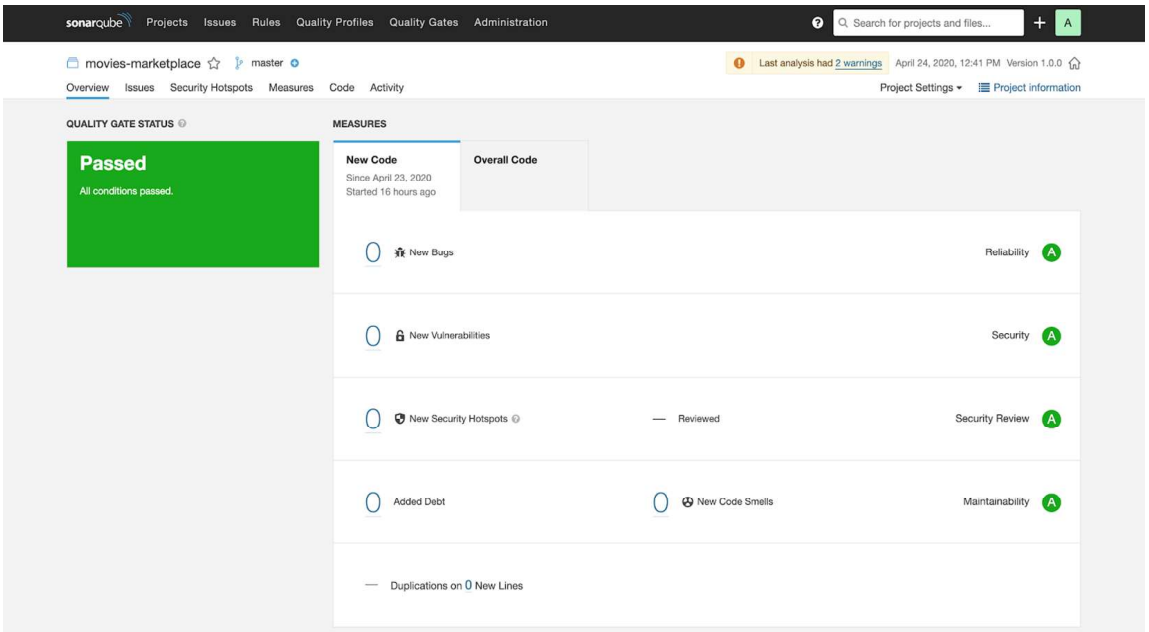

Figure 8.40 SonarQube project deep-dive metrics and issues

Also, when the job is completed, the SonarQube Scanner plugin will detect that a SonarQube analysis was made during the build. The plugin will then display a badge and a widget on the Jenkins job page with a link to the SonarQube dashboard as well as quality gate status, as shown in figure 8.41.

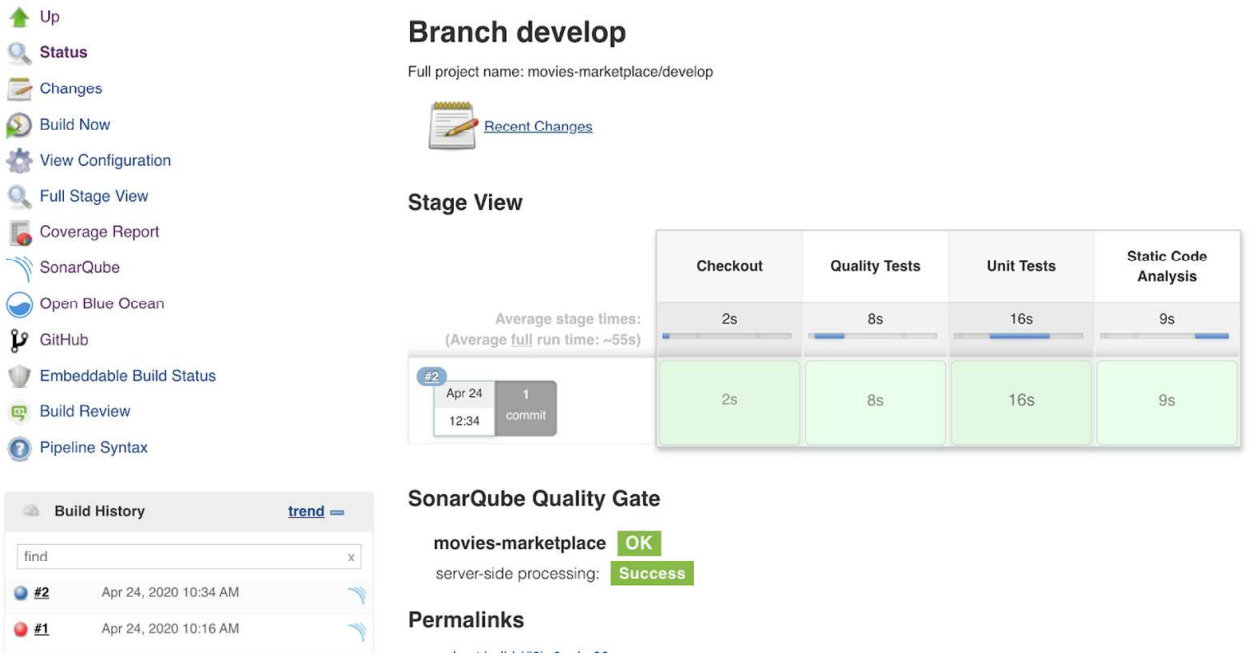

Figure 8.41 SonarQube integration with Jenkins

The SonarQube analysis was quick, but for larger projects, the analysis might take a few minutes to complete.

To wait for the analysis to be completed, we will pause the pipeline with the `waitForQualityGate` step, which waits for SonarQube analysis to be done. To notify the CI pipeline about the analysis completion, we need to create a webhook on SonarQube to notify Jenkins when project analysis is done, as shown in figure 8.42.

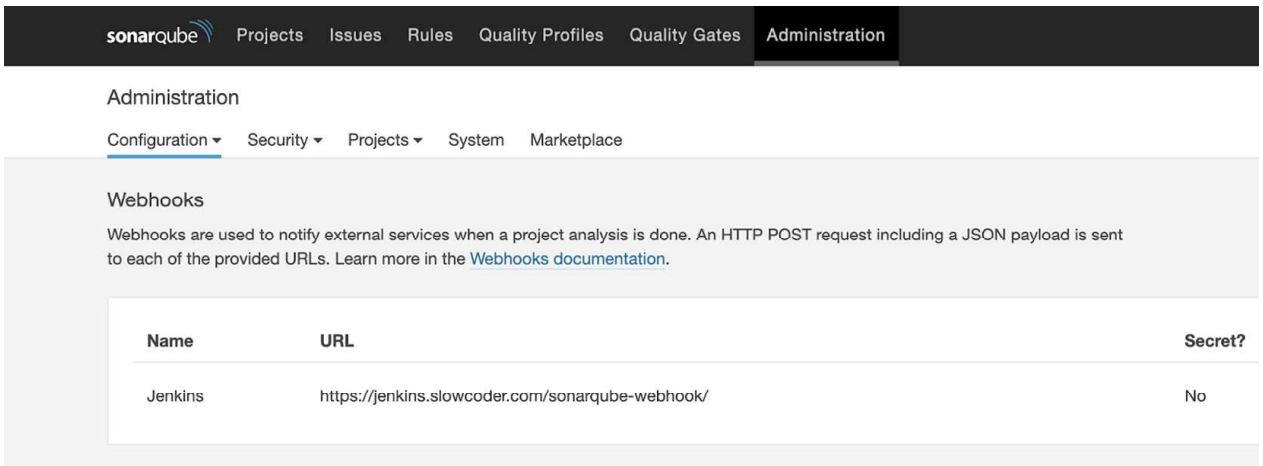

Figure 8.42 SonarQube webhook creation

Next, in the following listing, we update the Jenkinsfile to integrate the `waitForQualityGate` step that pauses the pipeline until SonarQube analysis is completed and returns the quality gate status.

Listing 8.27 Adding a quality gate to the Jenkinsfile

```
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}"
            }
        }
}
```

NOTE The quality gate can be moved outside the `node{}` block to avoid occupying a Jenkins worker waiting for SonarQube notification.

Commit the changes and push them to the remote repository. A new build will be triggered, and SonarQube analysis will be kicked off automatically. Once the analysis is completed, a notification will be sent to the CI pipeline to resume the pipeline stages, as shown in figure 8.43.

NOTE We can set up Post-build actions in Jenkins to notify the user about the test results.

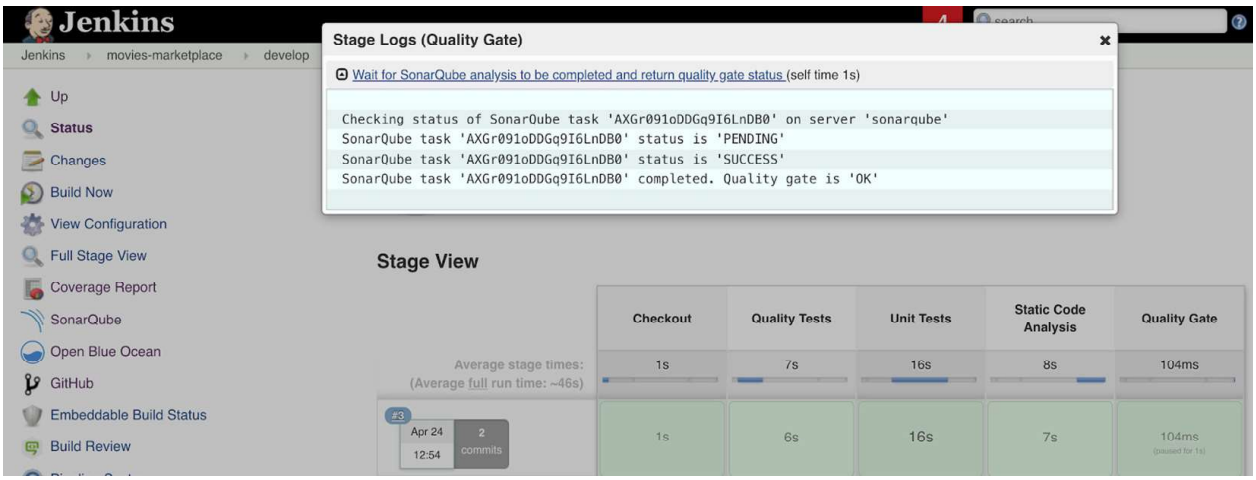

Figure 8.43 SonarQube project analysis status

As a result, as soon as a developer commits the code to GitHub, Jenkins will fetch/pull the code from the GitHub repository, perform static code analysis with the help of Sonar Scanner, and send analysis reports to the SonarQube server.

In this chapter, you learned how to run various automated tests and how to integrate external tools like Nancy and SonarQube to inspect code quality, detect bugs, and avoid potential security vulnerabilities while continuously building microservices within Jenkins CI pipelines. In the next chapter, we will build the Docker image after a successful run of tests and push the image to a private remote repository.

Summary

- Docker containers are used to run tests to avoid installing multiple runtime environments for each service we're integrating and keep a consistent execution environment across all Jenkins workers.
- Promoting traditional security practices into CI/CD workflows like external dependencies scanning can enable an additional security layer to avoid security breaches and vulnerabilities.
- Headless Chrome is a way to run UI tests in a headless environment without the full browser UI.
- The parallel DSL step gives the ability to easily run pipeline stages in parallel.
- SonarQube is a code-quality management tool that allows teams to manage, track, and improve the quality of their source code.

9 Building Docker images within a CI pipeline

This chapter covers

- Building Docker images inside Jenkins pipelines and best practices of writing Dockerfiles
- Using Docker agents as an execution environment in Jenkins declarative pipelines
- Integrating Jenkins build statuses into GitHub pull requests
- Deploying and configuring hosted and managed Docker private registry solutions
- Docker images life cycle within the development cycle and tagging strategies
- Scanning Docker images for security vulnerabilities within Jenkins pipelines

In the previous chapter, you learned how to run automated tests inside Docker containers within CI pipelines. In this chapter, we will finish the CI workflow by building a Docker image and storing it inside a private remote repository for versioning; see figure 9.1.

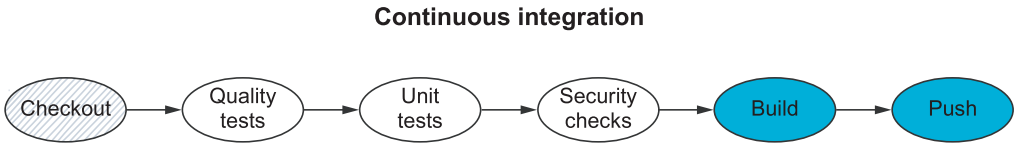

Figure 9.1 The Build and Push stages will be implemented in this chapter.

By the end of this chapter, you should be able to build a similar CI pipeline with these steps:

1. Check out the source code from a remote repository. The CI server fetches the code from the version-control system (VCS) on a push event.
2. Run pre-integration tests such as unit tests, security tests, quality tests, and UI tests inside a Docker container. These might include generating coverage reports and integrating quality-inspection tools like SonarQube for static code analysis.
3. Compile the source code and build a Docker image (automated packaging).
4. Tag the end image and store it in a private registry.

Figure 9.2 summarizes the end result of the CI workflow.

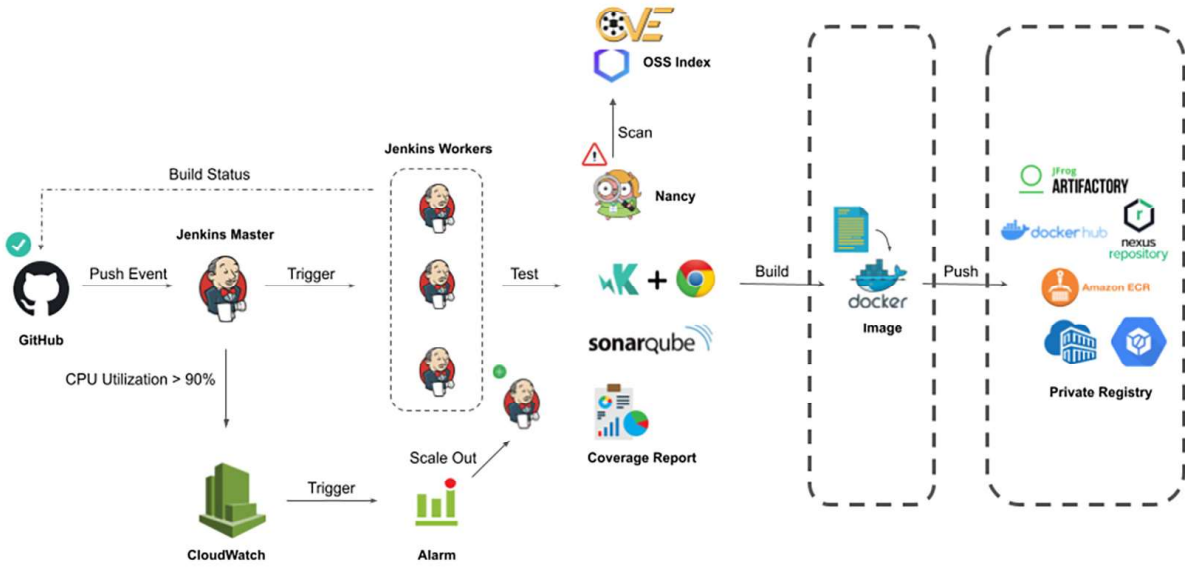

Figure 9.2 The CI pipeline process

The purpose of this CI pipeline is to automate the process of continuously building, testing, and uploading the Docker image to the private registry. Reporting for failures/success happens at every stage.

NOTE The CI design discussed in this chapter and previous ones can be modified to suit the needs of any type of project; the users just need to identify the right tools and configurations that can be used with Jenkins.

9.1 Building Docker images

For now, each push event to the remote repository triggers the pipeline on Jenkins. The pipeline will be executed based on stages defined in the Jenkinsfile. The first stage to be launched will be cloning the code from the remote repository, running automated tests, and publishing coverage reports. Figure 9.3 shows the current CI workflow for the movies-loader service.

Figure 9.3 Current CI workflow

If the tests are successful, the next stage will be building the artifact; in our case, it will be a Docker image.

> **NOTE** When you're building a Docker image for your application, you're building on top of an existing image. A broken base image can lead to production outages (security breaches, for instance). I recommend using an up-to-date and well-maintained image.

9.1.1 Using the Docker DSL

To build the main application Docker image, we need to define a Dockerfile with a set of instructions that specify the environment to use and the commands to run. Create a Dockerfile in the top-level directory of the movies-loader project, using the following code.

Listing 9.1 Movie loader's Dockerfile

```
FROM python:3.7.3
LABEL MAINTAINER mlabouardy
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY movies.json main.py ./
CMD python main.py
```

The Python-based application will use Python v3.7.3 as a base image, install the runtime dependencies with the pip manager, and set `python main.py` as the main command for the Docker image.

> **NOTE** To maintain the consistency of your image builds, create a requirements.txt file with transitively pinned versions of all used dependencies.

The order of instructions in a Dockerfile is important. The Docker image is rebuilt whenever any change occurs in the source code. That's why I placed the `pip install` command in listing 9.1, as the dependencies are not frequently changed. Therefore, Docker will rely on layer caching that will speed up the build time of the image. Refer to the official Docker documentation to learn more about the Docker build cache: http://mng.bz/B10J.

Finally, we add a `Build` stage in the Jenkinsfile, which uses the Docker DSL to build an image based on the Dockerfile in the repository:

```
stage('Build'){
        docker.build(imageName)
}
```

The `build()` method builds the Dockerfile in the current directory by default. You can override this by providing the Dockerfile path as the second argument of the `build()` method.

The changes are pushed to the develop branch with the following commands:

```
git add Jenkinsfile Dockerfile
git commit -m "building docker image"
git push origin develop
```

Then a new build should be triggered, and the image should be built, as shown in figure 9.4.

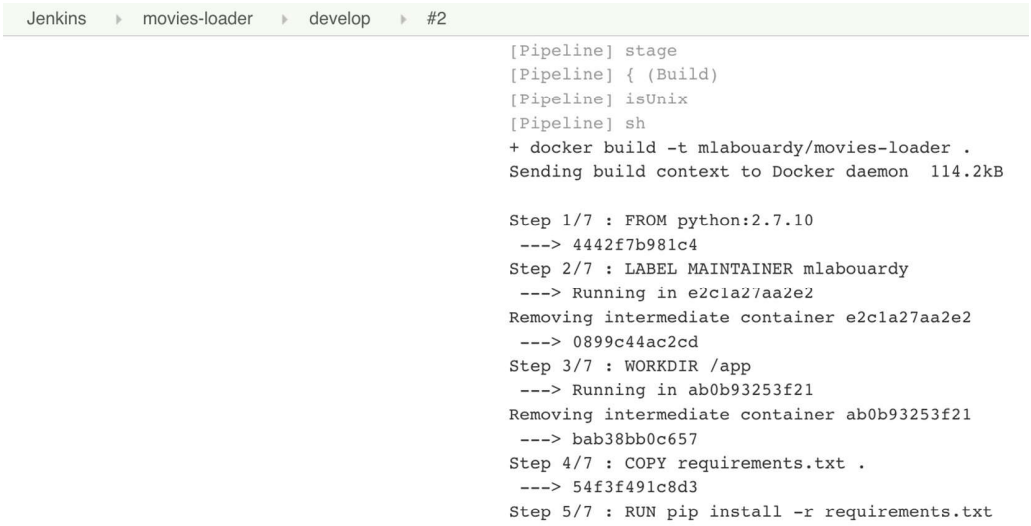

Figure 9.4 Python Docker image build logs

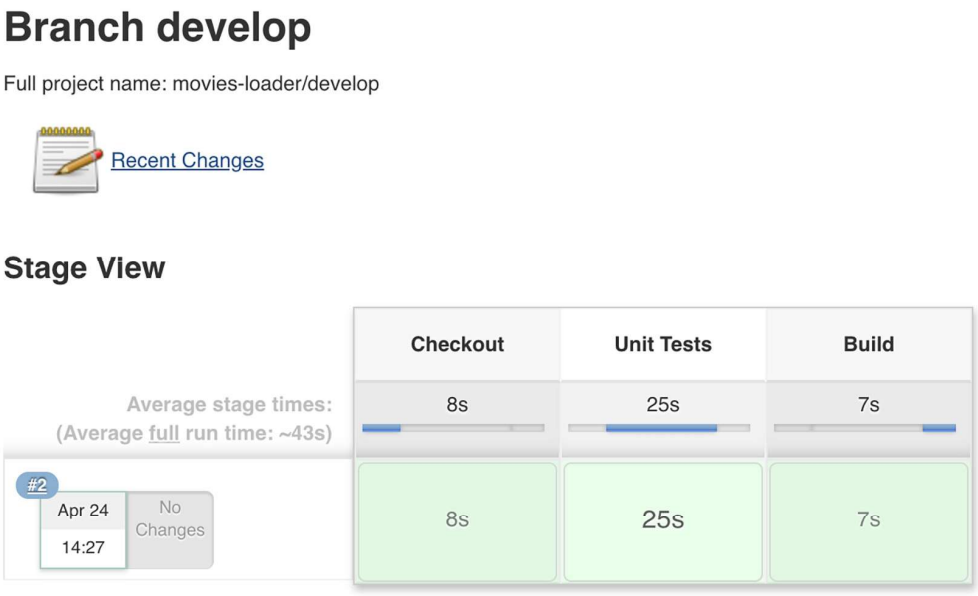

Figure 9.5 Movie loader CI pipeline

So far, we've defined the CI stages in figure 9.5 for the movies-loader CI pipeline. The movies-parser service's Dockerfile will be different, as it's written in Go. Because Go is a compiled language, we won't need it at the runtime of the service. Therefore, we will use Docker's multistage build feature to reduce the Docker image size, as shown in the following listing.

Listing 9.2 Multistage build usage

```
FROM golang:1.16.5
WORKDIR /go/src/github.com/mlabouardy/movies-parser
COPY main.go go.mod .
RUN go get -v
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app main.go

FROM alpine:latest
LABEL Maintainer mlabouardy
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/mlabouardy/movies-parser/app .
CMD ["./app"]
```

The Dockerfile is split into two stages. The first stage builds the binary with the `go build` command. The second stage uses Alpine as the base image, which is a lightweight image, and then copies the binary from the first stage.

The intermediate layer where the Go build tools and compilation happen is about 300 MB. The final image has a minimal footprint of 8 MB. The end result is the same tiny production image as before, with a significant reduction in complexity. The Go SDK and any intermediate artifacts are left behind and not saved in the final image.

NOTE The multistage build feature requires Docker engine 17.05 or higher on the daemon and client.

In the previous Dockerfile, stages are not named and are referred to by their integer number (starting with 0 for the first `FROM` instruction). However, we can name the stages by passing `AS` *`NAME`* to the `FROM` instruction, as shown in the following listing.

Listing 9.3 Naming Docker multistages

```
FROM golang:1.16.5 AS builder
WORKDIR /go/src/github.com/mlabouardy/parser
...
FROM alpine:latest
...
COPY --from=builder /go/src/github.com/mlabouardy/movies-parser/app .
```

Add the `Build` stage to the project Jenkinsfile, and push the changes to the develop branch. The pipeline will be triggered, and the result of the build should be similar to the one shown in figure 9.6.

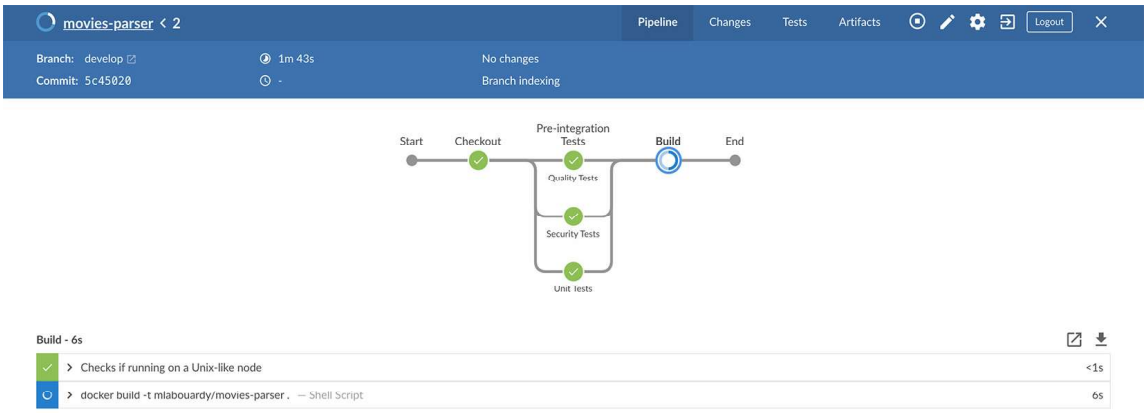

Figure 9.6 Movie parser CI pipeline

NOTE You could have just as easily based the final image on scratch or distroless images, but I prefer to have the convenience of Alpine. Plus, it's a safe choice for reducing image size.

The movies-store Docker image will use the Node.js base image from DockerHub; we're using the latest LTS node release at the time of writing. I prefer to name a specific version, rather than one of the floating tags like `node:lts` or `node:latest`, so that if you or someone else builds this image on a different machine, they will get the same version, rather than risking an accidental upgrade and attendant head-scratching.

NOTE In most cases, the best choice for a base image is from the official images available in DockerHub (https://hub.docker.com/). They tend to be better controlled than those created by the community.

Then, we install the needed dependencies for runtime by passing `--only=prod`. Finally, we set the `npm start` command to start the express server when the container is created, as shown in the following listing.

Listing 9.4 Movie store's Dockerfile

```
FROM node:14.17.0
WORKDIR /app
COPY package-lock.json package.json .
RUN npm i --only=prod
COPY index.js dao.js ./
EXPOSE 3000
CMD npm start
```

Note that, rather than copying the entire working directory, we are copying only the package.json and package-lock.json files. This allows us to take advantage of cached Docker layers. The package-lock.json file records the versions of all dependencies to ensure that the `npm install` command in Docker builds is consistent.

Once the pipeline changes are versioned and the execution is completed, the CI pipeline so far for movies-store should look similar to the Blue Ocean view in figure 9.7.

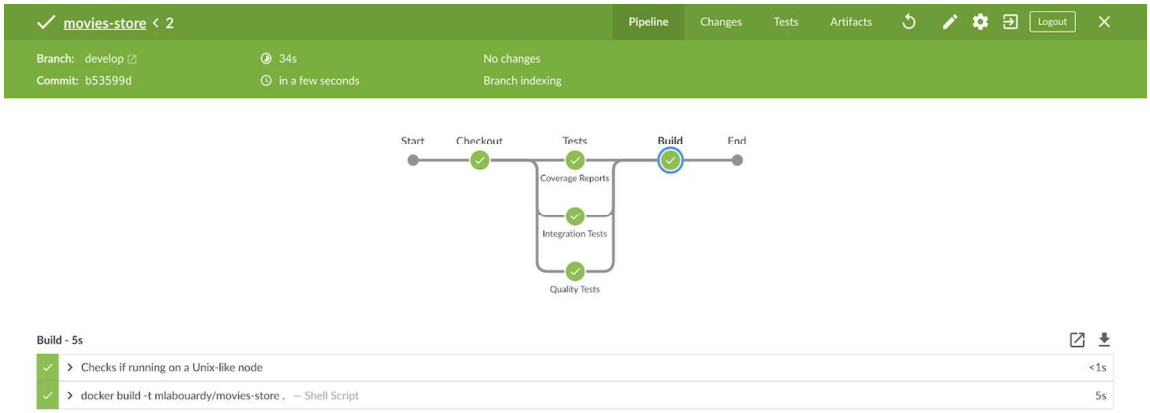

Figure 9.7 Movie store CI pipeline

NOTE During image build, Docker takes all files in the context directory. To increase the Docker build performance, exclude files and directories by adding a .dockerignore file to the context directory.

9.1.2 Docker build arguments

Finally, for the Angular application (aka movies-marketplace), we will once again use the multistage build feature to build the static folder with the `ng build` command. Then we'll copy the folder to an NGINX image to serve the content with a web server; see the following listing.

Listing 9.5 Movie marketplace's Dockerfile

```
FROM node:14.17.0 as builder
ARG ENVIRONMENT
ENV CHROME_BIN=chromium
WORKDIR /app
RUN apt-get update && apt-get install -y chromium
COPY package-lock.json package.json .
RUN npm i && npm i -g @angular/cli
COPY . .
RUN ng build -c $ENVIRONMENT

FROM nginx:alpine
RUN rm -rf /usr/share/nginx/html/*
COPY --from=builder /app/dist /usr/share/nginx/html
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

NOTE The `ENV` instruction is available during build and runtime. The `ARG` instruction (listing 9.5) is accessible only during build time.

Because we might have multiple Angular configurations (with different settings) based on the running environment, we will inject a build argument during the build time to specify the target environment as follows:

```
stage('Build'){
        docker.build(imageName, '--build-arg ENVIRONMENT=sandbox .')
}
```

When passing arguments to the `build()` method, the last value should end with the folder to use as the build context.

Finally, make sure to create a .dockerignore file in the root folder of the project to prevent local modules, debug logs, and temporary files from being copied into the Docker image. To exclude those directories, we create a .dockerignore file with the following content:

```
nodes_modules
coverage
dist
tmp
```

After pushing the changes, the pipeline should look like the Blue Ocean view in figure 9.8.

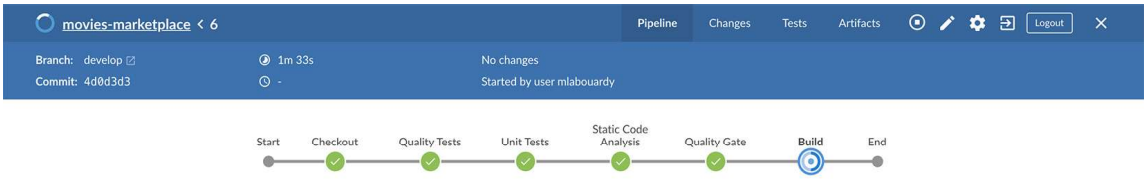

Figure 9.8 Movie marketplace CI pipeline

Now that the project Docker images are built, we need to store them somewhere. Therefore, we will deploy a private registry on which we will store all the images built through the development cycle of the project.

9.2 Deploying a Docker private registry

Continuous integration results in frequent builds and packages. Hence, we need a mechanism to store all this binary code (builds, packages, third-party plugins, and so on) in a system akin to a version-control system. Since VCSs such as Git and SVN store code and not binary files, we need a binary repository tool.

Many solutions exist, such as Nexus or Artifactory. However, they come with challenges including managing and hardening the instance. Fortunately, managed solutions also exist, depending on the cloud provider you're using, such as Amazon Elastic Container Registry (ECR), Google Container Registry, and Azure Container Registry.

NOTE You can also host your Docker images in DockerHub. If you go with this approach, you can skip this part.

9.2.1 Nexus Repository OSS

Nexus Repository OSS (www.sonatype.com/products/repository-oss) is a widely used open source, free artifact repository that can be used to store binaries and build artifacts. It can be used to distribute Maven/Java, npm, Helm, Docker, and more.

NOTE Since you're already familiar with Docker, you can run Nexus Repository OSS in a Docker container by using the Docker image from Sonatype.

To deploy Nexus Repository OSS, we need to bake a new machine image with Packer. The following listing provides the template.json content (the full template is available in chapter9/nexus/packer/template.json).

Listing 9.6 Nexus Repository OSS Packer template

```
{
    "variables" : {...},
    "builders" : [
        {
            "type" : "amazon-ebs",
            "ami_name" : "nexus-3.22.1-02",
            "ami_description" : "Nexus Repository OSS"
        }
    ],
    "provisioners" : [
        {
            "type" : "file",
            "source" : "./nexus.rc",
            "destination" : "/tmp/nexus.rc"
        },
        {
            "type" : "file",
```

```
            "source" : "./repository.json",
            "destination" : "/tmp/repository.json"
        },
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

This will create a temporary instance based on the Amazon Linux image and provision it with a shell script (listing 9.7) that installs the Nexus OSS version from the official repository and configures it to run a service with init.d, so it restarts after the instance reboots. This example uses version 3.30.1-01. The full script is available in chapter9/nexus/packer/setup.sh.

Listing 9.7 Installing the Nexus Repository OSS version (setup.sh)

```
NEXUS_USERNAME="admin"
NEXUS_PASSWORD="admin123"
echo "Install Java JDK 8"
yum update -y
yum install -y java-1.8.0-openjdk
echo "Install Nexus OSS"
wget https://download.sonatype.com/nexus/3/latest-unix.tar.gz -P /tmp
tar -xvf /tmp/latest-unix.tar.gz
mv nexus-* /opt/nexus
mv sonatype-work /opt/sonatype-work
useradd nexus
chown -R nexus:nexus /opt/nexus/ /opt/sonatype-work/
ln -s /opt/nexus/bin/nexus /etc/init.d/nexus
chkconfig --add nexus
chkconfig --levels 345 nexus on
mv /tmp/nexus.rc /opt/nexus/bin/nexus.rc
echo "nexus.scripts.allowCreation=true" >> nexus-default.properties
systemctl enable nexus
Systemctl start nexus
```

Defines Nexus OSS default credentials (admin/admin123)

Installs Java JDK 1.8.0, which is required to run Nexus OSS

Downloads Nexus OSS from the official repository and extracts the archive to the target

Then, the script will start Nexus server with the `service nexus restart` command and wait for it to be up and ready, as shown in the following listing.

Listing 9.8 Waiting for the Nexus server to be up (setup.sh)

```
until $(curl --output /dev/null
--silent --head --fail http://localhost:8081); do
    printf '.'
    sleep 2
done
```

Once the server responds, a POST request will be issued to the Nexus Script API to create a Docker hosted repository. The scripting API can be used to automate the creation of complex tasks for the Nexus Repository Manager, as shown next.

Listing 9.9 Nexus OSS script API (setup.sh)

```
curl -v -X POST -u $NEXUS_USERNAME:$NEXUS_PASSWORD
--header "Content-Type: application/json" 'http://localhost:8081/service/
     rest/v1/script'
-d @/tmp/repository.json
```

Performs a POST request on the Nexus server by including the default credentials in the request and the Docker repository config in the request payload

NOTE A comprehensive listing of Nexus REST API endpoints and functionality is documented through the NEXUS_HOST/swagger-ui endpoint.

The request payload is a Groovy script that exposes a Docker hosted registry on port 5000:

```
import org.sonatype.nexus.blobstore.api.BlobStoreManager;
import org.sonatype.nexus.repository.storage.WritePolicy;
repository.createDockerHosted('docker-registry',
5000, 443,
BlobStoreManager.DEFAULT_BLOBSTORE_NAME, true, true, WritePolicy.ALLOW, true)
```

Issue the `packer build` command to bake the AMI. Once the provisioning is finished, the Nexus AMI should be available in the Images section in the AWS Management Console, as shown in figure 9.9.

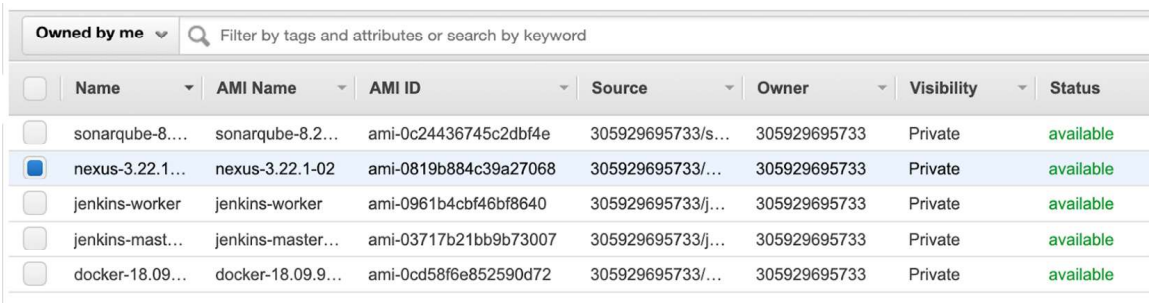

Figure 9.9 Nexus OSS AMI

From there, use Terraform to provision an EC2 instance based on the baked Nexus OSS AMI. Create a nexus.tf file with the content in the following listing.

Listing 9.10 Nexus EC2 instance resource

```
resource "aws_instance" "nexus" {
  ami                    = data.aws_ami.nexus.id
  instance_type          = var.nexus_instance_type
  key_name               = var.key_name
  vpc_security_group_ids = [aws_security_group.nexus_sg.id]
  subnet_id              = element(var.private_subnets, 0)
```

```
  root_block_device {
    volume_type           = "gp2"
    volume_size           = 50
    delete_on_termination = false
  }

  tags = {
    Author = var.author
   Name = "nexus"
  }
}
```

NOTE Running Nexus OSS without a problem requires a minimum of 8 GB of memory. Additionally, I strongly recommend using a dedicated EBS for blob storage (http://mng.bz/dr7Q).

Also, provision a public load balancer to forward incoming HTTP and HTTPS traffic to port 8081 of the EC2 instance, which is the port where the Nexus Repository Manager (dashboard) is exposed. Create a new file, loadbalancers.tf, with the following listing.

Listing 9.11 Nexus Repository Manager public load balancer

```
resource "aws_elb" "nexus_elb" {
  subnets                     = var.public_subnets
  cross_zone_load_balancing = true
  security_groups             = [aws_security_group.elb_nexus_sg.id]
  instances                   = [aws_instance.nexus.id]

  listener {
    instance_port      = 8081
    instance_protocol  = "http"
    lb_port            = 443
    lb_protocol        = "https"
    ssl_certificate_id = var.ssl_arn
  }

  health_check {
    healthy_threshold   = 2
    unhealthy_threshold = 2
    timeout             = 3
    target              = "TCP:8081"
    interval            = 5
  }

  tags = {
    Name   = "nexus_elb"
    Author = var.author
  }
}
```

Within the same file, add another public load balancer, as shown in the next listing. This will access the Docker private registry pointing to port 5000 of the hosted repository on the Nexus Repository Manager.

Listing 9.12 Docker registry public load balancer

```
resource "aws_elb" "registry_elb" {
  subnets                     = var.public_subnets
  cross_zone_load_balancing   = true
  security_groups             = [aws_security_group.elb_registry_sg.id]
  instances                   = [aws_instance.nexus.id]

  listener {
    instance_port      = 5000
    instance_protocol  = "http"
    lb_port            = 443
    lb_protocol        = "https"
    ssl_certificate_id = var.ssl_arn
  }
}
```

Use `terraform apply` to provision the AWS resources, the Nexus dashboard, and Docker Registry. URLs should be displayed at the end of the provisioning process in the `Outputs` section, as shown in figure 9.10.

```
aws_route53_record.nexus: Creating...
aws_route53_record.registry: Still creating... [10s elapsed]
aws_route53_record.nexus: Still creating... [10s elapsed]
aws_route53_record.registry: Still creating... [20s elapsed]
aws_route53_record.nexus: Still creating... [20s elapsed]
aws_route53_record.registry: Still creating... [30s elapsed]
aws_route53_record.nexus: Still creating... [30s elapsed]
aws_route53_record.registry: Creation complete after 34s [id=Z2TR95QTU3UIUT_registry.slowcoder.com_A]
aws_route53_record.nexus: Creation complete after 34s [id=Z2TR95QTU3UIUT_nexus.slowcoder.com_A]

Apply complete! Resources: 8 added, 0 changed, 0 destroyed.

Outputs:

nexus = https://nexus.slowcoder.com
registry = https://registry.slowcoder.com
```

Figure 9.10 Nexus Terraform resources

Point your favorite browser to the Nexus URL, and the web dashboard in figure 9.11 should be displayed. The default `admin` password can be found in /opt/sonatype-work/nexus3/admin.password.

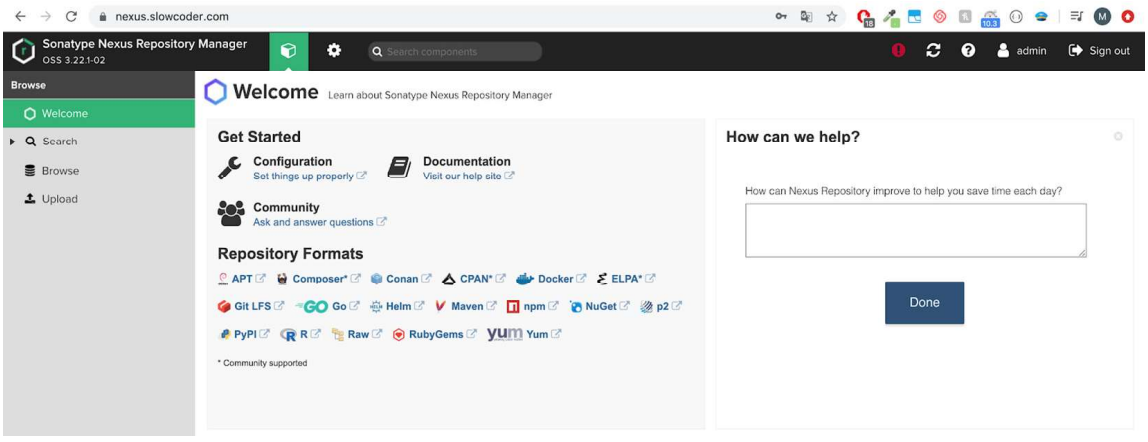

Figure 9.11 Nexus Repository Manager

If you jump to Settings from the cogwheel icon and then Repositories, a new Docker hosted repository should be created. The repository disables tag immutability and allows image tags to be overwritten by a subsequent image push using the same tag. If this option is enabled, an error will be returned if you attempt to push an image with a tag that already exists in the repository. The rest of the configurations should be similar to figure 9.12.

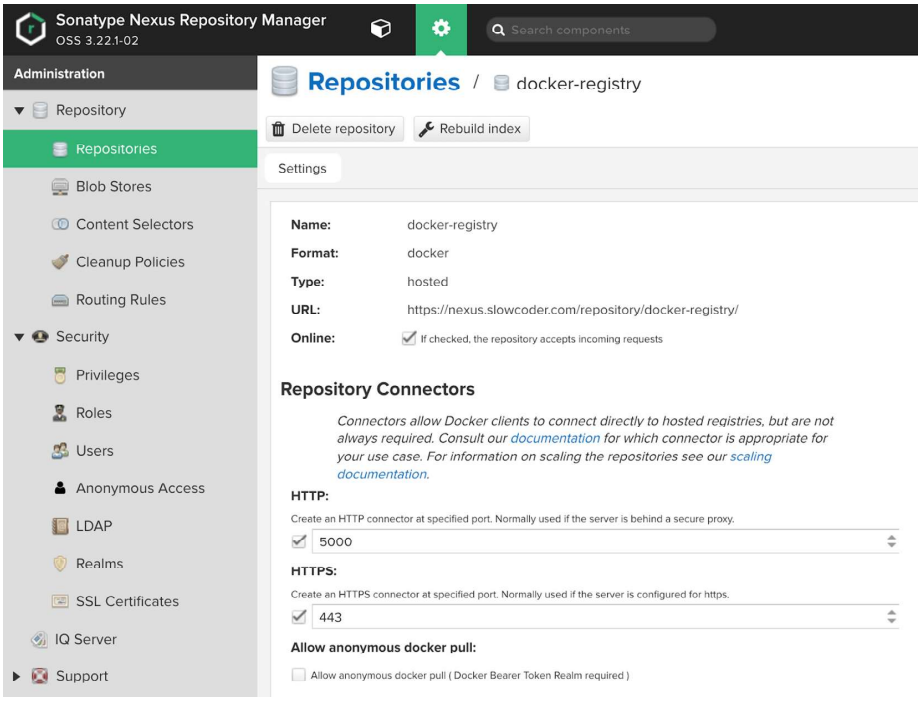

Figure 9.12 Docker-hosted registry on Nexus

To be able to pull and push Docker images to the registry, we will create a custom Nexus role from the Security section. This role, shown in figure 9.13, will give full access to the Docker hosted registry.

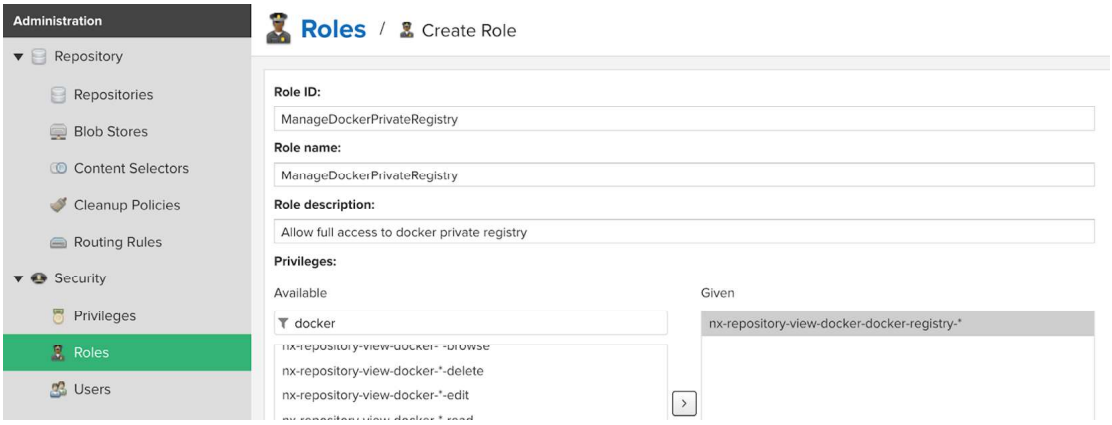

Figure 9.13 Nexus custom role for the Docker registry

NOTE For push and pull operations, only `nx-*-registry-add` and `nx-*-registry-read` permissions are required.

Next, we create a Jenkins user and assign to it the custom Nexus role we just created, as shown in figure 9.14.

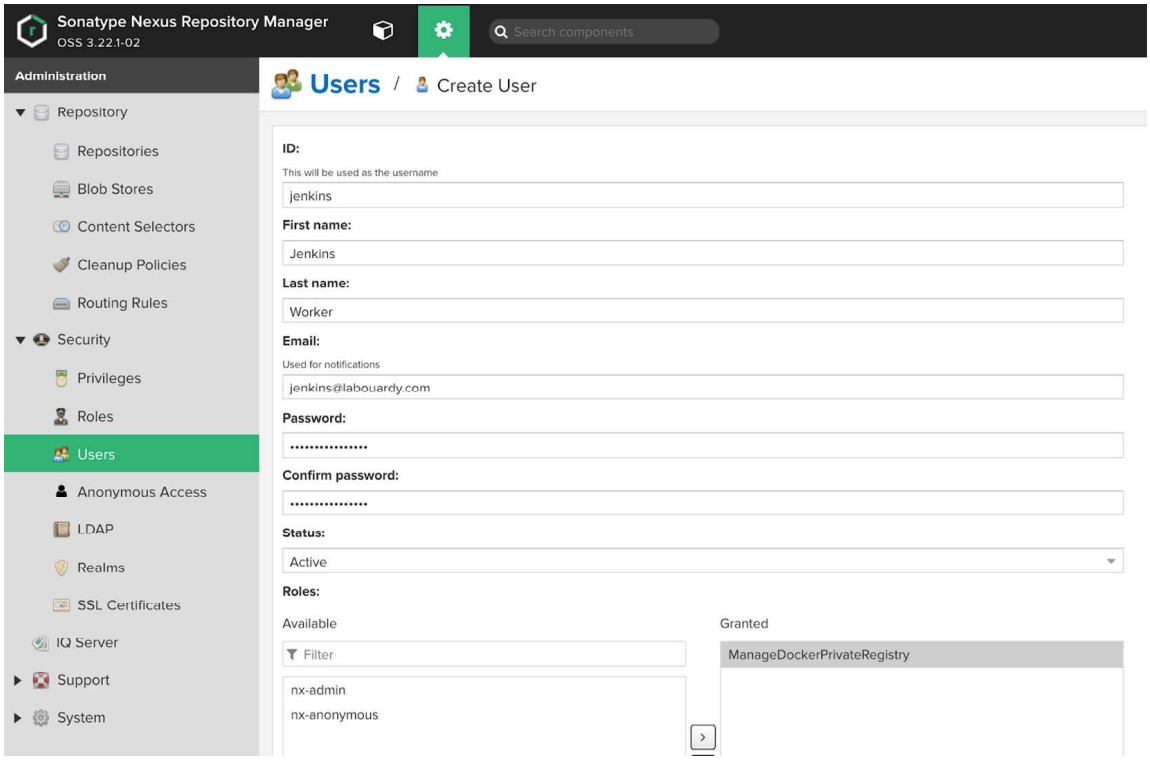

Figure 9.14 Docker registry credentials for Jenkins

We can test out the authentication by jumping back to the terminal session on the local machine and issuing the `docker login` command:

```
jenkins:terraform mlabouardy$ docker login https://registry.slowcoder.com
Username: jenkins
Password:
Login Succeeded
```

NOTE The hosted Docker repository is exposed on HTTPS by default. However, if you expose the private repository on a plain HTTP endpoint only, you need to configure the Docker daemon to allow insecure connections by passing the `–insecure-registry` flag to the Docker engine.

Finally, on Jenkins, create a registry credential of type Username with Password with the Nexus credentials we created so far for Jenkins (figure 9.15).

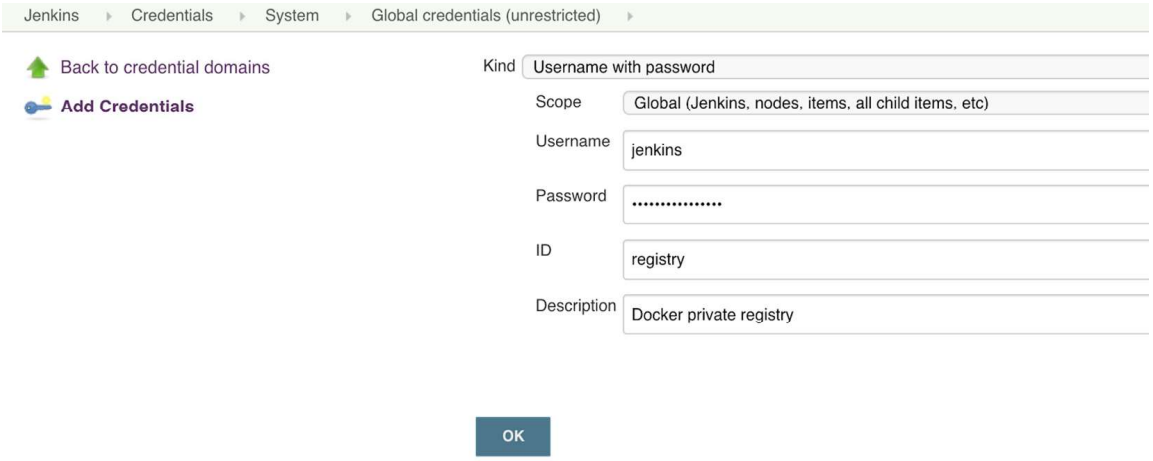

Figure 9.15 Docker registry credentials

Another alternative to Nexus Repository OSS is an AWS managed service.

9.2.2 *Amazon Elastic Container Registry*

If you're using AWS, as I am, you can use a managed AWS service called Elastic Container Registry (ECR) to host your private Docker images. From the AWS Management Console, navigate to Amazon ECR (https://console.aws.amazon.com/ecr/repositories). Then, create a repository for each Docker image you want to host or store. In our project, we need to create four repositories, one for each microservice. The service-loader repository, for instance, is shown in figure 9.16.

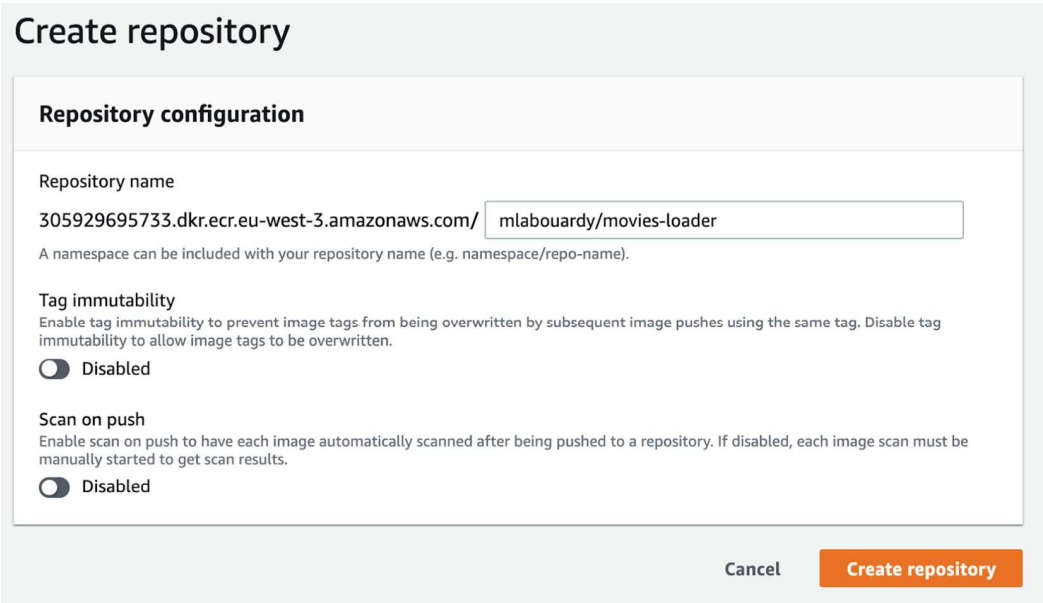

Figure 9.16 ECR new repository

Once the repository is created, you can click the View Push Commands button, and a dialog should pop up with a list of instructions on how to tag, push, and pull images to the remote repository; see figure 9.17.

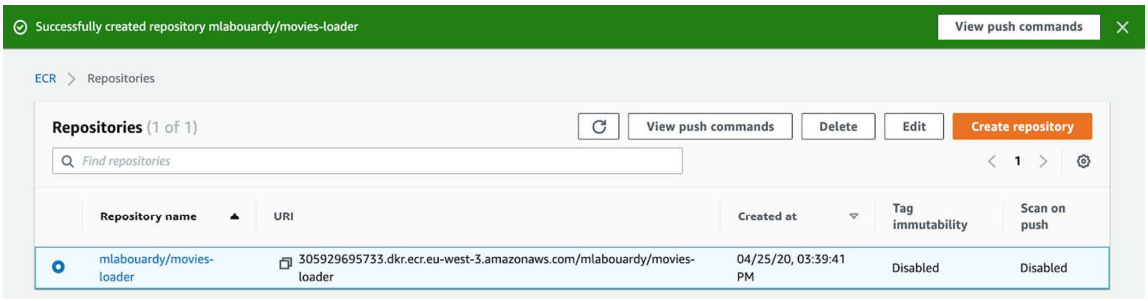

Figure 9.17 Movie loader ECR repository

Before interacting with the repository, you need to authenticate with ECR. The following command for Mac and Linux users can be used to log in to the remote repository:

```
aws ecr get-login-password --region REGION
| docker login --username AWS --password-stdin
ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/
mlabouardy/movies-loader
```

> **NOTE** Replace `ACCOUNT_ID` and `REGION` with your Amazon account ID and AWS region, respectively.

For Windows users, here is the command:

```
(Get-ECRLoginCommand).Password |
docker login --username AWS --password-stdin
ACCOUNT_ID.dkr.ecr.REGION.amazonaws.com/mlabouardy/movies-loader
```

Repeat the same procedure to create dedicated ECR repositories per microservice, as shown in figure 9.18.

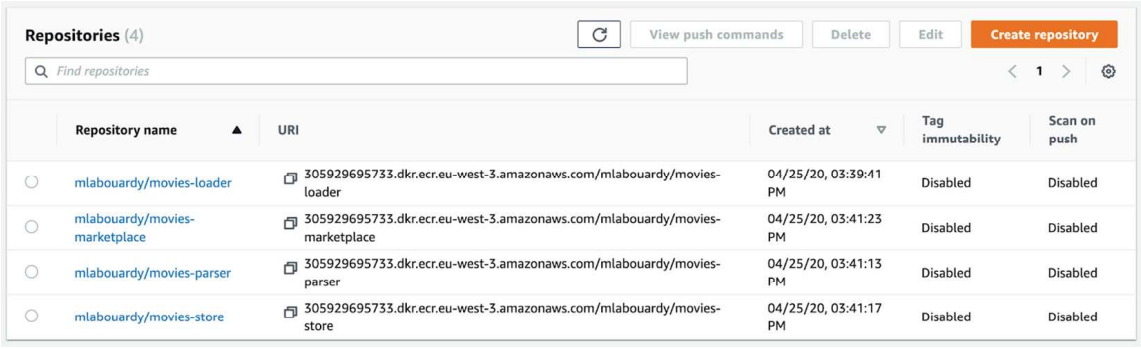

Figure 9.18 ECR repository for each microservice

9.2.3 Azure Container Registry

For Azure users, the Azure Container Registry service can be used to store container images without managing a private registry. On the Azure portal (https://portal.azure.com/), navigate to the Container Registries service and click the Add button to create a new registry. Specify the region where you want to deploy the registry and give it a name, as shown in figure 9.19.

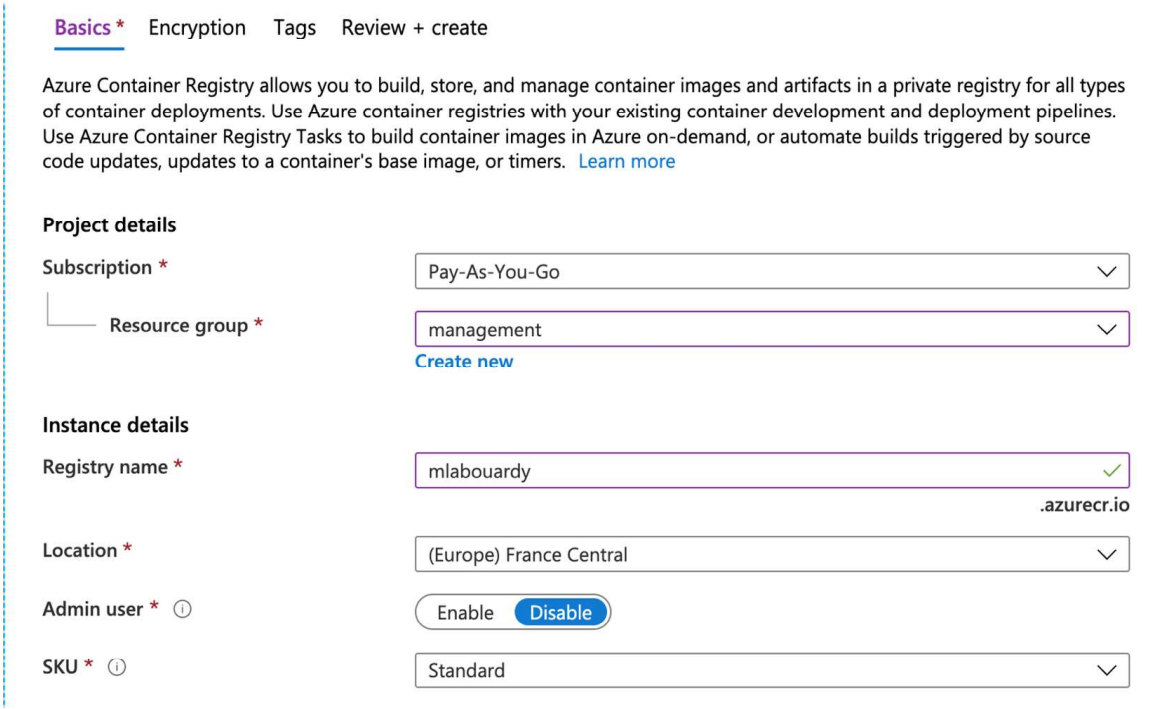

Figure 9.19 Azure new registry configuration

Leave other fields at the defaults and click Create. Once the registry is created, navigate to Access Keys under the Settings section, where you will find the admin username and password that you can use to authenticate to the registry to push or pull Docker images from Jenkins; see figure 9.20.

You can use those credentials in Jenkins to push the image within the CI pipeline. However, I recommend creating a token with granular access control by using role-based access control (RBAC), or the least privilege principle. The admin account is designed for only a single user to access the registry, mainly for testing purposes.

Navigate to the Tokens section and click the Add button to create a new access token. Give it a name and associate the `_repositories_push` scope to allow the execution of the `docker push` operation only (Jenkins will need to push only images to the registry); see figure 9.21.

Generate a password after you have created a token, as shown in figure 9.22. To authenticate with the registry, the token must be enabled and have a valid password.

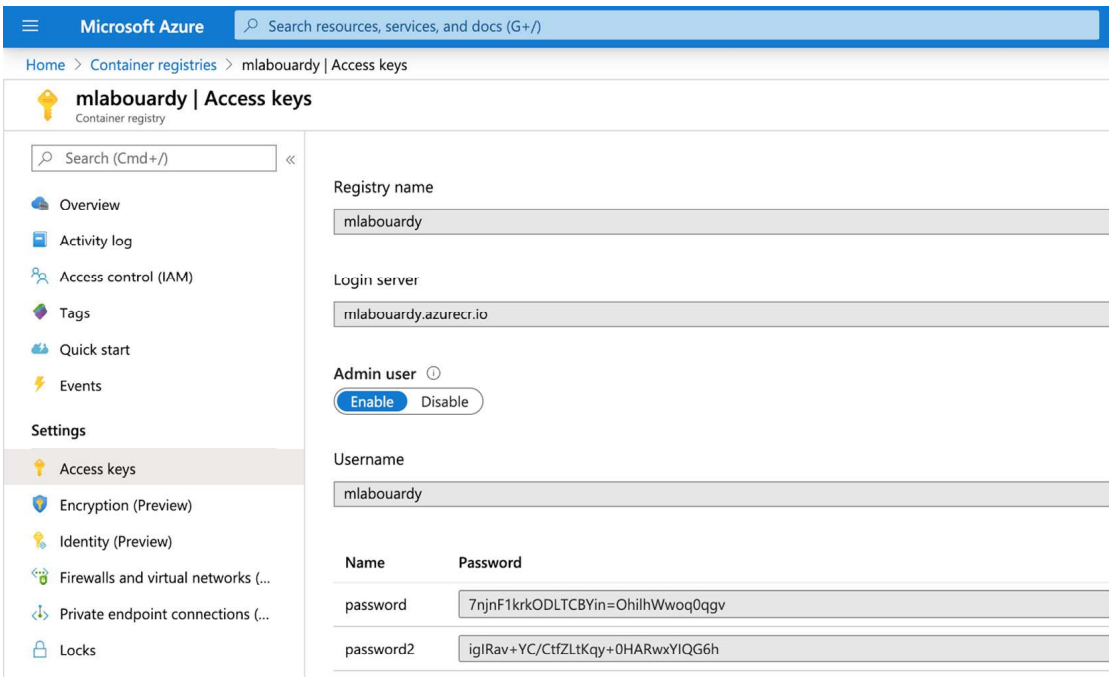

Figure 9.20 Azure Docker registry admin credentials

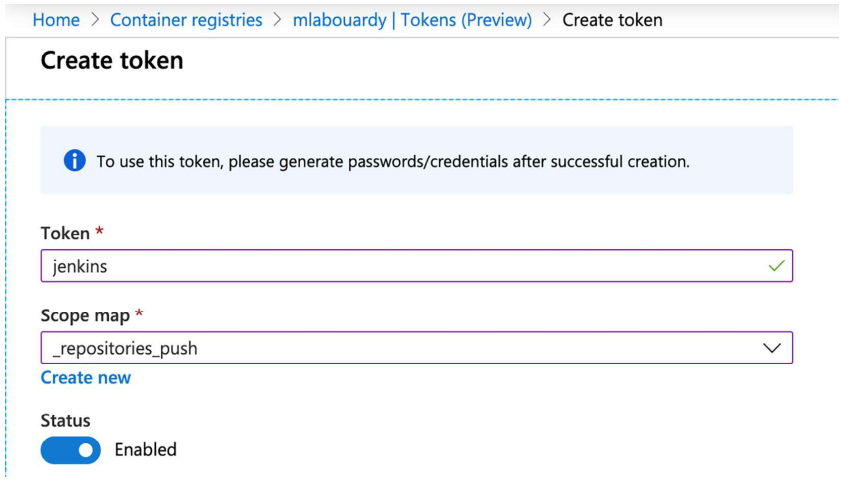

Figure 9.21 Azure Docker registry new access token

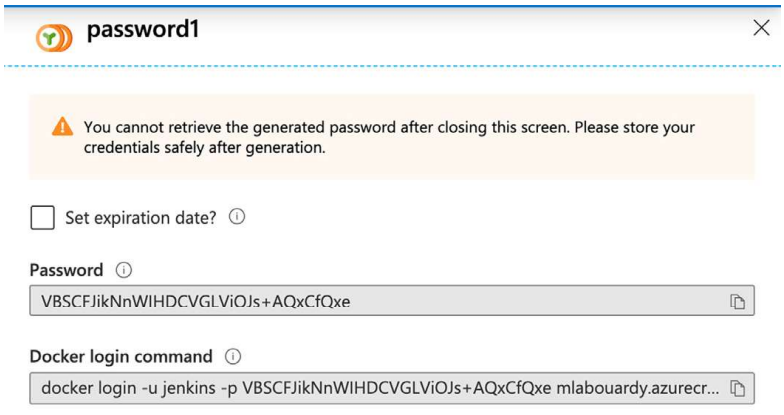

Figure 9.22 Azure Docker registry credentials

After generating a password, copy and save it as Jenkins credentials of type Username with Password. You can't retrieve a generated password after closing the dialog screen, but you can generate a new one.

9.2.4 Google Container Registry

For Google Cloud Platform users, a managed service called Google Container Registry (GCR) can be used to host Docker images. To get started, you need to enable API Container Registry (https://cloud.google.com/container-registry/docs/quickstart) for your GCP project and then install the `gcloud` command line. For Linux users, run the following listing.

Listing 9.13 `gcloud` installation

```
curl -O https://dl.google.com/dl/cloudsdk/channels/
rapid/downloads/google-cloud-sdk-344.0.0-linux-x86_64.tar.gz
tar zxvf google-cloud-sdk-344.0.0-linux-x86_64.tar.gz
 google-cloud-sdk
./google-cloud-sdk/install.sh
```

> **NOTE** For further instructions on how to install the Google Cloud SDK, read the official GCP guide at https://cloud.google.com/sdk/install.

Next, issue the following command to authenticate with the registry. The resulting authentication token is persisted in ~/.docker/config.json and reused for any subsequent interactions against that repository:

```
gcloud auth configure-docker
```

You need to tag the target images with the GCR URI (`gcr.io/[PROJECT-ID]`) and push the images with the `docker push` command. Figure 9.23 shows how to tag and push the movies-loader Docker image to GCR:

```
docker tag mlabouardy/movies-loader
eu.gcr.io/PROJECT_ID/mlabouardy/movies-loader
docker push eu.gcr.io/PROJECT_ID/mlabouardy/movies-loader
```

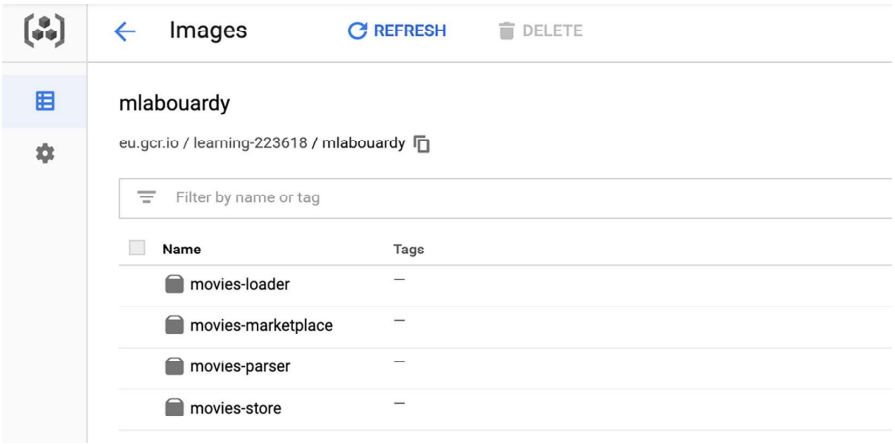

Figure 9.23 Google Container Registry images

Now that we've covered how to deploy a private Docker registry, we will update the Jenkinsfile for each service to push the image to the remote private registry at the end of a successful CI pipeline execution.

9.3 Tagging Docker images the right way

Add a new push stage to the Jenkinsfile with the `withRegistry` block, which authenticates against the registry URL provided in the first parameter by using the credentials provided in the second parameter. Then it persists the changes in ~/.docker/config.json. Finally, it pushes the image with a tag value equal to the build number ID (using the `env.BUILD_ID` keyword). The following listing is the Jenkinsfile for the movies-loader service after implementing the `Push` stage.

Listing 9.14 Publishing Docker image to a registry

```
def imageName = 'mlabouardy/movies-loader'
def registry = 'https://registry.slowcoder.com'
node('workers'){
    stage('Checkout'){
        checkout scm
    }

    stage('Unit Tests'){
        def imageTest= docker.build("${imageName}-test",
"-f Dockerfile.test .")
        imageTest.inside{
            sh 'python test_main.py'
        }
    }

    stage('Build'){
        docker.build(imageName)
    }

    stage('Push'){
        docker.withRegistry(registry, 'registry') {
           docker.image(imageName).push(env.BUILD_ID)
        }
    }
}
```

NOTE The `imageName` and `registry` values must be replaced with your own Docker private registry URL and name of the image to store, respectively.

For this example, the build number is 2; therefore, the movies-loader image is pushed to the registry after tagging it with a tag equal to 2, as shown in figure 9.24.

```
[Pipeline] { (Push)
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withDockerRegistry
$ docker login -u jenkins -p ******** https://registry.slowcoder.com
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /home/ec2-user/workspace/movies-loader_
86edd538b844/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
[Pipeline] {
[Pipeline] isUnix
[Pipeline] sh
+ docker tag mlabouardy/movies-loader registry.slowcoder.com/mlabouardy/movies-loader:2
[Pipeline] isUnix
[Pipeline] sh
+ docker push registry.slowcoder.com/mlabouardy/movies-loader:2
The push refers to repository [registry.slowcoder.com/mlabouardy/movies-loader]
908027c5b2f4: Preparing
c8924bb9cb10: Preparing
59bc756f6273: Preparing
3a9cf82366b7: Preparing
```

Figure 9.24 Docker push command logs

If we head back to the registry (for example, on Nexus Repository Manager), we can see that a movies-loader image has been successfully pushed (figure 9.25).

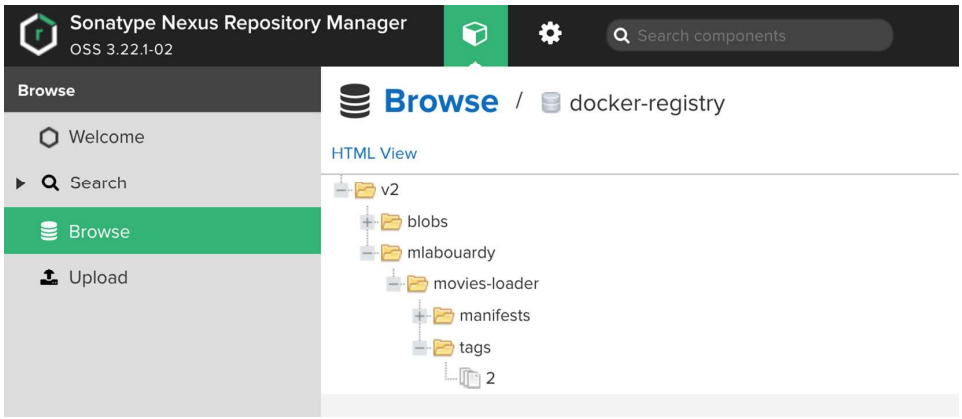

Figure 9.25 Docker image stored in Nexus

While the Jenkins build ID can be used to tag the images, it might not be handy. A better identifier is the Git commit ID. In this example, we will use it to tag the built Docker image. On a declarative and scripted pipeline, this information is not available out of the box. Therefore, we will create a function that uses the Git command line to fetch the commit ID and return it:

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

From there, we can update the `Push` stage to tag the image with the value returned by the `commitID()` function:

```
stage('Push'){
        docker.withRegistry(registry, 'registry') {
            docker.image(imageName).push(commitID())
        }
}
```

NOTE In chapter 14, we will cover how to create a Jenkins shared library with custom functions to avoid duplication of code in Jenkinsfiles.

Push the changes to the GitHub repository with the following commands:

```
git add Jenkinsfile
git commit -m "tagging docker image with git commit id"
git push origin develop
```

The new CI pipeline stages should look like figure 9.26 for the movies-loader service.

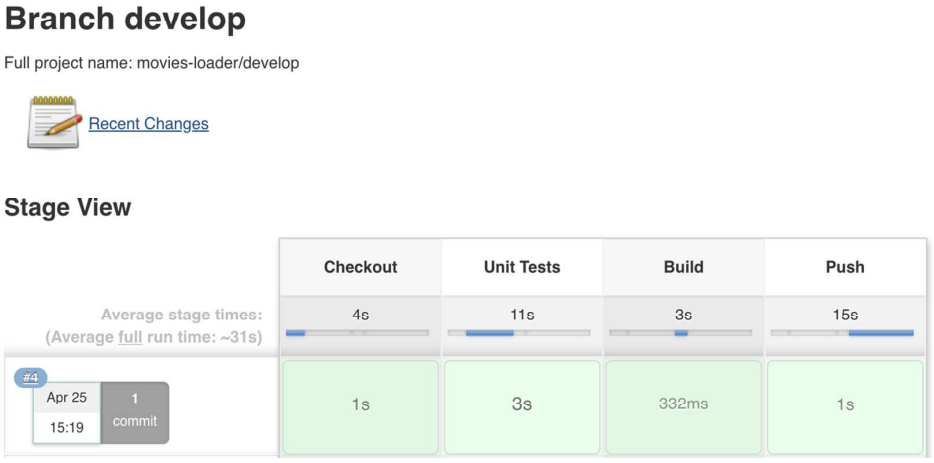

Figure 9.26 Movie loader CI pipeline

After a successful run on Nexus Repository Manager, a new image with a commit ID should be available (figure 9.27).

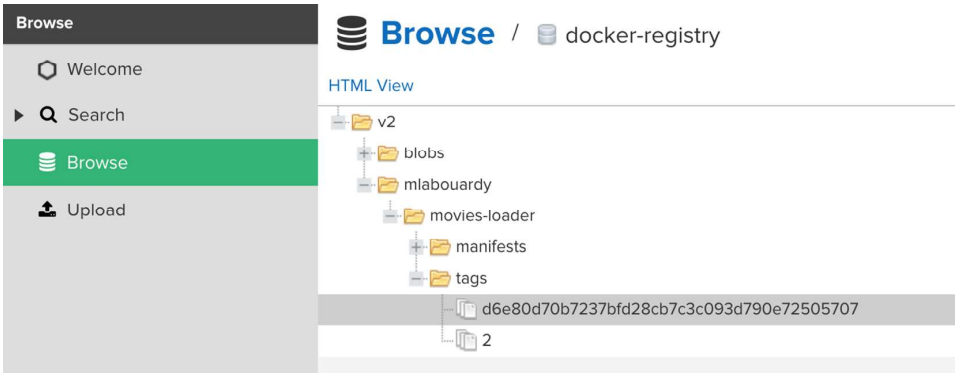

Figure 9.27 Commit ID image tag

We will take this further and push the same image with a tag based on the branch name. This tag will be helpful when we tackle continuous deployment and delivery. It will allow us to assign a particular tag per environment:

- *Latest*—Used to deploy the image to the production environment
- *Preprod*—Used to deploy the image to the staging or preproduction environment
- *Develop*—Used to deploy the image to the sandbox or development environment

The `Push` stage code block is as follows:

```
stage('Push'){
        docker.withRegistry(registry, 'registry') {
           docker.image(imageName).push(commitID())

            if (env.BRANCH_NAME == 'develop') {
                docker.image(imageName).push('develop')
            }
        }
}
```

The `env.BRANCH_NAME` variable contains the branch name. Also, you can just use `BRANCH_NAME` without the `env` keyword (it hasn't been required since Pipeline Groovy Plugin 2.18).

Lastly, if you're using Amazon ECR as a private registry, you need to authenticate first with the AWS CLI to the remote repository before issuing the push instructions. For AWS CLI 2 users, use the shell instruction in the following listing to invoke the `aws ecr` command.

Listing 9.15 Publishing the Docker image to ECR

```
def imageName = 'mlabouardy/movies-loader'
def registry = 'ACCOUNT_ID.dkr.ecr.eu-west-3.amazonaws.com'
def region = 'REGION'

node('workers'){
    ...
    stage('Push'){
        sh "aws ecr get-login-password --region ${region} |
docker login --username AWS
--password-stdin ${registry}/${imageName}"

        docker.image(imageName).push(commitID())
        if (env.BRANCH_NAME == 'develop') {
            docker.image(imageName).push('develop')
        }
    }
}
```

Make sure to substitute the `ACCOUNT_ID` and `REGION` variables with your own AWS account ID and AWS region, respectively. If you're using a 1.x version of the AWS CLI, use this code block instead:

```
stage('Push'){
        sh "\$(aws ecr get-login
--no-include-email --region ${region}) || true"
        docker.withRegistry("https://${registry}") {
            docker.image(imageName).push(commitID())
            if (env.BRANCH_NAME == 'develop') {
                docker.image(imageName).push('develop')
            }
        }
}
```

Before triggering the CI pipeline, you will need to give access to Jenkins workers to perform the push operation on the ECR registry. Therefore, you need to assign an IAM instance profile to Jenkins worker instances with the AmazonEC2ContainerRegistryFullAccess policy. Figure 9.28 illustrates the IAM instance profile assigned to Jenkins workers.

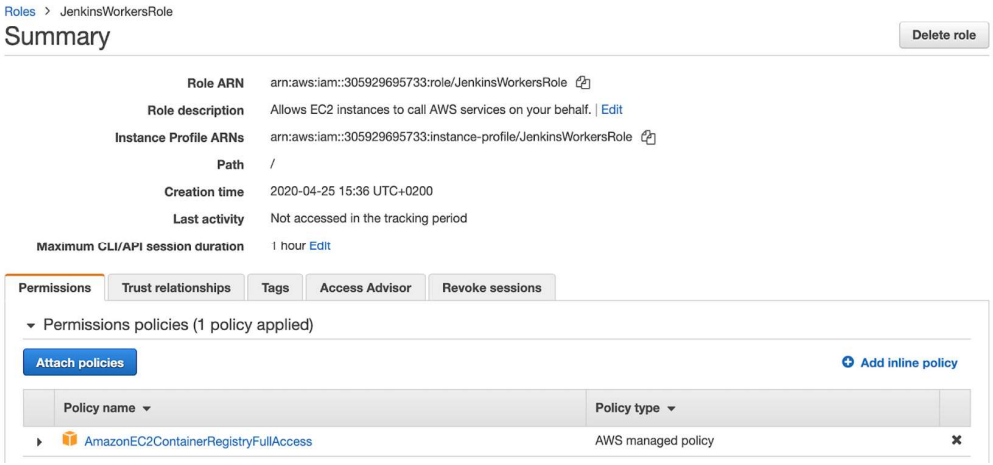

Figure 9.28 Jenkins workers' IAM instance profile

Once you've made the required changes, a new build should be triggered. A new image tag should be pushed to the ECR repository, at the end of the CI pipeline, as shown in figure 9.29.

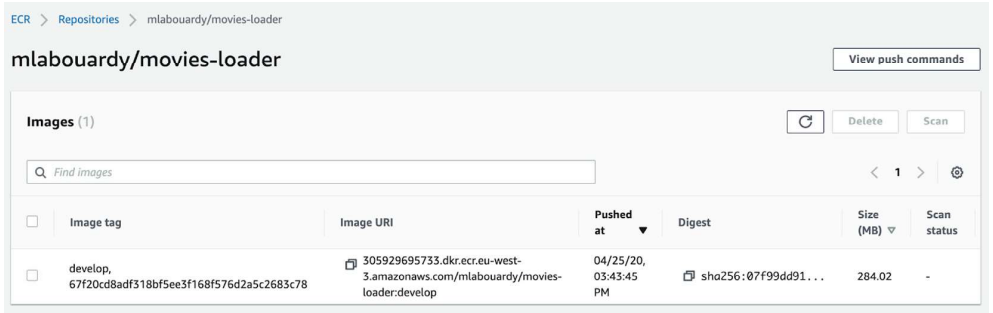

Figure 9.29 Movie loader ECR repository images

Repeat the same procedure for the rest of the microservices, to push their Docker image to the end of the CI pipeline, as shown in figure 9.30.

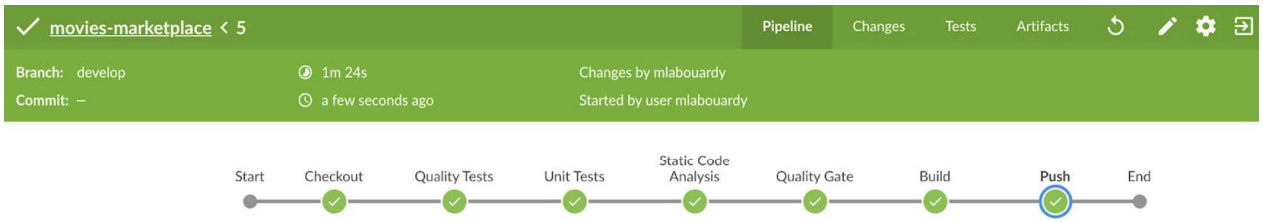

Figure 9.30 Movie marketplace CI pipeline

In a typical workflow, the Docker images should be analyzed, inspected, and scanned against security rules for compliance and auditing. That's why, in the upcoming section, we will integrate a container inspection and analytics platform within the CI pipeline to continuously inspect built Docker images for security vulnerabilities.

9.4 Scanning Docker images for vulnerabilities

Anchore Engine (https://github.com/anchore/anchore-engine) is an open source project that provides a centralized service for inspection, analysis, and certification of container images. You can run Anchore Engine as a standalone service or as a Docker container.

> **NOTE** A standalone installation will require at least 4 GB of RAM and enough disk space available to support the container images you intend to analyze.

You can bake your own AMI with Packer from scratch to install Anchore Engine and set up the PostgreSQL database. Then, use Terraform to deploy the stack, or you can simply deploy the configured stack out of the box with Docker Compose. Refer to chapters 4 and 5 for instructions on how to use Terraform and Packer.

Launch a private instance in the *management* VPC with Docker Community Edition (CE) pre-installed, and then install the Docker Compose tool from the Docker official guide page. Issue the following command to deploy Anchore Engine:

```
curl https://docs.anchore.com/current/docs/
engine/quickstart/docker-compose.yaml > docker-compose.yaml
docker-compose up -d
```

After a few moments, your Anchore Engine services should be up and running, ready to use. You can verify that the containers are running with the `docker-compose ps` command. Figure 9.31 shows the output. Make sure to allow inbound traffic on port 8228 (Anchore API) from the Jenkins master security group ID only, as shown in figure 9.32.

```
        Name                      Command                  State                Ports
-------------------------------------------------------------------------------------------
ec2-user_analyzer_1        /docker-entrypoint.sh anch ...   Up (health: starting)   8228/tcp
ec2-user_api_1             /docker-entrypoint.sh anch ...   Up (health: starting)   0.0.0.0:8228->8228/tcp
ec2-user_catalog_1         /docker-entrypoint.sh anch ...   Up (health: starting)   8228/tcp
ec2-user_db_1              docker-entrypoint.sh postgres    Up                      5432/tcp
ec2-user_policy-engine_1   /docker-entrypoint.sh anch ...   Up (health: starting)   8228/tcp
ec2-user_queue_1           /docker-entrypoint.sh anch ...   Up (health: starting)   8228/tcp
```

Figure 9.31 Docker Compose stack services

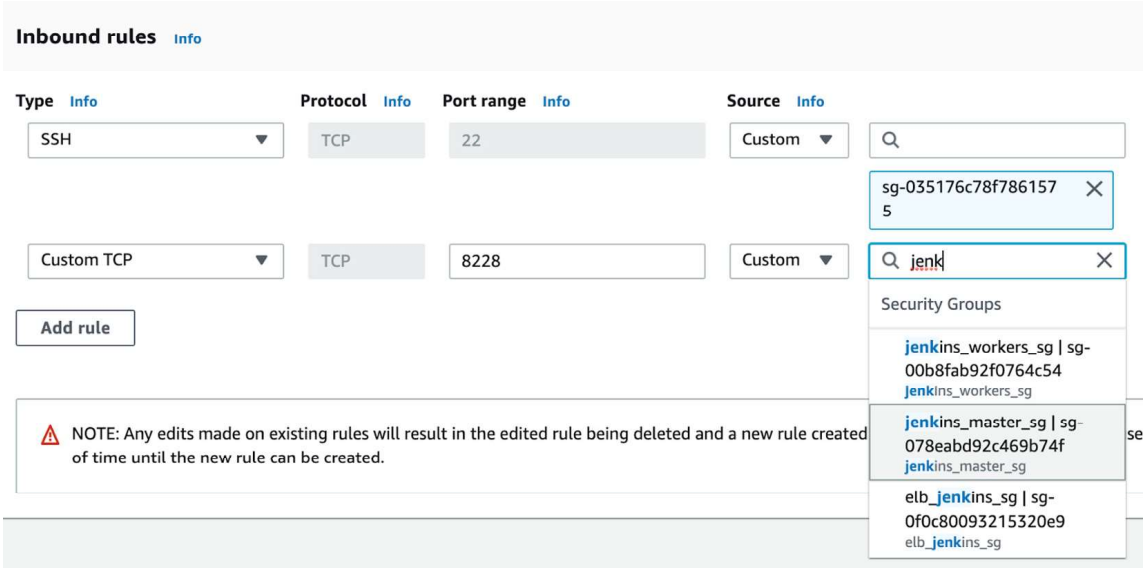

Figure 9.32 Anchore instance's security group

> **NOTE** You can take this further and deploy a load balancer in front of the EC2 instance and create an A record in Route 53 pointing to the load balancer FQDN.

When it comes to Jenkins, an available plugin already makes the integration much easier. From the main Jenkins menu, select Manage Jenkins and jump to the Manage Plugins section. Click the Available tab and install the Anchore Container Image Scanner plugin, as shown in figure 9.33.

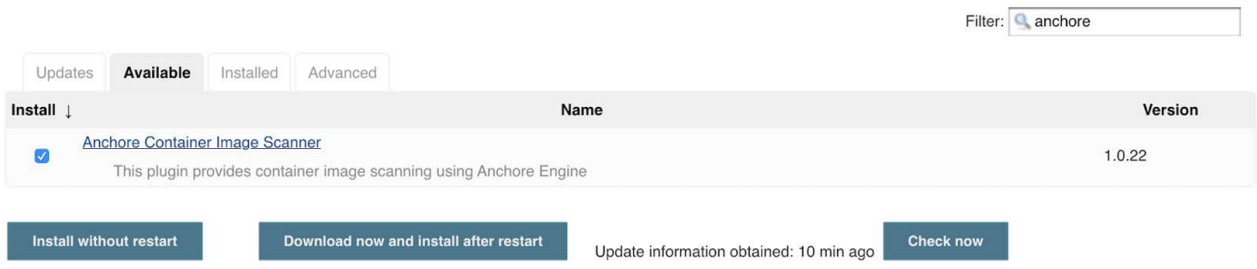

Figure 9.33 Anchore Container Image Scanner plugin

Next, from the Manage Jenkins menu, choose Configure System and scroll down to the Anchore Configuration. Then, set the Anchore URL with the /v1 route included and credentials (the default is admin/foobar), as shown in figure 9.34.

Figure 9.34 Anchore plugin configuration

Finally, integrate Anchore into the Jenkins pipeline by creating a file named *images* in the project workspace. This file should contain the name of the Docker image to be scanned and optionally include the Dockerfile. Then, call the Anchore plugin with the file created as a parameter, as shown in the following listing.

Listing 9.16 Analyzing Docker images with Anchore

```
stage('Analyze'){
            def scannedImage =
"${registry}/${imageName}:${commitID()}
${workspace}/Dockerfile"
            writeFile file: 'images', text: scannedImage
            anchore name: 'images'
}
```

Push the changes with the following commands to the remote repository on the develop branch:

```
git add Jenkinsfile
git commit -m "image scanning stage"
git push origin develop
```

The CI pipeline will be triggered upon the push event. After the image has been built and pushed to the registry, the Anchore Scanner should be called. It will throw an error due to Anchore not being able to pull the Docker image from the private registry for analysis and inspection.

Fortunately, Anchore integrates and supports analyzing images from any registry compatible with Docker v2. To allow access to the remote images from Anchore, install the `anchor-cli` binary from the Anchore EC2 instance:

```
yum install -y epel-release python-pip
pip install anchorecli
```

Next, we define credentials for the private Docker registry. Run this command; the `REGISTRY` parameter should include the registry's fully qualified hostname and port number (if exposed):

```
anchore-cli registry add REGISTRY USERNAME PASSWORD
```

> **NOTE** The same command can be used to configure a Docker registry hosted on Nexus or other solutions.

Since we're using Amazon ECR repositories and running Anchore from an EC2 instance, we will assign an IAM instance profile instead with the AmazonEC2ContainerRegistryReadOnly policy. In this case, we will pass `awsauto` for both `USERNAME` and `PASSWORD` and instruct the Anchore Engine to inherit the role from the underlying EC2 instance:

```
anchore-cli --u admin --p foobar registry add ACCOUNT_ID.dkr.ecr.REGION
     .amazonaws.com awsauto awsauto --registry-type=awsecr
```

To verify that credentials have been properly configured, run the following command to list the defined registries:

```
anchore-cli --u admin --p foobar registry list
```

```
[ec2-user@ip-10-0-0-229 ~]$ anchore-cli --u admin --p foobar registry add 305929695733.dkr.ecr.eu-west-3.amazonaws.com awsauto awsauto --registry-type=awsecr
Registry: 305929695733.dkr.ecr.eu-west-3.amazonaws.com
Name: 305929695733.dkr.ecr.eu-west-3.amazonaws.com
User: awsauto
Type: awsecr
Verify TLS: True
Created: 2020-05-15T16:48:03Z
Updated: 2020-05-15T16:48:03Z

[ec2-user@ip-10-0-0-229 ~]$ anchore-cli --u admin --p foobar registry list
Registry                                    Name                                        Type        User
305929695733.dkr.ecr.eu-west-3.amazonaws.com    305929695733.dkr.ecr.eu-west-3.amazonaws.com    awsecr      awsauto
[ec2-user@ip-10-0-0-229 ~]$
```

Rerun the pipeline with the Replay button. This time, Anchore will examine the contents of the image filesystem for vulnerabilities. If high-severity vulnerabilities are found, this will fail the image build, as shown in figure 9.35.

```
[Pipeline] anchore
2020-05-15T17:08:17.605 INFO   AnchoreWorker   Jenkins version: 2.204.1
2020-05-15T17:08:17.605 INFO   AnchoreWorker   Anchore Container Image Scanner Plugin version: 1.0.22
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [global] debug: false
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] engineurl: http://10.0.0.229:8228/v1
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] engineuser: admin
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] enginepass: ****
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] engineverify: false
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] name: images
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] engineRetries: 300
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] policyBundleId:
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] bailOnFail: true
2020-05-15T17:08:17.605 INFO   AnchoreWorker   [build] bailOnPluginFail: true
2020-05-15T17:08:17.614 INFO   AnchoreWorker   Submitting 305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-
loader:02c7fc2863f49d176a1738c722b2b601eb9d122f for analysis
2020-05-15T17:08:17.923 INFO   AnchoreWorker   Analysis request accepted, received image digest
sha256:c0ef9fd3ce1fa82adee2796cf53d2e467ff9e0a0739515357e34a8c05254fc3d
2020-05-15T17:08:17.924 INFO   AnchoreWorker   Waiting for analysis of 305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-
loader:02c7fc2863f49d176a1738c722b2b601eb9d122f, polling status periodically
```

Figure 9.35 Image scanning with Anchore

Once the scanning is finished, Anchore will return with a nonzero exit code if the image has any known high-severity issues. The result of the Anchore policy evaluation will be saved in JSON files. Also, the pipeline will show the status of the build (STOP, WARN, or FAIL), as shown in figure 9.36.

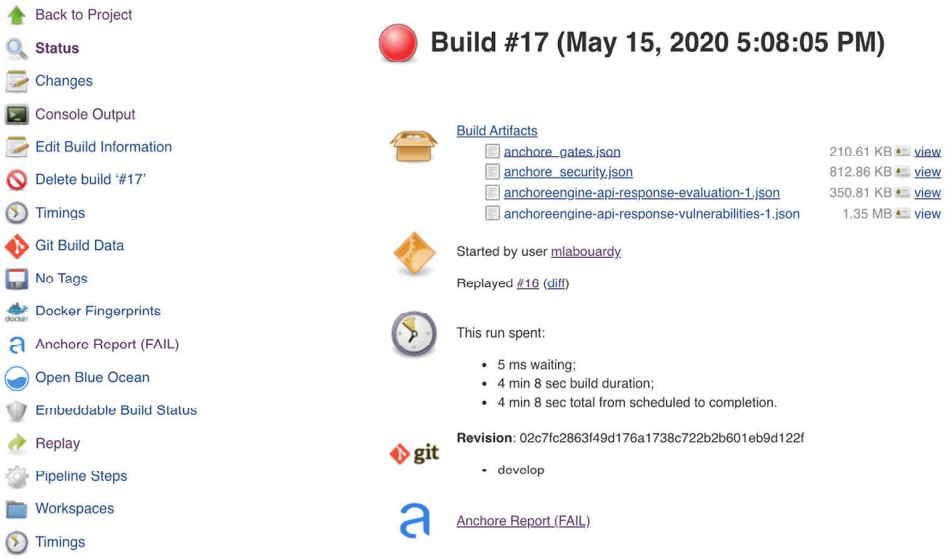

Figure 9.36 Anchore report results

The HTML report is automatically published, as well, on the newly created page. Clicking the Anchore Report link will display a graphical policy report showing the summary information and a detailed list of policy checks and results; see figure 9.37.

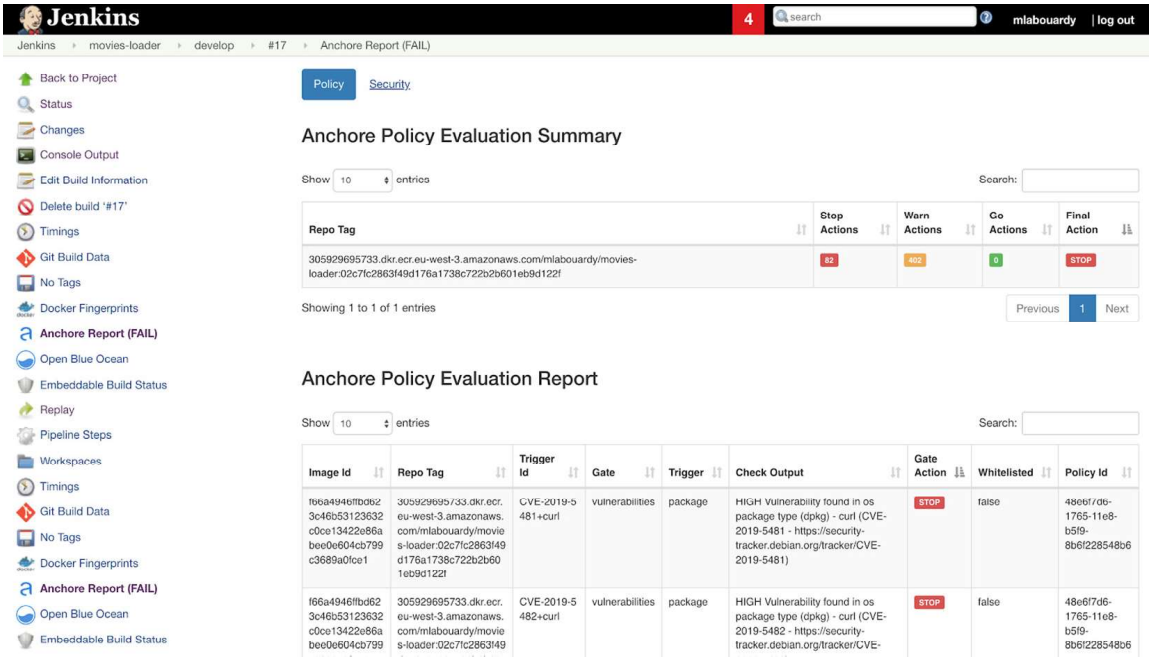

Figure 9.37 Anchore Common Vulnerabilities and Exposures (CVE) report