

CI & CD Pipeline & Jenkins

Introduction

Continuous integration (CI) and continuous delivery (CD) incorporate values, set of operating principles, and collection of practices that enable application development teams to deliver changes more reliably and regularly; this is also known as CI/CD pipeline.

What is Continuous Integration?

Continuous integration is an approach in which developers merge their code into a shared repository several times a day. For verification of the integrated code, automated tests and builds are run for it.

What is Continuous Delivery?

Continuous delivery is a strategy in which the development teams ensure the software is reliable to release at any time. On each commit, the software passes through the automated testing process. If it successfully passes the testing, it is then it is said to be ready for release into the production.

What is CI/CD Pipeline?

CI is the short form for Continuous Integration, and CD is the short form for Continuous Delivery. CI/CD Pipeline is a crucial part of the modern DevOps environment. The pipeline is a deployable path which the software follows on the way to its production with Continuous Integration and Continuous Delivery practices. It is a development lifecycle for software and includes the CI/CD pipeline, which has various stages or phases through which the software passes.

Version Control Phase

In this phase of the CI/CD pipeline, the code written by the developers is committed through a version control software or systems such as git, apache subversion, and more. It controls the commit history of the software code so that it can be changed if needed.

Build Phase

This phase is the first phase of this pipeline system. Developers build their code, and then they pass their code through the version control system or software. After this, the code returns to the build phase and gets compiled.

Unit Testing and Staging

When software reaches this stage, various tests are conducted on the software. One of the main tests is the Unit test, in which the units of software are tested. After successful testing, the staging phase begins. As the software has passed the tests to reach here, it is ready to be deployed into the staging process. Here, the software code is deployed to the staging environment/server. The code can be viewed and finalized here before the final tests can be conducted on the software.

Auto Testing Phase

After passing to the staging environment, another set of automated tests are prepared for the software. Only if the software completes these tests and is proved to be deployable, then it is sent to the next phase/stage, which is the deployment phase.

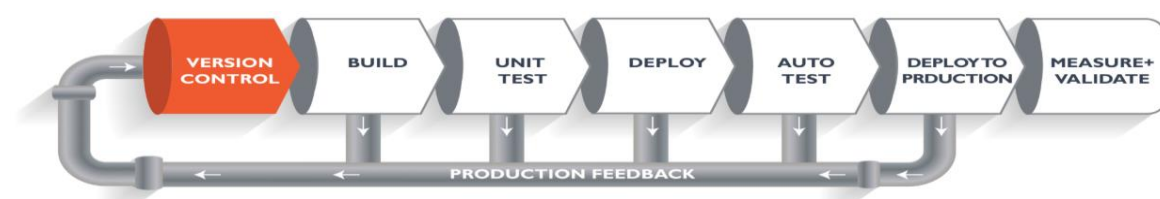
Deployment Phase

As the auto testing procedure is completed, then it is deployed to production. However, if any error occurs during the testing phase or the deployment phase, the software is sent to the version control procedure by the development team and checked for errors. If errors are found, then they need to be fixed. Other stages may be repeated if required.

CI & CD STEPS

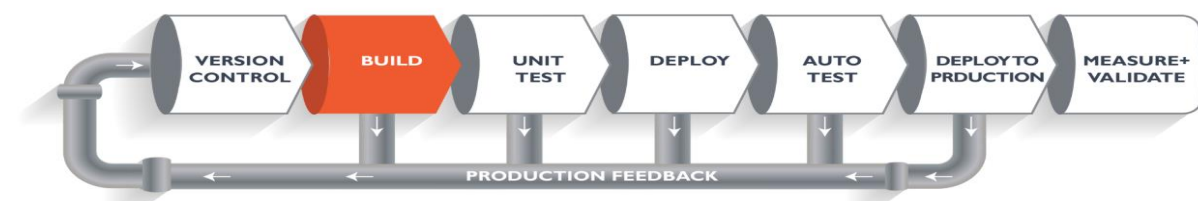
A CI/CD Pipeline implementation, or Continuous Integration/Continuous Deployment, is the backbone of the modern DevOps environment. It bridges the gap between development and operations teams by automating the building, testing, and deployment of applications. In this blog, we will learn what a CI/CD pipeline is and how it works.

CI stands for Continuous Integration and CD stands for Continuous Delivery/Continuous Deployment. You can think of it as a process like a software development lifecycle. Let us see how it works.



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.

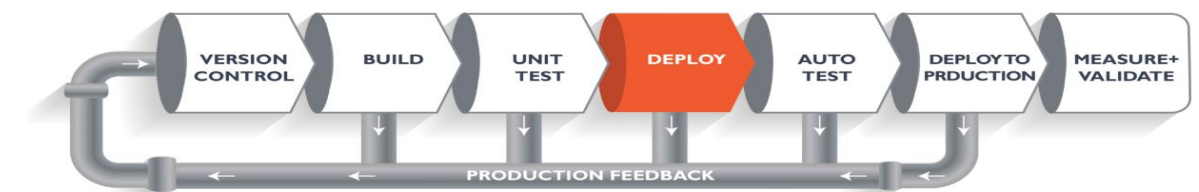
Let's take a scenario of a CI/CD Pipeline. Imagine you're going to build a web application which is going to be deployed on live web servers. You will have a set of developers responsible for writing the code, who will further go on and build the web application. Now, when this code is committed into a version control system (such as git, svn) by the team of developers. Next, it goes through the **build phase**, which is the first phase of the pipeline, where developers put in their code and then again, the code goes to the version control system with a proper version tag.



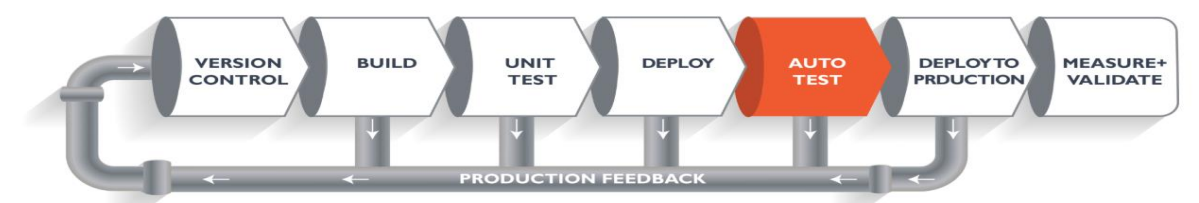
Suppose we have Java code and it needs to be compiled before execution. Through the version control phase, it again goes to the build phase, where it is compiled. You get all the features of that code from various branches of the repository, which merge them and finally use a compiler to compile it. This whole process is called the **build phase**.



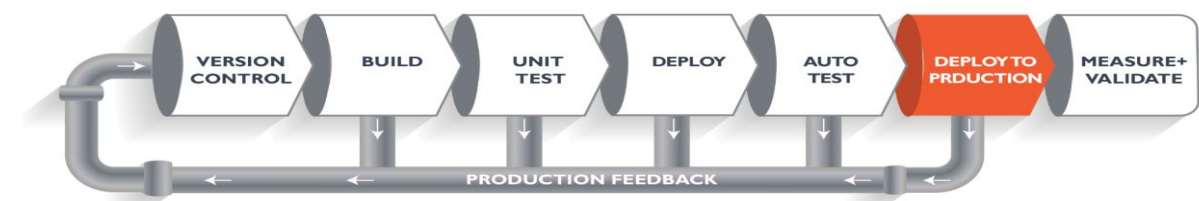
Once the build phase is over, then you move on to the **testing phase**. In this phase, we have various kinds of testing. One of them is the *unit test* (where you test the chunk/unit of software or for its sanity test).



When the test is completed, you move on to the **deploy phase**, where you deploy it into a staging or a test server. Here, you can view the code, or you can view the app in a simulator.

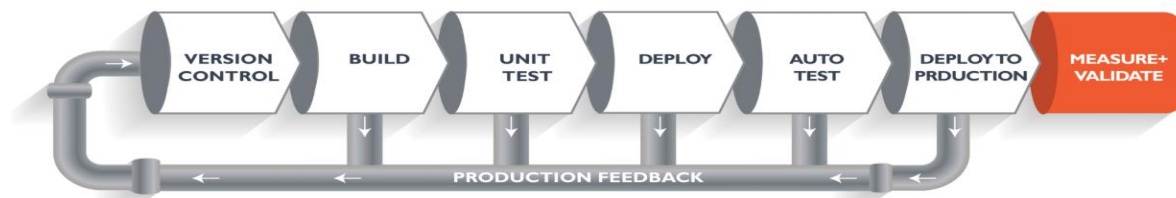


Once the code is deployed successfully, you can run another sanity test. If everything is accepted, then it can be deployed to production.



Meanwhile, in every step, if there is an error, you can shoot an email back to the development team so that they can fix it. Then they will push it into the version control system, and it goes back into the pipeline.

Once again, if there is any error reported during testing, the feedback goes to the dev team again, where they fix it and the process reiterates if required.

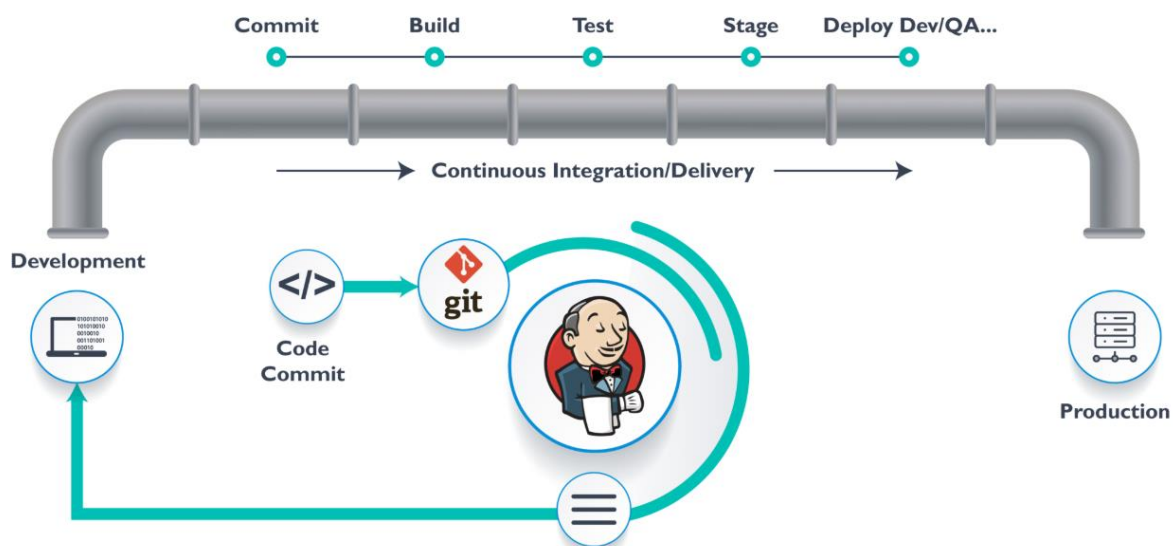


This lifecycle continues until we get code/a product which can be deployed to the production server where we measure and validate the code.

We now understand the CI/CD Pipeline and its working; now, we will move on to understand what Jenkins is and how we can deploy the demonstrated code using Jenkins and automate the entire process.

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.

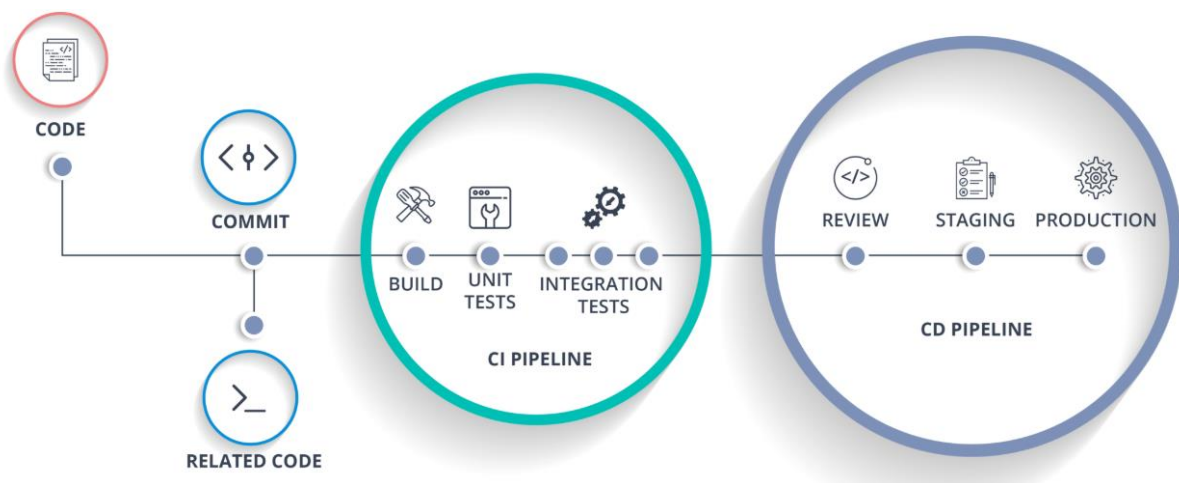


Jenkins provides us with various interfaces and tools in order to automate the entire process.

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 tool or for the 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.



What are the Continuous Integration and Continuous Delivery tools?

The CI/CD process needs to be automated to get the best results. Various tools help us in automating the process so that it can be done precisely and with the least effort. These tools are mostly open-source and are designed to help with collaborative software development. One of tool is Jenkins.

Jenkins

Jenkins is the most used tool for the CI/CD process. It is one of the earliest and most powerful CI tools. It has various interfaces and inbuilt tools, which help us in the automation of the CI/CD process. At first, it was introduced as a part of a project named Hudson, which was released in 2005. Then it was officially released as Jenkins, in 2011. It has a vast plugin ecosystem, which helps in delivering the features that we need.

- **Increased Flexibility**

With CI/CD, the errors are found quickly, and the product can be released more frequently. The flexibility to add new features increases. With automation, new changes can be adopted quickly and reliably.

What are the common pitfalls of CI/CD?

To shift from traditional models to DevOps, the existing organizations need to go through a transition process, which can be a long and difficult one. This process can take months, and even more if you don't follow the right transition steps. The steps to adopt CI/CD, the steps can be prioritized based on the following points-

- ✓ *The repetition frequency of the process.*
- ✓ *The dependencies involved in the process & delay produced by them.*
- ✓ *Length of the process.*
- ✓ *The urgency in process automation.*
- ✓ *If the process is prone to errors if not automated.*
- ✓ *These points can help in choosing processes to automate based on the priority.*

Confusion between Continuous Deployment and Continuous Delivery:

Many organizations fail to distinguish between continuous deployment and continuous delivery. They are two very different concepts. In the case of continuous deployment, the changes made in the code repository are passed through the pipeline, and if it is successful, the changes are immediately deployed to production. While in the case of continuous delivery, the changes made in code are built, tested, and then pushed to a non-production environment. In Continuous Deployment, deployment to the production environment is done without manual approval.

Inadequate coordination between continuous integration and continuous delivery

Continuous Delivery (CD) is the next step for Continuous integration (CI). They are two different items. But the implementation of CI/CD takes the collaboration of these two. Collaboration and communication cannot be automated.

What are the Best Practices in CI/CD?

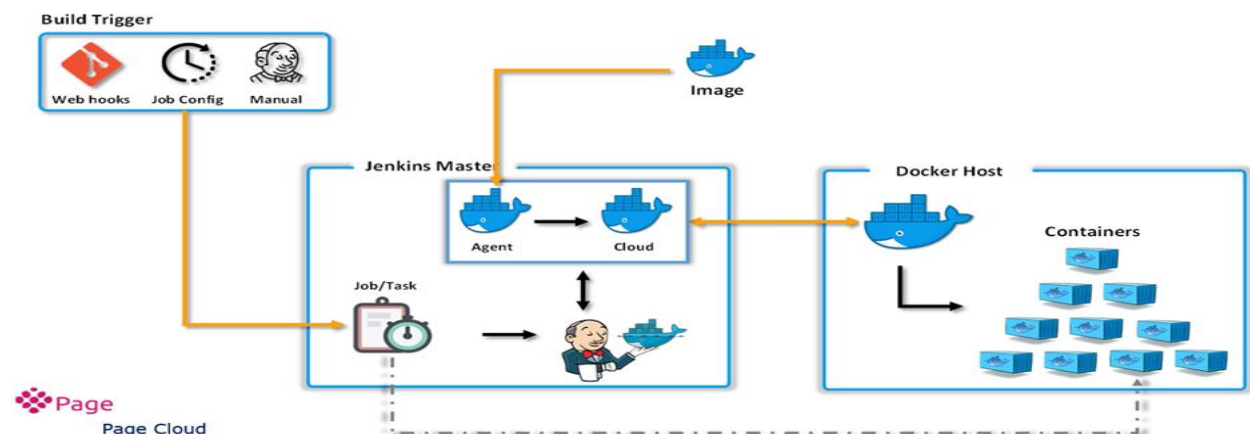
There are some practices in the case of the CI/CD process, which greatly enhance the performance of the process, and adhering to them can help in avoiding some common problems. These practices are:

- ✓ CI/CD should be made the only way to deploy to production.
- ✓ Fastest tests should be the earliest to run.
- ✓ Running the tests locally before committing at CI/CD pipeline.

Conclusion:

CI/CD is among the best practices for the devops teams to implement. Additionally, it's an agile methodology; it facilitates the development team to achieve the business requirements, best code quality, and security because the steps of deployment are automated.

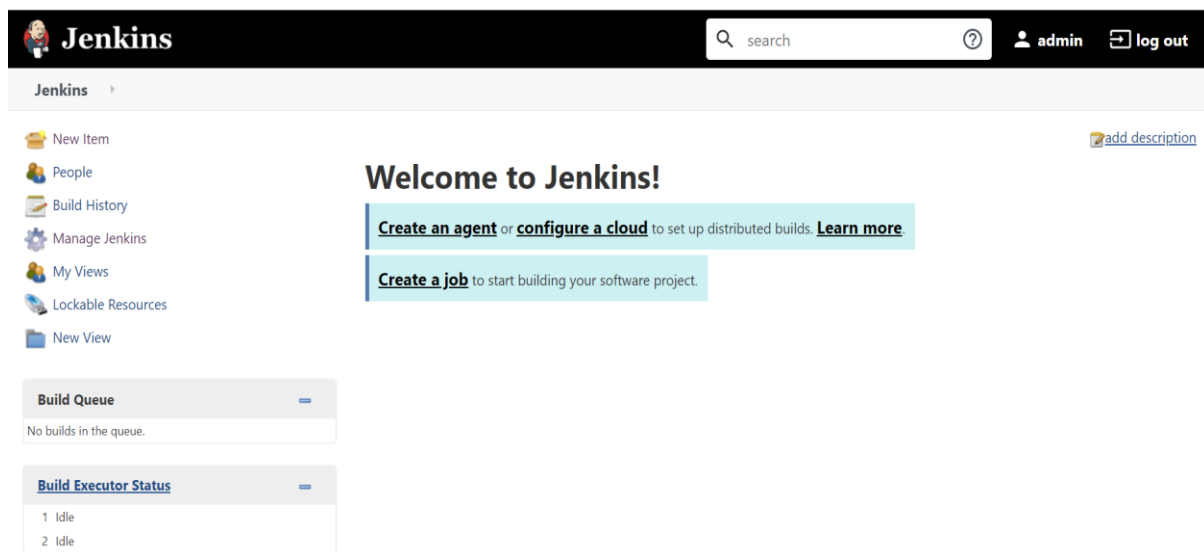
Our Target:



Jenkins Installation on CentOS7:

```
# wget http://pkg.jenkins.io/redhat/jenkins-2.252-1.1.noarch.rpm
# rpm -ivh jenkins-2.252-1.1.noarch.rpm
# yum install jenkins java-1.8.0-openjdk -y
# systemctl start jenkins
# systemctl enable jenkins
# firewall-cmd --permanent --add-port=8080/tcp --permanent
# firewall-cmd --permanent --add-service=http --permanent
# firewall-cmd --reload
```

Welcome to Jenkins!



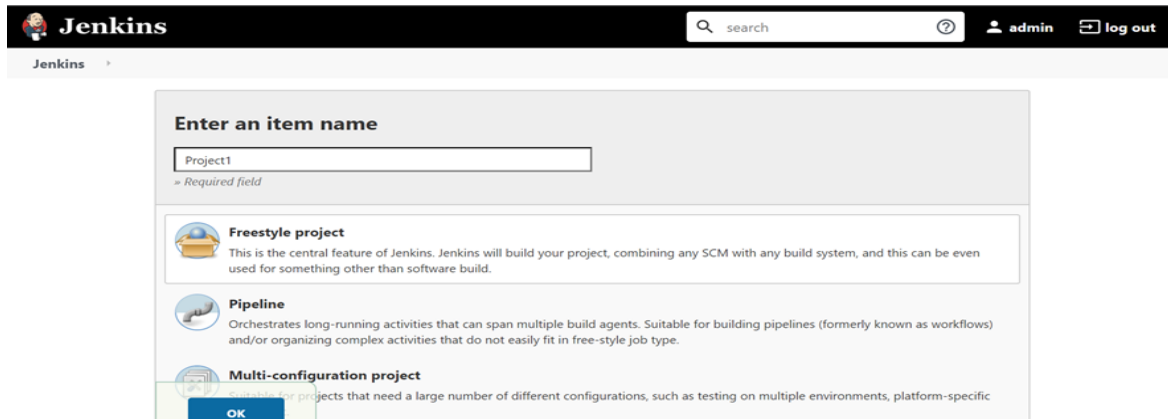
How to Create & Run a Basic job in jenkins

Steps:

1. Create Job

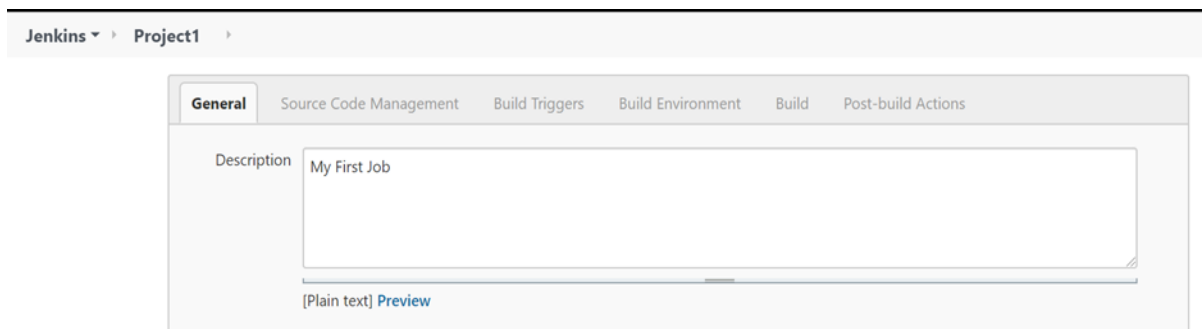


2. Enter an Item name
 - > FreeStyle Project.
 - > OK



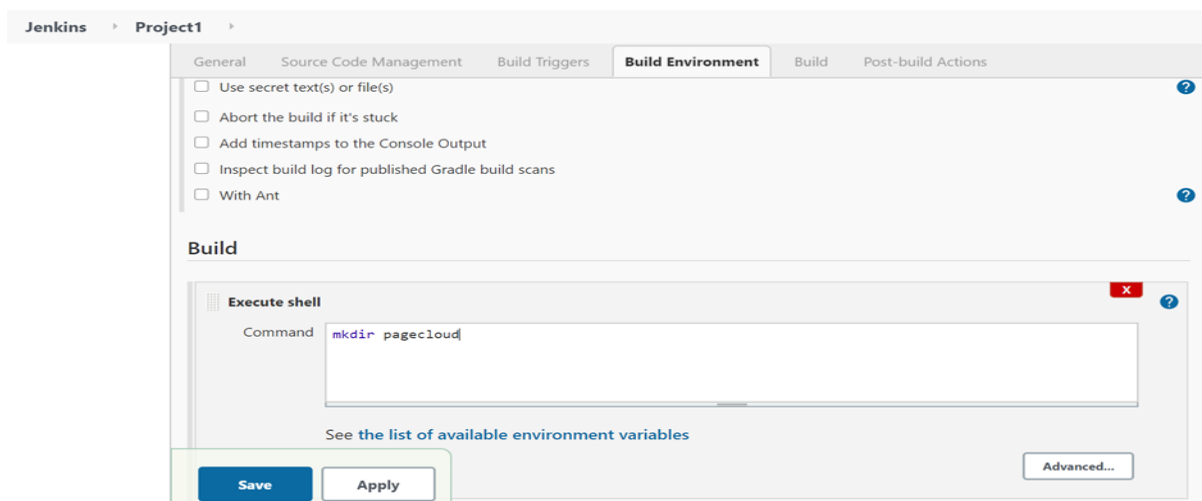
The screenshot shows the Jenkins 'Enter an item name' dialog. At the top, there's a search bar and user information (admin, log out). Below the title 'Enter an item name', there's a text input field containing 'Project1' with a '*' icon and the text '* Required field' below it. Underneath, there are three options: 'Freestyle project' (selected), 'Pipeline', and 'Multi-configuration project'. Each option has a brief description. At the bottom, there's a blue 'OK' button.

3. General > Description.



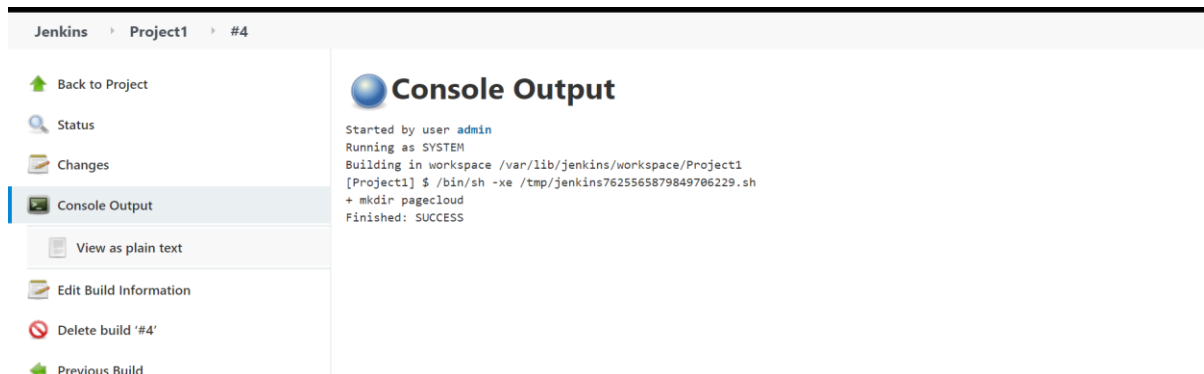
The screenshot shows the Jenkins 'Project1' configuration page, specifically the 'General' tab. The 'Description' field is a large text area containing 'My First Job'. Below the text area, there's a '[Plain text] Preview' link. The top navigation bar shows 'Jenkins > Project1' and the tabs 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'.

4. Build
 - > Add Build Step
 - > Execute Shell
 - > Command [mkdir pagecloud]
 - > Save & Apply.



The screenshot shows the Jenkins 'Project1' configuration page, specifically the 'Build Environment' tab. It has several checkboxes: 'Use secret text(s) or file(s)', 'Abort the build if it's stuck', 'Add timestamps to the Console Output', 'Inspect build log for published Gradle build scans', and 'With Ant'. Below these is the 'Build' section with a 'Execute shell' step. The 'Command' field contains 'mkdir pagecloud'. At the bottom, there are 'Save' and 'Apply' buttons, and an 'Advanced...' link.

5. Go to Dashboard
6. Click Project right and Build Now
7. Check the Console Output.

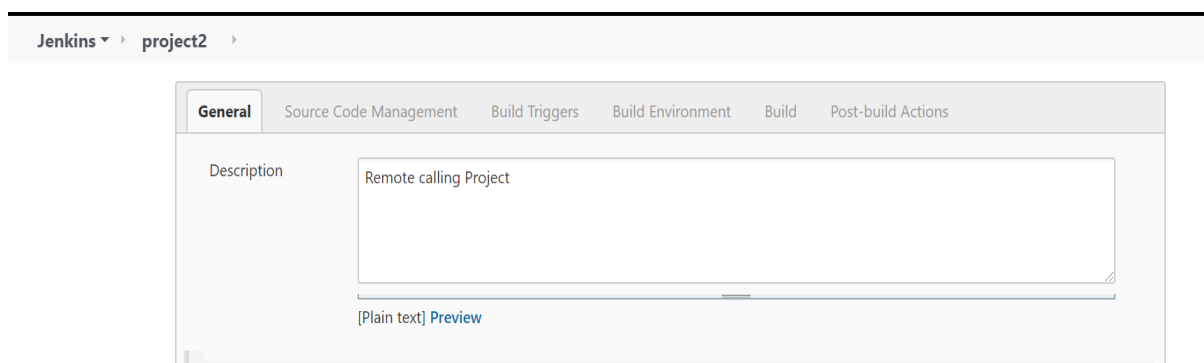


How to run the job remotely:

1. Create a Job



2. Enter an Item name
>FreeStyle Project.
>OK
3. General > Description.



4 Build Triggers

- > Auth Token



The image shows the 'Build Triggers' configuration page in Jenkins. The 'Trigger builds remotely (e.g., from scripts)' checkbox is checked. Below it, the 'Authentication Token' field contains the value '1234'. A text box provides instructions on how to use the token in a URL, showing examples like `JENKINS_URL/job/project2/build?token=TOKEN_NAME` and `/buildWithParameters?token=TOKEN_NAME`. It also mentions that one can optionally append `&cause=Cause+Text`. Below these instructions, there are four unchecked checkboxes: 'Build after other projects are built', 'Build periodically', 'GitHub hook trigger for GITScm polling', and 'Poll SCM'. Each checkbox has a help icon (a question mark in a blue circle) to its right.

Build Triggers

☒ Trigger builds remotely (e.g., from scripts) ?

Authentication Token

Use the following URL to trigger build remotely: `JENKINS_URL/job/project2/build?token=TOKEN_NAME` or `/buildWithParameters?token=TOKEN_NAME`

Optionally append `&cause=Cause+Text` to provide text that will be included in the recorded build cause.

☐ Build after other projects are built ?

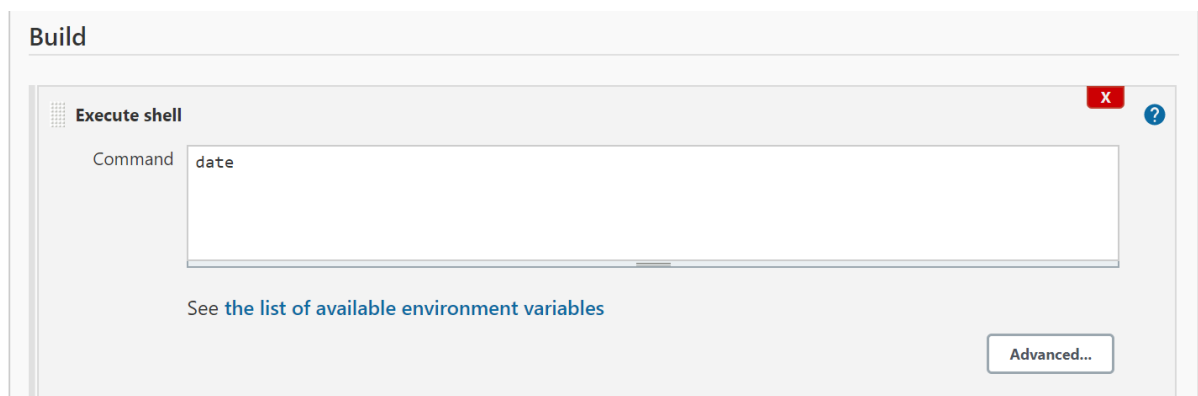
☐ Build periodically ?

☐ GitHub hook trigger for GITScm polling ?

☐ Poll SCM ?

5 Build

- > Add Build Step
- > Execute Shell
- > Command [date]
- > Save & Apply.



The image shows the 'Build' configuration page in Jenkins. The 'Execute shell' step is selected. The 'Command' field contains the value 'date'. Below the command field, there is a link to 'See the list of available environment variables'. At the bottom right, there is an 'Advanced...' button. A red 'X' icon and a help icon (a question mark in a blue circle) are visible in the top right corner of the configuration area.

Build

Execute shell X ?

Command

See [the list of available environment variables](#)

Advanced...

6. Go to Dashboard
7. Go to Browser and hit the url with auth token
8. Check the Console Output.

URL: <http://192.168.0.108:8080/job/project2/build?token=1234>

How to Create & Run a chain job in Jenkins

1. Create a Project

- > New Item.
 - > Job01
 - > Free Style.
 - > OK.
 - > General TAB > Description.
 - > Source Code Management > None.
 - > Build > Execute Shell.
- Apply & Save.

2. Create a Project.

- > New Item.
 - > Job02
 - > Free Style.
 - > OK.
 - > General TAB > Description.
 - > Source Code Management > None.
 - > Build Triggers:
 - > Build after other projects are built.
 - Projects to watch > Job01.
 - > Build > Execute Shell.
 - > Build Other Project
 - > Add post-build actions
 - > Projects to build > Job03
- Apply & Save.

1. Create a Project

- > New Item.
 - > Job03
 - > Free Style.
 - > OK.
 - > General TAB > Description.
 - > Source Code Management > None.
 - > Build > Execute Shell.
- Apply & Save.