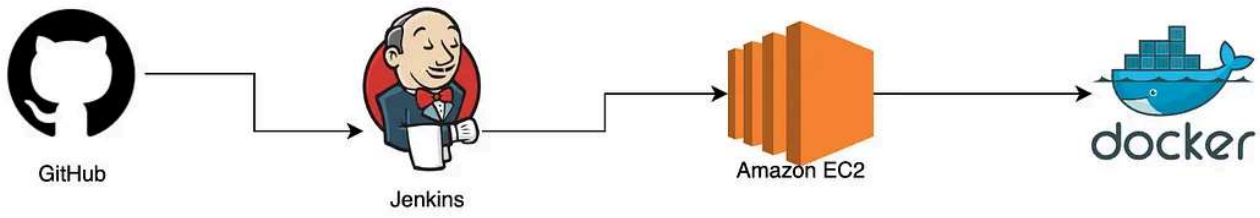
Creating a CI/CD Pipeline using Jenkins, Amazon EC2, GitHub and Docker — Part 2

Swetha Voora · Following
11 min read · Feb 23, 2024



CI/CD pipeline

You can find Part 1 here.

Now that you have verified that pushes to your repo's main branch trigger a build in the Jenkins Controller EC2 instance. Lets continue with the below steps for the further enhancing our CI/CD pipeline.

Configuring a Slave Node in Jenkins

Leveraging a Jenkins Slave node for building jobs is a strategic approach to distribute workload and optimize the utilization of resources within your CICD

pipeline. This section guides you through setting up an EC2 instance as a Jenkins Slave node.

Setting Up the Jenkins Slave Node

- Install Java openjdk version 11 on the second ec2 instance(Jenkins Slave Instance): run the commands

```
sudo apt-get update
sudo apt-get install openjdk-11-jdk
```

- Access Jenkins Controller Homepage: Navigate to http://<ec2-Ipv4-DNS>:8080 address of your Jenkins Controller on your web browser to access your Jenkins dashboard.
- Navigate to Nodes Configuration:Click on “Manage Jenkins” in the main menu. Select “Nodes”.
- Add a New Node:Click on “New Node”. Name the node, for example, “Slave-node-1”. Select “Permanent Agent” and proceed with “OK”.

Configure Node Details:

- Executors: Set the number of executors based on your instance’s vCPUs and expected workload. It’s advisable to have less than or equal executors to the number of vCPUs on your Jenkins Slave instance.
- Remote Root Directory: Connect to your Jenkins slave EC2 instance using SSH/EC2 Instance connect and run “cd ~” followed by “pwd” to print the root directory. Copy this path into the “Remote root directory” field.
- Labels: Assign labels to your node to control job execution. Labels help in job segregation and routing.(example : you can use “dev” for your Slave-node-1)
- Usage: Opt for “Only build jobs with label expressions matching this node” to ensure specific job execution.
- Launch Method: Select “Launch agent via SSH”.
- Host: On your second EC2 instance(Jenkins slave), use the below command to find the IP address of your slave instance and Enter this IP in the “Host” field.

```
hostname -i
```

- Credentials: Generate an SSH key pair on your Jenkins Slave EC2 instance by running the below command

```
ssh-keygen
```

- Navigate to the .ssh directory and use cat to display your public key and private key in your terminal.(Similar to jenkins job creation's Source Code Management (SCM) Section)
- Copy the public key and Enter the below command to list the keys and files in the .ssh folder

```
ll
```

- Run the below command to edit authorized_keys file

```
vim authorized_keys
```

- Press i on the keyboard to be able to insert into the file.
- Paste the public key at the end of the authorized_keys file to authorize access from Jenkins Controller.
- Press 'esc' key and type :wq and enter, You will come back to your terminal.
- Private Key: Copy the private key from your Jenkins Slave ec2 instance, generated earlier. In Jenkins, under the node configuration's "Credentials", choose "Add". Select "SSH Username with private key", input the Jenkins Slave EC2 instance's username(Example : root), and paste the private key.

- Host Key Verification Strategy: Choose “Non verifying verification strategy” for simplicity and to avoid host key verification issues.

Finalizing Node Setup:

After configuring the credentials and specifying the host details, save your node’s configuration.

By completing the above steps, you’ve successfully set up an additional node in Jenkins, designated for executing build jobs. This setup not only enhances the efficiency of your CICD pipeline by offloading build tasks from the Jenkins master (Controller) but also introduces flexibility in managing different types of jobs based on the labels assigned to them.

This slave node setup is part of an overarching strategy to optimize resource utilization, improve build times, and maintain a clean separation of duties within your Jenkins environment. As you progress with your CICD pipeline development, you can further customize and scale your Jenkins infrastructure by adding more slave nodes as required.

Additional Configuration in the Job, for Optimal Resource Management

After setting up your Jenkins Slave node, it’s crucial to revisit your job(Freestyle project job) configuration to ensure it runs on the designated slave node and to manage build artifacts efficiently. Here’s how to adjust your job settings for better resource management and to direct the job execution to the slave node.

Configuring Job to Use the Slave Node

1. Access Your Job Configuration: Navigate back to the Jenkins homepage at http://<ec2-Ipv4-DNS>:8080. Find the job you created for your CICD pipeline and click on its name. Click “Configure” to edit the job settings.

2. Discard Old Builds: Within the job configuration, locate the “General” section. Check the option for “Discard old builds” to enable automatic cleanup of build records. Under “Strategy”, select “Log Rotation”. Set “Max # of builds to keep” to 1. This setting helps in conserving storage space by retaining only the most recent build.

3. Restrict Job to Specific Node: Scroll down to check the option “Restrict where this project can run”. In the label expression field that appears, enter the label

you assigned to your slave node during its configuration (e.g., the label “dev” you used for “Slave-node-1”). Upon entering the label, Jenkins will indicate if the label correctly matches any of the available nodes with a note such as “Label is tied to 1 node” or “Label matches # of nodes”, ensuring that the job will only run on the specified node(s).

Finalizing the Job Configuration

After making these adjustments, your job is now optimized to automatically manage build artifacts and specifically target the configured slave node for execution. This ensures that your builds are processed on the appropriate infrastructure, leveraging the distributed nature of Jenkins to enhance performance and manage resources effectively.

By directing the job to run on a slave node, you alleviate the workload on the Jenkins Controller, allowing it to manage job scheduling and orchestration more efficiently. Additionally, by keeping only the most recent build, you maintain a lean environment that is easier to manage and quicker to troubleshoot.

Remember to click “Save” at the bottom of the job configuration page to apply these changes.

Checking the Repo Content Cloned into the Slave-Node EC2 Instance

Once your Jenkins CICD pipeline triggers a build on the slave node (Slave-node-1) due to a push to the main branch of your GitHub repository, the repository’s content will be cloned into the /root/workspace folder of your Jenkins slave EC2 instance. To facilitate the execution of your project, especially if it requires Node.js, you may need to install Node Version Manager (NVM) and Node.js. Here’s how to manage this setup and verify that your project has been correctly cloned and set up.

Installing NVM and Node.js

- Install NVM:On your slave EC2 instance, install NVM by executing the following command:

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

- After installation, you might need to close and reopen your terminal to start using NVM. This step ensures that the NVM commands are loaded into your shell session.
- Install Node.js: Install the latest version of Node.js using NVM by running:

```
nvm install node
```

- The command node is an alias for the latest version, ensuring you have the most current Node.js features and security updates.
- Install NPM: Node.js comes with NPM (Node Package Manager), but if required, you can update NPM to its latest version with:

```
sudo apt install npm
```

Preparing and Running Your Project

Assuming your GitHub repository includes a project utilizing Vite/React for the frontend, follow these steps to configure and run your application:

- Make sure to add port 3000 in your Jenkins slave EC2 instance's security group.

NOTE: (For a React application, you can use package.json to modify the host and port for the application. It should be something like this:)

```
"scripts": {
    "start": "HOST=0.0.0.0 PORT=3000 react-scripts start"
}
```

- Configure Vite for Development: Ensure your vite.config.js includes server configuration to listen on all network interfaces (0.0.0.0) and on a specified port (e.g., 3000). This configuration is crucial for accessing the application from outside the EC2 instance. Example vite.config.js setup:

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';
export default defineConfig({
    server: {
        host: '0.0.0.0',
        port: 3000 // Specify your desired port here.       },
    plugins: [react()],
});
```

- Install Project Dependencies: Change directory to your project's frontend folder in the Jenkins Slave EC2 instance(e.g., /path/from/root/to/frontend) and run the below command to fetch and install necessary dependencies.

```
npm install
```

- Verify Installation: Check if the node_modules directory is successfully created within your frontend folder, indicating that dependencies have been installed correctly.
- Run Your Application: Start your application by executing this command in your terminal.

```
npm run dev
```

- This command will launch your application in development mode, making it accessible via the Jenkins Slave EC2 instance's public DNS or IP address followed by the specified port (e.g., http://<EC2-Public-IPv4-DNS>:3000).

By following similar steps you can run your backend too. Once these steps are done, you've ensured that your project is correctly cloned and set up on the Jenkins Slave node. Additionally, you've prepared your development environment to run a full-stack application using Node.js and Vite, making your project accessible for testing and further development.

Configuring Docker for Deployment

Integrating Docker into your development and deployment workflow offers numerous benefits, including consistent environments, scalability, and isolation for your applications. By containerizing your frontend/backend, you ensure that it can be deployed consistently across any environment. Here's how to set up Docker for your frontend application.

Installing Docker

- Install Docker: Open your terminal on the Jenkins Slave EC2 Instance (or whichever instance you intend to deploy your application) and execute the following command to install Docker:

```
sudo apt install docker.io
```

- Ensure Docker is installed and running by checking its status or version with

```
sudo systemctl status docker or docker --version.
```

Preparing the Dockerfile

If you don't already have a Dockerfile in your repository's frontend/backend folder, follow these steps to create one:

NOTE: Ensure that you are adding this Dockerfile to your frontend/backend repos

Go to your Jenkins Slave EC2 instance.(Below is an Example for creating DockerFile for the frontend)

- Navigate to Your Frontend Directory: Change directory to your frontend folder where you want to create the Dockerfile. Run this command to create and open a new Dockerfile for editing.

```
sudo vim Dockerfile
```

Writing the Dockerfile:

- Press i on your keyboard to enter insert mode in Vim.
- Define the base image with the Node.js version you're using, for example, FROM node:21.6.2-alpine.
- Set the working directory inside the Docker container to match the path of your frontend directory using WORKDIR /path/to/frontend/directory.
- Copy your project files into the Docker image with COPY . .
- Install project dependencies using RUN npm install.
- Expose the port your application runs on, which is 3000, with EXPOSE 3000.
- Define the command to run your application using CMD ["npm","run","dev"].
- Example Dockerfile:

```
FROM node:21.6.2-alpine
WORKDIR /path/to/frontend/directory
COPY . .
RUN npm install
EXPOSE 3000
CMD ["npm","run","dev"]
```

Updating Github Repo Structure: Make sure to Include this Dockerfile inside the frontend folder in your repo.

Building and Running the Docker Container

*****NOTE : If you are not the root user. Please add sudo before the below docker commands, when running them in your Jenkins Slave EC2 instance*****

- Build Your Docker Image: After saving your Dockerfile, make sure you are inside your frontend/backend folder and build your Docker image in your Jenkins Slave instance by running the below command in the terminal:

```
docker build . -t your-desired-image-name
```

- This command creates a Docker image with the name you mentioned in the place of your-desired-image-name, based on the instructions in your Dockerfile.
- Run Your Docker Container: To run your Docker container, use:

```
docker run -d --name your-desired-container-name -p 3000:3000 your-image-name
```

- Replace your-desired-container-name with a name for your container. This command runs the container in detached mode, maps port 3000 inside the container to port 3000 on your host, and uses the image named your-image-name.

Verifying Deployment

- After your container is up and running, using the public IP address of your Jenkins Slaves EC2 instance, visit http://<EC2-Public-IPv4-DNS>:3000 in your web browser. You should now see your application running, served directly from the Docker container instead of your git clone. This verifies that the Docker containerization and deployment process is working correctly.

By following these steps, you've successfully integrated Docker into your project's deployment process, allowing for consistent and scalable deployments. In the next steps, you'll learn how to automate this process using Jenkins to build and deploy your Docker container upon code changes.

Continuous Deployment using Docker and Jenkins

To achieve Continuous Deployment (CD) using Docker and Jenkins, you will configure your Jenkins job to automatically build when commits are pushed to your main branch in GitHub, stop and remove an existing Docker container, and then run a new one with the latest code from your repository. This process ensures that your application is always up-to-date with the latest changes. Follow these steps to configure the CD process in your Jenkins job:

Configuring Build Actions for Docker Deployment

- Access Your Job Configuration: Go back to the Jenkins Controller instance's Jenkins dashboard and open the job you've created for your CICD pipeline. Click "Configure" to modify the job settings.
- Add Build Step for Container Management: Scroll down to the "Build Steps" section. Click "Add build step" and select "Execute shell". In the command box that appears, enter the following script to check if a Docker container with your-container-name exists. If it does, the script stops and removes the container to prepare for deploying the latest version:

```
# Check if the container exists
if docker ps -a | grep -q "your-container-name-here"; then
    # Stop the container
    docker stop your-container-name-here

    # Remove the container
    docker rm your-container-name-here
fi
```

Replace "your-container-name-here" with the actual name of your Docker container in quotes.

- Add Build Step for Image Removal: Click "Add build step" again and choose "Execute shell". Enter the following commands to remove the existing Docker image named your-image-name to ensure that you can rebuild the image from the latest code:

```
# Remove the frontend image
docker rmi your-desired-image-name
```

NOTE : Make sure to have a docker image (with the same name as "your-desired-image-name") in your Jenkins Slave instance, before trying to execute these shell commands as part of the automated process. Because, I had issues when there was no image in the EC2 instance, even though i tried to remove it conditionally(i.e., if the image exists)

- Add Build Step for Building and Running the Container: Add another “Execute shell” build step. This time, include the commands to build a new Docker image from your Dockerfile in the new git clone in the Jenkins Slave Instance and run a new container from this image:

```
docker build -f /path/from/root/to/frontend/Dockerfile -t your-image-name-here

docker run -d --name your-container-name-here -p 3000:3000 your-image-name-here
```

- Replace /path/from/root/to/frontend/Dockerfile with the actual path to your Dockerfile, your-image-name-here with the name you wish to give your Docker image (it can be any name you prefer), and your-container-name-here with the name you want for your Docker container.

- Save Your Configuration:After adding these build steps, click “Save” to apply the changes to your job.

Verifying Continuous Deployment

Now, whenever you push changes to your repository’s main branch, Jenkins will trigger this job. The job will build a new Docker image with the latest code and deploy it by running a new container, ensuring your application is always up-to-date. Consequently, your application remains updated without the necessity to manually intervene or access the EC2 instances. Your updated application can always be easily accessed at http://<Jenkins_Slave-Ipv4-DNS>:3000.

This setup forms the core of a Continuous Deployment pipeline using Jenkins and Docker, automating the deployment process and significantly reducing the manual effort required to update your application.

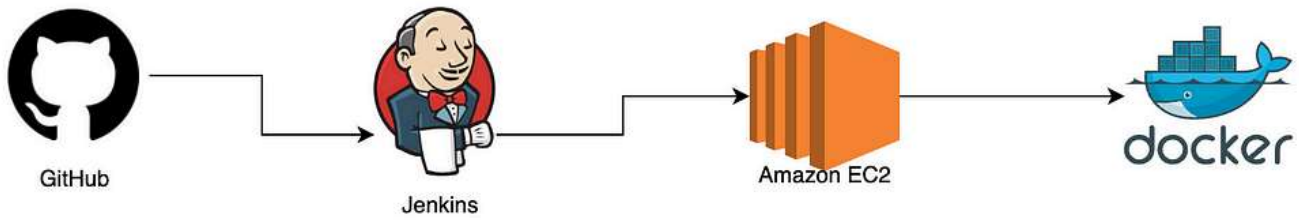
Jenkins | Continuous Integration | Continuous Delivery | Docker | Amazon Ec2

Following

Written by Swetha Voora

2 Followers

More from Swetha Voora



Swetha Voora

Creating a CI/CD Pipeline using Jenkins - Part1

Creating and Connecting to EC2 Instance via SSH/EC2 Instance Connect

9 min read · Feb 22, 2024

1

See all from Swetha Voora