1. **What will be the git branching strategy, elaborate why & why not with pros and cons?**

**Why do we need a Branching strategy? What is it?**

In a larger project with a multi-branching model, how do you manage the branching development model and enforce policies on various branches for a shorter development lifecycle and faster integration of work?

A good branching strategy leads the project source code towards **consistent** and **secured** data that is shareable and accessible by all collaborators in a shorter life cycle time.

It is necessary that you design your project model in a flexible way that is well **administered** for all the member roles and permissions.

An **ideal branching strategy**, in practice, should declare a set of guidelines protecting the **branches** against read and write permissions and restricting the unauthorized creation, deletion, forced pushes to the branches.

Whether you are maintaining your source code on a **Linux Server or** on a **web-hosted**, cloud-based source code control services, create your branches in a structured way and protect them for certain members and actions.

[Git Flow](https://nvie.com/posts/a-successful-git-branching-model/) describes multiple branches for development, releases and the orchestration between them. There are even [scripts and extensions](https://github.com/nvie/gitflow) provided to help use/maintain Git Flow.

* Need a set of rules/strategy to commit, merge and promote changes to a repository. This defines a git collaboration workflow.
* Let’s you define a structured delivery of work, increasing efficiency & reducing error
* Introduce opportunity for code review and protected branch.
* Exploring strategies to reduce merge conflicts
* Clean your working directory before doing a merge
* If you have work in progress, either commit it to a temporary branch or stash it
* Commit often, and merge often
* Create more of small topic branches rather than long running branches
* Push/pull often, publish your work often
* Rebase your work than merging only on your local repository and local branch.
* Separate different types of branches like: master, hotfixes, release, develop, feature & topic branches.
* Keep changes physically small. Small diffs, small amount of lines changed (easier to debug)

**Pros ?**

* Git Flow is used by a lot of distributed, open-source teams that have varying skill levels. The project maintainers can review and approve every line of code going into releases.
* Works well with in Continuous Integration or Continuous Delivery scenarios.
* Can release more often, including multiple times per day.
* Much tighter feedback loop for developers, business, and end-users.
* Run automated tests on all branches.
* Deploy depending on the exact strategy — it may be from the master branch, from the production branch, or from multiple branches.
* Ensures a clean state of branches at any given moment in the life cycle of project

**Cons ?**

* Git Flow can slow things down when having to look at large pull requests if you are trying to iterate quickly.
* It isn’t recommended when it needs to maintain single version in production.
* feature conflicts arises on environment branches which already has other in development features.

**git branching strategy**

A good branching strategy is the one that adapts to your project and business needs. Every organization has a set of its own defined SDLC processes.

Different teams like the Administration team, Development, Testing, Operations, and the QA team worked together harmoniously contributing to the project source code.

This approach lets you have only stable QA approved commits to be pushed from “**release**” branch into “**prod**” branch with the squash merge strategy. “**master\_prod**” only accepts final commits of the source code from the “**prod**” branch.

Main code development happens on the feature and topic branches.

"master\_prod" is the deployment-ready branch. A CD pipeline can be automated on this branch for continuous deployment in a DevOps environment setup.

**First we need to Develop and decide a Branching Model considering below 6 points**

* Decide on the VCS and the type of model architecture
* Select the git workflow model
* Select a branching strategy
* Create branches and define guidelines
* Define member roles and permissions
* Protect branches on Jenkins/gitlab and maintain branching guidelines

Depending upon the type of code development a branch is used for, branches may be briefly classified as:

* **Stable/Integration or long-running branches** – these generally have a clean history with no 3-way merge and have a short user-friendly history. “prod” and “master\_prod” are two such stable branches in our project example.
* **Release branch**merges the changed commits from all the feature and hotfix branches into one. This is where the conflicts are verified, reviewed, and resolved. (Refer: git resolve conflict- Point 6. Merge tools (diffmerge): resolving conflict).

Release branch ought to have a non-linear and messy commits history, which makes it easier to list the entire project commit history and analyze how the project developed and which commit was created by whom and what changes were made.

* **Feature branch**branches out from the main development branch (“master” branch more commonly). This is used to develop specific code by developers. These are short-lived branch targeting a specific code issue or a fix; such as feedback or even a new feature. Work on the feature branch is merged into the stable branch upon approval. My feature branches are “dev” and “uat”.
* **Unstable**or**hotfix branch**is short-lived private quick fix branches created to develop and test a bugfix. These are merged into the feature branches and deleted. “hotfix” is such an unstable branch in my project.

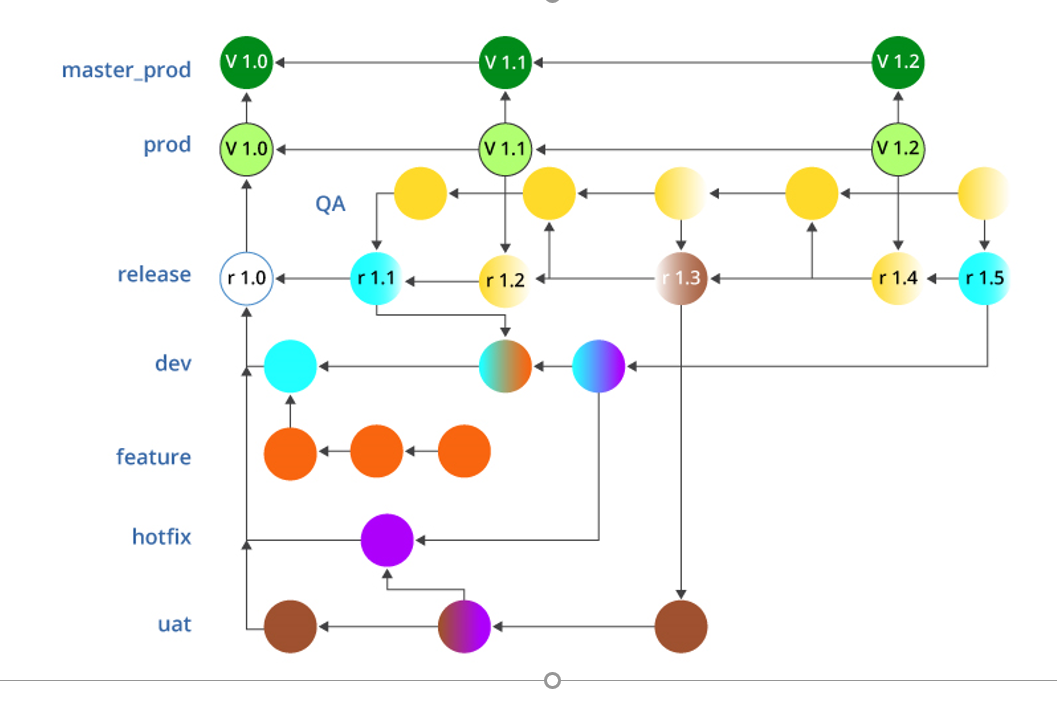
**Branches Explanation**:

* "**master-prod"** branch merge unique commits only from the "**prod"** branch
* "**prod"**branch merge "**QA"** approved commits only from the "**release**" branch with the squash merge strategy. Each commit in "**prod"**branch is tagged in the format: v1.0, v1.1 … v1.\*
* "**release"** branch merge unique commits from the branches "**dev**", "**uat**" and "**QA**". Each commit on this branch is tagged in the format: r1.0, r1.1 … r1.\*
* "**dev**" and "**uat**" branches never merge.
* "**hotfix**" branch commits are shared among any feature branches such as "**dev**" and "**uat**" or any future feature branches.
* Each hotfix or quick-fix branch is private to the feature branches and are dropped (deleted) after the merge.

Why **squash** merge technique?

The squash flag will pick up unique commits from the release branch and update the working tree and the Index area without actually making the commits.

**An example branching structural strategy that I have used in my one of project:**



* **Diagram:** Branching strategy
* Clone the project available at github:
* git clone http://github.com/suman/XXXX.git

**Branches Brief explanation:**

* “master-prod”: Accepts merges/code/commits only from the “prod” branch
* “prod”: Perform only a merge --squash from “release” branch.
* Merge only when approved by “QA”
* Tag every merge in the format: v1.0, v1.1 … v1.\*
* “release”: merge from the branches “dev”, “uat”, “QA”.
* Every release commit/project code version has to be approved by “QA”.
* Tag every merge in the format: r1.0, r1.1 … r1.\*
* “dev” and “uat” never merge with each other.
* “hotfix” branch commits are shared among any feature branches such as “dev” and “uat”
* “feature” branch is private to “dev” alone and is dropped after merging.
* CI/CD DevOps tools can be used to automate the above development and deployment to master\_prod.
* Every project release: r1.0 .. r1.x on the ‘release’ branch can be tracked by Jenkins CI tool and will trigger a build, on a successful build continuous testing suite
* cases will be triggered on the code. If the test passes the release will be delivered to ‘prod’ branch.
* Every source code delivered to ‘prod’ branch will be automatically deployed to ‘master\_prod’ branch.

All the steps will be mentioned in a Jenkins file on a branch ‘name’ condition.

## Member Roles and Permissions

**Jenkins or Gitlab** is an excellent cloud-based SCM and DevOps tool that provides an additional feature of protecting branches. Host your project on GitLab or Jenkins, which you could either import from GitHub or BitBucket or create a new project and push your source code from your Server machine using remote repository reference.

Gitlab/Jenkins allows you to define users as members with different project-level permissions and describes the roles – “Guest”, “Reporter”, “Developer”, “Maintainer” and “Owner” in detail.

Protect branches against permissions such as “allowed to merge” and “allowed to push” to the members.

**NOTE**: This is a private project with a maintainer and a developer; the maintainer has to personally give access to developers or a group as learned in Member role and permissions section.

**Tip**: Protected branches can only be deleted by the owner or maintainer of the project via the GitLab web interface.

The developer will fork the project and clone to his/her local repository. Note, how Developers have "merge" and "push" access only to some feature("dev", "uat" and "hotfix") branches; while stable branches are merged and pushed by Maintainers alone.

This gives a chance to review the merged changes from different developers and allow only tested code to be pushed into the stable branches.

## Further Development of the Project

1. With this protected branch strategy setup, each developer will have to clone this repository and work in their local working directory. Developers are allowed to create their own private branches and base their work on these existing branches.
2. Each developer then pushes his/her feature branch ("dev", "uat", "bug#302", "newFeature" etc...) into the main repository where Maintainer reviews and merge these feature branches into the "**release**" branch. While merging into the "release" branch maintainer has to resolve the possibilities of conflicts and collaborate with each developer’s work. The "release" branch generally has either a fast-forward or a recursive merge.
3. New commits from "release" branch are merged into the "**QA**" branch, upon a test and review in the QA branch; the QA approved changes/feedbacks are merged back into the "release" branch.
4. Only each "QA" approved commit is merged with the "**prod**" branch. To keep the "prod" branch history clean, short and linear we use the git **squash** merging strategy, like so -

=================================================================================================

1. **Which tool you will consider for CI/CD**

Considering DevOps to be an ideology towards achieving a quality product, every organization has its own guidelines and approach towards it.

Some of the popular tools I have used are:

GIT - Version control tool Docker - Containerization tool

Github or BitBucket - Version Control hub Kubernetes(K8’s) - Orchestration tool

Jenkins - CI/CD tool AWS or GCP - Cloud based service

Maven - Build tool Linux RHEL7/RHEL8

SonarQube - Code Analysis tool Shell scripting, Groovy scripting, YAML

Sonatype Nexus - Artifact repository Virtualization – VMware, Hyper-V

Apache Tomcat or Nginx - Application server Nagios, EFK, Prometheus – Monitoring Tool

Apache HTTPD – Web server Docker Registry – Image Repository

JIRA – Defect Tracking Tool K8s Package Manager - Helm Charts

Ansible or PUPPET - Configuration management tool

=================================================================================================

1. **What will be your build promotion plans (Dev - QA - Prod)**

**Goal:**

The DEV Jenkins build is running in a continuous integration set-up, as code is checked in to git, Jenkins will run a new build (clean, compile, test, deploy).

The idea is to be able to manually promote a successful DEV build to QA. QA build would check out the DEV's git Revision number, build, test, deploy to staging and finally create a branch in git.

Lastly the release manager could manually promote the QA build to PROD. PROD build would check out the branch from the previous QA build, deploy to PROD and tag the branch as a release.

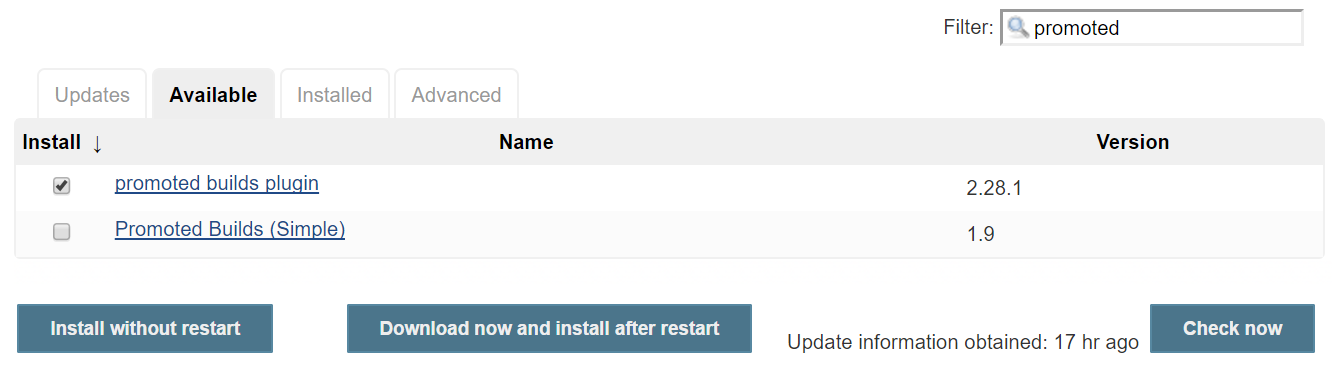
I have tried to use a combination of the Promotion Builds Plugin and the Paramterized Trigger Plugin.

**Setup and Configuration:**

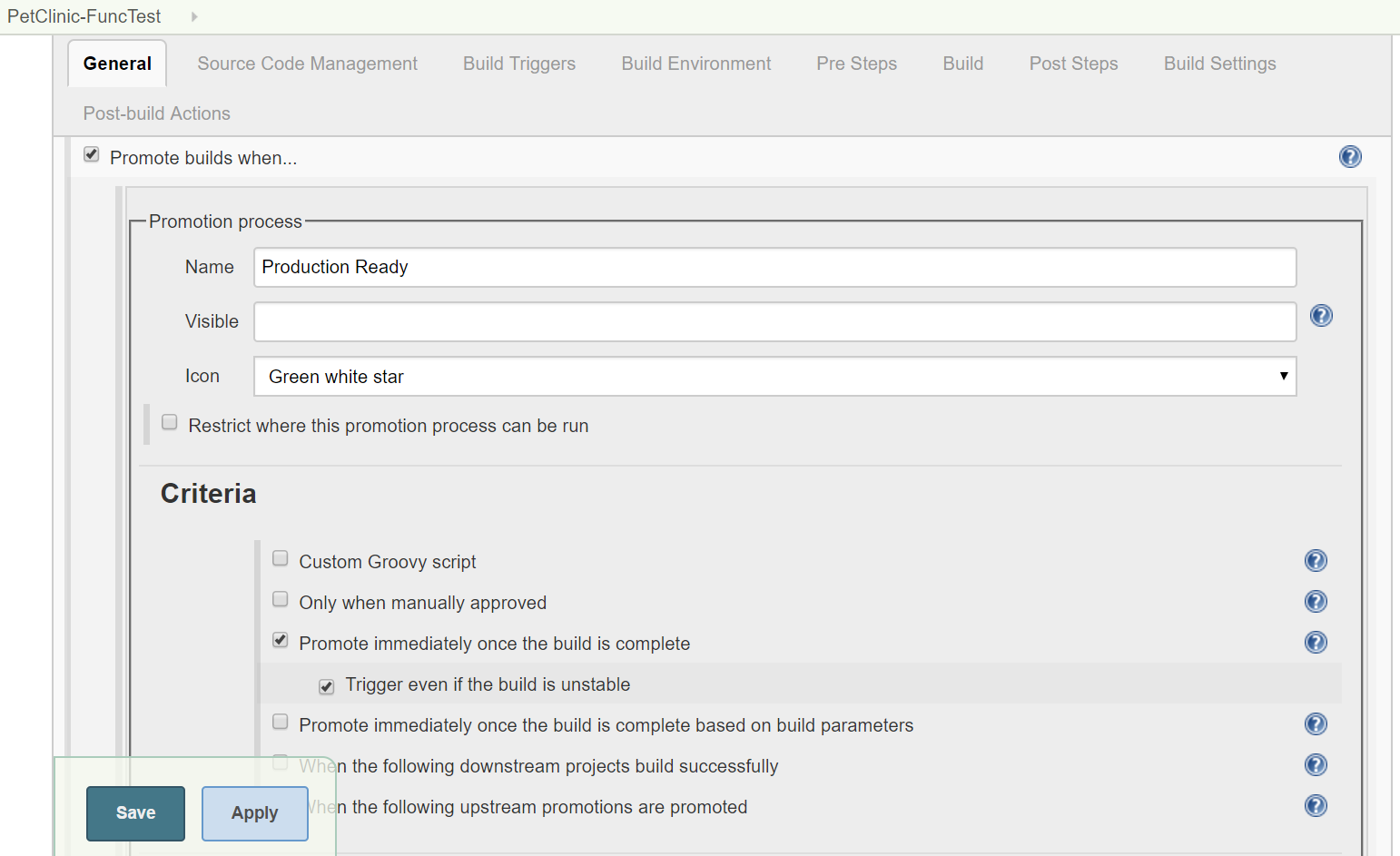
**Promoted builds**

The Promoted builds plugin allows us to tag the builds based on specific stages. This promotion can be manual or automated. We can identify promoted builds based on the star available on the project dashboard or the star available in Build History.

1. Go to **Manage Jenkins** and click on **Manage Plugins**.
2. Select **promoted builds plugin** and click on **Install without restart**:



1. Go to the **specific** build and open its configuration.
2. In the **General** section, click on **Promote builds when...**
3. Provide a name and select a star you want to associate build if the criteria is passed in the icon list box in the promotion process section.
4. Select **Promote immediately when build is** complete, as shown in following screenshot:



1. We can also select **Only when manually approved** and then we can give the Email ID of the approver.
2. Click on **Build Now** and observe the Jenkins dashboard. Look out for the green star in **Build History** when the build is executed successfully:

Another approach is to make use of the Artifact storage Jenkins provides coupled with the [Copy Artifact Plugin](https://wiki.jenkins-ci.org/display/JENKINS/Copy+Artifact+Plugin).

1. When a build is completed, you could instruct Jenkins to persist your application, either as a compressed zip/tar.gz or as an application bundle (jar/war)
2. Trigger a downstream job and use the Copy Artifact to retrieve the recorded artifact from the upstream job (or use parameterised builds)
3. Deploy/Unzip artifact as necessary using Build shell script/maven deploy
4. Retest application using the same sources/binaries as was created in step 1
5. Repeat for QA and Prod environments

**If you want to use in Job DSL format:**

The following environment variables are added for use in scripts, etc. These were retrieved from github [here](https://github.com/jenkinsci/promoted-builds-plugin/blob/master/src/main/java/hudson/plugins/promoted_builds/Promotion.java).

* PROMOTED\_URL - URL of the job being promoted
  + ex: <http://jenkins/job/job_name_being_promoted/77/>
* PROMOTED\_JOB\_NAME - Promoted job name
  + ex: job\_name\_being\_promoted
* PROMOTED\_NUMBER - Build number of the promoted job
  + ex: 77
* PROMOTED\_ID - ID of the build being promoted
  + ex: 2012-04-12\_17-13-03
* PROMOTED\_USER\_NAME - the user who triggered the promotion
* PROMOTED\_JOB\_FULL\_NAME - the full name of the promoted job

### **Example**

freeStyleJob('test-job') {

properties{

promotions {

promotion {

name('Development')

conditions {

manual('testuser')

}

wrappers {

timestamps()

}

actions {

shell('echo my-app;')

}

}

}

}

}

**Manual Approval for promotion**

**Build**:  This job includes the configuration for the building project, job triggers, scm location, jdk version to use, maven goals, artifact upload to repo like Nexus, Artifactory, and  email notification.

**Test**: This job can call test suites and decide to call a downstream job or not.

**Dev Deploy**: Simple job with a trigger to the Promotion Job if the deployment was successful.

This job can call the script to perform a deployment or use tools like Jenkins,

**QA Promotion**: This job includes a send email notification to the person/group responsible for approval. The email contains a link for promotion and an optional comment for approval notes.

Once approved, the deploy job will run and If you are a Jenkins Admin or have global privileges, you will see Force Execution Option.

After Re-execute Promotion Jenkins Admin will see the Force Promotion option and In Pipeline, the star icon will show that a particular build is promoted and by which user.

**QA Deploy:**  QA environment deployment job. This job can call a script to perform a deployment or use tools like Jenkins, Bamboo or Urbancode.

Similarly We can chain multiple Promotion Jobs and Deploy jobs to accomplish the need for another environment, for e.g**. Dev >> QA >> Prod**

**=================================================================================================**

**4. How will you manage module version dependencies**

Dependency management is a core feature of **Maven**. Managing dependencies for a single project is easy. Managing dependencies for multi-module projects and applications that consist of hundreds of modules is possible. Maven helps a great deal in defining, creating, and maintaining reproducible builds with well-defined classpaths and library versions.

## Transitive Dependencies Discovery

It is pretty often a case, when a library, say A, depends upon other library, say B. In case another project C wants to use A, then that project requires to use library B too.

Maven helps to avoid such requirements to discover all the libraries required. Maven does so by reading project files (pom.xml) of dependencies, figure out their dependencies and so on.

We only need to define direct dependency in each project pom. Maven handles the rest automatically.

With transitive dependencies, the graph of included libraries can quickly grow to a large extent. Cases can arise when there are duplicate libraries. Maven provides few features to control extent of transitive dependencies.

|  |  |
| --- | --- |
| **Sr.No.** | **Feature & Description** |
| 1 | **Dependency mediation**  Determines what version of a dependency is to be used when multiple versions of an artifact are encountered. If two dependency versions are at the same depth in the dependency tree, the first declared dependency will be used. |
| 2 | **Dependency management**  Directly specify the versions of artifacts to be used when they are encountered in transitive dependencies. For an example project C can include B as a dependency in its dependency Management section and directly control which version of B is to be used when it is ever referenced. |
| 3 | **Dependency scope**  Includes dependencies as per the current stage of the build. |
| 4 | **Excluded dependencies**  Any transitive dependency can be excluded using "exclusion" element. As example, A depends upon B and B depends upon C, then A can mark C as excluded. |
| 5 | **Optional dependencies**  Any transitive dependency can be marked as optional using "optional" element. As example, A depends upon B and B depends upon C. Now B marked C as optional. Then A will not use C. |

## Dependency Scope

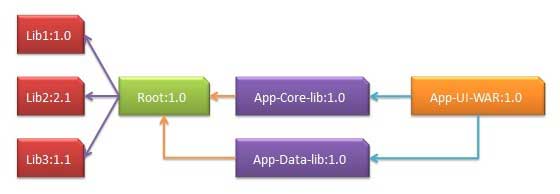
Transitive Dependencies Discovery can be restricted using various Dependency Scope as mentioned below.

|  |  |
| --- | --- |
| **Sr.No.** | **Scope & Description** |
| 1 | **compile**  This scope indicates that dependency is available in classpath of project. It is default scope. |
| 2 | **provided**  This scope indicates that dependency is to be provided by JDK or web-Server/Container at runtime. |
| 3 | **runtime**  This scope indicates that dependency is not required for compilation, but is required during execution. |
| 4 | **test**  This scope indicates that the dependency is only available for the test compilation and execution phases. |
| 5 | **system**  This scope indicates that you have to provide the system path. |
| 6 | **import**  This scope is only used when dependency is of type pom. This scope indicates that the specified POM should be replaced with the dependencies in that POM's <dependencyManagement> section. |

## Dependency Management

Usually, we have a set of project under a common project. In such case, we can create a common pom having all the common dependencies and then make this pom, the parent of sub-project's poms.

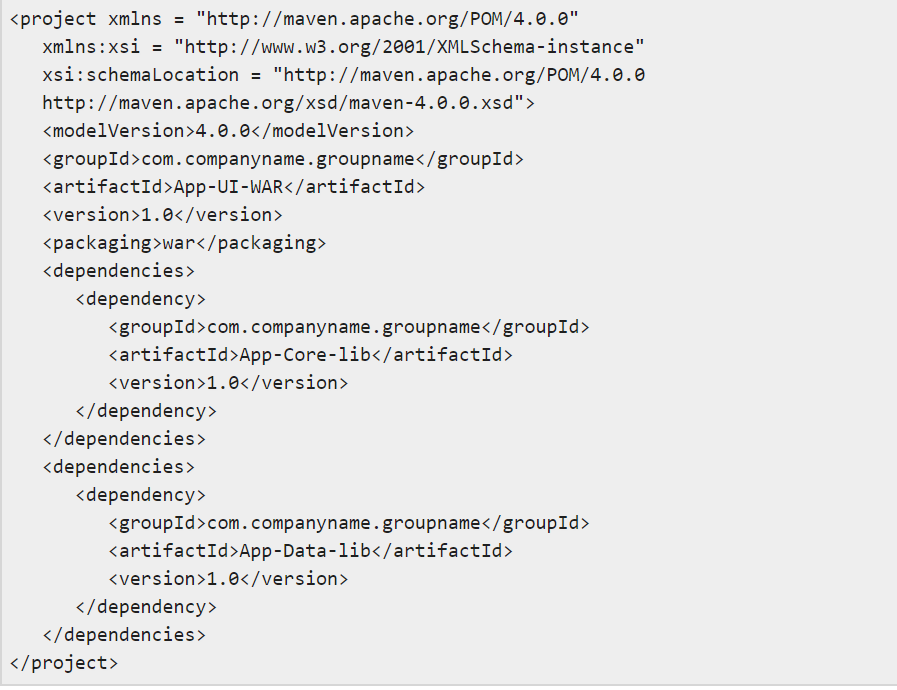
**Ex:**



Following are the detail of the above dependency graph −

* App-UI-WAR depends upon App-Core-lib and App-Data-lib.
* Root is parent of App-Core-lib and App-Data-lib.
* Root defines Lib1, lib2, Lib3 as dependencies in its dependency section.

**App-UI-WAR**



**App-Core-lib**



**App-Data-lib**



**Root**



Now when we build App-UI-WAR project, Maven will discover all the dependencies by traversing the dependency graph and build the application.

From above example, the key concepts are −

* Common dependencies can be placed at single place using concept of parent pom. Dependencies of **App-Data-lib** and **App-Core-lib** project are listed in *Root* project (See the packaging type of Root. It is POM).
* There is no need to specify Lib1, lib2, Lib3 as dependency in App-UI-WAR. Maven use the **Transitive Dependency Mechanism** to manage such detail.

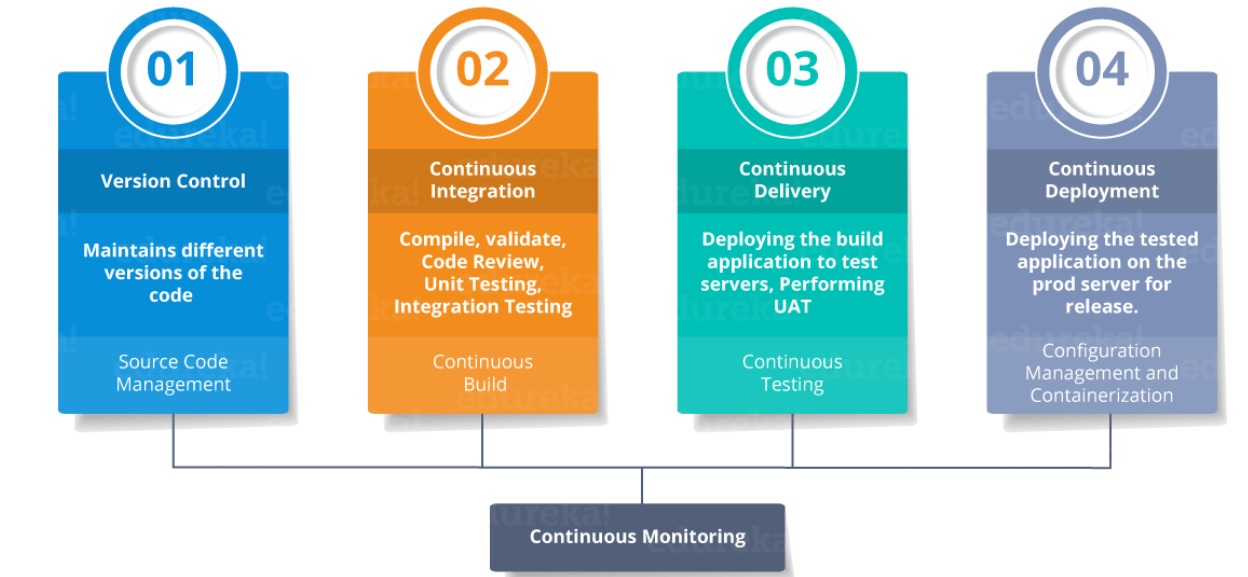
=================================================================================================

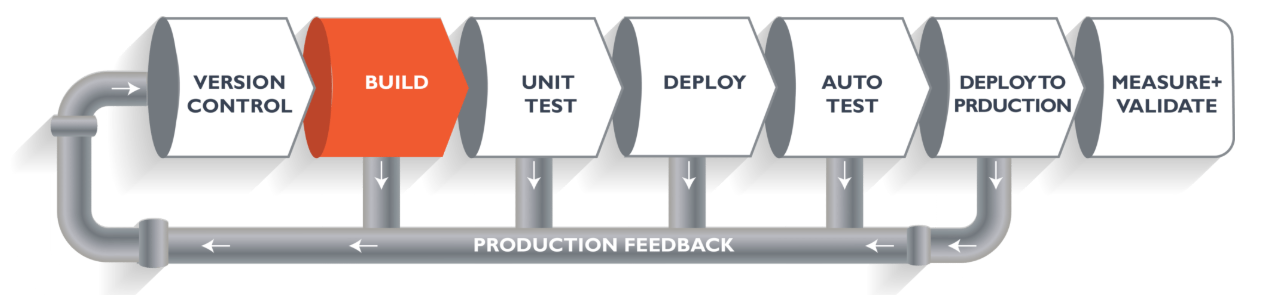
**5. Provide CI/CD implementation plans with stages**

A CI/CD Pipeline implementation, or Continuous Integration/Continuous Deployment, is the backbone of the modern DevOps environment. It bridges the gap between development and operations teams by automating the building, testing, and deployment of applications.



DevOps is a software development approach which involves continuous development, continuous testing, continuous integration, continuous deployment, and continuous monitoring of the software throughout its development lifecycle. This is the process adopted by all the top companies to develop high-quality software and shorter development lifecycles, resulting in greater customer satisfaction, something that every company wants.





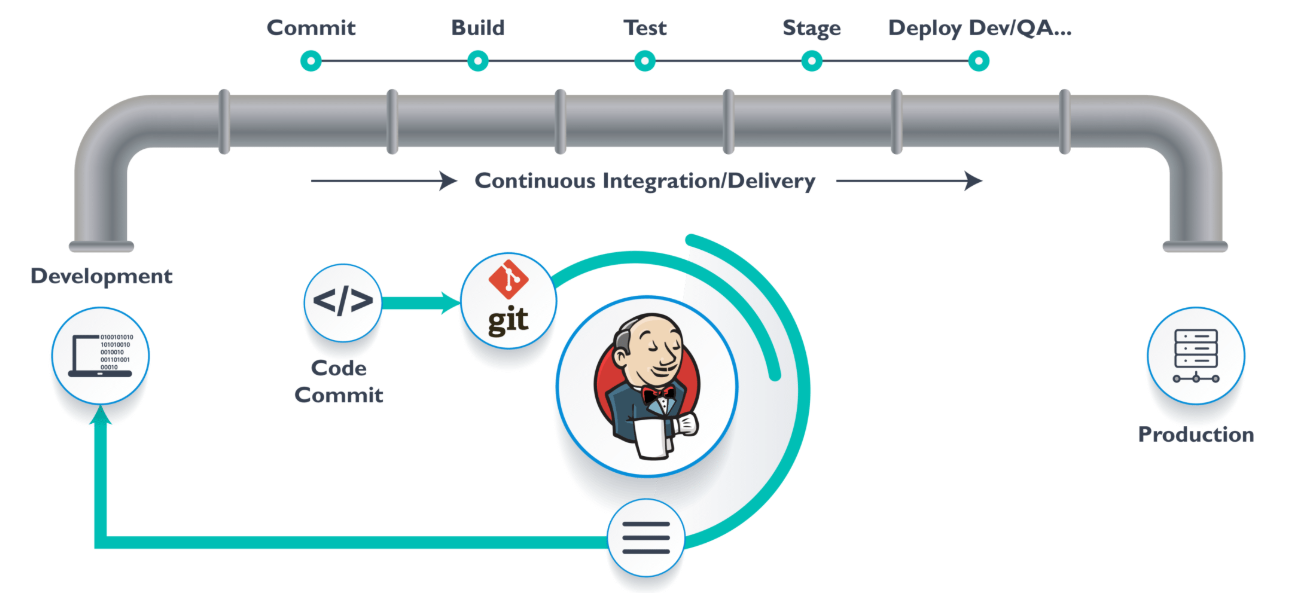
The above pipeline is a logical demonstration of how software will move along the various stages in this lifecycle before it is delivered to the customer or before it is live in production.

**Important phases to successful CI/CD pipelines:**

1. The ***Plan*** phase often combines practices from Scrum and Agile to enable frequent microincremental releases.
2. The ***Code*** phase focuses on core development tasks from within IDEs and appropriate sandboxing and frameworks.
3. The ***Build*** phase rapidly and incrementally merges code commits with some testing and security validation.
4. The ***Test*** phase focuses on automated verification of enhancements, often incorporating test-driven deployment practices. Testing is sometimes incorporated as part of the Build phase, and generally extends in some way to all phases of the CI/CD process to ensure continuous feedback and improvement.
5. The ***Release*** phase is centered around repository commits and adequate documentation of the changes.
6. The ***Deploy*** phase is the actual update to the codebase, with special thought given to issue and error avoidance.
7. The ***Operate*** phase occurs once the code is made live, and consists of monitoring and orchestration.
8. The ***Monitor & Optimize*** phase takes place parallel to the Operate phase and consists of data collection, analysis, and feedback to the start of the pipeline and to other phases as needed. In the most sophisticated environments, optimization is automated as ***continuous optimization*** (CO), an extension of CI/CD that leverages machine learning to eliminate risks and waste associated with manual infrastructure selection.

## The Ultimate CI Tool and Its Importance in the CI/CD Pipeline

Our task is to automate the entire process, from the time the development team gives us the code and commits it to the time we get it into production. We will automate the pipeline in order to make the entire software development lifecycle in DevOps/automated mode. For this, we will need automation tools



**Jenkins** provides us with various interfaces and tools in order to automate the entire process.

We have a Git repository where the development team will commit the code. Then, Jenkins takes over from there, a front-end tool where you can define your entire job or the task. Our job is to ensure the continuous integration and delivery process for that particular tool or for the particular application.

From Git, Jenkins pulls the code and then Jenkins moves it into the **commit phase**, where the code is committed from every branch. The **build phase** is where we compile the code. If it is Java code, we use tools like maven in Jenkins and then compile that code, which can be deployed to run a series of tests. These test cases are overseen by Jenkins again.

Then, it moves on to the staging server to deploy it using **Docker**. After a series of unit tests or sanity tests, it moves on to production.



[**Docker**](https://www.edureka.co/blog/docker-tutorial) is just like a virtual environment in which we can create a server. It takes a few seconds to create an entire server and deploy the artifacts we want to test.

As I mentioned earlier, you can run the entire cluster in a few seconds. We have a storage registry for images where you build your image and store it forever. You can use it anytime in any environment which can replicate itself.

**CI/CD pipeline with some stages using Jenkinsfile:**

pipeline {  
  
  environment {  
    registry = "[192.168.X.XX:5XX0/suman/myweb](http://192.168.1.81:5000/justme/myweb)"  
    dockerImage = ""  
  }  
  
  agent any  
  
  stages {  
  
    stage('Checkout Source') {  
      steps {  
        git '<https://github.com/sumandevops20/sumandevopschalenge.git>'  
      }  
    }  
  
    stage('Build image') {  
      steps{  
        script {  
          dockerImage = docker.build registry + ":$BUILD\_NUMBER"  
        }  
      }  
    }  
  
    stage('Push Image') {  
      steps{  
        script {  
          docker.withRegistry( "" ) {  
            dockerImage.push()  
          }  
        }  
      }  
    }  
  
    stage('Deploy App') {  
      steps {  
        script {  
          kubernetesDeploy(configs: "myweb.yaml", kubeconfigId: "mykubeconfig")  
        }  
      }  
    }  
  
  }  
  
}

=================================================================================================