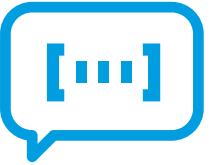


06

# CI/CD - Jenkins



# Index

**01 What is Jenkins**

**02 Git plugins**

**03 Freestyle project**

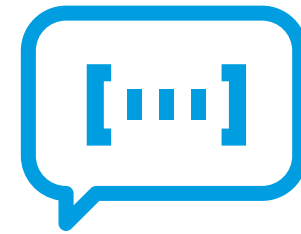
**04 Jenkins Pipelines**

**05 Continuous Integration With Jenkins  
And GitHub**

**06 Jenkins Metrics**

**07 Python and Jenkins**

# 01



## What is Jenkins

# Jenkins



- Is an **open-source Continuous Integration server** written in Java for orchestrating a **chain of actions** to achieve the Continuous Integration process in an automated fashion.
- Jenkins **supports the complete development life cycle** of software from building, testing, documenting the software, deploying, and other stages of the software development life cycle.
- Widely used application around the world.
- **Companies can accelerate their software development process**, as Jenkins can automate build and test at a rapid rate.

## OBJECTIVE

### Accessing Jenkins

## INSTRUCTIONS

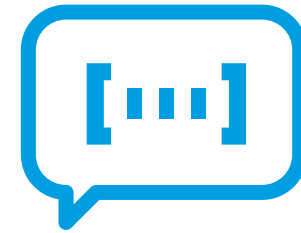
1. Open your browser and enter next url:
  1. `https //localhost 8080`
2. Unlock Jenkins following the instructions (entering the **initialAdminPassword**).
3. Install suggested plugins.
4. Finish the initial config process.
5. Review the interface.



15 min

# 02

## Git plugins



# Jenkins plugins



- Jenkins has **outstanding plugin support**. There are thousands of third-party application plugins available on their website:

<https://plugins.jenkins.io/>

- Jenkins **needs to have GitHub plugin** installed to be able to pull code from the GitHub repository.
  - You need not install a GitHub plugin if you have already installed the Git plugin in response to the prompt during the Jenkins' installation setup.

## OBJECTIVE

### **Installing Git Plugin in Jenkins**

## INSTRUCTIONS

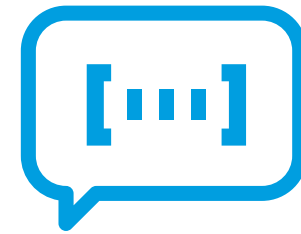
1. Follow this Guru99 tutorial for installing git plugin in Jenkins:
  - [https://www.guru99.com/jenkins-github-integration.html#how to install git plugin in jenkins](https://www.guru99.com/jenkins-github-integration.html#how%20to%20install%20git%20plugin%20in%20jenkins)



**15 min**



03




# Freestyle project


# Freestyle Project

Enter an item name


» This field cannot be empty, please enter a valid name

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**

Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**External Job**

This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.

**Multi-configuration project**

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

or Folder

- It is a **repeatable build job**, script, or pipeline that contains steps and post-build actions.
- It is an improved job or task that **can span multiple operations**.
- It allows **configuring build triggers** and offers project-based security for your Jenkins project.
- It also **offers plugins** to help building steps and post-build actions.

# Freestyle Project sections

The screenshot shows the 'General' tab of the Jenkins Freestyle Project Configuration page. At the top, there are tabs for 'General', 'Source Code Management', 'Build Triggers', 'Build Environment', 'Build', and 'Post-build Actions'. The 'General' tab is active. Below the tabs, there is a 'Description' field containing the text 'This is a demo freestyle project.' and a '[Plain text] Preview' button. Below the description, there is a list of checkboxes for various project settings: 'Commit agent's Docker container', 'Define a Docker template', 'Discard old builds', 'GitHub project', 'This build requires lockable resources', 'This project is parameterized', 'Throttle builds', 'Disable this project', and 'Execute concurrent builds if necessary'. Each checkbox has a blue question mark icon to its right. At the bottom right, there is an 'Advanced...' button.

Lets visit the freestyle project sections and its utility:

- 1. General**
- 2. Source Code Management**
- 3. Build Triggers**
- 4. Build Environment**
- 5. Build**
- 6. Post-build Actions**

## OBJECTIVE

### **Creating a Jenkins freestyle project**

## INSTRUCTIONS

1. Follow this dwops.com tutorial for creating a freestyle Project in Jenkins that generate a build for a Java source code:
  - <https://dwops.com/courses/beginners-jenkins-tutorial-basics-freestyle-projects-ci-cd-pipeline/lesson/creating-a-freestyle-project-using-a-github-repository/>



**20 min**

## OBJECTIVE

### Configuring Email notifications

## INSTRUCTIONS

1. Follow next steps for configuring email notifications when building a Project in Jenkins using a Gmail account.
2. Remember enabling **Less Secure App Access**. Go to this page [here](#).



20 min

Go to the Manage Jenkins > Configure System then find Extended E-mail Notification and fill in the following details:

1. ✓ SMTP server – smtp.gmail.com
2. ✓ SMTP Port – 465
  - Click on advanced below the SMTP Port box.
3. ✓ SMTP Username – any Gmail id. This Gmail account will be used to send email notifications.
4. ✓ SMTP Password – password of the Gmail id used above
5. ✓ Check the Use SSL box.
6. ✓ Default Recipient – Add a default email id.



## OBJECTIVE

### **Creating a parametrized job**

## INSTRUCTIONS

1. We are going to create a Jenkins job that will take input from the drop-down using parameters.



**20 min**

In the freestyle Job > Section General:

1. Create Parameters for dropdown

1. ✓ Check This project is parameterized in the General section
2. ✓ Click Add Parameter.
3. ✓ Choose Choice Parameter

2. Configure Choice Parameter

1. ✓ Name – give a suitable name for your parameter. For eg. OS
2. ✓ Choices – give the values one per line that will appear in the drop-down option.
3. ✓ Description – describe the parameter.

3. Configure Build Step

1. ✓ Choose Execute Shell and print something using the parameter. For eg.

```
echo 'The Jenkins is installed on ' $OS.
```

4. Build the project and choose the parameter

5. Check the Console Output





## OBJECTIVE

**Concat various freestyle projects**

## INSTRUCTIONS

1. Follow next instructions to generate a multiple-step freestyle Project.

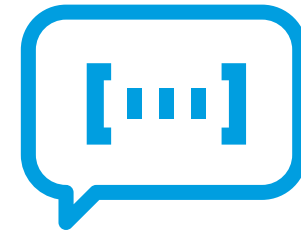


**20 min**

1. Generate a freestyle project (Step1) that clone next repository:
  - <https://github.com/ricardoahumada/SimuladorCoches>
  - Chage to standalone folder in the cloned repository.
2. Generate another freestyle project (Step2) that execute next order in shell:
  - `mvn clean test`
  - Go back to Step1 project and add a "Post-build Actions" that points to Step2.
3. Generate a final freestyle project (Step3) that execute next order in shell:
  - `mvn package`
  - Go back to Step3 project and add a "Post-build Actions" that points to Step3.
4. Build the project
5. Check the Console Output



# 05



## Jenkins Pipelines

# Jenkins Pipelines

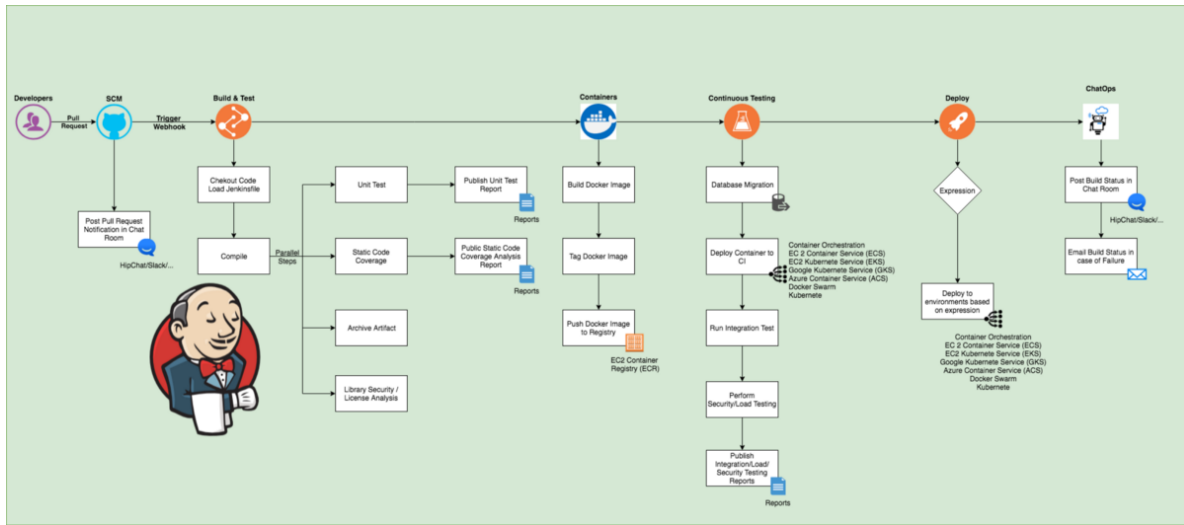


Image Source: medium.com

- Jenkins Pipeline (or simply "Pipeline" with a capital "P") is a **suite of plugins** which supports implementing and integrating continuous delivery pipelines into Jenkins.
- Pipeline **provides an extensible set of** tools for modeling simple-to-complex delivery pipelines "as code" via the Pipeline domain-specific language (DSL) syntax.
- The definition of a Jenkins Pipeline is written into a text file (called a **Jenkinsfile**) which in turn can be committed to a project's source control repository. [
- This is the foundation of "**Pipeline-as-code**"; treating the CD pipeline a part of the application to be versioned and reviewed like any other code.

# Pipeline concepts

- **Pipeline:** block which contains all the processes such as build, test, deploy, etc. It is a collection of all the stages in a Jenkinsfile.
- **Node:** A node is a machine that executes an entire workflow. It is a key part of the scripted pipeline syntax.
- **Agent:** It instructs Jenkins to allocate an executor for the builds. Helps to distribute the workload to different agents and execute several projects within a single Jenkins instance. Can be Any, None, Label, Docker.
- **Stages:** The work is specified in the form of stages. Each stage performs a specific task.
- **Steps:** A series of steps can be defined within a stage block. These steps are carried out in sequence to execute a stage.

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing..'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

# Jenkinsfile

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        echo 'Building..'
      }
    }
    stage('Test') {
      steps {
        echo 'Testing..'
      }
    }
    stage('Deploy') {
      steps {
        echo 'Deploying....'
      }
    }
  }
}
```

- A Jenkinsfile can be written using two types of syntax - **Declarative** and **Scripted**.
- Declarative and Scripted Pipelines are constructed fundamentally differently.
- **Declarative Pipeline** is a more recent feature of Jenkins Pipeline which:
  - provides **richer syntactical features** over Scripted Pipeline syntax, and
  - is designed to **make writing and reading** Pipeline code **easier**.
- Many of the individual syntactical components (or "steps") written into a Jenkinsfile, however, are common to both Declarative and Scripted Pipeline.

## OBJECTIVE

### **Scripted and Declarative Pipeline projects**

## INSTRUCTIONS

1. Follow next steps for creating sample scripted and declarative pipeline projects.



**15 min**

1. Create a new Item of type Pipeline.
  - Choose '**pipeline script**' for scripted pipeline.
  - Then copy and paste the previous code.
  - Build the project
  - Check the Console Output
  
2. Create another Item of type Pipeline.
  - Then select '**pipeline script from SCM**', for declarative pipeline.
  - Choose your SCM: <https://github.com/Zulaikha12/gitnew.git>
  - Build the project
  - Check the Console Output





# Declarative Pipeline

- In Declarative Pipeline syntax, the pipeline block defines all the work done throughout your entire Pipeline.

```
pipeline {
  agent any
  stages {
    stage('One') {
      steps {
        echo 'Hi, this is Zulaikha from edureka'
      }
    }
    stage('Two') {
      steps {
        input('Do you want to proceed?')
      }
    }
    stage('Three') {
      when {
        not {
          branch "master"
        }
      }
      steps {
        echo "Hello"
      }
    }
    stage('Four') {
      parallel {
        stage('Unit Test') {
          steps {
            echo "Running the unit test..."
          }
        }
        stage('Integration test') {
          agent {
            docker {
              reuseNode true
              image 'ubuntu'
            }
          }
          steps {
            echo "Running the integration test..."
          }
        }
      }
    }
  }
}
```

## OBJECTIVE

### **Trying a Pipelina project**

## INSTRUCTIONS

1. Create a new Item of type Pipeline.
2. Copy and paste the previous code.
3. Build the project
4. Check the Console Output



**5 min**

# Scripted Pipeline

- In Scripted Pipeline syntax, one or more node blocks do the core work throughout the entire Pipeline.
- Although this is not a mandatory requirement of Scripted Pipeline syntax, confining your Pipeline's work inside of a node block does two things:
  - Schedules the steps contained within the block to run by adding an item to the Jenkins queue. As soon as an executor is free on a node, the steps will run.
  - Creates a workspace (a directory specific to that particular Pipeline) where work can be done on files checked out from source control.

```
node {
    for (i=0; i<2; i++) {
        stage "Stage #"+i
        print 'Hello, world !'
        if (i==0)
        {
            git "https://github.com/Zulaikha12/gitnew.git"
            echo 'Running on Stage #0'
        }
        else {
            build 'Declarative pipeline'
            echo 'Running on Stage #1'
        }
    }
}
```

## OBJECTIVE

### **Trying a Pipelina project**

## INSTRUCTIONS

1. Create a new Item of type Pipeline.
2. Copy and paste the previous code.
3. Build the project
4. Check the Console Output



**5 min**

## OBJECTIVE

### **Comparing Declarative and Scripted Pipelines**

## INSTRUCTIONS

1. Look at next codes for the two types of pipelines. They are equivalents.
2. Which one is declarative and which one is scripted?
3. What this code do?



**5 min**

```

pipeline {
  agent any
  options {
    skipStagesAfterUnstable()
  }
  stages {
    stage('Build') {
      steps {
        sh 'make'
      }
    }
    stage('Test'){
      steps {
        sh 'make check'
        junit 'reports/**/*.xml'
      }
    }
    stage('Deploy') {
      steps {
        sh 'make publish'
      }
    }
  }
}

```

```

node {
  stage('Build') {
    sh 'make'
  }
  stage('Test') {
    sh 'make check'
    junit 'reports/**/*.xml'
  }
  if (currentBuild.currentResult == 'SUCCESS') {
    stage('Deploy') {
      sh 'make publish'
    }
  }
}

```



# Using environment variables

- Jenkins Pipeline exposes environment variables via the global variable **env**, which is available from anywhere within a Jenkinsfile.

```
pipeline {
  agent any
  stages {
    stage('Example') {
      steps {
        echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
      }
    }
  }
}
```

# Using environment variables

The full list of environment variables accessible from within Jenkins Pipeline is documented at [\*\*`\${YOUR\_JENKINS\_URL}/pipeline-syntax/globals#env`\*\*](https://YOUR_JENKINS_URL/pipeline-syntax/globals#env).

## **BUILD\_ID**

The current build ID, identical to BUILD\_NUMBER for builds created in Jenkins versions 1.597+

## **BUILD\_NUMBER**

The current build number, such as "153"

## **BUILD\_TAG**

String of jenkins-`${JOB_NAME}`-`${BUILD_NUMBER}`. Convenient to put into a resource file, a jar file, etc for easier identification

## **BUILD\_URL**

The URL where the results of this build can be found (for example `http://buildserver/jenkins/job/MyJobName/17/` )

## **EXECUTOR\_NUMBER**

The unique number that identifies the current executor (among executors of the same machine) performing this build. This is the number you see in the "build executor status", except that the number starts from 0, not 1

## **JAVA\_HOME**

If your job is configured to use a specific JDK, this variable is set to the JAVA\_HOME of the specified JDK. When this variable is set, PATH is also updated to include the bin subdirectory of JAVA\_HOME

## **JENKINS\_URL**

Full URL of Jenkins, such as `https://example.com:port/jenkins/` (NOTE: only available if Jenkins URL set in "System Configuration")

## **JOB\_NAME**

Name of the project of this build, such as "foo" or "foo/bar".

## **NODE\_NAME**

The name of the node the current build is running on. Set to 'master' for the Jenkins controller.

## **WORKSPACE**

The absolute path of the workspace



# Setting environment variables

- Setting an environment variable within a Jenkins Pipeline is accomplished differently depending on whether Declarative or Scripted Pipeline is used.
- Declarative Pipeline supports an **environment directive**, whereas users of Scripted Pipeline must use the **withEnv** step.

```
pipeline {
  agent any
  environment {
    CC = 'clang'
  }
  stages {
    stage('Example') {
      environment {
        DEBUG_FLAGS = '-g'
      }
      steps {
        sh 'printenv'
      }
    }
  }
}
```

```
node {
  /* .. snip .. */
  withEnv(["PATH+MAVEN=${tool 'M3'}/bin"]) {
    sh 'mvn -B verify'
  }
}
```

# Setting environment variables dynamically

- Environment variables can be set at run time and can be used by shell scripts (sh), Windows batch scripts (bat) and PowerShell scripts (powershell).
- Each script can either **returnStatus** or **returnStdout**.
- More information on scripts at
  - <https://www.jenkins.io/doc/pipeline/steps/workflow-durable-task-step/>
- Here is an example in a declarative pipeline using sh (shell) with both returnStatus and returnStdout.

```
pipeline {
  agent any
  environment {
    // Using returnStdout
    CC = """${sh(
      returnStdout: true,
      script: 'echo "clang"'
    )}"""
    // Using returnStatus
    EXIT_STATUS = """${sh(
      returnStatus: true,
      script: 'exit 1'
    )}"""
  }
  stages {
    stage('Example') {
      environment {
        DEBUG_FLAGS = '-g'
      }
      steps {
        sh 'printenv'
      }
    }
  }
}
```

# Handling credentials

- Credentials configured in Jenkins can be handled in Pipelines for immediate use.
- **For secret text, usernames and passwords, and secret files**
  - Jenkins' declarative Pipeline syntax has the `credentials()` helper method (used within the environment directive) which supports secret text, username and password, as well as secret file credentials. If you want to handle other types of credentials, refer to the For other credential types section (below).
- **Secret text**
  - The following Pipeline code shows an example of how to create a Pipeline using environment variables for secret text credentials.

```
pipeline {
    agent {
        // Define agent details here
    }
    environment {
        AWS_ACCESS_KEY_ID = credentials('jenkins-aws-secret-key-id')
        AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-key')
    }
    stages {
        stage('Example stage 1') {
            steps {
                echo $AWS_SECRET_ACCESS_KEY
            }
        }
        stage('Example stage 2') {
            steps {
                //
            }
        }
    }
}
```

# Username and passwords

This code actually sets the following three environment variables:

- **BITBUCKET\_COMMON\_CREDS:** contains a username and a password separated by a colon in the format username:password.
- **BITBUCKET\_COMMON\_CREDS\_USR**  
**R:** an additional variable containing the username component only.
- **BITBUCKET\_COMMON\_CREDS\_PSW**  
**W:** an additional variable containing the password component only.

```
environment {  
    BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-creds')  
}  
...  
  
echo $BITBUCKET_COMMON_CREDS  
  
echo $BITBUCKET_COMMON_CREDS_USR  
  
echo $BITBUCKET_COMMON_CREDS_PSW
```

# Secret files

A secret file is a credential which is stored in a file and uploaded to Jenkins. Secret files are used for credentials that are:

- too unwieldy to enter directly into Jenkins, and/or
- in binary format, such as a GPG file.

```
pipeline {
  agent {
    // Define agent details here
  }
  environment {
    // The MY_KUBECONFIG environment variable will be assigned
    // the value of a temporary file. For example:
    // /home/user/.jenkins/workspace/cred_test@tmp/secretFiles/546a5cf3-9b56-4165-a0fd-19e2afe6b31f/kubeconfig.txt
    MY_KUBECONFIG = credentials('my-kubeconfig')
  }
  stages {
    stage('Example stage 1') {
      steps {
        sh("kubectl --kubeconfig $MY_KUBECONFIG get pods")
      }
    }
  }
}
```

## OBJECTIVE

### **SSH in Jenkins**

## INSTRUCTIONS

1. Create a username/password for connecting your partner ssh server.
2. Generate a Jenkinsfile for uploading and run a Python script to the server.
3. Review the reference at:
  - <https://www.jenkins.io/doc/pipeline/steps/ssh-steps/>



**20 min**

# String interpolation

- Groovy's String interpolation support can be confusing to many newcomers to the language.
- While Groovy supports declaring a string with either single quotes, or double quotes.
- **Groovy string interpolation should never be used with credentials.!**

```
def username = 'Jenkins'
echo 'Hello Mr. ${username}'
echo "I said, Hello Mr. ${username}"
```

Would result in:

```
Hello Mr. ${username}
I said, Hello Mr. Jenkins
```

```
pipeline {
  agent any
  environment {
    EXAMPLE_CREDS = credentials('example-credentials-id')
  }
  stages {
    stage('Example') {
      steps {
        /* WRONG! */
        sh("curl -u ${EXAMPLE_CREDS_USR}:${EXAMPLE_CREDS_PSW} https://example.com/")
      }
    }
  }
}
```

# Handling parameters

- Declarative Pipeline supports parameters out-of-the-box, allowing the Pipeline to accept user-specified parameters at runtime via the parameters directive.
- Configuring parameters with Scripted Pipeline is done with the properties step, which can be found in the Snippet Generator.
- If you configured your pipeline to accept parameters using the Build with Parameters option, those parameters are accessible as members of the params variable.
- Assuming that a String parameter named "Greeting" has been configured in the Jenkinsfile, it can access that parameter via `${params.Greeting}`:

```
pipeline {
    agent any
    parameters {
        string(name: 'Greeting', defaultValue:
'Hello', description: 'How should I greet the
world?')
    }
    stages {
        stage('Example') {
            steps {
                echo "${params.Greeting} World!"
            }
        }
    }
}
```



# Handling failure

- Declarative Pipeline supports robust failure handling by default via its **post section** which allows declaring a number of different "post conditions" such as:
  - always, unstable, success, failure, and changed.
- The Pipeline Syntax section provides more detail on how to use the various post conditions.

```
pipeline {
  agent any
  stages {
    stage('Test') {
      steps {
        sh 'make check'
      }
    }
  }
  post {
    always {
      junit '**/target/*.xml'
    }
    failure {
      mail to: team@example.com, subject: 'The Pipeline failed :('
    }
  }
}
```

# Using multiple agents

- Pipeline allows utilizing multiple agents in the Jenkins environment from within the same Jenkinsfile, which can help for more advanced use-cases such as executing builds/tests across multiple platforms.

```
pipeline {
  agent none
  stages {
    stage('Build') {
      agent any
      steps {
        checkout scm
        sh 'make'
        stash includes: '**/target/*.jar', name: 'app'
      }
    }
    stage('Test on Linux') {
      agent {
        label 'linux'
      }
      steps {
        unstash 'app'
        sh 'make check'
      }
      post {
        always {
          junit '**/target/*.xml'
        }
      }
    }
    stage('Test on Windows') {
      agent {
        label 'windows'
      }
      steps {
        unstash 'app'
        bat 'make check'
      }
      post {
        always {
          junit '**/target/*.xml'
        }
      }
    }
  }
}
```

# Optional step arguments

- Pipeline follows the Groovy language convention of allowing parentheses to be omitted around method arguments.
- Many Pipeline steps also use the **named-parameter syntax** as a shorthand for creating a Map in Groovy, which uses the syntax **[key1: value1, key2: value2]**.
- For convenience, when calling steps taking only one parameter (or only one mandatory parameter), the parameter name may be omitted.

```
git url: 'git://example.com/amazing-project.git', branch: 'master'  
git([url: 'git://example.com/amazing-project.git', branch: 'master'])
```

```
sh 'echo hello' /* short form */  
sh([script: 'echo hello']) /* long form */
```

# Parallel execution

- Pipeline has built-in functionality for executing portions of Scripted Pipeline in parallel, implemented in the aptly named parallel step.
- Example: instead of executing the tests on the "linux" and "windows" labelled nodes in series, they will now execute in parallel assuming the requisite capacity exists in the Jenkins environment.

```
stage('Build') {
    /* .. snip .. */
}

stage('Test') {
    parallel linux: {
        node('linux') {
            checkout scm
            try {
                unstash 'app'
                sh 'make check'
            }
            finally {
                junit '**/target/*.xml'
            }
        }
    },
    windows: {
        node('windows') {
            /* .. snip .. */
        }
    }
}
```

# Scheduling jobs in Jenkins



- The scheduling function lets you schedule jobs to run automatically during off-hours or down times.
- Scheduling jobs can help you to scale your environment as Jenkins usage increases. This video provides insight on the scheduling function and its various configuration options.

## OBJECTIVE

**Scheduling a job every 10 mins**

## INSTRUCTIONS

1. Look at the next video on how to Schedule Jobs in Jenkins:
  - <https://www.youtube.com/watch?v=JhvVJtYFUm0>
2. Schedule you job for every 10mins.



**20 min**

# Branches and Pull Requests

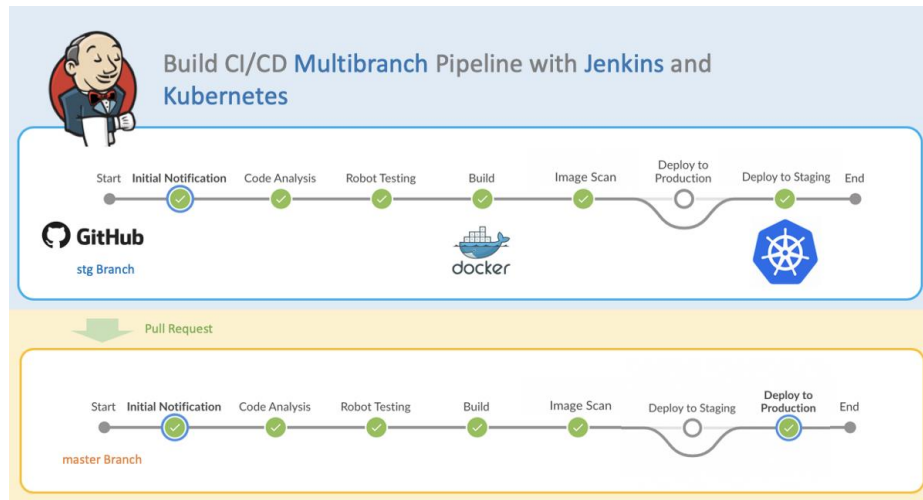


Image Source: [peiruwang.medium.com](https://peiruwang.medium.com)

- The Multibranch Pipeline project type enables you to implement different Jenkinsfiles for different branches of the same project.
- In a Multibranch Pipeline project, Jenkins automatically discovers, manages and executes Pipelines for branches which contain a Jenkinsfile in source control.

## OBJECTIVE

### **Generating a multibranch Pipeline**

## INSTRUCTIONS

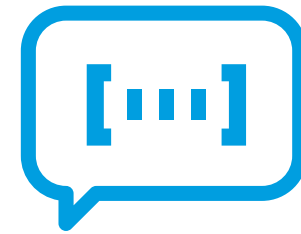
1. Follow next article on how to create a multibranch pipeline in Jenkins:
  - <https://www.jenkins.io/doc/book/pipeline/multibranch/>
2. Use your repository branches for creating a multibranch pipeline.



**20 min**



# 06



## Continuous Integration With Jenkins And GitHub

# What is a WebHook?

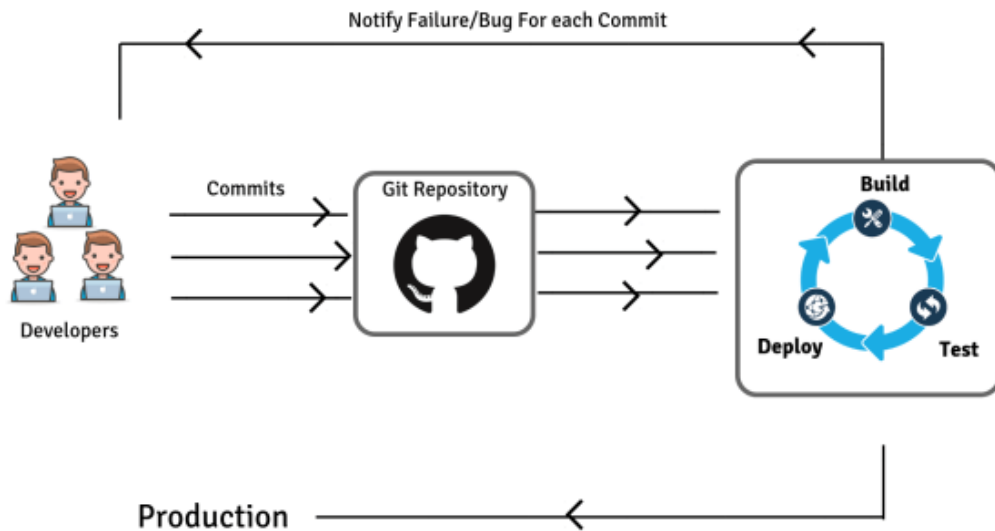


Image Source: qatouch.com

- One of the basic steps of implementing CI/CD is **integrating the Source Control Management tool with your CI tool.**
- This **saves time and keeps your project updated** all the time.
- We are wanting to **automatically trigger each build** on the Jenkins server, after each **Commit** or **Pull Request** is merged on your Git repository

# What is a WebHook?

- If you are familiar with RESTful APIs, then webhooks are nothing more than **a POST request** made by a web service to another service.
- In our case, the web service which is creating this request is **GitHub** and the service where this request is made to is our **Jenkins** server.

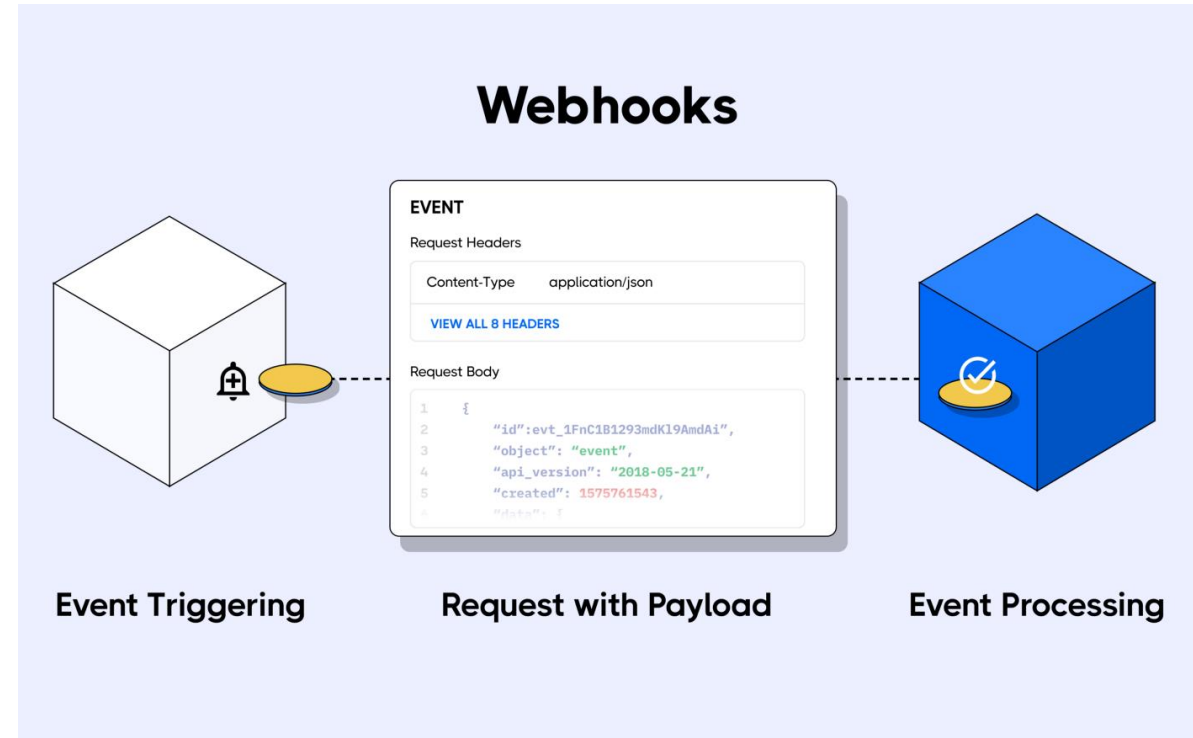


Image Source: medium.com

## OBJECTIVE

### **Configuring Jenkins freestyle project**

## INSTRUCTIONS

1. Follow next instructions for configuring Jenkins for receiving webhooks calls.



**10 min**

1. Create a new freestyle Item.
2. Go to the "Source Code Management" tab to change the settings for **Jenkins GitHub Webhook**.
3. Scroll down and under the "Source Code Management" section, choose the "**Git**" option to let Jenkins know that the new job is for Jenkins GitHub Webhook.
4. **Paste the URL** of the wanted repository in the field of "Repository URL".
5. Now, go to the "**Build Triggers**" tab where you can configure what action Jenkins GitHub Webhook should perform if any change is detected by Jenkins in the GitHub repository.
6. Here, choose the "**GitHub hook trigger for GITScm pulling**" option, which will listen for triggers from the given GitHub repository, as shown in the image below.
7. Now, click on the "Apply" button to save the changes and create a new Jenkins GitHub Webhook for your repository.



## OBJECTIVE

### **Setting Up GitHub Webhook**

## INSTRUCTIONS

1. Follow next instructions for configuring GitHub for sending webhooks calls.



**10 min**

1. Go to the "Settings" option on the right corner.
2. Select the "Webhooks" option and then click on the "Add Webhook" button.
3. Go to your Jenkins tab and copy the URL then paste it in the text field named "**Payload URL**", as shown in the image below.
  1. Append the **"/github-webhook/"** at the end of the URL.
  2. The final URL will look like in this format **"http://address:port/github-webhook/"**.
    1. **Address must be a internet accessible address or IP!**
  3. Select the "Content type" to "application/json" format
4. The "Secret" field is optional. Let's leave it blank for this Jenkins GitHub Webhook.
5. Next, choose one option under **"Which events would you like to trigger this webhook?"**.
6. click on the **"Add Webhook"** button to save Jenkins GitHub Webhook configurations.



## OBJECTIVE

**Test GitHub-Jenkins connection**

## INSTRUCTIONS

1. Follow next instructions for testing the Github-Jenkins connections.



**10 min**



## **Try with simple push**

1. Set the GitHub wekhook for simple push.
2. Now make some changes in your code.
3. Push to GitHub
4. Checkout the Jenkins build is automatically triggered.

## **Try with pull request**

1. Set the GitHub wekhook for pull-request merge.
2. Now make some changes in your code.
3. Enter in Pull request process with your partner
4. Checkout the Jenkins build is automatically triggered.



## OBJECTIVE

**Test GitHub-Jenkins connection with branches**

## INSTRUCTIONS

1. Create a webhook configuration for an specific branch in your repository.



**10 min**

## OBJECTIVE

### **Integrating GitHub webhook in a Jenkinsfile**

## INSTRUCTIONS

1. Follow next instructions for integrating GitHub webhook in a Jenkinsfile.

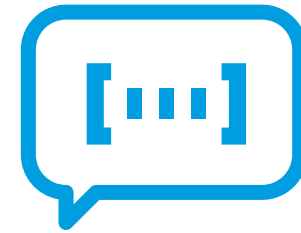


**15 min**

1. Create a Pipeline project in Jenkins
2. Choose Pipeline **script from SCM**
3. For the SCM dropdown, pick **Git**, and below, in the Repository URL, type (or paste) the full GitHub repo URL.
4. Now, type the **Script Path**, which is going to be the path of the Jenkinsfile.
  1. Add a pipeline script.
5. Test the pipeline manually.
6. Push some changes in your repository.
7. Checkout the Jenkins build is automatically triggered.



07



# Jenkins Metrics

# Jenkins Metrics Plugins



- There are various **plugins** which are available in Jenkins to showcase metrics for builds which are carried out over a period of time.
- These metrics are useful to **understand your builds and how frequently they fail/pass over time.**
- As an example, let's look at the 'Build History Metrics plugin'.

This plugin calculates the following metrics for all of the builds once installed:

- Mean Time To Failure (MTTF)
- Mean Time To Recovery (MTTR)
- Standard Deviation of Build Times

## OBJECTIVE

### **Installing Build History Metrics plugin**

## INSTRUCTIONS

1. Follow next instructions for installing and testing Build History Metrics plugin



**15 min**

1. Go to the Jenkins dashboard and click on Manage Jenkins.
2. Go to the Manage Plugins option.
3. Go to the Available tab and search for the plugin '**Build History Metrics plugin**' and choose to 'install without restart'.
  1. Wait for success installation confirmation.
4. When go to Job page, you will see a table with the calculated metrics. Metric's are shown for the last 7 days, last 30 days and all time.
5. Launch some builds forcing success and fail scenarios.
6. Watch at the metrics.





## OBJECTIVE

### **Installing Hudson global-build-stats plugin**

## INSTRUCTIONS

1. To see overall trends in Jenkins, there are plugins available to gather information from within the builds and Jenkins and display them in a graphical format. One example of such a plugin is the 'Hudson global-build-stats plugin'.
2. So let's go through the steps for this.



**15 min**

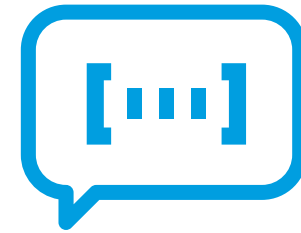
1. Go to the Jenkins dashboard and click on Manage Jenkins.
2. Go to the Manage Plugins option.
3. Go to the Available tab and search for the plugin '**Hudson global-build-stats plugin**' and choose to 'install without restart'.
  1. Wait for success installation confirmation.
4. Go to the Jenkins dashboard and click on Manage Jenkins > In the Manage Jenkins screen, scroll down and now you will now see an option called 'Global Build Stats'. Click on this link.
5. Click on the button 'Initialize stats'. This gathers all the existing records for builds which have already been carried out and charts can be created based on these results.
6. Once the data has been initialized, it's time to create a new chart. Click on the 'Create new chart' link.



7. A pop-up will come to enter relevant information for the new chart details.  
Enter the following mandatory information:
  1. Title – Any title information, for this example is given as 'Demo'
  2. Chart Width – 800
  3. Chart Height – 600
  4. Chart time scale – Daily
  5. Chart time length – 30 days
8. The rest of the information can remain as it is. Once the information is entered, click on Create New chart.
9. You will now see the chart which displays the trends of the builds over time.
10. If you click on any section within the chart, it will give you a drill down of the details of the job and their builds.



07



# Python and Jenkins

# Python and Jenkins



- Unlike compiled languages, **Python doesn't need a "build" per se**. Python projects can still benefit greatly from using Jenkins for continuous integration and delivery.
- In the Python ecosystem there are tools which can be integrated into Jenkins for installing, packaging, testing/reporting.

## OBJECTIVE

### Installing Python plugins

## INSTRUCTIONS

Install next plugins for Python.

1. **Phyton plugin:** <https://wiki.jenkins.io/JENKINS/Python-Plugin.html>
2. **Pylint:** <https://pylint.pycqa.org/en/latest/>



15 min



# EXERCISE

---

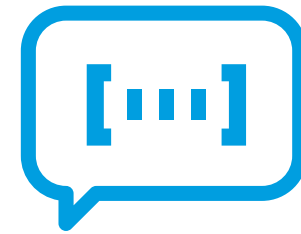
**More on Jenkins**

- Visit the next page for more jenkinsfile examples:
  - <https://www.jenkins.io/doc/pipeline/examples/>
- Try some of them.



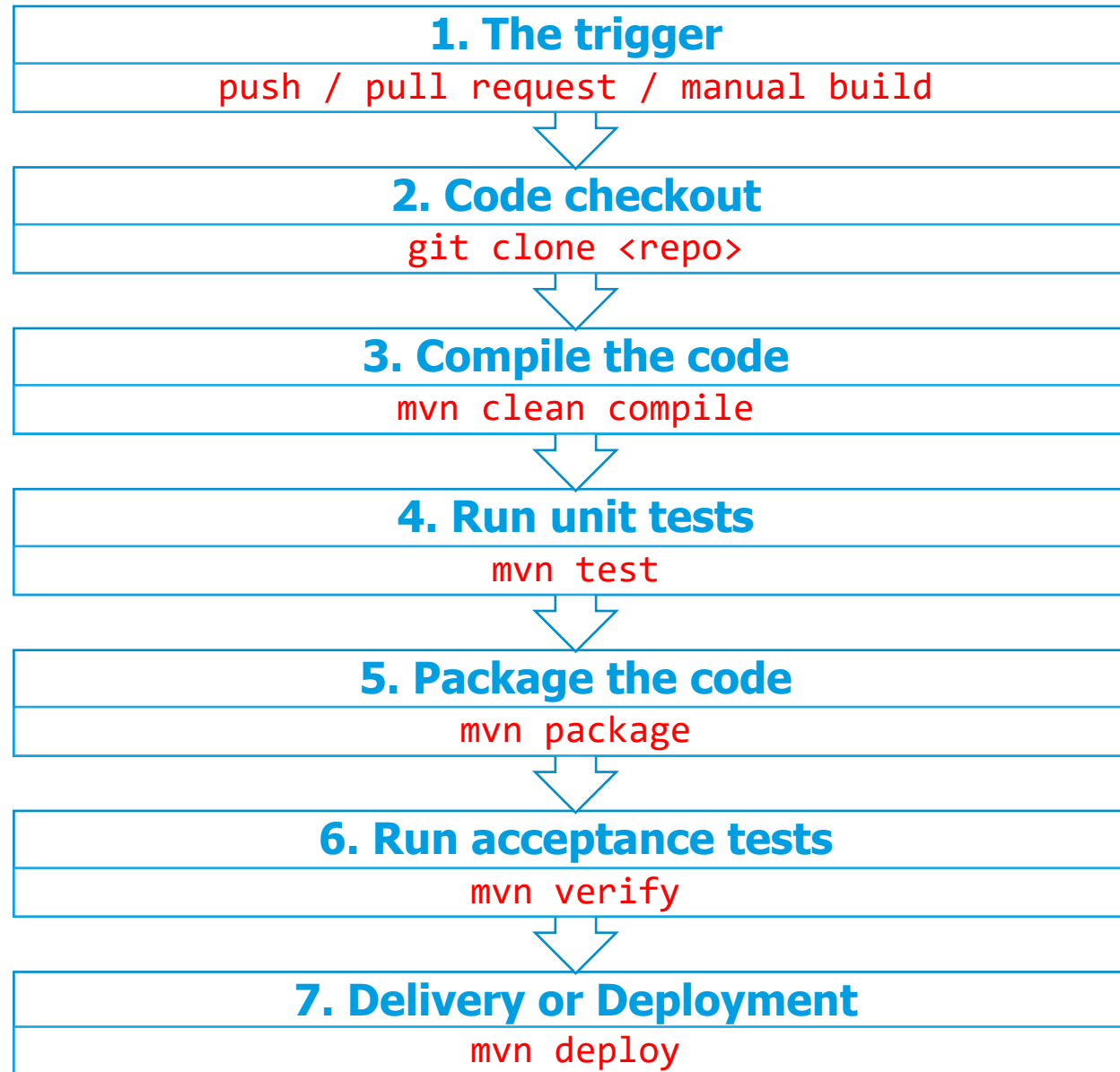


# A1



## Jenkinsfile guide

# 7 essential stages of a CI/CD pipeline



# Jenkinsfile structure

Our pipeline structure should correspond to the meaning stages in our project.

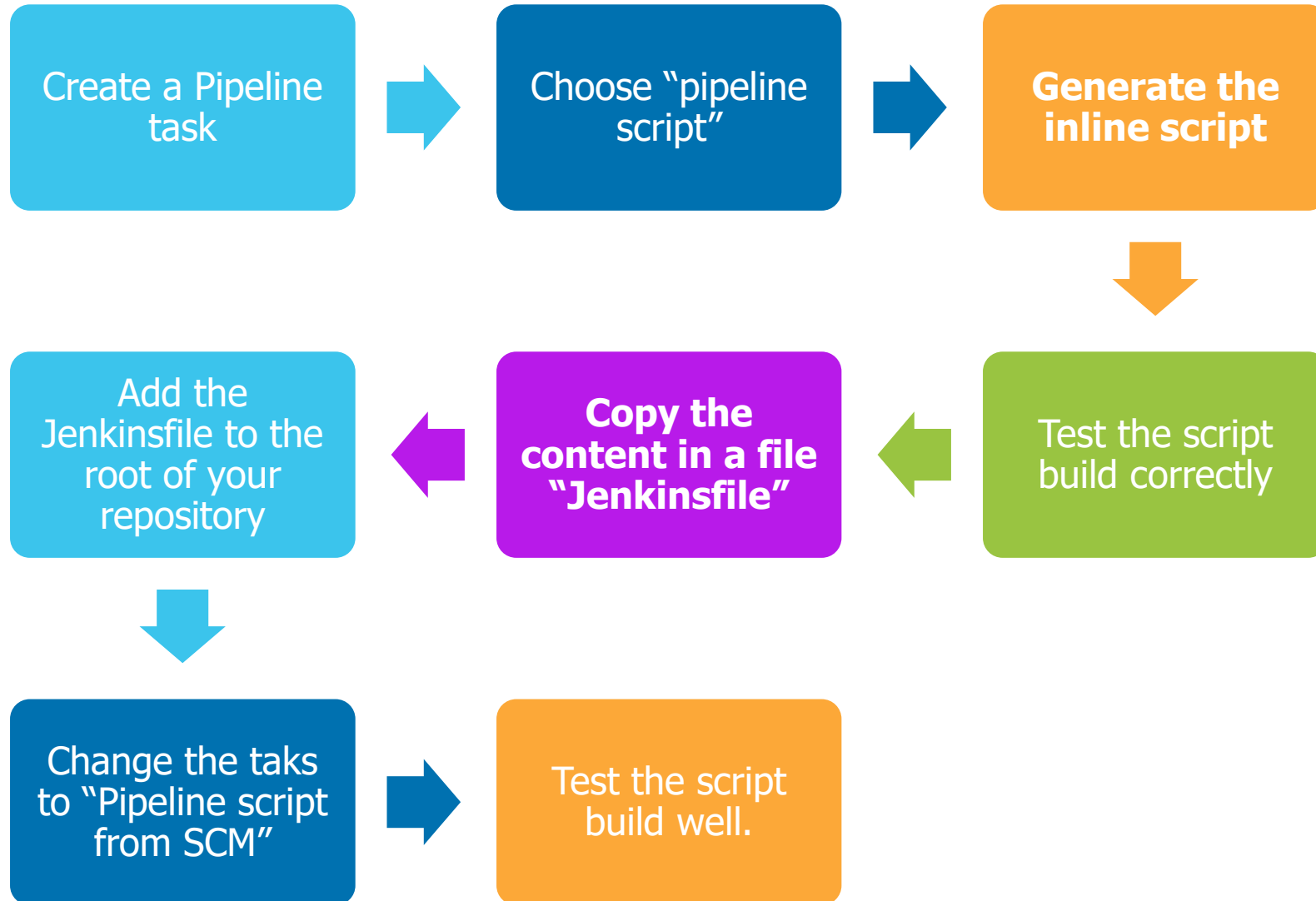
Pipeline:

- Checkout: this is needed to clone the code
- Compile
- Test
- ...

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                echo 'Checkout ...'
            }
        }
        stage('Compile') {
            steps {
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Package') {
            steps {
                echo 'Packaging...'
            }
        }
        stage('Acceptance test') {
            steps {
                echo 'Acceptance...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }
}
```

# How to create a Jenkinsfile?



# Common commands

## Checkout

```
git branch: '<branch>',url: '<repo>'
```

## execute a shell or bat command

```
bat '<command>'
```

```
sh '<command>'
```

## change dir and execute steps

```
dir ('standalone'){  
  bat 'dir'  
  bat 'mvn clean test'  
}
```

## environment variables

```
environment {  
  DB_ENGINE = 'sqlite'  
}  
...  
steps {  
  bat 'set | grep JAVA_HOME'  
  echo "Database engine is ${DB_ENGINE}"  
}
```

# Common commands

## post action

```
post {
  always {
    echo 'One way or another, I have finished'
    deleteDir() /* clean up our workspace */
  }
  success {
    echo 'I succeeded!'
  }
  failure {
    echo 'I failed :('
  }
}
```

## execution parameters

```
parameters {
  choice(
    choices: ['greeting', 'silence'],
    description: "",
    name: 'REQUESTED_ACTION')
}
```

# Common commands

## conditionals

```
stage ('Speak') {  
  when {  
    // Only say hello if a "greeting" is requested  
    expression { params.REQUESTED_ACTION == 'greeting' }  
  }  
  steps {  
    echo "Hello, bitwiseman!"  
  }  
}
```



# Next steps



# Thanks!

Follow us:

