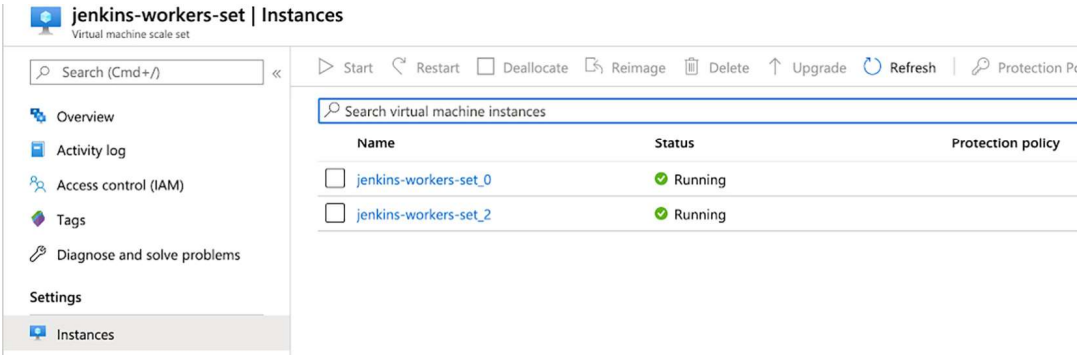


**Figure 6.32 Static number of Jenkins workers**

By default, two Jenkins workers will be up and running, as shown in figure 6.32.

To be able to scale workers based on build jobs and pipeline running, we will use Azure autoscale policies to trigger a scale-out or scale-in based on CPU utilization of the worker machines. Within jenkins_workers.tf, add the following `resource` block.

**Listing 6.34 Jenkins worker autoscaling policies**

```
resource "azurerm_monitor_autoscale_setting" "jenkins_workers_autoscale" {
  name                = "jenkins-workers-autoscale"
  resource_group_name = data.azurerm_resource_group.management.name
  location            = var.location
  target_resource_id  =
     azurerm_virtual_machine_scale_set.jenkins_workers_set.id

  profile {
    name = "jenkins-autoscale"
    capacity {
      default = 2
      minimum = 2
      maximum = 10
    }
    rule {
      metric_trigger {
        metric_name        = "Percentage CPU"
        metric_resource_id =
     azurerm_virtual_machine_scale_set.jenkins_workers_set.id
        time_grain         = "PT1M"
        statistic          = "Average"
        time_window        = "PT5M"
        time_aggregation   = "Average"
        operator           = "GreaterThan"
        threshold          = 80
      }
      scale_action {
        direction = "Increase"
        type      = "ChangeCount"
        value     = "1"
        cooldown  = "PT1M"
      }
    }
```

**Defines the minimum and maximum numbers of Jenkins workers**

**Monitors the CPU utilization of the workers—if it hits 80%, a new Jenkins worker's VM will be deployed.**

```
    rule {
      metric_trigger {
        metric_name        = "Percentage CPU"
        metric_resource_id =
     azurerm_virtual_machine_scale_set.jenkins_workers_set.id
        time_grain         = "PT1M"
        statistic          = "Average"
        time_window        = "PT5M"
        time_aggregation   = "Average"
        operator           = "LessThan"
        threshold          = 20
      }

      scale_action {
        direction = "Decrease"
        type      = "ChangeCount"
        value     = "1"
        cooldown  = "PT1M"
      }
    }
  }
}
```

**Monitors the CPU utilization of the workers—if it's below 20%, an existing Jenkins worker VM will be terminated.**

Apply the changes with `terraform apply`. Then, head over to the Jenkins worker scale set configuration. In the Scaling section, define a new autoscale policy, as shown in figure 6.33.

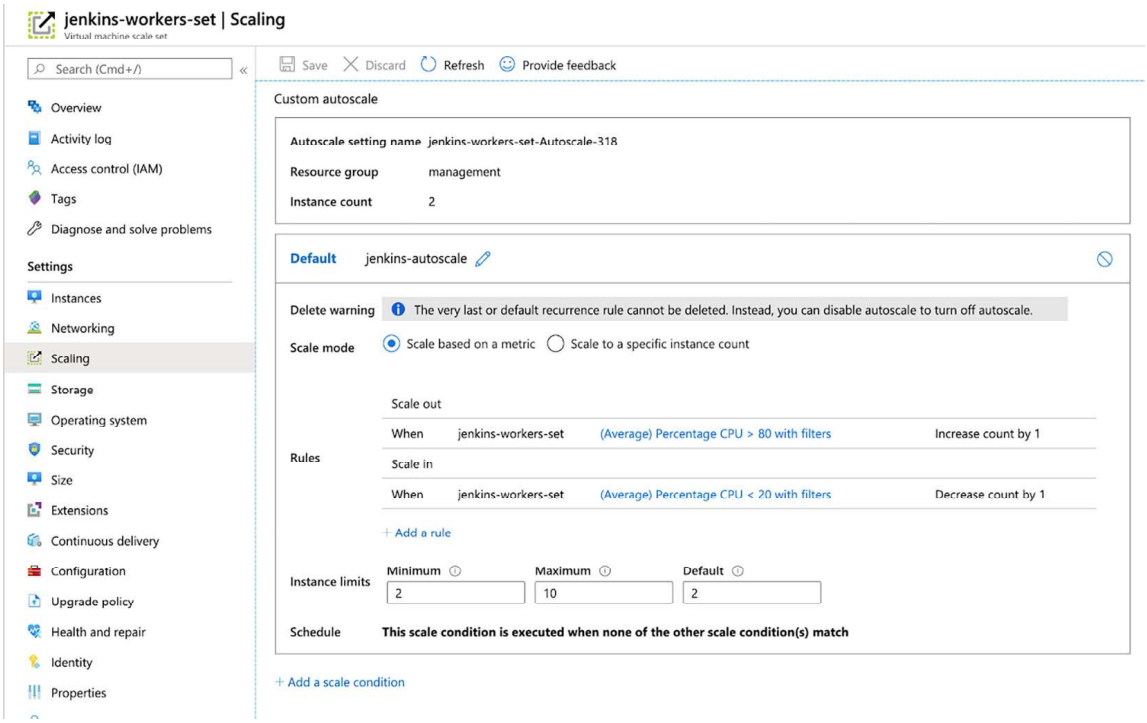


**Figure 6.33 Jenkins worker autoscaling policies**

**NOTE** Once you're finished playing with the Jenkins cluster, you will likely want to tear down everything that was created so that you don't incur any further costs.

Great! You are now able to deploy a self-healing Jenkins cluster on Microsoft Azure.

## 6.3 DigitalOcean

When we think of cloud computing providers, we are typically referring to the three giants in the industry: Azure, Google Cloud, and AWS. Unlike those providers that are known to everyone, DigitalOcean (www.digitalocean.com) is relatively new. You might be wondering why you should choose DigitalOcean over other providers. The reason lies in the differences between the three big players and DigitalOcean.

They differ in many aspects. One is small, while the others (AWS, GCP, and Azure) are huge. DigitalOcean provides virtual machines (called *Droplets*). There are no bells and whistles. You do not get lost in a catalog of services, since they are almost nonexistent. Plus, DigitalOcean's interface allows developers to quickly set up machines because of its friendly design. Moreover, it's affordable and has cheaper instances, which is a good starting point for beginner businesses and startups. (If you don't have a DigitalOcean account, you will need to create one; you will get $100 of free credits.)

To use Packer with DigitalOcean, we first need to generate a DigitalOcean API token. This can be done on the DigitalOcean Applications & API page. Click the Generate New Token button to obtain a token with read and write permissions, as shown in figure 6.34.

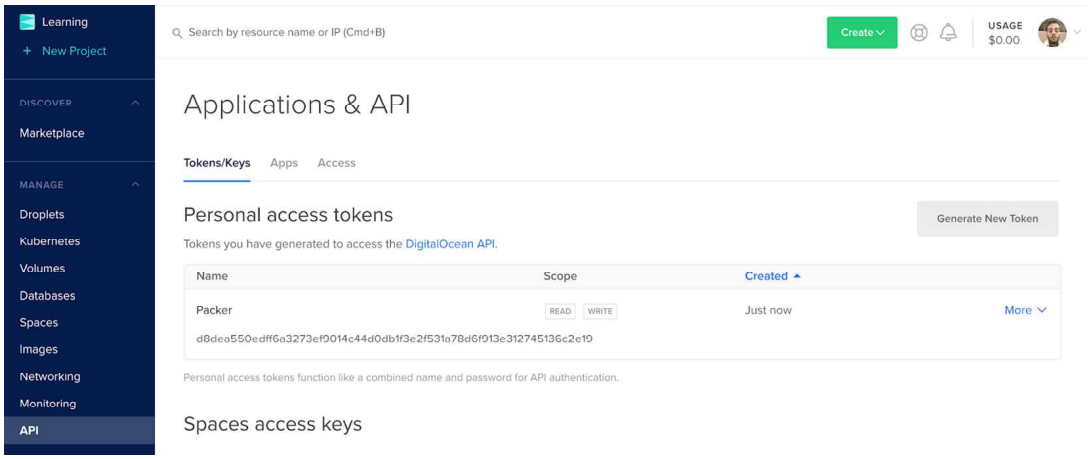


**Figure 6.34 Packer API access token**

### 6.3.1 Creating Jenkins DigitalOcean Snapshots

We're using the same template covered in listings 6.1 and 6.2; the only difference is the use of the `digitalocean` Packer builder to interact with the DigitalOcean API. The builder takes a CentOS source image and runs the provisioning necessary—installing

the tools required for building Jenkins jobs on the image after launching it—and then snapshots it into a reusable image; see the following listing. This reusable image can then be used as the foundation of new Jenkins workers that are launched within DigitalOcean by using Terraform.

**Listing 6.35 Jenkins worker image with DigitalOcean builder**

```
{
    "variables" : {
        "api_token" : "DIGITALOCEAN API TOKEN",
        "region": "DIGITALOCEAN REGION"
    },
    "builders" : [
        {
            "type": "digitalocean",
            "api_token": "{{user `api_token`}}",
            "image": "centos-8-x64",
            "region": "{{user `region`}}",
            "size": "512mb",
            "ssh_username": "root",
            "snapshot_name": "jenkins-worker"
        }
    ],
    "provisioners" : [
        {
            "type" : "shell",
            "script" : "./setup.sh",
            "execute_command" : "sudo -E -S sh '{{ .Path }}'"
        }
    ]
}
```

DigitalOcean API token and target region

The build Droplet will be based on CentOS 8.

Include your DigitalOcean API token and target region (refer to the official documentation for a list of supported regions: http://mng.bz/EDRJ). Then run the `packer build template.json` command. You'll get a working Jenkins worker image in your DigitalOcean account in a couple of minutes, as shown in figure 6.35.

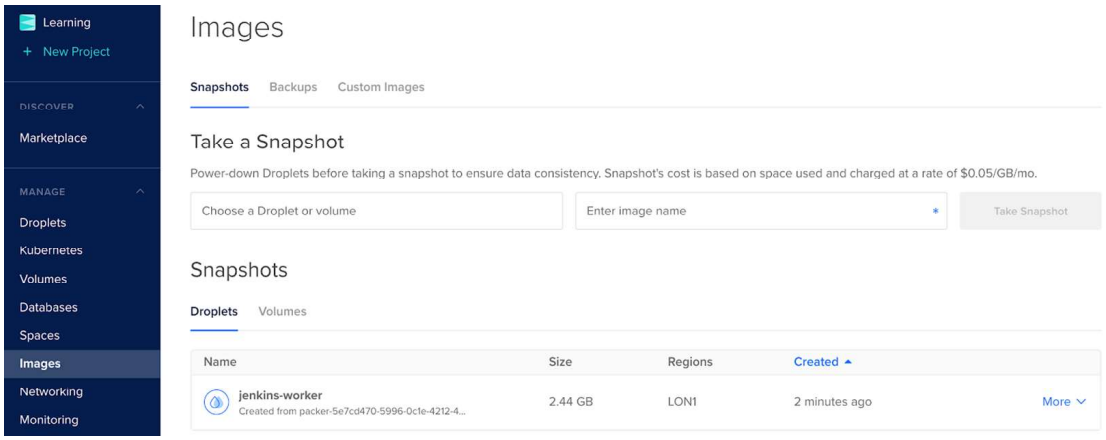


**Figure 6.35 Jenkins worker image snapshot**

Similarly, update the Jenkins master template referenced in listing 6.2 to use the `digitalocean` builder. The provisioning part creates a Jenkins credential based on a private SSH key used to deploy Jenkins workers. This is needed, as Jenkins needs to set up a bidirectional connection with workers via SSH.

**Listing 6.36 Jenkins master image with DigitalOcean builder**

```
{
    "variables" : {
        "api_token" : "DIGITALOCEAN API TOKEN",
        "region": "DIGITALOCEAN REGION",
        "ssh_key" : "PRIVATE SSH KEY FILE"
    },
    "builders" : [
        {
            "type": "digitalocean",
            "api_token": "{{user `api_token`}}",
            "image": "centos-8-x64",
            "region": "{{user `region`}}",
            "size": "2gb",
            "ssh_username": "root",
            "snapshot_name": "jenkins-master-2.204.1"
        }
    ],
    "provisioners" : [
        ...
    ]
}
```

This template has been cropped for brevity. The full JSON file can be downloaded from chapter6/digitalocean/packer/master/template.json.

Run the `packer validate` command to make sure that everything is copacetic. Then issue a `packer build` command. Once the build and provisioning part is finished, the Jenkins master snapshot should be ready to be used, as shown in figure 6.36.

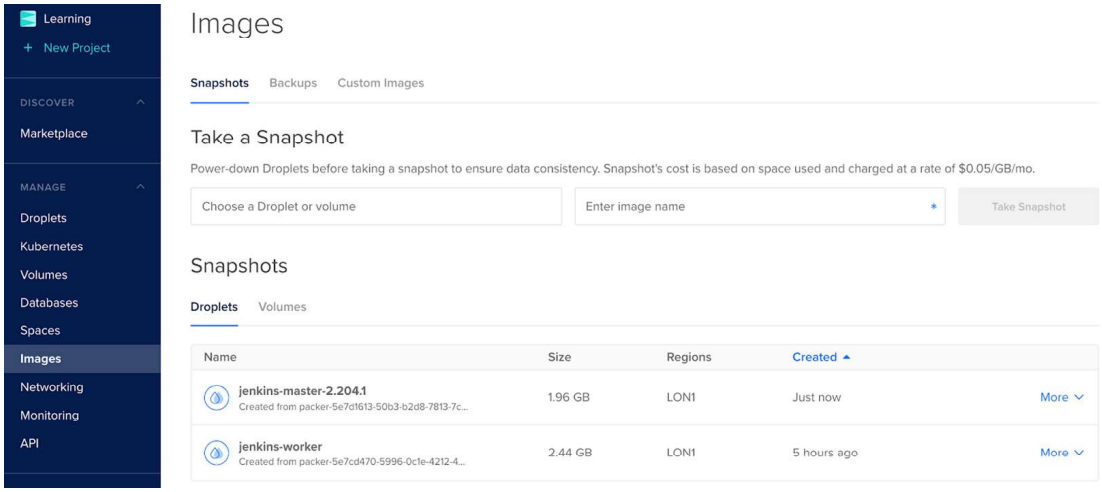


**Figure 6.36 Jenkins master image snapshot**

### 6.3.2 Deploying a Jenkins master Droplet

In this step, you'll write Terraform template files for automating Jenkins cluster Droplet deployments of the snapshot containing the Jenkins master and worker you just built using Packer.

Define a terraform.tf file and declare DigitalOcean as a provider. The provider needs to be configured with the proper API token before it can be used, as shown in the following listing.

**Listing 6.37 Defining the DigitalOcean provider**

```
provider "digitalocean" {
  token = var.token
}
```

Run `terraform init` to download the DigitalOcean plugin needed to translate the Terraform instructions into API calls:

```
Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "digitalocean" (terraform-providers/digitalocean) 1.15.1...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.digitalocean: version = "~> 1.15"

Terraform has been successfully initialized!
```

Define a single resource of the type `digitalocean_droplet` named `jenkins-master` in the jenkins_master.tf file, as shown in Listing 6.38. Then set its parameters according to the variable values and add an SSH key (using its fingerprint) from your DigitalOcean account to the Droplet resource. The deployed Droplet will be of type `s-1vcpu-2gb`, which comes up with 1 GB of RAM and 1vCPU.

For heavier workloads and larger projects, and to handle concurrent users connecting to the Jenkins web dashboard, a large Droplet type might be required. Refer to the official documentation for the list of available Droplet sizes: http://mng.bz/N4yD.

**Listing 6.38 Jenkins master Droplet**

```
data "digitalocean_image" "jenkins_master_image" {
  name = var.jenkins_master_image
}
```

```
resource "digitalocean_droplet" "jenkins_master" {
  name   = "jenkins-master"
  image  = data.digitalocean_image.jenkins_master_image.id
  region = var.region
  size   = "s-1vcpu-2gb"
  ssh_keys = [var.ssh_fingerprint]
}
```

Uses the Jenkins master image backed previously with Packer

Provisions a Droplet with 2 GB of RAM and 1vCPU

On DigitalOcean, you can upload your SSH public key to your account, which lets you add it to your Droplets at creation time (figure 6.37). This lets you log in to your Jenkins master without a password while still remaining secure.

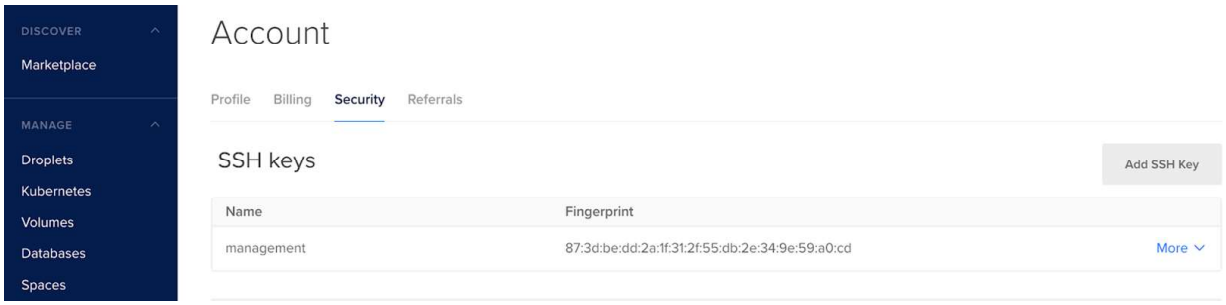


**Figure 6.37 Adding a public SSH key**

Next, attach a firewall to the Jenkins master Droplet with rules allowing inbound traffic on port 22 and 8080 from anywhere; see the following listing. For security purposes, I recommend limiting SSH incoming traffic to your CIDR network block.

**Listing 6.39 Jenkins master Droplet's firewall**

```
resource "digitalocean_firewall" "jenkins_master_firewall" {
  name = "jenkins-master-firewall"

  droplet_ids = [digitalocean_droplet.jenkins_master.id]

  inbound_rule {
    protocol         = "tcp"
    port_range       = "22"
    source_addresses = ["0.0.0.0/0", "::/0"]
  }

  inbound_rule {
    protocol         = "tcp"
    port_range       = "8080"
    source_addresses = ["0.0.0.0/0", "::/0"]
  }
```

Allows inbound traffic on port 22 (SSH) from anywhere

Allows inbound traffic on port 8080, where the Jenkins web dashboard is served from

```
  outbound_rule {
    protocol              = "tcp"
    port_range            = "1-65535"
    destination_addresses = ["0.0.0.0/0", "::/0"]
  }

  outbound_rule {
    protocol              = "udp"
    port_range            = "1-65535"
    destination_addresses = ["0.0.0.0/0", "::/0"]
  }

  outbound_rule {
    protocol              = "icmp"
    destination_addresses = ["0.0.0.0/0", "::/0"]
  }
}
```

**Allows outbound traffic on all ports from anywhere**

Paste the following code to the outputs.tf file to display the IP address of the Jenkins master Droplet when the deployment is complete.

**Listing 6.40 Jenkins master public IP address**

```
output "master" {
  value = digitalocean_droplet.jenkins_master.ipv4_address
}
```

Define the Terraform variables listed in table 6.3 in a new variable.tf file. Set their values in variables.tfvars to keep secrets and sensitive information out of template files.

**Table 6.3 DigitalOcean Terraform variables**

| Name | Type | Value | Description |
|---|---|---|---|
| `token` | String | None | This is the DigitalOcean API token. Alternatively, this can also be specified using `DIGITALOCEAN_TOKEN` environment variables. |
| `region` | String | None | The DigitalOcean region in which deploy the Jenkins master. |
| `jenkins_master_image` | String | None | The name of the Jenkins master image that was built previously with Packer. |
| `ssh_fingerprint` | String | None | SSH ID or fingerprint. To retrieve the info, head to the DigitalOcean Security dashboard. |

Run the `terraform plan` command to see the effect of the deployment before execution:

```
# digitalocean_droplet.jenkins_master will be created
+ resource "digitalocean_droplet" "jenkins_master" {
    + backups              = false
    + created_at           = (known after apply)
    + disk                 = (known after apply)
    + id                   = (known after apply)
    + image                = "61197862"
    + ipv4_address         = (known after apply)
    + ipv4_address_private = (known after apply)
    + ipv6                 = false
    + ipv6_address         = (known after apply)
    + ipv6_address_private = (known after apply)
    + locked               = (known after apply)
    + memory               = (known after apply)
    + monitoring           = false
    + name                 = "jenkins-master"
    + price_hourly         = (known after apply)
    + price_monthly        = (known after apply)
    + private_networking   = false
    + region               = "lon1"
    + resize_disk          = true
    + size                 = "s-1vcpu-2gb"
    + ssh_keys             = [
        + "87:3d:be:dd:2a:1f:31:2f:55:db:2e:34:9e:59:a0:cd",
      ]
    + status               = (known after apply)
    + urn                  = (known after apply)
    + vcpus                = (known after apply)
    + volume_ids           = (known after apply)
  }

Plan: 1 to add, 0 to change, 0 to destroy.
```

You can now move on to validating and deploying it on a Droplet with a `terraform apply` command. The deployment process should take a few seconds to finish. Then a new Jenkins master Droplet will be available in the Droplets console, and Terraform should display the IP address of the Jenkins master Droplet, as you can see in figure 6.38.

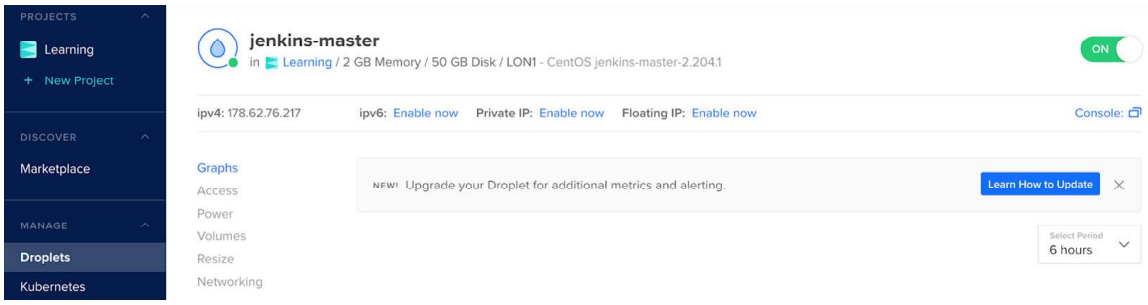


**Figure 6.38 Jenkins master Droplet**

Open your favorite browser and connect to the public IPv4 that was returned by the previous command. A preconfigured Jenkins dashboard should be displayed; see figure 6.39.

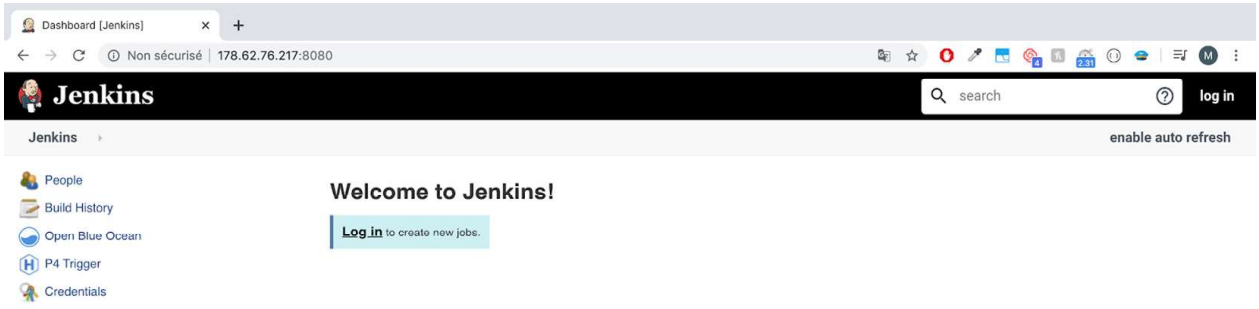


**Figure 6.39 Jenkins dashboard access with Droplet public IP**

### 6.3.3 *Building Jenkins worker Droplets*

Now to delegate build jobs to workers and offload the Jenkins master Droplet. Several build workers will be deployed to absorb the build activity.

Create a jenkins_workers.tf file where you define Jenkins worker Droplets. The workers will be launched from the Jenkins worker image.

**Listing 6.41 Jenkins worker Droplets**

```
data "digitalocean_image" "jenkins_worker_image" {
  name = var.jenkins_worker_image
}

data "template_file" "jenkins_worker_startup_script" {
  template = "${file("scripts/join-cluster.tpl")}"

  vars = {
    jenkins_url            = "http://
     ${digitalocean_droplet.jenkins_master.ipv4_address}:8080"
    jenkins_username       = var.jenkins_username
    jenkins_password       = var.jenkins_password
    jenkins_credentials_id = var.jenkins_credentials_id
  }
}
resource "digitalocean_droplet" "jenkins_workers" {
  count = var.jenkins_workers_count
  name   = "jenkins-worker"
  image  = data.digitalocean_image.jenkins_worker_image.id
  region = var.region
  size   = "s-1vcpu-2gb"
  ssh_keys = [var.ssh_fingerprint]
  user_data = data.template_file.jenkins_worker_startup_script.rendered
  depends_on = [digitalocean_droplet.jenkins_master]
}
```

The script is used to make the Droplet autojoin the cluster as a Jenkins agent/worker.

Indicates the number of Jenkins workers to create

In this Droplet configuration, we're using 1 GB of RAM and 1vCPU as configuration for Jenkins workers.

The launch script is passed in the user_data section so it can be executed the first time the Droplet is running.

The `count` variable is used to define the number of workers to deploy. Each Droplet will execute a shell script at startup. This script is similar to the one provided in previous sections, except for the use of the DigitalOcean metadata server to fetch the Droplet IP address and hostname:

```
INSTANCE_NAME=$(curl -s http://169.254.169.254/metadata/v1/hostname)
INSTANCE_IP=$(curl -s http://169.254.169.254/metadata/v1/
interfaces/public/0/ipv4/address)
```

Finally, to set up a bidirectional connection between Jenkins master and workers, we define a firewall allowing inbound traffic on TCP port 22.

**Listing 6.42 Jenkins worker firewall**

```
resource "digitalocean_firewall" "jenkins_workers_firewall" {
  name = "jenkins-workers-firewall"

  droplet_ids =
[for worker in digitalocean_droplet.jenkins_workers : worker.id   ]

  inbound_rule {
    protocol         = "tcp"
    port_range       = "22"
    source_droplet_ids = [digitalocean_droplet.jenkins_master.id]
  }
}
```

Allows the Jenkins master to SSH to the Jenkins workers

After a few minutes, the workers' Droplets will finish provisioning, and you'll see output similar to figure 6.40.

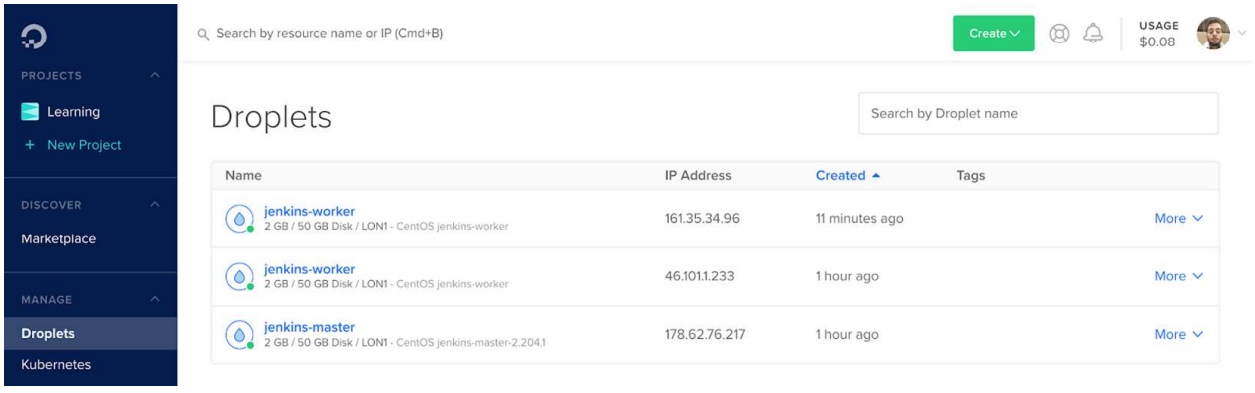


**Figure 6.40 Jenkins worker Droplets**

Go back to the Jenkins dashboard. The new deployed workers should join the cluster after executing the user data script covered in chapter 5's listing 5.7; see figure 6.41.

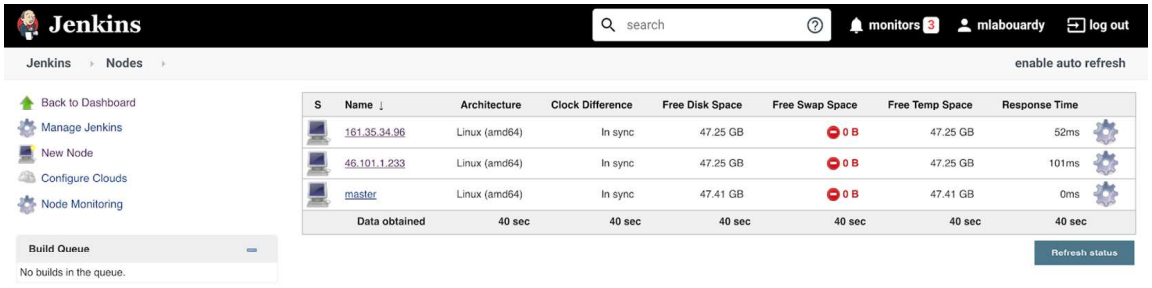


**Figure 6.41 Worker Droplets joining the cluster**

You can take this architecture further by deploying a load balancer in front of the Jenkins master Droplet to forward traffic to port 8080 and creating a DNS record pointing to the load balancer FQDN; see figure 6.42.

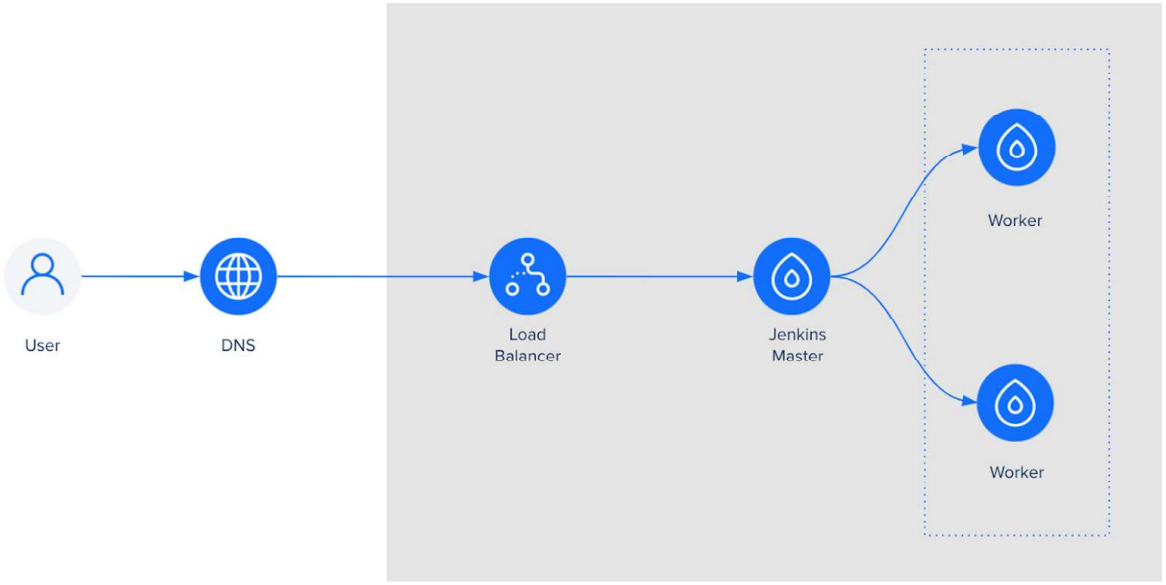


**Figure 6.42 Jenkins cluster architecture on DigitalOcean**

When you're finished, clean up the infrastructure by running the following:

```
terraform destroy --var-file=variables.tfvars
```

This chapter has covered how to deploy and operate a resilient and self-healing Jenkins cluster from scratch on numerous cloud providers with IaC tools. I've also explained how to architect Jenkins workers for scale with autoscaling policies and metrics alarms. In the next chapter, we will implement pipelines as code on Jenkins for numerous cloud-native applications such as Dockerized microservices and serverless applications.

## Summary

- The power of Packer comes from leveraging template files to create identical Jenkins machine images independently of the target platform.
- Deploying Jenkins on Google Cloud Platform comes with seamless native support for Kubernetes.
- Azure offers a variety of cloud-based services and might be a good alternative for running Jenkins on the cloud.
- Running Jenkins on DigitalOcean can be a cost-efficient solution for beginner businesses and startups.

# 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.

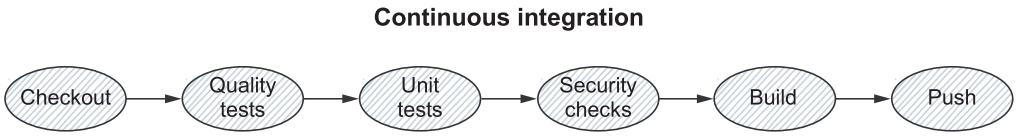


**Figure 7.1 Continuous integration stages**

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.

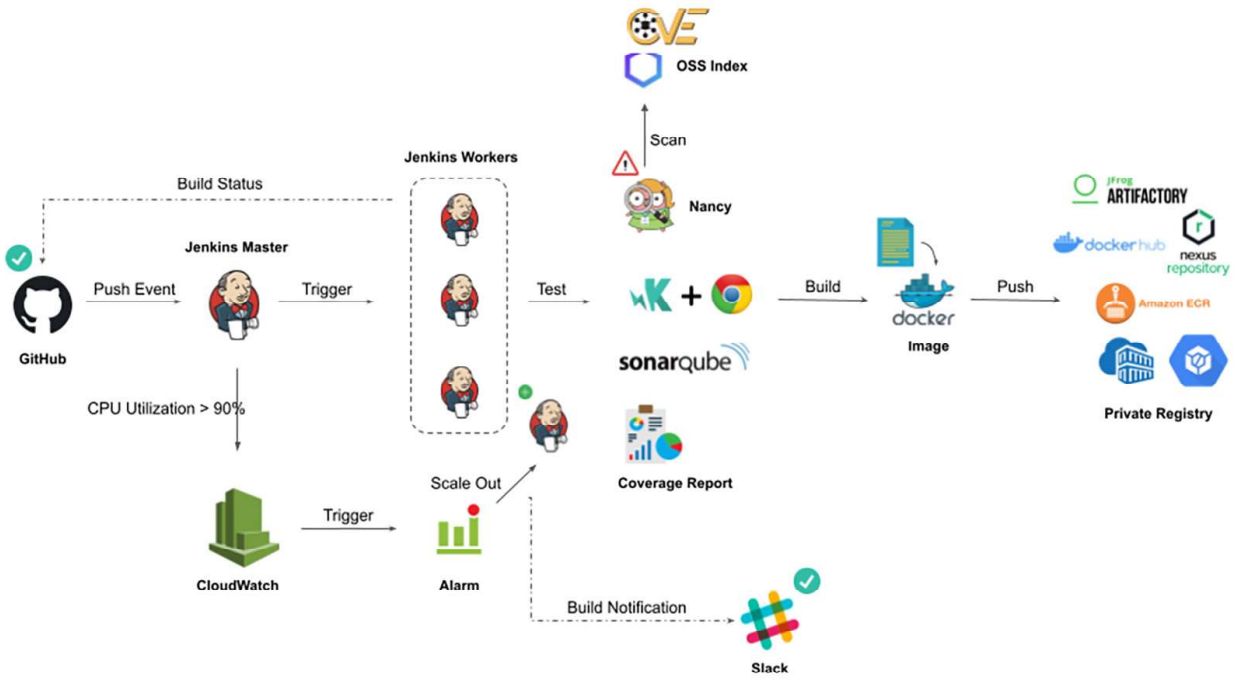


**Figure 7.2 Target CI pipeline**

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.

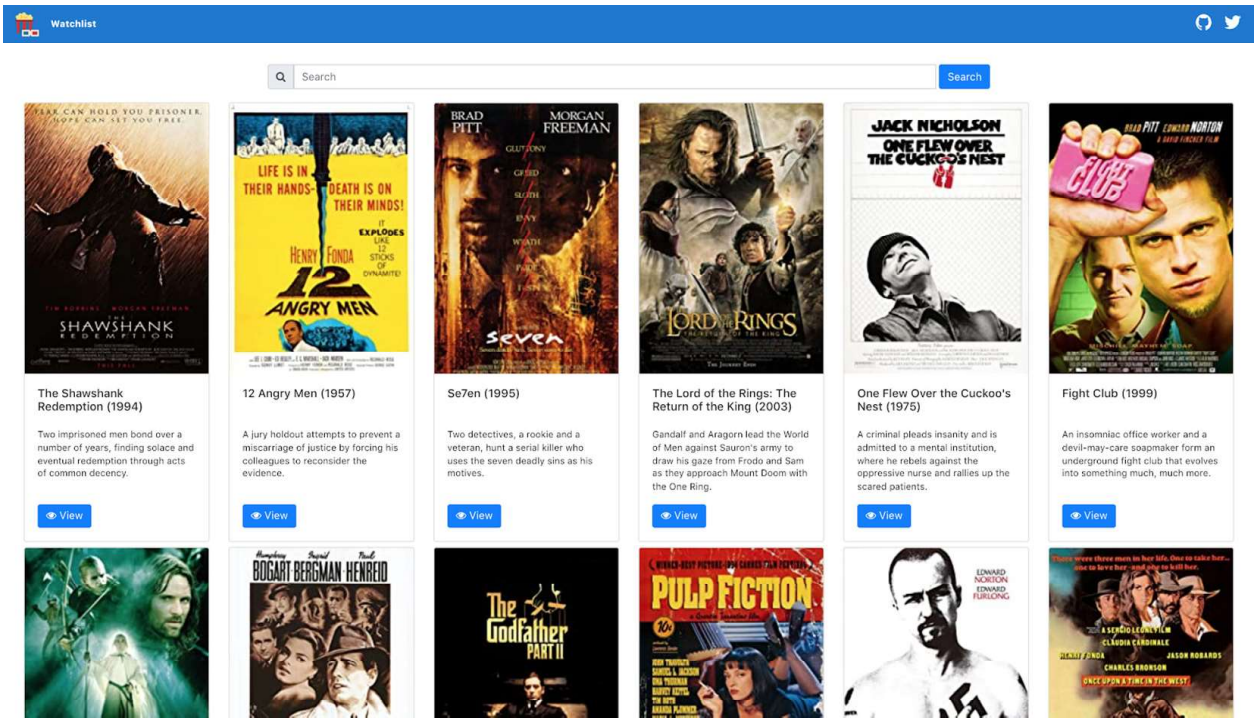


Figure 7.3 Watchlist marketplace UI

Figure 7.4 illustrates the application architecture and flow.

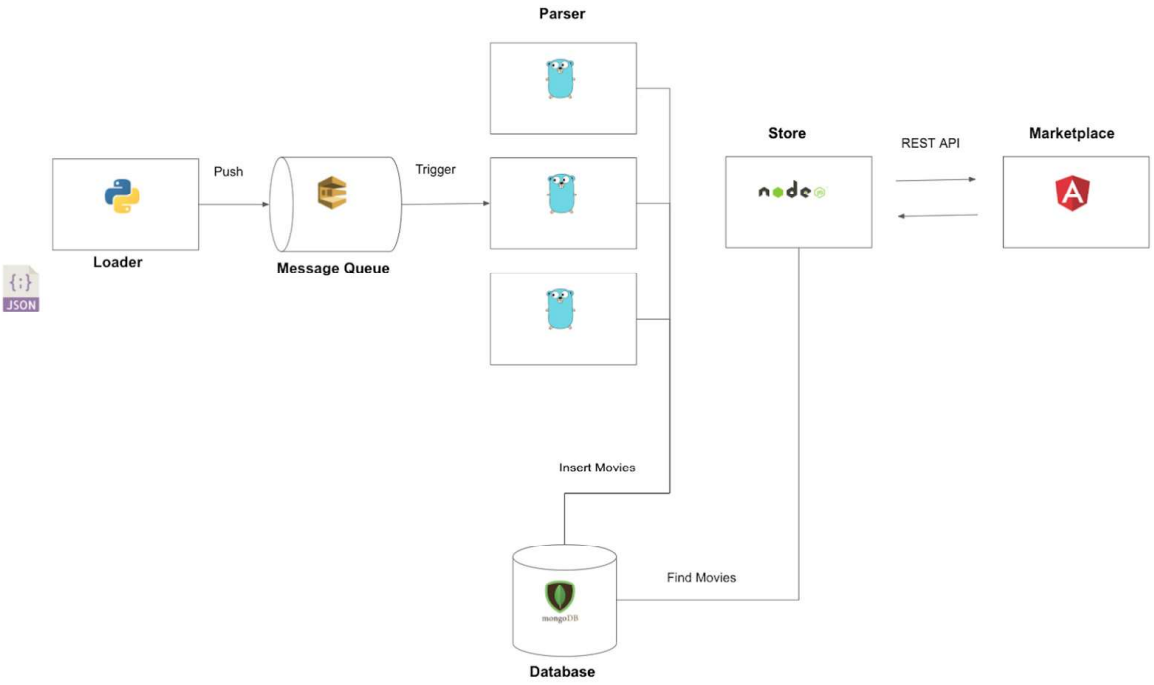


**Figure 7.4 The Loader service takes an array of movies in JSON format and forwards them one by one to a message queue (for example, Amazon SQS). From there, a Parser service will consume the items and fetch the movie's details from the IMDb database and save the result into MongoDB. Finally, the data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

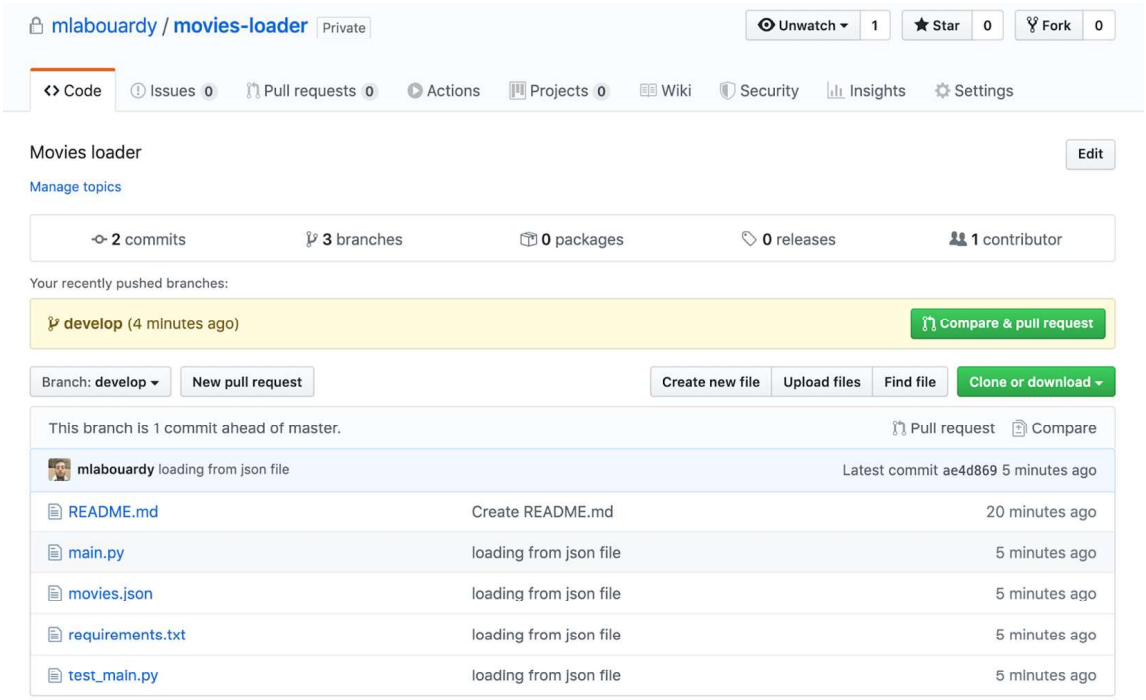


**Figure 7.5 The Loader GitHub repository has the service source's code.**

**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.

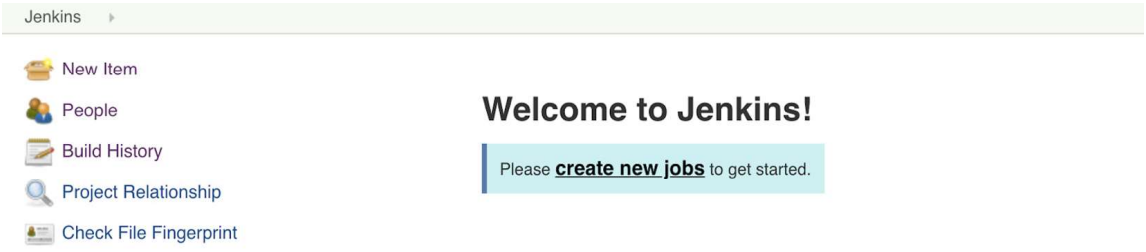


**Figure 7.6 Jenkins new job creation**

**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.

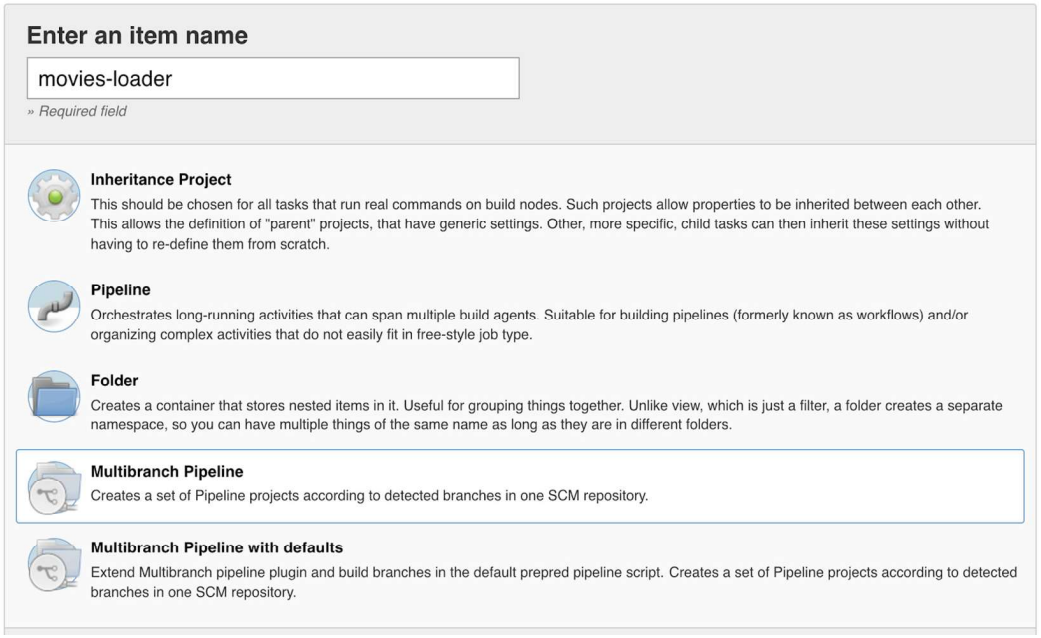


**Figure 7.7 Jenkins new job settings**

**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.

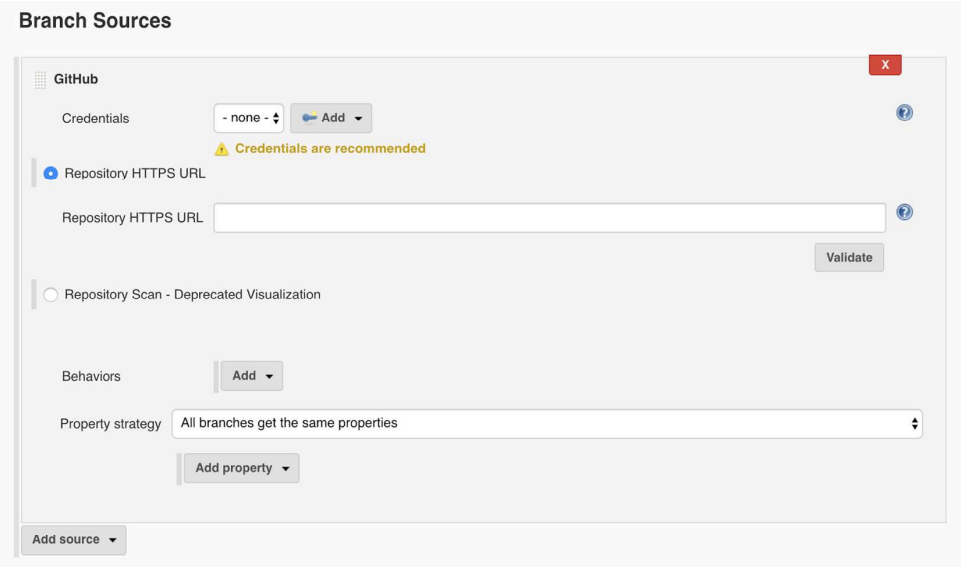


Figure 7.8 Branch Sources configuration

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.

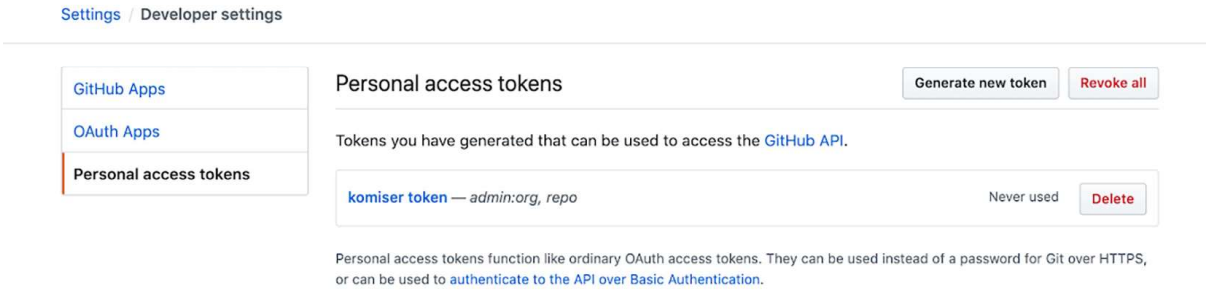


**Figure 7.10 GitHub personal access tokens**

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.

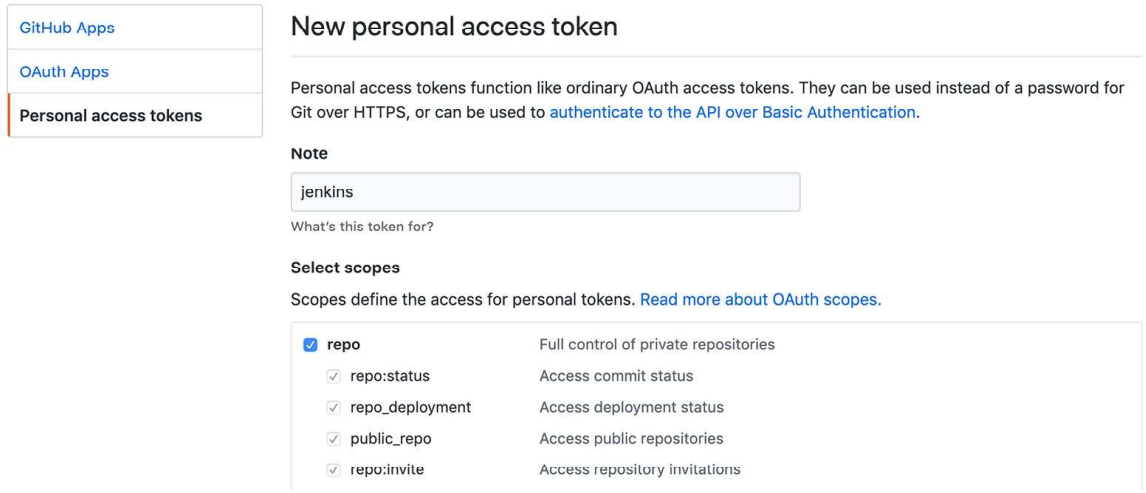


**Figure 7.11 Jenkins dedicated token for GitHub access**

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.

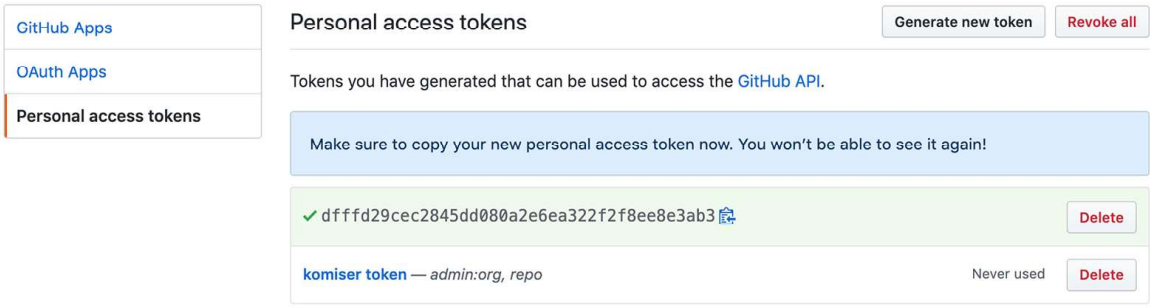


**Figure 7.12 Jenkins personal access token**

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.

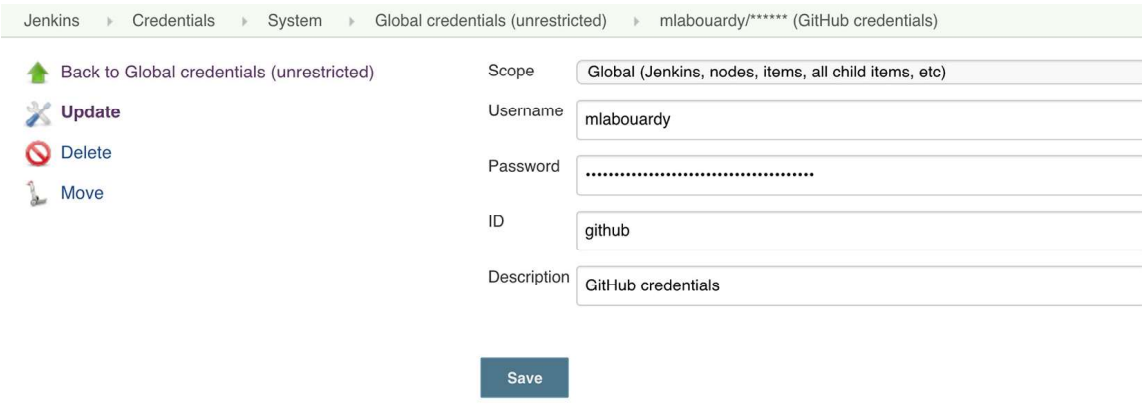


**Figure 7.13 GitHub credentials configuration on Jenkins**

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.

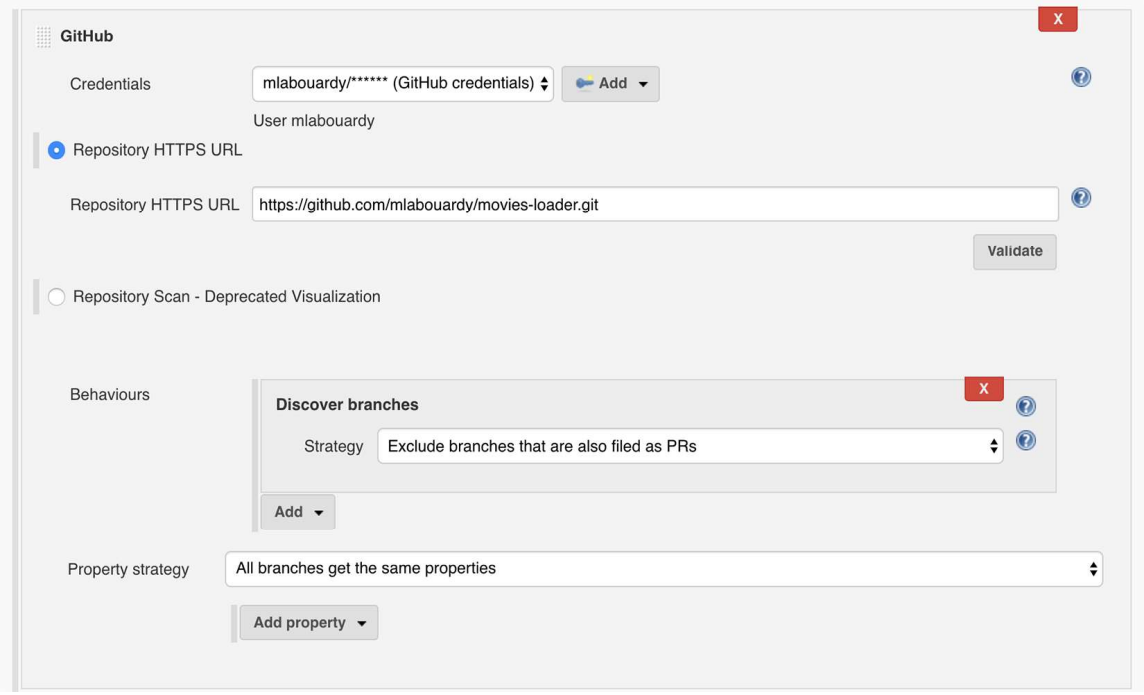


**Figure 7.14 GitHub repository configuration on Jenkins**

**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.

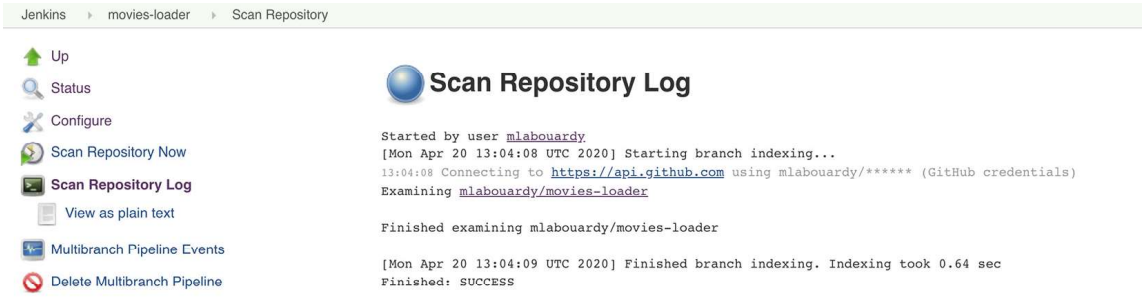


**Figure 7.15 Jenkins repository scanning logs**

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.

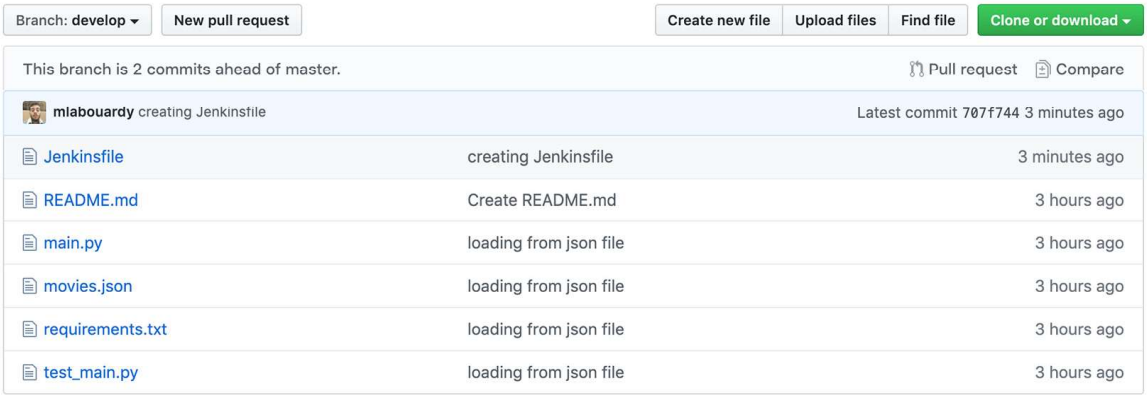


**Figure 7.16 Jenkinsfile is stored along with source code**

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.

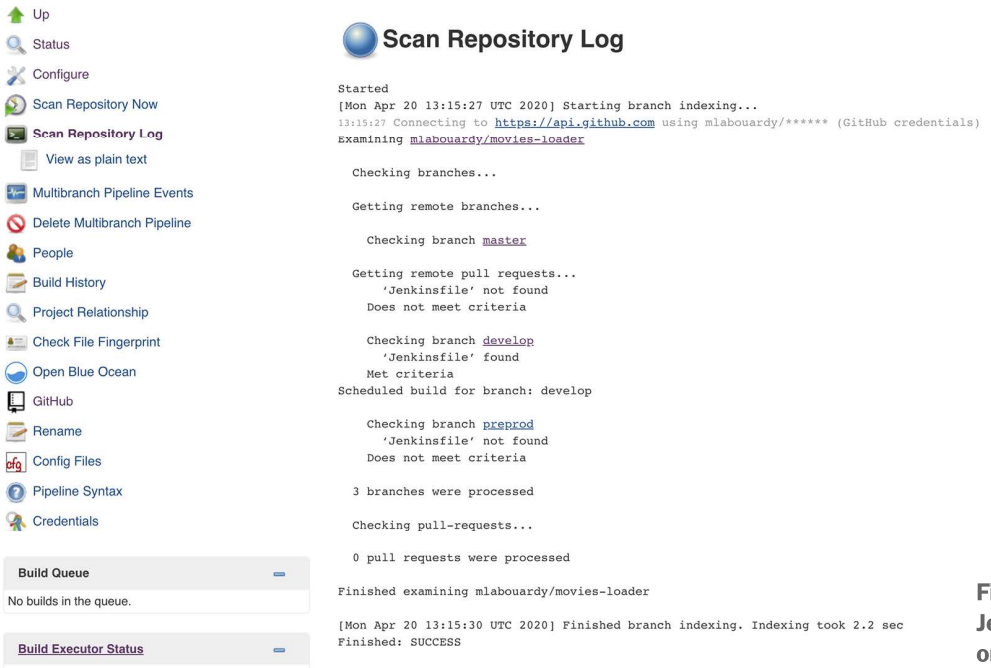


**Figure 7.17 Jenkinsfile detected on develop branch**

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.

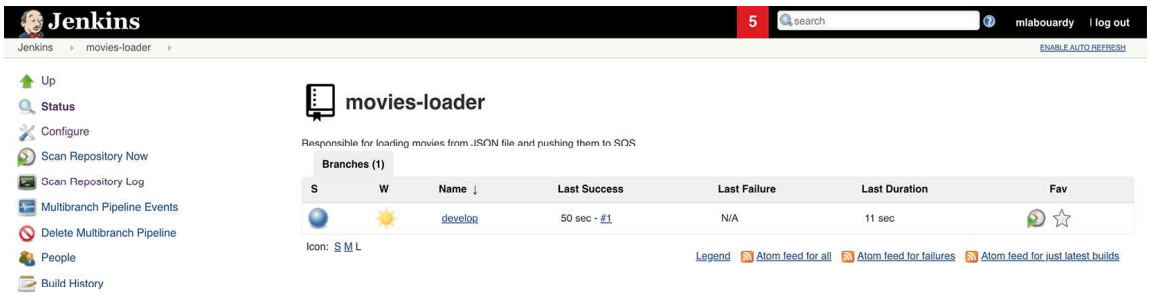


**Figure 7.18 Build job triggered on the develop branch**

**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.

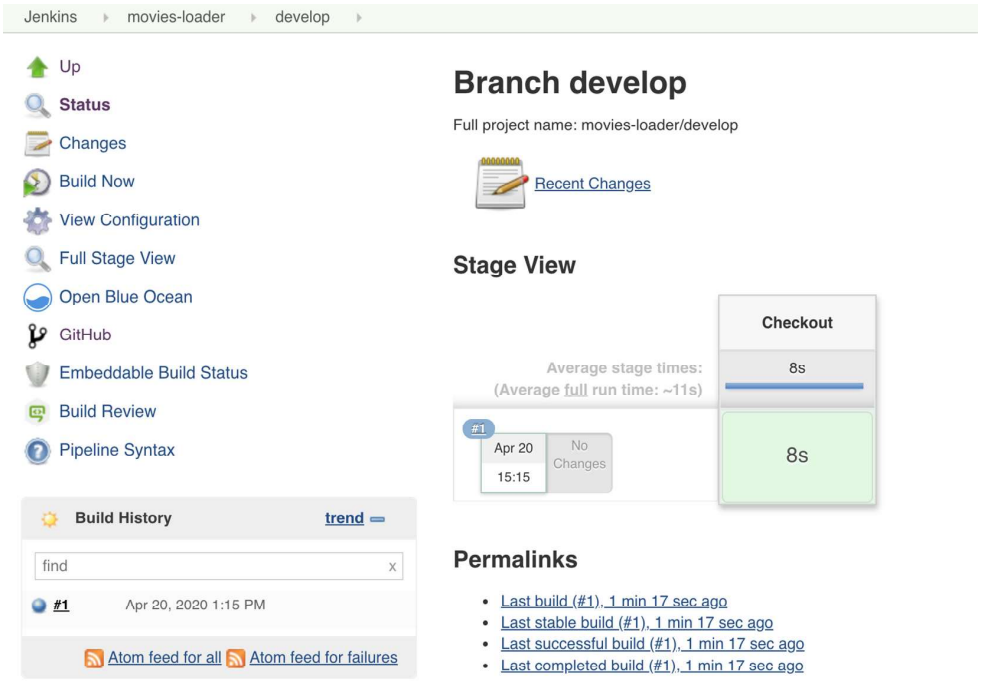


**Figure 7.19 Pipeline execution**

**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.

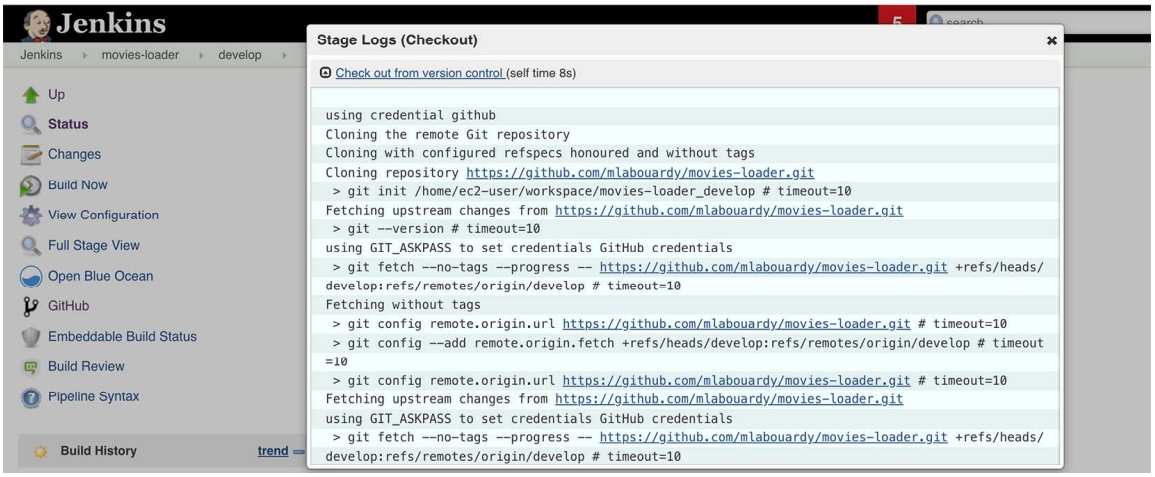


**Figure 7.20 Checkout stage logs**

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.

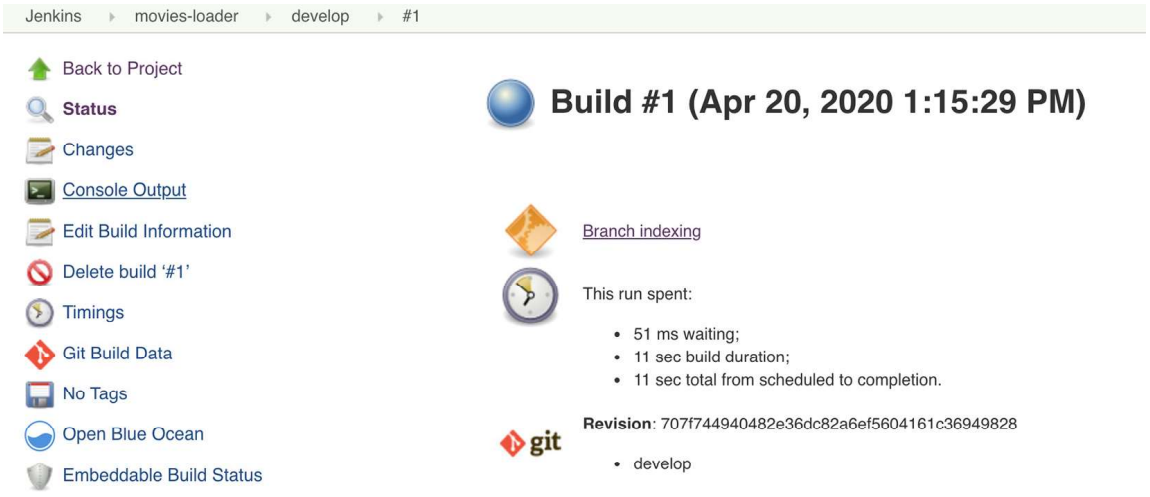


**Figure 7.21 Build number settings**

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

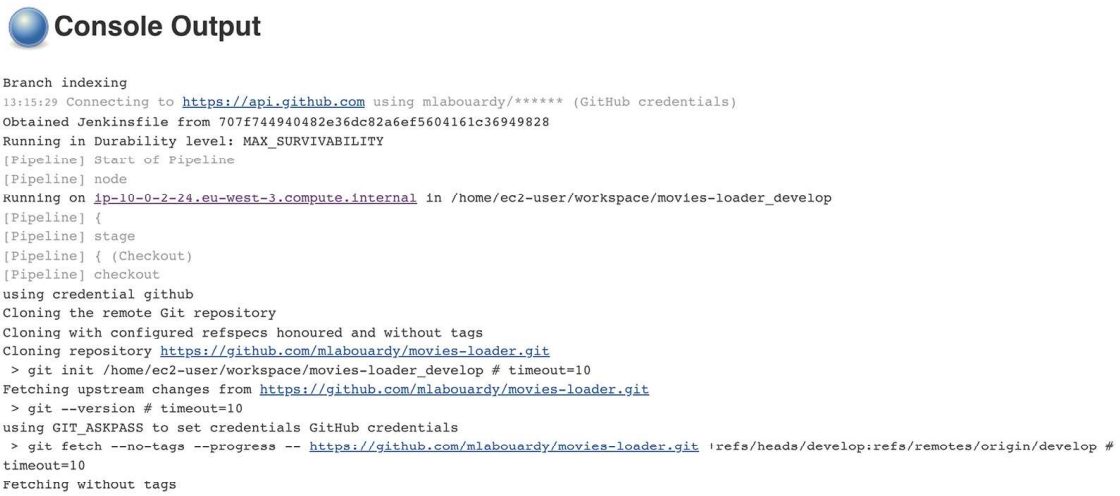


**Figure 7.22 Build console logs**

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.

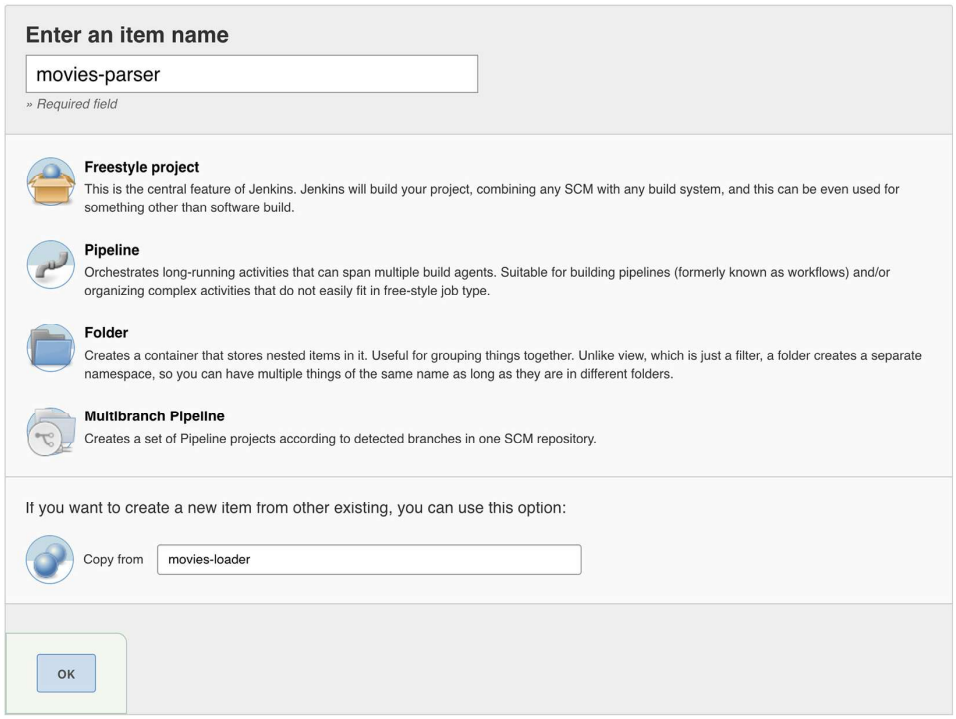


**Figure 7.23 Parser job's creation**

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.

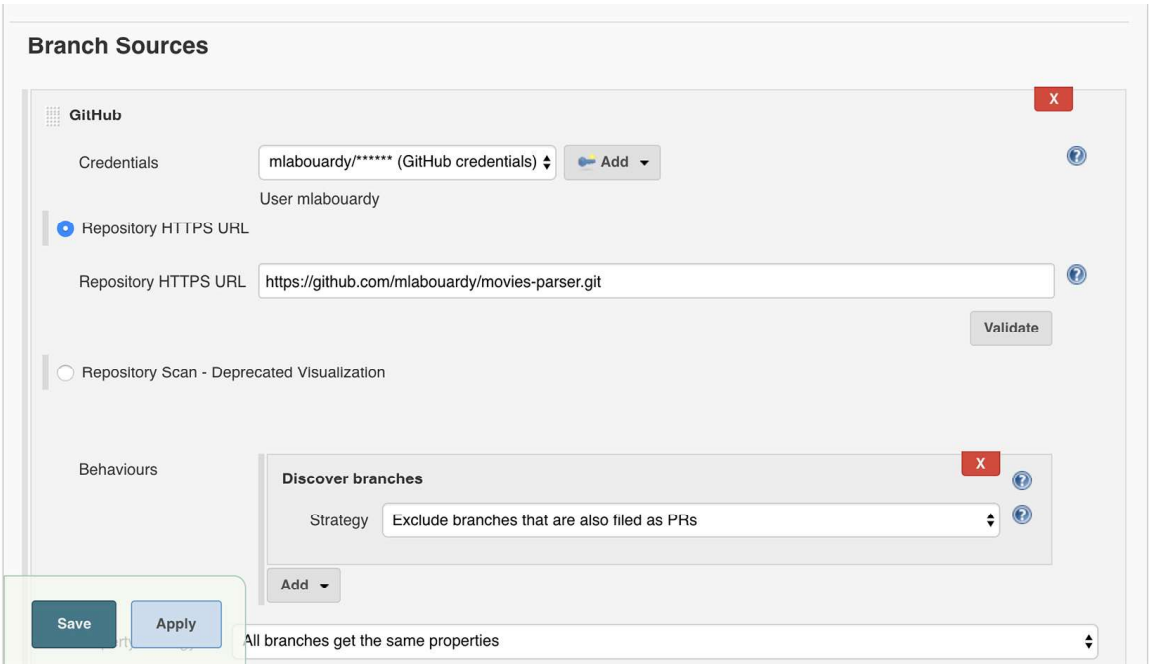


**Figure 7.24 Parser job GitHub configuration**

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.

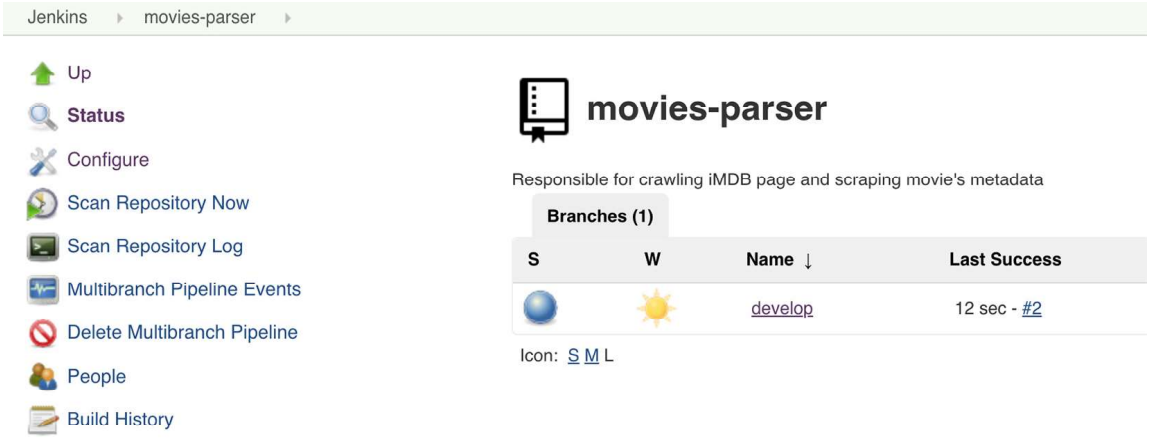


**Figure 7.25 Parser job list of active branches**

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.

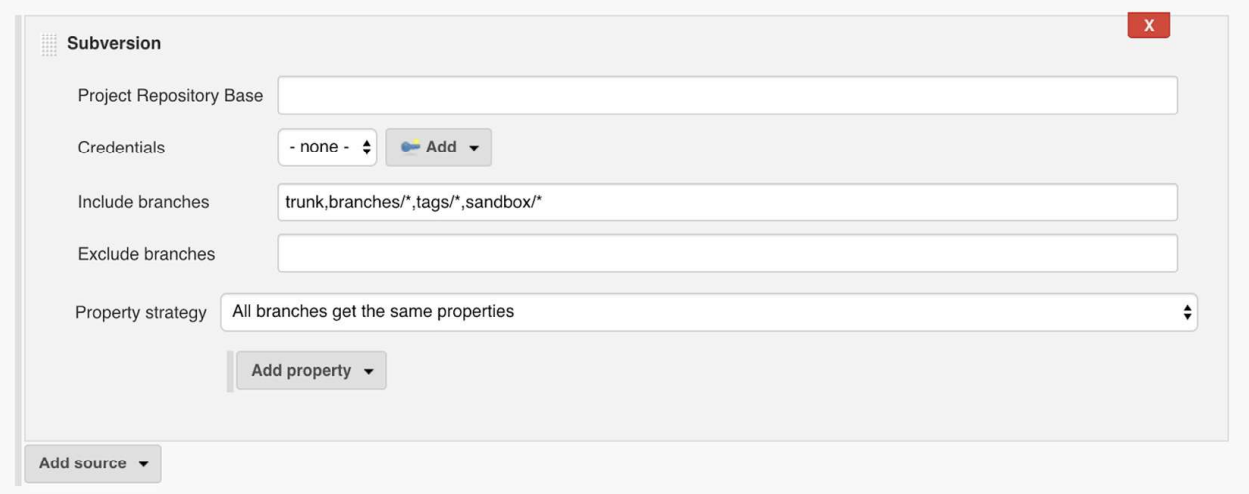


Figure 7.26 SVN repository configuration

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.

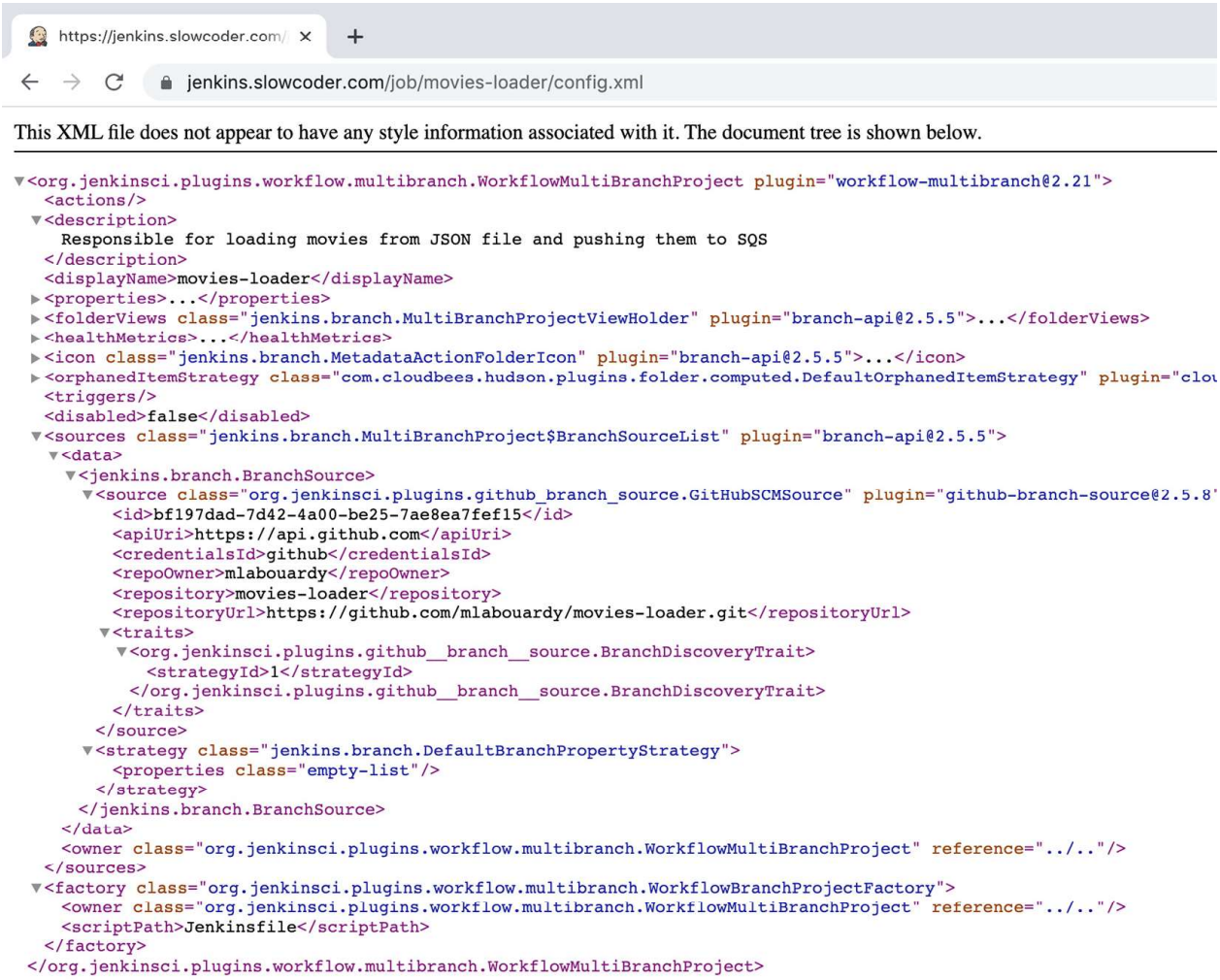


**Figure 7.27 Job XML configuration**

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.

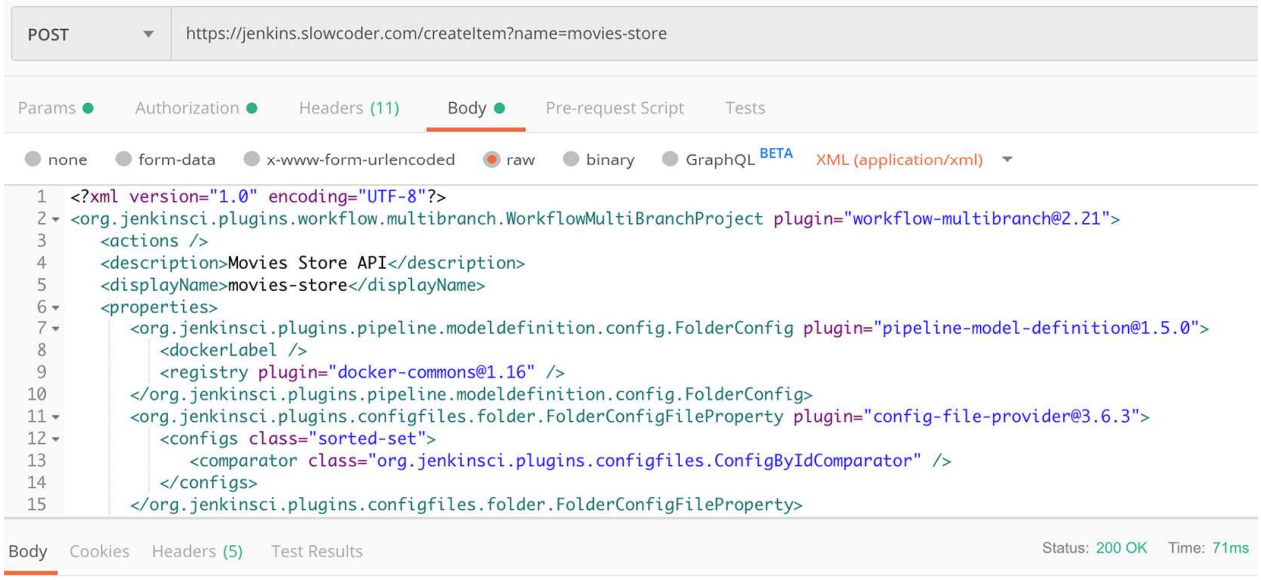


**Figure 7.28 Job creation Jenkins RESTful API with Postman**

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.

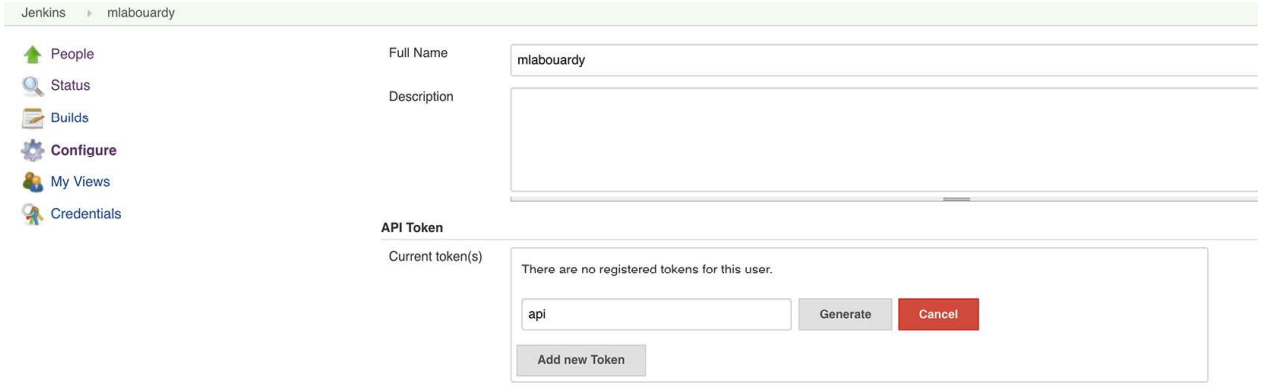


**Figure 7.29 Jenkins API token generation**

**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.

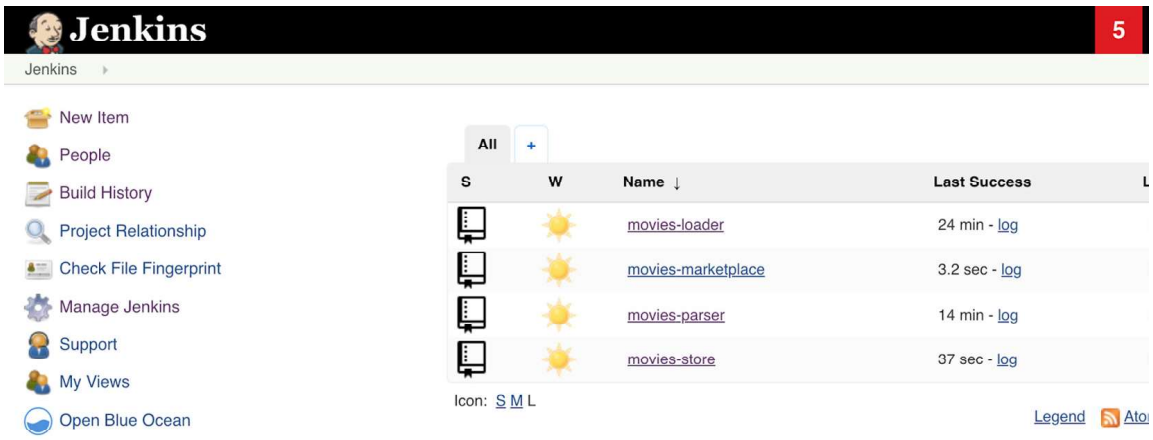


**Figure 7.30 Microservices jobs in Jenkins**

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.

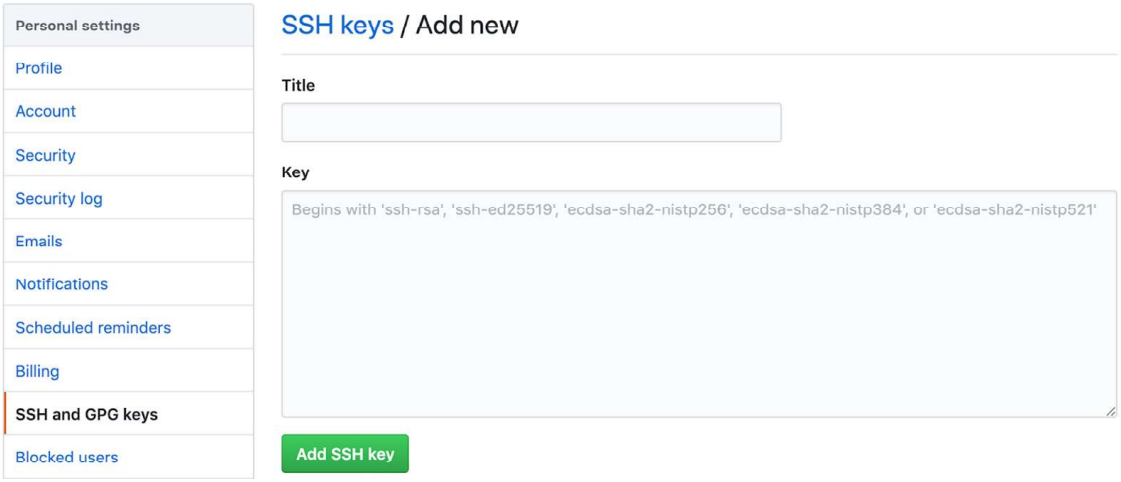


**Figure 7.31 GitHub SSH configuration**

> **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.