**Objective**

At the end of the post, you should be able to build a CI/CD pipeline for a sample spring-boot java application using Github as a version control system, Maven as a build tool, jenkins as an integration server, Docker as a container technology and Kubernetes on Azure as a platform. The end goal is to automate the following process:

* Checkout code.
* Compile code.
* Run test cases.
* Build docker images.
* Push images to docker registry.
* Pull new images from registry.
* Deploy the app on Kubernetes.

**Prerequisites**

In order to follow the steps in this guide, you will need the following:

* [GitHub Account](https://github.com): You need a GitHub account to fork/clone the ToDo app repository.
* [Azure Microsoft Account](https://azure.microsoft.com/en-us/): You need an Azure Microsoft account in order to use Azure.
* [Azure CLI](https://docs.microsoft.com/cli/azure/install-azure-cli?view=azure-cli-latest): You need to install the current version of Azure CLI  in order to create a Kubernetes cluster on Azure.
* [kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/): You need to install the kubectl command-line interface in order to run commands against the Kubernetes cluster.
* [Helm](https://github.com/kubernetes/helm/blob/master/docs/install.md): You need to install Helm and Tiller in order to help you install jenkins.

**Get the Sample Application from Github**

The sample application code is hosted on [Github](https://github.com/MirnaAlaisami/CICD). You can clone/fork the repo to your own GitHub account.

The application consists of 3 main components:

* UI (todoui)
* Main app (todobackend)
* DB (postgresdb)

*todoui* and *todobackend* are Spring Boot apps with the following external configuration possibilities:

todoui:

**backend.host=${BACKEND\_HOST:todobackend}  
backend.port=${BACKEND\_PORT:8090}  
backend.url=http://${backend.host}:${backend.port}**

todobackend:

**spring.datasource.url= jdbc:postgresql://${POSTGRES\_HOST:postgresdb}:5432/mydb**

The application directory contains the *Dockerfiles* that are used to containerize the app components, the *yaml files* that are used to deploy the app on Kubernetes and the *jenkinsfile* that contains the CI/CD pipeline code which will be executed by jenkins.

**Create and Configure a Managed Kubernetes Cluster with a Container Registry Instance on Azure**

In order to create a managed K8s cluster on Azure, execute the following steps using the installed Azure CLI:

* Sign in with your account credentials in the browser.

**az login**

* Create a resource group. The following example creates a resource group named CICDResourceGroup in the westeurope location.

**az group create ––name CICDResourceGroup ––location westeurope**

* Create an Azure Container Registry (ACR) instance and give it a name. In the following example acrCICD is used.

**az acr create ––resource-group CICDResourceGroup ––name acrCICD ––sku Basic**

* Get and note the loginServer name, which will be used as the Docker Registry URL later on.

**az acr list ––resource-group CICDResourceGroup ––query “[].{acrLoginServer:loginServer}” ––output table**

* Get your Container Registry credentials and note the Docker registry **username** and **password**, which will be needed later on.

**az acr credential show ––name acrCICD**

* Create a service principal in order to allow an AKS cluster to interact with other Azure resources. Record the resulted values of (**appId**, **password**)

**az ad sp create-for-rbac ––skip-assignment**

This is an example output:

{  
“appId”: “00000000-5ee0-9900-faee-0000b9ee0000”,  
“displayName”: “azure-cli-2019-06-28-10-58-10”,  
“name”: “http://azure-cli-2019-06-28-10-58-10”,  
“password”: “00000000-ed00-5500-0000-aaaa0000dddd”,  
“tenant”: “a33000000-5555-0000e-0000-66ffffddd00”  
}

* Get the ACR resource ID <**acrId**>.

**az acr show ––name acrCICD ––resource-group CICDResourceGroup ––query “id” ––output tsv**

This is an example output:

/subscriptions/11110000-9999-eeee-aaaa-0d0d0333000d/resourceGroups/CICDResourceGroup/providers/Microsoft.ContainerRegistry/registries/acrCICD

* Create a role assignment in order to grant the correct access for the AKS cluster to use images stored in ACR, using the resulted <**appId**> and <**acrId**> in previuos steps.

**az role assignment create ––assignee 00000000-5ee0-9900-faee-0000b9ee0000 ––scope /subscriptions/11110000-9999-eeee-aaaa-0d0d0333000d/resourceGroups/CICDResourceGroup/providers/Microsoft.ContainerRegistry/registries/acrCICD ––role AcrPush**

* Create the Kubernetes cluster, give it a name and grant it access to the ACR, using the <**appId**> and <**password**> created before. You can choose the desired number of nodes in your created cluster (for this guide, one node is enough). Note: this step can take some time to complete.

**az aks create ––name CICDAKSCluster ––resource-group CICDResourceGroup ––node-count 1 ––generate-ssh-keys ––service-principal 00000000-5ee0-9900-faee-0000b9ee0000 ––client-secret 00000000-ed00-5500-0000-aaaa0000dddd**

* Configure kubectl to connect to your Kubernetes cluster.

**az aks get-credentials ––name CICDAKSCluster ––resource-group CICDResourceGroup**

**Deploy Jenkins on Kubernetes**

In order to deploy Jenkins with Helm, execute the following steps:

* Deploy jenkins with Helm using default configuration.

**helm install ––name cicd stable/jenkins**

After executing the previous command, you will get a “Notes” part on your screen that guides you to get the admin-password and access jenkins.

If you want to customize the configuration before deploying jenkins, you can clone the [Jenkins Helm repository](https://github.com/helm/charts) and update the file stable/jenkins/values.yaml, or use helm’s “-set Parameter” for overwriting values.

**Create a GitHub Webhook**

In order to integrate the GitHub repository into Jenkins, a webhook can be used to run the Jenkins build whenever a code commit is made in GitHub. To create the GitHub webhook, execute the following steps:

* In Github, move to your repository and select **“Settings”**.
* On the left-hand side, select **“Webhooks”**.
* Click on the **“Add webhook”** button.
* In the **“Payload URL”,** enter http://<publicIp:8080>/github-webhook/, replacing the <publicIp> with the IP address of the Jenkins server.
* Leave the options with the default selection and click on the **“Add webhook”**.

**Configure Jenkins**

Install the plugins in Jenkins by executing the following steps within the Jenkins dashboard (some plug-ins are already installed) :

* Select **Manage Jenkins** > **Manage Plugins** > **Available**.
* Search for and install the **Github, Kubernetes Cli, Blue Ocean** plug-in.

In order to enable Jenkins to launch pods for running jobs, you have to configure the service account credentials:

* Using jenkins dashboard, navigate to **Manage Jenkins** > **Configure System**> **Cloud** > **Credentials**, then select **“Add”.**
* Choose the type **“Kubernetes Service Account“**.

Using the Docker registry **username** and **password** that you obtained in the previous steps, add your Azure Docker registry credentials as the type **“Username with password“**:

* Using jenkins dashboard, navigate to **Credentials** > **System** > **Global Credentials**> **Add** **Credentials**.
* Choose the type **“Username with password“**.

Now it’s the time to configure the Jenkins slaves which will execute the jobs within the different stages:

* Using jenkins dashboard, navigate to **Manage Jenkins** > **Configure System**> **Cloud** > **Kubernetes Pod Template.**
* Inside the default pod, create the necessary **Container Templates** for *jnlp, Docker, Kubectl* and *Maven* (by giving suitables names and docker images for them).
* Click on **“Save“**.

**Create and Understand the Pipeline**

Before we start with that, let’s understand some basics. Jenkins is composed of two main components, the master that schedules the builds and dispatches the jobs, and the slave(s) that executes the build jobs dispatched from master.

When Jenkins master schedules a new build, it creates a Jenkins slave pod, in which different containers will be created depending on the stages that you define in your pipeline, such that each stage will be executed using a specified container and once the jobs in a stage are done, the container is shutdown. This interaction between jenkins and the kubernetes cluster, in order to start and stop slave/agent containers, is done using the Jenkins Kubernetes plugin, using a kubernetes service account.

Using **Blue Ocean**, the new UI for jenkins, you can easily create a new project and connect it to the Github repository, following these steps:

* Create a **new pipeline** and select **GitHub** as your code store.
* Enter a **Personal Access Token** from GitHub (which can be generated for the first time using GitHub **New Personal Access Token** page), so that Jenkins can access your private repositories.
* Choose your **Github repository** that contains the code.
* Give your pipeline a **name** and **save** the changes.

Let us now have a deep look into the content of the *jenkinsfile*! But before we go more into the details, note that all the variables that are written in capital letters are environment variables which take their values by configuring them globally using Jenkins UI.

*As mentioned at the beginning of the Blogpost, in this way, you can build this pipeline using any private or public Docker registry and a K8s Cluster on any provider that you choose. All you have to do is to configure these variables correctly giving them the suitable values, thus you don’t need to touch the jenkinsfile at all!*

In order to configure any environment variable using jenkins UI, follow these steps:

* Navigate to **Manage Jenkins** > **Configure System**> **Global properties**> **Environment variables**
* Give the **Name** and the **Value** of the variable.
* Click on **Save**.

Now, let’s have a look on the different parts of the jenkinsfile:

* Define some variables that will be used within the stages of your jenkinsfile. Here we are defining the names of the apps, the image tags, the container names and the dockerfile names. Note that the value of the environment variable “REPOSITORY” should be configured in the Jenkins UI as mentioned before, so that you can use your chosen Docker registry later on. The “BUILD\_NUMBER” will take its proper value automatically. Using the values of the Docker registry created in this example, the Docker environment variables will have the following values:
  + REPOSITORY -> acrcicd.azurecr.io

Java



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //Define all variables  def app1\_name = 'todobackend'  def app2\_name = 'todoui'  def app1\_image\_tag = "${env.REPOSITORY}/${app1\_name}:v${env.BUILD\_NUMBER}"  def app2\_image\_tag = "${env.REPOSITORY}/${app2\_name}:v${env.BUILD\_NUMBER}"  def app1\_dockerfile\_name = 'Dockerfile-todobackend'  def app2\_dockerfile\_name = 'Dockerfile-todoui'  def app1\_container\_name = 'todobackend'  def app2\_container\_name = 'todoui' |

* Clone updates from the version control system, in our case here “Github”.



|  |  |
| --- | --- |
| 1  2  3  4 | //Stage 1: Checkout Code from Git      stage('Application Code Checkout from Git') {          checkout scm      } |

* Test the *todobackend* app with H2  in-memory mode using *Maven*. In order to execute the steps within this stage, we use the “maven” container, that we have already defined as a container Template in the “Configure System” section.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //Stage 2: Test Code with Maven/built-in Memory      stage('Test with Maven/H2') {          container('maven'){              dir ("./${app1\_name}") {                    sh ("mvn test -Dspring.profiles.active=dev")             }          }      } |

* Test the *todobackend* app with Postgres DB using *Maven*. In order to execute the steps within this stage, we first deploy a test Postgres DB as a pod in our K8s Cluster using the “kubectl” container. Then we test the connection of the app to it using “maven” as a container. Note that the values of the K8s environment variables should be configured in the Jenkins UI, so that you can connect to your K8s Cluster properly.  
  Using the values of the cluster created in this example,  the K8s environment variables will have the following values:
  + K8s\_CREDENTIALS\_ID -> (You should get this value from the credentials that you created in the “Configure Jenkins” section)
  + K8s\_SERVER\_URL -> https://cicdaksclu-cicdresourcegrou-000000-00000000.hcp.westeurope.azmk8s.io:443
  + K8s\_CONTEXT\_NAME -> CICDAKSCluster
  + CLUSTER\_NAME -> CICDAKSCluster

For the other DB Environment variables, consider the following values:

* + DB\_NAME -> mydb
  + DB\_USERNAME -> user
  + DB\_PASSWORD -> password



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //Stage 3: Test Code with Maven/DB      stage('Test with Maven/PSQL') {          container('kubectl'){              withKubeConfig([credentialsId: env.K8s\_CREDENTIALS\_ID,              serverUrl: env.K8s\_SERVER\_URL,              contextName: env.K8s\_CONTEXT\_NAME,              clusterName: env.K8s\_CLUSTER\_NAME]){                    sh("kubectl apply -f postgres\_test.yml")              }            }            container('maven'){              dir ("./${app1\_name}") {                  sh ("mvn test -Dspring.profiles.active=prod -Dspring.datasource.url=jdbc:postgresql://${env.PSQL\_TEST}/${env.DB\_NAME} -Dspring.datasource.username=${env.DB\_USERNAME} -Dspring.datasource.password=${env.DB\_PASSWORD}")              }          }      } |

* Build the application jars. After the tests have passed, it is the time to use “Maven” to build the code, compile it, package it and put the jar files in the appropriate folders.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | //Stage 4: Build with mvn      stage('Build with Maven') {          container('maven'){              dir ("./${app1\_name}") {                    sh ("mvn -B -DskipTests clean package")              }              dir ("./${app2\_name}") {                    sh ("mvn -B -DskipTests clean package")              }          }      } |

* Build docker images. In this stage, we build the Docker images of the *todobackend* and *todoui* apps using the created jar files in the previous stage.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | //Stage 5: Build Docker Image      stage('Build Docker Image') {          container('docker'){              sh("docker build -f ${app1\_dockerfile\_name} -t ${app1\_image\_tag} .")              sh("docker build -f ${app2\_dockerfile\_name} -t ${app2\_image\_tag} .")          }      } |

* Login to the Docker Registry and push the Docker images created before to the registry. Note that the values of the Docker environment variables should be configured in the Jenkins UI, so that you can connect to your Docker Registry properly.  
  Using the values of the Docker registry created in this example, the Docker environment variables will have the following values:
  + DOCKER\_CREDENTIALS\_ID -> (You should get this value from the credentials that you created in the “Configure Jenkins” section)
  + DOCEKR\_REGISTRY -> http://acrcicd.azurecr.io

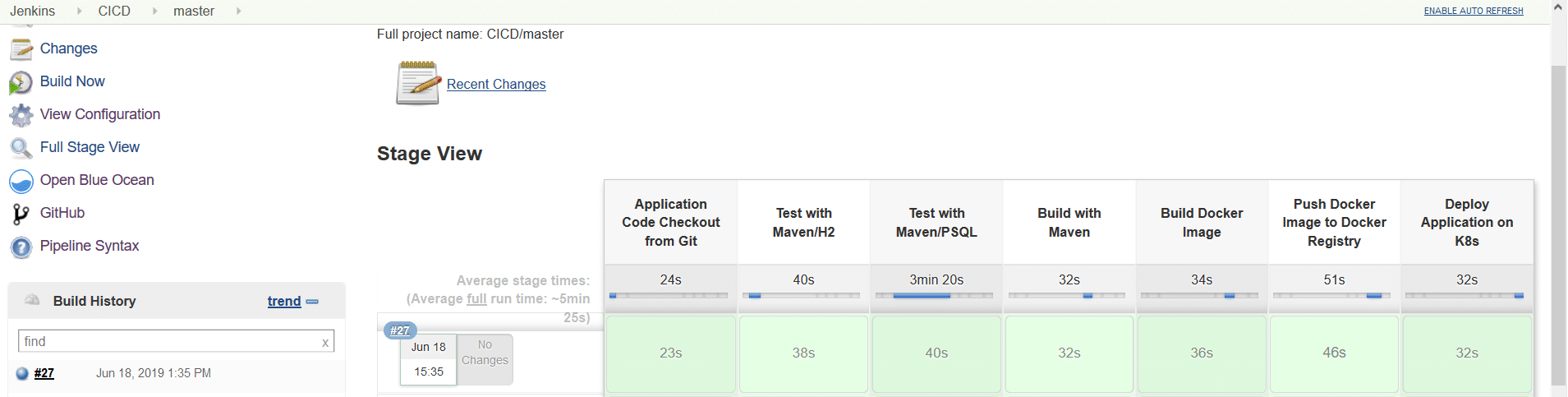


|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | //Stage 6: Push the Image to a Docker Registry      stage('Push Docker Image to Docker Registry') {          container('docker'){              withCredentials([[$class: 'UsernamePasswordMultiBinding',              credentialsId: env.DOCKER\_CREDENTIALS\_ID,              usernameVariable: 'USERNAME',              passwordVariable: 'PASSWORD']]) {                  docker.withRegistry(env.DOCEKR\_REGISTRY, env.DOCKER\_CREDENTIALS\_ID) {                      sh("docker push ${app1\_image\_tag}")                      sh("docker push ${app2\_image\_tag}")                  }              }          }      } |

* After previous stages have competed successfully, a deployment to kubernetes is triggered using the “kubectl” container. Note that the values of the K8s environment variables should have been configured in a previous step.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | //Stage 7: Deploy Application on K8s      stage('Deploy Application on K8s') {          container('kubectl'){              withKubeConfig([credentialsId: env.K8s\_CREDENTIALS\_ID,              serverUrl: env.K8s\_SERVER\_URL,              contextName: env.K8s\_CONTEXT\_NAME,              clusterName: env.K8s\_CLUSTER\_NAME]){                  sh("kubectl apply -f configmap.yml")                  sh("kubectl apply -f secret.yml")                  sh("kubectl apply -f postgres.yml")                  sh("kubectl apply -f ${app1\_name}.yml")                  sh("kubectl set image deployment/${app1\_name} ${app1\_container\_name}=${app1\_image\_tag}")                  sh("kubectl apply -f ${app2\_name}.yml")                  sh("kubectl set image deployment/${app2\_name} ${app2\_container\_name}=${app2\_image\_tag}")              }          }     } |

With that, you are ready to go! You can either change something in your Github repository and commit it, then the build of the pipeline will start automatically in jenkins, or you can start the build manually in jenkins UI. After it finishes, you should see a view similar to this, in which you can see the details of each stage (time it took, success or failure, logs) along with the Console Output of the build.

**Summary**

There you go! In this Blogpost, you have successfully created a CI/CD pipeline for a containerized spring-boot java application on Kubernetes using *Jenkins*. In Part 2 of this Blog-Series which will follow soon, we will set up a CI/CD pipeline using Bitbucket as a tool.

Top of Form



Bottom of Form

* [Cloud Computing](https://www.novatec-gmbh.de/en/blog/category/architecture-and-cloud/cloud-computing-en/)
* [Continuous Integration](https://www.novatec-gmbh.de/en/blog/category/development-programming-en/development-en/continuous-integration-en/)
* [Development & Programming](https://www.novatec-gmbh.de/en/blog/category/development-programming-en/)
* [DevOps](https://www.novatec-gmbh.de/en/blog/category/quality-engineering-en/devops-en/)
* [Education](https://www.novatec-gmbh.de/en/blog/category/education-en/)

### Related posts