CI/CD Automation Demo

Summary:

You are looking to improve the speed and quality of the software delivered to end users. Automating a CI/CD process for both application code as well as infrastructure management ensure that changes can be rapidly tested and deployed, and bugs or environment drift to be quickly remedied.

Approach:

* + Ansible to provision and install all the required resources and tools for the solution
  + GitHub as a code repository, ensuring robust and flexible change history and secure access to the code.
  + EC2 to host the compute needs, ensure speed/low latency, and scale as needed.
  + Jenkins to orchestrate the software lifecycle, triggered by code commits to build, test, package, deploy, and cleanup artifacts not needed after deployment.
  + Docker to containerize and run the application, ensuring reliability and ease of scalability based on future demand.

**GitHub Repository:** <https://github.com/StumptownRider/BMSDockerMaven.git>

Steps:

I. Initialize GitHub repository <https://github.com/StumptownRider/BMSDockerMaven.git>

i) Generate ssh keypair on local machine, and add it to GitHub ssh keys

* + - 1. ssh-keygen -t ed25519 ~/.ssh/github\_ed25519
      2. Enter a passphrase
      3. Print public key material to screen
      4. cat ~/.ssh/github\_ed25519.pub
      5. From github web access, go to Settings => SSH and GPG Keys => New SSH Key. Add the string form above step and save.

ii) Clone empty repo from github on local machine

* + - 1. git clone [git@github.com:StumptownRider/BMSDockerMaven.git](mailto:git@github.com:StumptownRider/BMSDockerMaven.git)

iii) Write/commit application code (for this example, boilerplate Spring Boot Web template from spring.io)

1. git add .

git commit -m “commited first code”

1. Synchronize changes with your GitHub project

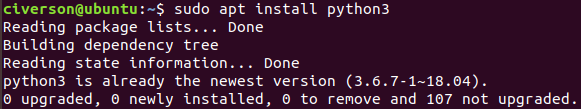
git push

II. Ansible automation setup: Install Ansible and Boto on machine used to administer infra configuration

1. Ensure Python is installed (typically included with full-featured linux distros)

sudo apt-get update

sudo apt install python3



python --version



ii) Ensure PIP is installed (likely included with python package installed above)

Text

Description automatically generated

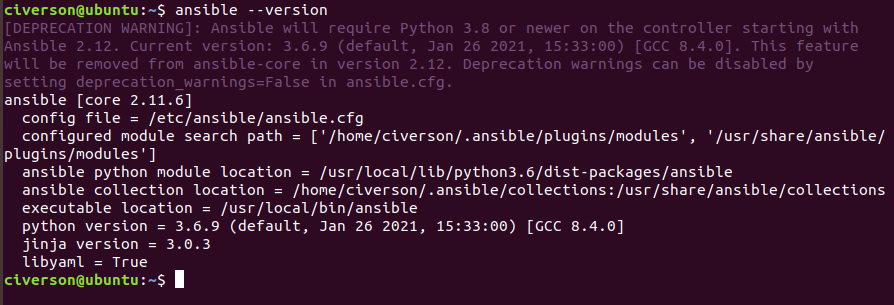
pip3 --version



iii) Install Ansible

sudo -H pip3 install ansible

ansible --version



iv) Install Boto (for Ansible to talk to AWS CLI)

pip3 install Boto3

Text

Description automatically generated

1. Create a file ~/.aws/credentials for AWS access keys on local machine used for administering the AWS provisioning via Ansible, and put the keys there. This will be picked up by boto.

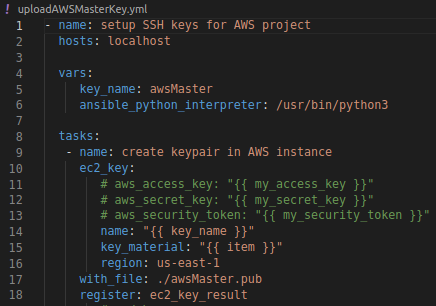
* 1. **Provision infra**

1. **Create playbook to upload public key to AWS (uploadAWSMasterKey.yml)**

ec2\_key module creates a Key Pair named from the key\_name variable with the file from 2b. Above

Presteps:

* ssh-keygen -t rsa -b 4096 -f awsMaster
* In this example, file is created within the local repository space, so it must be added to .gitignore. Better security practice to put in the ~/.ssh/ directory and point to that filepath in the playbook

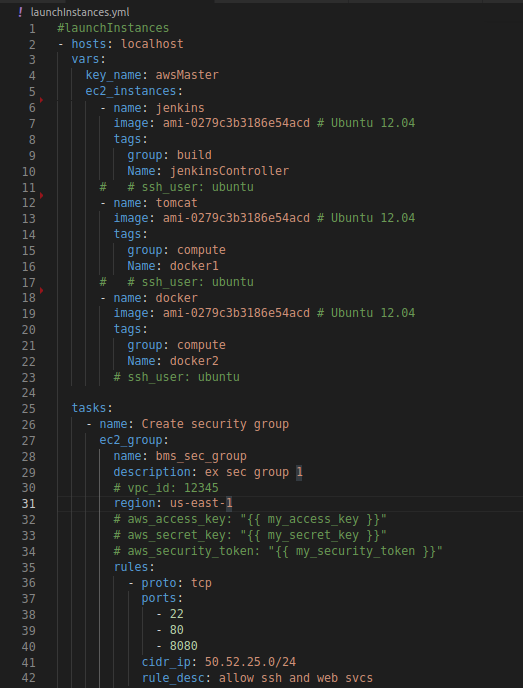


1. **Create playbook to create ec2 instances (launchInstances.yml)**

* **ec2\_instances module: defines the particular configuration for each machine (image/os, and tags for grouping them in dynamic inventory such that ansible can perform configuration across machines with similar functions)**

* ec2\_group module: Creates a security group which for project purposes exposes SSH and web access to my local machine, and more ports among instances in the group private network such that the various services can talk to each other (i.e. Docker and Jenkins agents)

* ec2 module: specifies the key pair, hardware, geographic region, number of instances, and tags which will be applied for each instance in ec2\_instances above.



1. **Create dynamic inventory file (exDynInv\_aws\_ec2.yml)**

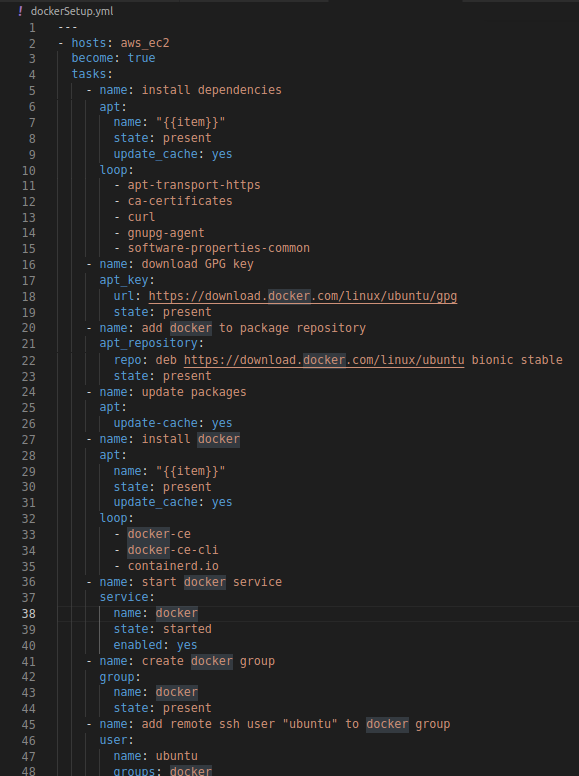
aws\_ec2 plugin pulls the instance fact data to construct an inventory file with. In this way, a set of ephemeral instances can be maintained reliably by the tags or other values we assign in playbooks or roles.

We declare which machines are to be captured, how they will be grouped, and what values to return for the hosts (in this case attribute is public ip defined for each instance)

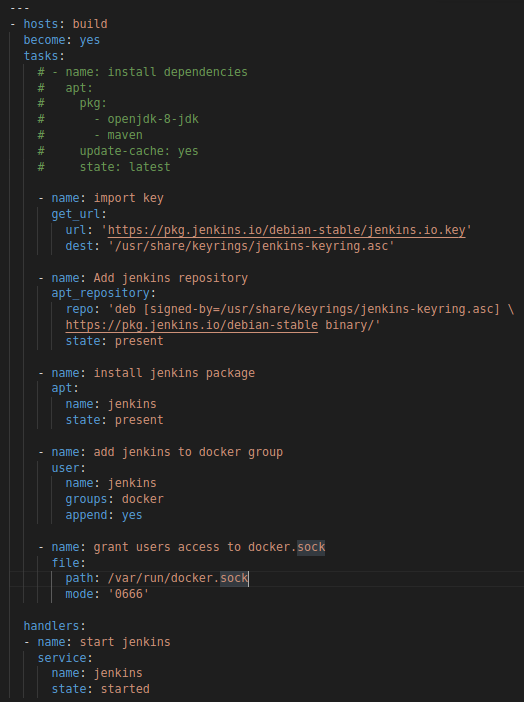
Text

Description automatically generated

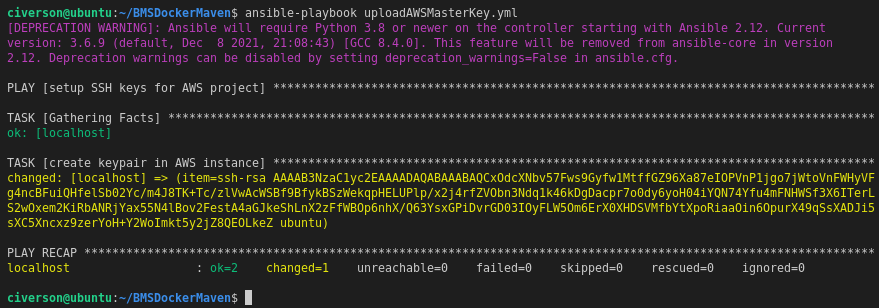
1. Create playbook to install docker and dependencies (dockerSetup.yml)
   * + apt module defines which packges to install (ubuntu/Debian machines)
     + service module is used to ensure that docker is running on the hosts that were just installed
     + group is used to create the docker "user" group
     + user module is used to ensure our default user can execute docker commands without sudo
     + Additional apt command at the end is to ensure some Jenkins supporting dependencies are installed. Initially openjdk-8-jdk was included here for the option of using any of the instances as ephemeral docker Jenkins agents, with Docker-in-Docker Jenkins Container. As approach is now a standard Jenkins installation, this final step could be moved to the jenkinsSetup.yml playbook.

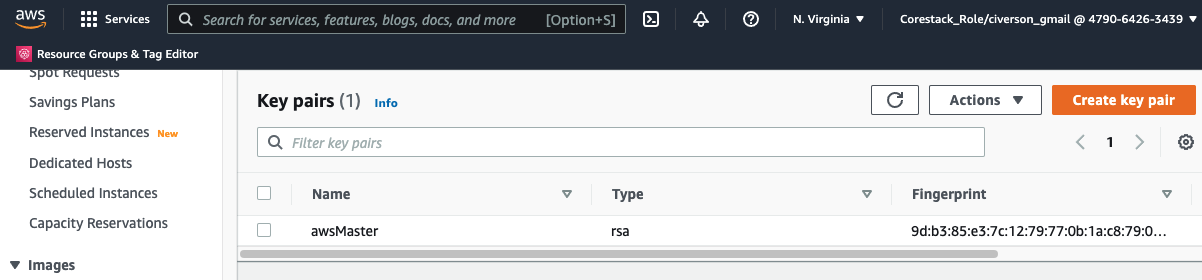


1. Create playbook to install Jenkins and dependancies (jenkinsSetup.yml)
   * + Import the jenkins repository signing key
     + Add Jenkins Repo
     + Install Jenkins Package
     + Ensure Jenkins can execute docker commands
       - Add Jenkins user to docker group
       - Grant users on Jenkins instance to docker.sock
       - Start Jenkins



1. Run playbooks
   * + Upload awsMaster public key as AWS Key Pair record

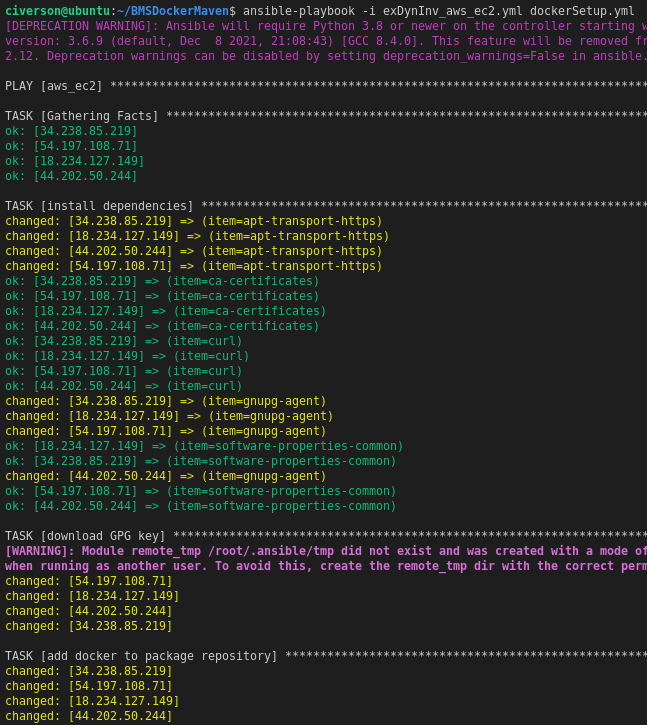




* + - Launch instances for Jenkins, jenkins agent, and Tomcat service ]



* + - Install docker and dependencies, and configure group/user permissions



* + - Install Jenkins on "build" host group

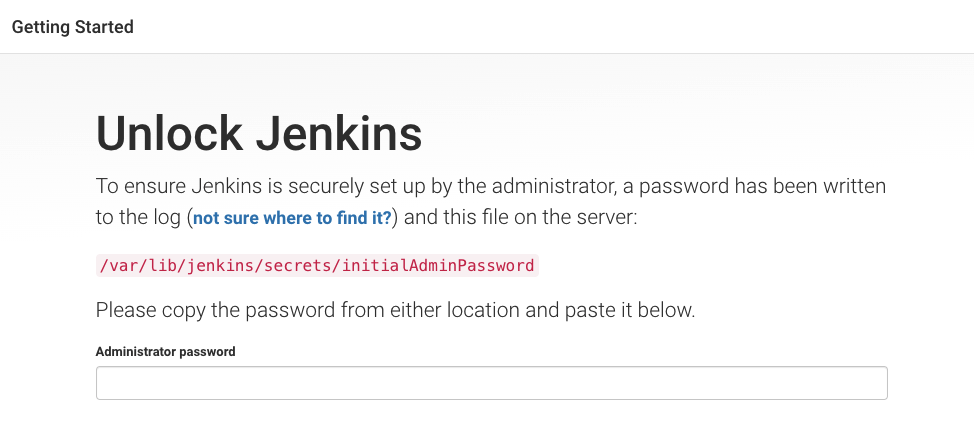
A picture containing text

Description automatically generated

* 1. **Jenkins Setup**

1. Jenkins admin account and install plugins
   1. Retreive initialAdminPassword from jenkins ec2 instance

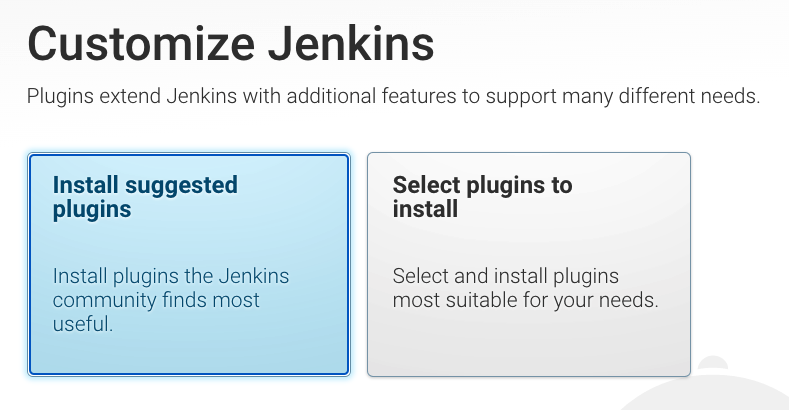
* Navigate to http://<ip-of-jenkins-controller-ec2-instance>:8080 in browser to reach the Jenkins WebUI. Landing page shows where to retrieve the password

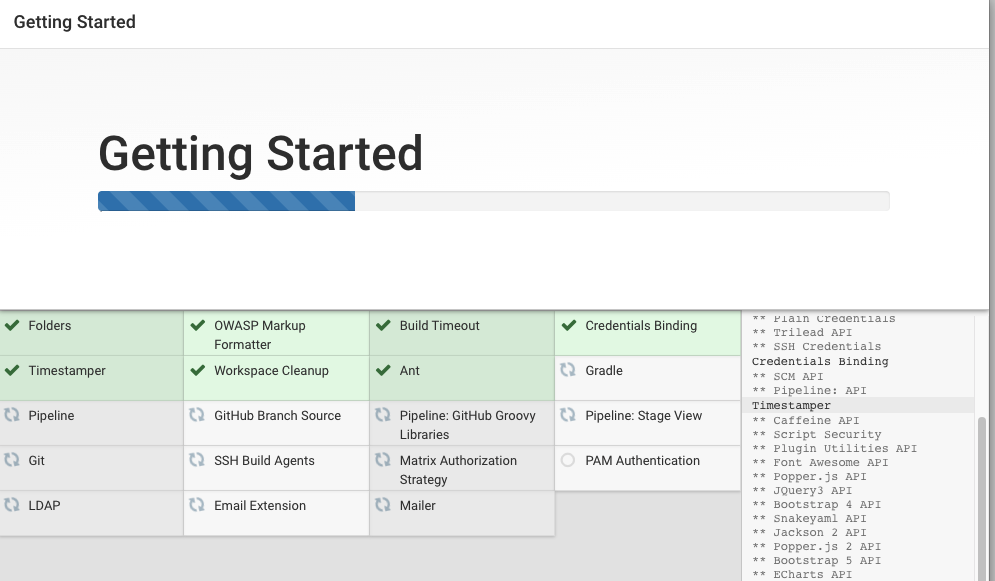


* Go back to terminal and print file from path above to the console, copy/paste into Jenkins form, and submit.

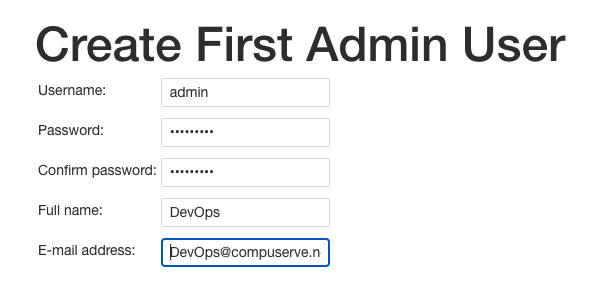


1. Install suggested plugins



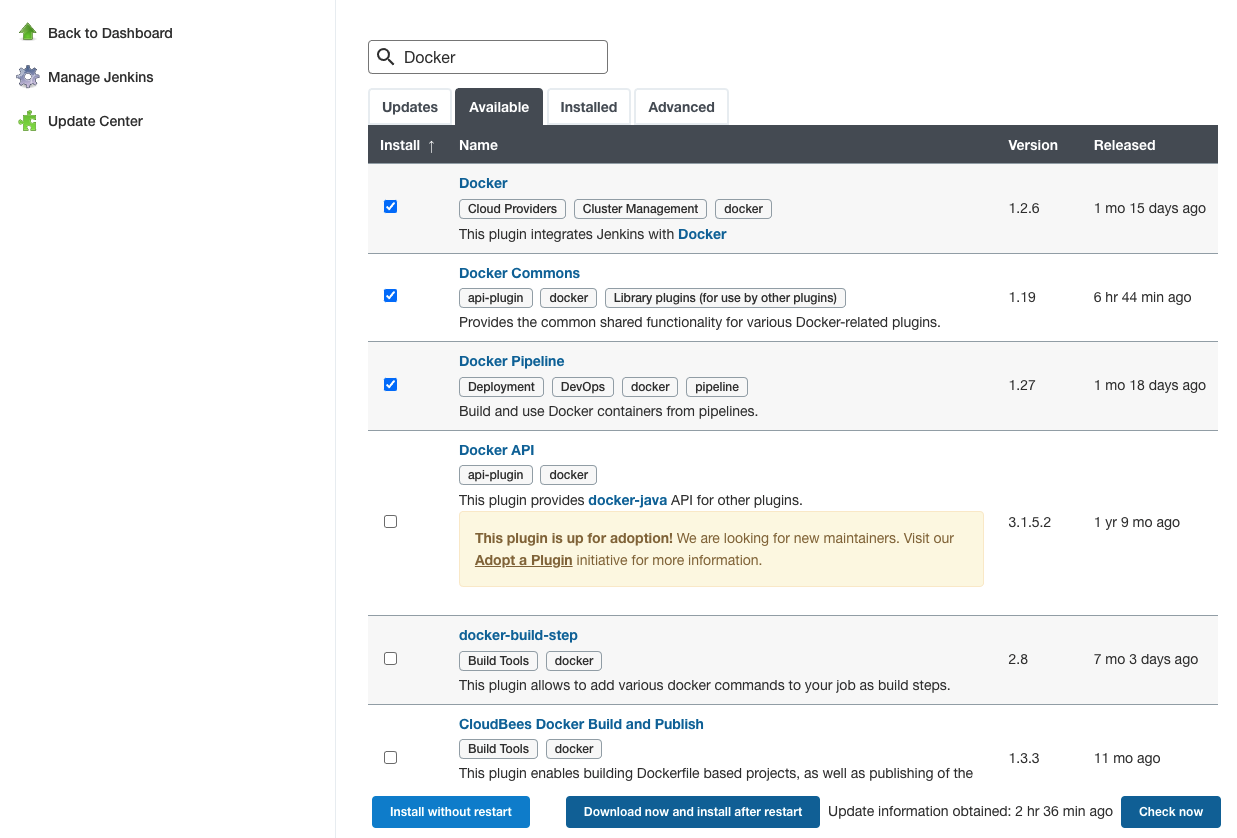


1. Create initial admin account

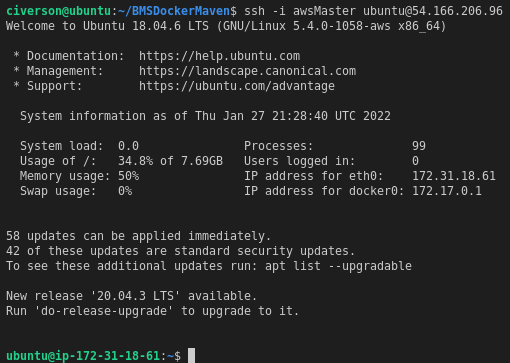


1. Fill out form and submit
   * + Complete following screens and proceed to Jenkins controller console

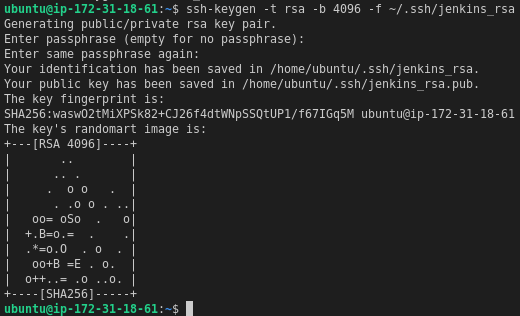
1. Via Jenkins => Manage Jenkins => Plugin Manager, "Install Without Restart" the following additional plugins to be used in the solution:
   * + Docker
     + Docker Pipeline
     + Docker Commons
     + SSH Agent



1. Generate ssh key on Jenkins controller machine and install on worker nodes for jenkins controller to gain access.
   * + Login to jenkins controller instance



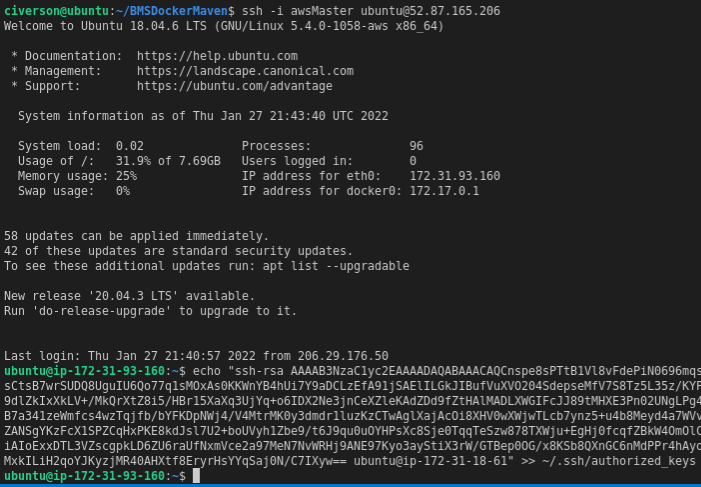
* + Create the new key

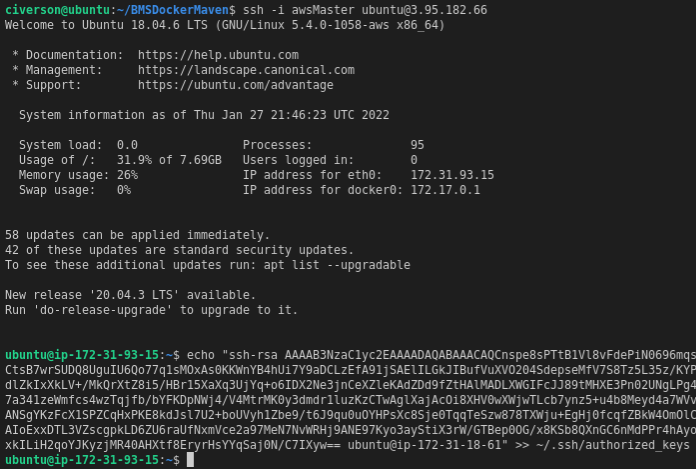


* + Retrieve public key string, and add to authorized\_hosts to remote hosts Jenkins machine will be connecting to.

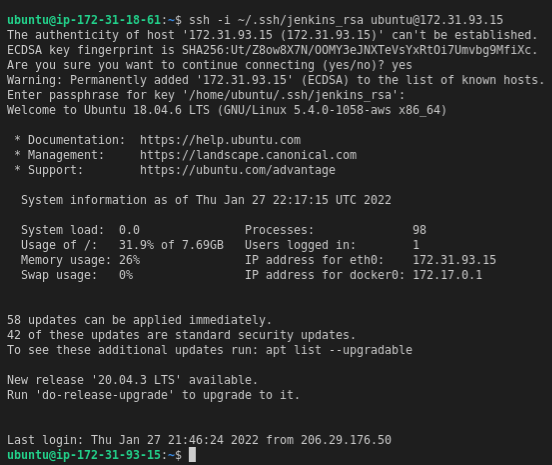
Text

Description automatically generated





* + Verify ssh is authorized from jenkins controller instance to worker instances (docker1, docker2)



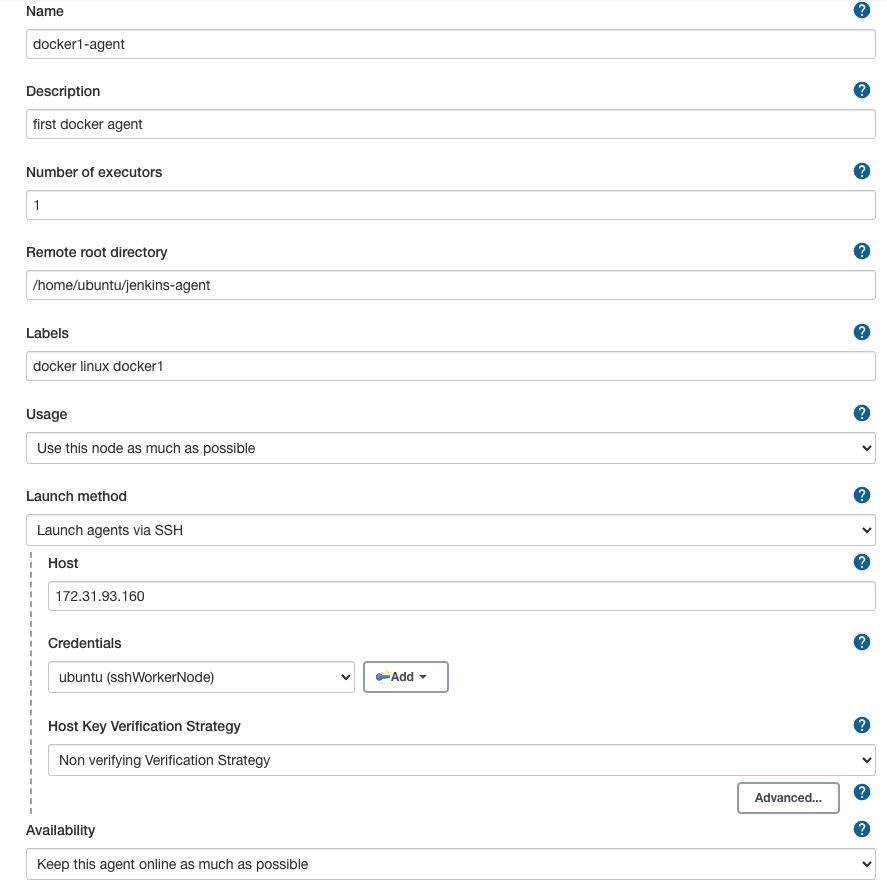
* + Print out jenkins\_rsa private key material and copy it, for adding credential in the next step



1. Configure credentials
   1. Add a jenkins credential for ssh connection to worker nodes
      * 1. Go to Jenkins => Manage Jenkins => Manage Credentials => Store:Jenkins => global credentials => add credentials [kind: ssh username with private key]
        2. Add a descriptive id to reference this in a jenkins pipeline (example: docker-worker-key)

* 1. Add a Jenkins credential for Github
     + 1. Generate personal access token for GitHub account access
       2. From github account => settings => developer => personal access tokens => generate new token
       3. Enter a name (i.e. "jenkins"), then select the permissions to be granted for this token, then click "Generate Token"
       4. Copy the token string returned by github
       5. Go back to jenkins => add credentials [kind: username with password]
       6. Username: <github-user>
       7. Password: paste <tokenValue>
       8. Add a descriptive id to reference this in a jenkins pipeline
  2. Generate a personal access token for DockerHub account access
     + 1. Login to dockerhub; go to Account Settings => Security => New Access Token. Copy returned token value
       2. Follow jenkins steps as in "Add a Jenkins credential for Github" above, then save the credential.

1. Set up build agent on worker instance/node docker1
   1. Go to jenkins => build executor status => newnode
   2. Complete the agent node connection details. Create memorable label values, these are used in a pipeline job to target particular agent(s) for execution of steps. Select ssh credentials created in "Configure credentials" step above



* 1. Verify node is successfully connected in the build executor status, or by checking the console output in the node options.
  2. Create the CI/CD pipeline:

We need to perform the following steps: Build, Test, Package, Install the package into a docker image with tomcat installed, deploy the image to one of the instances running docker, and remove the container stack once the job has completed.

1. Create a Jenkinsfile for defining the stages and steps of the process:
   1. Direct the execution to be performed on the jenkins agent. This ensures Jenkins controller is still performant for orchestrating other jobs and performing configuration tasks.
   2. Populate environment variables to pass credentials for accessing dockerhub
   3. Execute build, test, and package application code with maven
   4. Execute a Dockerfile which downloads a Tomcat base image for hosting the application, and copy the packaged application into the appropriate directory for serving it to users.
   5. Publish the compiled image to dockerhub
   6. Remove the container stack from the agent since it's no longer needed
   7. Via ssh, connect to the remaining instance "docker2" passing the ssh credentials from step 4c (generate ssh key for jenkins to access worker instances)
   8. Login to dockerhub
   9. Run the image to download it, and deploy the container to start up the Tomcat server. Ensure the port listening on the container is mapped to a logical port on the host machine (docker2). Tomcat is listening on :8080 by default.



1. Create a Dockerfile to install and configure the tomcat image which will run in a container
   * + Declare the Tomcat base image to be used
     + Ensure execution permission to the directory where Tomcat startup scripts are located
     + Copy the application package to the directory to the Tomcat webapps directory
     + Set conservative memory heap configuration values to ensure adequate memory for the instance kernel and docker service, as ec2 instance for this demonstration is small.
     + Execute the Tomcat startup script when the container is launched

Text

Description automatically generated

1. Save Jenkinsfile, Dockerfile to local repository, commit changes, and push to github target from the first step.

1. Configure the pipeline job
   * + Select Jenkins => New Item => Pipeline job. Give a name (i.e. BMSDockerMaven) and click OK.
     + Check "Poll SCM" under build triggers to ensure the pipeline is automatically run when new changes are submitted. Indicate an useful polling interval that doesn't impact performance (ex. 5 minutes), depending on how many other jobs may be polling from the Jenkins controller.
     + Navigate to Pipeline => Definition and select pipeline script from SCM
       - For SCM, select "GIT"
       - Paste the repository URL from step 1.
       - Select the credentials "github-personal" from step 4d.
       - Ensure the Branch Specifier matches the "release" or "production" branch. Github changed from "master" to "main" indicating the primary branch.
       - Save changes

Graphical user interface, text, application, email

Description automatically generated

1. Perform the build job
   * + Monitor the stages for any errors

Table

Description automatically generated

* + - When all stages are green, go to the URL of the tomcat server and app name

<example: <http://52.201.235.10:8080/BookMyShow/>>

1. Index page is rendered, deployment is succesful!

Graphical user interface, text, application, email

Description automatically generated

**Takeaways**

Ansible playbooks with Dynamic Inventory allow for quick change management of ephemeral cloud infrastructures.

* + - Public cloud providers are very fast, so provisioning goes quickly
    - Patterns/steps can be applied to both vertical and horizontal scaling. Just add instances, compute power, or additional regions to the playbook. Additional nodes/instances can picked up by the dynamic inventory for a wide range of host groups/functions.
    - With source control, infra configuration can quickly be tested and/or rolled back if needed.
    - Cost savings due to not adding resources until they are needed, and automatically maintaining environment state
    - Time savings due to the playbooks being documentation of
  1. CI/CD with Jenkins
     + Trigger a complete delivery/deployment automatically when changes are committed; when tests are automated, ensure both speed and reliability.
     + Additional tasks can be triggered on failures to ensure rapid feedback to the desired actor to address an issue.
     + Pipelines and build history logs make it much faster to compare differences and quickly identify what needs to be remedied.
  2. Docker
     + Applications become portable to any other machine with a docker host
     + Reduce conflicts when many containers running components of an application may function well with different tools/environments
     + Version your entire execution environment and customize as needed with Dockerfile and docker images

**Enhancement Recommendations**

* 1. Configure Jenkins agent as a k8s cluster with autoscaling, utilizing Docker images/containers for the execution environment.
     + Ensures builds will be performed very fast, and incur minimal cost when builds load is low
     + Easily customize images for quickly testing many different combinations of tools and versions compatibility
  2. Run ansible on the Jenkins controller
     + Build jobs can more easily be organized to reuse a mix of infra as code playbooks and roles, and automatically execute when changes are committed (example security updates/patches). Using roles, they can be rolled out in phases or groups to minimize/isolate impact and nearly eliminate downtime.