Continuous Integration (CI) - Continuous Delivery (CD) Pipeline

What is Continuous Integration?

Continuous Integration (CI) is a development practice where developers integrate code into a shared repository frequently, preferably several times a day. Each integration is then be verified by an automated build and automated tests.

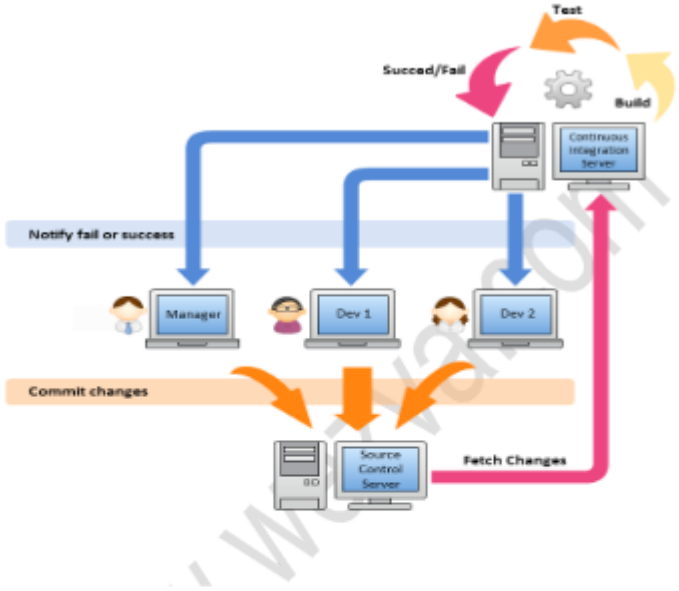
CI helps us to tell whether the code submitted by the developer is a good code which can be integrated with other code & a product can be created, as well as is the code good for QA testing.

CI Benefits

- Detect errors quickly and locate them more easily.
- Debugging build failures will be easy
- Helps Fasten Release cycle
- No separate integration step is needed in the lifecycle
- A deployable system at any given point
- Record of evolution of the project

Why Jenkins?

- Jenkins is an automation tool mainly used for setting up CICD, however it can be used to automate any manual tasks in which we give below details & it runs them accordingly
  - What to run
  - Where to run (machine or application)
  - When to run
- There are plenty of Plugins due to which its more flexible
- Acts as a dashboard to display all the job details
- Acts as a cron server replacement
- Stores complete history of every tasks
- Can be used for testing, deployment or scheduling day-day mundane tasks



Steps involved in Continuous Integration:

- Check SCM: At a regular frequency check from repository for new every commit, based on which the next steps would get triggered sequentially
- Integrate: All changes up until that point are combined into the project
- Build: The code is compiled into an executable or package
- Test: Automated test suites are run
- Code Quality: Generate Code Coverage & Code Analysis
- Archive: Versioned and stored so it can be distributed as is, if desired
- Deploy: Loaded onto a system where the developers can interact with it

Types of Software Builds?

I. Developer Builds

- Developers do it/ use it
- When ever the dev needs to verify his change
- Unit build & Unit test
- Done on developers machine
- Code based on Feature Branch

II. CI Builds

- Devops team Automates it
- We Reuse a workspace
- Incremental build
- Developers will use it
- Code based on Feature Branch
- Integration test
  - whenever a new checkin is submitted
  - poll scm (branch)
  - integrate the workspace
  - mvn package
    - compile
    - integration testing
    - package(war)
    - jacoco code coverage
  - sonarscanner code analysis
  - push to sonarqube
  - push war/artifacts to Jfrog Artifactory
  - notify to developers

III. Nightly Builds

- Devops team Automates it
- New workspace is used for every Build
- QA will use it
- Code based on Feature Branch
- We dp Full product build
  - Build at the cut off period i.e 1 or 2 times a day
  - Trigger at a scheduled time
  - generate new workspace
  - mvn package
    - compile
    - test (Skip)
    - package (war)
    - dockerfile - image
  - git tag (push to central repo)
  - push war/artifacts to Jfrog Artifactory

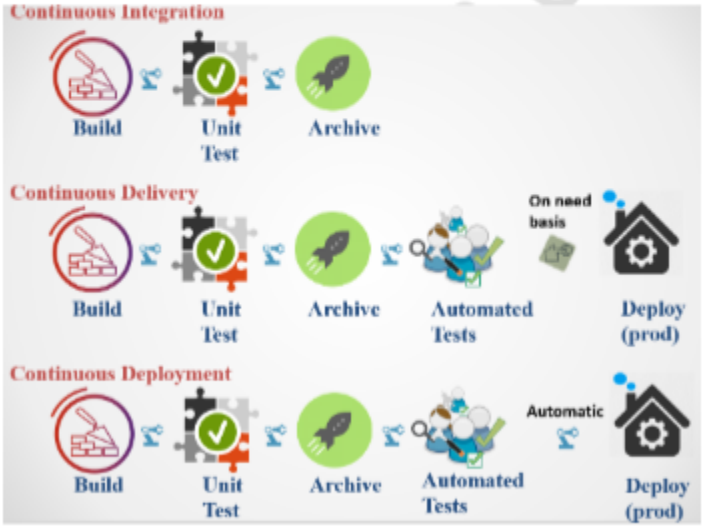
IV. Release Builds
- Code based on Release/Integration Branch
- Full product Build
- Customers use it or Production deployments happen from this
- Triggered when certified code is merged to Integration/Release branch

What is Continuous Delivery?

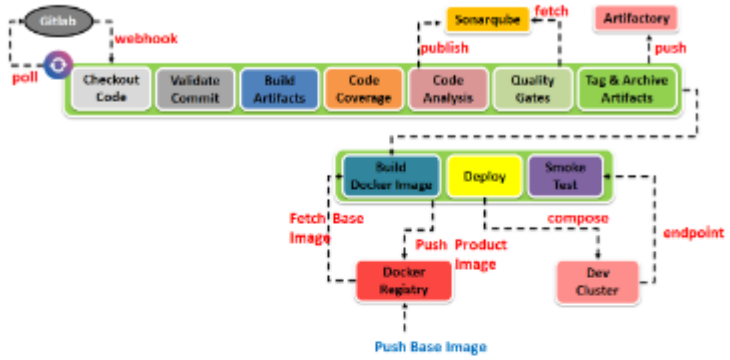
CI will helps to indentify whether the developer code submitted is a good code which can be tested, however due to faster release processes it is necessary to tell whether the code can be deployed.

Continuous Delivery is a process in which we ensure the code checkedin is not just ready for testing, but it is also ready for deployment at any point of time by running automated test suites.

Continuous Delivery = Continuous Integration + Automated Tests



CICD pipeline (Continuous Integration - Continuous Delivery



STAGE 0 (Triggering CI through Webhook)

- Install 'Gitlab Plugin' on Jenkins & restart Jenkins (https://plugins.jenkins.io/gitlab-plugin)
- Generate Personal Access Token in Gitlab i.e User settings -> Access tokens
- In Jenkins add credentials for Gitlab API token with the above value
- Once the API Access has been setup, we can configure the connection between Jenkins and GitLab
  - This happens in the Mange Jenkins -> Configure System -> Gitlab
  - Unchock the box "Enable authentication for '/project' end-point"
- Create Webhook:
  - Go to the Gitlab project -> Settings -> Integrations
  - Add URL "http:<JenkinsURL>:8080/project/<JobName>"
  - Enable "Push Events" under Trigger
  - Unchock "Enable SSL verification"

STAGE 1 (Checkout Code)

- Stage (Pre-Check): Evaluate whether to build the code or not, depending on the files modified in the commit

** REFER CLASS NOTES FOR DEMO SOURCECODE **

STAGE 2 (Build Artifacts)

** REFER CLASS NOTES FOR DEMO SOURCECODE **

- Build Artifacts & run Unit test
  $ mvn package
- Build Artifacts & skip Unit test
  $ mvn package -Dmaven.test.skip=true

STAGE 3 (Code Coverage)

- Add the following Maven Jacoco plugin in pom.xml of the project

```
<plugin>
 <groupId>org.jacoco</groupId>
 <artifactId>jacoco-maven-plugin</artifactId>
 <version>0.5.5.201112152213</version>
 <configuration>
  <destFile>${basedir}/target/coverage-reports/jacoco-unit.exec</destFile>
  <dataFile>${basedir}/target/coverage-reports/jacoco-unit.exec</dataFile>
 </configuration>
 <executions>
  <execution>
   <id>jacoco-initialize</id>
   <goals>
    <goal>prepare-agent</goal>
   </goals>
  </execution>
  <execution>
  <id>jacoco-site</id>
   <phase>package</phase>
   <goals>
    <goal>report</goal>
   </goals>
   </execution>
  </executions>
 </plugin>
```

- Invoke jacoco plugin directly
  $ mvn org.jacoco:jacoco-maven-plugin:0.5.5.201112152213:prepare-agent

STAGE 4 (Code Analysis)

- Install Docker & install sonarqube using docker image
  $ docker run -d --name sonarqube -p 9000:9000 sonarqube
- Access using <url>:9000 and credentials: admin/admin

- Install 'SonarQube Scanner' Plugin in Jenkins
- Install SonarQube Plugin in Jenkins
  (https://wiki.jenkins-ci.org/display/JENKINS/SonarQube+plugin)
  - Manage Jenkins -> Configure System -> SonarQube servers -> Add SonarQube -> Give URL of the sonarwube server
  - Manage Jenkins -> Global Tool Configuration -> SonarQube Scanner -> Path of scanner home dir

- Create 'sonar-project.properties' file under the parent dir of the project & add the below:

```
# Mandatory Medata required
  sonar.projectKey=WEZVA
  sonar.projectName=SonarDemo
  sonar.projectVersion=1.0
# path to the src dir of the maven project
  sonar.sources=src
  sonar.java.binaries=target\\classes
  sonar.jacoco.reportPath=target\\coverage-reports\\jacoco-unit.exec
```

STAGE 5 (Quality Gates)

- Create SonarQube User Token in SonarQube Server (top right corner)
  - User -> My Account -> Security -> Generate Tokens
- Configure SonarQube webhook for quality gate in Sonarqube
  - Administration -> Configuration -> Webhooks -> Create
  - Give url as "http:<JenkinsURL>:8080/sonarqube-webhook/"
- Install Quality Gates Plugin in Jenkins (https://wiki.jenkins-ci.org/display/JENKINS/Quality+Gates+Plugin)
- Add SonarQube Token to Jenkins
  - Manage Jenkins > Configure System > SonarQube
  - The server authentication token should be created as a 'Secret Text' credential

STAGE 6 (Archive Artifacts)

- Install Artifactory using Docker

```
$ mkdir -p ~/jfrog/artifactory/var/etc
$ chmod -R 777 ~/jfrog
$ touch ~/jfrog/artifactory/var/etc/system.yaml
$ chown -R 1030:1030 ~/jfrog/artifactory/var
$ docker run --name artifactory -d -v ~/jfrog/artifactory/var:/var/opt/jfrog/artifactory -p
8081:8081 -p 8082:8082 docker.bintray.io/jfrog/artifactory-oss:latest
```

- Access using <url>:8082/artifactory and credentials: admin/password
- Install Artifactory plugin in Jenkins
  https://wiki.jenkins-ci.org/display/JENKINS/Artifactory+Plugin
- Add Artifactory server details in Jenkins
  - Manage Jenkins -> Configure System -> Add Artifactory server -> URL

STAGE 7 (Build Image)

** REFER CLASS NOTES FOR DEMO Baseimage & Dockerfile **

- Add Dockerhub (Docker Registry) Credentials
  - Jenkins > Credentials > System > Global Credentials > Add Credentials > username and password
  - Give the dockerhub username & password

STAGE 8 (Deploy)

** REFER CLASS NOTES FOR DEMO DEPLOYMENT CONFIGURATION **

- Install docker-compose on Pilot
  $ apt install -y docker-compose

  If you get the following error "Error saving credentials: error storing credentials - err: exit status 1, out: `Cannot autolaunch D-Bus without X11 $DISPLAY`"
  Run the cmd:
  $ rm /usr/bin/docker-credential-secretservice

- Install "Docker Compose Build Step" plugin in Jenkins

STAGE 9 (Smoke Test)

** REFER CLASS NOTES FOR DEMO TESTSUITE **

Testcase1: Check for Application Endpoint URL

Testcase2: Check for core logs

Deploy using Kubernetes:

- Setup a Kubernetes cluster with 1 Master & 1 Node
  - Kubernetes Master SETUP:

--run as root--
$ apt install -y docker.io
$ usermod -a -G root ubuntu
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
$ cat << EOF > /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt update
$ apt install -y kubelet kubeadm kubectl
$ kubeadm init --pod-network-cidr=10.244.0.0/16 (Copy the output to run on worker)

---run this as non-root user---
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
$ kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

Kubernetes Slave SETUP:
--run as root--
$ apt install -y docker.io
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

```
$ cat << EOF > /etc/apt/sources.list.d/kubernetes.list
  deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ apt update
$ apt install -y kubeadm
$ Run the output obtainer from 'kubead init' on master in the above step
 Example: kubeadm join 172.31.36.238:6443 --token qvj91q.f4uqrlzylu0830q6 \
    --discovery-token-ca-cert-hash
sha256:4a2012a44cf8fd7a3941c7b9002366d1ad4a7e4746903d2c8ffaaf9e8651983a
```

- Add the Kubernetes Master as a Slave to Jenkins using ubuntu user
  - Enable passwordless ssh between Jenkins Master & Kubernetes Master
- Install the below plugin in Jenkins : "Kubernetes Continuous Deploy" (https://plugins.jenkins.io/kubernetes-cd/)
- Create a Jenkins Global Credential
  - Kind as "Kubernetes Configuration (kubeconfig)"
  - scope as "Global (Jenkins, nodes, items etc)"
  - give a desired ID string
  - kubeconfig, choose "Enter Directly" option & Copy the contents of ~/.kube/config from Kubernetes Master (ubuntu user's home dir)