What Does CI/CD Try To Solve?

CI/CD is a term that is often heard alongside other terms like DevOps, Agile, Scrum and Kanban, automation, and others. Sometimes, it's considered to be just part of the workflow without really understanding what it is or why it was adopted. Taking CI/CD for granted is common for young DevOps engineers who might have not seen the "traditional" way of software release cycles and, hence, cannot appreciate CI/CD.

CI/CD stands for Continuous Integration/Continous Delivery and/or Deployment. A team that does not implement CI/CD will have to pass through the following stages when it creates a new software product:

1. The product manager (representing the client's interests) provides the needed features that the product should have and the behavior it should follow. The documentation must be as thorough and specific as possible.
2. The developers with the business analysts start working on the application by writing codes, running unit tests, and committing the results to a version control system (for example, git).
3. Once the development phase is done, the project is moved to QA. Several tests are run against the product, like User Acceptance Tests, Integration Tests, performance tests among others. During that period, there should be no changes to the code base until the QA phase is complete. If there should be any bug, they're passed back to the developers to fix them, and hands the product back to QA.
4. Once QA is done, code is deployed to production by the operations team.

There is a number of shortcomings for the above workflow:

- First, it takes so long from the time the product manager makes her request until the product is ready for production.
- It's harder for developers to address bugs in code that has been written since quite a long time like a month or more ago. Remember, bugs are only spotted after the development phase is over and QA phase starts.
- When there's an *urgent* code change like a serious bug that needs a hotfix, the QA phase tends to be shortened due to the need to deploy as fast as possible.
- Since there's little collaboration between different teams, people start pointing fingers and blaming each other when bugs occur. Everybody starts caring only about his/her own part of the project and lose sight of the common goal.

CI/CD solves the above problems by introducing automation. Each change in the code once pushed to the version control system, gets tested, and further deployed to staging/UAT environments for even further testing before deploying it to production for users to consume. Automation ensures that the entire process is fast, reliable, repeatable, and much less error-prone.

So, What Is CI/CD Anyway?

Complete books have been written about this subject. The how, why, and when to implement it in your infrastructure. However, we always prefer less theory and more practice whenever possible. Having said that, the following is a brief description of the automated steps that should be executed once a code change is committed:

1. **Continuous Integration (CI):** the first step does not include QA. In other words, it does not focus on whether the code provides the features requested by the client. Rather, it ensures that the code is of good quality. Through unit tests, integration tests, the developers are quickly notified of any issues in the code quality. We can further augment the tests with code coverage and static analysis for even more quality assurance.
2. **User Acceptance Testing:** this is the first part of the CD process. In this phase, automated tests are performed on the code to ensure that it meets the client's expectations. For example, a web application may work without throwing any errors, but the client wants the visitor to come across a landing page where an offer is presented before navigating to the main page. The current code brings the visitor directly to the main page, which is a deviation from the client's demands. This sort of issue is pointed out by the UAT testing. In non-CD environments, this is the job of human QA testers.
3. **Deployment:** this is the second part of the CD process. It involves making changes to the servers/pods/containers that will host the application so that they reflect the updated version. This should be done in an automated way, preferably through a configuration management tool like Ansible, Chef, or Puppet.

And What Is A Pipeline?

A pipeline is a fancy term for a very simple concept; when you have a number of scripts that need to be executed in a certain order to achieve a common goal, they're collectively referred to as a "pipeline". For example, in Jenkins, a pipeline may consist of one or more *stages* that must all complete for a build to be successful. Using stages helps visualize the entire process, learn how long each stage is taking, and determine where exactly the build is failing.

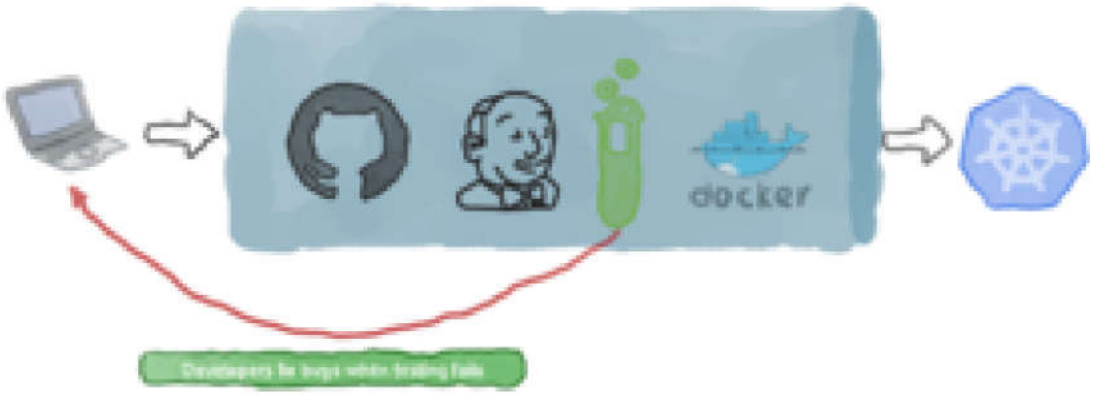
LAB: Create A Pipeline For A Golang App

In this lab, we are building a continuous delivery (CD) pipeline. We are using a very simple application written in Go. For the sake of simplicity, we are going to run only one type of test against the code. The prerequisites for this lab are as follows:

- A running Jenkins instance. This could be a cloud instance, a virtual machine, a bare metal one, or a docker container. It must be publicly accessible from the internet so that the repository can connect to Jenkins through web-hooks.

- Image registry: you can use Docker Registry, a cloud-based offering like ECR or GCR, or even a custom registry.
- An account on GitHub. Although we use GitHub in this example, the procedure can work equally with other repositories like Bitbucket with minor changes.

The pipeline can be depicted as follows:



Step 01: The Application Files

Our sample application will respond with ‘Hello World’ to any GET request. Create a new file called main.go and add the following lines:

```
package main

import (
   "log"
   "net/http"
)

type Server struct{}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
   w.WriteHeader(http.StatusOK)
   w.Header().Set("Content-Type", "application/json")
   w.Write([]byte(`{"message": "hello world"}`))
}

func main() {
   s := &Server{}
   http.Handle("/", s)
   log.Fatal(http.ListenAndServe(":8080", nil))
}
```

Since we are building a CD pipeline, we should have some tests in place. Our code is so simple that it only needs one test case; ensuring that we receive the correct string when we hit the root URL. Create a new file called main_test.go in the same directory and add the following lines:

```
package main

import (
   "log"
   "net/http"
)

type Server struct{}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
   w.WriteHeader(http.StatusOK)
   w.Header().Set("Content-Type", "application/json")
   w.Write([]byte(`{"message": "hello world"}`))
}

func main() {
   s := &Server{}
   http.Handle("/", s)
   log.Fatal(http.ListenAndServe(":8080", nil))
}
```

We also have a few other files that help us deploy the application, named:

The Dockerfile

This is where we package our application:

```
FROM golang:alpine AS build-env
RUN mkdir /go/src/app && apk update && apk add git
ADD main.go /go/src/app/
WORKDIR /go/src/app
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -ldflags '-extldflags "-stati

FROM scratch
WORKDIR /app
COPY --from=build-env /go/src/app/app .
ENTRYPOINT [ "./app" ]
```

The Dockerfile is a multistage one to keep the image size as small as possible. It starts with a build image based on golang:alpine. The resulting binary is used in the second image, which is just a scratch one. A scratch image contains no dependencies or libraries, just the binary file that starts the application.

The Service

Since we are using Kubernetes as the platform on which we host this application, we need at least a service and a deployment. Our service.yml file looks like this:

```
apiVersion: v1
kind: Service
metadata:
  name: hello-svc
spec:
  selector:
    role: app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 32000
  type: NodePort
```

There's nothing special about this definition. Just a Service that uses NodePort as its type. It will listen on port 32000 on the IP address of any of the cluster nodes. The incoming connection is relayed to the pod on port 8080. For internal communications, the service listens on port 80.

The deployment

The application itself, once dockerized, can be deployed to Kubernetes through a Deployment resource. The deployment.yml file looks as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-deployment
  labels:
    role: app
spec:
  replicas: 2
  selector:
    matchLabels:
      role: app
  template:
    metadata:
      labels:
        role: app
    spec:
      containers:
      - name: app
        image: ""
        resources:
          requests:
            cpu: 10m
```

The most interesting thing about this deployment definition is the image part. Instead of hardcoding the image name and tag, we are using a variable. Later on, we shall see how we can use this definition as a template for Ansible and substitute the image name (and any other parameters of the deployment) through the command line arguments.

The playbook

In this lab, we are using Ansible as our deployment tool. There are many other ways to deploy Kubernetes resources including Helm Charts, but I thought Ansible is a much easier option. Ansible uses playbooks to organize its instructions. Our playbook.yml file looks as follows:

```
- hosts: localhost
  tasks:
  - name: Deploy the service
    k8s:
      state: present
      definition: ""
      validate_certs: no
      namespace: default
  - name: Deploy the application
    k8s:
      state: present
      validate_certs: no
      namespace: default
      definition: ""
```

Ansible already includes the k8s module for handling communication with the Kubernetes API server. So, we don't need kubectl installed but we do need a valid kubeconfig file for connecting to the cluster (more on that later). Let's have a quick discussion about the important parts of this playbook:

- The playbook is used to deploy the service and the deployment of resources to the cluster.
- Since we need to inject data into the definition file on the fly while execution, we need to use our definition files as templates where variables can be supplied from outside.
- For that purpose, Ansible features the lookup function, where you can pass a valid YAML file as a template. Ansible supports many ways of injecting variables into templates. In this specific lab, we are using the command-line method.

Step 02: Install Jenkins, Ansible, And Docker

Let's install Ansible and use it to automatically deploy a Jenkins server and Docker runtime environment. We also need to install the openshift Python module to enable Ansible connect with Kubernetes.

Ansible's installation is very easy; just install Python and use pip to install Ansible:

1. Log in to the Jenkins instance
2. Install Python 3, Ansible, and the openshift module:

```
sudo apt update && sudo apt install -y python3 && sudo apt install -y pyt
```

3. By default, pip installs binaries under a hidden directory in the user's home folder. We need to add this directory to the $PATH variable so that we can easily call the command:

```
echo "export PATH=$PATH:~/.local/bin" >> ~/.bashrc && . ~/.bashrc
```

4. Install the Ansible role necessary for deploying a Jenkins instance:

```
ansible-galaxy install geerlingguy.jenkins
```

5. Install the Docker role:

```
ansible-galaxy install geerlingguy.docker
```

6. Create a playbook.yaml file and add the following lines:

```
- hosts: localhost
  become: yes
  vars:
    jenkins_hostname: 35.238.224.64
    docker_users:
    - jenkins
  roles:
    - role: geerlingguy.jenkins
    - role: geerlingguy.docker
```

7. Run the playbook through the following command: ansible-playbook playbook.yaml. Notice that we're using the public IP address of the instance as the hostname that Jenkins will use. If you are using DNS, you may need to replace this with the DNS name of the instance. Also, notice that you must enable port 8080 on the firewall (if any) before running the playbook.

8. In few minutes, Jenkins should be installed. You can check by navigating to the IP address (or the DNS name) of the machine and specifying port 8080:

9. Click on the Login link and supply "admin" as the username and "admin" as the password. Note that those are the default credentials set by the Ansible role that we used. You can (and should) change those defaults when using Jenkins in production environments. This can be done by setting the role variables. You can refer to the role official page.

10. The last thing you need to do is install the following plugins that will be used in our lab:

- git
- pipeline
- CloudBees Docker Build and Publish
- GitHub

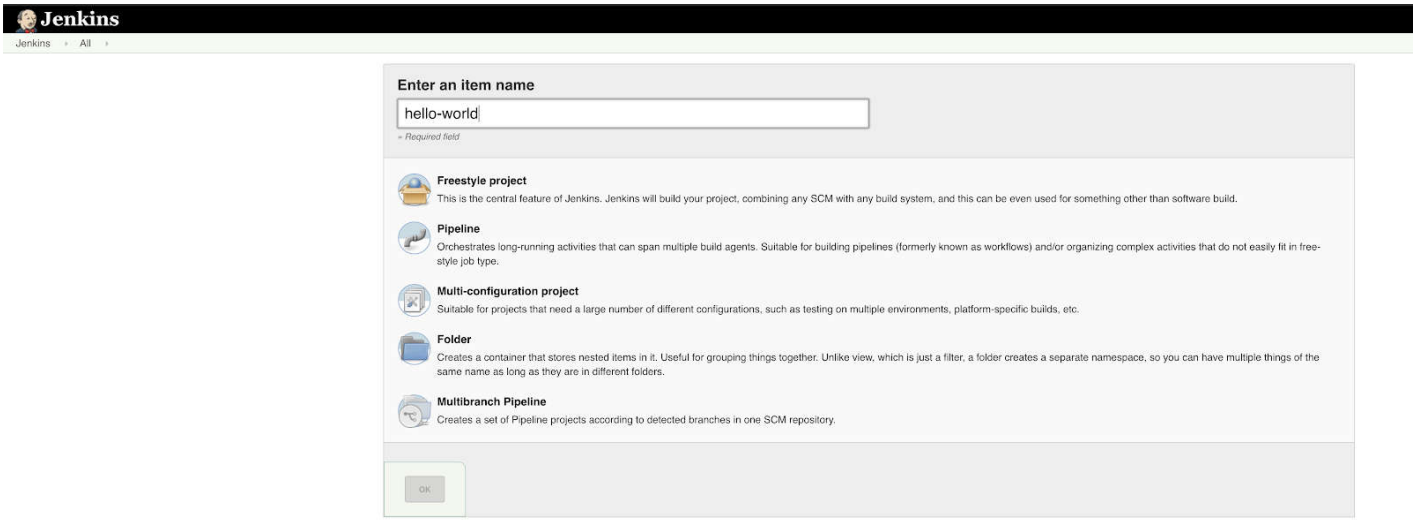
Step 03: Configuring Jenkins User To Connect To The Cluster

As mentioned, this lab assumes that you already have a Kubernetes cluster up and running. To enable Jenkins to connect to this cluster, we need to add the necessary kubeconfig file. In this specific lab, we are using a Kubernetes cluster that's hosted on Google Cloud so we're using the gcloud command. Your specific case may be different. But in all cases, we must copy the kubeconfig file to the Jenkins's user directory as follows:

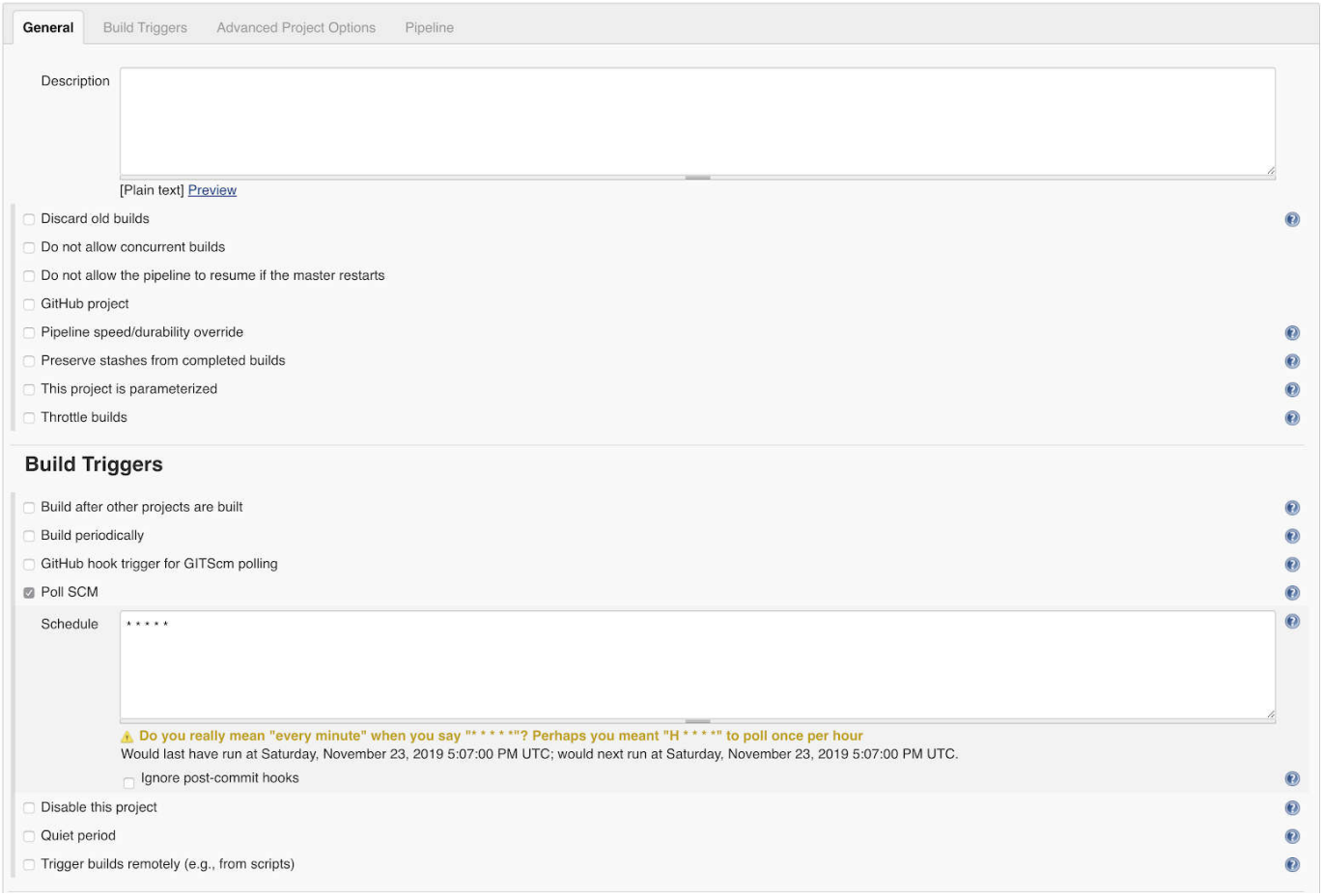
```
$ sudo cp ~/.kube/config ~jenkins/.kube/
$ sudo chown -R jenkins: ~jenkins/.kube/
```

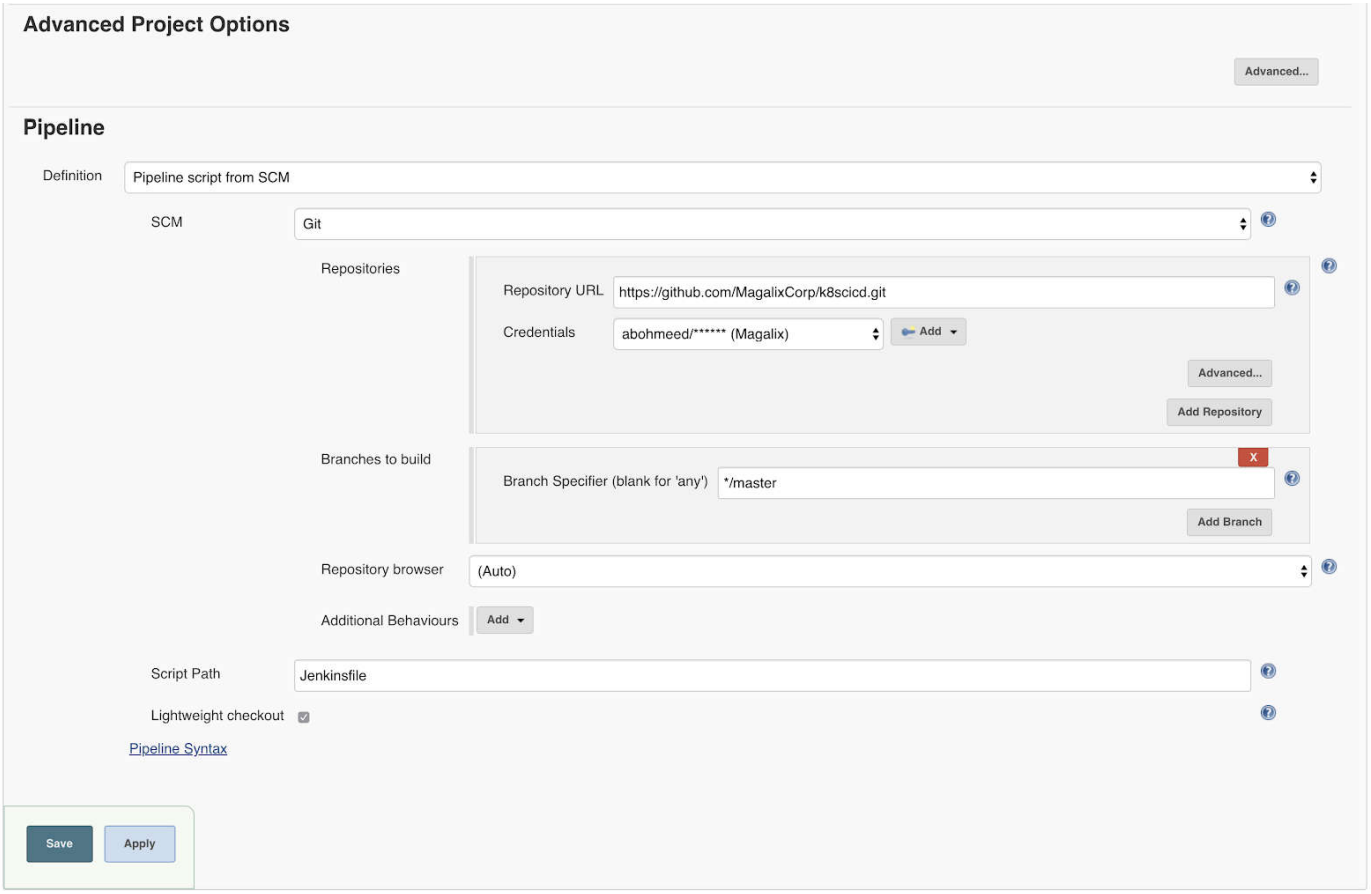
Note that the account that you'll use here must have the necessary permissions to create and manage Deployments and Services.

Step 04: Create The Jenkins Pipeline Job



Create a new Jenkins job and select the Pipeline type. The job settings should look as follows:



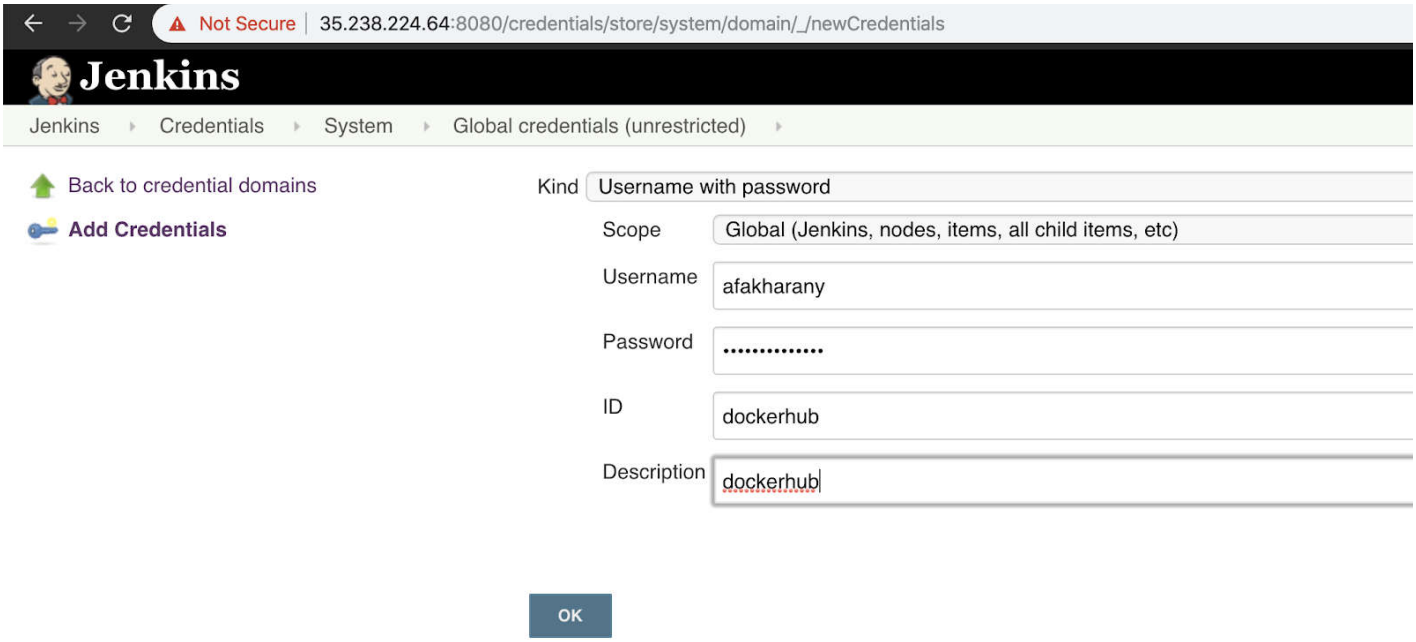


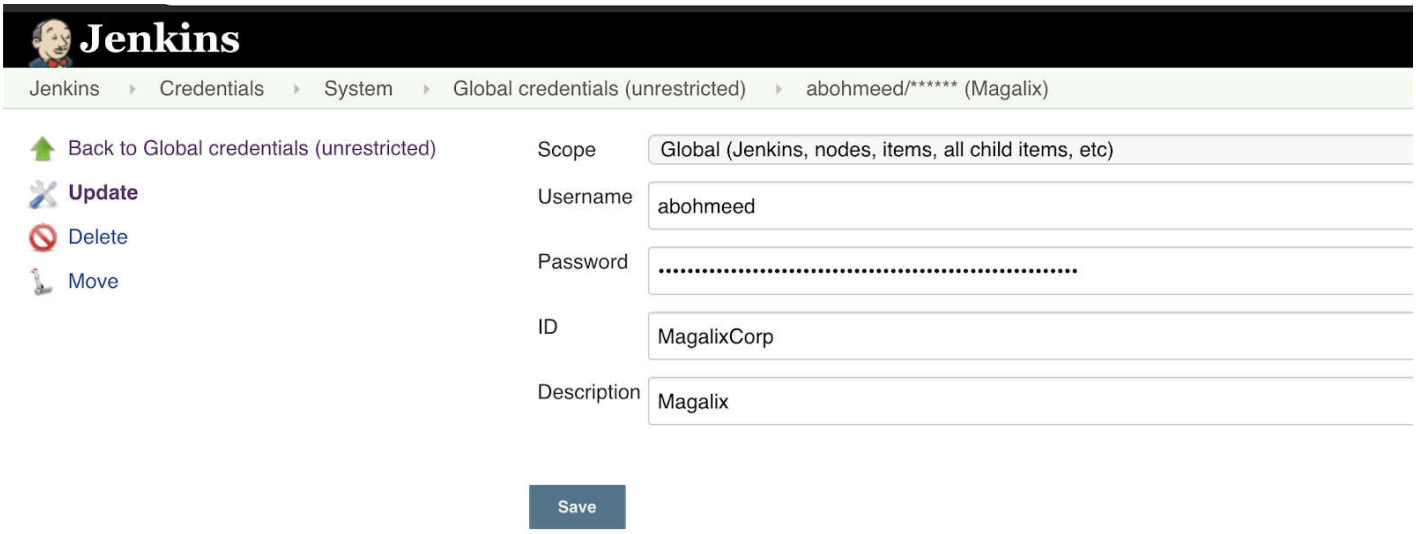
The settings that we changed are:

- We used the Poll SCM as the build trigger; setting this option instructs Jenkins to check the Git repository on a periodic basis (every minute as indicated by * * * * *). If the repo has changed since the last poll, the job is triggered.
- In the pipeline itself, we specified the repository URL and the credentials. The branch is master.
- In this lab, we are adding all the job's code in a Jenkinsfile that is stored in the same repository as the code. The Jenkinsfile is discussed later in this article.

Step 05: Configure Jenkins Credentials For GitHub and Docker Hub

Go to /credentials/store/system/domain/_/newCredentials and add the credentials to both targets. Make sure that you give a meaningful ID and description to each because you'll reference them later:





Step 06: Create The JenkinsFile

The Jenkinsfile is what instructs Jenkins about how to build, test, dockerize, publish, and deliver our application. Our Jenkinsfile looks like this:

```
pipeline {
   agent any
   environment {
       registry = "magalixcorp/k8scicd"
       GOCACHE = "/tmp"
   }
   stages {
       stage('Build') {
           agent {
               docker {
                   image 'golang'
               }
           }
           steps {
               // Create our project directory.
               sh 'cd ${GOPATH}/src'
               sh 'mkdir -p ${GOPATH}/src/hello-world'
               // Copy all files in our Jenkins workspace to our project directory.
               sh 'cp -r ${WORKSPACE}/* ${GOPATH}/src/hello-world'
               // Build the app.
               sh 'go build'
           }
       }
       stage('Test') {
           agent {
               docker {
                   image 'golang'
               }
           }
           steps {
               // Create our project directory.
               sh 'cd ${GOPATH}/src'
               sh 'mkdir -p ${GOPATH}/src/hello-world'
               // Copy all files in our Jenkins workspace to our project directory.
               sh 'cp -r ${WORKSPACE}/* ${GOPATH}/src/hello-world'
               // Remove cached test results.
               sh 'go clean -cache'
               // Run Unit Tests.
               sh 'go test ./... -v -short'
           }
       }
       stage('Publish') {
           environment {
               registryCredential = 'dockerhub'
           }
           steps{
               script {
                   def appimage = docker.build registry + ":$BUILD_NUMBER"
                   docker.withRegistry( '', registryCredential ) {
```

```
                        appimage.push()
                        appimage.push('latest')
                    }
                }
            }
        }
        stage ('Deploy') {
            steps {
                script{
                    def image_id = registry + ":$BUILD_NUMBER"
                    sh "ansible-playbook  playbook.yml --extra-vars \"image_id=${imag
                }
            }
        }
    }
}
```

The file is easier than it looks. Basically, the pipeline contains four stages:

1. Build is where we build the Go binary and ensure that nothing is wrong in the build process.
2. Test is where we apply a simple UAT test to ensure that the application works as expected.
3. Publish, where the Docker image is built and pushed to the registry. After that, any environment can make use of it.
4. Deploy, this is the final step where Ansible is invoked to contact Kubernetes and apply the definition files.

Now, let's discuss the important parts of this Jenkinsfile:

- The first two stages are largely similar. Both of them use the golang Docker image to build/test the application. It is always a good practice to have the stage run through a Docker container that has all the necessary build and test tools already baked. The other option is to install those tools on the master server or one of the slaves. Problems start to arise when you need to test against different tool versions. For example, maybe we want to build and test our code using Go 1.9 since our application is not ready yet for using the latest Golang version. Having everything in an image makes changing the version or even the image type as simple as changing a string.
- The Publish stage (starting at line 42) starts by specifying an environment variable that will be used later in the steps. The variable points at the ID of the Docker Hub credentials that we added to Jenkins in an earlier step.
- Line 48: we use the docker plugin to build the image. It uses the Dockerfile in our registry by default and adds the build number as the image tag. Later on, this will be of much importance when you need to determine which Jenkins build was the source of the currently running container.

- Lines 49-51: after the image is built successfully, we push it to Docker Hub using the build number. Additionally, we add the “latest” tag to the image (a second tag) so that we allow users to pull the image without specifying the build number, should they need to.
- Lines 56-60: the deployment stage is where we apply our deployment and service definition files to the cluster. We invoke Ansible using the playbook that we discussed earlier. Note that we are passing the image_id as a command-line variable. This value is automatically substituted for the image name in the deployment file.

Testing Our CD Pipeline

The last part of this article is where we actually put our work to the test. We are going to commit our code to GitHub and ensure that our code moves through the pipeline until it reaches the cluster:

1. Add our files: git add *
2. Commit our changes: git commit -m "Initial commit"
3. Push to GitHub: git push
4. On Jenkins, we can either wait for the job to get triggered automatically, or we can just click on “Build Now”.
5. If the job succeeds, we can examine our deployed application using the following commands:

Get the node IP address:

```
kubectl get nodes -o wide
NAME                                          STATUS   ROLES    AGE   VERSION          IN
gke-security-lab-default-pool-46f98c95-qsdj   Ready       7d    v1.13.11-gke.9   10.128.0
```

Now, let’s initiate an HTTP request to our app:

```
$ curl 35.193.211.74:32000
{"message": "hello world"}
```

OK, we can see that our application is working correctly. Let’s make an intentional error in our code and ensure that the pipeline will not ship faulty code to the target environment:

Change the message that should be displayed to be “Hello World!”, notice that we capitalize the first letter of each word and added an exclamation mark at the end. Since our client may not want the message to be displayed that way, the pipeline should stop at the Test stage.

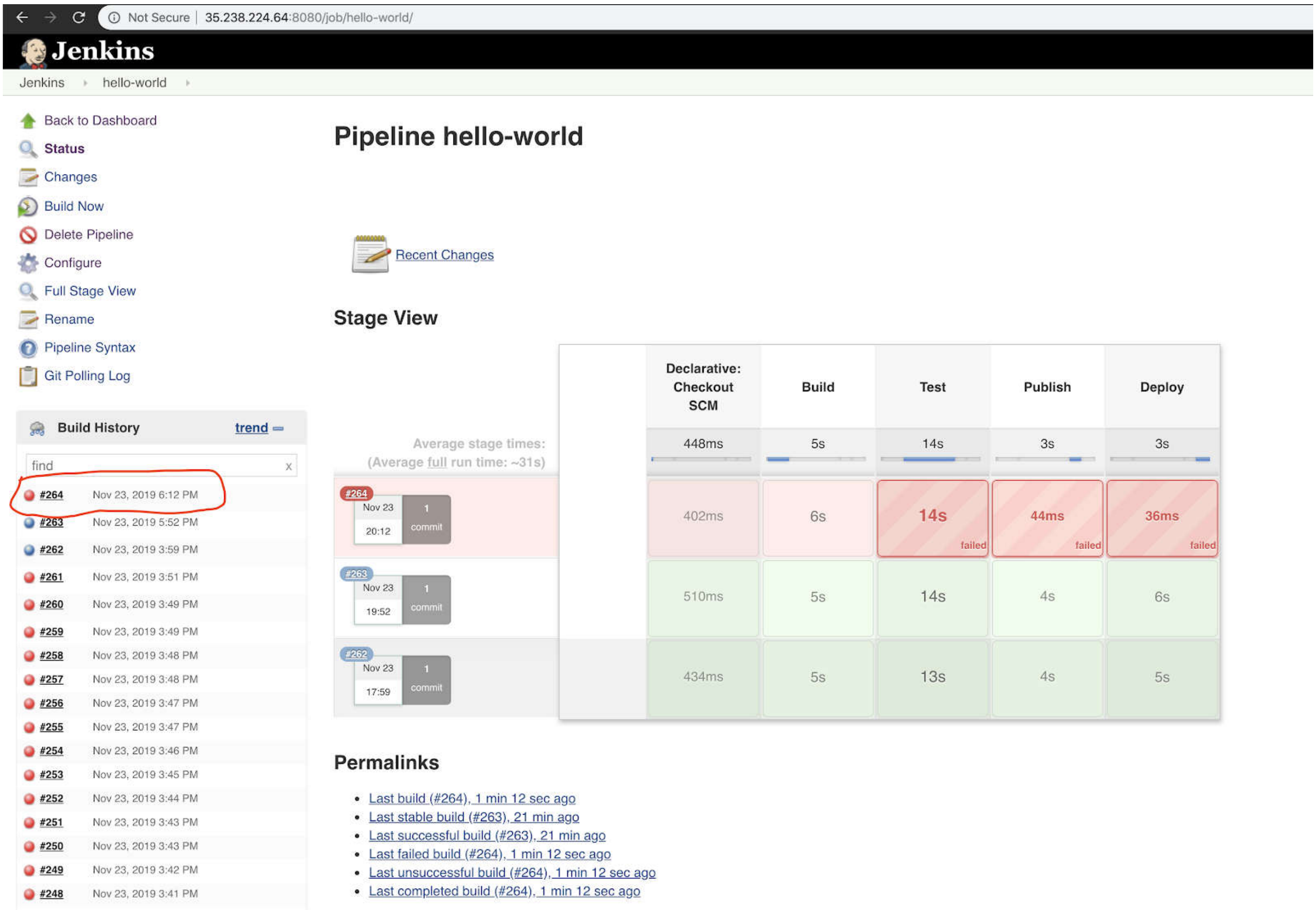
First, let’s make the change. The main.go file now should look like this:

```
package main

import (
   "log"
   "net/http"
)

type Server struct{}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
   w.WriteHeader(http.StatusOK)
   w.Header().Set("Content-Type", "application/json")
   w.Write([]byte(`{"message": "Hello World!"}`))
}

func main() {
   s := &Server{}
   http.Handle("/", s)
   log.Fatal(http.ListenAndServe(":8080", nil))
}
```

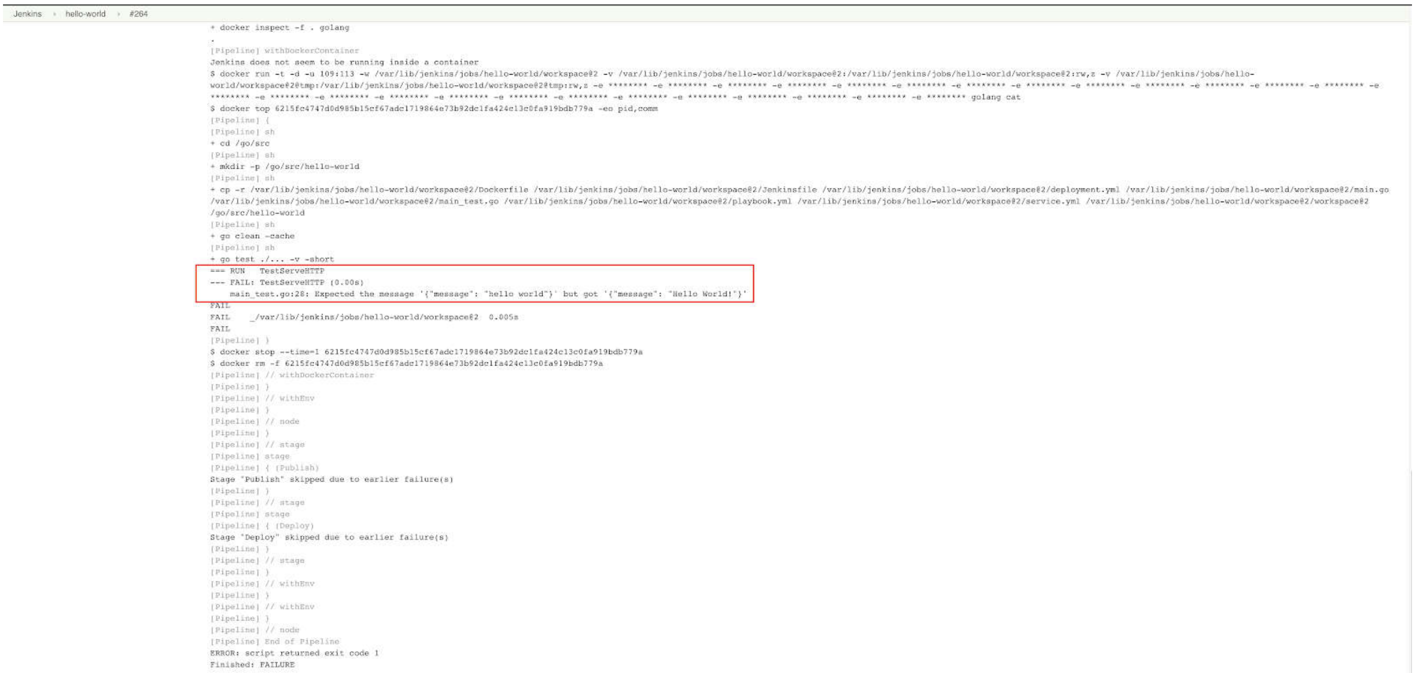
Next, let's commit and push our code:

```
$ git add main.go
$ git commit -m "Changes the greeting message"
[master 24a310e] Changes the greeting message
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 319 bytes | 319.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/MagalixCorp/k8scicd.git
   7954e03..24a310e  master -> master
```

Back to Jenkins, we can see that the last build has failed:



By clicking on the failed job, we can see the reason why it failed:



Our faulty code will never make its way to the target environment.

TL;DR

- CI/CD is an integral part of any modern environment that follows the Agile methodology.

- Through pipelines, you can ensure a smooth transition of code from the version control system to the target environment (testing/staging/production/etc.) while applying all the necessary testing and quality control practices.
- In this article, we had a practical lab where we built a continuous delivery pipeline to deploy a Golang application.
- Through Jenkins, we were able to pull the code from the repository, build and test it using a relevant Docker image.
- Next, we Dockerize and push our application - since it passed our tests - to Docker Hub.
- Finally, we used Ansible to deploy the application to our target environment, which is running Kubernetes.
- Using Jenkins pipelines and Ansible makes it very easy and flexible to change the workflow with very little friction. For example, we can add more tests to the Test stage, we can change the version of Go that's used to build and test the code, and we can use more variables to change other aspects in deployment and service definitions.
- The best part here is that we are using Kubernetes deployments, which ensures that we have zero downtime for the application when we are changing the container image. This is possible because Deployments use the rolling update method by default to terminate and recreate containers one at a time. Only when the new container is up and healthy does the Deployment terminate the old one