Under a production environment, you would replace the load balancer FQDN with an alias in Route 53. Refer to the official AWS documentation for instructions: http://mng.bz/Rq8P.

11.4 Packaging Kubernetes applications with Helm

So far, you have seen how to create one single chart for the microservices-based application and how to create a new release with Jenkins upon new Git commits. Another way of packaging the application is to create separate charts for each microservice, and then reference those charts as dependencies in the main chart (similar to a MongoDB chart). Figure 11.20 illustrates how Helm charts are packaged within a CI/CD pipeline.

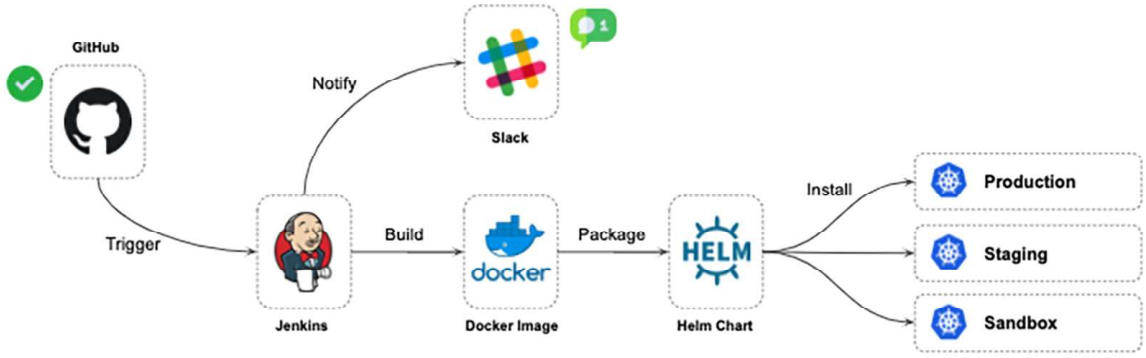

Figure 11.20 CI/CD of containerized application with Helm

On a push event, a Jenkins build will be triggered to build the Docker image and package the new release in a Helm chart. From there, the new chart is deployed to the corresponding Kubernetes environment. Along the way, a Slack notification is sent to notify the developers about the pipeline status.

On the movies-marketplace project, create a new Helm chart in the top-level directory by typing the following command:

```
helm create chart
```

It should create a new folder called chart with the following structure:

```
.
├── Chart.yaml
├── charts
├── templates
│   ├── deployment.yaml
│   └── service.yaml
└── values.yaml
```

As mentioned earlier, a Helm chart consists of metadata used to help describe the application, define constraints on the minimum required Kubernetes and/or Helm

version, and manage the version of the chart. All of this metadata lives in the Chart.yaml file (chapter11/microservices/movies-marketplace), shown in the following listing.

Listing 11.19 Movie loader chart

```
apiVersion: v2
name: movies-marketplace
description: UI to browse top 100 IMDb movies
type: application
version: 1.0.0
appVersion: 1.0.0
```

To be able to reference this chart from the main watchlist chart, we need to store it somewhere. Many open source solutions are available for storing Helm charts. GitHub can be used as a remote registry for Helm charts. Create a new GitHub repository called watchlist-charts and create an empty index.yaml file. This file will contain the metadata about available charts in the repository.

> **NOTE** Nexus Repository OSS supports Helm charts as well. You can publish charts to a Helm-hosted repository on Nexus.

Then, push this file to the master branch by issuing these commands:

```
git clone https://github.com/mlabouardy/watchlist-charts.git
cd watchlist-charts
touch index.yaml
git add index.yaml
git commit -m "add index.yaml"
git push origin master
```

The GitHub repository will look like figure 11.21.

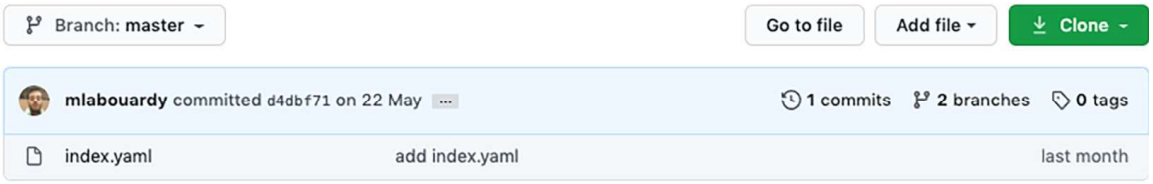

Figure 11.21 Helm charts GitHub repository

The Helm repository is an HTTP server that has a file index.yaml and all your chart files. To turn the GitHub repository into an HTTP server, we will enable GitHub pages.

Click the Settings tab. Scroll down to the GitHub Pages section and select the master branch as a source, as shown in figure 11.22.

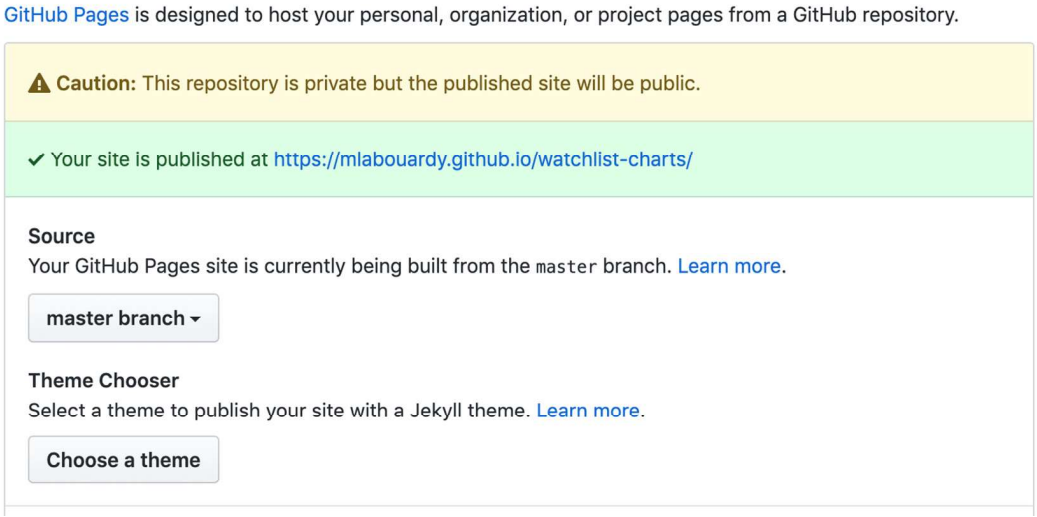

Figure 11.22 Enabling GitHub pages

With the private Helm repository ready to be used, let's package and publish our first Helm chart. On the movies-marketplace project, update the `Build` stage to use a parallel build to build the Docker image and the Helm chart. The `Build` stage should look like the following listing. (The complete Jenkinsfile is available at chapter11/pipeline/movies-marketplace/Jenkinsfile.)

Listing 11.20 Building the Docker image and Helm chart

```
stage('Build') {
 parallel(
  'Docker Image': {
   switch (env.BRANCH_NAME) {
    case 'develop':
       docker.build(imageName, '--build-arg ENVIRONMENT=sandbox .')
     break
    case 'preprod':
       docker.build(imageName, '--build-arg ENVIRONMENT=staging .')
     break
    ...
   }
  },
  'Helm Chart': {
     sh 'helm package chart'
  }
 )
}
```

Builds the appropriate Docker image by injecting the target environment settings

Packages the application in a Helm chart

The `helm package` command, as its name indicates, packages the chart directory into a chart archive (movies-marketplace-1.0.0.tgz). Finally, update the `Push` stage to use a parallel step as well, as shown in the following listing.

Listing 11.21 Storing the Docker image in a private registry

```
stage('Push') {
 parallel(
  'Docker Image': {
    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')
        }
        ...
    }
  },
  'Helm Chart': {
    ...
  }
 )
}
```

Authenticates with ECR in order to push the Docker images afterward

Tags and stores the image in ECR

Publishes the Helm chart to GitHub—see listing 11.22 for complete instructions.

The `Helm Chart` stage will clone the watchlist-charts GitHub repository with the `git clone` command, and add the metadata of the new packaged Helm chart to index.yaml with the `helm repo index` command. Then it pushes index.yaml and the archive chart to the Git repository; see the following listing.

Listing 11.22 Publishing the Helm chart to GitHub

```
'Helm Chart': {
    sh 'helm repo index --url https://mlabouardy.github.io/watchlist-charts/ .'
    sshagent(['github-ssh']) {
      sh 'git clone git@github.com:mlabouardy/watchlist-charts.git'
      sh 'mv movies-marketplace-1.0.0.tgz watchlist-charts/'
      dir('watchlist-charts'){
          sh 'git add index.yaml movies-marketplace-1.0.0.tgz
&& git commit -m "movies-marketplace"
&& git push origin master'
      }
    }
  }
```

Generates an index file, given a directory containing packaged charts

Provides SSH credentials to builds via an ssh-agent

Changes current directory to watchlist-charts folder

Commits and pushes the archive and index file to GitHub

If you push the new Jenkinsfile to the Git remote repository, a new pipeline will be triggered, as shown in figure 11.23. At the `Build` stage, the movies-marketplace Docker image and Helm chart will be packaged. Next, the `Push` stage will be executed to push the Docker image to the Docker private registry and the Helm chart to the GitHub repository.

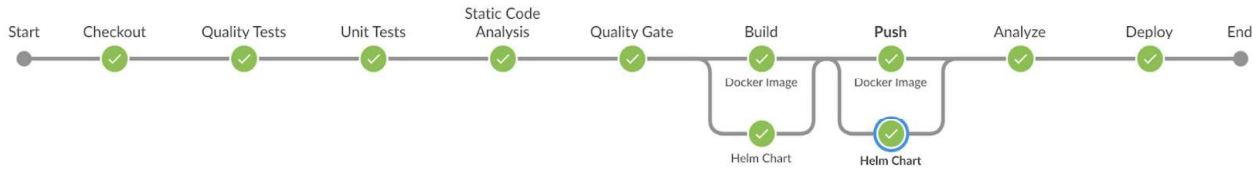

Figure 11.23 CI/CD workflow with Helm and Docker

Upon the completion of the CI/CD pipeline, a new archived chart will be available in the GitHub repository, as shown in figure 11.24.

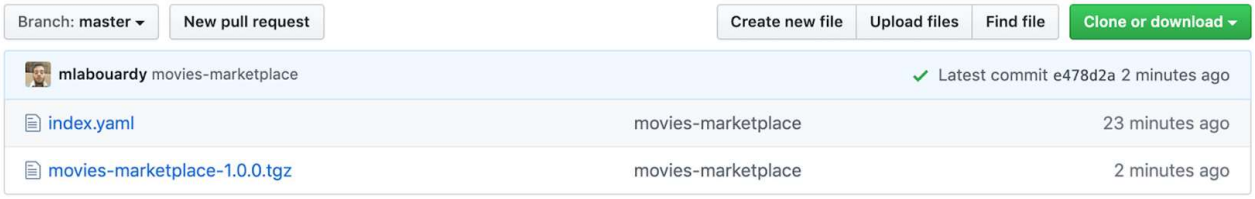

Figure 11.24 Packaging the Movies Marketplace chart

The index.yaml file will reference the newly built Helm chart under the `entries` section, as you can see in figure 11.25.

Figure 11.25 Helm repository metadata

You can override the chart version set in Chart.yaml by providing the new version with the `--version` flag at the time of packaging a Helm chart:

```
sh 'helm package chart --app-version ${appVersion} --version ${chartVersion}'
```

Repeat the same steps for other repositories to create a Helm chart per service. Once done, the Helm charts repository should contain four archived files (figure 11.26).

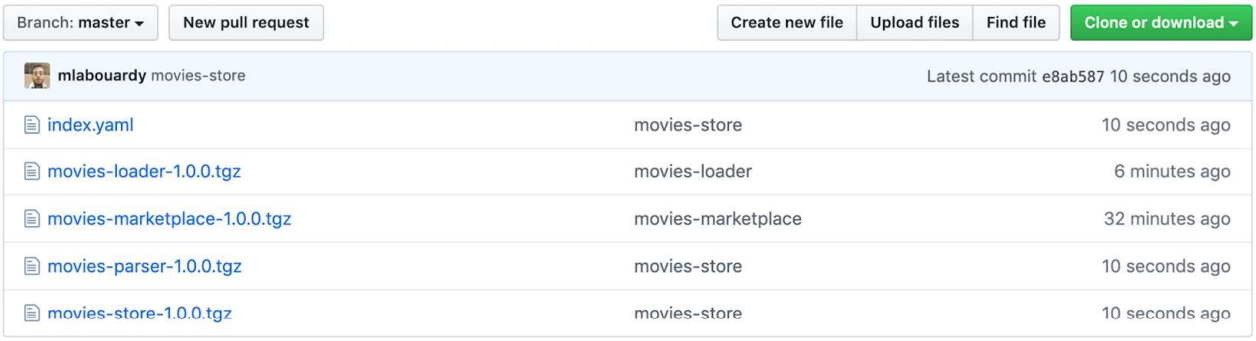

Figure 11.26 Application charts stored in the GitHub repository

Next, we configure the GitHub repository as a Helm repository:

```
helm repo add watchlist https://mlabouardy.github.io/watchlist-charts
```

Finally, we can reference these charts in the watchlist Chart.yaml file under the `dependencies` section, as shown in the following listing.

Listing 11.23 Watchlist application charts

```
apiVersion: v2
name: watchlist
description: Top 100 iMDB best movies in history
type: application
version: 1.0.0
appVersion: 1.0.0
maintainers:
    - name: Mohamed Labouardy
      email: mohamed@labouardy.com
dependencies:
  - name: mongodb
    version: 7.8.10
    repository: https://charts.bitnami.com/bitnami
    alias: mongodb
  - name: movies-loader
    version: 1.0.0
    repository: https://mlabouardy.github.io/watchlist-charts
  - name: movies-parser
    version: 1.0.0
    repository: https://mlabouardy.github.io/watchlist-charts
  - name: movies-store
    version: 1.0.0
```

```
    repository: https://mlabouardy.github.io/watchlist-charts
  - name: movies-marketplace
    version: 1.0.0
    repository: https://mlabouardy.github.io/watchlist-charts
```

Now that all pieces are running together and we checked the core functionality, let's validate that the solution is up for a typical GitFlow development process.

11.5 Running post-deployment smoke tests

The microservices are deployed. However, that doesn't mean these services are properly configured and correctly performing all the jobs that they're supposed to be doing.

You want to have a health check that indicates the current health operation of your services. You can set up a simple one by implementing an HTTP request to a service URL and check whether the response code is 200.

For instance, let's implement a health check for the movies-store service. Update the Jenkinsfile of the movies-store project (chapter11/pipeline/movies-store/Jenkinsfile) to add the function shown in the following listing.

Listing 11.24 Groovy function to return API URL

```
def getUrl(){
    switch(env.BRANCH_NAME){
        case 'preprod':
            return 'https://api.staging.domain.com'
        case 'master':
            return 'https://api.production.domain.com'
        default:
            return 'https://api.sandbox.domain.com'
    }
}
```

The function returns the service URL based on the current Git branch name. Finally, we add a `Healthcheck` stage at the end of the pipeline to issue a cURL command on the service URL:

```
stage('Healthcheck'){
    sh "curl -m 10 ${getUrl()}"
}
```

The `-m` flag is used to set a time-out of 10 seconds, to give Kubernetes enough time to pull the latest built image and deploy the changes into the cluster before checking the service health status.

Once you push the changes to the Git remote repository, a new build will be triggered. Upon the completion of the CI/CD pipeline, a cURL command will be executed with a GET request on the service URL, as shown in figure 11.27.

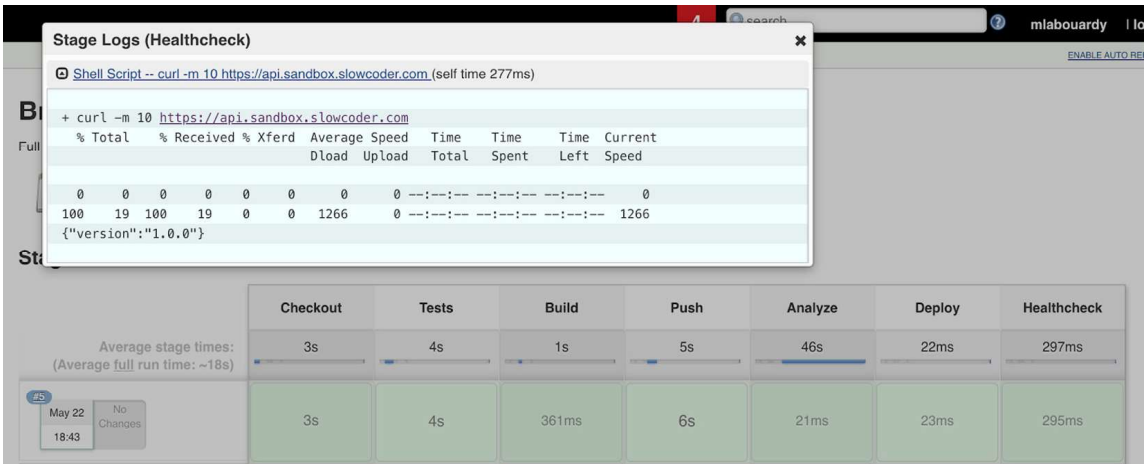

Figure 11.27 cURL command output

If the service responds before the expiration time-out, the cURL command will return a successful exit code. Otherwise, an error will be thrown to make the pipeline fail.

However, if the service is responding, that doesn't mean it's working correctly or a new version of the service has been successfully deployed.

To be able to issue advanced HTTP requests against the service URL, we will install the Jenkins HTTP Request plugin (www.jenkins.io/doc/pipeline/steps/http_request/) from the Jenkins Plugins page, as shown in figure 11.28.

Figure 11.28 Jenkins HTTP Request plugin

We can now update the movies-store's Jenkinsfile. The plugin offers an `httpRequest` DSL object that can be used to call a remote URL. In the following listing, `httpRequest` returns a response object that exposes the response body through a `content` attribute. Then, we use the `JsonSlurper` class to parse the response to a JSON object. The updated `Healthcheck` stage is shown in the following listing.

Listing 11.25 Movie store Healthcheck stage

```
stage('Healthcheck'){
    def response = httpRequest getUrl()
```

```
    def json = new JsonSlurper().parseText(response.content)
    def version = json.get('version')

    if version != '1.0.0' {
       error "Expected API version 1.0.0 but got ${version}"
    }
}
```

The service returns the version number deployed in Kubernetes. This value is fixed in the service source code, but you can inject the Jenkins build ID as a version number while building the Docker image of the service and check whether the returned version is equal to the Jenkins build ID at the `Healthcheck` stage.

Figure 11.29 shows the end result of the CI/CD pipeline of each microservice running in Kubernetes.

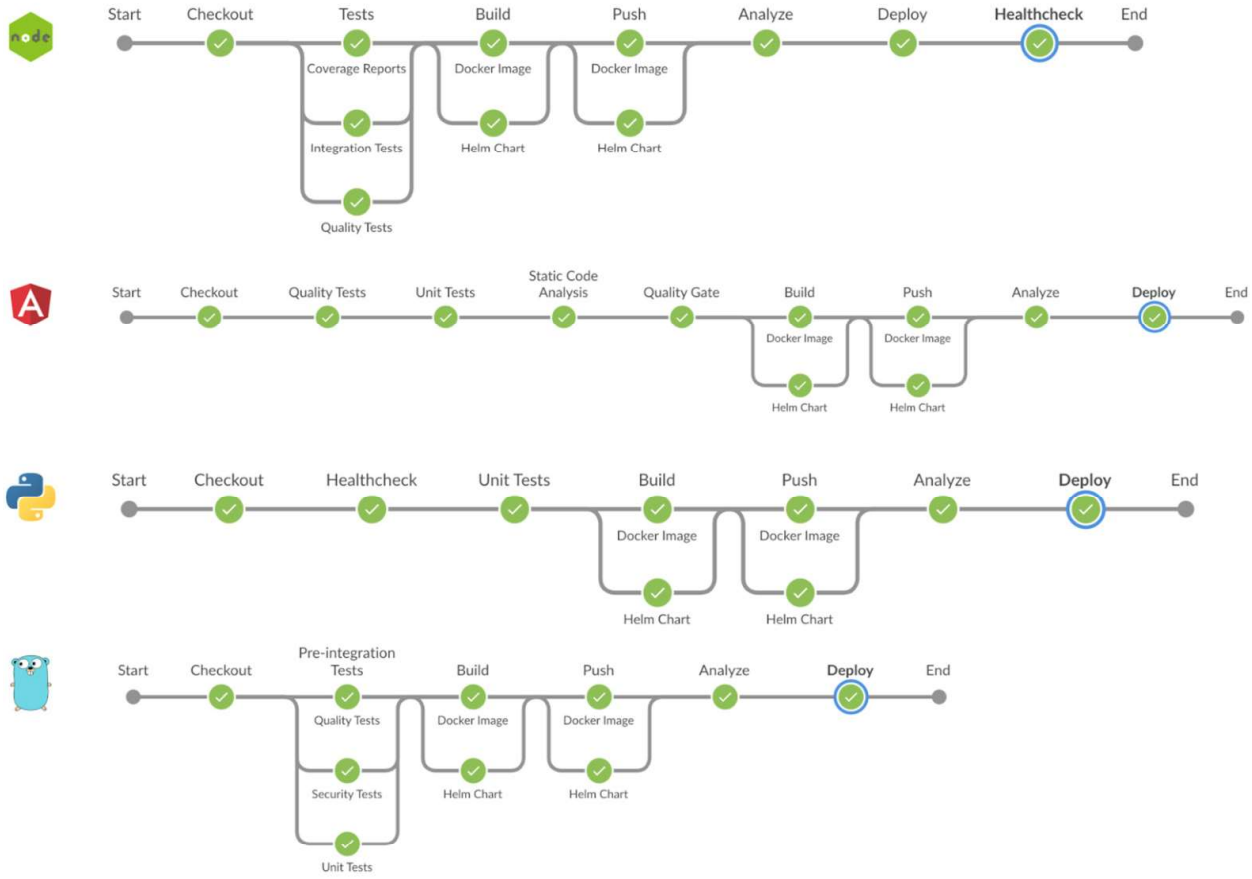

Figure 11.29 Complete CI/CD workflow for containerized microservices

When you opt for Jenkins to build cloud-native applications running in Kubernetes, you're required to create extensive configurations, as well as spending considerable time learning and using all of the necessary plugins to make it happen. Fortunately, Jenkins X comes into play to offer simplicity and ready-to-go templates.

11.6 Discovering Jenkins X

Jenkins X (https://jenkins-x.io/) is a CI/CD solution for modern cloud applications on Kubernetes. It's used to simplify the configurations and lets you harness the power of Jenkins 2.0. It also lets you use open source tools like Helm, Artifact Hub, ChartMuseum, Nexus, and Docker Registry to easily build cloud-native applications.

Jenkins X adds what's missing from Jenkins: comprehensive support for continuous delivery and managing the promotion of projects to preview, staging, and production environments running in Kubernetes. It uses GitOps to manage the configuration and version of the Kubernetes resources that are deployed to each environment. So each environment has its own Git repository that contains all the Helm charts, their versions, and the configuration for the applications to be run in the environment.

When following this methodology, Git is the single source of truth for both the infrastructure as code and the application code. All changes to the desired state are Git commits. So it's easy to see who made changes when, and more importantly, it's then easy to revert changes that cause bad things to happen.

With that being said, let's get our hands dirty and cover how Jenkins X works. To get started, install the Jenkins X CLI, and pick the most suitable instructions for your operating system: http://mng.bz/20ZX. Run `jx version --short` to make sure you're on the latest stable version. I'm using version 2.1.71 at the time of writing this book.

Jenkins X runs on a Kubernetes cluster. If you're running on one of the major cloud providers (Amazon EKS, GKE, or AKS), Jenkins X provides multiple approaches for creating this cluster:

```
jx create cluster eks --cluster-name=watchlist
Jx create cluster aks --cluster-name=watchlist
Jx create cluster gke --cluster-name=watchlist
Jx create cluster iks --cluster-name=watchlist
```

> **NOTE** You can run Jenkins X on the existing EKS cluster by referring to the official guide at https://jenkins-x.io/v3/admin/setup/operator/.

Install Jenkins X on a K8s cluster by issuing the following command from your terminal session:

```
jx boot
```

You will be asked a series of questions that will configure the installation, as shown in figure 11.30.

When the installation is done, you will be presented with useful links and the password for your Jenkins X–related services. Don't forget to save it somewhere for future use.

Jenkins X also deploys a set of supporting services, including the Jenkins dashboard, Docker Registry, ChartMuseum, and Artifact Hub to manage Helm charts, and Nexus, which serves as a Maven and npm repository.

```
Creating staging Environment in namespace jx
Created environment staging
Namespace jx-staging created
Created Jenkins Project: http://jenkins.jx.35.198.184.208.nip.io/job/mlabouardy/job/environment-watchlist-staging/

Note that your first pipeline may take a few minutes to start while the necessary images get downloaded!

Triggered Jenkins job:  http://jenkins.jx.35.198.184.208.nip.io/job/mlabouardy/job/environment-watchlist-staging/
Creating GitHub webhook for mlabouardy/environment-watchlist-staging for url http://jenkins.jx.35.198.184.208.nip.io/github-webhook/
Using Git provider github.com at https://github.com
? Using Git user name: mlabouardy
? Using organisation: mlabouardy
Creating repository mlabouardy/environment-watchlist-production
Creating Git repository mlabouardy/environment-watchlist-production
Pushed Git repository to https://github.com/mlabouardy/environment-watchlist-production

Creating production Environment in namespace jx
Created environment production
Namespace jx-production created
Created Jenkins Project: http://jenkins.jx.35.198.184.208.nip.io/job/mlabouardy/job/environment-watchlist-production/

Note that your first pipeline may take a few minutes to start while the necessary images get downloaded!

Triggered Jenkins job:  http://jenkins.jx.35.198.184.208.nip.io/job/mlabouardy/job/environment-watchlist-production/
Creating GitHub webhook for mlabouardy/environment-watchlist-production for url http://jenkins.jx.35.198.184.208.nip.io/github-webhook/

Jenkins X installation completed successfully


        ********************************************************

             NOTE: Your admin password is: u?zTUzZFGMH79UN0C~4u

        ********************************************************
```

Figure 11.30 Jenkins X installation output

The following is the output of the `kubectl get svc` command:

```
NAME              HOSTS                                    ADDRESS         PORTS   AGE
chartmuseum       chartmuseum.jx.34.89.183.25.nip.io       34.89.183.25    80      2m52s
docker-registry   docker-registry.jx.34.89.183.25.nip.io   34.89.183.25    80      2m53s
jenkins           jenkins.jx.34.89.183.25.nip.io           34.89.183.25    80      2m53s
nexus             nexus.jx.34.89.183.25.nip.io             34.89.183.25    80      2m53s
```

Point your browser to the Jenkins URL printed during the installation process and sign in with the admin username and password displayed previously in figure 11.30. The dashboard in figure 11.31 should be served.

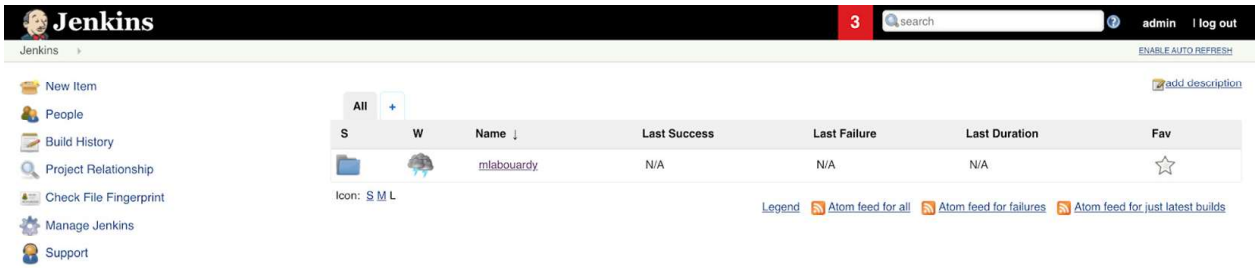

Figure 11.31 Jenkins web dashboard

It is possible to run Jenkins in serverless mode while installing Jenkins X. Then, instead of running the Jenkins web dashboard, which continuously consumes CPU and memory resources, you can run Jenkins only when you need it.

The Jenkins X installation also creates two Git repositories by default: one for your staging environment and one for production, as shown in figure 11.32:

- *Staging*—Any merge performed on the project master branch will automatically be deployed as a new version to staging (auto promote).
- *Production*—You will have to manually promote your staging application version into production by using a `jx promote` command.

Figure 11.32 Application deployment environments

Jenkins X uses these repositories to manage deployments to each environment, and promotions are done via Git pull requests. Each repository contains a Helm chart that specifies the applications to be deployed to the corresponding environment. Each repository also has a Jenkinsfile to handle promotions.

Now that you have a working cluster with Jenkins X installed, we are going to create an application that can be built and deployed with Jenkins X. For clarity, I have created a RESTful API in Go that serves an HTTP endpoint with a list of the top 100 IMDb movies. We will import this project inside Jenkins with this command:

```
jx import
```

If you wish to import a project that is already in a remote Git repository, you can use the `--url` argument:

```
jx import --url https://github.com/mlabouardy/jx-movies-store
```

The following is the output of the import command:

```
WARNING: No username defined for the current Git server!
[? Do you wish to use mlabouardy as the Git user name: Yes
The directory /Users/mlabouardy/github/jx-movies-store is not yet using git
[? Would you like to initialise git now? Yes
[? Commit message:  Initial import

Git repository created
performing pack detection in folder /Users/mlabouardy/github/jx-movies-store
--> Draft detected JSON (63.955137%)
--> Could not find a pack for JSON. Trying to find the next likely language match...
--> Draft detected Go (36.044863%)
selected pack: /Users/mlabouardy/.jx/draft/packs/github.com/jenkins-x-buildpacks/jenkins-x-kubernetes/packs/go
? Who should be the owner of the repository? mlabouardy
replacing placeholders in directory /Users/mlabouardy/github/jx-movies-store
app name: jx-movies-store, git server: github.com, org: mlabouardy, Docker registry org: crew-sandbox
skipping directory "/Users/mlabouardy/github/jx-movies-store/.git"
Draft pack go added
[? Would you like to define a different preview namespace? No
Using Git provider github.com at https://github.com
? Using organisation: mlabouardy
[? Enter the new repository name:  jx-movies-store
Creating repository mlabouardy/jx-movies-store
Pushed Git repository to https://github.com/mlabouardy/jx-movies-store
Created Jenkins Project: http://jenkins.jx.35.198.184.208.nip.io/job/mlabouardy/job/jx-movies-store/
```

Jenkins X will go over the code and choose the right default build pack for the project based on the programming language. Our project was developed in Go, so it will be a Go build pack. Jenkins X will generate a Jenkinsfile, Dockerfile, and Helm chart based on the project runtime environment. The import command will create a remote repository on GitHub, register a webhook, and push the code to the remote repository, shown in figure 11.33.

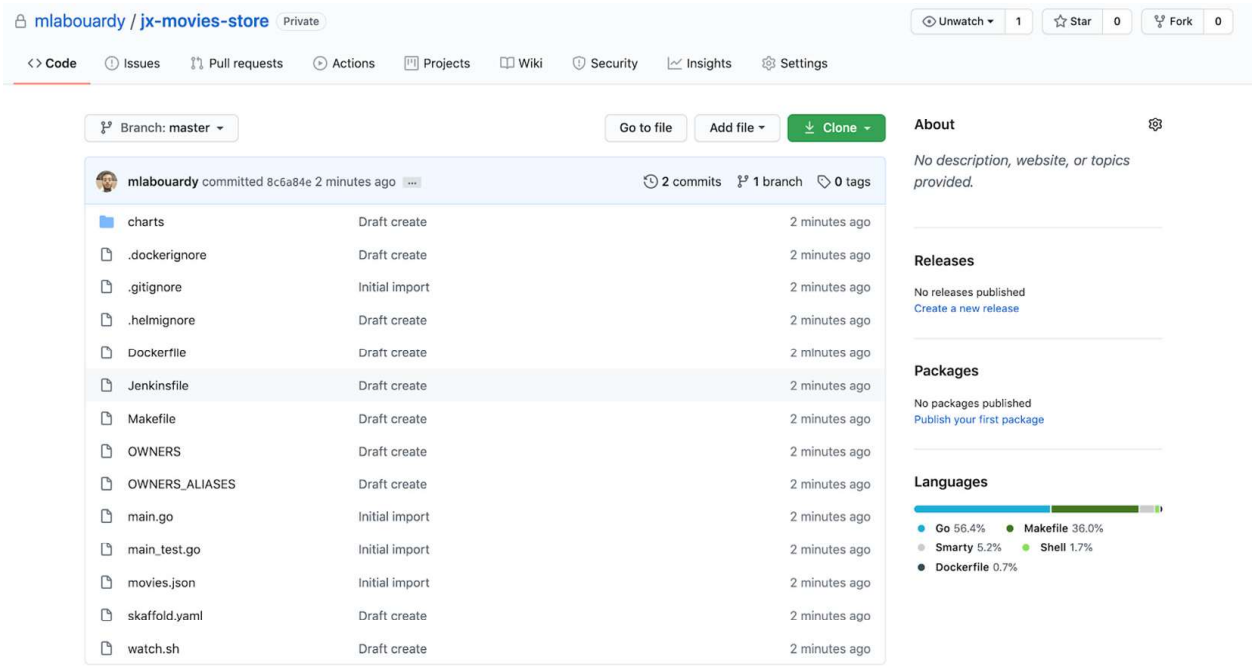

Figure 11.33 Application GitHub repository

Jenkins X will also automatically create a Jenkins multibranch pipeline job for the project, and the pipeline will be triggered. You can check the progress of the pipeline with this command:

```
jx get activity -f jx-movies-store -w
```

```
jenkins:jx-movies-store mlabouardy$ jx get activity -f jx-movies-store -w
STEP                                  STARTED AGO DURATION STATUS
mlabouardy/jx-movies-store/master #1         39s          Running
mlabouardy/jx-movies-store/master #1       2m13s          Running
  Checkout Source                             1s          Pending
mlabouardy/jx-movies-store/master #1       2m25s          Running
  Checkout Source                            13s      11s Succeeded
  CI Build and push snapshot                  2s          NotExecuted
  Build Release                               1s          Pending
```

You can also track the progress of the pipeline from the Jenkins dashboard by clicking the project job; figure 11.34 shows the result.

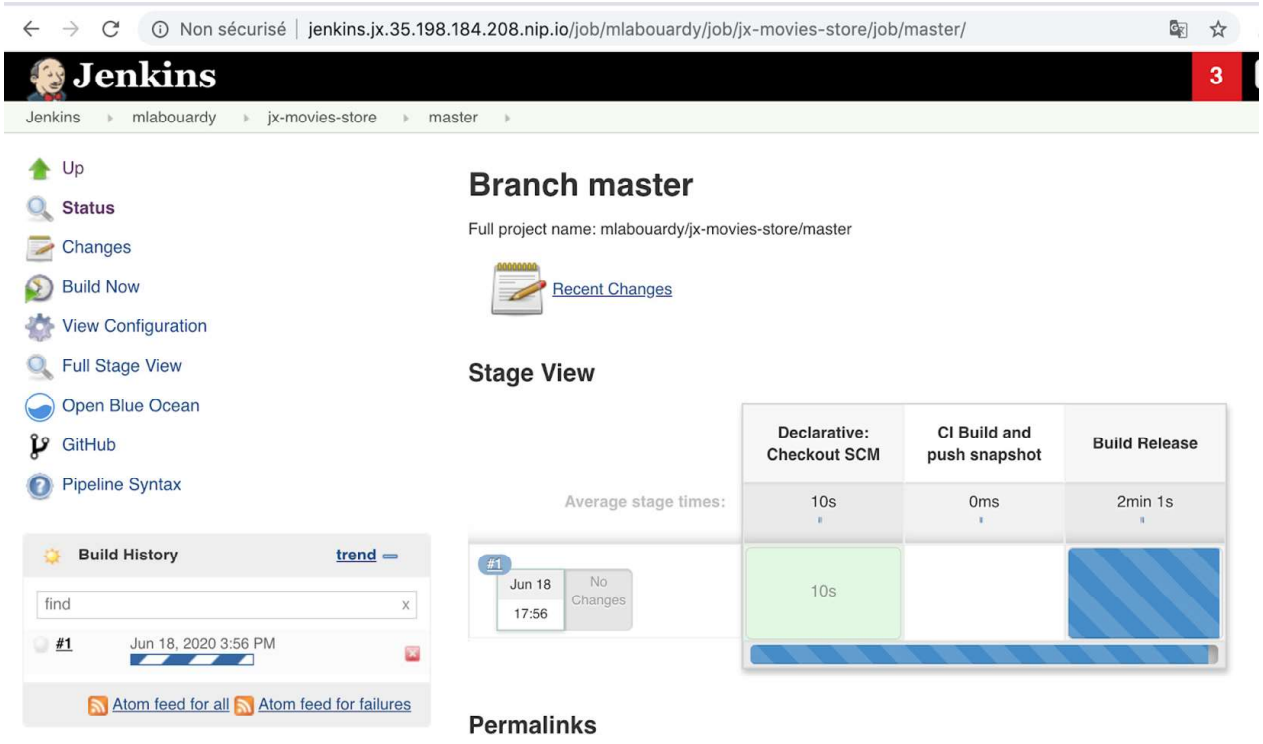

Figure 11.34 Application build pipeline

The pipeline stages are executed on a Kubernetes pod running in the Kubernetes cluster we provisioned earlier, as you can see in figure 11.35.

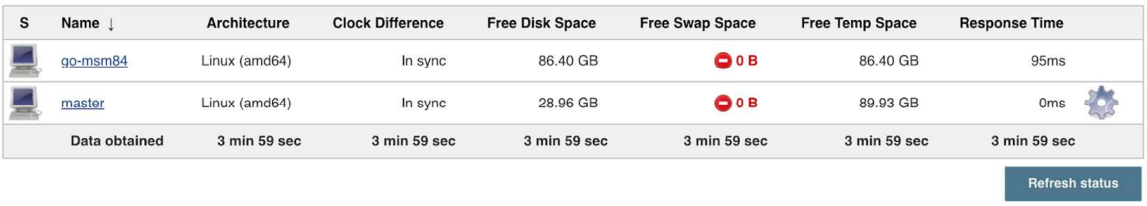

Figure 11.35 Jenkins workers based on Kubernetes pods

The executed pipeline will clone the repository, build the Docker image, and push it to a Docker registry, as shown in the following listing.

Listing 11.26 Build stage when an event occurs on master branch

```
stage('Build Release') {
      when {
        branch 'master'
      }
      steps {
```

```
        container('go') {
          dir('/home/jenkins/agent/go/src/
github.com/mlabouardy/jx-movies-store') {
            checkout scm
            sh "git checkout master"
            sh "git config --global credential.helper store"
            sh "jx step git credentials"
            sh "echo \$(jx-release-version) > VERSION"
            sh "jx step tag --version \$(cat VERSION)"
            sh "make build"
            sh "export VERSION=`cat VERSION`
&& skaffold build -f skaffold.yaml"
            sh "jx step post build --image $DOCKER_REGISTRY/$ORG/
     $APP_NAME:\$(cat VERSION)"
          }
        }
      }
}
```

A Helm chart will be packaged and pushed to the ChartMuseum repository, and a new release will be published on the project GitHub repository, as shown in figure 11.36. Jenkins X uses semantic versioning for tagging.

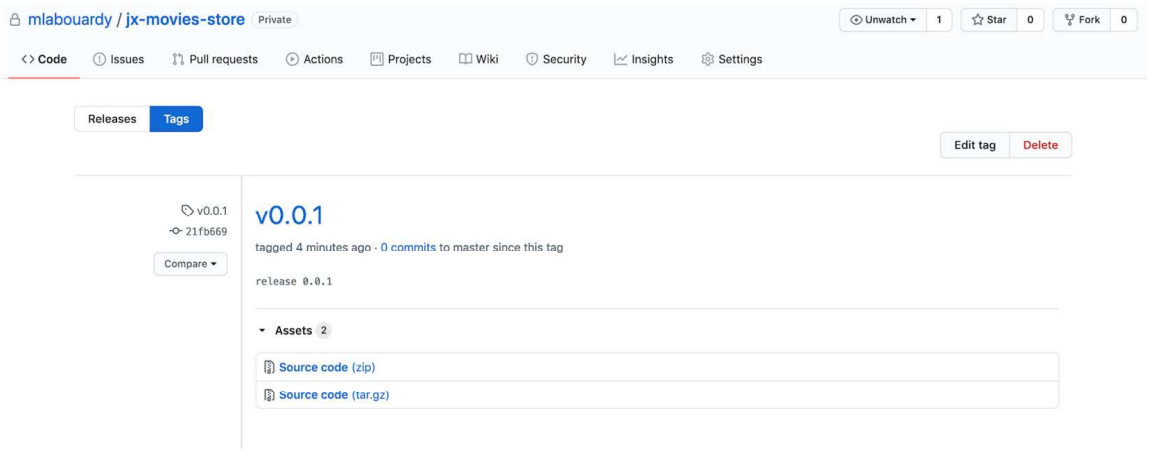

Figure 11.36 Publishing the application release

The release will be promoted automatically to the staging environment, as shown in figure 11.37.

Figure 11.37 Jenkins pipeline on the master branch

During the promotion stage, a new PR will be created by Jenkins X to deploy the new release to staging. This PR will add our application and its version in the env/requirements.yaml file inside the Git repository, as shown in figure 11.38.

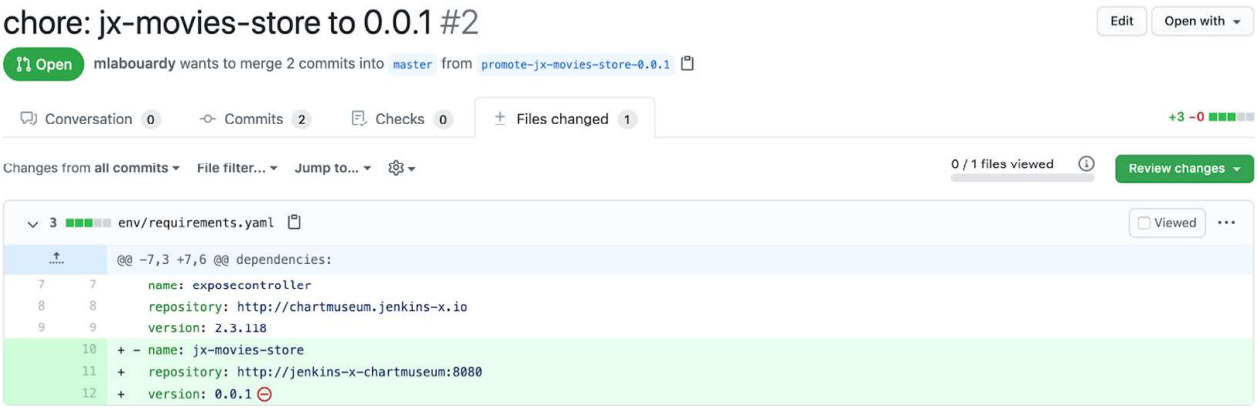

Figure 11.38 Promoting the application to staging

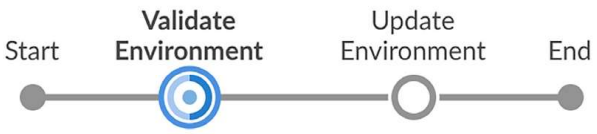

Figure 11.39 Deploying an application to staging

Now you can see that the multibranch jx-movies-store pipeline is triggered for the pull request. It will check out the PR, perform a `helm build`, and execute tests on the environment along with code review and approval. When it's successful, it will merge the PR with the master, see figure 11.39.

Once the application is deployed, we can type `jx get applications` to get the access URL for the application, as shown in figure 11.40.

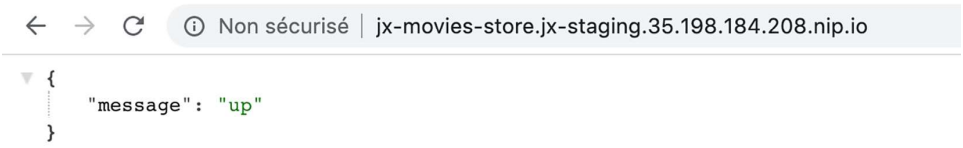

Figure 11.40 Application overall health status

Now we will update our application and see what will happen! Let's create a new feature branch:

```
git checkout -b feature/readme
git add README.md
git commit -m "update readme"
git push origin feature/readme
```

Jenkins X creates a GitHub webhook during the import of our application. This means we can just commit a change, and our application will be updated automatically, as shown in figure 11.41.

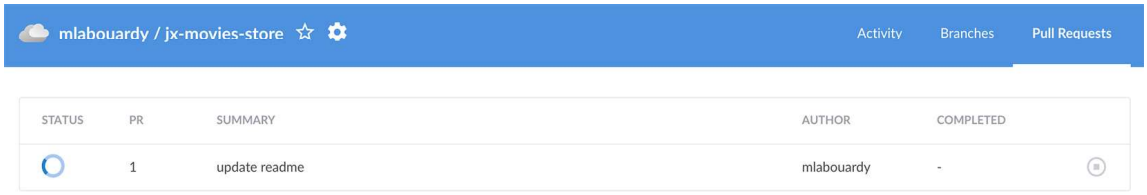

Figure 11.41 Building GitHub pull request

Jenkins X automatically spins up preview environments for our pull request, so we can get fast feedback before changes are merged to the master:

```
+ jx preview --app jx-movies-store --dir ../..
Creating a preview
Found commit author match for: mlabouardy with email address: mohamed@labouardy.com
Created environment mlabouardy-jx-movies-store-pr-1
Namespace jx-mlabouardy-jx-movies-store-pr-1 created
expose:
  Annotations:
    helm.sh/hook: post-install,post-upgrade
    helm.sh/hook-delete-policy: hook-succeeded
  config:
    domain: 35.198.184.208.nip.io
    exposer: Ingress
    http: "true"
preview:
  image:
    repository: 10.15.244.126:5000/crew-sandbox/jx-movies-store
    tag: 0.0.0-SNAPSHOT-PR-1-1
Cloning the Jenkins X versions repo https://github.com/jenkins-x/jenkins-x-versions.git with ref refs/heads/master to
/root/.jx/jenkins-x-versions
Updating PipelineActivities mlabouardy-jx-movies-store-pr-1-1 which has status Running
Preview application is now available at: http://jx-movies-store.jx-mlabouardy-jx-movies-store-pr-1.35.198.184.208.nip.io
```

Jenkins X creates a preview environment in the PR for the application changes and displays a link to evaluate the new feature, as shown in figure 11.42.

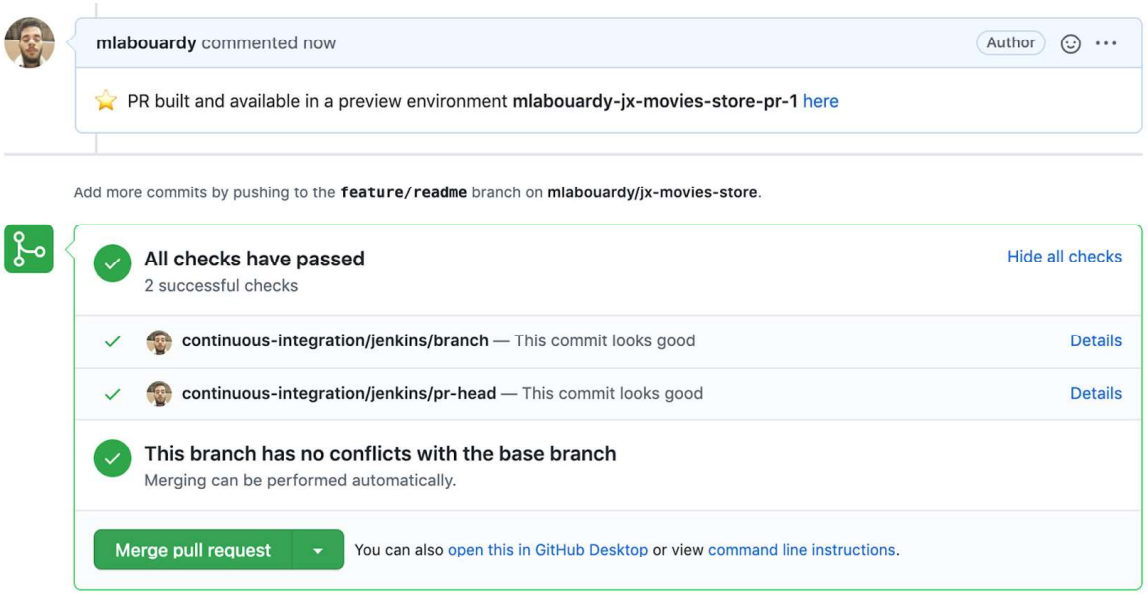

Figure 11.42 Pull request preview environment

The preview environment is created whenever a change is made to the repository, allowing any relevant user to validate or evaluate features, bug fixes, or security hotfixes. If we click the preview environment URL, we should have access to the service REST API, as shown in figure 11.43.

Figure 11.43 Movies Store API

Once the new changes are validated, we can confirm the code and functionality changes with an `/approve` comment, as shown in figure 11.44. This simple comment will merge the code changes back to the master branch and initiate a build on the master branch.

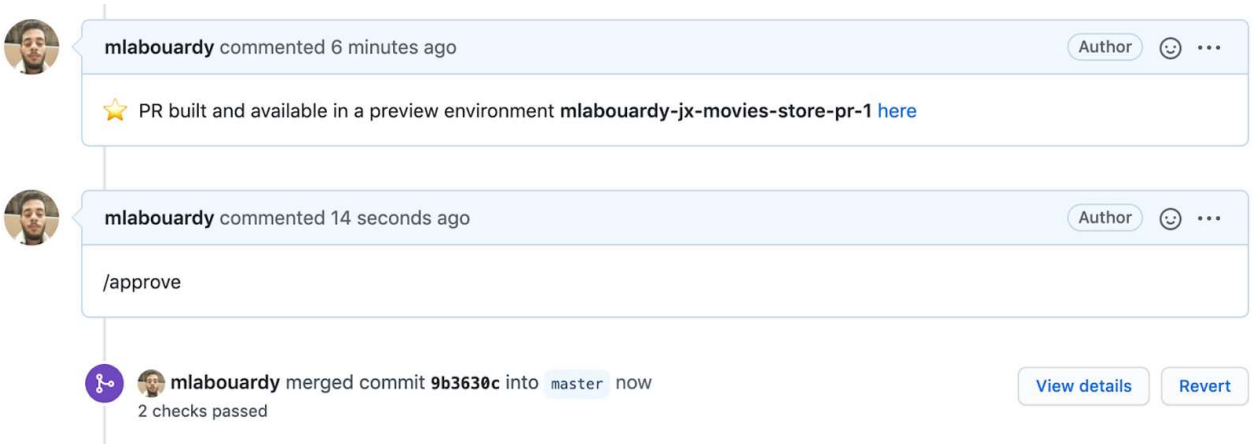

Figure 11.44 ChatOps commands within Git PR

Jenkins X offers multiple commands that can be used while managing pull requests. Each command triggers a specific action. Table 11.2 summarizes the most used commands.

Upon the completion of the build on the master branch, a new release will be published, as shown in figure 11.45.

Table 11.2 ChatOps commands

Command	Description
`/approve`	This PR can be merged. This command must be from someone in the repo OWNERS file.
`/retest`	Rerun any failed test pipeline contexts for this PR.
`/assign USER`	Assign the PR to the given user.
`/lgtm`	This PR looks good to me. This command can be from anyone with access to the repo.

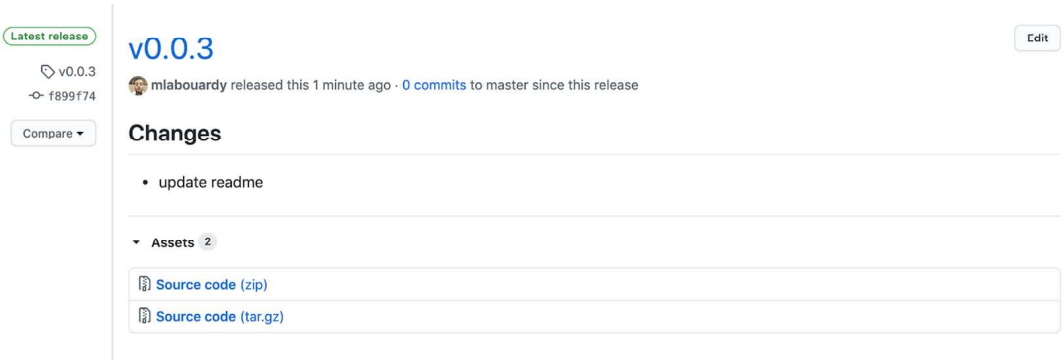

Figure 11.45 New application release

When you're satisfied with your application, you can use the `jx` CLI to promote the application to a different environment using a GitOps approach. For example, we can promote our application to production with the following command:

```
jx promote --app jx-movies-store --version 0.0.3 --env production
```

A new PR will be created, but this time on our production repository, and the environment-watchlist-production job is triggered, as shown in figure 11.46.

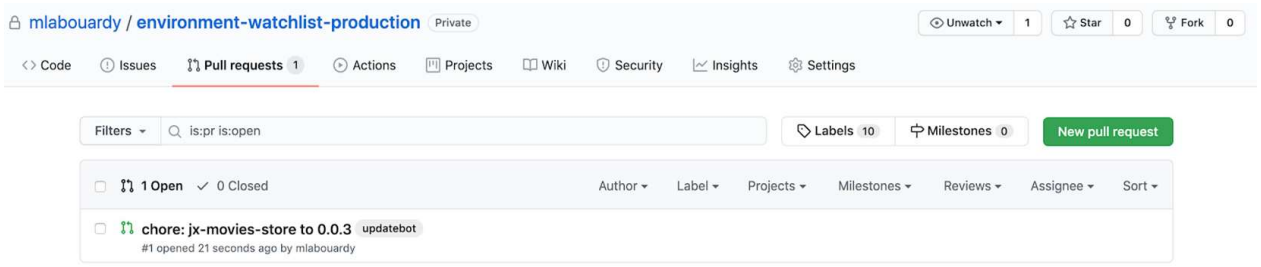

Figure 11.46 Promoting the application to production

Once the pull request is validated, the production pipeline runs Helm, which deploys the environment, pulling Helm charts from ChartMuseum, and Docker images from the Docker Registry. Kubernetes creates the project's resources, typically a pod, service, and ingress.

Jenkins X uses Git branch patterns to determine which branch names are automatically set up for CI/CD. By default, the master branch, and any branch starting with *PR-* or *feature* will be scanned. You can set up your own branch discovery mechanism with the following command:

```
jx import --branches "develop|preprod|master|PR-.*"
```

NOTE If you are done with your Amazon EKS cluster, you should delete it and its resources so that you do not incur additional charges. Issue a `terraform destroy` command to delete the AWS resources.

Summary

- Kubernetes manages containerized applications on clusters of nodes by helping operators deploy, scale, update, and maintain their services, and providing mechanisms for service discovery.
- The `kubectl apply` command can be used from Jenkins pipelines to perform deployments on K8s clusters.
- A Helm chart encapsulates Kubernetes object definitions and provides a mechanism for configuration at deployment time.
- GitHub pages have built-in support for installing Helm charts from an HTTP server.
- Jenkins X creates a Kubernetes pod for each agent started, defined by the Docker image to run, and stops it after each build.
- Jenkins X preview environments are used to get early feedback on changes to applications before the changes are merged into the master branch.
- Jenkins X does not aim to replace Jenkins but builds on it with best-of-breed open source tools. It's a great way to achieve CI/CD with batteries included, without having to assemble things together.

12 Lambda-based serverless functions

This chapter covers

- Implementing a CI/CD pipeline for a serverless-based application from scratch
- Setting up continuous deployment and delivery with AWS Lambda
- Separating multiple Lambda deployment environments
- Implementing API Gateway multistage deployments with Lambda alias and stage variables
- Delivering email notifications with attachments upon completion of CI/CD pipelines

In the previous chapters, you learned how to write a CI/CD pipeline for a containerized application running in both Docker Swarm and Kubernetes. In this chapter, you will learn how to deploy the same application written in a different architecture.

Serverless is the fastest-growing architectural movement right now. It allows developers to develop scalable applications faster by delegating the full responsibility of

managing the underlying infrastructure to the cloud provider. That said, going serverless carries several key challenges, one of which is CI/CD.

12.1 Deploying a Lambda-based application

Multiple serverless providers are out there, but to keep it simple, we'll use AWS—and specifically, AWS Lambda (https://aws.amazon.com/lambda/), which is the best known and most mature solution in the serverless space today. AWS Lambda, launched at AWS re:Invent 2014, was the first implementation of serverless computing. Users can upload their code to Lambda, which then performs operational and scaling activities on behalf of the users.

The service follows an event-driven architecture. This means the code deployed in Lambda can be triggered in response to events like HTTP requests coming from services like Amazon API Gateway (https://aws.amazon.com/api-gateway/).

Before going into further detail about how to create a CI/CD pipeline for a serverless application, we will look at the corresponding architecture. Figure 12.1 shows how serverless services like Amazon API Gateway, Amazon DynamoDB, Amazon S3, and AWS Lambda fit into the application architecture.

AWS Lambda empowers microservice development. That being said, each endpoint triggers a different Lambda function. These functions are independent of one another and can be written in different languages. Hence, this leads to scaling at the function level, easier unit testing, and loose coupling. All requests from clients first go through API Gateway. It then routes the incoming request to the right Lambda

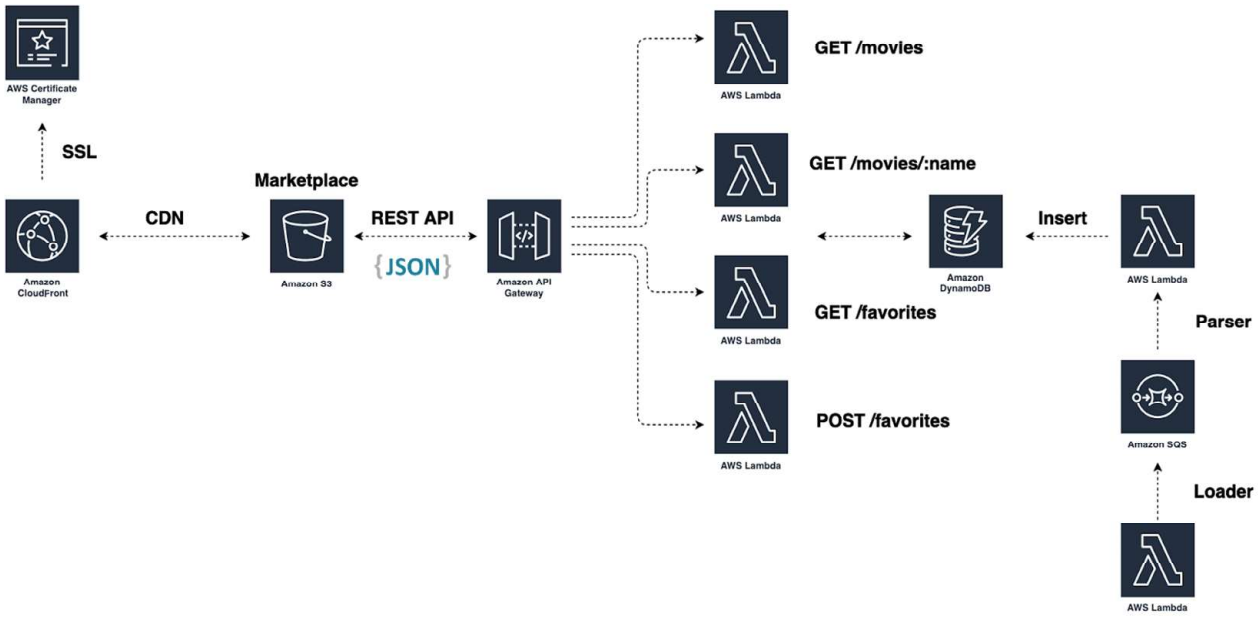

Figure 12.1 Watchlist application based on serverless architecture. Each Lambda function is responsible for a single API endpoint. The endpoints are managed through API Gateway and consumed by the Marketplace service hosted on an S3 bucket.

function accordingly. The functions are stateless, so that's where DynamoDB comes into the scene, to manage data persistence across Lambda functions. The Amazon S3 bucket is used to serve the marketplace static web application. Finally, an Amazon CloudFront distribution (optional) is used to deliver static assets such as Cascading Style Sheets (CSS) or JavaScript files from edge cache locations around the globe.

To deploy a Lambda function, we need to create an AWS Lambda resource and an IAM execution role with a list of AWS resources that the Lambda function has access to during runtime. For instance, the Lambda function `MoviesStoreListMovies` issues a `Scan` operation on a DynamoDB table to fetch a list of movies. Therefore, the Lambda execution role should grant access to the DynamoDB table.

To avoid duplication of code and provide a lightweight abstraction for creating Lambda functions, we will use Terraform modules. A *module* is a container for multiple resources that are used together.

> **NOTE** You can use Terraform Registry (https://registry.terraform.io/) to download well-tested modules built by the community or publish your own modules remotely.

The module responsible for creating an AWS Lambda resource is located under the modules folder (chapter12/terraform/modules). Create a new lambda.tf file with a module block for each Lambda function, as shown in the following listing. The module resource references the custom module through the `source` argument and overrides default variables such as the Lambda runtime environment and environment variables.

Listing 12.1 Creating Lambda functions with the Terraform module

```
module "MoviesLoader" {
  source = "./modules/function"
  name = "MoviesLoader"
  handler = "index.handler"
  runtime = "python3.7"
  environment = {
    SQS_URL = aws_sqs_queue.queue.id
  }
}

module "MoviesParser" {
  source = "./modules/function"
  name = "MoviesParser"
  handler = "main"
  runtime = "go1.x"
  environment = {
    TABLE_NAME = aws_dynamodb_table.movies.id
  }
}

module "MoviesStoreListMovies" {
  source = "./modules/function"
  name = "MoviesStoreListMovies"
```

```
  handler = "src/movies/findAll/index.handler"
  runtime = "nodejs14.x"
  environment = {
    TABLE_NAME = aws_dynamodb_table.movies.id
  }
}

module "MoviesStoreSearchMovies" {
  source = "./modules/function"
  name = "MoviesStoreSearchMovies"
  handler = "src/movies/findOne/index.handler"
  runtime = "nodejs14.x"
  environment = {
    TABLE_NAME = aws_dynamodb_table.movies.id
  }
}

module "MoviesStoreViewFavorites" {
  source = "./modules/function"
  name = "MoviesStoreViewFavorites"
  handler = "src/favorites/findAll/index.handler"
  runtime = "nodejs14.x"
  environment = {
    TABLE_NAME = aws_dynamodb_table.favorites.id
  }
}

module "MoviesStoreAddToFavorites" {
  source = "./modules/function"
  name = "MoviesStoreAddToFavorites"
  handler = "src/favorites/insert/index.handler"
  runtime = "nodejs14.x"
  environment = {
    TABLE_NAME = aws_dynamodb_table.favorites.id
  }
}
```

This code will provision a `MoviesLoader` Lambda function based on the Python 3.7 runtime environment, a `MoviesParser` function based on the Go runtime, and a `MoviesStoreListMovies` function based on the Node.js environment.

Next, we will deploy a RESTful API with Amazon API Gateway and define HTTP endpoints to trigger the Lambda functions upon incoming HTTP/HTTPS requests. The Terraform code in listing 12.2 exposes a GET method on the /movies resource. When a GET method is invoked on the /movies endpoint, the `MoviesStoreList-Movies` Lambda function will be triggered to return a list of IMDb movies stored on the DynamoDB table. Add the code shown in the following listing to apigateway.tf.

Listing 12.2 GET /movies endpoint definition

```
resource "aws_api_gateway_resource" "path_movies" {
   rest_api_id = aws_api_gateway_rest_api.api.id
   parent_id   = aws_api_gateway_rest_api.api.root_resource_id
```

```
   path_part   = "movies"
}
module "GetMovies" {
  source = "./modules/method"
  api_id = aws_api_gateway_rest_api.api.id
  resource_id = aws_api_gateway_resource.path_movies.id
  method = "GET"
  lambda_arn = module.MoviesStoreListMovies.arn
  invoke_arn = module.MoviesStoreListMovies.invoke_arn
  api_execution_arn = aws_api_gateway_rest_api.api.execution_arn
}
```

NOTE In addition to providing a unified entry point for Lambda functions, API Gateway comes with powerful features such as caching, cross-origin resource sharing (CORS) configuration, security, and authentication.

Define the rest of the API endpoints, or download the complete apigateway.tf file from chapter12/terraform/apigateway.tf.

The Movies Marketplace content—including HTML, CSS, JavaScript, images, and other files—will be hosted in an Amazon S3 bucket. The end users will then access the application by using the public website URL exposed by Amazon S3. Hence, we don't need to run any web server such as NGINX or Apache to make the web application available. The Terraform code in the following listing (s3.tf) creates an S3 bucket and enables website hosting.

Listing 12.3 S3 website hosting configuration

```
resource "aws_s3_bucket" "marketplace" {
  bucket = "marketplace.${var.domain_name}"
  acl    = "public-read"
  website {
    index_document = "index.html"
    error_document = "index.html"
  }
}
```

The bucket access-control list (ACL) must be set to `public-read`. The `website` block is where we define the index document for the application. Also, we grant access to the static content by attaching a bucket policy. The bucket policy grants `s3:GetObject` to all principals for any object in the bucket.

NOTE Unless you want to access the marketplace via the S3 bucket URL, you can use CloudFront on top of S3 to serve the application content by using a custom domain name over SSL.

Install the local modules with the `terraform init` command and run `terraform apply` to provision the AWS resources. Creating the whole infrastructure should take a few seconds. After the creation steps are complete, the API and marketplace URLs will be displayed in the `Outputs` section, as you can see in figure 12.2.

```
Apply complete! Resources: 57 added, 0 changed, 0 destroyed.

Outputs:

api = https://kvgfot7n4l.execute-api.eu-west-3.amazonaws.com/test
marketplace = marketplace.slowcoder.com.s3-website.eu-west-3.amazonaws.com
```

Figure 12.2 API Gateway and S3 website URLs

The `api` variable holds the RESTful API URL powered by API Gateway, and the `marketplace` variable is the S3 website URL for the marketplace application. If you head to AWS Lambda console (http://mng.bz/10Qg), the Lambda functions in figure 12.3 should be deployed.

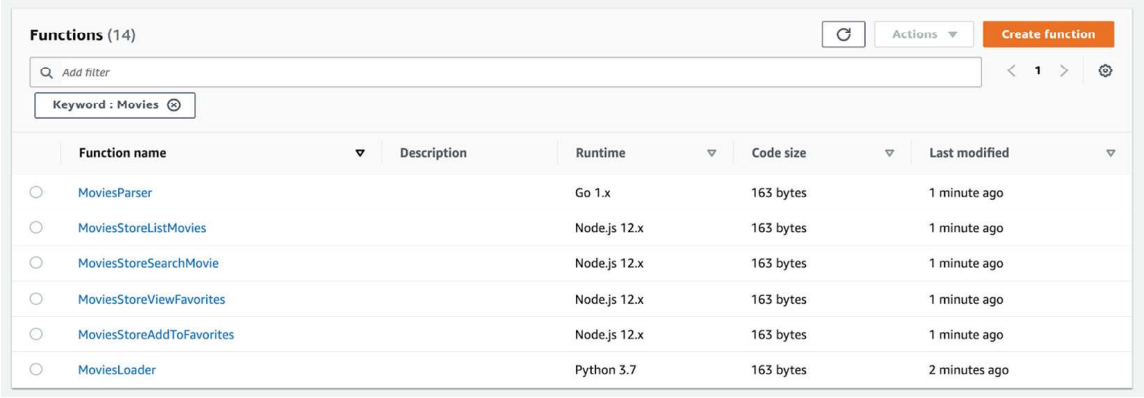

Figure 12.3 Watchlist application's Lambda functions

Point your favorite browser to the API Gateway URL, and navigate to the /movies endpoint. The HTTP request should trigger the `MoviesStoreListMovies` Lambda function responsible for listing movies. The error message in figure 12.4 will be displayed.

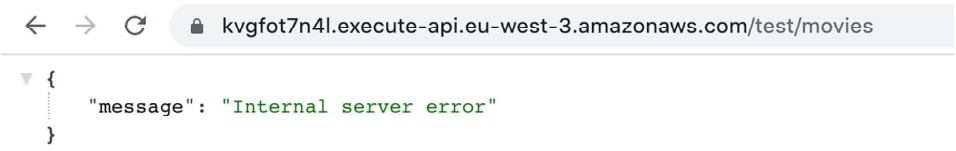

Figure 12.4 `MoviesStoreListMovies` HTTP response

Right now, no code is deployed to Lambda functions, so there would be nothing to see. To list movies, we need to deploy the function's code to the Lambda resource. In the upcoming section, we will create a CI/CD pipeline in Jenkins to automate the deployment of Lambda functions. Figure 12.5 illustrates the target CI/CD workflow.

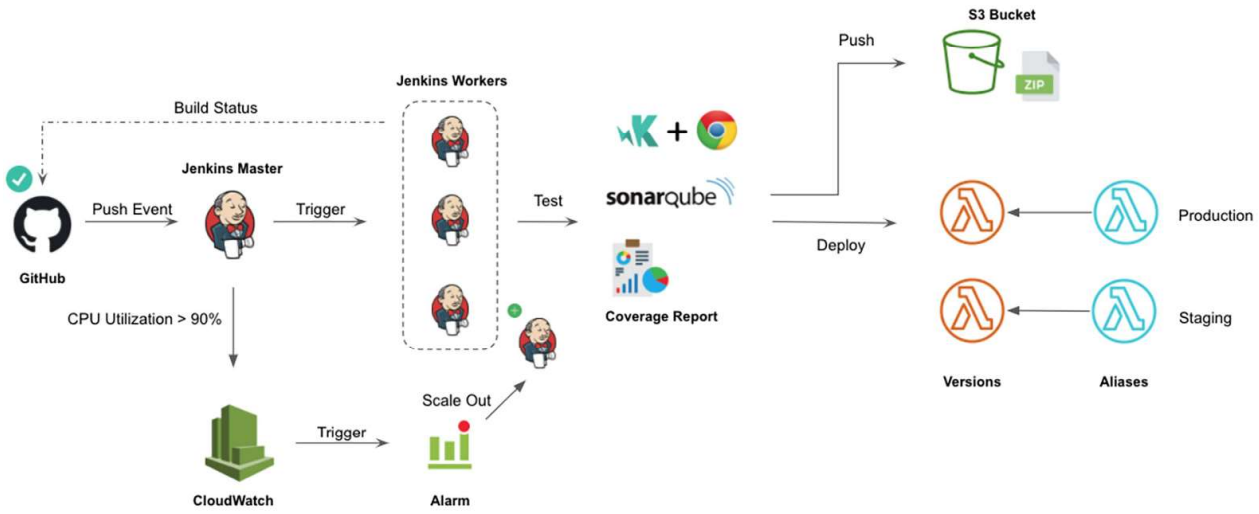

Figure 12.5 CI/CD workflow for a serverless application

A pipeline will be triggered whenever you make a change to your application's source code. The Jenkins master will schedule the build on one of the available Jenkins workers. The worker will execute the stages described in the Jenkinsfile located in the root directory of the application Git repository. The stages `Checkout` and `Tests` are given in chapter 8. The `Build` stage will compile the source code, install needed dependencies, and generate a deployment package (zip archive). Next, the `Push` stage will store the zip file in a remote S3 bucket and finally, the `Deploy` stage will be executed to update the Lambda function's code with the newest changes.

12.2 Creating deployment packages

Before integrating the serverless application in Jenkins, we need to store the Lambda functions' source code in a centralized remote repository for versioning. When it comes to serverless applications, two strategies are most used to organize functions into repositories:

- *Mono-repo*—Everything is put into the same repository; cohesive functions that work together to serve a business feature are grouped together under the same repository.
- *One repository per service*—Each Lambda function gets its own Git repository, with its own CI/CD pipeline.

This section doesn't go into the details around which is better, but instead shows how to build a CI/CD pipeline with the two approaches.

12.2.1 Mono-repo strategy

The MoviesLoader service, which consists of a single Lambda function written in Python, is responsible for loading a list of movies into a message queue. Create a GitHub repository, shown in figure 12.6, for the movies-loader Lambda function, and

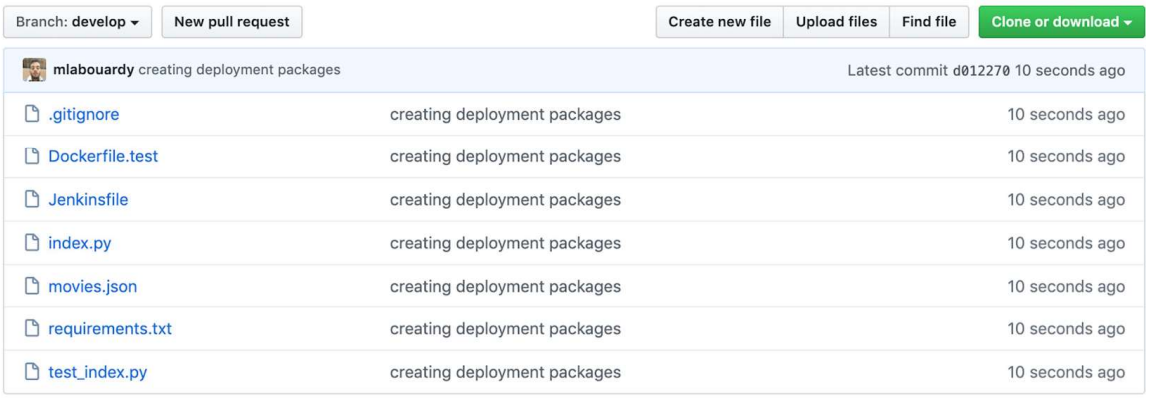

Figure 12.6 MoviesLoader Lambda function GitHub repository

then push the source code available in the book's repository (chapter12/functions) to the develop branch.

The Jenkinsfile (chapter12/functions/movies-loader/Jenkinsfile) is stored in the root repository. It's similar to the one provided in chapter 8's listing 8.3. Upon a push event, it will check out the function source code and run unit tests inside a Docker container. Having proper unit tests in place safeguards against subsequent Lambda code updates. This definition file, shown in the following listing, must be committed to the Lambda function's code repository.

Listing 12.4 Running function unit tests inside a Docker container

```
def imageName = 'mlabouardy/movies-loader'
node('workers'){
    try {
        stage('Checkout'){
            checkout scm
            notifySlack('STARTED')
        }
        stage('Unit Tests'){
            def imageTest= docker.build("${imageName}-test", "-f
     Dockerfile.test .")
            imageTest.inside{
                sh "python test_index.py"
            }
        }
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
    }
}
```

Sends a Slack notification when the build starts, by using the custom notifySlack method

When an error occurs, it's cached here, and the currentBuild.result variable is set to FAILED so the right Slack notification will be sent afterward.

When the pipeline is completed (success or failure), a Slack notification is sent to raise awareness about the pipeline status.

In listing 12.5, we create a deployment package, which is a zip file that includes both the Python code and any dependencies that the code needs to run. The `Build` stage generates a zip file and uses the Git commit ID as a name. Finally, we push the zip file to an S3 bucket for versioning and delete the file to save space.

Listing 12.5 Generating a deployment package

```
def functionName = 'MoviesLoader'
def imageName = 'mlabouardy/movies-loader'
def bucket = 'deployment-packages-watchlist'
def region = 'AWS REGION'

node('workers'){
    try {
        stage('Checkout'){...}

        stage('Unit Tests'){...}

        stage('Build'){
            sh "zip -r ${commitId}.zip index.py movies.json"
        }

        stage('Push'){
            sh "aws s3 cp ${commitId}.zip s3://${bucket}/${functionName}/"
        }
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
        sh "rm -rf ${commitId}.zip "
    }
}
```

The name of the S3 bucket where the deployment packages (zip files) are stored

Clones the Git repository. The instruction was omitted for brevity; see chapter12/functions/movies-loader/Jenkinsfile for the command.

Runs unit tests within a Docker container. See chapter12/functions/movies-loader/Jenkinsfile for instructions.

Creates an archive (zip file) with the function entrypoint (index.py) and the movies JSON array. The commitId function is used to create a unique ID for the archive based on the current Git commit ID.

Stores the archive to an S3 bucket

Deletes the archive at the end of the pipeline to save hard disk space

NOTE We use the Git commit ID as a name for the deployment package to give a meaningful and significant name for each release and be able to roll back to a specific commit if things go wrong.

On Jenkins, create a new multibranch pipeline job for the `MoviesLoader` lambda function (refer to chapter 7 for a step-by-step guide). Jenkins will discover the develop branch, and a new build will start; see figure 12.7.

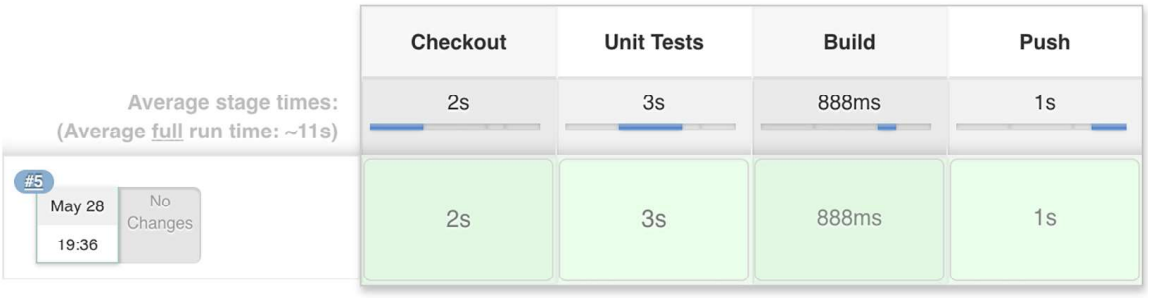

Figure 12.7 MoviesLoader Lambda function pipeline

You can drill down to see the steps on the UI that match our steps in the Jenkinsfile. While Jenkins is executing each stage, you can see the activity. You can see the tests running as part of the `Unit Tests` stage (figure 12.8). If tests are successful, a zip file will be generated and stored in an S3 bucket.

```
[Pipeline] sh
+ zip -r 5e8eb15d2de87a28572a4489e868b1b642e877b8.zip index.py movies.json
  adding: index.py (deflated 47%)
  adding: movies.json (deflated 79%)
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Push)
[Pipeline] sh
+ git rev-parse HEAD
[Pipeline] readFile
[Pipeline] sh
+ rm .git/commitID
[Pipeline] sh
+ aws s3 cp 5e8eb15d2de87a28572a4489e868b1b642e877b8.zip s3://deployment-packages-watchlist/MoviesLoader/
Completed 2.7 KiB/2.7 KiB (6.1 KiB/s) with 1 file(s) remaining
upload: ./5e8eb15d2de87a28572a4489e868b1b642e877b8.zip to s3://deployment-packages-
watchlist/MoviesLoader/5e8eb15d2de87a28572a4489e868b1b642e877b8.zip
```

Figure 12.8 Pipeline execution logs

Open the S3 console and click the bucket used by the pipeline for package storage. A new deployment package should be available with a key name identical to the Git commit ID, as shown in figure 12.9.

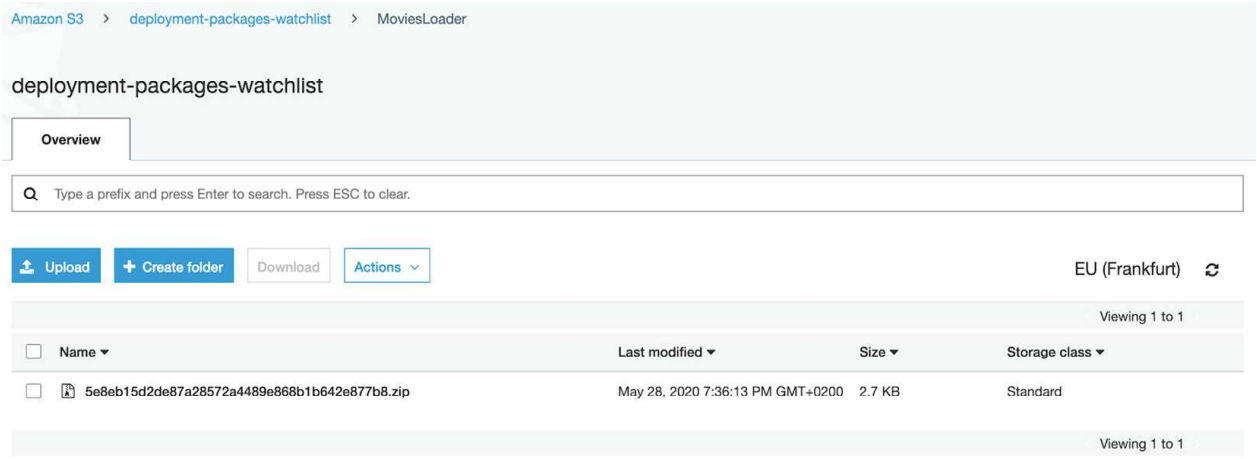

Figure 12.9 S3 bucket for deployment packages storage

Similarly for the movies-parser function, push the function source code to a dedicated GitHub repository, shown in figure 12.10.

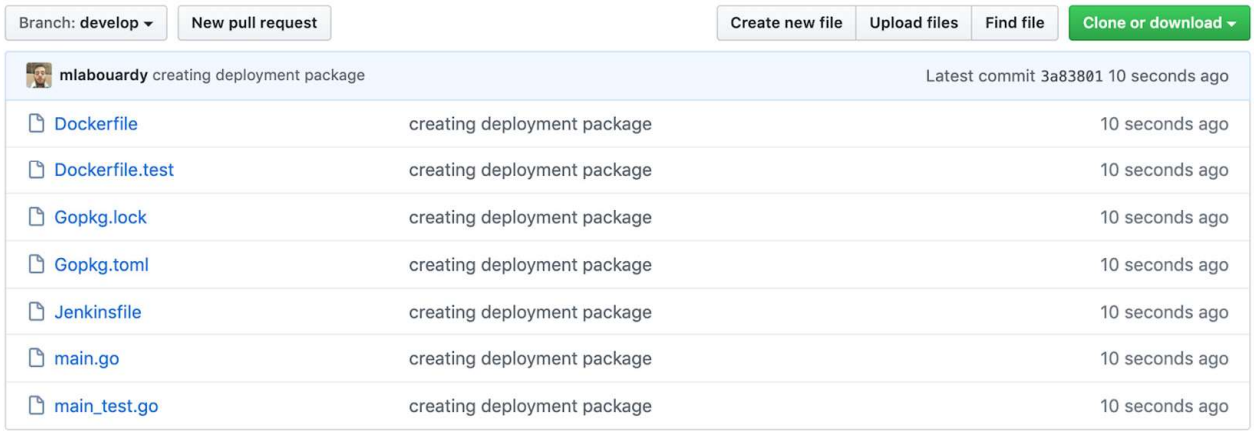

Figure 12.10 MoviesParser Lambda function GitHub repository

Create a Jenkinsfile (chapter12/functions/movies-parser/Jenkinsfile) with similar stages to chapter 8's listing 8.8 in the top-level directory of the Git repository; see the following listing.

Listing 12.6 Running function pre-integration tests in parallel

```
def imageName = 'mlabouardy/movies-parser'

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

        def imageTest= docker.build("${imageName}-test",
"-f Dockerfile.test .")
        stage('Pre-integration Tests'){
            parallel(
                'Quality Tests': {
                    imageTest.inside{
                        sh 'golint'
                    }
                },
                'Unit Tests': {
                    imageTest.inside{
                        sh 'go test'
                    }
                },
                'Security Tests': {
                    imageTest.inside('-u root:root'){
                       sh 'nancy /go/src/github/mlabouardy/
movies-parser/Gopkg.lock'
                    }
                }
            )
```

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

The function is written in Go, so we need to build a binary with the Docker multistage build feature, as explained in listing 9.2. Then, we copy the built binary from the Docker container and generate a zip package. Finally, we push the deployment package to the S3 bucket, as shown in the following listing.

Listing 12.7 Building a Go-based Lambda deployment package

```
def functionName = 'MoviesParser'
def imageName = 'mlabouardy/movies-parser'
def region = 'eu-west-3'

node('workers'){
    try{
     stage('Checkout'){...}                          Refer to listing 12.6
     stage('Pre-integration Tests'){...}             for the instructions.

     stage('Build'){
       sh """
        docker build -t ${imageName} .
        docker run --rm ${imageName}
        docker cp ${imageName}:/go/src/github.com/mlabouardy/
movies-parser/main main
        zip -r ${commitID()}.zip main
       """
      }

      stage('Push'){
        sh "aws s3 cp ${commitID()}.zip s3://${bucket}/${functionName}/"
      }
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
        sh "rm ${commitID()}.zip"
    }
}
```

Push the changes to the movies-parser Git repository, and create a new multibranch pipeline job for movies-parser. The pipeline stages should be executed. Upon completion, the pipeline should look like figure 12.11 in the Blue Ocean view.

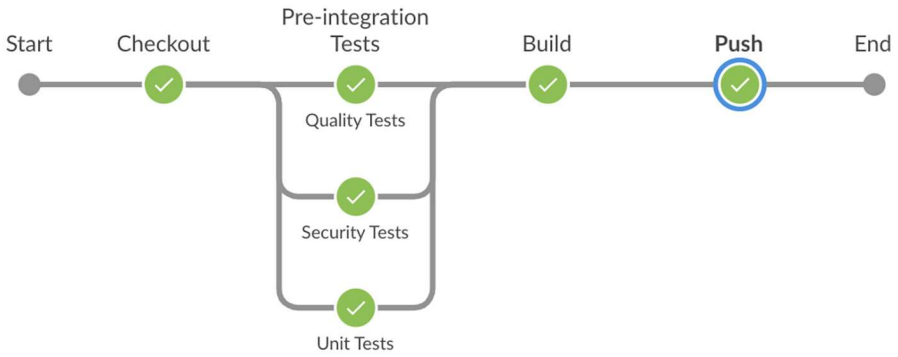

Figure 12.11 MoviesParser Lambda function workflow

Figure 12.12 shows the console output of the `Push` stage. The function deployment package will be stored under the MoviesParser subfolder.

Console Output

```
+ aws s3 cp 1b073885e88d279e5948dce042ebc364862252ac.zip s3://deployment-packages-watchlist/MoviesParser/
Completed 256.0 KiB/6.9 MiB (540.0 KiB/s) with 1 file(s) remaining
Completed 512.0 KiB/6.9 MiB (1.0 MiB/s) with 1 file(s) remaining
Completed 768.0 KiB/6.9 MiB (1.5 MiB/s) with 1 file(s) remaining
Completed 1.0 MiB/6.9 MiB (2.0 MiB/s) with 1 file(s) remaining
Completed 1.2 MiB/6.9 MiB (2.5 MiB/s) with 1 file(s) remaining
Completed 1.5 MiB/6.9 MiB (3.0 MiB/s) with 1 file(s) remaining
Completed 1.8 MiB/6.9 MiB (3.5 MiB/s) with 1 file(s) remaining
Completed 2.0 MiB/6.9 MiB (3.8 MiB/s) with 1 file(s) remaining
```

Figure 12.12 Publishing deployment package to S3

The obvious counterpart to the multi-repo pattern is the mono-repo approach. In this pattern, a single repository holds a collection of Lambda functions grouped by business capabilities.

12.2.2 Multi-repo strategy

The Movies Store API is split into multiple Lambda functions (`MoviesStoreListMovies`, `MoviesStoreSearchMovie`, `MoviesStoreViewFavorites`, `MoviesStoreAddToFavorites`). The easiest way to share code among these functions is by having them all together in a single repository. Create a new GitHub repository (chapter12/functions/movies-store), shown in figure 12.13.

The src/ folder at the root is made up of a collection of services. Each service deals with a relatively small and self-contained function. For instance, the movies/findAll folder is responsible for serving a list of movies from the DynamoDB table. The package.json file is located at the root of the repo. However, it is fairly common to have a separate package.json inside each service directory.

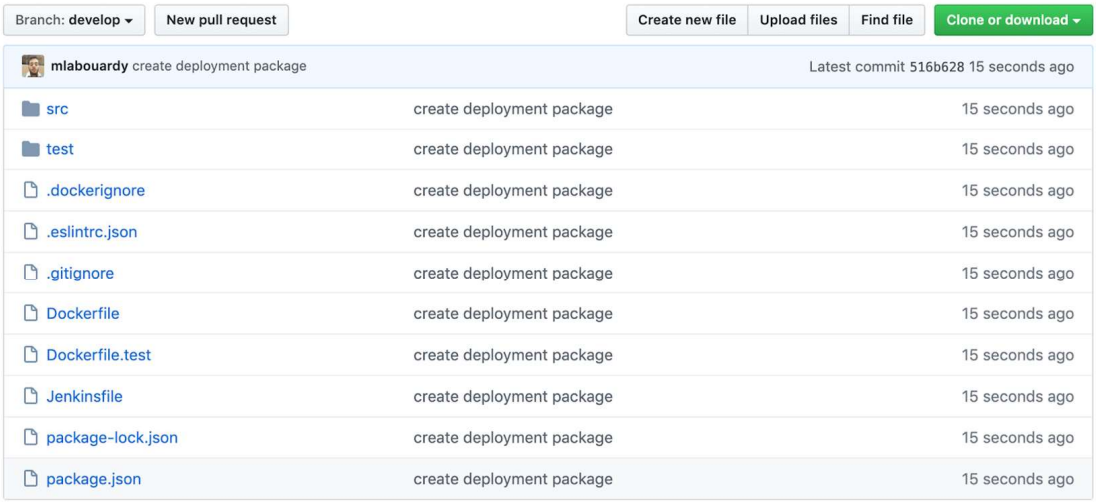

Figure 12.13 Multiple Lambda functions stored in single repository

On the movies-store repository, create a Jenkinsfile (chapter12/functions/movies-store/Jenkinsfile) by using your favorite text editor or IDE with the content in the following listing. Refer to listing 8.14 for more details about the implemented stages.

Listing 12.8 Running quality tests and generating code coverage reports

```
def imageName = 'mlabouardy/movies-store'
node('workers'){
    try {
        stage('Checkout'){
            checkout scm
            notifySlack('STARTED')
        }

        def imageTest= docker.build("${imageName}-test",
"-f Dockerfile.test .")

        stage('Tests'){
            parallel(
                'Quality Tests': {
                    sh "docker run --rm ${imageName}-test npm run lint"
                },
                'Unit Tests': {
                    sh "docker run --rm ${imageName}-test npm run test"
                },
                'Coverage Reports': {
                    sh "docker run --rm
-v $PWD/coverage:/app/coverage ${imageName}-test
npm run coverage"
                    publishHTML (target: [
                        allowMissing: false,
                        alwaysLinkToLastBuild: false,
                        keepAll: true,
                        reportDir: "$PWD/coverage",
                        reportFiles: "index.html",
```

```
                        reportName: "Coverage Report"
                    ])
                }
            )
        }
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
    }
}
```

Next, we run a Docker container from a Node.js base image to install external dependencies by running the `npm install` command. Then, we copy the node_modules folder from the running container to the current workspace and create a zip file, as shown in the next listing. The deployment package size will impact the functions' cold start. To keep the deployment package size smaller, we install only the runtime dependencies by passing `--prod=only` to the `npm install` command.

Listing 12.9 Building a Node.js-based Lambda deployment package

```
stage('Build'){
    sh """
      docker build -t ${imageName} .
      containerName=\$(docker run -d ${imageName})
      docker cp \$containerName:/app/node_modules node_modules
      docker rm -f \$containerName
      zip -r ${commitID()}.zip node_modules src
    """
}
```

NOTE One drawback of dynamic provisioning is a phenomenon called *cold start*. Essentially, functions that haven't been used for a while take longer to start up and to handle the first request.

Then, in the following listing, we push the generated zip file to an S3 bucket, use a loop to go through each function name, and save the zip in an S3 bucket under the function folder. You can use the Serverless framework (www.serverless.com) to create a zip file per function and exclude unused dependencies and files.

Listing 12.10 Publishing Node.js deployment packages to S3

```
def functions = ['MoviesStoreListMovies',
'MoviesStoreSearchMovie',
'MoviesStoreSearchMovie',
'MoviesStoreAddToFavorites']
def bucket = 'deployment-packages-watchlist'

node('workers'){
    try {
        stage('Checkout'){...}
        stage('Tests'){...}
```

```
        stage('Build'){...}

        stage('Push'){
            functions.each { function ->
                sh "aws s3 cp ${commitID()}.zip s3://${bucket}/${function}/"
            }
        }
    } catch(e){
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        notifySlack(currentBuild.result)
        sh "rm --rf ${commitID()}.zip"
    }
}
```

Head back to the Jenkins dashboard, create a new multibranch pipeline job for the movies-store project, and commit the changes to the develop branch. In a few seconds, a new build should be triggered on the movies-store job for the develop branch. On the resultant page, you will see the Stage view for the develop branch pipeline, shown in figure 12.14.

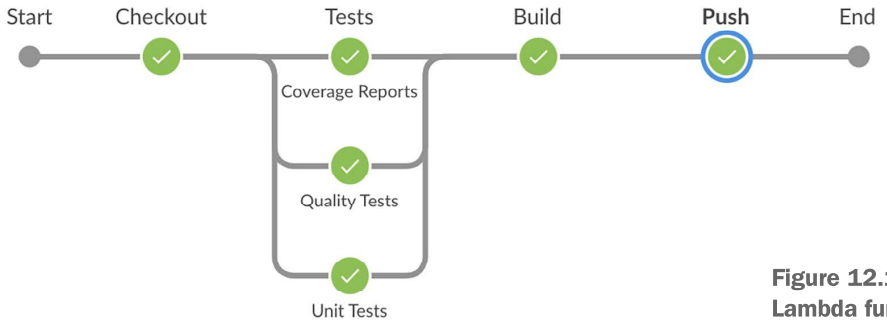

Figure 12.14 MoviesStore Lambda functions CI workflow

For common situations, the build and push stages can take a good amount of the CI/CD execution time. Therefore, we can use the `parallel` key, as shown in the following listing, to run the push stage in parallel, to keep the pipeline turnaround time short.

Listing 12.11 Parallel directive with a map structure

```
stage('Push'){
    def fileName = commitID()
    def parallelStagesMap = functions.collectEntries {
      ["${it}" : {
         stage("Lambda: ${it}") {
           sh "aws s3 cp ${fileName}.zip s3://${bucket}/${it}/"
         }
      }]
    }
    parallel parallelStagesMap
}
```

Sets the archive's name to the Git commit ID

Runs the stages in parallel

Parallel directive is expecting a map structure, so we're building one. We iterate over the functions list and create the corresponding command to store the archive file to the appropriate S3 folder.

The `parallel` directive takes a map of the string and closure. The string is the display name of the parallel execution (name of the function), and the closure is the actual `aws s3 cp` instruction to copy the deployment package to the corresponding function folder in S3. As a result, storing the deployment packages for each function will be run in parallel, as shown in figure 12.15.

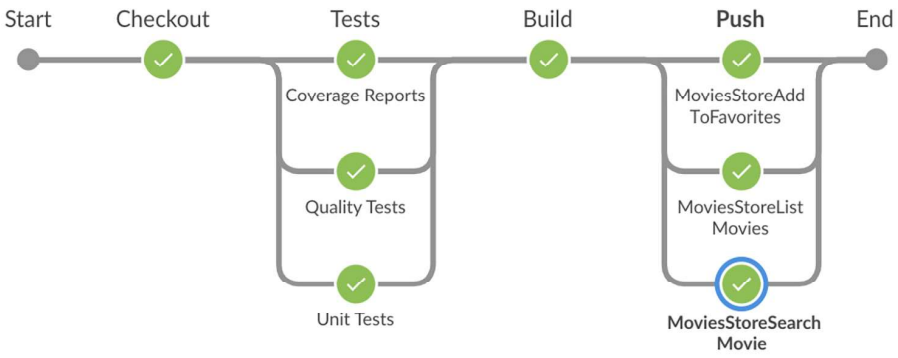

Figure 12.15 MoviesStore CI workflow

Once the pipeline execution is completed, in the S3 bucket, a deployment package should be stored for each Lambda function, as shown in figure 12.16.

Figure 12.16 Lambda functions deployment packages

By now, the deployment packages are stored in an S3 bucket, so we can go ahead and update the Lambda function source code with the built zip files.

12.3 Updating Lambda function code

For `MoviesLoader` and `MoviesParser` Lambda functions, add the following `Deploy` stage to their Jenkinsfiles (chapter12/functions/movies-loader/Jenkinsfile and chapter12/functions/movies-parser/Jenkinsfile). The stage uses the AWS Lambda CLI to issue an `update-function-code` command to update the function code with the zip file stored previously in the S3 bucket; see the following listing.

Listing 12.12 Updating the Lambda function's code with AWS CLI

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

The command takes as an argument the name of the S3 bucket where the zip file is stored as well as the Amazon S3 key of the deployment package.

Once you push the changes to the Git remote repository, Jenkins will update the Lambda function's code with the `update-function-code` command. The output in figure 12.17 confirms that.

Console Output

```
+ aws lambda update-function-code --function-name MoviesLoader --s3-bucket deployment-packages-watchlist --s3-key
MoviesLoader/5e8eb15d2de87a28572a4489e868b1b642e877b8.zip --region eu-west-3
{
    "FunctionName": "MoviesLoader",
    "LastModified": "2020-05-29T15:43:27.094+0000",
    "RevisionId": "46414431-f7c6-4199-9ade-80a385ccbe3d",
    "MemorySize": 128,
    "Environment": {
        "Variables": {
            "SQS_URL": "https://sqs.eu-west-3.amazonaws.com/305929695733/movies_to_parse"
        }
    },
    "Version": "$LATEST",
    "Role": "arn:aws:iam::305929695733:role/MoviesLoaderRole",
    "Timeout": 10,
    "Runtime": "python3.7",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "udtoEDeFVkn73aQr9S9kmoBgqbyQzLB7pA+RCeyMmK0=",
    "Description": "",
    "CodeSize": 2725,
    "FunctionArn": "arn:aws:lambda:eu-west-3:305929695733:function:MoviesLoader",
    "Handler": "index.handler"
}
```

Figure 12.17 `UpdateFunction-Code` operation logs

The CI/CD pipelines for the `MoviesLoader` and `MoviesParser` functions should contain the stages shown in figure 12.18.

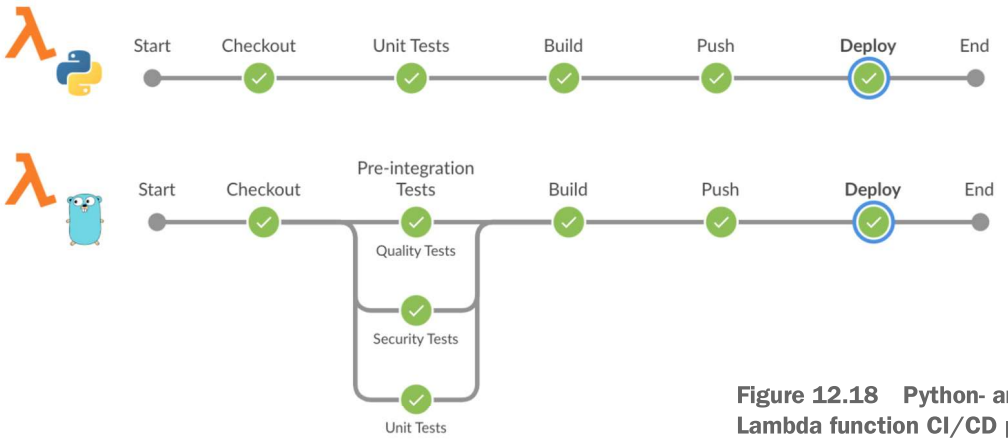

Figure 12.18 Python- and Go-based Lambda function CI/CD pipelines

NOTE The Serverless framework (https://serverless.com/) or AWS Serverless Application Model (SAM) can also be used to write and deploy Lambda functions within Jenkins pipelines.

Similarly, add the same stage to the `MoviesStore` Lambda functions—except this time, we will wrap the `update-function-code` command with a `for` loop to update each function versioning within the same GitHub repository; see the following listing.

Listing 12.13 Updating multiple Lambda functions

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

When the new stage is committed, the pipeline will be triggered upon a push event, and the CI/CD stages in figure 12.19 will be executed.

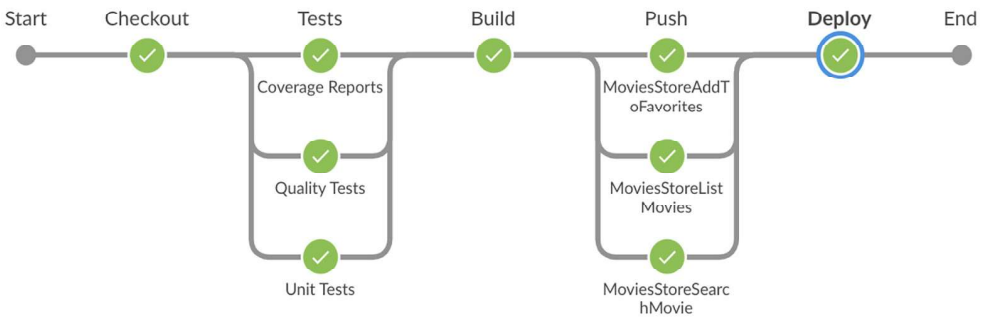

Figure 12.19 MoviesStore CI/CD pipeline

Before we automate the deployment of the marketplace, we need to load some data into the DynamoDB table. Trigger the `MoviesLoader` function from AWS Management Console, or by issuing the following command from your terminal session:

```
aws lambda invoke --function-name MoviesLoader --payload '{}' response.json
```

NOTE Make sure to assign the `AWSLambda_FullAccess` policy to the IAM user configured with your AWS CLI.

The preceding command will invoke the `MoviesLoader` function and save the function's output in the response.json file. The function will load movies to SQS and trigger the `MoviesParser` Lambda function, which will crawl the movie's IMDb page and store its information in the Movies DynamoDB table, shown in figure 12.20.

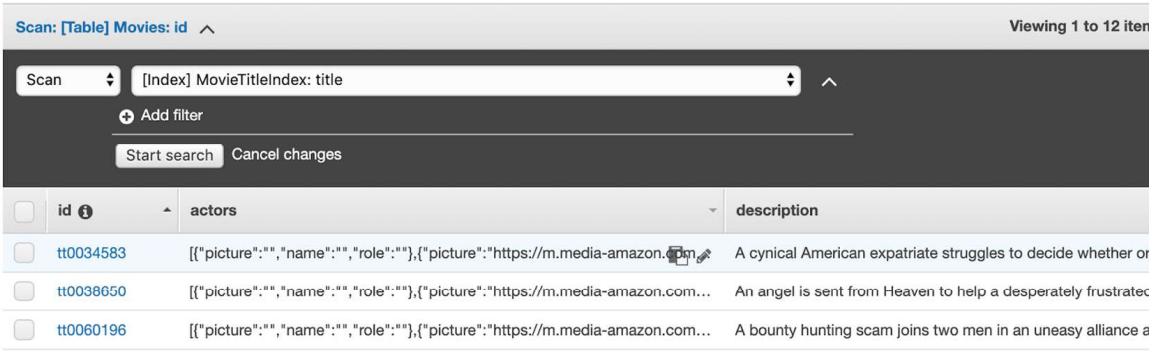

Figure 12.20 Movies DynamoDB table

Each message in SQS will invoke the `MoviesParser` function; once the queue is empty, the DynamoDB table should contain the top 100 IMDb movies.

12.4 Hosting a static website on S3

The Movie Marketplace is a single-page application (SPA), written in TypeScript, using the Angular framework. The application serves static content (HTML, JavaScript, and CSS files), which can be a good fit for S3 website-hosting features.

Let's automate the deployment of the marketplace to an S3 bucket, as shown in the next listing. First, create a GitHub project to version the marketplace source code. Then, write a Jenkinsfile to run quality, unit tests, and static code analysis with SonarQube. Refer to chapter 8 for more details.

Listing 12.14 Integrating an Angular application with the Jenkinsfile

```
def imageName = 'mlabouardy/movies-marketplace'
def region = 'AWS REGION'

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

        def imageTest= docker.build("${imageName}-test",
"-f Dockerfile.test .")
        stage('Quality Tests'){
            sh "docker run --rm ${imageName}-test npm run lint"
        }
        stage('Unit Tests'){
            sh "docker run --rm
-v $PWD/coverage:/app/coverage
${imageName}-test npm run test"
```

Builds a Docker image based on Dockerfile.test to run automated tests

Runs the code linting process

Runs unit tests and generates a coverage report