NOTE You can customize Anchore Engine to use your own security policies to allow/block external packages, OS scanning, and so forth.

And that's how to define a continuous integration pipeline on Jenkins from scratch for Dockerized microservices.

NOTE An alternative solution is Aqua Trivy (https://github.com/aquasecurity/trivy), which is a freely available community edition. Paid solutions also can be integrated easily with Jenkins such as Sysdig (https://sysdig.com/) and Aqua.

9.5 Writing a Jenkins declarative pipeline

Along with the previous chapters, we have used the scripted pipeline approach to define the CI pipeline for our project because of the flexibility it gives while using Groovy syntax. This section covers how to get the same pipeline output with a declarative pipeline approach. This is a simplified and friendlier syntax with specific statements for defining them, without a need to learn or master Groovy language.

Let's take as an example the scripted pipeline used for the movies-loader service. The following listing provides the service Jenkinsfile (cropped for brevity).

Listing 9.17 Jenkinsfile scripted pipeline

```
node('workers'){
    stage('Checkout'){
        checkout scm
    }
    stage('Unit Tests'){
        def imageTest= docker.build("${imageName}-test",
"-f Dockerfile.test .")
        imageTest.inside{
            sh "python main_test.py"
        }
    }
    stage('Build'){
        docker.build(imageName)
    }
    stage('Push'){
        docker.withRegistry(registry, 'registry') {
            docker.image(imageName).push(commitID())

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

This scripted pipeline can be easily converted to a declarative version, by following these steps:

1. Replace the `node('workers')` instruction with a `pipeline` keyword. All valid declarative pipelines must be enclosed within a `pipeline` block.

2. Define an `agent` section at the top level inside the `pipeline` block, to define the execution environment where the pipeline will be executed. In our example, the execution will be on Jenkins workers.
3. Wrap `stage` blocks with a `stages` section. The `stages` section contains a stage for each discrete part of the CI pipeline, such as `Checkout`, `Test`, `Build`, and `Push`.
4. Wrap each given `stage` command and instruction with a `steps` block.

Create a Jenkinsfile.declarative file with the required changes. The end result should look like the following listing.

Listing 9.18 Jenkinsfile declarative pipeline

```
pipeline{
    agent{
        label 'workers'
    }
    stages{
        stage('Checkout'){
            steps{
                checkout scm
            }
        }
        stage('Unit Tests'){
            steps{
                script {
                    def imageTest= docker.build("${imageName}-test",
"-f Dockerfile.test .")
                    imageTest.inside{
                        sh "python test_main.py"
                    }
                }
            }
        }
        stage('Build'){
            steps{
                script {
                    docker.build(imageName)
                }
            }
        }
        stage('Push'){
            steps{
                script {
                    docker.withRegistry(registry, 'registry') {
                        docker.image(imageName).push(commitID())

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

Defines where the pipeline should be executed. In the example, the pipeline stages will be performed on the agents with the workers label.

Clones the GitHub repository configured in the Jenkins's job settings

Builds a Docker image based on Dockerfile.test and provisions a container from the image to run the Python unit tests

Builds the application Docker image from the Dockerfile

Authenticates with the Docker remote repository and pushes the application image to the repository

NOTE The declarative pipeline might also contain a `post` section to perform post-build steps such as notification or cleaning up the environment. This section is covered in chapter 10.

Update the Jenkins job configuration to use the new declarative pipeline file instead by updating the Script Path field, as shown in figure 9.38.

Figure 9.38 Jenkinsfile path configuration

Push the declarative pipeline to the remote repository with these commands:

```
git add Jenkinsfile.declarative
git commit -m "pipeline with declarative approach"
git push origin develop
```

The GitHub webhook will notify Jenkins upon the push event, and the new declarative pipeline should be executed, as you can see in figure 9.39.

Figure 9.39 Jenkinsfile declarative pipeline execution

You can now restart any completed declarative pipeline from any top-level stage that ran in that pipeline. You can go to the side panel for a run in the classic UI and click Restart from Stage, as shown in figure 9.40.

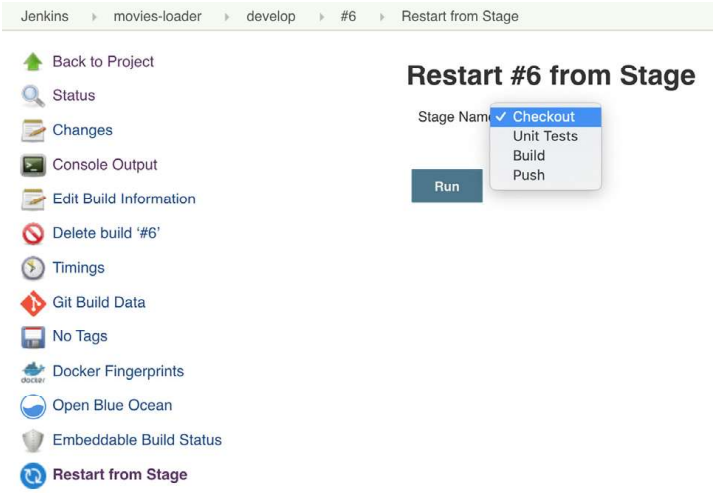

Figure 9.40 Restart from Stage feature

You will be prompted to choose from a list of top-level stages that were executed in the original run, in the order they were executed. This allows you to rerun a pipeline from a stage that failed because of transient or environmental considerations.

> **NOTE** Restarting stages can also be done in the Blue Ocean UI, after your pipeline has completed, whether it succeeds or fails.

Docker can also be used as an execution environment for running CI/CD pipelines in the `agent` section, as shown in the following listing.

Listing 9.19 Declarative pipeline with a Docker agent

```
pipeline{
    agent{
        docker {
            image 'python:3.7.3'
        }
    }
    stages{
       stage('Checkout'){
            steps{
                checkout scm
            }
        }
        stage('Unit Tests'){
            steps{
                script {
                    sh 'python test_main.py'
                }
            }
        }
    }
}
```

If we try to execute this pipeline, the build will quickly fail because the pipeline assumes that any configured machine/instance is capable of running Docker-based pipelines. In this example, the build ran in the master machine. However, because Docker is not installed in this machine, the pipeline failed:

```
+ docker inspect -f . python:3.7.3
/var/lib/jenkins/workspace/movies-loader_develop@tmp/durable-efd13a52/script.sh: line 1: docker: command not f
[Pipeline] isUnix
[Pipeline] sh
+ docker pull python:3.7.3
/var/lib/jenkins/workspace/movies-loader_develop@tmp/durable-7f4fd486/script.sh: line 1: docker: command not f
```

To run the pipeline on Jenkins workers only, update the Pipeline Model Definition settings from the Jenkins job configuration and set the `workers` label on the Docker Label field, as shown in figure 9.41.

When the pipeline executes, Jenkins will automatically start the specified container and execute the steps defined within it. This pipeline executes the same stages and the same steps.

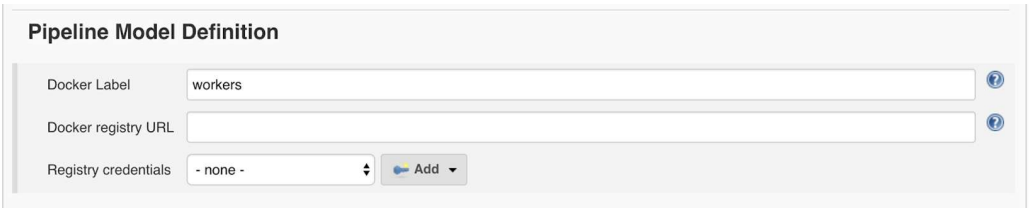

Figure 9.41 Pipeline model definition

9.6 Managing pull requests with Jenkins

For now, we push directly to the develop branch; however, we should create feature branches and then create pull requests to run tests and provide feedback to GitHub and block submission approval if tests fail. Let's see how to set up a review process with Jenkins for pull requests.

Create a new feature branch from the develop branch with the following command:

```
git checkout -b feature/featureA
```

Make some changes; in this example, I have updated the README.md file. Then, commit the changes and push the new feature branch to the remote repository:

```
git add README.md
git commit -m "update readme"
git push feature/featureA
```

Head over to the GitHub repository, and create a new pull request to merge the feature branch to the develop branch, as shown in figure 9.42.

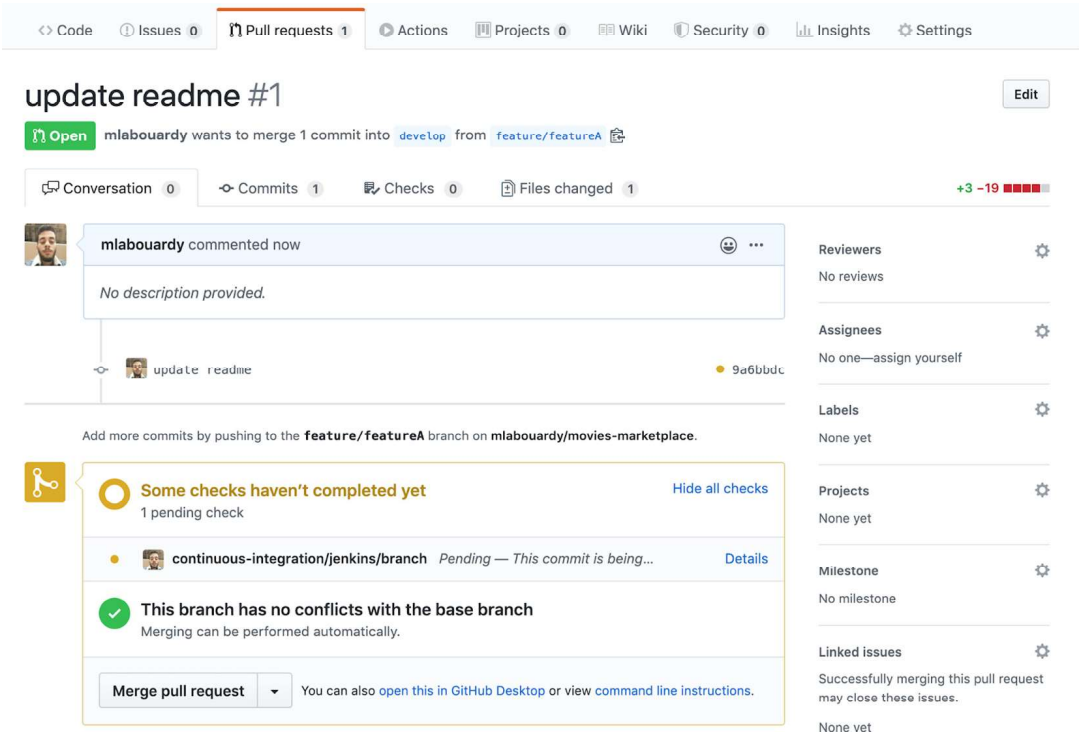

Figure 9.42 New pull request

On Jenkins, a new build will be triggered on the feature branch, as you can see in figure 9.43.

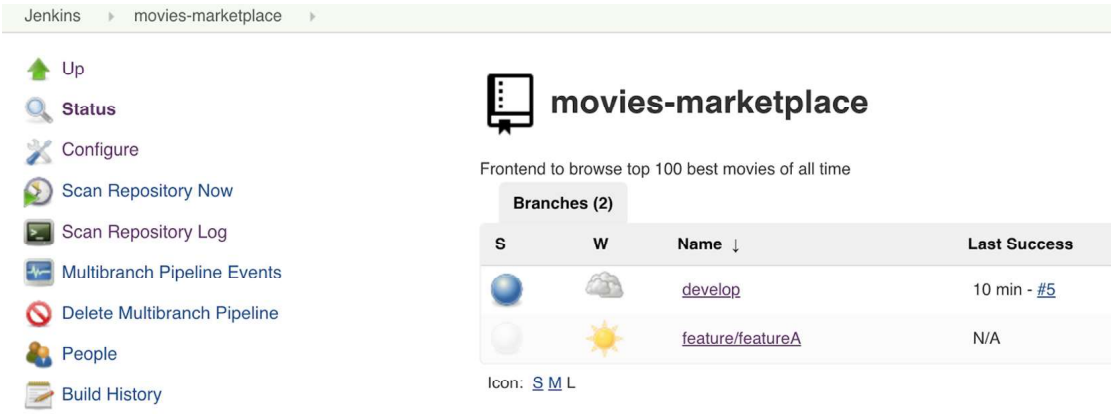

Figure 9.43 Build execution on the feature branch

Once the CI is finished, Jenkins will update the status on GitHub (figure 9.44). The build indicator in GitHub will turn either red or green, based on the build status.

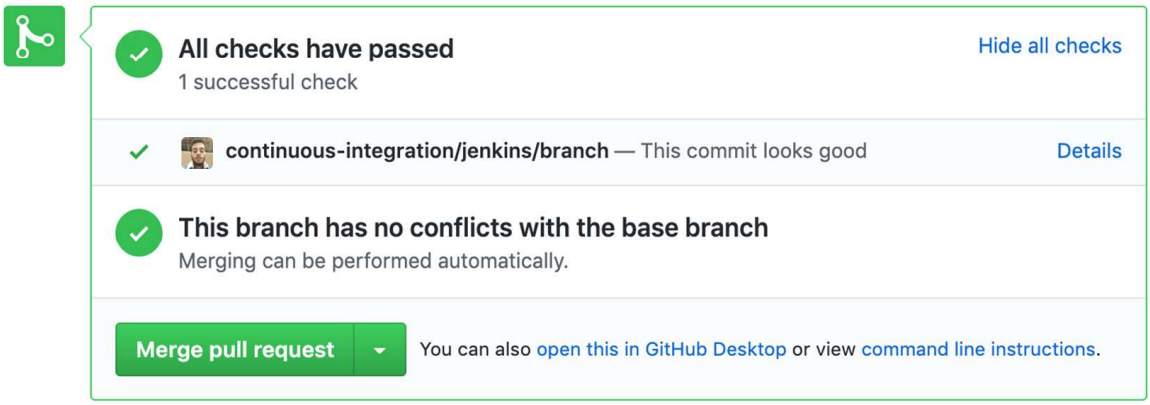

Figure 9.44 Jenkins post-build status on GitHub PR

> **NOTE** You can also configure SonarQube to analyze pull requests so you can ensure that the code is clean and approved for merging.

This process allows you to run a build and subsequent automated tests at every check-in so only the best code gets merged. Catching bugs early and automatically reduces the number of problems introduced into production, so your team can build better, more efficient software. We can now merge the feature branch and delete it; see figure 9.45.

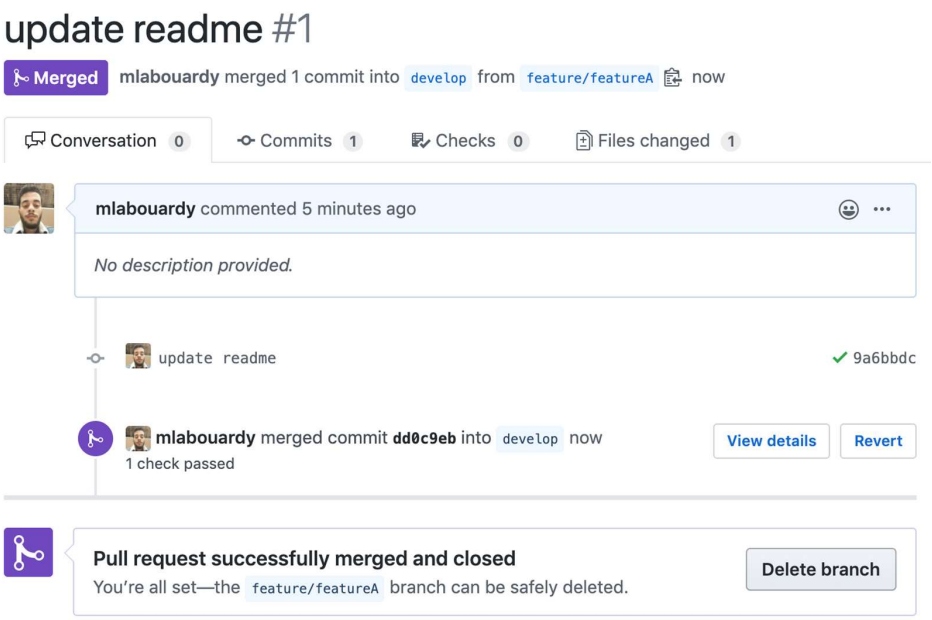

Figure 9.45 Merge and delete the feature branch.

And that will trigger another build on the develop branch, which will trigger the CI stages and push the image with the `develop` tag to the remote Docker registry.

Once the build is completed, we can check the status of previous commits by clicking the Commits section from the GitHub repository. A green, yellow, or red check mark should be displayed, depending on the state of the build; see figure 9.46.

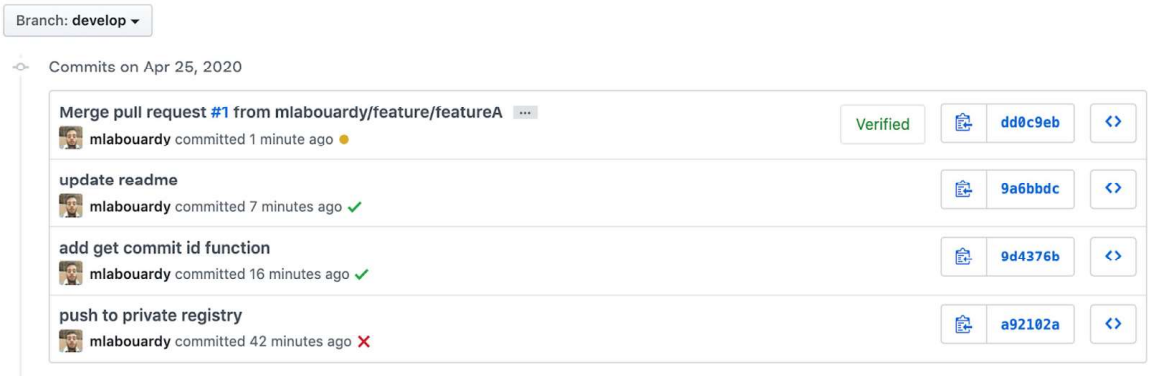

Figure 9.46 Jenkins build status history

Finally, to disable developers from pushing directly to the develop branch and also merging without a Jenkins build being passed, we will create a new rule to protect the develop branch. On the GitHub repository settings, jump to the Branches section and add a new protection rule that requires the Jenkins status check to be successful before merging. Figure 9.47 shows the rule configuration.

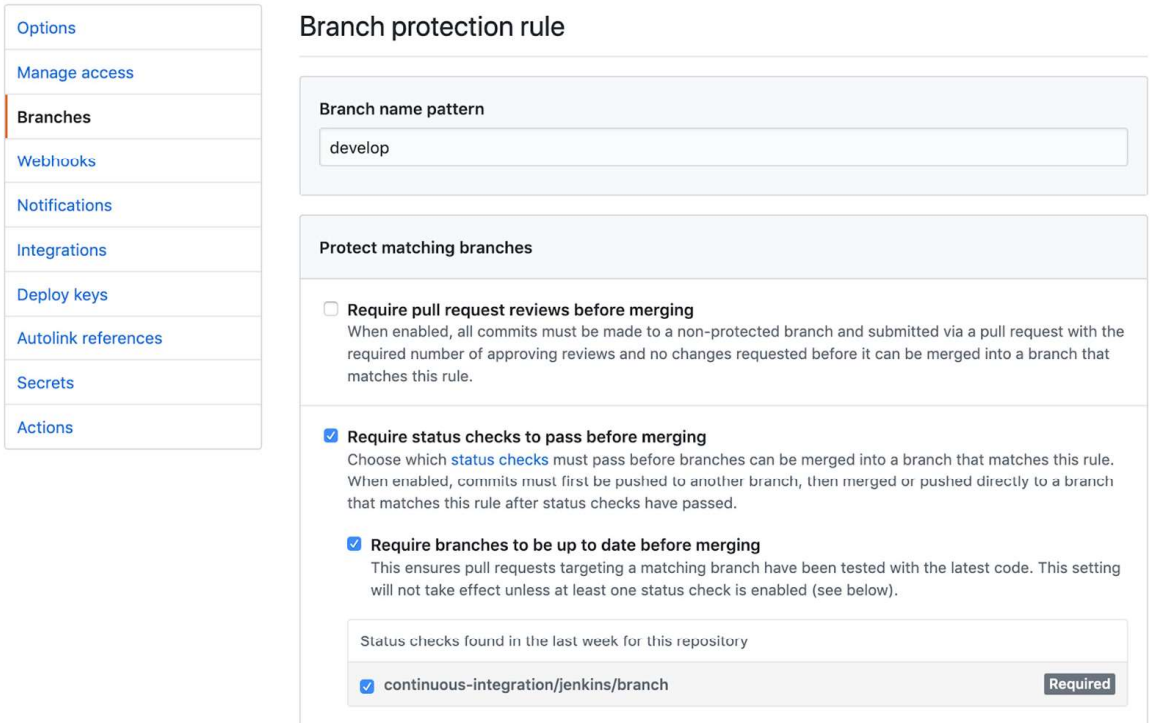

Figure 9.47 GitHub branch protection

Apply the same rule for the preprod and master branches. Then, repeat the same procedure for the rest of the GitHub repositories of the project.

With the Docker images safely stored in the private registry and the build status posted to GitHub, we've completed the implementation of the CI pipeline of Dockerized microservices with Jenkins multibranch pipelines. The next two chapters cover how to implement continuous deployment and delivery practices with Jenkins for two of the most used container orchestration platforms for cloud-native applications: Docker Swarm and Kubernetes.

Summary

- You can optimize Docker images for production with Docker caching layers, multistage build features, and lightweight base images such as an Alpine base image.
- The commit ID and Jenkins build ID can be used to tag Docker images for versioning and rollback to a working version in case of application deployment failure.
- Binary repository tools like Nexus and Artifactory can manage and store build artifacts for later use.
- Anchore Engine is an open source tool that lets you scan Docker images for security vulnerabilities during CI workflow.
- In a CI environment, the frequency of a build is too high, and each build generates a package. Since all the built packages are in one place, developers are at liberty to choose what to promote and what not to promote in higher environments.

10 Cloud-native applications on Docker Swarm

This chapter covers

- Deploying a self-healing Swarm cluster on AWS and using an S3 bucket for node discovery
- Running SSH-based commands within Jenkins pipelines and configuring SSH agents
- Automating deployment of Dockerized applications to Swarm
- Integrating Slack to manage releases and build notifications of CI/CD pipelines
- Continuous delivery to production and user manual approvals within Jenkins

The previous chapter covered how to set up a continuous integration pipeline for a containerized microservice application with Jenkins. This chapter covers how to automate the deployment and manage multiple application environments. By the end of this chapter, you will be familiar with continuous deployment and delivery (figure 10.1) for containerized microservices running in a Docker Swarm cluster.

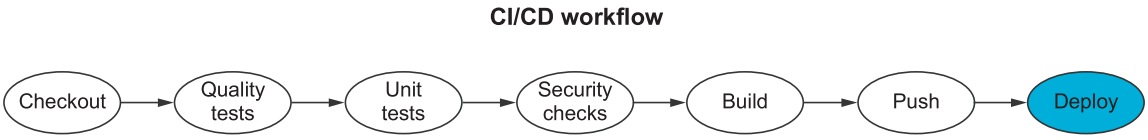

Figure 10.1 A complete CI/CD pipeline workflow

One of the basic solutions to run multiple containers across a set of machines is Swarm (https://docs.docker.com/engine/swarm/), which comes bundled with Docker Engine. By the end of this chapter, you should be able to build a CI/CD pipeline from scratch for services running inside a Docker Swarm cluster, as shown in figure 10.2.

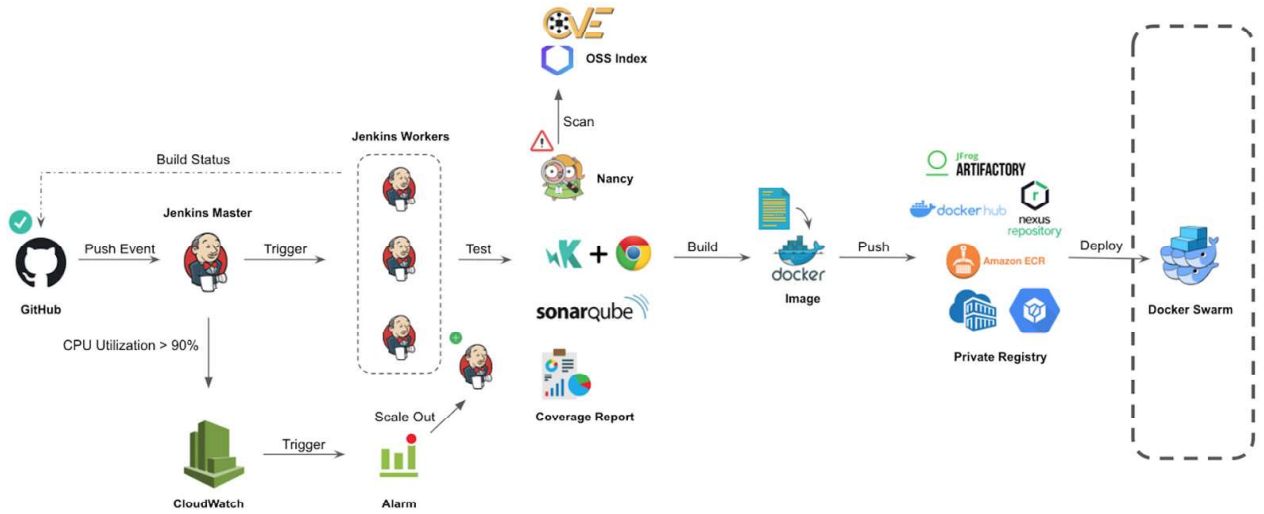

Figure 10.2 Target CI/CD pipeline

10.1 Running a distributed Docker Swarm cluster

Docker Swarm was originally released as a standalone product that ran master and agent containers on a cluster of servers to orchestrate the deployment of containers. This changed with the release of Docker 1.12 in 2016. Docker Swarm became officially part of Docker Engine and was built right into every Docker installation.

> **NOTE** This is just a brief overview of the capabilities of Docker Swarm in Docker. For further reading, feel free to explore the Docker Swarm official documentation (https://docs.docker.com/engine/swarm/).

To illustrate the deployment of containers into a Swarm cluster from a CI/CD pipeline defined in Jenkins, we need to deploy a Swarm cluster.

The Swarm cluster will be deployed inside a VPC with two Auto Scaling groups: one for Swarm managers and another for Swarm workers. Both ASGs will be deployed within private subnets that spin up across multiple availability zones for resiliency.

Once the ASGs are created, setting up the Swarm requires manual initialization of the managers, and adding new nodes to the cluster requires additional information (a cluster join token) provided by the first manager when the Swarm is created.

This step can be automated with configuration management tools like Ansible or Chef. However, it requires manual interaction. To address this, and to provide automatic Swarm initialization, we will run a one-shot Docker container on instance launch; the container uses an S3 bucket as a cluster discovery registry to find active managers and join tokens.

Figure 10.3 summarizes the architecture we will deploy. We will focus on AWS, but the same architecture can be applied in other cloud providers or locally.

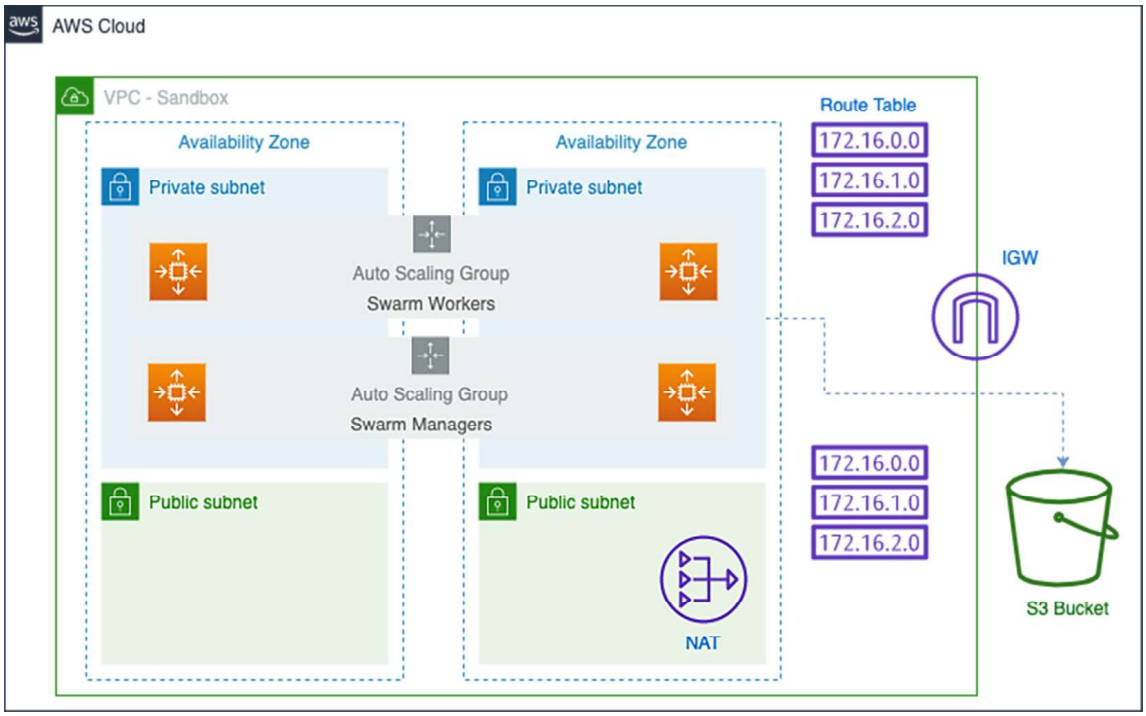

Figure 10.3 Swarm architecture in AWS

> **NOTE** A distributed, consistent key-value store such as etcd (https://etcd.io/), HashiCorp's Consul (www.consul.io), or Apache ZooKeeper (https://zookeeper.apache.org/) can be used as service discovery to make the nodes auto-join the Swarm cluster.

To deploy Swarm instances, we need to provide an AMI with Docker Engine preinstalled. By now, you should be familiar with Packer. We will create a template.json file with the content in the following listing. (The full template can be downloaded from chapter10/swarm/packer/docker-ce/template.json.)

Listing 10.1 Docker AMI's Packer template

```
{
    "variables" : {},
    "builders" : [
        {
            "type" : "amazon-ebs",
            "profile" : "{{user `aws_profile`}}",
            "region" : "{{user `region`}}",
            "instance_type" : "{{user `instance_type`}}",
            "source_ami" : "{{user `source_ami`}}",
            "ssh_username" : "ec2-user",
            "ami_name" : "18.09.9-ce",
            "ami_description" : "Docker engine AMI",
        }
    ],
    "provisioners" : [
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

The base image is Amazon Linux 2, which will be provisioned with a shell script that installs the most recent Docker Community Edition package. Then it adds the `ec2-user` username to the `docker` group, to be able to execute Docker commands without using the `sudo` command; see the following listing.

Listing 10.2 Docker Community Edition installation

```
#!/bin/bash
yum update -y
yum install docker -y
usermod -aG docker ec2-user
systemctl enable docker
```

Issue a `packer build` command to bake the Docker AMI. Once the provisioning process is completed, the new baked AMI should be available on the Images section on the AWS Management Console (figure 10.4).

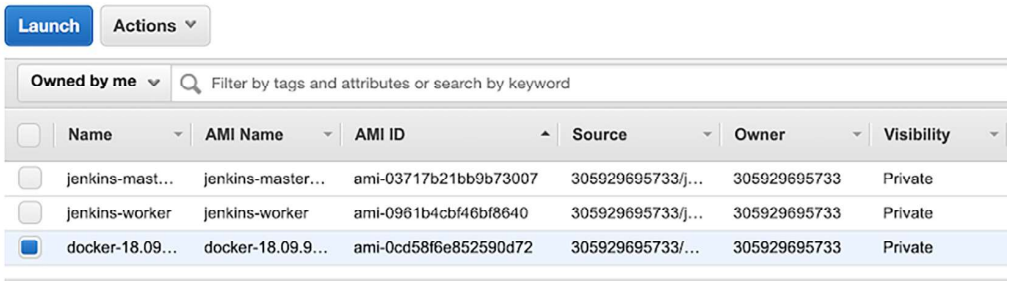

Figure 10.4 Docker Community Edition AMI

Next, deploy the infrastructure with Terraform, and create a dedicated VPC called `sandbox` with a 10.1.0.0/16 CIDR block to isolate the sandbox application and workload. Define the block in listing 10.3 in the vpc.tf file.

> **NOTE** Deploying the cluster on a different VPC is not mandatory, but following the best practices by isolating your workload environments for auditing and security compliance is strongly recommended.

Listing 10.3 Sandbox VPC resource

```
resource "aws_vpc" "sandbox" {
  cidr_block           = var.cidr_block
  enable_dns_hostnames = true

  tags = {
    Name   = var.vpc_name
    Author = var.author
  }
}
```

The Swarm manager needs a way of passing the worker token to the workers after it has initialized. The best way to do that is to have the Swarm manager's user data trigger generating the token and putting it into an S3 bucket. Define a private S3 bucket resource in s3.tf with the code in the following listing.

Listing 10.4 Swarm discovery S3 bucket resource

```
resource "aws_s3_bucket" "swarm_discovery_bucket" {
  bucket = var.swarm_discovery_bucket
  acl    = "private"

  tags = {
    Author = var.author
    Environment = var.environment
  }
}
```

> **NOTE** The AWS Systems Manager Parameter Store (http://mng.bz/r6GX) can also be used as a shared encrypted store to store and retrieve the join token for Swarm workers.

An IAM instance profile is necessary for EC2 instances to be able to interact with the S3 bucket to store or fetch the Swarm token for an autojoin operation. Define an IAM role policy within the iam.tf file, as shown in the next listing.

Listing 10.5 Swarm nodes IAM policy

```
resource "aws_iam_role_policy" "discovery_bucket_access_policy" {
  name = "discovery-bucket-access-policy-${var.environment}"
  role = aws_iam_role.swarm_role.id
```

```
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "s3:*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
EOF
}
```

Then, we create a launch configuration for Swarm managers that uses the Docker AMI baked with Packer and run a startup script configured on user data. Use the following listing to define the code in swarm_managers.tf.

Listing 10.6 Swarm managers launch configuration

```
resource "aws_launch_configuration" "managers_launch_conf" {
  name                 = "managers_config_${var.environment}"
  image_id             = data.aws_ami.docker.id
  instance_type        = var.manager_instance_type
  key_name             = var.key_name
  security_groups      = [aws_security_group.swarm_sg.id]
  user_data            = data.template_file.swarm_manager_user_data.rendered
  iam_instance_profile = aws_iam_instance_profile.swarm_profile.id

  root_block_device {
    volume_type = "gp2"
    volume_size = 20
  }

  lifecycle {
    create_before_destroy = true
  }
}
```

The startup script uses the name of the cluster discovery S3 bucket and the role of the running instance (manager or worker), as shown in the next listing. Based on the instance role, the `docker swarm join` command will use the right token (`workers` token or `managers` token).

Listing 10.7 Swarm managers user data

```
data "template_file" "swarm_manager_user_data" {
  template = "${file("scripts/join-swarm.tpl")}"
  vars = {
    swarm_discovery_bucket = "${var.swarm_discovery_bucket}"
    swarm_name             = var.environment
```

```
    swarm_role              = "manager"
  }
}
```

The shell script joint-swarm.tpl, shown in the following listing, uses EC2 metadata to fetch the instance private IP address. The script then executes a container that uses the S3 bucket to store the state of the Swarm once it's created or creates a new Swarm if no state already exists in the bucket.

Listing 10.8 Swarm nodes startup script

```
#!/bin/bash
NODE_IP=$(curl -fsS http://169.254.169.254/latest/meta-data/local-ipv4)
docker run -d --restart on-failure:5 \
    -e SWARM_DISCOVERY_BUCKET=${swarm_discovery_bucket} \
    -e ROLE=${swarm_role} \
    -e NODE_IP=$NODE_IP \
    -e SWARM_NAME=${swarm_name} \
    -v /var/run/docker.sock:/var/run/docker.sock \
    mlabouardy/swarm-discovery
```

NOTE The mlabouardy/swarm-discovery full Python script and Dockerfile is given in the GitHub repository: pipeline-as-code-with-jenkins/tree/master/chapter10/discovery.

From there, we will create an ASG of managers. By default, we will create one manager for the cluster. But I recommend using an odd number when running Swarm in production, as a majority vote is needed among managers to agree on proposed management tasks. An odd—rather than even—number is strongly recommended to have a tie-breaking consensus. However, for a sandbox cluster, we will keep it simple and go with one Swarm manager. In swarm_mangers.tf, define the ASG resource as shown in the following listing.

Listing 10.9 Swarm managers Auto Scaling group

```
resource "aws_autoscaling_group" "swarm_managers" {
  name                 = "managers_asg_${var.environment}"
  launch_configuration = aws_launch_configuration.managers_launch_conf.name
  vpc_zone_identifier  = [for subnet in aws_subnet.private_subnets:
     subnet.id]
  depends_on = [aws_s3_bucket.swarm_discovery_bucket]
  min_size             = 1
  max_size             = 3
  lifecycle {
    create_before_destroy = true
  }
}
```

NOTE You can define autoscaling policies with CloudWatch alarms to trigger scale-out or scale-in events based on CPU utilization or custom metrics of the Swarm nodes.

Similarly, we will create an ASG for workers, and we will go with two Swarm workers. Note the use of the `depends_on` keyword to create an implicit dependency on the `swarm_managers` resource. Terraform uses this information to determine the correct order for creating resources.

In this example, Terraform will create Swarm managers first. That way, we guarantee the Swarm initialization and the availability of a join token in the S3 bucket. Add the resource in the following listing in the swarm_workers.tf file.

Listing 10.10 Swarm workers ASG

```
resource "aws_autoscaling_group" "swarm_workers" {
  name                 = "workers_asg_${var.environment}"
  launch_configuration = aws_launch_configuration.workers_launch_conf.name
  vpc_zone_identifier  = [for subnet in aws_subnet.private_subnets:
     subnet.id]
  min_size             = 2
  max_size             = 5
  depends_on = [aws_autoscaling_group.swarm_managers]
  lifecycle {
    create_before_destroy = true
  }
}
```

Finally, allow the firewall rules in table 10.1 on the security group assigned to the Swarm cluster instances.

Table 10.1 Swarm cluster security group rules

Protocol	Port	Source	Description
TCP	2377	Swarm	Cluster management and raft sync communications
TCP	7946	Swarm	Control-plane gossip discovery communication among all nodes
UDP	7946	Swarm	Container network discovery from other Swarm nodes
UDP	4789	Swarm	Data-plane VXLAN overlay network traffic
TCP	22	Jenkins and Bastion SGs	SSH traffic from Jenkins master and bastion security groups

The following listing provides the security group definition.

Listing 10.11 Swarm nodes security group

```
resource "aws_security_group" "swarm_sg" {
  name        = "swarm_sg_${var.environment}"
  description = "Allow inbound traffic for
swarm management and ssh from jenkins & bastion hosts"
  vpc_id      = aws_vpc.sandbox.id
```

```
  ingress {
    from_port       = 22
    to_port         = 22
    protocol        = "tcp"
    security_groups = [var.bastion_sg_id, var.jenkins_sg_id]
  }
  ingress {
    from_port   = "2377"
    to_port     = "2377"
    protocol    = "tcp"
    cidr_blocks = [var.cidr_block]
  }
  …
  egress {
    from_port   = "0"
    to_port     = "0"
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

NOTE I recommend using an S3 backend with encryption and versioning enabled to remotely store the Terraform state files.

Define the required Terraform variables in variables.tfvars as listed in table 10.2.

Table 10.2 Swarm Terraform variables

Variable	Type	Value	Description
region	String	None	The name of the region, such as eu-central-1, in which to deploy the Swarm cluster
shared_credentials_file	String	~/.aws/credentials	The path to the shared credentials file. If this is not set and a profile is specified, ~/.aws/credentials will be used.
aws_profile	String	profile	The AWS profile name as set in the shared credentials file
author	String	None	Name of the owner of the Swarm cluster. It's optional, but recommended, to tag your AWS resources to track the monthly costs by owner or environment.
key_name	String	None	SSH key pair
availability_zones	List	None	Availability zone where you'll spin up the VPC subnet
bastion_sg_id	String	None	The bastion host security group ID
jenkins_sg_id	String	None	The Jenkins master security group ID
vpc_name	String	sandbox	The name of the VPC

Table 10.2 Swarm Terraform variables *(continued)*

Variable	Type	Value	Description
`environment`	String	`sandbox`	The runtime environment name
`cidr_block`	String	`10.1.0.0/16`	The VPC CIDR block
`cluster_name`	String	`sandbox`	The Swarm cluster's name
`public_subnets_count`	Number	2	The number of public subnets to create
`private_subnets_count`	Number	2	The number of private subnets to create
`swarm_discovery_bucket`	String	`swarm-discovery-cluster`	The S3 bucket where the Swarm tokens will be stored
`manager_instance_type`	String	`t2.small`	The EC2 instance type for Swarm managers
`worker_instance_type`	String	`t2.large`	The EC2 instance type for Swarm workers

Then, use the `terraform apply` command to start the deployment process. Once deployed, the ASGs will be created, the Swarm discovery container will be launched on each instance, and the first manager to be run will execute the `swarm init` command and store the token on the S3 bucket (figure 10.5), which will be used by other instances to join the cluster.

> **NOTE** You can have as many or as few worker groups as you wish, running in as many different configurations as you choose (CPU or memory-optimized workers alongside general-purpose Swarm workers).

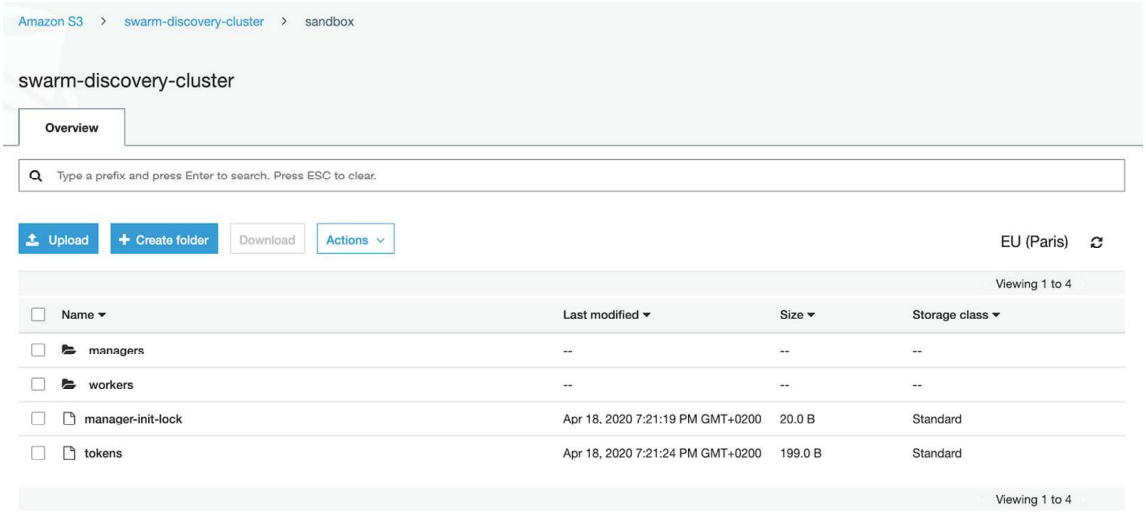

Figure 10.5 Swarm state stored in an S3 bucket

If you decide to create a dedicated VPC for the Swarm cluster, you need to set up VPC peering between management and sandbox VPCs, as shown in figure 10.6. For a step-by-step guide on how to set up peering with Terraform, refer to the official Terraform documentation at http://mng.bz/VBw5.

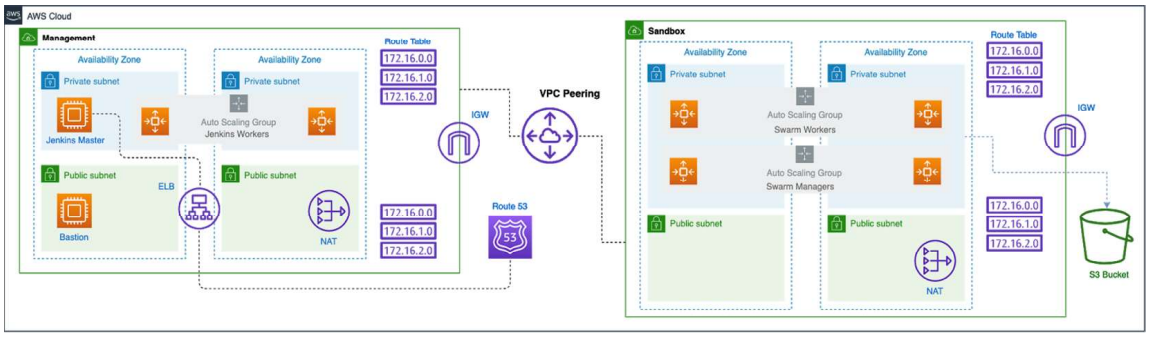

Figure 10.6 VPC peering between management and sandbox VPCs

> **NOTE** If you intend to use the VPC peering connection, make sure the VPCs don't have matching or overlapping IPv4 CIDR blocks. In our example, the management and sandbox CIDR blocks are 10.0.0.0/16 and 10.1.0.0/16, respectively.

From the VPC dashboard, navigate to Peering Connections and create a new one. Configure the peering as shown in figure 10.7.

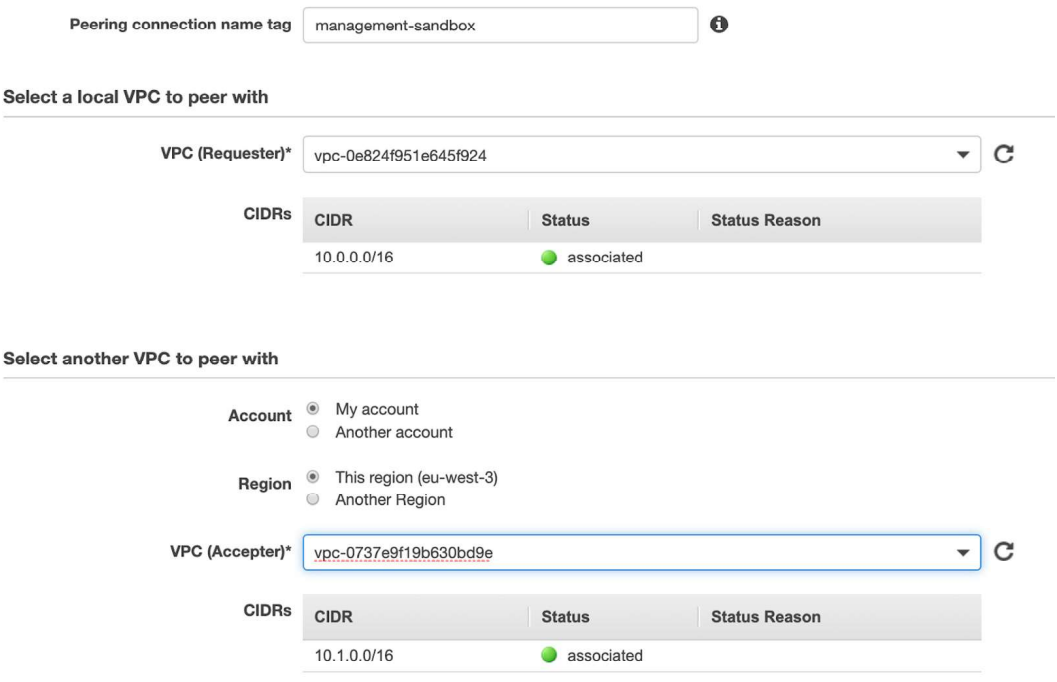

Figure 10.7 Configuring the peering of management and sandbox VPCs

After creating the peering connection, you'll see Pending Acceptance in the status bar. If you are using a different account or different region, go to the corresponding VPC console, where you can see Pending Acceptance in the status bar of the peering connection. From the Actions drop-down, choose Accept Request, as shown in figure 10.8. Then, in the Accept VPC Peering Connection Request prompt box, click Yes, Accept.

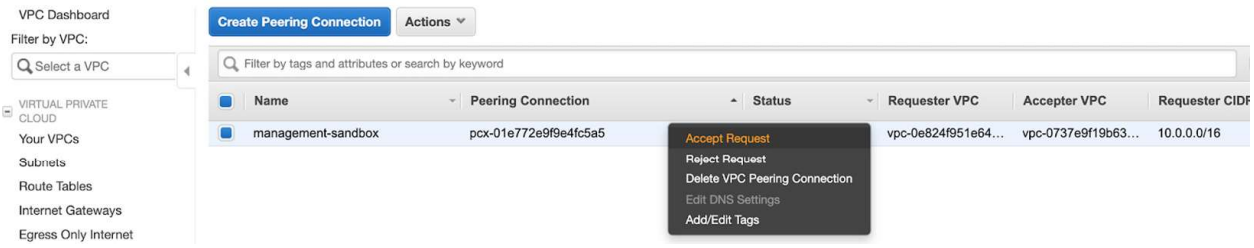

Figure 10.8 Accepting VPC peering request

To send and receive traffic across this VPC peering connection, you must add a route to the peered VPC in one or more of your VPC route tables. In the route tables associated with the subnets of the VPC, create a route with the CIDR block of the peer VPC as a destination, and the ID of the VPC peering connection as a target.

Repeat the same setups for all other VPC route tables. Once everything is set up, your routing table will look like figure 10.9.

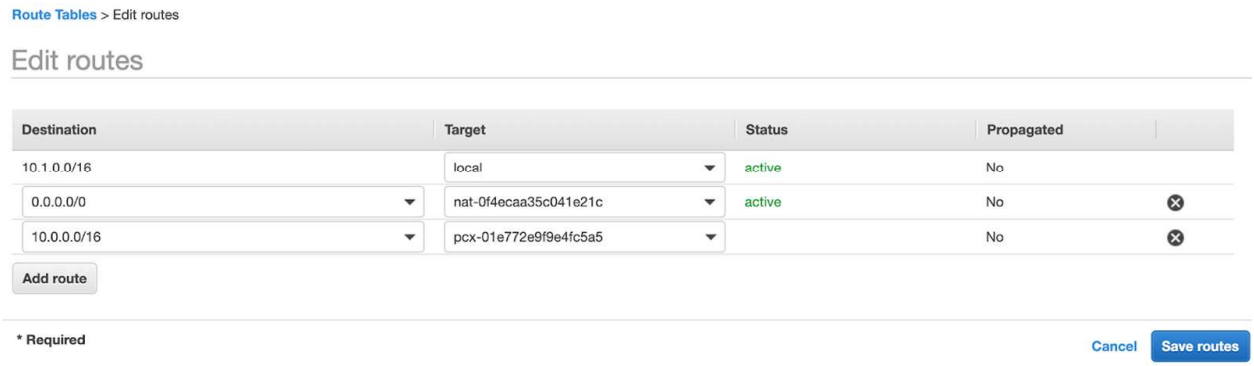

Figure 10.9 Sandbox VPC's route table update

To view the Swarm state, set up an SSH tunnel by using the bastion host deployed in chapter 5's section 5.2.4:

```
ssh -N 3000:SWARM_MANAGER_IP:22 ec2-user@BASTION_IP
ssh ec2-user@localhost -p 3000
```

Replace `SWARM_MANAGER_IP` with the Swarm manager private IP address. Once connected, if you type the `docker info` command, the `Swarm: active` attribute should confirm that Swarm has been properly configured:

```
[ec2-user@ip-10-1-2-168 ~]$ docker info
Containers: 1
 Running: 0
 Paused: 0
 Stopped: 1
Images: 1
Server Version: 18.09.9-ce
Storage Driver: overlay2
 Backing Filesystem: extfs
 Supports d_type: true
 Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
 Volume: local
 Network: bridge host macvlan null overlay
 Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
Swarm: active
 NodeID: zv2gyvahz61mthzr3z29e16v1
 Is Manager: true
```

Run `docker node ls` from the manager machine to view your Swarm's connected nodes. As you can see in figure 10.10, we now have one manager and two workers.

```
docker node ls
```

```
[ec2-user@ip-10-1-2-168 ~]$ docker node ls
ID                            HOSTNAME            STATUS              AVAILABILITY        MANAGER STATUS      ENGINE VERSION
uoxlczrw3f17bsuegqdcst4o1     ip-10-1-0-91        Ready               Active                                  18.09.9-ce
wyo0uo4sjigu0fa030aim809m     ip-10-1-2-155       Ready               Active                                  18.09.9-ce
zv2gyvahz61mthzr3z29e16v1 *   ip-10-1-2-168       Ready               Active              Leader              18.09.9-ce
[ec2-user@ip-10-1-2-168 ~]$
```

Figure 10.10 Swarm cluster nodes list

With our Swarm up and running, let's deploy the Dockerized-based application with Jenkins.

10.2 Defining a continuous deployment process

Create a new GitHub repository for deployment. Because deployment options are often changed, we will store the deployment part on a different Git repo. Then, create three main branches: develop, preprod, and master, as in figure 10.11.

Docker Swarm mode now integrates directly with Docker Compose v3 and officially supports the deployment of *stacks* (groups of services) via docker-compose.yml files. The same docker-compose.yml file you would use to test your application locally can now be used to deploy your application to Swarm.

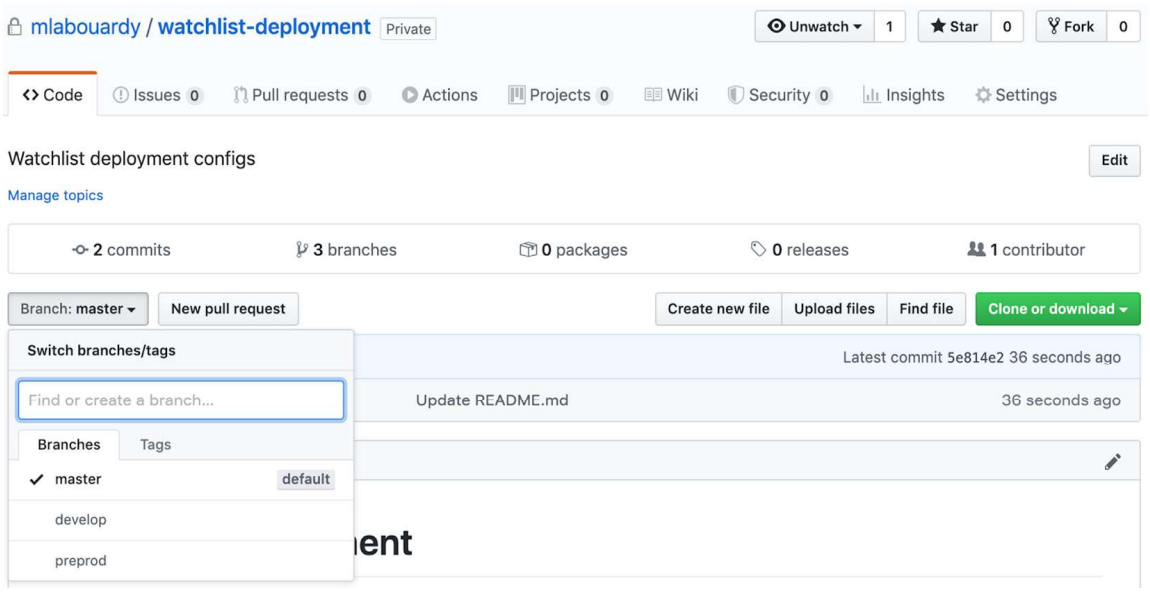

Figure 10.11 GitHub deployment repository

To do a Docker Swarm deployment from Jenkins, we need a docker-compose file that contains the references to Docker images along with the configuration settings such as port, network name, labels, and constraints. To run this file, we need to execute the `docker stack deployment` command over SSH on a manager machine.

On the develop branch, create a docker-compose.yml file by using your favorite text editor or IDE, with the content in the following listing.

Listing 10.12 Application Docker Compose

```
version: "3.3"
services:
  movies-loader:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-loader:develop
    environment:
      - AWS_REGION=REGION
      - SQS_URL=https://sqs.REGION.amazonaws.com/ID/movies_to_parse_sandbox

  movies-parser:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-loader:develop
    environment:
      - AWS_REGION=REGION
      - SQS_URL=https://sqs.REGION.amazonaws.com/ID/movies_to_parse_sandbox
      - MONGO_URI=mongodb://root:root@mongodb/watchlist
      - MONGO_DATABASE=watchlist
    depends_on:
      - mongodb

  movies-store:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-store:develop
```

```
    environment:
      - MONGO_URI=mongodb://root:root@mongodb/watchlist
    ports:
      - 3000:3000
    depends_on:
      - mongodb

  movies-marketplace:
    image: ID.dkr.ecr.REGION.amazonaws.com/USER/movies-marketplace:develop
    ports:
      - 80:80

  mongodb:
    image: bitnami/mongodb:latest
    environment:
      - MONGODB_USERNAME=root
      - MONGODB_PASSWORD=root
      - MONGODB_DATABASE=watchlist
```

NOTE Substitute the `ID`, `REGION`, and `USER` with your own AWS Account ID, AWS region, and ECR URI.

Each service uses the image we built in chapter 9 and references the `develop` tag. This tag is dedicated to sandbox deployment and contains the codebase of the develop branch. Also, we have defined a MongoDB service that will be used by both the movies-store and movies-parser services.

The MongoDB service credentials are in plaintext. However, you shouldn't commit sensitive information under any circumstances and opt for managed solutions like HashiCorp Vault or AWS SSM Parameter Store to encrypt your credentials and access tokens. You can also use an integrated feature of Docker called Secrets to create database credentials:

```
openssl rand -base64 12 | docker secret create mongodb_password -
```

And update docker-compose.yml to use the secret instead of the plaintext password:

```
mongodb:
    image: bitnami/mongodb:latest
    environment:
      - MONGODB_USERNAME=root
      - MONGO_ROOT_PASSWORD_FILE: /run/secrets/mongodb_password
      - MONGODB_DATABASE=watchlist
```

NOTE If the MongoDB service crashes for unknown reasons or has been removed, its data will be lost. To avoid this loss of data, you should mount a persistent volume. Depending on the cloud provider used, Docker volumes support use of external persistent storage such as Amazon EBS.

To decouple the crawling and parsing of HTML pages, we are using a distributed queue between the movies-loader and movies-parser services. In addition to its high availability, this will allow us to deploy additional movies-parser workers based on the

number of HTML pages to parse. Create an SQS for the sandbox environment called `movies_to_parse_sandbox` with Terraform (chapter10/swarm/terraform/sqs.tf), as shown in figure 10.12. This queue will be used by movies-loader to push movies into, and then it will be consumed by movies-parser workers.

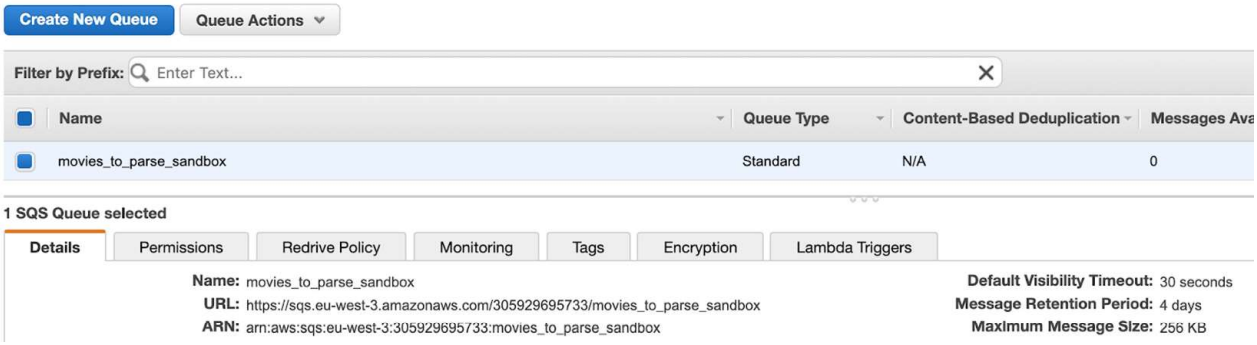

Figure 10.12 Sandbox queue settings

With Docker Compose out of the way, we can proceed and create a Jenkinsfile, shown in listing 10.13, with these steps:

1. Clone the GitHub repository (chapter10/deployment/sandbox/Jenkinsfile) and check out the develop branch.
2. Send the docker-compose.yml file over SSH to the manager node and execute the command `docker stack deploy`.

NOTE We use the master label to constrain the pipeline to be executed on the Jenkins master only. Workers' machines might also be used for this job.

Listing 10.13 Deployment Jenkinsfile

```
def swarmManager = 'manager.sandbox.domain.com'
def region = 'AWS REGION'                                  <-- Replace with your own AWS default region.
node('master'){
    stage('Checkout'){
        checkout scm
    }
    stage('Copy'){
        sh "scp -o StrictHostKeyChecking=no
docker-compose.yml ec2-user@${swarmManager}:/home/ec2-user"
    }
    stage('Deploy stack'){
        sh "ssh -oStrictHostKeyChecking=no ec2-user@${swarmManager}
     '\$(\$(aws ecr get-login --no-include-email
     --region ${region}))' || true"
```

```
        sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager} docker stack deploy
--compose-file docker-compose.yml
--with-registry-auth watchlist"
    }
}
```

This Jenkinsfile uses Amazon ECR as a private registry. If you're using a private registry that requires username and password authentication (such as Nexus, DockerHub, Azure, or Cloud Container Registry), you can use the Credentials Binding plugin https://plugins.jenkins.io/credentials-binding/), which is installed by default, to allow registry credentials to be bounded to `USERNAME` and `PASSWORD` variables. Then, pass those variables to the `docker login` command for authentication:

```
stage('Deploy'){
    withCredentials([[
$class: 'UsernamePasswordMultiBinding',
credentialsId: 'registry',
usernameVariable: 'USERNAME',
passwordVariable: 'PASSWORD']]) {
        sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager}
docker login --password $PASSWORD --username $USERNAME
${registry}"
        sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager}
docker stack deploy --compose-file docker-compose.yml
--with-registry-auth watchlist"
    }
}
```

Push the Jenkinsfile and docker-compose.yml files to the develop branch with the following commands:

```
git add .
git commit -m "deploy watchlist stack to sandbox"
git push origin develop
```

Head over to Jenkins, and create a new multibranch pipeline job called watchlist-deployment.

> **NOTE** For a step-by-step guide on how to create and configure multibranch pipeline jobs on Jenkins, check out chapter 7.

Set the GitHub repository HTTPS clone URL and allow Jenkins to discover all branches looking for a Jenkinsfile on the root repository, as shown in figure 10.13.

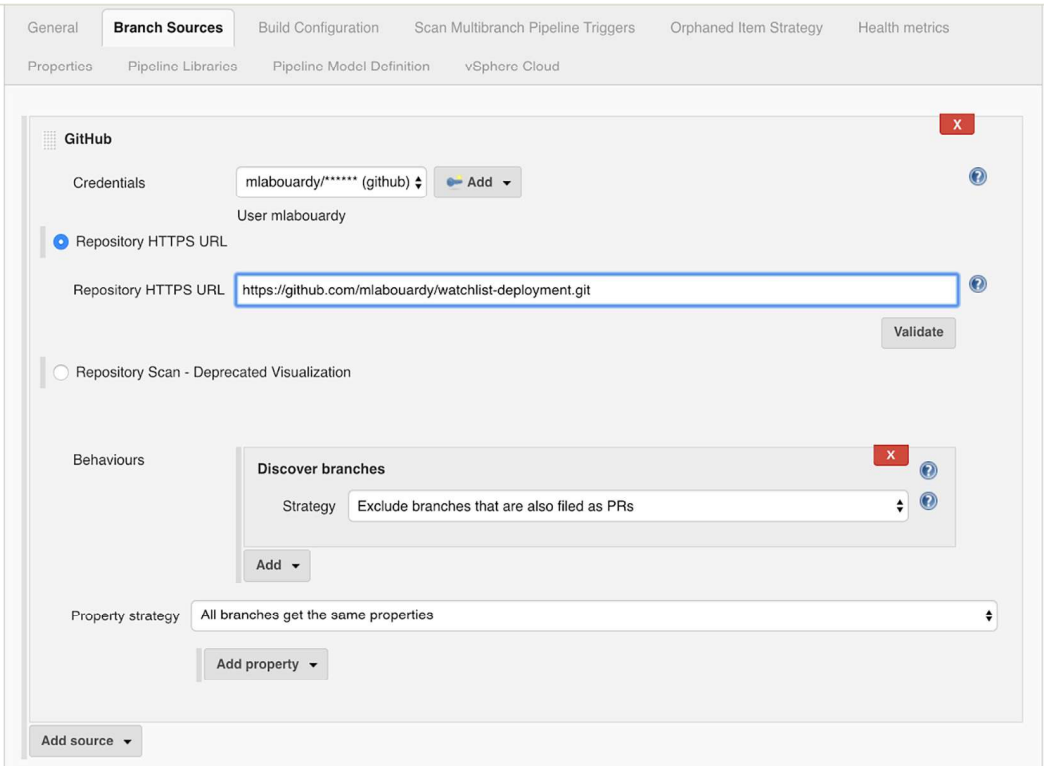

Figure 10.13 Branch sources configuration

For now, the job pipeline should discover the develop branch and execute the stages defined in the Jenkinsfile, as shown in figure 10.14.

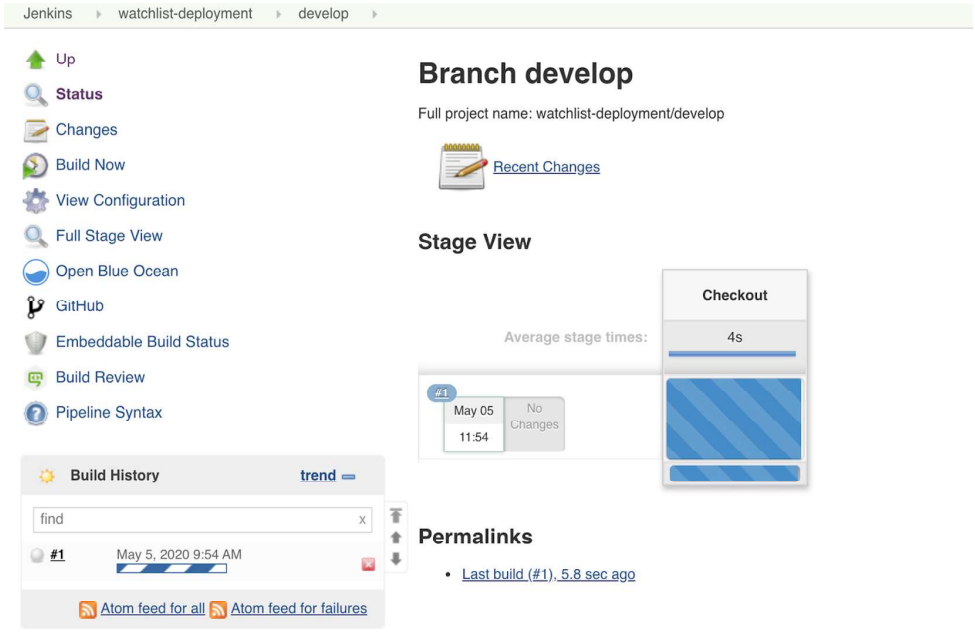

Figure 10.14 Deployment job on Jenkins

The pipeline should fail and turn red at the Copy stage, as shown in figure 10.15. The Jenkins master cannot SSH to the Swarm manager because the Jenkins master has the wrong private SSH key.

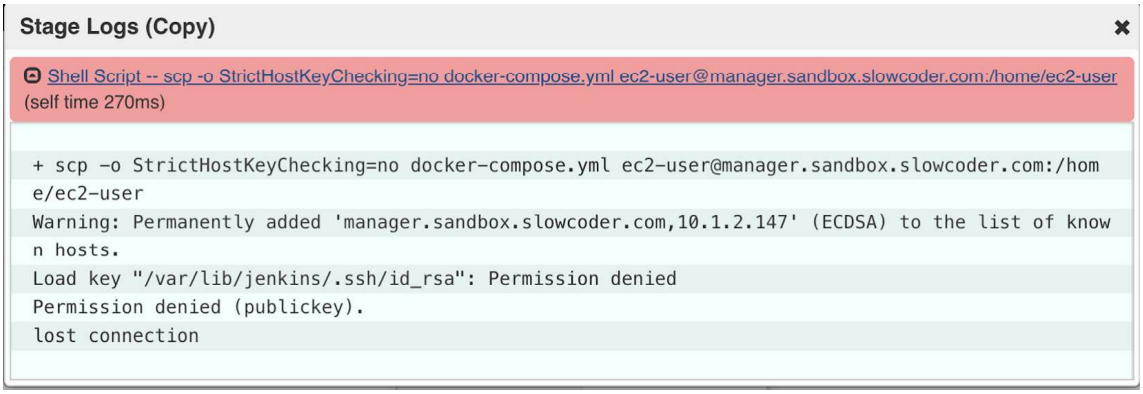

Figure 10.15 SCP command logs

For Jenkins to continuously deploy to the Swarm, it needs access to the Swarm manager. Create a new credential of type SSH Username with Private Key on Jenkins to access the Swarm sandbox. On a private-key field, paste the content of the key pair used while creating Swarm EC2 instances. Then, call it `swarm-sandbox`, as shown in figure 10.16.

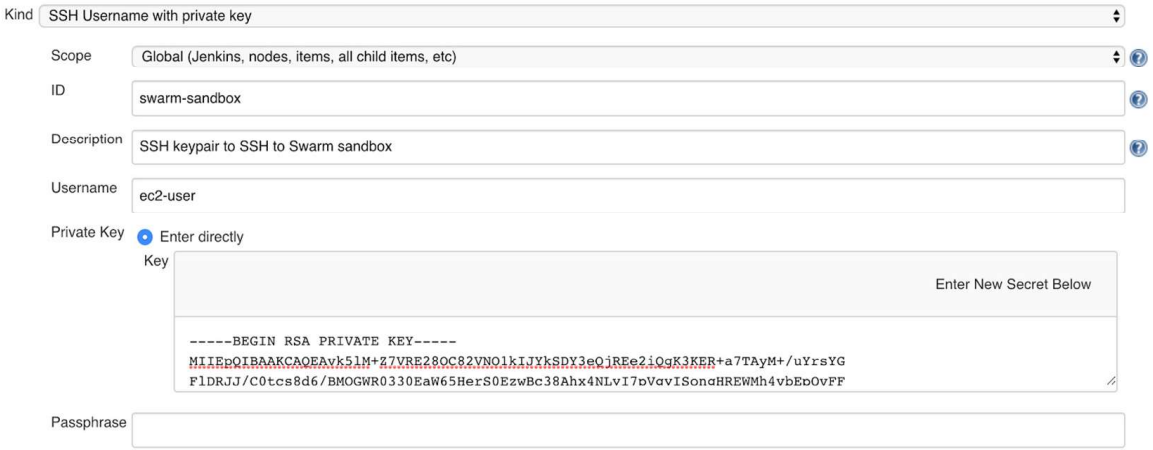

Figure 10.16 Jenkins credential with Swarm SSH key pair

NOTE Jenkins would need access to only the Swarm manager. The other nodes are managed by the Swarm manager, so Jenkins does not need direct access to them.

Update the Jenkinsfile to use the SSH agent plugin (Credentials Binding plugin) to inject the credentials. The `sshagent` block should wrap all SSH- and SCP-based commands, as shown in the following listing.

Listing 10.14 SSH agent configuration

```
sshagent (credentials: ['swarm-sandbox']){
   stage('Copy'){
     sh "scp -o StrictHostKeyChecking=no
docker-compose.yml ec2-user@${swarmManager}:/home/ec2-user"
   }

   stage('Deploy stack'){
     sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager}
'\$(\$(aws ecr get-login --no-include-email --region ${region}))'
|| true"
     sh "ssh -oStrictHostKeyChecking=no
ec2-user@${swarmManager}
docker stack deploy --compose-file docker-compose.yml
--with-registry-auth watchlist"
   }
}
```

Push the changes to the develop branch. A new build should be triggered on the develop branch's nested job of the watchlist-deployment item.

> **NOTE** For continuous deployment, create a GitHub webhook on the GitHub repository to notify Jenkins on push events.

This time, the pipeline should be successful and turns green (figure 10.17).

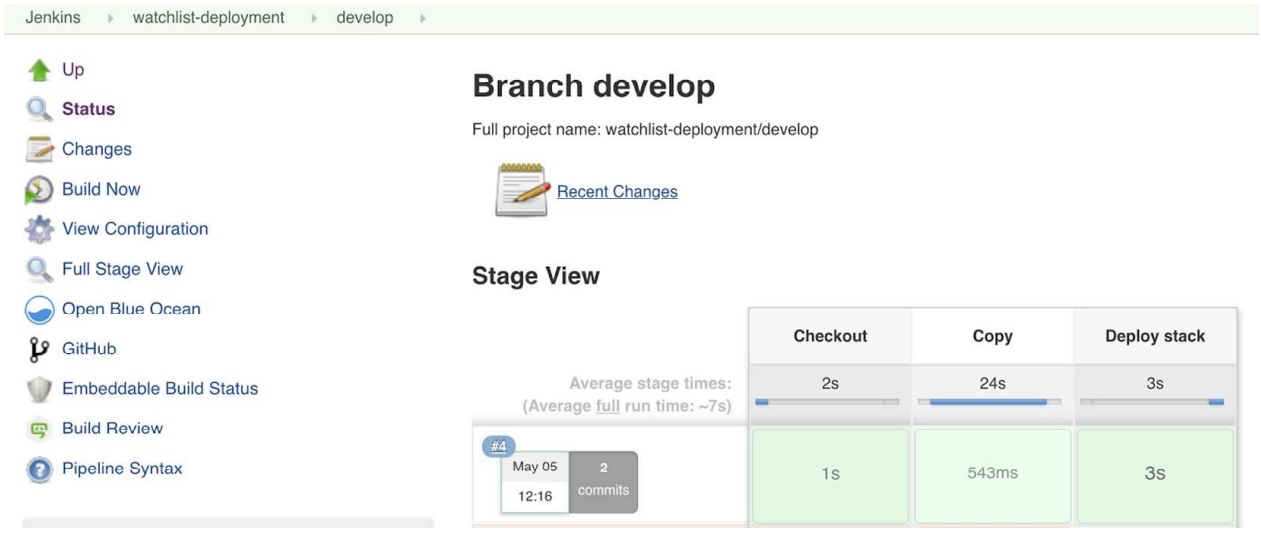

Figure 10.17 Continuous deployment pipeline

On the build logs side, Jenkins will run `docker stack deploy` over SSH on the Swarm manager, and the services in figure 10.18 will be deployed based on the `develop` tag image.

```
+ ssh -oStrictHostKeyChecking=no ec2-user@manager.sandbox.slowcoder.com docker stack deploy --compose-file docker-compose.yml --with-
registry-auth watchlist
Creating network watchlist_default
Creating service watchlist_mongodb
Creating service watchlist_movies-loader
Creating service watchlist_movies-parser
Creating service watchlist_movies-store
Creating service watchlist_movies-marketplace
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
$ ssh-agent -k
unset SSH_AUTH_SOCK;
unset SSH_AGENT_PID;
echo Agent pid 4018 killed;
[ssh-agent] Stopped.
[Pipeline] // sshagent
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
```

Figure 10.18 Output from `docker stack deploy`

> **NOTE** If you plan to use Amazon ECR as a remote repository, you need to assign an ECR IAM policy to the IAM instance profile assigned to Swarm instances.

On Swarm, type the following command, and we should be able to view the status of the stack and the services running within it:

```
docker service ls
```

The four microservices should be deployed alongside a MongoDB service, as shown in figure 10.19.

```
[ec2-user@ip-10-1-2-147 ~]$ docker service ls
ID             NAME                        MODE         REPLICAS   IMAGE
ro2ee4qwup7k   watchlist_mongodb           replicated   1/1        bitnami/mongodb:latest
rkmslcb50mr9   watchlist_movies-loader      replicated   0/1        305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-loader:develop
rkymrina4zi6   watchlist_movies-marketplace replicated   1/1        305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-marketplace:develop
afsaadidilfs   watchlist_movies-parser      replicated   0/1        305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-loader:develop
ftlt8tunlir2   watchlist_movies-store       replicated   1/1        305929695733.dkr.ecr.eu-west-3.amazonaws.com/mlabouardy/movies-store:develop
[ec2-user@ip-10-1-2-147 ~]$
```

Figure 10.19 Stack successfully deployed on Swarm sandbox

Next, we will deploy an open source tool called Visualizer to visualize Docker services across a set of machines. Execute these commands on the Swarm manager machine:

```
docker service create --name=visualizer
--publish=8080:8080/tcp
--constraint=node.role==manager \
     --mount=type=bind,src=/var/run/docker.sock,dst=/var/run/docker.sock \
     dockersamples/visualizer
```

Once the service is deployed, we will create a public load balancer to forward incoming HTTP and HTTPS (optional) traffic to port 8080, which is the port the Visualizer UI is exposed to. Declare the ELB resource in the following listing or download the resources file from chapter8/services/loadbalancers.tf.

Listing 10.15 Visualizer load balancer

```
resource "aws_elb" "visualizer_elb" {
  subnets                     = var.public_subnets
  cross_zone_load_balancing = true
  security_groups             = [aws_security_group.elb_visualizer_sg.id]
  listener {
    instance_port      = 8080
    instance_protocol  = "http"
    lb_port            = 443
    lb_protocol        = "https"
    ssl_certificate_id = var.ssl_arn
  }
  listener {
    instance_port      = 8080
    instance_protocol  = "http"
    lb_port            = 80
    lb_protocol        = "http"

  }
  health_check {
    healthy_threshold   = 2
    unhealthy_threshold = 2
    timeout             = 3
    target              = "TCP:8080" resource "aws_autoscaling_attachment"
     "cluster_attach_visualizer_elb" {
  autoscaling_group_name = var.swarm_managers_asg_id
  elb                     = aws_elb.visualizer_elb.id
}

    interval            = 5
  }
}
```

Then, we attach the load balancer to the ASG of the Swarm managers. The load balancer can also be assigned to the Swarm workers. In fact, all of the nodes within the Swarm cluster are aware of the location of every container within the cluster via the gossip network. If an incoming request hits a node that is not currently running the service for which that request was intended, the request will be routed to a node that is running a container for that service.

This is so nodes don't have to be purpose-built for specific services. Any node can run any service, and every node can be load balanced equally, reducing complexity and the number of resources needed for an application. This feature is called *mesh routing*:

```
resource "aws_autoscaling_attachment" "cluster_attach_visualizer_elb" {
  autoscaling_group_name = var.swarm_managers_asg_id
  elb                     = aws_elb.visualizer_elb.id
}
```

The following listing (chapter8/services/dns.tf) is not mandatory, but can be used to create a friendly DNS record pointing to the Visualizer load balancer FQDN.

Listing 10.16 Visualizer DNS configuration

```
resource "aws_route53_record" "visualizer" {
  zone_id = var.hosted_zone_id
  name    = "visualizer.${var.environment}.${var.domain_name}"
  type    = "A"
  alias {
    name                   = aws_elb.visualizer_elb.dns_name
    zone_id                = aws_elb.visualizer_elb.zone_id
    evaluate_target_health = true
  }
}
```

NOTE Update the security group of the Swarm cluster to allow incoming inbound traffic on port 8080 from the load balancer security group. Add an ingress rule for port 8080 and use `terraform apply` for changes to take effect.

Once changes are issued, point the browser to the load balancer URL displayed in the `Outputs` section in your terminal session. This handy tool, shown in figure 10.20, helps you see which containers are running, and on which nodes.

NOTE This tool works only with Docker Swarm mode in Docker Engine 1.12.0 and later. It does not work with the separate Docker Swarm project.

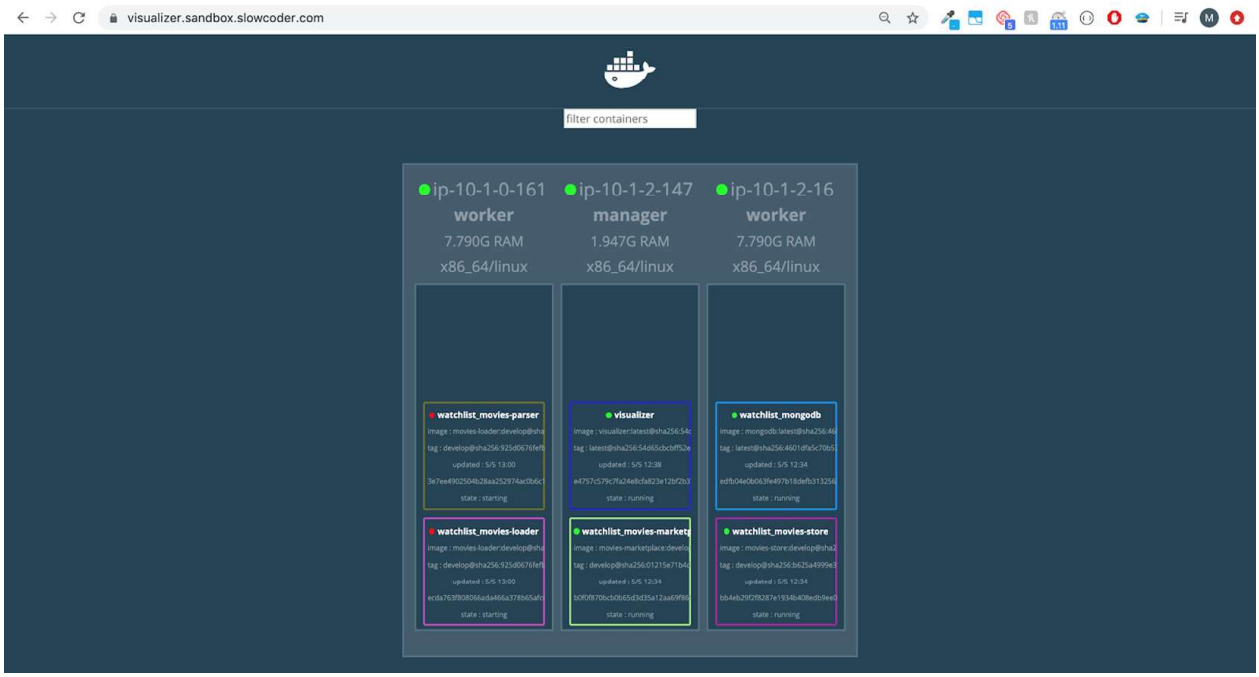

Figure 10.20 Visualizer dashboard

NOTE Containers are deployed on the manager, too. If you want to restrict deployment to workers, use Docker constraints with labels.

We have successfully deployed our application stack to Swarm. However, for now, the deployment is triggered manually. Ultimately, we want the deployment job to be executed at the end of each CI pipeline's successful execution.

To do so, update the Jenkinsfile (chapter10/pipelines/movies-loader/Jenkinsfile) to trigger the external job with the `build job` keyword. For example, on the movies-loader Jenkinsfile, add the following `Deploy` stage code block to the end of the pipeline:

```
stage('Deploy'){
        if(env.BRANCH_NAME == 'develop'){
            build job: "watchlist-deployment/${env.BRANCH_NAME}"
        }
}
```

Commit and push the changes to a feature branch. Then create a pull request (PR) to merge to develop. A new build should be triggered on the feature branch, and once it's done, Jenkins will post the build status on the PR, as shown in figure 10.21.

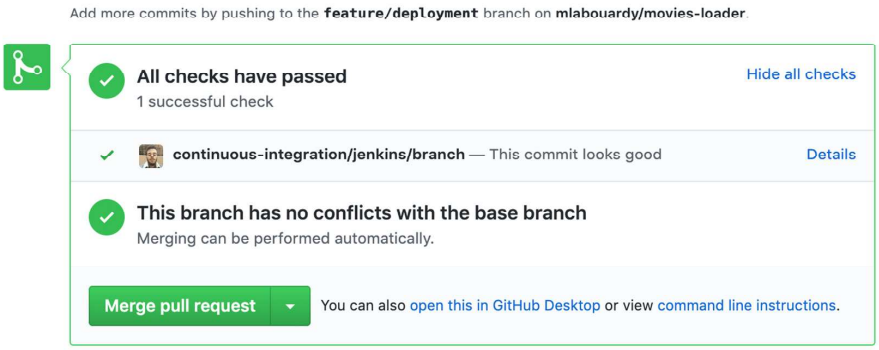

Figure 10.21 Pull request build status

Once the pull request is validated, we merge to the develop branch, and a new build will be triggered on that branch, as shown in figure 10.22.

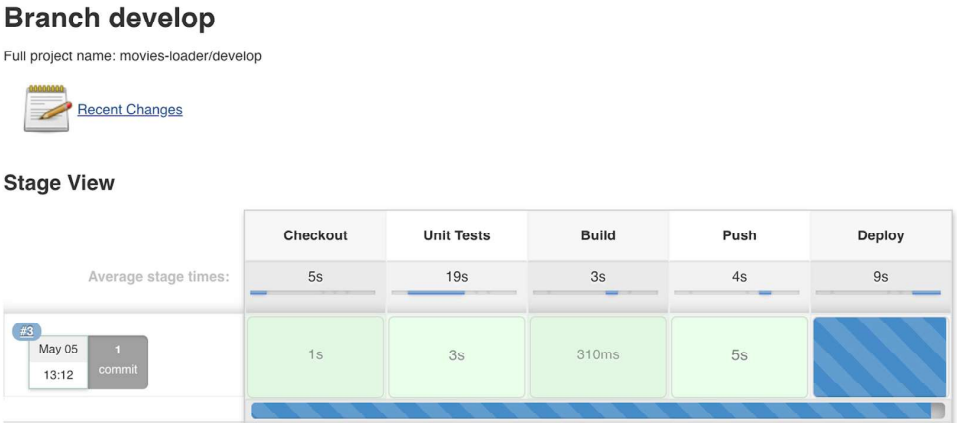

Figure 10.22 Jenkins CI/CD pipeline for the movies-loader project

At the end of the CI pipeline, the deploy stage will be executed, and watchlist-deployment will be triggered on the develop branch, as shown in figure 10.23.

Figure 10.23 External job triggering

That will trigger the deployment job, which will deploy the stack and force the pull of new Docker images with the `develop` tag. Repeat the same process for other GitHub repositories. In the end, each repository will trigger a deployment to sandbox if the CI is successfully executed, as shown in figure 10.24.

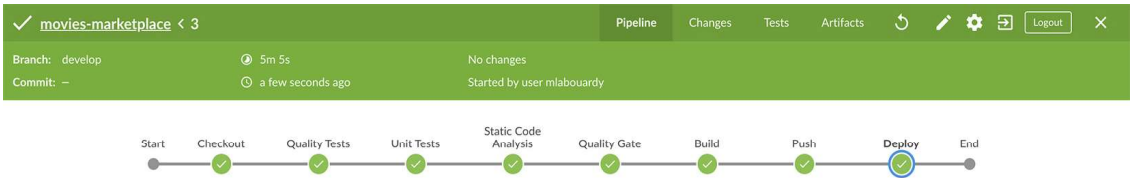

Figure 10.24 Marketplace CI/CD pipeline execution

> **NOTE** In chapters 11 and 12, we will cover how to run automated health checks and post-integration tests on the deployed application from Jenkins within the CI/CD pipeline.

By now, our application is deployed to the Swarm sandbox environment. To access the application, we need to create two public load balancers: one for the API (movies-store) and another for the frontend (movies-marketplace). Use Terraform template files available in the GitHub repository (under the /chapter8/services folder) to create the AWS resources, and then issue `terraform apply` to provision the resources. At the end of the deployment process, the marketplace and store API access URLs will be displayed in the `Outputs` section, as shown in figure 10.25.

```
aws_route53_record.movies_store: Creating...
aws_route53_record.movies_marketplace: Creating...
aws_route53_record.movies_marketplace: Still creating... [10s elapsed]
aws_route53_record.movies_store: Still creating... [10s elapsed]
aws_route53_record.movies_store: Still creating... [20s elapsed]
aws_route53_record.movies_marketplace: Still creating... [20s elapsed]
aws_route53_record.movies_store: Still creating... [30s elapsed]
aws_route53_record.movies_marketplace: Still creating... [30s elapsed]
aws_route53_record.movies_store: Still creating... [40s elapsed]
aws_route53_record.movies_marketplace: Still creating... [40s elapsed]
aws_route53_record.movies_store: Creation complete after 47s [id=Z2TR95QTU3UIUT_api.sandbox.slowcoder.com_A]
aws_route53_record.movies_marketplace: Creation complete after 48s [id=Z2TR95QTU3UIUT_marketplace.sandbox.slowcoder.com_A]

Apply complete! Resources: 2 added, 0 changed, 0 destroyed.

Outputs:

marketplace = https://marketplace.sandbox.slowcoder.com
store = https://api.sandbox.slowcoder.com
visualizer = https://visualizer.sandbox.slowcoder.com
```

Figure 10.25 Terraform apply output

NOTE Make sure to allow inbound traffic on ports 80 (frontend), 8080 (visualizer), and 3000 (API) from the security group attached to the Swarm EC2 instances.

For the marketplace to be able to interact with the RESTful API to show a list of crawled movies, we need to inject the API URL at the build time of the marketplace Docker image. The source code of the marketplace contains multiple files based on the target environment (figure 10.26).

Figure 10.26 Angular environment files

Each file contains the right API URL. For the sandbox environment, the environment .sandbox.ts file will be used, as shown in the following listing.

Listing 10.17 Marketplace sandbox environment variables

```
export const environment = {
  production: false,
  apiURL: 'https://api.sandbox.slowcoder.com',
};
```

The marketplace Docker image will be built using the `ng build -c sandbox` flag, which will replace the environment.ts file with environment.sandbox.ts values; see figure 10.27.

Figure 10.27 Docker image build execution

Once the new image is deployed to Swarm, point your browser to the marketplace URL. It should display the top 100 IMDb best movies in history, as shown in figure 10.28.

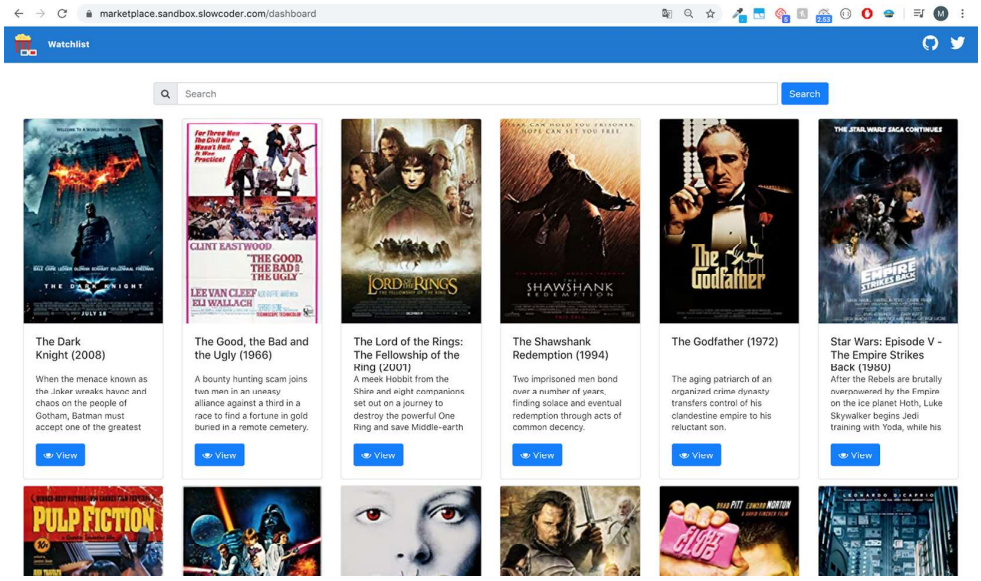

Figure 10.28 Watchlist marketplace dashboard

That's how to reach continuous deployment. However, we want to alert the development and product teams of the deployment and CI/CD status of the project.

10.3 Integrating Jenkins with Slack notifications

At certain stages of the pipeline, you may decide you want to send out a Slack notification to your team to inform them of the build status. To send Slack messages through Jenkins, we need to provide a way for our job to authorize itself with Slack.

Luckily for us, Slack has a prebuilt Jenkins integration that makes things pretty easy. Install the plugin from http://mng.bz/xXOB. Replace `WORKSPACE` with your Slack workspace name, as shown in figure 10.29.

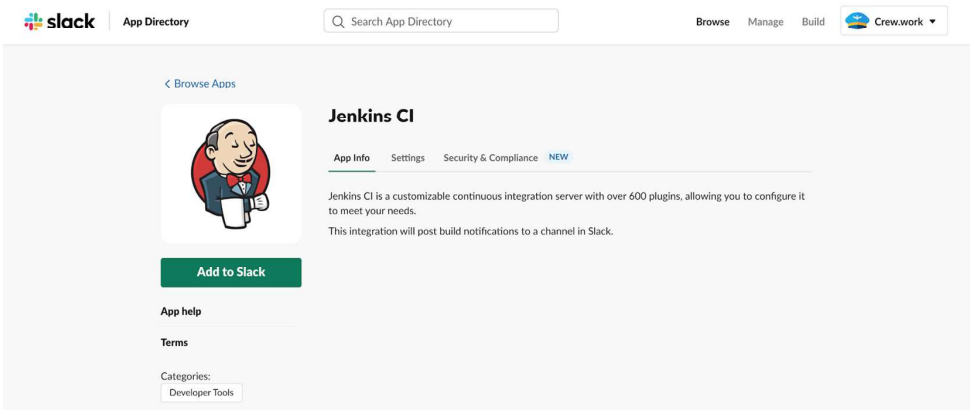

Figure 10.29 Jenkins CI Slack integration

Click the Add to Slack button. Then select the channel on which you want Jenkins to send notifications, as shown in figure 10.30.

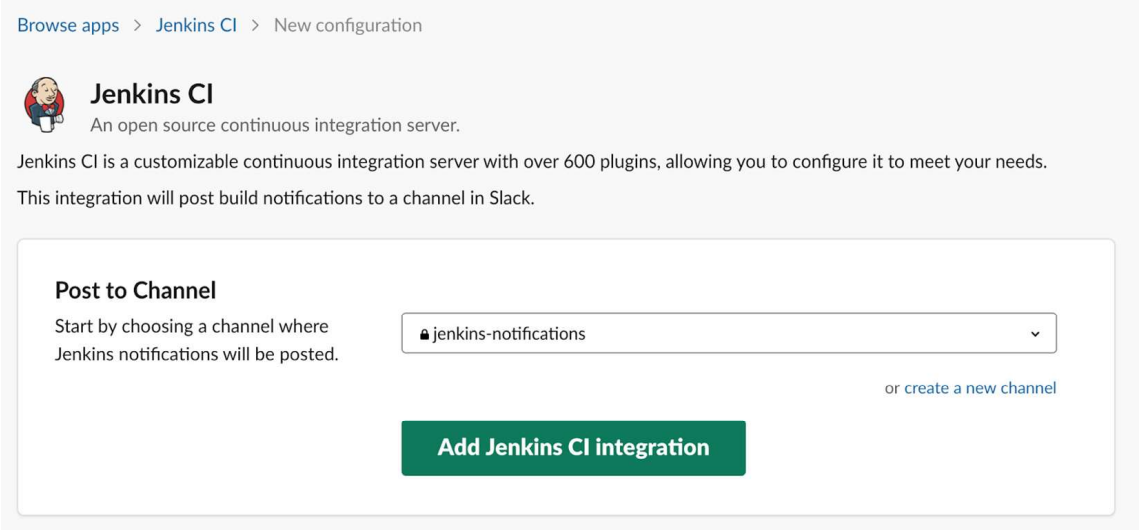

Figure 10.30 Slack channel configuration

After that, we need to set the configuration on the Jenkins Slack Notification plugin (https://plugins.jenkins.io/slack/), which is already installed on the baked Jenkins master machine image. Enter the team workspace name, integration token created on your slack, and channel name, as shown in figure 10.31, and click the Apply and Save buttons.

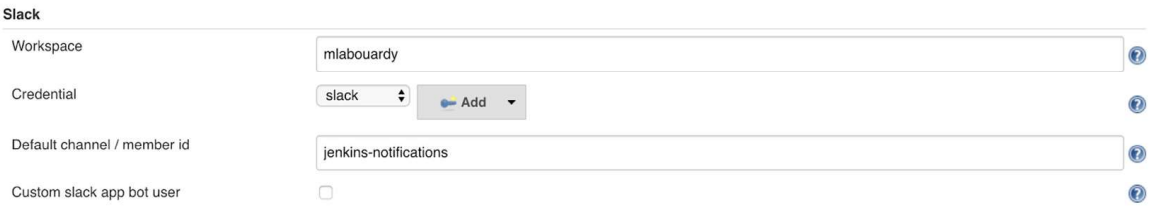

Figure 10.31 Jenkins Slack Notification plugin

Now that we have Slack properly configured in Jenkins, we can configure our CI/CD pipeline to send a notification to broadcast the status of the build with the following method:

```
slackSend (color: colorCode, message: summary)
```

Let's add this instruction at the end of the CI/CD pipeline for the movies-loader service as an example; see the following listing.

Listing 10.18 Jenkins Slack plugin DSL

```
node('workers'){
    stage('Checkout'){}

    stage('Unit Tests'){}

    stage('Build'){}

    stage('Push'){}

    stage('Deploy'){}

    slackSend (color: '#2e7d32',
message: "${env.JOB_NAME} has been successfully deployed")
}
```

NOTE For simplicity, I skipped steps that run unit tests, build the image, and push the image to the registry. You're advised to put them inside the workflow we are about to explore.

Push the changes to a feature branch, and then merge to develop. At the end of the pipeline, a new Slack notification will be sent, as shown in figure 10.32.

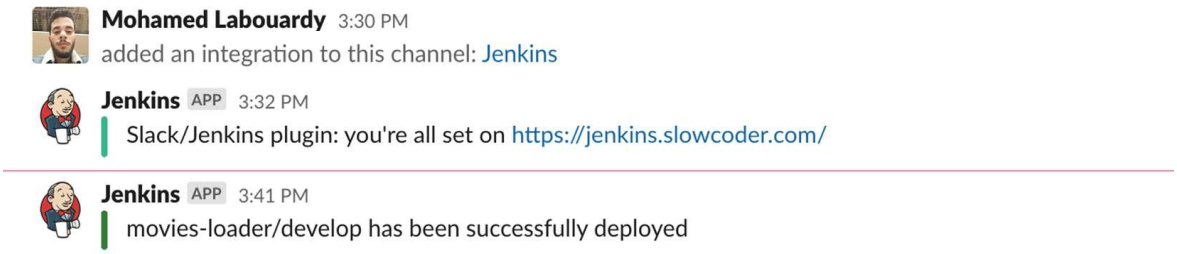

Figure 10.32 Jenkins Slack notification

While this works, we also want to be notified when the pipeline fails. That's where `try-catch` blocks come into play to handle errors thrown by pipeline stages; see the following listing.

Listing 10.19 Slack notifications within Jenkins

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

        stage('Unit Tests'){}
        stage('Build'){}
```

```
        stage('Push'){}
        stage('Deploy'){}
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
    }
}
```

This time, a `notifySlack()` method is used, which sends a notification with a different color based on the pipeline build status, as shown in the following listing.

Listing 10.20 Custom Slack notification message color

```
def notifySlack(String buildStatus){
    buildStatus =  buildStatus ?: 'SUCCESSFUL'
    def colorCode = '#FF0000'

    if (buildStatus == 'STARTED') {
        colorCode = '#546e7a'
    } else if (buildStatus == 'SUCCESSFUL') {
        colorCode = '#2e7d32'
    } else {
        colorCode = '#c62828c'
    }
    slackSend (color: colorCode,
message: "${env.JOB_NAME} build status: ${buildStatus}")
}
```

Colors the border along the left side of the message

Sends a Slack message with the job name by using the env.JOB_NAME, and build status by using the buildStatus variable

Based on your build result, the code sends Slack notifications as shown in figure 10.33.

Figure 10.33 Build status notification

Let's simulate a build failure by throwing an error, by adding the following instruction to the `Build` stage:

```
error "Build failed"
```

Push the changes to GitHub. The pipeline will fail at the `Build` stage (figure 10.34).

Figure 10.34 Throwing an error within the Jenkins pipeline

On the Slack channel, this time we will receive a notification with the build status set to Failure, as you can see in figure 10.35.

Figure 10.35 Build failure Slack notification

In the following listing, we take this further. We'll add more information to the notification, such as the author of the push event, Git commit ID, and message.

Listing 10.21 Custom Slack notification message attributes

```
def notifySlack(String buildStatus){
    buildStatus =  buildStatus ?: 'SUCCESSFUL'
    def colorCode = '#FF0000'
    def subject = "Name: '${env.JOB_NAME}'\n
Status: ${buildStatus}\nBuild ID: ${env.BUILD_NUMBER}"
    def summary = "${subject}\nMessage: ${commitMessage()}
\nAuthor: ${commitAuthor()}\nURL: ${env.BUILD_URL}"

    if (buildStatus == 'STARTED') {
        colorCode = '#546e7a'
    } else if (buildStatus == 'SUCCESSFUL') {
        colorCode = '#2e7d32'
    } else {
        colorCode = '#c62828c'
    }
    slackSend (color: colorCode, message: summary)
}
```

Displays the job's name, its status, and build number

Holds the subject's value and Git info (author, commit message) and build URL

The `notifySlack()` method will call `commitAuthor()` and `commitMessage()` to get the appropriate information. The `commitAuthor()` method will return the name of the commit author by executing the `git show` command, as shown in the following listing.

Listing 10.22 Git helper function to fetch the author

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

Displays the commit message's author with the git show command, saves the output to the commitAuthor file

Reads the commitAuthor file and trims extra spaces

And the `commitMessage()` method will use the `git log` command alongside the `HEAD` flag to fetch the commit message description; see the following listing.

Listing 10.23 Git helper function to fetch the commit message

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

Displays the last commit message description and saves the output in a commitMessage file

Reads the commitMessage content and trims extra spaces

If we push the changes, at the end of the CI/CD pipeline, the Slack notifications should contain the name of Jenkins job, build ID and its status, author name, and commit description, as shown in figure 10.36.

Figure 10.36 Slack notification with Git commit details