



Search Medium



Published in Better Programming

You have **2** free member-only stories left this month.

[Sign up for Medium and get an extra one](#)



Alfonso Valdes Carrales

[Follow](#)

Jun 22, 2022 · 23 min read

[Listen](#)[Save](#)

# How to Create a CI/CD Pipeline With Jenkins, Containers, and Amazon ECS

Learn the step-by-step process of deploying a sample Nodejs containerized application

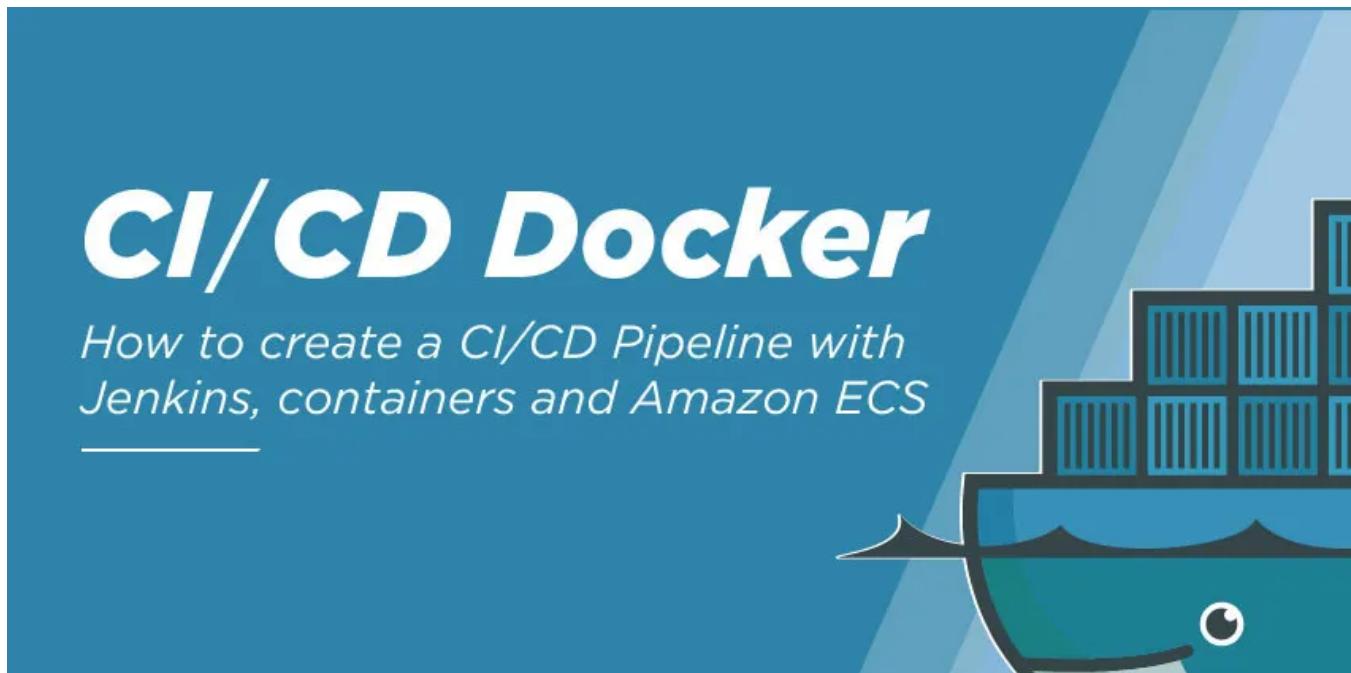


image by author

If you're still building and delivering your software applications the traditional way then you are missing out on a major innovation in the software development process or software development lifecycle.

To show you what I'm talking about, in this article I will share “How to create a CI/CD Pipeline with Jenkins, Containers, and Amazon ECS” that deploys your application and overcomes the limitations of the traditional software delivery model.

This innovation greatly affects deadlines, time to market, quality of the product, etc. I will take you through the whole step-by-step process of setting up a CI/CD Docker pipeline for a sample Nodejs application.

A CI/CD Pipeline (or Continuous Integration Continuous Delivery) pipeline is a set of instructions to automate the process of software tests, builds, and deployments. Here are a few benefits of implementing CI/CD in your organization.



### 1. Smaller Code Change

The ability of CI/CD pipelines to allow the integration of a small piece of code at a time helps developers to recognize any potential problem before too much work is completed.

### 2. Faster Delivery

Multiple daily releases or continual releases can be made a reality using CI/CD pipelines.

### 3. Observability

Having automation in place that generates extensive logs in each stage of the development process helps to understand if something goes wrong.

### 4. Easier Rollbacks

There are chances that the code that has been deployed may have issues. In such cases, it is very crucial to get back to the previous working release as soon as possible. One of the biggest advantages of using the CI/CD pipelines is that you can quickly and easily roll back to the previous working release.

### 5. Reduce Costs

Having automation in place for repetitive tasks frees up the developer and operation people's time that could be spent on product development.

These are just a few benefits of having CI/CD pipelines for builds and deployments. In this video, you can continue learning the CI/CD benefits and why you should use a CI/CD in the first place.

Now, before we proceed with the steps to set up a CI/CD Pipeline with Jenkins, Containers, and Amazon ECS, let's see in short what tools and technologies we will be using.

## CI/CD Docker Tool Stack

### 1. GitHub

It is a web-based application or a cloud-based service where people or developers collaborate, store and manage their application code using Git. V will create and store our sample Nodejs application code here.



### 2. AWS EC2 Instance

AWS EC2 is an Elastic Computer Service provided by Amazon Web Services used to create virtual machines or virtual instances on AWS Cloud. We will create an EC2 instance and install Jenkins and other dependencies in it.

### 3. Java

This will be required to run Jenkins Server.

### 4. AWS CLI

aws-cli i.e AWS Command Line Interface is a command-line tool used to manage AWS Services using commands. We will be using it to manage AWS ECS Task and ECS Service.

### 5. Nodejs and Npm

Nodejs is a backend JavaScript runtime environment and Npm is a package manager for Node. We will be creating a CI/CD Docker pipeline for the Nodejs application.

### 6. Docker

Docker is an open source containerization platform used for developing, shipping, and running applications. We will use it to build Docker images of our sample Nodejs application and push/pull them to/from AWS ECR.

## 7. Jenkins

Jenkins is an open source, freely available automation server used to build, test, and deploy software applications. We will be creating our CI/CD Docker pipeline to build, test and deploy our Nodejs application on AWS ECS using Jenkins

## 8. AWS ECR

AWS Elastic Container Registry is a Docker image repository fully managed by AWS to easily store, share, and deploy container images. We will be using AWS ECR to store Docker images of our sample Nodejs application.

## 9. AWS ECS



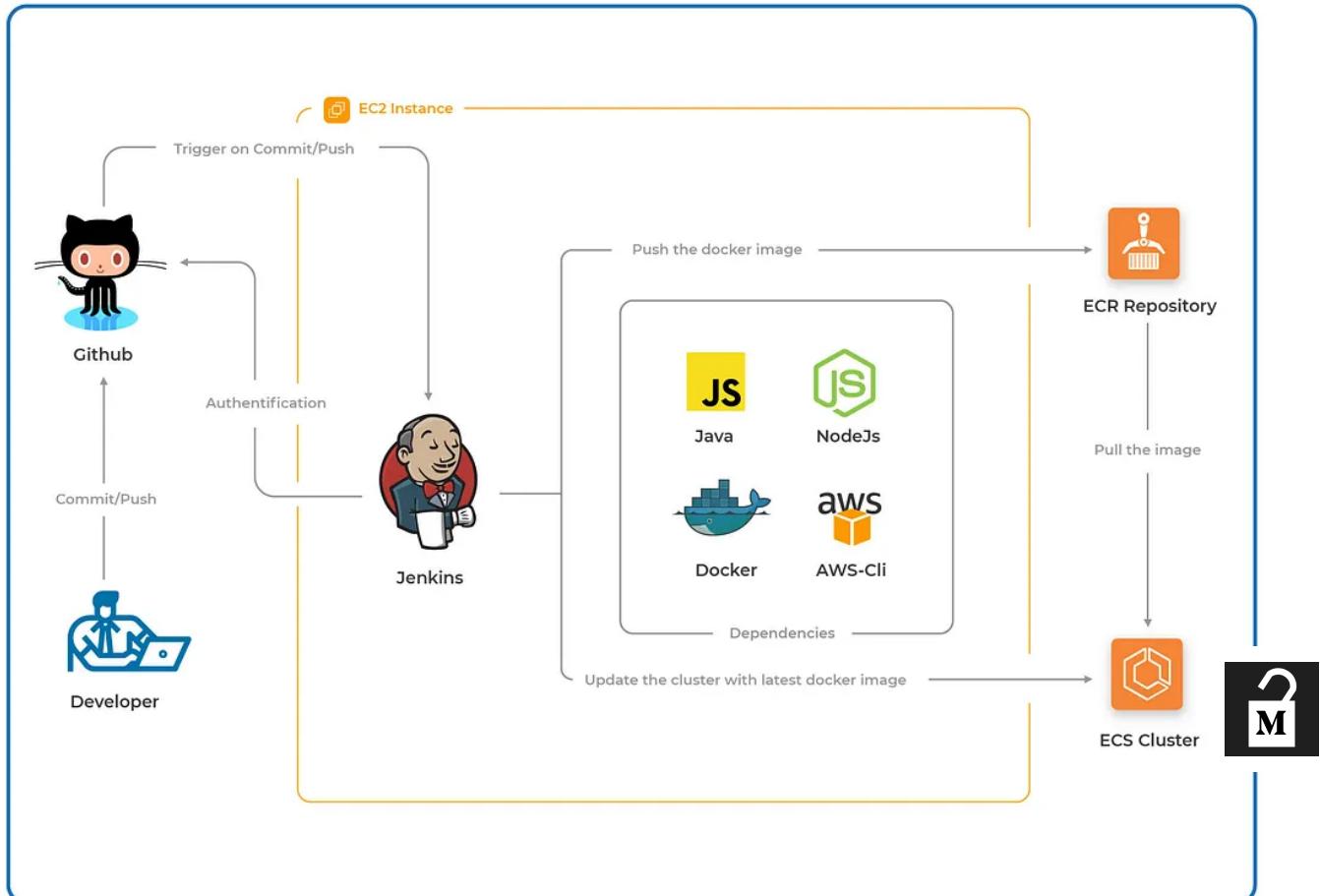
AWS Elastic Container Service is a container orchestration service fully managed by AWS to easily deploy, manage, and scale containerized applications. We will be using it to host our sample Nodejs application.

Also read: [CI/CD Pipeline and workflow on AWS, Kubernetes, and Docker](#)

## Architecture

This is what our architecture will look like after setting up the CI/CD Pipeline with Docker.

After the CI/CD Docker Pipeline is successfully set up, we will push commits to our GitHub repository and in turn, GitHub Webhook will trigger the CI/CD Pipeline on Jenkins Server. Jenkins Server will then pull the latest code, carry out unit tests, build a docker image and push it to AWS ECR. After the image is pushed to AWS ECR, the same image will be deployed in AWS ECS by Jenkins.



## CI/CD Workflow and Phases

### Workflow

CI and CD Workflow allows us to focus on development while it carries out the tests, build, and deployments in an automated way.

#### 1. Continuous Integration

This allows the developers to push the code to the version control system or source code management system, build and test the latest code pushed by the developer, and generate and store artifacts.

#### 2. Continuous Delivery

This is the process that lets us deploy the tested code to the production whenever required.

#### 3. Continuous Deployment

This goes one step further and releases every single change without any manual intervention to the customer system every time the production pipeline passes all the tests.

## Phases

The primary goal of the automated CI/CD pipeline is to build the latest code and deploy it. There can be various stages as per the need. The most common ones are mentioned below:

### 1. Trigger

The CI/CD pipeline can do its job on the specified schedule when executed manually or triggered automatically on a particular action in the code repository.



### 2. Code Pull

In this phase, the pipeline pulls the latest code whenever the pipeline is triggered.

### 3. Unit Tests

In this phase, the pipeline performs tests that are there in the codebase, this is also referred to as unit tests.

### 4. Build or Package

Once all the tests pass, the pipeline moves forward and builds artifacts or docker images in case of dockerized applications.

### 5. Push or Store

In this phase, the code that has been built is pushed to the artifactory or Docker repository in the case of dockerized applications.

### 6. Acceptance Tests

This phase or stage of the pipeline validates if the software behaves as intended. It is a way to ensure that the software or application does what it is meant to do.

### 7. Deploy

This is the final stage in any CI/CD pipeline. In this stage, the application is ready for delivery or deployment.

## Deployment strategy

A deployment strategy is a way in which containers of the micro-services are taken down and added. There are various options available. However, we will only discuss the ones that are available and supported by ECS

### Rolling updates

In rolling updates, the scheduler in the ECS Service replaces the currently running tasks with new ones. The tasks in the ECS cluster are nothing but running containers created out of the task definition. Deployment configuration controls the number of tasks that Amazon ECS adds or removes from the service.

The lower and the upper limit on the number of tasks that should be running is controlled by `minimumHealthyPercent` and `maximumPercent` respectively.



1. `minimumHealthyPercent` example: If the value of `minimumHealthyPercent` is 50 and the desired task count is 4, then the scheduler can stop two existing tasks before starting two new tasks
2. `maximumPercent` example: If the value of `maximumPercent` is 4 and the desired task is 4, then the scheduler can start four new tasks before stopping four existing tasks.

If you want to learn more about this, visit the official documentation [here](#).

### Blue/Green Deployment

Blue/Green deployment strategy enables the developer to verify a new deployment before sending traffic to it by installing an updated version of the application as a new replacement task set.

There are primarily three ways in which traffic can shift during blue/green deployment.

1. Canary – Traffic is shifted in two increments, percentage of traffic shifted to your updated task set in the first increment and the interval, in minutes, before the remaining traffic is shifted in the second increment.

2. Linear – Traffic is shifted in equal increments, the percentage of traffic shifted in each increment, and the number of minutes between each increment.
3. All-at-once – All traffic is shifted from the original task set to the updated task set all at once.

To learn more about this, visit the official documentation [here](#).

Out of these two strategies, we will be using the rolling-updates deployment strategy in our demo application.

## Dockerize Node.js app

Now, let's get started and make our hands dirty.



The Dockerfile for the sample Nodejs application is as follows. There is no need to copy-paste this file, it is already available in the sample git repository that you cloned previously.

Let's just try to understand the instructions of our Dockerfile.

1. FROM node:12.18.4-alpine

This will be our base image for the container.

2. WORKDIR /app

This will be set as a working directory in the container.

3. ENV PATH /app/node\_modules/.bin:\$PATH

The PATH variable is assigned a path to /app/node\_modules/.bin .

4. COPY package.json ./

Package.json will be copied into the working directory of the container.

5. RUN npm install

Install dependencies.

6. COPY . ./

Copy files and folders with dependencies from the host machine to the container.

## 7. EXPOSE 3000

Allow to port 300 of the container.

## 8. CMD [“node”, “./src/server.js”]

Start the application

This is the Docker file that we will use to create a docker image.

## Setup GitHub Repositories

### Create a new repository

1. Go to <https://github.com/>, create an account if you don't have it already else log in to your account and create a new repository. You can name it as per your choice; however, I would recommend using the same name to avoid any confusion.



A screenshot of the GitHub 'Create a new repository' interface. The page has a light gray header with the GitHub logo and navigation links like 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the header, there's a search bar and a 'Create a New Repository' button. The main form is titled 'Create a new repository'. It asks for the 'Owner' (set to 'shivalkarrahul') and 'Repository name' ('demo-nodejs-app'). A note says 'Great repository names are short and descriptive'. There's an optional 'Description' field and a choice between 'Public' (selected) and 'Private'. Under 'Initialize this repository with:', there are three options: 'Add a README file', 'Add .gitignore', and 'Choose a license'. At the bottom is a large green 'Create repository' button.

2. You will get the screen as follows, copy the repository URL and keep it handy. Call this URL as a GitHub Repository URL and note it down in the text file on your system.

The screenshot shows a GitHub repository page for 'shivalkarrahal/demo-nodejs-app'. A prominent blue box at the top contains the text 'Quick setup — if you've done this kind of thing before' and provides options to 'Set up in Desktop' or 'HTTPS' or 'SSH'. It also includes the URL 'https://github.com/shivalkarrahal/demo-nodejs-app.git'. Below this, a note says 'Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.'

Note: Create a new text file on your system and note down all the details that will be required later.

## Create a GitHub Token

This will be required for authentication purposes. It will be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over basic authentication.



1. Click on the user icon in the top-right, go to “Settings,” then click on the “Developers settings” option in the left panel.

The screenshot shows the GitHub 'Settings' page under 'Developer settings'. On the left, there's a sidebar with options like Account settings, Profile, Account, Appearance, etc. The main area is titled 'Public profile' and contains fields for Name, Profile picture, Bio, URL, Twitter username, Company, and Location. To the right, a sidebar shows the user is signed in as 'shivalkarrahal' and lists options such as Your profile, Your repositories, Your codespaces, Your projects, Your stars, Your gists, Upgrade, Feature preview, Help, Settings, and Sign out.

2. Click on the “Personal access tokens” options and “Generate new token” to create a new token.

The screenshot shows the GitHub developer settings page. The 'Personal access tokens' tab is selected. A note at the top says: 'Need an API token for scripts or testing? [Generate a personal access token](#) for quick access to the [GitHub API](#).'. Below this, it states: 'Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#)'. There is a 'Generate new token' button.

3. Tick the “repo” checkbox, the token will then have “full control of private repositories”

The screenshot shows the GitHub developer settings page with the 'New personal access token' form. The 'Note' field contains 'demo-token'. The 'Expiration' dropdown is set to '30 days' with a note that it will expire on 'Tue, Nov 9 2021'. The 'Select scopes' section is expanded, showing various options under the 'repo' scope. A large 'M' icon with a lock symbol is visible on the right.

Scope	Description
<input checked="" type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input type="checkbox"/> admin:org	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input type="checkbox"/> admin:public_key	Full control of user public keys
<input type="checkbox"/> write:public_key	Write user public keys
<input type="checkbox"/> read:public_key	Read user public keys

4. You should see your token created now.

The screenshot shows the GitHub developer settings page with the 'Personal access tokens' list. It shows one token named 'demo-token' which has the scope 'repo' and was generated on 'Tue, Nov 9 2021'. It has never been used and has a 'Delete' button.

## Clone the Sample Repository

1. Check your present working directory.

```
pwd
```

Note: You are in the home directory, i.e., /home/ubuntu.

1. Clone my sample repository containing all the required code.

```
git clone https://github.com/shivalkarrahul/nodejs.git
```

2. Create a new repository. This repository will be used for CI/CD Pipeline setup.

```
git clone https://github.com/shivalkarrahul/demo-nodejs-app.git
```

3. Copy all the code from my nodejs repository to the newly created demo-nodejs-app repository.

```
cp -r nodejs/* demo-nodejs-app/
```

4. Change your working directory.

```
cd demo-nodejs-app/
```



Note: For the rest of the article, do not change your directory. Stay in the same directory. Here it is, /home/ubuntu/demo-nodejs-app/, and execute all the commands from there.

1. ls -l

2. git status

```
[ubuntu@ip-172-31-24-191:~$ pwd
/home/ubuntu
[ubuntu@ip-172-31-24-191:~$ git clone https://github.com/shivalkarrahul/nodejs.git
Cloning into 'nodejs'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 15 (delta 2), reused 14 (delta 1), pack-reused 0
Unpacking objects: 100% (15/15), done.
[ubuntu@ip-172-31-24-191:~$ git clone https://github.com/shivalkarrahul/demo-nodejs-app.git
Cloning into 'demo-nodejs-app'...
warning: You appear to have cloned an empty repository.
[ubuntu@ip-172-31-24-191:~$ cp -r nodejs/* demo-nodejs-app/
[ubuntu@ip-172-31-24-191:~$ cd demo-nodejs-app/
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ ls -l
total 28
-rw-rw-r-- 1 ubuntu ubuntu 361 Oct 10 07:35 Dockerfile
-rw-rw-r-- 1 ubuntu ubuntu 1370 Oct 10 07:35 Jenkinsfile
-rw-rw-r-- 1 ubuntu ubuntu 535 Oct 10 07:35 README.md
-rw-rw-r-- 1 ubuntu ubuntu 327 Oct 10 07:35 package.json
-rwxrwxr-x 1 ubuntu ubuntu 1286 Oct 10 07:35 script.sh
drwxrwxr-x 3 ubuntu ubuntu 4096 Oct 10 07:35 src
-rw-rw-r-- 1 ubuntu ubuntu 510 Oct 10 07:35 task-definition.json
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    Dockerfile
    Jenkinsfile
    README.md
    package.json
    script.sh
    src/
    task-definition.json

nothing added to commit but untracked files present (use "git add" to track)
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ ]
```

## Push Your First Commit to the Repository

1. Check your present working directory; it should be the same. Here it is,

```
/home/ubuntu/demo-nodejs-app/
```

```
pwd
```

2. Set a username for your git commit message.

```
git config user.name "Rahul"
```

3. Set an email for your git commit message.

```
git config user.email "<test@email.com>"
```

4. Verify the username and email you set.

```
git config -list
```

5. Check the status, see files that have been changed or added to your git repository.



```
git status
```

6. Add files to the git staging area.

```
git add
```

7. Check the status, see files that have been added to the git staging area.

```
git status
```

8. Commit your files with a commit message.

```
git commit -m "My first commit"
```

9. Push the commit to your remote git repository.

```
git push
```

```
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ pwd
/home/ubuntu/demo-nodejs-app
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git config user.name "Rahul"
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git config user.email "rahul@rahul.com"
core.repositoryformatversion=0
core.filemode=true
core.ignorecase=true
core.symlinks=false
core.packedgit=true
remote.origin.url=https://github.com/shivalkarrahul/demo-nodejs-app.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.master.remote=origin
branch.master.merge=refs/heads/master
user.name=Rahul
user.email="rahul@rahul.com"
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    Dockerfile
    Jenkinsfile
    README.md
    package.json
    script.sh
    src/
    task-definition.json

nothing added to commit but untracked files present (use "git add" to track)
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git add .
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:  Dockerfile
    new file:  Jenkinsfile
    new file:  README.md
    new file:  package.json
    new file:  script.sh
    new file:  src/server.js
    new file:  src/test/server.test.js
    new file:  task-definition.json

ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git commit -m "My first commit"
[master (root-commit) 8ec6c2] My first commit
  8 files changed, 214 insertions(+)
create mode 100644 Dockerfile
create mode 100644 Jenkinsfile
create mode 100644 README.md
create mode 100644 package.json
create mode 100644 script.sh
create mode 100644 src/server.js
create mode 100644 src/test/server.test.js
create mode 100644 task-definition.json
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git push
Username for 'https://github.com: [REDACTED]@github.com':
Password for 'https://[REDACTED]@github.com':[REDACTED]
Counting objects: 12, done.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (12/12), 2.91 KiB | 2.91 MiB/s, done.
Total 12 (delta 0), reused 0 (delta 0)
To https://github.com/shivalkarrahul/demo-nodejs-app.git
 * [new branch]  master -> master
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$
```



## Setup the AWS Infrastructure

### Create an IAM User With a Programmatic Access

1. Create an IAM user with programmatic access in your AWS account and note down the access key and secret key in your text file for future reference. Provide administrator permissions to the user.  
We don't need admin access. However, to avoid permission issues and for the sake of the demo, let's proceed with administrator access.

**Summary**

User ARN: arn:aws:iam:::user/demo-admin-user

Path: /

Creation time: 2021-10-10 12:52 UTC+0530

Permissions Groups Tags Security credentials Access Advisor

Permissions policies (1 policy applied)

Add permissions Add inline policy

Policy name: AdministratorAccess

Attached directly: AdministratorAccess

Policy type: AWS managed policy

## Create an ECR Repository

1. Create an ECR Repository in your AWS account and note its URL in your text file for future reference.



Successfully created repository demo

Amazon Container Services > ECR > Repositories

Private Public

Private repositories (1 of 1)

Repository name	URI	Created at	Tag immutability	Scan on push	Encryption type
demo	dkr.ecr.eu-west-3.amazonaws.com/demo	10 October 2021, 12:09:29 (UTC+05:5)	Disabled	Disabled	AES-256

## Create an ECR Repository

1. Create an ECR Repository in your AWS account and note its URL in your text file for future reference.

Successfully created repository demo

Amazon Container Services > ECR > Repositories

Private Public

Private repositories (1 of 1)

Repository name	URI	Created at	Tag immutability	Scan on push	Encryption type
demo	dkr.ecr.eu-west-3.amazonaws.com/demo	10 October 2021, 12:09:29 (UTC+05:5)	Disabled	Disabled	AES-256

## Create an ECS Cluster

1. Go to ECS Console and click on “Get Started” to create a cluster.

The screenshot shows the 'Clusters' section of the Amazon ECS console. On the left, there's a sidebar with links like 'Amazon ECS', 'Clusters', 'Task Definitions', etc. In the main area, there's a brief description of what an ECS cluster is, followed by a 'Create Cluster' button and a 'Get Started' button. Below these are three view options: 'View', 'List', and 'Card'. A message 'No clusters found' is displayed, along with a 'Get Started' button. At the bottom right, there are navigation buttons for 'view all' and '0 - 0 of 0'.

2. Click on the “Configure” button available in the “custom” option under “Container definition.”



The screenshot shows the 'Container definition' configuration page. It has two sections: 'Container definition' and 'Task definition'. In the 'Container definition' section, there are several pre-defined container images like 'sample-app', 'nginx', and 'tomcat-webserver', each with its own configuration details. Below them is a 'custom' section with a 'Configure' button. In the 'Task definition' section, there's a 'Task definition name' field set to 'first-run-task-definition' and other fields like 'Network mode' (awsvpc), 'Task execution role' (Create new), 'Compatibilities' (FARGATE), 'Task memory' (0.5GB (512)), and 'Task CPU' (0.25 vCPU (256)). At the bottom, there are 'Required' and 'Next' buttons.

3. Specify a name to the container as “nodejs-container,” the ECR Repository URL in the “Image” text box, “3000” port in the Port mappings section, and then click on the “Update” button. You can specify any name of your choice for the container.

4. You can now see the details you specified under “Container definition.” Click on the “Next” button to proceed.

5. Select “Application Load Balancer” under “Define your service” and then click on the “Next” button.

Diagram of ECS objects and how they relate

```
graph LR; Container[Container definition] --- Task[Task definition]; Task --- Service[Service]; Service --- Cluster[Cluster]
```

Define your service

A service allows you to run and maintain a specified number (the "desired count") of simultaneous instances of a task definition in an ECS cluster.

Service name  Edit

Number of desired tasks

Security group  Automatically create new  
Two security groups are created to secure your service: An Application Load Balancer security group that allows all traffic on the Application Load Balancer port and an Amazon ECS security group that allows all traffic ONLY from the Application Load Balancer security group. You can further configure security groups and network access outside of this wizard.

Load balancer type  None  Application Load Balancer

Load balancer listener port

Load balancer listener protocol

\*Required Cancel Previous Next

6. Keep the cluster name as “default” and proceed by clicking on the “Next” button. You can change the cluster name if you want.

Getting Started with Amazon Elastic Container Service (Amazon ECS) using Fargate

Diagram of ECS objects and how they relate

Step 1: Container and Task  
Step 2: Service  
**Step 3: Cluster**  
Step 4: Review

Configure your cluster

The infrastructure in a Fargate cluster is fully managed by AWS. Your containers run without you managing and configuring individual Amazon EC2 instances.

To see key differences between Fargate and standard ECS clusters, see the [Amazon ECS documentation](#).

Cluster name **default**

Cluster names are unique per account per region. Up to 255 letters (uppercase and lowercase), numbers, and hyphens are allowed.

VPC ID Automatically create new

Subnets Automatically create new

\*Required Cancel Previous Next



7. Review the configuration and it should look as follows. If the configurations match, then click on the “Create” button. This will initiate the ECS Cluster creation.

Review

Review the configuration you've set up before creating your task definition, service, and cluster.

**Task definition**

Task definition name **first-run-task-definition**

Network mode **awsvpc**

Task execution role **Create new**

Container name **nodejs-container**

Image **dkr.ecr.eu-west-3.amazonaws.com/demo**

Memory **512**

Port **3000**

Protocol **HTTP**

**Service**

Service name **nodejs-container-service**

Number of desired tasks **1**

Load balancer listener port **3000**

Load balancer listener protocol **HTTP**

**Cluster**

Cluster name **default**

VPC ID Automatically create new

Subnets Automatically create new

\*Required Cancel Previous Create

8. After a few minutes, you should have your ECS cluster created and the Launch Status should be something as follows.

## Create an EC2 Instance for setting up the Jenkins Server

1. Create an EC2 Instance with Ubuntu 18.04 AMI and open its Port 22 for you and Port 8080 for 0.0.0.0/0 in its Security Group. Port 22 will be required for ssh g\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\h\\ and 8080 for accessing the Jenkins Server. Port 8080 is where GitHub Webhook will try to connect to on Jenkins Server, hence we need to allow it for 0.0.0.0/0 .



## Setup Jenkins on the EC2 Instance

After the instance is available, let's install Jenkins Server on it along with all the dependencies.

## Prerequisites of the EC2 Instance

1. Verify if the OS is Ubuntu 18.04 LTS

```
cat/
```

2. Check the RAM, a minimum of 2 GB is what we require.

```
free -m
```

3. The User that you use to log in to the server should have sudo privileges. "ubuntu" is the user available with sudo privileges for EC2 instances created using "Ubuntu 18.04 LTS" AMI.

```
whoami
```

#### 4. Check your present working directory, it will be your home directory.

```
pwd
```

```
[ubuntu@ip-172-31-24-191:~$ cat /etc/issue
Ubuntu 18.04.5 LTS \n \l

[ubuntu@ip-172-31-24-191:~$ free -m
      total        used        free      shared  buff/cache   available
Mem:       1985         149       1307          0         528       1690
Swap:          0          0          0

[ubuntu@ip-172-31-24-191:~$ whoami
ubuntu
[ubuntu@ip-172-31-24-191:~$ pwd
/home/ubuntu
ubuntu@ip-172-31-24-191:~$ ]
```

### Install Java, JSON Processor jq, Nodejs/NPM and aws-cli on the EC2 Instance

1. Update your system by downloading package information from all configuration sources.



```
sudo apt update
```

2. Search and Install Java 11

```
sudo apt search openjdk
sudo apt install openjdk-11-jdk
```

3. Install jq command, the JSON processor.

```
sudo apt install jq
```

4. Install Nodejs 12 and NPM

```
curl -sL https://deb.nodesource.com/setup\_12.x | sudo -E bash -
sudo apt install nodejs
```

5. Install the AWS CLI tool.

```
sudo apt install awscli
```

6. Check the Java version.

```
java -version
```

7. Check the jq version.

```
jq -version
```

## 8. Check the Nodejs version

```
node --version
```

## 9. Check the NPM version

```
npm --version
```

## 10. Check the AWS CLI version

```
aws --version
```

```
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ java --version
openjdk 11.0.11 2021-04-20
OpenJDK Runtime Environment (build 11.0.11+9-Ubuntu-0ubuntu2.18.04)
OpenJDK 64-Bit Server VM (build 11.0.11+9-Ubuntu-0ubuntu2.18.04, mixed mode, sharing)
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ jq --version
jq-1.5-1-a5b5cbe
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ node --version
v12.22.6
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ npm --version
6.14.15
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ aws --version
aws-cli/1.18.69 Python/3.6.9 Linux/5.4.0-1045-aws botocore/1.16.19
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$
```



Note: Make sure all your versions match the versions seen in the above image.

## Install Jenkins on the EC2 Instance

### 1. Jenkins can be installed from the Debian repository

```
wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.d/jenkins.list'
```

### 2. Update the apt package index

```
sudo apt-get update
```

### 3. Install Jenkins on the machine

```
sudo apt-get install jenkins
```

### 4. Check the service status if it is running or not.

```
service jenkins status
```

### 5. You should have your Jenkins up and running now. You may refer to the official

documentation [here](#) if you face any issues with the installation.

```
ubuntu@ip-172-31-24-191:~$ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | sudo apt-key add -
OK
ubuntu@ip-172-31-24-191:~$ sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.d/jenkins.list'
ubuntu@ip-172-31-24-191:~$ sudo apt-get update
Hit:1 http://eu-west-3.ec2.archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 http://eu-west-3.ec2.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:3 http://eu-west-3.ec2.archive.ubuntu.com/ubuntu bionic-backports InRelease
Hit:4 http://security.ubuntu.com/ubuntu bionic-security InRelease
Ign:5 http://pkg.jenkins-ci.org/debian binary/ InRelease
Get:6 http://pkg.jenkins-ci.org/debian binary/ Release [2044 B]
Get:7 http://pkg.jenkins-ci.org/debian binary/ Release.gpg [833 B]
Get:8 http://pkg.jenkins-ci.org/debian binary/ Packages [39.5 kB]
Fetched 42.4 kB in 1s (65.4 kB/s)
Reading package lists... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  daemon
The following NEW packages will be installed:
  daemon jenkins
0 upgraded, 2 newly installed, 0 to remove and 88 not upgraded.
Need to get 71.8 MB of archives.
After this operation, 72.5 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://eu-west-3.ec2.archive.ubuntu.com/ubuntu bionic/universe amd64 daemon amd64 0.6.4-1build1 [99.5 kB]
Get:2 http://pkg.jenkins-ci.org/debian binary/ jenkins 2.315 [71.7 MB]
Fetched 71.8 MB in 1min 3s (1143 kB/s)
Selecting previously unselected package daemon.
(Reading database ... 60167 files and directories currently installed.)
Preparing to unpack .../daemon_0.6.4-1build1_amd64.deb ...
Unpacking daemon (0.6.4-1build1) ...
Selecting previously unselected package jenkins.
Preparing to unpack .../archives/jenkins_2.315_all.deb ...
Unpacking jenkins (2.315) ...
Setting up daemon (0.6.4-1build1) ...
Setting up jenkins (2.315) ...
Processing triggers for system (237-3ubuntu10.46) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
Processing triggers for ureadahead (0.100.0-21)
ubuntu@ip-172-31-24-191:~$ service jenkins status
● jenkins.service - LSB: Start Jenkins at boot time
  Loaded: loaded (/etc/init.d/jenkins; generated)
    Active: active (exited) since Sun 2021-10-10 07:04:44 UTC; 40s ago
      Docs: man:systemd-sysv-generator(8)
        Tasks: 0 (limit: 2347)
       CGroub: /system.slice/jenkins.service
Oct 10 07:04:43 ip-172-31-24-191 systemd[1]: Starting LSB: Start Jenkins at boot time...
Oct 10 07:04:43 ip-172-31-24-191 jenkins[8683]: Correct java version found
Oct 10 07:04:43 ip-172-31-24-191 jenkins[8683]: * Starting Jenkins Automation Server jenkins
Oct 10 07:04:43 ip-172-31-24-191 su[8734]: Successful su for jenkins by root
Oct 10 07:04:43 ip-172-31-24-191 su[8734]: + ?? root:jenkins
Oct 10 07:04:43 ip-172-31-24-191 su[8734]: pam_unix(su:session): session opened for user jenkins by (uid=0)
Oct 10 07:04:43 ip-172-31-24-191 su[8734]: pam_unix(su:session): session closed for user jenkins
Oct 10 07:04:44 ip-172-31-24-191 jenkins[8683]: ...done.
Oct 10 07:04:44 ip-172-31-24-191 systemd[1]: Started LSB: Start Jenkins at boot time.
ubuntu@ip-172-31-24-191:~$
```



## Install Docker on the EC2 Instance

1. Install packages to allow apt to use a repository over HTTPS:

```
sudo apt-get install apt-transport-https ca-certificates curl gnupg lsb-
release
```

2. Add Docker's official GPG key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

3. Set up the stable repository

```
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-
keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs)
stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

4. Update the apt package index

```
sudo apt-get update
```

5. Install the latest version of Docker Engine and containerd,

```
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

6. Check the docker version.

```
docker -version
```

7. Create a “docker” group, this may exit.

```
sudo groupadd docker
```

8. Add “ubuntu” user to the “docker” group

```
sudo usermod -aG docker ubuntu
```

9. Add “jenkins” user to the “docker” group

```
sudo usermod -aG docker jenkins
```



10. Test if you can create docker objects using “ubuntu” user.

```
docker run hello-world
```

11. Switch to “root” user

```
sudo -i
```

12. Switch to “jenkins” user

```
su jenkins
```

13. Test if you can create docker objects using “jenkins” user.

```
docker run hello-world
```

14. Exit from “jenkins” user

```
exit
```

15. Exit from “root” user

```
exit
```

16. Now you should be back in “ubuntu” user. You may refer to the official documentation [here](#) if you face any issues with the installation.

```
[ubuntu@ip-172-31-24-191:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:9ade9cc2e26189a19c2e8854b9c8f1e14829b51c55a630ee675a5a9540ef6ccf
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

```
[ubuntu@ip-172-31-24-191:~$ sudo -i
[root@ip-172-31-24-191:~# su jenkins
[jenkins@ip-172-31-24-191:/root$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

```
[jenkins@ip-172-31-24-191:/root$ exit
exit
[root@ip-172-31-24-191:~# exit
logout
ubuntu@ip-172-31-24-191:~$ ]
```



## Configure the Jenkins Server

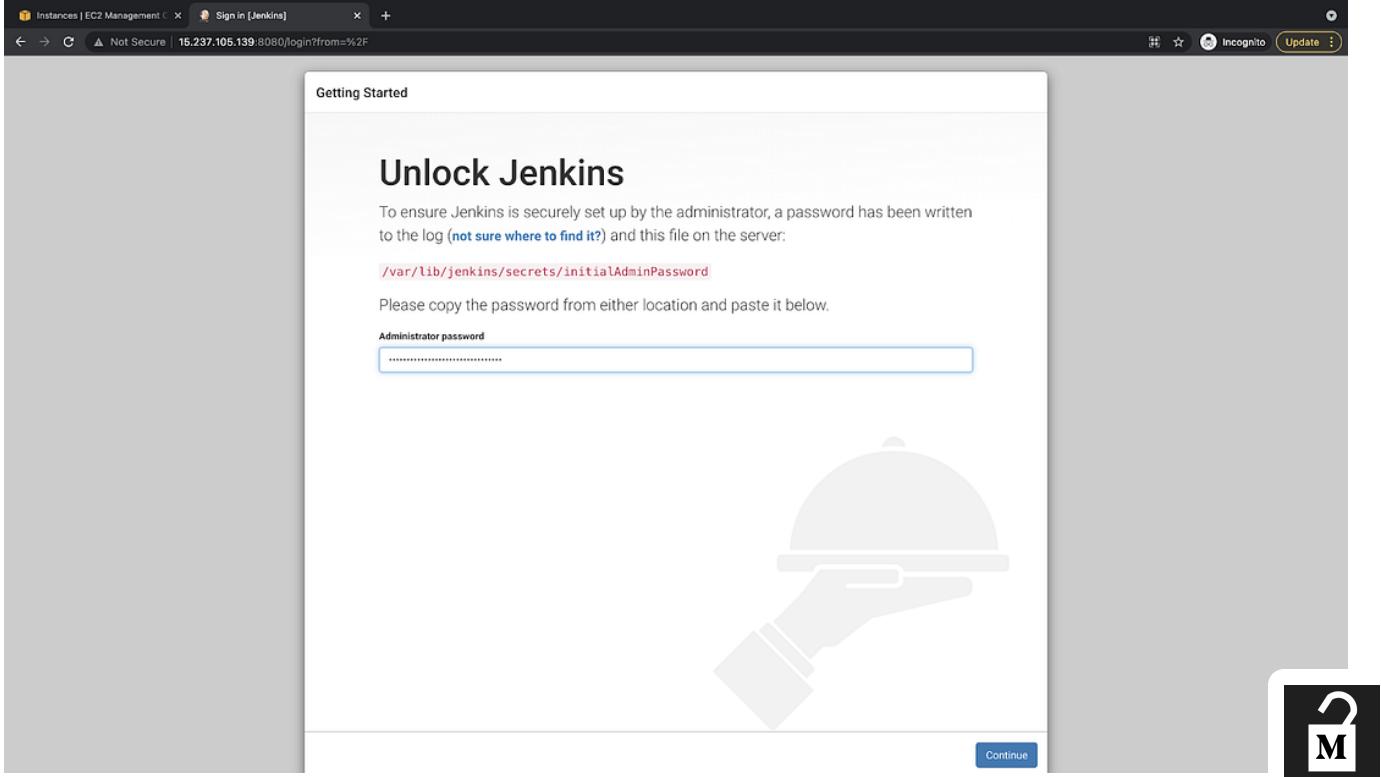
- After Jenkins has been installed, the first step is to extract its password.

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

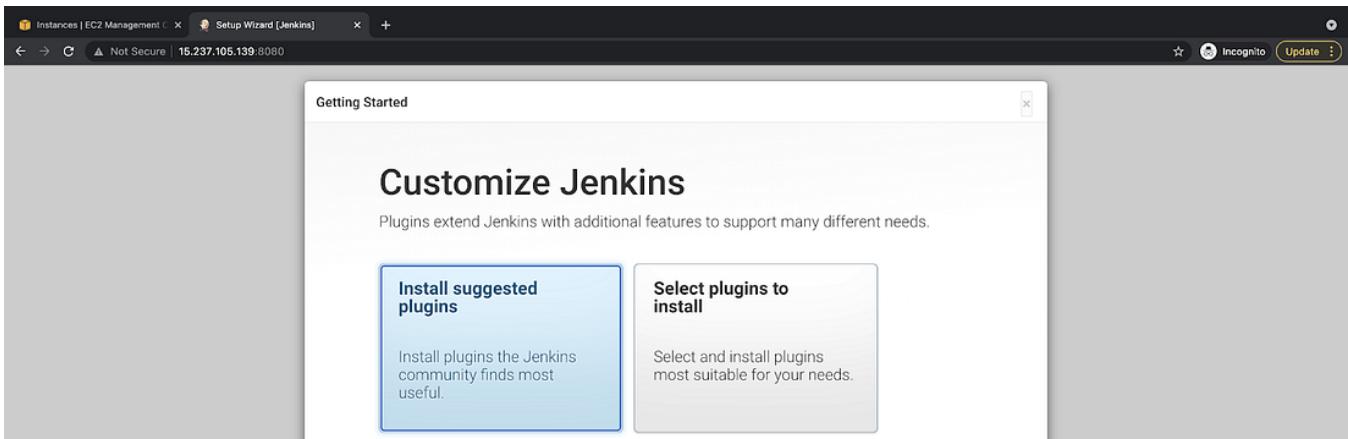
```
[ubuntu@ip-172-31-24-191:~$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword
218ff266f45840e180495cb1dd93b339
ubuntu@ip-172-31-24-191:~$ ]
```

- Hit the URL in the browser

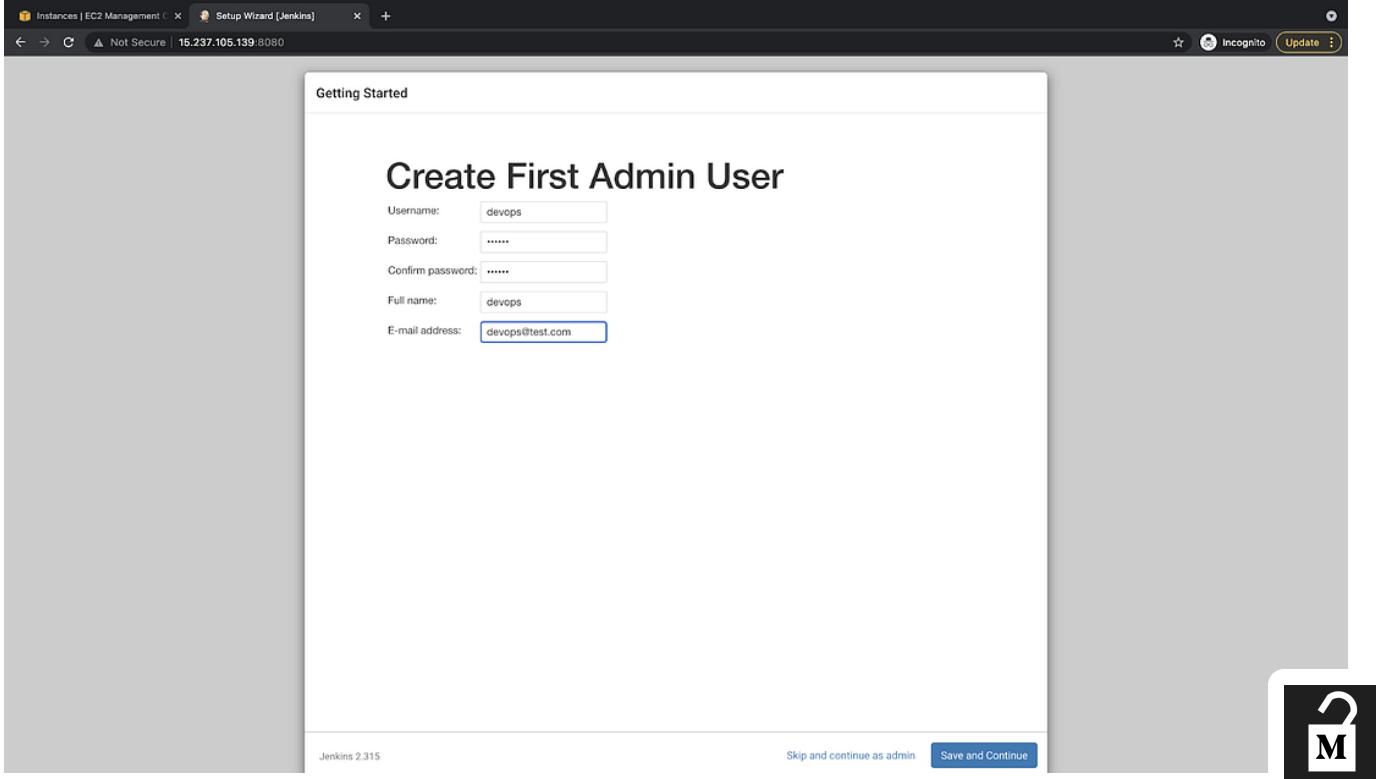
Jenkins URL: <http://<public-ip-of-the-ec2-instance>:8080>



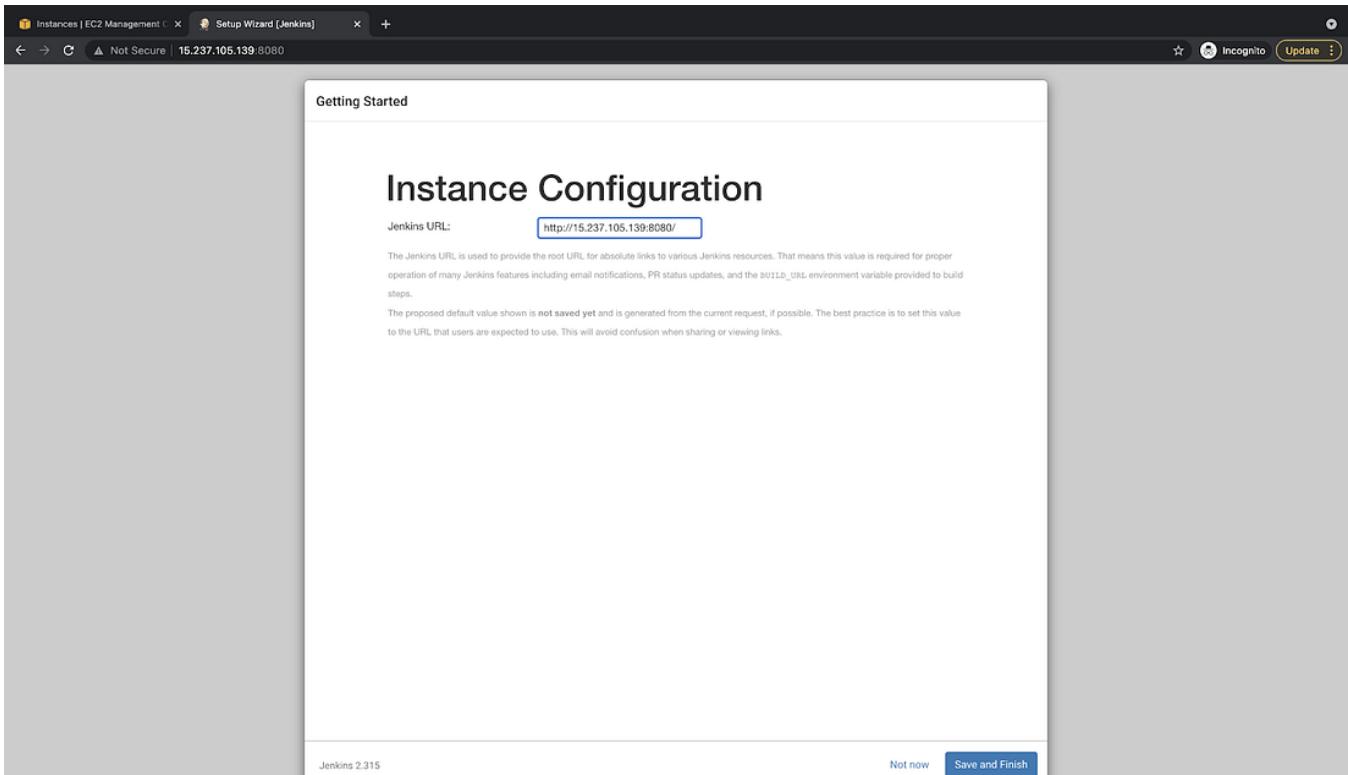
### 3. Select the “Install suggested plugins” option



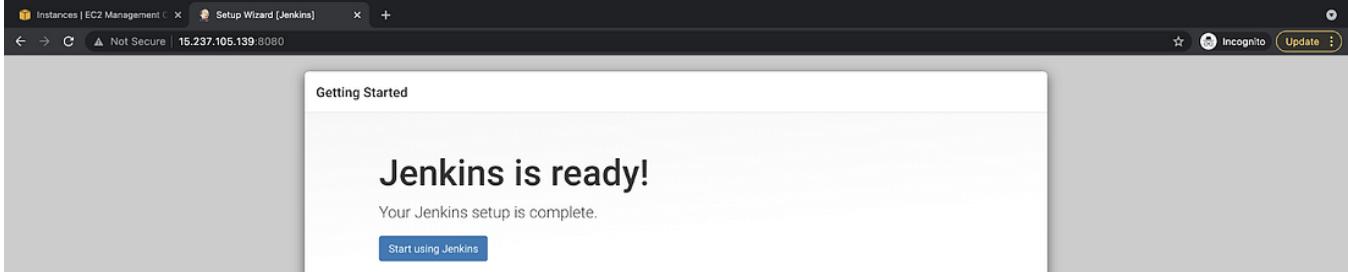
### 4. Specify user-name, password for the new admin user to be created. You can use this user as an admin user.



5. This URL field will be auto-filled, click on the “Save and Finish” button to proceed.



6. Your Jenkins Server is ready now.



7. Here is what its Dashboard looks like:

The dashboard features a sidebar with links for New Item, People, Build History, Manage Jenkins, My Views, Lockable Resources, and New View. The main area includes sections for "Welcome to Jenkins!", "Start building your software project" (with a "Create a job" button), "Set up a distributed build" (with "Set up an agent" and "Configure a cloud" buttons), and "Learn more about distributed builds". A large "M" logo is visible in the top right.

## Install Plugins

1. Let's install all the plugins that we will need. Click on “Manage Jenkins” in the left panel.

The Manage Jenkins page shows the "System Configuration" section with links for "Configure System", "Global Tool Configuration", and "Manage Plugins". A note at the top states: "Building on the built-in node can be a security issue. You should set up distributed builds. See [the documentation](#)." Buttons for "Set up agent", "Set up cloud", and "Dismiss" are also present.

2. Here is a list of plugins that we need to install

1. CloudBees AWS Credentials:

Allows storing Amazon IAM credentials, keys within the Jenkins Credentials API.

## 2. Docker Pipeline:

This plugin allows building, testing, and using Docker images from Jenkins Pipeline.

## 3. Amazon ECR:

This plugin provides integration with AWS Elastic Container Registry (ECR)  
Usage:

## 4. AWS Steps:

This plugin adds Jenkins pipeline steps to interact with the AWS API.

3. In the “Available” tab, search all these plugins and click on “Install without restart.”

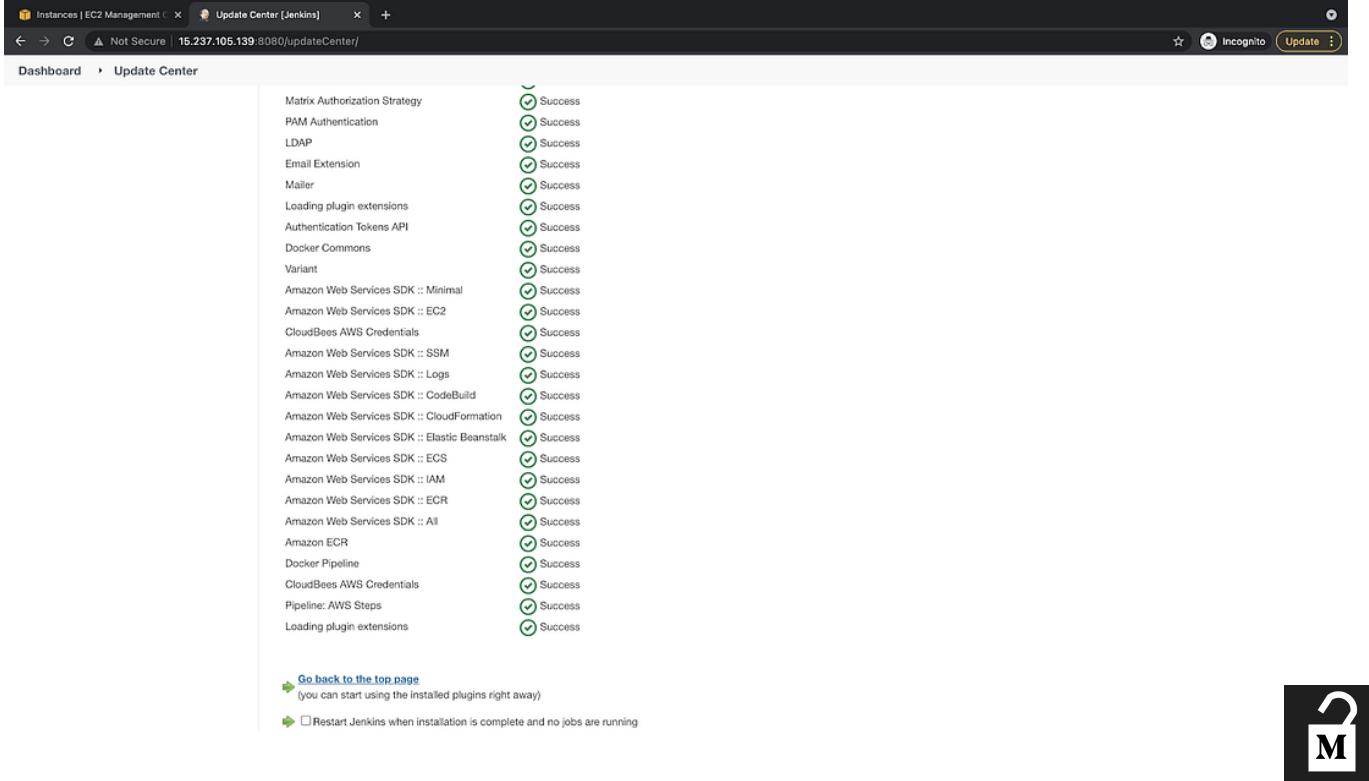


The screenshot shows the Jenkins Plugin Manager interface. The top navigation bar has tabs for 'Available Plugins [Jenkins]' and 'Available'. Below the navigation is a search bar with the query 'AWS Steps'. The main content area displays a list of available plugins:

Name	Version	Released
CloudBees AWS Credentials	1.32	8 days 17 hr ago
Docker Pipeline	1.26	7 mo 17 days ago
Amazon ECR	1.6	4 yr 4 mo ago
Pipeline: AWS Steps	1.43	10 mo ago

At the bottom of the list, there are three buttons: 'Install without restart', 'Download now and install after restart', and 'Check now'.

4. You will see the screen as follows after the plugins have been installed successfully.



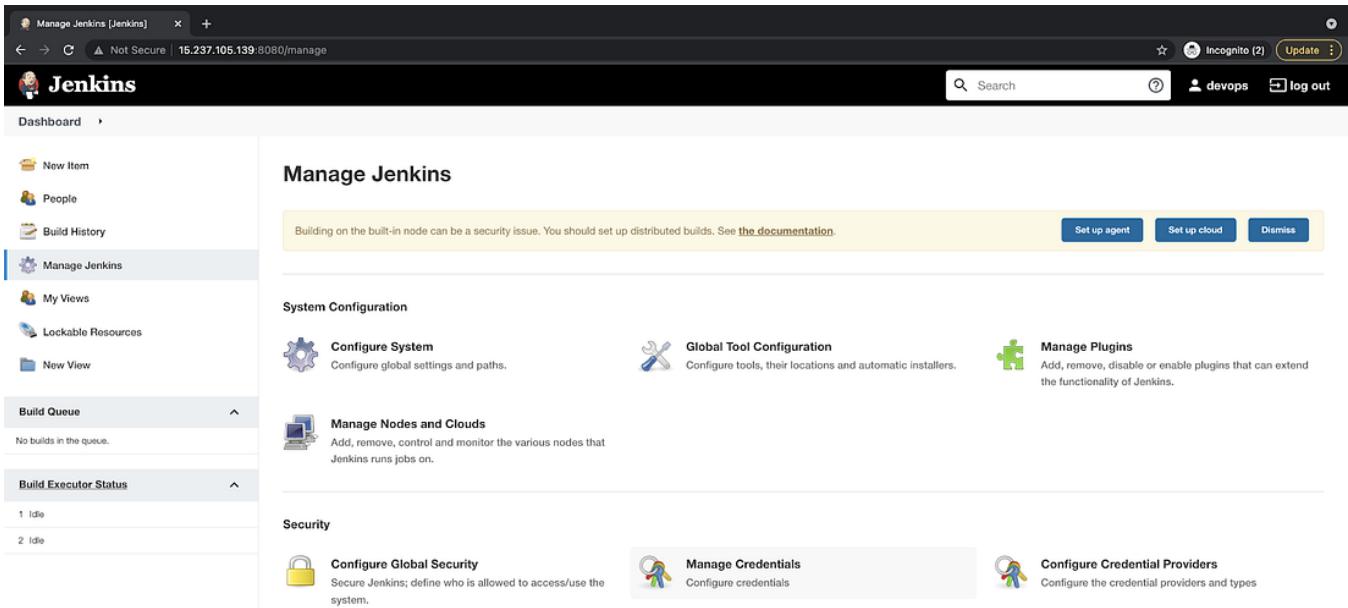
The screenshot shows the Jenkins Update Center interface. On the left, there's a sidebar with links like 'Dashboard', 'Update Center', 'Manage Jenkins', etc. The main area lists various Jenkins plugins with their status: 'Success'. Some of the listed items include:

- Matrix Authorization Strategy
- PAM Authentication
- LDAP
- Email Extension
- Mailer
- Loading plugin extensions
- Authentication Tokens API
- Docker Commons
- Variant
- Amazon Web Services SDK :: Minimal
- Amazon Web Services SDK :: EC2
- CloudBees AWS Credentials
- Amazon Web Services SDK :: SSM
- Amazon Web Services SDK :: Logs
- Amazon Web Services SDK :: CodeBuild
- Amazon Web Services SDK :: CloudFormation
- Amazon Web Services SDK :: Elastic Beanstalk
- Amazon Web Services SDK :: ECS
- Amazon Web Services SDK :: IAM
- Amazon Web Services SDK :: ECR
- Amazon Web Services SDK :: All
- Amazon ECR
- Docker Pipeline
- CloudBees AWS Credentials
- Pipeline: AWS Steps
- Loading plugin extensions

At the bottom, there are two buttons: 'Go back to the top page' and 'Restart Jenkins when installation is complete and no jobs are running'. To the right of the main content area is a large black square icon containing a white 'M' with a lock symbol over it.

## Create Credentials in Jenkins

1. CloudBees AWS Credentials plugin will come to the rescue here. Go to “Manage Jenkins”, and then click on “Manage Credentials.”



The screenshot shows the Jenkins Manage Jenkins dashboard. On the left, there's a sidebar with links like 'Dashboard', 'New Item', 'People', 'Build History', 'Manage Jenkins' (which is currently selected), 'My Views', 'Lockable Resources', and 'New View'. The main area has a title 'Manage Jenkins' and a message: 'Building on the built-in node can be a security issue. You should set up distributed builds. See [the documentation](#).'. Below this, there are several configuration sections:

- System Configuration**
  - Configure System**: Configure global settings and paths.
  - Global Tool Configuration**: Configure tools, their locations and automatic installers.
  - Manage Plugins**: Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
- Security**
  - Configure Global Security**: Secure Jenkins; define who is allowed to access/use the system.
  - Manage Credentials**: Configure credentials.
  - Configure Credential Providers**: Configure the credential providers and types.

2. Click on “(global).” “Add credentials.”

The screenshot shows the Jenkins 'Credentials' page. On the left, there's a sidebar with links like 'New Item', 'People', 'Build History', etc. The main area has a title 'Credentials' with a search bar. Below it is a table with columns 'Domain', 'ID', and 'Name'. A sub-section titled 'Stores scoped to Jenkins' shows a table with a single row for 'Jenkins'. At the bottom right of this section is a blue button labeled 'Add credentials'.

3. Select Kind as “AWS Credentials” and provide ID as “demo-admin-user”. This can be provided as per your choice, keep a note of this ID in the text file. Specify the Access Key and Secret Key of the IAM user we created in the previous steps.

Click on “OK” to store the IAM credentials.

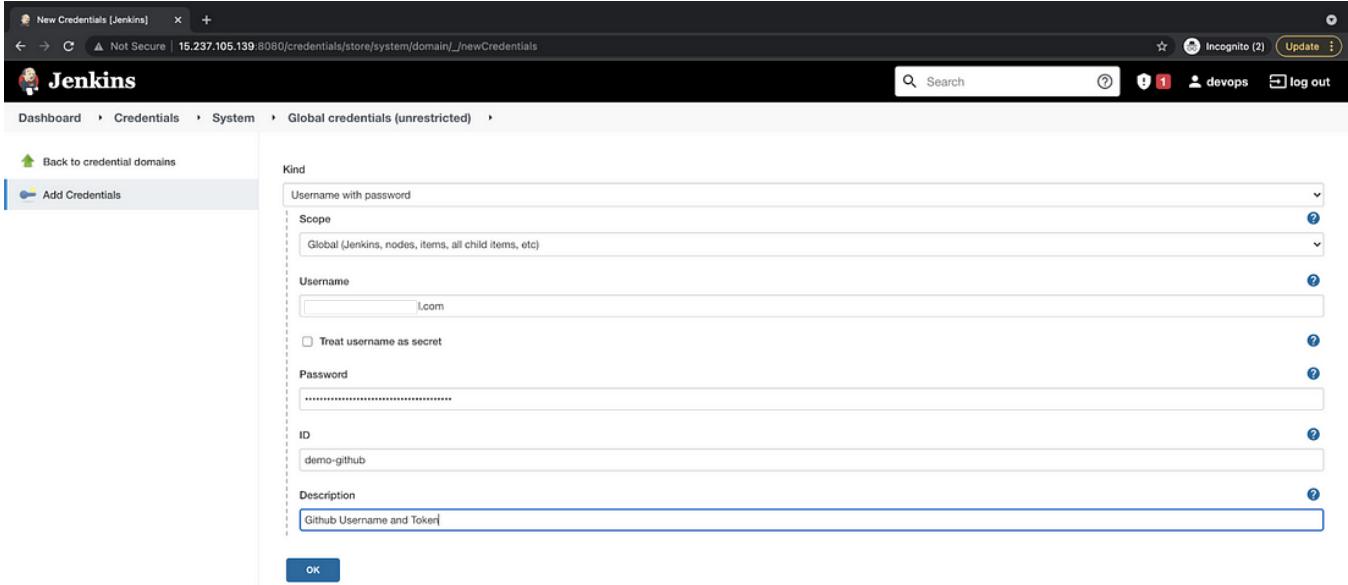
The screenshot shows the 'New Credentials' configuration page for Jenkins. The URL is '15.237.105.139:8080/credentials/store/system/domain/\_/newCredentials'. The form is for 'AWS Credentials' with the following fields filled in:

- Scope:** Global (Jenkins, nodes, items, all child items, etc)
- ID:** demo-admin-user
- Access Key ID:** AKIAQ6GAIASXHOMTRYLW
- Secret Access Key:** (Redacted)
- IAM Role Support:** (checkbox)

A red error message at the bottom says: "Please specify the Secret Access Key". At the bottom right is a blue 'OK' button.

4. Follow the same step and this time select Kind as “Username with password” to store the GitHub Username and Token we created earlier.

Click on “OK” to store the GitHub credentials.

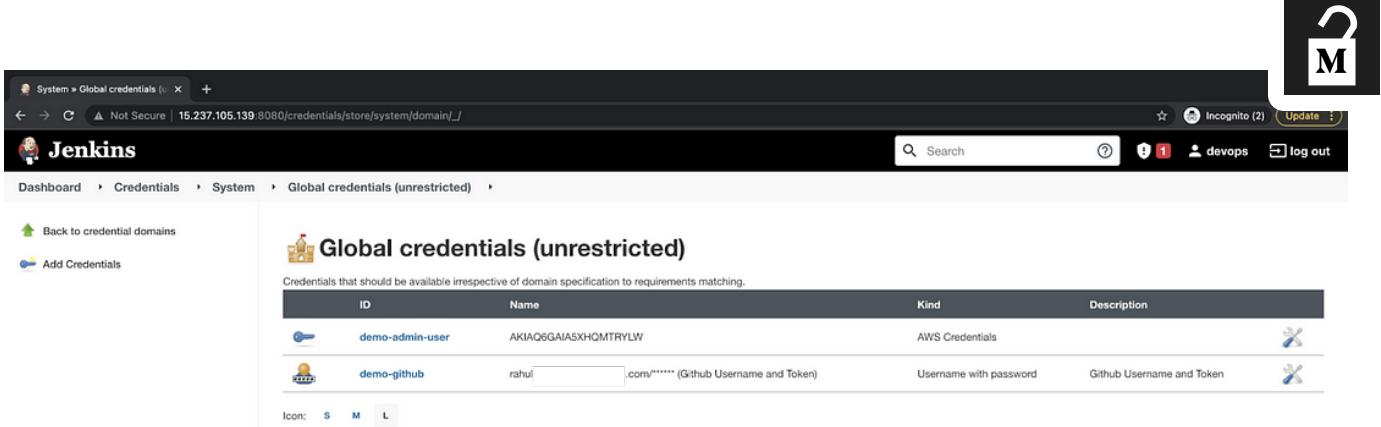


The screenshot shows the Jenkins 'New Credentials' interface. A 'Username with password' credential is being created with the following details:

- Scope:** Global (Jenkins, nodes, items, all child items, etc)
- Username:** l.com
- Password:** (redacted)
- ID:** demo-github
- Description:** Github Username and Token

An 'OK' button is at the bottom.

5. You should now have IAM and GitHub credentials in your Jenkins.

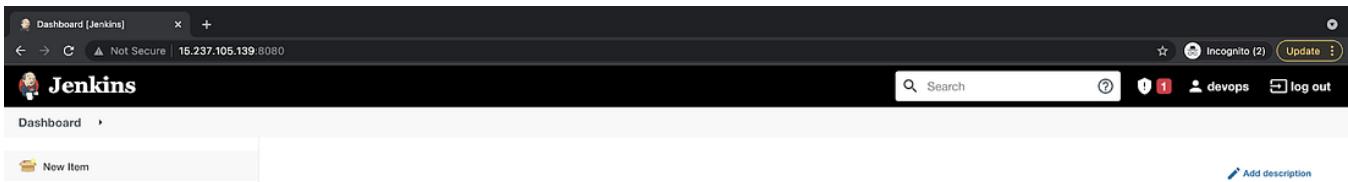


The screenshot shows the Jenkins 'Global credentials (unrestricted)' page listing two credentials:

ID	Name	Kind	Description
demo-admin-user	AKIAQ6GAIASXHQMTRYLW	AWS Credentials	(Icon)
demo-github	rahul.l.com/***** (Github Username and Token)	Username with password	Github Username and Token (Icon)

## Create a Jenkins Job

1. Go to the main dashboard, and click on “New Item” to create a Jenkins Pipeline.



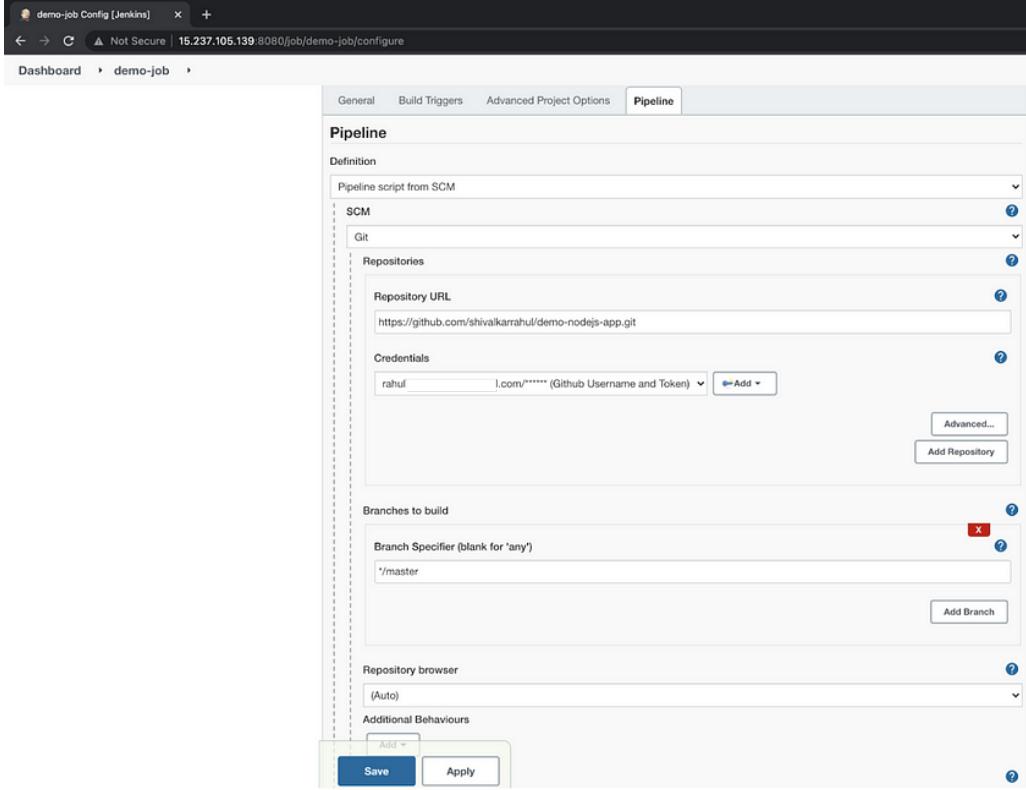
The screenshot shows the Jenkins dashboard with a 'New Item' button highlighted.

2. Select the “Pipeline” and name it “demo-job” or provide a name of your choice.

3. Tick the “GitHub project” checkbox under the “General” tab, provide the GitHub Repository URL of the one we created earlier. Also, tick the checkbox “GitHub hook trigger for GitScm polling” under the “Build Trigger” tab.

4. Under the “Pipeline” tab, select “Pipeline script from the SCM” definition, specify our repository URL and select the credential we created for GitHub. Check the branch name if it matches the one you will be using for your commits.

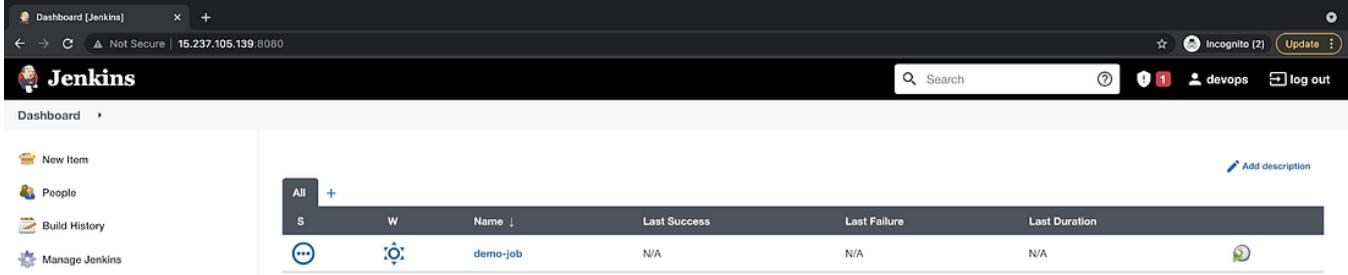
Review the configurations and click on “Save” to save your changes to the pipeline.



The screenshot shows the Jenkins Pipeline configuration page. The job name is 'demo-job'. The 'Pipeline' tab is selected. In the 'Definition' section, 'Pipeline script from SCM' is chosen, and 'Git' is selected under 'SCM'. The 'Repository URL' is set to <https://github.com/shivalkarrahul/demo-nodejs-app.git>. Under 'Credentials', a credential named 'rahul' is selected. The 'Branches to build' section shows a branch specifier '/master'. The pipeline has been saved.



5. Now you can see the pipeline we just created.



The screenshot shows the Jenkins dashboard. The 'All' tab is selected, displaying a list of pipelines. One pipeline, 'demo-job', is shown with a status of 'In Progress' (indicated by a blue circle icon). Other columns include 'Name' (sorted by name), 'Last Success', 'Last Failure', and 'Last Duration'. The pipeline 'demo-job' has a status of 'N/A' for all these metrics.

## Integrate GitHub and Jenkins

The next step is to integrate GitHub with Jenkins so that whenever there is an event on the GitHub Repository, it can trigger the Jenkins Job.

1. Go to the settings tab of the repository, and click on “Webhooks” in the left panel. You can see the “Add webhook” button, click on it to create a webhook.

The screenshot shows the GitHub repository settings page for 'shivalkarrahal / demo-nodejs-app'. The 'Webhooks' tab is selected in the navigation bar. On the left, there's a sidebar with options like Options, Manage access, Repository roles, Security & analysis, Branches, and Webhooks (which is highlighted). The main content area is titled 'Webhooks' and contains a brief description: 'Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#)'. A 'Add webhook' button is located at the top right of this section.

2. Provide the Jenkins URL with context as “/github-webhook/”. The URL will look as follows.

Webhook URL: `http://<Jenkins-IP>:8080/github-webhook/`

You can select the events of your choice. However, for the sake of simplicity I have chosen “Send me everything.”



Make sure the “Active” checkbox is checked.

Click on “Add webhook” to create a webhook that will trigger the Jenkins job whenever there is any kind of event in the GitHub Repository.

The screenshot shows the 'Add webhook' form within the GitHub repository settings. The sidebar on the left has 'Webhooks' selected. The main form area is titled 'Webhooks / Add webhook' and contains the following fields:

- Payload URL \***: `http://15.237.105.139:8080/github-webhook/`
- Content type**: `application/x-www-form-urlencoded`
- Secret**: An empty input field.
- Which events would you like to trigger this webhook?**
  - Just the push event.
  - Send me everything.
  - Let me select individual events.
- Active**: A checked checkbox with the note: "We will deliver event details when this hook is triggered."

A green 'Add webhook' button is at the bottom of the form.

3. You should see your webhook. Click on it to see if it has been configured correctly or not.

The screenshot shows the GitHub repository settings page for 'shivalkarrahal / demo-nodejs-app'. The 'Webhooks' section is active. A message at the top says 'Okay, that hook was successfully created. We sent a ping payload to test it out! Read more about it at https://docs.github.com/webhooks/#ping-event.' Below this, a list of webhooks shows one entry: 'http://15.237.105.139:8080/github... (all events)' with 'Edit' and 'Delete' buttons.

- Click on the “Recent Deliveries” tab and you should see a green tick mark. The green tick mark shows that the webhook was able to connect to the Jenkins Server.

The screenshot shows the GitHub repository settings page for 'shivalkarrahal / demo-nodejs-app'. The 'Webhooks / Manage webhook' section is active. The 'Recent Deliveries' tab is selected, showing a single entry with a green checkmark and a timestamp: '2021-10-10 13:21:19'. To the right of the list is a large padlock icon with a 'M' inside.

## Deploy the Nodejs Application to the ECS Cluster

Before we trigger the Pipeline from GitHub Webhook. Let's try to manually execute it.

### Build the Job Manually

- Go to the Job we created and Build it.

The screenshot shows the Jenkins pipeline interface for 'Pipeline demo-job'. The left sidebar shows 'Status', 'Changes', 'Build Now' (with a 'Build Now' button), and 'Recent Changes'. The main area displays the pipeline stages: 'Pipeline demo-job' (status: 'Not Started'). On the right, there are 'Add description' and 'Disable Project' buttons.

- If you see its logs, you will see that it failed. The reason is, that we have not yet assigned values to variables we have in our Jenkinsfile.

## Push Your Second Commit

Reminder: For the rest of the article, do not change your directory. Stay in the same directory, i.e., /home/ubuntu/demo-nodejs-app, and execute all the commands from here.

### Assign values to the variable in the Jenkinsfile

1. To overcome the above error, you need to make some changes to the Jenkinsfile.

We have variables in that file and we need to assign values to those variables to deploy our application to the ECS cluster we created. Assign correct values to the variables having “CHANGE\_ME.”

cat Jenkinsfile

```
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ cat Jenkinsfile
pipeline {
    agent any
    environment {
        AWS_ACCOUNT_ID="CHANGE_ME"
        AWS_DEFAULT_REGION="CHANGE_ME"
        CLUSTER_NAME="CHANGE_ME"
        SERVICE_NAME="CHANGE_ME"
        TASK_DEFINITION_NAME="CHANGE_ME"
        DESIRED_COUNT="CHANGE_ME"
        IMAGE_REPO_NAME="CHANGE_ME"
        IMAGE_TAG="${env.BUILD_ID}"
        REPOSITORY_URI = "${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/${IMAGE_REPO_NAME}"
        registryCredential = "CHANGE_ME"
```



2. Here is the list of variables for your convenience.

We have the following variables in the Jenkinsfile.

1. AWS\_ACCOUNT\_ID="CHANGE\_ME"

Assign your AWS Account Number here.

2. AWS\_DEFAULT\_REGION="CHANGE\_ME"

Assign the region you created your ECS Cluster in

3. CLUSTER\_NAME="CHANGE\_ME"

Assign the name of the ECS Cluster that you created.

4. SERVICE\_NAME="CHANGE\_ME"

Assign the Service name that got created in the ECS Cluster.

5. TASK\_DEFINITION\_NAME="CHANGE\_ME"

Assign the Task name that got created in the ECS Cluster.

## 6. DESIRED\_COUNT="CHANGE\_ME"

Assing the number of tasks you want to be created in the ECS Cluster.

## 7. IMAGE\_REPO\_NAME="CHANGE\_ME"

Assign the ECR Repository URL

## 8. IMAGE\_TAG="\${env.BUILD\_ID}"

Do not change this.

## 9. REPOSITORY\_URI =

`"${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/${IMAGE_REPO_NAME}"`

Do not change this.



## 10. registryCredential = "CHANGE\_ME"

Assign the name of the credentials you created in Jenkins to store the AWS Access Key and Secret Key

## 3. Check the status to confirm that the file has been changed.

```
git status
cat Jenkinsfile
```

```
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   Jenkinsfile

no changes added to commit (use "git add" and/or "git commit -a")
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ cat Jenkinsfile
pipeline {
  agent any
  environment {
    AWS_ACCOUNT_ID="████████"
    AWS_DEFAULT_REGION="eu-west-3"
    CLUSTER_NAME="default"
    SERVICE_NAME="nodejs-container-service"
    TASK_DEFINITION_NAME="first-run-task-definition"
    DESIRED_COUNT="1"
    IMAGE_REPO_NAME="demo"
    IMAGE_TAG="${env.BUILD_ID}"
    REPOSITORY_URI = "${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/${IMAGE_REPO_NAME}"
    registryCredential = "demo-admin-user"
  }
}
```

## 4. Add a file to the git staging area, commit it and then push it to the remote GitHub Repository.

```
git status
git add Jenkinsfile
```

```
git commit -m "Assigned environment specific values in Jenkinsfile"
git push
```

## Error on Jenkins Server

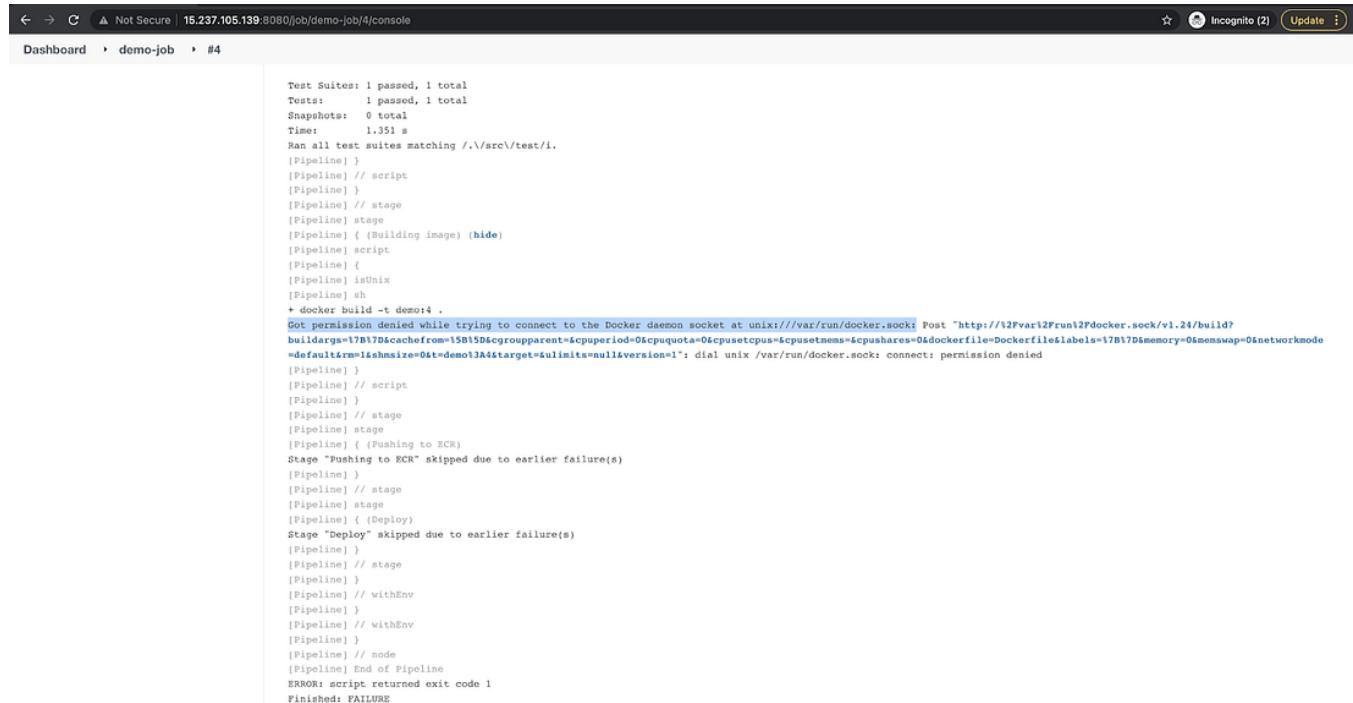
After pushing the commit, the Jenkins Pipeline will get triggered.

However, you will see an error “*Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock*” in your Jenkins Job.

The reason for this is that a “jenkins” user that is used by the Jenkins Job is not allowed to create docker objects. To give permission to a “jenkins” user, we added it to the “docker” group in the previous step. However, we did not restart the Jenkins service after that.



I kept this deliberately so that I could show you the need to add the “jenkins” user to the “docker” group in your EC2 Instance.



```
Test Suites: 1 passed, 1 total
Tests:    1 passed, 1 total
Snapshots: 0 total
Time:    1.351 s
Ran all test suites matching ./src/test/i.
[Pipeline]
[Pipeline] // script
[Pipeline]
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Building image) (hide)
[Pipeline] script
[Pipeline] {
[Pipeline] isUnix
[Pipeline] sh
+ docker build -t demo:4 .
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock; Post "http://127.0.0.1:2375/docker/v1.24/build?buildargs=&BuildCacheFrom=15b15d4cgrouparent=&CpuPeriod=0&CpuQuota=0&CpusetCpus=&CpusetMems=&Cpushares=0&Dockerfile=Dockerfile&Labels=17b17d5memory=0&MemSwap=0&NetworkMode=default&rm=1&ShmSize=0&t=demo13a4&target=&ulimits=null&version=1": dial unix /var/run/docker.sock: connect: permission denied
[Pipeline]
[Pipeline] // script
[Pipeline]
[Pipeline] // stage
[Pipeline] stage
[Pipeline] stage
[Pipeline] { (Pushing to ECR)
Stage "Pushing to ECR" skipped due to earlier failure(s)
[Pipeline]
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
Stage "Deploy" skipped due to earlier failure(s)
[Pipeline]
[Pipeline] // stage
[Pipeline] // withEnv
[Pipeline]
[Pipeline] // withEnv
[Pipeline]
[Pipeline] // node
[Pipeline] End of Pipeline
ERROR: script returned exit code 1
Finished: FAILURE
```

Now you know what needs to be done to overcome the above error.

### 1. Restart the Jenkins service.

```
sudo service jenkins restart
```

## 2. Check if the Jenkins service has started or not.

```
sudo service jenkins status
```

```
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ sudo service jenkins restart
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ sudo service jenkins status
● jenkins.service - LSB: Start Jenkins at boot time
  Loaded: loaded (/etc/init.d/jenkins; generated)
    Active: active (exited) since Sun 2021-10-10 09:28:41 UTC; 4s ago
      Docs: man:systemd-sysv-generator(8)
  Process: 26618 ExecStop=/etc/init.d/jenkins stop (code=exited, status=0/SUCCESS)
  Process: 26666 ExecStart=/etc/init.d/jenkins start (code=exited, status=0/SUCCESS)

Oct 10 09:28:40 ip-172-31-24-191 systemd[1]: Stopped LSB: Start Jenkins at boot time.
Oct 10 09:28:40 ip-172-31-24-191 systemd[1]: Starting LSB: Start Jenkins at boot time...
Oct 10 09:28:40 ip-172-31-24-191 jenkins[26666]: Correct java version found
Oct 10 09:28:40 ip-172-31-24-191 jenkins[26666]: * Starting Jenkins Automation Server jenkins
Oct 10 09:28:40 ip-172-31-24-191 su[26717]: Successful su for jenkins by root
Oct 10 09:28:40 ip-172-31-24-191 su[26717]: + ??? root:jenkins
Oct 10 09:28:40 ip-172-31-24-191 pam_unix(su:session): session opened for user jenkins by (uid=0)
Oct 10 09:28:40 ip-172-31-24-191 su[26717]: pam_unix(su:session): session closed for user jenkins
Oct 10 09:28:41 ip-172-31-24-191 jenkins[26666]: ...done.
Oct 10 09:28:41 ip-172-31-24-191 systemd[1]: Started LSB: Start Jenkins at boot time.
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ ]
```

## Push Your Third Commit

1. Make some changes in README.md to commit, push and test if the Pipeline gets triggered automatically or not.

```
vim README.md
```

2. Add, commit and push the file.

```
git status
```

```
git diff README.md
```

```
git add README.md
```

```
git commit -m "Modified README.md to trigger the Jenkins job after restarting the Jenkins service"
```

```
git push
```



```
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

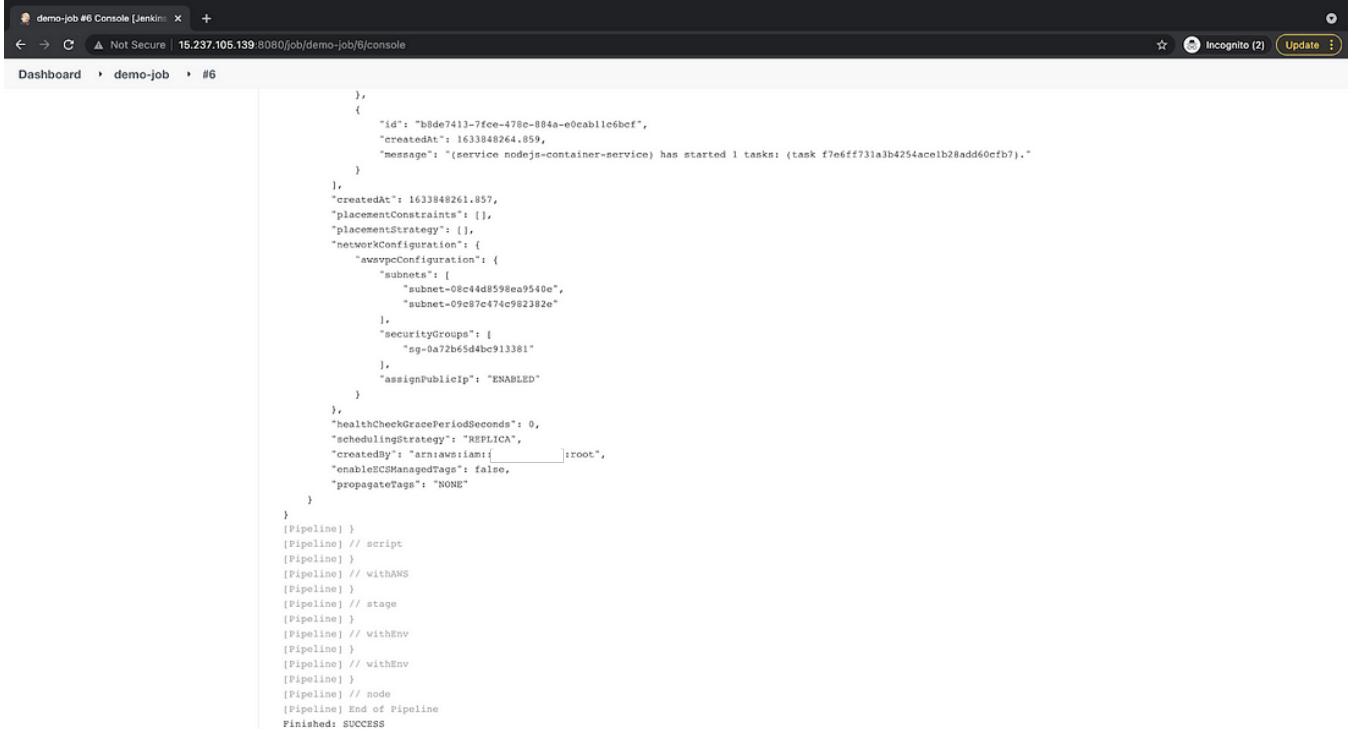
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git diff README.md

@@ -36,3 +36,5 @@ Application will be deployed in AWS ECS.

## Notes
+
+1. Do not forget to restart Jenkins after you add jenkins user to docker group
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git add README.md
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git commit -m "Modified README.md to trigger the Jenkins job after restarting the Jenkins service"
[master 598b756] Modified README.md to trigger the Jenkins job after restarting the Jenkins service
 1 file changed, 2 insertions(+)
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git push
Username for 'https://github.com': [REDACTED].com
Password for 'https://[REDACTED]@github.com':
Counting objects: 3, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 408 bytes | 408.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/shivalkarraahul/demo-nodejs-app.git
 45c5b72..598b756 master -> master
ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ ]
```

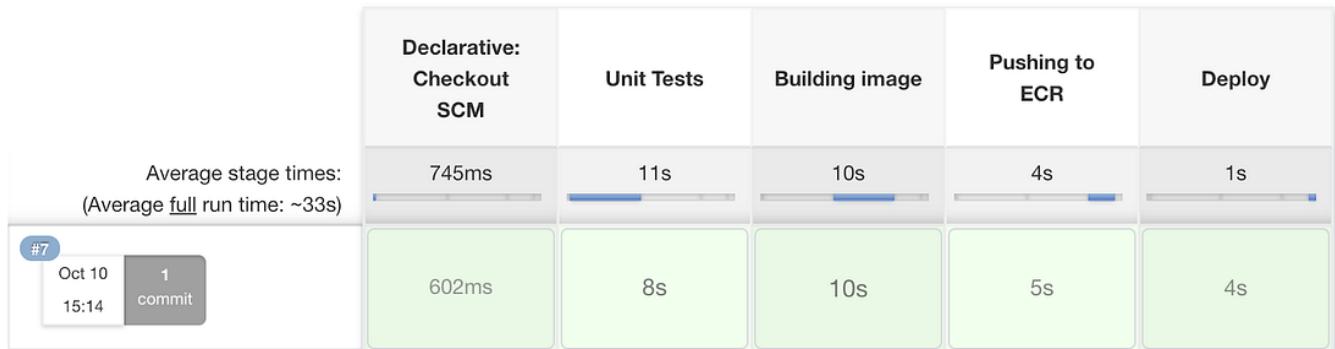
3. This time you can observe that the job must have got triggered automatically. ↗ to the Jenkins job and verify the same.

```
{
  "id": "b0de7413-7fce-478c-804a-e0cab1c6bcf",
  "createdAt": 1633848264.859,
  "message": "(service nodejs-container-service) has started 1 tasks: (task f7eff731a3b4254acelb28add60cfb?)."
},
{
  "createdAt": 1633848261.857,
  "placementConstraints": {},
  "placementStrategy": {},
  "networkConfiguration": {
    "awsvpcConfiguration": {
      "subnets": [
        "subnet-08c44d8598ea9540e",
        "subnet-09e87c474c982382e"
      ],
      "securityGroups": [
        "sg-0a72b65d4bc913381"
      ],
      "assignPublicIp": "ENABLED"
    }
  },
  "healthCheckGracePeriodSeconds": 0,
  "schedulingStrategy": "REPLICAS",
  "createdBy": "arn:aws:iam:[REDACTED]:root",
  "enableCSEmanagedTags": false,
  "propagateTags": "NONE"
}
}
[Pipeline]
[Pipeline] // script
[Pipeline]
[Pipeline] // withAWS
[Pipeline]
[Pipeline] // stage
[Pipeline]
[Pipeline] // withEnv
[Pipeline]
[Pipeline] // withEnv
[Pipeline]
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

4. This is what the Stage View looks like. It shows us the stages that we have specified in our Jenkinsfile.

## Stage View



## Check Status of the Task in the ECS Cluster

1. Go to the Cluster, click on the “Tasks” tab, and then open the running “Task.”

Cluster ARN: arn:aws:ecs:eu-west-3:i:cluster/default

Status: ACTIVE

Registered container instances: 0

Pending tasks count: 0 Fargate, 0 EC2, 0 External

Running tasks count: 1 Fargate, 0 EC2, 0 External

Active service count: 1 Fargate, 0 EC2, 0 External

Draining service count: 0 Fargate, 0 EC2, 0 External

Services	Tasks	ECS Instances	Metrics	Scheduled Tasks	Tags	Capacity Providers			
<a href="#">Run new Task</a>	<a href="#">Stop</a>	<a href="#">Stop All</a>	<a href="#">Actions</a>	Last updated on October 10, 2021 3:07:49 PM (0m ago)					
Desired task status: <a href="#">Running</a> Stopped									
<a href="#">Filter in this page</a> <a href="#">Launch type</a> ALL									
Task	Task definition	Container insta...	Last status	Desired status ...	Started at	Started By	Group	Launch type	Platform versio...
9e646ce1557c45...	first-run-task-definition:24	--	RUNNING	RUNNING	2021-10-10 15:0...	ecs-svc/240421...	service:nodejs-c...	FARGATE	1.4.0

2. Click on the “JSON” tab and verify the image, the image tag should match with the Jenkins Build number. In this case, it is “6” and it matches with my Jenkins Job Build number.

The screenshot shows the AWS ECS Task Definitions console. On the left, there's a sidebar with options like 'New ECS Experience', 'Amazon ECS Clusters', 'Task Definitions' (which is selected and highlighted in orange), 'Account Settings', 'Amazon ECR Repositories', 'AWS Marketplace Discover software', and 'Subscriptions'. The main area is titled 'Task Definition: first-run-task-definition:24'. It contains a message: 'View detailed information for your task definition. To modify the task definition, you need to create a new revision and then make the required changes to the task definition'. Below this are two tabs: 'Create new revision' (blue) and 'Actions' (grey). Underneath are three sub-tabs: 'Builder' (selected and highlighted in orange), 'JSON' (grey), and 'Tags' (grey). The 'JSON' tab displays the following JSON configuration:

```

{
  "image": "dkr.ecr.eu-west-3.amazonaws.com/demo:6",
  "startTimeout": null,
  "firelensConfiguration": null,
  "dependsOn": null,
  "disableNetworking": null,
  "interactive": null,
  "healthCheck": null,
  "essential": true,
  "links": null,
  "hostname": null,
  "extraHosts": null,
  "pseudoTerminal": null,
  "user": null,
  "readonlyRootFilesystem": null
}

```

3. Hit the ELB URL to check if the Nodejs application is available or not. You should get the message as follows in the browser after hitting the ELB URL.



## Push Your Fourth Commit

1. Open the “`rc/server.js`” file and make some changes in the display message to test the CI/CD Pipeline again.

```
vim src/server.js
```

2. Check the files that have been changed. In this case, only one file can be seen as changed.

```
git status
```

3. Check the difference that your change has caused in the file.

```
git diff src/server.js
```

4. Add the file that you changed to the git staging area.

```
git add src/server.js
```

5. Check the status of the local repository.

```
git status
```

6. Add a message to the commit.

```
git commit -m "Updated welcome message"
```

## 7. Push your change to the remote repository.

```
git push
```

```
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ vim src/server.js
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   src/server.js

no changes added to commit (use "git add" and/or "git commit -a")
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git diff src/server.js

@@ -3,7 +3,7 @@ var http = require('http');
 var server = http.createServer(function(request, response) {
   response.writeHead(200, {"Content-Type": "text/plain"});
-   response.end("Hello, Welcome to DevOps CI CD");
+   response.end("Hello, Welcome to DevOps CI CD. Tried to rebuild and deploy using CI/CD");
 });

[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git add src/server.js
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   src/server.js

[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git commit -m "Updated welcome message"
[master fff7ed4] Updated welcome message
 1 file changed, 1 insertion(+), 1 deletion(-)
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ git push
Username for 'https://github.com': [REDACTED].com
Password for 'https://[REDACTED].com@github.com':
Counting objects: 4, done.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 416 bytes | 416.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/shivalkarrahal/demo-nodejs-app.git
 598b756..fff7ed4 master -> master
[ubuntu@ip-172-31-24-191:~/demo-nodejs-app$ ]
```



8. Go to the Task, this time you will see two tasks running. One with the older revision and one with the newer revision. You see two tasks because of the rolling-update deployment strategy configured by default in the cluster.

Details	Tasks	Events	Auto Scaling	Deployments	Metrics	Tags	Last updated on October 10, 2021 3:16:11 PM (0m ago)		
Task status: <span style="color: green;">Running</span> Stopped									
<input type="button" value="Filter in this page"/>									
Task	Task Definition	Last status	Desired status	Group	Launch type	Platform version	< 1-2 >	Page size	50
9e646ce1557c454595ee874...	first-run-task-definition:24	RUNNING	RUNNING	service:nodejs-container-ser...	FARGATE	1.4.0			
4b1e3a297c6d4c3aa5239b0...	first-run-task-definition:25	RUNNING	RUNNING	service:nodejs-container-ser...	FARGATE	1.4.0			

Note: Your revision numbers may differ.

9. Wait around 2–3 minutes, and you should only have one task running with the latest revision.

10. Again, hit the ELB URL and you should see your changes. In this case, we changed the display message.



A newsletter covering the best programming articles published across Medium [Take a look.](#)

**Congratulations!** You have a working Jenkins CI/CD Pipeline to deploy your Node.js application on AWS ECS whenever there is a change in your source code.



[Get this newsletter](#)

## Cleanup the Resources We Created

If you were just trying out to set up a CI/CD pipeline to get familiar with it or for POC purposes in your organization and no longer need it, it is always better to delete the resources you created while carrying out the POC. As part of this CI/CD pipeline, we created a few resources.

We created the below list to help you delete them:

[Get the Medium app](#)

1. Delete the GitHub Repository



Download on the  
App Store



GET IT ON  
Google Play

2. Delete the GitHub Token

3. Delete the IAM User

4. Delete the EC2 Instance

5. Delete the ECR Repository

6. Delete the ECS Cluster

7. Deregister the Task Definition

## Summary

And finally, here is the summary of what you have to do to set up a CI/CD Docker pipeline to deploy a sample Nodejs application on AWS ECS using Jenkins.

1. Clone the existing sample GitHub Repository
2. Create a new GitHub Repository and copy the code from the sample repository in it
3. Create a GitHub Token
4. Create an IAM User
5. Create an ECR Repository
6. Create an ECS Cluster 
7. Create an EC2 Instance for setting up the Jenkins Server
8. Install Java, JSON processor jq, Nodejs, and NPM on the EC2 Instance
9. Install Jenkins on the EC2 Instance
10. Install Docker on the EC2 Instance
11. Install Plugins
12. Create Credentials in Jenkins
13. Create a Jenkins Job
14. Integrate GitHub and Jenkins
15. Check the deployment
16. Cleanup the resources

## Conclusion

A CI/CD Pipeline serves as a way of automating your software applications' builds, tests, and deployments. It is the backbone of any organization with a DevOps

culture. It has numerous benefits for software development and it boosts your business greatly.

In this article, we demonstrated the steps to create a Jenkins CI/CD Docker Pipeline to deploy a sample Nodejs containerized application on AWS ECS. We saw how GitHub Webhooks can be used to trigger the Jenkins pipeline on every push to the repository which in turn deploys the latest docker image to AWS ECS.

CI/CD pipelines with Docker are best for your organization to improve code quality and deliver software releases quickly without any human errors.

We hope this article helped you learn more about the integral parts of the CI/CD Docker Pipeline.



*Original post: [https://www.clickittech.com/devops/ci-cd-docker/?utm\\_source=ci+cd+docker&utm\\_id=Blog+medium](https://www.clickittech.com/devops/ci-cd-docker/?utm_source=ci+cd+docker&utm_id=Blog+medium)*