



**Build Automation &  
CI/CD with Jenkins**

## Key Takeaways

# Introduction to Build Automation

# What is Build Automation? - 1

Build automation is the **process of automating** the



*retrieval of source code*



*executing automated tests*



*compiling into binary code/build docker image*



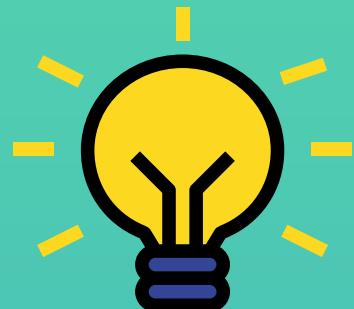
*push artifact to repository*



*deploy artifact*



*...*



Build Automation and CI/CD  
is in the heart of DevOps

# What is Build Automation? - 2



Doing this **manually** each time would be **super inconvenient!**



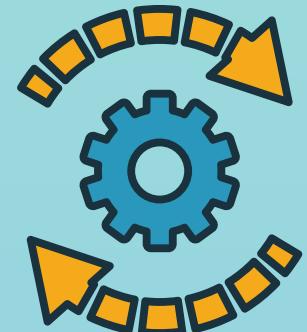
- stash working changes and move to main branch
- login to correct docker repository
- set up the test environment to run tests
- execute tests (meanwhile you CAN'T code)
- build Docker Image and push to repository



So we have a **dedicated server** executing this

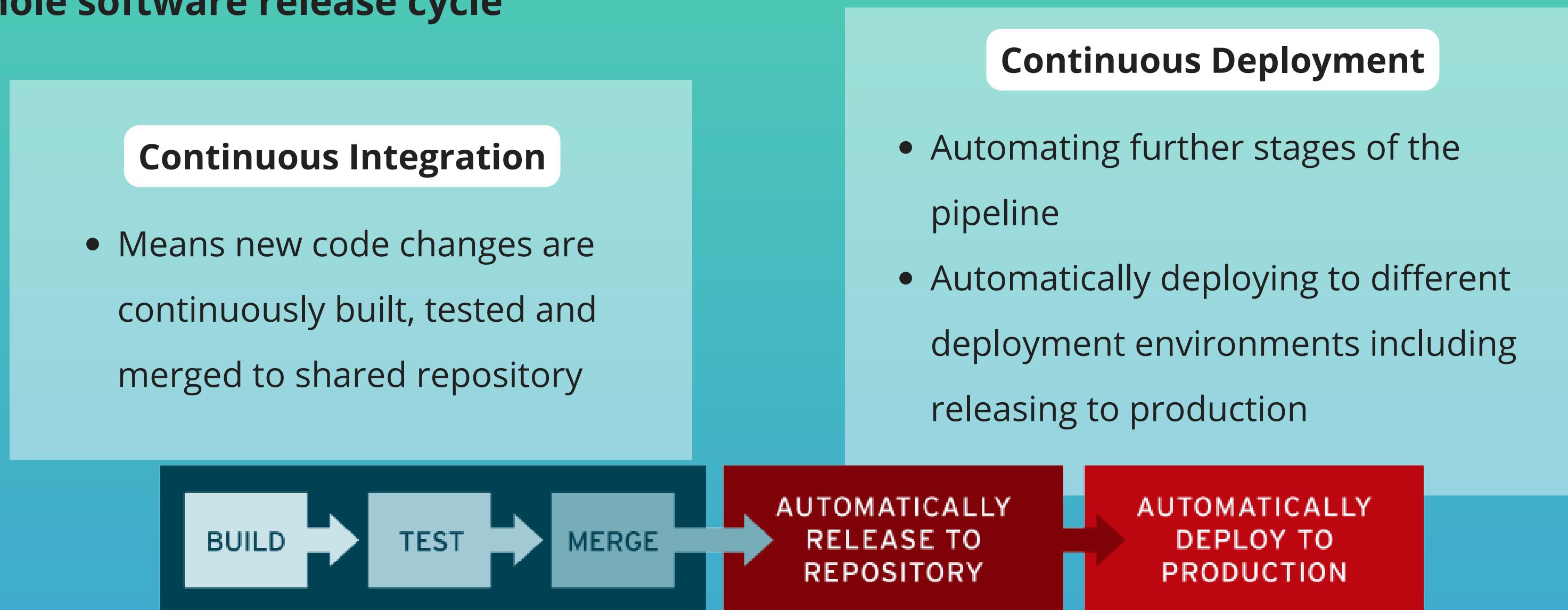


- Test environment prepared
- Docker credentials configured
- All necessary tools installed
- And trigger this build automatically



# What is CI / CD ?

- CI/CD stands for **Continuous Integration / Continuous Deployment**
- Goal of CI/CD is to "release early and often". And the way to do this, is by **automating the whole software release cycle**



- Build automation is a basic component of CI/CD pipelines

# Jenkins = most used in the industry

- Jenkins is currently one of the most used build automation CI/CD tool in the industry



Many others available:



Travis CI

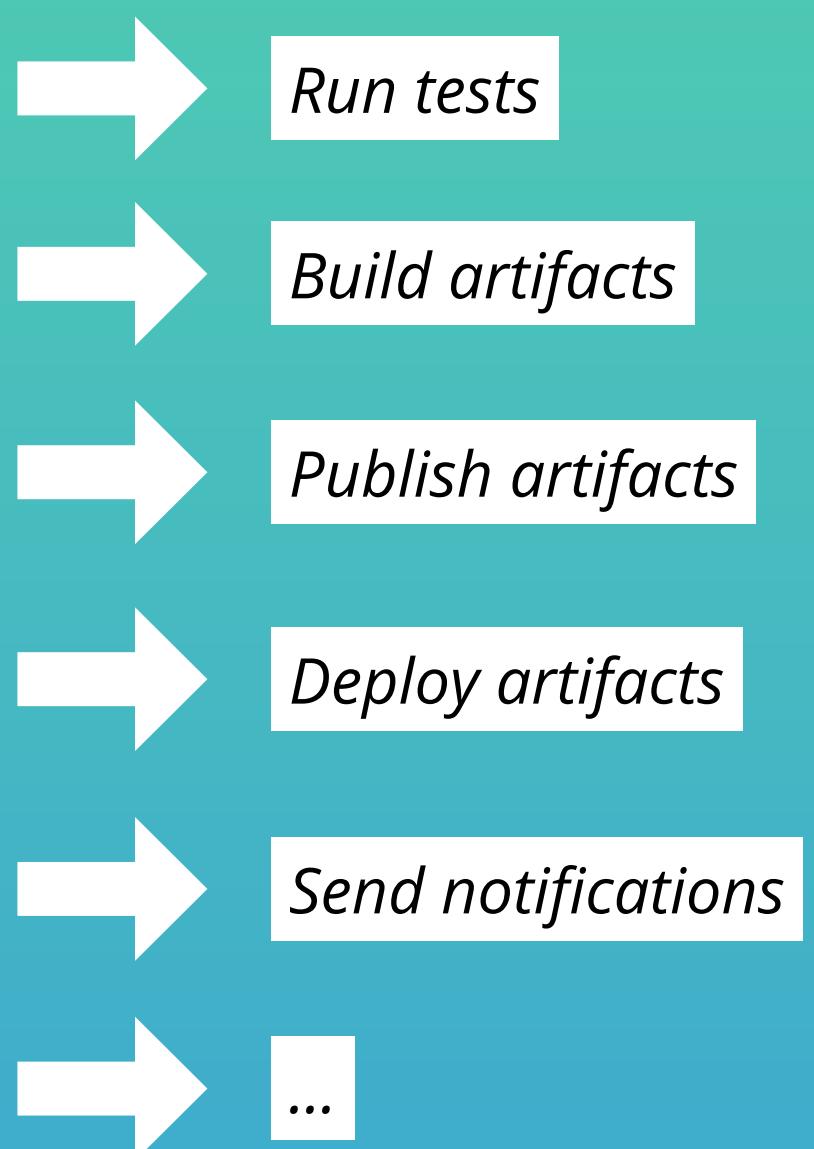


TeamCity

# Introduction to Jenkins - 1

- 1.Jenkins is a software that you **install on a dedicated server**
- 2.It has a **UI** to configure your builds
- 3.You need to **install all the tools** you need to get the tasks done (Docker, Gradle/Maven/npm, etc.)
- 4.Then you need to **configure the tasks** (run tests, build app, deployment etc.)
- 5.And **configure the automatic trigger** of the workflow

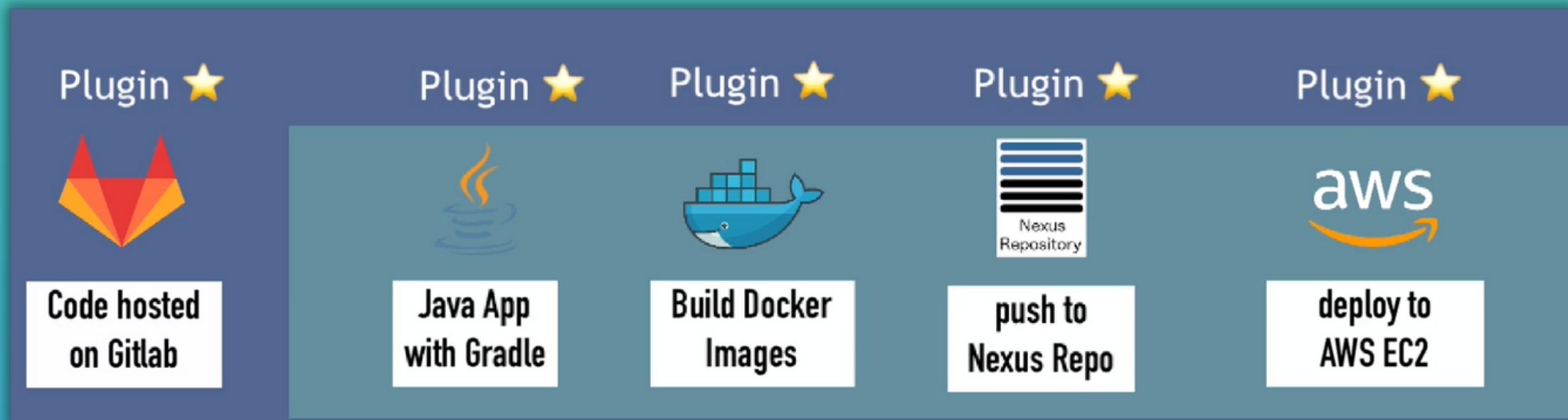
## What you can do with Jenkins



# Introduction to Jenkins - 2

One of most important capabilities a build automation needs to have is  
**its integration with other technologies**

- Jenkins makes this integrations possible with **Plugins**. Plugin = **extension to Jenkins functionality**
- There are plugins for Docker, Build Tools, Repositories, Deployment Servers etc.



- Plugins help you to execute each of those steps

# Introduction to Jenkins - 3

- **DevOps Engineer task is to administer Jenkins** (plugins, setup etc.), create or help developers to set up build automation jobs in Jenkins

## How it works:

RUN TESTS



Build Tools need to be available

BUILD ARTIFACTS



Build Tools or Docker need to be available

PUBLISH ARTIFACTS



Store credentials in Jenkins



Jenkins User must have access to all these technologies and platforms

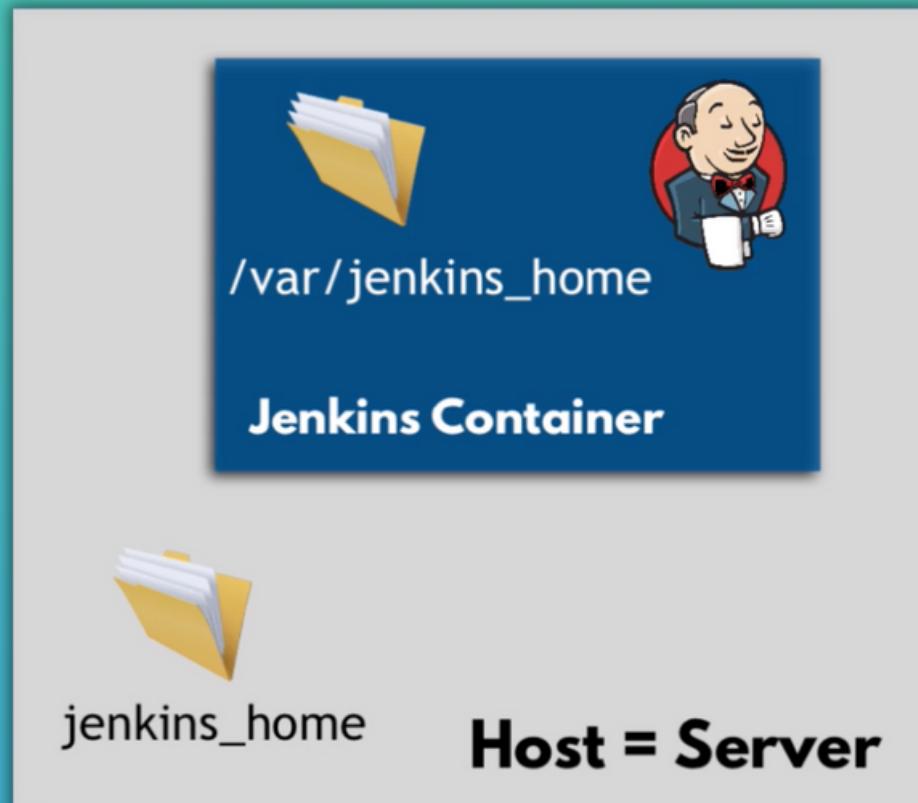
## What to configure:

- Install Jenkins and prepare everything
- The setup needs to be done only once
- Plugins, Credentials etc. can be used for different projects

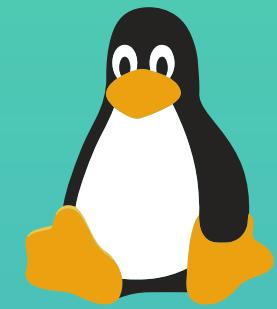
# Jenkins Installation

2 ways to install Jenkins on a server:

1) Run Jenkins as Docker container



2) Install Jenkins directly on OS



```
Bootcamp — root@ubuntu-s-2vcpu-2gb-fra1-01: ~ — ssh root@207.154.235.187 — 95x26
[root@ubuntu-s-2vcpu-2gb-fra1-01: ~]# docker run -p 8080:8080 -p 50000:50000 -d \
> -v jenkins_home:/var/jenkins_home jenkins/jenkins:lts
```

# Getting Started

Getting Started

## Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

`/var/jenkins_home/secrets/initialAdminPassword`

Please copy the password from either location and paste it below.

**Administrator password**

Enter password

Getting Started

## Getting Started

✓ Folders	✓ OWASP Markup Formatter	✓ Build Timeout	✓ Credentials Binding
✓ Timestamper	✓ Workspace Cleanup	✓ Ant	✓ Gradle
✓ Pipeline	✓ GitHub Branch Source	✓ Pipeline: GitHub Groovy Libraries	✓ Pipeline: Stage View
✓ Git	✓ SSH Build Agents	○ Matrix Authorization Strategy	○ PAM Authentication
LDAP	○ Email Extension	✓ Mailer	

\*\* Pipeline: Declarative Extension Points API  
\*\* JSch dependency  
\*\* Git client  
\*\* GIT server  
\*\* Pipeline: Shared Groovy Libraries  
\*\* Branch API  
\*\* Pipeline: Multibranch  
\*\* Pipeline: Stage Tags Metadata  
\*\* Pipeline: Declarative  
\*\* Lockable Resources  
Pipeline  
\*\* OkHttp  
\*\* GitHub API  
Git  
\*\* GitHub  
GitHub Branch Source  
Pipeline: GitHub Groovy Libraries  
Pipeline: Stage View  
Git  
SSH Build Agents  
\*\* - required dependency

Welcome to Jenkins!

Create an agent or configure a cloud to set up distributed builds. [Learn more](#).

Create a job to start building your software project.

Some plugins are installed automatically

Successfully logged in

## Jenkins Administrator

### Operations or DevOps teams

- **Administers** and manages Jenkins
- Sets up Jenkins cluster
- Installs plugins
- Backups Jenkins data
- ...

The screenshot shows the Jenkins System Configuration interface. On the left, there's a sidebar with links for People, Build History, Manage Jenkins, My Views, Lockable Resources, and New View. Below that are sections for Build Queue (empty), Build Executor Status (1 Idle, 2 Idle), and Status Information (System Information, System Log, Load Statistics). The main content area is titled "System Configuration" and contains several management options:

- Configure System**: Configure global settings and paths.
- Global Tool Configuration**: Configure tools and their locations and automatic installers.
- Manage Plugins**: Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
- Manage Nodes and Clouds**: Add, remove, control and monitor the various nodes that Jenkins runs jobs on.
- Configure Global Security**: Secure Jenkins; define who is allowed to access/use the system.
- Manage Credentials**: Configure credentials.
- Configure Credential Providers**: Configure the credential providers and types.
- Manage Users**: Create/delete/modify users that can log in to this Jenkins.

# Roles

## Jenkins User

### Developers or DevOps teams

- Creating the **actual jobs** to run workflows

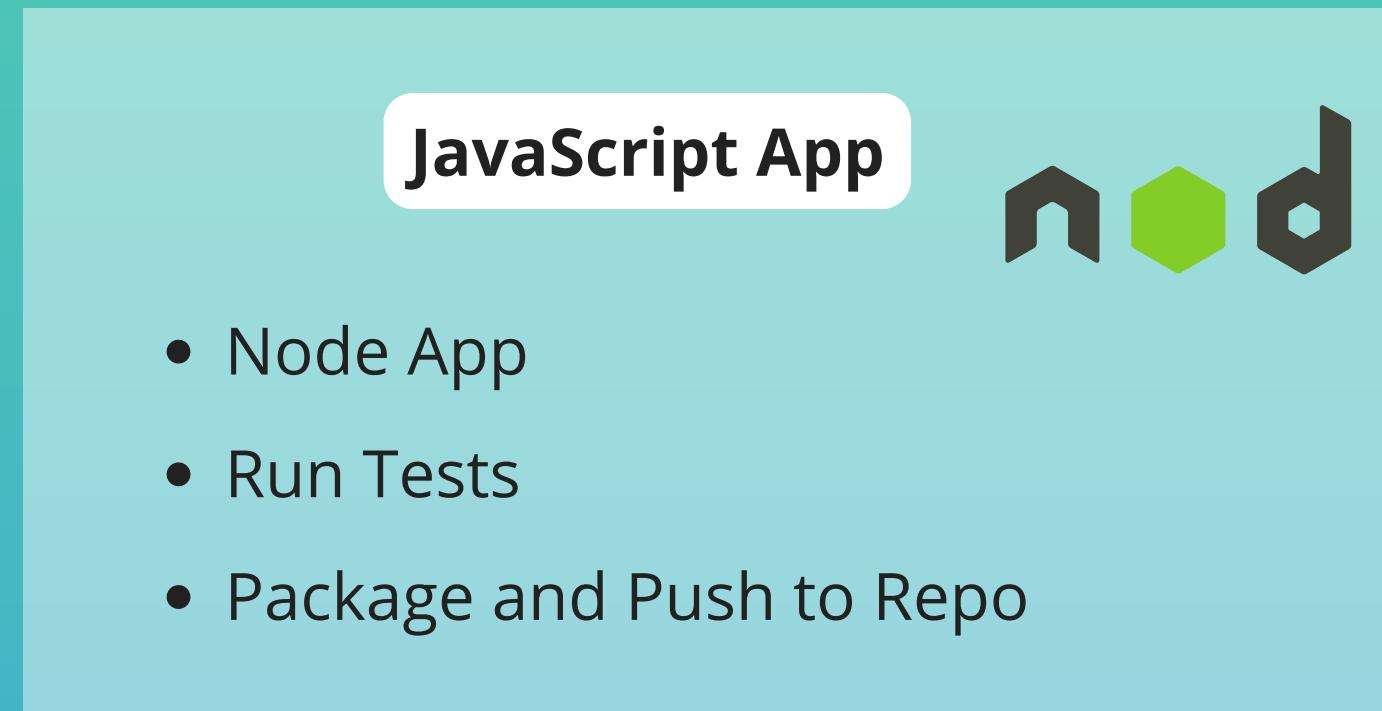
The screenshot shows the Jenkins "Enter an item name" page. It features a search bar with the placeholder "» Required field". Below it is a list of project types with icons and descriptions:

- Freestyle project**: This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can even be used for something other than software build.
- Pipeline**: Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
- Multi-configuration project**: Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform specific builds, etc.
- Folder**: Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.
- GitHub Organization**: Scans a GitHub organization (or user account) for all repositories matching some defined markers.
- Multibranch Pipeline**: Creates a set of Pipeline projects according to detected branches in one SCM repository.

# Deep Dive into Jenkins

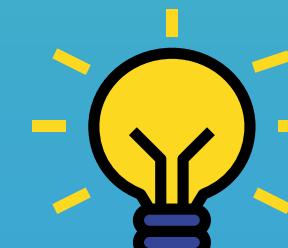
# Install Build Tools in Jenkins - 1

- Jenkins is used to **build applications**
- So first we need to **install and configure the build tools** in Jenkins



→ ***Maven needs to be available on Jenkins***

→ ***npm needs to be available on Jenkins***



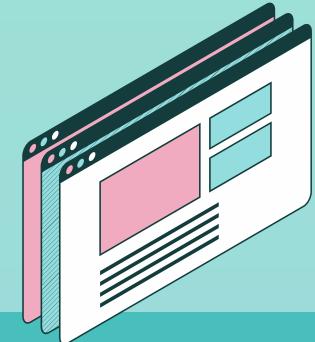
Depending on your app (**programming language**) you need to have **different tools** installed and configured on Jenkins

# Install Build Tools in Jenkins - 2

- **2 ways** to install and configure those tools:

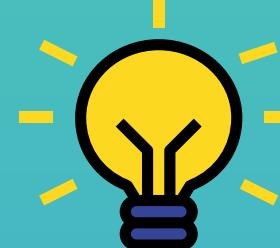
## 1) Jenkins Plugins

- Just install plugin (via UI) for your tool

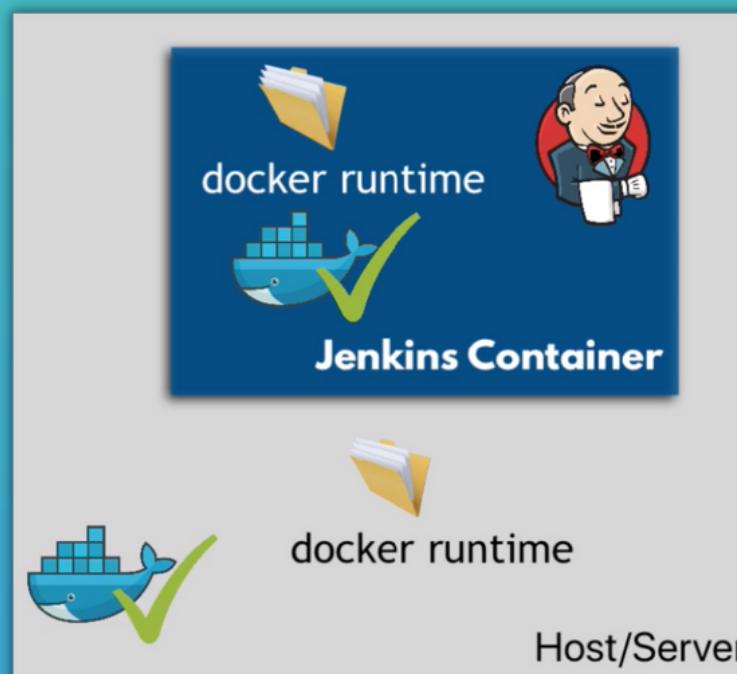


## 2) Install Tools directly on server

- Enter in remote server and install
- Inside the Docker container, when Jenkins runs as container



How to make Docker available in Jenkins container

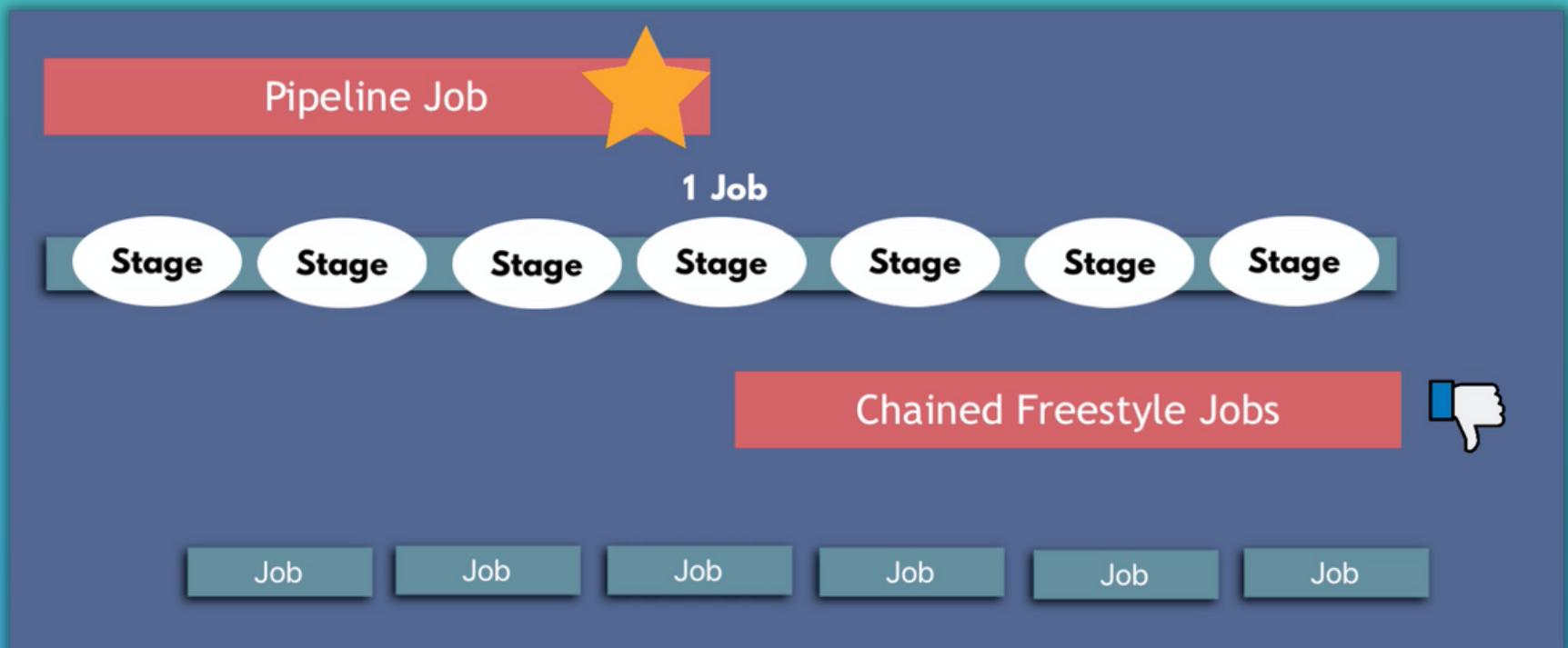


- To execute Docker commands inside Jenkins, we need Docker available. One way is to **mount Docker from host into Jenkins**:

```
root@ubuntu-s-2vcpu-4gb-fra1-01:~# docker run -p 8080:8080 -p 50000:50000 -d \
> -v jenkins_home:/var/jenkins_home \
> -v /var/run/docker.sock:/var/run/docker.sock \
> -v $(which docker):/usr/bin/docker jenkins:lts
```

# Jenkins Job Types - 1

- **Build jobs** are at the heart of Jenkins build process
- A job is a task or step in your build process
- "Job" is a deprecated term, but still mostly used (called "Project" now)
- The **2 of 3 most important jobs and its differences:**



## Freestyle job

- To orchestrate **simple jobs** for a project
- Configuration: Via UI
- Very limited, limited to the input fields of plugin

## Pipeline job

- For more complex workflows, so **suitable for CI/CD pipelines**
- Ability to divide the tasks into different stages
- Configuration: UI and Scripting (Pipeline as Code)
- Much more powerful

# Jenkins Job Types - 2

- More advantages or use cases for Pipeline job:

- ✓ You want to execute 2 tasks in parallel
- ✓ Usage of user input or conditional statements
- ✓ Set variables
- ✓ ...

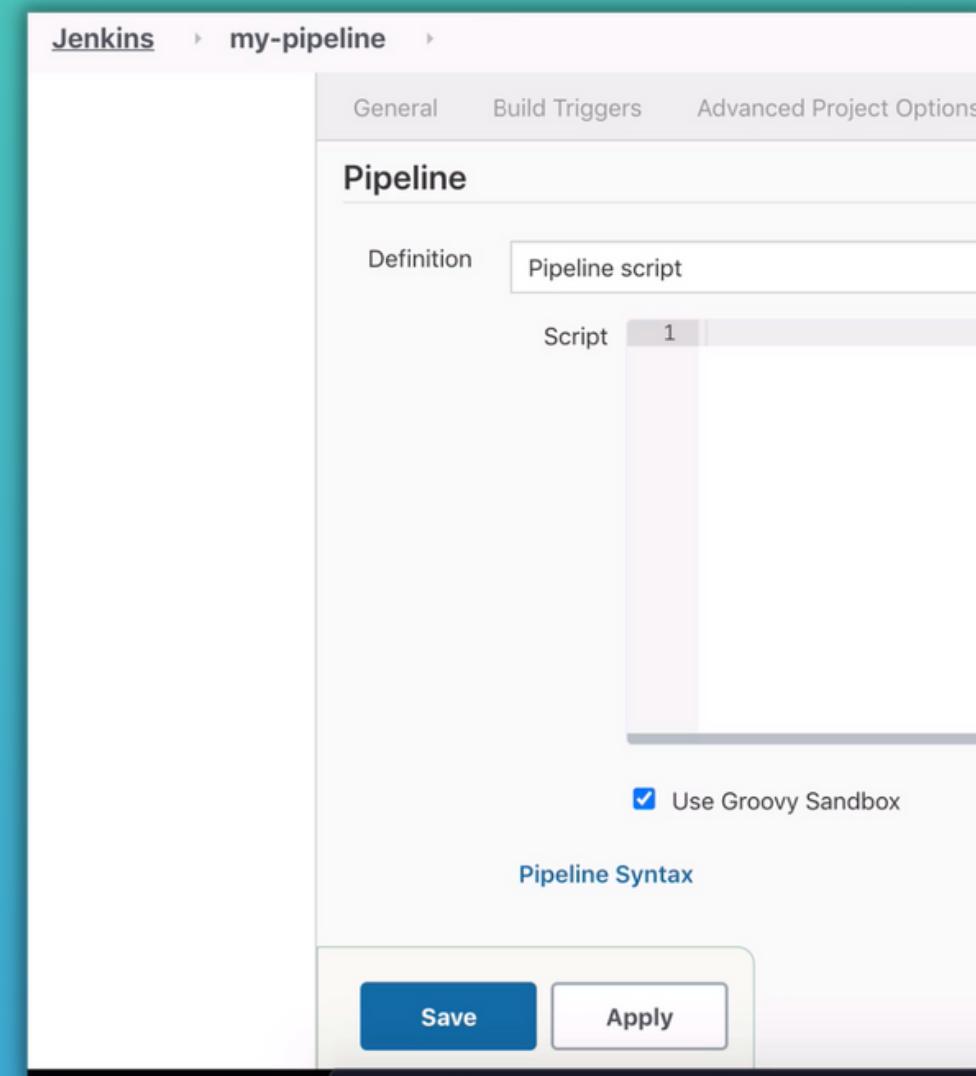


```
1 CODE_CHANGES = getGitChanges()  
2 pipeline {  
3     agent any  
4     stages {  
5         stage("build") {  
6             when {  
7                 expression {  
8                     BRANCH_NAME == 'dev' && CODE_CHANGES == true  
9                 }  
10            }  
11            steps {  
12                echo 'building the application...'  
13            }  
14        }  
15    }  
16}
```

# "Pipeline as Code"

- With Pipeline job you have the ability to **script your pipeline** instead of UI configuration
- "Pipeline as Code", which is current best practice in industry: "**Everything as Code**"  
(Infrastructure as Code, Configuration as Code, Policy as Code, Security as Code, ...)

1) Write code directly in UI:



2) Write code in a **Jenkinsfile**:



# Jenkinsfile

- Jenkinsfile is a **text file that contains the definition of a Jenkins Pipeline**
- Is checked into project's git repository
- **2 ways to write a Jenkinsfile:**

## Scripted

- First syntax
- Groovy engine
- Advanced scripting capabilities, high flexibility
- Difficult to start

## Declarative Pipeline

- Easier to get started, but not that powerful
- **Pre-defined structure**

```
Y dev / Jenkinsfile
1 pipeline {
2   agent any
3   stages {
4     stage("build") {
5       steps {
6       }
7     }
8   }
9 }
10 }
11 }
12 }
13 }
14 }
15 }
```

# Jenkinsfile Syntax - 1

```
Y dev / Jenkinsfile

1 pipeline {
2
3     agent any
4
5     stages {
6
7         stage("build") {
8             steps {
9
10            }
11        }
12    }
13
14 }
15
16 node {
17     // groovy script
18 }
```

## Required fields of Jenkinsfile

- "**pipeline**" - must be top-level
- "**agent**" - where to execute
- "**stages**" - where the "work" happens
  - "stage" and "steps"

# Jenkinsfile Syntax - 2

## post

- Execute some logic **AFTER** all stages executed
- Available **conditions**: Execute "always", only on build "success" or build "failure"

```
20
21     stage("deploy") {
22         steps {
23             echo 'deploying the application...'
24         }
25     }
26 }
27 }
28 post {
29     always {
30         // ...
31     }
32     failure {
33     }
34 }
35 }
36 }
37 }
```

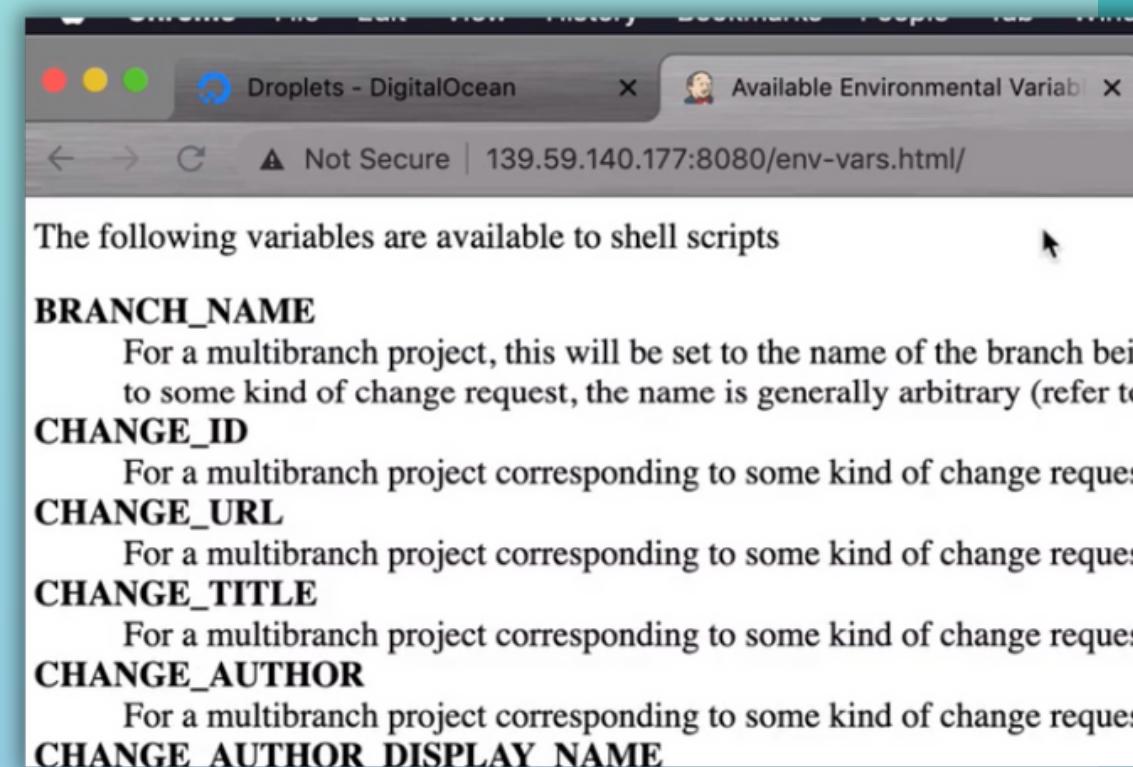
## Conditions

- **Logical control flow** in stages of what to execute under which condition

```
agent any
stages {
    stage("build") {
        steps {
            echo 'building the application...'
        }
    }
    stage("test") {
        when {
            expression {
                BRANCH_NAME == 'dev'
            }
        }
        steps {
            echo 'testing the application...'
        }
    }
    stage("deploy") {
        steps {
            echo 'deploying the application...'
        }
    }
}
```

## Environment Variables

- Jenkins offers information on build. some out of the box through **ENV VARS**
- See available env vars at */env-vars.html*



### BRANCH\_NAME

For a multibranch project, this will be set to the name of the branch being built. In some kind of change request, the name is generally arbitrary (refer to the CHANGE\_ID variable).

### CHANGE\_ID

For a multibranch project corresponding to some kind of change request.

### CHANGE\_URL

For a multibranch project corresponding to some kind of change request.

### CHANGE\_TITLE

For a multibranch project corresponding to some kind of change request.

### CHANGE\_AUTHOR

For a multibranch project corresponding to some kind of change request.

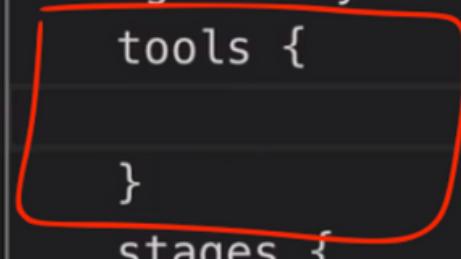
### CHANGE\_AUTHOR\_DISPLAY\_NAME

# Jenkinsfile Syntax - 3

## tools

- To access build tools for your projects, like npm, gradle, maven

```
❶ Jenkinsfile ●  
❷  
❸ 1 pipeline {  
❹ 2     agent any  
❺ 3     tools {  
❻ 4         }  
❽ 5     stages {  
❾ 6         stage("build") {  
❿ 7             steps {  
❻ 8                 echo 'building the  
❽ 9             }
```



## parameters

- You can parameterize your build to make it more re-usable
- **Types of parameter:**

- string, choice, booleanParam



```
❶ Jenkinsfile ●  
❷  
❸ 1 pipeline {  
❹ 2     agent any  
❺ 3     parameters {  
❻ 4         string(name: 'VERSION', defaultValue: '', description: 'version to deploy on prod')  
❽ 5         choice(name: 'VERSION', choices: ['1.1.0', '1.2.0', '1.3.0'], description: '')  
❽ 6         booleanParam(name: 'executeTests', defaultValue: true, description: '')  
❽ 7     }  
❽ 8     stages {  
❽ 9         stage("build") {
```

# Jenkinsfile Syntax - 4

## Using external Groovy script

- For complete flexibility you can write any groovy script and include it in the Jenkinsfile



```
1 def gv
2
3 pipeline {
4     agent any
5     parameters {
6         choice(name: 'VERSION', choices: ['1.1.0', '1.2.0'])
7         booleanParam(name: 'executeTests', defaultValue: true)
8     }
9     stages {
10         stage("init") {
11             steps {
12                 script {
13                     gv = load "script.groovy"
14                 }
15             }
16         }
17         stage("build") {
18             steps {
19                 script {
20                     gv.buildApp()
21                 }
22         }
```

```
1 def buildApp() {
2     echo 'building the application...'
3 }
4
5 def testApp() {
6     echo 'testing the application...'
7 }
8
9 def deployApp() {
10    echo 'deploying the application...'
11    echo "deploying version ${params.VERSION}"
12 }
13
14 return this
```

# Multibranch Pipeline

- Enables you to implement **different Jenkinsfiles** for **different branches** of the same project
- Jenkins automatically discovers, manages and executes Pipelines for branches, which **contain a Jenkinsfile** in Git repository (no need to manually create the Pipeline)

The screenshot shows the Jenkins Multibranch Pipeline interface for a project named "my-multi-branch-pipeline". On the left, there's a sidebar with options like "Up", "Status", "Configure", "Scan Multibranch Pipeline Now", "Scan Multibranch Pipeline Log", "Multibranch Pipeline Events", "Delete Multibranch Pipeline", and "People". The main area displays "Branches (3)" with the following data:

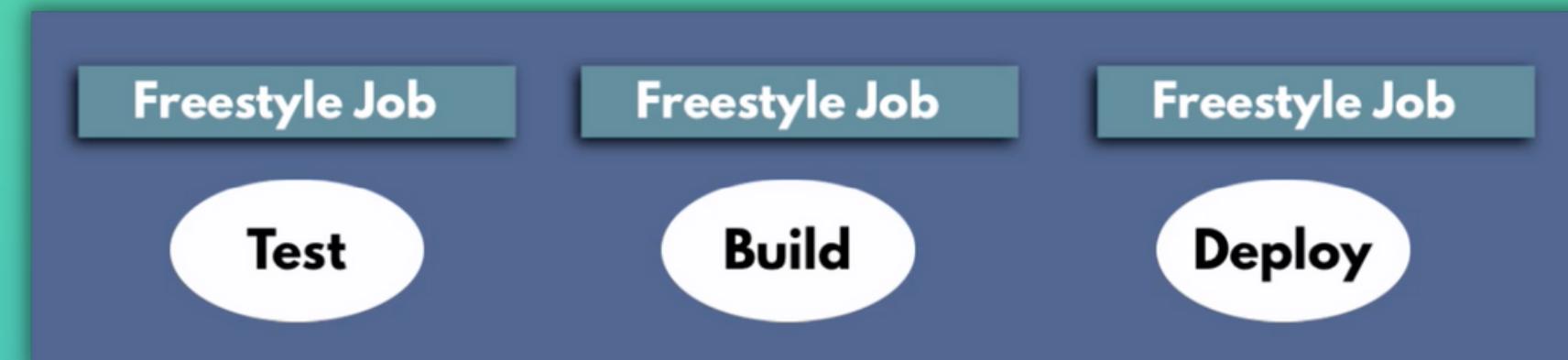
S	W	Name ↓	Last Success	Last Failure	Last Duration
🔴	🌧	bugfix/jenkins-pipeline	N/A	2 min 14 sec - #1	2.1 sec
🔵	☀️	jenkins-jobs	2 min 14 sec - #1	N/A	23 sec
⚪	☀️	master	N/A	N/A	N/A

Below the interface is a diagram illustrating the Multibranch Pipeline concept. It shows a red box labeled "Multibranch Pipeline" with three arrows pointing to three teal boxes labeled "Pipeline". To the right, there's a diagram of a Jenkins pipeline structure with nodes for "feature-1" and "bugfix-1" merging into a "master" node, each with its own "Jenkinsfile" icon.

# Summary of all 3 Job Types

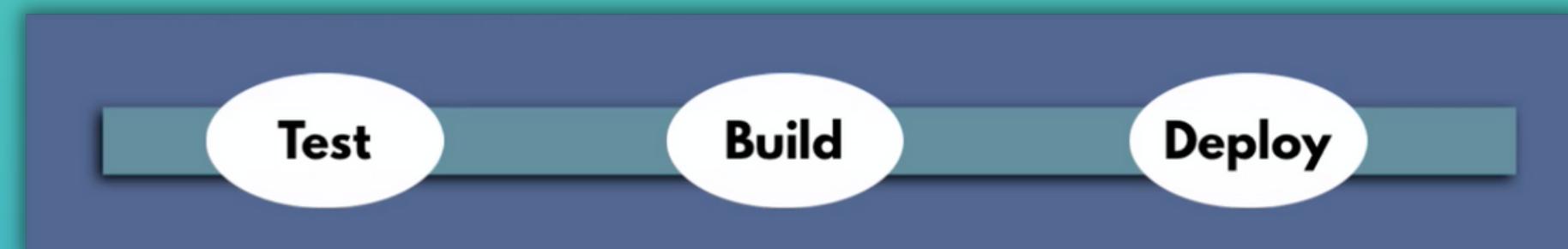
## Freestyle

- Executing a single task



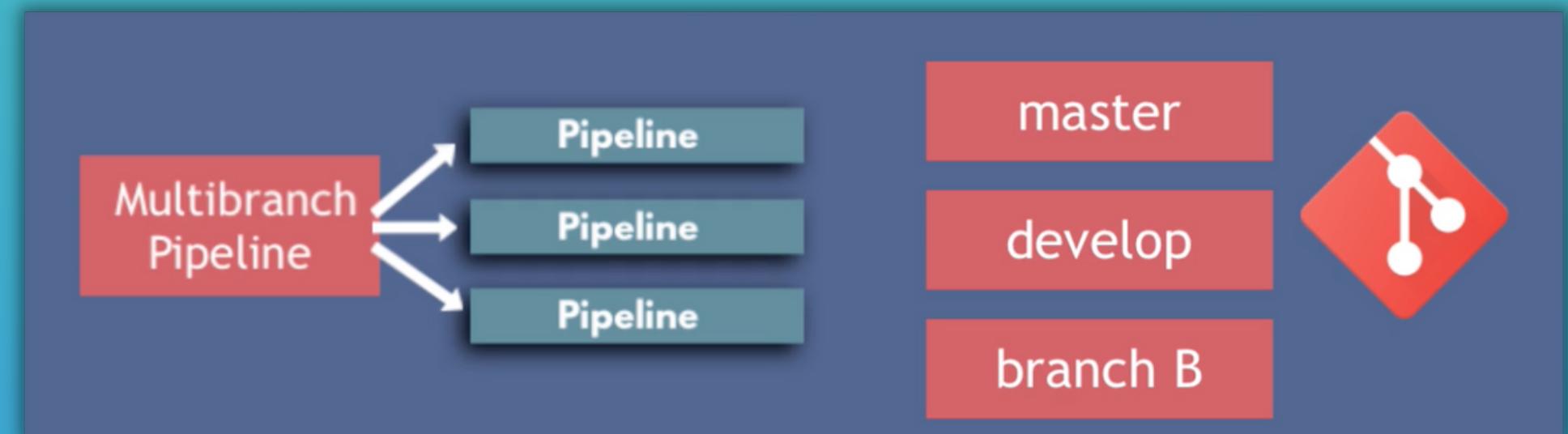
## Pipeline

- Better solution for CI/CD
- 1 branch



## Multibranch Pipeline

- Parent of Pipeline Jobs



# Jenkins Important Terms



- **Job** = A deprecated term, synonymous with Project
- **Project** = A user-configured description of work which Jenkins should perform, such as building a piece of software, etc.
- **Build** = Result of a single execution of a Project
- **Pipeline** = A user-defined model of a continuous delivery pipeline
- **Plugin** = An extension to Jenkins functionality provided separately from Jenkins Core

# Credentials - 1

Jenkins **needs credentials for all the tasks** in the pipeline, like

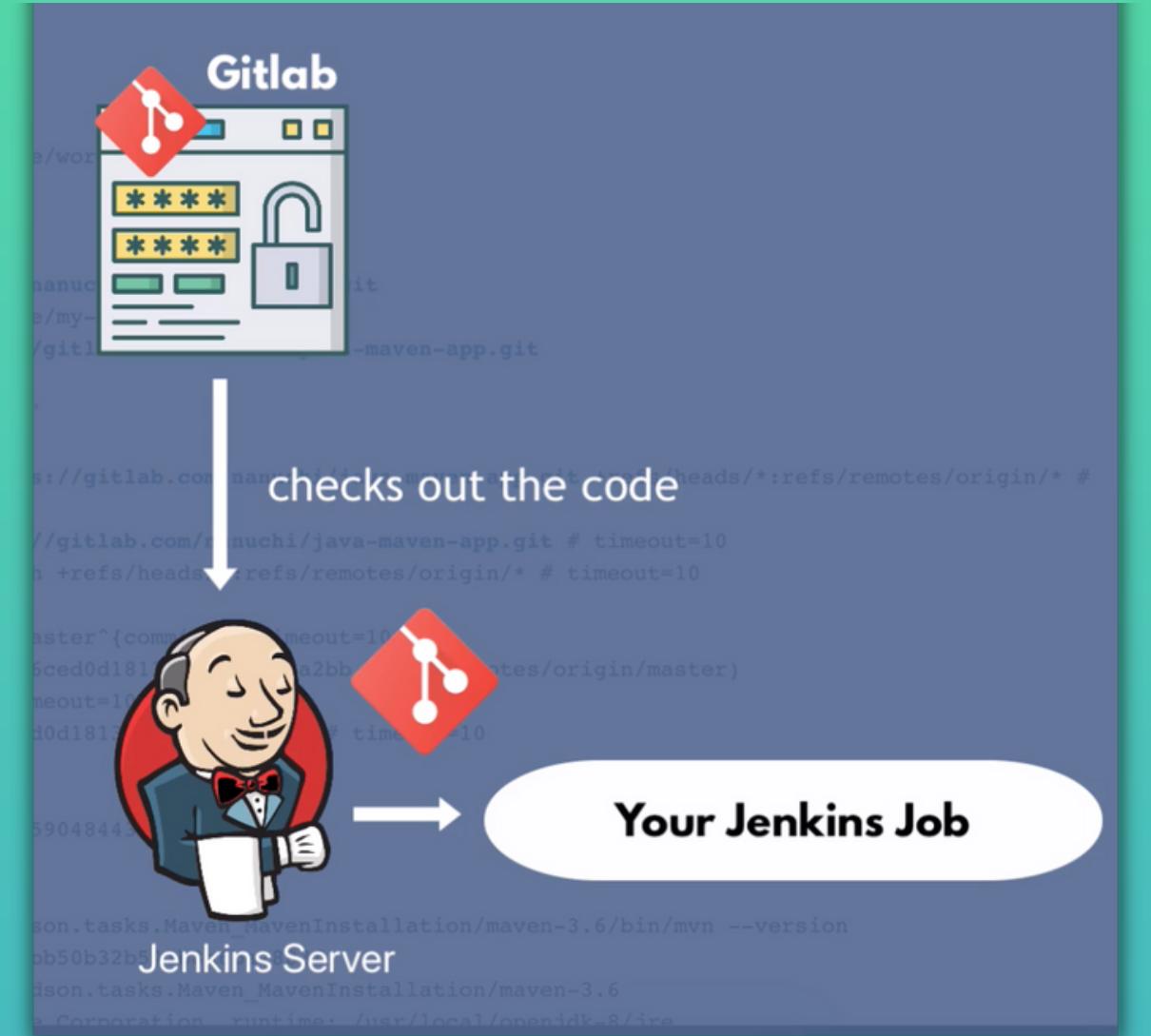
- fetch code of git repository,
- login to docker registry to push images,
- SSH to remote server to deploy

## Credentials Scopes

Credentials in Jenkins have a **scope**:

- **System:** Only available on Jenkins server, NOT for Jenkins jobs
- **Global:** Everywhere accessible

The screenshot shows the Jenkins global credentials configuration interface. The URL is `Jenkins > Credentials > System > Global credentials (unrestricted)`. The 'Kind' dropdown is set to 'Username with password'. The 'Scope' dropdown is set to 'Global (Jenkins, nodes, items, all child items, etc)', which is highlighted with a red border. Below these are fields for 'Username' and 'Password'.



# Credentials - 2

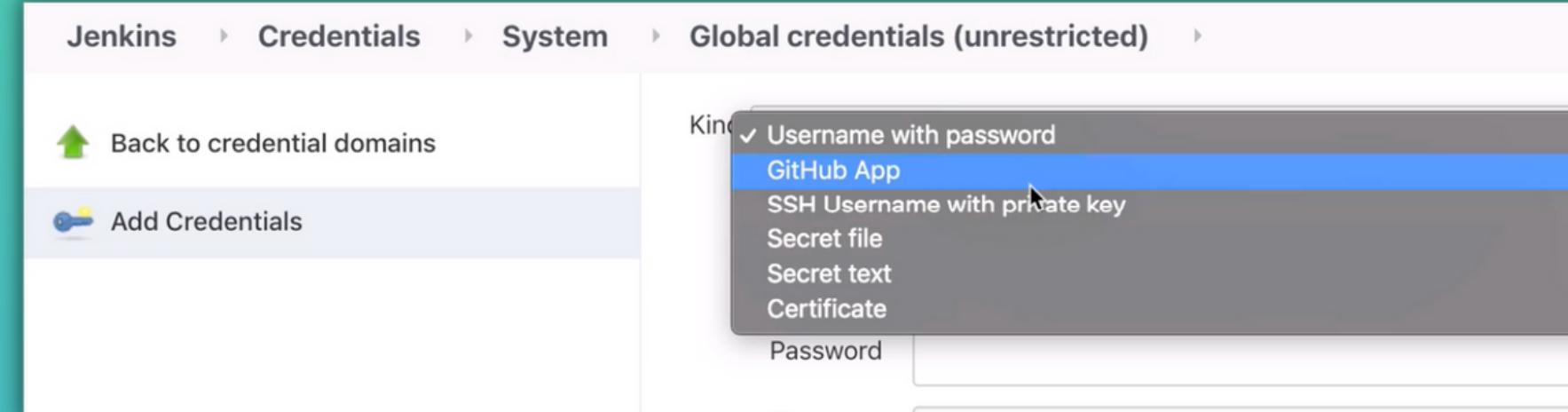
## Credential Types

Jenkins can store the following types of credentials

- **Secret text** - a token such as an API token
- **Username and password** - which could be handled as separate components or as a colon separated string in the format username:password
- **Secret file** - secret content in a file
- **SSH Username with private key** - an SSH public/private key pair
- **Certificate** - a PKCS#12 certificate file and optional password
- **Docker Host Certificate Authentication** credentials.



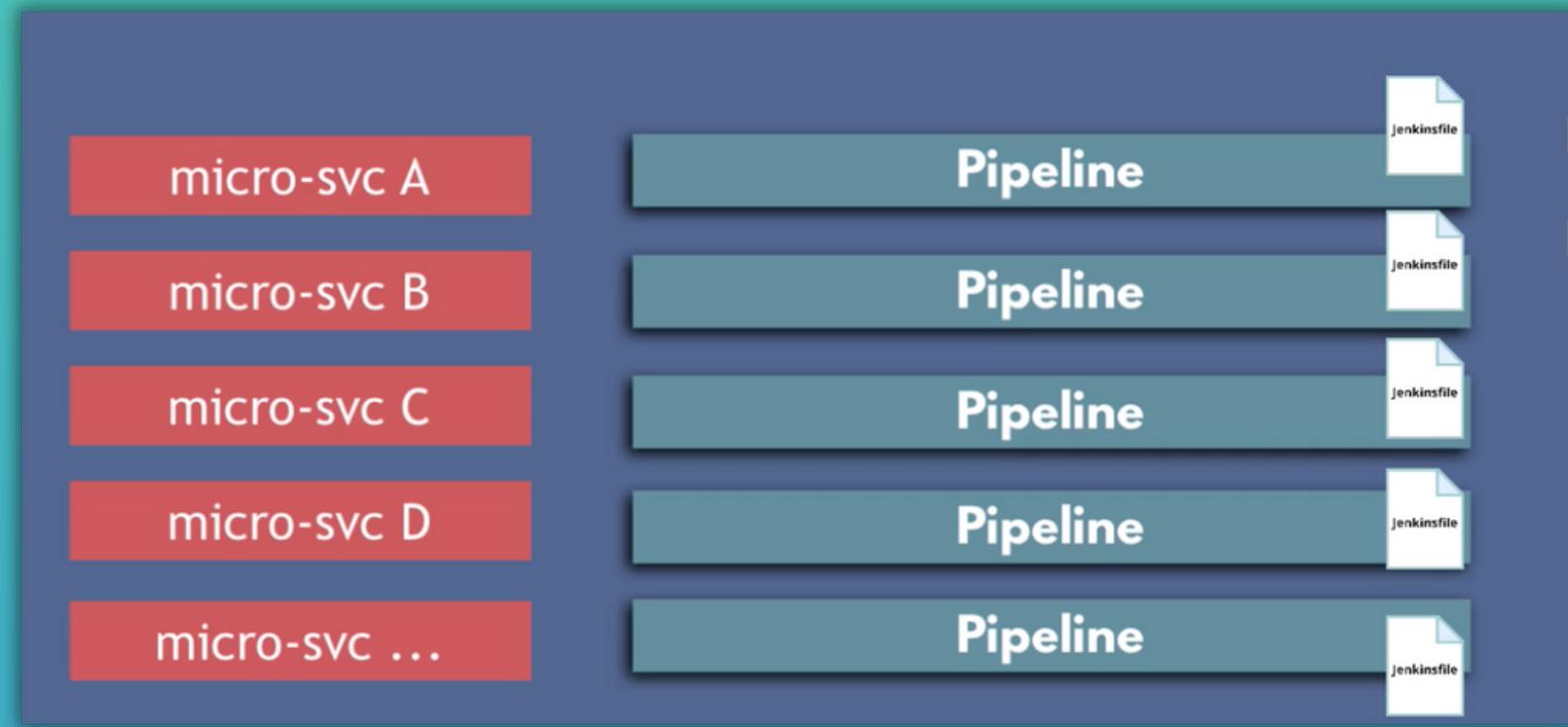
New types based on plugins



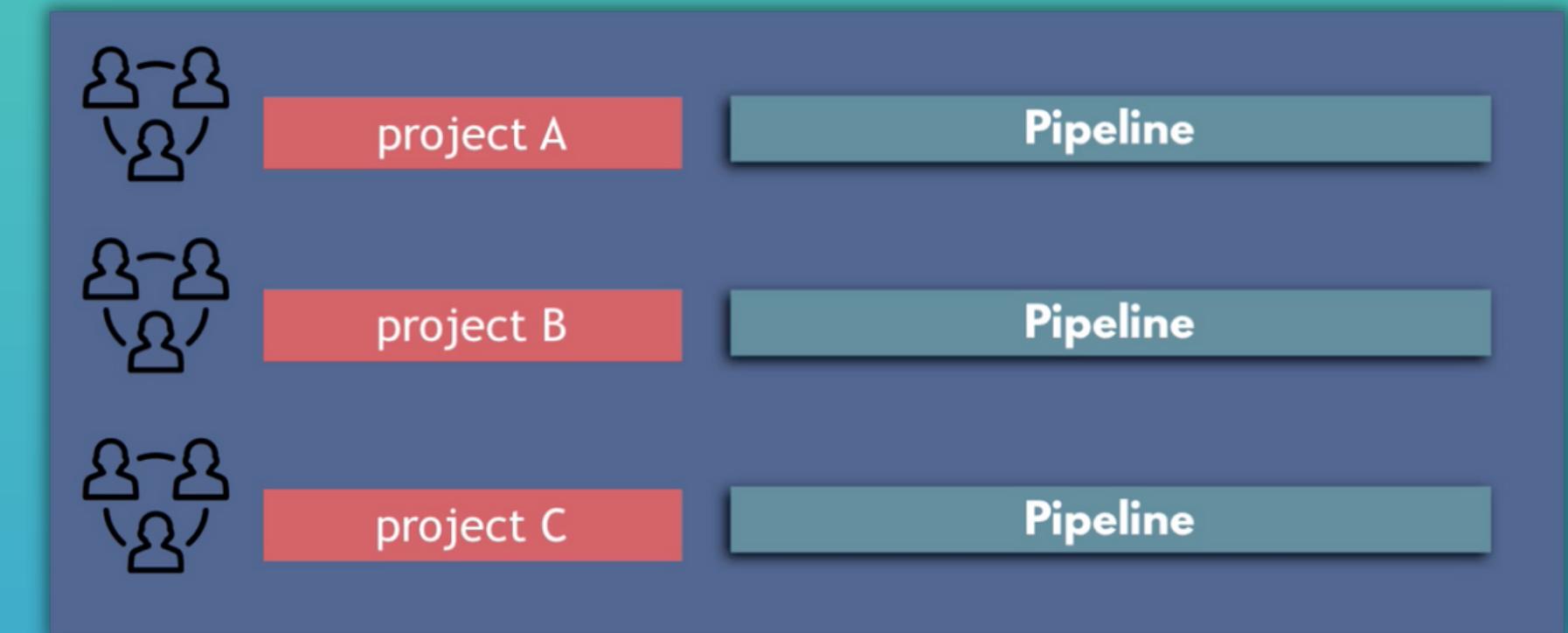
Domain	ID	Name
(global)	gitlab-credentials	nanuchi/*********
(global)	docker-hub-repo	nanajanashia/*
(global)	nexus-docker-repo	nana/*********
(global)	server-credentials	server-user/***
(global)	global	global/*********

# Jenkins Shared Library - 1

- Used to share parts of Pipelines between various projects to reduce duplication
- "Shared Libraries" can be defined in external git repositories and loaded into existing Pipelines



**Use Case 1:** Same logic in different microservices



**Use Case 2:** Many different projects in a company with some common tasks and logic

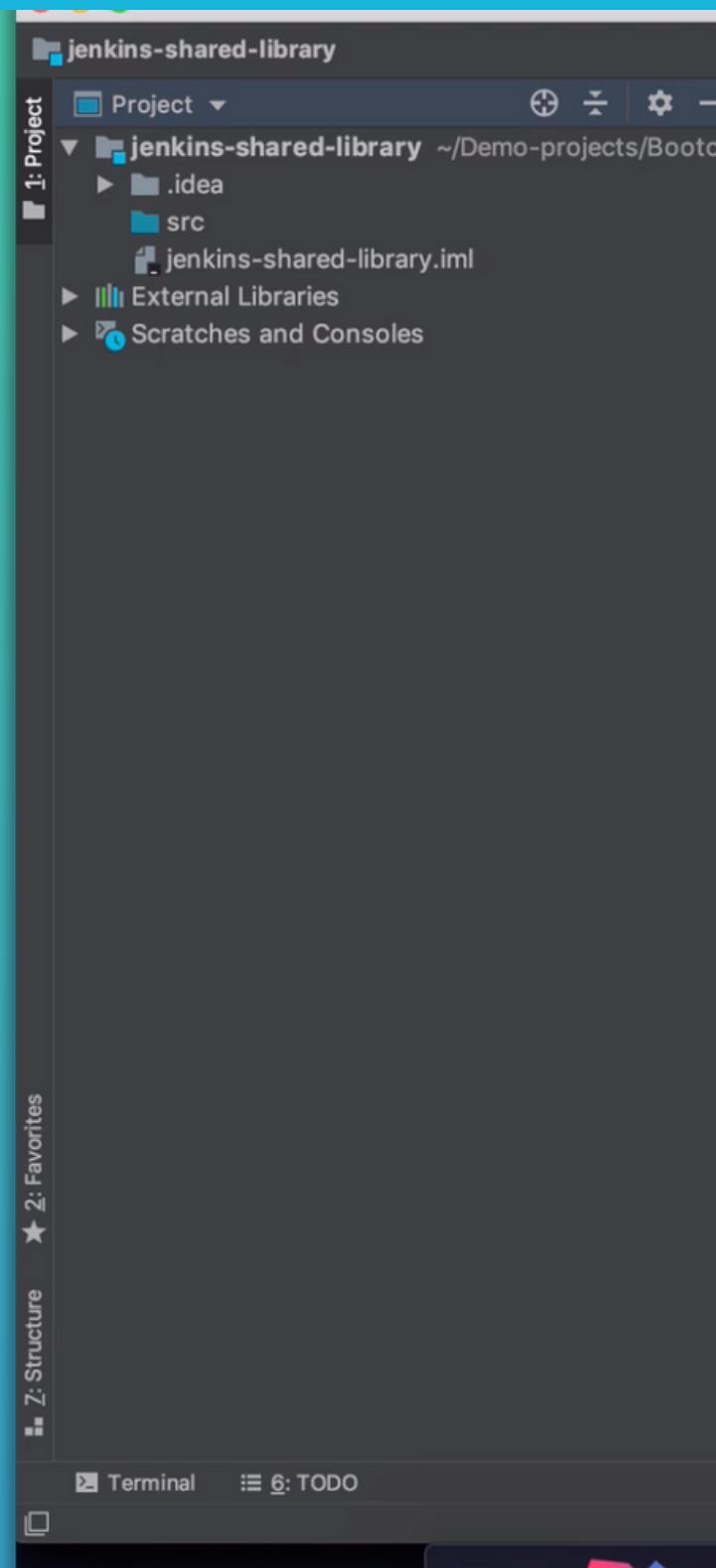
# Jenkins Shared Library - 2

## Steps to use Shared Library

- 1. Create the Shared Library (SL)**
  - a. Create Repository
  - b. Write the Groovy Code
- 2. Make the SL available in Jenkins globally or for project**
- 3. Use the SL in Jenkinsfile to extend the Pipeline**

# Jenkins Shared Library - 3

## 1) Create the Shared Library (SL)



### Structure of Shared Library

#### vars folder

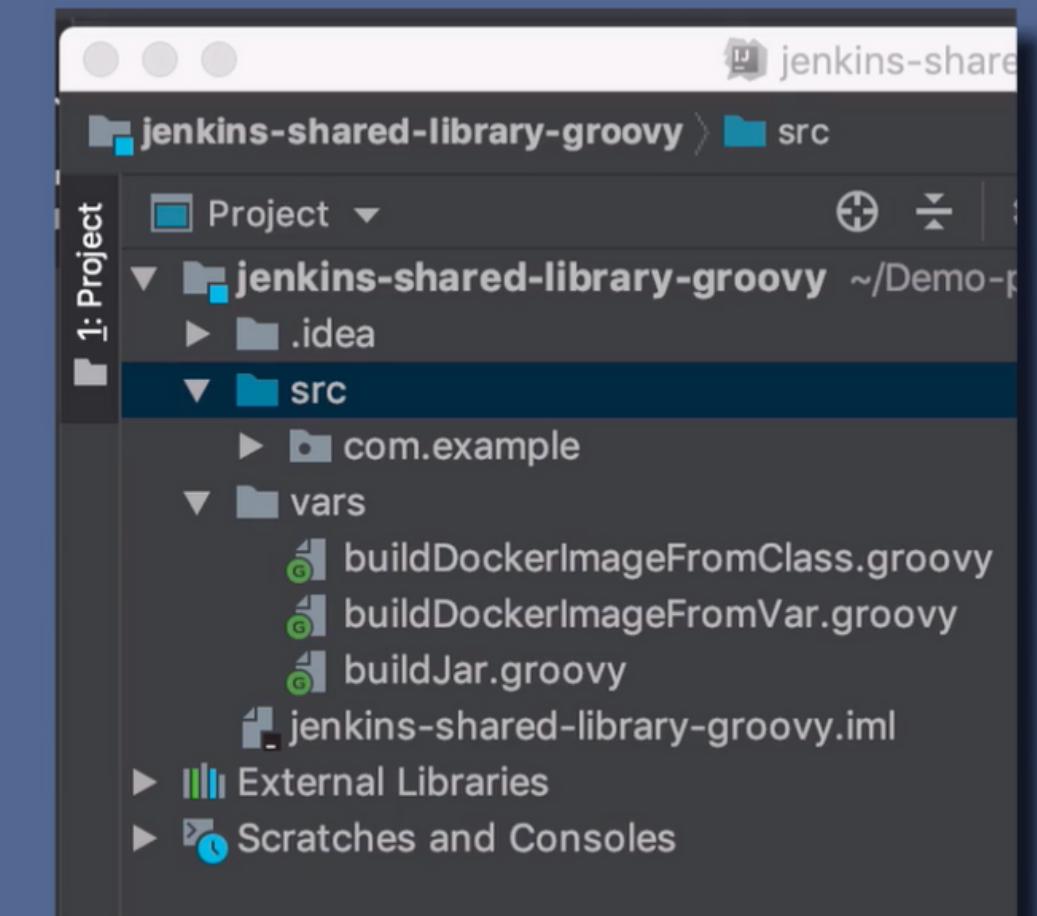
- ▶ functions that we call from Jenkinsfile
- ▶ Each function/execution step is its own Groovy file

#### src folder

- ▶ helper code

#### resources folder

- ▶ use external libraries
- ▶ non groovy files



# Jenkins Shared Library - 4

## 2) Configure globally in Jenkins

Jenkins > configuration

Add

### Global Pipeline Libraries

Sharable libraries available to any Pipeline jobs running on this system. These libraries will be trusted, meaning they run without "sandbox" restrictions and may use @Grab.

Library	Name	Default version	?
	jenkins-shared-library	master	
		Cannot validate default version until after saving and reconfiguring.	
Load implicitly	<input type="checkbox"/>		
Allow default version to be overridden	<input checked="" type="checkbox"/>		
Include @Library changes in job recent changes	<input checked="" type="checkbox"/>		

### Retrieval method

Modern SCM

Git

Project Repository: <https://gitlab.com/nanuchi/jenkin>

Credentials: nanuchi/\*\*\*\*\*\*\*\*

Behaviours

Discover branches

Save Apply

### Retrieval method

Modern SCM

### Source Code Management

Git

Project Repository: <https://gitlab.com/nanuchi/jenkin>

Credentials: nanuchi/\*\*\*\*\*\*\*\*

Add

Behaviours

Discover branches

GitHub

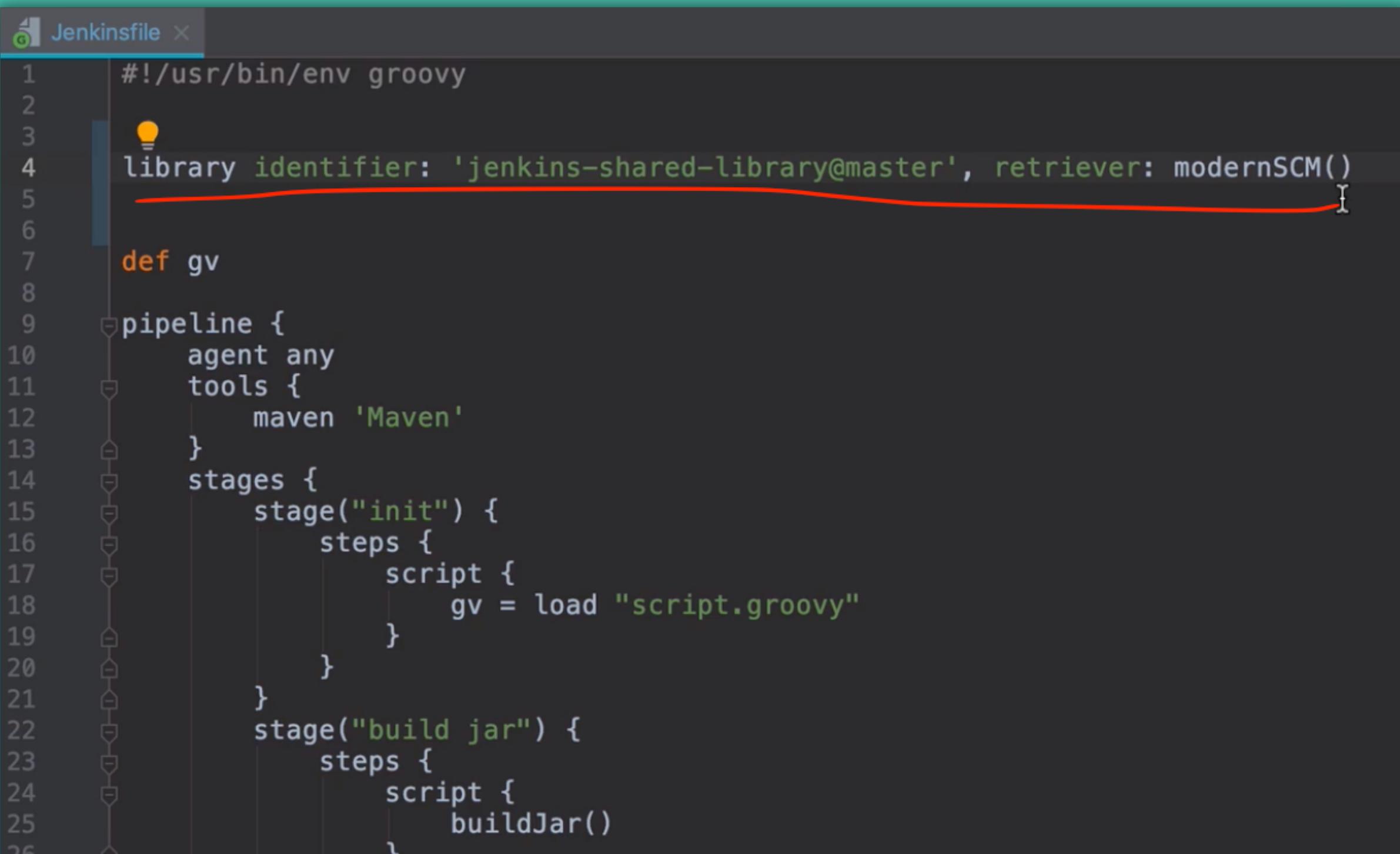
Legacy SCM

Add

Delete

# Jenkins Shared Library - 5

## 3) Use the configured JSL in any of your pipeline scripts



```
1 #!/usr/bin/env groovy
2
3     library identifier: 'jenkins-shared-library@master', retriever: modernSCM()
4
5 def gv
6
7 pipeline {
8     agent any
9     tools {
10         maven 'Maven'
11     }
12     stages {
13         stage("init") {
14             steps {
15                 script {
16                     gv = load "script.groovy"
17                 }
18             }
19         }
20         stage("build jar") {
21             steps {
22                 script {
23                     buildJar()
24                 }
25             }
26         }
27     }
28 }
```

# Trigger Jenkins Job

3 ways to trigger Jenkins jobs:

## Manually



- Use case may be **for production pipelines**

## Automatically



- Trigger **automatically when changes happen in the git repository (Webhook)**
- Jenkins and Gitlab needs to be configured for that

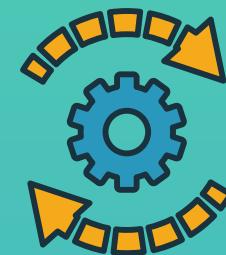
## Scheduling



- Trigger job **on scheduled times**
- For example:
  - For long running tests
  - Tasks that need to run only weekly/monthly

# Webhook

- Mechanism to automatically trigger the build of a Jenkins project upon a commit pushed in a Git repository



Configure in Git repository:

Nana Janashia > java-maven-app > Webhook Settings

**Webhooks**

Webhooks enable you to send notifications to web applications in response to events in a group or project. We recommend using an integration in preference to a webhook.

**URL**  
http://example.com/trigger-ci.json

URL must be percent-encoded if necessary.

**Secret token**

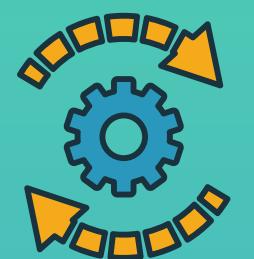
Use this token to validate received payloads. It is sent with the request in the X-Gitlab-Token HTTP header.

**Trigger**

**Push events**  
Branch name or wildcard pattern to trigger on (leave blank for all)  
URL is triggered by a push to the repository

**Tag push events**  
URL is triggered when a new tag is pushed to the repository

**Comments**  
URL is triggered when a comment is added to a pull request



Configure in Jenkins Job:

Dashboard > java-maven-app >

General Branch Sources Build Configuration Scan Multibranch

Properties Pipeline Libraries

by Jenkinsfile

Script Path  
Jenkinsfile

**Scan Multibranch Pipeline Triggers**

Periodically if not otherwise run

Scan by webhook

Trigger token  
gitlabtoken

The token to match with webhook token. Receive any HTTP request. JEN

# Software Versioning

# Software Versioning

- Each **build tool** or package manager tool **keeps the project's version**

 Gradle

```
build.gradle x
46
47     defaultTasks 'bootRun'
48
49     group = 'com.myapp'
50     version = '1.0.0-SNAPSHOT'
51
52     description = ''
```

 npm

```
package.json x
1 {
2     "name": "test-js-app",
3     "version": "0.0.1",
4     "description": "Beta",
5     "private": true,
6     "license": "UNLICENSED",
7     "cacheDirectories": [
8         "node_modules"
9     ],
10    "dependencies": {
11        "@babel/polyfill": "7.8
```

 Maven™

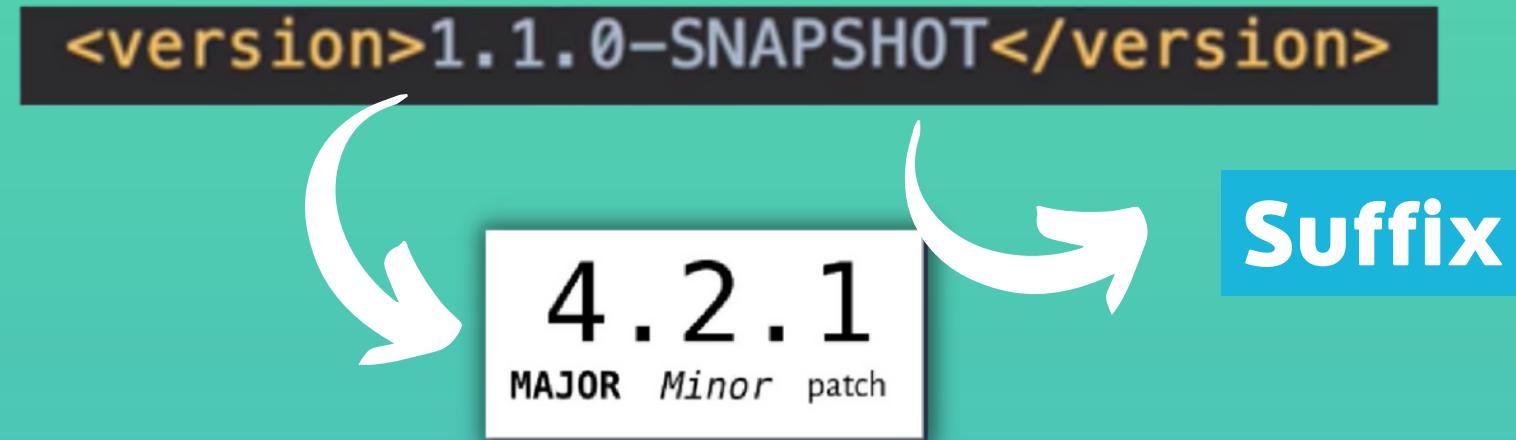
```
pom.xml x
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5             http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7
8   <groupId>maven</groupId>
9   <artifactId>test-maven</artifactId>
10  <version>1.0-SNAPSHOT</version>
11
12 </project>
```

- You and the team decide how you version your application



1.1.0-SNAPSHOT

# 3 Parts of a version



## Major Version

- When you increase this part, means the **new software** **version contains big changes**, breaking changes, NOT backward-compatible

## Minor Version

- New, but backward-compatible changes, new API features

## Patch Version

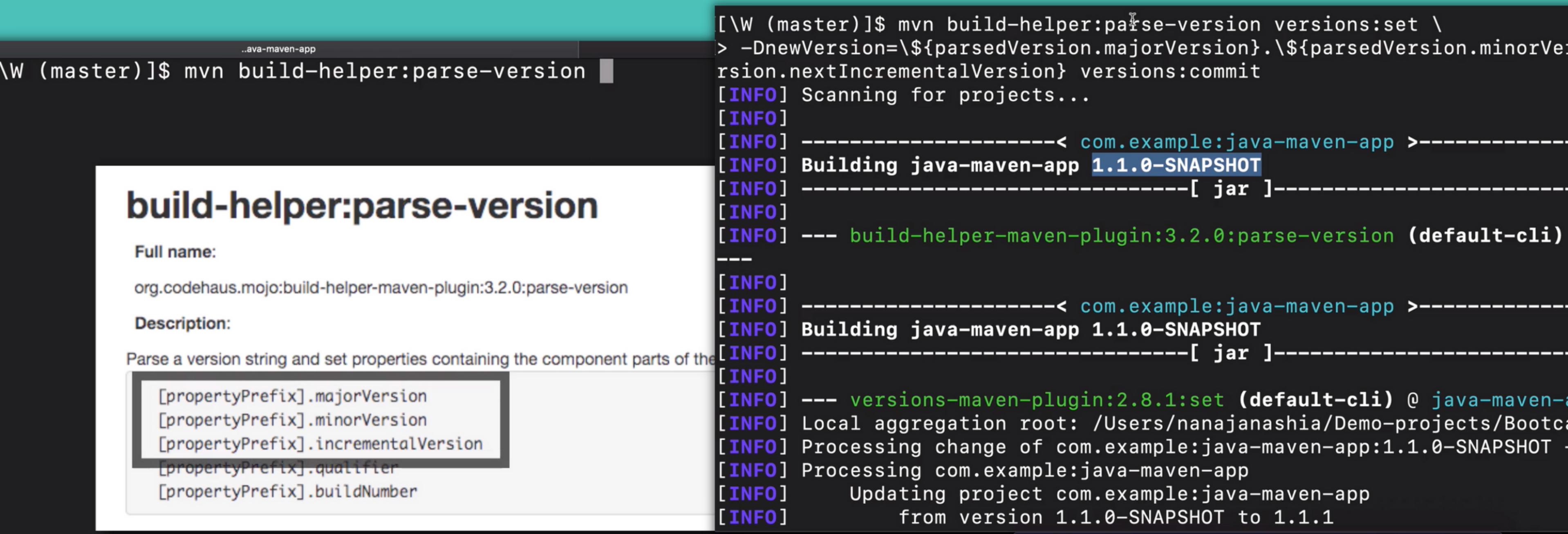
- Minor changes and **bug fixes**, doesn't change API

## Suffix

- Gives **more information**
- For example, "SNAPSHOT" term means it's a "pre-release" version indicating that the version is **unstable** and still under development. When code is ready and it's time to release it, you will remove the "SNAPSHOT" suffix
- Another example "alpha"

# Dynamically increment application version - 1

- Instead of updating the version manually, you will configure the build automation tool to **increment the version automatically**
- Build Tools have commands to increment the version



The screenshot shows a terminal window with two panes. The left pane displays the Maven command:

```
\W (master)]$ mvn build-helper:parse-version
```

The right pane shows the command's output:

```
[\\W (master)]$ mvn build-helper:parse-version versions:set \
> -DnewVersion=\${parsedVersion.majorVersion}.\${parsedVersion.minorVer
rsion.nextIncrementalVersion} versions:commit
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example:java-maven-app >-----
[INFO] Building java-maven-app 1.1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO] --- build-helper-maven-plugin:3.2.0:parse-version (default-cli)
[INFO]
[INFO] -----< com.example:java-maven-app >-----
[INFO] Building java-maven-app 1.1.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- versions-maven-plugin:2.8.1:set (default-cli) @ java-maven-a
[INFO] Local aggregation root: /Users/nanajanashia/Demo-projects/Bootca
[INFO] Processing change of com.example:java-maven-app:1.1.0-SNAPSHOT -
[INFO] Processing com.example:java-maven-app
[INFO] Updating project com.example:java-maven-app
[INFO] from version 1.1.0-SNAPSHOT to 1.1.1
```

Below the terminal window, a tooltip for the **build-helper:parse-version** command is visible, providing details about its full name, description, and properties.

**build-helper:parse-version**

**Full name:** org.codehaus.mojo:build-helper-maven-plugin:3.2.0:parse-version

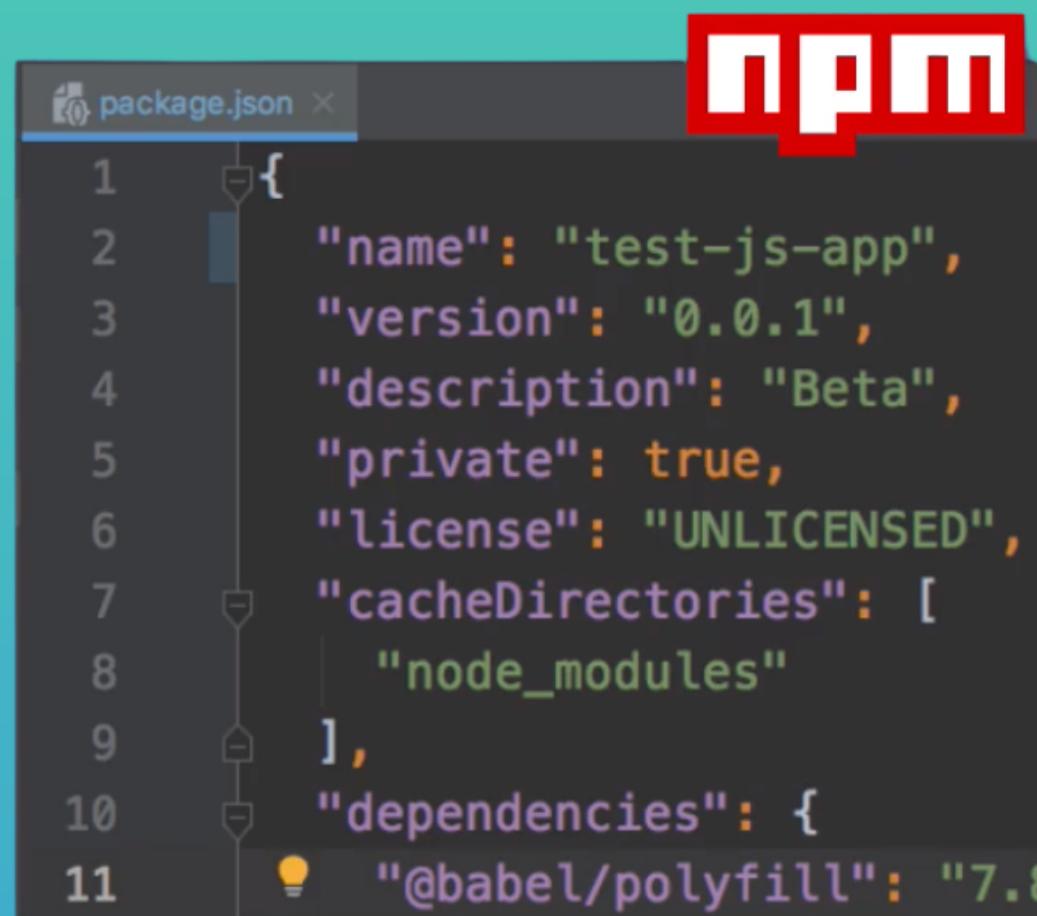
**Description:** Parse a version string and set properties containing the component parts of the version string.

**Properties:**

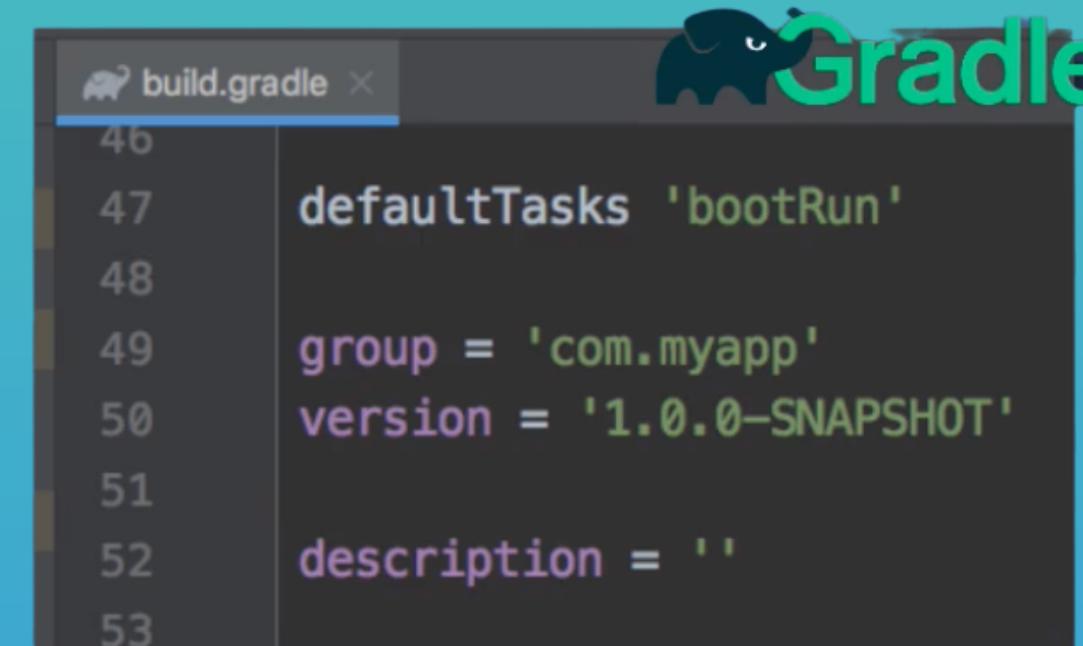
- [propertyPrefix].majorVersion
- [propertyPrefix].minorVersion
- [propertyPrefix].incrementalVersion
- [propertyPrefix].qualifier
- [propertyPrefix].buildNumber

# Dynamically increment application version - 2

- Different package manager/build tools have the **same concept**
- For npm, you can read version from package.json file, increment and save back



```
1 {  
2   "name": "test-js-app",  
3   "version": "0.0.1",  
4   "description": "Beta",  
5   "private": true,  
6   "license": "UNLICENSED",  
7   "cacheDirectories": [  
8     "node_modules"  
9   ],  
10  "dependencies": {  
11    "@babel/polyfill": "7.8
```



```
46  
47 defaultTasks 'bootRun'  
48  
49 group = 'com.myapp'  
50 version = '1.0.0-SNAPSHOT'  
51  
52 description = ''  
53
```

# Best Practices - 1

## Pipeline Script/Jenkinsfile related best practices:

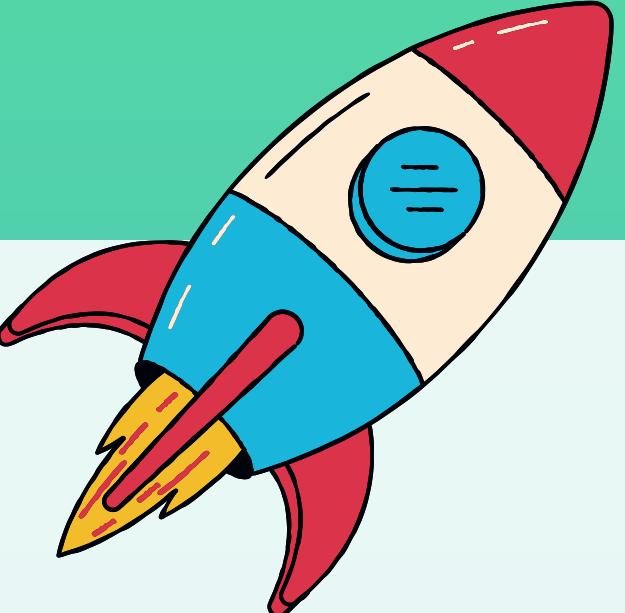
- Use Pipeline as Code: Store your Jenkinsfile in Git Repository (instead of inline script in Jenkins). Good for history, working in a team etc.
- Call your Pipeline Script the default name: Jenkinsfile
- Non-Setup work should occur within a stage block. Makes your builds easier to visualize, debug etc.
- Make sure to put `#!/usr/bin/env groovy` at the top of the file so that IDEs, GitHub diffs, etc properly detect the language and do syntax highlighting for you.
- Input Parameter: input shouldn't be done within a node block. It's recommended to use timeout for the input step in order to avoid waiting for an infinite amount of time, and also control structures (try/catch/finally).



# Best Practices - 2

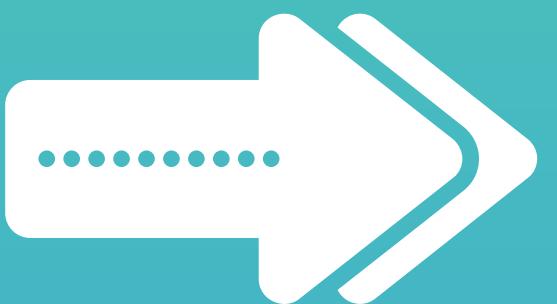
## Other best practices:

- Use Automatic versioning
- Store common pipeline code in a Shared Library, so that it can be used by other projects/teams
- More best practices and how to put it to practice: <https://www.lambdatest.com/blog/jenkins-best-practices/>





**CI Part of CI/CD**



**CD Part of CI/CD in next Module**