# Before jenkins:

* Code integration was not accurate
* Entire software has to be developed, then tested to check for errors
* Developers have to wait for development to complete, then look at bugs, in such cases solving bugs is not easy (but complex)
* No iterative improvement of code
* Software delivery process was slow

# What is Jenkins:

* Jenkins is a **continuous integration tool** that allows **continuous development, test, and deployment** of newly created codes
* Before Jenkins nightly build and integration is followed, with Jenkins continuous build and integration is followed
* Along with the newly developed code, you are putting your tests and verification services in to the build environment, so you can always test your code, therefore making integration more robust with less bugs and errors

Graphical user interface, application

Description automatically generated

## What is Continuous Integration:

Diagram

Description automatically generated

Developer pulls the code from repository and makes new changes to code and then pushes it to CI server (you can make use of another CI server, if the current code sent to CI server by you or another developer is taking too long, then you can use another CI server there by making the process efficient and productive), if the tests In the CI server are failed, The code is sent back to developers, else if the tests pass, then it will be passed to tester, if there are code standard errors or other errors then the code is sent back to developers from testers, else if there are no errors, then the code is ready for deployment.

# Various CI tools:

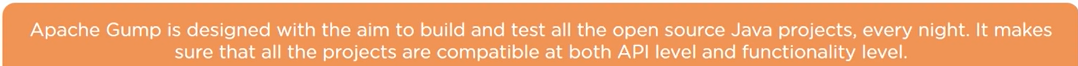
Bamboo:

A CI tool that can **run multiple builds in parallel for faster compilation**. It has the functionality to connect with repositories, and has build tasks for ant, Maven, etc.

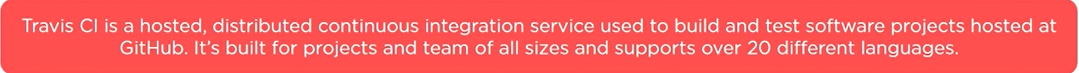
Buildbot:

An open source framework used to automate software build, test and release processes, built using python, supports distributed, parallel execution of jobs across multiple platforms

Apache gump:



Travis CI:



Jenkins:

An open source automation server written in JAVA. It automates software development process via Continuous integration and facilitates Continuous Delivery

# Features of Jenkins:

1. Easy Installation: Jenkins is a self contained java based program, ready to run with packages for Windows, Mac OS, Unix like OS
2. Easy configuration: can be easily setup and configured with the help of web interface, which includes error checks and built in help
3. Plugins: it has hundreds of plugins in update center of Jenkins and integrates with every tool in the CI CD tool chain
4. Extensible: Jenkins can be extended via its plugin architecture and provides infinite possibilities of what it can dox
5. Distributed: it can easily distribute work across multiple machines, thereby facilitating faster builds, tests and deployments across multiple platforms

# Jenkins Pipeline:

Timeline

Description automatically generated

# Jenkins architecture:

Graphical user interface, application

Description automatically generated

If you have multiple files (java, html and …) that need different builds then Jenkins need to have multiple build servers with different compatible versions

## Jenkins Master-Slave architecture:

Diagram

Description automatically generated

After pulling code from repository the Jenkins server master distributes load to all its slaves (different Operating systems) so that it can carry out builds and tests, produce test reports

# Jenkins File:

* Instead of configuring Jenkins UI to schedule jobs, you can script that build and its configurations in a file, so Jenkins file is pipeline as a code (PAAC), a scripted pipeline which is part of this whole infrastructure as a code concept
* Jenkins file can be declarative/scripted
* Scripted vs declarative:

Declarative has pre-defined structure

* + A picture containing timeline

    Description automatically generated
* Declarative structure:
  + Pipeline in below image states that the file is a pipeline
  + Agent any- states that it can be run on any node
  + Graphical user interface

    Description automatically generated
  + Stages can be multiple: ex: build, test, deploy, steps is where you include steps to be covered for each stage
  + Graphical user interface, text, application

    Description automatically generated
* Scripted structure:
  + Text

    Description automatically generated

## Post:

We can have some conditions which can be executed after all the stages have been executed

Always: irrespective of batch execution status, the code in this block will be executed

Success: the code in this block gets executed if the batch execution is successful

Failure: the code in this block gets executed if the batch execution is a failure

## Unstable:

A build is unstable if The Build had some errors but they were not fatal. A Build is unstable if it was built successfully, and one or more publishers report it unstable.

You can have unstable section in post section like this:

Post {

Unstable {

error(“code quality checks failed”)

}

}

We can also manually trigger the unstable state [here](https://www.jenkins.io/doc/pipeline/steps/workflow-basic-steps/#unstable-set-stage-result-to-unstable)

Stage(‘temp’) {

Steps {

unstable(message: ‘custom message you wanna give’

}

}

### Triggering unstable condition using warn pipeline setup:

Similar to raise exception in python, we can create a warning using the [warn pipeline setup](https://www.jenkins.io/doc/pipeline/steps/workflow-basic-steps/#warnerror-catch-error-and-set-build-and-stage-result-to-unstable)

If you give exit code 0, it will be a successful build,

If you give exit code 1, it should be an error, Jenkins catches that error and will flag the build as unstable

Stage(‘temp’){

Steps{

warnError(message: :‘watch this’) {

sh ‘Exit 1’

}}}

## When expression:

If you have a scenario where you have to execute a stage only if it is a dev or a master branch, then you can make use of **when expression** block as below, here **BRANCH\_NAME** is an environment variable which stores the name of the branch which is being considered,also you can use **or, and** to join one or more expressions an environment variable can also be declared as env.BRANCH\_NAME, environment variables can be declared in Capital letters (afaik)

Graphical user interface

Description automatically generated

## **Evaluating when before entering agent in a stage**

By default, the when condition for a stage will be evaluated after entering the agent for that stage, if one is defined. However, this can be changed by specifying the beforeAgent option within the when block. If beforeAgent is set to true, the when condition will be evaluated first, and the agent will only be entered if the when condition evaluates to true.

Example

when {

beforeAgent true ;

allOf {

anyOf {

changeset 'requirements.txt'

changeset 'env/docker/Dockerfile.base'

expression { return params.FORCE\_REQ\_BUILD }

expression {

!(currentBuild.previousBuild != null

&& "SUCCESS".equals(currentBuild.previousBuild.result))

}

}

}

}

## Environment Variables:

There are many env variables in Jenkins, they can be referenced in the link: localhost:<portnumber>/env-vars.html

### Environment Block:

You can use an environment block as a place where you store all the information related the repeating variables

Text

Description automatically generated

## Tools:

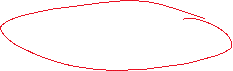
When you have both front end and backend for your application, then you might need certain tools to install related packages, that is done with the help of tools

Tools are needed when you want to install packages which are application specific: ex: maven, gradle, jdk

Below you can see maven command being used in build stage

Text

Description automatically generated



## Parameters:

To have user defined choices while deploying a build we can use parameters, in the below scenario, you can see, you might want to deploy 1.1.0 or 1.2.0 or any other version, also you can have the option to perform test stage if you select the checkbox while build is happening

Text

Description automatically generated

Text

Description automatically generated

The above choice line is getting displayed like this with all the possible values populated

A picture containing text

Description automatically generated

The above params.executeScripts is getting displayed as checkbox here

Graphical user interface, text

Description automatically generated with medium confidence

## Embedding groovy scripts inside Jenkins File:

<https://youtu.be/7KCS70sCoK0?t=1649>

groovy script will be a separate file, all env variables available in jenkin files will be available in groovy script

## Changeset:

changeset <path> : this will check Whether the files in the specified path has been changed or not, if changed, it will become true, else it will become false

can be used in blocks like anyOf, allOf, to set some execution flow

## Kubernetes agent:

Link: https://plugins.jenkins.io/kubernetes/#plugin-content-declarative-pipeline

We have a lot of configurations we do for an Kubernetes agent we run, but major configurations present in pod and container parts, they are explained here:

### Pod template:

* **cloud** The name of the cloud as defined in Jenkins settings. Defaults to kubernetes
* **name** The name of the pod. This is only used for inheritance.
* **namespace** The namespace of the pod.
* **label** The node label. This is how the pod template can be referred to when asking for an agent through the node step. In a pipeline, it is recommended to omit this field and rely on the generated label that can be referred to using the POD\_LABEL variable defined within the podTemplate block.
* **yaml** [yaml representation of the Pod](https://kubernetes.io/docs/reference/kubernetes-api/workload-resources/pod-v1/), to allow setting any values not supported as fields
* **yamlMergeStrategy** merge() or override(). Controls whether the yaml definition overrides or is merged with the yaml definition inherited from pod templates declared with inheritFrom. Defaults to override() (for backward compatibility reasons).
* **containers** The container templates part of the pod (see below for details).
* **serviceAccount** The service account of the pod.
* **nodeSelector** The node selector of the pod.
* **nodeUsageMode** Either NORMAL or EXCLUSIVE, this controls whether Jenkins only schedules jobs with label expressions matching or use the node as much as possible.
* **volumes** Volumes that are defined for the pod and are mounted by **ALL** containers.
  + configMapVolume : a read only volume that is mounted from a ConfigMap.
  + dynamicPVC() : a persistent volume claim managed dynamically. It is deleted at the same time as the pod.
  + emptyDirVolume (default): an empty dir allocated on the host machine
  + hostPathVolume() : a host path volume
  + nfsVolume() : a nfs volume
  + persistentVolumeClaim() : an existing persistent volume claim by name.
  + secretVolume : a read only volume that is mounted from a Kubernetes secret.
* **envVars** Environment variables that are applied to **ALL** containers.
  + **envVar** An environment variable whose value is defined inline.
  + **secretEnvVar** An environment variable whose value is derived from a Kubernetes secret.
* **imagePullSecrets** List of pull secret names, to [pull images from a private Docker registry](https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/).
* **annotations** Annotations to apply to the pod.
* **inheritFrom** List of one or more pod templates to inherit from (more details below).
* **slaveConnectTimeout** Timeout in seconds for an agent to be online (more details below).
* **podRetention** Controls the behavior of keeping agent pods. Can be 'never()', 'onFailure()', 'always()', or 'default()' - if empty will default to deleting the pod after activeDeadlineSeconds has passed.
* **activeDeadlineSeconds** If podRetention is set to never() or onFailure(), the pod is deleted after this deadline is passed.
* **idleMinutes** Allows the pod to remain active for reuse until the configured number of minutes has passed since the last step was executed on it.
* **showRawYaml** Enable or disable the output of the raw pod manifest. Defaults to true
* **runAsUser** The user ID to run all containers in the pod as.
* **runAsGroup** The group ID to run all containers in the pod as.
* **hostNetwork** Use the hosts network.
* **workspaceVolume** The type of volume to use for the workspace.
  + dynamicPVC() : a persistent volume claim managed dynamically. It is deleted at the same time as the pod.
  + emptyDirWorkspaceVolume (default): an empty dir allocated on the host machine
  + hostPathWorkspaceVolume() : a host path volume
  + nfsWorkspaceVolume() : a nfs volume
  + persistentVolumeClaimWorkspaceVolume() : an existing persistent volume claim by name.

### Container template:

Container templates are part of pod. They can be configured via the user interface or in a pipeline and allow you to set the following fields:

* **name** The name of the container.
* **image** The image of the container.
* **envVars** Environment variables that are applied to the container **(supplementing and overriding env vars that are set on pod level)**.
  + **envVar** An environment variable whose value is defined inline.
  + **secretEnvVar** An environment variable whose value is derived from a Kubernetes secret.
* **command** The command the container will execute. Will overwrite the Docker entrypoint. A typical value is sleep.
* **args** The arguments passed to the command. A typical value is 99999999.
* **ttyEnabled** Flag to mark that tty should be enabled.
* **livenessProbe** Parameters to be added to a exec liveness probe in the container (does not support httpGet liveness probes)
* **ports** Expose ports on the container.
* **alwaysPullImage** The container will pull the image upon starting.
* **runAsUser** The user ID to run the container as.
* **runAsGroup** The group ID to run the container as.

# Misc:

If you want to decide a set of stages should execute or not based on some certain conditions, so that these stages will only execute if the specified conditions hold true, then we can use nested stages, whereas in the first stage you should specify the conditions based on which the next stages execution state depends upon.

disableConcurrentBuilds- Disallow concurrent executions of the Pipeline. Can be useful for preventing simultaneous accesses to shared resources

def <variable name>: this is way of declaring variables, before they are used (can also be used without declaring variables)

allOf: if we have some conditions under allOf, the steps/stages in the current stage will execute only if all the statements are executed true.

anyOf: if we have some conditions under anyOf, the steps/stages in the current stage will execute only if at least one of the statements are executed true.

Stages inside stages are useful when you want to execute/avoid build based on some set of conditions

We can have post for stages and stage as well

Aborted: The Build was interrupted before it reaches its expected end. For example, the user has stopped it manually or there was a time-out. Useful in a post section so that if the stage/stages are aborted then we can execute this set of things

aborted {

\\statements

}

* when you execute some tests (using makefile or maven file or any other file) some reports related to it are generated in xml format, we need to check whether there are any errors, for us to know if there are any errors we can use Jenkins which can read those reports in xml format, **Jenkins can read the reports just after all the steps are executed and before the next stage starts, i.e., in the post block of the current stage**