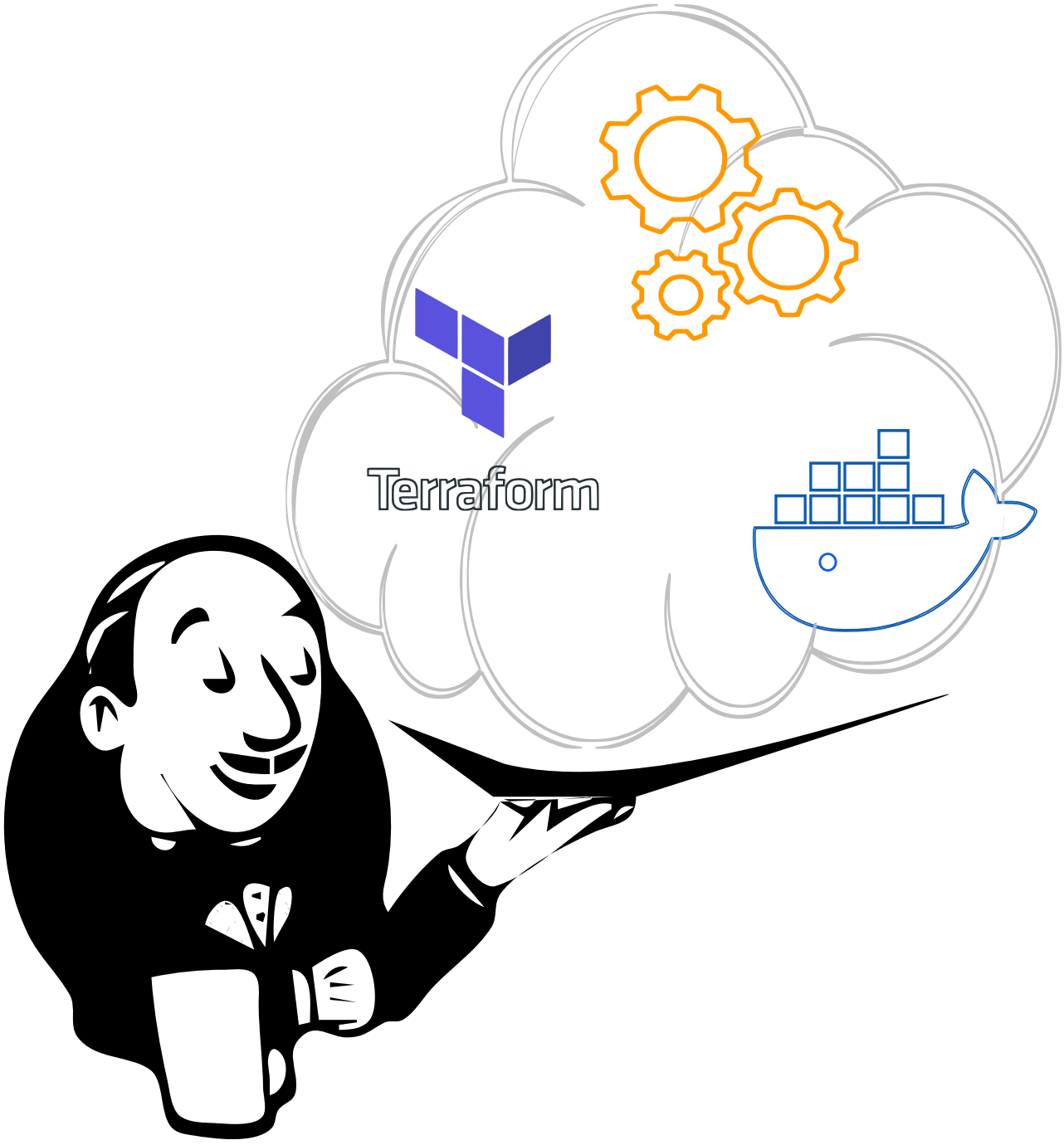
Mastek

Jenkins-serviced CI/CD for Cloud Platforms



Author: Aji Sreekumar (Technical Architect)

COPYRIGHT

Author Bio: Aji Sreekumar is an information technology professional with 18 years of experience in Java/J2ee and related technologies. Currently, she is a Java/Cloud Technical Architect at Mastek UK. In addition to technology, she is passionate about employing automation to improve process maturity and efficiency at all levels in the software development process.

Mastek UK Ltd, Pennant House, 2 Napier Court, Reading RG1 8BW
Mastek UK Ltd (Leeds), 4th Floor, 36 Park Row, Leeds, LS1 5JL, United Kingdom
Mastek US, 15601 Dallas Parkway, Suite 250, Dallas, TX 75001
Mastek Ltd (Seepz), 106, SDF IV,Seepz, Andheri (East), Mumbai – 400 096

Contents

1 INTRODUCTION

The DevOps terminology has grown from a mere buzzword to an indispensable process methodology for many organizations. It presents varied significance to different users; however, in a general sense 'DevOps' implies a composite software delivery team using automation for improved quality and quick turnaround of the build and deployment process. An established DevOps reference architecture is shown in Figure 1

The most important points to address here are: what to automate, what technologies to be employed, and what processes the DevOps framework is to fulfil. These questions are of varied priorities to the many adopters. Despite this, DevOps aims to gain feedback more often, fast track builds and deployments, facilitate quick recovery in the event of failure, and gives more reliable and faster notifications of errors.

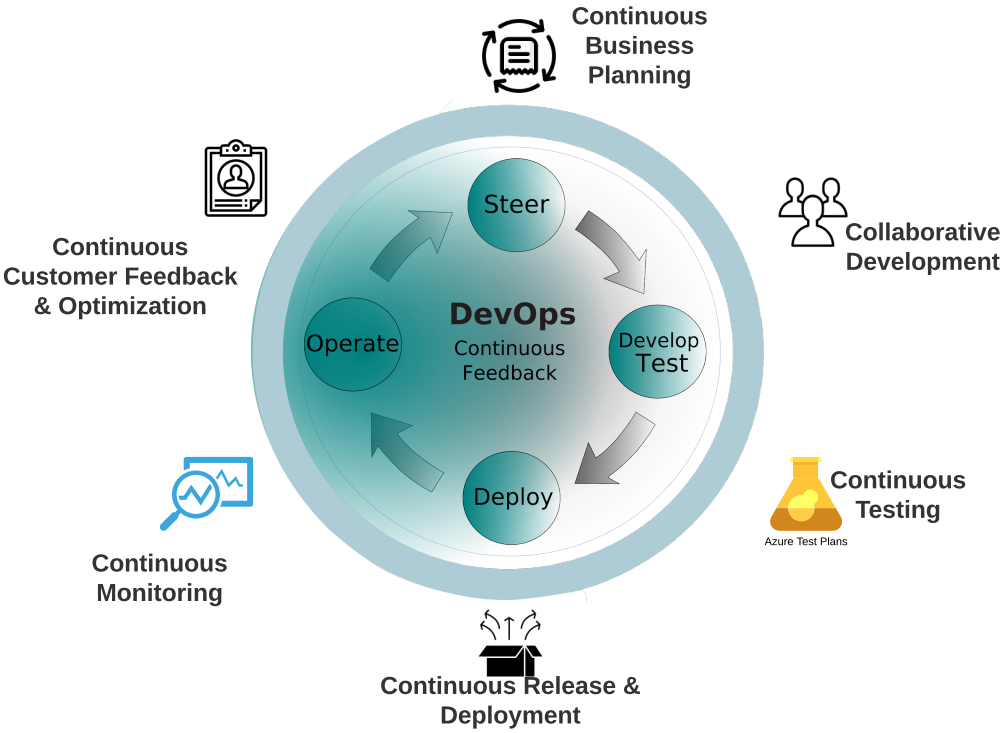


Figure 1: DevOps reference architecture [1]

The "traditional environment team" structure adopted by medium and low maturity organizations consist of separate development and operations teams, with operations performed manually or through individual scripts for every

activity. Each change in version necessitates a change of configuration values at 'n' places with more mature teams exploiting the power of single configuration files at the very best.

For organizations that plan, say, once in a quarter or so deployments, these processes are adequate. However, for today's businesses with production releases expected even weekly, only mature CI/CD processes can handle the pace. The overheads caused by the manual trial and errors, misconfigurations, the cost of time and effort in rebuilding and re-deploying applications is proportionately enormous. This is very happily eliminated in a CI/CD mature organization.

The Cloud platforms have ushered in an era of soft infrastructure that can be spun up and destroyed at will for near-zero cost. They also handle the scaling of applications to manage the load, which was an architect's nightmare, in the past. All cloud platforms: AWS, GCP and Azure offer seamless integration of CI/CD technologies for the best of ease build and deployment.

'Infrastructure as code' is the ability to 'provision' the necessary infrastructure just as a developer would spin up an application. Once the required configurations are in place, the subsequent step is to deploy it. The term 'provisioning' creates the virtual environment on the cloud platform, pooling the required mix of CPU computing and memory requirements, specified in the code.

In the overall scheme of software project development, three tools play a significant role to streamline the processes: Jenkins, Terraform, and Docker. While Terraform provisions infrastructure through code, Docker builds the application code into container images and deploys them to the cloud. Jenkins orchestrates the activities of the other two through its pipelines as shown in Figure 2.

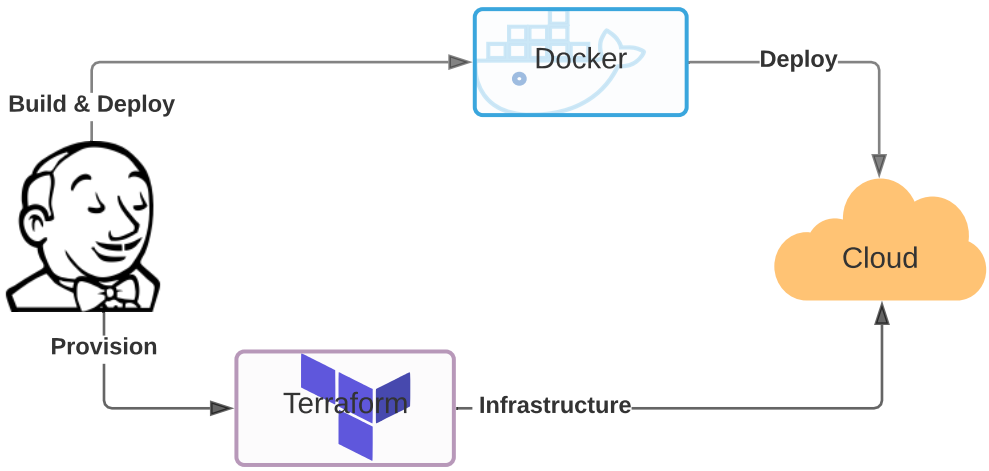


Figure 2: Conceptual view of the pipelines

This short book serves as a design and configuration document; it is also used as a readyreckoner. The entire code base required for the CI/CD engine is presented and is reusable with minimum changes.

1.1 Objectives

The objective is to create an engine with these tech stacks on AWS, to prove the concept and also share the learning to a broader audience. Anyone attempting to achieve CI/CD with this stack, gets a head start with the necessary scripts and configurations shared herein.

On a longer-term, the CI/CD engine discussed here can:

1. Automate build and deployment of applications
2. Automate provisioning of cloud infrastructure
3. Improve DevOps maturity in the organization

1.2 Document Structure

This book is a guide to implement the CI/CD pipeline for a cloud environment; but for demonstration an AWS environment and is outlined, as shown in Figure 3.

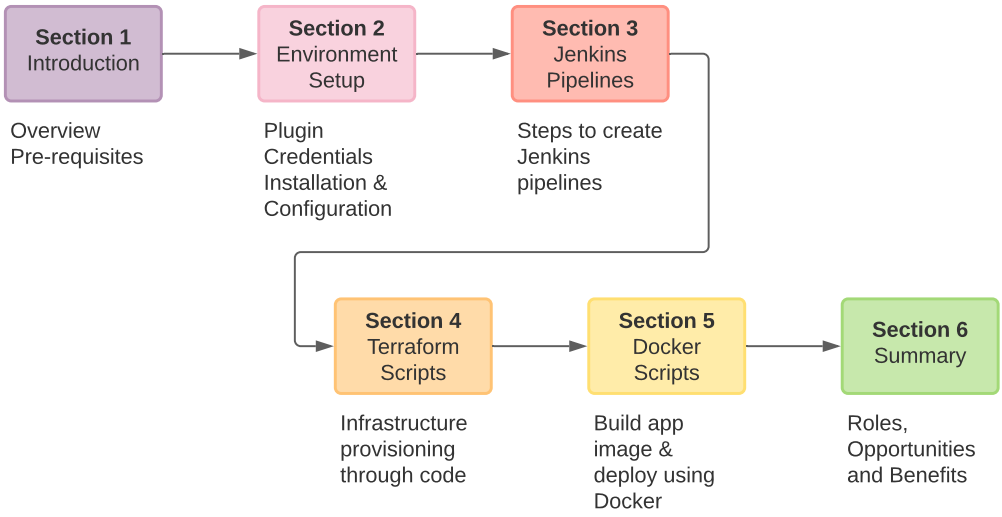


Figure 3: Book Layout

- **Section 1** gives an overview of the technologies and the objectives of this exercise.
- **Section 2** provides guidance on the required plugins and libraries needed to get the environment setup and their configurations.
- **Section 3** covers the process of creating Jenkins pipelines in a series of steps. Continuing on from the previous section, this section also covers configuring the different tools and giving credentials to authenticate Jenkins to trigger the other services as appropriate.
- **Section 4** details the provisioning of the infrastructure on the AWS platform employing code with the help of Terraform.
- **Section 5** shows how to use Docker to manage the build and deploy the application code as images into the provisioned environment.
- **Section 6** discusses the challenges in brief and weighs the opportunities from using the automated procedure for provisioning infrastructure, build and deployment.

2 ENVIRONMENT SETUP

This section helps to get the Jenkins continuous integration (CI) servers configured. Jenkins provides hundreds of plugins to build, test and deploy applications. We cover the three most important aspects of setting up the CI server, here:

1. Installing Plugins
2. Configuring libraries for Jenkins to work with Terraform and Docker
3. Credential management setup to authorize Jenkins to access external tools and services

Jenkins Continuous Integration server can be deployed on-premises or on the cloud. The current experiment assumes this infrastructure to be on-premises. However, the approach is not much different from a cloud-based setup.

After evaluating the prerequisites, we go on to configure Jenkins to trigger the 'provisioning', build & deployment services as appropriate.

No	Tools	Version	Use
1	Maven plugin	3.6.3	Compile and package (jar/war) application code
2	Jenkins	2.249.3	Continuous Integration and Deployment Server
3	Docker plugin	1.25	Containerize the compiled and packaged code
4	AWS ECR plugin	1.6	Integrate Jenkins with AWS ECR repository
5	Terraform plugin	3.14.1	Run Terraform commands in the pipeline script
6	SonarQube Scanner	2.13	Generate Sonar report in the sonar server

2.1 Pre-requisites

1. Access to Jenkins service
2. Access to a Cloud account (here, AWS)
3. Access to Software Configuration management tool- Git
4. A suitable IDE to work with Jenkins script (Visual Studio code preferred)
5. Application code checked into a Git repository with Dockerfile and Jenkinsfile in the root folder.

6. Access to Sonar server

Step 1

To begin, go to the Jenkins login page in your home installation. This book follows an on-prem Jenkins installation.

On the local machine, the login page is available on the "https://localhost:8081/login" as shown in Figure 4.

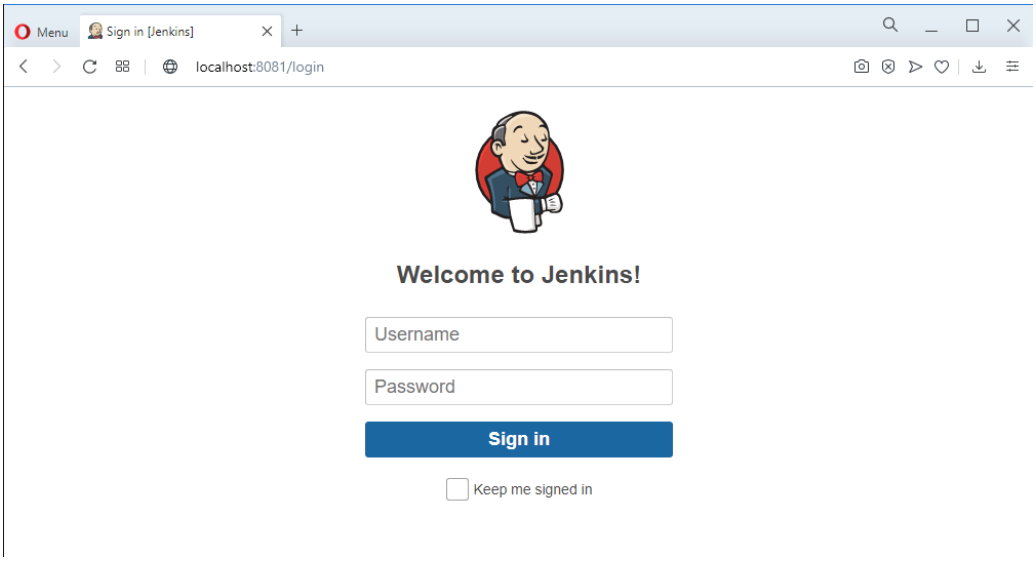


Figure 4: Jenkins login page

Step 2

a) Log on to Jenkins

The 'Jenkins Dashboard' gives options to create pipelines and manage Jenkins operations; shown in Figure 5.

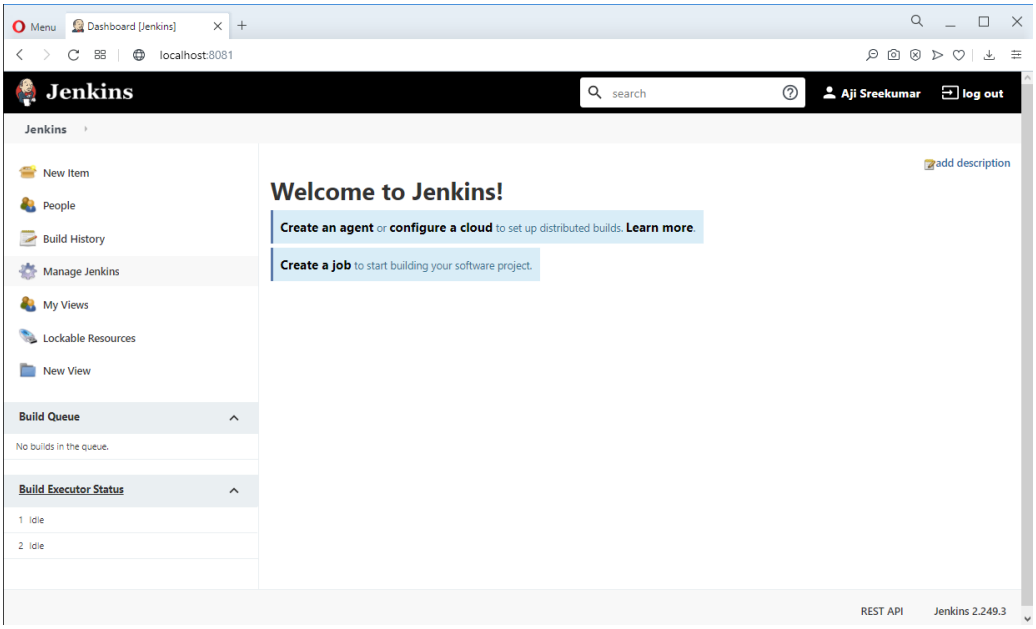


Figure 5: The Jenkins Dashboard

b) Click 'Manage Jenkins' menu on the left sidebar.

This takes the user to the 'Manage Jenkins' page as shown in Figure 6.

Many features are available in the Manage Jenkins page. The three important ones discussed here are:

1. **Manage Plugins:** This helps to install the required plugins/libraries for the pipeline implementation
2. **Global Tool Configurations:** This helps to configure the required libraries/tools
3. **Manage Credentials:** This helps to configure the required credentials to access external systems.

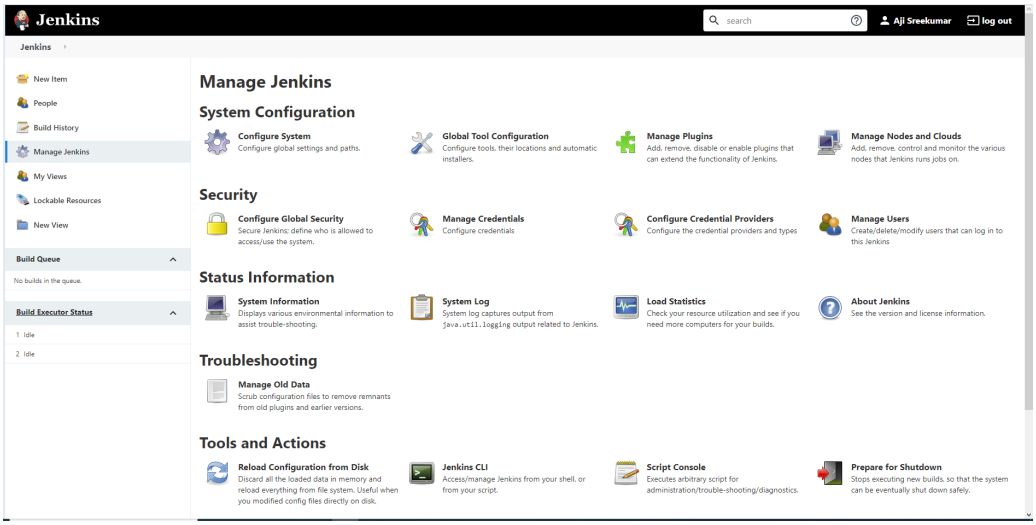


Figure 6: The Manage Jenkins page

Step 3

a) Click Manage Jenkins → Manage Plugins

The Plugins Manager page has a massive list of plugins. For the current setup, plugins related to Docker, Terraform, and Amazon-ECR are the ones to be installed. It is recommended to install plugins for each group one by one.

To limit the list, enter Docker in the search box to list the available plugins for Docker in the 'Available' tab. The partial list is seen in Figure 7.

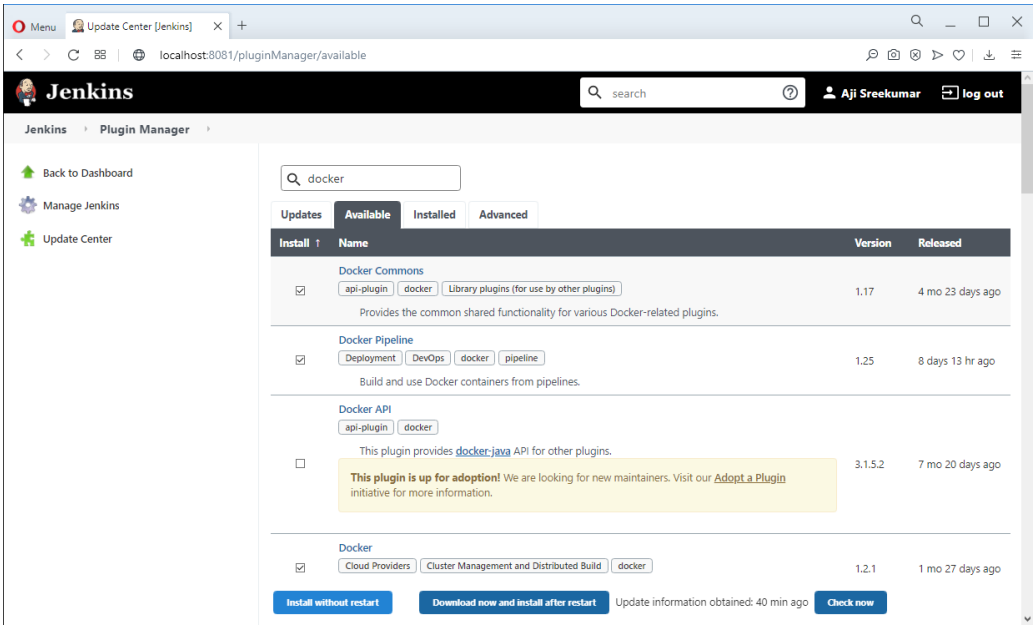


Figure 7: The Plugin Manager

In the same tab ('Available Plugins'), select the following plugins (not all of them are displayed in the image), and 'Install without a restart' :

1. Amazon-ECR
2. Terraform
3. Docker Commons
4. Docker Compose build setup
5. Docker Pipeline
6. Docker plugin
7. Docker Slaves Plugin
8. Docker Build Setup
9. SonarQube Scanner

b) Follow the previous step to select and install plugins for Amazon-ECR, Terraform and Sonar plugins → Install without restart

It will take a few minutes for the plugins to be installed. Once done, the installation is confirmed on 'Update Center' page, as shown in Figure 8.

Then, go to Step 4.

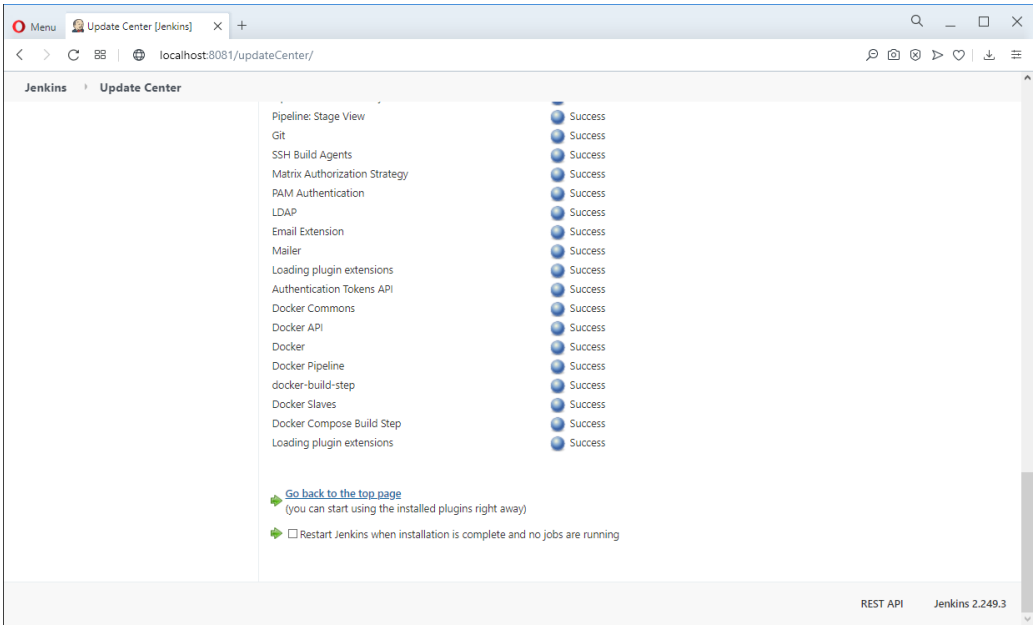


Figure 8: Plugin installation confirmation on 'Update Center'

Step 4

Once the plugins are configured, libraries for Maven, Docker and Terraform libraries need to be made available to Jenkins. This is done on the Global Tool Configuration page. The libraries configured here are public for pipelines created under any projects.

a) Click Manage Jenkins → Global Tool Configurations:

The Global Tools configuration page shown in Figure 9 also lists tools that are required by Jenkins. Make sure these tools were installed and available as part of the initial Jenkins setup.

This page has options to add tools to run the pipeline. Add Maven, Docker and Terraform tools to be configured from this page.

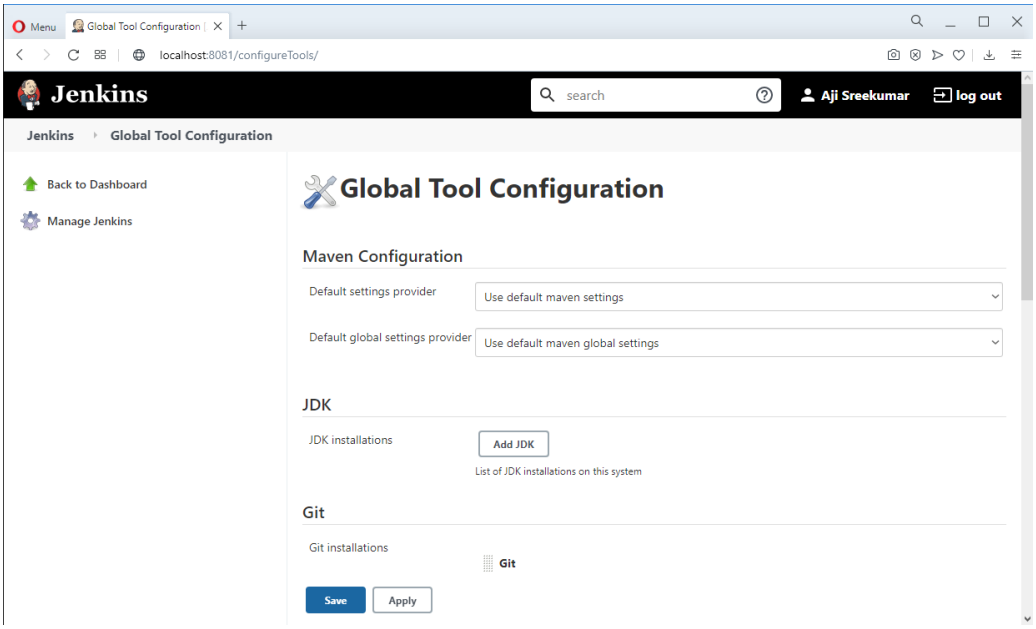


Figure 9: Global Tools Configuration page with default tools

Further, execute the steps in the following order:

b) Add Maven → Select Install automatically checkbox → Give a suitable name

The same name would need to be entered in the Jenkins' script (Jenkinsfile) within the environment tag. The current step configures the Maven service to work with Jenkins. The script file will be created as part of the code base eventually, and Jenkins looks up this path when it wants to trigger the Maven Service. So, the names need to be identical.

A specimen script file is given at the end of the book for reference.

c) Add Docker → Select Install automatically checkbox → Give a suitable name

The same name would need to be given in the Jenkins script within the docker tag.

d) Add Terraform → Select Install automatically checkbox → Give a suitable name

The version of the Terraform plugin should correspond to the version of the OS in which Jenkins is installed. This is selected from the select box in the "Add Terraform" section.

As in the case of Maven and Docker, the same name should go into the Jenkins script that is part of the codebase.

e) Apply and Save.

This is shown in Figure 10. With this, all required plugins are configured.

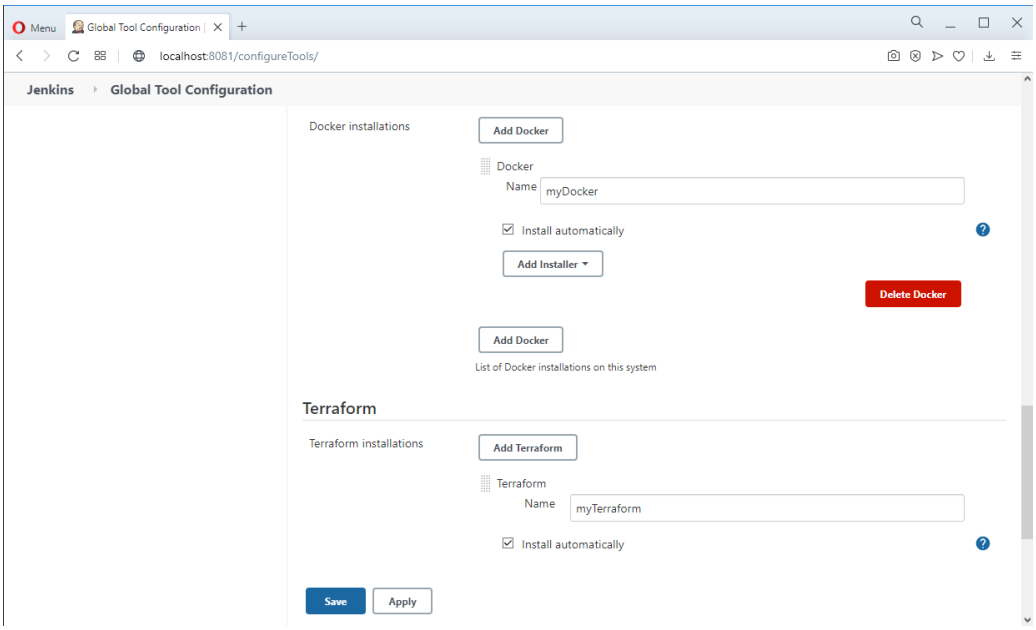


Figure 10: Global tools configuration page (add Docker and Terraform)

Step 4

After installing Maven, Docker and Terraform, credentials need to be set up to authorize Jenkins to access external services such as AWS and Git. This is accomplished from the 'Manage Credentials' page as shown in Figure 11

a) Click Manage Jenkins → Manage Credentials

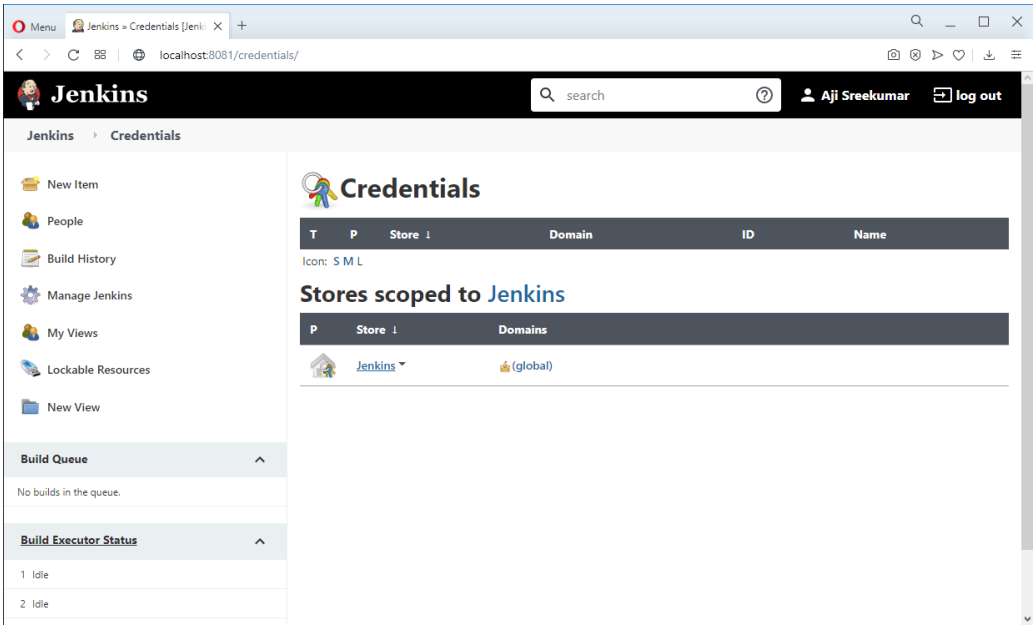


Figure 11: The manage credentials page

b) Click Jenkins → System → Global Credentials/unrestricted link

The unrestricted credentials allow any number of credentials to be configured in Jenkins. Also, credentials added to the Global configuration will be available to all items configured in the Jenkins server. See Figure 12 for a view of the screen.

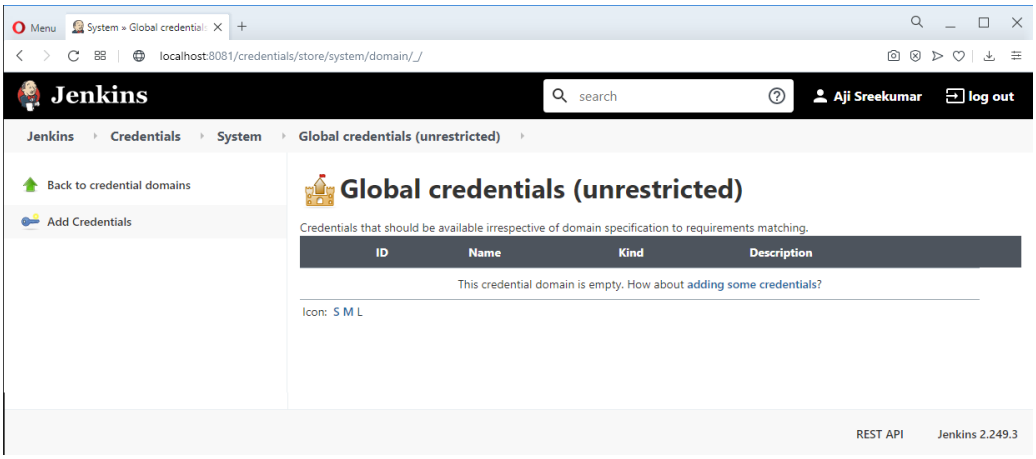


Figure 12: Global credentials configuration

c) Click Add Credentials → Select 'AWS Credentials' from the select → create Jenkins credentials for AWS

Use the AWS account keys to create Jenkins credentials (access key and secret key) as displayed in Figure 13.

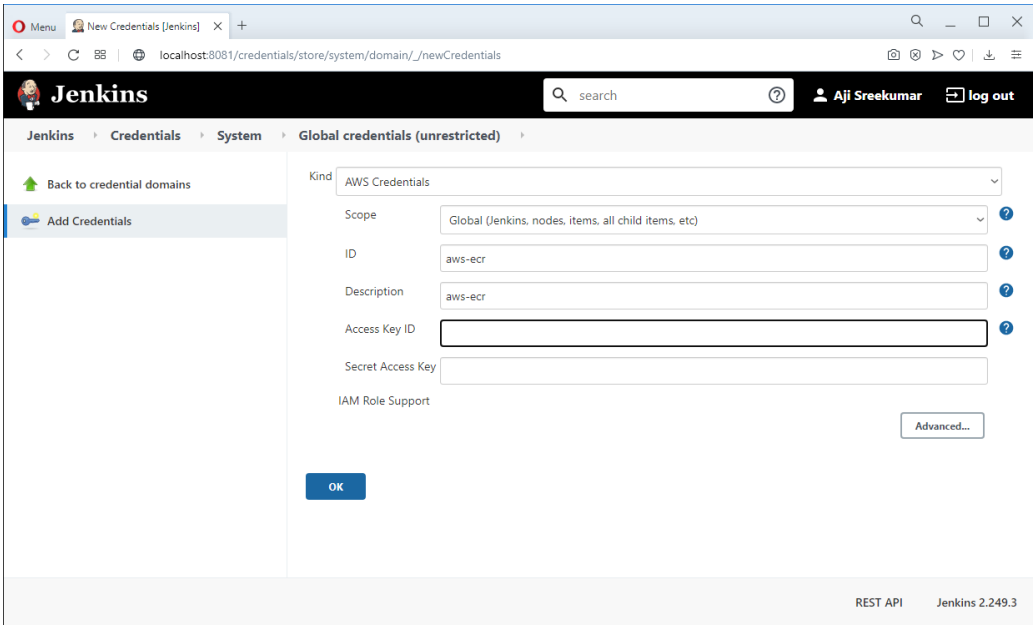


Figure 13: Configuring AWS credentials

Credentials for Docker and Git are configured by selecting 'Kind' as 'Username with password'. Docker credential needs to be configured only if the image needs to be pushed to the Docker Hub. The Git credential may be avoided while using public repositories (as in the case of proof of concepts).

This completes the environment setup required. The oncoming chapters cover the steps creating the pipelines for build, deployment and infrastructure provisioning.

3 JENKINS PIPELINES

A Jenkins Pipeline (is a set of processes) packages an application code for build and deployment, and/or manages the infrastructure. It automates getting the code from Git, compiles, tests, and packages it, and builds the docker image for deployment. Jenkins does this in a structured and repeatable manner.

3.1 Pipeline Creation Steps

Two pipelines are created as part of this setup:

1. One is to automate the real time provision of the pre-defined infrastructure
2. The second one is to automate the build and deployment of code

Step 1

On the Jenkins dashboard:

a) Click New Item → Select pipeline and provide a suitable name → Ok

This takes the user to the Pipeline configuration page as in Figure 14.

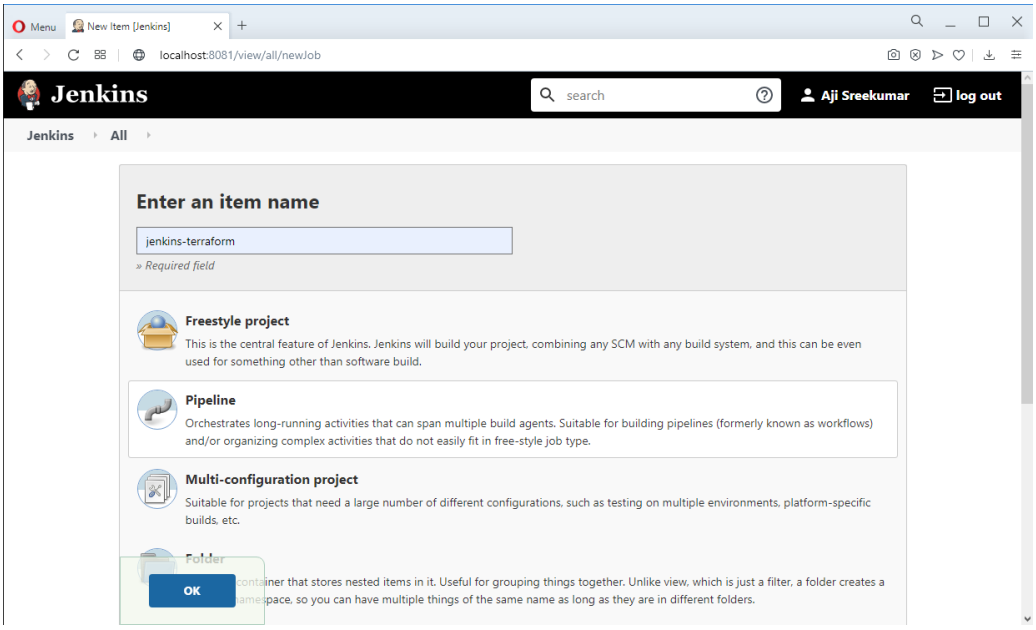


Figure 14: The Create Pipeline screen

Step 2

On the Pipeline configuration page (Figure 15):

a) General tab → Give a Description → Select the checkbox for 'Discard old build' and 'Do not allow concurrent build'

ℹ Note : These are optional. Jenkins will automatically clean up unused builds after a scheduled period.

Next, scroll down to go to the 'Build Triggers' section:

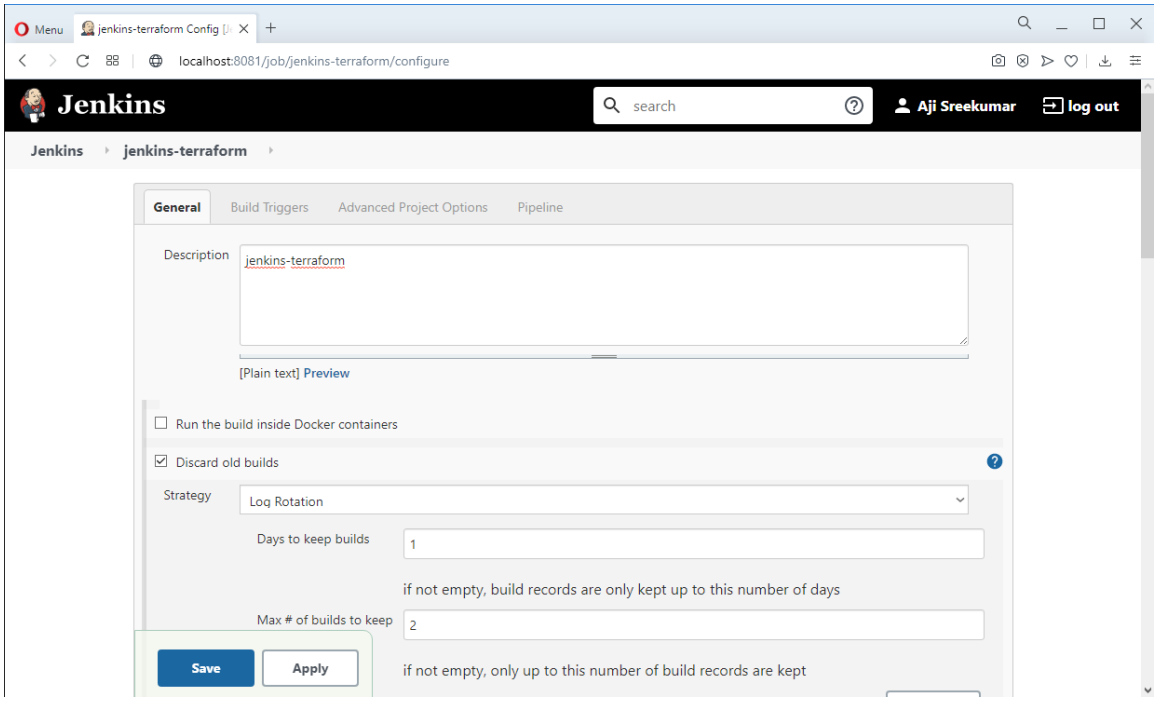


Figure 15: Configuring pipelines: the General tab

b) Select Build Triggers → Poll SCM → Schedule:

Enter Schedule. For instance: If the Schedule is hourly, enter H****. Other schedule options are available like daily, monthly, and by the minute or with a chosen delay. To schedule once every 15 minutes, give H/15****

Figure 16 shows the screen from which the triggers can be scheduled.

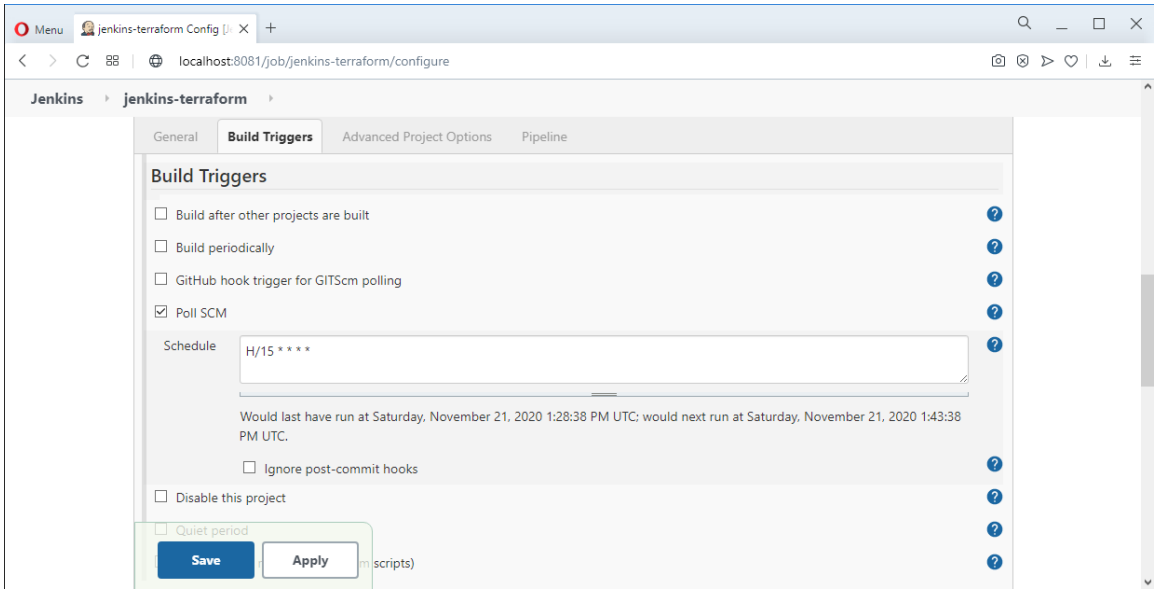


Figure 16: Configuring pipelines: the Build Triggers tab

c) Choose Pipeline →

1. Select 'Definition' as 'Pipeline script from SCM'
2. Select SCM as Git
3. Repositories → Repository URL → <enter Git Repository URL>
4. Repositories *Credentials*: <enter credentials (not mandatory for public repositories)>
5. Branches to build → <enter the branch to build>
6. Script Path → < enter Jenkinsfile, this is available in the text box by default. If not, provide name as Jenkinsfile along with the path>

The Figure 17 shows the configuration screen.

i Note : Figures 15, 16, and 17 shows the configuration available on the same page as individual tabs.

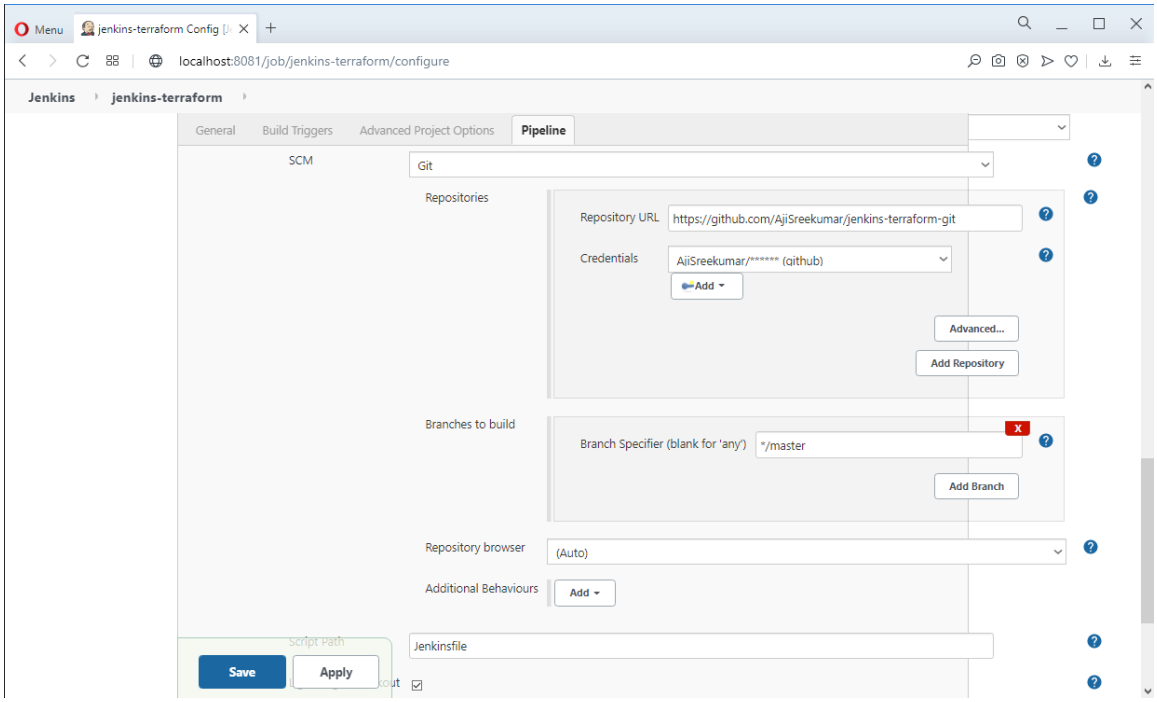


Figure 17: Configuring pipelines: the Pipeline tab

d) Click Save

The pipeline is configured and displayed on the dashboard.

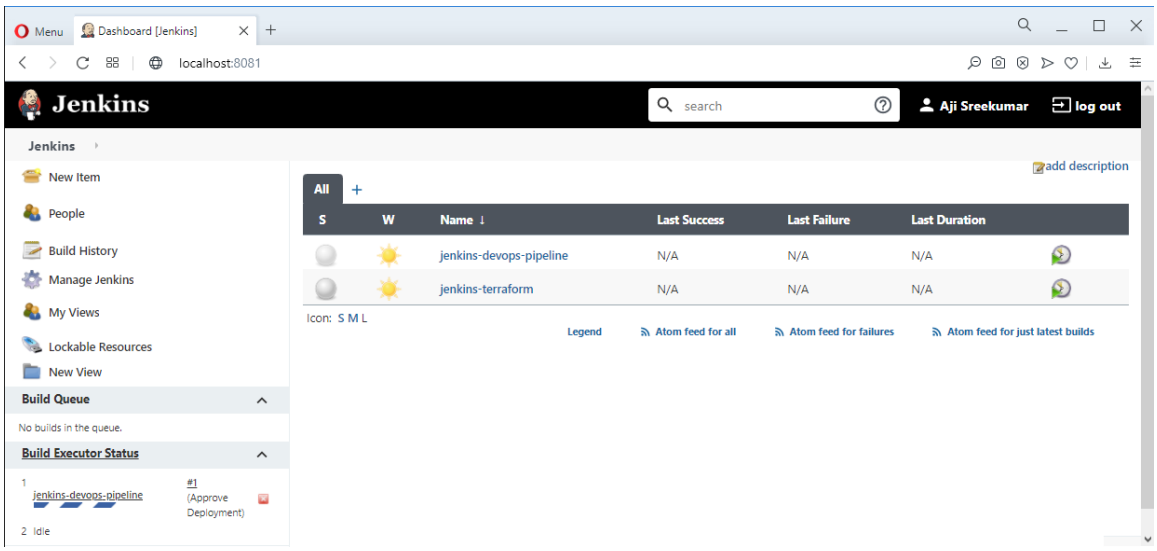


Figure 18: Saved pipeline configurations

At the end of this section, the pipeline for infrastructure provisioning is created and configured. Follow the same steps for the second pipeline (build and deployment).

4 INFRASTRUCTURE PIPELINE - (TERRAFORM-IAC)

Terraform helps to build and maintain versions for infrastructure in an optimal way. It achieves this using configuration scripts that outline the needed infrastructure. The most attractive feature of Terraform is its ability to pick up changes to configuration files and modify the infrastructure without requiring redeployment of the existing applications.

The configuration code can manage many aspects of the infrastructure: the number of computing instances, networking configurations (DNS), storage etc.

4.1 The Script Files

This section covers the steps and code needed to provision environments on AWS using Terraform code. The configuration is split into 'main.tf' and 'backend.tf' as two script files that are reusable with minor modifications. How these files lie within the Git repository and connected to Jenkins is shown in Figure 19.

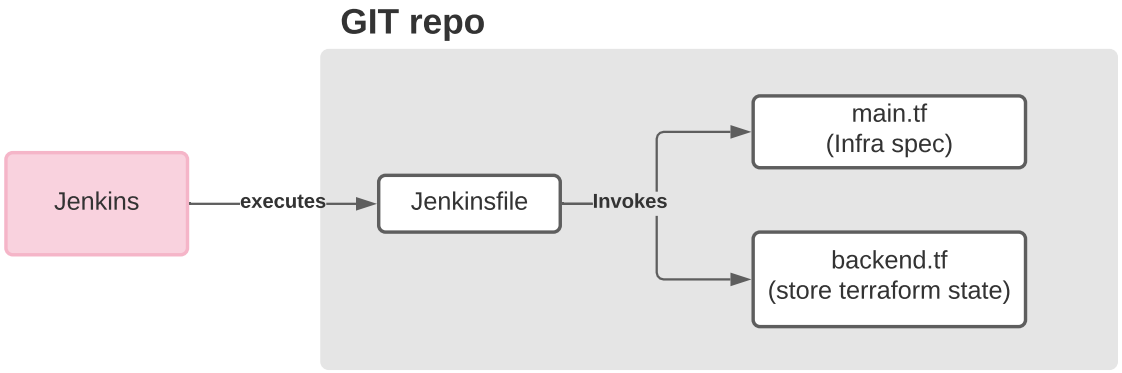


Figure 19: Jenkins script and related components

1. **Jenkinsfile[2]:** holds the Terraform commands to provision, refresh, and destroy infrastructure. There are two ways to create a Jenkinsfile: scripted and declarative.

   'Scripted' is the traditional way of writing scripts using Groovy. 'Declarative' pipeline script is used here as it is more user friendly and easy to understand

2. **main.tf [3]:** covers the main script. This Terraform file holds the specification for the minimum infrastructure required to run a microservice application.

3. **backend.tf:** stores the Terraform state in the AWS bucket so that it is available to all users. This ensures consistency through reuse of the same state across the same environment (say, DEV or PROD).

4.2 Pipeline Creation Steps

The pipeline creation scripts and the steps are detailed in this section.

⚠ **Warning:** Terraform scripts follow style conventions. The reader is encouraged to format the script files given in the code boxes before executing them.

a) Jenkinsfile

```
//DECLARATIVE
pipeline{
    agent any
    environment {
        TERRAFORM_HOME = tool 'myTerraform'
    }
    stages{
        stage('Check Terraform Path and Version') {
            steps {
                sh 'echo $TERRAFORM_HOME'
                sh '${TERRAFORM_HOME}/terraform --version'
            }
        }
        stage('Terraform Init') {
            steps {
                sh "${TERRAFORM_HOME}/terraform init
                -input=false"
            }
        }
        stage('Terraform Plan') {
            steps {
                sh  "${TERRAFORM_HOME}/terraform plan
                -out tfplan"
            }
        }
```

```
        stage('Approve Create') {
            steps {
                script {
                    def userInput = input(id: 'confirm', message:
                    'Apply Terraform?', parameters: [ [$class:
                    'BooleanParameterDefinition', defaultValue: false,
                    description:'Apply terraform', name: 'confirm']])
                }
            }
        }
        stage('Terraform Apply') {
            steps {
                sh "${TERRAFORM_HOME}/terraform apply
                -refresh=false tfplan"
            }
        }
        stage('Approve Destroy') {
            steps {
                script {
                    def userInput = input(id: 'confirm', message:
                    'Destroy Terraform?', parameters:[[$class:
                    'BooleanParameterDefinition', defaultValue: false,
                    description:'Destroy terraform', name: 'confirm']])
                }
            }
        }
        stage('Terraform Destroy') {
            steps {  sh "${TERRAFORM_HOME}/terraform
                    destroy -auto-approve" }
        }
    }

    post{ success{echo('Run when successful')}
        failure{echo('Run when fail')}
    }
}
```

b) main.tf

```
provider "aws" {
  region     = "eu-west-2"
  version    = "~> 3.14.1"
  access_key =
  secret_key =
}

# used default vpc
resource "aws_default_vpc" "default_vpc" {
}

# define data source for subnets
data "aws_subnet_ids" "default_subnets" {
  vpc_id = aws_default_vpc.default_vpc.id
}

# define data source for iam role policy
data "aws_iam_policy_document" "assume_role_policy" {
  statement {
    actions = ["sts:AssumeRole"]
    principals {
      type        = "Service"
      identifiers = ["ecs-tasks.amazonaws.com"]
    }
  }
}

#create ECR repo
resource "aws_ecr_repository" "ecr_repo" {
  name                 = "ecr_repo"
  image_tag_mutability = "MUTABLE"
  image_scanning_configuration {
    scan_on_push = true
  }
}
```

```
# create ECS cluster
resource "aws_ecs_cluster" "ecs_cluster" {
  name = "ecs_cluster"
}

# create Cloud watch log group
resource "aws_cloudwatch_log_group" "ecs-service-log-group" {
  name = "ecs-service-log-group"
}

# create ECS task def and attach  docker image from ecr registry
and cloud watch log to it

resource "aws_ecs_task_definition" "ecs_service_task" {
  family = "ecs_service_task"
  container_definitions    = <<DEFINITION
[
  {
    "name": "ecs_service_task",
    "image": "${aws_ecr_repository.ecr_repo.repository_url}:latest",
    "essential": true,
    "portMappings":[
      {
        "containerPort":8000,
        "hostPort": 8000
      }
    ],
    "environment": [
      {
         "name": "spring.profiles.active",
         "value": "dev"
      }
    ],
```

```
"logConfiguration": {
      "logDriver": "awslogs",
      "options": {
        "awslogs-group": "ecs-service-log-group",
        "awslogs-region": "eu-west-2",
        "awslogs-stream-prefix": "myecs"
      }
},
    "cpu": 400,
    "memory": 2048,
    "memoryReservation": 1024
  }
]

DEFINITION
  requires_compatibilities = ["FARGATE"]
  network_mode             = "awsvpc"
  memory                   = 2048
  cpu                      = 512
  execution_role_arn       = aws_iam_role.ecsTaskExecutionRole.arn
}

# create a IAM role for task execution
  resource "aws_iam_role" "ecsTaskExecutionRole" {
  name               = "ecsTaskExecutionRole"
  assume_role_policy =
  data.aws_iam_policy_document.assume_role_policy.json
}
# attach policy to the role
  resource "aws_iam_role_policy_attachment"
  "ecsTaskExecutionRole_policy" {
  role       = aws_iam_role.ecsTaskExecutionRole.name
  policy_arn =
  "arn:aws:iam::aws:policy/service-role/
  AmazonECSTaskExecutionRolePolicy"
}
```

```
#create security group for application load balancer
resource "aws_security_group" "alb_sg" {
  name   = "alb_sg"
  vpc_id = aws_default_vpc.default_vpc.id
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    egress {
        from_port   = 0
        to_port     = 0
        protocol    = "-1"
        cidr_blocks = ["0.0.0.0/0"]
      }
}

#create application load balancer and attach security group
resource "aws_alb" "alb" {
  name               = "alb"
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default_subnets.ids
  security_groups    = [aws_security_group.alb_sg.id]
}

#create target group
resource "aws_lb_target_group" "alb_target_group" {
  name        = "alb-target-group"
  port        = 80
  protocol    = "HTTP"
  target_type = "ip"
  vpc_id      = aws_default_vpc.default_vpc.id
}
```

```
#create listener and associate with target group
resource "aws_lb_listener" "alb_listener" {
  load_balancer_arn = aws_alb.alb.arn
  port              = "80"
  protocol          = "HTTP"
  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.alb_target_group.arn
  }
}

#create ECS service and attach task definition
#and security group
resource "aws_ecs_service" "ecs_service" {
  name            = "ecs_service"
  cluster         = aws_ecs_cluster.ecs_cluster.id
  desired_count   = 1
  task_definition = aws_ecs_task_definition.ecs_service_task.arn
  launch_type     = "FARGATE"
  load_balancer {
    target_group_arn = aws_lb_target_group.alb_target_group.arn
    container_name = aws_ecs_task_definition.ecs_service_task.family
    container_port   = 8000
  }

  network_configuration {
    subnets         = data.aws_subnet_ids.default_subnets.ids
    assign_public_ip = true
    security_groups = [aws_security_group.ecs_service_sg.id]
    }
    depends_on = [aws_lb_listener.alb_listener]
}
```

```
#create security group for service to communicate with
#alb security group
resource "aws_security_group" "ecs_service_sg" {
  ingress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"
    security_groups = [aws_security_group.alb_sg.id]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

c) Backend file- backend.tf

```
terraform {
  backend "s3" {
    access_key      =
    secret_key      =
    bucket         = "jenkins-terraform-tfstate"
    key            = "terraform.tfstate"
    region         = "eu-west-2"
    encrypt         = true
  }
}
```

Step 1

With the above scripts available, on the Jenkins dashboard:

a) Click the pipeline name → Click Build Now → Approve if asked for

The approval prompt is shown in Figure 20.

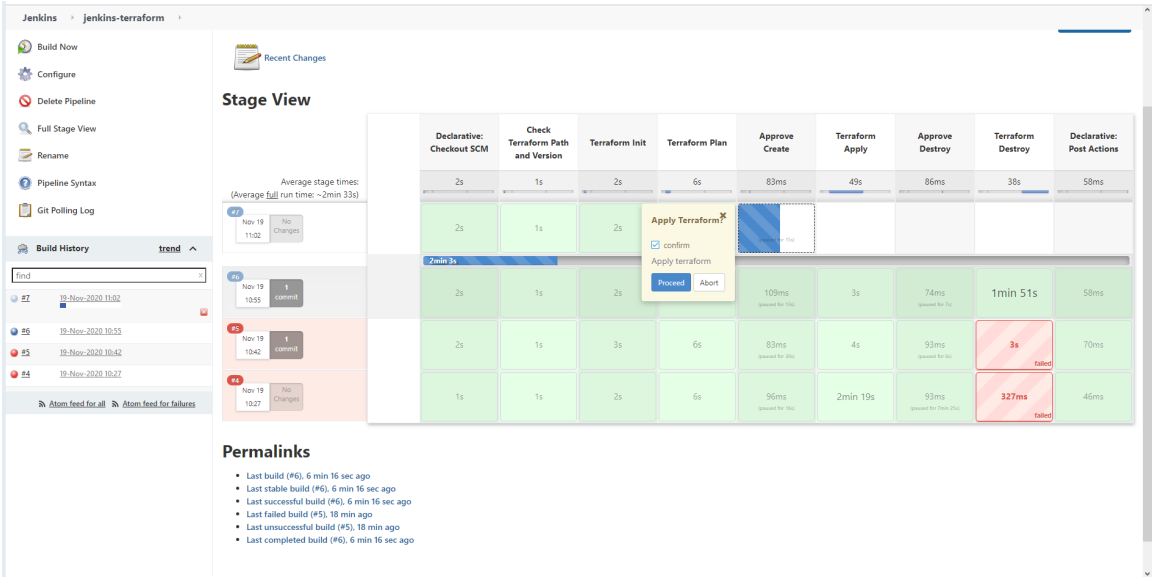


Figure 20: The approval prompt page

With this, the pipeline is completed, and the infrastructure is provisioned in AWS. The completed pipeline is shown in Figure 21.

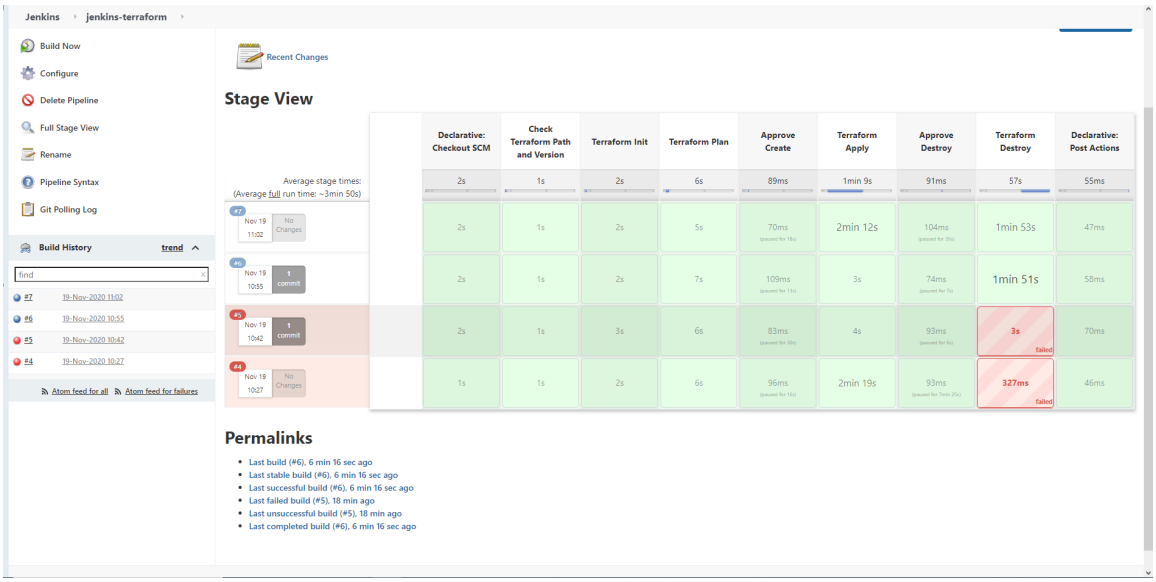


Figure 21: The completed pipeline

The provisioned Infrastructure in AWS is shown in Figure 22.

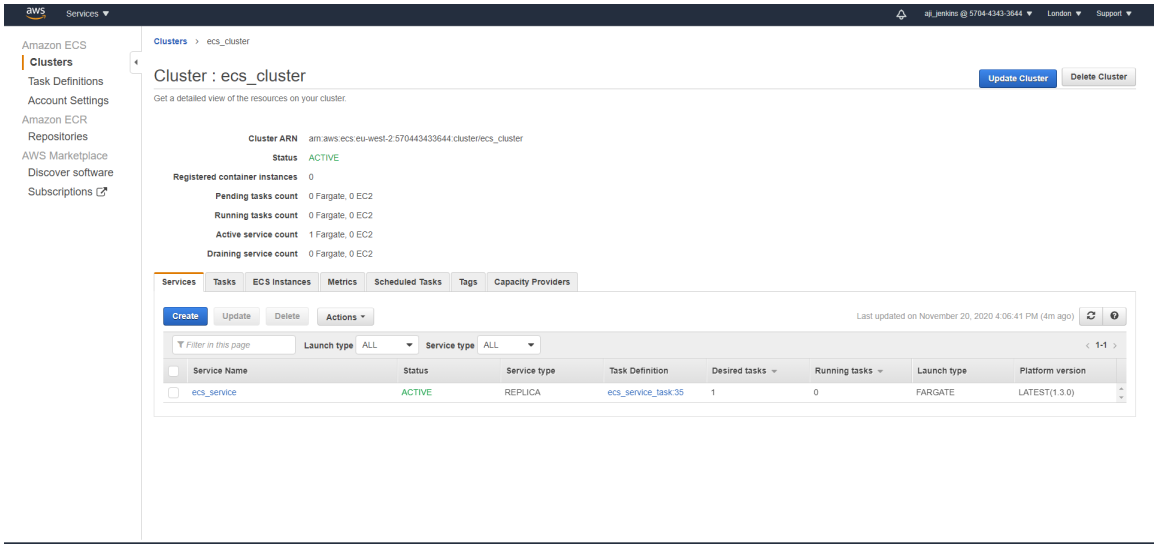


Figure 22: Newly created infrastructure

The Terraform state file created in AWS S3 bucket through the configurations provided in backend.tf file is shown in Figure 23.

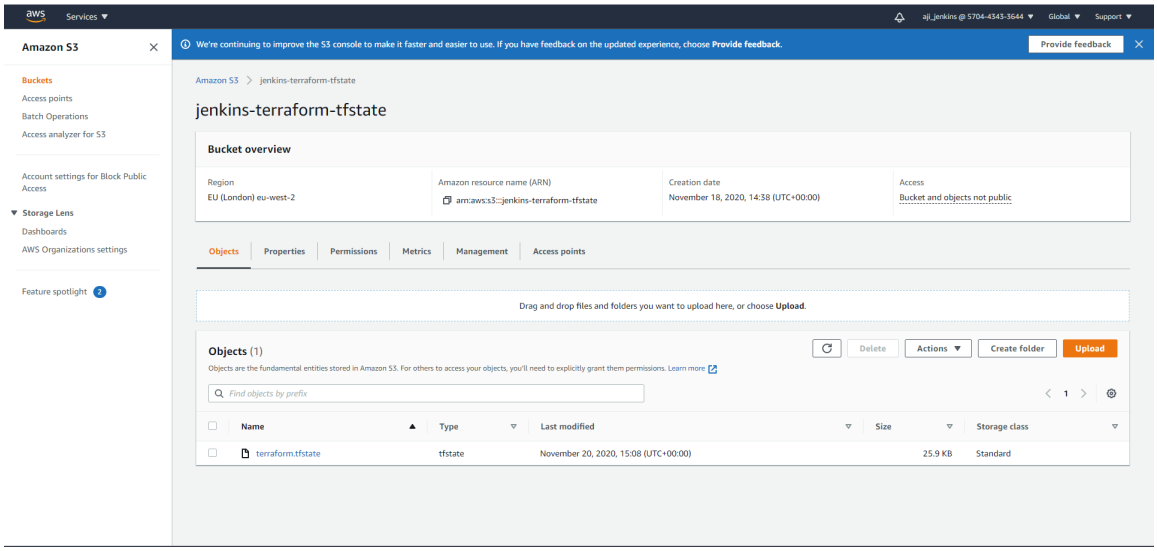


Figure 23: Terraform state in AWS S3 created by backend.tf

5 BUILD AND DEPLOYMENT PIPELINE (DOCKER)

The build and deployment of the application code using Docker are covered in this section. The steps described assumes the application code and Jenkinsfile are present in the Git repository.

5.1 Jenkinsfile and Application Code

The layout of the Jenkinsfile and the application code within the Git repository is illustrated in Figure 24. Jenkins checks out the application code from the Git repository, based on the Jenkinsfile configuration, builds it and deploys it to the AWS environment created previously using the Terraform code. Pipeline executes the script written in the Jenkinsfile kept at the root folder of source code.

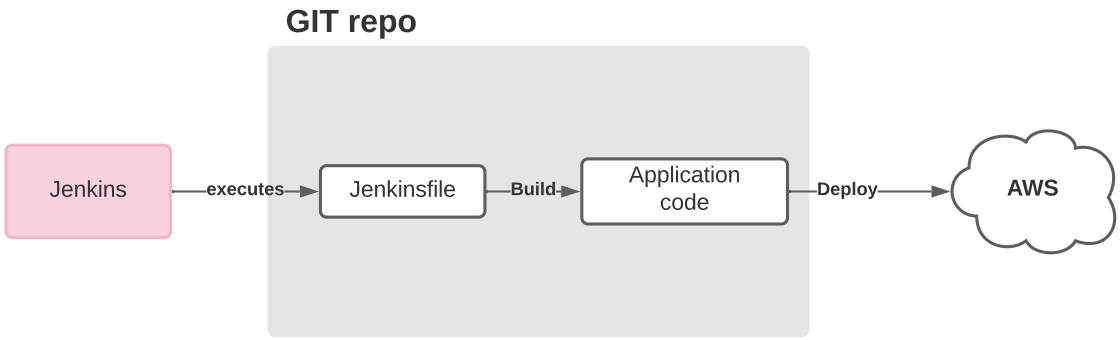


Figure 24: Build and deploy application code

The **Jenkinsfile** holds the maven and docker scripts that compile the application code, package it, build the image and deploys it on the AWS.

a) Jenkinsfile

```
//DECLARATIVE
pipeline{
    agent any
    environment {
        dockerHome = tool <provide the name configured in
        Global tool configuration>
        mavenHome = tool  <provide the name configured in
        Global tool configuration>
        PATH = "$dockerHome/bin:$mavenHome/bin:$PATH"
    }
    stages{
        stage('Build Details'){
            steps{
                sh 'mvn --version'
                sh 'docker --version'
                echo "Build"
                echo "PATH - $PATH"
                echo "BUILD_NUMBER - $env.BUILD_NUMBER"
                echo "BUILD_ID - $env.BUILD_ID"
                echo "JOB_NAME - $env.JOB_NAME"
                echo "BUILD_TAG - $env.BUILD_TAG"
                echo "BUILD_URL - $env.BUILD_URL"
            }
        }
        stage('Compile'){
            steps{
                sh "mvn clean compile"
            }
        }
        stage('Run Unit Test'){
            steps{
                sh "mvn test"
            }
        }
```

```
        stage('Generate Sonar Report'){
            steps{
                sh "mvn sonar:sonar -Dsonar.host.url=
                <provide the url of the Sonar server>"
            }
        }
        stage('Approve Sonar'){
            steps {
             script {
             def userInput = input(id: 'confirm', message:
             'Approve Sonar ?', parameters:[[$class:
             'BooleanParameterDefinition', defaultValue: false,
             description: 'Approve Sonar', name: 'confirm'] ])
             }
            }
        }
        stage('Integration Test'){
            steps{
                sh "mvn failsafe:integration-test failsafe:verify"
            }
        }
        stage('Package'){
            steps{
                echo "Package"
                sh "mvn package -DskipTests"
            }
        }
        stage('Build Docker Image'){
            steps{
                echo "Build Docker Image"
                script{
                    dockerImage = docker.build("<provide the AWS ECR
                    repo name>:${env.BUILD_TAG}")
                }
            }
        }
```

```
        stage('Approve Deployment'){
           steps {
                script {
                    def userInput = input(id: 'confirm', message:
                    'Proceed deployment ?', parameters: [ [$class:
                    'BooleanParameterDefinition', defaultValue: false,
                    description: 'Proceed deployment', name: 'confirm'] ])
                }
            }
        }

        stage('Push to ECR'){
            steps{
                echo "Push Docker Image to AWS ECR"
                script{
                        docker.withRegistry(<AWS docker registry url>,
                        'ecr:<region >:<AWS credentials
                        configured in Jenkins>'){
                        dockerImage.push();
                        dockerImage.push('latest')
                    }
                }
            }
        }
    }

    post{
        success{ echo('Run when successful') }
        failure{ echo('Run when fail') }
    }
}
```

Step 1

On the Jenkins Dashboard:

a) Click the pipeline name created for build and deploy → Build Now → Approve when asked for

The approval prompt for deployment is shown in Figure 25.

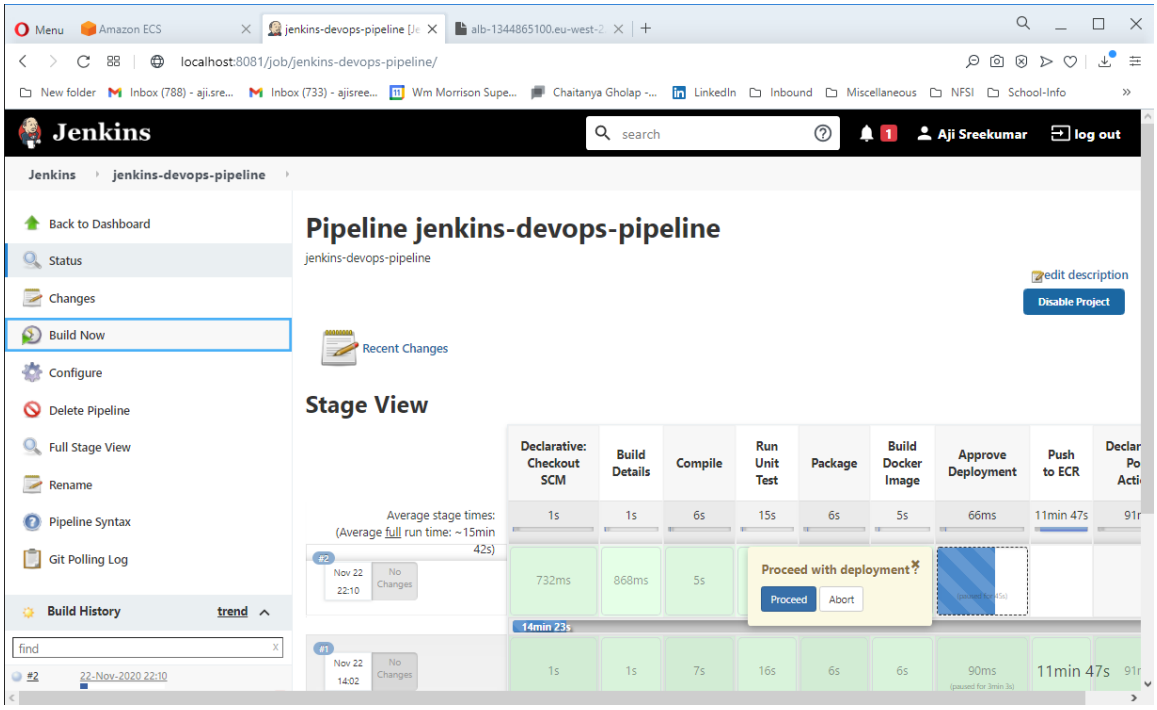


Figure 25: Approve build and deploy pipeline

The build and deployment pipeline created in this step is shown in Figure 26.

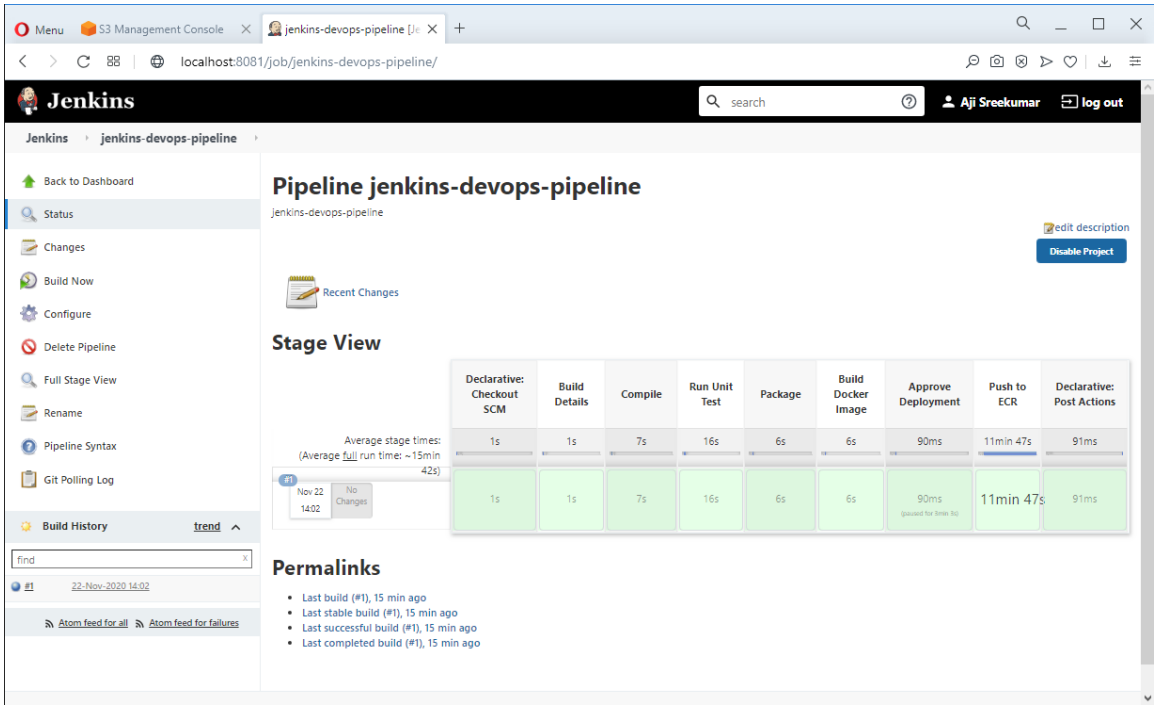


Figure 26: The completed pipeline

The image is available in the AWS ECR repository as in Figure 27

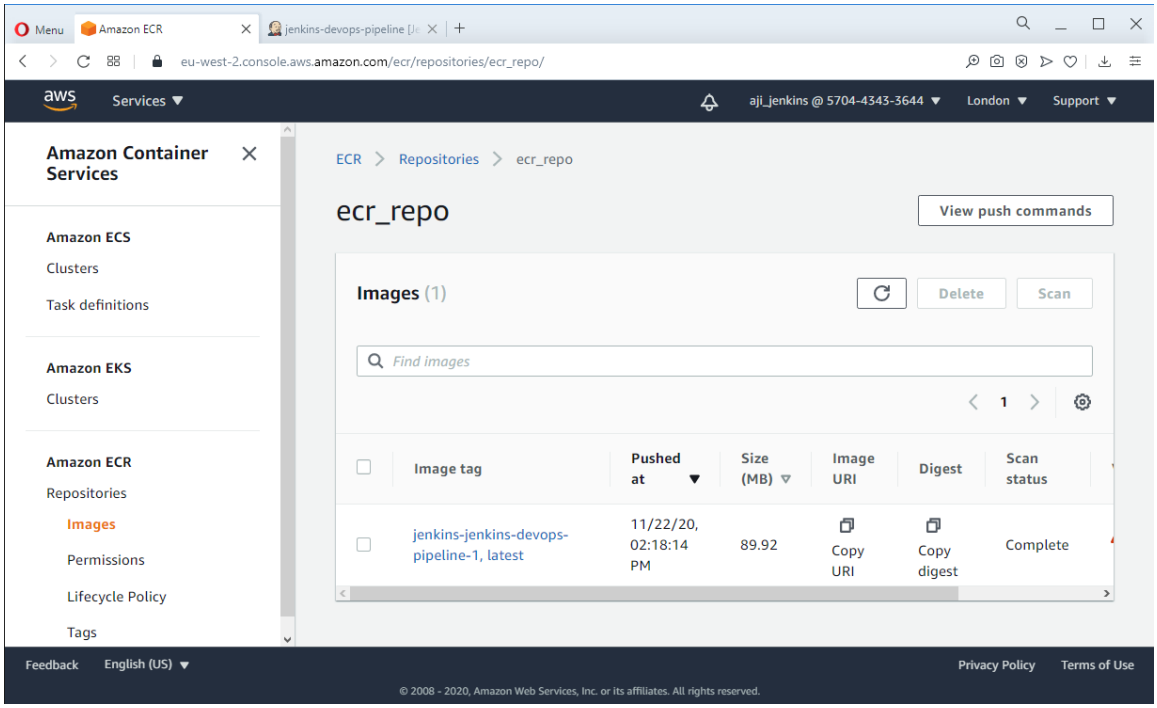


Figure 27: The image in AWS ECR

6 SUMMARY

A CI/CD engine with Jenkins, Terraform, and Docker provides the seamless capability to create infrastructure, build and deploy code. This simple setup can improve the development maturity of any team, large or small, to a high degree.

6.1 The Role of each Tool in the CI/CD Process

The role of each of the components covered in this book within the CI/CD process is illustrated in Figure 28.

The book helps to create a CI pipeline to integrate and deploy application code in AWS, and a Terraform pipeline for provisioning the required infrastructure, in five sections. The configuration files needed are also detailed out in their respective sections.

6.2 Opportunities and Benefits:

The CI/CD methodologies bring in many advantages through increased process maturity. The following are a few direct benefits observed, irrespective of the size of the project.

1. **Reduced error rate:** Considerable decrease in errors due to reduced manual intervention
2. **Cost Savings:** Cost savings due to reduced rework effort from the developers
3. **Quick turnaround :** Gives opportunity for faster, and more frequent production deployments
4. **Increased Productivity:** Developers can now focus on more productive activities like creating functionalities rather than mundane, repetitive tasks

5. **Testing quality:** Is increased since automation scripts provide considerable coverage through regression testing in every release cycle. With very little manual intervention, this improvement is available at the least cost.

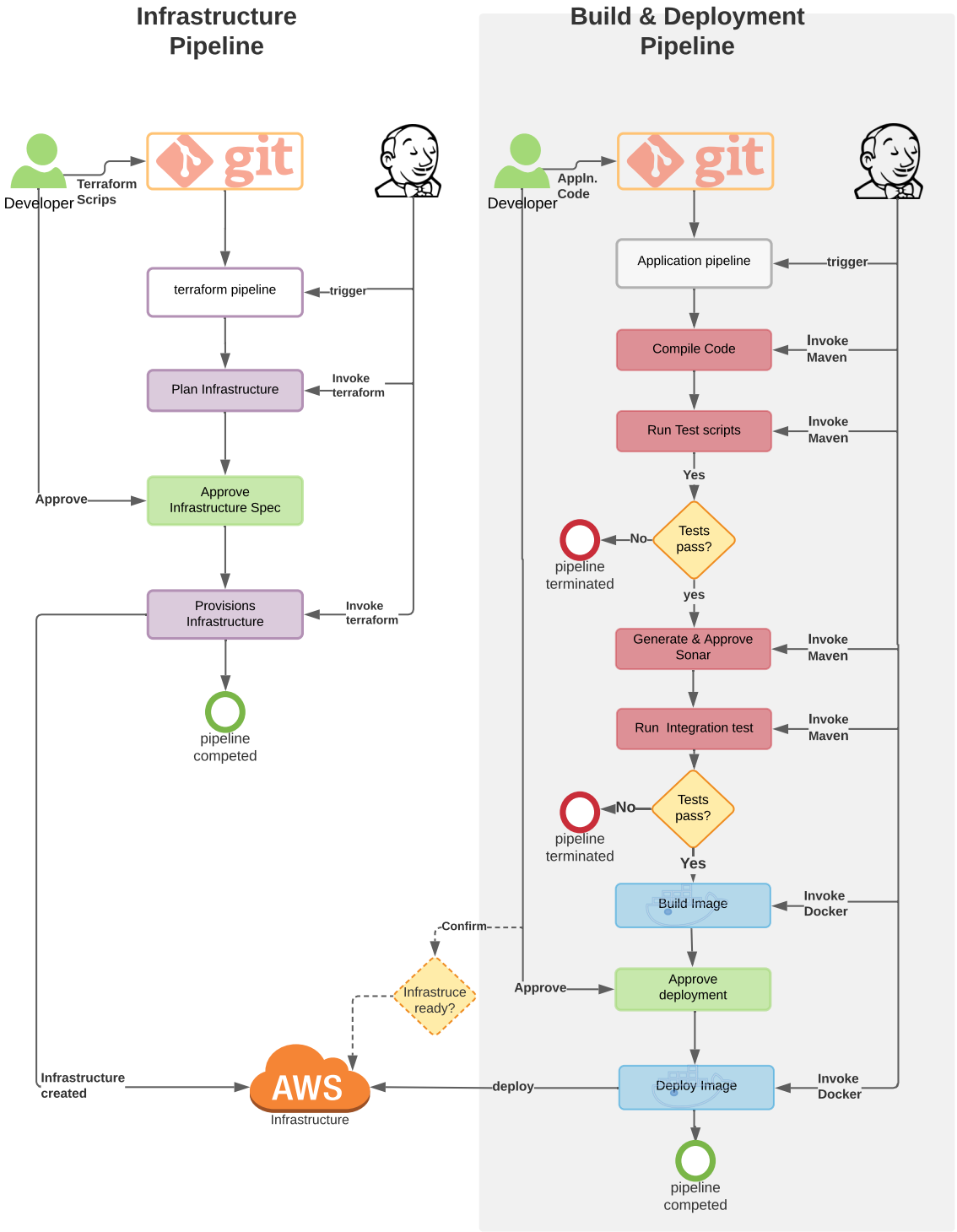


Figure 28: The role of Jenkins, Terraform, and Docker in the overall CI/CD process

6.3 Notable Technical Issues and Learning:

A few critical issues that surfaced while integrating the tools are noted here for the benefit of new users:

1. There might be cases where Jenkins fails to recognize the Terraform commands in the declarative pipeline script. This can be resolved by adding the Terraform path to the commands.

2. The Docker 'push' to cloud repository takes away the entire available network bandwidth resulting in an outage. This can be resolved by setting the parameter "max-concurrent-uploads" to 1 in the Docker daemon configuration file [4].

3. The AWS credentials provided in the main Terraform file (main.tf) might not be accessible to the backend Terraform file (backend.tf). To resolve this, credentials can be locally set in the backend.tf to connect to AWS.

6.4 Conclusion

This book has covered the most essential details needed to get a CI/CD engine up and running quickly for a software development project. While it is assumed that the reader is familiar with Git, Maven and other related technologies required, vital scripts are provided that can be reused with very little or no changes at all. The step by step instructions, with screenshots and minimum verbiage helps him to move ahead at a fast pace.

While institutionalizing this CI/CD engine, the development team is also encouraged to establish a formal Git version control process for the maximum benefit and a seamless DevOps experience. I hope this book will motivate the reader to adopt the Jenkins, Docker, Terraform based CI/CD process for its simplicity and maturity, and straightforward setup.

References

[1] Sharma,DevOps 3rd IBM Limited Edition. IBM Limited, Mar. 2017.

[2] R. Karanam, Master DevOps with Docker, Kubernetes and Azure DevOps. Udemy, 2020.

[3] Hashicorp, "Hashicorp terraform registry." https://registry.terraform.io. Accessed: 2020-11-19.

[4] Stackoverflow, "Stackoverflow questions." https://stackoverflow.com. Accessed: 2020-11-20.

Mastek UK Ltd
Pennant House | 2 Napier Court
Reading | RG1 8BW
United Kingdom