**Integrating a Node.js application running on AWS Beanstalk as a docker container with Gitlab CI/CD**

Many large enterprise organizations constantly look for various AWS services that will help them quickly and reliably build their web applications in a cost-effective way. One such service to build and deploy these web applications in a quicker way is AWS Elastic Beanstalk. With AWS Elastic Beanstalk, we can quickly deploy and manage these applications in AWS cloud without worrying about the infrastructure. AWS Elastic Beanstalk supports applications developed in various programming languages and can be deployable as Docker containers. At the same time, organizations want to automate their development process using continuous integration and continuous delivery and deployment methods. GitLab CI/CD is such kind of tool built into GitLab for software development through continuous methodologies.

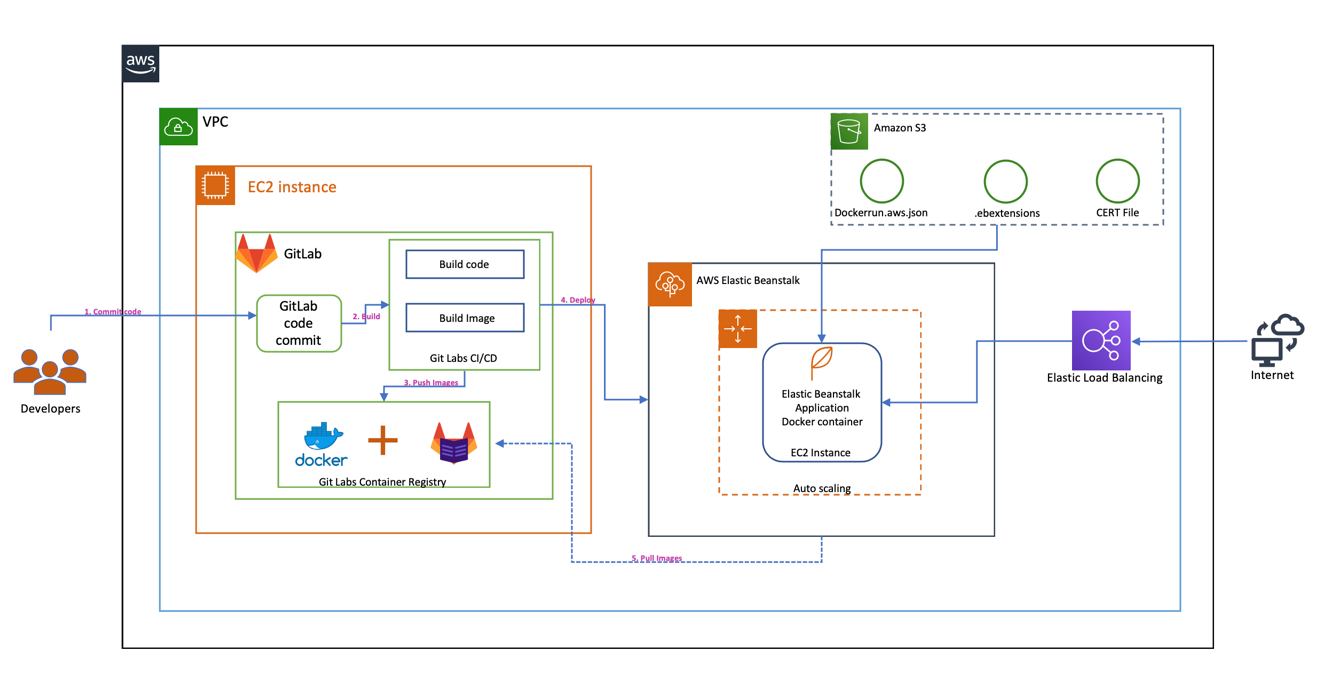
In this blog post, we will walk you through step by step process to deploy a simple Node.js application as a docker container hosted in Gitlab’s container registry as a Docker image into AWS Elastic Beanstalk service. We will see a process to create a pipeline where pushing an image into GitLab container registry, will automatically update in the AWS Elastic Beanstalk environment. Basically, any update to the Node.js application code, or the docker file or any other configuration file committed to GitLab repository, will build a new docker image and pushed to Gitlab container registry and eventually AWS Beanstalk will deploy the new artifact onto the environment.

# Solution Overview

The steps we will follow in this blog post are:

1. Create a Virtual Private Cloud (VPC), and an Amazon S3 bucket with a CloudFormation script.
2. Provision a sample Gitlab environment with container registry, Gitlab runner and a docker service in an EC2 instance with a CloudFormation script.
3. Create Route 53 hosted zone and a record set pointing to the Gitlab hosted instance.
4. Upload your ssh key and configure a personal access token in the Gitlab.
5. Download a simple Node.js application and configure it.
6. Create a docker image and push it to Gitlab container registry.
7. Provision a sample AWS Elastic Beanstalk application and environment with a CloudFormation script.
8. Configure Dockerrun.aws.json and, .platform/hooks files to deploy it into AWS Beanstalk environment.
9. Update the sample code and verify the if it is updated the environment in AWS Elastic Beanstalk.

The following diagram explains how the services work together.



# Prerequisites and assumptions

To follow the steps outlined in this blog post, you need the following:

* An AWS account that provides access to AWS services.
* The templates and code are intended to work in the US-EAST-1 region only and they are only for demonstration purpose only and not for production use.

Additionally, be aware of the following:

* We configure all services in the same VPC to simplify networking considerations.
* **Important**: The [AWS CloudFormation](https://aws.amazon.com/cloudformation/) templates and the sample code that we provide use hard-coded user names and passwords and open security groups. These are just for testing purposes and aren't intended for production use without any modifications.

**1. Use the AWS CloudFormation template to configure Amazon VPC and S3**

In this step, we set up a VPC, public subnet, internet gateway, route table, and a security group. The security group has one inbound access rule. The inbound rule allows access to any TCP port from any host within the same security group. We use this VPC and subnet for all other services that are created in the next steps. This template also creates a standard Amazon S3 bucket with a provided bucket name to store the input data and processed data.

You can use this [downloadable](https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-makeshift/cloudformations/step-1-vpc-dynamo-s3.yaml) AWS CloudFormation template to set up the previous components. To launch directly through the console, choose Launch Stack.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/new?stackName=CF-Root-Makeshift-blog-Step1&templateURL=https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-makeshift/cloudformations/step-1-vpc-dynamo-s3.yaml)

After you specify the template details, choose Next. On the Review page, choose Create.

When the stack launch is complete, it should return outputs similar to the following.

|  |  |
| --- | --- |
| Key | Value |
| StackName | Name |
| VPCID | vpc-xxxxxxxx |
| SubnetIDA | subnet-xxxxxxxx |
| SubnetIDB | subnet-xxxxxxxx |
| SubnetIDC | subnet-xxxxxxxx |
| VPCSubnets | VPCSubnetsList |
| AWSBLOGBEANAccessSecurityGroup | Security group |

Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:

$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-east-1 --query 'Stacks[0].Outputs'

**2. Use the AWS CloudFormation template to configure EC2 instance and setup gitlab on the host.**

In this step, we set up an EC2 instance and install gitlab software and also configures runner, docker as well. <Need to add more information>  
  
You can use this [downloadable](https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-makeshift/cloudformations/step-1-vpc-dynamo-s3.yaml) AWS CloudFormation template to set up the previous components. To launch directly through the console, choose Launch Stack.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/new?stackName=CF-Root-Makeshift-blog-Step1&templateURL=https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-makeshift/cloudformations/step-1-vpc-dynamo-s3.yaml)

Provide EC2 Key pair. After you specify the template details, choose Next. On the Review page, choose Create.

When the stack launch is complete, it should return outputs similar to the following.

|  |  |
| --- | --- |
| Key | Value |
| StackName | Name |
| GitEc2PublicDNS | ec2-xx-xx-xx-xx.compute-1.amazonaws.com |
| GitEc2PublicIp | xx-xx-xx-xx |
| ExpS3Bucket | <bucket-that-was-created> |

**Note:** To install and configure GitLab, it takes around ~10 minutes. Please wait while the GitLab is completely configured and running.

Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:

$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-east-1 --query 'Stacks[0].Outputs'

**Prepare .dockercfg file:**

To provide access to GitLab registry from ELB environment, we need to create a “.dockercfg” file and store this in an Amazon S3 bucket.

To get the configuration, login to GitLab host where the docker is also configured.

* $ ssh -i <pem\_file> ec2-xx-xx-xx-xx.compute-1.amazonaws.com
* $ Run: “sudo docker login --username=root --password=changeme registry.beangitlab.com”
* $ sudo cat ~/.docker/config.json

You will see an output like below.

$ sudo cat ~/.docker/config.json

{

"auths": {

"registry.beangitlab.com": {

"auth": "cmXXXDpjaGFXXXXXVtZQ=="

}

}

}

* Create a new .dockercfg file on your local machine or somewhere.

vi docker.cfg  
  
From the above config.json output, copy the contents into docker.cfg file.  
{

"auths": {

"registry.beangitlab.com": {

"auth": "cmXXXDpjaGFXXXXXVtZQ=="

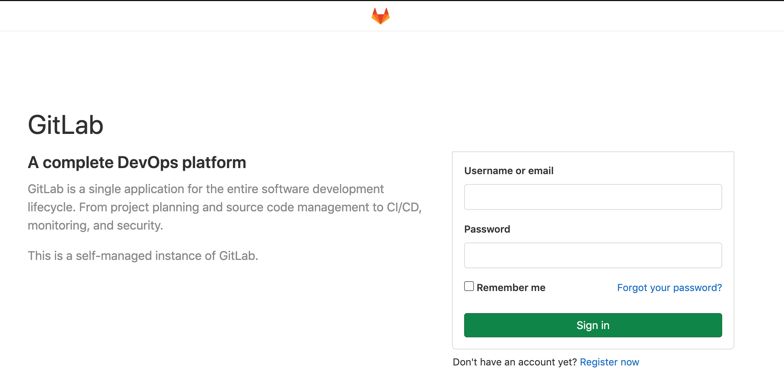
}

}

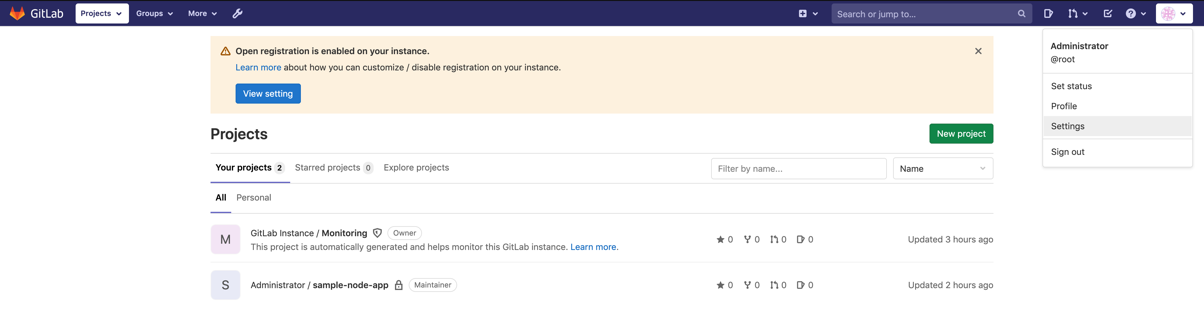
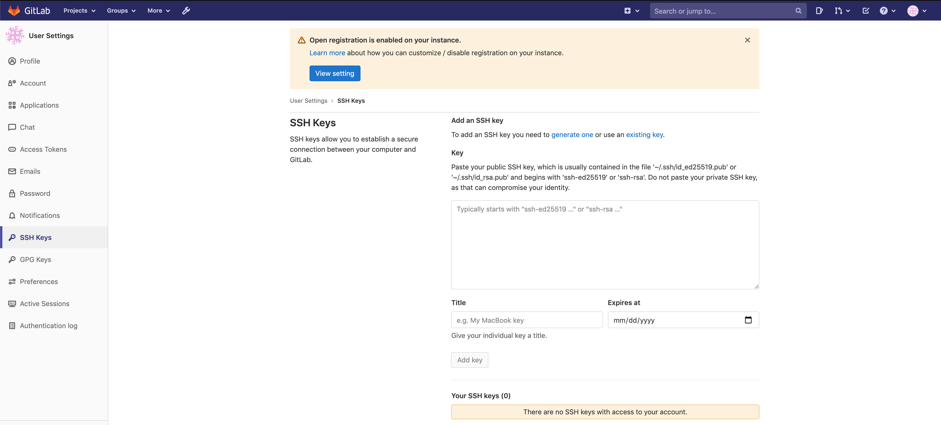
}

* Now copy the .dockercfg file to the S3 bucket that was created in the previous step.  
  From your mac or local machine.  
  aws s3 cp .dockercfg <S3\_BUCKET>

**Connecting to Gitlab URL:**

Now connect to the gitlab running on ec2 instance using the PublicDNS address that was shown in the output from the above cloud formation template. Open your browser and enter the PublicDNS in the address bar.  
You will see the below page. Provide username as “root” and password as “changeme” to login to the gitlab platform. Please note that these are set in the “gitlab-setup.sh” script.  
****

**Update SSH Key in GitLab:**

Once you login to the Gitlab platform we need to add your local host ssh key to establish a secure connection between your local computer/mac and the GitLab.  
  
Open settings page by selecting the “Settings” link as shown in the below picture.  
****Click on “SSH Keys” on the left side menu and it will show the below page.  
****

Get your public ssh key from your mac/computer and paste it under the “Key” section. On a mac computer, you can get the public key by running the below command at the “terminal”. Once you paste the ssh key, click on “Add Key” button.

**cat ~/.ssh/id\_rsa.pub**

**Setting up a sample node.js application and the AWS Elastic Beanstalk configuration files**

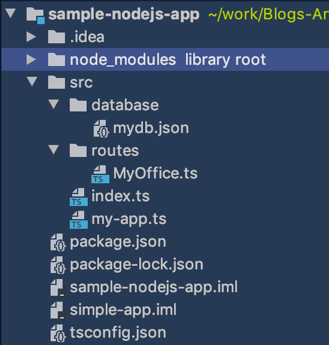
Download the sample typescript based nodejs application from [here](https://skkodali-proserve.s3.amazonaws.com/beanstalk-gitlab/sample-nodejs-app.zip) onto your local machine.

Make sure you have “npm” and “node” setup on your computer.

Unzip the sample app that was downloaded.

* cd ~
* Copy the above downloaded zip file here.
* unzip sample-nodejs-app.zip

Once you unzip the folder, the project structure looks like this.



Run the below command to install the “express” module. Express.js is a framework built on top of Node.js to setup routes easily. It has a lot of features, utility methods and middleware to help us create scalable and robust APIs quickly.

* Open the command shell and navigate to the “sample-nodejs-app” folder.
* npm install @types/node @types/express @types/body-parser --save-dev

To quickly test the application, you can do the below steps.

* npm run dev

It will show output something similar to this. And the application is running on port number 5000.

“  
hello, Srikanth

My app port number is : 5000.

“

You can use the below curl command to query the api and see if the application is working as expected.

* curl <http://localhost:5000/emps>
* curl [http://localhost:5000/emps/2](http://localhost:5000/emps/2eiifccrckvuftdhijdjnffbnvtifunvicjbvcvtbibht)

**Compiling the application using “tsc” command:**

The tsc command invokes the typescript compiler. It uses the tsconfig.json file to compile the application. Once the compilation is complete, it generates the “dist” directory.

**Converting a sample standalone application into a docker container:**

Our end goal is to deploy a docker container into Amazon Beanstalk service, lets convert this standalone application into a docker container.

**Dockerfile**

The Dockerfile already exists in the zip file. The contents of the file are as below.

FROM node:13.8.0-slim  
WORKDIR /app  
COPY dist/ /app/dist/  
COPY src/ /app/src/  
COPY \*.json /app/  
COPY .ebextensions/\* /app/.ebextensions/  
  
RUN npm ci --quiet && npm run build && npm prune --production  
  
ENV ***NODE\_ENV***=production  
# COPY myca.crt /usr/local/share/ca-certificates  
# RUN update-ca-certificates  
  
CMD node ./dist/index.js  
  
EXPOSE 5000

The above docker file copies the contents of the sample application files into “/app” directory in the container. And whenever the docker container starts, it will execute the index.js file under “dist” directory.

**Dockerrun.aws.json template file:**

A Dockerrun.aws.json file describes how to deploy a remote Docker image as an Elastic Beanstalk application. In our case, our docker image is stored in the “GitLab container registry” and we will point to this “image” location and specify that in this file. For more information about Dockerrun.aws.json file contents, please check this [link](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/single-container-docker-configuration.html). The important section in our case is “Image” section in this configuration file. It specifies the Docker base image on an existing Docker repository from which you’re building a docker container.

Since we are using “GitLab” for CI/CD, whenever we make some changes to the code, the pipeline will execute and will create a new container image file in the GitLab container registry. Since we do not know the “image” name ahead, we cannot hardcode this value. For this we will create a template file first and will replace the “image” name dynamically with in the GitLab CI/CD pipeline using “.gitlab-ci.yml” file.

***{*** "AWSEBDockerrunVersion": "1",  
 "Authentication": ***{*** "Bucket": "$s3\_bucket\_name",  
 "Key": ".dockercfg"  
 ***}***,  
 "Ports": ***[  
 {*** "ContainerPort": "8080"  
 ***}  
 ]***,  
 "Image": ***{*** "Name": "$image\_name",  
 "Update": "true"  
 ***}  
}***

Here we are specifying the “.dockercfg” file under “Authentication” section. The location needs to be in an S3 bucket. The s3 bucket name is also defined in the .gitlab-ci.yml file and will be passed to this template file to generate the actual file. Note that “.dockercfg” file was created in the previous section.

In the “Image” section, we are not hardcoding the “image” name. Instead we are specifying this as a variable and in the “.gitlab-ci.yml” file, we will update this value dynamically whenever the new image is created in the GitLab container registry.

**.gitlab-ci.yml file:**

GitLab Ci/CD pipelines are configured using a YAML file called “.gitlab-ci.yml”. We need to create this file in the application root directory. It defines structure and order of the pipeline and decides what to execute in the “GitLab Runner”. This file is also provided in the above sample-nodejs-app zip file.

In our case, we divided the structure of the pipeline into four main sections. These are

1. variables
2. before\_script
3. stages
4. build
5. deploy-dev

**Variables section:**

In this section, update the “S3\_BUCKET\_NAME” with the s3 bucket name that was created in the previous section.

**before-script section:**

In the “before\_script” section, we need to update the /etc/hosts file in the container with the docker host’s IP address. In our case, the docker is running on the same server as GitLab runner. This entry is needed for name resolution, otherwise it will use the “nameserver” ip address defined in the /etc/resolv.conf” file and it will fail to connect.

echo XX.XX.XX.XX ec2-XX-XX-XX-XX.compute-1.amazonaws.com registry.beangitlab.com

>> /etc/hosts

Copy the “ca.cert” file that was created when we created the GitLab Runner on an ec2 instance. The ca.cert file was created under “/etc/gitlab-runner/config/certs/ca.crt”. The default location to store the certs is “/usr/local/ca-certificates/”. We need to copy this file onto docker container’s “/usr/local/ca-certificates/ folder.

- cp /etc/gitlab-runner/config/certs/ca.crt /usr/local/share/ca-certificates/ca.crt

**stages:**

This section is used to define stages that contain jobs and allows for having flexible multi stage pipelines. In our “stages” section, we have two stages: “build” and “deploy”. First, it will execute all the steps under “build” stage and then it will execute the steps under “deploy” stage.

stages:  
 - build  
 - deploy

**build:**

In this stage, we will update the ca-certificates, connect to docker and build a Docker image from the given Dockerfile. And we use “docker push” to share the image to the self-hosted GitLab registry.

build:  
 stage: build  
 services:  
 - name: docker:dind  
 script:  
 - update-ca-certificates  
 - echo "Certificates are updated."  
 - docker login -u $CI\_REGISTRY\_USER -p $CI\_REGISTRY\_PASSWORD $CI\_REGISTRY  
 - docker build -t registry.srikanth.com/root/simple-node-app .  
 - docker push registry.srikanth.com/root/simple-node-app

**deploy-dev:**In this stage, we will install the necessary packages on the docker container including python2, docker-ce and an open source tool called “[sigil](https://github.com/gliderlabs/sigil/)”.

This “[sigil](https://github.com/gliderlabs/sigil/)” tool will be used to update the “image” name entry dynamically in the “Dockerrun.aws.json” file that was created earlier. This tool will read the latest Docker container image name using the $CI\_REGISTRY\_IMAGE variable and updates the image\_name in the given template file name and creates a new file with the name “Dockerrun.aws.json”.

# To download the sigil software from github

- curl -L "https://github.com/gliderlabs/sigil/releases/download/v0.6.0/sigil\_0.6.0\_$(uname -sm|tr \ \_).tgz" | tar -zxC /usr/local/bin

# use sigil command to update value of the “image\_name” in Dockerrun.aws.json file.  
- sigil -p -f Dockerrun.aws.json.template "image\_name=$CI\_REGISTRY\_IMAGE" > Dockerrun.aws.json

We also install “awsebcli” to easily manage beanstalk applications.

**Deploying an artifact:**

You can tell the EB CLI to deploy a Docker Image from a container registry. In our case our container registry is Gitlab Registry. We can do this by specifying the following lines in “.elasticbeanstalk/config.yml” file.

* cd ~/sample-nodejs-app/
* mkdir -p .elasticbeanstalk
* cd .elasticbeanstalk
* touch config.yml

Append the below lines in the “config.yml” file.

deploy:  
 artifact: Dockerrun.aws.json

**Automating the SSL certification installation on Elastic BeanStalk instances:**

Since our AWS Elastic Beanstalk application is connecting to GitLab Registry to access the Docker image, it uses SSL certificate to connect to GitLab Registry. So, the SSL certificates needs to be copied to Elastic Beanstalk instances automatically whenever the new instances are created.

To achieve this, first we need to copy the necessary certificate files on to an S3 bucket. This was already done as part of the GitLab Registry setup. In the code repository file gitlab-setup.sh file, which was used to create the GitLab Registry node, the below function will copy the necessary certificate file to the specified S3 bucket. You can see the code snippet below.

\_upLoadToS3Path***()  
{*** ${AWS} $***{***S3\_COPY***}*** $***{***GITLAB\_CERTS\_DIR***}***/$***{***CERT\_CRT***} s3://***$***{***1***}/myca.crt  
}***

Then, we need to create a “*.ebextensions*” folder in the application root directory. In that directory, we need to create a “*.config*” file with any prefix. For example, in our case, we will use “*mycommands.config*” file. Whatever the commands you specify in this file, it will be executed on the Elastic Beanstalk instances whenever they first created. The sample-nodejs-app.zip file contains this .ebextensions/mycommands.config file.

In this file, we will specify from which location we need to get the SSL certificate file and where we need to copy on the AWS Elastic Beanstalk application instance. To access an Amazon S3 bucket, AWS Beanstalk application instances requires an instance profile that should be attached.

We already created this Instance profile using a CloudFomration script as part of the initial GitLab Registry setup. This instance profile has read access on the S3 bucket that was created in the same setup. This bucket contains the actual SSL certificate that we need to access to.

**Update *mycommands.config* file:**

* cd ~/sample-nodejs-app/
* Open the mycommands.config file using your IDE or terminal.

**commands section:**

The elastic beanstalk instances should also able to connect to GitLab Registry server. To enable this connectivity, we need add the below entry in /etc/hosts file whenever they created. To achieve this we can use the following syntax.

commands:  
 01-myCommand:  
 command: "echo 'XX.XX.XX.XX ec2-XX-XX-XX.compute-1.amazonaws.com registry.beangitlab.com' >> /etc/hosts"

Replace the XX.XX.XX.XX with the valid IP address of the EC2 instance where the Gitlab Registry was setup.

We also need to add this entry to Docker container as well, whenever it gets created. To do this, we can use the following syntax.

container\_commands:  
 01-myContainerCommand:  
 command: "echo 'XX.XX.XX.XX ec2-XX-XX-XX.compute-1.amazonaws.com registry. beangitlab.com' >> /etc/hosts"

Next we need to use Resources Key to add an authentication method named “S3Auth” and use it to download the SSL certificate file from the specified Amazon S3 bucket.

Resources:  
 AWSEBAutoScalingGroup:  
 Metadata:  
 AWS::CloudFormation::Authentication:  
 S3Auth:  
 type: "s3"  
 buckets: ["<S3\_BUCKET\_THAT\_WAS\_USED\_TO\_STORE\_SSL\_CERT>"]  
 roleName:  
 "Fn::GetOptionSetting":  
 Namespace: "aws:autoscaling:launchconfiguration"  
 OptionName: "IamInstanceProfile"  
 DefaultValue: "<INSTANCE\_PROFILE\_ROLE\_THAT\_WAS\_CREATED>"  
  
files:  
 "/usr/local/share/ca-certificates/ca.crt" :  
 mode: "000400"  
 owner: root  
 group: root  
 authentication: "S3Auth"  
 source: https://s3.amazonaws.com/<S3\_BUCKET\_THAT\_WAS\_USED\_TO\_STORE\_SSL\_CERT>/myca.crt

This section of the config file will give read permissions to the specified S3 bucket and copies the “myca.crt” file from that S3 bucket to “/usr/local/share/ca-certificates/” folder in the docker container that will be created inside AWS Elastic Beanstalk instances.

**Creating an AWS Elastic Beanstalk application and environment using a CloudFormation template.**

Now lets create a AWS Beanstalk application and a AWS Beanstalk environment to deploy a sample application. When we first initialize the AWS Beanstalk application, we will use a sample application.

You can use this [downloadable](https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-makeshift/cloudformations/step-1-vpc-dynamo-s3.yaml) AWS CloudFormation template to set up the previous components. To launch directly through the console, choose Launch Stack.

[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/new?stackName=CF-Root-Makeshift-blog-Step1&templateURL=https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-makeshift/cloudformations/step-1-vpc-dynamo-s3.yaml)

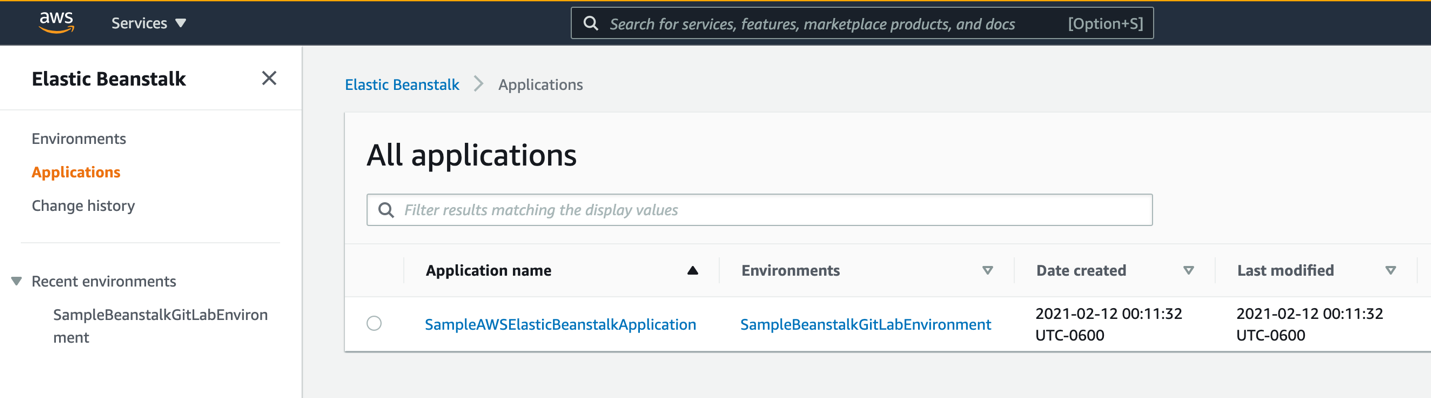
After you specify the template details, choose Next. On the Review page, choose Create.

**Note:** This Cloudformation template will take around 10 minutes to completely create the AWS Beanstalk application and an environment with a sample application.

Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:

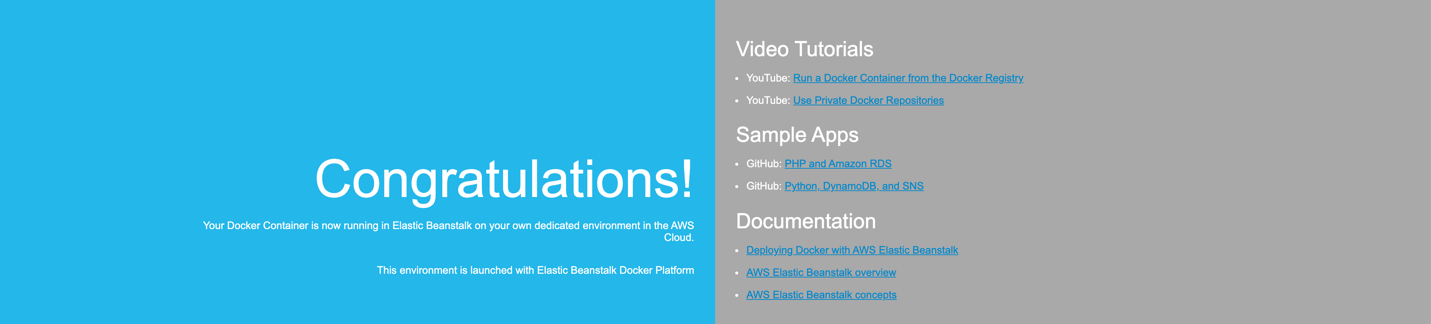
$ aws cloudformation describe-stacks --stack-name *<stack\_name>* --region us-east-1 --query 'Stacks[0].Outputs'

Once the Cloudformation template’s execution is successful, you can see the newly created “Application” and “Environment” in the AWS Elastic Beanstalk service UI.



It also creates an Elastic Load Balancer and we can use ELB URL to connect to the application.

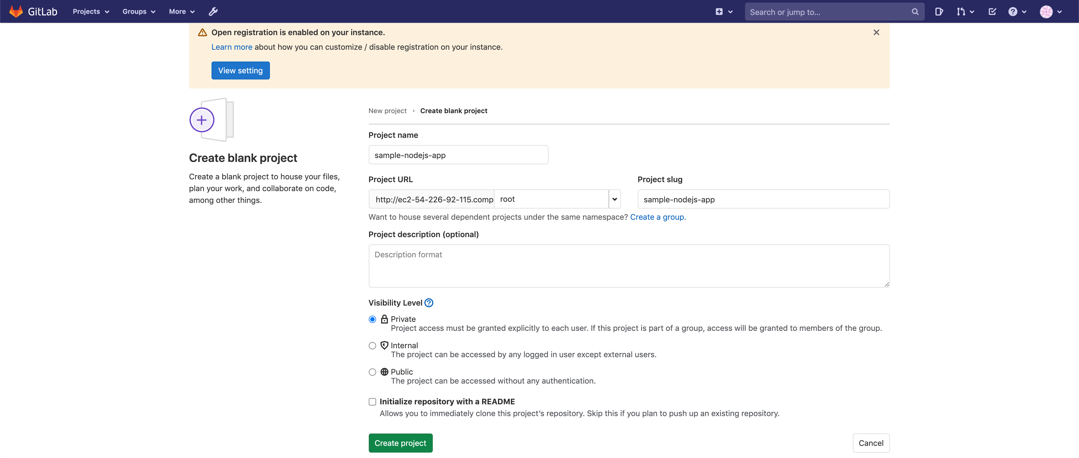
Go to Amazon Elastic Load Balancer console, and select the newly created ELB and copy the DNS name. Use DNS name and paste it in the browser address bar. It will open a default page like below.



**Setup Git repo for the sample-nodejs-app application:**

Let’s setup the Git repo for our sample-nodejs-app application. Login to GitLab UI.

Click on “New Project” button and select “Create blank project”. Provide “sample-nodejs-app” for Project Name and click on “Create Project”.



Let’s push our existing code to the GitLab repo. As soon as we push the code to the repo, it will start executing the pipeline.

On your local computer/mac, run the below git commands to push the code to the GitLab repository.

* cd ~/sample-nodejs-app/
* git init
* git remote add origin [git@ec2-xx-xx-xx-xx.compute-1.amazonaws.com:root/sample-nodejs-app.git](mailto:git@ec2-xx-xx-xx-xx.compute-1.amazonaws.com:root/sample-nodejs-app.git)
* git add .
* git commit -m “Initial commit”
* git push -u origin master

**Update the sample-nodejs-app application and push it to the gitlab:**

section

**Conclusion**