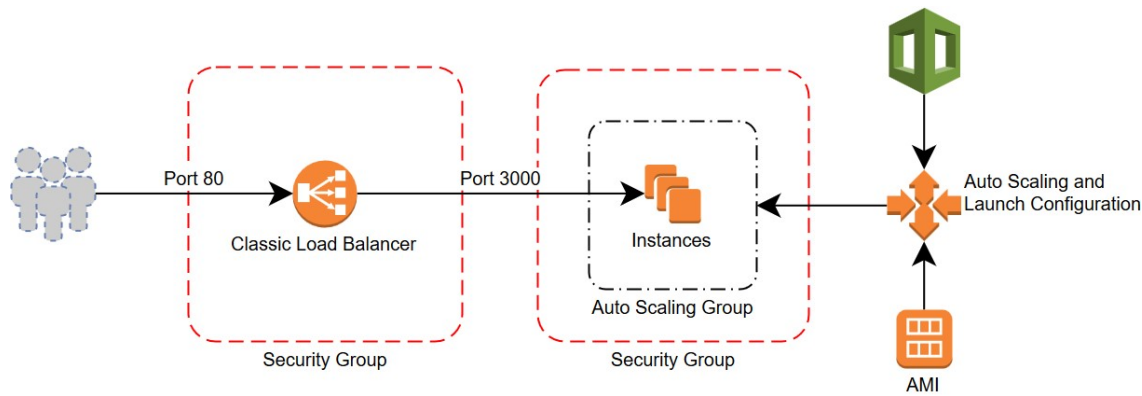


Using Jenkins, an open source automation server, application code can be committed to a git repository and then a deployed to AWS using Hashicorp's Packer and Terraform. Since Terraform creates a CloudFormation stack in AWS, additional commits to modify the application will trigger rolling updates and automatically update the instances in the stack.



## Configure Jenkins

This guide uses the docker container for Jenkins' [Blue Ocean](#), specifically designed as a simplified GUI for the Jenkins Pipeline.

### Installation

[Jenkins](#) can be run as a remote server, or in this case, locally using [docker](#).

#### Start the container

The following command runs the Jenkins server in the background with the name `docker-jenkins` for easy access. On the first run, a docker [volume](#) called `jenkins-data` is created and mounted as the Jenkins home directory.

##### Linux

```
docker run \
  -d \
  --name docker-jenkins \
  -u root \
  -p 8080:8080 \
  -v jenkins-data:/var/jenkins_home \
  -v /var/run/docker.sock:/var/run/docker.sock \
  jenkinsci/blueocean
```

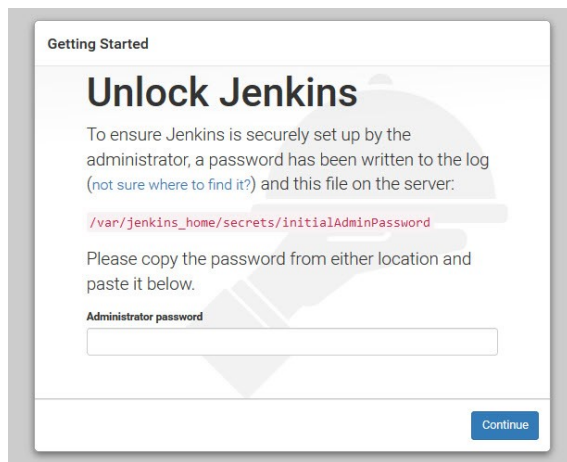
##### Windows

```
docker run ^
  -d ^
  --name docker-jenkins ^
  -u root ^
  -p 8080:8080 ^
  -v jenkins-data:/var/jenkins_home ^
  -v /var/run/docker.sock:/var/run/docker.sock ^
  jenkinsci/blueocean
```

After startup, the server's web portal can be accessed via <http://localhost:8080>.

Note: Since Jenkins will be running in docker, and docker will also be utilized within the Jenkins pipeline to run [Packer](#) and [Terraform](#), it is also important to mount `docker.sock` from the host. That way, the docker controller in Jenkins uses the host's docker daemon to run its containers.

#### Get the admin password



The admin password for Jenkins is created when the container is started with a new volume. If running Jenkins in the foreground, you'll get the following message toward the end of the output:

Please use the following password to proceed to installation:

(redacted)

This may also be found at: `/var/jenkins_home/secrets/initialAdminPassword`

If running Jenkins in the background, you can access the admin password with this command:

#### Linux

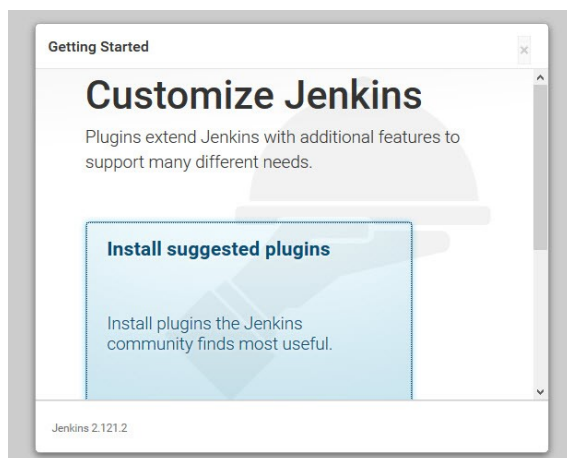
```
docker exec -it docker-jenkins \  
/bin/bash -c "cat /var/jenkins_home/secrets/initialAdminPassword"
```

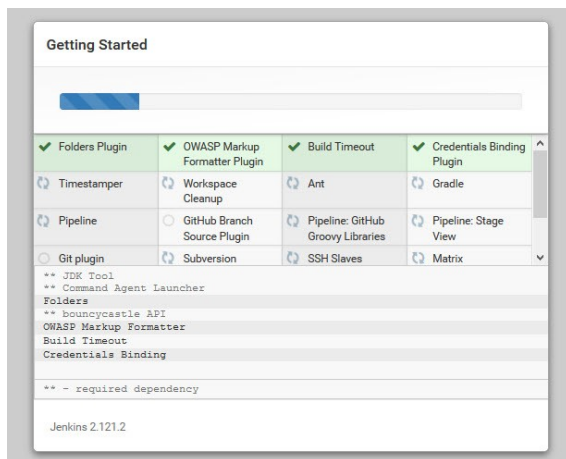
#### Windows

```
docker exec -it docker-jenkins ^  
/bin/bash -c "cat /var/jenkins_home/secrets/initialAdminPassword"
```

## Install default plugins

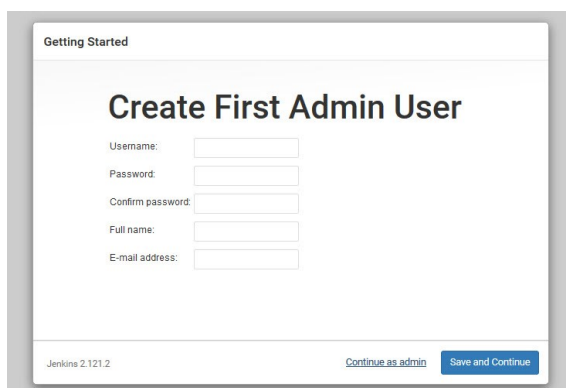
Selecting the default plugins is fine for now, only one other plugin is required, which will be installed later.



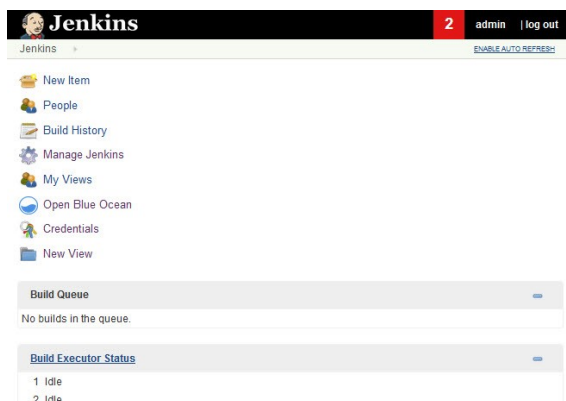
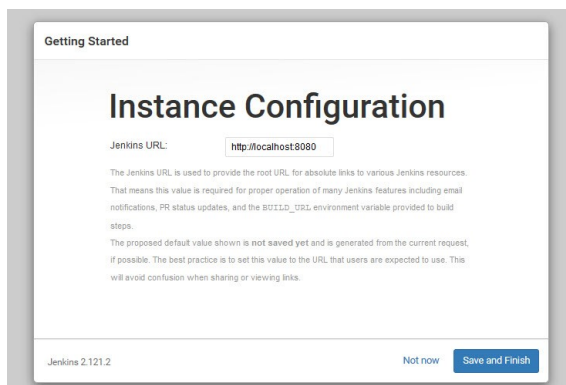


## Finish installation

Create a user account for Jenkins, or 'Continue as admin' (bottom right) with the admin password from earlier.



Since Jenkins is running in a container with port 8080 mapped, the URL <http://localhost:8080> is fine.



## Install CloudBees AWS Credentials plugin

The only required plugin, CloudBees AWS Credentials will allow binding AWS credentials to steps in the pipeline so that Packer and Terraform can access AWS as an IAM user. In the Jenkins side menu, click 'Manage Jenkins', scroll down halfway, and click 'Manage Plugins'.



### Manage Plugins

Add, remove, disable or enable plugins that can extend the functionality of Jenkins.

There are updates available

Select the available tab, and enable the [CloudBees Amazon Web Services Credentials](#) plugin. Then click 'Install without restart'.

Filter:

Updates	Available	Installed	Advanced
Install			
	Name	Version	
<input type="checkbox"/>	<a href="#">Chef Identity</a> This plugin manages user credentials for a <a href="#">Chef</a> server.	1.0.0	
<input type="checkbox"/>	<a href="#">Amazon S3 Bucket Credentials</a>	0.2.2	
<input type="checkbox"/>	<a href="#">Amazon ECR</a> This plugin generates Docker authentication token from Amazon Credentials to access Amazon ECR.	1.6	
<input checked="" type="checkbox"/>	<a href="#">CloudBees Amazon Web Services Credentials</a>	1.23	
<input type="checkbox"/>	<a href="#">Azure Credentials</a> Manage Azure credentials using Jenkins Credentials API	1.6.0	
<input type="checkbox"/>	<a href="#">Azure PublisherSettings Credentials</a> This plugin manage Azure PublisherSettings using Jenkins Credentials API.	1.2	
<input type="checkbox"/>	<a href="#">CloudBees Credentials</a> CloudBees Credentials Plugin This plugin is used to manage credentials within Jenkins. This plugin is used by CloudBees plugins to access CloudBees services.	3.3	

## Add credentials to Jenkins

In the credentials section, you can click the down arrow next to the Jenkins Provider's (global) store.



### Credentials

T	P	Store	Domain	ID	Name
---	---	-------	--------	----	------

Icon: [S](#) [M](#) [L](#)

#### Stores scoped to [Jenkins](#)

P	Store	Domains
	<a href="#">Jenkins</a>	<a href="#">(global)</a>

Or click directly on the Jenkins Provider, and there is a option to 'Add Credentials' in the left menu.

**Jenkins** 2  [admin](#) [log out](#)

[Jenkins](#) [Credentials](#) [System](#) [Global credentials \(unrestricted\)](#)

[Back to credential domains](#) [Add Credentials](#)

**Global credentials (unrestricted)**

Credentials that should be available irrespective of domain specification to requirements matching.

Name	Kind	Description
This credential domain is empty. How about <a href="#">adding some credentials</a> ?		

Icon: [S](#) [M](#) [L](#)

## Github

To allow Jenkins (and the build-container agent) access to checkout this repository from Github, enter a Github username and personal access token for the password. Use 'node-app-git-credentials' as the ID, as it is referenced in the [Jenkinsfile](#).

Kind:

Scope:

Username:

Password:

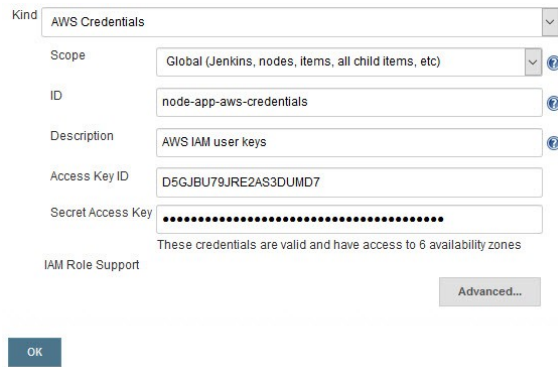
ID:

Description:

OK

## AWS

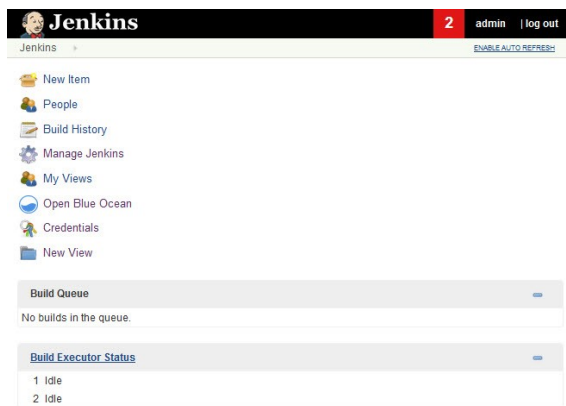
Follow the same steps above, then select 'AWS Credentials' from the 'Kind' dropdown. Use 'node-app-aws-credentials' as the ID, as it is referenced in the [Jenkinsfile](#). Note: No modifications to 'Advanced...' settings or 'IAM Role Support' are required.



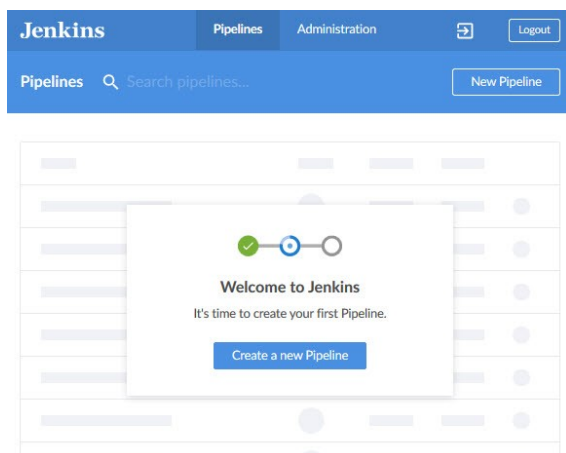
The screenshot shows the 'Add Credentials' form in Jenkins. The 'Kind' dropdown is set to 'AWS Credentials'. The 'Scope' is 'Global (Jenkins, nodes, items, all child items, etc)'. The 'ID' is 'node-app-aws-credentials'. The 'Description' is 'AWS IAM user keys'. The 'Access Key ID' is 'D5GJBU79JRE2AS3DUMD7'. The 'Secret Access Key' is masked with dots. Below the fields, a message states: 'These credentials are valid and have access to 6 availability zones'. There is an 'Advanced...' button at the bottom right of the form. An 'OK' button is visible below the form.

## Create a pipeline

A pipeline is the CI/CD component of Jenkins. Jenkins' Blue Ocean makes creating pipelines (and their Jenkinsfile representations) simple. User's can create a pipeline from a straightforward UI, and then export that pipeline as a [Jenkinsfile](#).



From the Jenkins home screen, click 'Blue Ocean' in the left menu or use the URL <http://localhost:8080/blue/organizations/jenkins>.



Click 'Create a new Pipeline' and select from a list of Git providers. For this repository, select 'Github'.

Where do you store your code?

Bitbucket Cloud	Bitbucket Server
GitHub	GitHub Enterprise
Git	

Jenkins needs access to the Github repository via an access token. Clicking on 'Create an access token here', will take you to the personal access token page in Github, and pre-select the specific permissions required for working with Jenkins.

Where do you store your code?

Bitbucket Cloud	Bitbucket Server
GitHub	GitHub Enterprise
Git	

Connect to GitHub

Jenkins needs an access token to authorize itself with GitHub.  
[Create an access token here.](#)

Your GitHub access token

Complete

After importing a Github personal access token, select the appropriate organization and repository containing the application when prompted.

Connect to GitHub

Jenkins needs an access token to authorize itself with GitHub.  
[Create an access token here.](#)

.....

Which organization does the repository belong to?

craigsands

Complete

Which organization does the repository belong to?

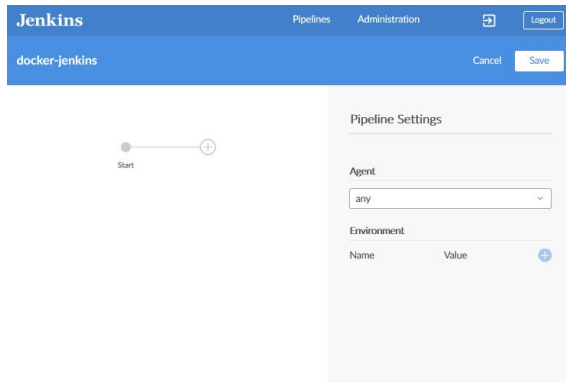
craigsands

Choose a repository

Loaded 6 repositories

Complete

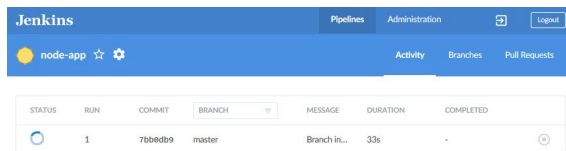
Jenkins then searches the repository for an existing Jenkinsfile. If none exist in the repository selected, Jenkins will start a new one. (In the below screenshot, a separate project, 'docker-jenkins', did not contain a Jenkinsfile.)



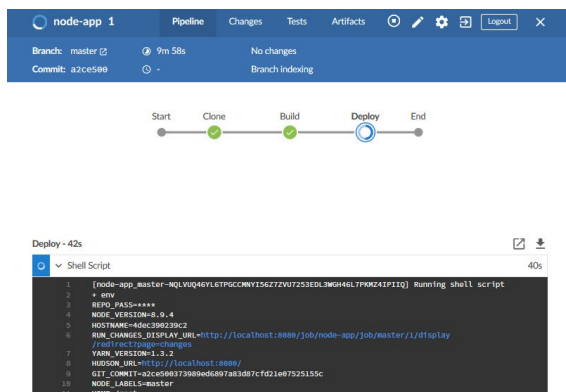
Otherwise, the pipeline will be built from the Jenkinsfile. (This repository, 'node-app' has one in the project root.)



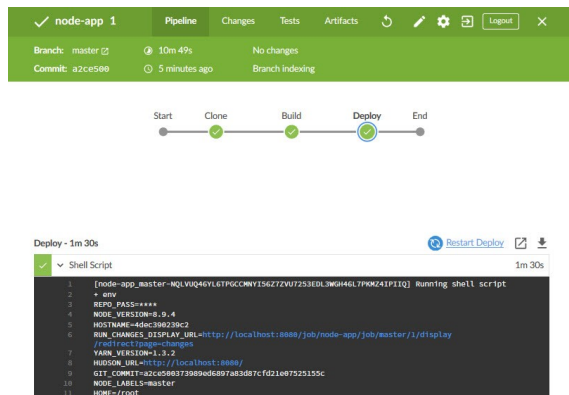
Once the pipeline is created, Jenkins will start the pipeline.



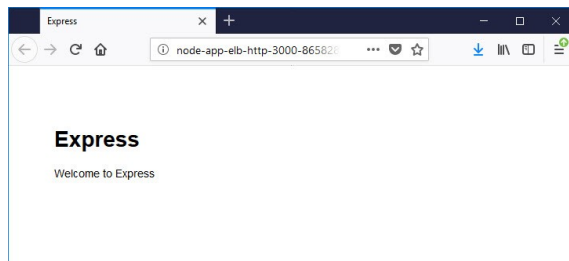
You can then click on it to view the status step-by-step.







Following a successful deployment, the node-app can be accessed by its AWS ELB (Elastic Load Balancer).



## Understand the Jenkinsfile

The [Jenkinsfile](#) is the declarative representation of a pipeline, with the advantage that it can be checked into source control and tracked. See <https://jenkins.io/doc/book/pipeline/jenkinsfile/> for more information on creating a Jenkinsfile from scratch.

The Jenkinsfile included in this repository provisions an [AWS AMI \(Amazon Machine Image\)](#) using [Packer](#) and deploys an [Auto Scaling Group](#) (managed by [CloudFormation](#)) with [Terraform](#).

The Jenkinsfile included consists of the following structure:

```
pipeline {
  agent { dockerfile true }
  environment {} // environment variables
  stages {
    stage('Clone') { steps {...} }
    stage('Deploy-TF-Backend') { steps {...} }
    stage('Commit-TF-Backend-State') { steps {...} }
    stage('Build-Node-App') { steps {...} }
    stage('Deploy-Node-App') { steps {...} }
  }
}
```

## Agent

An [agent](#) at the top-level defines what environment the pipeline will run in. Specifying `dockerfile true` instructs Jenkins to build an image from the included [Dockerfile](#) in the root of the repository. Since deploying this application requires Git, Packer, [Ansible](#), and Terraform, the Dockerfile includes the instructions for installing those prerequisites in the docker image.

The remainder of the Jenkinsfile contains the stages (with arbitrary names) and steps needed to complete the deployment process. These stages and steps are all completed within the docker image (agent) container.

## Clone

As this repository is the primary source for every component of the application, the first stage clones this repository in the build container.

```
stage('Clone') {
  steps {
    checkout scm
  }
}
```

## Deploy-TF-Backend

Terraform uses a `.tfstate` file for keeping track of the AWS resources that were created. By default, the `.tfstate` file is saved in the current working directory. However, Terraform allows for configuring 'backends' for keeping remote state. For this node-app, the state will be saved in an S3 bucket with versioning (as part of the node-app deployment, the final stage of this pipeline). Since the S3 bucket needs to be created, along with a DynamoDb table for locking while Terraform is running, Terraform can be used for that as well. For the S3 and DynamoDb table creation, Terraform will use the `.tf` files in [config/backend](#).

Since Terraform requires access to AWS, the Jenkinsfile includes a class `AmazonWebServicesCredentialsBinding` to bind the AWS credentials specified previously in the Jenkins Credentials section. The credentials are directly referenced by the ID `node-app-aws-credentials`, entered when specifying the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. The `AmazonWebServicesCredentialsBinding` class is provided by the [CloudBees Amazon Web Services Credentials](#) plugin, and automatically adds the credentials as environment variables.

Since the build container is built on demand, the `config/backend` directory is initialized with Terraform first, then applied.

```
stage('Deploy-TF-Backend') {
  steps {
    // https://jenkins.io/doc/pipeline/steps/credentials-binding/
    withCredentials([
      $class: 'AmazonWebServicesCredentialsBinding',
      credentialsId: 'node-app-aws-credentials'
    ]) {
      sh '''
        cd config/backend
        terraform init
        terraform apply \
          -auto-approve \
          -var "aws_region=${AWS_REGION}" \
          -var "lock_table_name=${LOCK_TABLE_NAME}" \
          -var "s3_bucket_name=${S3_BUCKET_NAME}"
        ...
      '''
    }
  }
}
```

## Commit-TF-Backend-State

After the S3 and DynamoDb deployment process is complete, Terraform wrote the state to a local `terraform.tfstate` file. S3 will be used for the backend state for the node-app, but this deployment's state file can be committed to the original (or another) repository for reference and version control. Since committing the file to Github, the `UsernamePasswordMultiBinding` class is used (directly referenced by the ID `node-app-git-credentials`) to allow the step to push the Terraform state file into the repository. Unlike the CloudBees AWS credentials plugin, variables for the username and password are explicitly defined (in this case `REPO_USER` and `REPO_PASS`), and then used in the following shell steps.

```
stage('Commit-TF-Backend-State') {
  steps {
    // https://jenkins.io/doc/pipeline/steps/credentials-binding/
```

```

withCredentials([[
  $class: 'UsernamePasswordMultiBinding',
  credentialsId: 'node-app-git-credentials',
  usernameVariable: 'REPO_USER',
  passwordVariable: 'REPO_PASS'
]]) {
  sh '''
    git add config/backend/terraform.tfstate
    git \
      -c user.name="Craig Sands" \
      -c user.email="craigsands@gmail.com" \
      commit \
      -m "terraform backend state update from Jenkins"
    git push https://${REPO_USER}:${REPO_PASS}@github.com:craigsands/node-app.git master
  '''
}
}
}

```

## Build-Node-App

The fourth stage builds the AWS AMI using Packer. Packer also requires access to AWS, and uses the same CloudBees Amazon Web Services Credentials plugin as the previous Terraform stage. This step uses shell commands to validate the [template file](#) and build the AMI.

```

stage('Build-Node-App') {
  steps {
    // https://jenkins.io/doc/pipeline/steps/credentials-binding/
    withCredentials([[
      $class: 'AmazonWebServicesCredentialsBinding',
      credentialsId: 'node-app-aws-credentials'
    ]]) {
      sh '''
        packer validate \
          -var "aws_region=${AWS_REGION}" \
          ami.json
      '''
      sh '''
        packer build \
          -var "aws_region=${AWS_REGION}" \
          ami.json
      '''
    }
  }
}

```

## Deploy

The final stage again binds AWS credentials so that Terraform can build the appropriate resources in AWS. The [Terraform code](#) references the AMI built in the previous step to use in the deployment. This stage deploys the resources necessary to create an Auto Scaling Group managed by CloudFormation.

```

stage('Deploy-Node-App') {
  steps {
    // https://jenkins.io/doc/pipeline/steps/credentials-binding/
    withCredentials([[
      $class: 'AmazonWebServicesCredentialsBinding',
      credentialsId: 'node-app-aws-credentials'
    ]]) {
      sh '''
        cd config/node-app
        terraform init \

```

```
    -backend-config="region=${AWS_REGION}" \
    -backend-config="bucket=${S3_BUCKET_NAME}" \
    -backend-config="dynamodb_table=${LOCK_TABLE_NAME}"
  terraform apply \
    -auto-approve \
    -var "aws_region=${AWS_REGION}"
  ...
}
}
```