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.

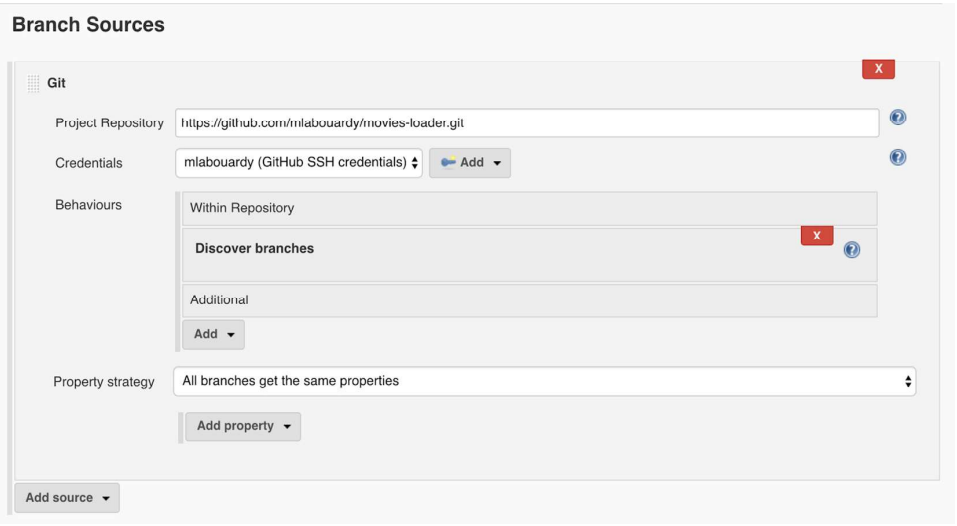


Figure 7.33 Configuring the Jenkins job to use SSH keys

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.

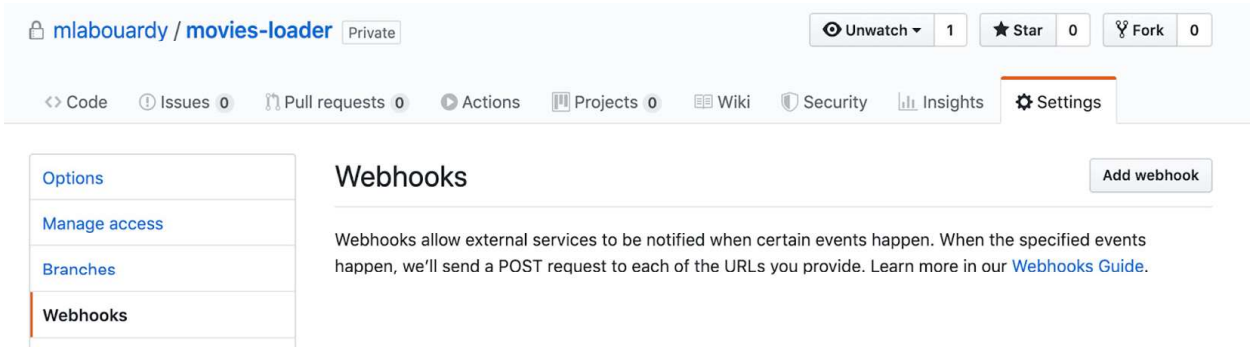


Figure 7.35 GitHub Webhooks section

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).

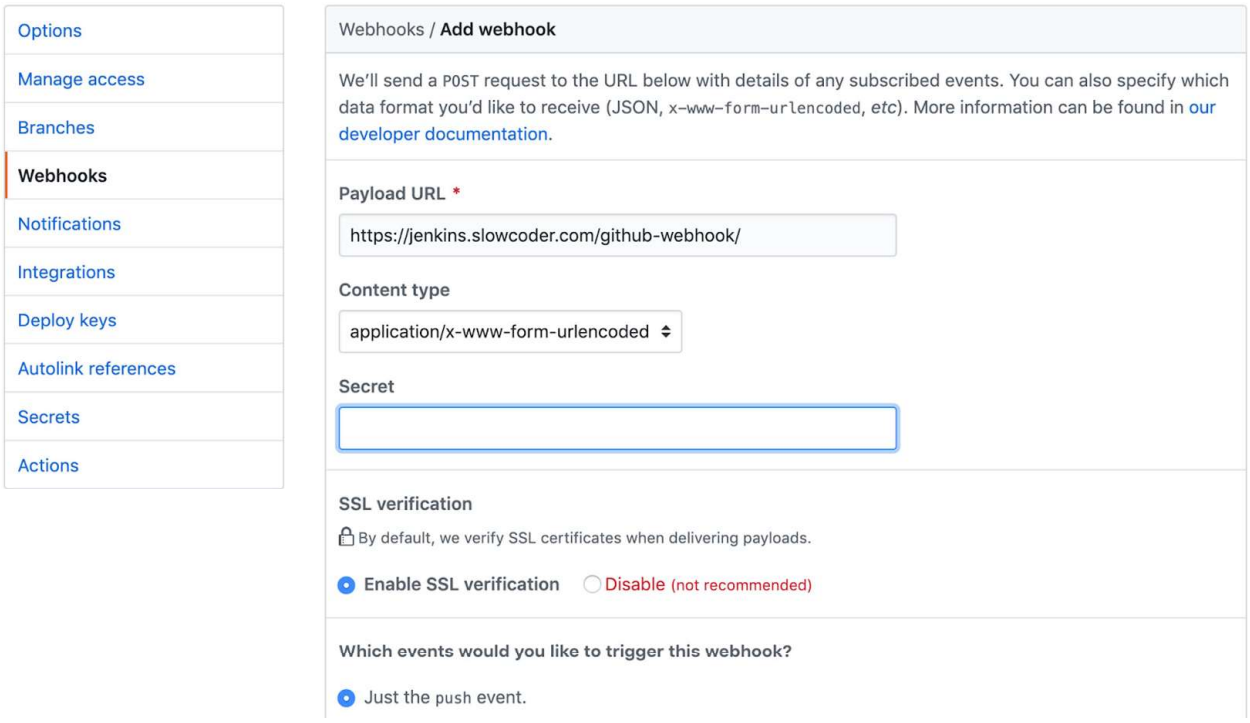


Figure 7.36 Jenkins webhook settings

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.

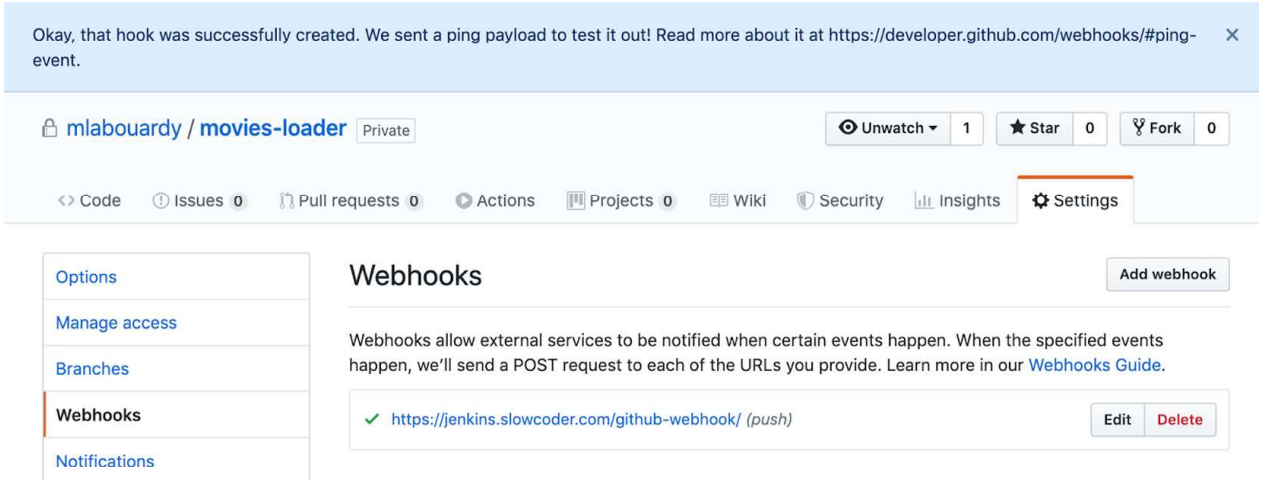
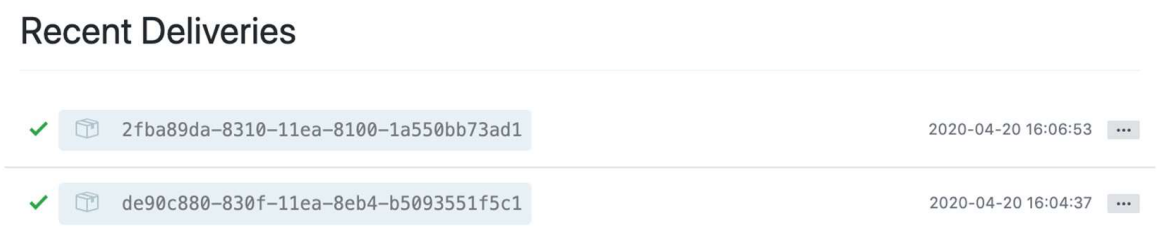


Figure 7.37 Jenkins webhook settings

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.

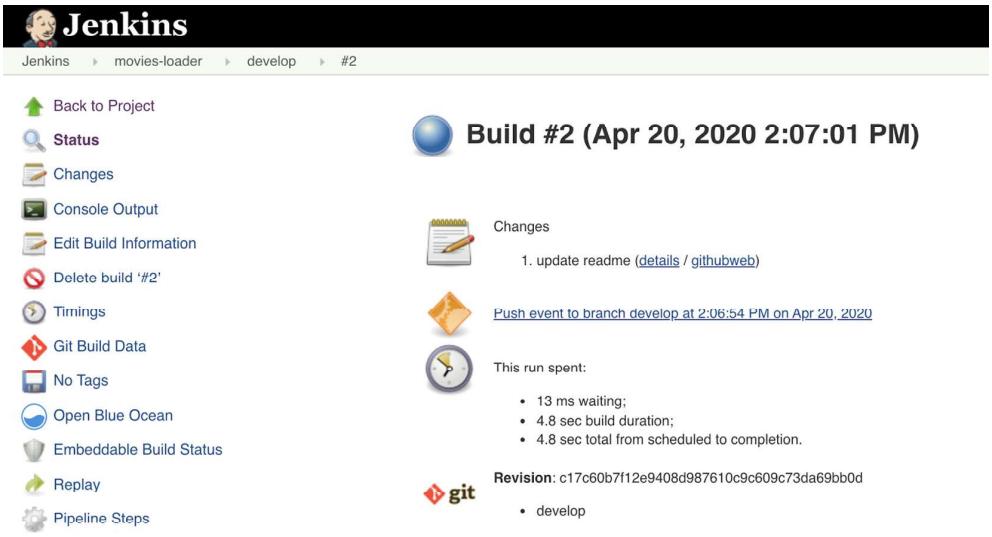


Figure 7.38 GitHub push event

> **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.

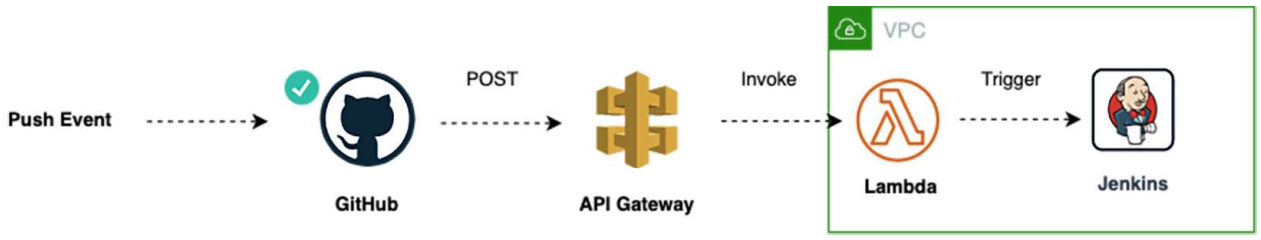


Figure 7.39 GitHub webhook setup with API Gateway

> **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.

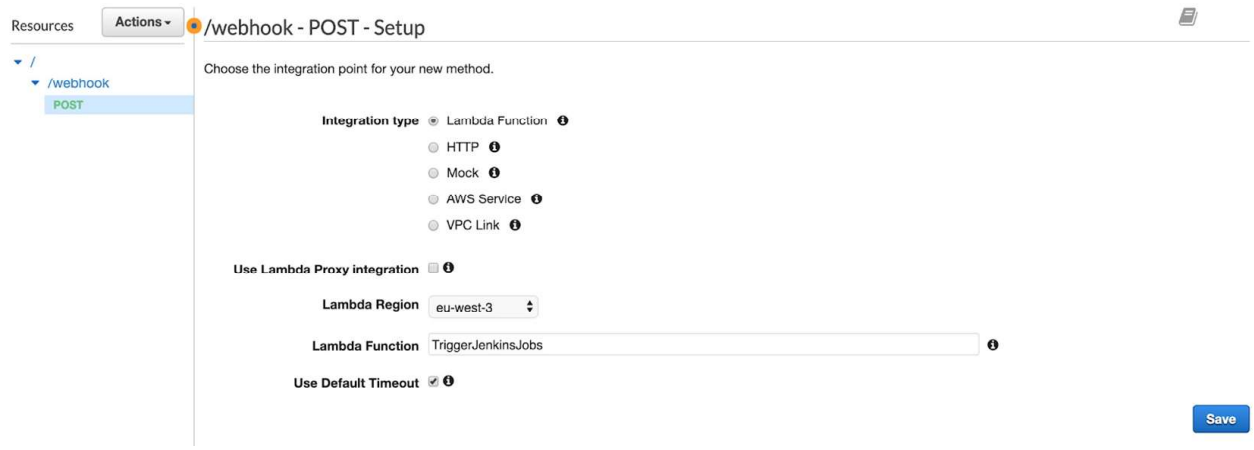


Figure 7.40 Triggering a Lambda function with API Gateway

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.

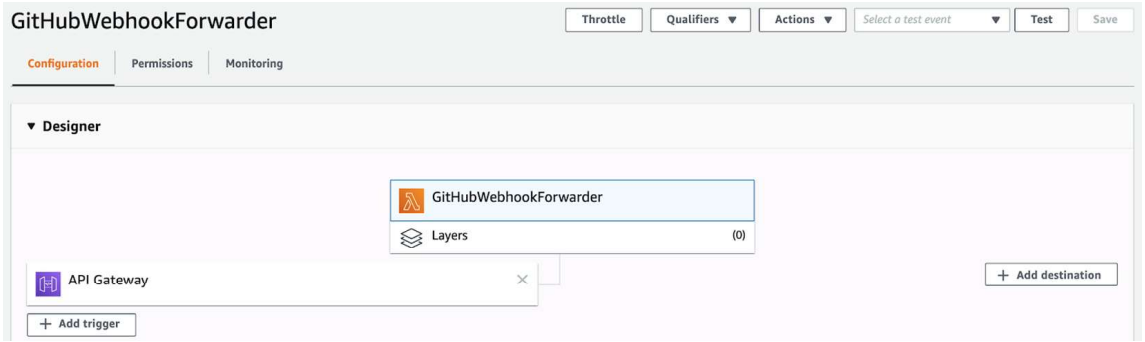


Figure 7.41 `GitHubWebhookForwarder` Lambda function

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.

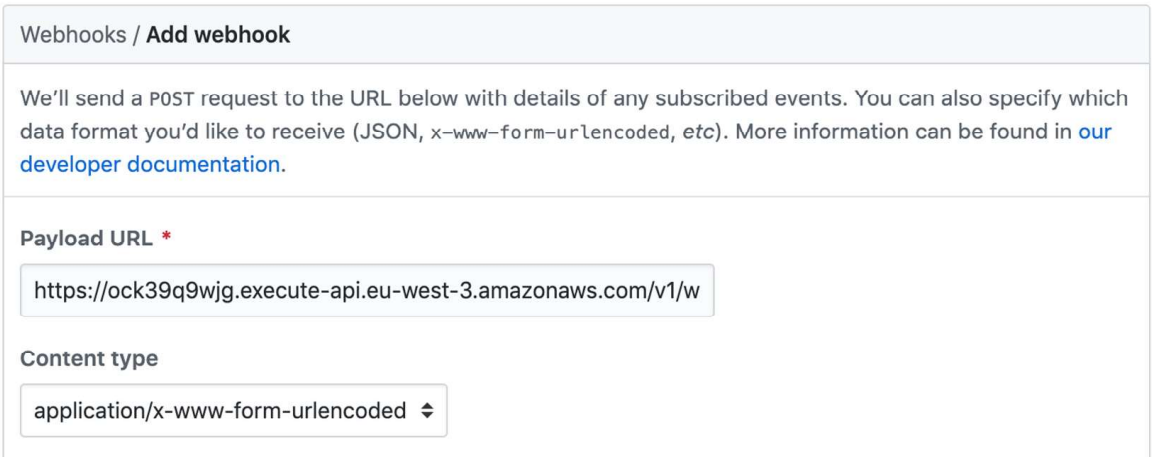


Figure 7.42 GitHub webhook based on API Gateway URL

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.

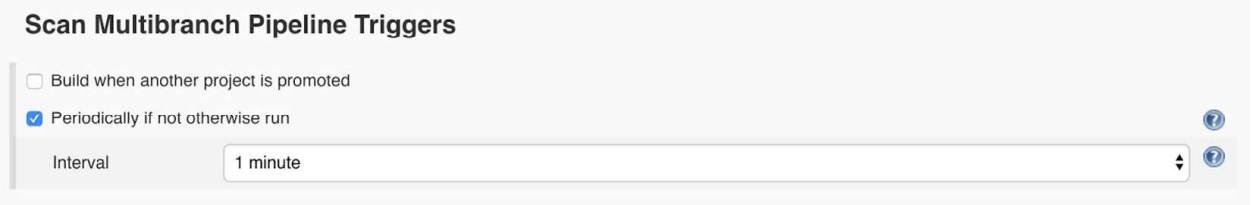


Figure 7.43 Under the job's settings, you can define the interval of checks.

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.

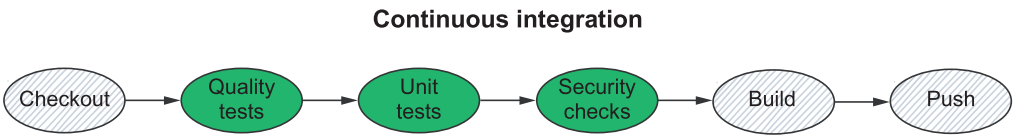


Figure 8.1 The test stages covered in this chapter

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.

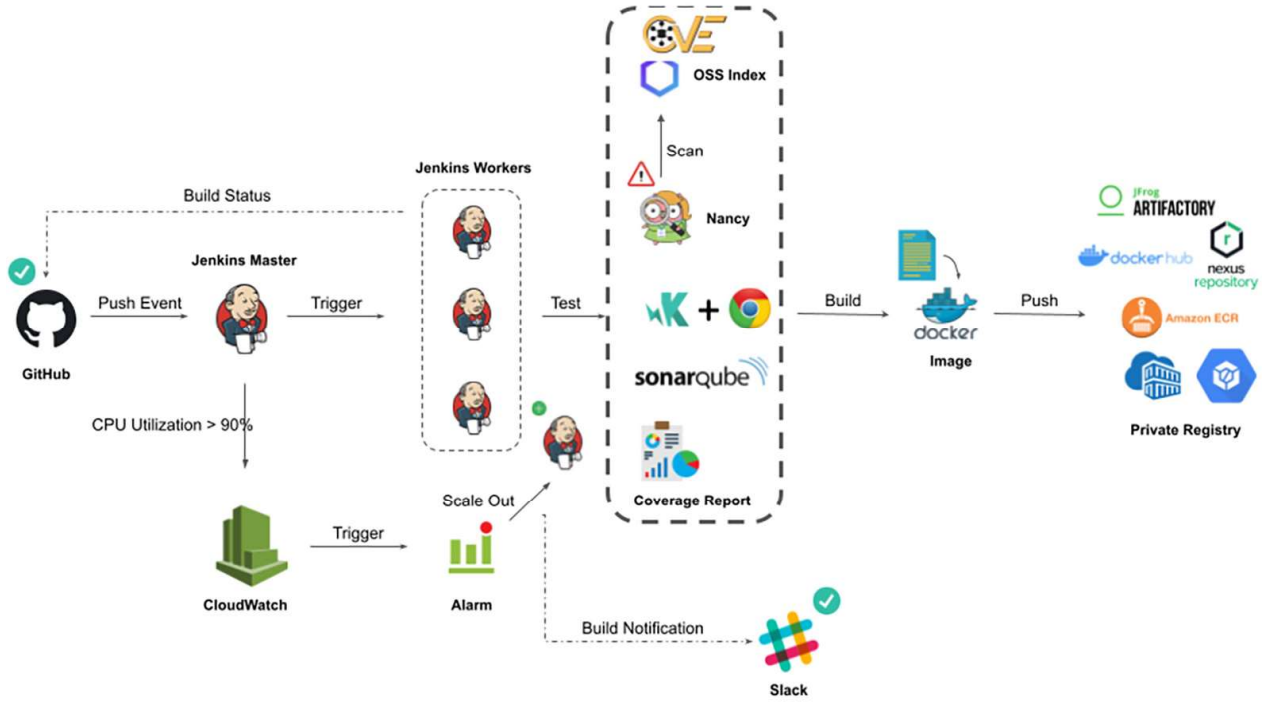


Figure 8.2 Target CI pipeline

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.

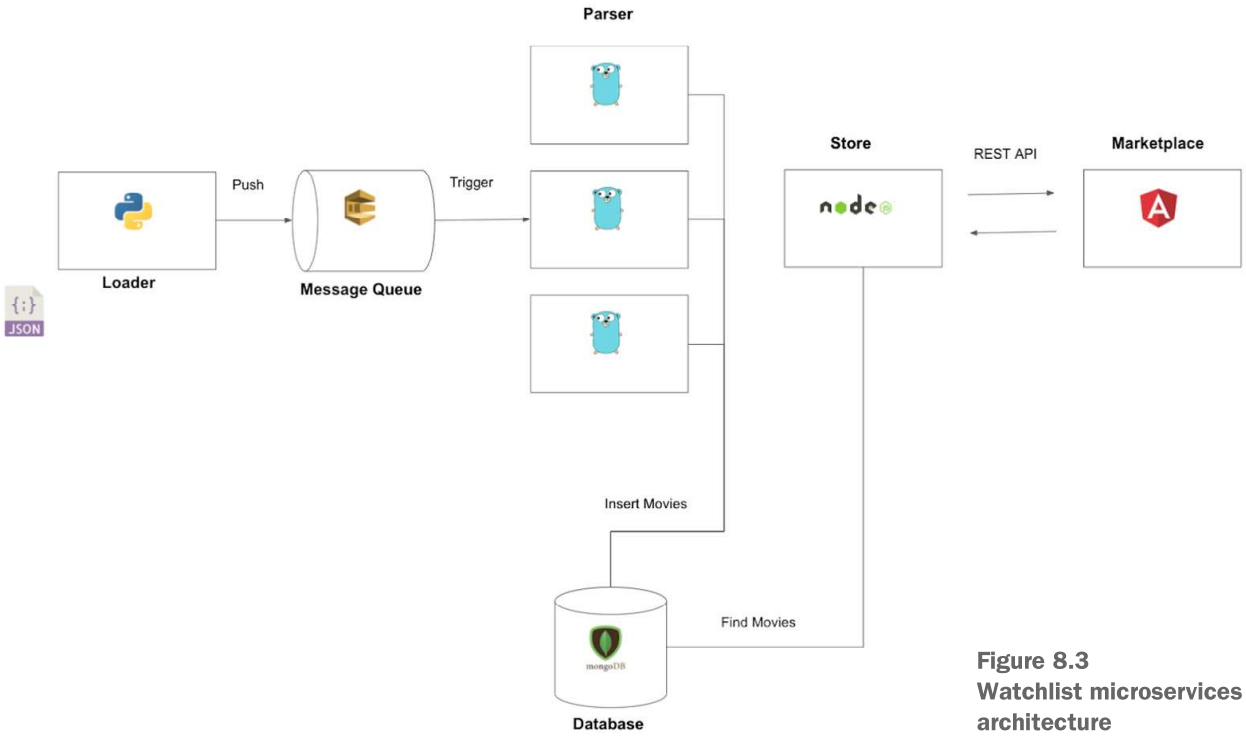


Figure 8.3 Watchlist microservices architecture

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.

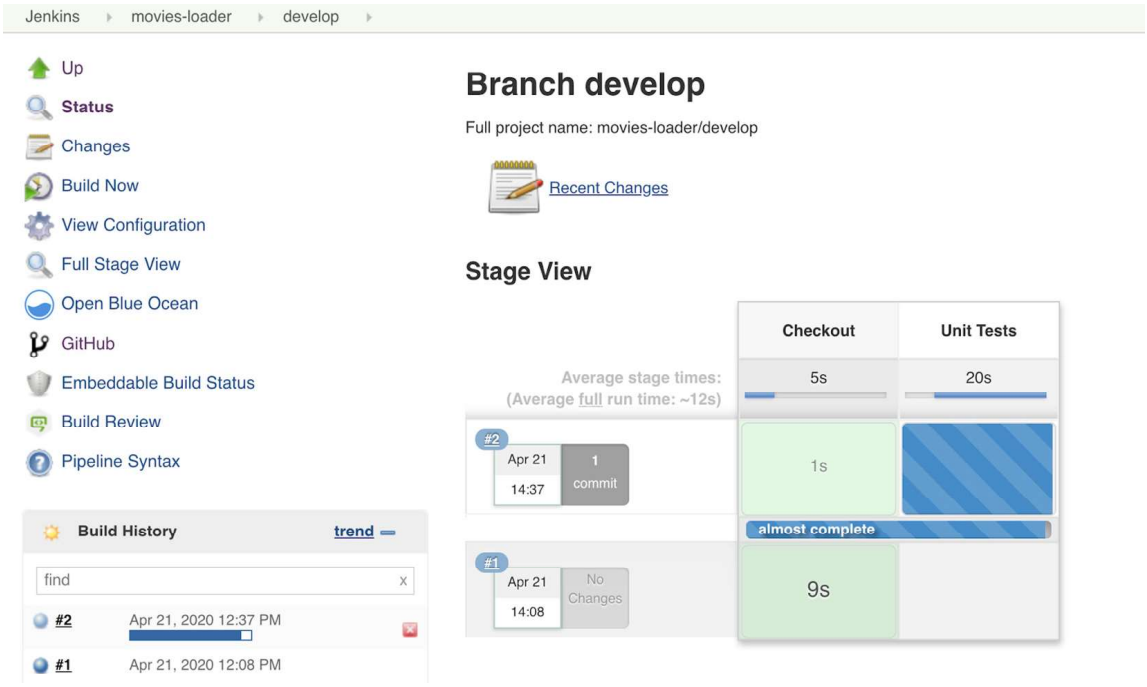


Figure 8.4 Unit test stage execution

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.


```
+ docker build -t mlabouardy/movies-loader-test -f Dockerfile.test .
Sending build context to Docker daemon  97.79kB

Step 1/5 : FROM python:2.7.10
 ---> 4442f7b981c4
Step 2/5 : WORKDIR /app
 ---> Using cache
 ---> 32b2206925a5
Step 3/5 : COPY test_main.py .
 ---> Using cache
 ---> 4b832ce1cf99
Step 4/5 : COPY movies.json .
 ---> Using cache
 ---> 00b9cf9b6b7e
Step 5/5 : CMD python test_main.py
 ---> Using cache
 ---> 29c2f7ca651d
Successfully built 29c2f7ca651d
Successfully tagged mlabouardy/movies-loader-test:latest
[Pipeline] sh
+ docker run --rm mlabouardy/movies-loader-test
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline

GitHub has been notified of this commit's build result

Finished: SUCCESS
```

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.

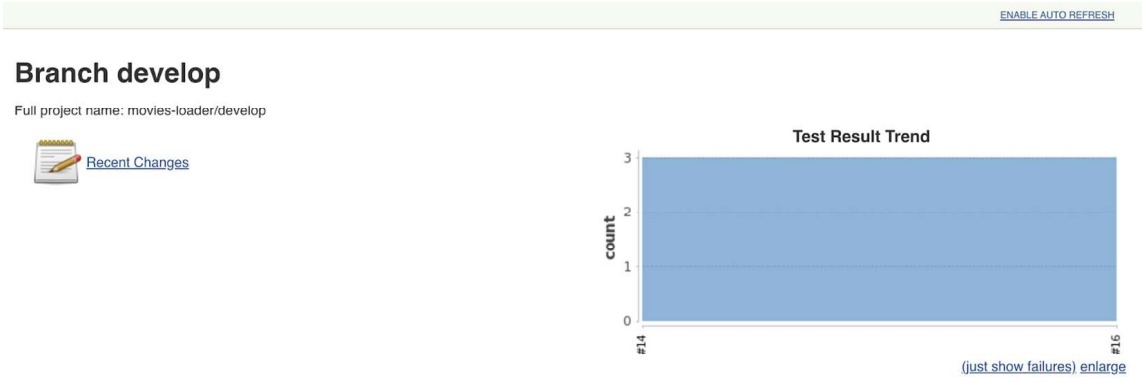


Figure 8.7 JUnit test chart analyzer

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.

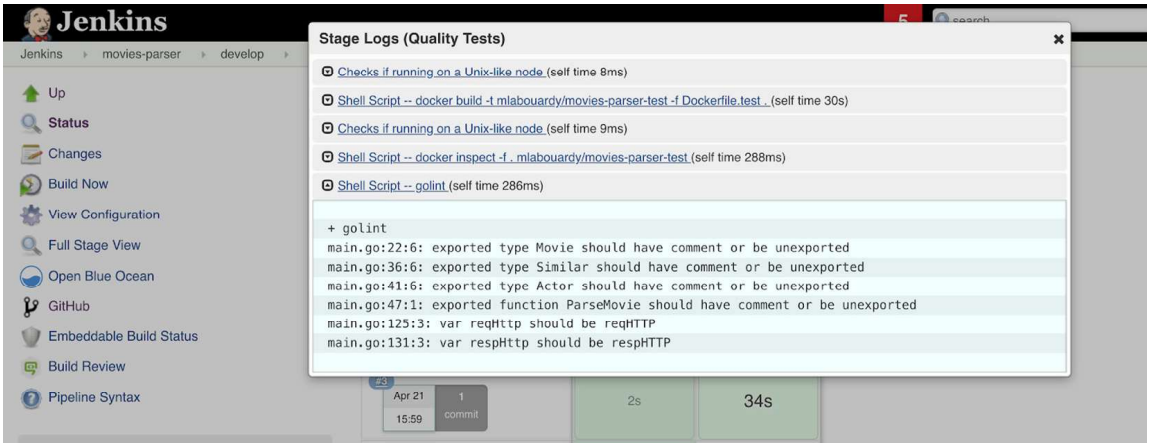


Figure 8.8 The `golint` command output identifies the missing comments

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.

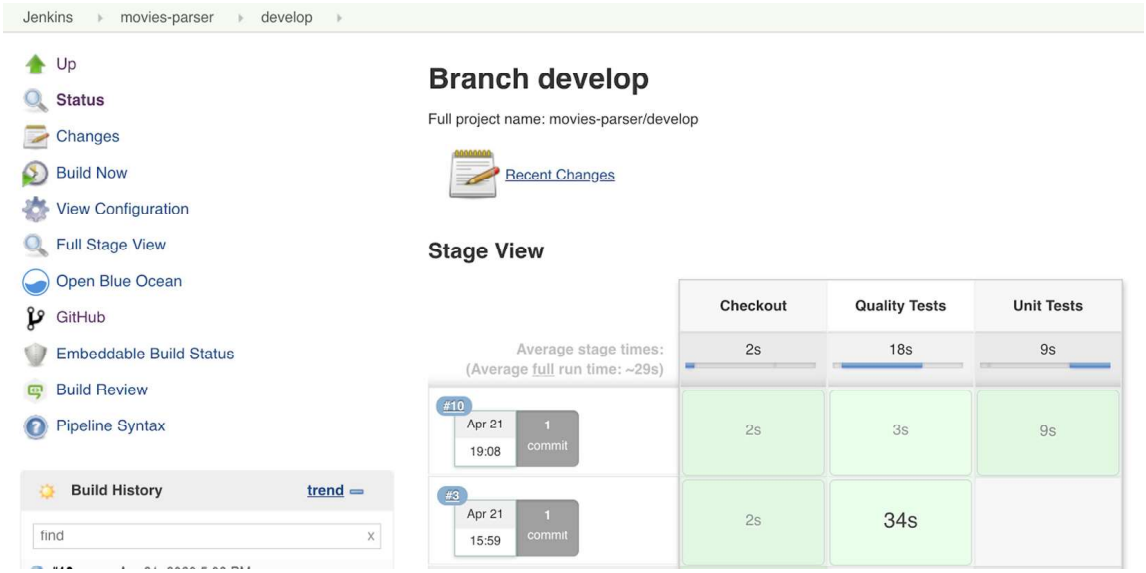


Figure 8.9 Go CI pipeline

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

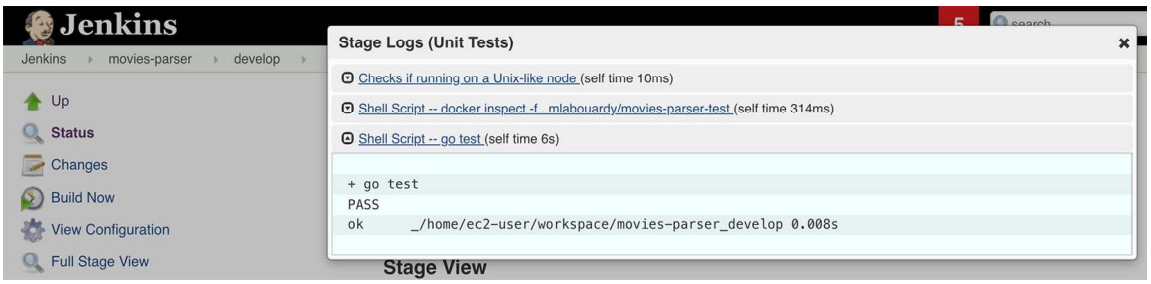


Figure 8.10 `go test` command output

> **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
```

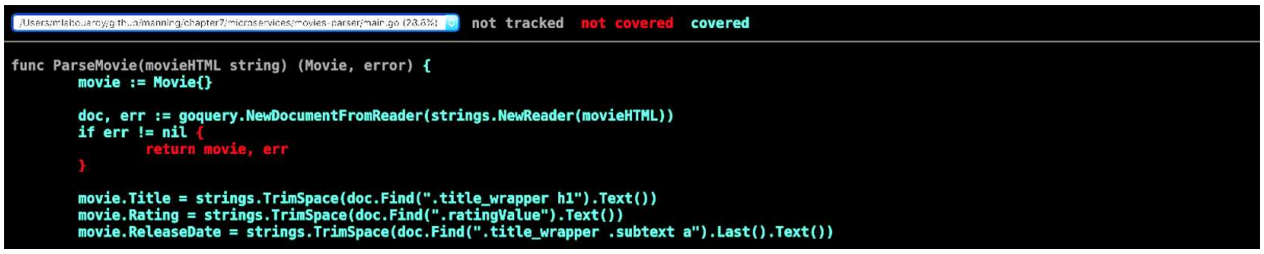


Figure 8.11 The coverage.html content can be served from the Jenkins dashboard at the end of the test stage.

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.

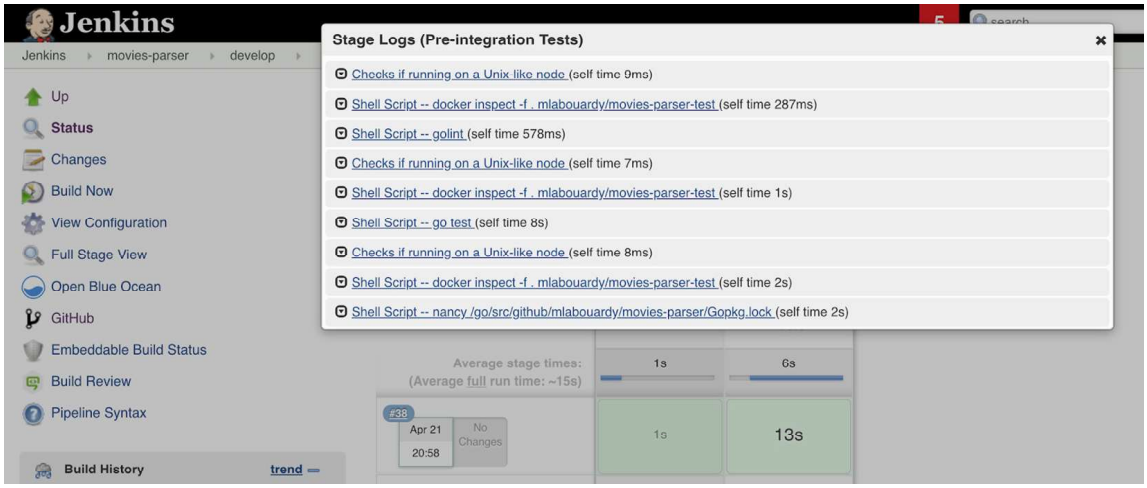


Figure 8.14 Pre-integration tests' parallel execution

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

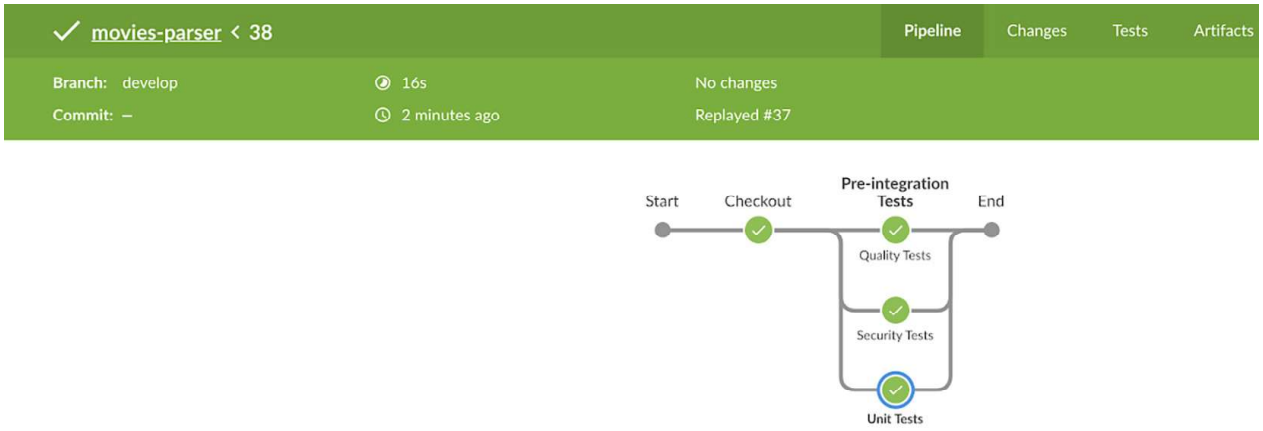


Figure 8.15 Parallel stages in Blue Ocean

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.

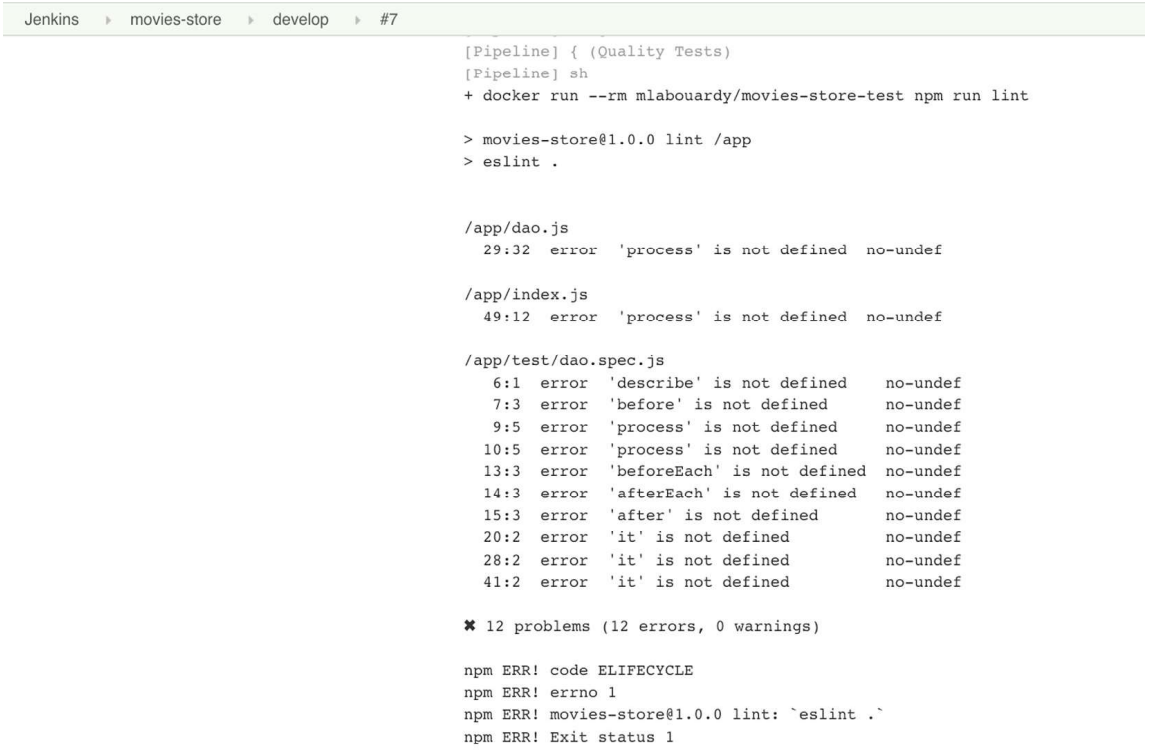


Figure 8.16 ESLint problem detection

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.

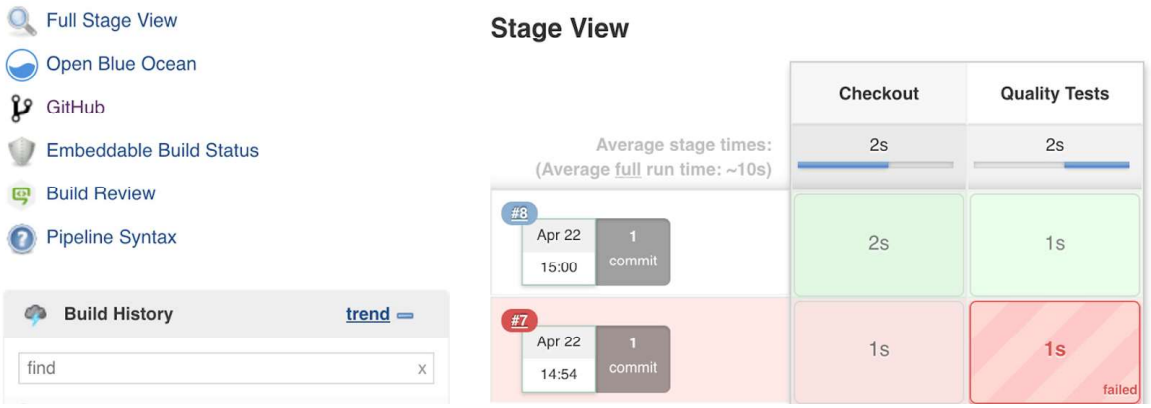


Figure 8.17 CI pipeline execution after fixing ESLint errors

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"
}
```

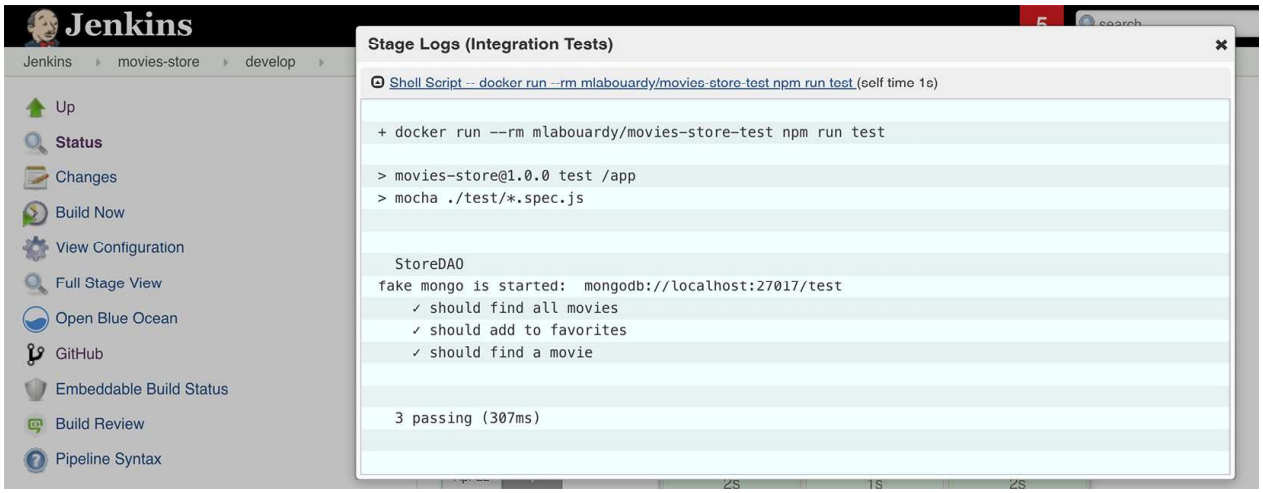


Figure 8.18 Unit testing using the Mocha framework

> **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.

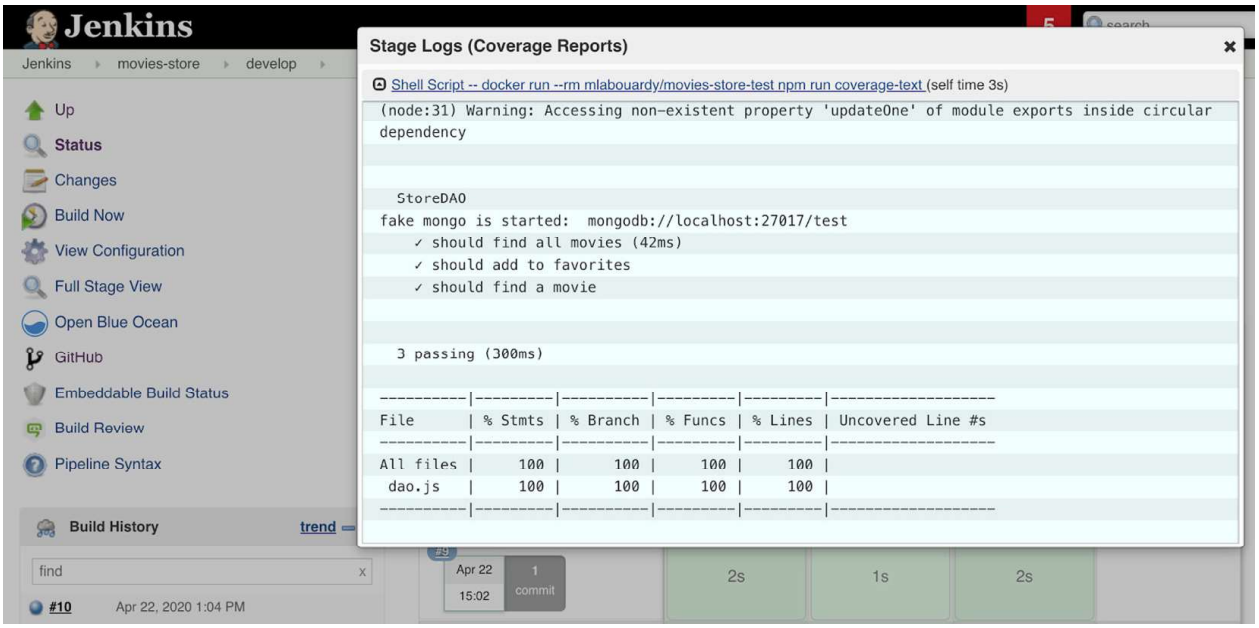


Figure 8.19 Istanbul coverage reports in text format

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.

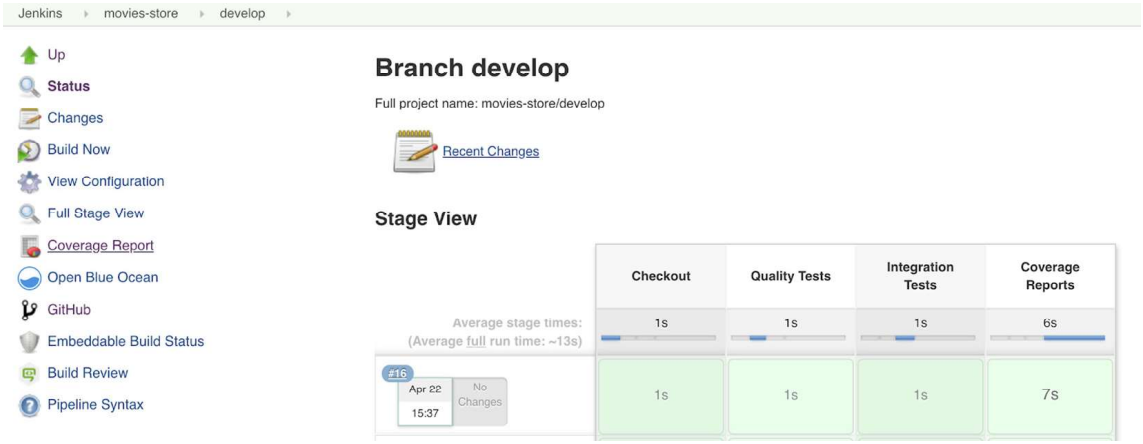


Figure 8.21 The coverage report can be accessible from the Jenkins panel.

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.

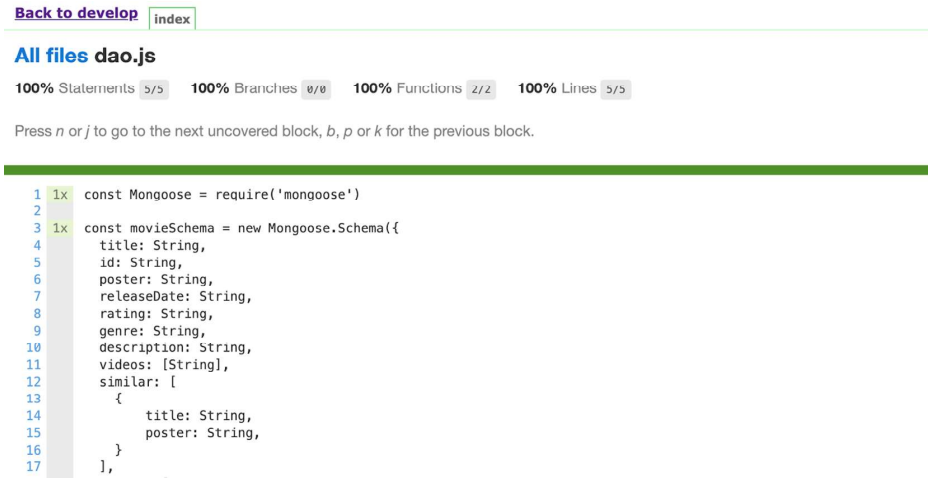


Figure 8.22 Deep dive inside the coverage report

> **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.

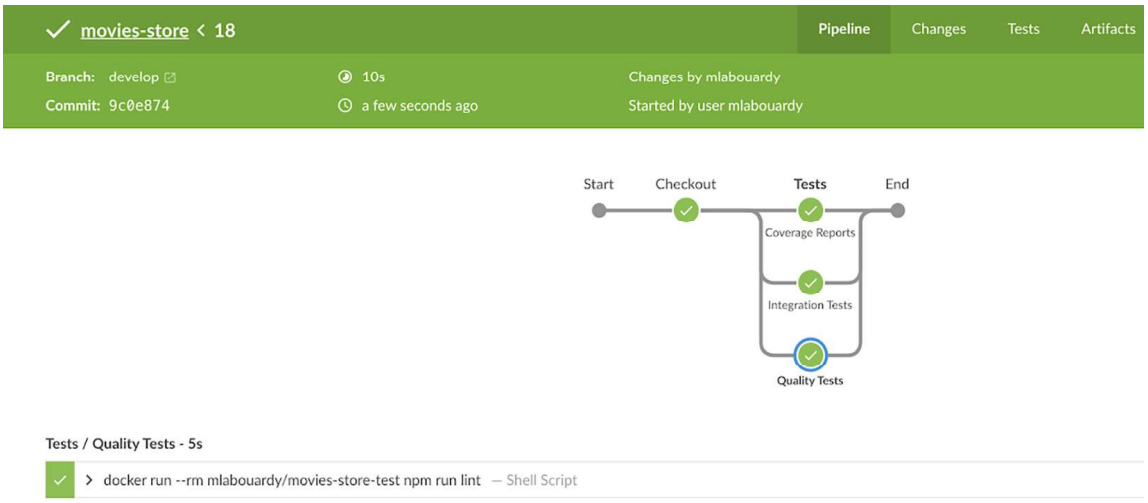


Figure 8.23 Running tests in parallel

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.

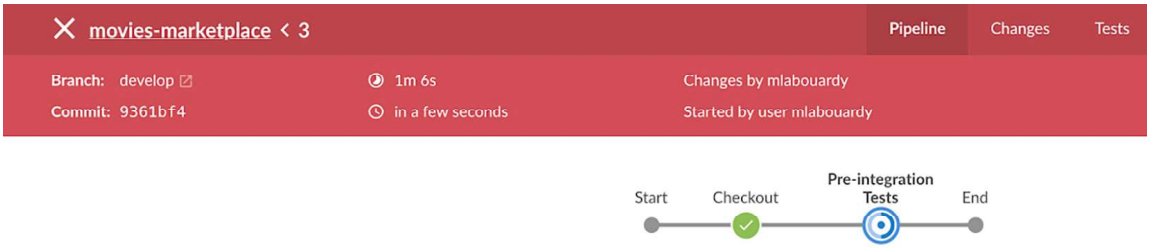


Figure 8.24 CI pipeline failure

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.

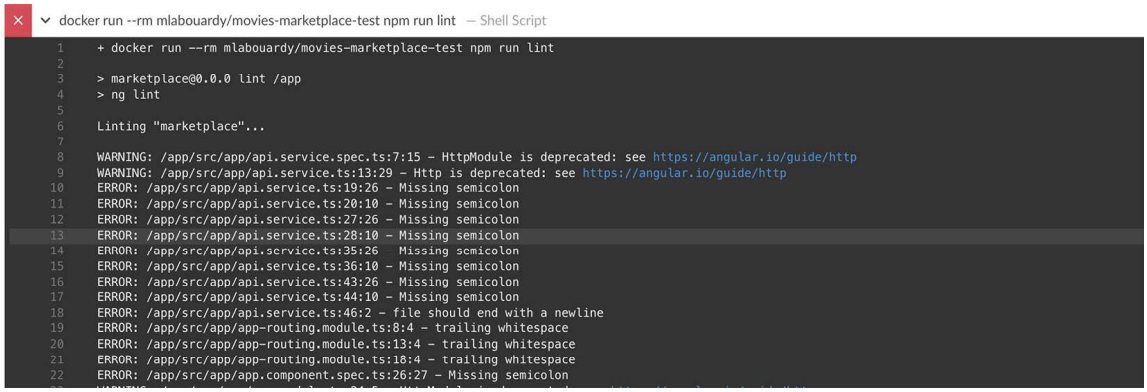


Figure 8.25 Angular linting output logs

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.

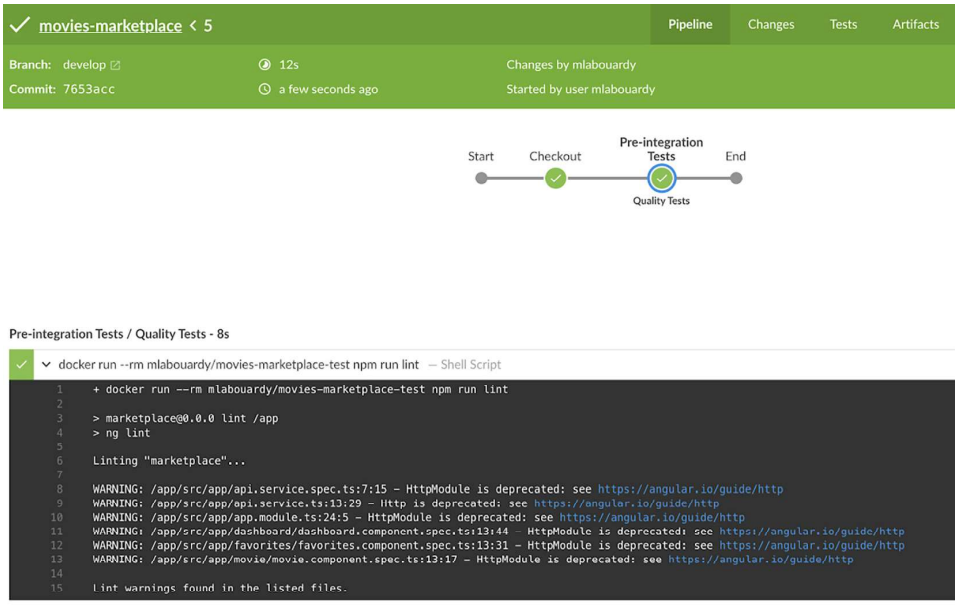


Figure 8.26 Angular linting output logs

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.

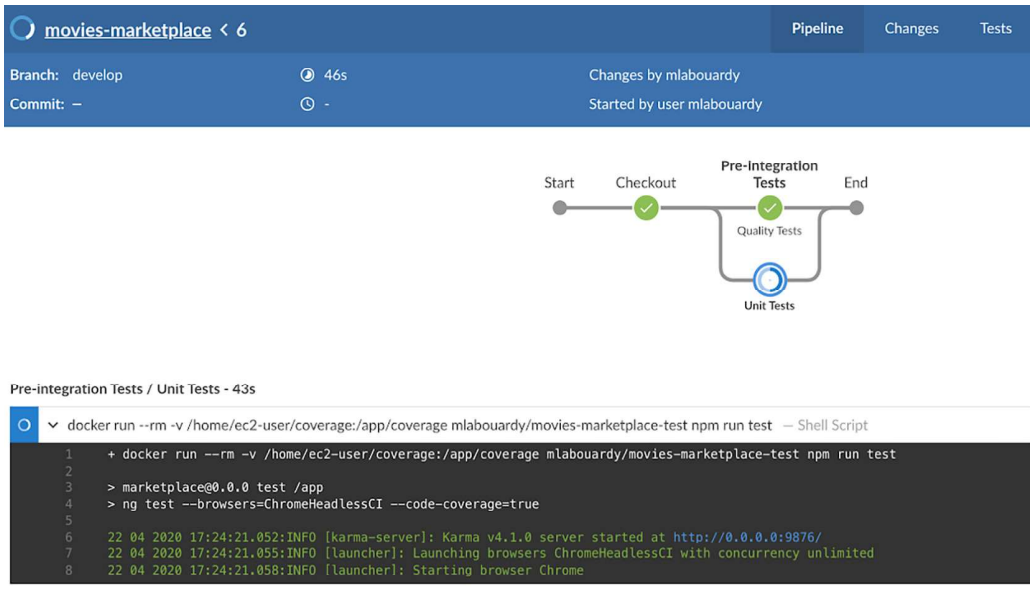


Figure 8.27 Running headless Chrome inside a Docker container

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

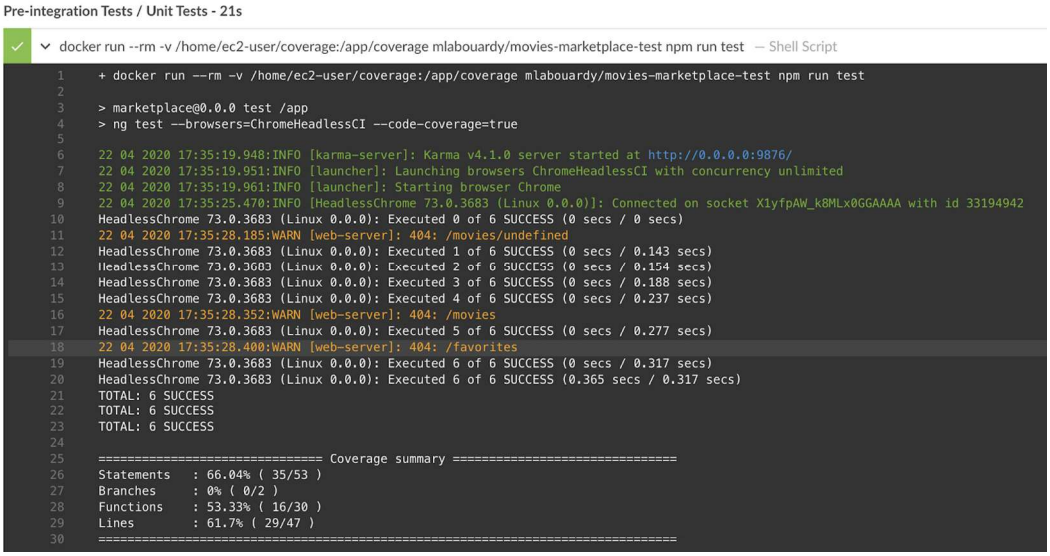


Figure 8.28 Successful execution of the Karma unit tests

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

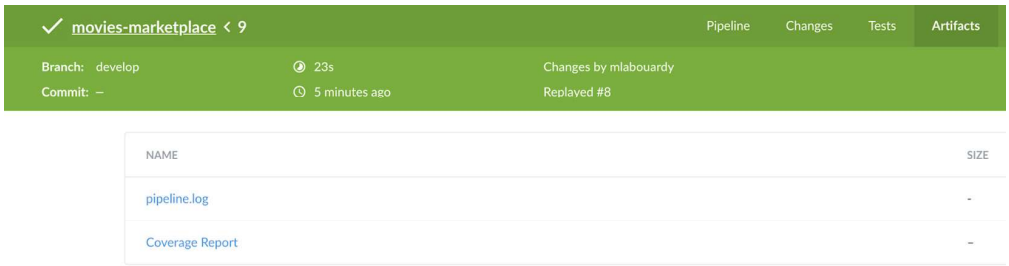


Figure 8.29 Coverage report alongside other artifacts

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.

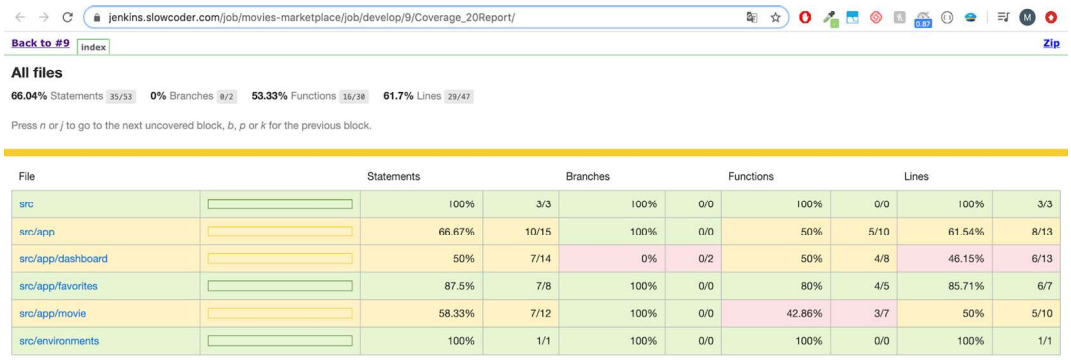


Figure 8.30 Coverage statistics by filename

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.