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

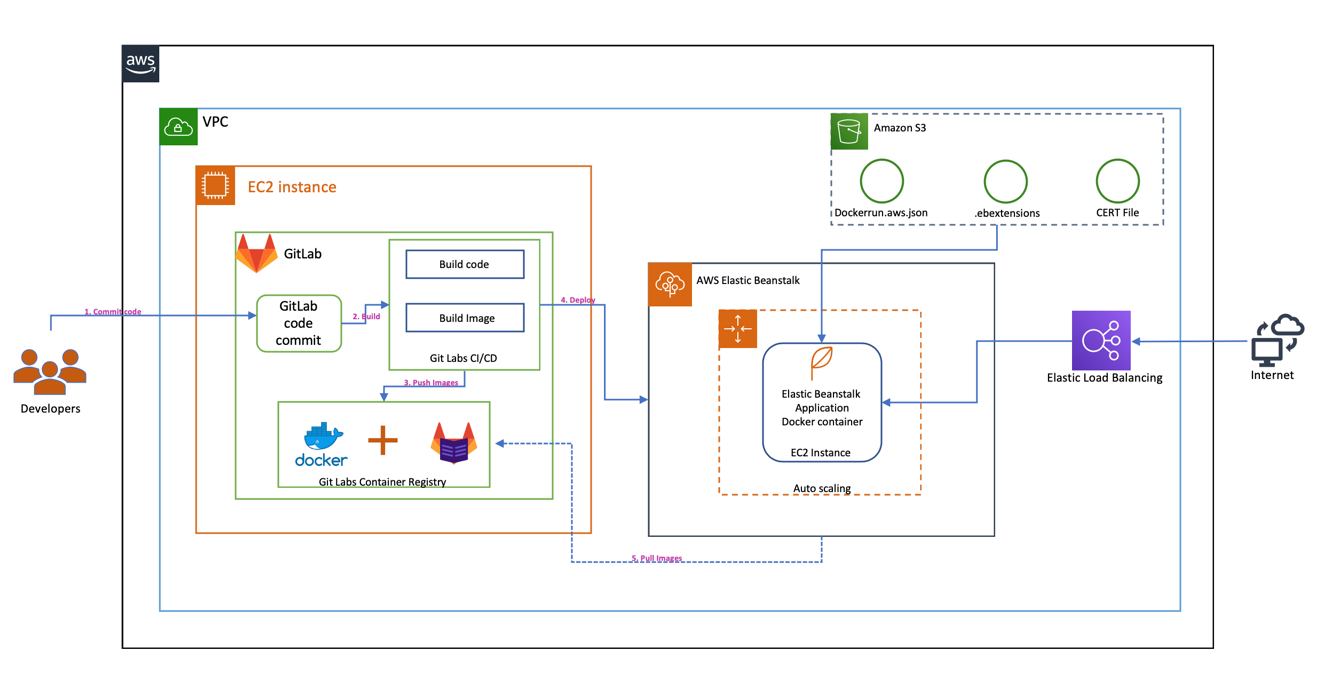
Many large enterprise organizations constantly look for various AWS services that will help them quickly and reliably build their web applications in a cost-effective way. One such service to build and deploy these web applications in a quicker way is AWS Elastic Beanstalk. With AWS Elastic Beanstalk, 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 to push a docker 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.
* You need to have Node.js and its package manager “npm” installed on your local machine. For installing Node.js and npm on mac, you can run “brew update” and “brew install node” commands on your terminal.
* You also need to have TypeScript compiler (tsc) installed on your local machine. Our sample application is developed using TypeScript which is a superset of JavaScript that compiles to plan JavaScript. To install TypeScript compiler on your local mac, you can run “npm install -g typescript” command on your terminal.
* Update with “git” requirement.

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-beanstalk-gitlab/cloudformations/step1-vpc.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=Step1-Bean-GitLab-blog&templateURL=https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-beanstalk-gitlab/cloudformations/step1-vpc.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-bigdat) 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=Step2-Bean-GitLab-blog&templateURL=https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-beanstalk-gitlab/cloudformations/step2-create-ec2-instance.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.

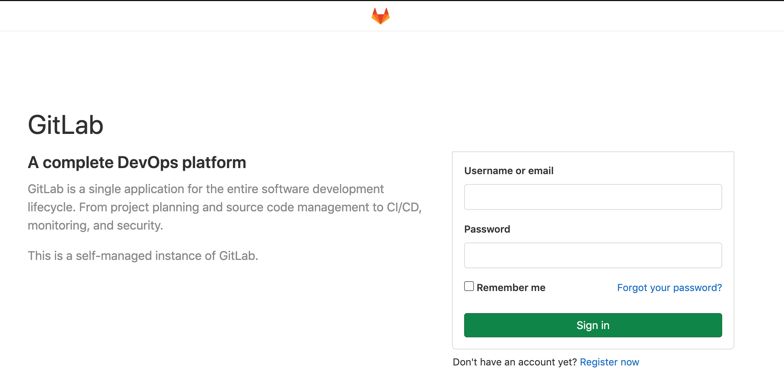
|  |  |
| --- | --- |
| Key | Value |
| StackName | Name |
| GitEc2PublicDNS | ec2-xx-xx-xx-xx.compute-1.amazonaws.com |
| GitEc2PublicIp | xx-xx-xx-xx |
| ExpS3Bucket | <bucket-that-was-created> |

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

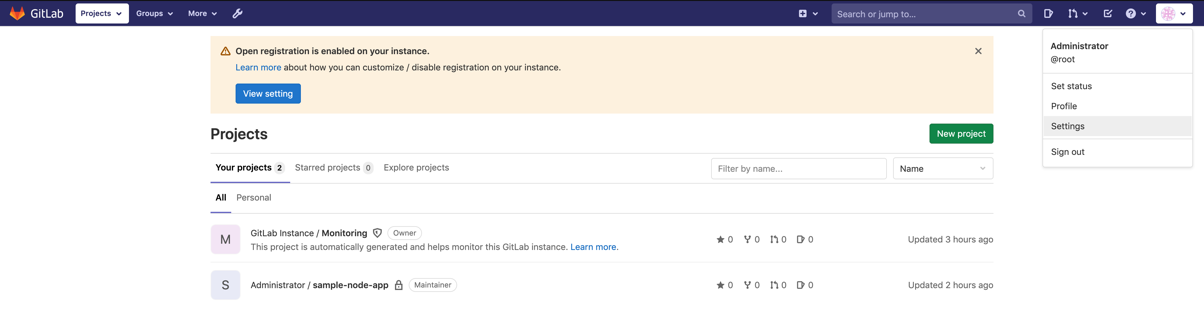
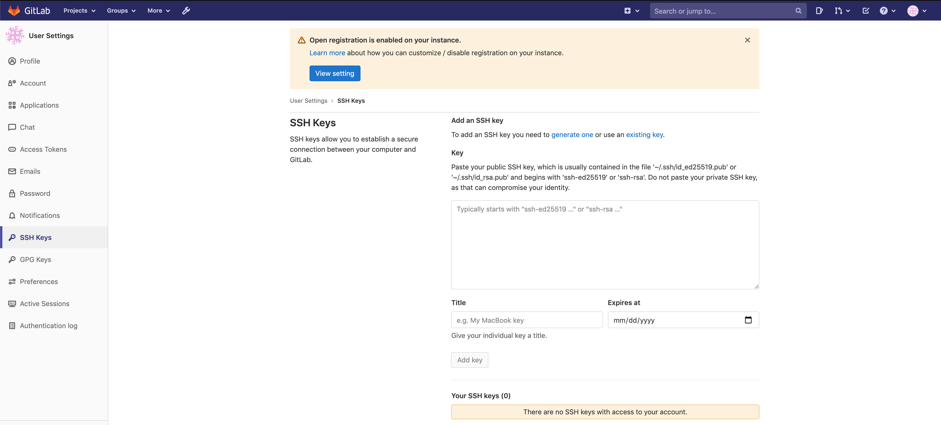
Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:

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

**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.  
****

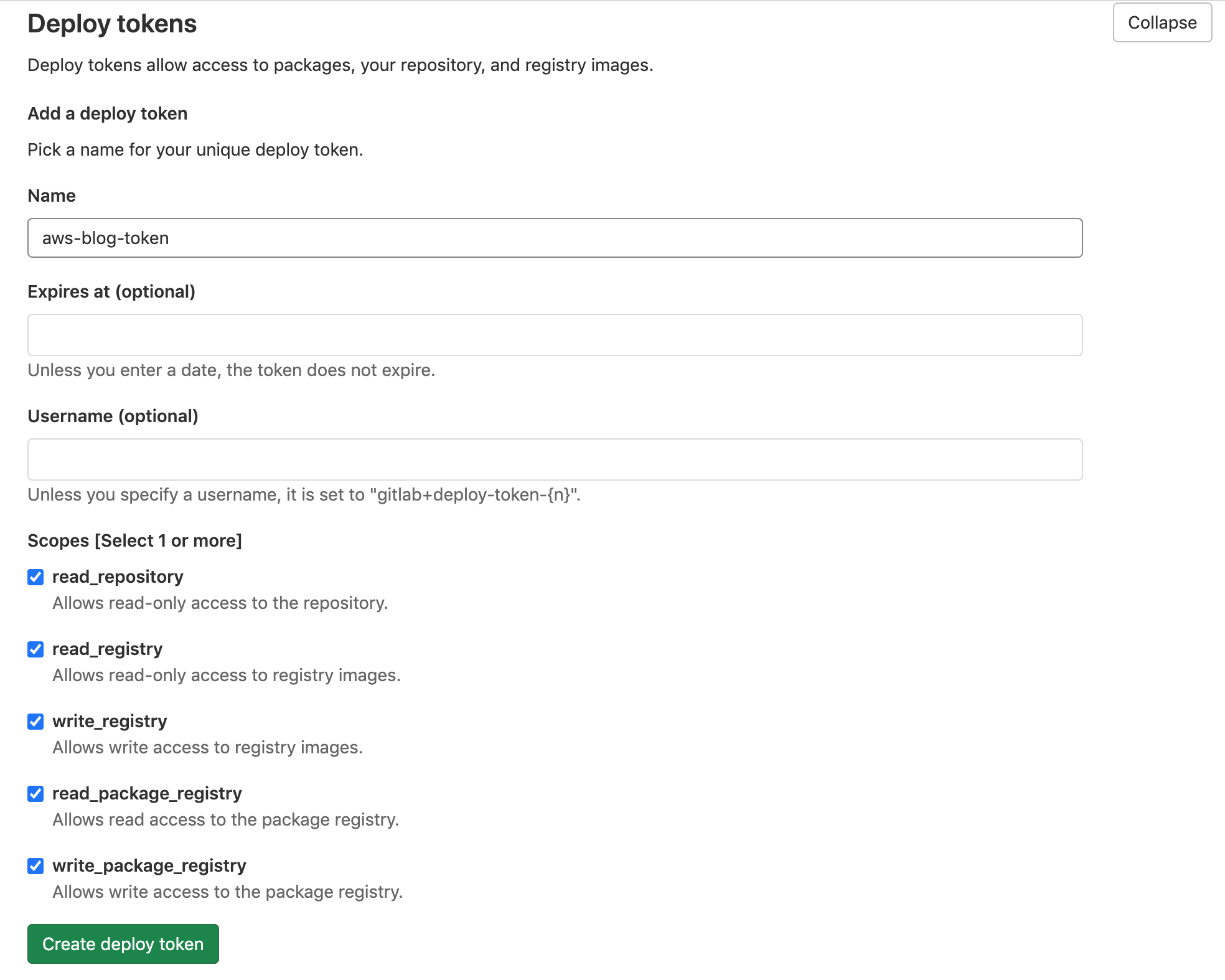
Get your public ssh key from your mac/computer and paste it under the “Key” section. On a mac computer, you can get the public key by running the below command at the “terminal”. Once you paste the ssh key, click on “Add Key” button.

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

**Sample node.js application and the AWS Elastic Beanstalk configuration files**

As part of the Ec2 instance creation and gitlab setup, the AWS CloudFormation script in step2, will also downloads a sample application and pushes the code into gitlab repository. Go to “Projects” menu in gitlab home page after logging in, and click on “Your Projects” and you will see “sample-nodejs-app” project as well.

**Adding Deployment tokens in the Gitlab console**

In order for Amazon Elastic Beanstalk to pull the docker image of the “sample-nodejs-app” that is built on Gitlab and stored in the Gitlab Private Docker registry, we need to create “[GitLab Deploy Tokens](https://docs.gitlab.com/ee/user/project/deploy_tokens/)”. Deploy tokens allow access to packages, your repository and registry images. Once you sign in to your Gitlab account, Select “sample-nodejs-app” -> Go Settings -> Repository. Expand “Deploy tokens”. Provide “Name” and select all options under “Scopes” section as shown in the below screenshot. Click on “Create deploy token” button.  
  


It will create the username as “gitlab+deploy-token-1” and token with some random alpha numeric characters. Save these values before navigating to some other screen as the token cannot be recovered.

We use the “username” and “generated token” to connect to docker using “docker login <registry.domain>” and it will generate the Docker “config.json” file. Once this file is generated, we use the populated token along with the domain name to S3 bucket. This part is of generating this config.json file and uploading to s3 bucket is automatically done inside the “.gitlab-ci.yml” file. You do not need to manually do these steps. Here is the code snippet in the “.gitlab-ci.yml” file.

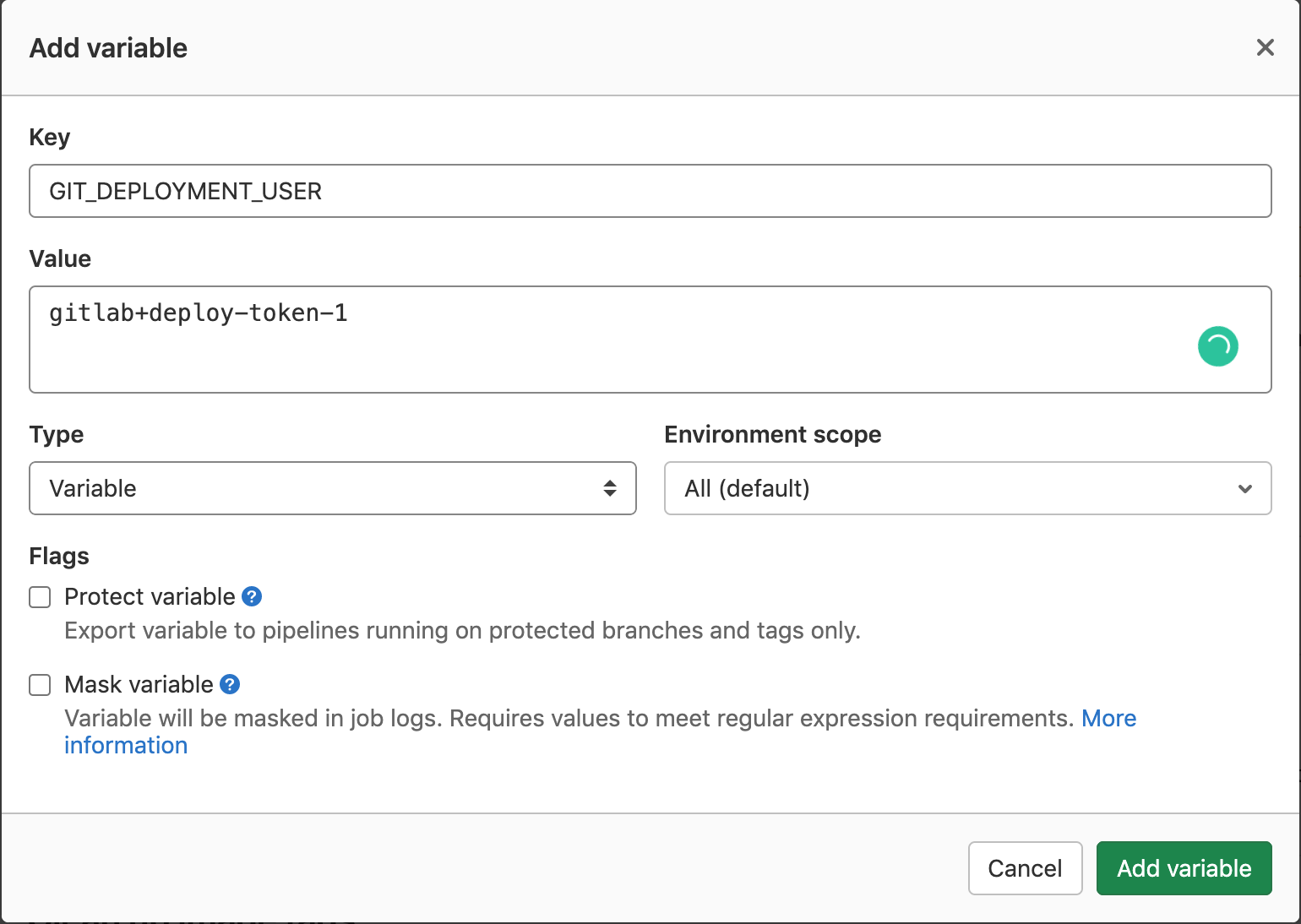
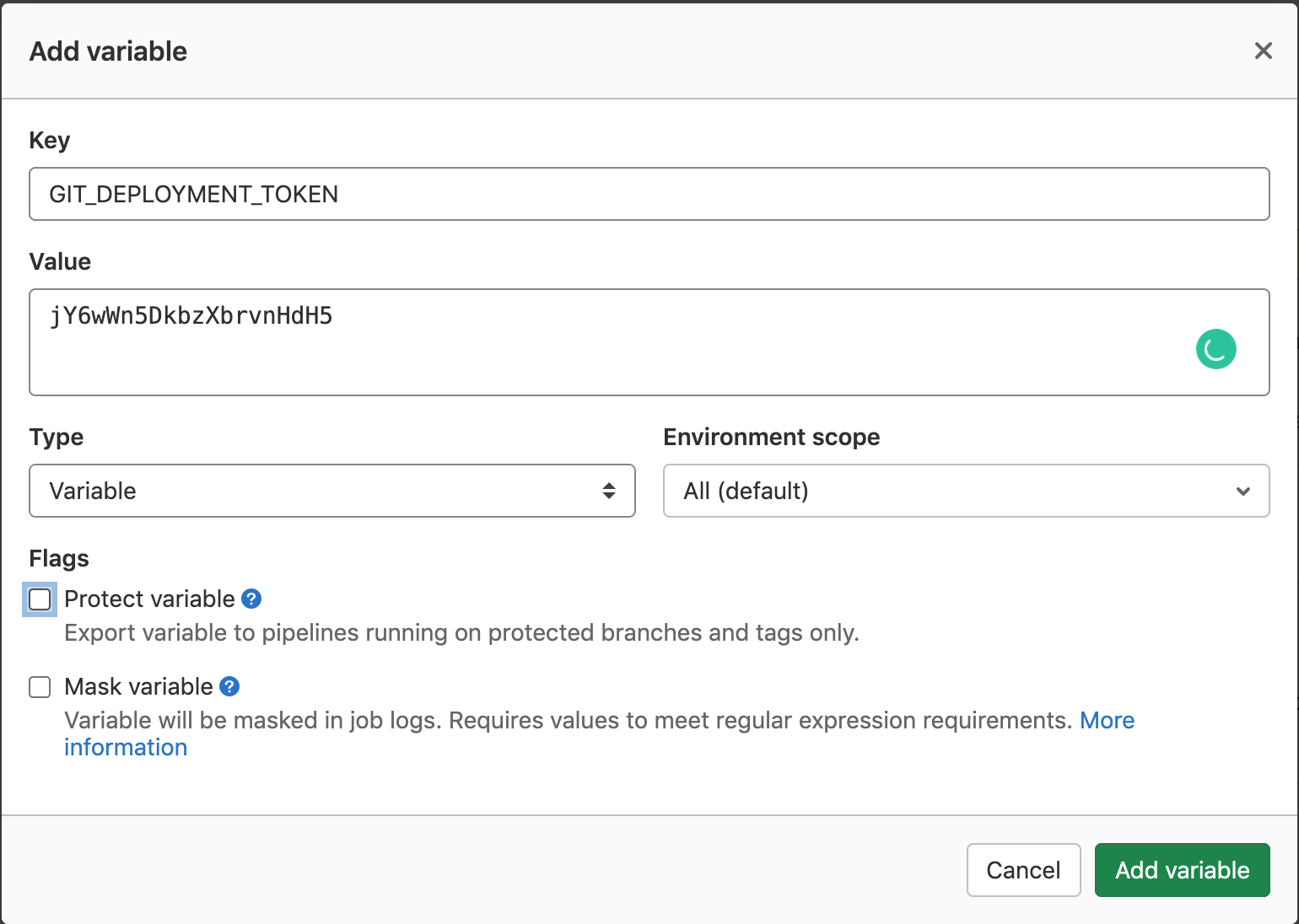
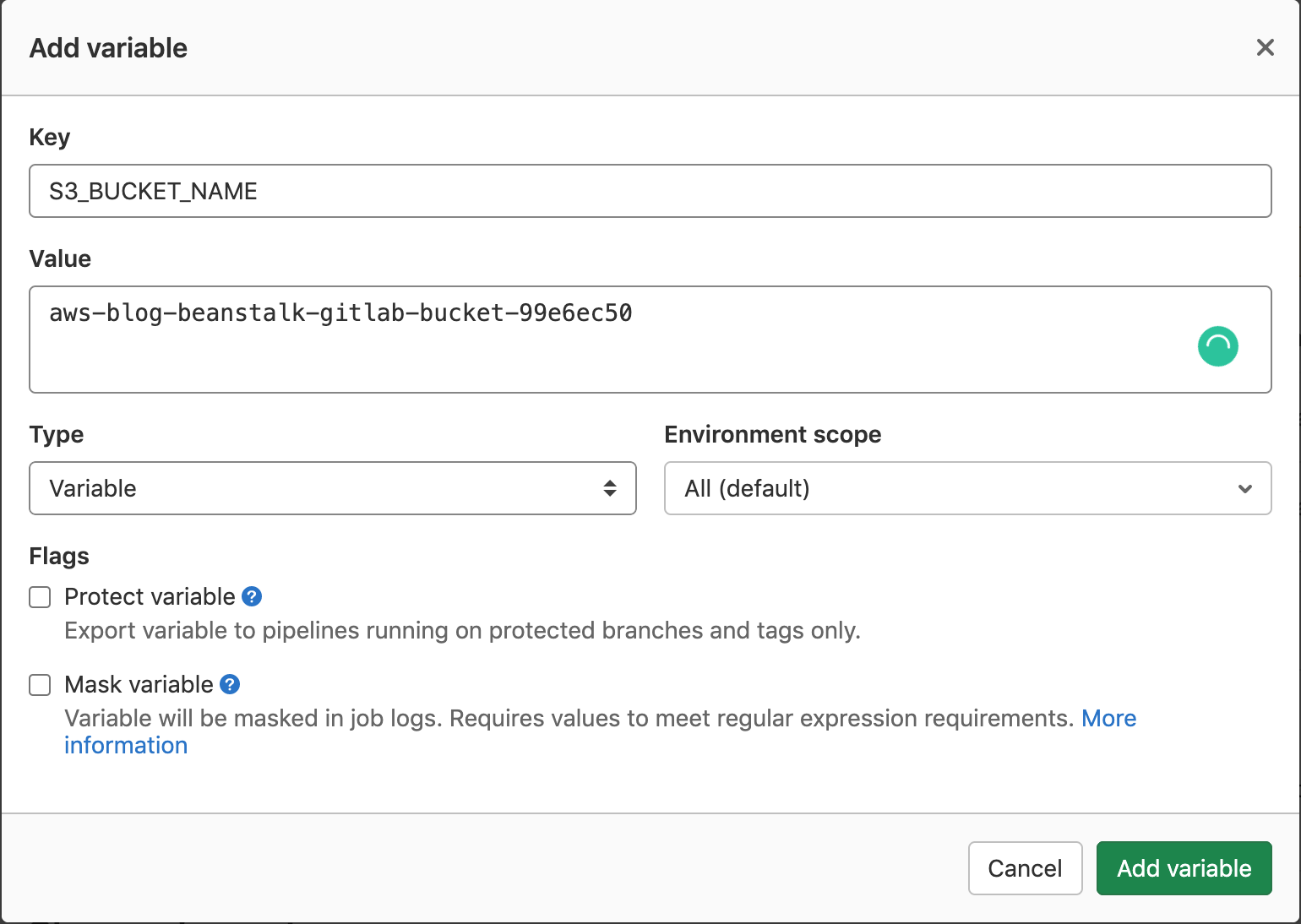
*#- docker login -u gitlab+deploy-token-1 -p "mPi9hmYWM6vsKNRne3T5" $CI\_REGISTRY*- docker login -u $GIT\_DEPLOYMENT\_USER -p $GIT\_DEPLOYMENT\_TOKEN $CI\_REGISTRY  
- cat /root/.docker/config.json  
- echo $(cat /root/.docker/config.json | jq 'del(.HttpHeaders)') > dockercfg.json  
- cat dockercfg.json  
- aws s3 cp dockercfg.json s3://$S3\_BUCKET\_NAME/.dockercfg  
- aws s3 ls s3://$S3\_BUCKET\_NAME/.dockercfg

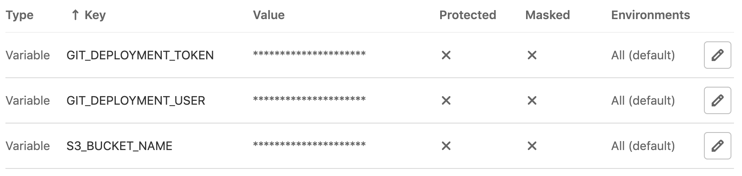
**Adding CI/CD variables in the Gitlab console**

Our “.gitlab-ci.yml” file accesses some custom variables as GitLab CI/CD environment variables. These variables are part of the environment in which pipelines and jobs run. We can setup custom variables in the [Gitlab console](https://docs.gitlab.com/ee/ci/variables/#create-a-custom-variable-in-the-ui). Go to your project once you sign in to Gitlab console, then click on Settings -> CI/CD and expands “Variables” section. Our pipeline needs three variables to be added in this section. Make sure you “unchecked” “Protected variable” check box as shown in the below screen shots.

* Use the username value for “GIT\_DEPLOYMENT\_USER” that was generated as part of creating the deployment token.
* Use the token value for GIT\_DEPLOYMENT\_TOKEN” that was generated as part of creating the deployment token.
* Use the bucket name for S3\_BUCKET\_NAME that was created during second AWS CloudFormation script. You can find the S3\_BUCKET\_NAME in the “outputs” section of the AWS cloud formation console.

GIT\_DEPLOYMENT\_USER: "$GIT\_DEPLOYMENT\_USER"  
GIT\_DEPLOYMENT\_TOKEN: "$GIT\_DEPLOYMENT\_TOKEN"  
S3\_BUCKET\_NAME: "$S3\_BUCKET\_NAME"

Once all these variables are created, you can see the below variables.  


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

Now let’s create a AWS Beanstalk application and a AWS Beanstalk environment to deploy a sample application. When we first initialize the AWS Beanstalk application, it will use the default sample application provided by AWS.

You can use this [downloadable](https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-beanstalk-gitlab/cloudformations/step3-setup-beanstalk.yaml) AWS CloudFormation template to set up the previous components. To launch directly through the console, choose Launch Stack.

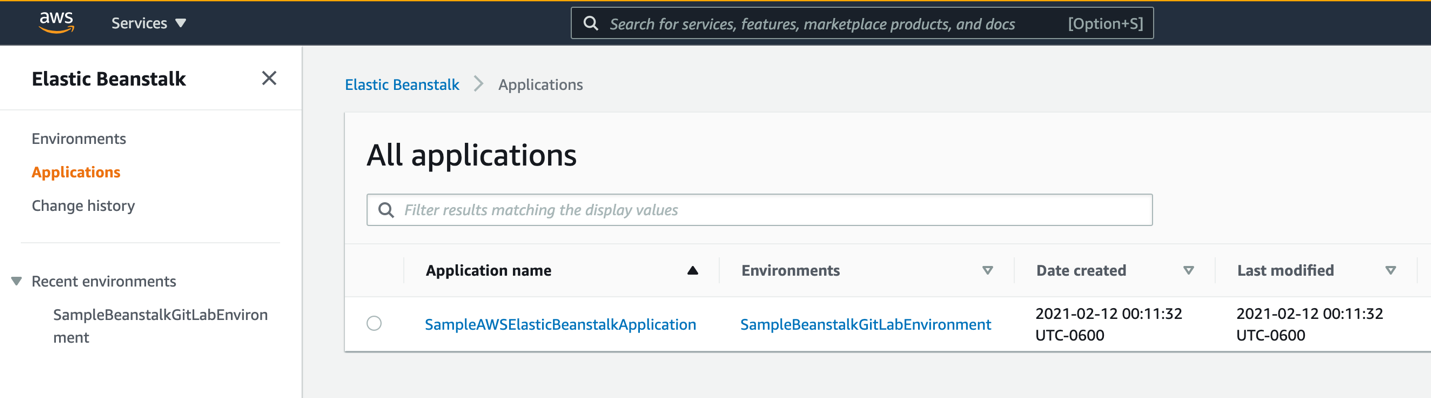
[launch_stack](https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/new?stackName=Step3-Bean-GitLab-blog&templateURL=https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-beanstalk-gitlab/cloudformations/step3-setup-beanstalk.yaml)

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.

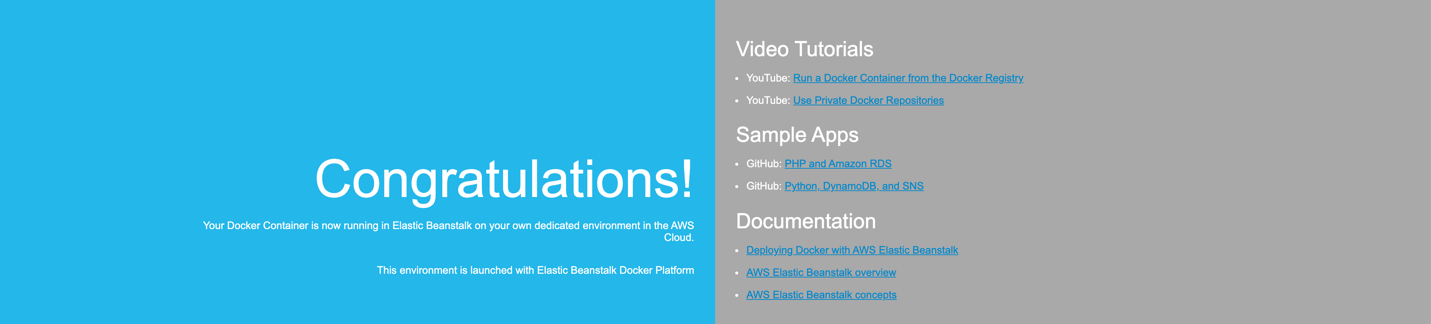
Make a note of the output, because you use this information in the next step. You can [view the stack outputs](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-console-view-stack-data-resources.html) on the AWS Management Console or by using the following AWS CLI command:

$ 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.



**Clone the sample-nodejs-app application to your local machine**

Now we will clone the **sample-nodejs-app** application to our local machines and update some changes and will push to the Gitlab repo. Sign in to GitLab and go to “Your projects” and select the “sample-nodejs-app.

On your local machine (showing the below steps on mac), create a directory where you want to download your repository.  
  
> mkdir -p ~/test/  
> cd ~/test/  
> git clone [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) (Make sure you have port 22 is opened to this instance from your network). Update the host DNS with the EC2 instance that was created in Step2 cloud formation script. This is your GitLab instance.  
  
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.

* cd ~/test/sample-nodejs-app/
* 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 80.

“  
hello, AWS

My app port number is : 80.

“

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

* curl <http://localhost:80/emps>
* curl <http://localhost:80/emps/2>

**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.

* cd ~/test/sample-nodejs-app/
* tsc

**Configuration files**

**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 in the GitLab Private registry from which you’re downloading the 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 private registry. Go to “Packages & Registries” -> Container Registry section to see the images that will get created.

Since we do not know the “image” name before it gets generated, 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 script using “.gitlab-ci.yml” file.

***{*** "AWSEBDockerrunVersion": "1",  
 "Authentication": ***{*** "Bucket": "$s3\_bucket\_name",  
 "Key": ".dockercfg"  
 ***}***,  
 "Ports": ***[  
 {*** "ContainerPort": "80"  
 ***}  
 ]***,  
 "Image": ***{*** "Name": "$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 gets generated in the same “.gitlab-ci.yml” file.

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”. We will see some of the important sections in this file.

**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:  
 - docker login -u $GIT\_USER -p $DOCKER\_PASSWORD $CI\_REGISTRY  
 - docker build -t git.beanblog-aws1.com/root/sample-nodejs-app .  
 - docker push git.beanblog-aws1.com/root/sample-nodejs-app

**deploy-dev:**In this stage, we will install the necessary packages on the Gitlab runner’s 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

**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](https://s3.amazonaws.com/aws-bigdata-blog/artifacts/awsblog-beanstalk-gitlab/shell/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  
}***

When the Amazon Elastic Beanstalk creates the EC2 instances to run the application, we need to download the SSL cert file from Amazon S3 bucket. To download this file whenever the instance is created, we can use the [platform hooks](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/platforms-linux-extend.html) to extend Amazon Beanstalk environment. We will have the below shell script to be executed during “prebuild” phase of the provisioning stage. This shell script(01\_pre\_deploy\_script.sh) is already in your repository under “.platform/hooks/prebuild” directory.

*#!/bin/bash*

S3\_BUCKET\_NAME=$(/opt/elasticbeanstalk/bin/get-config environment -k AWS\_S3\_BUCKET\_VALUE)

echo "copying myca.crt file from s3 bucket."

echo "Bucket name exported from the environment variable that was set in the Beanstalk environment configuration through Cloudforamtion script :::: ${S3\_BUCKET\_NAME}"

*#- Namespace: "aws:elasticbeanstalk:application:environment"*

*# OptionName: AWS\_S3\_BUCKET\_VALUE*

*# Value:*

*# Fn::ImportValue: "exp-s3-bucket"*

sudo mkdir -p /etc/docker/certs.d

sudo aws s3 cp s3://${S3\_BUCKET\_NAME}/myca.crt /etc/docker/certs.d/ca.crt

sudo cp -r -p /etc/docker/certs.d/ca.crt /etc/pki/tls/certs/

ls -ltra /etc/docker/certs.d/ca.crt

ls -ltra /etc/pki/tls/certs/ca.crt

sudo service docker stop

sudo service docker start

sudo service docker status

The final source bundle that will be used to create our sample application will have the below files. Note that we are *not* including the actual application source files in the final source bundle, instead we are instructing Amazon Elastic Beanstalk to use the “Dockerrun.aws.json” file to retrieve the application’s docker image from the Gitlab’s private registry.  
  
.platform  
└── hooks  
 └── prebuild  
 └── 01\_pre\_deploy\_script.sh  
Dockerrun.aws.json

**Push the sample-nodejs-app application to Gitlab repository.**

Now login to your local machine and navigate to the directory where you copied the sample application. Since we already compiled the code above, it will create the “dist” directory and it will trigger changes to the source code.

* cd ~/test/sample-nodejs-app
* git add .
* git commit -m "compiled application changes"
* git push -u origin master

Once the code push is completed, sign in to the “Gitlab” console and select “sample-nodejs-app” and go to CI/CD->Pipelines. You will see the pipeline gets executed and it will deploy the new version to Elastic Beanstalk environment.

**Cleanup**

To clean up the environment that was created, follow the below steps.

Go to Amazon S3 console and look for the bucket with the name prefix: “aws-blog-beanstalk-gitlab-bucket”. Open the bucket and remove the files in this bucket.

Caution: Please make sure you are deleting the contents from the correct S3 bucket.

Then go to Amazon Cloudformation console and delete “Step3-Bean-GitLab-blog”, “Step2-Bean-GitLab-blog” and “Step1-Bean-GitLab-blog” stacks one after the other.

**Conclusion**