Pipeline as Code

Continuous delivery with Jenkins, Kubernetes, and Terraform

MOHAMED LABOUARDY

MANNING
SHELTER ISLAND

Part 3

Hands-on CI/CD pipelines

You've smashed through parts 1 and 2 but you're still hungry for more. I understand. Thankfully, this part is designed to give you a lot to chew on.

You'll implement CI/CD workflows for real-world, cloud-native applications. In the next few chapters, you'll run automated tests with Docker, analyze your Docker images for security vulnerabilities, and deploy containerized microservices on Docker Swarm and Kubernetes. You'll learn how to automate the deployment process for your serverless applications. This is just a tiny glimpse, so roll up your sleeves and let's dive into this!

7 Defining a pipeline as code for microservices

This chapter covers

- Using a Jenkins multibranch pipeline plugin and GitFlow model
- Defining multibranch pipelines for containerized microservices
- Triggering a Jenkins job on push events using GitHub webhooks
- Exporting Jenkins jobs configuration as XML and cloning Jenkins jobs

The previous chapters covered how to deploy a Jenkins cluster on multiple cloud providers by using automation tools: HashiCorp Packer and Terraform. In this chapter, we will define a continuous integration (CI) pipeline for Dockerized microservices.

In chapter 1, you learned that CI is continuously testing and building all changes of the source code before integrating them into the central repository. Figure 7.1 summarizes the stages in this workflow.

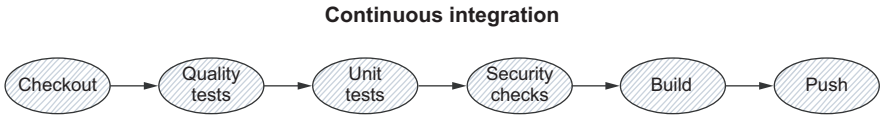


Figure 7.1 Continuous integration stages

Every change to the source code triggers the CI pipeline, which launches the automated tests. This comes with many benefits:

- Detecting bugs and issues earlier, which results in a dramatic decrease in maintenance time and costs
- Ensuring that the codebase continues to work and meets the spec requirements as the system grows
- Improving team velocity by establishing a fast-feedback loop

While automated tests come with multiple benefits, they're extremely time-consuming to implement and execute. Therefore, we will use a testing framework based on the target service runtime and requirements.

Once tests are successful, the source code is compiled and an artifact is built. Then it will be packaged and stored in a remote registry for version control and deployment later.

Chapter 8 covers how to write a classic CI pipeline for containerized microservices. The end result will look like the CI pipeline in figure 7.2.

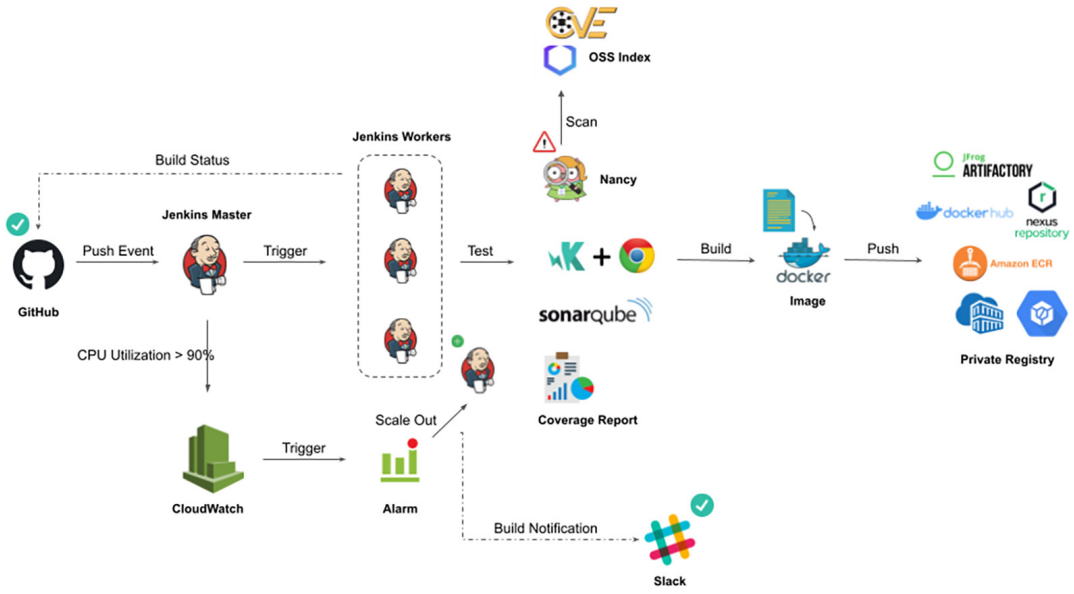


Figure 7.2 Target CI pipeline

These steps cover the most basic flow of a continuous integration process. In the following chapters, once you are comfortable with this workflow, we'll go even further. We'll start by creating our multibranch pipeline from scratch with Jenkins and continuously running pipelines with GitHub webhooks.

7.1 Introducing microservices-based applications

It can be challenging to create a reliable CI/CD process for a microservices architecture. The goal of the pipeline is to allow teams to build and deploy their services quickly and independently, without disrupting other teams or destabilizing the application as a whole.

To illustrate how to define a CI/CD pipeline from scratch for containerized microservices, I have implemented a simple web application based on a microservices architecture. We are going to integrate and deploy a web-based application called Watchlist, where users can browse the top 100 greatest movies of all time and add them to their watching list.

The project includes tests, benchmarks, and everything needed to run the application locally and on the cloud. The deployed application will look like figure 7.3.

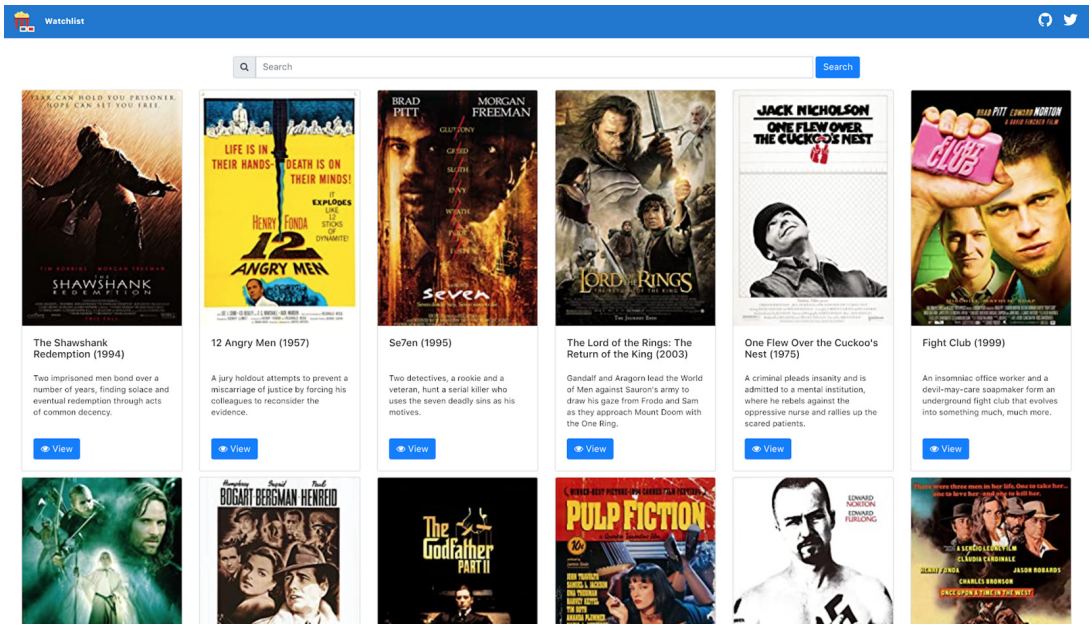


Figure 7.3 Watchlist marketplace UI

Figure 7.4 illustrates the application architecture and flow.

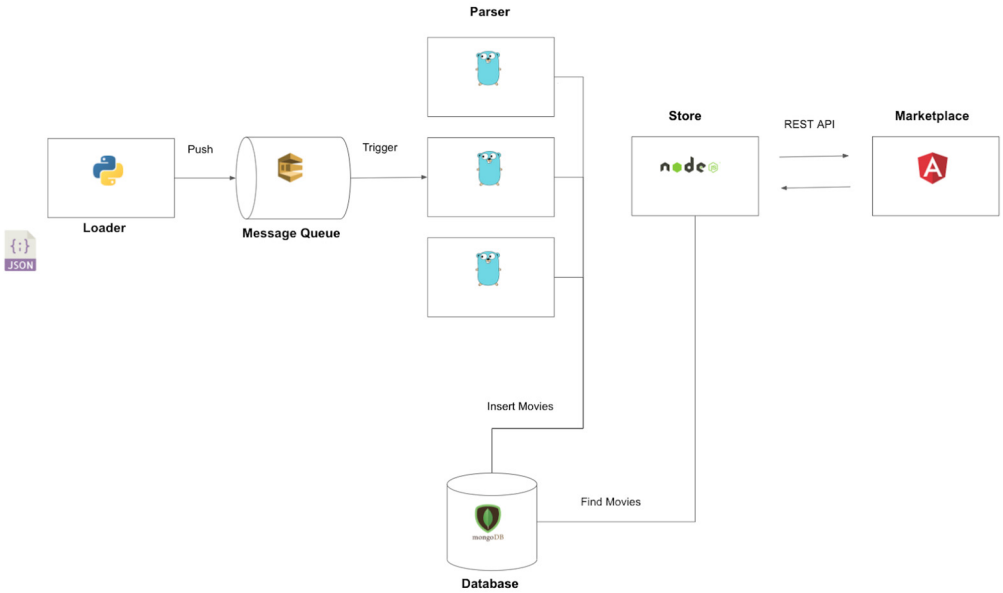


Figure 7.4 The Loader service takes an array of movies in JSON format and forwards them one by one to a message queue (for example, Amazon SQS). From there, a Parser service will consume the items and fetch the movie's details from the IMDb database and save the result into MongoDB. Finally, the data is served through a RESTful API by the Store service and visualized with the Marketplace UI.

> **NOTE** *Amazon Simple Queue Service* (SQS) is a distributed message queuing service. It is intended to provide a highly scalable managed message queue to resolve issues arising from producer-consumer problems and to decouple distributed application services. See https://aws.amazon.com/sqs/ for more details.

The architecture is composed of multiple services written in different languages to illustrate the advantages of the microservices paradigm and the use of Jenkins to automate the build and deployment process of different runtime environments. Table 7.1 lists the microservices.

Table 7.1 Application microservices

Service	Language	Description
Loader	Python	Responsible for reading a JSON file containing a list of movies and pushing each movie item to Amazon SQS.
Parser	Golang	Responsible for consuming movies by subscribing to SQS and scraping movie information from the IMDb website (www.imdb.com) and storing the metadata (movie's name, cover, description, and so forth) into MongoDB.

Table 7.1 Application microservices *(continued)*

Service	Language	Description
Store	Node.js	Responsible for serving a RESTful API with endpoints to fetch a list of movies and insert new movies into the watch list database in the MongoDB server.
Marketplace	Angular and TypeScript	Responsible for serving a frontend to browse movies by calling the Store RESTful API.

Before we dig deeper into the CI workflow for the application, let's see how the distributed application source code will be organized. When you start moving to microservices, one of the big challenges you will be facing is the organization of the codebase.

Do you create a repository for each service or a single repo for all services? Each pattern has its own advantages and disadvantages:

- *Multiple repositories*—You can have multiple teams independently developing a service (clear ownership). Plus, smaller codebases are easier to maintain, test, and deploy with less team coordination. However, having independent teams might create localized knowledge across the organization and result in teams lacking an understanding of the bigger picture of the project.
- *Mono repository*—Having a single source-control repository comes with a simplified project organization with less overhead from managing project dependencies. It also improves the overall work culture when teams work on a mono repository. However, versioning might become more complicated, and performance and scalability issues may arise.

Both patterns have pros and cons, and neither is a silver bullet. You should understand their benefits and limitations, and use them to make an informed decision on what's best for you and your project.

The way you structure your codebase will impact the design of the CI/CD pipeline. Having a project hosted on a single repository might result in a single pipeline with fairly complex stages. Pipeline size and complexity are often a huge pain point. As the number of services evolves within an organization, the management of pipelines becomes a bigger issue as well. In the end, most pipelines end as a spaghetti mix of npm, pip, and Maven scripts sprinkled with some bash scripts all over the place. On the other side, adopting a multiple-repositories strategy might result in multiple pipelines to manage and code duplication. Fortunately, solutions are available to reduce pipeline management, including using shared pipeline segments and shared Groovy scripts.

NOTE Chapter 14 covers how to write a shared library in Jenkins to share common code and steps across multiple pipelines.

This book illustrates how to build CI/CD pipelines for both patterns. For microservices, we will adopt the multiple repositories strategy. We will cover the mono-repo approach while building CI/CD pipelines for serverless functions.

First, create four Git repositories to store the source code for each service (Loader, Parser, Store, and Marketplace). In this book, I'm using GitHub, but any SCM system can be used, such as GitLab, Bitbucket, or even SVN. Make sure you have Git installed on the machine that you will use to perform the steps mentioned in the following section.

> **NOTE** Throughout this book, we will use the GitFlow model for branch management. For more information, read chapter 2.

Once the repositories are created, clone them to your workspace and create three main branches: develop, preprod, and master branches to help organize the code and isolate the under-development code from the one running in production. This branching strategy is a slimmer version of the GitFlow workflow branching model.

> **NOTE** The complete Jenkinsfile for each service can be found in the chapter7/microservices folder within the book's GitHub repository.

Use the following commands to create the target branches and push them to the remote repository:

```
git clone https://github.com/mlabouardy/movies-loader.git
cd movies-loader
git checkout -b preprod
git push origin preprod
git checkout -b develop
git push origin develop
```

To view the branches in the Git repository, run this command in your terminal:

```
git branch -a
```

An asterisk (*) will be next to the branch that you're currently on (develop). Output similar to the following should be displayed in your terminal session:

```
jenkins:movies-loader mlabouardy$ git branch -a
* develop
  preprod
  remotes/origin/develop
  remotes/origin/master
  remotes/origin/preprod
jenkins:movies-loader mlabouardy$
```

Next, copy the code from the book's GitHub repository to each Git repository on the develop branch, and then push the changes to the remote repository:

```
git add .
git commit -m "loading from json file"
git push origin develop
```

The GitHub repository should look like figure 7.5.

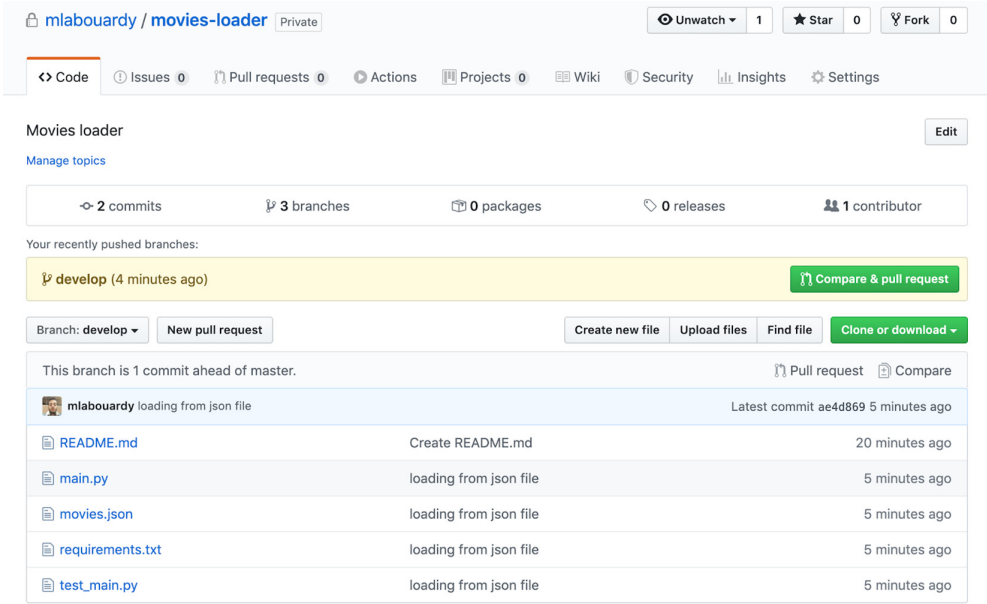


Figure 7.5 The Loader GitHub repository has the service source's code.

NOTE For now, we push the changes directly to the develop branch. Later, you will see how to create pull requests and set up a review process with Jenkins.

The movies-loader source code is available in the chapter7/microservices/movies-loader folder. Repeat the same process to create the movies-parser, movies-store, and movies-marketplace GitHub repositories.

7.2 Defining multibranch pipeline jobs

To integrate the application source code with Jenkins, we need to create Jenkins jobs to continuously build it. Head over to Jenkins web dashboard and click the New Item button at the top-left corner, or click the Create New Jobs link to create a new job, as shown in figure 7.6.

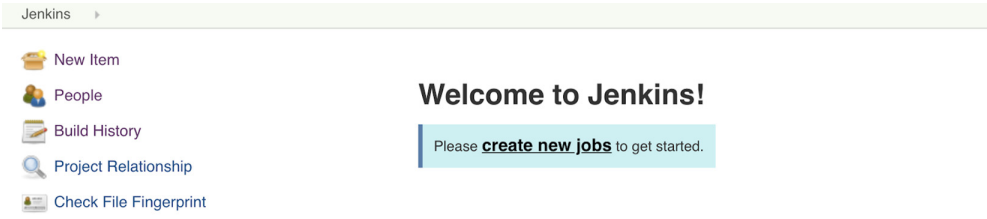


Figure 7.6 Jenkins new job creation

NOTE For a step-by-step guide on deploying Jenkins, refer to chapter 5.

On the resultant page, you will be presented with various types of Jenkins jobs to choose from. Enter the name of the project, scroll down, select Multibranch Pipeline, and click the OK button. The Multibranch Pipeline option allows us to automatically create a pipeline for each branch on the source-control repository.

Figure 7.7 shows the multibranch job pipeline for the movies-loader service.

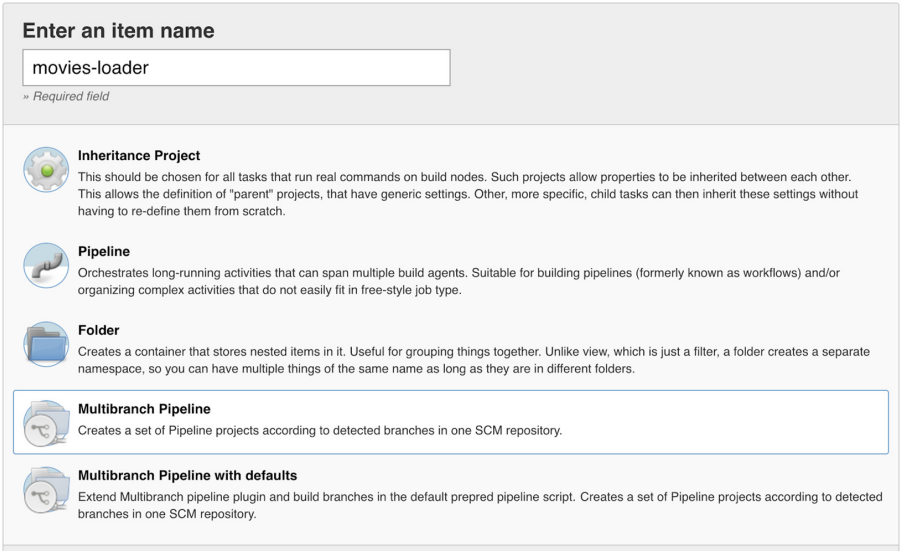


Figure 7.7 Jenkins new job settings

NOTE The Jenkins Multibranch Pipeline plugin (https://plugins.jenkins.io/workflow-multibranch/) is installed by default on the baked Jenkins master AMI.

I'll briefly summarize the new job types here and then explain each in more detail in upcoming chapters:

- *Freestyle project*—This is a classic way of creating a Jenkins job, wherein each CI stage is represented by using UI components and forms. The job is a web-based configuration, and any modification is done through the Jenkins dashboard.
- *Inheritance project*—The purpose of this project type is to bring true inheritance of properties between multiple job definitions to Jenkins. It allows you to share common properties only once and create Jenkins jobs to inherit them across many projects.
- *Pipeline*—This job type lets you either paste a Jenkinsfile directly into the job UI or reference a single Git repository as the source and then specify a single branch where the Jenkinsfile is located. This job can be useful if you plan to use a trunk-based workflow to manage your project source code.

- *Folder*—This is a way to group multiple projects together rather than a type of project itself. This is different from the view tabs on the Jenkins dashboard, which provide just a filter. Rather, this is like a directory folder on the server, storing nested items.
- *Multibranch pipeline*—This is a type of project we will use through this book. As its name indicates, it allows us to automatically create nested jobs for each Git branch containing a Jenkinsfile.
- *Organization*—Certain source-control platforms provide a mechanism for grouping multiple repositories into organizations. This project type allows you to use a Jenkinsfile in the repositories within an organization and execute a pipeline based on the Jenkinsfile. Currently, the project type supports only GitHub and Bitbucket organizations.

NOTE The trunk-based strategy uses one central repository with a single entry (called a *trunk* or *master*) for all changes to the project.

To be clear, having these new job types available depends on having the requisite plugins installed. If you baked the Jenkins master machine image with the list of plugins provided in chapter 4's section 4.3.2, you will get all the job types discussed in the preceding list.

7.3 Git and GitHub integration

The pipeline script (Jenkinsfile) will be versioned in GitHub. Therefore, we need to configure the Jenkins job to fetch it from the remote repository.

Set a name and description in the General section. Then, select the code source from the Branch Sources section. Configure the pipeline to refer to GitHub for source-control management by selecting GitHub from the drop-down list; see figure 7.8.

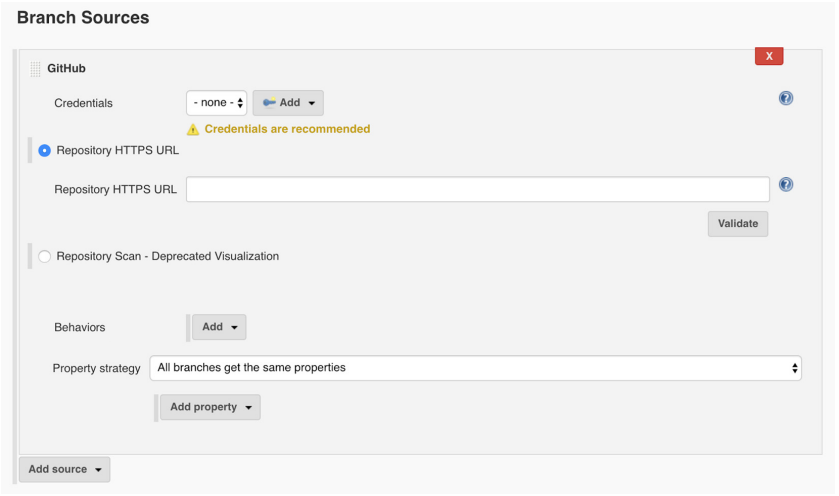


Figure 7.8 Branch Sources configuration

For checkout credentials, open a new tab and go to the Jenkins dashboard. Click Credentials and then System. On the Global Credentials page, from the menu on the left, click the Add Credentials link. Next, create a new Jenkins global credential of type Username and Password to access the microservices projects in Git. The GitHub username and password can be set as shown in figure 7.9. However, it's not recommended to use a personal GitHub account.

NOTE The Jenkins Credentials plugin (https://plugins.jenkins.io/credentials/) is installed by default on the baked Jenkins master machine image. It is part of the essential plugins listed in chapter 4's section 4.3.2.

Figure 7.9 Jenkins credentials provider

Therefore, I have created a dedicated Jenkins service account on GitHub and used an access token instead of the account password. You can create the access token by signing in with the GitHub credentials and navigating to Settings. Then, from the left menu, select Developer Settings and select Personal Access Tokens, as shown in figure 7.10.

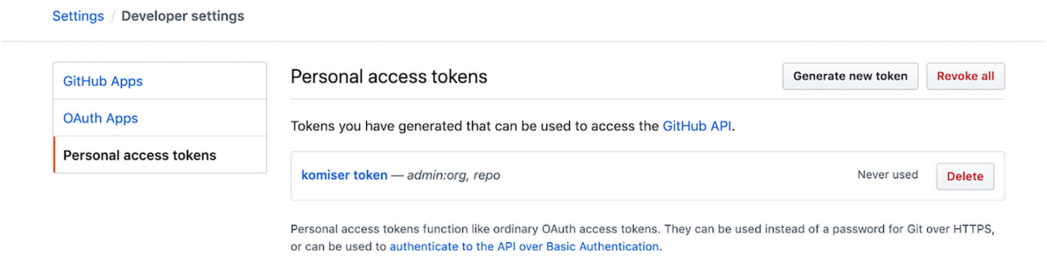


Figure 7.10 GitHub personal access tokens

Click the Generate New Token button, give a name to the access token, and select the `repo` access from the list of authorized scopes, as shown in figure 7.11. For private repositories, you must ensure that the `repo` scope is selected, and not just the `repo:status` and `public_repo` scopes. The token name is helpful, as you'll likely have many of these tokens for many applications.

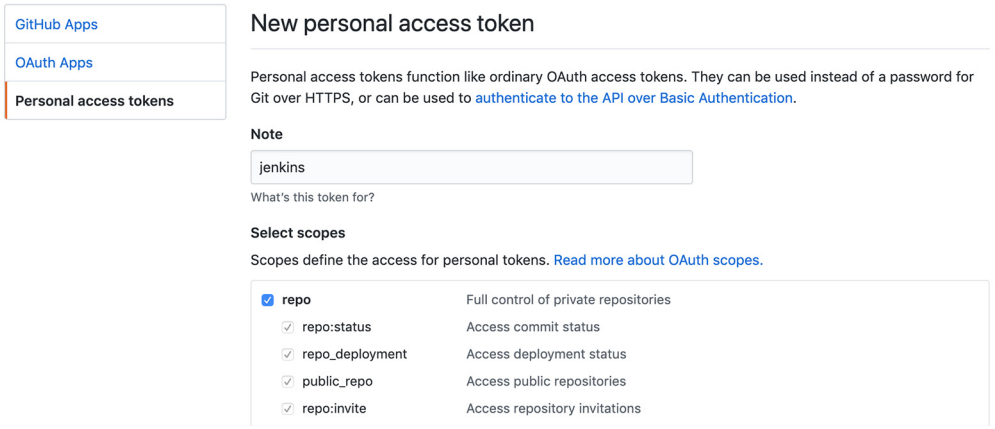


Figure 7.11 Jenkins dedicated token for GitHub access

As the GitHub warning in figure 7.12 indicates, you must copy the token after you generate it, as you won't be able to see it again. If you fail to do so, your only recourse will be to regenerate the token.

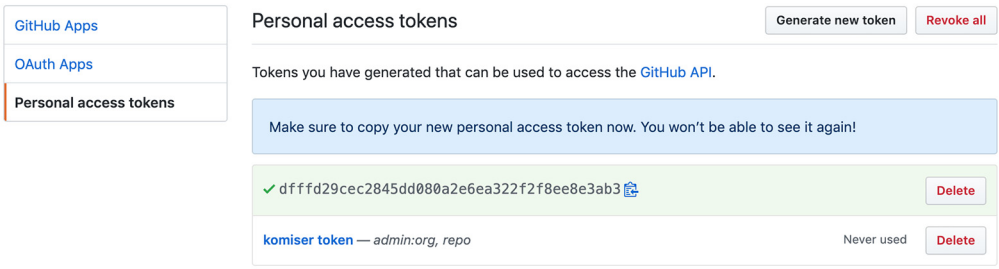


Figure 7.12 Jenkins personal access token

Paste in the GitHub personal access token to the Password field. Give a unique ID to your GitHub credentials by typing a string in the ID field and add a meaningful description to the Description field, as shown in figure 7.13. Then click the Save button.

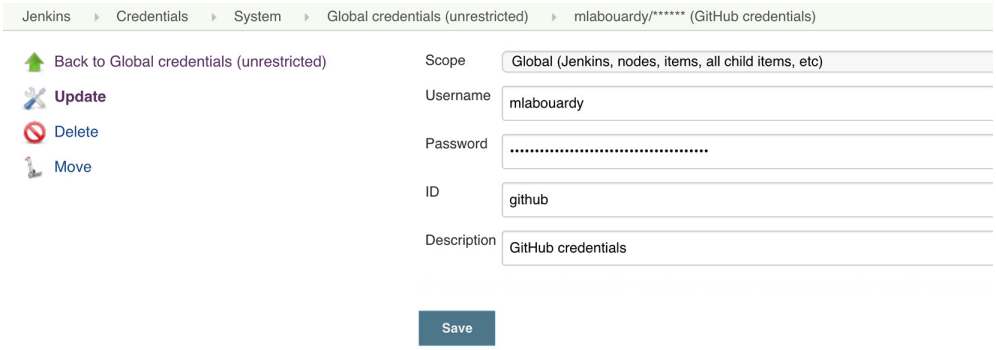


Figure 7.13 GitHub credentials configuration on Jenkins

Go back to the job configuration tab, shown in figure 7.14, and select the credentials you created from the Credentials drop-down list. Set the repository HTTPS clone URL and set the discovering behavior to allow scanning of all repository branches. Then, scroll all the way down and click the Apply and Save buttons.

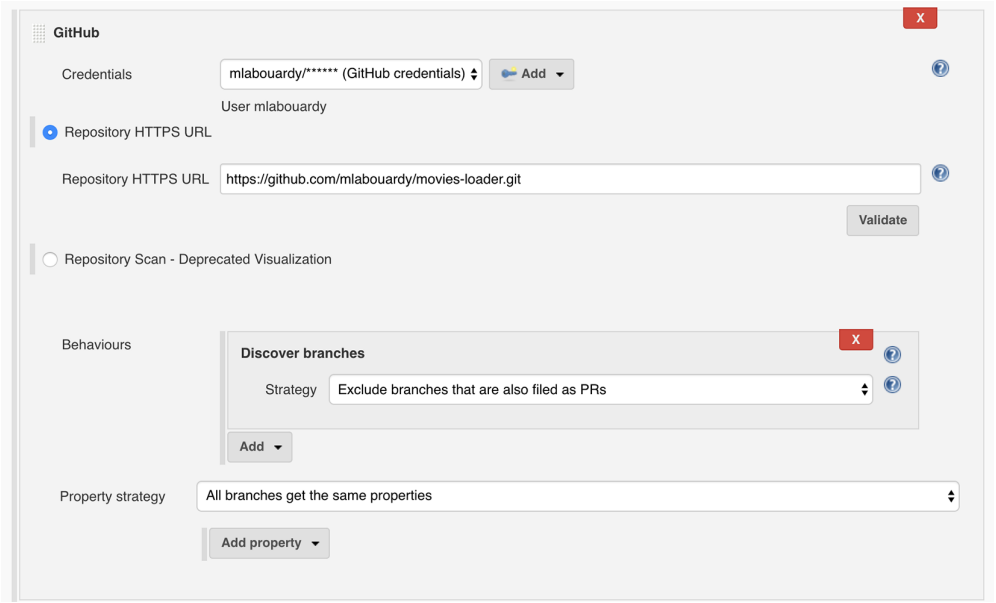


Figure 7.14 GitHub repository configuration on Jenkins

NOTE We cover Jenkins advanced scanning behaviors and strategies in chapter 9.

Jenkins will scan the GitHub repository, looking for branches with a Jenkinsfile in the root repository. So far, there are none, and we can check that by clicking the Scan Repository Log button from the left sidebar.

> **NOTE** In this book, we will use the concept of pipeline as code instead of representing each CI stage within the UI as in a Jenkins classic freestyle job. The pipeline will be described in a Jenkinsfile.

The log output confirms that no Jenkinsfile has been found yet in the GitHub repository, as shown in figure 7.15.

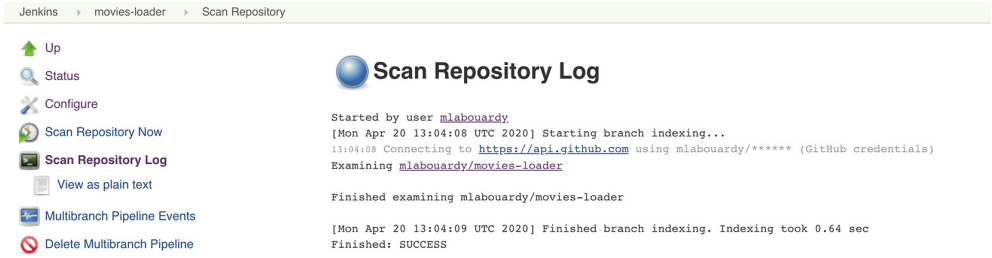


Figure 7.15 Jenkins repository scanning logs

It's time to create a Jenkinsfile. Using your favorite text editor or IDE, create and save a new text file with the name `Jenkinsfile` at the root of your local movies-loader Git repository. Copy the following scripted pipeline code and paste it into your empty Jenkinsfile.

Listing 7.1 Jenkinsfile using a scripted approach

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

> **NOTE** We are using scripted pipeline syntax to write most of the Jenkinsfile. However, the declarative approach will be given when the CI pipeline is completed.

The `Checkout` stage, as its name indicates, will simply check out the code at the reference point that triggered the run. You can customize the checkout process by providing additional parameters. Also, the stages will be executed on Jenkins workers—hence, the use of the `workers` label on the node block. We're assuming we have a Jenkins worker already set up on the Jenkins instance labeled `workers`. If no label is provided, Jenkins will run the pipeline on the first executor that becomes available on any machine (master or worker).

Save your edited Jenkinsfile and push the changes to the develop branch by running the following commands:

```
git add Jenkinsfile
git commit -m "creating Jenkinsfile"
git push origin develop
```

The Jenkinsfile lives with the source code in GitHub. Therefore, like any code, it can be peer-reviewed, commented on, and approved before being merged into main branches; see figure 7.16.

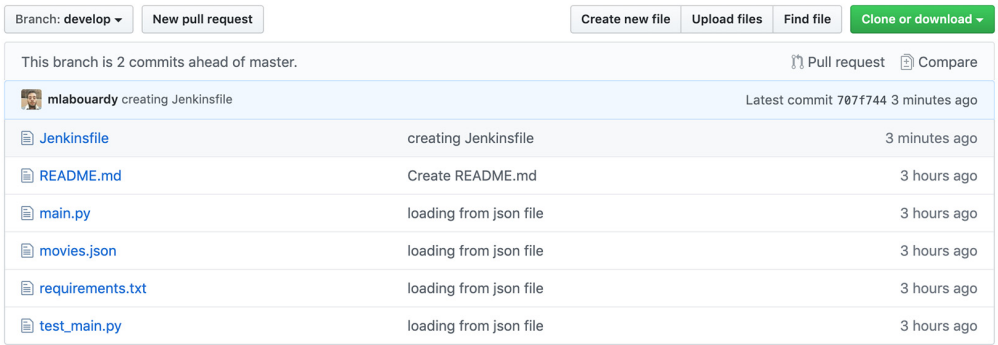


Figure 7.16 Jenkinsfile is stored along with source code

Go back to the Jenkins dashboard, and to trigger the scanning again, click the Scan Repository Now button. By default, this will automatically trigger builds for all newly discovered branches, as shown in figure 7.17.

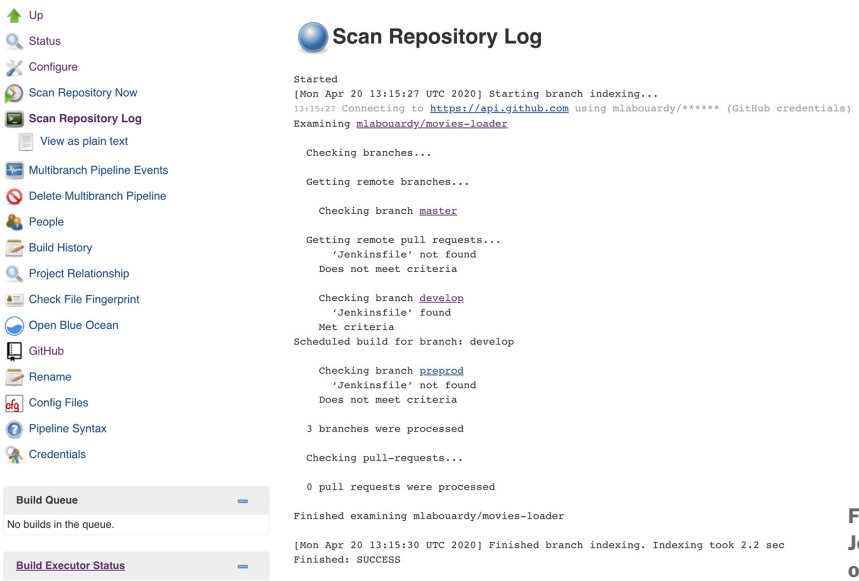


Figure 7.17 Jenkinsfile detected on develop branch

In our current setup, a Jenkinsfile has been found only on the develop branch. If we click the movies-loader job again. Jenkins should have created a nested job for the develop branch, as you can see in figure 7.18. There was no pipeline scheduled for the preprod and master branches since there was no Jenkinsfile on them yet.

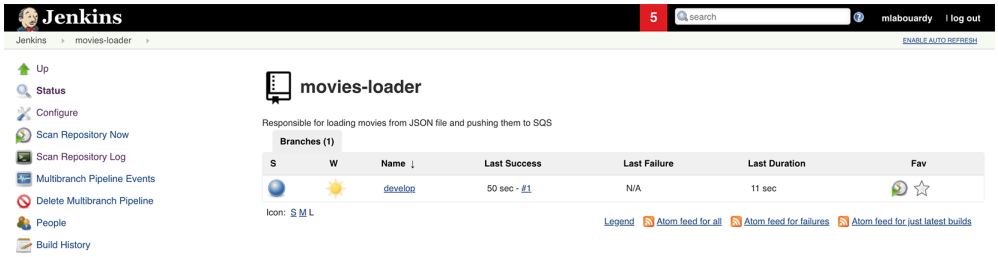


Figure 7.18 Build job triggered on the develop branch

NOTE If you ever have problems with jobs for branches not being created or built automatically, check the Scan Repository Log item from the left job sidebar.

The build should be triggered on the develop branch automatically, and the checkout stage will be executed and turned green. Note that the Git client should be installed on the worker where the build is executed.

The Jenkins Stage view, shown in figure 7.19, lets us visualize the progress of various stages of the pipeline in real-time.

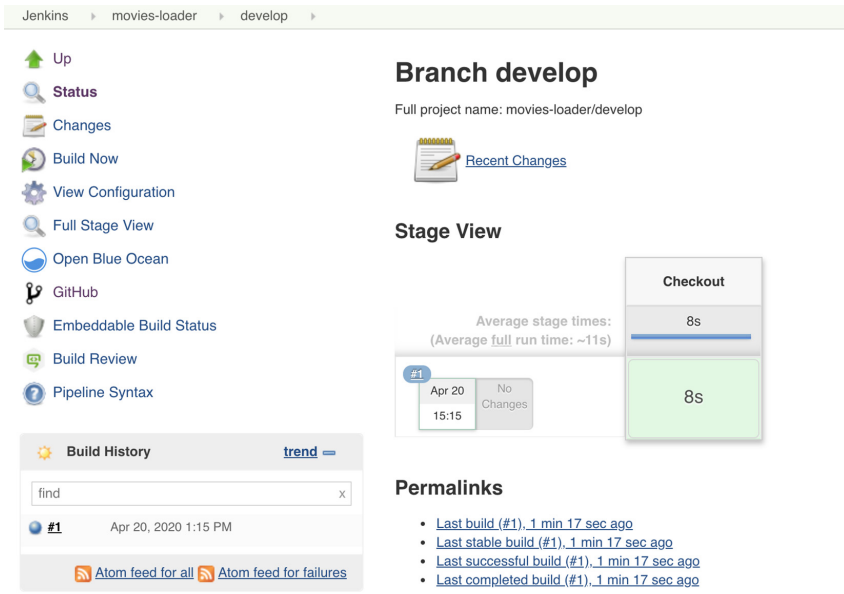


Figure 7.19 Pipeline execution

NOTE The Jenkins Stage view is a new feature that comes as a part of release 2.*x*. It works only with Jenkins Pipeline and Jenkins Multibranch pipeline jobs.

Click the Checkout stage column to view the stage's logs. You can see that Jenkins has cloned the movies-loader GitHub repository and checked out the develop branch to fetch the latest source code changes from the remote repository, as shown in figure 7.20.

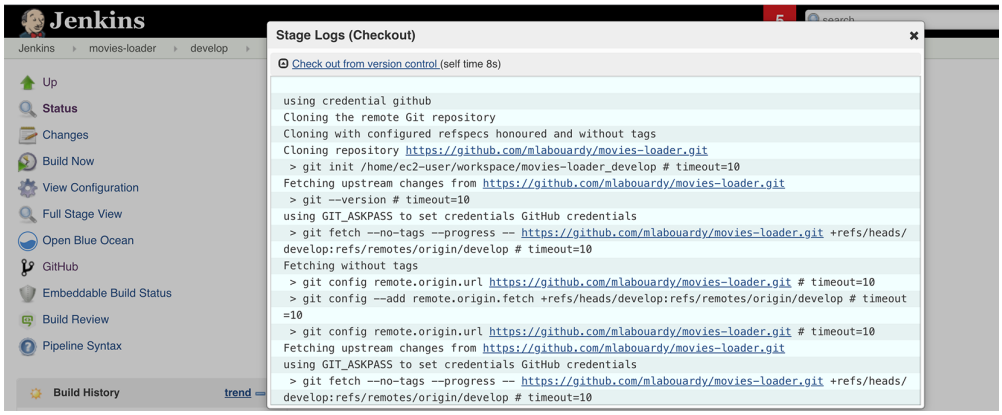


Figure 7.20 Checkout stage logs

To view the complete build log, look for the Build History on the left side. The Build History tab will list all the builds that have been run. Click the last build number; see figure 7.21.

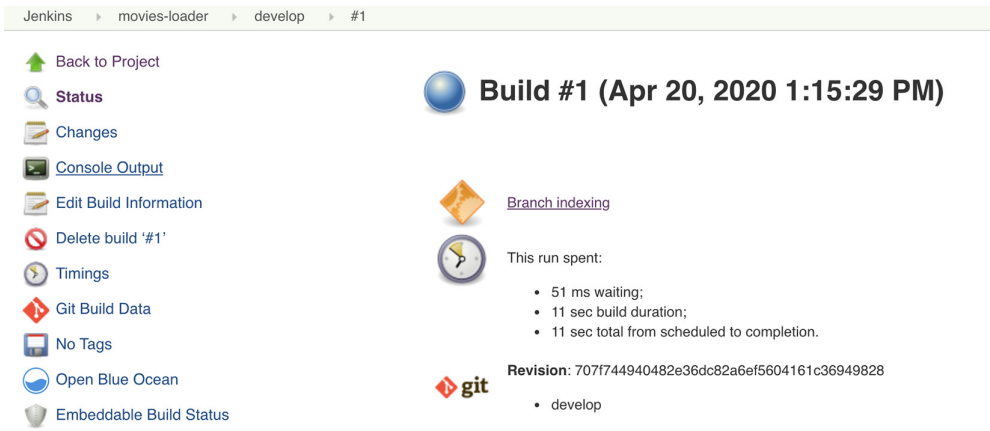


Figure 7.21 Build number settings

Then, click the Console Output item from the left corner. The complete build logs will be displayed, as shown in figure 7.22.

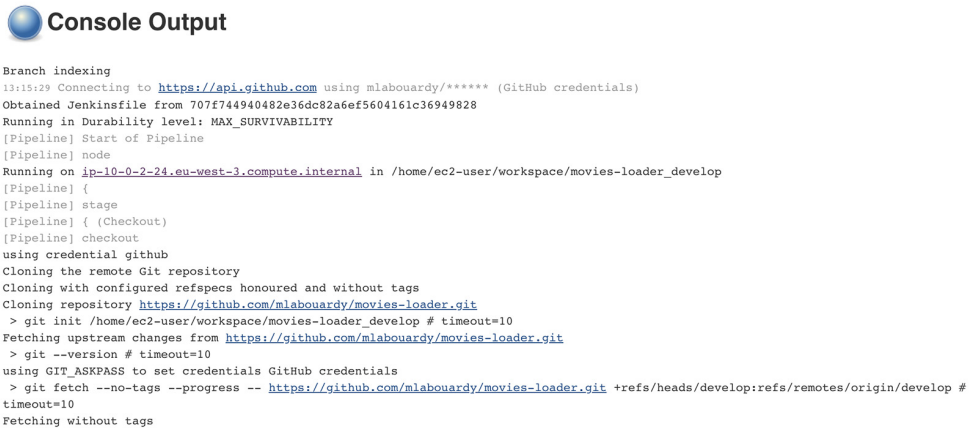


Figure 7.22 Build console logs

Now that we have created a Jenkins job for movies-loader, let's create another Jenkins job for the movies-parser service; once again, head over to Jenkins main page and click the New Item button. However, to save time, copy the configuration from the previous job, as shown in figure 7.23.

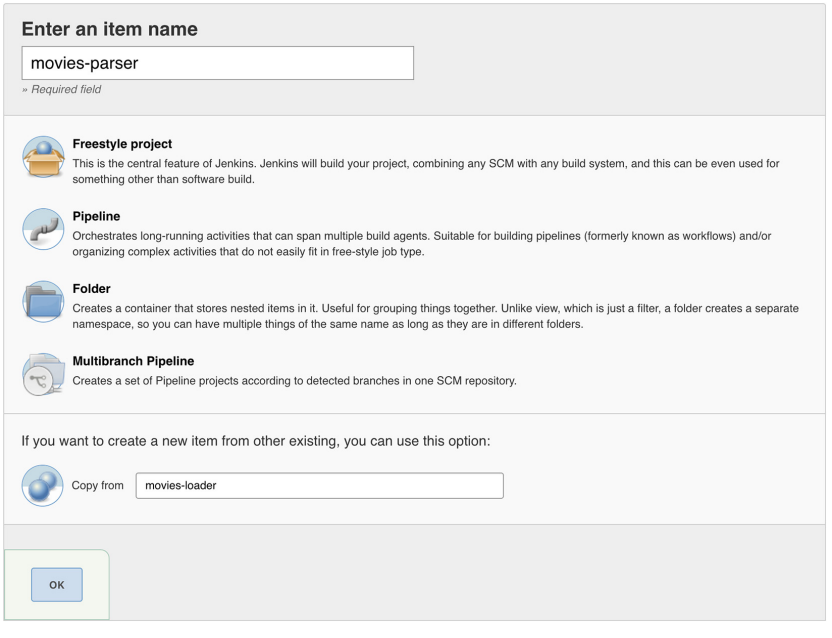


Figure 7.23 Parser job's creation

Click the OK button. The movies-parser job will reflect all features of the cloned movies-loader job. Update appropriately the GitHub repository HTTPS clone URL, job description, and display name, as shown in figure 7.24.

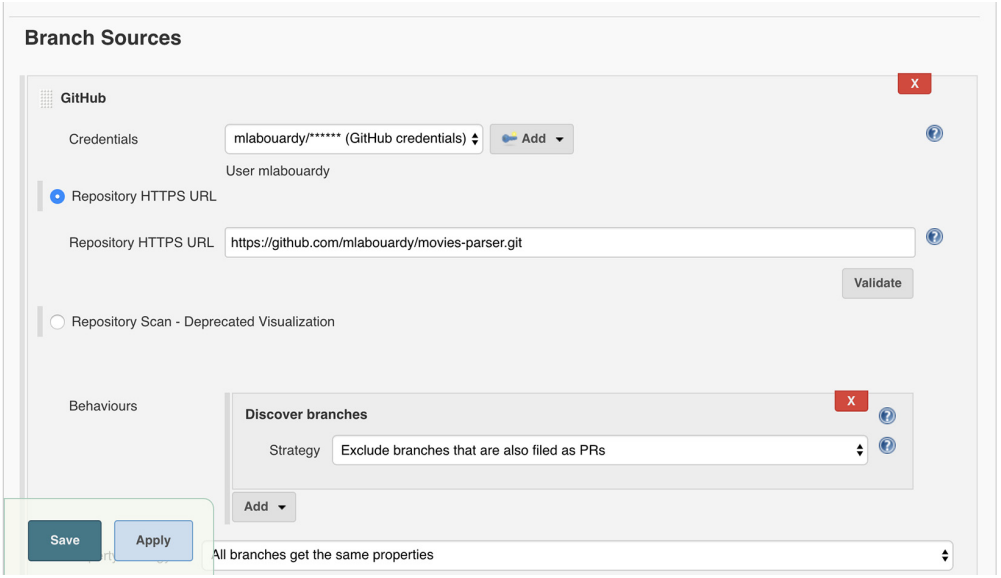


Figure 7.24 Parser job GitHub configuration

Push the same Jenkinsfile used in the previous job to the develop branch of the movies-parser GitHub repository. Then click Apply for changes to take effect.

After saving, the build will always run from the current version of Jenkinsfile into the repository, as shown in figure 7.25.

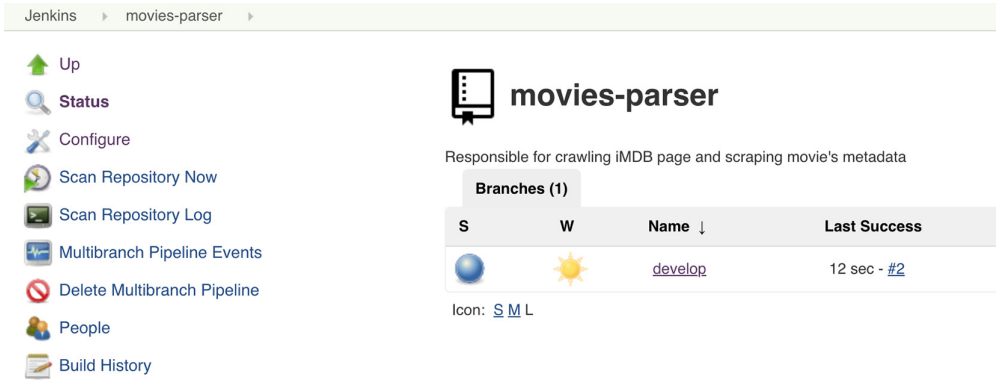


Figure 7.25 Parser job list of active branches

Follow the same steps to create Jenkins jobs for the movies-store and movies-marketplace services.

While Git is the most used distributed version control nowadays, Jenkins comes with built-in support for Subversion. To use source code from a Subversion repository, you simply provide the corresponding Subversion URL—it will work fine with any of the three Subversion protocols of HTTP, SVN, or File. Jenkins will check that the URL is valid as soon as you enter it. If the repository requires authentication, you can create a Jenkins credential of type Username with Password, and select it from the Credentials drop-down list, as shown in figure 7.26.

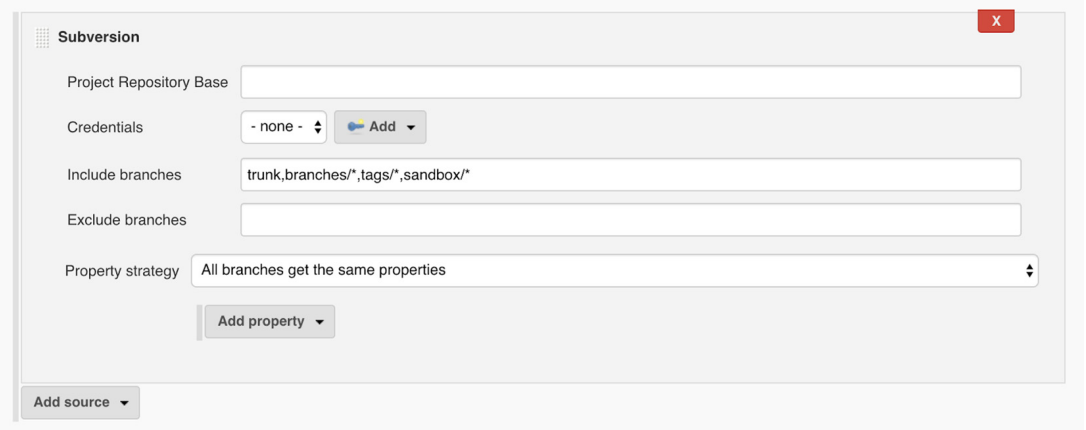


Figure 7.26 SVN repository configuration

You can fine-tune the way Jenkins obtains the latest source code from your Subversion repository by selecting an appropriate value in the Check-out Strategy drop-down list.

7.4 Discovering Jenkins jobs' XML configuration

Another way to create or clone a multibranch pipeline job is to export the config.xml file of an existing job. The XML file contains, as you might expect, the configuration details for the build job.

You can view the XML configuration of a job by pointing your browser to JENKINS_DNS/job/JOB_NAME/config.xml. It should dump the job XML definition in the browser page, as shown in figure 7.27.

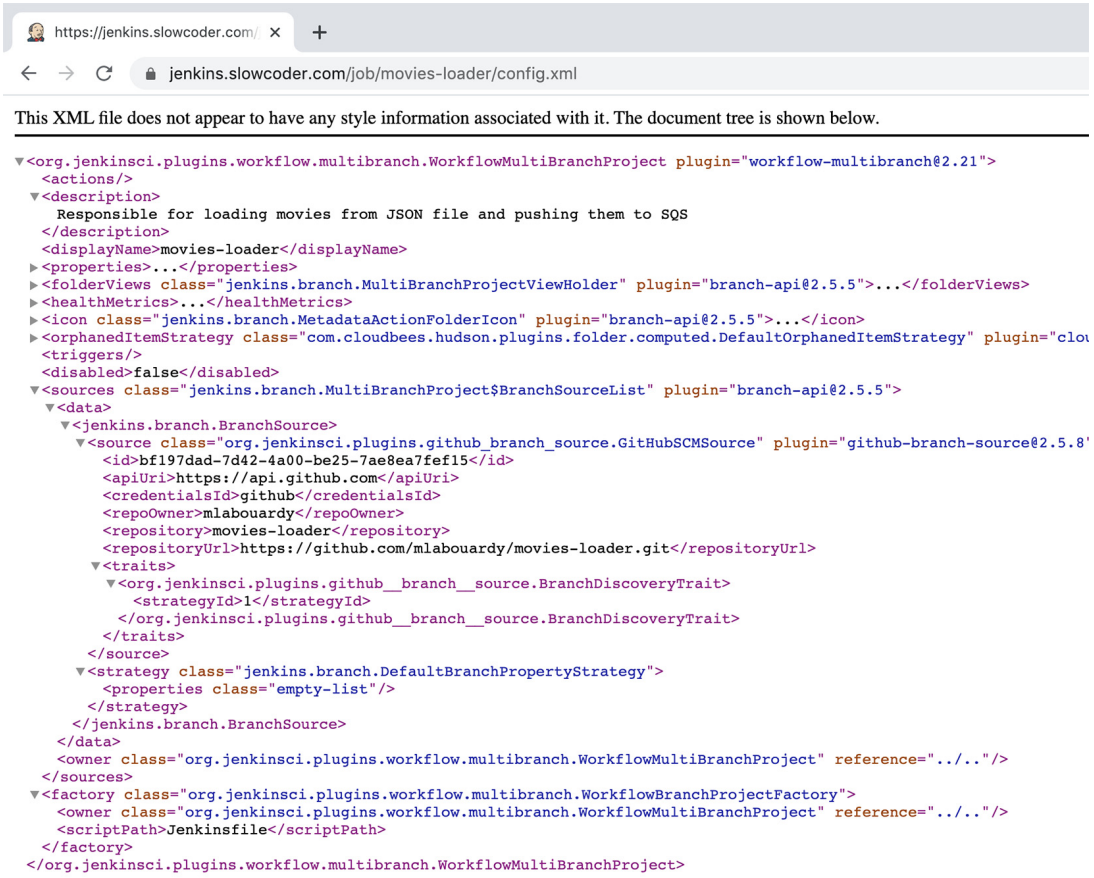


Figure 7.27 Job XML configuration

Save the job definition in an XML file and update the XML tags in table 7.2 with the appropriate values based on the target Jenkins job you're planning to create.

Table 7.2 XML tags

XML tag	Description
`<description>`	Meaningful description explaining in a few words the purpose of the Jenkins job
`<displayName>`	Jenkins job's display name; general practice is to use the name of the repository storing the source code as a value for display name
`<repository>`	Name of the GitHub repository holding the source code, such as movies-store
`<repositoryURL>`	GitHub repository HTTPS clone URL, set in the following format: https://github.com/username/repository.git

NOTE In chapter 14, we will cover how to use the Jenkins CLI to automate the import and export of multiple jobs and plugins in Jenkins.

The following listing is an example of an XML config file for the movies-store job. It illustrates a typical structure of a Jenkins job XML configuration.

Listing 7.2 Movies store config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<org.jenkinsci.plugins.workflow
.multibranch.WorkflowMultiBranchProject plugin="workflow-multibranch@2.21">
   <actions />
   <description>Movies store RESTful API</description>
   <displayName>movies-store</displayName>
   <sources class="jenkins.branch
.MultiBranchProject$BranchSourceList" plugin="branch-api@2.5.5">
      <data>
         <jenkins.branch.BranchSource>
            <source class="org.jenkinsci.plugins
.github_branch_source.GitHubSCMSource" plugin="github-branch-source@2.5.8">
               <id>bf197dad-7d42-4a00-be25-7ae8ea7fef15</id>
               <apiUri>https://api.github.com</apiUri>
               <credentialsId>github</credentialsId>
               <repoOwner>mlabouardy</repoOwner>
               <repository>movies-store</repository>
               <repositoryUrl>
https://github.com/mlabouardy/movies-store.git
               </repositoryUrl>
               <traits>

     <org.jenkinsci.plugins.github__branch__source.BranchDiscoveryTrait>
                     <strategyId>1</strategyId>
</org.jenkinsci.plugins.github__branch__source.BranchDiscoveryTrait>
               </traits>
            </source>
         </jenkins.branch.BranchSource>
      </data>
   </sources>
</org.jenkinsci.plugins.workflow.multibranch.WorkflowMultiBranchProject>
```

Defines the job's name and description

Defines the project GitHub repository URL (HTTPS format)

Tells Jenkins to scan all branches in the GitHub repository looking for a Jenkinsfile

NOTE The XML has been cropped for brevity. The full job XML definition is available in the GitHub repository in chapter7/jobs/movies-store.xml.

Once you have updated the config.xml file with the appropriate values, issue an HTTP POST request with the job XML definition as a payload to the Jenkins URL with a query parameter `name` equal to the target job's name. Figure 7.28 shows an example for creating a movies-store job with a Postman HTTP API client.

NOTE If CSRF protection is enabled on Jenkins, you will need to create an API token instead of a crumb issuer token. For more information, refer to chapter 2.

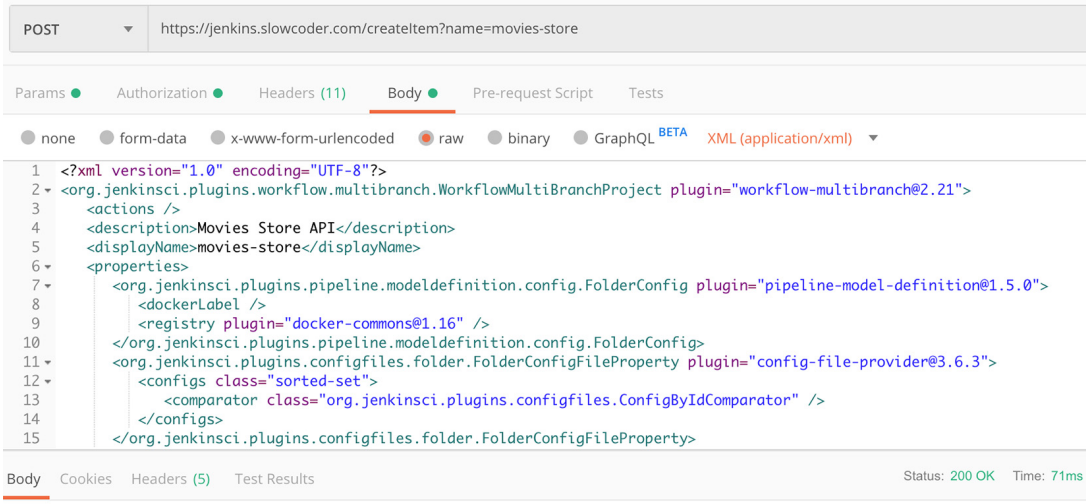


Figure 7.28 Job creation Jenkins RESTful API with Postman

A one-line cURL command can also be used to clone and create a new job:

```
curl -s https:///<USER>:<API_TOKEN>@JENKINS_HOST/job/JOBNAME/config.xml
| curl -X POST 'https:///<USER>:<API_TOKEN>@JENKINS_HOST/
     createItem?name=JOBNAME'
--header "Content-Type: application/xml" -d @-
```

The Jenkins API token (`API_TOKEN` variable) can be created from the Jenkins dashboard by logging with the user that you want to generate the API token for. Then open the user profile page and click Configure to open the user configuration page.

Locate the Add new Token button, give a name to the new token, and click the Generate button, as shown in figure 7.29. Retrieve the token and replace the `API_TOKEN` variable in the preceding cURL commands with the generated token value.

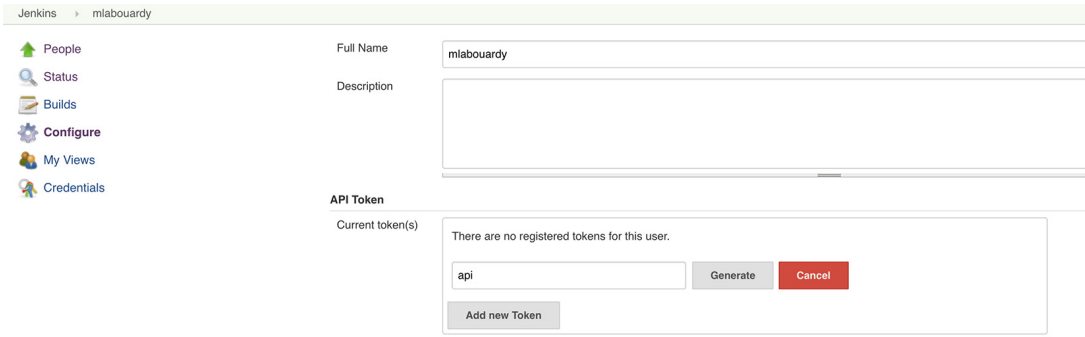


Figure 7.29 Jenkins API token generation

NOTE Jenkins jobs can also be created by copying the XML file directly to the /var/lib/jenkins/jobs/<*Job name*> folder on the Jenkins master instance and restarting Jenkins with the `service jenkins restart` command for changes to take effect.

Once the four Jenkins jobs are created, you should have the jobs shown in figure 7.30 on the Jenkins main page. You can organize these jobs in one view by creating a Jenkins folder. You can create a folder named Watchlist and move these jobs to it.

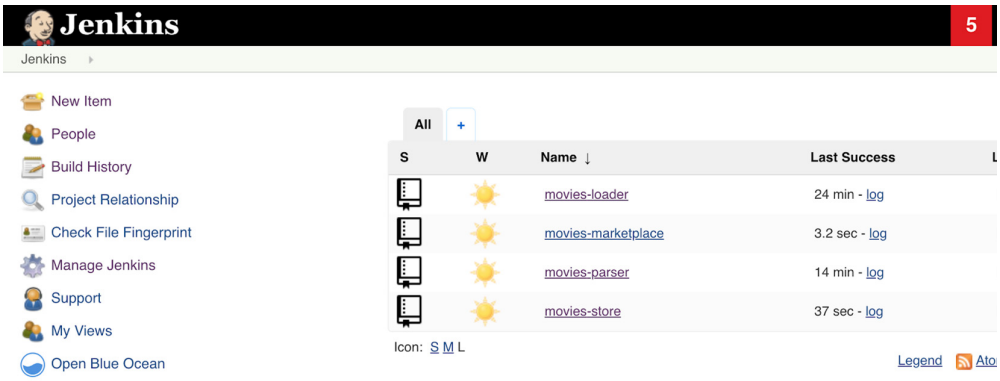


Figure 7.30 Microservices jobs in Jenkins

To do so, follow these steps: From the sidebar, click New Item, enter `Watchlist` as a name in the text box, and select Folder to create the folder. To move the existing jobs to the folder, click the arrow to the right of the job and select Move. Select Watchlist as the desired folder and click Move.

The microservices jobs will be accessible with the following URL format: JENKINS_DNS/job/Watchlist/job.

The Jenkins CLI can be used to import or export a job even if its usage is deprecated and not recommended for security vulnerabilities (at least for Jenkins 2.53 and older versions). You can run this command to import your Jenkins job XML file:

```
java -jar jenkins-cli.jar -s JENKINS_URL
-auth USERNAME:PASSWORD
create-job movies-marketplace < config.xml
```

An alternative authentication method is to use an access token by replacing the `-auth` option with the `username:token` argument.

7.5 Configuring SSH authentication with Jenkins

Previously, you learned to configure GitHub on Jenkins with username and password credentials. We also covered how to create a GitHub API access token with granular

permissions. This section covers how to use SSH keys instead to authenticate with project repositories.

> **NOTE** You can generate a one-purpose SSH key for SSH authentication with remote Git repositories by using the `ssh-keygen` command.

First, configure the Jenkins public SSH key on GitHub. You can configure SSH on the GitHub repository by going to the repository settings and adding a deploy key from the Deploy Keys section. Or simply configure the SSH key globally from the user profile settings. Give a name such as `Jenkins` and paste the public key (from the id_rsa.pub file); see figure 7.31.

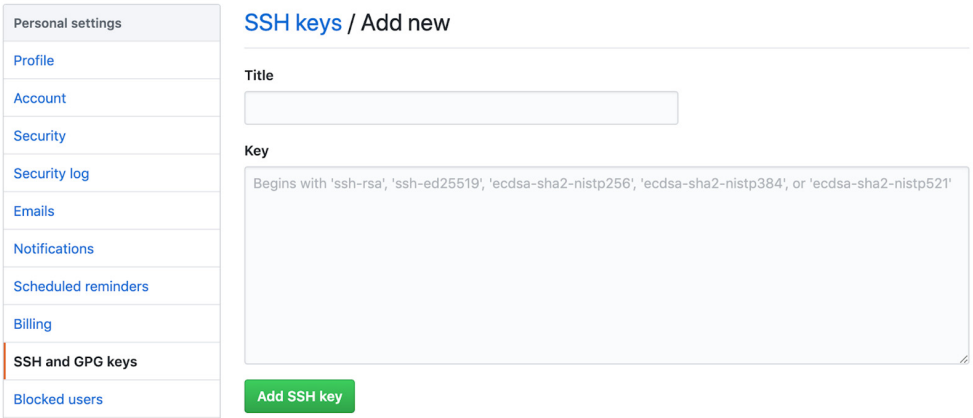


Figure 7.31 GitHub SSH configuration

> **NOTE** Once a key has been attached to one repository as a deploy key, it cannot be used on another repository.

To determine whether the key is successfully configured, type the following command on your Jenkins SSH session. Use the `-i` flag to provide the path to the Jenkins private key:

```
ssh -T -ai PRIVATE_KEY_PATH git@github.com
```

If the response looks something like `Hi username`, the key has been properly configured.

Now go to Credentials from the left pane inside the Jenkins console and click Global. Then select Add Credentials and create a credential of type SSH Username with Private Key. Give it a name and set the value of the SSH private key, as shown in figure 7.32. The Username should be the username for the GitHub account that hosts the project. In the Passphrase text box, write the passphrase given while generating the SSH RSA key. If not set, leave it blank.

Figure 7.32 Configuring GitHub SSH credentials on Jenkins

Head back to the Jenkins job, and under Branch Sources, choose Git from the drop-down list, set the repository SSH clone URL, and select the saved credentials title name; see figure 7.33.

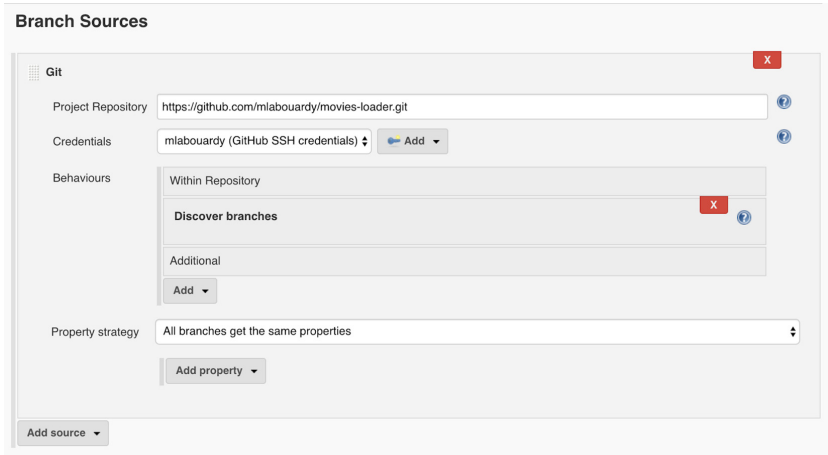


Figure 7.33 Configuring the Jenkins job to use SSH keys

If you go to the build output, it should clearly list that the SSH key is being used for authentication. The following is sample output highlighting the same:

```
Branch indexing
 > git rev-parse --is-inside-work-tree # timeout=10
Setting origin to git@github.com:mlabouardy/movies-loader.git
 > git config remote.origin.url git@github.com:mlabouardy/movies-loader.git # timeout=10
Fetching origin...
Fetching upstream changes from origin
 > git --version # timeout=10
 > git config --get remote.origin.url # timeout=10
using GIT_SSH to set credentials GitHub SSH credentials
 > git fetch --tags --progress -- origin +refs/heads/*:refs/remotes/origin/* # timeout=10
```

Until now, the `Checkout` stage has been using the credentials and settings configured in the current Jenkins job. If you want to customize the settings and use specific credentials, you can replace it with the following listing.

Listing 7.3 Customized git clone command

```
stage('Checkout') {
    steps {
        git branch: 'develop',
            credentialsId: 'github-ssh',
            url: 'git@github.com:mlabouardy/movies-loader.git'
    }
}
```

This example will clone the develop branch of the movies-loader GitHub repository, using the SSH credentials saved in the github-ssh Jenkins credentials.

7.6 Triggering Jenkins builds with GitHub webhooks

So far, we have always built the pipeline manually by clicking the Build Now button. It works but is not very convenient. All team members would have to remember that after committing to the repository, they need to open Jenkins and start the build.

To trigger the jobs by push event, we will create a webhook on the GitHub repository of each service, as illustrated in figure 7.34. Remember, a Jenkinsfile should also be present on the respective branch to tell Jenkins what it needs to do when it finds a change in the repository.

> **NOTE** *Webhooks* are user-defined HTTP callbacks. They are triggered by an event in a web application and can facilitate integrating different applications or third-party APIs.

Figure 7.34 Webhook explained

Navigate to the GitHub repository that you want to connect to Jenkins and click the repository Settings option. In the menu on the left, click Webhooks, as shown in figure 7.35.

GitHub webhooks allow you to notify external services when certain Git events happen (push, merge, commit, fork, and so forth) by sending a POST request to the configured service URL.

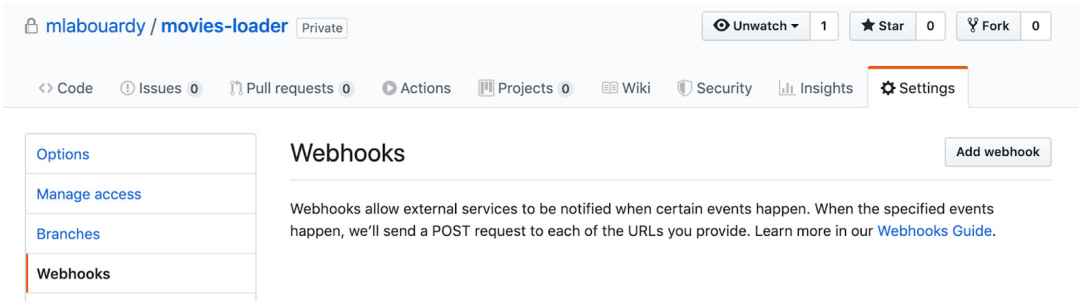


Figure 7.35 GitHub Webhooks section

Click the Add Webhook button to bring up the associated dialog, shown in figure 7.36. Fill in the form with the following values:

- The payload URL should be in the following format: JENKINS_URL/github-webhook/ (make sure it includes the last forward slash).
- The content type can be either application/json or application/x-www-form-urlencoded.
- Select the push event as a trigger and leave the Secret field empty (unless a secret has been created and configured in the Jenkins Configure System > GitHub Plugin section).

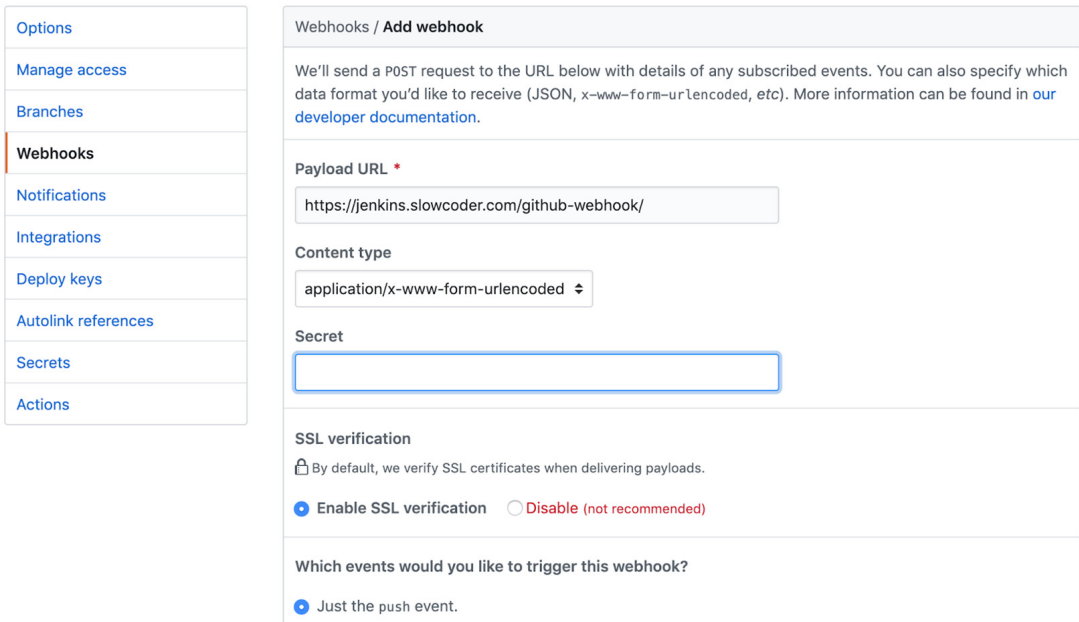


Figure 7.36 Jenkins webhook settings

Leave the rest of the options at their default values and then click the Add Webhook button. A test payload should be sent to Jenkins to set up the hook. If the payload is successfully received by Jenkins, you should see the webhook with a green check mark, as shown in figure 7.37.

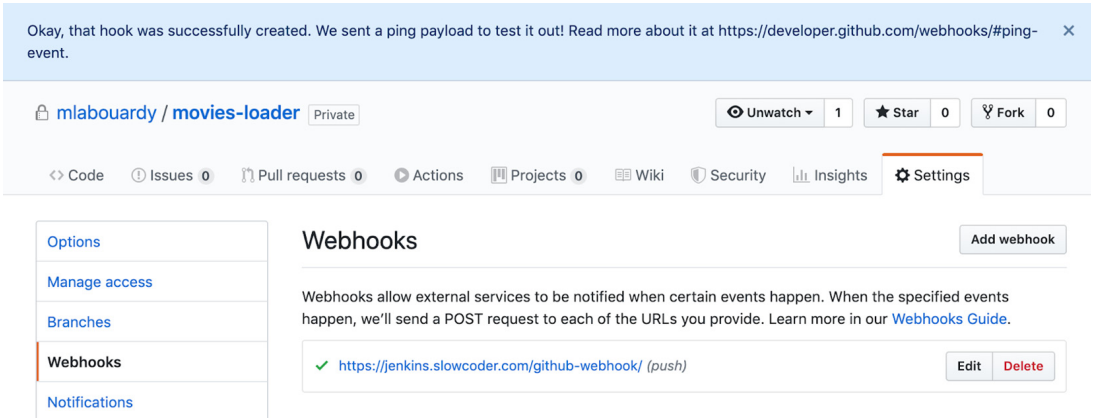
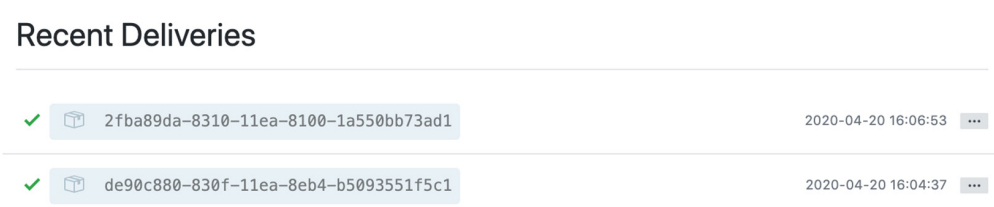


Figure 7.37 Jenkins webhook settings

With these GitHub updates done, if you push some changes to the Git repository, a new event should get kicked off automatically. In this scenario, we update the README.md file:



Go back to your Jenkins project, and you'll see that a new job was triggered automatically from the commit we made at the previous step. Click the little arrow next to the job and choose Console Output. Figure 7.38 shows the output.

The update readme message confirms that the build was triggered automatically upon pushing the new README.md to the GitHub repository. Now, every time you publish your changes to your remote repository, GitHub will trigger your new Jenkins job. Create a similar webhook on the remaining GitHub repositories by following the same procedure.

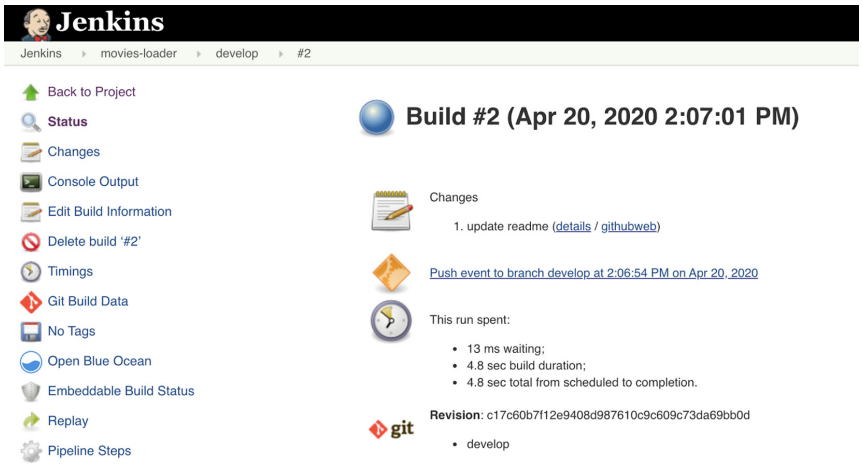


Figure 7.38 GitHub push event

NOTE If you want SVN users to continuously trigger Jenkins jobs after every commit, you can either configure Jenkins to periodically poll the SVN server or set up a post-commit hook on the remote repository.

In a different situation, the Jenkins dashboard might not be accessible from a public network. Instead of executing jobs manually, you can set up a public reverse proxy as middleware between the GitHub server and Jenkins, and configure the GitHub webhook to use the middleware URL. Figure 7.39 explains how to use AWS managed services to set up a webhook forwarder for a Jenkins instance within a VPC.

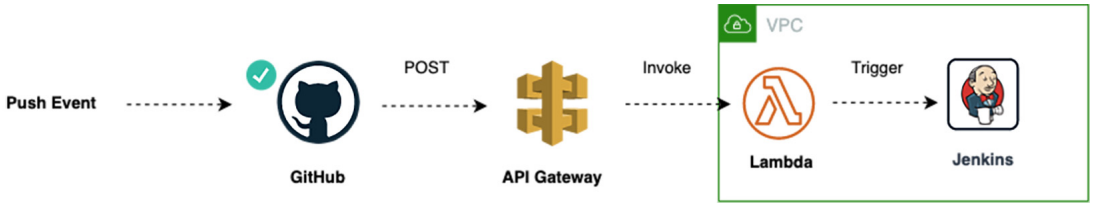


Figure 7.39 GitHub webhook setup with API Gateway

NOTE You can generalize this approach to other services too, such as Bitbucket or DockerHub—or anything, really, that emits webhooks.

If you're using AWS as a cloud provider, you can use a managed proxy called Amazon API Gateway to invoke a Lambda function when a POST request is invoked on a specific endpoint, as shown in figure 7.40.

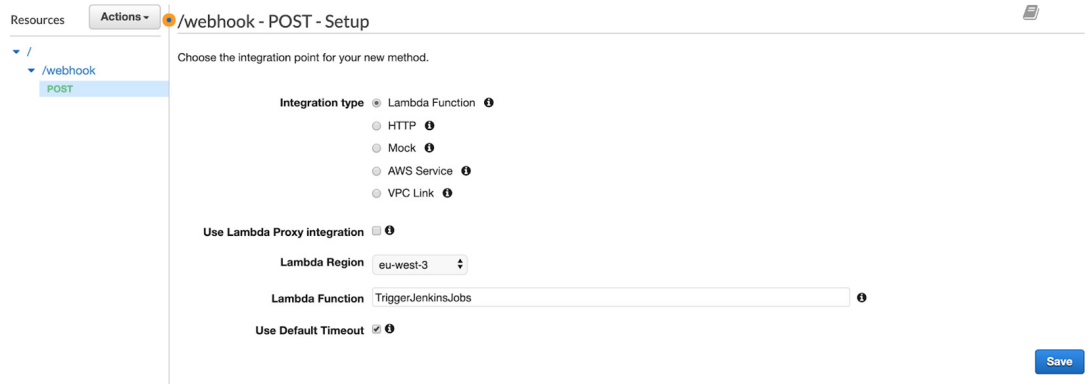


Figure 7.40 Triggering a Lambda function with API Gateway

The Lambda function will receive the GitHub payload from API Gateway and relay it to the Jenkins server. The following listing is a function entry point written in JavaScript.

Listing 7.4 Lambda function handler

```
const Request = require('request');
exports.handler = (event, context, callback) => {
    Request.post({
        url: process.env.JENKINS_URL,
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "X-GitHub-Event": event.headers["X-GitHub-Event"]
        },
        json: JSON.parse(event.body)
    }, (error, response, body) => {
        callback(null, {
            "statusCode": 200,
            "headers": {
                "content-type": "application/json"
            },
            "body": "success",
            "isBase64Encoded": false
        })
    })
};
```

To deploy the GitHub webhook and AWS resources, we will use Terraform. But first, we need to create a deployment package with the Lambda function index.js entry point. The deployment package is a zip file that can be generated with the following command:

```
zip deployment.zip index.js
```

> **NOTE** This section assumes you're familiar with the usual Terraform plan/apply workflow. If you're new to Terraform, refer to chapter 5.

Next, we define a lambda.tf file containing the Terraform resource definition for an AWS Lambda function. We set the runtime to be a Node.js runtime environment (the Lambda handler is written in JavaScript). We define an environment variable named `JENKINS_URL` with a value pointing to the Jenkins web dashboard URL, as shown in the next listing.

Listing 7.5 Lambda function based on Node.js runtime

```
resource "aws_lambda_function" "lambda" {
  filename = "../deployment.zip"
  function_name = "GitHubWebhookForwarder"
  role = aws_iam_role.role.arn
  handler = "index.handler"
  runtime = "nodejs14.x"
  timeout = 10
  environment {
    variables = {
      JENKINS_URL = var.jenkins_url
    }
  }
}
```

Then, we define an API Gateway RESTful API to trigger the preceding Lambda function when a POST request occurs on the /webhook endpoint. Create a new file, apigateway.tf, in the same directory as our lambda.tf from the previous step and paste the following content.

Listing 7.6 API Gateway RESTful API

```
resource "aws_api_gateway_rest_api" "api" {
  name        = "GitHubWebHookAPI"
  description = "GitHub Webhook forwarder"
}

resource "aws_api_gateway_resource" "path" {
   rest_api_id = aws_api_gateway_rest_api.api.id
   parent_id   = aws_api_gateway_rest_api.api.root_resource_id
   path_part   = "webhook"
}

resource "aws_api_gateway_integration" "request_integration" {
  rest_api_id = aws_api_gateway_rest_api.api.id
  resource_id = aws_api_gateway_method.request_method.resource_id
  http_method = aws_api_gateway_method.request_method.http_method
  type        = "AWS_PROXY"
  uri         = aws_lambda_function.lambda.invoke_arn
  integration_http_method = "POST"
}
```

Finally, in the following listing, we create an API Gateway deployment to activate the configuration and expose the API at a URL that can be used for webhook configuration. We use a Terraform output variable to display the API deployment URL by referencing the API deployment stage.

Listing 7.7 API new deployment stage

```
resource "aws_api_gateway_deployment" "stage" {
   rest_api_id = aws_api_gateway_rest_api.api.id
   stage_name  = "v1"
}

output "webhook" {
    value = "${aws_api_gateway_deployment.stage.invoke_url}/webhook"
}
```

Before issuing the `terraform apply` command, you need to define the variables used in the preceding resources. The variables.tf file will contain the list of variables, which are detailed in table 7.3.

Table 7.3 GitHub webhook proxy's Terraform variables

Variable	Type	Value	Description
`region`	String	`none`	The AWS region in which to deploy AWS resources. It can also be sourced from the `AWS_REGION` environment variable.
`shared_credentials_file`	String	`none`	The path to the shared credentials file. If this is not set and a profile specified, ~/.aws/credentials will be used.
`aws_profile`	String	`profile`	The AWS profile name as set in the shared credentials file.
`jenkins_url`	String	`none`	The Jenkins URL, which has the format http://IP:8080, or uses HTTPS if an SSL certificate is being used.

When Terraform finishes deploying the AWS resources, a new Lambda function called `GitHubWehookForwarder` should be created with a trigger of type API Gateway, as shown in figure 7.41.

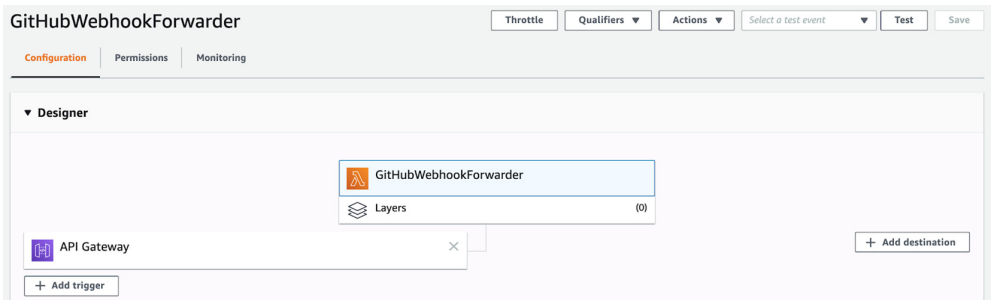


Figure 7.41 `GitHubWebhookForwarder` Lambda function

Furthermore, Terraform will display the RESTful API deployment URL, which you can use to create a webhook on the target GitHub repository, as shown in figure 7.42.

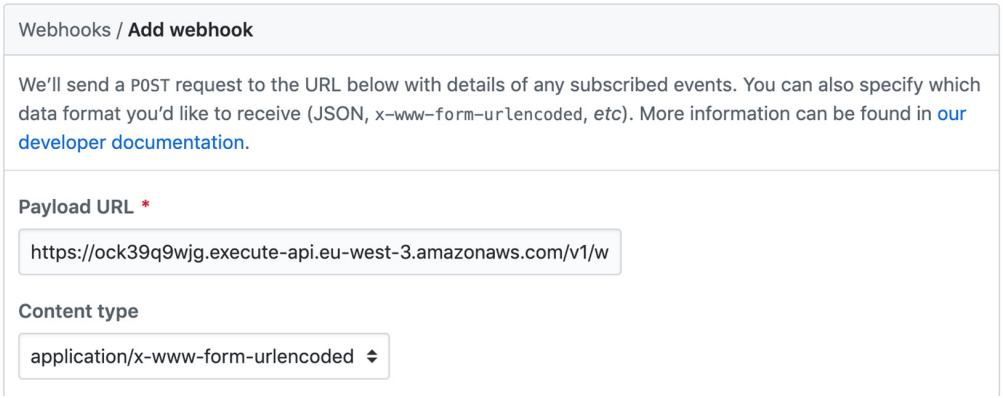


Figure 7.42 GitHub webhook based on API Gateway URL

Webhooks should be flowing now. You can make a change to your repository and check that a build starts soon after. You also can add an extra security layer, by requiring a request secret and validating the incoming request signature on the Lambda function side.

If you're running Jenkins locally, you can use a build trigger to poll SCM and schedule it to run periodically, as shown in figure 7.43. In such a case, Jenkins would regularly check the repository, and if anything changed, it would run the job.

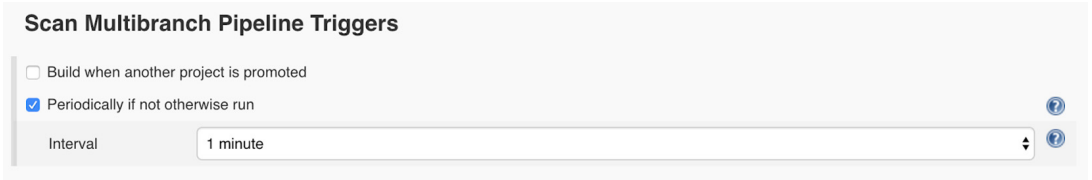


Figure 7.43 Under the job's settings, you can define the interval of checks.

After running the pipeline manually for the first time, the automatic trigger is set. Then it checks GitHub every minute, and for new commits, starts a build. To test that it works as expected, you can commit and push anything to the GitHub repository and see that the build starts.

> **NOTE** Polling SCM, even if it's less intuitive, might be useful if Git commits are frequent and the build takes a long time, so executing a build upon a push event every time would cause an overload.

So far, you have learned how to integrate Git repositories with Jenkins and define multibranch pipeline jobs. And we have ended up creating our first complete commit pipeline. However, with the current state, it doesn't do much. In the following chapters, we will see what improvements can be made to make the commit pipeline even better, and we will start by running automated tests within the Jenkins pipelines.

Summary

- A webhook is a mechanism to automatically trigger the build of a Jenkins project upon a commit pushed in a remote Git repository.
- The development workflow should be carefully chosen inside the team or organization because it affects the CI process and defines the way the code is developed.
- Using multi-repo or mono-repo strategies to organize the codebase will define the complexity of a CI/CD pipeline as the number of applications evolves within an organization.
- A pipeline can go through the standard code development process (code review, pull requests, automated testing, and so forth) when a Jenkinsfile and application source code live together on the same Git repository.
- Jenkins stores configuration files for the jobs it runs in an XML file. Editing these XML configuration files has the same effect as editing Jenkins jobs through the web dashboard.
- A reverse proxy can be useful to let Git webhooks reach a running Jenkins server behind a firewall.

8 Running automated tests with Jenkins

This chapter covers

- Implementing CI pipelines for Python, Go, Node.js, and Angular-based services
- Running pre-integration tests and automated UI testing with Headless Chrome
- Executing SonarQube static code analysis within Jenkins pipelines
- Running unit tests inside a Docker container and publishing code coverage reports
- Integrating dependency checks in a Jenkins pipeline and injecting security in DevOps

In the previous chapter, you learned how to set up multibranch pipeline jobs for containerized microservices and for continuously triggering Jenkins upon push events with webhooks. In this chapter, we will run automated tests within the CI pipeline. Figure 8.1 summarizes the current CI workflow stages.

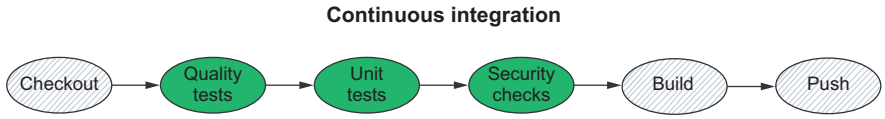


Figure 8.1 The test stages covered in this chapter

Test automation is widely considered a cornerstone of Agile development. If you want to release fast—even daily—with reasonable quality, you have to move to automated testing. On the other hand, giving less importance to testing can result in customer dissatisfaction and a delayed product. However, automating the testing process is a bit more difficult than automating the build, release, and deployment processes. Automating nearly all the test cases used in an application usually takes a lot of effort. It is an activity that matures over time. It is not always possible to automate all the testing. But the idea is to automate whatever testing is possible.

By the end of this chapter, we will implement the test stage in the target CI pipeline shown in figure 8.2.

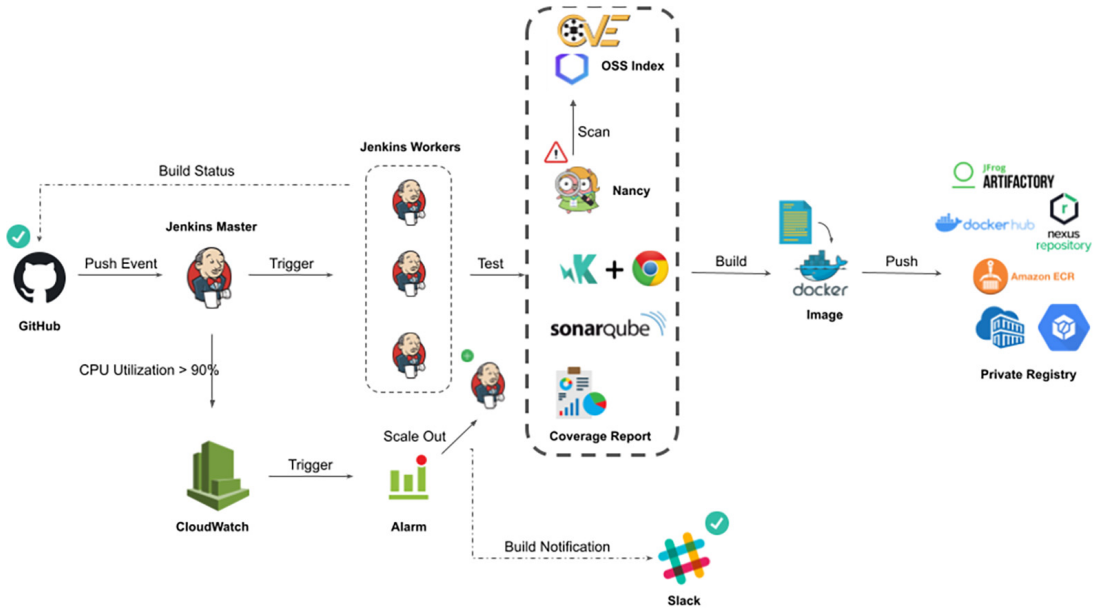


Figure 8.2 Target CI pipeline

Before resuming the CI pipeline implementation, a quick reminder regarding the web distributed application we're integrating with Jenkins: it's based on a microservices architecture and split into components/services written in different programming languages and frameworks. Figure 8.3 illustrates this architecture.

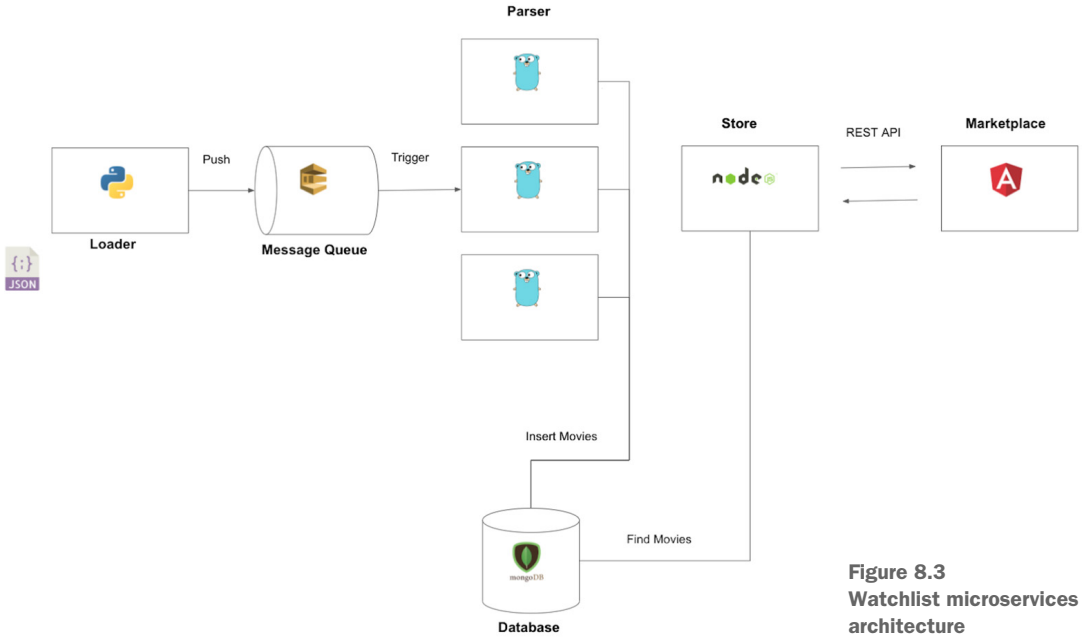


Figure 8.3 Watchlist microservices architecture

In the following sections, you will learn how to integrate various types of tests in our CI workflow. We will start with unit testing.

8.1 Running unit tests inside Docker containers

Unit testing is the frontline effort to identify issues as early as possible. The test needs to be small and quick to execute to be efficient.

The movies-loader service is written in Python. To define unit tests, we're going to use the unittest framework (it comes bundled with the installation of Python). To use it, we import the unittest module, which offers a rich set of methods to construct and run tests. The following listing, test_main.py, demonstrates a short unit test to test the JSON loading and parsing mechanism.

Listing 8.1 Unit testing in Python

```
import unittest
import json

class TestJSONLoaderMethods(unittest.TestCase):
    movies = []

    @classmethod
    def setUpClass(cls):
        with open('movies.json') as json_file:
            cls.movies = json.load(json_file)

    def test_rank(self):
        self.assertEqual(self.movies[0]['rank'], '1')
```

```
    def test_title(self):
        self.assertEqual(self.movies[0]['title'], 'The Shawshank Redemption')

    def test_id(self):
        self.assertEqual(self.movies[0]['id'], 'tt0111161')

if __name__ == '__main__':
    unittest.main()
```

The `setUpClass()` method allows us to load the movies.json file before the execution of each test method. The three individual tests are defined with methods whose names start with the prefix `test`. This naming convention informs the test runner about which methods represent tests. The crux of each test is a call to `assertEqual()` to check for an expected result. For instance, we check whether the first movie's title attribute parsed from the JSON file is `The Shawshank Redemption`.

To run the test, we can execute the `python test_main.py` command on Jenkins. However, it requires Python 3 to be installed. To avoid installing the runtime environment for each service we are building, we will run the tests inside a Docker container. That way, we will be using Docker as an execution environment across all Jenkins workers.

On the movies-loader repository, create a Dockerfile.test file by using your favorite text editor or IDE with the following content.

Listing 8.2 Movie loader's Dockerfile.test

```
FROM python:3.7.3
WORKDIR /app
COPY test_main.py .
COPY movies.json .
```

The Dockerfile is built from a Python 3.7.3 official image. It sets a working directory called app, and copies the test files to the working directory.

> **NOTE** The name convention *Dockerfile.test* is used to avoid name conflict with *Dockerfile*, which is used to build the main application's Docker image.

Now, update the Jenkinsfile given in listing 7.1 and add a new `Unit Test` stage, as shown in the following listing. The stage will create a Docker image based on Dockerfile .test and then spin up a Docker container from the created image to run the `python test_main.py` command to launch unit tests. The `Unit Test` stage uses a DSL-like syntax to define the shell instructions.

Listing 8.3 Movie loader's Jenkinsfile

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

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

```
    stage('Unit Tests'){
        sh "docker build -t ${imageName}-test -f Dockerfile.test ."
        sh "docker run --rm ${imageName}-test"
    }
}
```

The `docker build` and `docker run` commands are used to create an image and build a container from the image, respectively.

> **NOTE** The `--rm` flag in the `docker run` command is used to automatically clean up the container and remove the filesystem when the container exits.

You can use the `powershell` step in your pipeline on a Windows worker. This step has the same options as the `sh` instruction.

Commit the changes to the develop branch with the following commands:

```
git add Dockerfile.test Jenkinsfile
git commit -m "unit tests execution"
git push origin develop
```

In a few seconds, a new build should be triggered on the movies-loader job for the develop branch. From the movies-loader Multibranch Pipeline job, click the respective develop branch. On the resultant page, you will see the Stage view for the develop branch pipeline, as shown in figure 8.4.

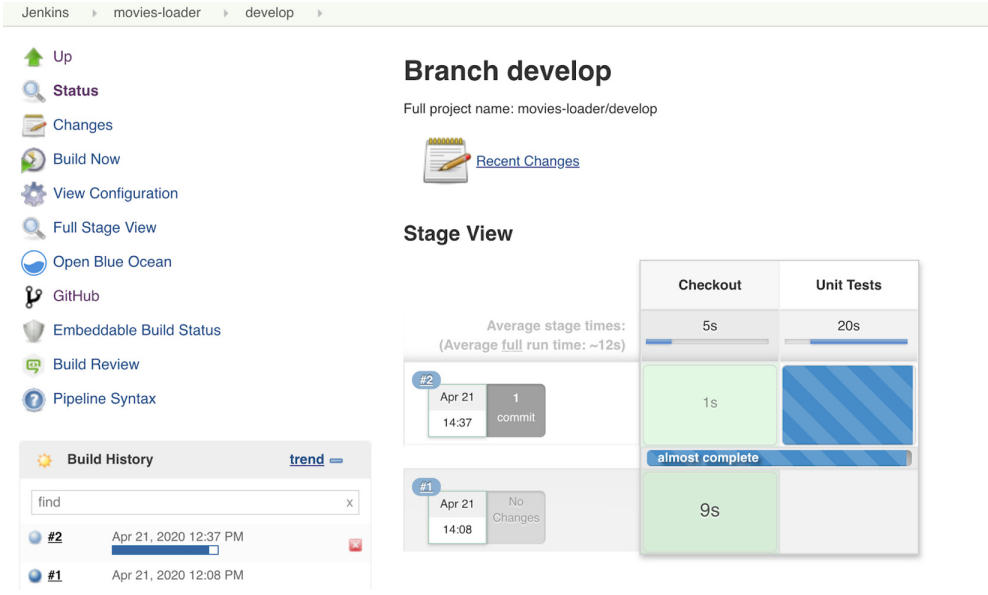


Figure 8.4 Unit test stage execution

Click the Console Output option to view the test results. All three test cases ran, and the status shows as `SUCCESS` in the logs, as you can see in figure 8.5.

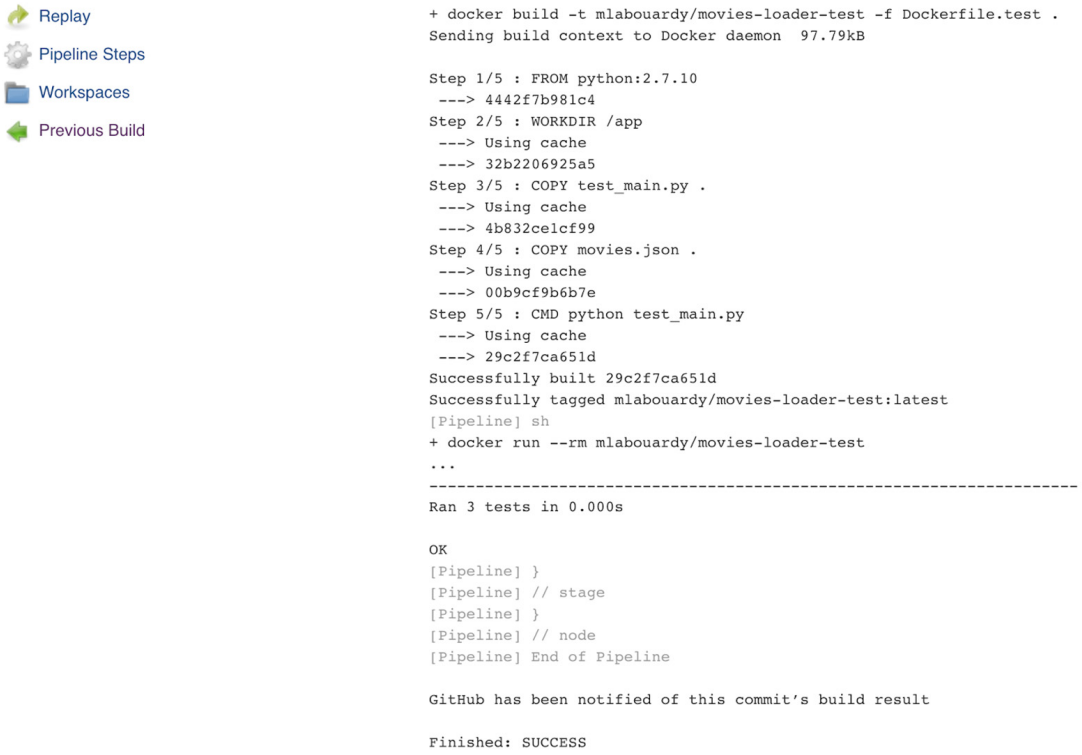


Figure 8.5 Unit test successful execution logs

The shell commands can be replaced with Docker DSL instructions. I advise using them where appropriate instead of running Docker commands via the shell, because they provide high-level encapsulation and ease of use:

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

The `docker.build()` method is similar to running the `docker build` command. The returned value of the method can be used for a subsequent call to create a Docker container and run the unit tests. Figure 8.6 shows a successful run of the pipeline.

```
$ docker run -t -d -u 500:500 -w /home/ec2-user/workspace/movies-loader_develop -v /home/ec2-user/workspace/movies-
loader_develop:/home/ec2-user/workspace/movies-loader_develop:rw,z -v /home/ec2-user/workspace/movies-loader_develop@tmp:/home/ec2-
user/workspace/movies-loader_develop@tmp:rw,z -e ******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ********
******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ******** -e ********
******** -e ******** -e ******** mlabouardy/movies-loader-test cat
$ docker top da810b2d196fb6e261456ad1ca42ed5ca7a07ebeca697aae513b29c382cfac63 -eo pid,comm
[Pipeline] {
[Pipeline] sh
+ python test_main.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
[Pipeline] }
$ docker stop --time=1 da810b2d196fb6e261456ad1ca42ed5ca7a07ebeca697aae513b29c382cfac63
$ docker rm -f da810b2d196fb6e261456ad1ca42ed5ca7a07ebeca697aae513b29c382cfac63
[Pipeline] // withDockerContainer
[Pipeline] }
```

Figure 8.6 Using the Docker DSL to run tests

To show results in a graphical, visual way, we can use the JUnit report integration plugin on Jenkins to consume an XML file generated by Python unit tests.

> **NOTE** The JUnit report integration plugin (https://plugins.jenkins.io/junit/) is installed by default in the baked Jenkins master machine image.

Update the test_main.py file to use the xmlrunner library, and pass it to the unittest .main method:

```
import xmlrunner
...
if __name__ == '__main__':
    runner = xmlrunner.XMLTestRunner(output='reports')
    unittest.main(testRunner=runner)
```

This will generate test reports in the reports directory. However, we need to address a problem: the test container will store the result of the tests that it executes within itself. We can resolve this by mapping a volume to the reports directory. Update the Jenkinsfile to tell Jenkins where to find the JUnit test report:

```
stage('Unit Tests'){
        def imageTest= docker.build("${imageName}-test",
 "-f Dockerfile.test .")
        sh "docker run --rm -v $PWD/reports:/app/reports ${imageName}-test"
        junit "$PWD/reports/*.xml"
}
```

> **NOTE** You can also get the report results by using the `docker cp` command to copy the report files into the current workspace. Then, set the workspace as an argument for the JUnit command.

Let's go ahead and execute this. This will add a chart to the project page in Jenkins after the changes are pushed to the develop branch and CI execution is completed; see figure 8.7.

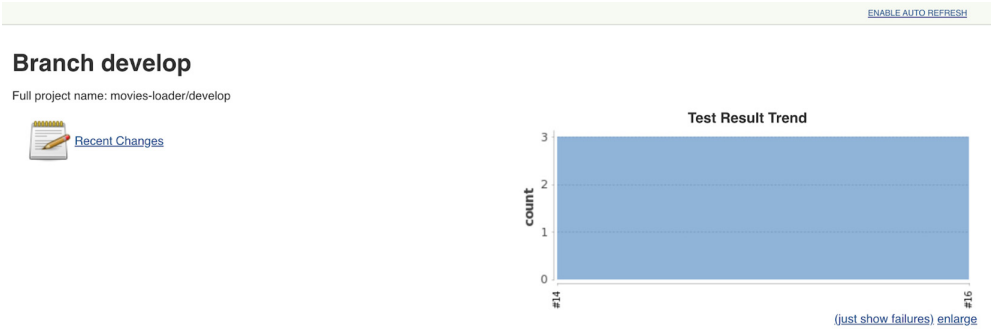


Figure 8.7 JUnit test chart analyzer

The historic graph shows several metrics (including failure, total, and duration) related to the test execution over a period of time. You can also click the chart to get more details about individual tests.

8.2 Automating code linter integration with Jenkins

Another example of tests to implement within CI pipelines is *code linting*. Linters can be used to check the source code and find typos, syntax errors, undeclared variables, and calls to undefined or deprecated functions. They can help you write better code and anticipate potential bugs. Let's see how to integrate code linters with Jenkins.

The movies-parser service is written in Go, so we can use a Go linter to make sure that the code respects the code style. A linter may sound like an optional tool, but for larger projects, it helps to keep a consistent style over your project.

Dockerfile.test uses golang:1.13.4 as a base image, and installs the `golint` tool and service dependencies, as shown in the following listing.

Listing 8.4 Movie parser's Dockerfile.test

```
FROM golang:1.13.4
WORKDIR /go/src/github.com/mlabouardy/movies-loader
ENV GOCACHE /tmp
WORKDIR /go/src/github/mlabouardy/movies-parser
RUN go get -u golang.org/x/lint/golint
COPY . .
RUN go get -v
```

Add the `Quality Tests` stage to the Jenkinsfile to build a Docker image based on Dockerfile.test with the `docker.build()` command, and then use the `inside()` instruction on the built image to start a Docker container in daemonized mode to execute the `golint` command:

```
def imageName = 'mlabouardy/movies-parser'
node('workers'){
    stage('Checkout'){
        checkout scm
    }
```

```
    stage('Quality Tests'){
       def imageTest= docker.build("${imageName}-test", "-f Dockerfile.test .")
        imageTest.inside{
            sh 'golint'
        }
    }
}
```

NOTE If an `ENTRYPOINT` instruction is defined in Dockerfile.test, the `inside()` instruction will pass the commands defined in its scope as an argument to the `ENTRYPOINT` instruction.

The `golint` execution will result in the logs shown in figure 8.8.

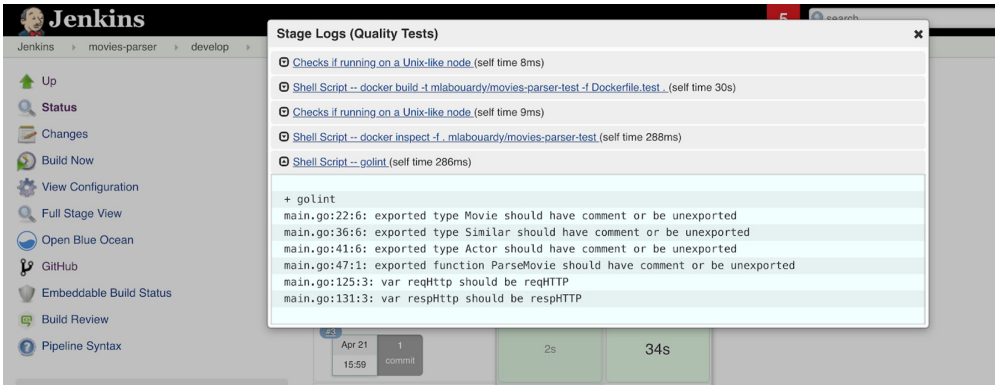


Figure 8.8 The `golint` command output identifies the missing comments

By default, `golint` prints only the style issues, and returns (with a 0 exit code), so the CI never considers that something went wrong. If you specify `-set_exit_status`, the pipeline will fail if an issue is reported by `golint`.

We can also implement a unit test for the movies-parser service. Go has a built-in testing command called `go test` and the package *testing*, which combine to give a minimal but complete unit-testing experience.

Similarly to the movies-loader service, we will write a Dockerfile.test file to execute the `go test` command that will execute tests written in the main_test.go file. The code in the following listing has been cropped for brevity and to highlight the main parts. You can browse the full code in chapter7/microservices/movies-parser/main_test.go.

Listing 8.5 Movie parser's unit test

```
package main

import (
    "testing"
)
```

```
const HTML = `
<div class="plot_summary ">
    <div class="summary_text">
        An ex-hit-man comes out of retirement to track down the gangsters
that killed his dog and took everything from him.
    </div>
    ...
</div>
`
func TestParseMovie(t *testing.T) {
    expectedMovie := Movie{
            Title:       "John Wick (2014)",
            ReleaseDate: "24 October 2014 (USA)",
            Description: "An ex-hit-man comes ...",
    }

    currentMovie, err := ParseMovie(HTML)
    if expectedMovie.Title != currentMovie.Title {
     t.Errorf("returned wrong title: got %v want %v"
, currentMovie.Title, expectedMovie.Title)
    }
}
```

This code shows the basic structure of a unit test in Go. The built-in testing package is provided by Go's standard library. A unit test is a function that accepts the argument of type `*testing.T` and calls the `t.Error()` method to indicate a failure. This function must start with a `Test` keyword, and the latter name should start with an uppercase letter. In our use case, the function tests the `ParseMovie()` method, which takes as a parameter `HTML` and returns a `Movie`'s structure.

8.3 Generating code coverage reports

The `Unit Tests` stage is straightforward: it will execute `go test` inside the Docker container created from the Docker test image. Instead of building the test image on each stage, we move the `docker.build()` instruction outside the stage to speed up the pipeline execution time, as you can see in the following listing.

Listing 8.6 Movie parser's Jenkinsfile

```
def imageName = 'mlabouardy/movies-parser'
node('workers'){
    stage('Checkout'){
        checkout scm
   }

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

```
        imageTest.inside{
            sh 'go test'
        }
    }
}
```

Push the changes to the develop branch, and the pipeline should be triggered to execute the three stages defined on the Jenkinsfile, as shown in figure 8.9.

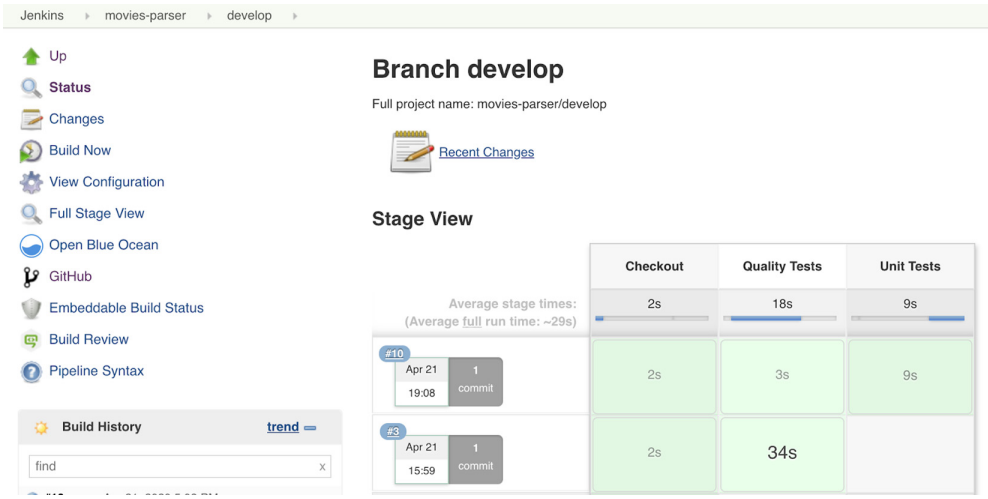


Figure 8.9 Go CI pipeline

The `go test` command output is shown in figure 8.10.

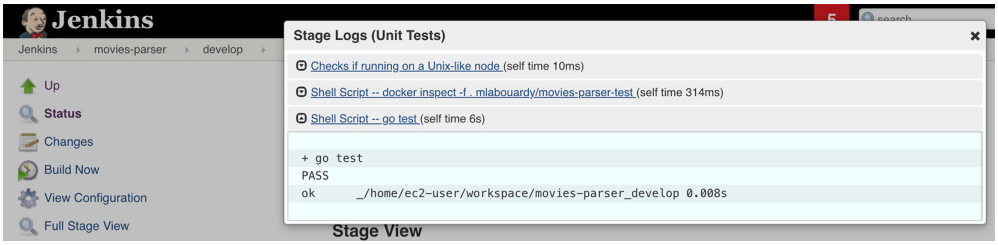


Figure 8.10 `go test` command output

> **NOTE** Go provides the `-cover` flag to the `go test` command as a built-in functionality to check your code coverage.

If we want to get the coverage report in HTML format, you need to add the following command:

```
go test -coverprofile=cover/cover.cov
go tool cover -html=cover/coverage.cov -o coverage.html
```

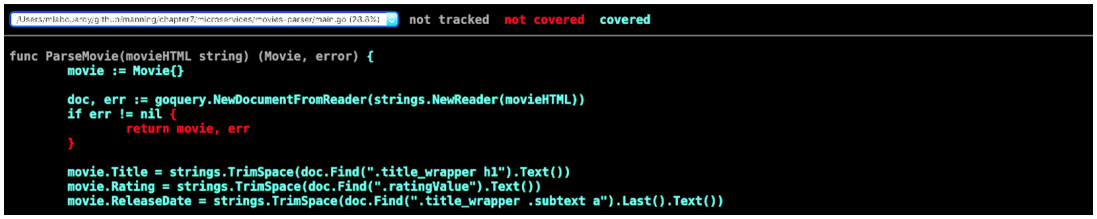


Figure 8.11 The coverage.html content can be served from the Jenkins dashboard at the end of the test stage.

The commands render an HTML page, shown in figure 8.11, that visualizes line-by-line coverage of each affected line in the main.go file.

You can include the previous command in the CI workflow to generate coverage reports in HTML format.

8.4 Injecting security in the CI pipeline

It's important to make sure that no vulnerabilities are published to production—at least no critical or major ones. Scanning project dependencies within a CI pipeline can ensure this additional level of security. Several dependency scanning solutions exist, commercial and open source. In this part, we'll go with Nancy.

Nancy (https://github.com/sonatype-nexus-community/nancy) is an open source tool that checks for vulnerabilities in your Go dependencies. It uses Sonatype's OSS Index (https://ossindex.sonatype.org/), a mirror of the Common Vulnerabilities and Exposures (CVE) database, to check your dependencies for publicly filed vulnerabilities.

> **NOTE** Chapter 9 covers how to use the OWASP Dependency-Check plugin on Jenkins to detect references to dependencies that have been assigned CVE entries.

Step one in the process is to install a Nancy binary from the official release page. Update Dockerfile.test for the movies-parser project to install Nancy version 1.0.22 (at the time of writing this book) and configure the executable on the `PATH` variable, as shown in the following listing.

Listing 8.7 Movie parser's Dockerfile.test

```
FROM golang:1.13.4
ENV VERSION 1.0.22
ENV GOCACHE /tmp
WORKDIR /go/src/github/mlabouardy/movies-parser
RUN wget https://github.com/sonatype-nexus-community/nancy/releases/download/
     $VERSION/nancy
```

```
linux.amd64-$VERSION -O nancy && \
    chmod +x nancy && mv nancy /usr/local/bin/nancy
RUN go get -u golang.org/x/lint/golint
COPY . .
RUN go get -v
```

To start using the tool, add a `Security Tests` stage on the Jenkinsfile to run Nancy with the Gopkg.lock file as parameter, which contains a list of used Go dependencies in the movies-parser service:

```
stage('Security Tests'){
        imageTest.inside('-u root:root'){
           sh 'nancy /go/src/github/mlabouardy/movies-parser/Gopkg.lock'
        }
}
```

Push the changes to the remote repository. A new pipeline will be started. At the `Security Tests` stage, Nancy will be executed, and no dependency security vulnerability will be reported, as shown in figure 8.12.

```
+ nancy /go/src/github/mlabouardy/movies-parser/Gopkg.lock
 __  __
/\ \/\ \
\ \ `\\ \      __      ___     ___    __  __
 \ \ , ` \   /'__`\   /' _ `\   /'___\ /\ \/\ \
  \ \ \`\ \ /\ \L\.\_ /\ \/\ \ /\ \__/ \ \ \_\ \
   \ \_\ \_\\ \__/.\_\\ \_\ \_\\ \____\ \/`____ \
    \/_/\/_/ \/__/\/_/ \/_/\/_/ \/____/  `/___/> \
                                           /\___/
                                           \/__/
  _       _                          _    _
 /_)     /_` _  _  _ _/_     _  _    (/   /_` _ . _  _    _/  _
/_) /_/  ._/ /_// //_|/  /_/ /_//'  (_X  /   / / /_'/ //_/ _\
    _/                 _/ /
Nancy version: 0.2.0
[31m!!!!! WARNING !!!!!
Scanning cannot be completed on the following package(s) since they do not use semver.
[0m[1/9] [1mpkg:golang/github.com/golang/snappy@master [0m
[2/9] [1mpkg:golang/github.com/jmespath/go-jmespath@c2b33e84 [0m
[3/9] [1mpkg:golang/github.com/xdg/scram@master [0m
[4/9] [1mpkg:golang/github.com/xdg/stringprep@master [0m
[5/9] [1mpkg:golang/golang.org/x/crypto@master [0m
[6/9] [1mpkg:golang/golang.org/x/lint@master [0m
[7/9] [1mpkg:golang/golang.org/x/net@master [0m
[8/9] [1mpkg:golang/golang.org/x/sync@master [0m
[9/9] [1mpkg:golang/golang.org/x/tools@master [0m

[1/8] [1mpkg:golang/github.com/PuerkitoBio/goquery@1.5.1 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[2/8] [1mpkg:golang/github.com/andybalholm/cascadia@1.1.0 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[3/8] [1mpkg:golang/github.com/aws/aws-sdk-go-v2@0.20.0 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[4/8] [1mpkg:golang/github.com/go-stack/stack@1.8.0 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[5/8] [1mpkg:golang/github.com/klauspost/compress@1.10.4 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[6/8] [1mpkg:golang/github.com/pkg/errors@0.9.1 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[7/8] [1mpkg:golang/go.mongodb.org/mongo-driver@1.3.2 [0m [38;5;251m   No known vulnerabilities against package/version
[0m[8/8] [1mpkg:golang/golang.org/x/text@0.3.2 [0m [38;5;251m   No known vulnerabilities against package/version
[0m
Audited dependencies:8,Vulnerable: [1;31m0 [0m
```

Figure 8.12 Dependencies scanning for known vulnerabilities

If Nancy finds a vulnerability in one of your dependencies, it will exit with a nonzero code, allowing you to use Nancy as a tool in your CI/CD process, and fail builds.

While you should aim to resolve all security vulnerabilities, some security scan results may contain false positives. For example, if you see a theoretical denial-of-service attack under obscure conditions that don't apply to your project, it may be safe to schedule a fix a week or two into the future. On the other hand, a more serious vulnerability that may grant unauthorized access to customer credit card data should be fixed immediately. Whatever the case, arm yourself with knowledge of the vulnerability so you and your team can determine the proper course of action to mitigate the security threat.

Adding the dependency scanning to your pipeline (figure 8.13) is a simple first step to reduce your attack surface. This is easy to implement, as it requires no server reconfigurations or additional servers to work. In its most basic form, simply install the Nancy binary and roll it out.

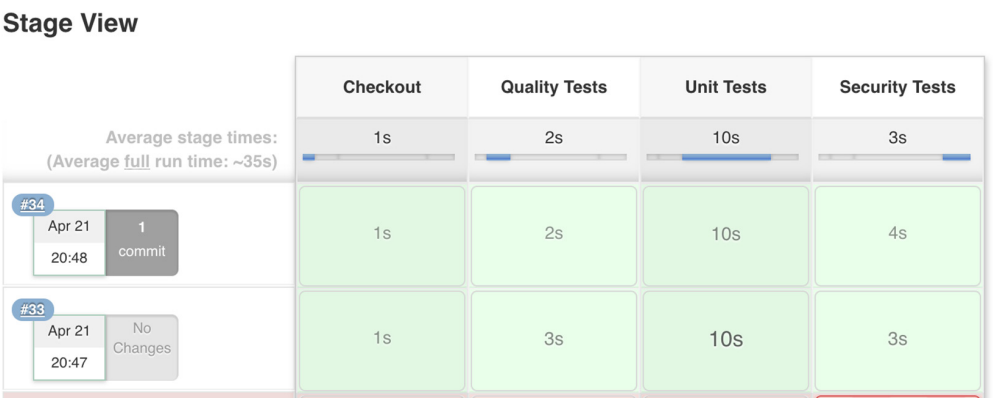


Figure 8.13 Security injection in CI pipeline

8.5 Running parallel tests with Jenkins

So far, pre-integration tests are running sequentially. One problem we always encounter is how to run all the tests needed to ensure high-quality changes while still keeping pipeline times reasonable and changes flowing smoothly. More tests mean greater confidence, but also longer wait times.

> **NOTE** In chapter 9, we will cover how to use the Parallel Test Execution plugin to run tests in parallel across multiple Jenkins workers.

One of the features of Jenkins pipelines that you see advertised quite frequently is its ability to run parts of your build in parallel by using the `parallel` DSL step.

Update the Jenkinsfile to use the `parallel` keyword, as shown in the following listing. The `parallel` section contains a list of nested test stages to be run in parallel.

Also, you can force your parallel stages to all be aborted when any one of them fails, by adding a `failFast true` instruction.

Listing 8.8 Running tests in parallel

```
node('workers'){
    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 Gopkg.lock'
                    }
                }
            )
    }
}
```

If you push those changes to the remote repository, a new build will be invoked (figure 8.14). However, one disadvantage of the standard pipeline view is that you can't easily see how the parallel steps progress, because the pipeline is linear, like a pipeline. This issue has been addressed by Jenkins by providing an alternate view: Blue Ocean.

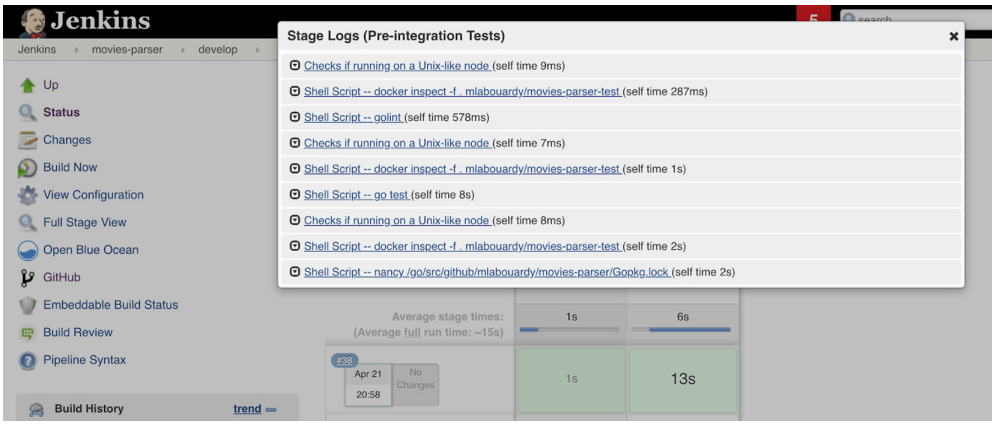


Figure 8.14 Pre-integration tests' parallel execution

Figure 8.15 shows the results for the same pipeline, with parallel test execution in Blue Ocean mode.

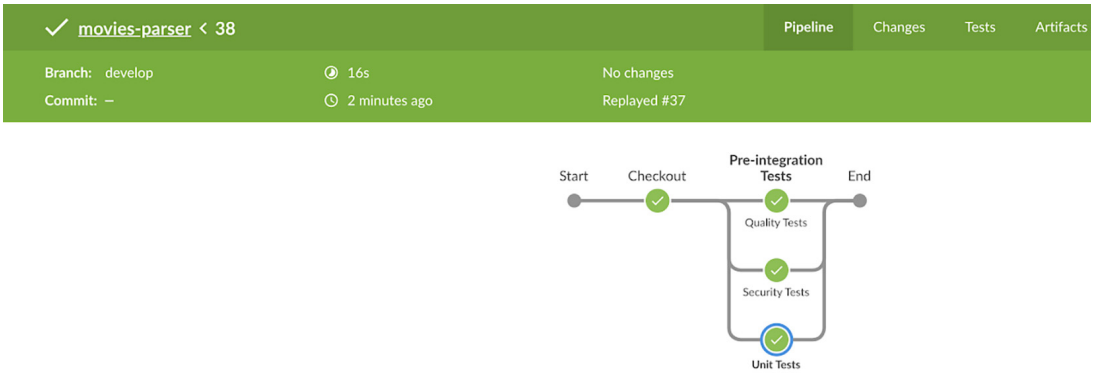


Figure 8.15 Parallel stages in Blue Ocean

This looks nice and provides great visualization for parallel pipeline stages.

8.6 Improving quality with code analysis

Apart from continuously integrating code, CI pipelines nowadays also include tasks that perform continuous inspection—inspecting code for its quality in a continuous approach.

The movies-store application is written with TypeScript. We will use Dockerfile.test to build the Docker image to run automated tests, as shown in the following listing.

Listing 8.9 Movie store's Dockerfile.test

```
FROM node:14.0.0
WORKDIR /app
COPY package-lock.json .
COPY package.json .
RUN npm i
COPY . .
```

The first category of tests will be linting the source code. As you saw earlier in this chapter, linting is the process of checking the source code for programmatic, syntactic, stylistic errors. Linting puts the whole service in a uniform format. The code linting can be achieved by writing some rules. Many linters are available, including JSLint, JSHint, and ESLint.

When it comes to linting TypeScript code, ESLint (https://eslint.org/) has a higher-performing architecture than others. For that reason, I'm using ESLint for linting the Node.js project, as shown in the following listing.

Listing 8.10 Movie store's Jenkinsfile

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

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

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

    stage('Quality Tests'){
        imageTest.inside{
            sh 'npm run lint'
         }
    }
}
```

Copy this content to the movies-store Jenkinsfile and push the changes to the develop branch. A new build should be triggered. At the `Quality Tests` stage, we'll see the errors regarding undefined keywords (figure 8.16) such as `describe` and `before`, which are part of the Mocha (https://mochajs.org/) and Chai (www.chaijs.com) JavaScript frameworks. These frameworks are used to describe unit tests (located under the test folder) efficiently and handily.

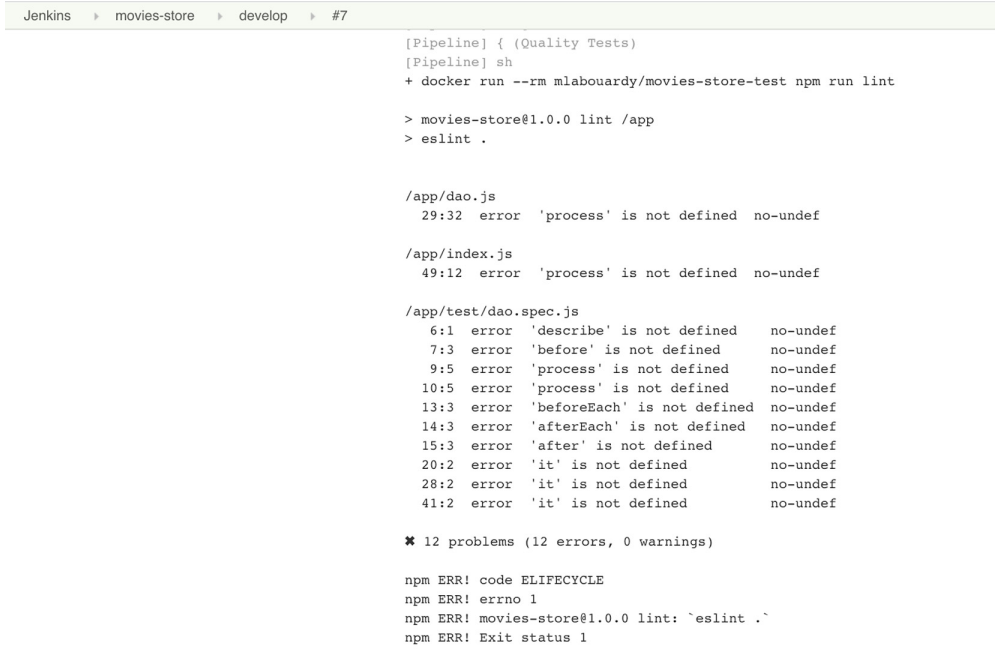


Figure 8.16 ESLint problem detection

ESLint will return an exit 1 code error, which will break the pipeline. To fix the spotted errors, extend ESLint rules by enabling the Mocha environment for ESLint. We use the `key` attribute in eslintrc.json to specify the environments we want to enable by setting `mocha` to `true`:

```
{
    "env": {
        "node": true,
        "commonjs": true,
        "es6": true,
        "mocha": true
    },

}
```

If you push the changes, this time the static code analysis results will be successful, as you can see in figure 8.17.

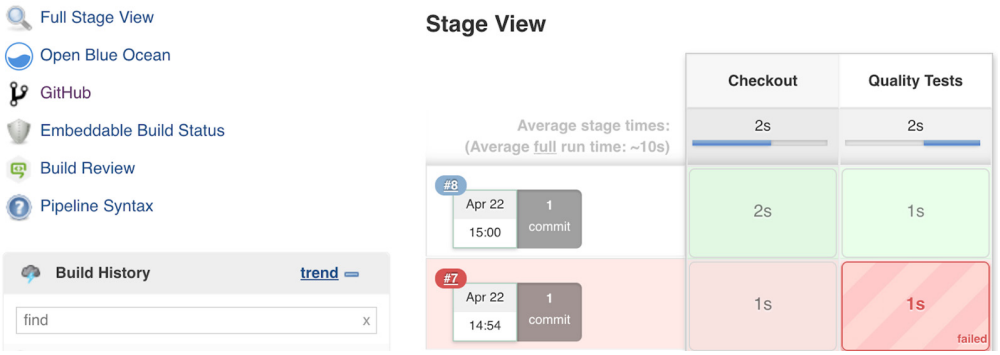


Figure 8.17 CI pipeline execution after fixing ESLint errors

8.7 Running mocked database tests

While many developers focus on 100% coverage with unit tests, the code you write must not be tested just in isolation. Integration and end-to-end tests give you that extra confidence by testing parts of your application together. These parts may be working just fine on their own, but in a large system, units of code rarely work separately.

Typically, for integration or end-to-end tests, your scripts will need to connect to a real, dedicated database for testing purposes. This involves writing code that runs at the beginning and end of every test case/suite to ensure that the database is in a clean, predictable state.

Using a real database for testing does have some challenges: database operations can be relatively slow, the testing environment can be complex, and operational

overhead may increase. Java projects widely use DbUnit with an in-memory database for this purpose (for example, H2, www.h2database.com/html/main.html). Reusing a good solution from another platform and applying it to the Node.js world can be the way to go here.

Mongo-unit (www.npmjs.com/package/mongo-unit) is a Node.js package that can be installed by using Node Package Manager (npm) or Yarn. It runs MongoDB in memory. It makes integration tests easy by integrating well with the Mocha framework and providing a simple API to manage the database state.

> **NOTE** In chapter 9 and 10, we will run sidecar containers in Jenkins pipelines, such as a MongoDB database, to run end-to-end tests.

The following listing is a simple test (/chapter7/microservices/movies-store/test/dao.spec.js), written with Mocha and Chai, that uses the mongo-unit package to simulate MongoDB by running an in-memory database.

Listing 8.11 Mocha and Chai unit tests

```
const Expect = require('chai').expect
const MongoUnit = require('mongo-unit')
const DAO = require('../dao')
const TestData = require('./movies.json')

describe('StoreDAO', () => {
  before(() =>  MongoUnit.start().then(() => {
    process.env.MONGO_URI = MongoUnit.getUrl()
    DAO.init()
  }))
  beforeEach(() => MongoUnit.load(TestData))
  afterEach(() => MongoUnit.drop())
  after(() => {
    DAO.close()
    return MongoUnit.stop()
  })
 it('should find all movies', () => {
   return DAO.Movie.find()
    .then(movies => {
      Expect(movies.length).to.equal(8)
      Expect(movies[0].title).to.equal('Pulp Fiction (1994)')
    })
 })
})
```

Next, we update the Jenkinsfile to add a new stage that executes the `npm run test` command:

```
stage('Integration Tests'){
        sh "docker run --rm ${imageName}-test npm run test"
}
```

The `npm run test` command is an alias; it runs the Mocha command line against test cases in the test folder (figure 8.18). The command is defined in package.json, provided in the following listing.

Listing 8.12 Movie store's package.json

```
"scripts": {
    "start": "node index.js",
    "test": "mocha ./test/*.spec.js",
    "lint": "eslint .",
    "coverage-text": "nyc --reporter=text mocha",
    "coverage-html": "nyc --reporter=html mocha"
}
```

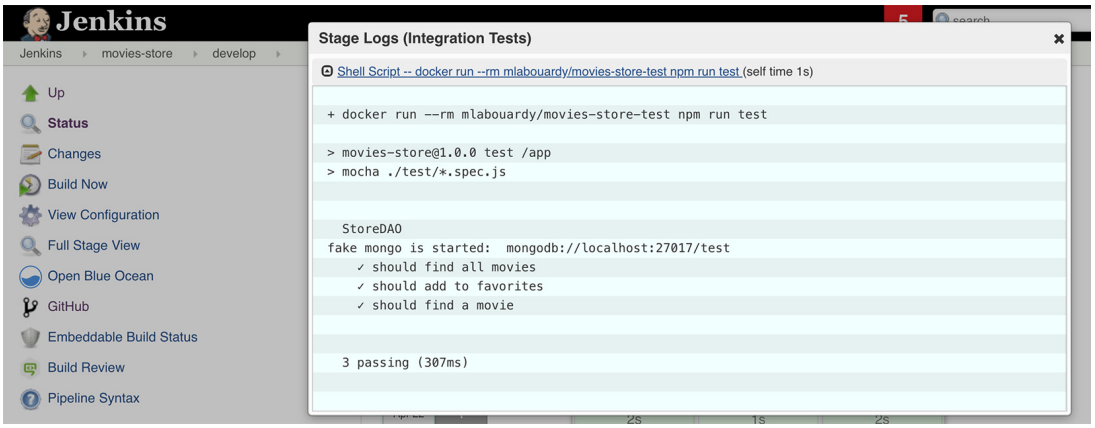


Figure 8.18 Unit testing using the Mocha framework

> **NOTE** If your tests depend on other services, Docker Compose can be used to simplify the startup and connection of all the services that the application depends on.

8.8 Generating HTML coverage reports

We create a new stage to run the coverage tool with a text output format:

```
stage('Coverage Reports'){
        sh "docker run --rm ${imageName}-test npm run coverage-text"
}
```

This will output the text report to the console output, as shown in figure 8.19.

> **NOTE** Istanbul is a JavaScript code coverage tool. For more information, refer to the official guide at https://istanbul.js.org.

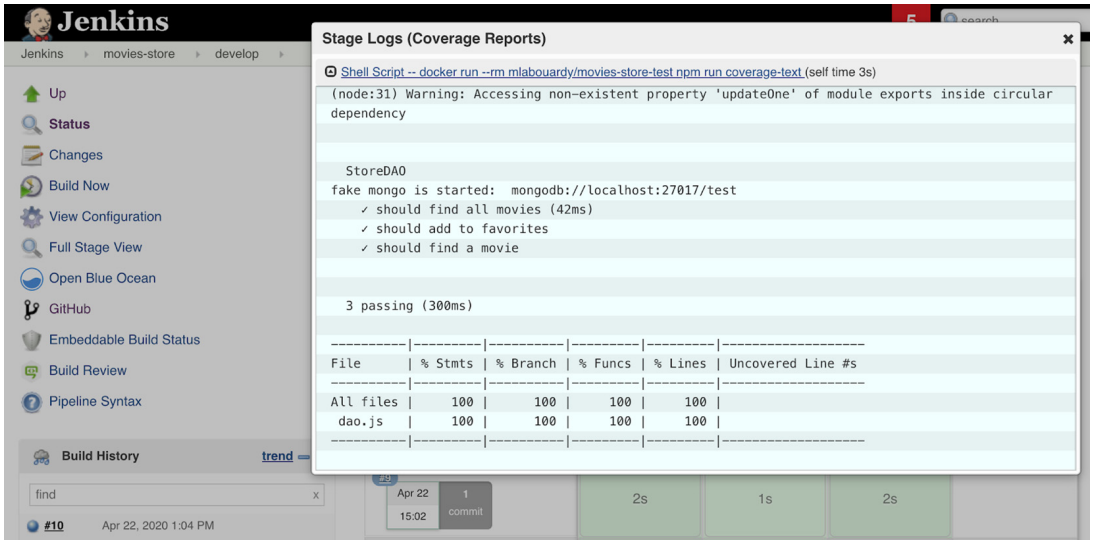


Figure 8.19 Istanbul coverage reports in text format

The metrics that you might see in your coverage reports could be defined as in table 8.1.

Table 8.1 Coverage report metrics

Metric	Description
Statements	The number of statements in the program that are truly called, out of the total number
Branches	The number of branches of the control structures executed
Functions	The number of functions called, out of the total number of functions defined
Lines	The number of lines of source code that are being tested, out of the total number of lines present inside the code

By default, Istanbul uses a text reporter, but various other reporters are available. You can view the full list at http://mng.bz/DKoE.

To generate the HTML format, we will map a volume to /app/coverage, which is the folder in which Istanbul will generate the reports. Then, we'll use the Jenkins HTML Publisher plugin to display the generated code coverage reports, as shown in the following listing.

Listing 8.13 Publishing code coverage HTML reports

```
stage('Coverage Reports'){
        sh "docker run --rm
-v $PWD/coverage:/app/coverage ${imageName}-test
```

```
npm run coverage-html"
        publishHTML (target: [
          allowMissing: false,
          alwaysLinkToLastBuild: false,
          keepAll: true,
          reportDir: "$PWD/coverage",
          reportFiles: "index.html",
          reportName: "Coverage Report"
        ])
}
```

The `publishHTML` command takes the `target` block as the main parameter. Within that, we have several subparameters. The `allowMissing` parameter is set to `false`, so if something goes wrong while generating the coverage report and the report is missing, the `publishHTML` instruction will throw an error.

At the end of the CI pipeline, an HTML file will be generated and consumed by the HTML Publisher plugin, as shown in figure 8.20.

```
[Pipeline] publishHTML
[htmlpublisher] Archiving HTML reports...
[htmlpublisher] Archiving at BUILD level /home/ec2-user/coverage to /var/lib/jenkins/jobs/movies-
store/branches/develop/builds/16/htmlreports/Coverage_20Report
```

Figure 8.20 HTML report generation with Istanbul

The HTML report will then be accessible from Jenkins, by clicking the Coverage Report item from the left panel; see figure 8.21.

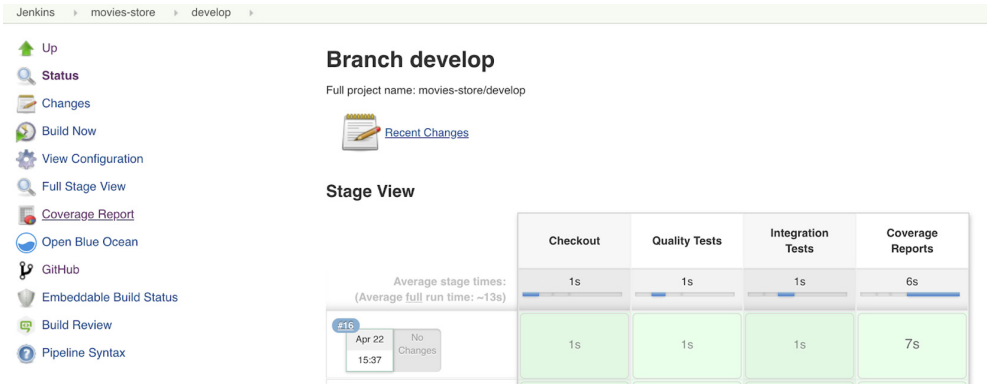


Figure 8.21 The coverage report can be accessible from the Jenkins panel.

NOTE The Cobertura plugin (https://plugins.jenkins.io/cobertura/) can also be used to publish HTML reports. Both plugins show the same results.

We can drill down to identify the uncovered lines and functions, as shown in figure 8.22.

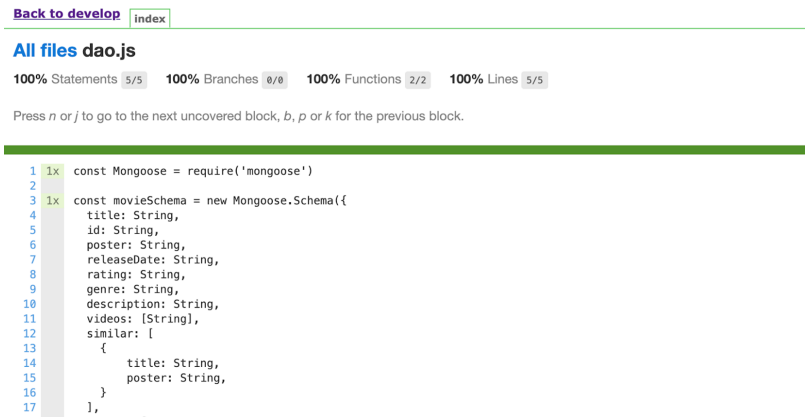


Figure 8.22 Deep dive inside the coverage report

> **NOTE** Several tools exist to create coverage reports, depending on the language you use (for example, SimpleCov for Ruby, Coverage.py for Python, and JaCoCo for Java).

You can take this further and run stages in parallel to reduce the waiting time of running tests, as shown in the following listing.

Listing 8.14 Running pre-integration tests in parallel

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

Figure 8.23 shows the end result of running this job in the Blue Ocean view.

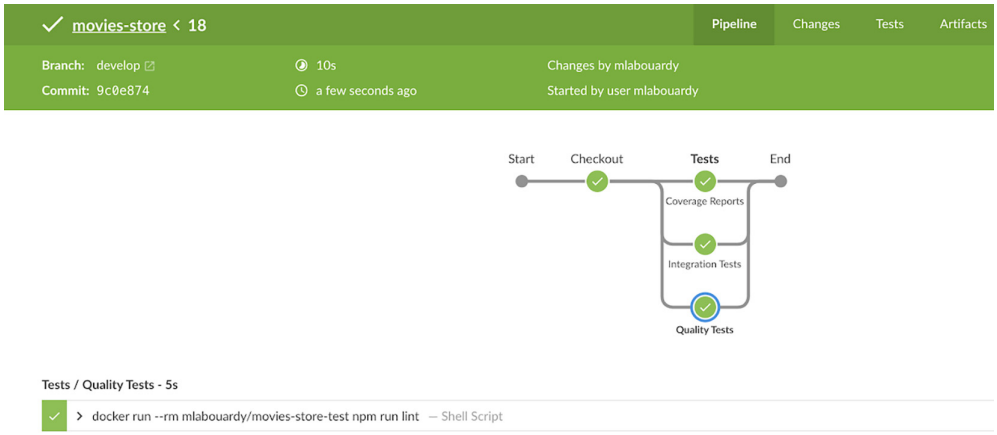


Figure 8.23 Running tests in parallel

8.9 Automating UI testing with Headless Chrome

For the Angular application, we will create a Dockerfile.test file that installs the Angular CLI (https://angular.io/cli) and the needed dependencies to run automated tests; see the following listing.

Listing 8.15 Movie marketplace's Dockerfile.test

```
FROM node:14.0.0
ENV CHROME_BIN=chromium
WORKDIR /app
COPY package-lock.json .
COPY package.json .
RUN npm i && npm i -g @angular/cli
COPY . .
```

The linting state is similar to the previous part; we will use the TSLint linter, which comes installed by default for Angular projects. Hence, we will run the `npm run lint` alias command defined in package.json, as shown in the following listing.

Listing 8.16 Movie marketplace's package.json

```
"scripts": {
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test --browsers=ChromeHeadlessCI --code-coverage=true",
    "lint": "ng lint",
    "e2e": "ng e2e"
  }
```

We update the Jenkinsfile with the following content.

Listing 8.17 Movie marketplace's Jenkinsfile

```
def imageName = 'mlabouardy/movies-marketplace'
node('workers'){
    stage('Checkout'){
        checkout scm
    }

    def imageTest= docker.build("${imageName}-test", "-f Dockerfile.test .")
    stage('Pre-integration Tests'){
        parallel(
            'Quality Tests': {
                sh "docker run --rm ${imageName}-test npm run lint"
            }
        )
    }
}
```

Let's save this config and run a build. The pipeline should fail and turn red because of the forced rules on TSLint, as shown in figure 8.24.

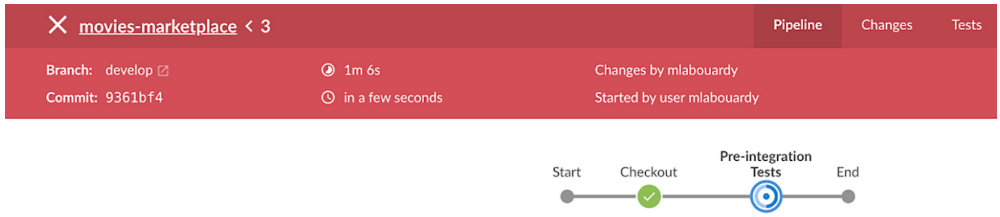


Figure 8.24 CI pipeline failure

If you click the Quality Tests stage logs, the logs should display errors regarding missing semicolons and trailing whitespace, as shown in figure 8.25.

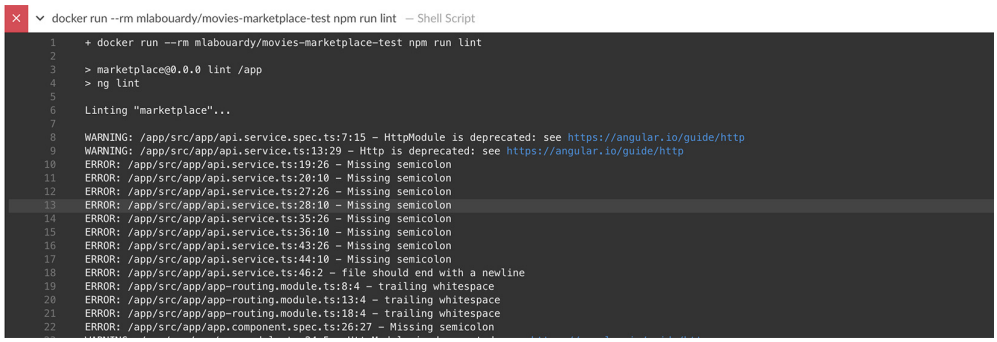


Figure 8.25 Angular linting output logs

If you wish to let TSLint pass within your code (figure 8.26), you need to update tslint.json to disable forced rules or add the `/* tslint:disable */` instruction at the beginning of each file for TSLint to skip the linting process on those files.

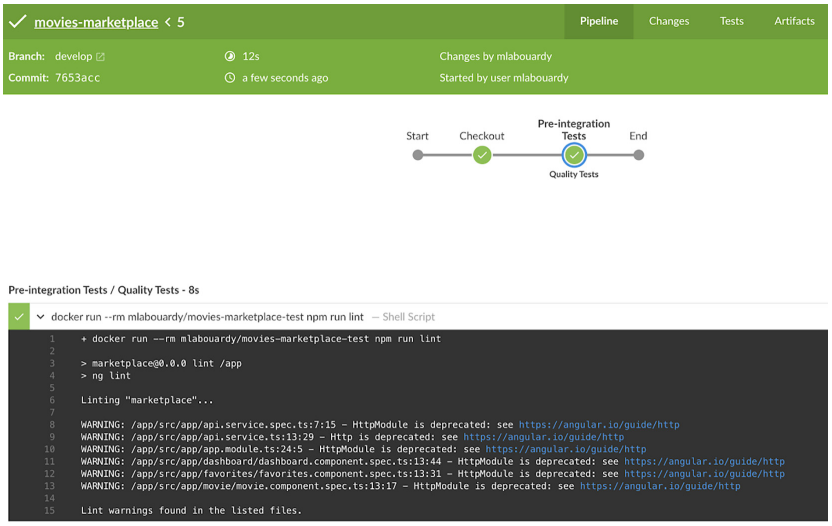


Figure 8.26 Angular linting output logs

For Angular unit testing, we will use the Jasmine (https://jasmine.github.io/) and Karma (https://karma-runner.github.io/latest/index.html) frameworks. Both testing frameworks support the BDD practice, which describes tests in a human-readable format for nontechnical people. The sample unit test (chapter7/microservices/movies-marketplace/src/app/app.component.spec.ts) in the following listing is self-explanatory. It tests whether the app component has a property `text` with the value `Watchlist` that is rendered in the HTML inside a `span` element tag.

Listing 8.18 Movie marketplace's Karma tests

```
import { TestBed, async } from '@angular/core/testing';
import { RouterTestingModule } from '@angular/router/testing';
import { AppComponent } from './app.component';

describe('AppComponent', () => {
  beforeEach(async(() => {
    TestBed.configureTestingModule({
      imports: [
        RouterTestingModule
      ],
      declarations: [
        AppComponent
      ],
    }).compileComponents();
  }));
```

```
  it('should create the app', () => {
    const fixture = TestBed.createComponent(AppComponent);
    const app = fixture.debugElement.componentInstance;
    expect(app).toBeTruthy();
  });
  it('should render title', () => {
    const fixture = TestBed.createComponent(AppComponent);
    fixture.detectChanges();
    const compiled = fixture.debugElement.nativeElement;
    expect(compiled.querySelector('.toolbar
     span').textContent).toContain('Watchlist');
  });
});
```

NOTE When creating Angular projects with the Angular CLI, it defaults to creating and running unit tests by using Jasmine and Karma.

Running unit tests for frontend web applications requires them to be tested in a web browser. While it's not an issue on a workstation or host machine, it can become tedious when running in a restricted environment such as a Docker container. In fact, these execution environments are generally lightweight and do not contain any graphical environment.

Fortunately, Karma tests can be run with a UI-less browser, and two main options can be used: Chrome Headless or PhantomJS. The example in the following listing uses Chrome Headless with Puppeteer, which can be configured on a simple flag in the Karma config (chapter7/microservices/movies-marketplace/karma.conf.js).

Listing 8.19 Karma runner configuration

```
module.exports = function (config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine', '@angular-devkit/build-angular'],
    customLaunchers: {
      ChromeHeadlessCI: {
        base: 'Chrome',
        flags: [
          '--headless',
          '--disable-gpu',
          '--no-sandbox',
          '--remote-debugging-port=9222'
        ]
      }
    },
    browsers: ['ChromeHeadless', 'Chrome'],
    singleRun: true,  });
};
```

Headless Chrome needs sudo privileges to be run unless the `--no-sandbox` flag is used. Next, we need to update Dockerfile.test to install Chromium:

```
RUN apt-get update && apt-get install -y chromium
```

NOTE Chromium/Google Chrome has shipped with the headless mode since version 59.

Then, we update the Jenkinsfile to run unit tests with the `npm run test` command. The command will fire up Headless Chrome and execute Karma.js tests. Next, we generate a coverage report in HTML format that will be consumed by the HTML Publisher plugin, as shown in the following listing.

Listing 8.20 Mapping the workspace folder with the Docker container volume

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

Once changes are pushed to the GitHub repository, a new build will be triggered and unit tests will be executed, as shown in figure 8.27.

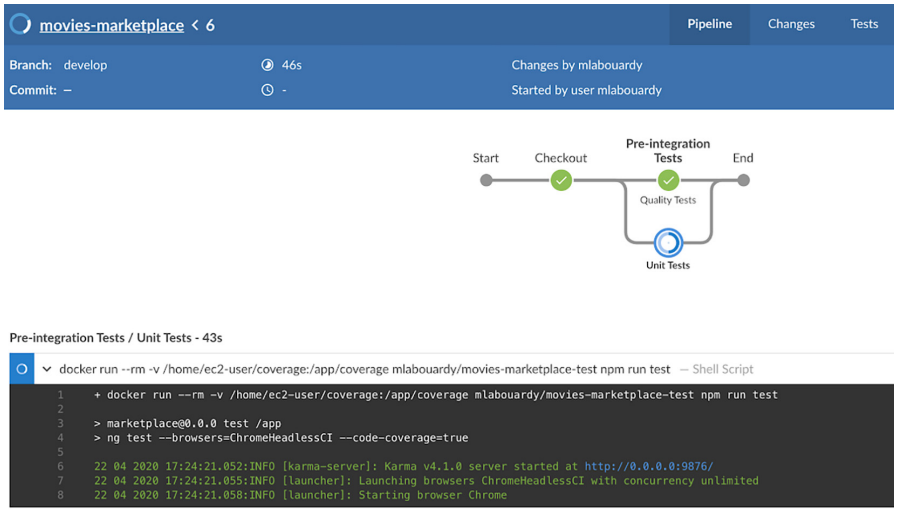


Figure 8.27 Running headless Chrome inside a Docker container

The Karma launcher will run the tests on the Headless Chrome browser and display the code coverage statistics, as shown in figure 8.28.

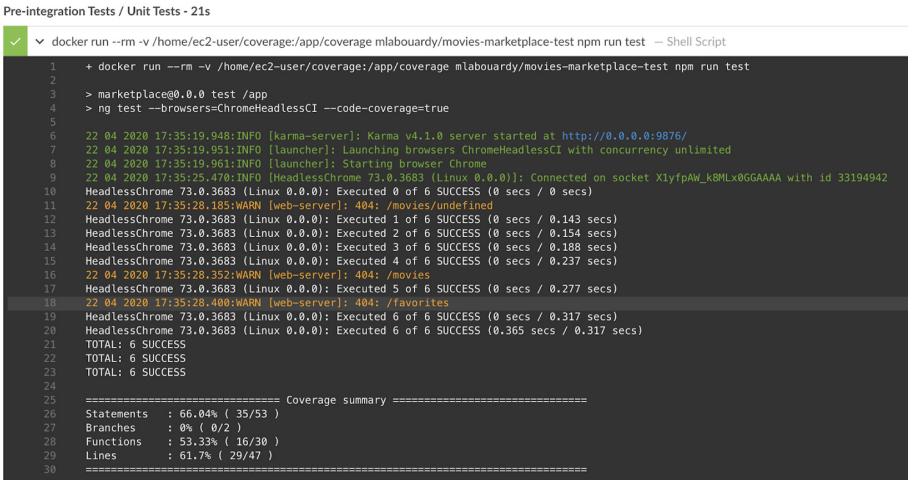


Figure 8.28 Successful execution of the Karma unit tests

Also, a generated HTML report will be available in the Artifacts section in the Blue Ocean view, shown in figure 8.29.

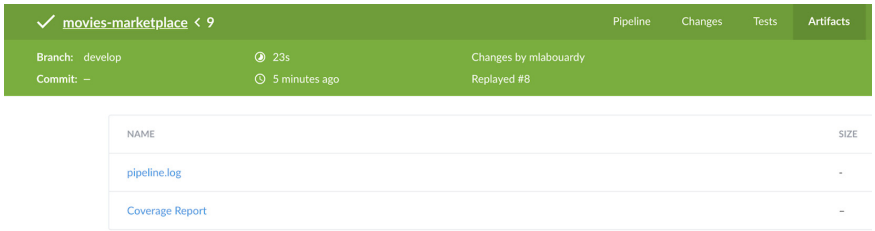
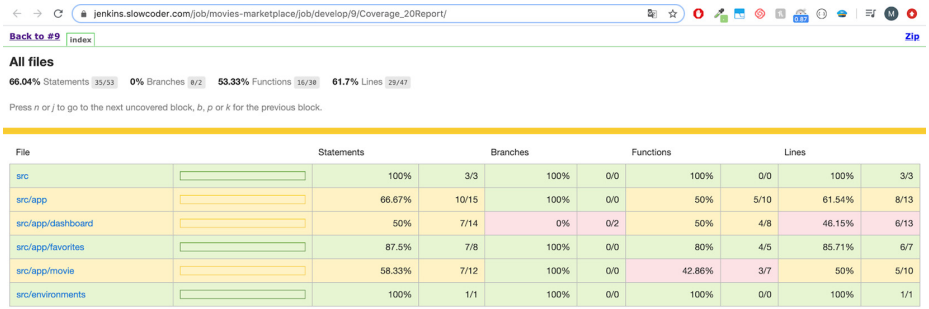


Figure 8.29 Coverage report alongside other artifacts

If you click the coverage report link, it should display the statements and functions coverage by Angular components and services, as shown in figure 8.30.



File	Statements		Branches		Functions		Lines	
src	100%	3/3	100%	0/0	100%	0/0	100%	3/3
src/app	66.67%	10/15	100%	0/0	50%	5/10	61.54%	8/13
src/app/dashboard	50%	7/14	0%	0/2	50%	4/8	46.15%	6/13
src/app/favorites	87.5%	7/8	100%	0/0	80%	4/5	85.71%	6/7
src/app/movie	58.33%	7/12	100%	0/0	42.86%	3/7	50%	5/10
src/environments	100%	1/1	100%	0/0	100%	0/0	100%	1/1

Figure 8.30 Coverage statistics by filename

With this done, it is now possible to run the unit tests with Chromium inside a Docker container.

8.10 Integrating SonarQube Scanner with Jenkins

While code linters can give you a high-level overview of the quality of your code, they're still limited if you want to perform deep static code analysis and inspection to detect potential bugs and vulnerabilities. That's where SonarQube comes into play. it will give you a 360-degree vision of the quality of the codebase by integrating external libraries like PMD, Checkstyle, and FindBugs. Every time code gets committed, code analysis is performed.

> **NOTE** SonarQube can be used to inspect code in more than 20 programming languages, including Java, PHP, Go, and Python.

To deploy SonarQube, we will bake a new AMI with Packer. Similarly to previous chapters, we create a template.json file with the content in the following listing (chapter8/sonarqube/packer/template.json).

Listing 8.21 Jenkins worker'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" : "ubuntu",
            "ami_name" : "sonarqube-8.2.0.32929",
            "ami_description" : "SonarQube community edition"
        }
    ],
    "provisioners" : [
        {
            "type" : "file",
            "source" : "sonar.init.d",
            "destination" : "/tmp/"
        },
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

The temporary EC2 instance will be based on Amazon Linux AMI and uses a shell script to provision the instance to install SonarQube and configure the needed dependencies.

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

Listing 8.22 Installing SonarQube LTS

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

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

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

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

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

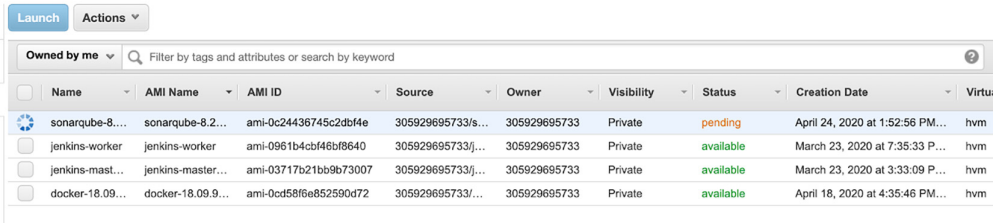


Figure 8.31 SonarQube machine image

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

Listing 8.23 SonarQube EC2 instance resource with Terraform

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

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

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

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

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

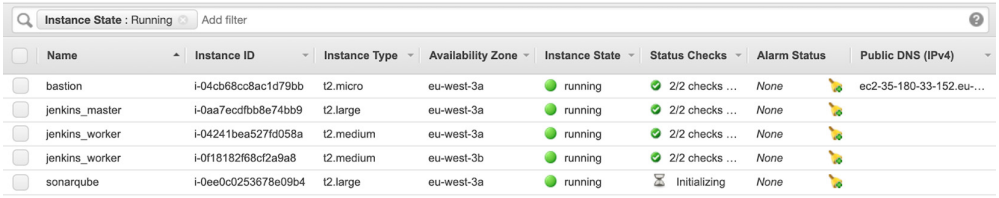


Figure 8.32 SonarQube private EC2 instance

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

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

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

Outputs:

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

Figure 8.33 SonarQube DNS URL

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

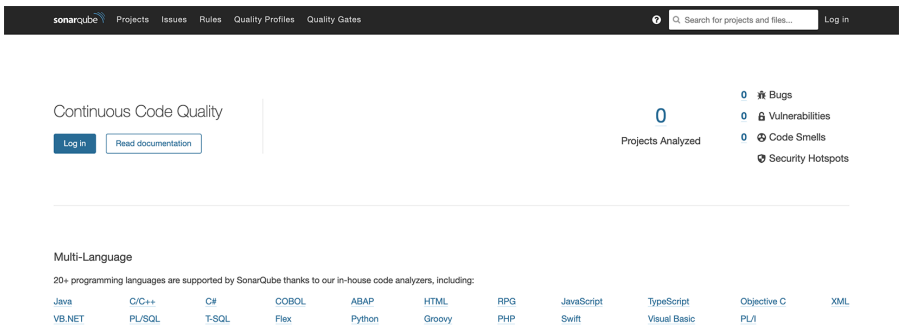


Figure 8.34 SonarQube web dashboard

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

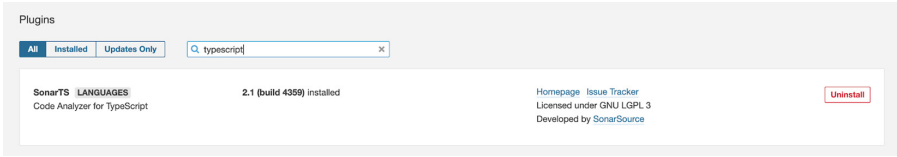


Figure 8.35 SonarQube TypeScript analyzer plugin

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

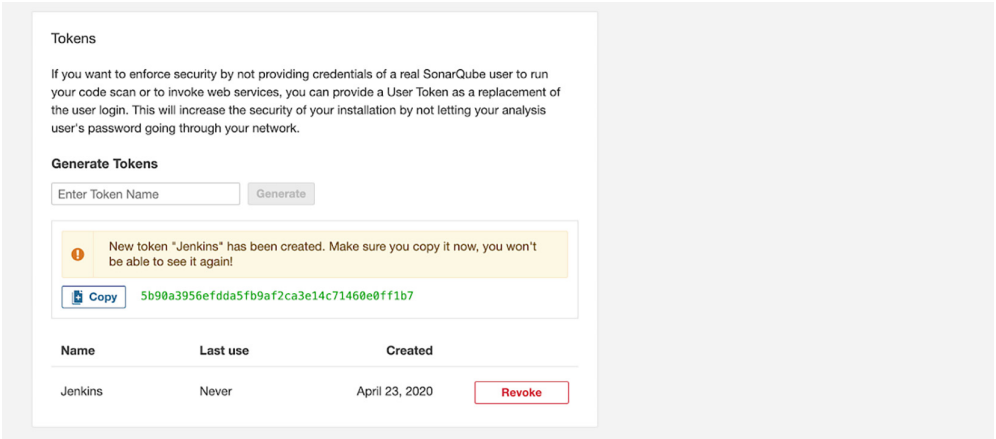


Figure 8.36 SonarQube Jenkins dedicated token

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

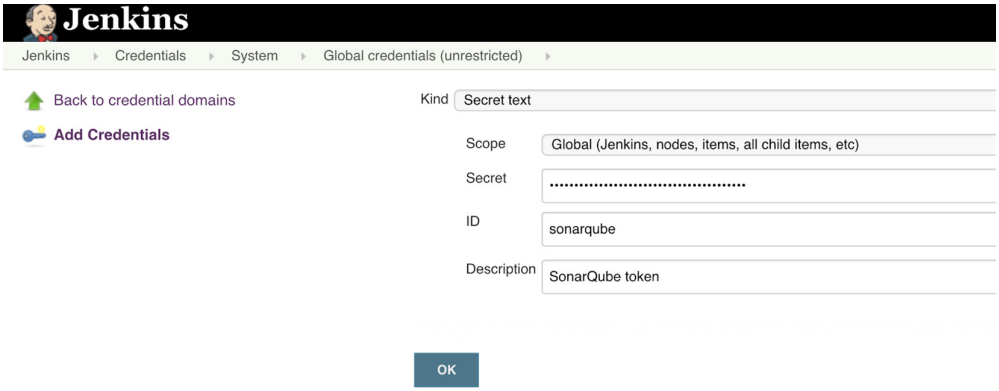


Figure 8.37 SonarQube secret text credentials

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

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

Listing 8.24 SonarQube Scanner installation

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

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

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

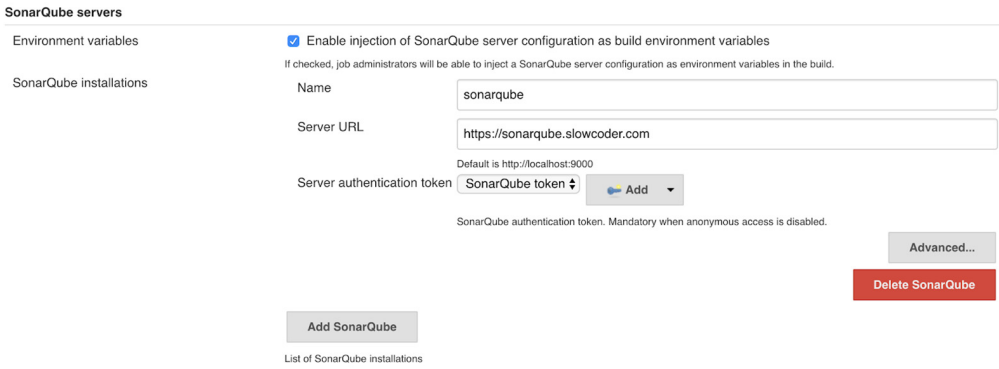


Figure 8.38 SonarQube server settings

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

Listing 8.25 SonarQube project configuration

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

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

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

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

Listing 8.26 Triggering SonarQube analysis

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

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

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

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

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

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

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

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

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

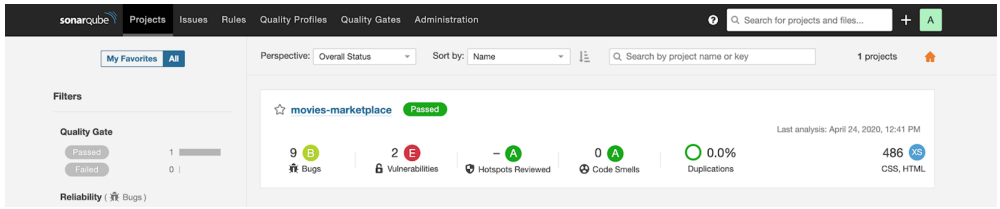


Figure 8.39 SonarQube project metrics

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

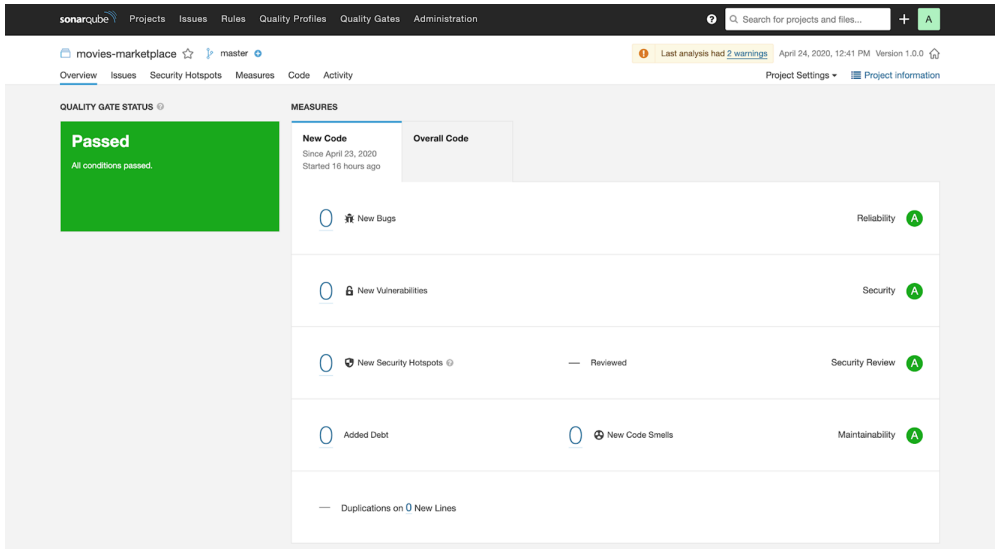


Figure 8.40 SonarQube project deep-dive metrics and issues

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

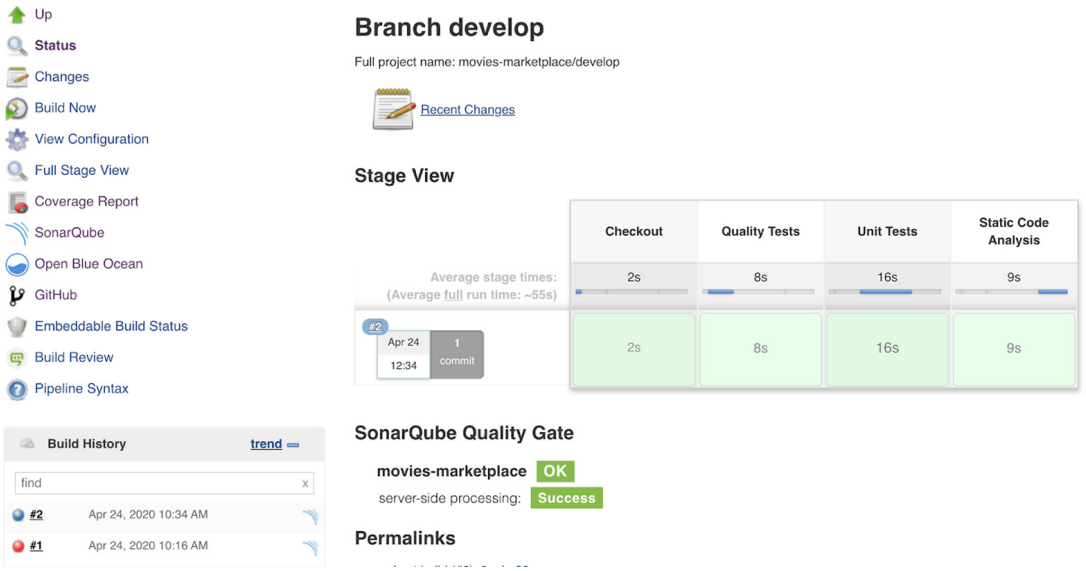


Figure 8.41 SonarQube integration with Jenkins

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

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

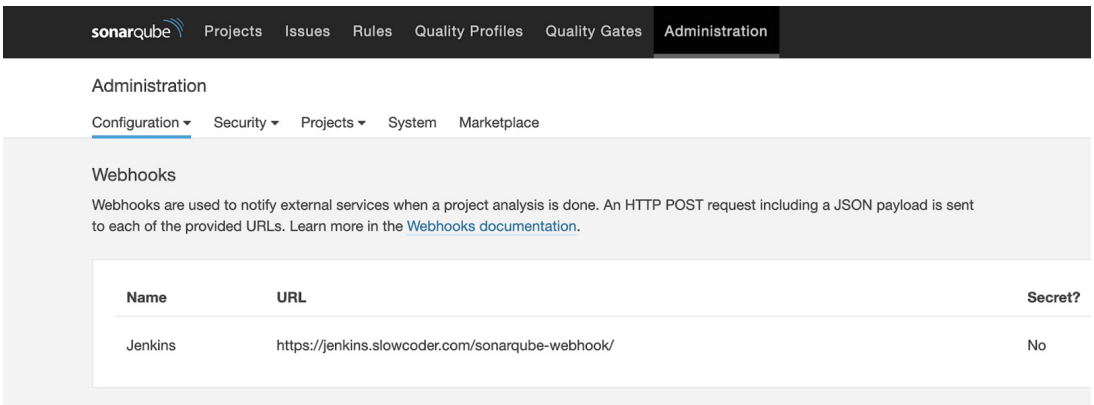


Figure 8.42 SonarQube webhook creation

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

Listing 8.27 Adding a quality gate to the Jenkinsfile

```
stage('Static Code Analysis'){
        withSonarQubeEnv('sonarqube') {
            sh 'sonar-scanner'
        }
}
stage("Quality Gate"){
        timeout(time: 5, unit: 'MINUTES') {
            def qg = waitForQualityGate()
            if (qg.status != 'OK') {
                error "Pipeline
aborted due to quality gate failure: ${qg.status}"
            }
        }
}
```

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

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

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

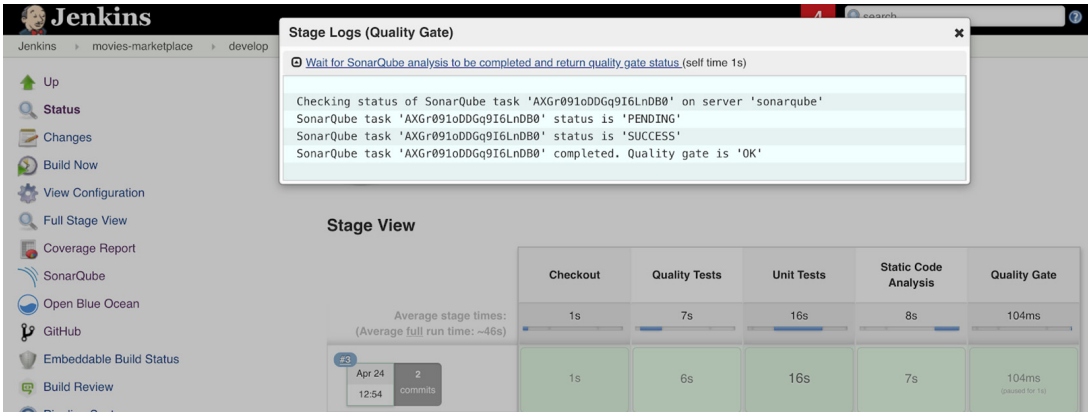


Figure 8.43 SonarQube project analysis status

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

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

Summary

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

9 Building Docker images within a CI pipeline

This chapter covers

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

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

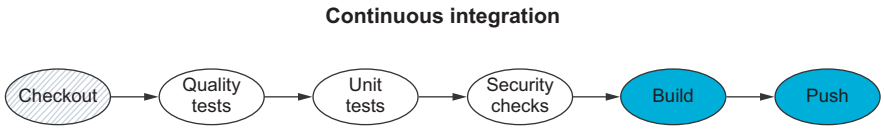


Figure 9.1 The Build and Push stages will be implemented in this chapter.

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

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

Figure 9.2 summarizes the end result of the CI workflow.

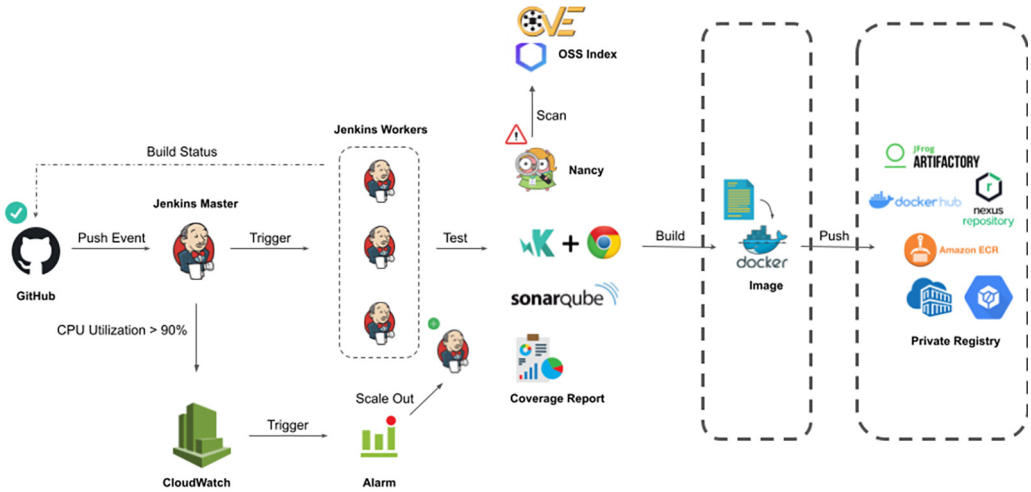


Figure 9.2 The CI pipeline process

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

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

9.1 Building Docker images

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

Figure 9.3 Current CI workflow

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

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

9.1.1 Using the Docker DSL

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

Listing 9.1 Movie loader's Dockerfile

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

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

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

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

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

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

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

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

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

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

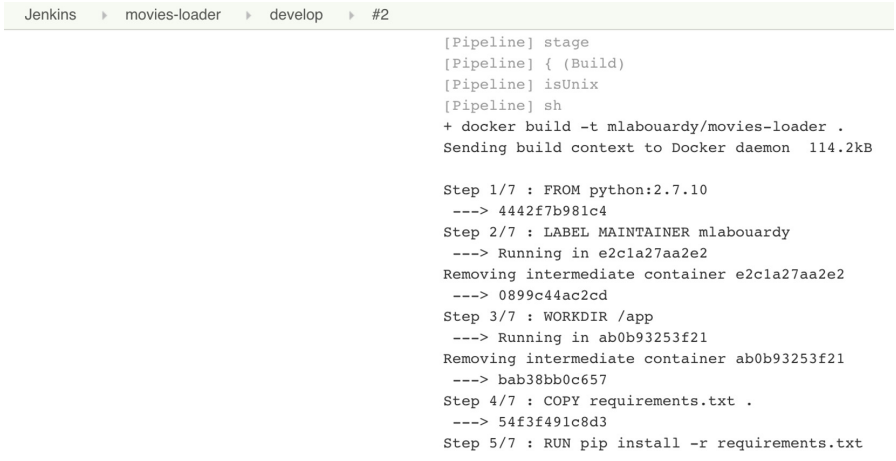


Figure 9.4 Python Docker image build logs

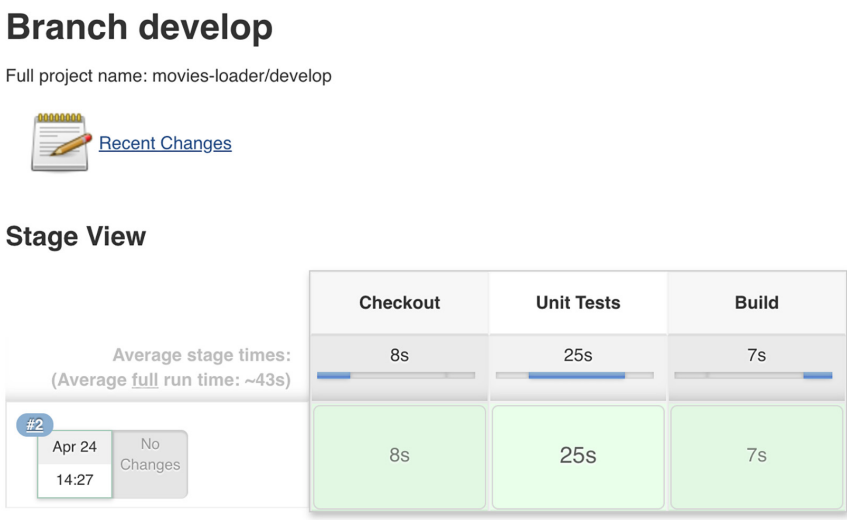


Figure 9.5 Movie loader CI pipeline

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

Listing 9.2 Multistage build usage

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

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

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

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

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

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

Listing 9.3 Naming Docker multistages

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

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

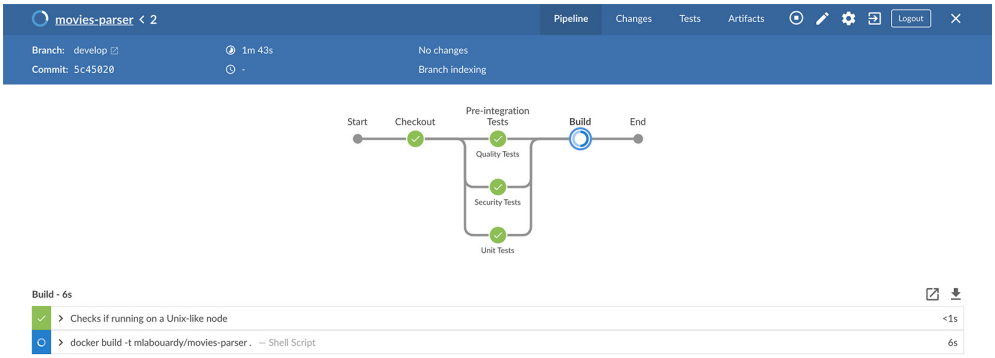


Figure 9.6 Movie parser CI pipeline

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

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

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

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

Listing 9.4 Movie store's Dockerfile

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

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

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

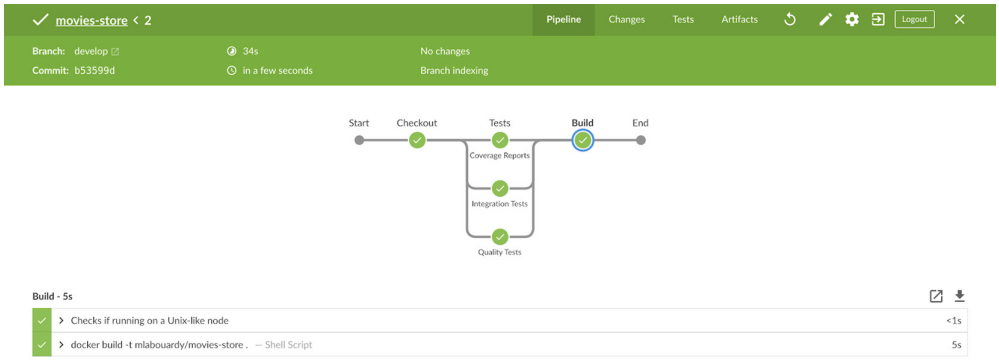


Figure 9.7 Movie store CI pipeline

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

9.1.2 Docker build arguments

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

Listing 9.5 Movie marketplace's Dockerfile

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

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

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

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

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

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

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

```
nodes_modules
coverage
dist
tmp
```

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

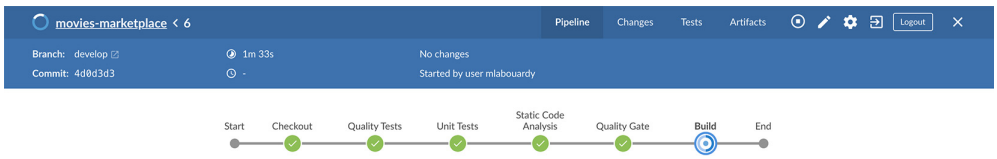


Figure 9.8 Movie marketplace CI pipeline

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

9.2 Deploying a Docker private registry

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

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

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

9.2.1 Nexus Repository OSS

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

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

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

Listing 9.6 Nexus Repository OSS Packer template

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

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

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

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

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

Defines Nexus OSS default credentials (admin/admin123)

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

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

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

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

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

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

Listing 9.9 Nexus OSS script API (setup.sh)

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

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

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

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

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

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

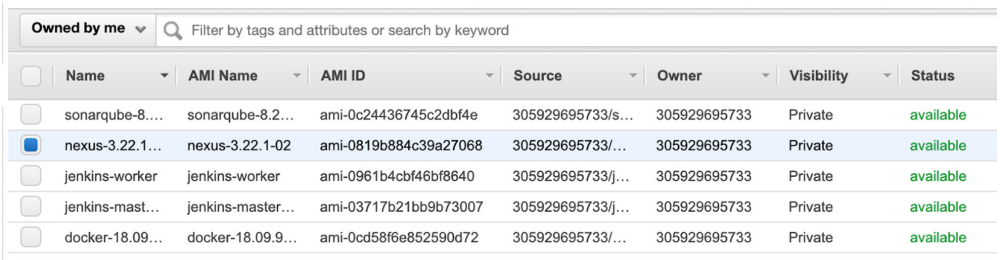


Figure 9.9 Nexus OSS AMI

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

Listing 9.10 Nexus EC2 instance resource

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

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

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

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

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

Listing 9.11 Nexus Repository Manager public load balancer

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

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

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

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

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

Listing 9.12 Docker registry public load balancer

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

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

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

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

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

Outputs:

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

Figure 9.10 Nexus Terraform resources

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

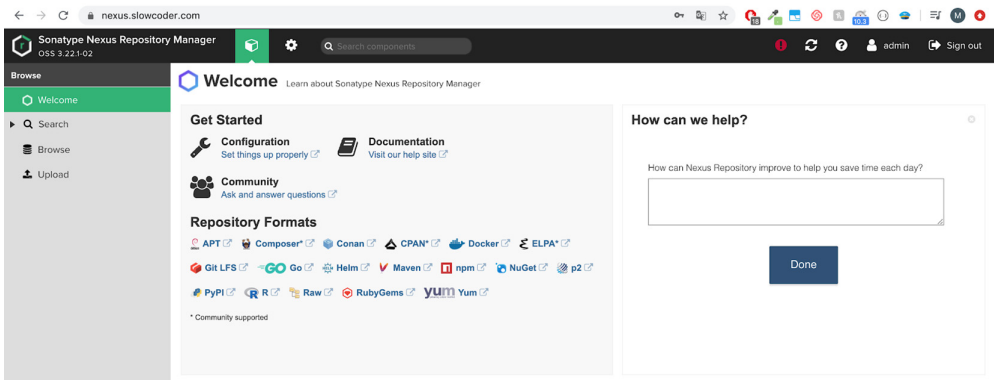


Figure 9.11 Nexus Repository Manager

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

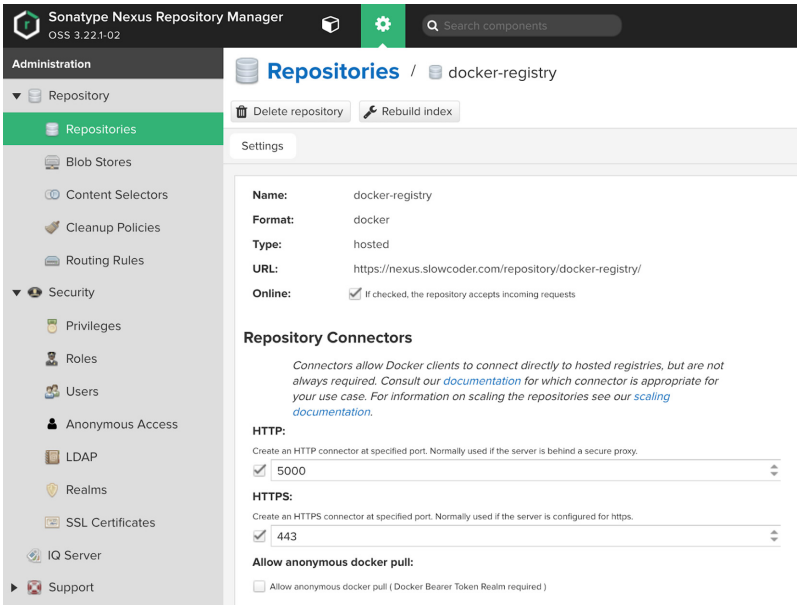


Figure 9.12 Docker-hosted registry on Nexus

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

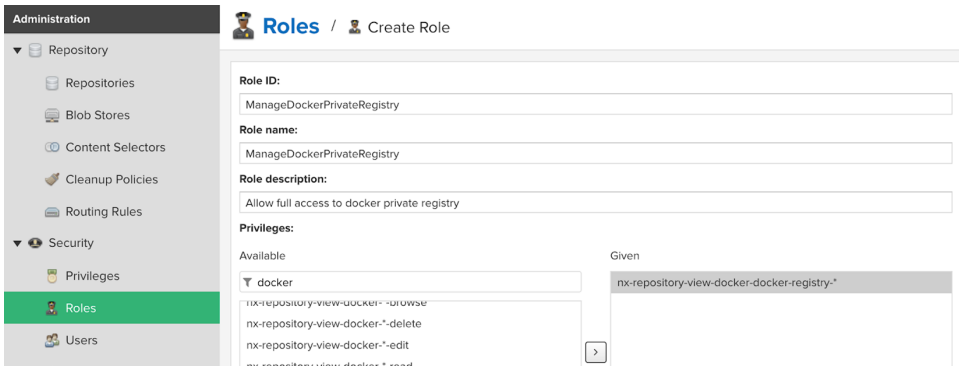


Figure 9.13 Nexus custom role for the Docker registry

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

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

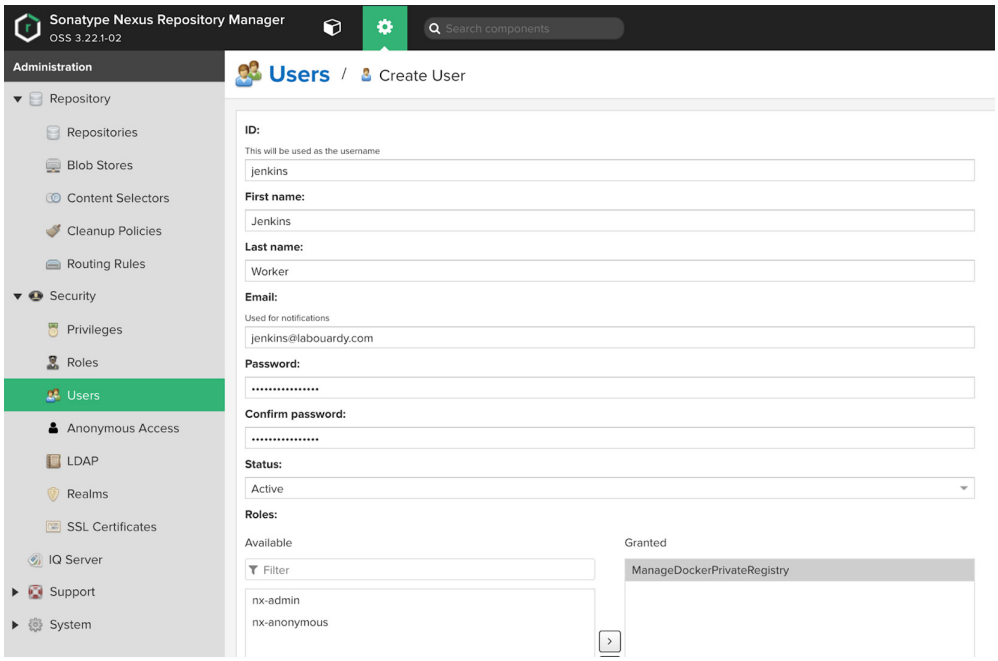


Figure 9.14 Docker registry credentials for Jenkins

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

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

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

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

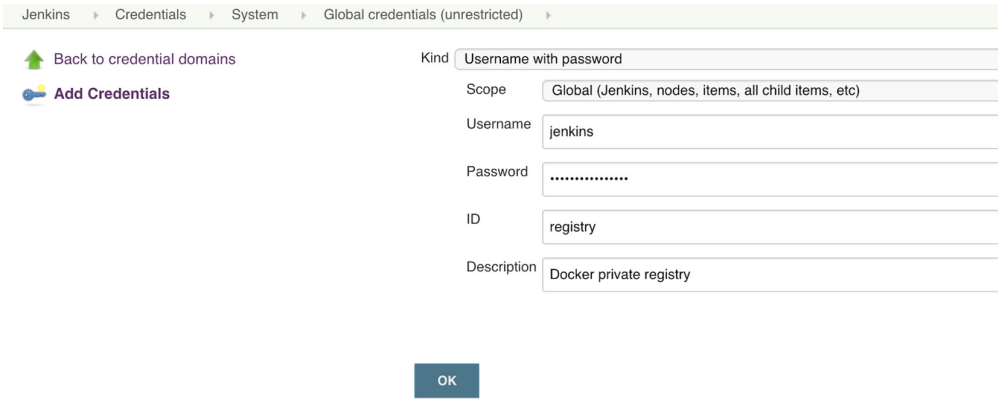


Figure 9.15 Docker registry credentials

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

9.2.2 *Amazon Elastic Container Registry*

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

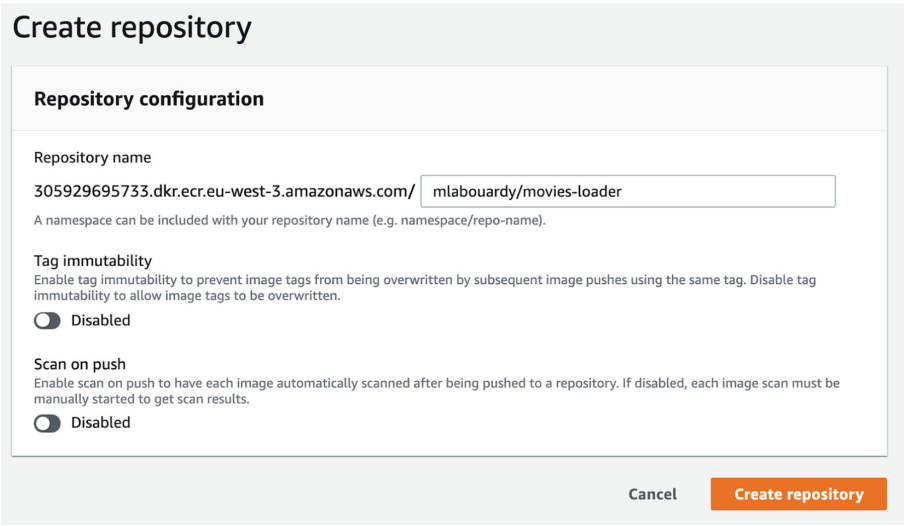


Figure 9.16 ECR new repository

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

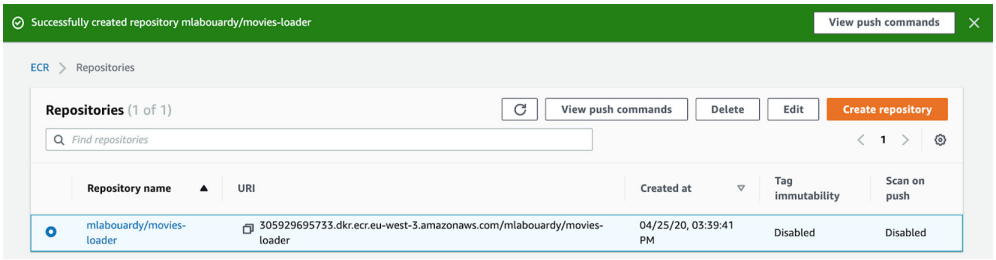


Figure 9.17 Movie loader ECR repository

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

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

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

For Windows users, here is the command:

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

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

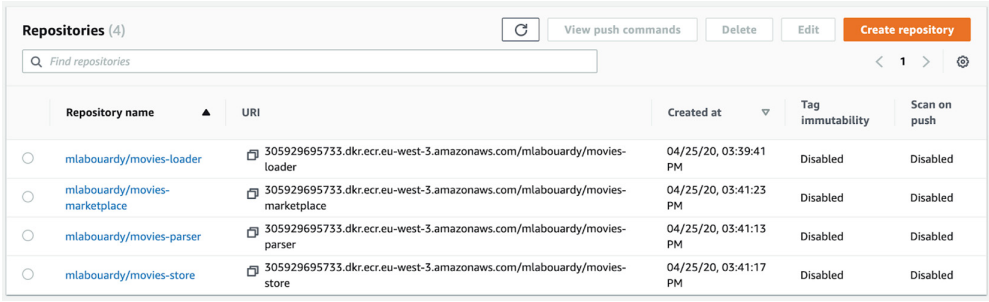


Figure 9.18 ECR repository for each microservice

9.2.3 Azure Container Registry

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

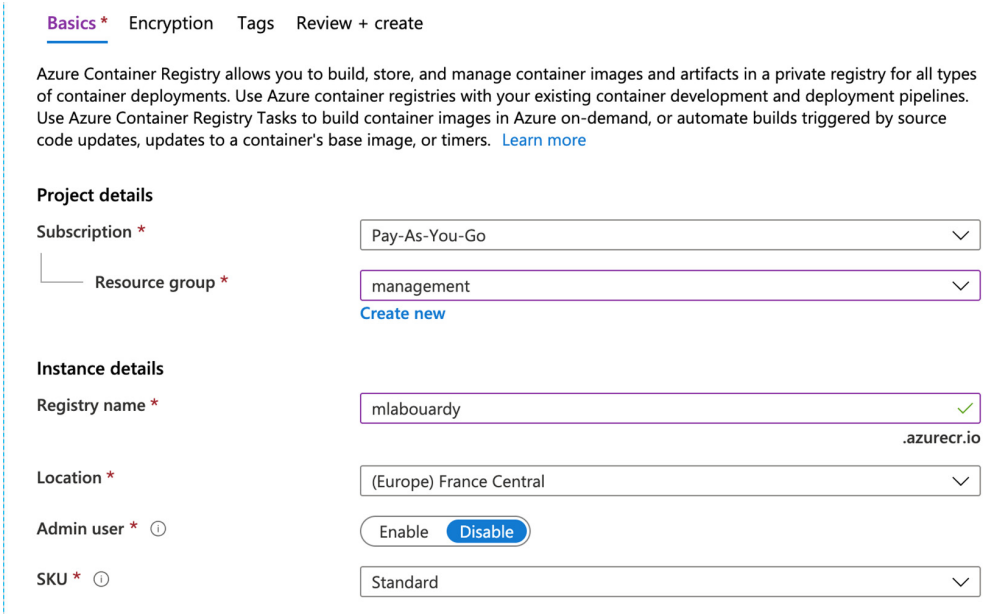


Figure 9.19 Azure new registry configuration

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

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

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

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

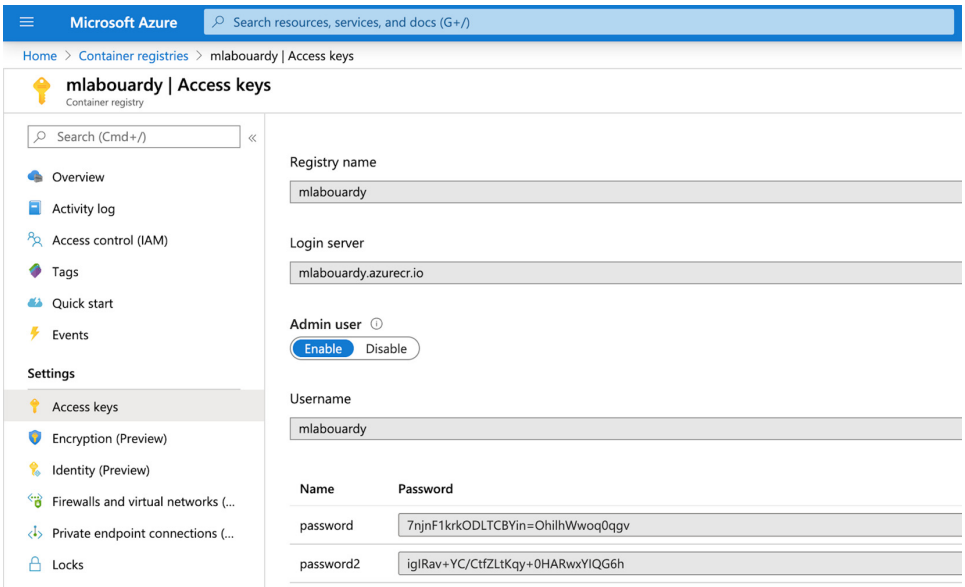


Figure 9.20 Azure Docker registry admin credentials

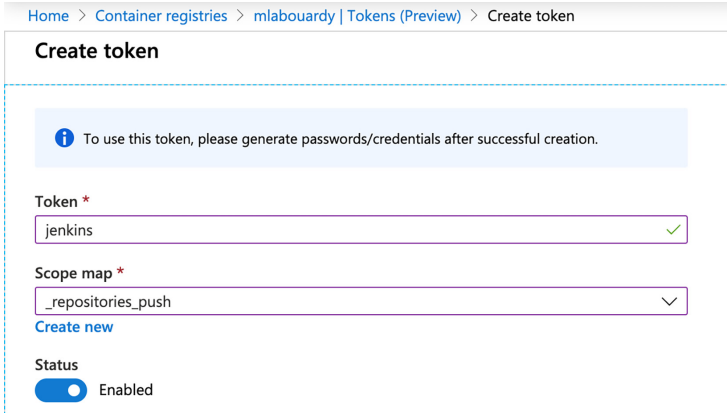


Figure 9.21 Azure Docker registry new access token

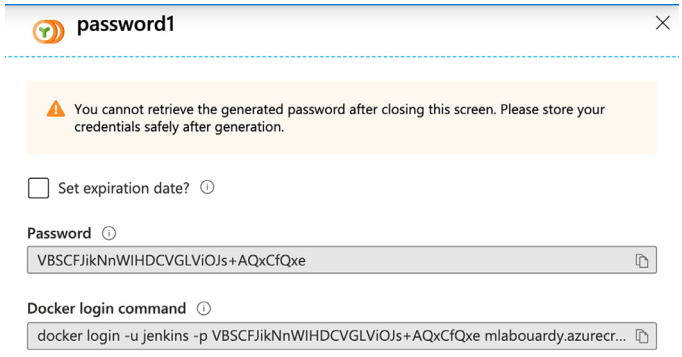


Figure 9.22 Azure Docker registry credentials

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

9.2.4 Google Container Registry

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

Listing 9.13 `gcloud` installation

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

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

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

```
gcloud auth configure-docker
```

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

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

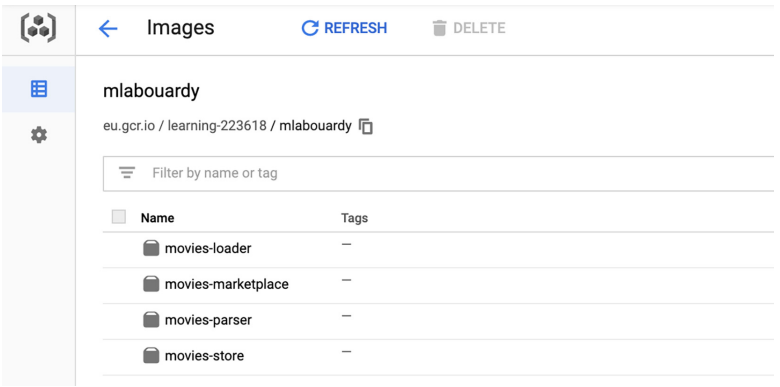


Figure 9.23 Google Container Registry images

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

9.3 Tagging Docker images the right way

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

Listing 9.14 Publishing Docker image to a registry

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

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

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

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

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

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

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

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

Figure 9.24 Docker push command logs

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

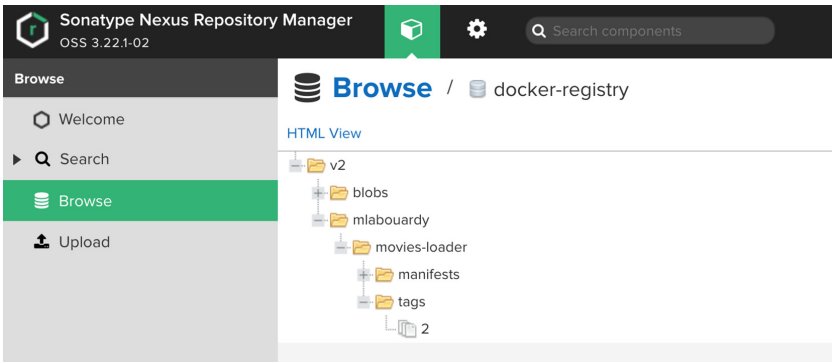


Figure 9.25 Docker image stored in Nexus

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

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

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

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

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

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

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

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

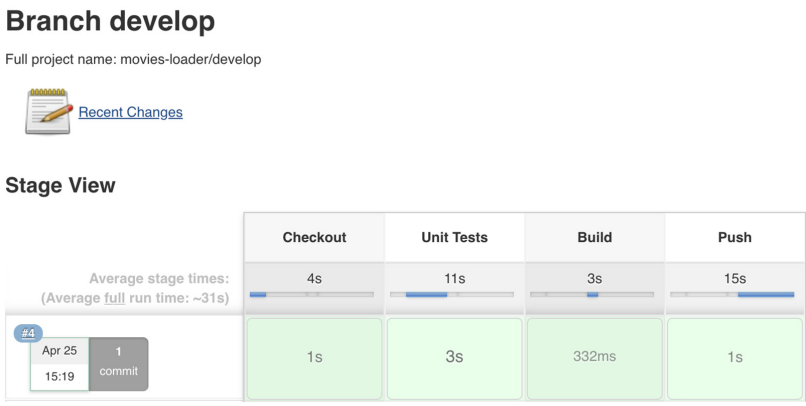


Figure 9.26 Movie loader CI pipeline

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

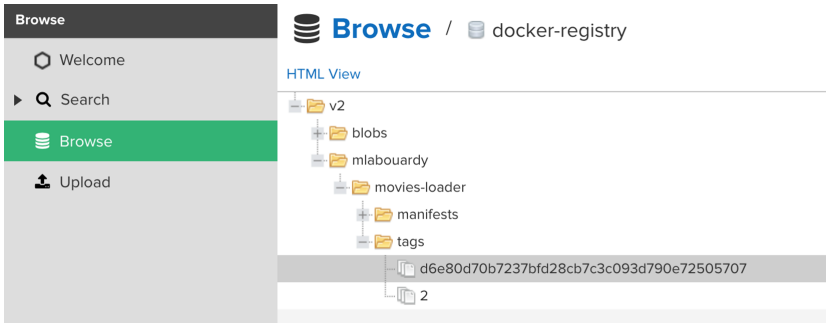


Figure 9.27 Commit ID image tag

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

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

The `Push` stage code block is as follows:

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

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

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

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

Listing 9.15 Publishing the Docker image to ECR

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

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

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

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

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

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

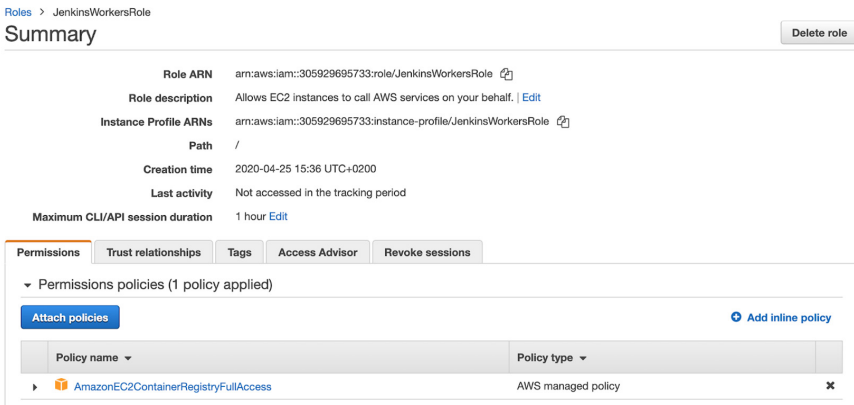


Figure 9.28 Jenkins workers' IAM instance profile

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

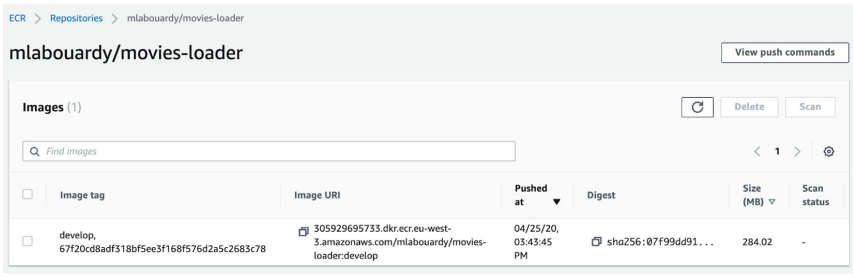


Figure 9.29 Movie loader ECR repository images

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

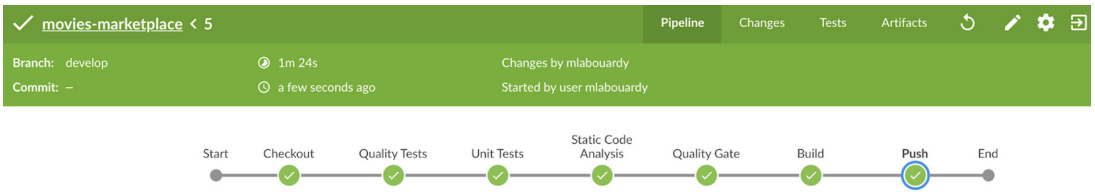


Figure 9.30 Movie marketplace CI pipeline

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

9.4 Scanning Docker images for vulnerabilities

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

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

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

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

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

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

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

Figure 9.31 Docker Compose stack services

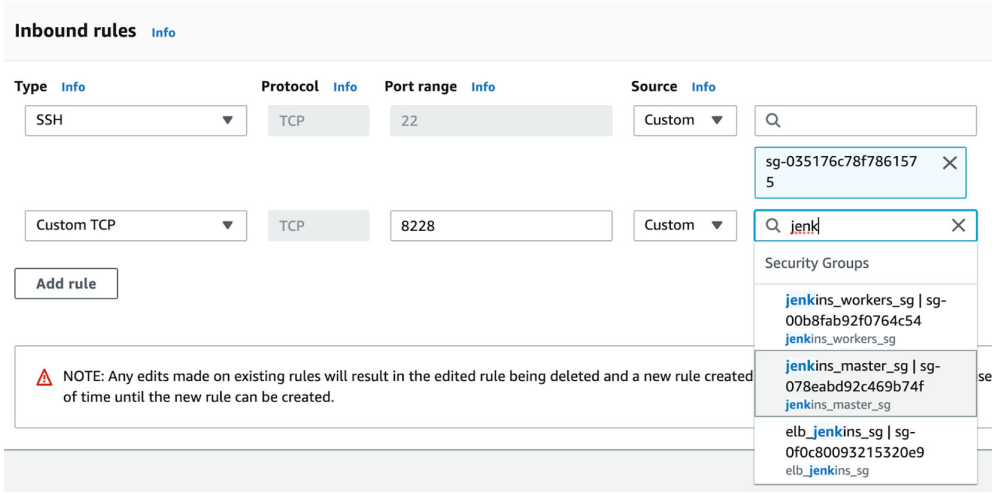


Figure 9.32 Anchore instance's security group

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

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

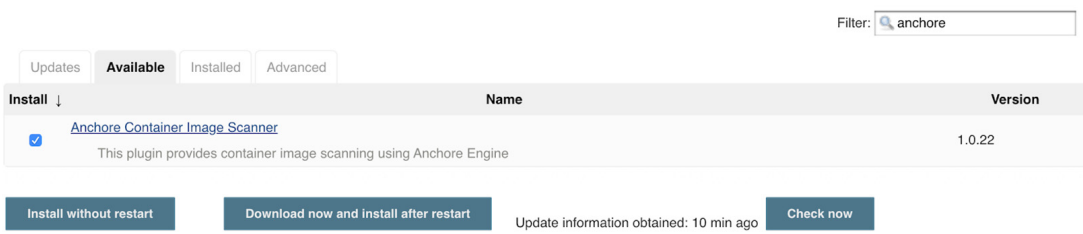


Figure 9.33 Anchore Container Image Scanner plugin

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

Figure 9.34 Anchore plugin configuration

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

Listing 9.16 Analyzing Docker images with Anchore

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9.35 Image scanning with Anchore

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

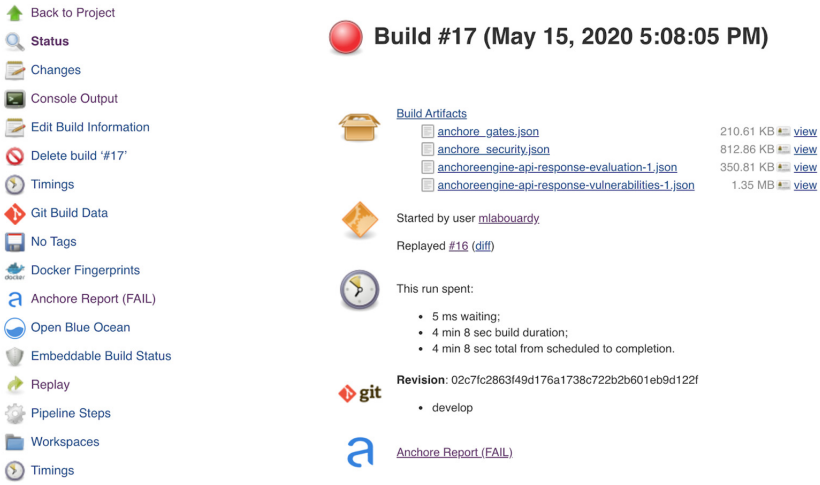


Figure 9.36 Anchore report results

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

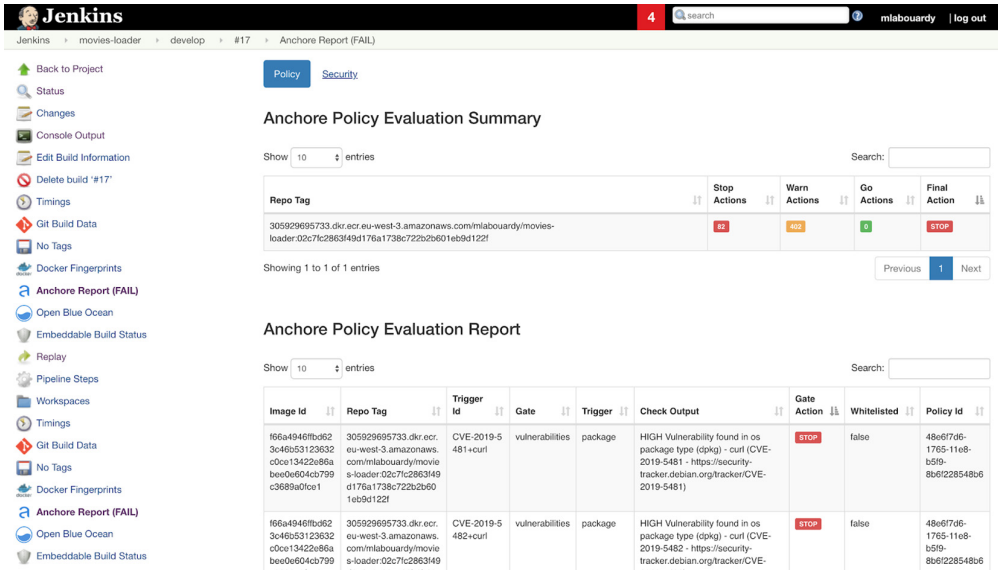


Figure 9.37 Anchore Common Vulnerabilities and Exposures (CVE) report

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.

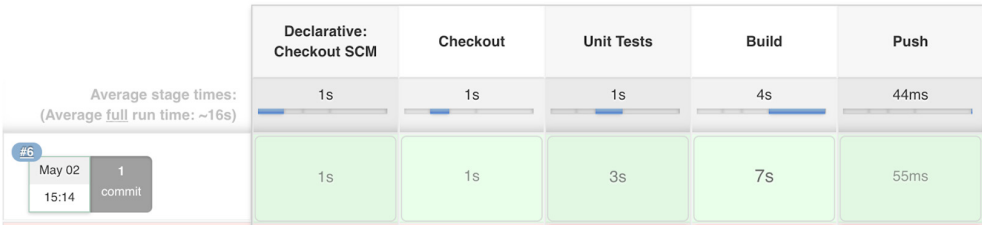


Figure 9.39 Jenkinsfile declarative pipeline execution

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.

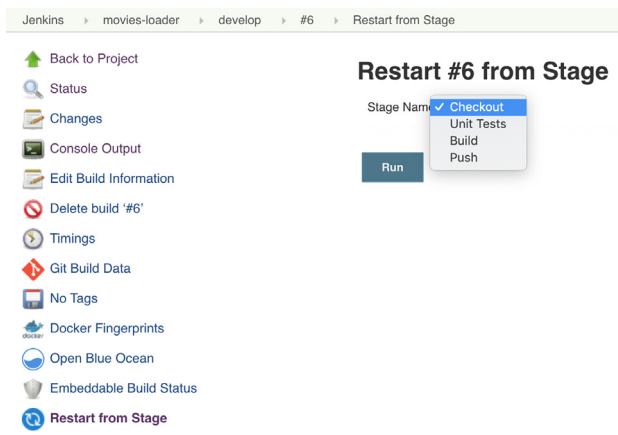


Figure 9.40 Restart from Stage feature

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.

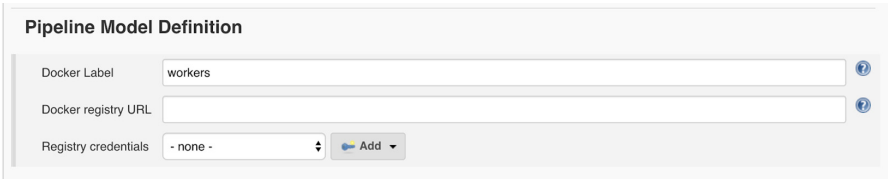


Figure 9.41 Pipeline model definition

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.

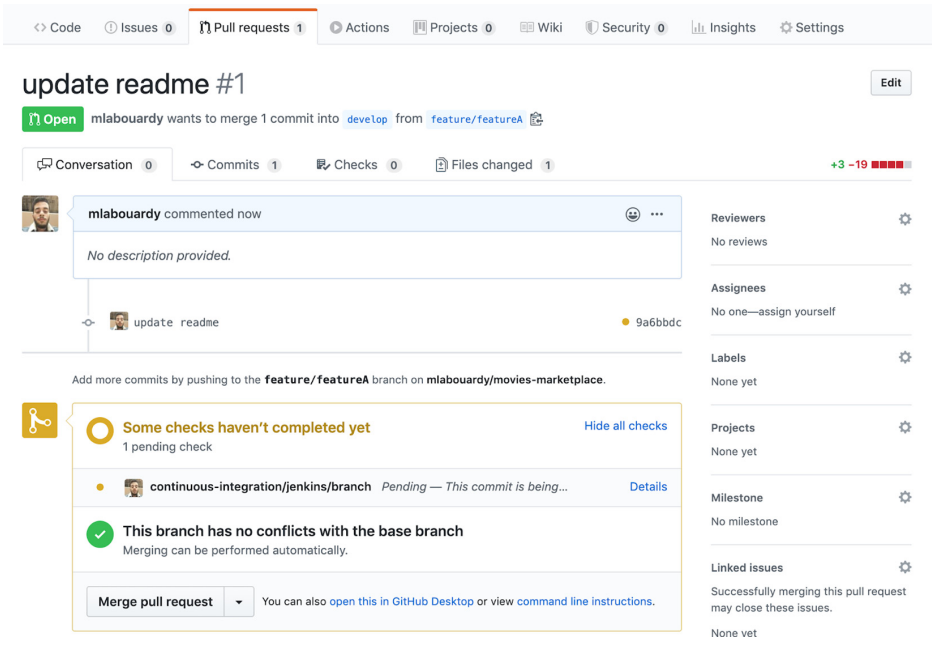


Figure 9.42 New pull request

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

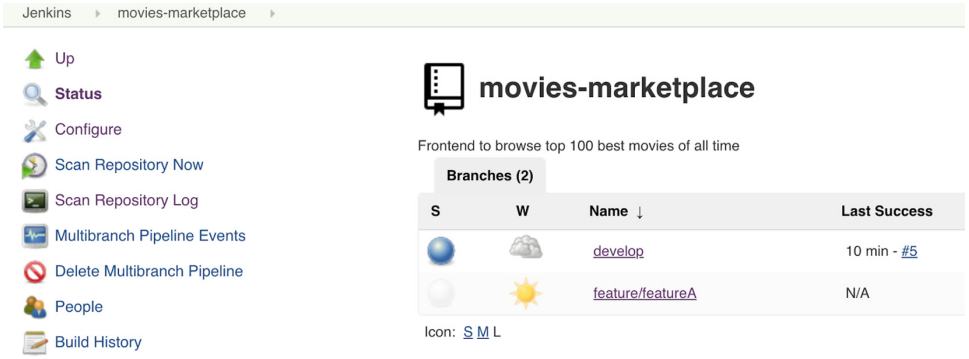


Figure 9.43 Build execution on the feature branch

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.

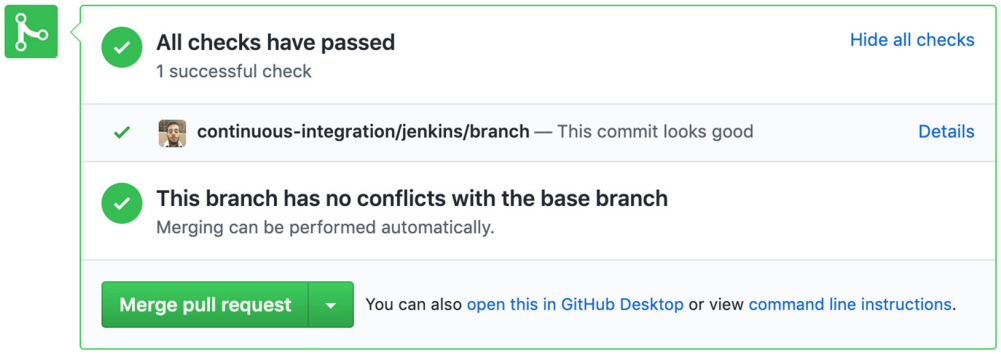


Figure 9.44 Jenkins post-build status on GitHub PR

> **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.

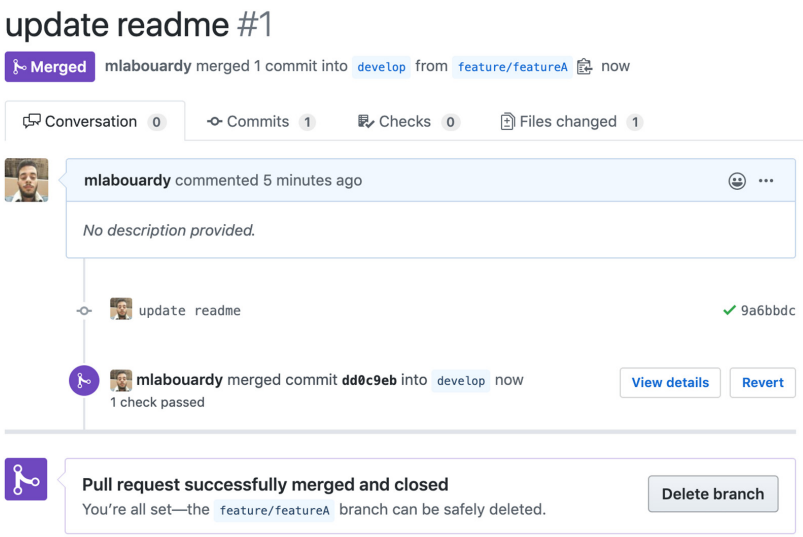


Figure 9.45 Merge and delete the feature branch.

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.

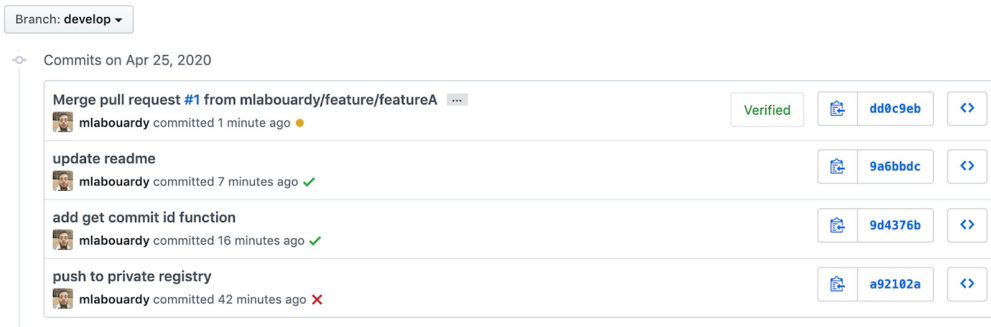


Figure 9.46 Jenkins build status history

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.

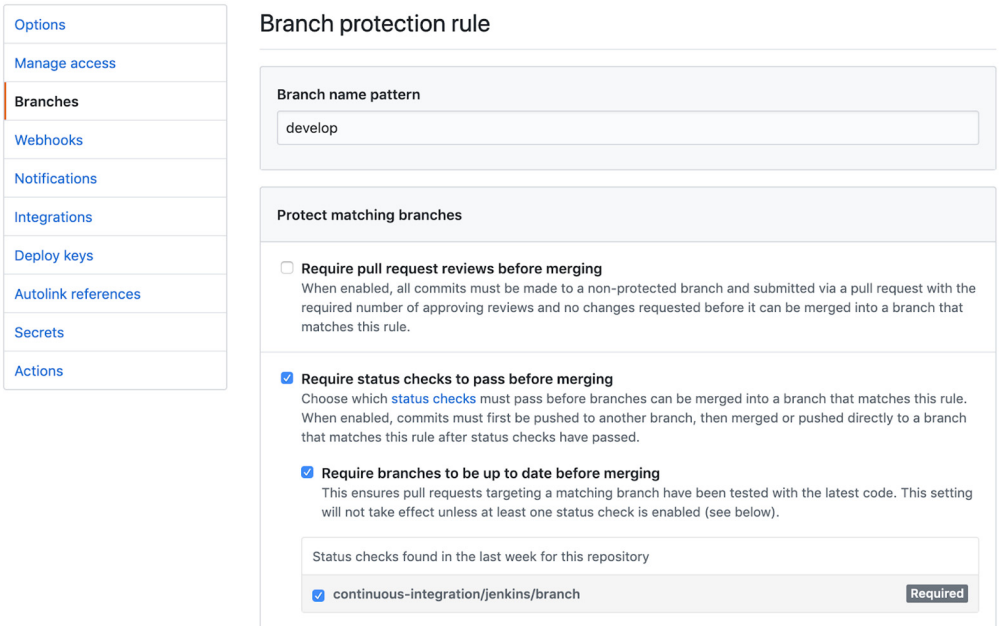


Figure 9.47 GitHub branch protection

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.