**WHAT IS JENKINS**

* #1 Continuous Integration and Delivery server
* Created by Kohsuke Kawaguchi in 2006
  + Original project: "Hudson", renamed "Jenkins" in 2011
* An **independent** and **active** community ([https://jenkins.io](https://jenkins.io/))

## CONTINUOUS DELIVERY (CD)

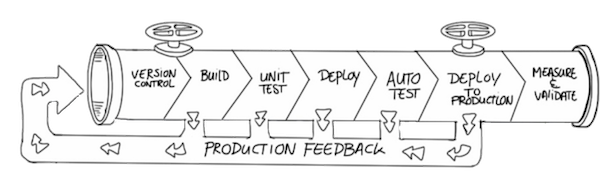
* Teams produce software in short cycles
* Software is built and tested frequently
  + SCM hooks/webhooks detect source code changes and trigger the Pipeline to run
* Software can be reliably released at any time
* Applications in production can be incrementally updated
* Requires a straightforward and repeatable process
* Continuous Delivery means that every change is ready to be deployed

## CONTINUOUS DEPLOYMENT

* Extends Continuous Delivery
* Every change is automatically deployed into production

**WHAT IS CD FLOW?**

* CD is about setting a "flow", from SCM commit to deployed application



* Flow begins when new code is checked into Source Code Management

## PIPELINE AND CD FLOW

* New code is built and tested with the code that is already in SCM
  + One Pipeline can build the code for various platforms such as JVM 8, JVM 11, and JVM 13;  
    or Linux, macOS and Windows; or iOS and Android
* Several test stages are usually defined in the flow
  + One Pipeline can run different levels of tests on different platforms
* When the code passes all tests, it is ready to be deployed to production

## PROJECT TYPES

The Continuous Delivery build flow is defined as a project, of one of the following types:

* Freestyle Projects
* Pipeline Projects
  + Declarative Pipeline
  + Scripted Pipeline

## FREESTYLE ("CHAINED") PROJECTS

* Original Jenkins method for defining the build flow
* Use job orchestration tools such as the Job DSL plugin or Jenkins Job Builder
* Are still supported but have limitations for Continuous Delivery and Deployment

## FREESTYLE LIMITATIONS

* Provides only sequential steps
* A job does not "survive" upon restarts (even planned restarts)
* Chaining job with upstream/downstream:
  + Is only GUI-based; not code
  + Does not provide centralized configuration

## FREESTYLE CD FLOW

A Real-world CD Flow requires features that Freestyle Projects cannot provide:

* Requires (Complex) Conditional Logic
* Requires Resources Allocation And Cleanup
* Involves Human Interaction For Manual Approval
* Should Be Resumable At Some Point On Failure

## JENKINS PIPELINE

[jenkins.io/doc/book/pipeline](https://jenkins.io/doc/book/pipeline/)

* Jenkins Pipeline is a tool for defining your Continuous Delivery/Deployment flow as code
  + New fundamental "type" in Jenkins that conceptualizes and models  
    a Continuous Delivery process
  + Two syntaxes:
    - **Scripted**: sequential execution, using Groovy expressions for flow control
    - **Declarative**: uses a framework to control execution
  + Uses the Pipeline DSL (Domain-specific Language)
    - programmatically manipulate Jenkins Objects
* Captures the entire continuous delivery process as code
  + Pipeline is not a job creation tool like the Job DSL plugin or Jenkins Job Builder

## PIPELINE BENEFITS (1/2)

The pipeline functionality is:

* **Durable:** The Jenkins master can restart and the Pipeline continues to run
* **Pausable:** can stop and wait for human input or approval
* **Versatile:** supports complex real-world CD requirements (fork, join, loop, parallelize)
* **Extensible:** supports custom extensions to its "DSL" (Domain-specific Language)

## PIPELINE BENEFITS (2/2)

* Reduces number of jobs
* Easier maintenance
* Decentralization of job configuration
* Easier specification through code

## PIPELINE-AS-CODE

* A Pipeline is defined in a Jenkinsfile
  + Uses a DSL based on Apache Groovy syntax
* Deployment flow is expressed as code
  + Can express complex flows, conditionals and such
* The Jenkinsfile is stored on an SCM
  + Works with SCM conventions such as Branches and Git Pull Requests
  + Applies versioning, testing and merging against your CD Pipeline definition
  + Classic Web UI configuration is possible

## JENKINS VOCABULARY 1/2

* **Master:**
  + Computer, VM or container where Jenkins is installed and run
  + Serves requests and handles build tasks
* **Agent:** (formerly "slave")
  + Computer, VM or container that connects to a Jenkins Master
  + Executes tasks when directed by the Master
  + Has a number and scope of operations to perform.
* **Node** is sometimes used to refer to the computer, VM or container used for the  
  Master or Agent; be careful because "Node" has another meaning for Pipeline

## PIPELINE BENEFITS (2/2)

* Reduces number of jobs
* Easier maintenance
* Decentralization of job configuration
* Easier specification through code

## PIPELINE-AS-CODE

* A Pipeline is defined in a Jenkinsfile
  + Uses a DSL based on Apache Groovy syntax
* Deployment flow is expressed as code
  + Can express complex flows, conditionals and such
* The Jenkinsfile is stored on an SCM
  + Works with SCM conventions such as Branches and Git Pull Requests
  + Applies versioning, testing and merging against your CD Pipeline definition
  + Classic Web UI configuration is possible

## JENKINS VOCABULARY 1/2

* **Master:**
  + Computer, VM or container where Jenkins is installed and run
  + Serves requests and handles build tasks
* **Agent:** (formerly "slave")
  + Computer, VM or container that connects to a Jenkins Master
  + Executes tasks when directed by the Master
  + Has a number and scope of operations to perform.
* **Node** is sometimes used to refer to the computer, VM or container used for the  
  Master or Agent; be careful because "Node" has another meaning for Pipeline

## JENKINS PIPELINE SECTIONS

* The Jenkinsfile that defines a Pipeline uses a DSL based on Apache Groovy syntax
  + Is structured in sections, called stages
  + Each stage includes steps
  + steps include the actual tests/commands to run
  + An agent defines where the programs and scripts execute
* This is illustrated in the following simple Jenkinsfile:
* pipeline {
* agent { label 'linux' }
* stages {
* stage('MyBuild') {
* steps {
* sh './jenkins/build.sh'
* }
* }
* stage('MySmalltest') {
* steps {
* sh './jenkins/smalltest.sh'
* }
* }
* }

}

## DECLARATIVE VERSUS SCRIPTED PIPELINE

* Two different syntaxes for Pipeline
* Both are built on same Pipeline subsystem
* Both definitions are stored in a Jenkinsfile under SCM ("Pipeline-as-code")
* Both rely on the Pipeline DSL, which is based on Apache Groovy syntax
* Both can use steps built into Pipeline or provided in plugins
* Both support shared libraries

## SCRIPTED PIPELINE

* Executed serially, from top down
* Relies on Groovy expressions for flow control
  + Requires extensive knowledge of Groovy syntax
* Very flexible and extensible
  + Limitations are mostly Groovy limitations
  + Power users with complex requirements may need it
* Novice users can easily mess up the syntax

## DECLARATIVE PIPELINE

* Stricter, pre-defined structure
* Execution can always be resumed after an interruption,  
  no matter what caused the interruption
* Requires only limited knowledge of Groovy syntax
  + Using Blue Ocean simplifies the Pipeline creation process even more
* Encourages a declarative programming model
* Reduces the risk of syntax errors
* Use the **script** step to include bits of scripted code within a Declarative Pipeline only when you have needs that are beyond the capabilities of Declarative syntax

## TOOLS FOR WORKING WITH PIPELINE

* Classic Web UI
* Blue Ocean

## CLASSIC WEB UI

* You code the Jenkinsfile using the text editor of your choice
* Use the Jenkins dashboard to run and configure the Pipeline
* Provides some tools to simplify the process, such as the Declarative Directive Generator and Snippet Generator that generate a line of code based on the task that is required and information you input to an appropriate form
* Supports all features for Declarative and Scripted Pipelines

## BLUE OCEAN PIPELINE EDITOR

* Visual editor for creating and editing Declarative Pipelines
* Round-trip to the Jenkinsfile file that is the Pipeline
* Add/remove configuration, stages and steps with a GUI
  + Also includes a code editor
* Supports Git, GitHub, GitHub Enterprise and Bitbucket Server and Cloud
* Supports most features that the Classic Web UI supports
* Provides visualization and analysis for the Pipeline run
* Generates a Jenkinsfile and stores it in the source code repository

## BLUE OCEAN EDITOR LIMITATIONS

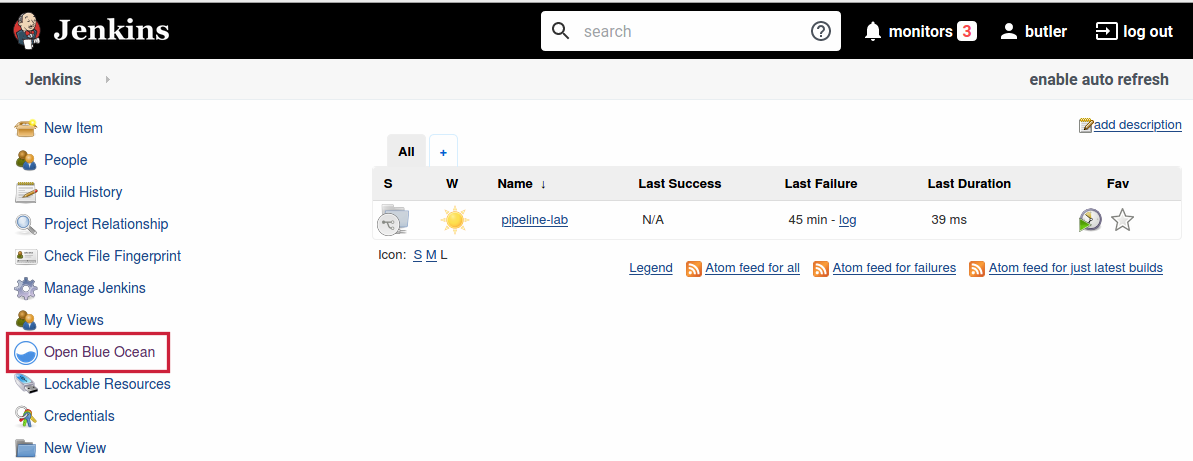
* The Blue Ocean graphical editor does not support all features
* Use the Blue Ocean code editor to incorporate such features into your Pipeline
* The Directive Editor can help with the syntax
* Blue Ocean can run a Pipeline that includes these features
  + Specify environment variables within a block
  + post section that defines actions that run at the end of a Pipeline run or an individual stage
  + Apply options to the Pipeline
  + Use the when directive
  + Define and access credentials

## FIRST PIPELINE

* For our first pipeline, we will:
  + Create and run a new Pipeline for the sample project using Blue Ocean graphical editor
    - This Pipeline builds, tests and deploys the software
  + Edit and run an existing Pipeline using Blue Ocean Editor
  + Save changes to the master branch of the Git repository and see them run
  + View resulting Jenkinsfile on the Git repository
* Note that you can instead use a text editor to create a Declarative Pipeline

## INITIALIZE BLUE OCEAN

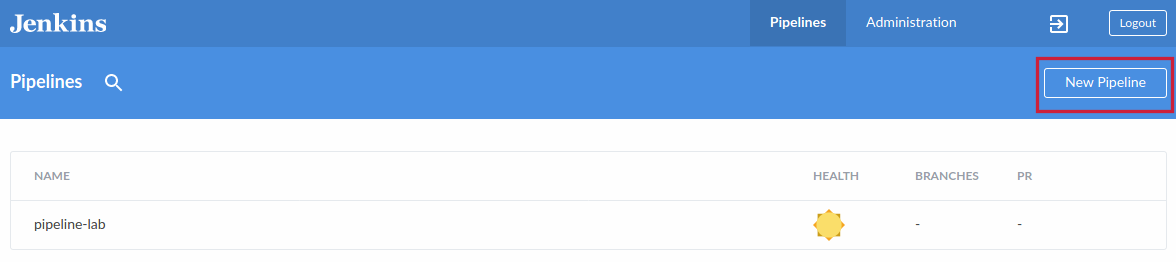
* When you open Jenkins, the console is displayed:



* Click on "Open Blue Ocean" in the left sidebar of the Jenkins console

## ENTERING BLUE OCEAN

* The following screen appears:

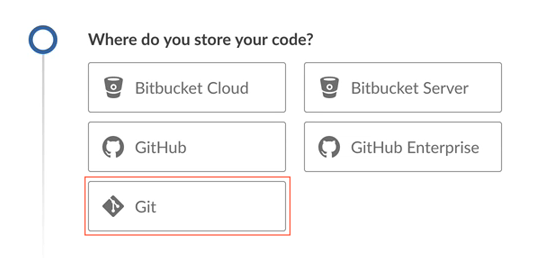


* Click "New Pipeline"

## REGISTER WITH SOURCE CODE MANAGEMENT

## SELECT GIT AS YOUR PIPELINE SOURCE

* Before you can create a Pipeline, you must register Blue Ocean with the  
  Source Code Management repository you are using.
* The first step is to identify the type of repository you are using.
  + For this class, select "Git":



## ABOUT SOURCE CODE MANAGEMENT

* Jenkins can work with almost any Source Code Management system.
* An advanced Git server such as Bitbucket or GitHub is required for pull/merge  
  request integration in Jenkins, which validates the Pipeline code by Pull Request
* Examples in this course use Gitea to host a Git SCM so you can use standard  
  Git commands without having any external service access

## USING A DIFFERENT SCM

* If you are using a different SCM in your production environment, you must:
  + Install the appropriate plug-in for that SCM
  + Adjust some of the commands and practices for interacting with that SCM
* Pipeline principles are the same

## CONNECT REPOSITORY

* When you select "Git", Blue Ocean provides a box where you can put the URL of the Git repository. The next slides show you how to get the URL that you can copy into this box

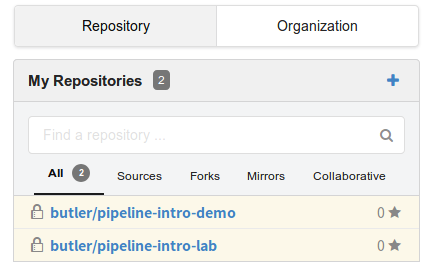


## PICK REPOSITORY (1/4)

* To provide the repository address (URL), access the Git repository page for the  
  demo project by clicking on the "> Gitserver" link on your "CloudBees Lab" home page
  + Direct link: <http://localhost:5000/gitserver/butler/pipeline-demo>
* Click on "Sign In" in the upper right of the screen

## PICK REPOSITORY (2/4)

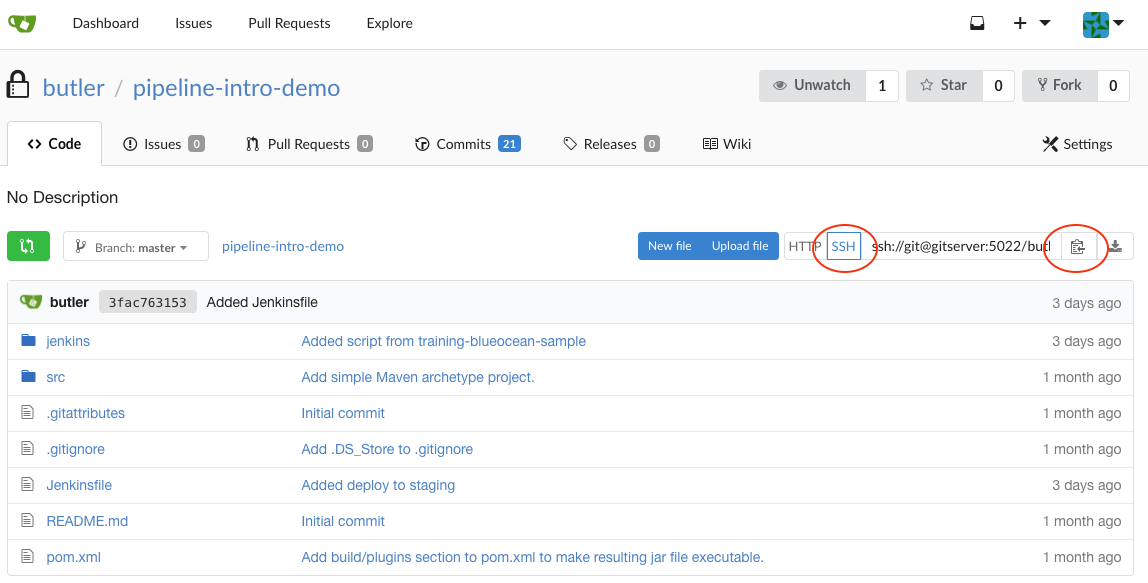
* After you sign in, find the "Repository" box in the right portion of the display:



* Click on the repository you want to use.
  + To follow along with the presentation, choose "butler/pipeline-demo"

## PICK REPOSITORY (3/4)

* On the right end of the repository description bar, select the SSH button to display the SSH address
* Click on the copy icon at the right end of the box to copy the SSH address
  + SSH address: ssh://git@gitserver:5022/butler/pipeline-demo.git



## PICK REPOSITORY (4/4)

* Go back to your Blue Ocean page
* Paste the URL into the box on the screen
* When supplied with a valid URL, Jenkins detects that the repository  
  is available for SSH access and auto-generates an SSH key for you
* You need to put the public part of this SSH key in the remote project
* Gitea, the local Git Server, supports SSH out of the box
  + GitHub, Bitbucket, GitLab also support SSH access.

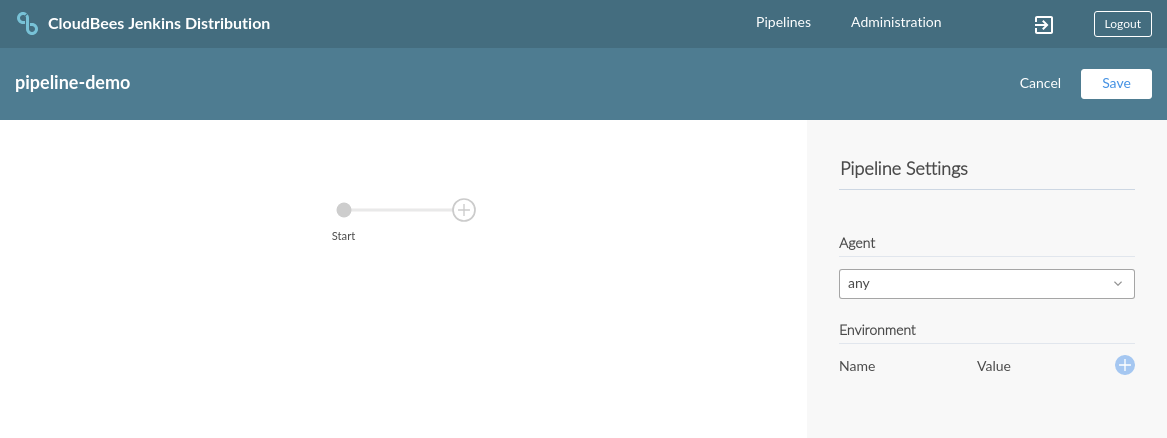
## AUTOGENERATED SSH KEY

* Blue Ocean displays the generated SSH key for the user butler
  + This key is used by Jenkins to authenticate on the Gitea Server to be able to access  
    the source code.
* Your environment is already configured: the SSH key is already loaded in the Gitea Server.
  + In Blue Ocean, you can click on the "Create Pipeline" button at the bottom of the screen.

## CREATING YOUR FIRST PIPELINE

## BLUE OCEAN PIPELINE EDITOR

* When you click on the "Create Pipeline" button after registering your SSH key with Gitea, the following screen appears:



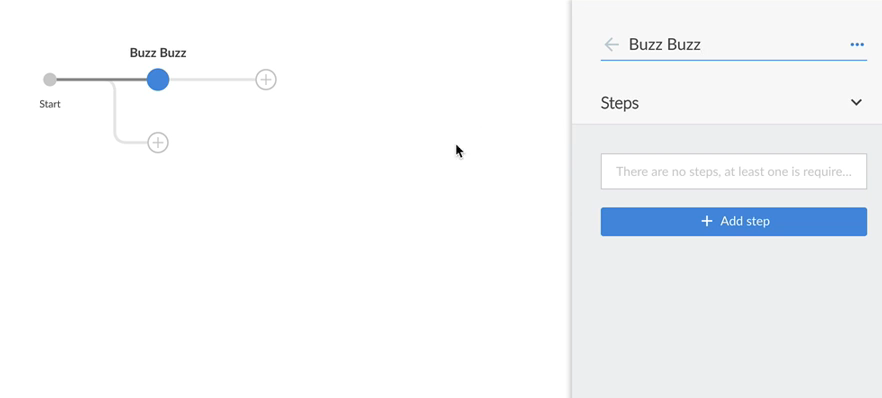
## STAGES AND STEPS

The basic Declarative Pipeline structure is a sequence of stages which include steps:

* A stage groups tasks to be done; it includes steps
  + Stages are the logical segmentation of a Pipeline
* A step defines an actual task, such as execute a script or program
* Any non-setup work should occur within a stage block
  + Stages are the logical segmentation of a Pipeline
* The Pipeline code itself executes on the Jenkins Master node
* Most of the code inside stage blocks executes on agent nodes

## ADD BUZZ BUZZ STAGE

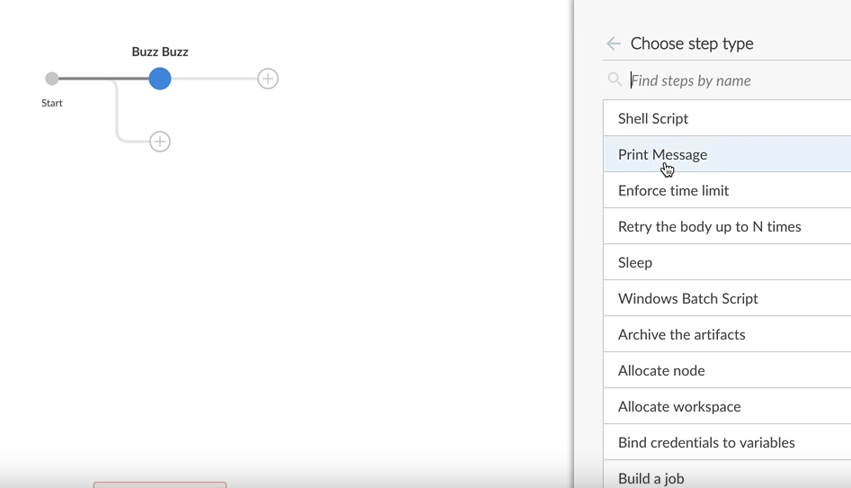
* Click on the circle with the "+" sign to create a stage.
* In the right frame of the screen, set the name of the stage to "Buzz Buzz".
* The "agent" defines the environment to use to run this stage; for now, we will set this as "any".



## ADD STEP TO STAGE

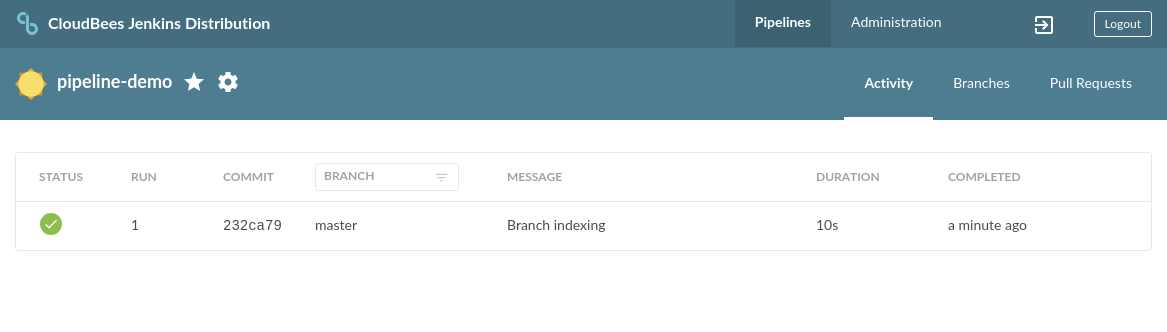
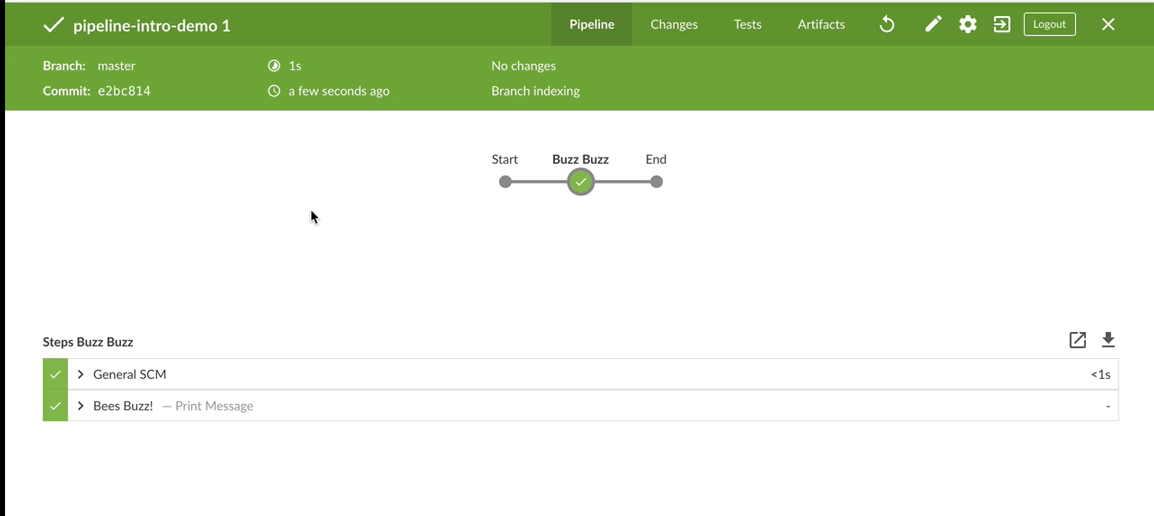
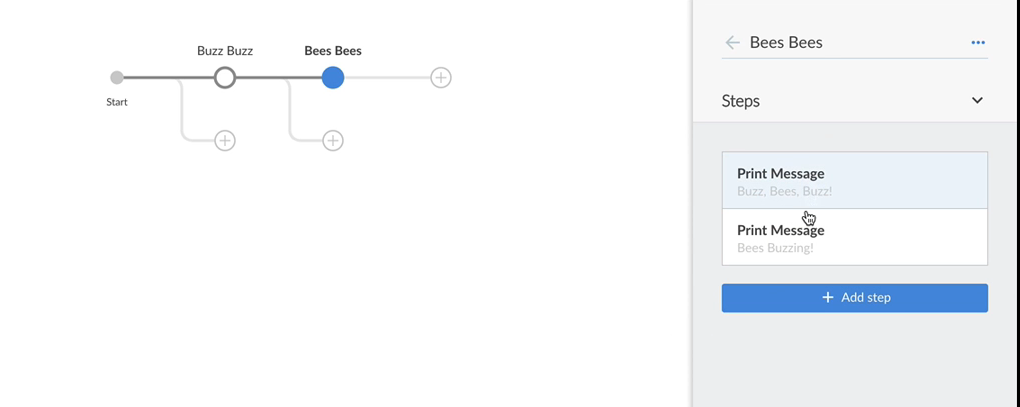
* Click on "Add a step" in the right frame.
* Choose the "Print Message" step type.
* In the box that is displayed, type:

Bees Buzz!



## SAVE PIPELINE



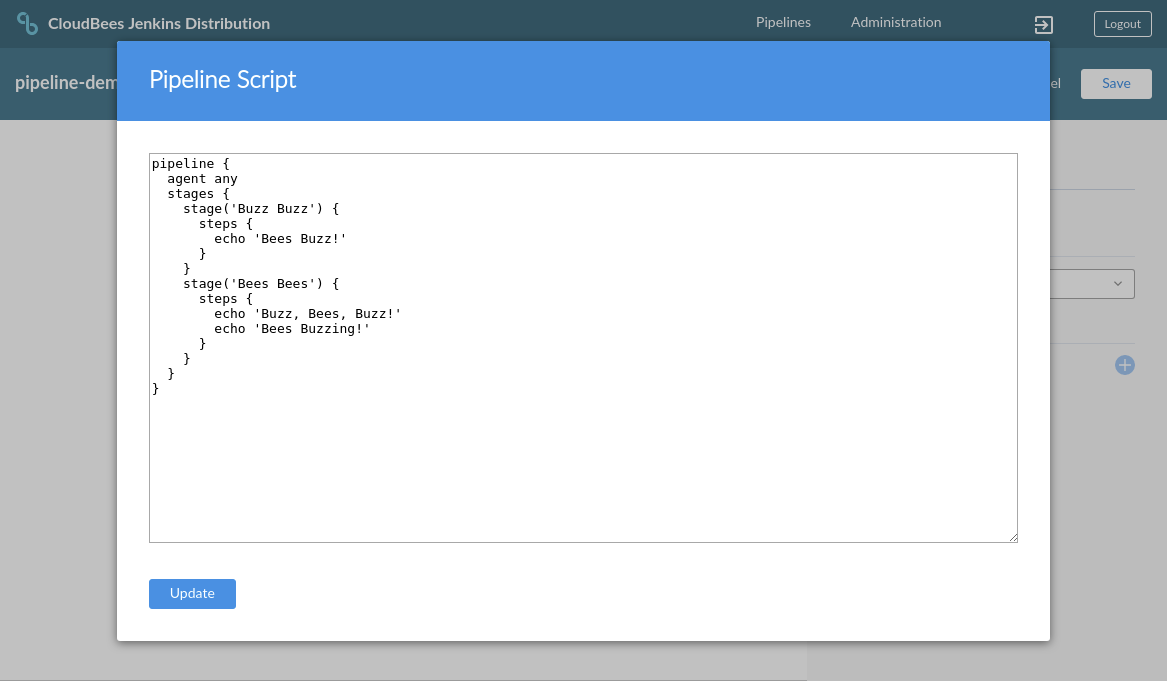
* Describe what you did: "Created new pipeline"
* Leave the default "Commit to master" set
* Click on "Save & run"
* **WATCHING PIPELINE RUN**
* 
* **IEW PIPELINE RUN DETAILS**
* 
* **ADD BEES BEES STAGE**
* 

## INSTRUCTIONS: ADD BEES BEES STAGE

* Click the circle with the "+" sign to the right.
* Name this stage "Bees Bees".
* Click "Add a step" and choose "Print Message". A box is displayed where you type the content of the message; type "Buzz, Bees, Buzz!"
* Add another "Print Message" step with the text "Bees Buzzing!"
* To delete a step, select that step, click on the three dots at the upper right of the right frame, then click on the "Delete" box.

## VIEW JENKINSFILE

* Click ctrl-S (Linux or Windows) or cmd-S (macOS) to pop an edit window that displays an editable version of the current Jenkinsfile



## JENKINSFILE STRUCTURE

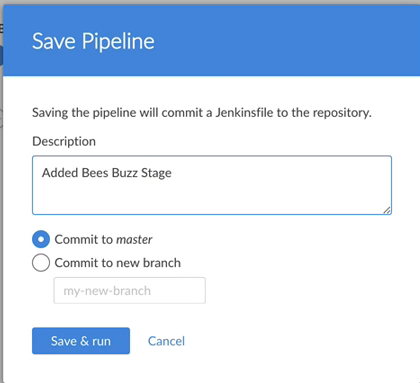
* The Jenkinsfile is your Pipeline (Pipeline-as-code)
* It uses a DSL based on Apache Groovy syntax.
* Text file that can be edited with any text editor
* For Declarative Pipeline, the first line is the **pipeline** block
* The first line under **pipeline** defines the global agent for the Pipeline; we will discuss this more below.

## JENKINS PIPELINE SECTIONS

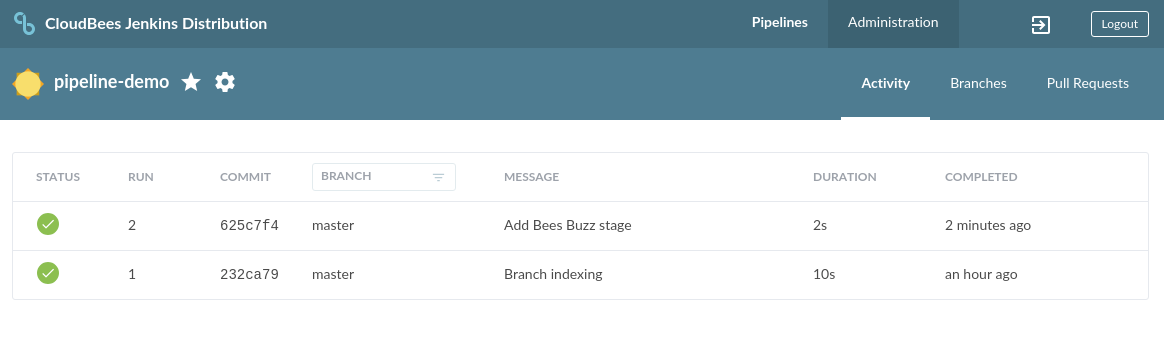
* **agent** — Specifies where the Pipeline or a specific **stage** executes
  + Declarative Pipeline requires a global declaration of **agent** as the first line in the pipeline block. It is possible to use different agents for different stages but the global agent statement is always required.
* **stage** — A conceptually distinct subset of the Pipeline, such as "MyBuild", "MyTest", or "MyDeploy". Used to present Pipeline status/progress. Stage labels should be significant for your application.
* **steps** — A series of distinct tasks inside a stage.

The Jenkinsfile can also include Sections, Directives, Options and so forth; for full information about Pipeline syntax, see [jenkins.io/doc/book/pipeline/syntax](https://jenkins.io/doc/book/pipeline/syntax)

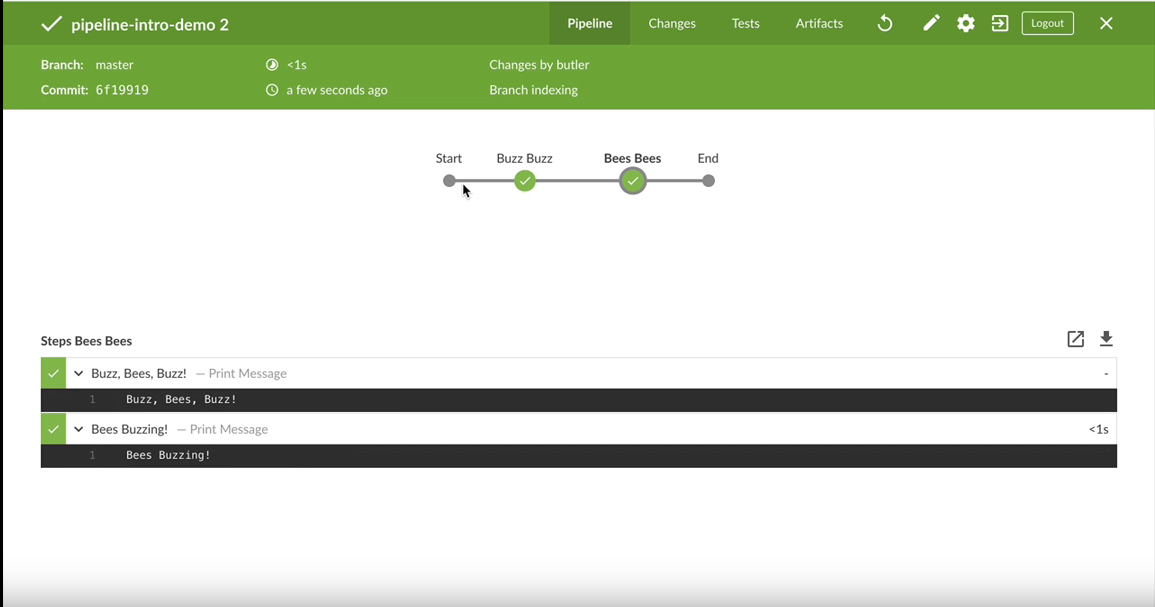
## SAVE AND RUN PIPELINE



**WATCHING REVISED PIPELINE RUN**



**VIEW REVISED PIPELINE RUN DETAILS**



**LAB EXERCISE**

[Create and Edit an Example Pipeline](https://s3.amazonaws.com/cloudbees-training-materials/training-pipeline-fundamentals/master/selfpaced/labs/Create_and_Edit_an_Example_Pipeline.html)

1.17

Lab ExerciseCreate and Edit an Example Pipeline



## Solution Groovey script

pipeline {

agent any

stages {

stage('Fluffy Build') {

steps {

echo 'Placeholder'

sh 'echo Edited Placeholder.'

}

}

stage('Fluffy Test') {

steps {

sh 'sleep 5'

sh 'echo Success!'

}

}

stage('Fluffy Deploy') {

steps {

echo 'Placeholder'

}

}

}

}

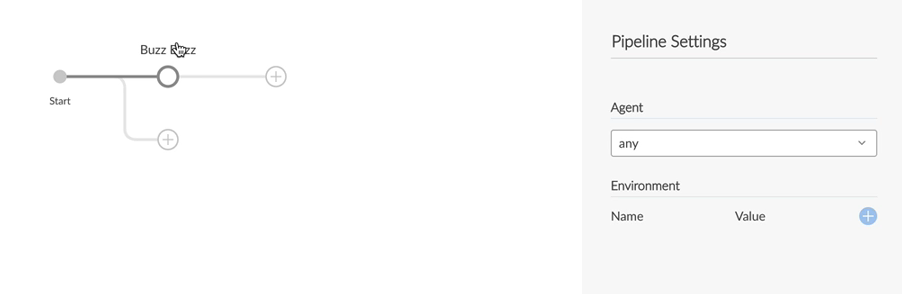
## SAVE PIPELINE TO A BRANCH

## USING BRANCHES

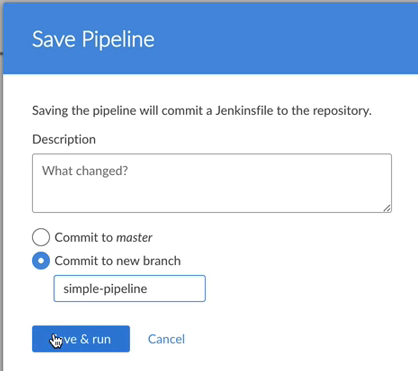
* The master branch contains the current "official" code for your application.
* You and your colleagues should develop and test new code in a branch other than master
  + You do not affect the official master branch as you work on new code
  + If many people are working on the same repo, you do not trip over each other
  + You can work on several items at the same time, but isolate the code for each in its own branch.
* When your Pipeline is complete and you have tested it extensively, you can merge it to master and it becomes part of your "official" code base

## STRIP OUR PIPELINE

* Delete all but the "Buzz Buzz" stage and it’s single step from the pipeline:



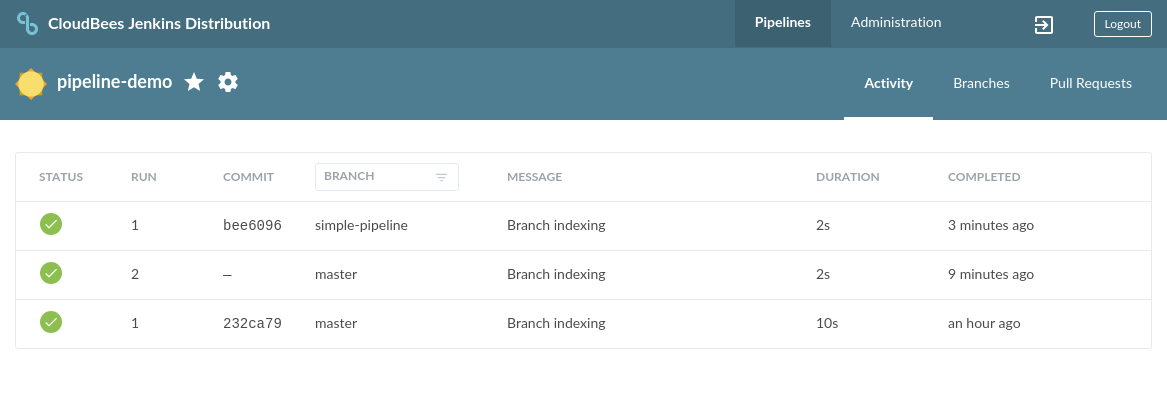
## SAVE TO A BRANCH



## INSTRUCTIONS: SAVE TO A BRANCH

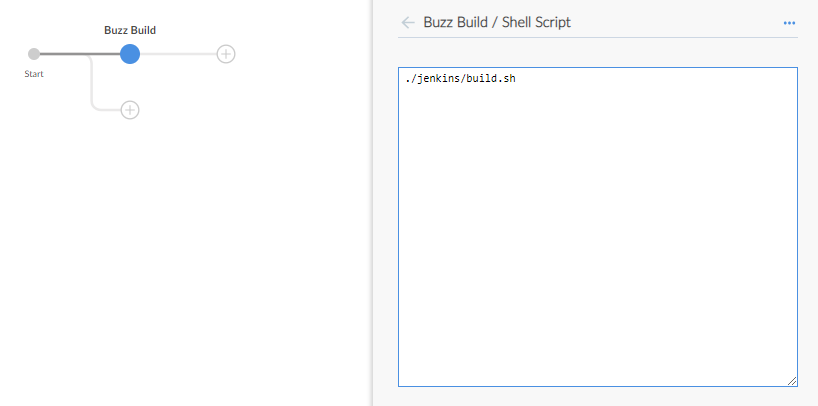
* Click the pencil icon to go back to the Blue Ocean Editor
  + Then click "Save" to save the pipeline
* Type "Start simple pipeline" in the Description box
* Click "Commit to new branch"
* Type the name of your new branch in the box; for example, "simple-pipeline"
* Click "Save & run" when the information is complete.

## WATCH PIPELINE RUN



## CREATE SIMPLE PIPELINE

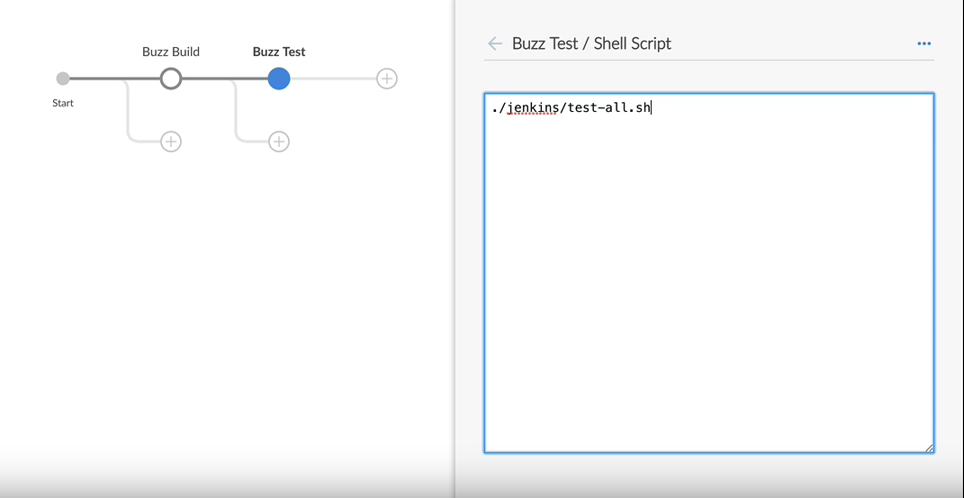
**CREATE BUZZ BUILD STAGE**



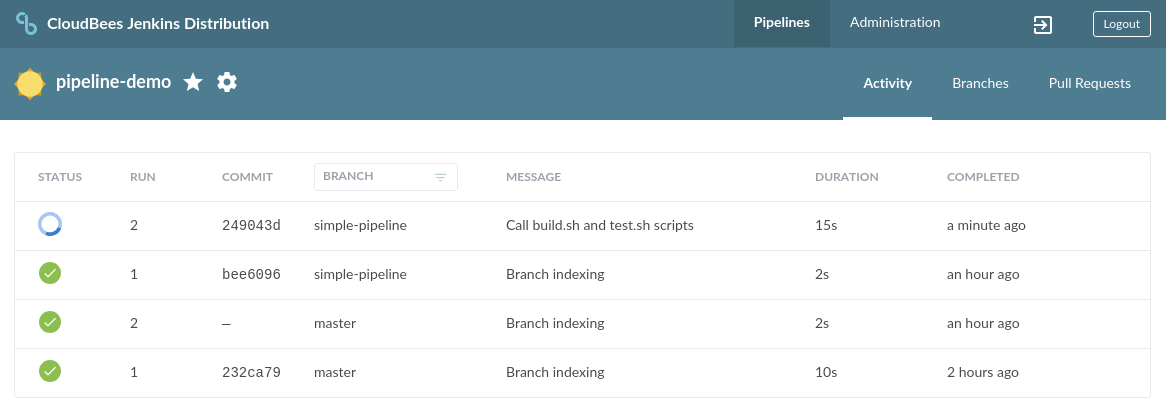
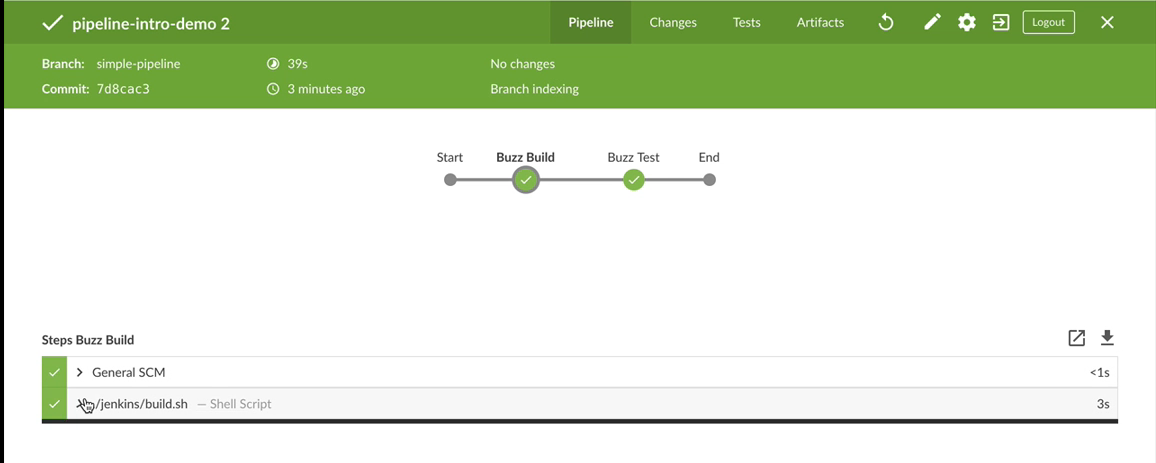
## INSTRUCTIONS: CREATE BUZZ BUILD STAGE

* Rename the "Buzz Buzz" stage to "Buzz Build"
* Delete the "Print Message" step for this stage
* Create a Shell Script step
* Type in ./jenkins/build.sh

## ADD BUZZ TEST STAGE



## INSTRUCTIONS: ADD BUZZ TEST STAGE

* Add another stage called "Buzz Test"
* Add a step to "Buzz Test" to run the Shell Script, ./jenkins/test-all.sh
* Save & run the Pipeline, using "Initial build and test" as the Description
  + Note that Blue Ocean by default saves to the branch we last used.
* **WATCH PIPELINE RUN**
* 
* **VIEW PIPELINE RUN DETAILS**
* 

## ARTIFACTS AND FINGERPRINTS

## WHAT ARE ARTIFACTS?

* An artifact is a file produced as a result of a Jenkins build.
  + The name comes from Maven naming conventions
* A single Jenkins build can produce many artifacts
* By default, they are stored where they are created, so they are deleted  
  when the workspace is wiped, unless they are archived

## ARCHIVE ARTIFACTS

* Jobs can be configured to **archive** artifacts based on filename patterns
* Archived artifacts are available for testing and debugging after the pipeline run finishes
* Archived artifacts are kept **forever** unless a **retention** policy is applied to builds  
  to delete them periodically

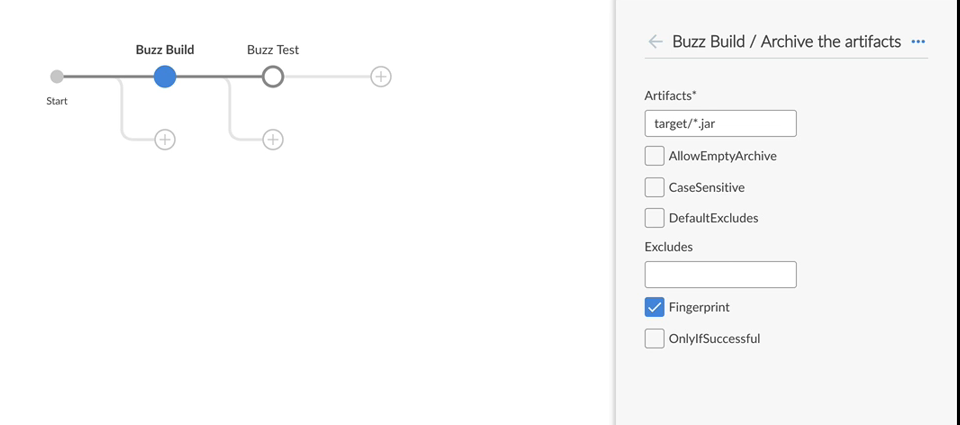
## HOW TO ARCHIVE ARTIFACTS ?

* Use the pipeline step archiveArtifacts
* Requires a pattern to know which artifacts to archive
  + my-app.zip:
    - The file my-app.zip, at the workspace’s root
  + images/\*.png:
    - All files with .png extension in the images folder at the workspace’s root
  + target/\*\*/\*.jar:
    - All files with .jar extension, recursively under the target folder, at the workspace’s root
* Archiving keeps those files in ${JENKINS\_HOME} **forever** unless you delete them

## FINGERPRINTS

* A fingerprint is the **MD5** checksum of an artifact
* Each archived artifact can be fingerprinted; merely check the "Fingerprint" box  
  when you create the archiving step
* Jenkins uses **Fingerprints** to keep track of artifacts without any ambiguity
* A database of all fingerprints is managed on the Master
  + Located in the ${JENKINS\_HOME}/fingerprints directory

## ARCHIVE ARTIFACTS IN THE PIPELINE



## INSTRUCTIONS: ARCHIVE ARTIFACTS IN THE PIPELINE

* In the "Buzz Build" stage, click "+ Add step"
* Search for "archive" to find the "Archive the artifacts" step type
  + Type target/\*.jar in the "Artifacts" box
  + Click the "Fingerprint" box
* This creates an archive that contains the artifact that is the output of the build.sh script

## CODE: ARCHIVE ARTIFACTS

* The Jenkinsfile code that is created includes the archiveArtifacts step  
  with fingerprint: true specified:
* pipeline {
* agent any
* stages {
* stage('Buzz Build') {
* steps {
* sh './jenkins/build.sh'
* archiveArtifacts(artifacts: 'target/\*.jar', fingerprint: true)
* }
* }
* }

}

## ARCHIVES — INTERNAL DETAILS

* Artifacts can be archived in the post section of the Pipeline  
  or at the end of the stage that generated the artifact
* Jenkins tracks these artifacts **forever**, across builds, jobs, nodes and folders
* Once archived, an archive is attached to the build that produced it:
  + All artifacts of a build:

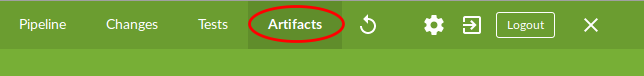
http://${JENKINS\_URL}/job/${YOUR\_JOB}/${BUILD\_NUMBER}/artifact

## USING ARCHIVED ARTIFACTS

* In a production environment, your build chain system (Maven, Gradle, Make, etc.)  
  publishes artifacts to an artifact repository such as Artifactory, Nexus, etc.
* Teams can also deploy artifacts from Jenkins to test environments
* Use the [Copy Artifact Plugin](https://plugins.jenkins.io/copyartifact) to take artifacts from one project (Pipeline run)  
  and copy them to another Project

## ACCESS ARCHIVED ARTIFACTS - BLUE OCEAN

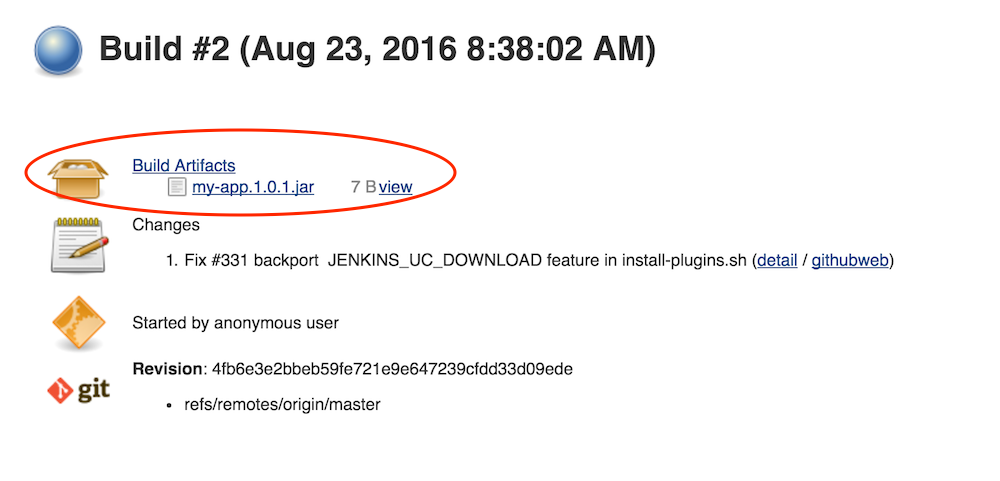
* Artifacts can be viewed in Blue Ocean on the Artifact screen for each build of the Pipeline:



* Each Pipeline generates a pipeline.log artifact when it runs
* Additional artifacts are listed here if the Pipeline is coded to create them

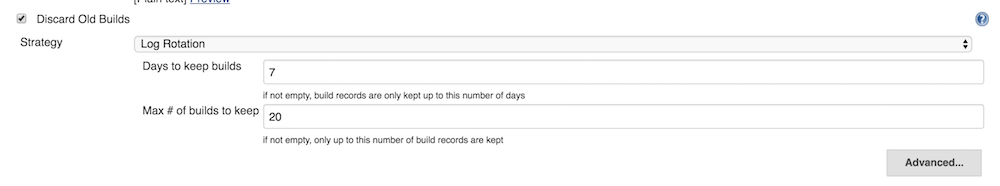
## ACCESS ARCHIVED ARTIFACTS - CLASSIC WEB UI

* Artifacts are also visible on the main Build page in the Classic Web UI:



## ARTIFACT RETENTION POLICY

* Coupled to build retention policy
  + Deleting a build deletes attached artifacts
* **Good practice:** discard build and clean it
  + Driven by age: # days to keep a build
  + Driven by number: Max # of builds to keep
* Important builds can individually be Kept Forever



## GOING FURTHER

* To learn more about artifacts, read the following:
  + Documentation for the [Copy Artifact](https://plugins.jenkins.io/copyartifact) plugin
  + [A Very Quick Guide to Deploying Artifacts with Jenkins](http://codurance.com/training/2014/10/03/guide-to-deploying-artifacts-with-jenkins/)
  + CloudBees Support, [Best Strategy for Disk Space Management: Clean Up Old Builds](https://support.cloudbees.com/hc/en-us/articles/215549798-Deleting-Old-Builds-Best-Strategy-for-Cleanup-and-disk-space-management)

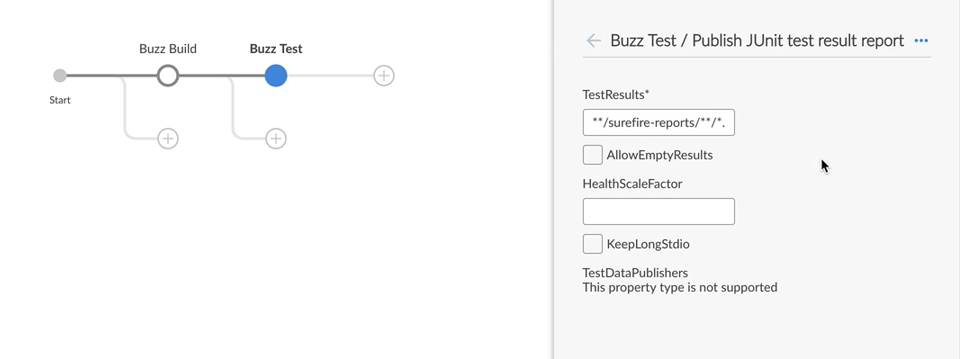
## JUNIT

## WHAT IS JUNIT?

* As an external application, JUnit is a common testing framework for Java programs
* In the Jenkins context, JUnit is a publisher that consumes XML test reports
  + Generates some graphical visualization of the historical test results
  + Provides a web UI for viewing test reports, tracking failures and so forth
  + Useful for tracking test result trends
  + Works with all supported build tools
    - You must specify the appropriate path name for the XML test reports  
      generated by the build tool you are using
* JUnit in this course means the Jenkins publisher

## ADDING JUNIT STEP

* JUnit is available as a step for your Pipeline when the JUnit Plugin is installed on your system.



## ADDING JUNIT STEP — INSTRUCTIONS

* Select the Buzz Test Stage and add a Step: "Archive JUnit-formatted test results"
* Type the path of the XML files that contain your test reports in the "Test Results" box
  + This Pipeline is using Apache Maven so the path is \*\*/surefire-reports/\*\*/\*.xml
  + For other build tools, specify the appropriate path
* The other configuration options are discussed on the JUnit Plugin page
  + In most cases, you can just use the default values
* Save and Run the Pipeline, using "Added test results and artifacts" for the "Description"

## VIEW JENKINSFILE JUNIT

* The JUnit step is coded in the Pipeline as:
* steps {
* sh './jenkins/test-all.sh'
* junit '\*\*/surefire-reports/\*\*/\*.xml'

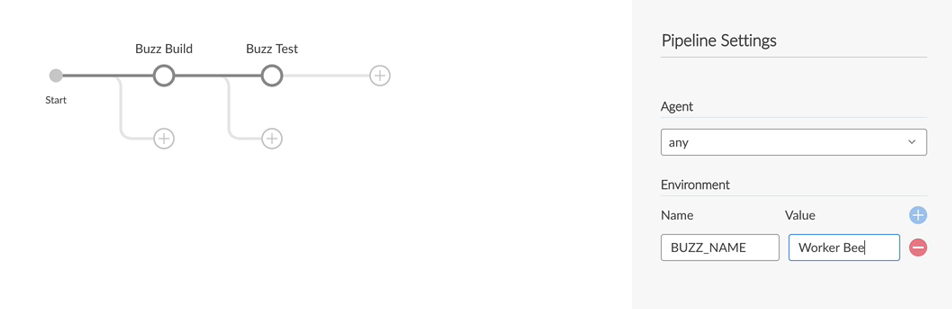
}

## SET ENVIRONMENT VARIABLES

## JENKINS ENVIRONMENT VARIABLES

* Jenkins recognizes some environment variables set in the shell
* Jenkins sets its own specific environment variables
  + For example, BUILD\_NUMBER, NODE\_NAME and JOB\_NAME
* Many plugins define environment variables

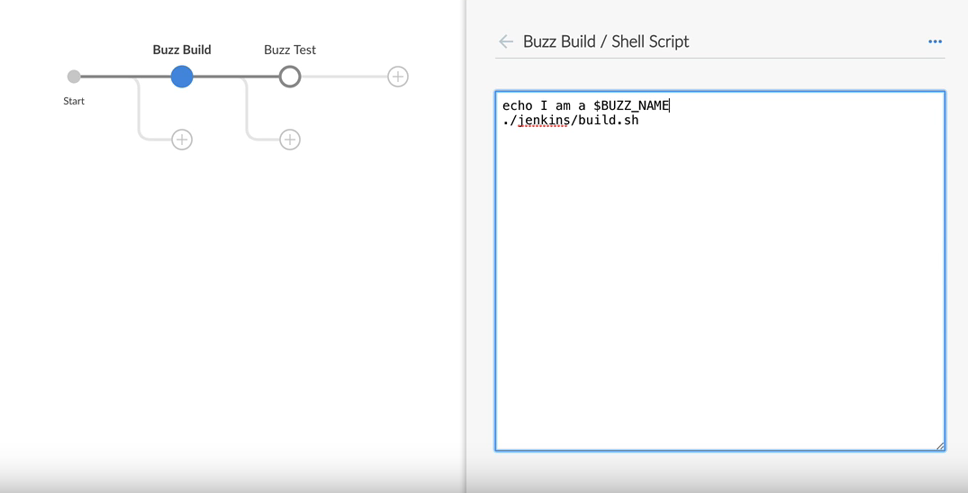
## SET YOUR OWN ENVIRONMENT VARIABLES

* You can set additional environment variables for your Pipeline
* Environment variables are set as directives that define key-value pairs
  + Can be set in the pipeline block (applies to the entire Pipeline)
  + Can be set per stage (applies only to that stage)
* **SET ENVIRONMENT VARIABLE**
* 

## SET ENVIRONMENT VARIABLE

* Use the Pipeline Settings screen
* Click the + sign to the right of the "Name/Value" line under "Environment"
* Type the name of the variable (all capital letters) in the "Name" box
* Type the value of the variable in the "Value" box.
* For our Demo, set the BUZZ\_NAME environment variable to have the Worker Bee value.

## USE ENVIRONMENT VARIABLE



## INSTRUCTIONS: USE ENVIRONMENT VARIABLE

* A simple use of our environment variable:
  + Click on the "Buzz Build" stage
  + Select the existing shell script for edit
  + Add the following line above the existing text in the script:

echo "I am a ${BUZZ\_NAME}"

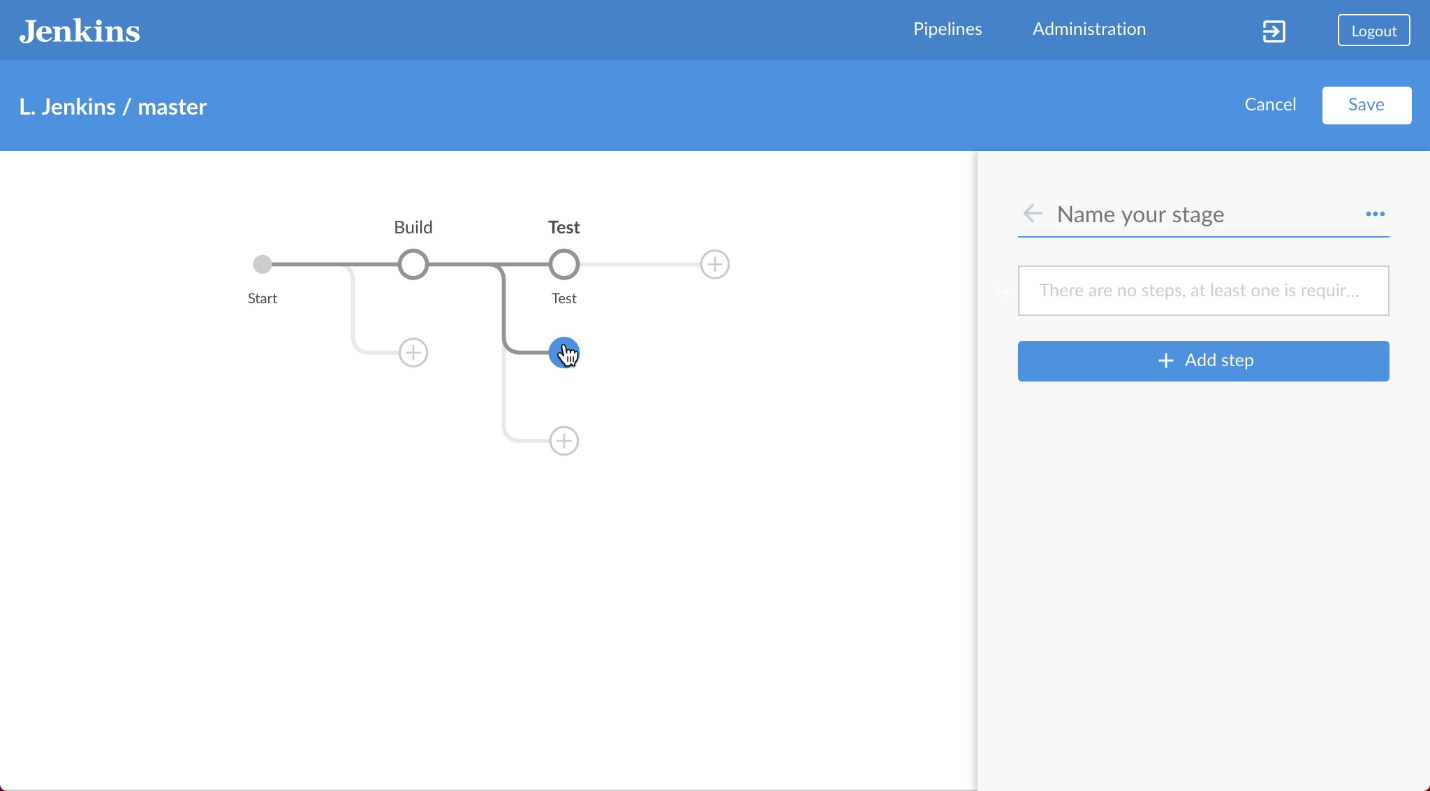
* Now save and run the Pipeline, using the Description: "Added env sample"

## PARALLEL STAGES

## PARALLEL STAGES

* Stages can be run in parallel:
  + Long running stages
  + Builds for different target architectures or OSes (Debian/Fedora, Android/iOS)  
    or different software versions (JVM8/11), et cetera
  + Builds of independent modules of your code
  + Unit and integration tests are tied to the code; run them in parallel

## DECLARATIVE PARALLEL SYNTAX

* Each "parallelized branch" is a stage
* A stage can use either **steps** or **parallel** at the top level; it cannot use both
* A stage within a **parallel** stage can contain **agent** and **steps** sections
* A **parallel** stage cannot contain **agent** or **tools**
* Add failfast true to force all parallel processes to abort if one fails
* **ADDING PARALLEL STAGES**
* 

## INSTRUCTIONS: ADDING PARALLEL STAGES

* Click on a + button under an existing stage to add a parallel stage
  + In this case, click on the + button under "Buzz Test"
* Then add steps to that new stage
* Add as many parallel stages as makes sense
* Note that Pipeline does not support arbitrary parallel stage depth

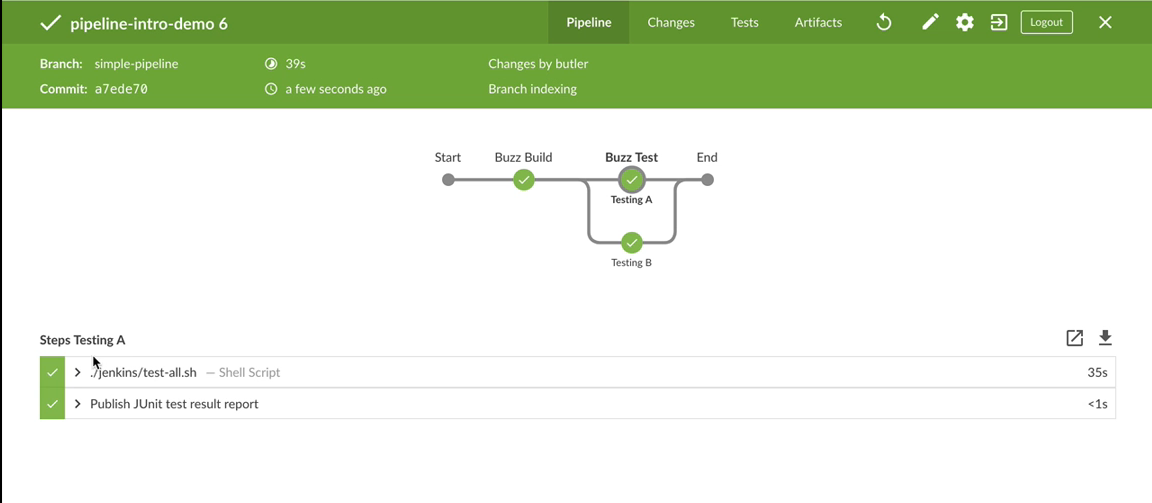
## ADD TESTING A AND TESTING B TO PIPELINE

* Add a parallel stage; name it "Testing B"
* Note that, when you add this parallel stage, Blue Ocean adds a first stage  
  that is named "Buzz Test"; rename that stage to be "Testing A"
* Add a step to "Testing B"; we’re not going to do anything very interesting  
  at this point; just choose "Shell Script" and type in
* sleep 10

echo done.

* Save & run the Pipeline; type "Added Parallel stage" as the description.

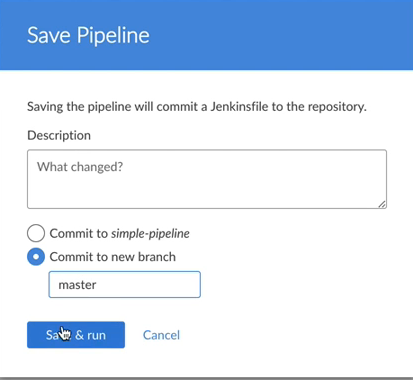
## PIPELINE RUN DETAILS FOR PARALLEL



## HOW PARALLEL STAGES ARE SCHEDULED

* By default, Jenkins tries to allocate a stage to the last node on which it  
  (or a similar stage) executed.
  + This may mean that multiple parallel steps execute on the same node  
    while other nodes in the cluster are idle
* Use the [Least Load Plugin](https://wiki.jenkins.io/display/JENKINS/Least+Load+Plugin) to replace the default load balancer with one  
  that schedules a new stage to nodes with the least load

## SAVE PIPELINE TO MASTER



## INSTRUCTIONS: SAVE PIPELINE TO MASTER

* Click the "Save" button
* No description is required
* Click on "Commit to new branch" and type "master" in the box
* Click "Save and Run"

## DECLARATIVE WITHOUT BLUE OCEAN

* You can create and/or modify a Declarative Pipeline using a text editor rather than Blue Ocean
* Manually configure this Pipeline
  + Must be a Multibranch pipeline
  + Define the SCM, credentials, hooks, etc on the Jenkins Dashboard
  + Configure "githooks" or their equivalent on the SCM itself

## CHECK DECLARATIVE PIPELINE INTO SCM

* After creating/modifying a Declarative Pipeline outside Blue Ocean:
  + Push a PR for the new/modified Jenkinsfile
  + Jenkins runs the Jenkinsfile and reports any errors
  + PRs that modify the Jenkinsfile should always be reviewed by  
    someone other than the author before being merged
    - This protects against a Jenkinsfile that calls a malicious script

## LAB EXERCISE

[Create and Edit a Simple Pipeline with Parallel Stages](https://s3.amazonaws.com/cloudbees-training-materials/training-pipeline-fundamentals/master/selfpaced/labs/Create_and_Edit_a_Simple_Pipeline_with_Parallel_Stages.html)

**Solution**

pipeline { agent any stages { stage('Fluffy Build') { steps { sh './jenkins/build.sh' archiveArtifacts 'target/\*.jar' } } stage('Fluffy Test') { parallel { stage('Backend') { steps { sh './jenkins/test-backend.sh' junit 'target/surefire-reports/\*\*/TEST\*.xml' } } stage('Frontend') { steps { sh './jenkins/test-frontend.sh' junit 'target/test-results/\*\*/TEST\*.xml' } } stage('Performance') { steps { sh './jenkins/test-performance.sh' } } stage('Static') { steps { sh './jenkins/test-static.sh' } } } } stage('Fluffy Deploy') { steps { sh './jenkins/deploy.sh staging' } } } }

## SCRIPTED PIPELINE

## WHAT IS SCRIPTED?

* Older, "lower-level" Pipeline syntax
* Similar to Declarative
  + Uses the same Pipeline engine
  + Defined in a Jenkinsfile
* Differs from Declarative
  + More flexible, allows more complexity
  + Can be more difficult to learn and use

## WHY NOT USE SCRIPTED PIPELINE?

* Novice users easily make mistakes
* Risks putting too much code in Pipeline
  + Put build, test, and deploy in your scripts
  + Call your scripts from Pipeline
* Call your build scripts from Pipeline
  + npm, gulp, make, maven, ant, scons, etc.

## SAMPLE SCRIPTED PIPELINE

* A small Jenkinsfile for a Scripted Pipeline looks like:
* stage('Build') {
* parallel linux: {
* node('linux') {
* checkout scm
* try {
* sh 'make'
* }
* finally {
* junit '\*\*/target/\*.xml'
* }
* }
* },
* windows: {
* node('windows') {
* /\* .. snip .. \*/
* }
* }

}

## NOTES ABOUT SCRIPTED SYNTAX

* No pipeline block that includes the entire pipeline
* Uses node where Declarative uses agent
* Scripted Pipeline code must explicitly handle the checkout scm step
* Many steps are similar to what is used in Declarative

## HOW TO CREATE A SCRIPTED PIPELINE

* Inline — Use the Classic Web UI; type text into the **Definition** box
  + Easier method, appropriate for quick prototyping of any type of Pipeline
* Text editor — Type the Pipeline code into the text editor of your choice
  + Normal practice for production Pipelines

## HINTS ABOUT EDITING A JENKINSFILE

* Uses a custom DSL based on Apache Groovy syntax
* You need some expertise in Groovy to create a Pipeline using Scripted syntax
* The **Snippet Generator** is a pipeline syntax helper that simplifies the process

## TRIVIAL SCRIPTED PIPELINE

* To create a trivial Scripted Pipeline
  + Open the Classic Web UI
  + Click "New Item"
  + Type a name for your Pipeline and select "Pipeline"; click "OK"
  + Type the Pipeline code into the "Definition" box
    - Use the Snippet Generator for help with pipeline syntax
  + Click the "Save" button
    - This applies the configuration change and redirects you to the job’s main page
  + Launch a build of this pipeline manually

## ENTER THE CLASSIC WEB UI

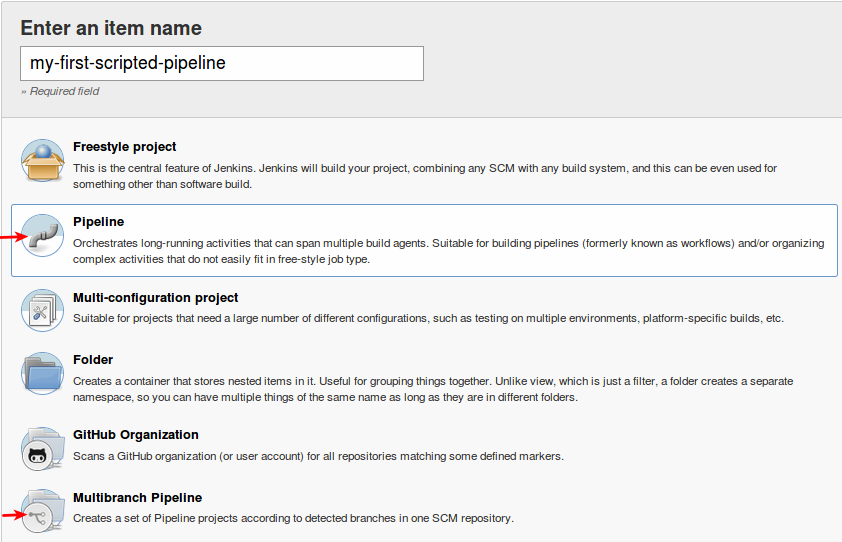
* To escape to the Classic Web UI from Blue Ocean, click the arrow in a box icon on the Blue Ocean header



* To return to Blue Ocean, use the "Open Blue Ocean" link in the left column of the Classic Web UI

## START A NEW SCRIPTED PIPELINE

* Click "New Item" in the left frame to open the following screen:



* Choose "Pipeline"
  + Note the "Multibranch Pipeline" option at the bottom of the list

## DEFINE A NEW SCRIPTED PIPELINE

* Type the name of your new Pipeline in the Enter an item name box
  + "my-first-scripted-pipeline" in this example
* Select "Pipeline" from the list of available job types
* Click the "OK" button

## USING THE INLINE PIPELINE EDITOR

* You can compose your Jenkinsfile directly in the GUI or in a text file you edit with a text editor
* Click the arrow next "try sample pipeline" to see some sample code
* Reset that field to "try sample pipeline" and type your pipeline code in the "Script" box under "Definition"
* Click "Pipeline Syntax" at the bottom of the screen to open the "Snippet Generator"



## SNIPPET GENERATOR

* Generates valid code for Scripted Pipeline steps, based on the desired task
  + You can also use the Snippet Generator for Declarative Pipelines
* This code can be pasted into the Jenkinsfile
* Includes Pipeline steps provided by Jenkins and installed plugins
* Has more options and, in some cases, more depth than Blue Ocean
* Does not have a search capability like Blue Ocean has

## USING THE SNIPPET GENERATOR: OVERVIEW

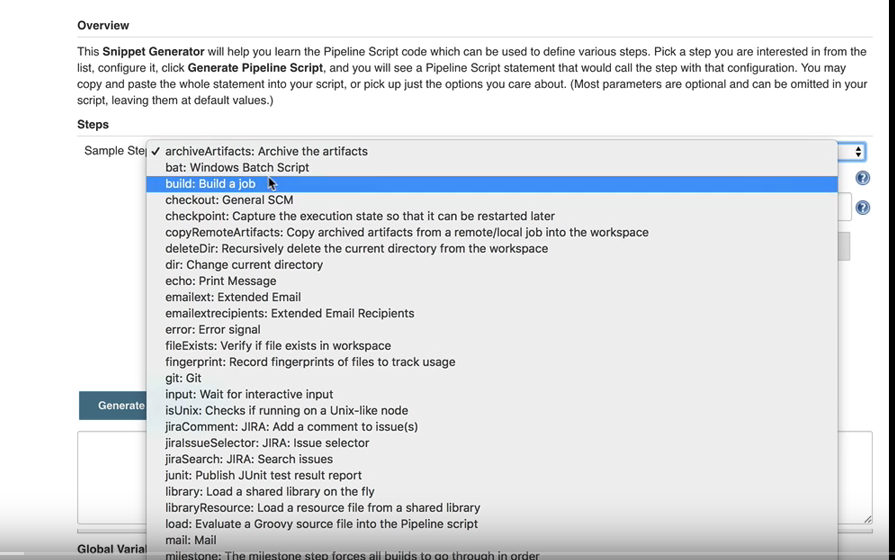
This is an overview of the Snippet Generator usage:

* Select a step you want to execute
* Customize its arguments
* Generate the snippet
* Copy and paste the snippet into your Pipeline

## USING THE SNIPPET GENERATOR: SELECT A STEP

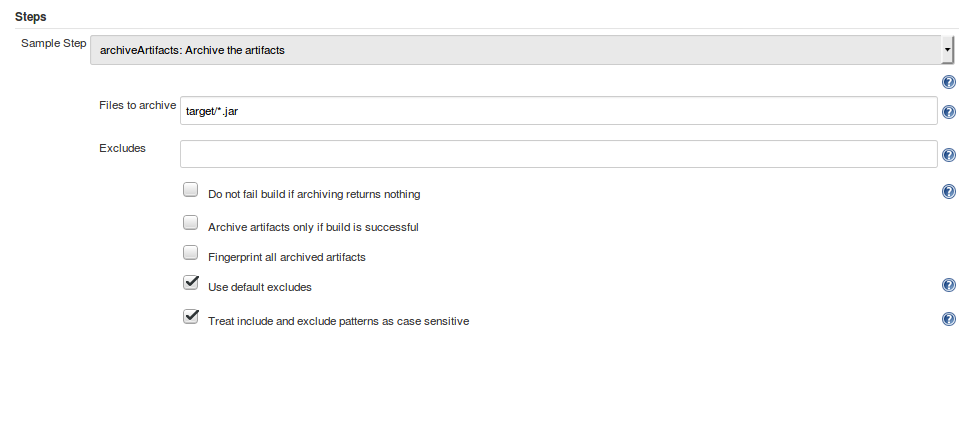
* Click on the "Sample Step" box to display a drop-down menu of steps you might want to execute.
* Scroll through the items displayed to select the task you want

## SNIPPET GENERATOR



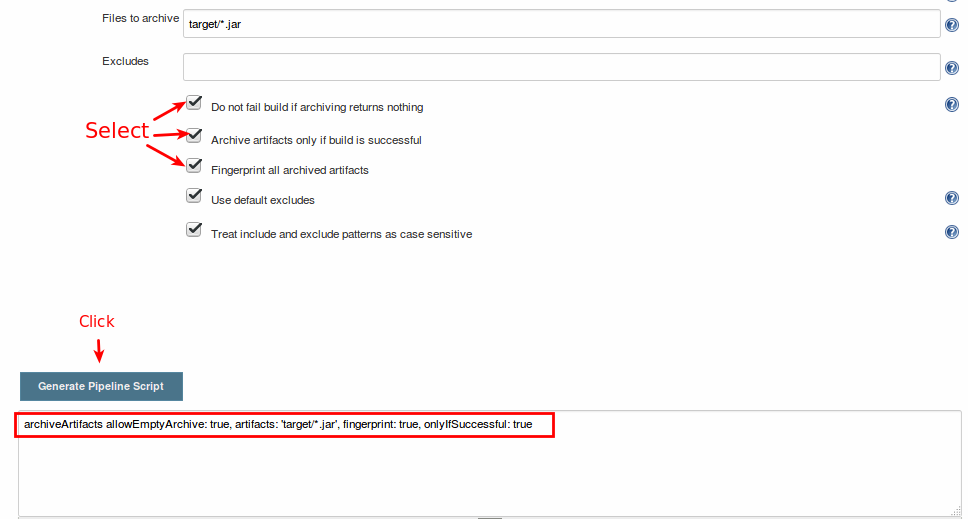
## USING THE SNIPPET GENERATOR: CUSTOMIZE THE STEP

* When appropriate, the Snippet Generator provides forms and/or checklists where you can  
  provide arguments for fields used for the selected step.
* For example, if you choose the "archiveArtifacts" step and then click the "Advanced" button  
  in the display, you get:



## USING THE SNIPPET GENERATOR: GENERATE THE SNIPPET

* Click the three empty boxes so that all boxes are selected
* Click the "Generate Pipeline Script" button; a snippet appears below the button



## USING THE SNIPPET GENERATOR: USE THE SNIPPET

* Copy and paste this snippet into the appropriate part of the Pipeline Script
* You could also
  + Paste this snippet into a file you are editing in a text editor
  + Paste this snippet into your Blue Ocean code editor

## CREATE TRIVIAL SCRIPTED PIPELINE

* Our trivial Scripted Pipeline needs the following steps:
  + Checkout code from SCM
    - Declarative Syntax automatically provides this step but you must  
      implement it explicitly for Scripted Pipelines
  + Steps to echo the main tasks to be performed:
    - Build code
    - Run unit tests on code
    - Run regression tests
    - Deploy code

## STEP: BUILD CODE

* Choose the "echo: Print Message" item
* Type the message you want into the box that is provided:



* Click the "Generate Pipeline Script" box
* Copy the generated code to your Pipeline (Jenkinsfile)

## FINISH THE TRIVIAL SCRIPTED PIPELINE

* Follow the same process to create steps to test and deploy the software
* When you have completed all the steps, save your Jenkinsfile.
* Commit your new code and create a PR that contains it
* Jenkins runs your new Scripted Pipeline and displays the St

## SCRIPTED PIPELINES IN PRODUCTION

* This exercise is a short introduction to Scripted Pipelines
* Scripted Pipelines used in production have the following characteristics:
  + Jenkinsfile must be created and modified in a text editor
  + Should be Multibranch Pipeline
    - Requires Git, Bitbucket or SVN
    - Must be configured to the SCM repository, with credentials, hooks, etc
  + Must be committed to the SCM like any other code

## WHAT IS MULTIBRANCH?

* Recommended for all new Pipelines
* All Pipelines created with Blue Ocean are multibranch
* Requires Git, Bitbucket or SVN
* Configures a set of branches
  + Jenkins creates a subproject for each branch in the SCM repository

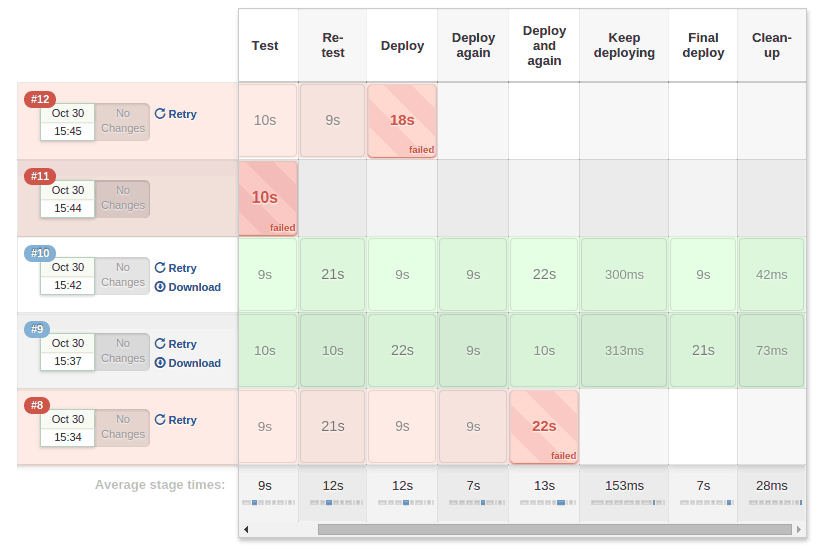
## WHY MULTIBRANCH

* Automatic Workflow creation for each new branch in the repository
  + Assumes that webhooks are registered from the SCM to Jenkins
* Builds are specific to that child/branch and its unique SCM change and build history
* Automatic job pruning/deletion for branches deleted from the repository, according to the settings
* Ability to override the parent properties for a specific branch, if necessary

## PIPELINE STAGE VIEW

## PIPELINE STAGE VIEW

* GUI on the **Job Page**
* Provides Pipeline visualization relying on **stages**
* Shows a matrix with build history and stages as dimensions



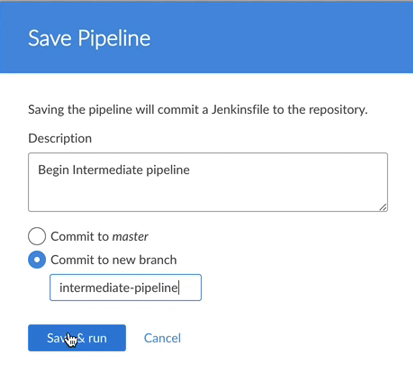
## ABOUT PIPELINE STAGE VIEW

* An alternative to the Blue Ocean visual display
* Shows the Pipeline build history
* You can also click on "Full Stage View" in the left frame to get a full-screen view
* Displays pipeline data, just as Blue Ocean does:
  + Date, time, changes (with changes links) per build
  + Execution time per build and per stage
  + Status and Output logs per stage

## SET UP NEW BRANCH

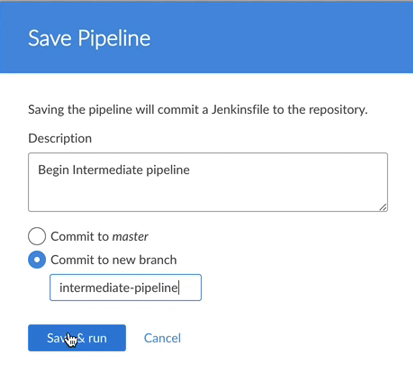
## START A NEW BRANCH

* Save your Pipeline to master
* Create a new branch for the exercises in this section



## START A NEW BRANCH

* Save your Pipeline to master
* Create a new branch for the exercises in this section

****

## REFERENCE DOCUMENTATION

* [Pipeline Syntax](https://jenkins.io/doc/book/pipeline/syntax/)
  + Describes the basic syntax of the Declarative Pipeline, including Sections and Directives.
  + Includes sections about flow control, differences from plain Groovy and a syntax comparison  
    with Scripted syntax
* [Pipeline Basic Steps](https://jenkins.io/doc/pipeline/steps/workflow-basic-steps/)
  + Describes the steps that are available to all Jenkins Pipelines
* [Pipeline Steps Reference](https://jenkins.io/doc/pipeline/steps/)
  + Links to descriptions of steps provided by plugins

## AGENTS

## WHAT IS AN AGENT?

* In a Jenkins cluster, a **node** is a machine ready to take any build workload.
* The agent step specifies the "node" on which a stage executes.
* The global agent section must be defined at the top-level, inside the pipeline block
* The **agent** section can also be defined for each stage
  + When building and testing for different platforms, use agent to specify the appropriate environment for each step.

## PARAMETERS

* The agent section takes a parameter to specify the type of node to assign. In Blue Ocean, you type the appropriate parameter into the box.
  + agent any (default) — Stage can execute on any node
  + agent none — Do not define an agent
    - Use for global agent to allow you to specify particular nodes for each stage
    - Use for a stage that must execute without an agent context and its associated workspace
  + Do not use agent none for steps that execute a shell command or do other activities that require an agent context and the associated workspace; the results can be unpredictable
  + agent { label } — Use an agent identified by the specified label
  + agent { node } — Similar to agent { label } but allows additional options

## DOCKER AGENT

* Docker is a tool that can package an application and its dependencies  
  in a virtual container that can run on any Linux server
* A Docker container is:
  + Self sufficient
  + Runs as an isolated process
  + Easier to deploy
  + Secure
* Docker containers can be used as the agent to provide a build environment
  + A docker agent runs on a node with Docker installed
  + It is a recommended method for running agents

## SPECIFY DOCKER AGENT IN PIPELINE

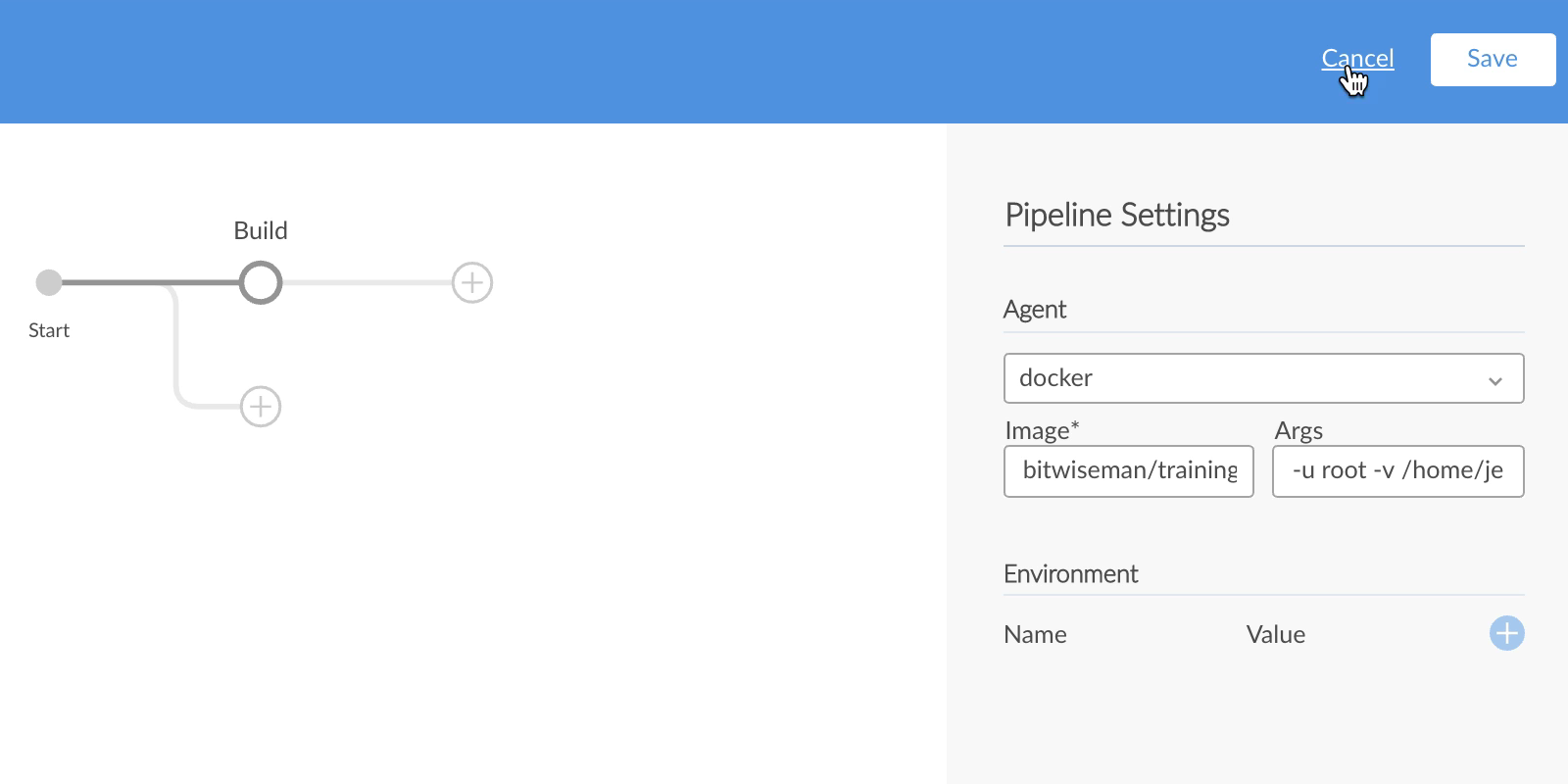
* agent { docker }
  + Execute the Pipeline or stage with the specified container
  + The container will be dynamically provisioned on the node
* agent { dockerfile }
  + Execute the Pipeline or stage with a container built  
    from the specified Dockerfile

## DOCKER CONTAINER DURABILITY

* When you specify a docker container for an agent, Jenkins calls APIs directly
  + These commands are serialized so they can resume after a master restart
* When you use a shell step to run a Docker command directly, the step is bound  
  to the durable task of the shell
  + The Docker container and any tasks running in it are terminated when the shell terminates

## CONFIGURING AGENTS IN BLUE OCEAN

Blue Ocean allows you to specify parameters for an agent and to specify any subsequential arguments, as shown here:



## SIMPLE SYNTAX

* Run all Pipeline stages on any node machine:
* pipeline {
* agent any
* ....

}

* Run all Pipeline stages on a node machine tagged as bzzzmaven:
* pipeline {
* agent { label 'bzzzmaven' }
* ....

}

* Run all Pipeline stages on a container based on the image bzzzcentos:7:
* pipeline {
* agent { docker 'bzzzcentos:7' }
* ....

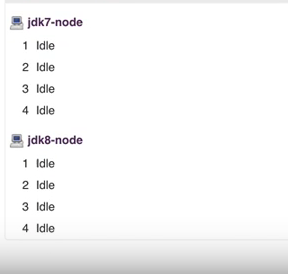
}

## PER-STAGE AGENT SYNTAX

* Do not run stages on an agent by default
* Run the stage Build on a node machine tagged as bzzzmaven
* Run the stage Deploy on a node machine tagged as bzzzproduction:
* pipeline {
* agent none
* stages {
* stage ('Build') {
* agent { label 'bzzzmaven' }
* steps { ... }
* }
* stage ('Deploy') {
* agent { label 'bzzzproduction' }
* steps { ... }
* }
* }

}

## SPECIFY AGENTS FOR OUR PIPELINE

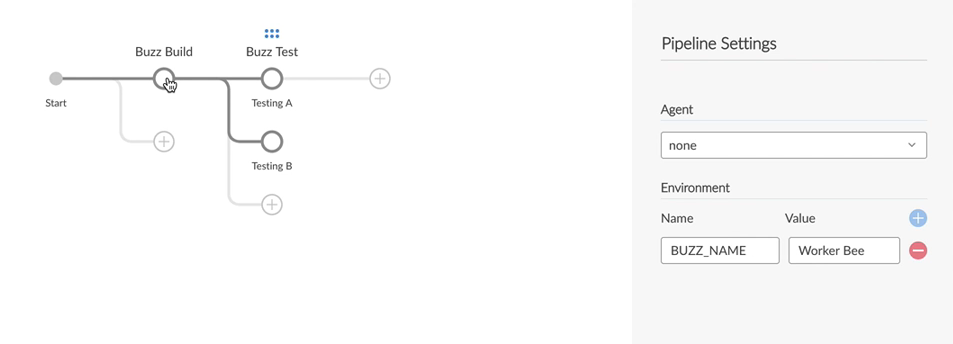
* We are building our application to work with both JDK7 and JDK8. This means that  
  we need a JDK7 environment and a JDK8 environment. To implement this:
  + Change the global agent to be agent none; in other words, there is no global agent
  + Specify the appropriate agent (labeled either jdk7 or jdk8) for the build and test steps  
    that need the specific JDK environment.
  + Specify agent any for stages that do not require a specific JDK version.
* **WHAT NODES ARE AVAILABLE**
* Available nodes are listed at the bottom of the left frame of the Classic Web UI:
* 

## MORE DETAILS ABOUT NODES

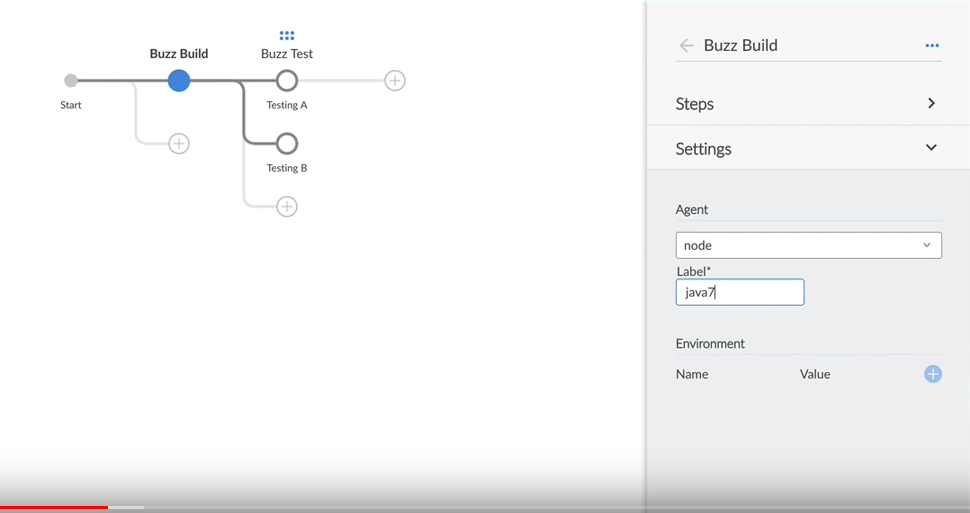
Click on one of the nodes to see details about the node:



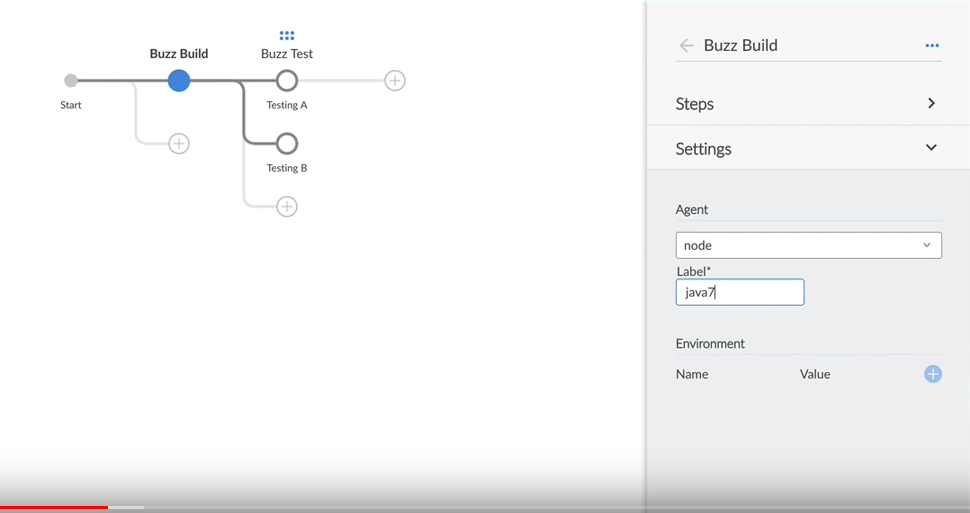
## RESET GLOBAL AGENT INSTRUCTIONS



## RESET GLOBAL AGENT INSTRUCTIONS

* Click on the grey dot above "Start" at the left hand size of the pipeline visualization
* Under "Pipeline Settings", change the Agent value to none
* **SPECIFY JAVA7 FOR EACH STAGE**
* 

## SPECIFY JAVA7 FOR EACH STAGE

****

## SPECIFY JAVA7 FOR EACH STAGE — INSTRUCTIONS

* Select the "Buzz Build" stage
* In "Settings", choose "node" from the drop-down menu
* Specify the java7 node
* Now do the same for the "Testing A" and "Testing B" stages

## ADD JDK8 BUILD AND TEST

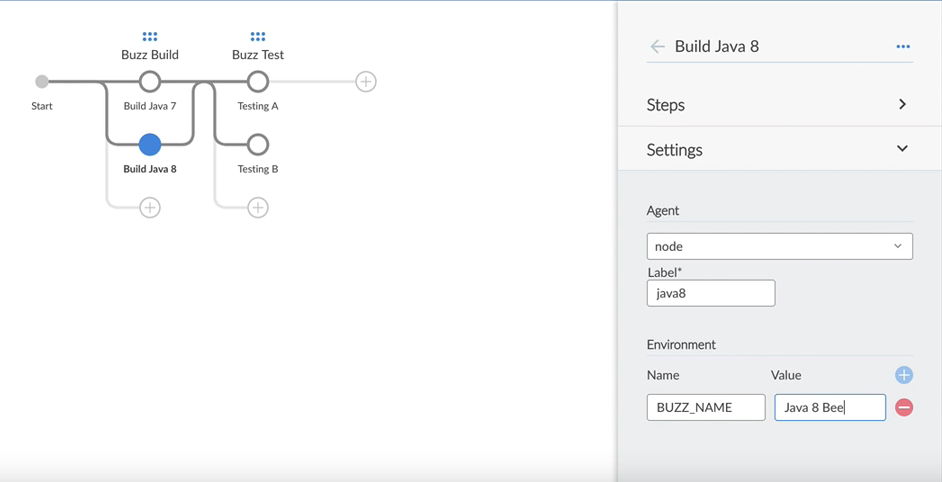


## ADD JDK8 BUILD AND TEST INSTRUCTIONS

* Add the "Build Java 8" stage to execute in parallel to the existing stage under "Buzz Build"
* Rename the existing stage to be "Build Java 7"
* Select the "Build Java 8" stage, open "Settings", and set it to run on node java8
* Open the "Build Java 7" stage, open the first step, and copy the contents.
* Open the "Build Java 8" stage, create a "Shell Script" step, and paste the contents  
  from the Java 7 step into that file
* Repeat that process to copy the contents of the second step in "Build Java 7"  
  to a step in "Build Java 8"

## SET STAGE-SPECIFIC ENVIRONMENT VARIABLE

Override the global value of the "BUZZ\_NAME" environment variable just for the "Build Java 8" stage:



## STASH/UNSTASH

* Use stash to save a set of files for use later in the same build, but in another stage that executes on another node/workspace
* Use unstash to restore the stashed files into the current workspace of the other stage
* Stashed files are discarded at the end of the build
* Stashed files are archived in a compressed tar file, which works well for small (<5 Mb) files
  + Stashing large files consumes significant resources (especially CPU) on both the Master and agent nodes
  + For large files, consider using:
    - the [External Workspace manager](https://jenkins.io/doc/pipeline/steps/external-workspace-manager/) plugin
    - an external repository manager such as Nexus or Artifactory
    - the [Artifact Manager on S3](https://plugins.jenkins.io/artifact-manager-s3) plugin ([README](https://github.com/jenkinsci/artifact-manager-s3-plugin/blob/master/README.md))

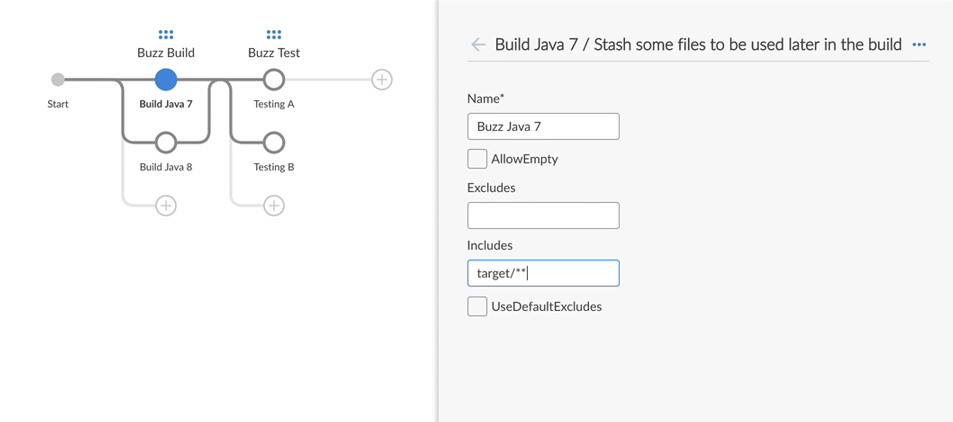
## CALLING SYNTAX

* stash and unstash are implemented as steps within a stage
* stash requires one parameter, name, which is a simple identifier for the set of files being stashed
* By default, stash places all workspace files into the named stash
* To store files from a different directory (or a subset of the workspace):
  + Use the optional includes parameter to give the name of the files or directories to store. This accepts a set of Ant-style include patterns.
  + Other optional parameters are documented on the [Pipeline: Basic Steps](https://jenkins.io/doc/pipeline/steps/workflow-basic-steps/#stash-stash-some-files-to-be-used-later-in-the-build) reference page
* To unstash files:
  + Optionally, use the dir step to create a directory where the files will be written.
  + Call unstash, using the name you assigned with the stash step above.

## IMPLEMENT STASH/UNSTASH (1/2)

* We are going to make our Pipeline build and test our software for both JDK 7 and JDK 8.
  + We will use stash/unstash to ensure that the JDK 7 tests are run against the JDK 7 build and that the JDK 8 tests are run against the JDK 8 build:
* Add a step to the "Build Java 7" stage: "Stash some files to be used later in the build"
  + Name the stash "Buzz Java 7"; enter target/\*\* in the "Includes" box to save everything under the target directory
* Follow the same procedure to stash "Buzz Java 8" in the "Build Java 8" stage

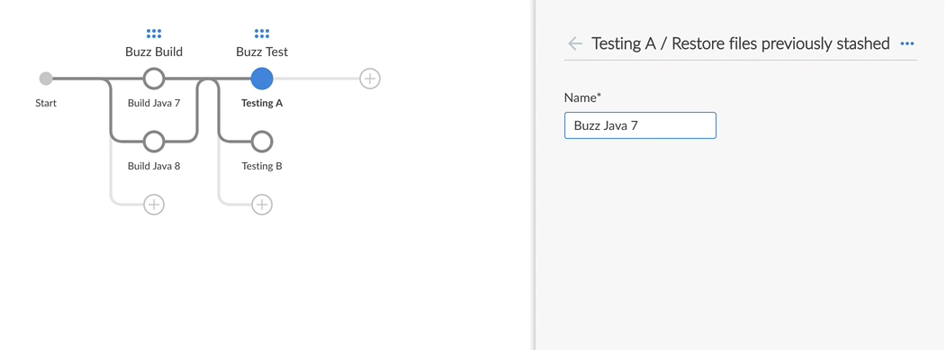
## STASH SCREEN



## IMPLEMENT STASH/UNSTASH (2/2)

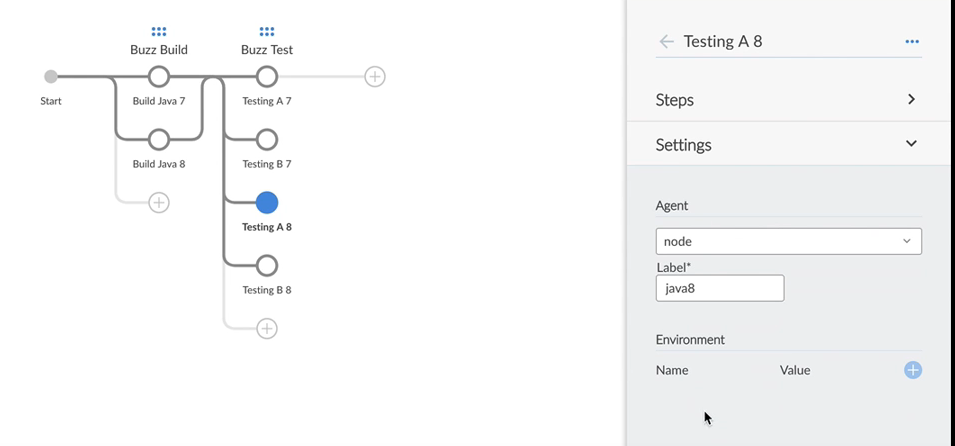
* Add a step to the "Testing A" stage: "Restore files previously stashed"
  + Use the "Buzz Java 7" name from the stash we previously created
* Follow the same procedure to unstash "Buzz Java 8" in the "Testing B" stage

## UNSTASH SCREEN



## EDIT JENKINSFILE

* Use ctrl-S to open the Jenkinsfile in the edit window
* Move the unstash step so it runs before the test shell script in both the "Testing A" and "Testing B" stages
* Rename the testing stages to be "Testing A 7" and "Testing B 7"
* Copy and paste the code for the "Testing A 7" and "Testing B 7" stages
* Edit the copy to create "Testing A 8" and "Testing B 8" stages
* Edit the agent labels and the stash/unstash names to use "8" rather than "7"
* **RESULTANT PIPELINE**



## WAIT FOR INTERACTIVE INPUT

## WHAT IS INTERACTIVE INPUT? (1/2)

* Jenkins provides the ability to pause the pipeline to wait for input from a human.
  + For example, after the build and tests run successfully, you may want a human  
    to confirm that the software should be deployed before executing the stage  
    that actually does the deployment

## WHAT IS INTERACTIVE INPUT? (2/2)

* Implemented as a step, usually within its own stage
* The input step should run on agent none
  + It must execute without an *agent* context and its associated workspace
  + It should not block an executor
* Add the parameter for additional notes and echo the value
* Use timeout to avoid waiting for an infinite amount of time
* In most cases, you want to define specific people who can approve moving forward

## ADD WAIT FOR INPUT TO PIPELINE

* For our Pipeline, we will create a stage to deploy the software to a staging server  
  after all tests have passed
* Before we deploy, we want a human to approve the deployment
* "Wait for input" cannot be implemented in the same stage as the deployment because:
  + "Wait for input" must run with agent none
  + The deployment stage requires an *agent* context

## IMPLEMENT WAIT FOR INPUT

* Create a new stage; name it "Confirm Deploy to Staging"
* Add a step of type "Wait for interactive input"
* Fill out the form that is provided:

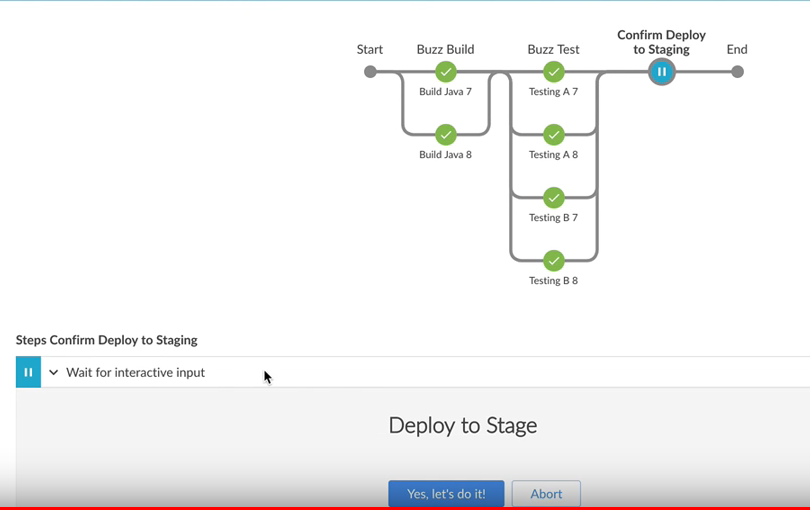


## INSTRUCTIONS: IMPLEMENT WAIT FOR INPUT

* To fill out the "Wait for interactive input" form that is provided:
  + Type the text of the question to be printed in the "Message" box
    - For example: Do you want to deploy to staging?
  + Type the text to be in the button used to respond in the "OK" box
    - For example, Yes, let’s do it!
  + Use the "Submitter" box to define who can authorize this operation.
* Click "Settings" at the bottom of the right frame
  + Set agent none to be the agent for this step

## WHAT HAPPENS WHEN IT RUNS?

* Note the "Deploy to Stage" bar at the bottom of the screen, along with the text you supplied when you created that button



## CODE: WAIT FOR INPUT

* The code in the Jenkinsfile looks like:
* stage('Confirm Deploy to Staging') {
* agent none
* steps {
* input(message: 'Deploy to Stage', ok: "Yes, let's do it!")
* }

}

* Note that Blue Ocean uses single quotes to enclose the text in the "Message"  
  and "OK" boxes and automatically escapes apostrophes in the text
* To make the code cleaner, you can use the code editor to use double quotes  
  to enclose the text

## DEPLOYING FROM JENKINS PIPELINE

## DEPLOYMENT

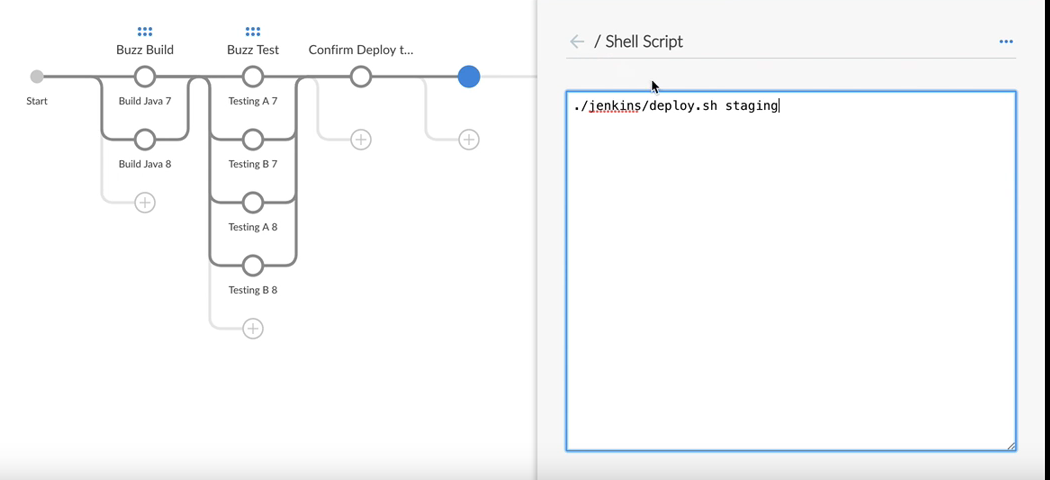
* Don’t reinvent your deployment system in Jenkins Pipeline
  + Think of Pipeline as glue
* "Deployment" may have different meanings
  + Deploy to AWS/Azure
  + Deploy to a physical datacenter
  + Upload to an app store
  + Upload to an internal artifact server

## ADD DEPLOY STAGE INSTRUCTIONS

* Create a new stage named "Deploy to Staging"
  + This cannot be combined in the stage used to "Wait for Input" because we need an agent context to deploy the software but "Wait for Input" must run with agent none
* Configure this stage to run on a java8 node
* Unstash the built software (Buzz Java 8) as the first step in this stage
* Use a "Shell script" step to call the deploy script

## CALL DEPLOY SCRIPT

* Set the "Shell Script" field to ./jenkins/deploy.sh staging



## LAB EXERCISE

[Section 4: Multi-environment Pipeline](https://s3.amazonaws.com/cloudbees-training-materials/training-pipeline-fundamentals/master/selfpaced/labs/Section_4_Multi_environment_Pipeline.html)

* pipeline {
* agent none
* stages {
* stage('Fluffy Build') {
* parallel {
* stage('Build Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* sh './jenkins/build.sh'
* stash(name: 'Java 8', includes: 'target/\*\*')
* }
* }
* stage('Build Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* sh './jenkins/build.sh'
* archiveArtifacts 'target/\*.jar'
* stash(name: 'Java 7', includes: 'target/\*\*')
* }
* }
* }
* }
* stage('Fluffy Test') {
* parallel {
* stage('Backend Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-backend.sh'
* junit 'target/surefire-reports/\*\*/TEST\*.xml'
* }
* }
* stage('Frontend') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-frontend.sh'
* junit 'target/test-results/\*\*/TEST\*.xml'
* }
* }
* stage('Performance Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-performance.sh'
* }
* }
* stage('Static Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-static.sh'
* }
* }
* stage('Backend Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-backend.sh'
* junit 'target/surefire-reports/\*\*/TEST\*.xml'
* }
* }
* stage('Frontend Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-frontend.sh'
* junit 'target/test-results/\*\*/TEST\*.xml'
* }
* }
* stage('Performance Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-performance.sh'
* }
* }
* stage('Static Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-static.sh'
* }
* }
* }
* }
* stage('Confirm Deploy') {
* steps {
* input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!')
* }
* }
* stage('Fluffy Deploy') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/deploy.sh staging'
* }
* }
* }
* }

## POST SECTION

**WHAT IS A POST SECTION?**

* post section contains steps to be executed at the end of a Pipeline run or stage
* post section is divided into conditions such as always, success, failure
* Each condition lists steps to be executed
  + See [Pipeline Syntax](https://jenkins.io/doc/book/pipeline/syntax/) for a full description of supported conditions
* An error in a post section does not make the Pipeline run unsuccessful
* Part of Declarative Pipeline syntax but not yet integrated into Blue Ocean Editor

**HOW TO IMPLEMENT POST BLOCK**

* Use ctrl-S/cmd-S to open the Blue Ocean code editor
* Type in the post section block code
* Particularly useful for archiving artifacts and storing test results
* Note that post section blocks are not displayed in the Blue Ocean Editor  
  but they are in the Jenkinsfile

**OUR SAMPLE**

* Modify the "Build Java 7" stage
* Create post sections with two conditions:
  + always archive the artifacts
  + stash the build only on success
* In the real world, should do this for all the Build and Test stages

## SAMPLE POST CODE

pipeline {

stages {

stage('Buzz Build') {

parallel {

stage('Build Java 7') {

steps {

sh """

echo I am a $BUZZ\_NAME!

./jenkins/build.sh

"""

}

post {

always {

archiveArtifacts(artifacts: 'target/\*.jar', fingerprint: true)

}

success {

stash(name: 'Buzz Java 7', includes: 'target/\*\*')

}

}

}

...

}

## ENVIRONMENT DIRECTIVE

## WHAT IS AN ENVIRONMENT DIRECTIVE?

* Specifies a sequence of key-value pairs that are defined as environment variables
  + Can be specified globally for the Pipeline, to apply to all steps
  + Can be specified for an individual stage to apply only to that stage
  + By manually coding withEnv inside a steps block, a specific environment variable  
    can be specified for one or more (but not all) steps within a stage

## WHERE ARE ENVIRONMENT VARIABLES DEFINED?

* Jenkins defines a set of environment variables
  + Examples include BUILD\_NUMBER, JENKINS\_URL and EXECUTOR\_NUMBER.
  + These are listed under the Web UI
  + Follow the the Manage Jenkins → System information link
  + All Jenkens environment variables are listed on the [Building a Software Project](https://wiki.jenkins.io/display/JENKINS/Building+a+software+project) page
* The Blue Ocean Editor can be used to easily define environment variables in most cases.
* Many plugins define environment variables that are useful when using that plugin.

## SINGLE AND DOUBLE QUOTES

* The Pipeline DSL uses [Apache Groovy syntax](http://groovy-lang.org/syntax.html). Variables are dereferenced according to  
  whether they are enclosed in single quotes or double quotes:
  + Single quotes: The dereferencing syntax (${VARIABLE\_NAME}) is passed literally  
    to the Pipeline sh step; the shell interpreter on the agent dereferences the variable.
  + Double quotes: The Pipeline code dereferences the variable on the master’s JVM thread  
    then passes the calculated string to the sh step.

## ENVIRONMENT EXAMPLE

pipeline {

agent any

environment {

CC = 'clang'

}

stages {

stage('Example') {

environment {

AN\_ACCESS\_KEY = credentials('my-predefined-secret-text')

}

steps {

sh 'printenv'

* The first use of environment applies to the entire Pipeline
* The second use of environment applies only to the Example stage

## NOTIFICATIONS

## JENKINS AND NOTIFICATIONS

* Jenkins can send notifications to alert you when a build starts and whether it succeeded or failed. For example:
  + Email
  + Slack
* Jenkins can also update JIRA tickets after a build
* Other notification channels are available through plugins
* More on this subject on the Jenkins.io Blog: [Jenkins.io Blog Articles related to Notifications](https://jenkins.io/node/tags/notifications/)

## ABOUT OUR EXAMPLE

* Send notifications when build starts, when build succeeds, when build fails
* Notifications sent to Slack channel and Email recipients
* Our sample notifications use Jenkins environment variables to define the  
  job name, build number and URL of the build
* You can use other environment variables to add additional information.

## PREREQUISITES

* Must install and configure the appropriate plugin for each channel
  + [Slack Plugin](https://plugins.jenkins.io/slack)
  + [Email-ext Plugin](https://plugins.jenkins.io/email-ext)

## NOTIFICATIONS WHEN BUILD STARTS (1/2)

stages {

stage ('Start') {

steps {

// send build started notifications

slackSend (color: '#FFFF00', message: "STARTED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})")

}

}

}

## NOTIFICATIONS WHEN BUILD STARTS (2/2)

/\* ... unchanged ... \*/

// send to email

emailext (

subject: "STARTED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>STARTED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

/\* ... unchanged ... \*/

## NOTIFICATIONS WHEN BUILD COMPLETES

* Create a post section in the Pipeline for notifications to be sent after the Pipeline run is done
* Create a success condition to define notifications to be sent when the Pipeline completed successfully
* Create a failure condition to define notifications to be sent when the Pipeline fails

## NOTIFICATIONS WHEN BUILD SUCCEEDS

post {

success {

slackSend (color: '#00FF00', message: "SUCCESSFUL: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})")

emailext (

subject: "SUCCESSFUL: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>SUCCESSFUL: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

}

}

## NOTIFICATIONS WHEN BUILD FAILS

failure {

slackSend (color: '#FF0000', message: "FAILED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})")

emailext (

subject: "FAILED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>FAILED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

}

## CONFIGURE NOTIFICATIONS

* Your Jenkins administrator must configure notifications for Jenkins in order for the notifications to be delivered.
* Slack uses an API token
  + Integration points for each generate the token
  + That generated token must be copied into the Jenkins configuration
* The Slack configuration provides a button that can be used to test the configured connection

## WHEN DIRECTIVE

## WHAT IS THE "WHEN" DIRECTIVE?

* Specifies conditions that must be met for Pipeline to execute the stage
* Must contain at least one condition
* Supports nested conditions

## BUILT-IN CONDITIONS

* branch — Execute the stage when the branch being built matches the branch pattern given:
* when {
* branch 'master'

}

* environment — Execute the stage when the specified environment variable is set to the given value:
* when {
* environment name: 'DEPLOY\_TO', value: 'production'

}

* expression — Execute the stage when the specified expression evaluates to true:
* when {
* expression {
* return params.DEBUG\_BUILD
* }

}

## BUILT-IN NESTED CONDITIONS

* allOf — Execute the stage when all nested conditions are true:
* when {
* allOf {
* branch 'master'
* environment name: 'DEPLOY\_TO', value: 'production' // AND
* }

}

* anyOf — Execute the stage when at least one of the nested conditions is true
* when {
* anyOf {
* branch 'master'
* branch 'staging' // OR
* }

}

* not — Execute the stage when the nested condition is false.

when { not { branch 'master' } }

## MULTIPLE CONDITIONS

* If the when directive contains more than one condition, all  
  specified conditions must return true for the stage to execute.
* This is identical to using the allOf condition

## IMPLEMENTING "WHEN" DIRECTIVE

* The "Confirm Deploy to Staging" and "Deploy to Staging" steps are only appropriate  
  when building from the master branch.
* Use Ctrl+S (Cmd+S on macOS) to open the Blue Ocean code editor and add  
  when directives that specify the branch condition to these two stages.
* The Blue Ocean Editor does **not** show the when directive but it is in the Jenkinsfile.
* When we save and run the Pipeline, these steps are skipped because we are  
  not working on the master branch.

## "WHEN" DIRECTIVE EXAMPLE

...

stage('Confirm Deploy to Staging') {

when {

branch 'master'

}

steps {

input(message: 'Deploy to Stage', ok: 'Yes, let\'s do it!')

}

}

stage('Deploy to Staging') {

agent {

node {

label 'java8'

}

}

when {

branch 'master'

}

steps {

unstash 'Buzz Java 8'

sh './jenkins/deploy.sh staging'

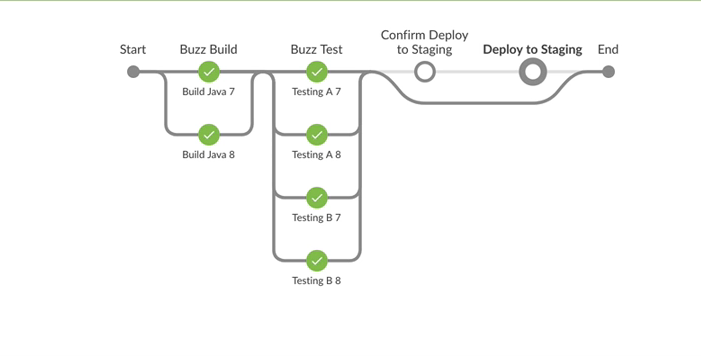
}

}

...

**VIEW PIPELINE RUN DETAILS**

The visualization shows that we skipped these two stages because we are not working in the master branch:



## GIT ENVIRONMENT VARIABLES

## ABOUT GIT VARIABLES

* Git environment variables
  + Are defined in the Git plugin
  + Are documented in the [Environment Variables](https://wiki.jenkins.io/display/JENKINS/Git+Plugin#GitPlugin-Environmentvariables) section of the Git plugin docs
  + Are set automatically at the beginning of every Pipeline run if you are using Git as your SCM
* GIT\_COMMIT, GIT\_BRANCH and GIT\_URL are examples of Git environment variables
* These variables can then be used within the Pipeline
* We will look at two examples to illustrate how this is done

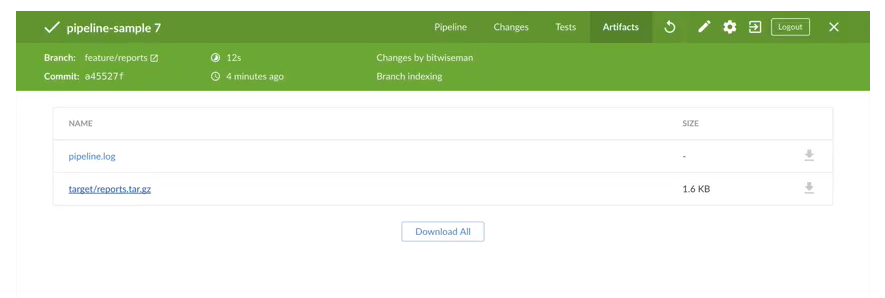
## EXAMPLE USE OF GIT VARIABLES

* The first example uses the GIT\_COMMIT environment variable in a shell step  
  to give a unique name to a file generated by the Pipeline
* Let’s look at a Pipeline snippet that creates and archives a file of reports.
* The flow is:
  + Generate some files
  + Add those files to a \*.tar.gz file
  + Archive that file for later review
* Pipeline code looks like:
* stage('Generate Reports') {
* steps {
* sh './jenkins/generate-reports.sh'
* sh 'tar -czv target/reports.tar.gz target/reports'
* archiveArtifacts 'target/\*.tar.gz'
* echo "Finished run for commit ${ env.GIT\_COMMIT.substring(0,6) }"
* }

}

## VIEWING THE ARTIFACT

* Blue Ocean shows us this file after the Pipeline runs:



* The tar file shows up in the build’s details

## GIT COMMIT NUMBER

* Blue Ocean displays the associated Git Commit number for each run on the left side of the  
  Details Header part of the navigation bar
* Jenkins lets us access that Git Commit number using the GIT\_COMMIT environment variable

## USING GIT COMMIT NUMBER

* If you want to download and compare the contents of this archive from several runs,  
  you need to add some unique identifier to the name of each file.
* Adding the Git Commit number identifies this particular file and associates it  
  with the commit version
* Use bash shell conventions (0:6) to use just the first six characters of  
  the commit hash in the file name rather than the full (rather long) commit hash
* stage('Generate Reports') {
* steps {
* sh './jenkins/generate-reports.sh'
* sh 'tar -czf target/reports-${GIT\_COMMIT:0:6}.tar.gz target/reports'
* archiveArtifacts 'target/\*.tar.gz'
* echo "Finished run for commit ${ env.GIT\_COMMIT.substring(0,6) }"
* }

}

## RESULTS USING GIT\_COMMIT

* When the Pipeline runs, you can see the tar command, using the first six digits of the commit hash as part of the file name.
* This hash value matches the "Commit" value that is displayed in the left end of the Details Header



## USING GIT VARIABLES IN A NON-SHELL STEP

* This code sends an email each time the Pipeline run finishes
* We use interpolated strings to add Git variables.
* The subject line uses a Jenkins environment variable to give the build number, plus the Git branch and the first six characters of the Git commit hash, read from the Pipeline’s env object
* Note that double quotes are used to surround the value of the subject: parameter

post {

always {

mail(

to: 'bitwiseman@bitwiseman.io',

subject: "Finished run for build ${ env.BUILD\_ID }, commit ${ env.GIT\_BRANCH } (${ env.GIT\_COMMIT.substring(0,6) })",

body: 'Placeholder' )

}

}

## OTHER GIT\_\* VARIABLES

* To see the full list of GIT variables available to a Pipeline, use an sh step in a Pipeline  
  with code like this:
* steps {
* sh '''env | grep GIT\_
* ./jenkins/build.sh
* '''

}

* The most frequently used GIT\_\* variables, with example values, are:
* GIT\_PREVIOUS\_SUCCESSFUL\_COMMIT=b47e27bdc4022e19736b1101d8e9c615889b38f2
* GIT\_COMMIT=bb7216c58cfff23d94e3917fb004137324955edd
* GIT\_BRANCH=feature/test-cond
* GIT\_URL=https://github.com/bitwise-jenkins/pipeline-sample.git

GIT\_PREVIOUS\_COMMIT=b47e27bdc4022e19736b1101d8e9c615889b38f2

* Other GIT\_\* variables include GIT\_AUTHOR\_EMAIL, GIT\_AUTHOR\_NAME, GIT\_COMMITTER\_EMAIL and GIT\_COMMITTER\_NAME, although some of these  
  have known bugs.

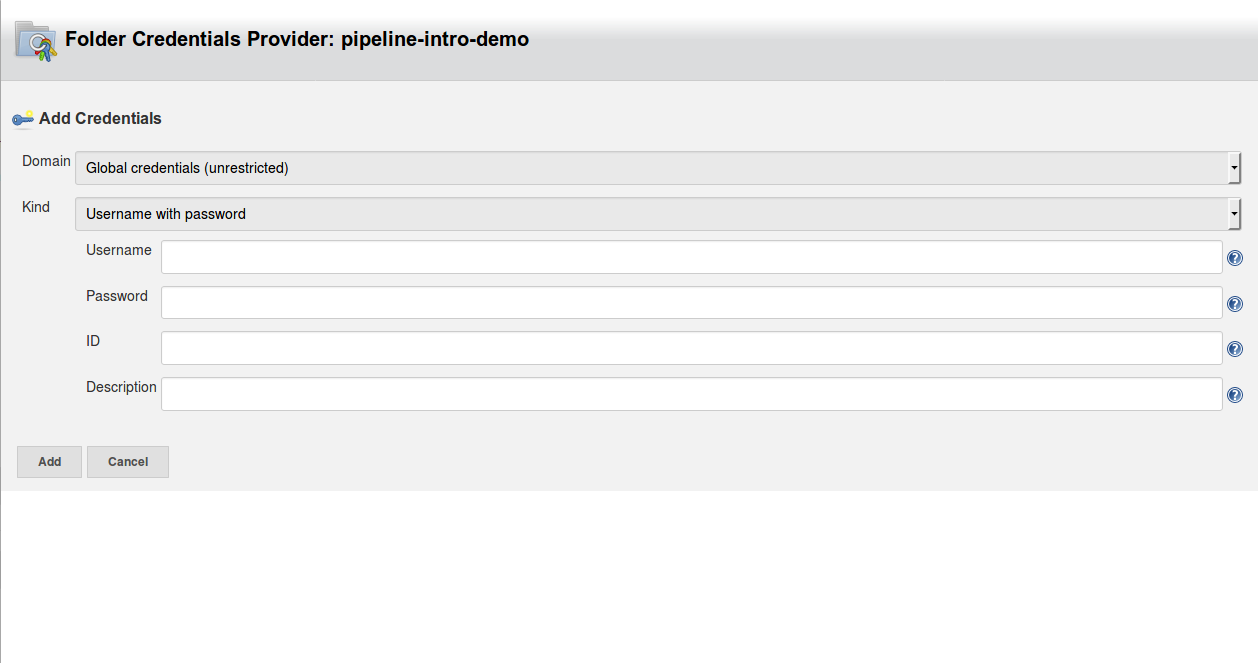
## CREDENTIALS

## ABOUT CREDENTIALS

* Are used to get trusted access to other resources without having to share the actual passwords, et cetera, with all users
* Implemented with plugins that are on the "Recommended Plugins" list when you install Jenkins
* Defined under "Configure" on the Classic Web UI
* Are stored in an obfuscated form on the Jenkins Master
* Are tied to and encrypted by the instance ID so can not be migrated to a new instance
* The canonical documentation about credentials is [Credentials API Plugin documentation](https://github.com/jenkinsci/credentials-plugin/tree/master/docs), written and maintained by Stephen Connolly, the author of the Credentials plugin.

## ADD CREDENTIALS

* To create a credential, fill in the following form under "Configure" on the Web UI



## NOTES ABOUT ADDING CREDENTIALS

* Assign an ID that can be used to access the credential. The Pipeline uses this ID to apply these credentials
  + If you do not assign an ID, Jenkins assigns an ID that is a GUID
  + IDs should be 40 characters or less
  + Only use alpha-numeric characters plus the separator characters - , \_ .
  + IDs are case sensitive
* Scope defines the places where this credential can be used
  + System: Credential is available only to the object with which the credential is associated. Examples include email authorization or node connection, where the Jenkins instance itself is using the credential
  + Global: Credential is available to the object with which the credential is associated plus all objects that are children of that object. Most Credentials that are called in Pipelines are defined as Global.

## CREDENTIALS TYPE

* The Jenkins Credentials Type defines the behavior. It must match what the target object expects:
  + Secret Text type:
    - MYVARNAME contains the path of the file that contains the Secret Text
  + Standard username and password type:
    - MYVARNAME is set to <username>:<password> (conjoined name and password string)
    - MYVARNAME\_USR is set to just the username
    - MYVARNAME\_PSW is set to just the password

## MORE ABOUT CREDENTIAL TYPES

* Each credential type supported for the environment directive:
  + Must be supported in the credentials binding plugin
  + Must have a special handler for Declarative Pipeline
* Standard Declarative syntax currently supports the following credentials:
  + Username and Password (plugin type StandardUsernamePasswordCredentials)
  + Secret file (FileCredentials)
  + Secret text (StringCredentials)
* Use the sshagent step for SSH credentials, which withCredentials does not support

## PLUGINS USE OF CREDENTIALS

* Plugins may define other credential types
* A plugin that supports credentials may not support all credential types.
  + For example, the Git Plugin (and Git Client Plugin) can only use username/password  
    credentials and private key credentials
  + Attempts to use secret text, certificate, or other credential types fail for the Git plugin

## EXAMPLE USE OF CREDENTIALS

* Pipeline needs to deploy results to an external resource such as Nexus or Elasticsearch.
* Access to that external resource is protected by a username and password
* Sharing that username/password with 20 or 100 users is undesirable
  + It is a security risk to have multiple people share a password
  + If the password is changed, all those people must be informed, which is a hassle
* Instead, define a credential that knows the username and password
* The Pipeline uses that credential to supply the username and password, but people  
  who run the Pipeline do not know what the password is

## EXAMPLE CODE

* Credentials are implemented in a Pipeline using the withCredentials() method
* Use the Blue Ocean code editor to add the withCredentials() method; the Blue Ocean graphical editor does not yet support it
* ...
* stage('Deploy Reports')
* steps {
* ...
* withCredentials(bindings: [string(credentialsId: 'my-elastic-key', variable: 'ELASTIC\_ACCESS\_KEY')]) {
* // Environment Variable available in the remote shell
* sh "env | grep ELASTIC\_ACCESS\_KEY"
* sh "echo ${ELASTIC\_ACCESS\_KEY} > secret-file.txt"
* }
* ...

}

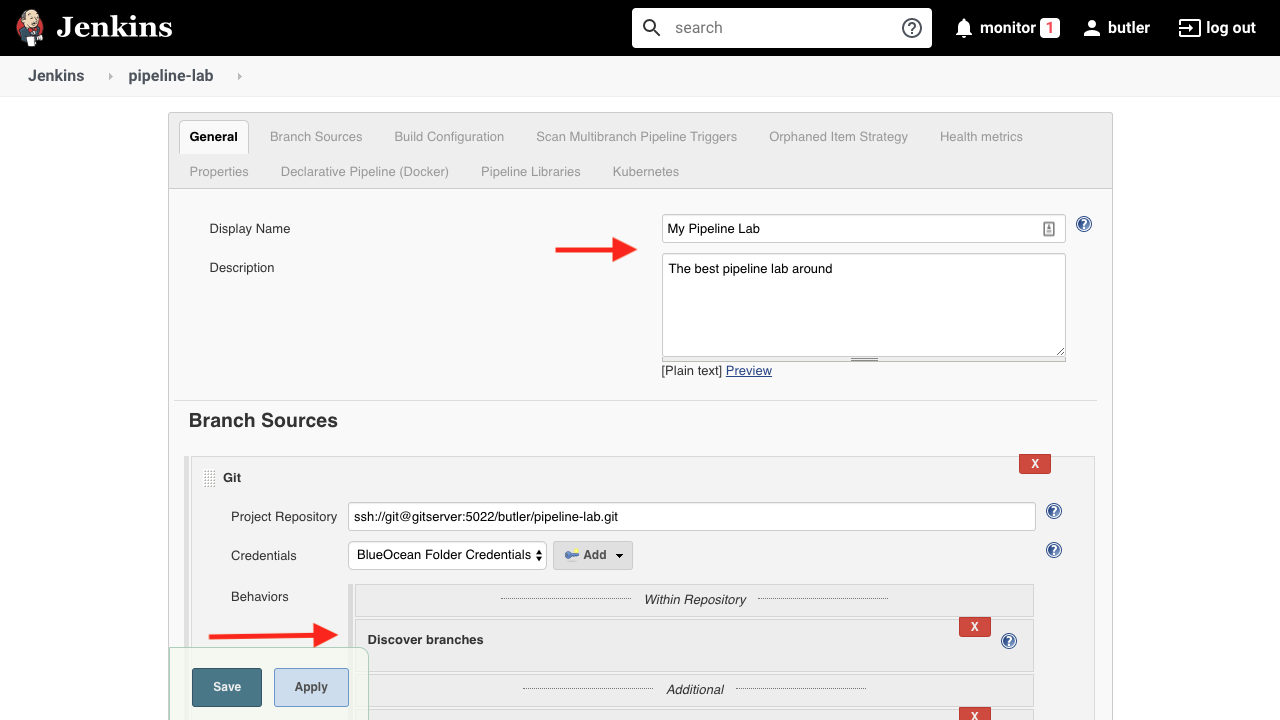
* Uses the credential that is defined with the ID of my-elastic-key
* withCredentials binds that to a local Pipeline variable called ELASTIC\_ACCESS\_KEY
* The steps uses the ELASTIC\_ACCESS\_KEY local Pipeline variable to establish  
  the credentials that it needs

## OPTIONS AND CONFIGURATIONS

## WHAT ARE OPTIONS AND CONFIGURATIONS?

* Methods for controlling the characteristics and behaviors of the Pipeline
* Options are set inside the Jenkinsfile for a Declarative Pipeline
* Configurations are set from the Classic Web UI on the Jenkins Master

## CONFIGURATIONS

* Mostly for administration use
* Click on "Configure" in the left frame of the Classic Web UI
* Frequently used configurations:
  + Display name for this Pipeline
  + Behaviors for check out
* **CONFIGURATION EXAMPLE**
* 

## OPTIONS DIRECTIVE

* Documented on the Pipeline Syntax page
  + Look under Declarative Directive Generator
* Use the Blue Ocean code editor to add the options directive

## LET’S GO THROUGH AN OPTIONS EXAMPLE

## SETTING TIMEOUTS IN A PIPELINE

* Timeouts allow for healthy cleanup of the Pipeline
* The entire Pipeline should have a timeout
* All inputs should be wrapped in a timeout

## EXAMPLE: SET TIMEOUT

* timeout for the entire Pipeline (Note the double quotes around **DAYS**
* options {
* timeout(time: 3, unit: "DAYS")

}

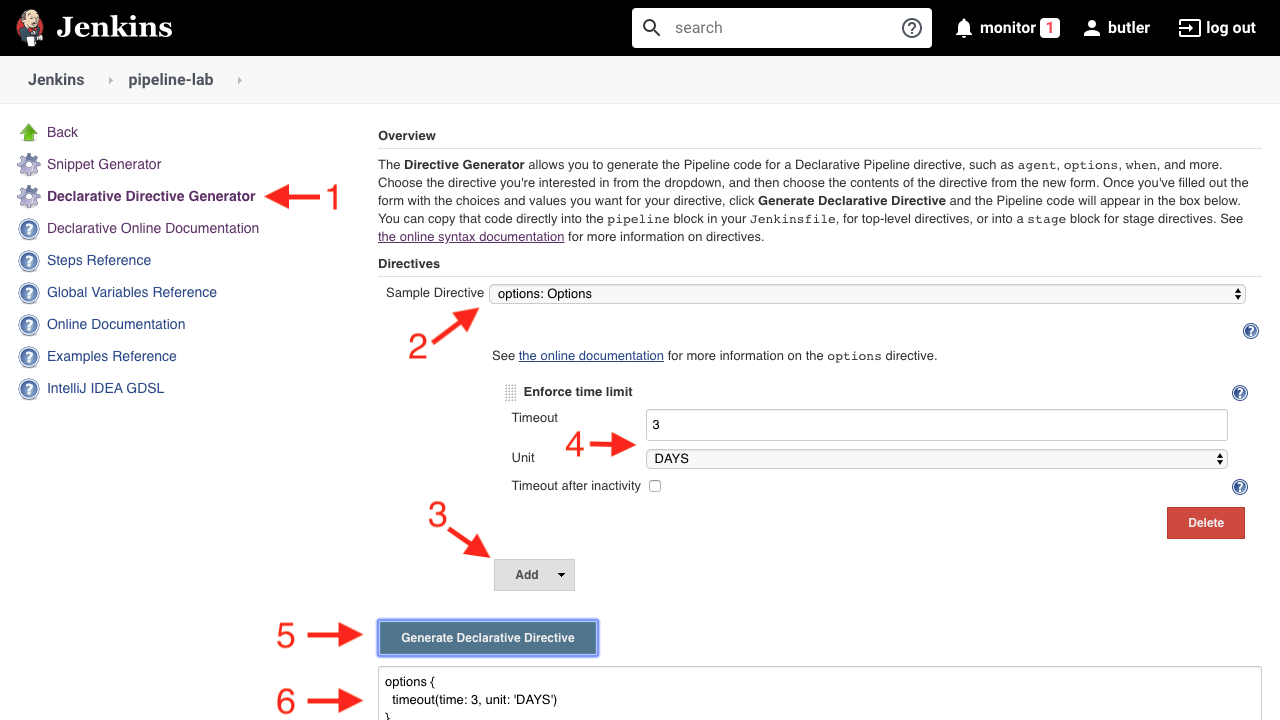
* timeout for an "input" stage
* steps {
* timeout(time:3, unit:"DAYS") {
* input(message: "Deploy to Stage", ok: "Yes, let's do it!")
* }

}

**BUT WAIT, I DON’T KNOW THE SYNTAX TO SET THE OPTIONS FOR THE ENTIRE PIPELINE**

**DECLARATIVE DIRECTIVE GENERATOR CAN HELP**

**USING THE DECLARATIVE DIRECTIVE GENERATOR**

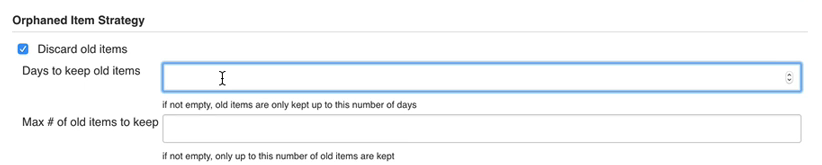


## LET’S LOOK AT ANOTHER EXAMPLE

## HOW LONG TO KEEP "OLD ITEMS"

### CONFIGURATION

* Under "Orphaned Item Strategy"



## HOW LONG TO KEEP "OLD ITEMS"

### DECLARATIVE SYNTAX

* Use the buildDiscarder option to define the number of items to keep:
* options {
* buildDiscarder(logRotator(numToKeepStr: '1'))

}

## SO, WHAT DOES ALL THIS MEAN?

## WHEN TO USE AN OPTION AND WHEN TO USE A CONFIGURATION

* If the property can be managed inside of a Jenkinsfile, put it in the Jenkinsfile
* If the property cannot be managed inside of a Jenkinsfile,  
  manually configure the property in the job configuration

## PARAMETERS

## WHAT ARE PARAMETERS

* parameters directive provides a list of parameters a user should provide when triggering the Pipeline
* The user-specified values are made available to the Pipeline through the params object.
* Can be set using the Blue Ocean code editor
* Blue Ocean can not access parameters on the first run after they are coded; they are recognized on all subsequent runs

## EXAMPLE OF PARAMETERS

pipeline {

agent none

parameters {

string(name: 'DEPLOY\_ENV', defaultValue: 'staging', description: '')

}

stages {

stage ('Deploy') {

echo "Deploying to ${DEPLOY\_ENV}"

}

}

}

## BUILD TOOLS TO CREATE CODE FOR STEPS

## SUPPORTED BUILD TOOLS

* Make
* Apache Ant
* Apache Maven
* Gradle
* NPM

## MAKE

* The original UNIX build script
* It still works for very simple projects
* Other options are better for large, complex projects
* See the [make man page](https://linux.die.net/man/1/make)

## APACHE ANT

* Uses an XML file to describe the project
* Includes built-in tasks
* Allows you to define custom tasks
* Uses Apache Ivy for dependency management

## USING APACHE ANT IN PIPELINE

* Requires the [Ant Plugin](https://wiki.jenkins.io/display/JENKINS/Ant+Plugin)
* On Linux and macOS, call as a "Shell Script" (sh) step.
  + For example, to compile, build and run the project:

sh 'ant clean compile jar run'

* + On Windows, call as a bat step:

bat 'ant clean compile jar run'

* See the [Apache Ant Manual](http://ant.apache.org/manual/index.html)
* Apache AntUses an XML file to describe the projectIncludes built-in tasksAllows you to define custom tasksUses Apache Ivy for dependency management

## APACHE MAVEN

* Most popular project build tool for Java
* Can also be used for Scala, Ruby, C# and other languages
* Uses an XML file to describe the project
* Includes pre-defined targets for common tasks such as compiling and packaging
* Downloads libraries and plugins for the appropriate repositories and puts them  
  in cache on your local machine
* The [Maven Project Plugin](https://wiki.jenkins.io/display/JENKINS/Maven+Project+Plugin) provides an advanced integration for Maven 2/3 projects; it is not required but is recommended.

1.4

## USING APACHE MAVEN IN PIPELINE

* On Linux and macOS, call as a "Shell Script" (sh) step:

sh 'mvn clean install'

* On Windows, call as a bat step:

bat 'mvn clean install'

* See the [Apache Maven Documentation](http://maven.apache.org/guides/index.html)

## GRADLE

* Expands on Apache Maven and Apache Ant
* Uses domain-specific language (DSL) for project configuration
* Used for all builds in Android projects
* Supports very large, multi-project builds
* Allows for incrementally adding to the build
* Claims to be faster than Maven for the same build

## USING GRADLE IN PIPELINE

* Requires the [Gradle Plugin](https://wiki.jenkins.io/display/JENKINS/Gradle+Plugin)
* On Linux and macOS, call as a "Shell Script" (sh) step:

sh './gradlew build'

* On Windows, call as a bat step:

bat './gradlew build'

* [Gradle Documentation](https://gradle.org/docs/)

## NPM

* Package manager for JavaScript
* Requires the [NodeJS Plugin](https://wiki.jenkins.io/display/JENKINS/NodeJS+Plugin)
* Default package manager for the Node.js runtime environment
* Can install packages, manage dependencies and manage versions of packages
* Consists of a command line client and a registry of packages
* Is required to run the Grunt build tool for JavaScript and the Bower package manager for the web

## USING NPM IN PIPELINE

* On Linux and macOS, call as a "Shell Script" (sh) step:

sh 'npm build'

or

sh 'npm test'

or

sh 'npm run build'

* On Windows, call as a bat step:

bat 'npm build'

or

bat 'npm test'

or

bat 'npm run build'

* See the [NPM documentation](https://docs.npmjs.com/)

## ADDING TOOLS TO PIPELINE

* The tools section defines the tools to auto-install and put on the PATH.
* It is ignored if agent none is specified.
* It can be specified
  + In the pipeline block to apply to the entire pipeline
  + In a stage block to apply to just that stage
* The following tools are supported:
  + maven
  + jdk
  + gradle
  + nodejs

## EXAMPLE TOOLS SECTION

* Each tool specified in the tools section takes an argument that defines the version to use:
* pipeline {
* agent any
* tools {
* maven 'Maven 3.3.9'
* jdk 'jdk8'
* }
* ...

}

* The values used to define each tool must first be configured on the WebUI under  
  "Managing Jenkins" → "Global Tool Configuration"
* A common practice is to define the Maven version in the tools section in the pipeline  
  block, then define the JDK version in tools sections located in individual stage sections.
  + This is how to create a Pipeline that builds and tests the code for multiple JDK versions

## CHECKING THE TOOLS

The output of the following code shows the values of the PATH and M2\_HOME  
environment variables, which show which version of the tools is being used:

* stages {
* stage ('Initialize') {
* steps {
* sh '''
* echo "PATH = ${PATH}"
* echo "M2\_HOME = ${M2\_HOME}"
* '''
* }
* }

}

* The version configured in the tools section will be first on the path.

## CALLING THE MAVEN BUILD

* Now we can call the Maven build code:
* stage ('Build') {
* steps {
* sh 'mvn -Dmaven.test.failure.ignore=true install'
* }
* post {
* always {
* junit 'target/surefire-reports/\*\*/\*.xml'
* }
* }

}

* install is one of the phases defined for the "default" [Maven Build Lifecycle](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html)

## MORE ABOUT THE MAVEN BUILD

* When you run Maven with a phase (provided as argument), Maven executes, in order,  
  each life cycle phase up to the one specified.
  + The call to install runs validate, compile, test, package, verify  
    and finally install.
* This Pipeline always archives the JUnit test reports for display in the Jenkins Web UI  
  because most people want the test results whether the mvn call succeeds or not.
  + If you only want to archive the reports when the Maven build succeeds,  
    use success rather than always here.

## RUNNING A MAVEN BUILD

* Here is the full pseudo-code for what we have discussed on the previous slides:
* pipeline {
* agent any
* tools {
* maven 'Maven 3.3.9'
* jdk 'jdk8'
* }
* stages {
* stage ('Initialize') {
* steps {
* sh '''
* echo "PATH = ${PATH}"
* echo "M2\_HOME = ${M2\_HOME}"
* '''
* }
* }
* stage ('Build') {
* steps {
* sh 'mvn -Dmaven.test.failure.ignore=true install'
* }
* post {
* always {
* junit 'target/surefire-reports/\*\*/\*.xml'
* }
* }
* }

}

## CONSOLE OUTPUT

* The console output of running this Pipeline is:
* Console Output
* { ... truncated lines ... }
* + mvn install
* [INFO] Scanning for projects...
* [WARNING] The POM for org.jenkins-ci.tools:maven-hpi-plugin:jar:1.119 is missing, no dependency information available
* [WARNING] Failed to build parent project for org.jenkins-ci.plugins:junit:hpi:1.20-SNAPSHOT
* [INFO]
* [INFO] ------------------------------------------------------------------------
* [INFO] Building JUnit Plugin 1.20-SNAPSHOT
* [INFO] ------------------------------------------------------------------------
* [INFO]
* [INFO] --- maven-hpi-plugin:1.119:validate (default-validate) @ junit ---
* [INFO]
* [INFO] --- maven-enforcer-plugin:1.3.1:display-info (display-info) @ junit ---
* [INFO] Maven Version: 3.3.9
* [INFO] JDK Version: 1.8.0\_92 normalized as: 1.8.0-92
* [INFO] OS Info: Arch: x86\_64 Family: mac Name: mac os x Version: 10.12.3
* [INFO]
* { ... }
* [INFO] ------------------------------------------------------------------------
* [INFO] BUILD SUCCESS
* [INFO] ------------------------------------------------------------------------
* [INFO] Total time: 03:25 min
* [INFO] Finished at: 2017-02-06T22:43:41-08:00
* [INFO] Final Memory: 84M/1265M

[INFO] ------------------------------------------------------------------------

## USING CATEGORIES

* The previous Pipeline runs all the build and tests functionality in one large block
* Use Categories to split the build and tests into separate stages
  + Categories are assigned inside the test source code
  + This avoids splitting tests into multiple files
  + Maven can then run different categories of tests at different times in the project build, test and deliver cycle
  + Use options to the mvn test steps in Pipeline to control which subset of tests should be included in or excluded from the run
* For more information about using Categories:
  + [How to run JUnit tests in Maven?](https://stackoverflow.com/questions/3100924/how-to-run-junit-tests-by-category-in-maven)
  + [How to run Categorized Junit tests using Gradle?](https://stackoverflow.com/questions/40588381/how-to-run-categorized-junit-tests-using-gradle?noredirect=1&lq=1)

Solution:

pipeline {

agent none

stages {

stage('Fluffy Build') {

parallel {

stage('Build Java 8') {

agent {

node {

label 'java8'

}

}

steps {

sh './jenkins/build.sh'

}

post {

success {

stash(name: 'Java 8', includes: 'target/\*\*')

}

}

}

stage('Build Java 7') {

agent {

node {

label 'java7'

}

}

steps {

sh './jenkins/build.sh'

}

post {

success {

archiveArtifacts 'target/\*.jar'

stash(name: 'Java 7', includes: 'target/\*\*')

}

}

}

}

}

stage('Fluffy Test') {

parallel {

stage('Backend Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-static.sh'

}

}

stage('Backend Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-static.sh'

}

}

}

}

stage('Confirm Deploy') {

when {

branch 'master'

}

steps {

input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!')

}

}

stage('Fluffy Deploy') {

when {

branch 'master'

}

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh "./jenkins/deploy.sh ${params.DEPLOY\_TO}"

}

}

}

parameters {

string(name: 'DEPLOY\_TO', defaultValue: 'dev', description: '')

}

}

## INTRODUCING SHARED LIBRARIES

## USE PIPELINE SHARED LIBRARIES

* Allows you to share and reuse Pipeline code
* Scales your Jenkins Pipeline usage
  + Supports collaboration between a large number of teams working on a large number of projects
* Helps administrators manage code sprawl
  + Write once, propagate everywhere
  + Pipeline as code everywhere
* Use tooling to avoid "silos"
  + Collaborate instead of enforcing

## WHAT IS A PIPELINE SHARED LIBRARY ?

* A separate SCM repo that contains reusable functions which can be called from Pipelines
* Configured once per Jenkins instance
* Cloned at build time
* Loaded and used as code libraries for Jenkins Pipelines
* Modifications made to a shared library function are applied to all Pipelines that call that function

## WHY SHARED LIBRARIES ARE USEFUL

* Most companies build many applications that are built on a similar technology stack  
  or deployed in a standard way (for example, pre-packaged as Docker images)
* The various integration and delivery pipelines may be quite similar
* A Shared Library is a custom DSL for your installation
  + Defines functions that can be called in other pipelines
    - The alternative is to copy-and-paste Pipeline code from one Jenkinsfile to another
  + Calling a function rather than copying a few lines of Pipeline code leads to  
    smaller Jenkinsfiles that are easier to read and maintain

## NOTES ABOUT SHARED LIBRARIES

* Extremely powerful
* Learning curve
  + First step is not easy
  + Requires deeper understanding of Pipeline
* Take time to read the [Extending with Shared Libary](https://jenkins.io/doc/book/pipeline/shared-libraries/) documentation
* The "Pipelines - Intermediate" class concentrates on implementing Shared Libraries

## BEST PRACTICES

## KEEP IT SIMPLE!

* Pipeline code is glue
  + Use the minimum amount of code to connect the Pipeline steps and integrate tools
    - Delegate more to agent and reduce the load on masters
  + This makes the Pipeline code:
    - Easier to maintain
    - More robust against bugs

## USE EXTERNAL SCRIPTS AND TOOLS

* Avoid embedding complex logic in the Pipeline itself
* Instead, use external scripts and tools for complex or CPU-expensive processing
  + Offloads work from the master to external executors
  + Allows for easy scale-out of hardware resources
  + Simplifies testing
    - Components can be tested in isolation without the full on-master execution environment

## WHEN TO USE COMMAND-LINE TOOLS

* Processing data
* Communicating interactively with REST APIs
* Parsing/templating larger XML or JSON files
* Nontrivial integration with external APIs
* Simulations and complex calculations
* Business logic

## COMMAND-LINE CLIENTS FOR APIS

 Many software vendors provide easy command-line clients for their tools in various  
programming languages

* These are often robust, performant and easy to use

 Use shell or batch steps to integrate these tools, which can be written in any language

* For a Java client, use a command like:

sh "java -jar client.jar $endPointUrl $inputData"

 Avoid inputs that might contain shell metacharacters. A construction like the following  
solves this problem:

writeFile file: 'input.json', text: inputData

sh 'java -jar client.jar $endPointUrl input.json'

* For a Python client, use a command like:

sh "python jiraClient.py $issueId $someParam"

## CONSIDER EXISTING PLUGINS

* Jenkins has a wealth of plugins
  + Especially for source control, artifact management, deployment systems and system automation
* Quality of plugins varies widely
  + Look for number of installations and how frequently and recently updates appear in the changelog
  + Poorly-maintained plugins with limited installations may actually be worse than writing a little custom Pipeline code

## REDUCE THE NUMBER OF STEPS IN THE PIPELINE

* Most well-formed Pipelines contain less than 300 steps
* Reducing the number of steps that are called can improve Pipeline and overall Jenkins performance
  + Each call to sh or bat incurs ~ 200ms of overhead
  + Information about each step run is written to disk, which can be slower for Jenkins to process.
* Other advantages to fewer steps:
  + Simplify the test and debug process
  + Simplify the logic of the Pipeline

## HOW TO REDUCE THE NUMBER OF STEPS IN A PIPELINE

* Consolidate several sequential sh or bat steps into a single, external  
  helper script that the Pipeline calls as a single step
  + Version this script and store it in the source code repository
  + This script can be tested independently of the Pipeline itself
* The tradeoff here is that you cannot just read down the Pipeline  
  to quickly see each step that it executes

## SCRIPTED SYNTAX

* When writing a Pipeline using Scripted syntax:
  + Do not write code that runs network calls. For example:

def html = "http://google.com".toURL().text

* + Do not write code that attempts to do disk read/write directly
  + Do not launch Thread operations
    - These may interfere with normal operations
    - May fail when Jenkins restarts

## AVOID COMPLEX GROOVY FEATURES

* Avoid complex or advanced Groovy features
  + Such features can introduce difficult-to-resolve problems  
    that interfere with the Pipeline engine
  + For example, Groovy Traits are not supported
  + Such features also make Pipelines harder to read

## INTERNAL JENKINS APIS

* Do NOT use shared libraries, Declarative script blocks or Scripted Pipeline  
  to invoke internal Jenkins APIs. An example of a bad call is

Jenkins.getInstance().getComputer("someNodeName")

* + This often results in problems that destabilize Jenkins
  + It also breaks isolation between Pipelines
  + Long-term support for such usages is not guaranteed
* The appropriate way to invoke Jenkins internal APIs is to write a new Pipeline step  
  (assuming no existing step does what is required).

## MANAGE LOG DATA

* Some people recommend limiting the amount of data written directly to logs by Pipelines
  + Rather than write several MB of log data (such as from a build tool), write this data  
    to an external file, compress it, and archive it as a build artifact
  + Log data is written to files on the master node so can generate significant network traffic.
    - This can degrade performance for this Pipeline and the full installation
* Others argue that the convenience of viewing the log of each step directly is worth the performance cost
  + The practice described above means that, after the fact, one must unload and  
    uncompress artifacts to understand a failure

Solution:

pipeline {

agent none

stages {

stage('Fluffy Build') {

parallel {

stage('Build Java 8') {

agent {

node {

label 'java8'

}

}

steps {

sh './jenkins/build.sh'

}

post {

success {

stash(name: 'Java 8', includes: 'target/\*\*')

}

}

}

stage('Build Java 7') {

agent {

node {

label 'java7'

}

}

steps {

sh './jenkins/build.sh'

}

post {

success {

archiveArtifacts 'target/\*.jar'

stash(name: 'Java 7', includes: 'target/\*\*')

}

}

}

}

}

stage('Fluffy Test') {

parallel {

stage('Backend Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-static.sh'

}

}

stage('Backend Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-static.sh'

}

}

}

}

stage('Confirm Deploy') {

when {

branch 'master'

}

steps {

input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!')

}

}

stage('Fluffy Deploy') {

when {

branch 'master'

}

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh "./jenkins/deploy.sh ${params.DEPLOY\_TO}"

}

}

}

parameters {

string(name: 'DEPLOY\_TO', defaultValue: 'dev', description: '')

}

}

## JENKINS PIPELINE - ADVANCED FEATURES

* Covered today:
  + Build Triggers
* Going further with the course "Jenkins Pipeline - Intermediate"

## TRIGGERS DIRECTIVE

## WHAT IS THE TRIGGERS DIRECTIVE?

* Defines special conditions when the Pipeline should be re-triggered; in other words, used to schedule specialized runs of a Pipeline
* Do NOT use triggers as a polling mechanism to detect source code changes
  + Use Git/Bitbucket webhooks or other hooks to trigger a Pipeline run each time a source code modification is pushed

## USE CASES FOR TRIGGERS

* Perform periodic tasks such as periodically regenerating metadata files
* Trigger builds remotely using curl and an auth token
* Set a build to launch after another Pipeline completes. Note that the "child" Pipeline sets this rather than having the "parent" Pipeline call other Pipelines.

## SUPPORTED TRIGGERS

* cron — Schedule a run at a specified time
* pollSCM — Define a regular interval at which Jenkins should check for new source changes; run the Pipeline if such changes are found
* upstream — Define conditions when a Pipeline should run because of the results of another Pipeline run. See [Result class](http://javadoc.jenkins-ci.org/hudson/model/Result.html) for details about specifying the threshold used.

## IMPLEMENT A TRIGGER

Triggers are specified at the top level of the Pipeline. For example:

pipeline {

agent any

triggers {

cron('H \*/4 \* \* 1-5')

}

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

}

## Course 2 (Jenkins Pipeline - Intermediate)

## COURSE OVERVIEW

Jenkins Pipeline - Intermediate

## COURSE OBJECTIVES

* After completing this training module, you should understand the following:
  + How to create and modify Pipeline code without using Blue Ocean
  + How to create and configure a Shared Library
  + How to call a Shared Library custom step from a Pipeline
  + How to create and use a Resource File
  + More about creating robust, maintainable Pipelines

## COURSE MODULES

1. Recap of Pipeline Fundamentals
2. Prepare for Shared Libraries
3. Create a Shared Library
4. Call Shared Library Functions
5. Best Practices

## PREREQUISITES

* "[Jenkins - Fundamentals](https://standard.cbu.cloudbees.com/cloudbees-university-jenkins-fundamentals)" course or CJE/CCJE certification
* "[Pipeline - Fundamentals](https://standard.cbu.cloudbees.com/cloudbees-university-jenkins-pipeline-fundamentals)" course
* Students should also have some familiarity with [ancillary technologies](https://standard.cbu.cloudbees.com/ancillary-technologies) that are used in this course:
  + Docker
  + Git
  + Apache Maven, Gradle, Ant or NPM
  + Apache Groovy
* The class has been structured so you can do the exercises even if you are not familiar with these tools but learning them will help you implement your Pipelines when you go back to work.

## APPROACH

* This course teaches you how to create and run a Jenkins Declarative Pipelineusing shared libraries.
  + Students modify a real-life Pipeline to build, test and deploy a Pipeline.
* The course uses one project:
  + Lab project: Students are given a list of tasks and are expected to figure out  
    how to implement a Pipeline that implements those tasks.

## CLASSROOM

* Feel free to ask questions if you do not understand the material
* Please avoid questions that are not directly related to the material
* One 15-minute break in the morning session
* One hour for lunch
* Two 15-minute breaks in the afternoon session

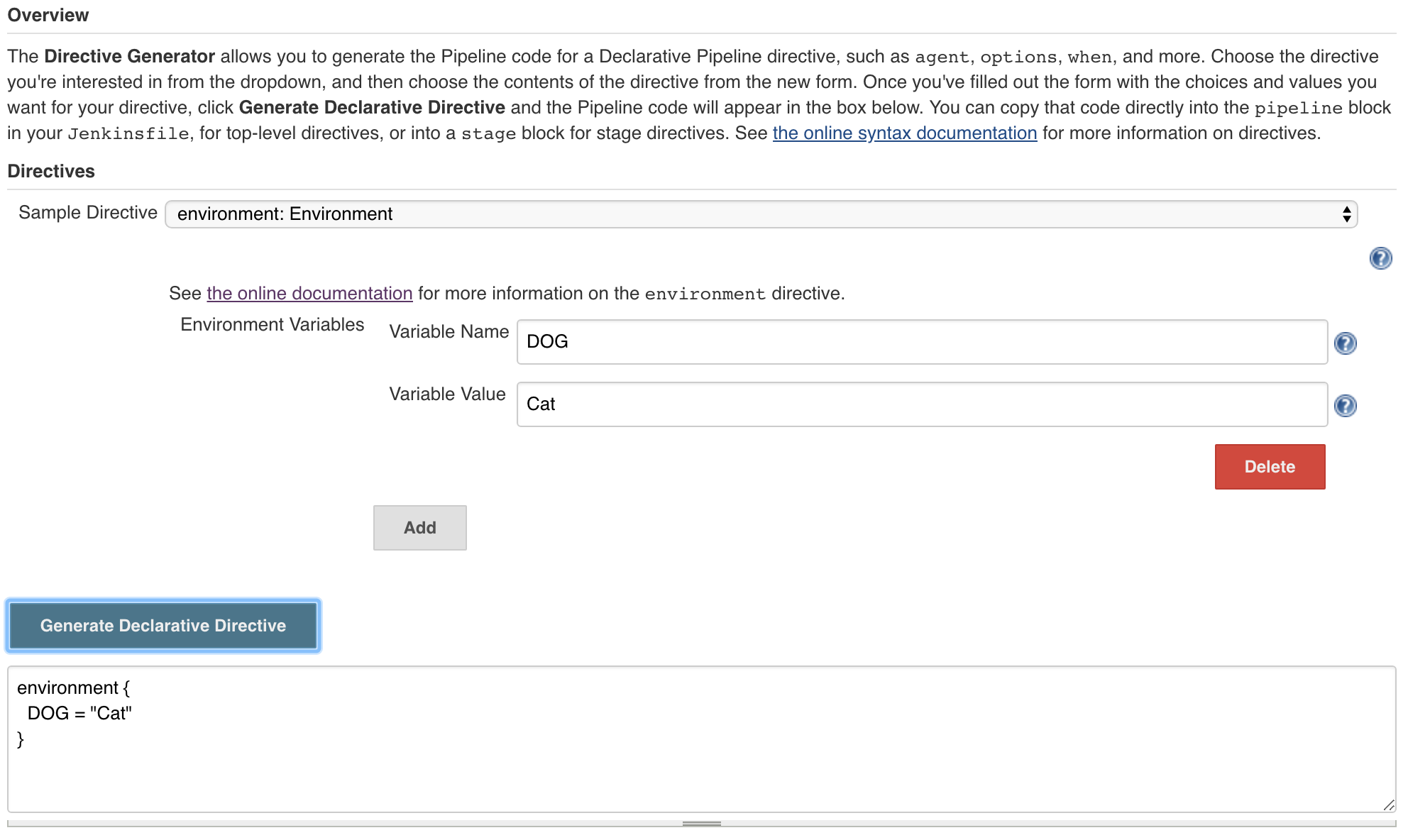
## HINTS FOR SELF-PACED

* We recommend that you right-click on links to labs and other materials  
  to open the new section in a separate browser tab
* Use the contents in the left frame to navigate between sections
* If you do not want to complete the entire class in one sitting, you can stop at any time
  + When you log back into the class, you are taken to the place where you stopped  
    and the work you did on previous sections is available

## LAB EXERCISE

[Introduction](https://s3.amazonaws.com/cloudbees-training-materials/training-pipeline-intermediate/master/selfpaced/labs/Introduction.html)

## RECENT FEATURES ADDED TO DECLARATIVE

* Declarative Directive Generator
* New when Conditions
* New post Conditions
* New options
* input Directive to Stage
* **DECLARATIVE DIRECTIVE GENERATOR**
* 

## NEW WHEN CONDITIONS

* equals
* changeRequest
* buildingTag
* tag
* beforeAgent

## EQUALS

* Compares two values and returns true if they’re equal
* You can also do "not equals" comparisons using the not { equals …​ } syntax

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'make package'

}

}

stage('Test') {

when { equals expected: 2, actual: currentBuild.number }

steps {

sh 'make check'

}

}

stage('Deploy') {

steps {

echo 'Deploying only because this commit is tagged...'

sh 'make deploy'

}

}

}

}

## CHANGEREQUEST

* Returns true if this Pipeline is building a change request, such as a GitHub or Bitbucket pull request
  + when { changeRequest() }
* More detailed checks by using a filter against the change request, allowing you to ask "was this change request created by [joe@example.com](mailto:joe@example.com)?"
  + when { changeRequest authorEmail: "[joe@example.com](mailto:joe@example.com)" }
* You can also do pattern matching against the filters using a comparator to determine if the pull request was from anyone with the email address ending in @example.com
  + when { changeRequest authorEmail: "[\\w\_-.]+@example.com", comparator: 'REGEXP' }

## BUILDINGTAG

* A simple condition that just checks if the Pipeline is running against a tag in SCM, rather than a branch or a specific commit reference
  + when { buildingTag() }

## TAG

* A more detailed equivalent of buildingTag, allowing you to check against the tag name itself

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'make package'

}

}

stage('Test') {

when { equals expected: 2, actual: currentBuild.number }

steps {

sh 'make check'

}

}

stage('Deploy') {

when { tag "release-\*" }

steps {

echo 'Deploying only because this commit is tagged...'

sh 'make deploy'

}

}

}

}

## BEFOREAGENT

* Allows you to specify that the when conditions should be evaluated before entering the agent for the stage
* When beforeAgent true is specified, you will not have access to the agent’s workspace, but you can avoid unnecessary SCM checkouts and waiting for a valid agent to be available

pipeline {

agent none

stages {

stage('Example Build') {

steps {

echo 'Hello World'

}

}

stage('Example Deploy') {

agent {

label "some-label"

}

when {

beforeAgent true

branch 'production'

}

steps {

echo 'Deploying'

}

}

}

}

## NEW POST CONDITIONS

* fixed
* regression

## FIXED

* Checks to see if the current run is successful and if the previous run was either failed or unstable

## REGRESSION

* Checks to see if the current run’s status is worse than the previous run’s status
* If the previous run was successful and the current run is unstable, this fires and its block of steps executes
* It also runs if the previous run was unstable and the current run is a failure, etc

## NEW OPTIONS

* checkoutToSubdirectory
* newContainerPerStage

## CHECKOUTTOSUBDIRECTORY

* Allows you to override the location that the automatic SCM checkout uses
* Using checkoutToSubdirectory("foo"), your Pipeline checks out your repository to $WORKSPACE/foo, rather than the default of $WORKSPACE

## NEWCONTAINERPERSTAGE

* If you are using a top-level docker or dockerfile agent and want to ensure that each of your stages runs in a fresh container of the same image

## INPUT DIRECTIVE TO STAGE

pipeline {

agent any

stages {

stage('Example') {

input {

message "Should we continue?"

ok "Yes, we should."

submitter "alice,bob"

parameters {

string(name: 'PERSON', defaultValue: 'Mr Jenkins', description: 'Who should I say hello to?')

}

}

when {

equals expected: "Fred", actual: "${PERSON}"

}

steps {

echo "Hello, ${PERSON}, nice to meet you."

}

}

}

## USING DOCKER WITH PIPELINE

Pipeline has built-in support for interacting with Docker from within a Jenkinsfile

## CUSTOMIZE THE EXECUTION ENVIRONMENT

pipeline {

agent {

docker { image 'node:7-alpine' }

}

stages {

stage('Test') {

steps {

sh 'node --version'

}

}

}

}

## CACHING DATA FOR CONTAINERS

* Many build tools download external dependencies and cache them locally for future re-use
* Pipeline supports adding custom arguments that are passed to Docker, allowing users  
  to specify custom Docker Volumes to mount
  + These can be used for caching data on the agent between Pipeline runs

## CACHING DATA FOR CONTAINERS

pipeline {

agent {

docker {

image 'maven:3-alpine'

args '-v $HOME/.m2:/root/.m2'

}

}

stages {

stage('Build') {

steps {

sh 'mvn -B'

}

}

}

}

## USING MULTIPLE CONTAINERS

* It is increasingly common for code bases to rely on multiple, different technologies
* A repository might have both a Java-based back-end API implementation and a JavaScript-based front-end implementation
* Combining Docker and Pipeline allows a **Jenkinsfile** to use multiple types of technologies by combining the agent {} directive, with different stages

## USING MULTIPLE CONTAINERS

pipeline {

agent none

stages {

stage('Back-end') {

agent {

docker { image 'maven:3-alpine' }

}

steps {

sh 'mvn --version'

}

}

stage('Front-end') {

agent {

docker { image 'node:7-alpine' }

}

steps {

sh 'node --version'

}

}

}

}

## USING A DOCKERFILE

Dockerfile

FROM node:7-alpine

RUN apk add -U subversion

Jenkinsfile

pipeline {

agent { dockerfile true }

stages {

stage('Test') {

steps {

sh 'node --version'

sh 'svn --version'

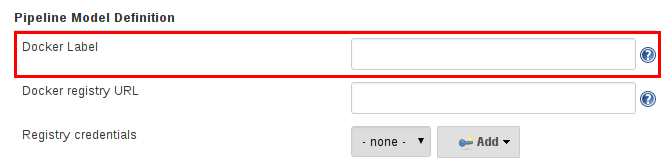
}

}

}

}

## SPECIFY A DOCKER LABEL

* By default, Pipeline assumes that any configured agent is capable of running  
  Docker-based Pipelines
* This can be problematic if some of your agents do not have Docker
* Pipeline enables you to specify the agents (by Label) to use when running Docker-based Pipelines
  + This can be specified as a global option in the Manage Jenkins page and on the Folder level,
* **SPECIFY A DOCKER LABEL**
* 

## SCRIPTED PIPELINE

## OVERVIEW

* Scripted syntax is a domain specific language based on Apache Groovy
  + **Most** functionality provided by the Groovy language is made available to users of Scripted syntax, which means it can be a very expressive and flexible tool for authoring  
    continuous delivery pipelines
  + Scripted syntax offers a tremendous amount of flexibility and extensibility to Jenkins users
  + The learning curve for the scripted pipeline syntax is steep, which is not typically desirable  
    for all members of a given team
* Declarative syntax offers a simpler and more opinionated syntax for authoring Jenkins Pipeline
* We are not going to discuss how to implement a Scripted Pipeline in this class

## DECLARATIVE AND SCRIPTED

* Have the same Pipeline sub-system underneath
  + Both are durable implementations of "Pipeline as code"
  + Both can use steps built into Pipeline or provided by plugins
  + Both can utilize Shared Libraries

## DECLARATIVE VS SCRIPTED

* Declarative limits what is available to the user with a more strict and pre-defined structure, making it an ideal choice for simpler continuous delivery pipelines
* Scripted provides very few limits
  + The only limits on structure and syntax are defined by Groovy itself, not by Pipeline-specific systems
  + Useful when you have more complex requirements than Declarative can support out of the box
  + BUT it has few safeguards against errors you might make
* Use the script step to execute a block of Scripted syntax in a Declarative Pipeline

## SO WHAT DOES ALL THIS MEAN?

* Always start with Declarative syntax
* Extend with Shared Libraries
* Use a script step to introduce Scripted syntax only when you really need to

## FOR FURTHER READING

* <https://jenkins.io/doc/book/pipeline/syntax/#scripted-pipeline>

## USING A JENKINSFILE

## BENEFITS OF USING A JENKINSFILE

* Code review/iteration on the Pipeline
* Audit trail for the Pipeline
* Single source of truth for the Pipeline, which can be viewed  
  and edited by multiple members of the project

## WORKING WITH YOUR JENKINSFILE

## STRING INTERPOLATION

* Jenkins Pipeline uses rules identical to Groovy for string interpolation
* Groovy’s String interpolation support can be confusing

## STRING INTERPOLATION

* Groovy supports declaring a string with either single quotes or double quotes:
* def singlyQuoted = 'Hello'

def doublyQuoted = "World"

## STRING INTERPOLATION

* String interpolation only works for strings in double-quotes, not for strings in single-quotes.
  + For example, this code:
  + def username = 'Jenkins'
  + echo 'Hello Mr. ${username}'

echo "I said, Hello Mr. ${username}"

* + Results in:
  + Hello Mr. ${username}

I said, Hello Mr. Jenkins

* You can see that the dollar-sign ($) based string interpolation works for the string that is in double quotes but does not work for the string in single quotes

## USING ENVIRONMENT VARIABLES

pipeline {

agent any

stages {

stage('Example') {

steps {

echo "Running ${env.BUILD\_ID} on ${env.JENKINS\_URL}"

}

}

}

}

## SETTING ENVIRONMENT VARIABLES

* An environment directive used in the top-level pipeline block  
  applies to all steps within the Pipeline
* An environment directive defined within a stage applies only  
  to the given environment variables for steps within the **stage**

## SETTING ENVIRONMENT VARIABLES

pipeline {

agent any

environment {

CC = 'clang'

}

stages {

stage('Example') {

environment {

DEBUG\_FLAGS = '-g'

}

steps {

sh 'printenv'

}

}

}

}

## CREDENTIALS

You might be used to seeing something like:

pipeline {

agent any

stages {

stage("test") {

steps {

withCredentials([usernameColonPassword(variable: 'SERVICE\_CREDS', credentialsId: 'my-cred-id')]) {

sh """

echo "Service user is $SERVICE\_CREDS\_USR"

echo "Service password is $SERVICE\_CREDS\_PSW"

curl -u $SERVICE\_CREDS https://myservice.example.com

"""

}

}

}

}

}

## CREDENTIALS

* The **environment** directive supports a special helper method credentials()
  + This can be used to access pre-defined credentials by their identifier  
    in the Jenkins environment

## USERNAME AND PASSWORD

The environment variable specified is set to username:password and two additional environment variables are defined automatically: MYVARNAME\_USR and MYVARNAME\_PSW respectively.

pipeline {

agent any

environment {

SERVICE\_CREDS = credentials('my-prefined-username-password')

}

stages {

stage("test") {

steps {

sh """

echo "Service user is $SERVICE\_CREDS\_USR"

echo "Service password is $SERVICE\_CREDS\_PSW"

curl -u $SERVICE\_CREDS https://myservice.example.com

"""

}

}

}

}

## CREDENTIALS

* The **environment** directive supports a special helper method credentials()
  + This can be used to access pre-defined credentials by their identifier  
    in the Jenkins environment

## USERNAME AND PASSWORD

The environment variable specified is set to username:password and two additional environment variables are defined automatically: MYVARNAME\_USR and MYVARNAME\_PSW respectively.

pipeline {

agent any

environment {

SERVICE\_CREDS = credentials('my-prefined-username-password')

}

stages {

stage("test") {

steps {

sh """

echo "Service user is $SERVICE\_CREDS\_USR"

echo "Service password is $SERVICE\_CREDS\_PSW"

curl -u $SERVICE\_CREDS https://myservice.example.com

"""

}

}

}

}

## SECRET TEXT

The environment variable specified will be set to the Secret Text content

pipeline {

agent any

environment {

SOME\_SECRET\_TEXT = credentials('jenkins-secret-text-id')

}

stages {

stage("test") {

steps {

sh """

echo "secret text is $SOME\_SECRET\_TEXT"

"""

}

}

}

}

## SECRET FILE

The environment variable specified will be set to the location of the file that is temporarily created

pipeline {

agent any

environment {

SOME\_SECRET\_FILE = credentials('jenkins-secret-file-id')

}

stages {

stage("test") {

steps {

sh """

echo "secret file location is $SOME\_SECRET\_FILE"

"""

}

}

}

}

## SSH WITH PRIVATE KEY

The environment variable specified will be set to the location of the SSH key file that is temporarily created and two additional environment variables may be automatically defined: MYVARNAME\_USR and MYVARNAME\_PSW (holding the passphrase).

pipeline {

agent any

environment {

SSH\_CREDS = credentials('my-prefined-ssh-creds')

}

stages {

stage("test") {

steps {

sh """

echo "SSH private key is located at $SSH\_CREDS"

echo "SSH user is $SSH\_CREDS\_USR"

echo "SSH passphrase is $SSH\_CREDS\_PSW"

"""

}

}

}

}

## WHAT IF MY CREDENTIALS ISN’T ONE OF THESE FOUR?

Unsupported credentials type causes the pipeline to fail with the message:  
  
org.jenkinsci.plugins.credentialsbinding.impl.CredentialNotFoundException: No suitable binding handler could be found for type <unsupportedType>  
  
and you’ll continue to use **withCredentials**

## PARAMETERS

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

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 :('

}

}

}

## OPTIONAL STEP ARGUMENTS

Pipeline follows the Groovy language convention of allowing parentheses to be omitted around method arguments

## OPTIONAL STEP ARGUMENTS

These two statements are functionally equivalent:

git url: 'git://example.com/amazing-project.git', branch: 'master'

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

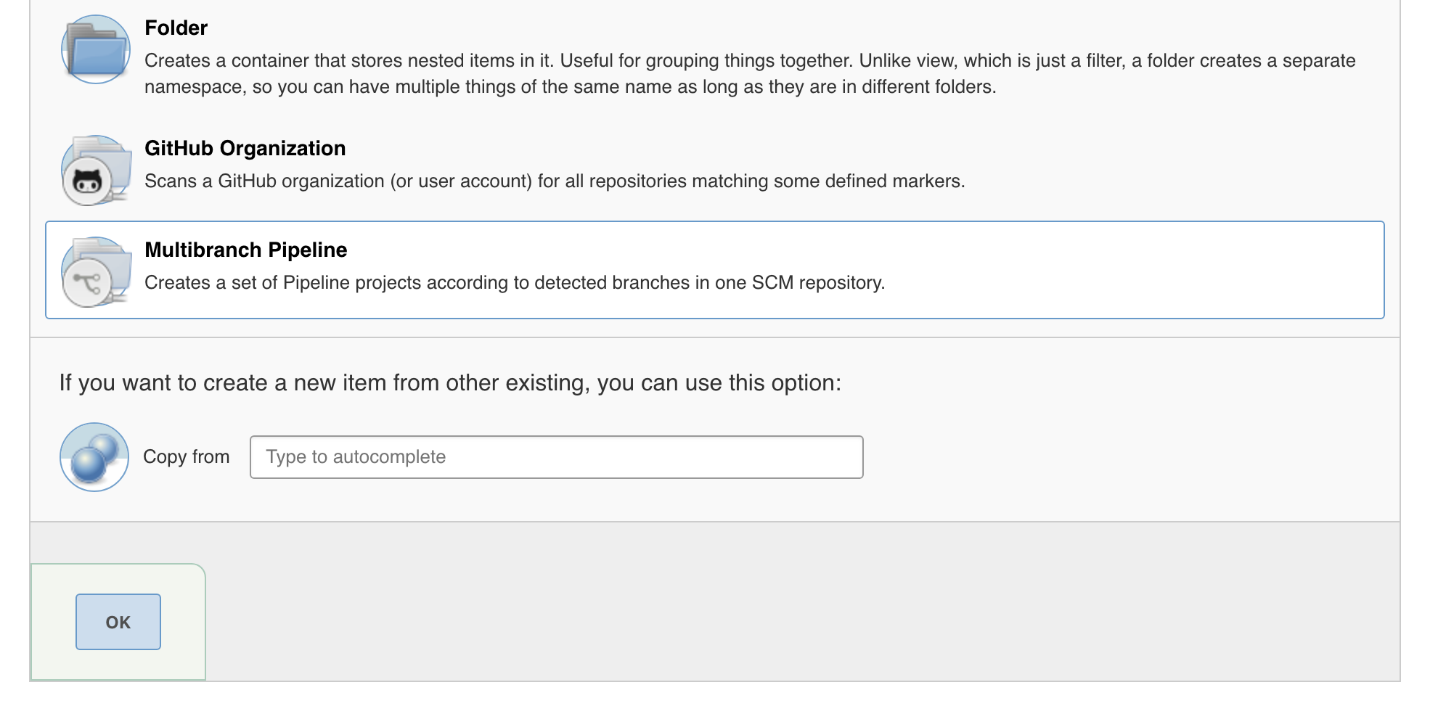
## WHAT IS A MULTIBRANCH PIPELINE ?

* Configured to point to a SCM
* Contains Pipeline Jobs
  + One Pipeline **per** SCM branch with a Jenkinsfile
    - Without Multibranch, each Pipeline maps to only one branch of the SCM
  + Supports Pull Requests as well
  + Jobs are **automatically** created/deleted
    - Without Multibranch, no automatic discovery
* Is implemented as a Jenkins job type
  + Basically: it is a **folder**
* All new Pipelines should be created as Multibranch Pipelines
  + All Pipelines created with Blue Ocean are Multibranch

## CREATE A MULTIBRANCH PIPELINE USING THE CLASSIC UI

* From the Jenkins Dashboard, click on **New Item** in the left frame
  + Enter the name of your new Pipeline in the box that is provided
  + Choose "Multibranch Pipeline" from the list provided and click "OK"
* Choose your SCM from the list under "Branch Sources"
  + Fill in the fields that are displayed to configure your SCM
  + (Optional) Configure a webhook from SCM
* Push a Jenkinsfile on any branch
  + Merge branch: jobs automatically managed
* Everything is automated, which greatly reduces the administrative tasks

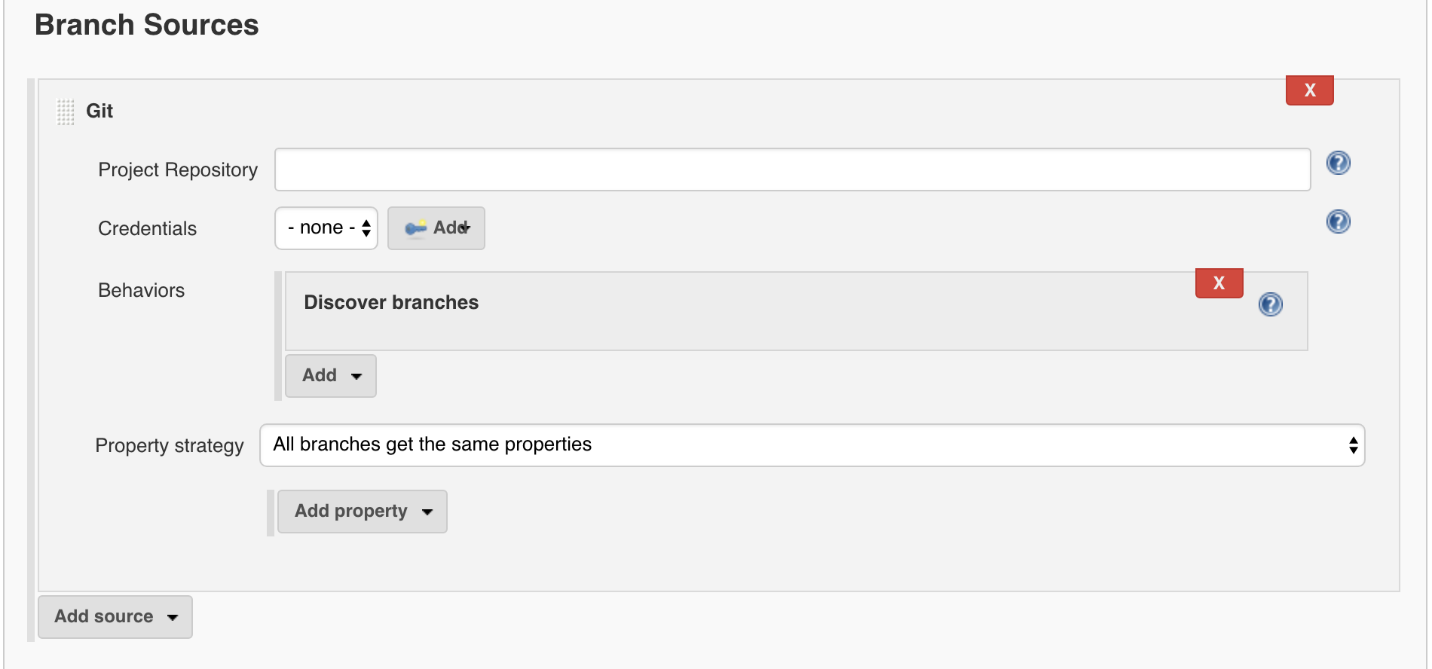
## CREATE A NEW JOB OF TYPE "MULTIBRANCH PIPELINE"

****

## CONFIGURE THE BRANCH SOURCE (SCM)

****

## CONFIGURE THE BRANCH SOURCE (SCM)

****

## MULTIBRANCH PIPELINES CONFIGURATIONS

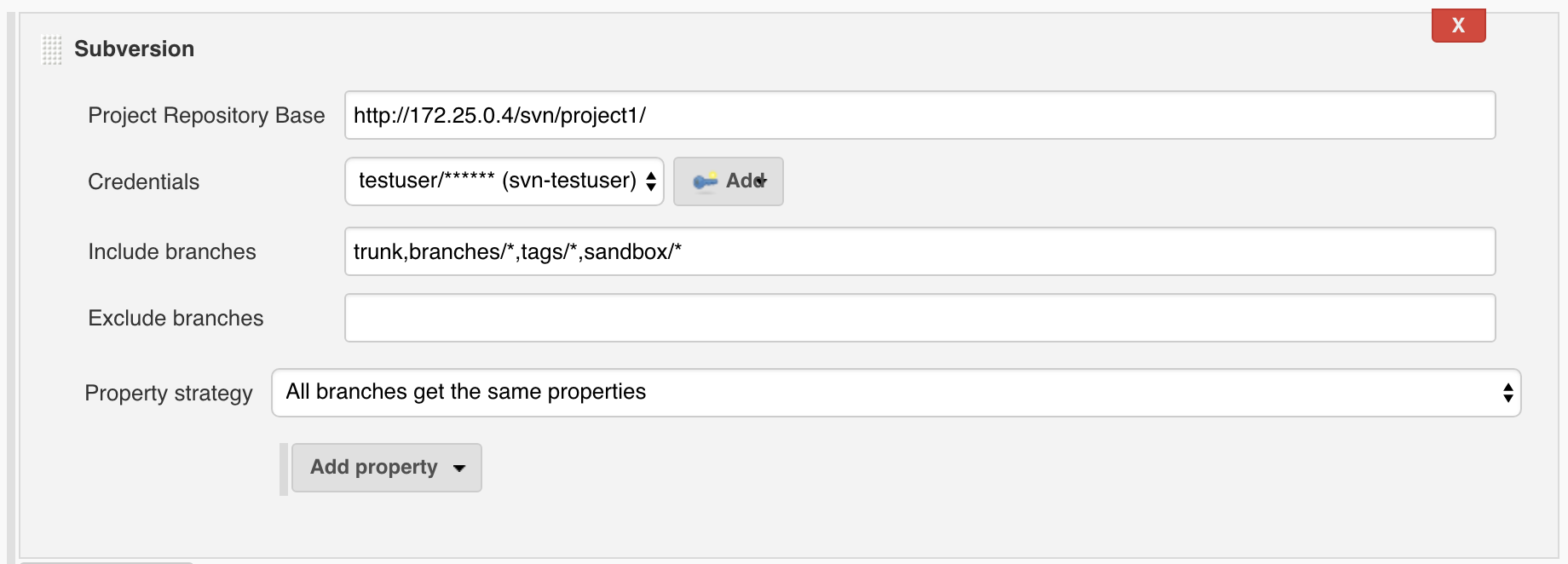
* **Customizable** retention policy
  + "Orphaned Item Strategy" configuration section
* **Triggers**
  + If you aren’t using webhooks to trigger jobs, you can tell the job how often to run

## ORGANIZATION SCANNING

* Currently only works with GitHub Organization folders and Bitbucket Team/Project folders
  + Corresponding branch source plugins must be installed
  + Other SCMs may be supported in the future
* Admin selects the job type associated with the SCM type
  + One credential (API token generally) needed
  + Maps to an "organization folder" or "team/project" as top level
* Each repository maps to a Multibranch pipeline
  + **Inside** the "organization folder" or "team/project"
  + More **automation**
  + Automate **webhooks** creation

## BUT WHAT IF I’M STILL USING SUBVERSION?

* You can still use Multibranch!

****

## FOR FURTHER READING

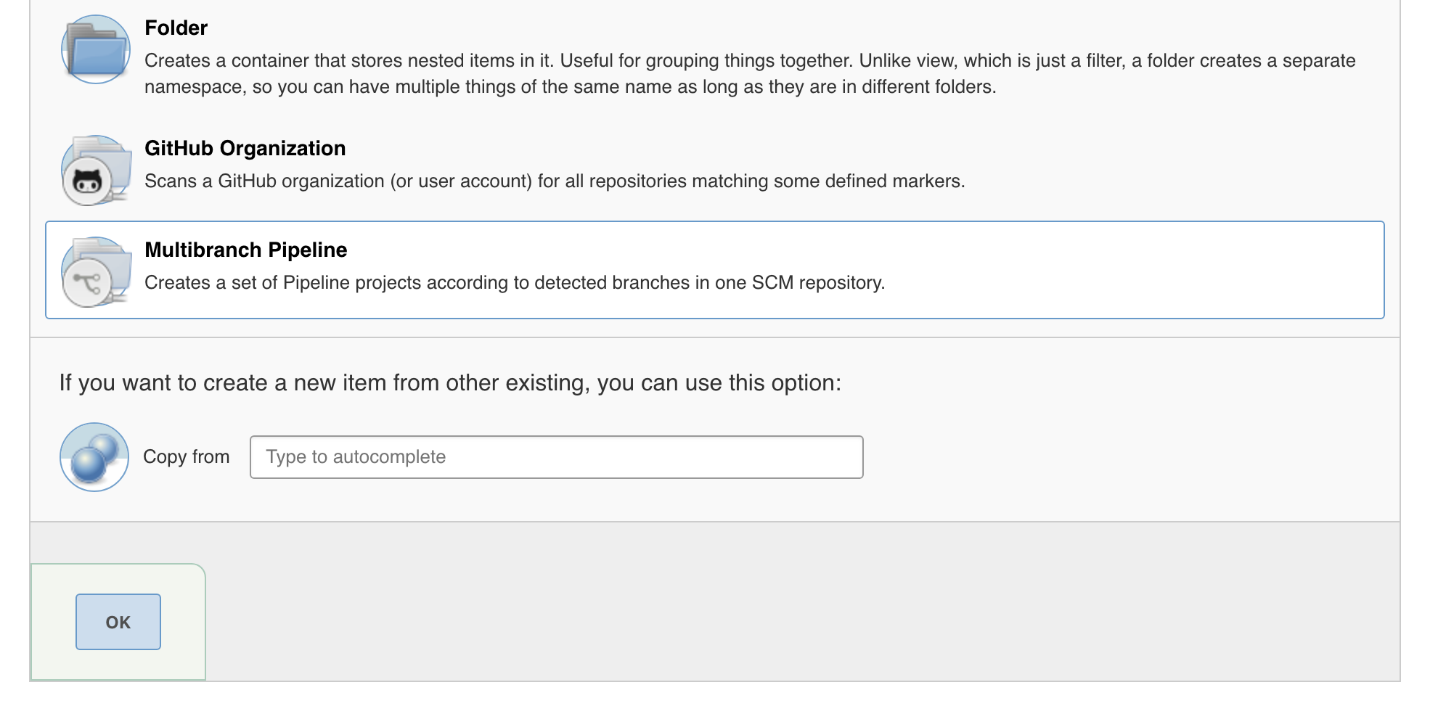
* Some recommended readings on this subject:
  + [Getting started with Blue Ocean](https://jenkins.io/doc/book/blueocean/getting-started)
  + [Branches and Pull Requests](https://jenkins.io/doc/book/pipeline/multibranch)
  + [Pipeline-as-code with Multibranch Workflows in Jenkins](https://jenkins.io/blog/2015/12/03/pipeline-as-code-with-multibranch-workflows-in-jenkins)

## PIPELINE WITHOUT BLUE OCEAN

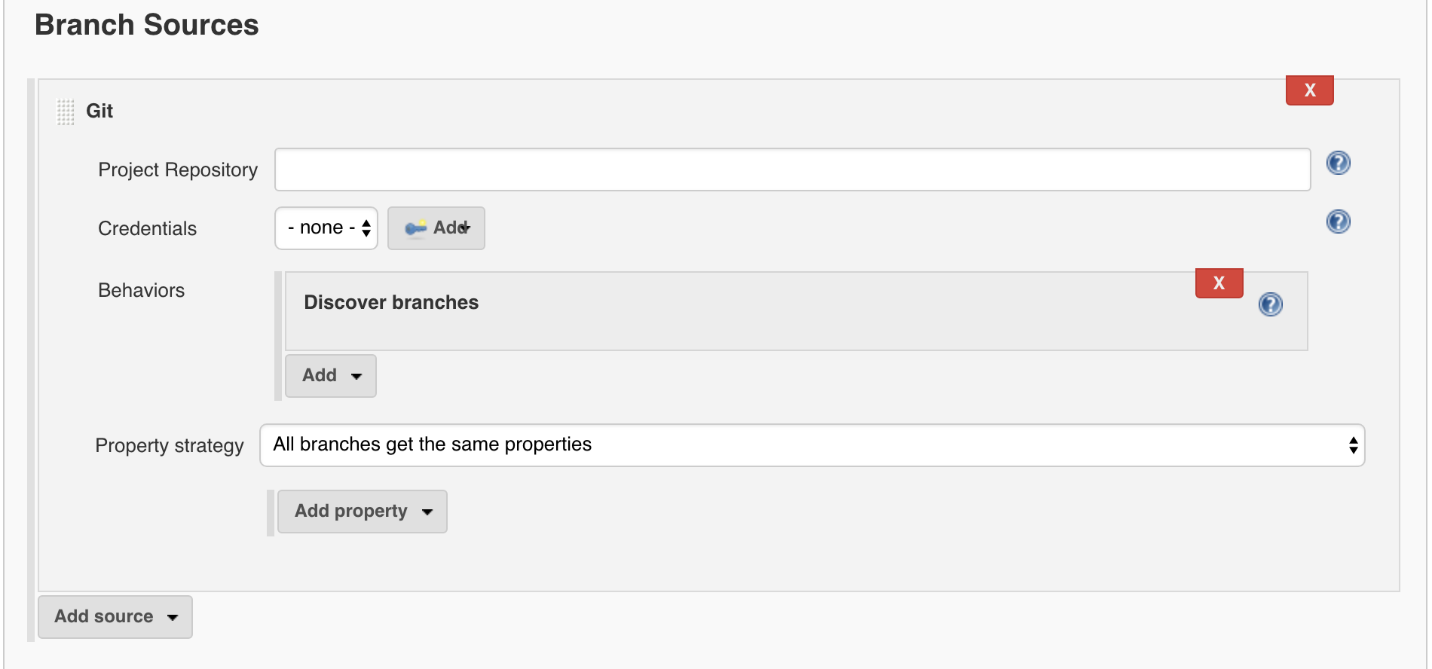
## OVERVIEW

* Pipelines can be implemented and modified outside Blue Ocean
* Use your favorite editor to maintain job syntax

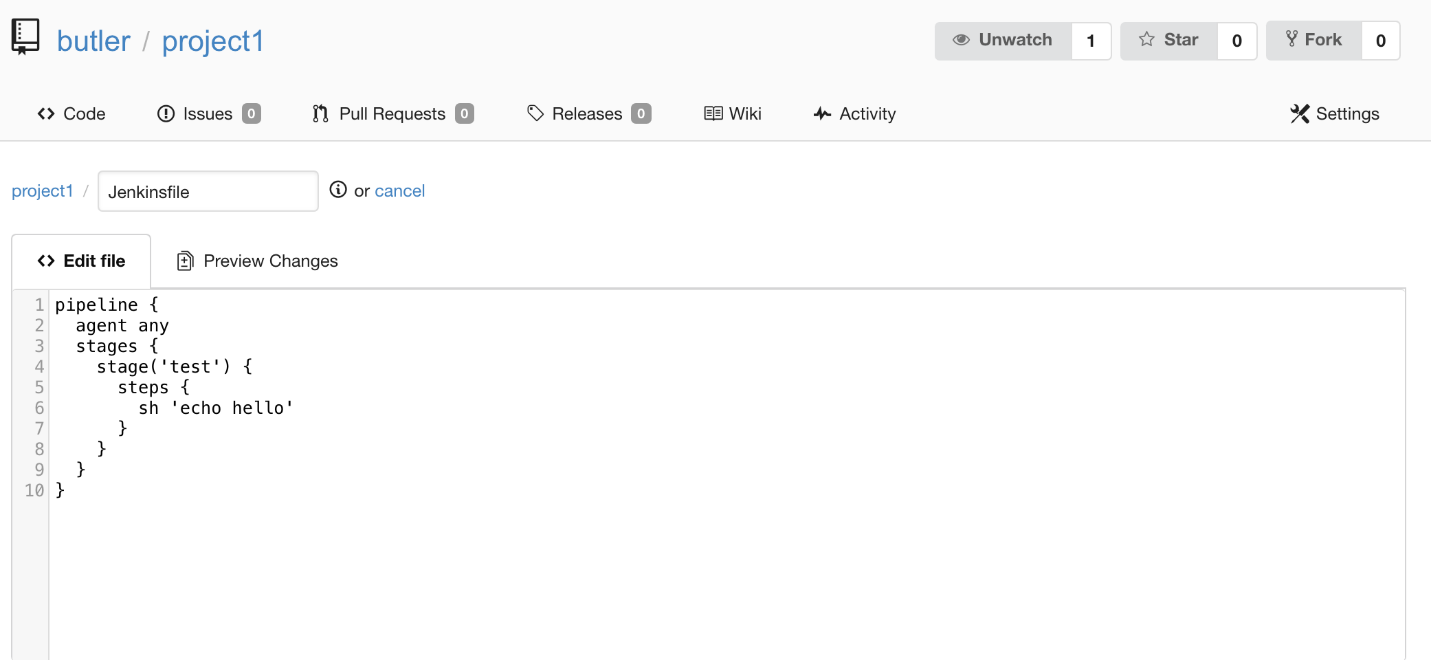
## CREATE A NEW MULTIBRANCH JOB

****

## REGISTER THE SCM REPOSITORY

****

**CREATE JENKINSFILE**

****

## INTRODUCTION TO SHARED LIBRARIES

## WHY USE SHARED LIBRARIES ?

* Allow you to share and reuse Pipeline code
* Scale your Jenkins Pipeline usage
  + Supports collaboration between a large number of teams working on a large number of projects
* Help administrators manage code sprawl
  + Write once, propagate everywhere
  + Pipeline as code everywhere
* Use tooling to avoid silos
  + Collaborate instead of enforcing

## NOTES ABOUT SHARED LIBRARIES

* Extremely powerful
* Learning curve
  + First step is not easy
  + Requires deeper understanding of Pipeline
* Adds some overhead
  + Testing
  + Maintenance
* Many uses
  + Take time to read the documentation

## FOR FURTHER READING

* [Extending with Shared Libraries](https://jenkins.io/doc/book/pipeline/shared-libraries)

## IMPLEMENT SHARED LIBRARIES

## HOW TO IMPLEMENT PIPELINE SHARED LIBRARIES

1. Create a separate SCM repository for the shared library
2. Configure a Global Pipeline Library in Jenkins
3. Code the custom step and check it into the shared library SCM repository
4. Call the custom step from your Pipeline

## CREATE AN SCM REPOSITORY

## SCM DIRECTORY STRUCTURE

* The directory structure of a Shared Library repository is as follows:
* (root)
* +- src # Groovy source files
* | +- org
* | +- foo
* | +- Bar.groovy # for org.foo.Bar class
* +- vars
* | +- foo.groovy # for global 'foo' variable
* | +- foo.txt # help for 'foo' variable
* +- resources # resource files (external libraries only)
* | +- org
* | +- foo

| +- bar.json # static helper data for org.foo.Bar

## SRC DIRECTORY

* src directory uses a standard Java source directory structure
* This directory is added to the classpath when executing Pipelines
* You should **rarely** (preferably never) add anything to the src directory

## VARS DIRECTORY

* vars directory contains scripts that define **custom steps** accessible from a Pipeline.
* All custom steps are defined in the root of the vars directory
  + You can not use subfolders to var
* Each file should define one step
  + The name should be the name of that step, camelCased, with the .groovy suffix.
* The matching .txt file, if present, can contain documentation
  + This will be processed through the system’s configured markup formatter

## RESOURCES DIRECTORY

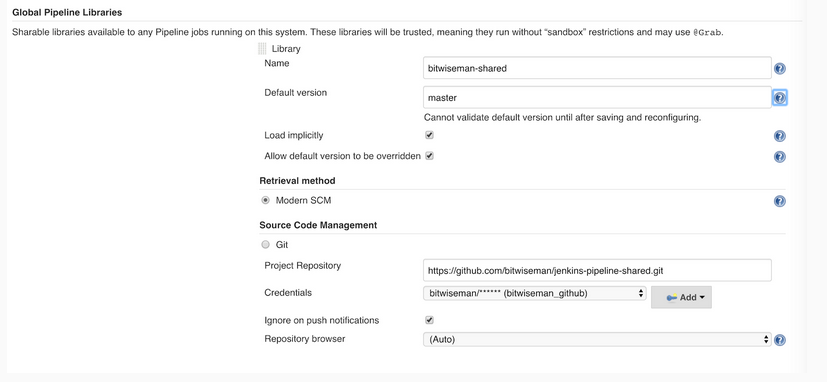
* The **libraryResource** step reads files from the resources directory  
  and returns the content as a plain string
* You can use subdirectories in the resources directory
  + Be sure to give each directory a name that is meaningful to you

## OTHER DIRECTORIES

* Other directories under the root of the shared library  
  are reserved for future enhancements

## CONFIGURE THE SHARED LIBRARY

* **HOW TO CONFIGURE A SHARED LIBRARY**
* Navigate to *Manage Jenkins* → *Configure System* on the Manage Jenkins page:



## SHARED LIBRARY CONFIGURATION NOTES

* Global Libraries configured in Jenkins are considered trusted
  + Steps from this library run **outside** the Groovy sandbox
* Libraries configured at multibranch/folder level are considered not trusted
  + Steps from this library run **inside** the Groovy sandbox
  + Prefer libraries at multibranch/folder level to reduce risk to Jenkins server  
    from libraries outside the sandbox
* Set "Default version" to **master** to have Pipelines call custom steps from the master branch
  + If "Allow default version to be overridden" is enabled, a Pipeline can override this  
    to call custom steps from other branches using the @Library annotation
* When Load implicitly is enabled, the default branch is automatically available to all Pipelines; custom steps can also be loaded manually using a @Library annotation

## WRITE SHARED LIBRARY CUSTOM STEPS

## CREATE A CUSTOM STEP

* Create a file that has the desired name of our custom step
* Add code to a call() method inside that file
  + Code the custom step exactly as you would code it in a Pipeline
  + If the custom step is for code you created in a Pipeline, you can  
    basically copy-and-paste that code
* Check the file into the SCM repository
  + For testing, check the new custom step into a branch other than **master**

## CREATE A HELLOWORLDSIMPLE CUSTOM STEP

Jenkinsfile

pipeline {

agent any

stages {

stage('hello') {

steps {

sh "echo Hello world, Fred. It is Friday."

}

}

}

}

## CREATE A HELLOWORLDSIMPLE CUSTOM STEP

vars/helloWorldSimple.groovy

def call(String name, String dayOfWeek) {

sh "echo Hello World ${name}. It is ${dayOfWeek}."

}

## CALL THE SHARED LIBRARY CUSTOM STEP

## LOADING THE LIBRARY

* You can load a shared library from your Pipeline
  + By Annotation
  + By DSL keyword
  + Implicitly

## HELLOWORLDSIMPLE EXAMPLE

Jenkinsfile

@Library('shared-starter') \_

pipeline {

agent any

stages {

stage('hello') {

steps {

helloWorldSimple("Fred","Friday")

}

}

}

}

vars/helloWorldSimple.groovy

def call(String name, String dayOfWeek) {

sh "echo Hello World ${name}. It is ${dayOfWeek}."

}

## HELLOWORLD EXAMPLE

Jenkinsfile

@Library('shared-starter') \_

pipeline {

agent any

stages {

stage('hello') {

steps {

helloWorld(name: "Fred", dayOfWeek: "Friday")

}

}

}

}

vars/helloWorld.groovy

def call(Map config = [:]) {

sh "echo Hello World ${config.name}. It is ${config.dayOfWeek}."

}

## BACK TO THE FUTURE

* In the "Pipelines - Fundamentals" course, we created code to send email and Slack notifications when a build starts, when it completes and when it fails
* Let’s look at that code and then turn it into a custom step that any Pipeline can call

## NOTIFICATIONS WHEN BUILD STARTS

stages {

stage ('Start') {

steps {

// send build started notifications

slackSend (

color: '#FFFF00',

message: "STARTED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})"

)

// send to email

emailext (

subject: "STARTED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>STARTED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

}

}

}

## NOTIFICATIONS WHEN BUILD SUCCEEDS

post {

success {

slackSend (

color: '#00FF00',

message: "SUCCESSFUL: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})"

)

emailext (

subject: "SUCCESSFUL: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>SUCCESSFUL: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

}

}

**NOTIFICATIONS WHEN BUILD FAILS**

post {

failure {

slackSend (

color: '#FF0000',

message: "FAILED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})"

)

emailext (

subject: "FAILED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>FAILED: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

}

}

**HERE WE GO**

**CUSTOM STEP FOR SENDNOTIFICATIONS**

vars/sendNotifications.groovy

def call(Map config = [:]) {

slackSend (

color: "${config.slackSendColor}",

message: "${config.message}: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]' (${env.BUILD\_URL})"

)

// send to email

emailext (

subject: "${config.message}: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]'",

body: """<p>${config.message}: Job '${env.JOB\_NAME} [${env.BUILD\_NUMBER}]':</p>

<p>Check console output at &QUOT;<a href='${env.BUILD\_URL}'>${env.JOB\_NAME} [${env.BUILD\_NUMBER}]</a>&QUOT;</p>""",

recipientProviders: [[$class: 'DevelopersRecipientProvider']]

)

}

**NOTIFICATIONS WHEN BUILD STARTS**

Jenkinsfile

stages {

stage ('Start') {

steps {

sendNotifications(

slackSendColor: "#FFFF00",

message: "STARTED"

)

}

}

}

**NOTIFICATIONS WHEN BUILD SUCCEEDS**

Jenkinsfile

post {

success {

sendNotifications(

slackSendColor: "#00FF00",

message: "SUCCESSFUL"

)

}

}

**NOTIFICATIONS WHEN BUILD FAILS**

Jenkinsfile

post {

failure {

sendNotifications(

slackSendColor: "#FF0000",

message: "FAILED"

)

}

}

**LET’S MAKE THE JENKINSFILE EVEN MORE CONCISE**

**NOTIFICATIONS WHEN BUILD STARTS**

Jenkinsfile

stages {

stage ('Start') {

steps {

sendNotificationsStart()

}

}

}

vars/sendNotificationsStart.groovy

def call() {

sendNotifications(

slackSendColor: "#FFFF00",

message: "STARTED"

)

}

**NOTIFICATIONS WHEN BUILD SUCCEEDS**

Jenkinsfile

post {

success {

sendNotificationsSuccess()

}

}

vars/sendNotificationsSuccess.groovy

def call() {

sendNotifications(

slackSendColor: "#00FF00",

message: "SUCCESSFUL"

)

}

Solution For Notifications

**@Library**('shared-library') \_

pipeline {

agent none

stages {

stage('Fluffy Build') {

parallel {

stage('Build Java 8') {

agent {

node {

label 'java8'

}

}

steps {

sh "./jenkins/build.sh"

}

post {

success {

stash(name: 'Java 8', includes: 'target/\*\*')

}

}

}

stage('Build Java 7') {

agent {

node {

label 'java7'

}

}

steps {

sh './jenkins/build.sh'

}

post {

success {

postBuildSuccess(stashName: "Java 7")

}

}

}

}

}

stage('Fluffy Test') {

parallel {

stage('Backend Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-static.sh'

}

}

stage('Backend Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-static.sh'

}

}

}

}

stage('Confirm Deploy') {

when {

branch 'master'

}

steps {

timeout(time: 3, unit: 'MINUTES' ) {

input(message: "Okay to Deploy to Staging?", ok: "Let's Do it!")

}

}

}

stage('Fluffy Deploy') {

when {

branch 'master'

}

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh "./jenkins/deploy.sh ${params.DEPLOY\_TO}"

}

}

}

parameters {

string(name: 'DEPLOY\_TO', defaultValue: 'dev', description: '')

}

}

## MORE SHARED LIBRARY EXAMPLES

**SIMPLIFYING JENKINSFILES**

**DEFINE THE WHOLE PIPELINE AS A CUSTOM STEP**

Jenkinsfile

@Library('shared-starter') \_

helloWorldPipeline(name: "Fred", dayOfWeek: "Friday")

vars/helloWorldPipeline.groovy

def call(Map pipelineParams) {

pipeline {

agent any

stages {

stage('hello') {

steps {

helloWorld(name: "${pipelineParams.name}", dayOfWeek: "${pipelineParams.dayOfWeek}")

}

}

}

}

}

**PIPELINE HAS A (NOT SO) WELL KEPT SECRET**

* Pipeline gives you the ability to add your own DSL elements
* Pipeline is itself a DSL, so you can extend it

## WHY YOU WOULD WANT YOUR OWN DSL

* To reduce boilerplate by encapsulating common items you do in one DSL statement
* To provide a DSL that provides a prescriptive way that builds should happen across  
  your team or company

## PREVIOUS EXAMPLE AS DSL

Jenkinsfile

@Library('shared-starter') \_

helloWorldPipeline {

name = "Fred"

dayOfWeek = "Friday"

}

vars/helloWorldPipeline.groovy

def call(body) {

def pipelineParams= [:]

body.resolveStrategy = Closure.DELEGATE\_FIRST

body.delegate = pipelineParams

body()

pipeline {

agent any

stages {

stage('hello') {

steps {

helloWorld(name: "${pipelineParams.name}", dayOfWeek: "${pipelineParams.dayOfWeek}")

}

}

}

}

}

## LAB EXERCISE

[Create a Corporate Pipeline](https://s3.amazonaws.com/cloudbees-training-materials/training-pipeline-intermediate/master/selfpaced/labs/Create_a_Corporate_Pipeline.html)

**def** call(body) {

**def** pipelineParams= [:]

body.resolveStrategy = Closure.DELEGATE\_FIRST

body.delegate = pipelineParams

body()

pipeline {

agent none

stages {

stage('Fluffy Build') {

parallel {

stage('Build Java 8') {

agent {

node {

label 'java8'

}

}

post {

success {

stash(name: 'Java 8', includes: 'target/\*\*')

}

}

steps {

runLinuxScript(name: "build.sh")

}

}

stage('Build Java 7') {

agent {

node {

label 'java7'

}

}

post {

success {

postBuildSuccess(stashName: "Java 7")

}

}

steps {

runLinuxScript(name: "build.sh")

}

}

}

}

stage('Fluffy Test') {

parallel {

stage('Backend Java 8') {

agent {

node {

label 'java8'

}

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-backend.sh'

}

}

stage('Frontend') {

agent {

node {

label 'java8'

}

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-frontend.sh'

}

}

stage('Performance Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 8') {

agent {

node {

label 'java8'

}

}

steps {

unstash 'Java 8'

sh './jenkins/test-static.sh'

}

}

stage('Backend Java 7') {

agent {

node {

label 'java7'

}

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-backend.sh'

}

}

stage('Frontend Java 7') {

agent {

node {

label 'java7'

}

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-frontend.sh'

}

}

stage('Performance Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 7') {

agent {

node {

label 'java7'

}

}

steps {

unstash 'Java 7'

sh './jenkins/test-static.sh'

}

}

}

}

stage('Confirm Deploy') {

when {

branch 'master'

}

steps {

timeout(time: 3, unit: 'MINUTES') {

input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!')

}

}

}

stage('Fluffy Deploy') {

agent {

node {

label 'java7'

}

}

when {

branch 'master'

}

steps {

unstash 'Java 7'

sh "./jenkins/deploy.sh ${pipelineParams.deployTo}"

}

}

}

}

}

## DURABILITY

## DURABILITY AND PIPELINE SPEED

* By default, Pipeline writes transient data to disk FREQUENTLY
  + Running pipelines lose very little data from a system crash or an unexpected Jenkins restart
  + This frequent disk I/O can severely degrade Pipeline performance
* Speed/Durability settings allow you to improve performance by reducing the frequency at which data is written to disk
  + This incurs the risk that some data may be lost if the system crashes or Jenkins is restarted

## WHEN HIGHER-PERFORMANCE DURABILITY SETTINGS HELP

* Most basic Pipelines that just build and test the code
  + They frequently write build and test data and can easily be rerun
* Your Jenkins instance shows high iowait numbers
* Your Jenkins instance uses a networked file system or magnetic storage
* You run many Pipelines at the same time
* You run Pipelines with many steps (more than several hundred)

**WHEN NOT TO USE HIGHER-PERFORMANCE DURABILITY SETTINGS**

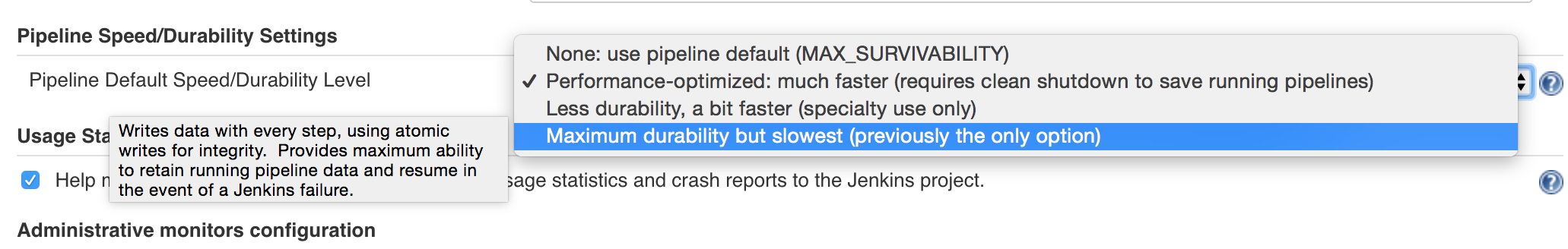
* Higher-Performance Durability Settings do not help if:
  + Your Pipelines mostly wait for shell/batch scripts to finish
  + Your Pipelines are writing large amounts of data to logs
    - This setting does not affect logging
  + You are not running Pipelines
* Higher-Performance settings are NOT recommended for:
  + Pipelines that modify the state of critical infrastructure
  + Pipelines that deploy code to a production

**DURABILITY SETTINGS**

* **MAX\_SURVIVABILITY** (default) - Slowest option
  + Writes data frequently
  + Little or no data is lost if Jenkins has a dirty shutdown
* **SURVIVABLE\_NONATOMIC** - A bit faster
  + Writes data with every step but avoids atomic writes
  + Faster than MAX\_SURVIVABILITY mode, especially on networked filesystems
* **PERFORMANCE\_OPTIMIZED** - Fastest option
  + Greatly reduces disk I/O
  + Data may be lost if Jenkins has a dirty shutdown

**HOW TO SET - GLOBALLY**

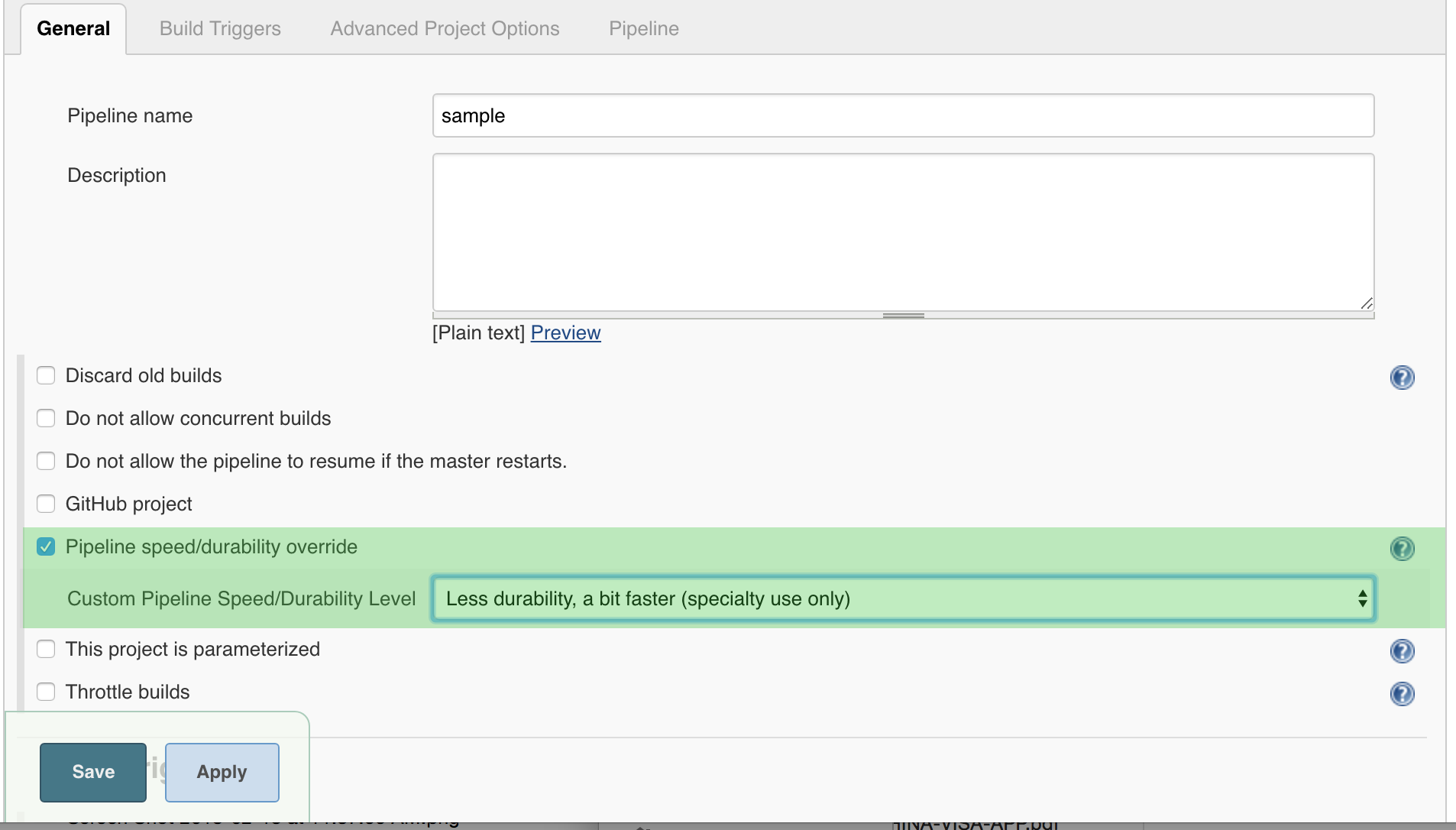
* Set under "Manage Jenkins" page on dashboard

****

**HOW TO SET - PIPELINE JOB TYPE**

* Overrides global setting
* Set under "Pipeline speed/durability override" at the top of the job configuration

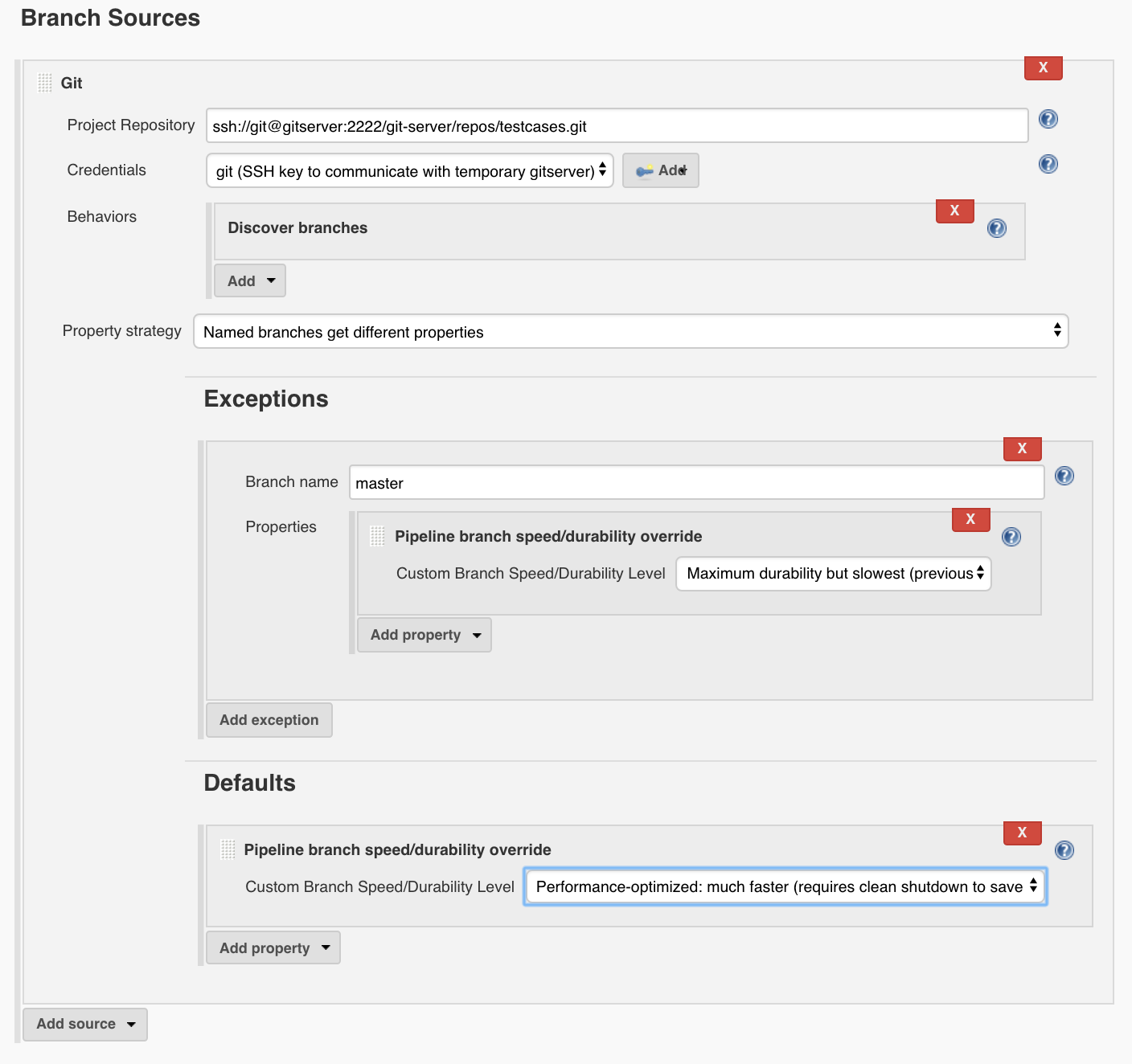
**HOW TO SET - PIPELINE JOB TYPE**

****

**HOW TO SET - MULTIBRANCH JOB TYPE**

* Overrides global setting
* Configure a custom Branch Property Strategy under the SCM

## HOW TO SET - MULTIBRANCH JOB TYPE

****

## HOW TO SET - JENKINSFILE

pipeline {

agent any

stages {

stage('Example') {

steps {

echo 'Hello World'

}

}

}

options {

durabilityHint('PERFORMANCE\_OPTIMIZED')

}

}

## BEST PRACTICES FOR DURABILITY SETTINGS

* Use PERFORMANCE\_OPTIMIZED mode for most build/test Pipelines
  + Set MAX\_SURVIVABILITY for Pipelines that modify critical infrastructure
  + You can set PERFORMANCE\_OPTIMIZED for the global setting then use the options step to choose a more durable setting for Pipelines where data preservation is more critical
* Use either MAX\_SURVIVABILITY or SURVIVABLE\_NONATOMIC for auditing
  + These modes record every step that is run
  + You can set one of these modes for the global setting then use the options step to choose PERFORMANCE\_OPTIMIZED for build/test Pipelines
* You can force a Pipeline to persist data by pausing it

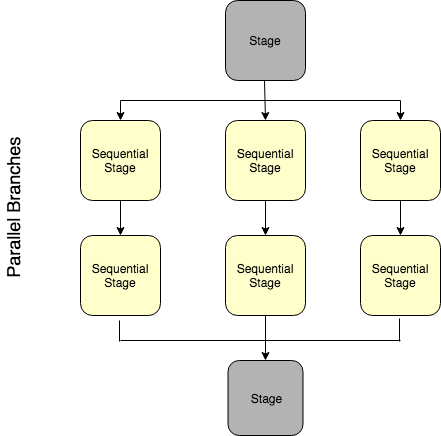
## FOR FURTHER READING

* [Scaling Pipelines](https://jenkins.io/doc/book/pipeline/scaling-pipeline/)
* **def** call(body) {
* **def** pipelineParams = [:]
* body.resolveStrategy = Closure.DELEGATE\_FIRST
* body.delegate = pipelineParams
* body()
* pipeline {
* agent none
* stages {
* stage('Fluffy Build') {
* parallel {
* stage('Build Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* post {
* success {
* stash(name: 'Java 8', includes: 'target/\*\*')
* }
* }
* steps {
* runLinuxScript(name: "build.sh")
* }
* }
* stage('Build Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* post {
* success {
* postBuildSuccess(stashName: "Java 7")
* }
* }
* steps {
* runLinuxScript(name: "build.sh")
* }
* }
* }
* }
* stage('Fluffy Test') {
* parallel {
* stage('Backend Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* post {
* always {
* junit 'target/surefire-reports/\*\*/TEST\*.xml'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-backend.sh'
* }
* }
* stage('Frontend') {
* agent {
* node {
* label 'java8'
* }
* }
* post {
* always {
* junit 'target/test-results/\*\*/TEST\*.xml'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-frontend.sh'
* }
* }
* stage('Performance Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-performance.sh'
* }
* }
* stage('Static Java 8') {
* agent {
* node {
* label 'java8'
* }
* }
* steps {
* unstash 'Java 8'
* sh './jenkins/test-static.sh'
* }
* }
* stage('Backend Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* post {
* always {
* junit 'target/surefire-reports/\*\*/TEST\*.xml'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-backend.sh'
* }
* }
* stage('Frontend Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* post {
* always {
* junit 'target/test-results/\*\*/TEST\*.xml'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-frontend.sh'
* }
* }
* stage('Performance Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-performance.sh'
* }
* }
* stage('Static Java 7') {
* agent {
* node {
* label 'java7'
* }
* }
* steps {
* unstash 'Java 7'
* sh './jenkins/test-static.sh'
* }
* }
* }
* }
* stage('Confirm Deploy') {
* when {
* branch 'master'
* }
* steps {
* timeout(time: 3, unit: 'MINUTES') {
* input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!')
* }
* }
* }
* stage('Fluffy Deploy') {
* agent {
* node {
* label 'java7'
* }
* }
* when {
* branch 'master'
* }
* steps {
* unstash 'Java 7'
* sh "./jenkins/deploy.sh ${pipelineParams.deployTo}"
* }
* }
* }
* options {
* durabilityHint('MAX\_SURVIVABILITY')
* }
* }
* }

**SEQUENTIAL STAGES**

* Another way to specify stages nested within other stages
* You can run multiple stages in each parallel branch
  + This gives you more visibility into the progress of your Pipeline

**SEQUENTIAL STAGES**

****

**SEQUENTIAL STAGES**

pipeline {

agent none

stages {

stage("build and deploy on Windows and Linux") {

parallel {

stage("windows") {

agent { label "windows" }

stages {

stage("build") {

steps {

bat "run-build.bat"

}

}

stage("deploy") {

when { branch "master" }

steps {

bat "run-deploy.bat"

}

}

}

}

stage("linux") {

agent { label "linux" }

stages {

stage("build") {

steps {

sh "./run-build.sh"

}

}

stage("deploy") {

when { branch "master" }

steps {

sh "./run-deploy.sh"

}

}

}

}

}

}

}

}

## Solution for Sequential stages

## def call(body) { def pipelineParams= [:] body.resolveStrategy = Closure.DELEGATE\_FIRST body.delegate = pipelineParams body() pipeline { agent none stages { stage('Build and Test Java') { parallel { stage('java8') { agent { label 'java8' } stages { stage("build8") { steps { runLinuxScript(name: "build.sh") } post { success { stash(name: 'Java 8', includes: 'target/\*\*') } } } stage('Backend Java 8') { steps { unstash 'Java 8' sh './jenkins/test-backend.sh' } post { always { junit 'target/surefire-reports/\*\*/TEST\*.xml' } } } stage('Frontend') { steps { unstash 'Java 8' sh './jenkins/test-frontend.sh' } post { always { junit 'target/test-results/\*\*/TEST\*.xml' } } } stage('Performance Java 8') { steps { unstash 'Java 8' sh './jenkins/test-performance.sh' } } stage('Static Java 8') { steps { unstash 'Java 8' sh './jenkins/test-static.sh' } } } } stage('java7') { agent { label 'java7' } stages { stage("build7") { steps { runLinuxScript(name: "build.sh") } post { success { postBuildSuccess(stashName: "Java 7") } } } stage('Backend Java 7') { steps { unstash 'Java 7' sh './jenkins/test-backend.sh' } post { always { junit 'target/surefire-reports/\*\*/TEST\*.xml' } } } stage('Frontend Java 7') { steps { unstash 'Java 7' sh './jenkins/test-frontend.sh' } post { always { junit 'target/test-results/\*\*/TEST\*.xml' } } } stage('Performance Java 7') { steps { unstash 'Java 7' sh './jenkins/test-performance.sh' } } stage('Static Java 7') { steps { unstash 'Java 7' sh './jenkins/test-static.sh' } } } } } } stage('Confirm Deploy') { when { branch 'master' } steps { timeout(time: 3, unit: 'MINUTES') { input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!') } } } stage('Fluffy Deploy') { agent { label 'java7' } when { branch 'master' } steps { unstash 'Java 7' sh "./jenkins/deploy.sh ${pipelineParams.deployTo}" } } } options { durabilityHint('MAX\_SURVIVABILITY') } } }

## RESTART FROM A STAGE

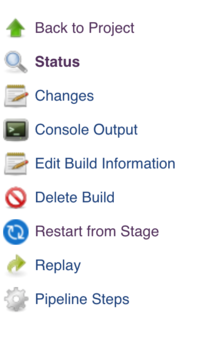
 You can restart any completed Declarative Pipeline from any top-level stage  
that ran in that Pipeline

 This allows you to rerun a Pipeline from a stage that failed due to transient or  
environmental considerations

**HOW TO USE**

* No additional configuration is needed in the Jenkinsfile to allow you to  
  restart stages in your Declarative Pipelines
* Once your Pipeline has completed, whether it succeeds or fails, you can go to  
  the side panel for the run in the classic UI and click on "Restart from Stage"

**HOW TO USE**

****

**HOW TO USE**

* You are prompted to choose from a list of top-level stages that were executed  
  in the original run, in the order they were executed
* Stages that were skipped due to an earlier failure are not available to be restarted,  
  but stages that were skipped due to a when condition not being satisfied are available
* The parent stage for a group of **parallel** stages or a group of nested **stages** to be  
  run sequentially are also not available - only top-level stages are allowed

**HOW TO USE**

****

**HOW TO USE**

* Once you choose a stage from which to restart and click submit,  
  a new build, with a new build number, starts
  + All inputs are the same, including SCM information, build parameters,  
    and the contents of any stash artifacts
* All stages before the selected stage are skipped and the Pipeline  
  starts executing at the selected stage
* From that point on, the Pipeline runs as normal

**PRESERVING STASHES FOR USE WITH RESTARTED STAGES**

* Normally, when you run the stash step in your Pipeline, the resulting stash of artifacts  
  is cleared when the Pipeline completes, regardless of the result of the Pipeline
* Since stash artifacts are not accessible outside of the Pipeline run that created them,  
  this has not created any limitations on usage
* With Declarative stage restarting, you may want to be able to **unstash** artifacts  
  from a stage that ran before the stage from which you are restarting
* To enable stash preservation, use the preserveStashes job property
  + This allows you to configure a maximum number of completed runs  
    whose stash artifacts should be preserved for reuse in a restarted run
* You can specify anywhere from 1 to 50 as the number of runs to preserve

## PRESERVING STASHES FOR USE WITH RESTARTED STAGES

options {

preserveStashes()

// or

preserveStashes(buildCount: 5)

}

* When a Pipeline completes, it checks to see if any previously completed runs should have their **stash** artifacts cleared

## LAB EXERCISE

[Restart Stage](https://s3.amazonaws.com/cloudbees-training-materials/training-pipeline-intermediate/master/selfpaced/labs/Restart_Stage.html)

## Solution

**def** call(body) {

**def** pipelineParams= [:]

body.resolveStrategy = Closure.DELEGATE\_FIRST

body.delegate = pipelineParams

body()

pipeline {

agent none

stages {

stage('Build and Test Java') {

parallel {

stage('java8') {

agent { label 'java8' }

stages {

stage("build8") {

steps {

runLinuxScript(name: "build.sh")

}

post {

success {

stash(name: 'Java 8', includes: 'target/\*\*')

}

}

}

stage('Backend Java 8') {

steps {

unstash 'Java 8'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend') {

steps {

unstash 'Java 8'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 8') {

steps {

unstash 'Java 8'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 8') {

steps {

unstash 'Java 8'

sh './jenkins/test-static.sh'

}

}

}

}

stage('java7') {

agent { label 'java7' }

stages {

stage("build7") {

steps {

runLinuxScript(name: "build.sh")

}

post {

success {

postBuildSuccess(stashName: "Java 7")

}

}

}

stage('Backend Java 7') {

steps {

unstash 'Java 7'

sh './jenkins/test-backend.sh'

}

post {

always {

junit 'target/surefire-reports/\*\*/TEST\*.xml'

}

}

}

stage('Frontend Java 7') {

steps {

unstash 'Java 7'

sh './jenkins/test-frontend.sh'

}

post {

always {

junit 'target/test-results/\*\*/TEST\*.xml'

}

}

}

stage('Performance Java 7') {

steps {

unstash 'Java 7'

sh './jenkins/test-performance.sh'

}

}

stage('Static Java 7') {

steps {

unstash 'Java 7'

sh './jenkins/test-static.sh'

}

}

}

}

}

}

stage('Confirm Deploy') {

when { branch 'master' }

steps {

timeout(time: 3, unit: 'MINUTES') {

input(message: 'Okay to Deploy to Staging?', ok: 'Let\'s Do it!')

}

}

}

stage('Fluffy Deploy') {

agent { label 'java7' }

when { branch 'master' }

steps {

unstash 'Java 7'

sh "./jenkins/deploy.sh ${pipelineParams.deployTo}"

}

}

}

options {

durabilityHint('MAX\_SURVIVABILITY')

preserveStashes(buildCount: 5)

}

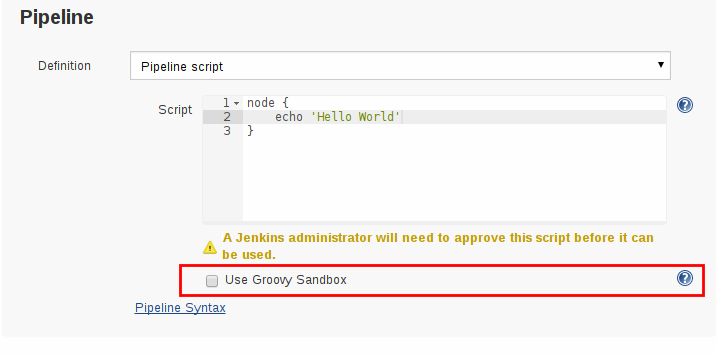
}

}

## WHAT IS THE GROOVY SANDBOX

* A limited execution environment that is enabled by default
* Allows anyone to run a test script without risking damage to the Jenkins environment
  + Not necessary to wait for an administrator to approve the script
  + When it runs, each method call, object construction and field access is checked  
    against a whitelist of approved operations
    - If an unapproved operation is attempted, the script is killed

## HOW TO USE SANDBOX

****

## WHY DOES PIPELINE NEED A SANDBOX?

* The sandbox provides a safe location to test Scripted Pipeline code that  
  has not been thoroughly tested and reviewed
* "Unsafe" Pipeline code includes calls that are not known to be safe
  + The code itself may actually be benign
* The mischief that unsafe code can do includes:
  + Disclosure of information (secrets, proprietary information, or other  
    confidential information being accessed by the Pipeline)
  + Modification/deletion of data in the Jenkins master
* Unsafe code can be inserted in a Pipeline intentionally or accidentally

## WHITELIST

* The whitelist defines each method call, object construction and field access that can be used
  + The Script Security plugin includes a small [default whitelist](https://github.com/jenkinsci/script-security-plugin/blob/master/src/main/resources/org/jenkinsci/plugins/scriptsecurity/sandbox/whitelists/generic-whitelist)
  + Plugins may add operations to that list
  + Administrators may add operations to that list
  + Method signatures can be pre-whitelisted with Groovy either on boot (using init.groovy.d)  
    or in the script console

## ADMINISTRATION OF WHITELIST

* When a script fails because it uses an operation that is not in the whiteliest,  
  that operation is added to an approval queue
  + The administrator can approve the script and it will run
* The administrator is also given a list of pending operation approvals
  + Click **Approve** next to an operation to add it to the whitelist
  + This makes that operation available to all sandboxed scripts on the Jenkins instance

## GUIDELINES FOR WHITELIST

* Consider carefully which operations are white listed
  + Most operations that change the state of persisted objects (such as Jenkins jobs) should not be whitelisted
  + Most **getSomething** methods are harmless
    - Some "getter" methods check specific permissions (using an Access Control List or ACL) whereas scripts are often run by a system pseudo-user to whom all permissions are granted
    - Unconditional whitelisting of such an operation may allow access to resources that should be secured

## EXAMPLE OF UNSAFE GETTER METHOD

* **hudson.model.AbstractItem getParent** obtains the folder or Jenkins root that contains a job
  + Is inherently safe
* The possible follow-up call method **hudson.model.ItemGroup getItems** lists jobs by name within a folder by checking Jobs/Read
  + Should not be whitelisted unconditionally because it enables the user to read at least some information from any jobs in that folder, even those that are protected by the ACL

## SAFE HANDLING OF GETTER METHOD

* Administrator can instead click **Approve assuming permission check** for **getItems**
  + The call is permitted when run as an actual user who is on the ACL
  + The call is forbidden when run as the system user
  + This button is shown only for method calls and constructors
  + Use it only when you know that Jenkins is doing a permission check

## MORE REMARKS ABOUT SCRIPT SECURITY

* **Do not** use Permissive Script Security
  + This makes it trivially easy to root your master
* The Sandbox can be annoying because each script fails when the first item that is  
  not whitelisted fails so it can take several iterations to get all methods, etc. whitelisted
  + Even so, **DO NOT DISABLE THE SANDBOX!**
  + This would make your Jenkins installation very vulnerable to a catastrophic security breach
* Script Security and the Sandbox are most important for Scripted Pipeline
  + Declarative Pipeline can include code that is subject to script security,  
    although it is less common and harder to do

## MORE REMARKS ABOUT SCRIPT SECURITY

* Global Shared Libraries with a locked-down repo can be used  
  to execute unsafe Pipeline code without doing mass whitelisting

## FOR FURTHER READING

* We have only scratched the surface of this topic. Additional information is available:
  + R. Tyler Croy’s [Do not disable the Groovy sandbox](http://unethicalblogger.com/2017/08/03/donut-disable-groovy-sandbox.html) blog is a fun discussion of  
    why sandboxes are important and includes an example script that could  
    destroy your Jenkins instance were it allowed to run.
  + The [Script Security Plugin](https://wiki.jenkins.io/display/JENKINS/Script+Security+Plugin) documentation provides detailed information about  
    the issues and use of sandboxes and other related topics.
  + [Content Security Policy](https://wiki.jenkins.io/display/JENKINS/Configuring+Content+Security+Policy) describes the Content-Security-Policy header and  
    gives guidelines and instructions for relaxing the rules.

## KEEP IT SIMPLE!

* Limit the amount of complex logic embedded in the Pipeline
* Scripted Syntax is NOT a general-purpose programming language

*There’s nothing wrong with using Scripted syntax if what you’re trying to do is really not a good fit for Declarative syntax — but it’s often a sign you’re getting too fancy and should simplify your approach.*

Pipeline code is glue

* Use just enough Scripted Syntax to connect the Pipeline steps and integrate tools
  + Delegate more to agent and reduce the load on masters
* This makes the Pipeline code:
  + Easier to maintain
  + More robust against bugs

## USE COMMAND-LINE TOOLS FOR XML AND JSON PARSING

* Avoid Pipeline XML or JSON parsing using Groovy’s XmlSlurper and JsonSlurper
  + Groovy implementations are complex and very brittle for Pipeline usage
  + XmlSlurper and JsonSlurper carry a high memory and CPU cost in Pipelines
* **xmllint** and **XMLStarlet** are command-line tools offering XML extraction using XPath
* **jq** offers the same functionality for JSON
* These extraction tools may be coupled with **curl** or **wget** to fetch information from an HTTP API

## USE EXTERNAL SCRIPTS AND TOOLS

* Avoid embedding complex logic in the Pipeline itself
* Instead, use external scripts and tools for complex or CPU-expensive processing
  + Offloads work from the master to external executors
  + Allows for easy scale-out of hardware resources
  + Simplifies testing
    - Components can be tested in isolation without the full on-master execution environment

## WHEN TO USE COMMAND-LINE TOOLS

* Processing data
* Communicating interactively with REST APIs
* Parsing/templating larger XML or JSON files
* Nontrivial integration with external APIs
* Simulations and complex calculations
* Business logic

## COMMAND-LINE CLIENTS FOR APIS

* Many software vendors provide easy command-line clients for their tools  
  in various programming languages
  + These are often robust, performant and easy to use
* Use shell or batch steps to integrate these tools, which can be written in any language
  + For a Java client, use a command like:

sh "java -jar client.jar $endPointUrl $inputData"

* Avoid inputs that might contain shell metacharacters. A construction like the following  
  solves this problem
* writeFile file: 'input.json', text: inputData

sh 'java -jar client.jar $endPointUrl input.json'

## REDUCE THE NUMBER OF STEPS IN THE PIPELINE

* Most well-formed Pipelines contain less than 300 steps
* Reducing the number of steps that are called can improve Pipeline and overall Jenkins performance
  + Each call to sh or bat incurs about 200ms of overhead
  + Information about each step run is written to disk, which can be slower for Jenkins to process.
* Other advantages to fewer steps:
  + Simplify the test and debug process
  + Simplify the logic of the Pipeline

## HOW TO REDUCE THE NUMBER OF STEPS IN A PIPELINE

* Consolidate several sequential sh or bat steps into a single, external helper  
  shell script that the Pipeline calls as a single step
  + Version this script and store it in the source code repository
  + This script can be tested independently of the Pipeline itself
* The tradeoff here is that you cannot just read down the Pipeline to quickly  
  see each step that it executes

# Jenkins Administration - Fundamentals (Course-3)

## COURSE SECTIONS

* This course covers the following topics:
  1. Install, set up, and upgrade Jenkins
  2. Explore the "Manage Jenkins" screens
  3. Manage plugins
  4. Configure notifications
  5. Implement distributed builds with Masters, Nodes, and Agents
  6. Manage security on your Jenkins instance
  7. Use folders to segregate builds by project and team
  8. Monitor Jenkins
  9. Back up your Jenkins instance
  10. Use the Jenkins command line interface (CLI) to automate tasks

## SCOPE OF COURSE

* Understand and practice:
  + How to efficiently administer Jenkins
  + Best practices
  + Where to find additional information
  + This course covers open source features available in Jenkins

## AUDIENCE

* DevOps Engineers
* System Administrators
* Build and Release Engineers
* QA Engineers
* Technical Project Managers
* Architects

## PREREQUISITES

* Students should understand the principles of Continuous Integration and Continuous Delivery (CI/CD), Source Code Management, and modern testing theories that are used in this course:
* Students should also understand ancillary technologies that are mentioned in this class:
  + Docker; Git; Java; Apache Maven, Gradle, Apache Ant or npm; Apache Groovy
  + The class has been structured so you can do the exercises even if you are not familiar with these tools but learning them will help you implement your Pipelines when you go back to work.

## REQUIREMENTS AND PLANNING

**MINIMAL REQUIREMENTS: JENKINS**

* Jenkins runs on current versions of:
  + Linux
  + Windows
  + macOS
  + Unix/BSD
* Java 8 or Java 11 is required
  + Jenkins Long Term Support (LTS) runs on Java 11 beginning in Release 2.164.1
  + Java 9 and 10 are not supported

## RUNTIME REQUIREMENTS: JVM

* Java Virtual Machine (JVM) must be tuned:
  + Memory Heap size: -Xms1G -Xmx2G
  + G1 garbage collector for heap > 4GB : -XX:+UseG1GC
* Check your Java Memory documentation for details

## RUNTIME REQUIREMENTS: OPERATING SYSTEM

* Beware of operating system Limits
  + Maximum number of open files
  + Maximum number of forked processes
  + Network tuning (packet size, TCP timeouts)

## JENKINS IS DISK I/O INTENSIVE

* Jenkins writes a lot of files as it processes builds
  + Some of these can be offloaded to external repositories such as Nexus and Artifactory
* Disk usage grows over time
  + Especially when you start to host jobs from people who are not close to you
  + Be sure you add more disks when necessary

## DISK REQUIREMENTS

* No need to waste money on 15000rpm SCSI disks
  + But bigger disk is nice
  + Low latency is the big winner (SSD)
* Can be network mounted
  + NFS
  + SAN

## DISK SPACE

* Artifacts, logs and other files written by builds can be very large
  + Consider external storage such as Nexus or Artifactory for storing  
    these large files off the Jenkins master
  + Retention policy affects the amount of space these consume
    - If you need to retain all (or most) artifacts for a long time, the  
      disk consumption issues are more serious

## DISK SPACE FOR BACKUPS

* Be sure to configure enough disk space for backups
  + Can configure a separate disk on the master just to hold backups
  + Can back up Jenkins files to a local directory then copy those files to an external location
    - Must have enough space on the disk to accommodate the backed up files
    - Recommend creating a separate filesystem (mountpoint) for the backup directory

## EXPANDABLE VOLUME

* “Spanned volume” on Windows
* LVM on Linux
  + This needs the most upfront planning
* ZFS
  + There’s no viable free Solaris flavor
* The Jenkins home directory ($JENKINS\_HOME) should be its own filesystem
  + If it’s too late, symlink is your friend

## INSTALL JENKINS

## DOWNLOAD JENKINS

* Jenkins is distributed on many channels:
  + OS native packages (RPM, DEB…​)
  + WAR file
  + Docker image
  + Cloud templates (AWS, Azure…​)
* Start from the [Jenkins download page](https://jenkins.io/download)

## INSTALL JENKINS: LINUX PACKAGES

* Native packages available for major Linux distributions. The major ones are:
  + RPM for Red Hat family
  + Deb for Debian/Ubuntu
  + Many others: Gentoo, OpenSuse…​
* Pattern with package managers:
  + Add the Jenkins Package Repository
  + Install and start Jenkins
* Example for Red Hat family:

# Add the Jenkins Yum Package Repository

$ wget -O /etc/yum.repos.d/cje.repo "http://<...>/rpm/jenkins.repo"

$ rpm --import "http://<...>/rpm/jenkins.io.key"

# Install it

$ yum install jenkins

# Start it

$ service jenkins start

## MORE ABOUT LINUX NATIVE PACKAGES

* Based on **stand-alone jenkins.war** (with embedded Jetty Application Server)
* Benefits:
  + Creates jenkins user
  + Sets up service scripts (init.d, upstart or systemd)
  + Configuration files follow native conventions
  + Provides log rotation out of the box
* Where are files?
  + Settings in /etc/default/jenkins or /etc/sysconfig/jenkins
  + $JENKINS\_HOME defaults to /var/lib/jenkins

## INSTALLING JENKINS: NATIVE WINDOWS

* Run either:
  + setup.exe
  + jenkins.msi if .NET 2.0 runtime is already available
* Install Jenkins as a Windows service
  + Files are installed into %JENKINS\_HOME%

## RUNNING JENKINS: WAR

* The Web Application Archive (WAR) distribution of Jenkins  
  can be run:
  + as a stand-alone application
  + in a servlet container

## RUNNING JENKINS: WAR AS A STAND-ALONE APP

* Run from the command line
* Uses an embedded application server (Jetty)
* Provides some extra features (restart from the web interface,…​)

$ java ${JAVA\_OPTS} -jar jenkins.war ${JENKINS\_OPTS}

## JENKINS AS A STAND-ALONE APP: COMMAND LINE OPTIONS

* Options configured by startup flags (JENKINS\_OPTS):

**--prefix $PREFIX (default: /)**

Runs Jenkins to include the $PREFIX at the end of the URL

**--httpPort $PORT (default: 8080)**

Jenkins listens on $PORT port.

**--httpListenAddress $HTTP\_HOST (default: 0.0.0.0)**

Binds Jenkins to the IP address represented by $HTTP\_HOST.

**--logfile $LOGFILE**

write to $LOGFILE instead of stdout

## MORE ABOUT COMMAND LINE OPTIONS

* These "options" are actually flags
  + Passed as members of the JENKINS\_OPTS variable to the "java -jar" command line  
    mentioned above
* See [Starting and Accessing Jenkins](https://wiki.jenkins.io/display/JENKINS/Starting+and+Accessing+Jenkins) for a complete list of start-up flags that are available
* Caveat: If you misspell an option, Jenkins ignores it rather than generate an error

## JENKINS AS A STAND-ALONE APP: SIGNAL

* Signal management:
  + Reacts to SIGTERM and SIGINT to initiate proper shutdown
  + When receiving SIGALRM, log file gets reopened; allow Log rotation.

## JENKINS AS A STAND-ALONE APP: EXAMPLE

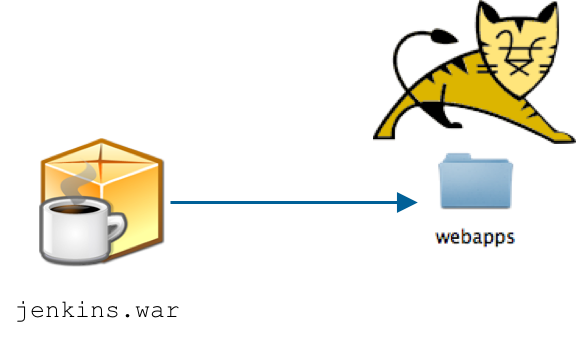
* Configuration Example:

java -jar jenkins.war --httpPort=8081 --prefix=/ci --httpListenAddress=127.0.0.1

* The Jenkins instance will be reachable only on <http://127.0.0.1:8081/ci> .
* More information in the [documentation](https://wiki.jenkins.io/display/JENKINS/Starting+and+Accessing+Jenkins)

## RUNNING JENKINS: ON APPLICATION SERVER

* Deploy the Jenkins WAR file to an existing application server
* May fit in better with existing infrastructures
* Deploy the jenkins.war file in the usual manner

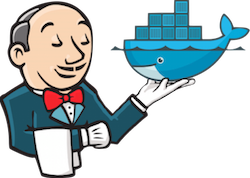


## INSTALLING JENKINS: CONTAINERS

* Jenkins can be installed in a container
  + Docker
  + Kubernetes
* Both container types provide:
  + Weekly and Long Term Support (LTS) Jenkins versions
  + Extensible; can be used to build your own Docker or Kubernetes Image
  + Based on standalone Jenkins WAR

## INSTALLING JENKINS: DOCKER CONTAINER

* Official Docker image on DockerHub: <https://hub.docker.com/r/jenkins/jenkins/>
  + Docker infrastructure **native** integration



## INSTALLING JENKINS: KUBERNETES CONTAINER

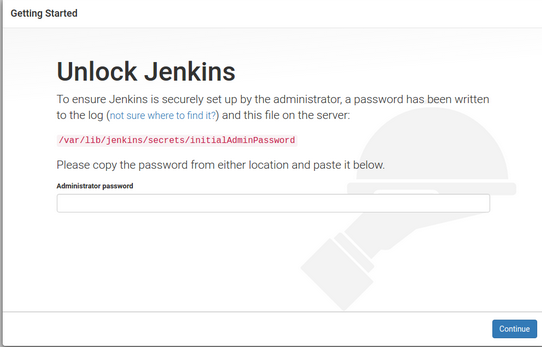
* Official Helm Chart: <https://github.com/kubernetes/charts/tree/master/stable/jenkins>
  + Helm is the package manager used to find, share and use software built for Kubernetes
* Jenkins-X project: [https://jenkins-x.io](https://jenkins-x.io/)
  + Jenkins X is an opinionated platform for providing CI / CD on top of Kubernetes.

## POST-INSTALLATION WIZARD

* The post-installation wizard takes you through a few quick initial configuration steps  
  that are required before you can view the Jenkins Home Page:
  + Bring up a web browser and go to the http://<myServer>:8080 URL
    - Replace <myServer> with the name of the system running Jenkins
  + Follow the instructions on the screen to:
    - Unlock Jenkins
    - Install essential plugins
    - Create first admin user
* See [Setup Wizard](https://jenkins.io/doc/book/installing/#setup-wizard) for more details

## UNLOCK JENKINS

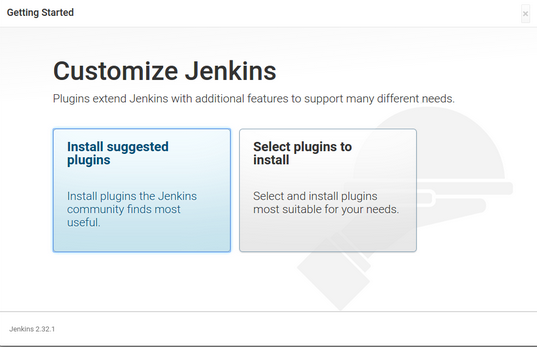
* Follow the instructions on the screen to access the administrative password that was set



* Note that you may need to issue the sudo chmod 0755 name\_of\_dir command  
  to access the file that contains the initial password

## INSTALL ESSENTIAL PLUGINS

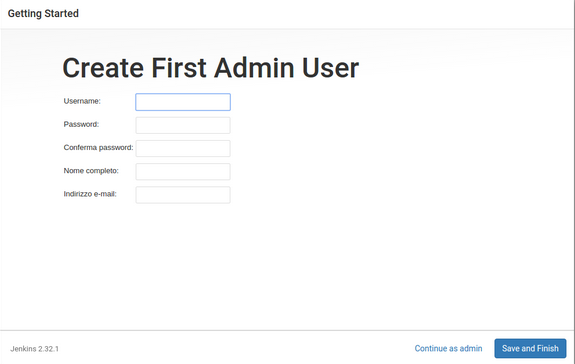
* Many of the features of Jenkins are implemented as plugins
* Before opening Jenkins, you need to install a set of essential plugins that are required:



* You can install additional plugins later

## CREATE FIRST ADMIN USER

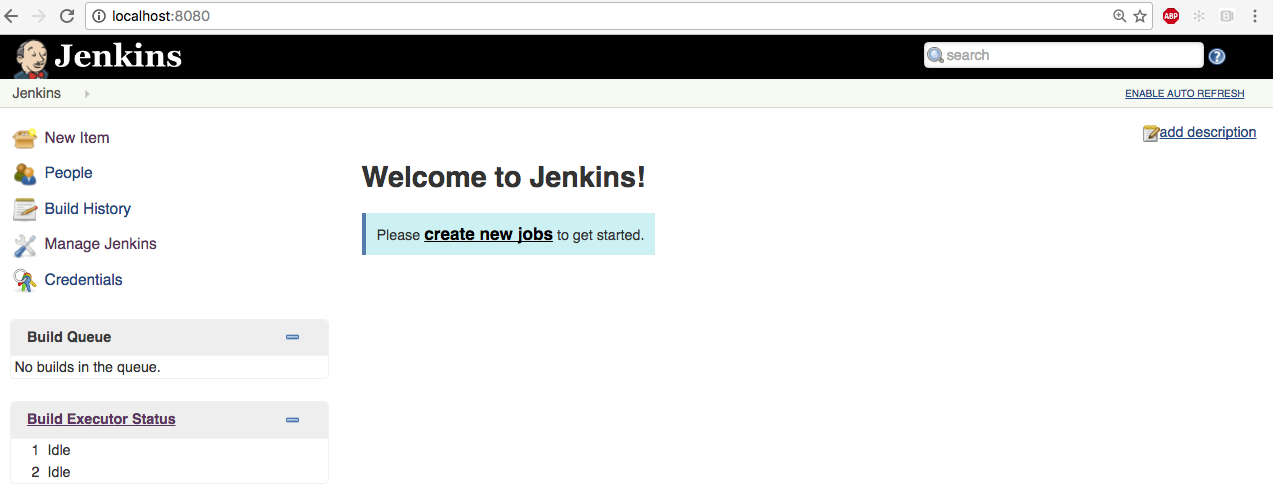
* Create the first admin user by filling out the form on this screen, then click "Save and Finish":



* You can create additional users after Jenkins is running.

## ACCESSING HOMEPAGE

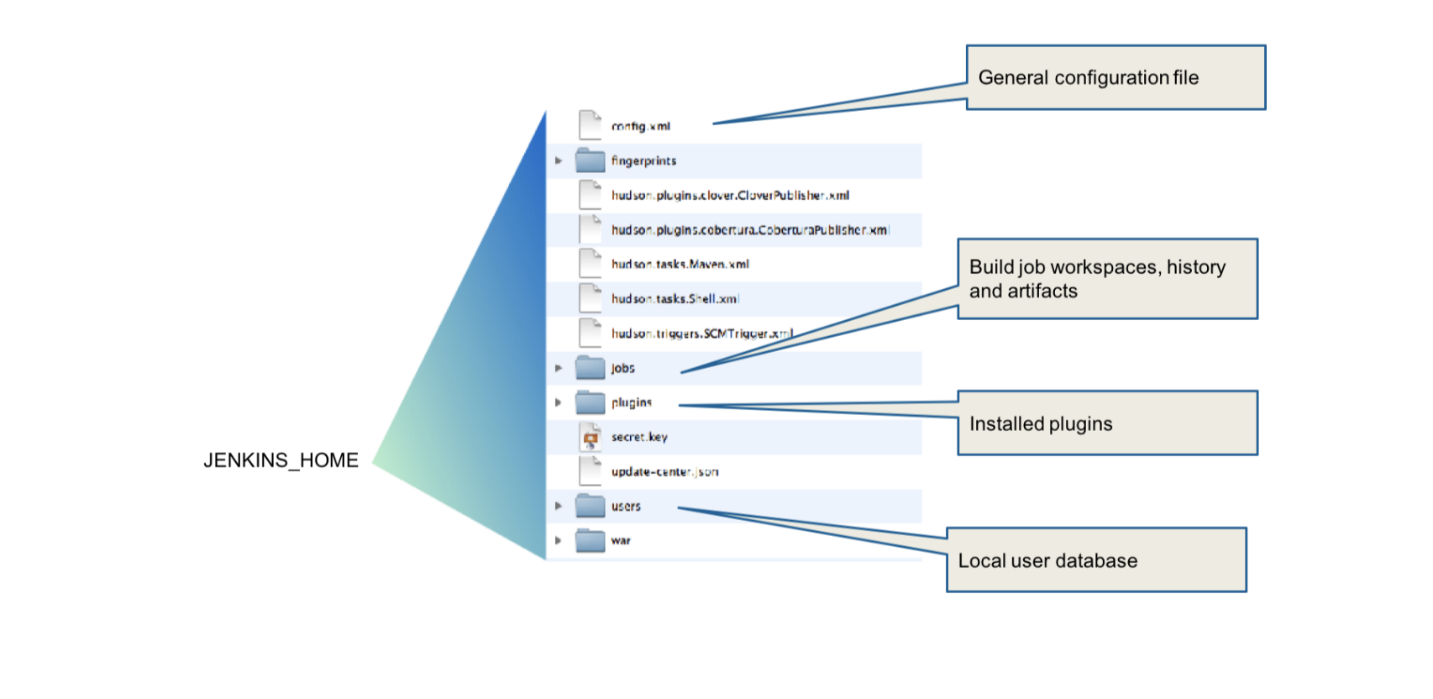
* You can now access the Jenkins Home Page, also called  
  the Jenkins Dashboard or the Classic Web UI



## JENKINS HOME DIRECTORY

* This is where all the important data goes
  + Need to be backed up, etc.
* Can be set in several ways
  + JENKINS\_HOME system property
  + JENKINS\_HOME environment variable
  + Falls back to ~/.jenkins
* Can be verified from system config page

## JENKINS HOME DIRECTORY STRUCTURE



## WHAT DID WE LEARN ?

* Jenkins is distributed in many channels, even Docker
* Try to use the standalone version
* Tune your JVM and operating system and keep an eye on this
* Know your Jenkins Configuration and Jenkins Home locations

## GOING FURTHER

Some recommended readings on this subject:

* [Jenkins Home Page](https://jenkins.io/)
* [Installing and Starting Jenkins](https://jenkins.io/doc/book/installing/)
* Oracle, [Understanding Memory Management](https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html)
* [limits.conf(5) man page](http://linux.die.net/man/5/limits.conf)
* [Docker Repository for Jenkins](https://hub.docker.com/r/jenkins/jenkins/)

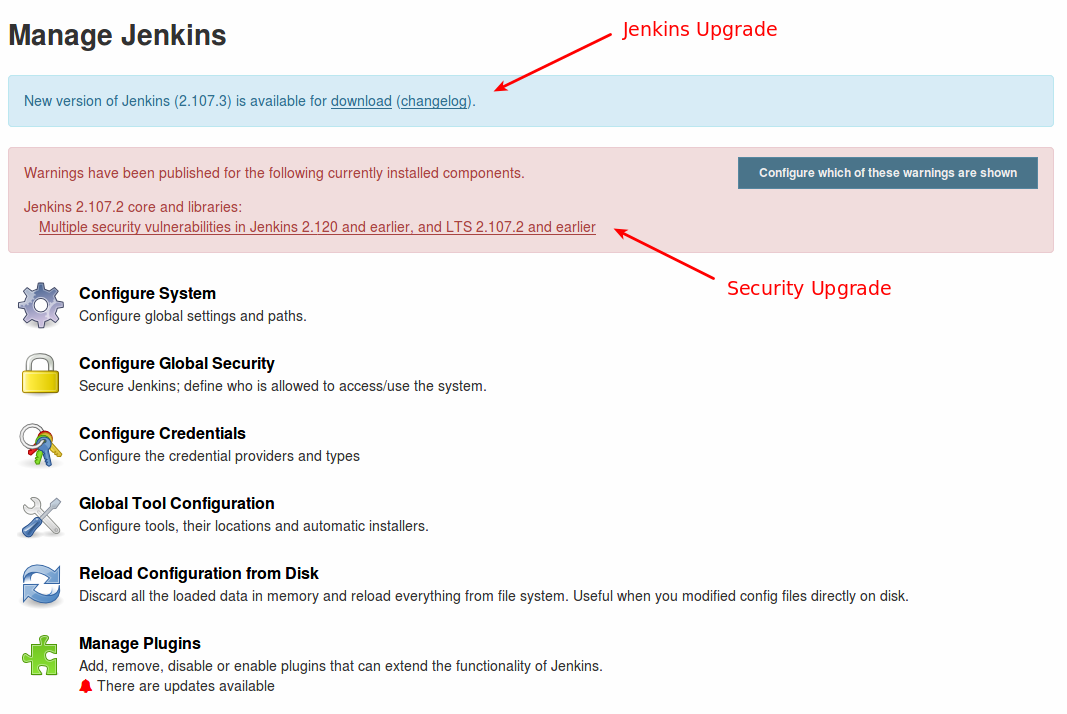
## JENKINS UPGRADES

## ABOUT JENKINS UPGRADES

* Upgrade packages for Jenkins are published regularly
  + Long-term Support (LTS) packages are released every 12 weeks
  + Interim packages are released weekly to make critical bug-fixes  
    and features available quickly to users who need them
* Upgrade to the latest LTS release to stay up-to-date
  + Always check the Release Notes and changelog before you upgrade
* Security releases are published as soon as possible

## UPGRADE ALERTS

* The "Manage Jenkins" page tells you when upgrades are available:



## INSTALL JENKINS UPGRADES

* Right-click on "changelog" in the alert panel to open the changelog for this release
  + Read this to be aware of changes this upgrade may make to your system
* Click on "download" to download the upgrade software
* Use the "Prepare for Shutdown" screen under "Manage Jenkins" to initiate  
  a graceful shutdown of Jenkins
* Replace the jenkins.war file on your system with the new file
  + On Linux systems, this is located in the /usr/share/jenkins directory by default
* Restart Jenkins to apply the upgrade

## MANAGE JENKINS

## OVERVIEW

* From the Jenkins Dashboard, click **Manage Jenkins**
  + This displays the screen from which most administrative tasks are done
* We are going to take a quick tour of this page now
  + We will discuss major sections in detail later

## "MANAGE JENKINS" SCREEN



## NOTIFICATIONS ABOUT UPDATES

* The top of the screen may contain "Monitors" that alert you when a new version of the Jenkins software or a security update is available.
* You should apply all updates as soon as possible, according to your processes (staging environment, validation, backup/restore). We will cover this later.
  + New features, bug fixes and security updates
  + Read the changelog for information about what is in the package
  + Click the link to download and install the system

## SYSTEM CONFIGURATION

* You can configure many aspects of your Jenkins server:
  + Security
  + JDK installations
  + Build tools - Ant and Maven installations
  + Version control tools
  + Email configuration
  + Plugin-specific configuration
  + …​ and a lot more …​

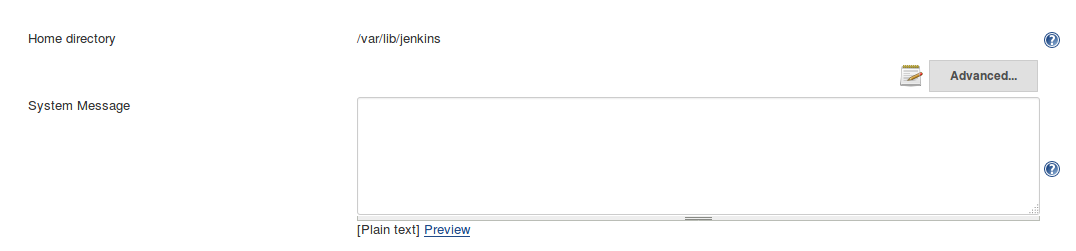
## CONFIGURE SYSTEM

* Use the "Configure System" page to check and modify some basic system parameters



## ABOUT SYSTEM CONFIGURATION

* Sets most of the fundamental tools that Jenkins uses
* Plugins may add sections to this page

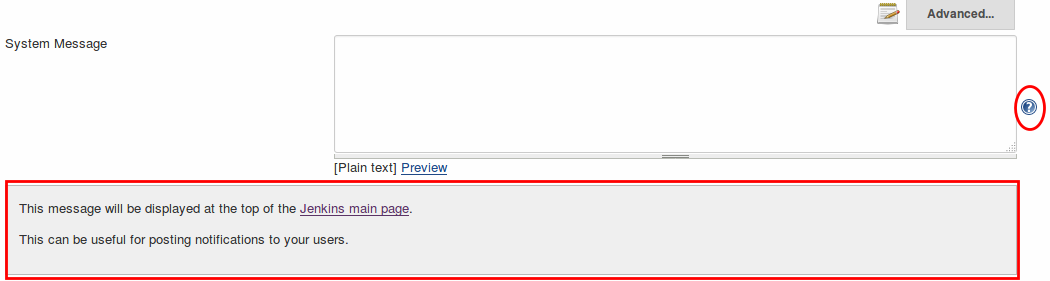


## GENERAL SYSTEM-WIDE CONFIGURATIONS

* **Home Directory** — Display the Jenkins home directory
  + To change this location, modify the value of the JENKINS\_HOME environment variable
* **System Message** — Text that is displayed at the top of your Jenkins home page
  + Can include HTML tags
  + Use this page to display the name of the server and a short description of its purpose
  + May include a pointer to a Wiki page and information about who to contact
  + Display announcements about scheduled down time or other information of interest to all Jenkins users

## INLINE HELP

* Click on the "?" icon on the right of each field for information about this field.
* For example, the help screen for the "System Message" box is:



* Click on the "?" icon again to hide the help text

## SYSTEM CONFIGURATION - # OF EXECUTORS

* What is an ‘Executor’
  + Unit of task execution on a computer
  + Defines upper bound of concurrent build execution
* Typically, you set this to # of CPU cores
  + Unless your builds are highly parallel in itself
* This configures the number of executors on the master
  + Set this to 0 to prevent builds from running on the master
  + We will talk later about configuring executors for builds on remote nodes
* Watch out for memory requirement

## SYSTEM CONFIGURATION - QUIET PERIOD

* Waits for this many seconds before actually starting the build
* Groups adjacent commits
  + This can collapse multiple changes into one build

## SYSTEM CONFIGURATION - SCM TOOLS

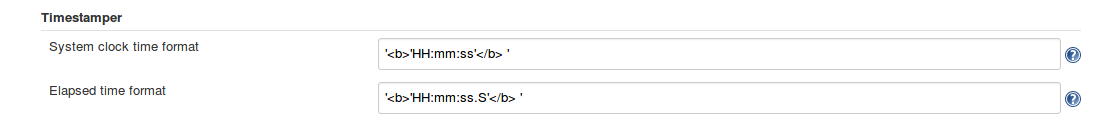
* Git is supported out-of-the-box
* Plug-in support for others:
  + Subversion, Mercurial, Perforce, Team Foundation Server, …
* Most require the tools to be installed separately

## SCM AND AUTHENTICATION

* Most SCM tools support/require authentication when talking to servers
  + How to configure Jenkins depends on SCM
  + Most use Credentials plugin (Git, SVN, Mercurial, Team Foundation Server, …​)
* Perforce
  + Each project needs you to enter credentials separately

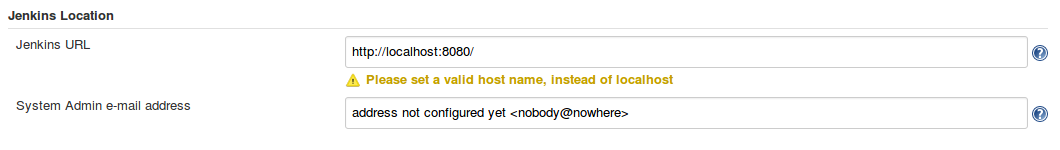
## TIMESTAMPER

* Scroll down the page to the "Timestamper" area
* Use these fields to define the format used to display time stamp information



## JENKINS LOCATION

* Scroll down further to the "Jenkins Location" area



* Replace the localhost:8080 string with the actual URL that people use to access Jenkins
* Define the email address to which email about general Jenkins issues are sent
  + This can be an individual’s email address or an email alias that contains multiple email addresses

## INVEST IN GOOD URL

* If your users cannot see Jenkins, much of the benefit is lost
  + Make the URL easier to remember:
    - Bad: [http://tjek-02.int.rev.example.org](http://tjek-02.int.rev.example.org/)
    - Good: [https://jenkins.example.org](https://jenkins.example.org/)

## USE NAME SERVICE

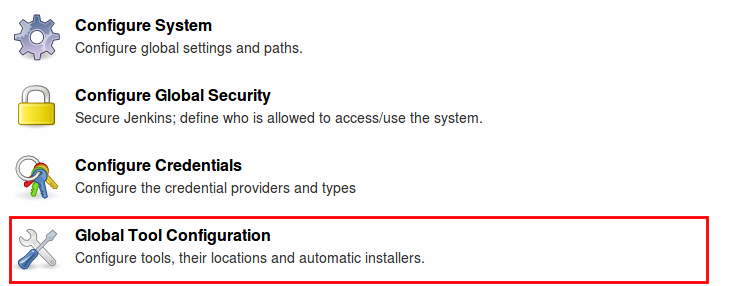
* Get the host alias rather than use the primary machine name
  + If your IT cannot provide this, use dynamic DNS
  + This also makes your service relocatable

## MORE ABOUT SYSTEM CONFIGURATION

* Jenkins is very flexible and powerful
  + Default configurations can be used while you are starting
  + Settings on this page give you fine control of how Jenkins executes
    - As you learn more about Jenkins and your applications, use these settings to fine-tune the behavior of your Jenkins instance
* When you are done modifying this screen, scroll to the bottom and click the Save button to apply your configuration changes

## GLOBAL TOOLS CONFIGURATION

* Use the "Global Tools Configuration" page to configure the tools used for Pipeline development:

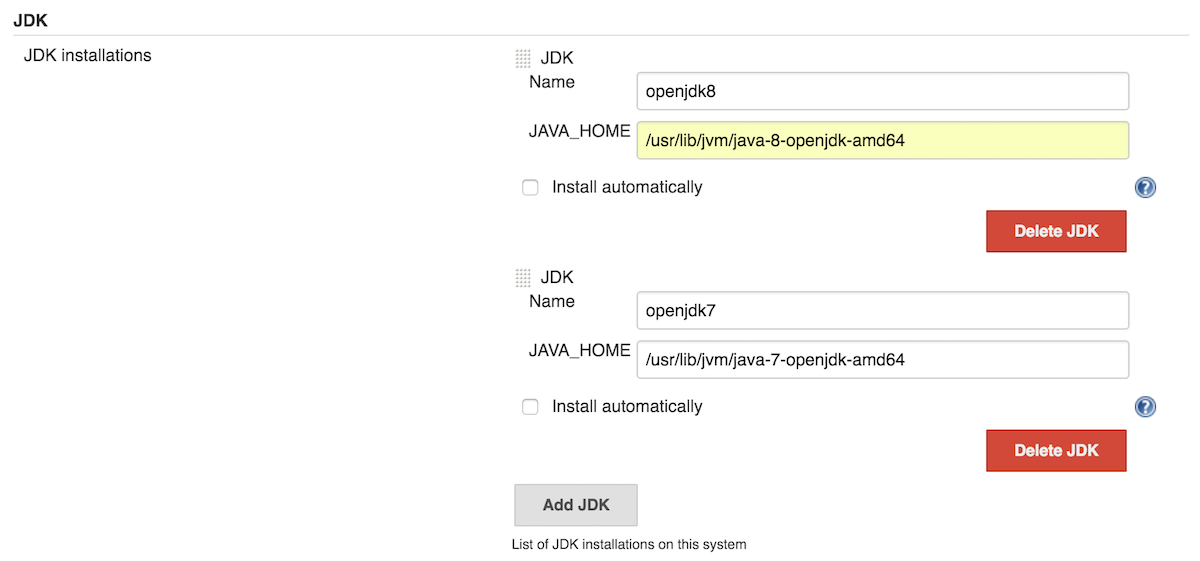


## WHAT TOOLS ARE CONFIGURED?

* Some tools are listed here by default; others only if the appropriate plugins are installed
  + JDK and other languages
  + Build tools: Maven, Gradle, Ant and others
  + SCM: Git, Mecurial and others
  + Containers: Docker and Kubernetes (when installed)
* When you select a tool to install, Jenkins provides boxes for you to supply information that is required to install that tool
* Many support multiple versions of the tool
* Some support auto installation

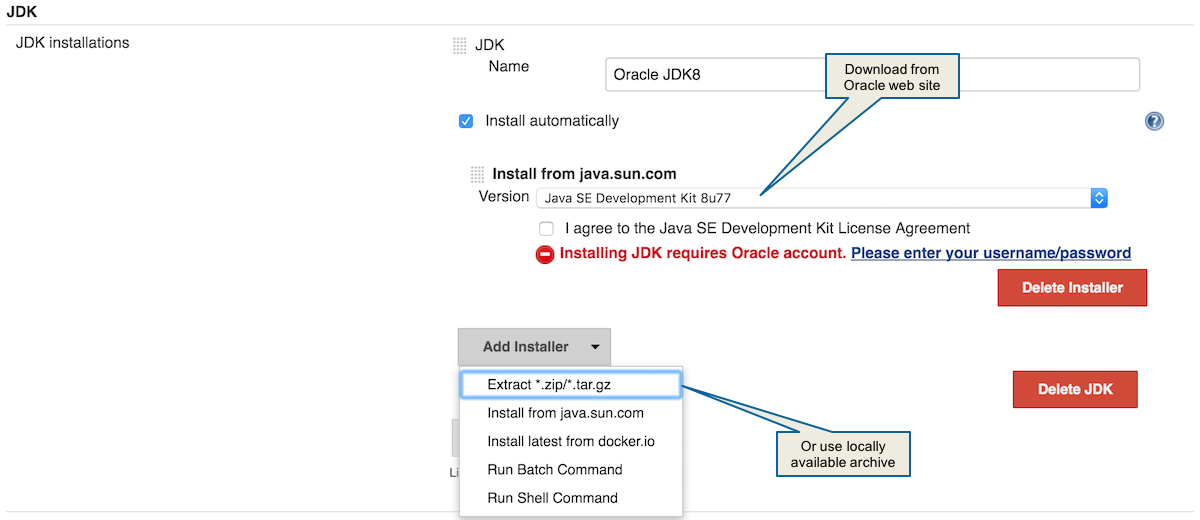
## GLOBAL TOOLS CONFIGURATION - JDK

* Define the JDK versions you need for your projects
* You can define multiple versions for different projects



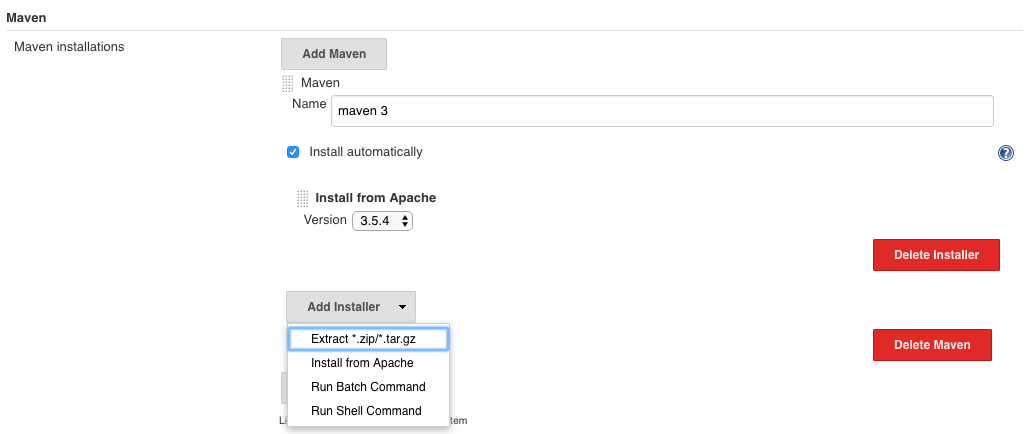
## GLOBAL TOOLS CONFIGURATION - JDK

* Install JDK versions automatically



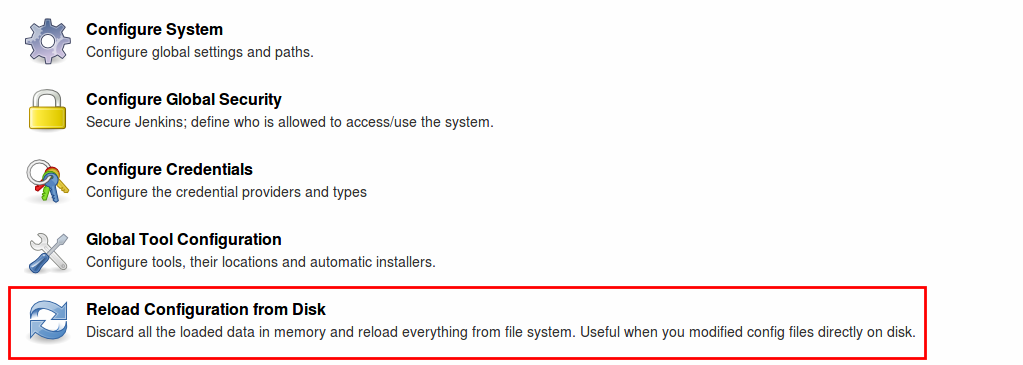
## GLOBAL TOOLS CONFIGURATION - MAVEN

* Install Maven automatically



## RELOAD CONFIGURATION FROM DISK

* Click on "Reload Configuration from Disk" to refresh the Jenkins configuration files and directory structure without restarting Jenkins



* Use this when you modify Jenkins or its environment outside of the UI
  + For example, when you edit config files from the command line or move jobs between folders using the command line
* Not necessary when you modify the configuration from the UI or restart Jenkins after modifying the configuration

## SECURITY

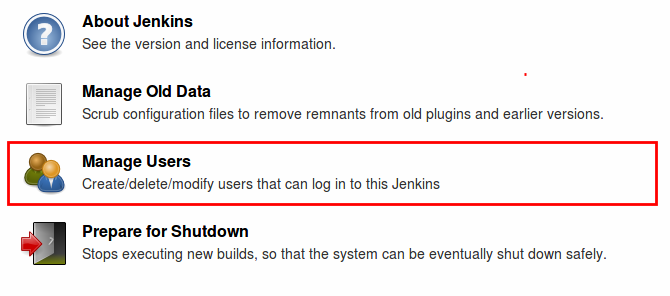
## CONFIGURE GLOBAL SECURITY

* The "Configure Global Security" screen is used to:
  + Define how users are authenticated and what they are authorized to do
  + Control other security settings



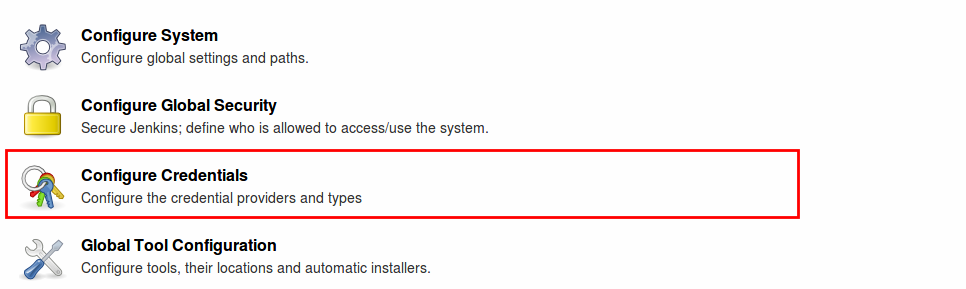
## MANAGE USERS SCREEN

* The "Manage Users" screen is used to add user to the Jenkins user database and lists all users who are in that database



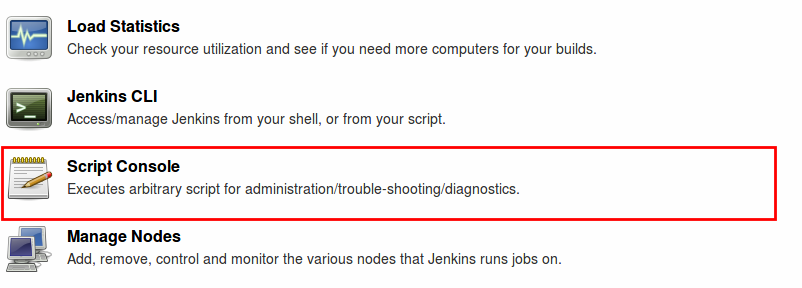
## CONFIGURE CREDENTIALS

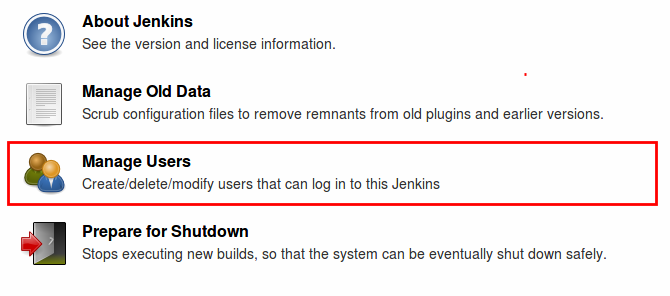
* Jenkins credentials control access to third-party sites and applications such as artifact repositories and cloud-based storage systems.
* Use the "Configure Credentials" screen to manage credentials:



* We will discuss credentials at more detail later

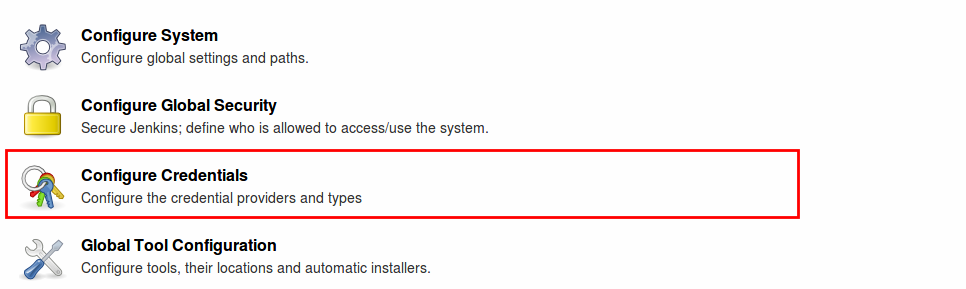
## SCRIPT CONSOLE

* The Script Console allows you to type in and execute an arbitrary Groovy script on the server.
* Apache Groovy is the foundation of the DSL used for Jenkins Pipelines
* 
* This is useful for troubleshooting and diagnostics
* Scripts execute in a Groovy Sandbox that limits the internal APIs that are accessible
* With the Script Security plugin, administrators can use these results to manage which unsafe methods (if any) are allowed in the Jenkins environment
* **Manage Users Screen**
* The "Manage Users" screen is used to add user to the Jenkins user database and lists all users who are in that database



## CONFIGURE CREDENTIALS

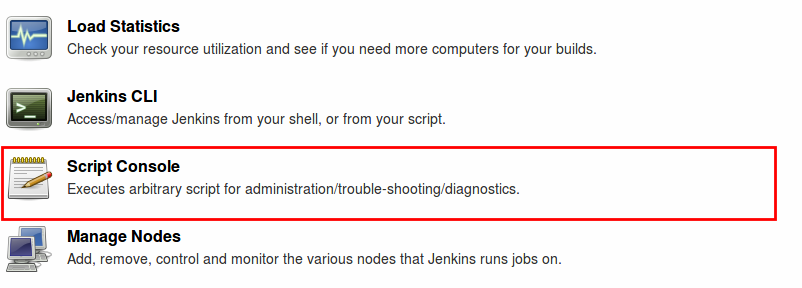
* Jenkins credentials control access to third-party sites and applications such as artifact repositories and cloud-based storage systems.
* Use the "Configure Credentials" screen to manage credentials:



* We will discuss credentials at more detail later

## SCRIPT CONSOLE

* The Script Console allows you to type in and execute an arbitrary Groovy script on the server.
  + Apache Groovy is the foundation of the DSL used for Jenkins Pipelines



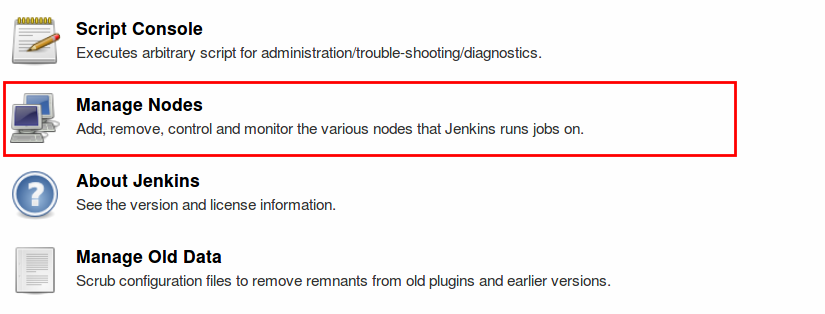
* This is useful for troubleshooting and diagnostics
* Scripts execute in a Groovy Sandbox that limits the internal APIs that are accessible
* With the Script Security plugin, administrators can use these results to manage which unsafe methods (if any) are allowed in the Jenkins environment

## PROVIDE BUILD ENVIRONMENTS

* Administrators must configure build environments that Pipeline developers can use.
* Two main elements:
  + Configure the tools that Pipeline developers can use
  + Configure the nodes and agents on which Pipelines execute

## MANAGE NODES

* Use the "Manage Nodes" screen to create, configure and view nodes:

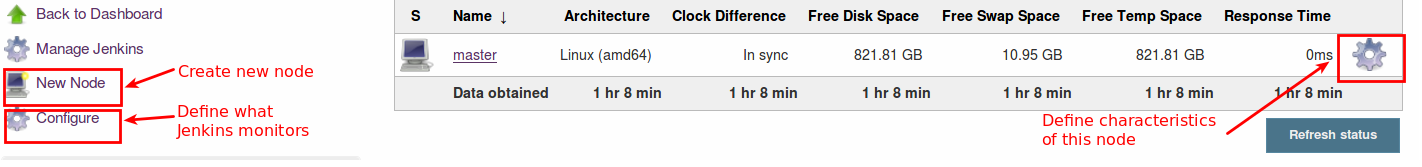


## RECAP OF NODES, AGENTS AND EXECUTORS

* The "Jenkins - Fundamentals" class discusses these terms. To recap:
  + A **node** is a server where Jenkins runs jobs on **executors**
  + The **agent** is the tool that manages the **executors** on a remote **node**, on behalf of Jenkins.
  + The Jenkins master also runs on a **node**.

## CREATE, CONFIGURE AND VIEW NODES

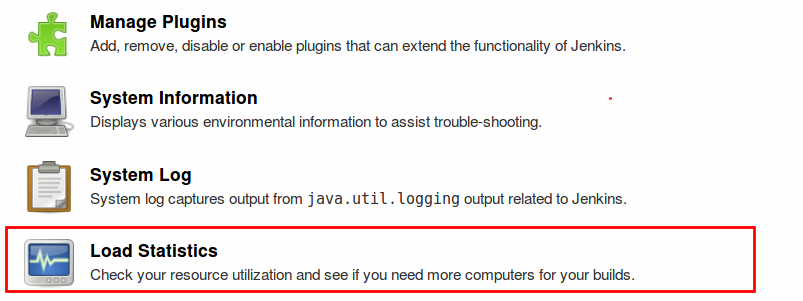
* The "Manage Nodes" screen opens to display information about nodes that are currently configured:



* Initially, this just shows the node where the Jenkins master runs.
* Click "New Node" in the left frame to create a new node
* Click the flywheel to the right of the node line to pop a screen where you can define the number of executors and other characteristics of this node
* Click the flywheel in the left frame to configure the items Jenkins monitors for all nodes it runs

## MONITORING NODE USAGE

* Use the "Load Statistics" page to monitor node utilization for your Jenkins instance:



## NODE USAGE INFORMATION

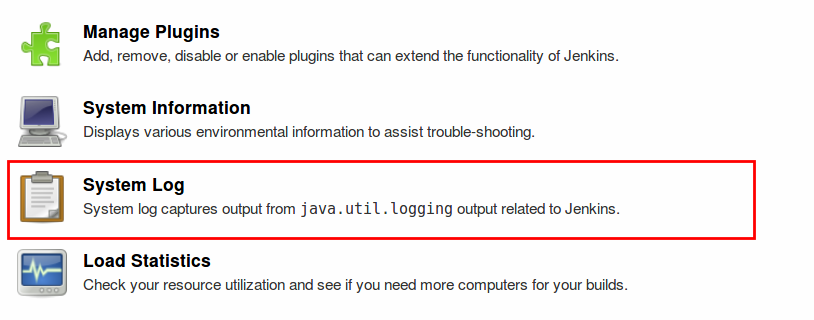
* The "Load Statistics" reports the following information about the nodes configured on your Jenkins instance:
  + Number of online executors
  + Number of busy executors
  + Number of available executors
  + Queue length (number of jobs that are waiting for an available executor)
* Use this information to determine when you need to add more nodes and/or executors to your instance to improve throughput

## INFORMATION PAGES

* Some of the "Manage Jenkins" pages provide reference information that will be useful as you administer your Jenkins cluster:
  + System Log
  + About Jenkins
  + System Information
  + Jenkins CLI

## SYSTEM LOG

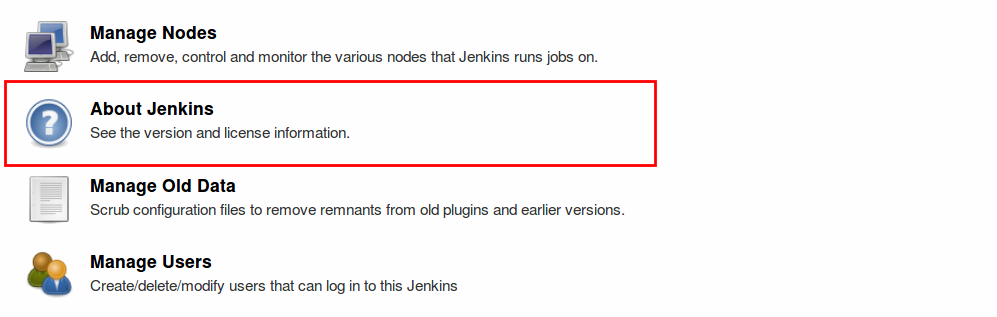
* The "System Log" page includes a link to the log about activities on the Jenkins instance itself:



* Note that this information is independent of the build logs that are presented for each job or Pipeline that runs

## ABOUT JENKINS

* The "About Jenkins" page shows the current release of Jenkins on your system plus information about licenses for all components:

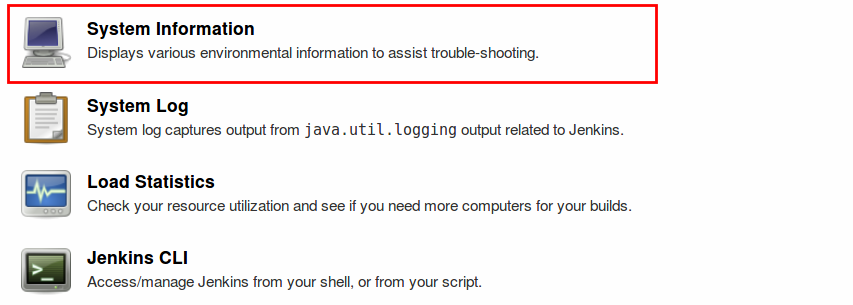


## ABOUT JENKINS INFORMATION

* Release and version of Jenkins that is running
* List of all third-party libraries used for this release of Jenkins
  + Links to licensing details about each library
* List of installed plugins
  + Includes a link that shows all third-party dependencies for each plugin
  + That list includes a link to licensing details about each library

## SYSTEM INFORMATION

* The "System Information" page provides detailed information what is available on this Jenkins instance:

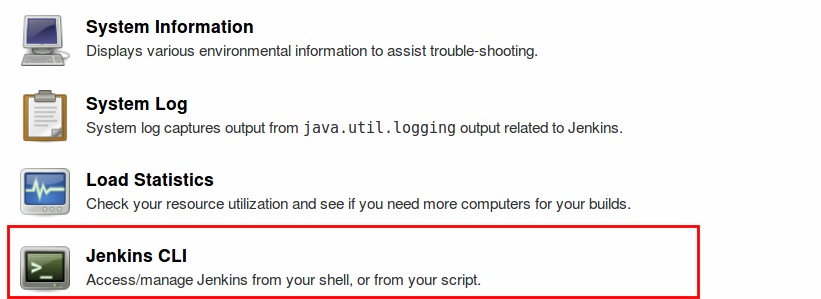


## WHAT IS ON SYSTEM INFORMATION PAGE

* This page contains the following lists:
  + **System Properties** that can be used as arguments to the command line used to start Jenkins
  + **Environment Variables** with current values
  + List of **Plugins** installed on the system

## JENKINS CLI

* The Jenkins CLI page lists supported CLI commands:



## WHAT IS JENKINS CLI

* The Jenkins Command Line Interface (CLI) provides commands that perform functions that are usually executed using the UI
* These can be used in scripts that automate administrative tasks
* Can be accessed in either of the following ways:
  + Over SSH
  + With the Jenkins CLI client, which is a .jar file distributed with Jenkins
* See [Command Line Interface](https://jenkins.io/doc/book/managing/cli/) in the Jenkins Handbook for more information

## MANAGEMENT: SERVICE LIFECYCLE

Managing Options used to Start/Stop/Reload Jenkins service

* Reload Configuration From Disk:
  + Read configuration again from $JENKINS\_HOME without Jenkins restart
* Prepare for shutdown:
  + Prevents new jobs from being started
  + Waits for running jobs to complete (clean job termination)

## JENKINS UPGRADE

* Management page provides Jenkins Upgrade management
  + With standalone server: 1-click upgrade
  + Without: direct link to latest WAR
* "Manage Old Datas" helps to take care of dropped configurations items after an upgrade

## WHAT DID WE LEARN ?

* Jenkins has a dedicated Management page with many sections, providing:
  + System, Tool configurations
  + Security options
  + Metrics, Logs, Configuration Item overview
  + Lifecycle Management utilities
* Jenkins can be managed with Apache Groovy Code
  + Helps integrating with Configuration Management
  + Automate large scale Jenkins deployment

## LAB EXERCISE

[Configuring Global Tools](https://s3.amazonaws.com/cloudbees-training-materials/training-admin-fundamentals/master/selfpaced/labs/Configuring_Global_Tools.html)

## PLUGINS

## JENKINS PLUGINS

* Jenkins uses plugins to provide much of its functionality
  + Many "classical" Jenkins functions are implemented as plugins
  + More than 1700 plugins are available
* This "modular" architecture means that your Jenkins installation includes the features and functionality you need without being bloated with functionality you do not need
* Managing plugins is a critical part of managing Jenkins
  + Many features are implemented with multiple plugins
  + Many plugins have dependencies on other plugins

## FEATURES IMPLEMENTED WITH PLUGINS

* Source code management tools
* Build Tools
* Reporting tools
  + Code coverage, static code analysis
* Online source code browsers
* Issue tracker
* Notification tools
* Views and UI customizations
* Distributed builds

## WHAT ARE PLUGINS ?

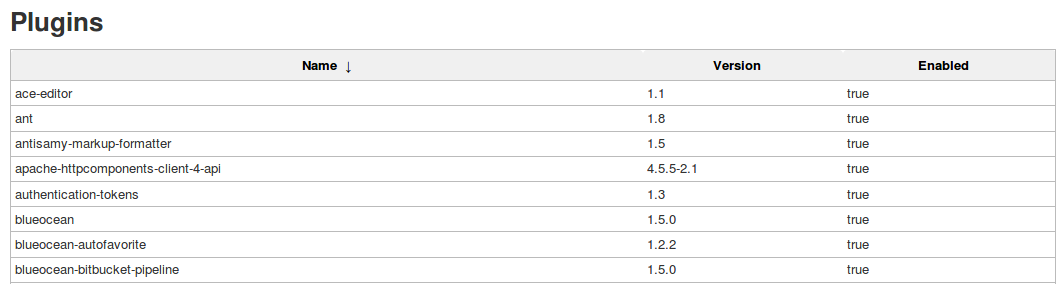
* A plugin is a JAR file with some special conventions (e.g. no web.xml)
* Contained in a file with an hpi or jpi extension
* Stored in the ${JENKINS\_HOME}/plugins directory
  + unless you use --pluginroot to change where the binary of the plugins are "exploded"
* May have dependencies (mandatory or optional)
* Plugins are versioned artifacts that can be upgraded
  + If necessary, they can be downgraded but this is not generally recommended

## SUGGESTED PLUGINS

* The Installation Wizard provides a list of "Suggested Plugins" that everyone should install
  + These are plugins that provide commonly-used functionality
  + Are widely used, well-maintained, and have been examined for security issues
* This list is a feature of the Installation Wizard
  + When you upgrade to later Jenkins releases, it is up to you to identify new plugins that you may want to install

## LIST OF INSTALLED PLUGINS

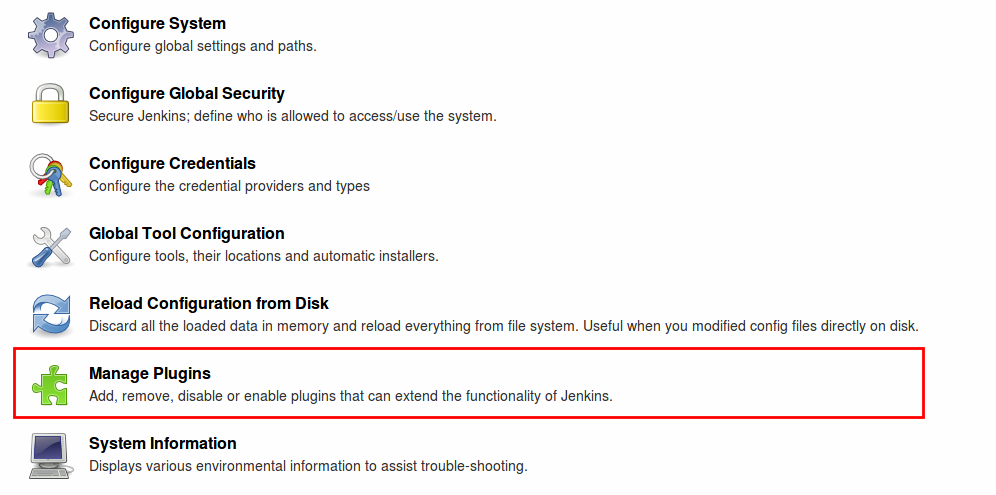
* Click on **Manage Jenkins → System Information** to see a list of installed plugins; for example:



* A list of installed plugins is also available from the **Manage Plugins** page we will discuss next

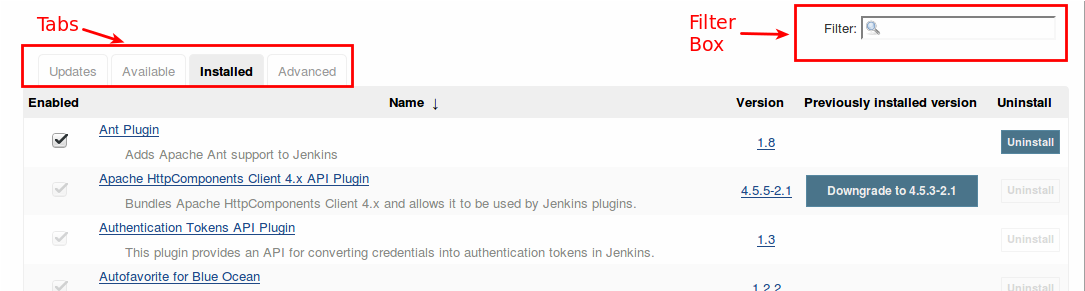
## MANAGE PLUGINS

* Use the "Manage Plugins" page to add, remove, update, enable or disable plugins:



## MANAGE PLUGINS — DETAILS

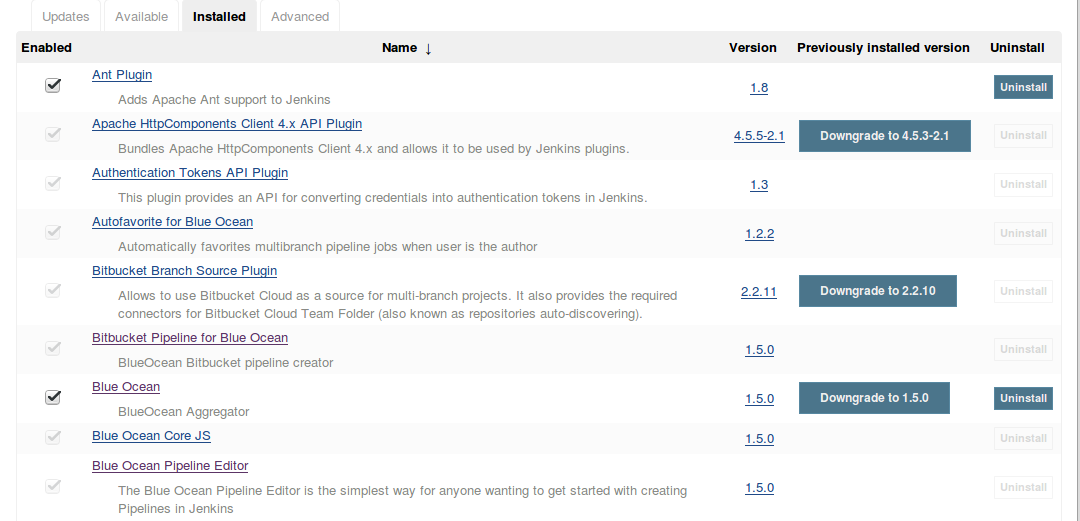
* Tabs at the top of the page define what is displayed:



* Use the "Filter Box" to search for specific plugins

## MANAGE PLUGINS — INSTALLED TAB

* The "Installed" tab lists the plugins that are installed on this instance:



## INSTALLED TAB — COLUMN HEADERS

* **Name** — Name of the plugin with a brief description
  + Click on the **Name** column header to toggle between alphabetical order and reversed alphabetical order
  + Right-click on the plugin name to view the official documentation for the plugin
* **Version** — Version of each installed plugin
* **Previously Installed Version** — Links used to "roll back" a plugin to the previously installed version
  + Do not try to roll back a plugin unless absolutely necessary
  + Additional cleanup is required after a rollback
* **Enabled** — Check mark indicates that the plugin is enabled
  + Greyed-out check marks indicate plugins that are required by other plugins
* **Uninstall** — Links used to uninstall plugins
  + Lines with no "Uninstall" button are for plugins that are required by other plugins

## UNINSTALLING A PLUGIN

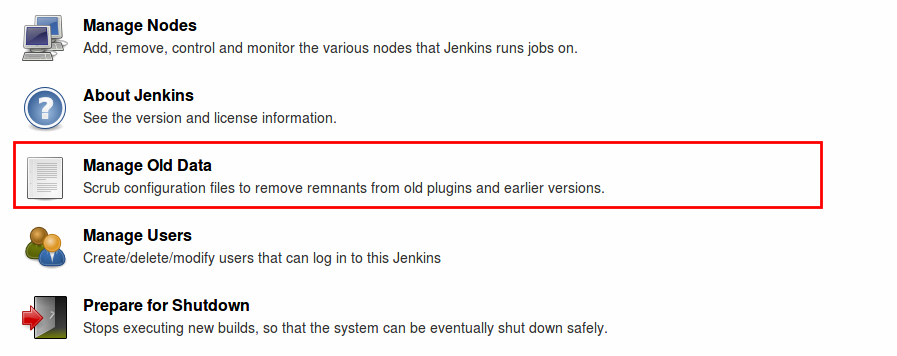
* Removes the plugin binary (jpi or hpi file) from the disk
  + Plugin continues to function until you restart Jenkins
  + After a restart, plugin does not appear in the UI and all extensions it contributed disappear
* Does not remove the configuration that the plugin may have created
  + If the plugin contributed extensions to any jobs/agents/views/builds/etc, Jenkins reports unrecognized fragments in configuration files then ignores such fragments
  + Until you remove those configuration files, you can re-install the plugin and restart Jenkins to restore the configurations
  + Use the "Manage Old Data" screen to scrap configurations left by an uninstalled plugin

## DISABLING A PLUGIN

* Disabling a plugin is a softer way to retire a plugin
  + The extension for the plugin binary is changed to jpi.disabled or hpi.disabled
  + Jenkins does not start the plugin
  + Extensions contributed by this plugin are not visible
  + The plugin still appears in the "Installed" list
    - The box under the **Enabled** column is not checked
* Fragments contributed to configuration files by a disabled plugin are handled like those for uninstalled plugins
* Reenable the plugin by checking the box in the **Enabled** column
  + You may need to restart the instance to make the plugin operational

## MANAGE OLD DATA

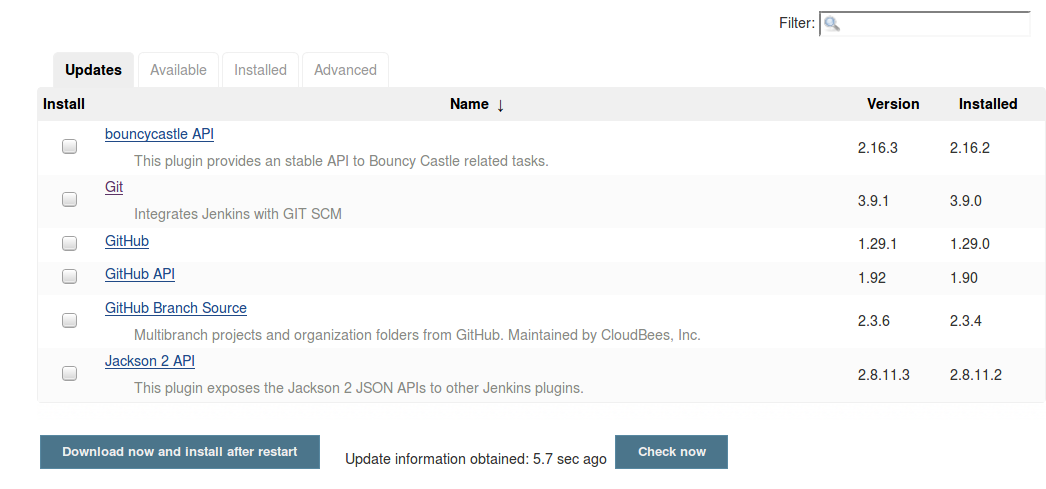
* Use the "Manage Old Data" page to clean up configurations left by uninstalled and disabled plugins:



* The page lists any "old data" on your Jenkins instance and discusses they types of old data and how to handle each

## UPDATES TAB

* The Updates tab lists installed plugins for which an update is available:



* Click the box under "Install" for each update to be installed
* Then click the "Download now and install after restart" button

## UPDATE PLUGINS REGULARLY

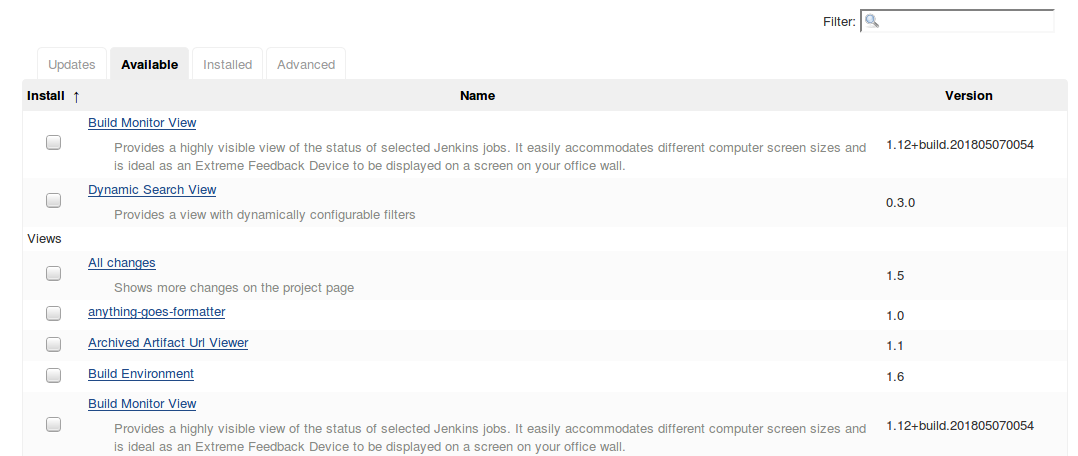
* Many plugins are updated frequently
  + Enhanced functionality, bug fixes, security fixes
* These should be applied as soon as is feasible
  + Always read the changelog information about new plugins before installing
  + Use a staging environment to validate updated plugins before they are applied to the production environment
  + Take a backup of your instance before updating a plugin
* Many updates take effect only after Jenkins is restarted

## JENKINS UPDATE CENTER

* The full set of open source plugins are available at the [Plugins Center](https://plugins.jenkins.io/)
  + Also called the "Update Center"
* When you download a plugin:
  + Jenkins downloads the plugin index from the configured Update Center
  + Jenkins filters the index by the core version
  + If you install the plugin, the proper binary is downloaded

## AVAILABLE TAB

* The "Available" tab lists plugins that are available for installation
* Click the "Install" column header to determine whether the plugins are listed alphabetically or sorted by category:



## WHAT PLUGINS ARE LISTED AS AVAILABLE?

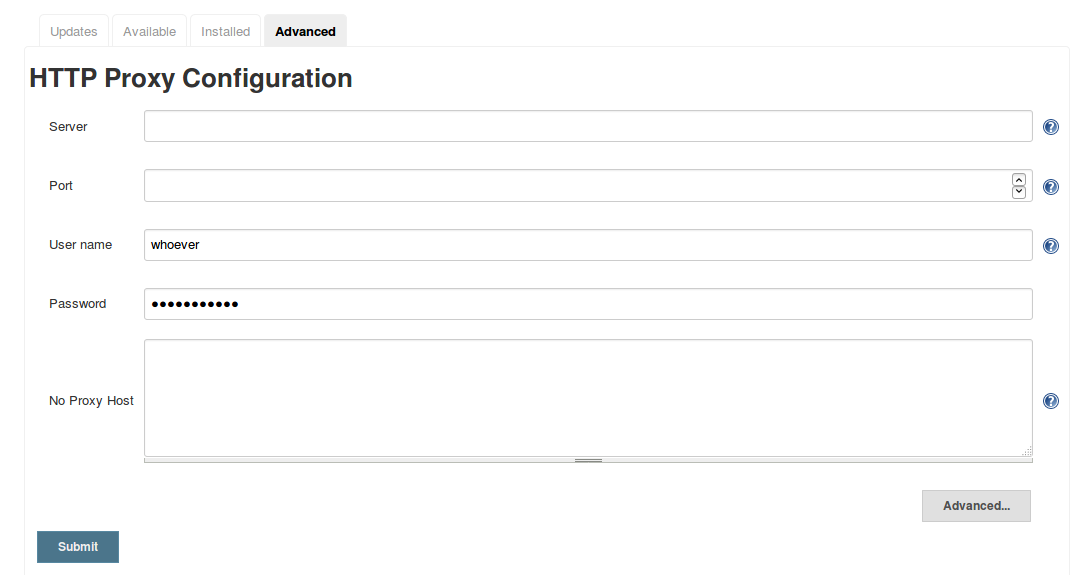
* The "Available" tab contains a filtered list of links to the Update Center
* This list is NOT curated
  + Inclusion on this list means only that the plugin can be installed
  + Look for the number of installations and how frequently and recently updates appear to evaluate the reliability of each plugin before you download it.
* The list of available plugins is updated for each Jenkins release but is not updated between releases

## INSTALL NEW PLUGIN

* To illustrate how to install a new plugin, we are going to install the beer plugin
  + This is a harmless plugin that adds a page on the Jenkins root to print some jokes about beer
  + It has no dependencies and is lightweight.
* You MUST have Internet access (HTTP proxy supported) to download a new plugin from this page

## CONFIGURE HTTP PROXY ACCESS

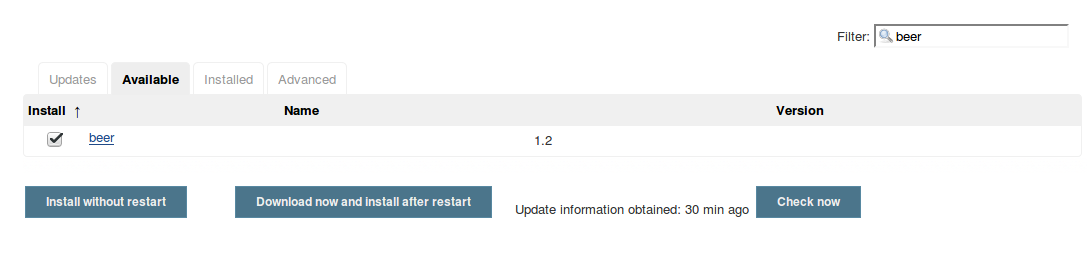
* If you are running under an HTTP proxy, select the **Advanced** tab to configure HTTP proxy access:



* Click the **Advanced** button on the bottom right to enable the **Validate Proxy** configuration fields before clicking "Submit" to submit the configuration

## INSTALL BEER PLUGIN

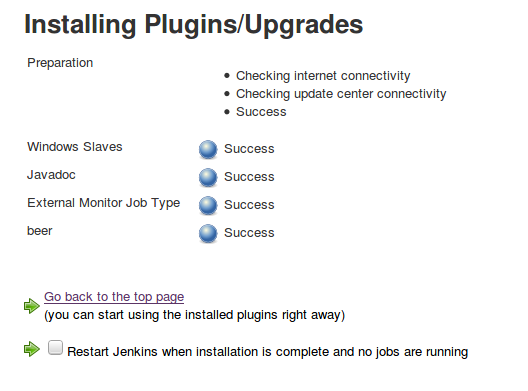
* Type "beer" in the filter box to locate the plugin:



* Click the box in the "Install" column
* Click "Install without restart"

## INSTALLING PLUGINS/UPGRADES SCREEN

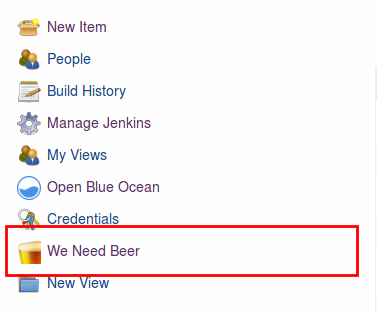
* When you click "Install without restart", the following screen is displayed:



* Note the checkbox "Restart Jenkins when no job running"
* When all components show "Success", you are done

## RUN THE PLUGIN

* Return to the Jenkins dashboard
* Note the new item in the left frame:



* Click on "We Need Beer" and enjoy!

## LAST REMARKS ABOUT PLUGINS

* You can install plugins manually, without using the "Manage Plugins" UI; this is not covered here
* Plugins may have dependencies — sometimes a lot of them
  + The documentation for each plugin lists all dependencies
  + The plugin manager installs all required dependencies
  + You must manually install optional dependencies if you want them
* Some plugins require a Jenkins restart before they can be run.
  + Plan accordingly when installing plugins in a production environment

## LAB EXERCISE

[Installing necessary plugins](https://s3.amazonaws.com/cloudbees-training-materials/training-admin-fundamentals/master/selfpaced/labs/Installing_necessary_plugins.html)

## CONFIGURE NOTIFICATIONS

## NOTIFICATIONS FOR CI/CD

* Primary feedback mechanism
* Can be delivered through email, Slack, etc
* Pipeline code defines the content of notifications for that build
* Administrator must configure and manage the notifications

## PREPARE FOR NOTIFICATIONS

* Install required plugin(s)
* Configure globally in Jenkins Management:
  + Notification Service (Email Extension, Slack, etc)
  + Credentials (generally an API token)
  + Other provider-specific options (rooms, team domains)

## DEFINE A NOTIFICATION

* Per Project, specify:
  + When do you want to trigger a message ?
    - Status change, warning, other event..
  + Recipients for the messages ?
    - Can depend on trigger (notify only the developer who broke the build)
  + What do you want to say ?
    - This is about the content of the message.
    - Use templates and environment variables to have the "right" information; for example, you can use the Jenkins BUILD\_NUMBER environment variable to include the build number in the subject and body of the message; if you are using Git as your SCM, you can use the GIT\_COMMIT environment variable to specify the Git commit

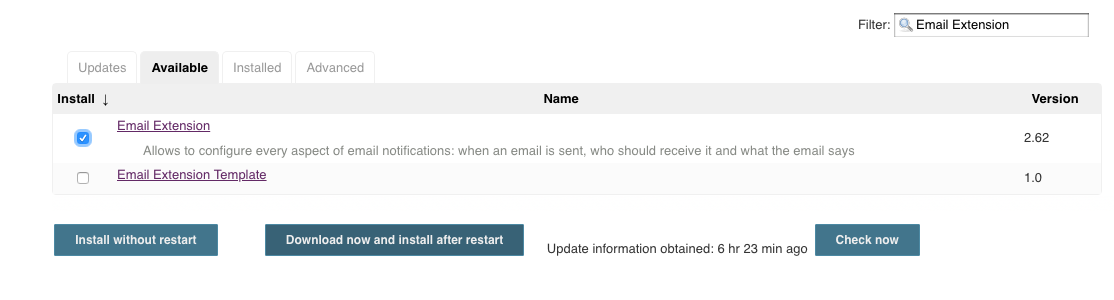
## BUILT-IN EMAIL NOTIFICATIONS

* Main configuration is about SMTP (sending) server
  + Hostname, port, encryption, authentication, SPAM configuration
* Built-in Jenkins Mailer:
  + 1. Every failed build triggers a new e-mail
    - Never fails a build
  + 2. A successful build after a failed (or unstable) build triggers a new e-mail
    - Crisis is over
  + 3. An unstable build after a successful build triggers a new e-mail
    - A wild regression appears
  + 4. Unless configured, every unstable build triggers a new e-mail
    - Regression still there
* Project configuration level: assign recipients to the whole project

## EMAIL-EXT PLUGIN

* E-mail Ext plugin extends the built-in Jenkins Mailer capability
* Better customization for:
  + Triggers add more granularity: 2nd failure, aborted build, Not built, Test Improvement…​
  + Content has many more templates and variables. Can attach scripts.
  + Recipients can change depending on notifications. Managed at both global and jobs levels.

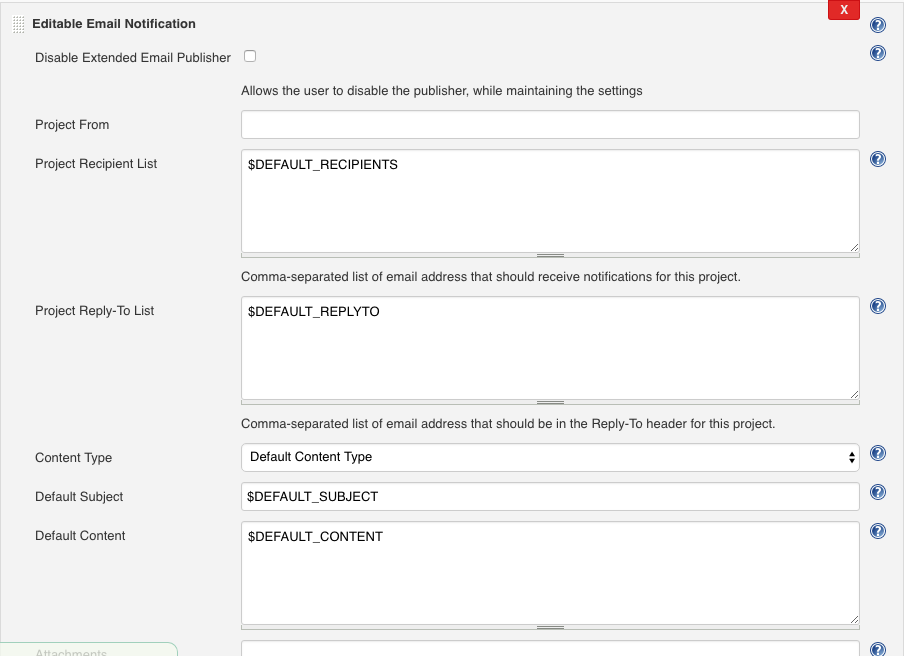
## INSTALLING EMAIL-EXT PLUGIN



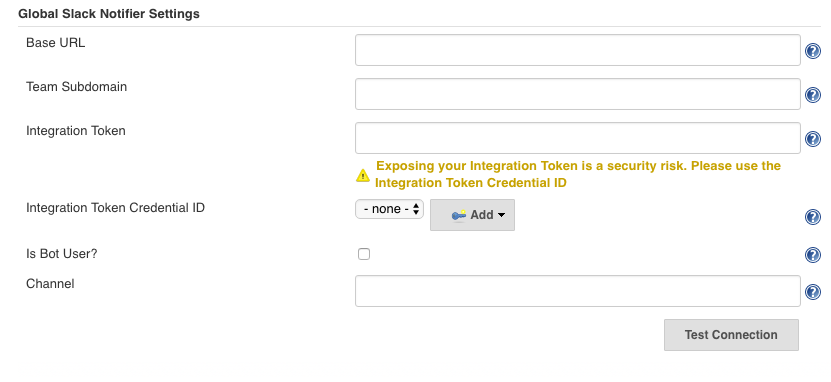
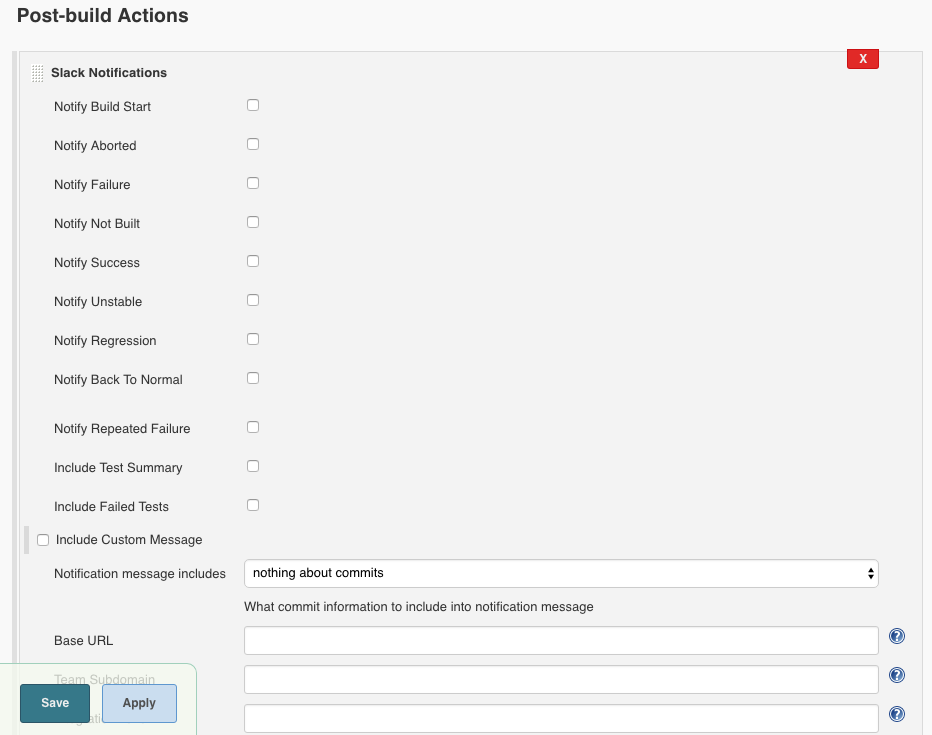
## 1.5CONFIGURING EMAIL-EXT GLOBALLY



**CONFIGURE EMAIL-EXT PER PROJECT**



## SLACK NOTIFICATION PLUGIN

* Slack Notification plugin allows to post build notifications to a Slack channel
* Better customization for:
  + Notifications can be be configured to be sent for any build status:
    - Start, Aborted, Failure, Unstable…
  + Custom content message can be included in the notifications:
    - Jenkins Environment Variables, Commit list, Authors
  + Notifications can be sent via bot users
* Managed at both global and job levels
* **INSTALLING SLACK NOTIFICATION PLUGIN**
* 
* **CONFIGURING SLACK NOTIFICATIONS - GLOBAL**
* 
* **CONFIGURING SLACK NOTIFICATIONS - PER PROJECT**
* 

## WHAT DID WE LEARN ?

* Jenkins allows you to send notifications
* Configure the appropriate plugins for how you receive notifications
  + Emails, Slack etc
* Goals of notifying:
  + Notify Developers when a Build Fails
  + Failure Notification to Developers Who Break a Build
* Take in account reduction of the Notification Spam
  + "Right" message, to "right" recipient at the "right" time

## GOING FURTHER

Some recommended readings on this subject:

* [Jenkins: Notifications](https://www.previous.cloudbees.com/jenkins/about/notifications)
* [Cleaning up and notifications](https://jenkins.io/doc/pipeline/tour/post/)
* [Notification Plugin](https://wiki.jenkins.io/display/JENKINS/Notification+Plugin)
* [Mailer Plugin](https://wiki.jenkins.io/display/JENKINS/Mailer)
* [Slack Plugin](https://wiki.jenkins.io/display/JENKINS/Slack+Plugin)

## DISTRIBUTED BUILD ARCHITECTURE

## WHAT IS DISTRIBUTED BUILD ARCHITECTURE

* Distributed Builds run on nodes other than the master node.
  + Additional nodes are configured and the master node manages these nodes to do the actual buildwork
* Distributed build architecture is recommended for all production systems
  + For demonstration purposes, you can run builds on the master node

## JENKINS COMPONENTS

* The **Jenkins Master** is the Jenkins service itself.
  + It is a webserver that also acts as a "brain" for deciding how, when and where to run tasks.
* A **node** is a server where Jenkins runs jobs on **executors**.
* The **agent** is the tool that manages the **executors** on a remote **node**, on behalf of Jenkins.
* Note that the Jenkins master also runs on a **node**.

## JENKINS MASTER

* This is where Jenkins is installed
* Management tasks (configuration, authorization/authentication)  
  are executed on the master
* Files written when a Pipeline executes are written to the filesystem  
  on the master unless they are off-loaded to an artifact repository  
  such as Nexus or Artifactory

## PROPER USE OF JENKINS MASTER

* Do not run the workload of building projects on the Master  
  in a production environment
  + You can do this for demonstration and study purposes
* Some administrative tasks (such as running a Backup)  
  are executed on the node that hosts the Jenkins Master
  + Add an executor to the master to perform the task
    - You can label this executor (for example, backup) and only allow it to be used by jobs that specify it
    - You can also create the executor, use it, then delete the executor to prevent build jobs from running on the master

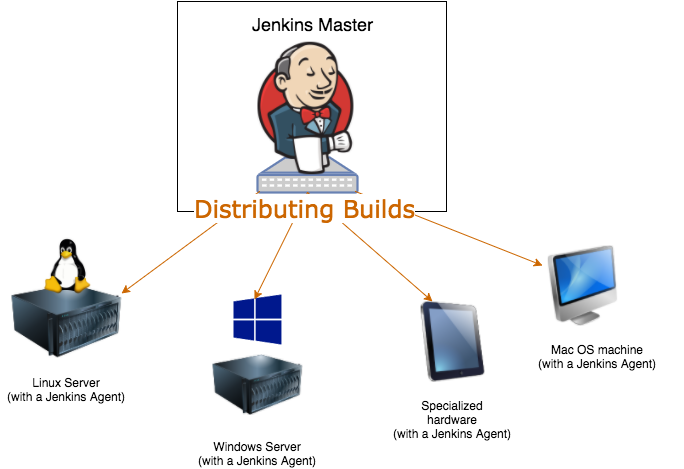
## NODES

* Provide the tools and packages to be used during your builds:
  + Application SDKs such as the JDK or C++ build tools
  + Apache Maven, Gradle, npm, shells, Docker, etc.
* Jenkins monitors each attached node for:
  + Disk space, free temp space, free swap
  + Clock time/sync and response time
* A node is taken offline if any of these values go outside the configured threshold

## WHAT ARE DISTRIBUTED BUILDS ?

* Distributed Builds are builds that run on nodes other than the master node.
  + Master
    - Serves HTTP requests
    - Stores all important information
  + Agents
    - 170KB single jar
    - Assumed to be unreliable
    - We know of clusters with 500+ nodes online at the same time
      * Memory, network bandwidth, disk I/O speed, etc on the master usually limit the number of builds that can be run simultaneously on the cluster more than the number of agents does

## ARCHITECTURE OF DISTRIBUTED BUILDS



## ADVANTAGES OF DISTRIBUTED BUILDS

* Your ${JENKINS\_HOME} is protected from malicious builds
* Makes the Jenkins instance more scalable
  + If you do not have enough resources to run your builds, just add more agents
  + Distributes the build load for better resource utilization
* You can define specialized nodes
  + Run different operating systems, use different CPU architectures, or different JDK versions

1.7

## WHAT ARE DISTRIBUTED BUILDS: AGENTS

* Build agent is a small Java program
  + Runs locally on the **Node** (machine) that it controls
  + Accesses files, forks processes, etc., on behalf of master
  + Uses the slave.jar file
    - Fetchable from the Jenkins Master
      * Point your browser to jnlpJars/slave.jar under $JENKINS\_URL

## MASTER AND AGENT COMMUNICATION

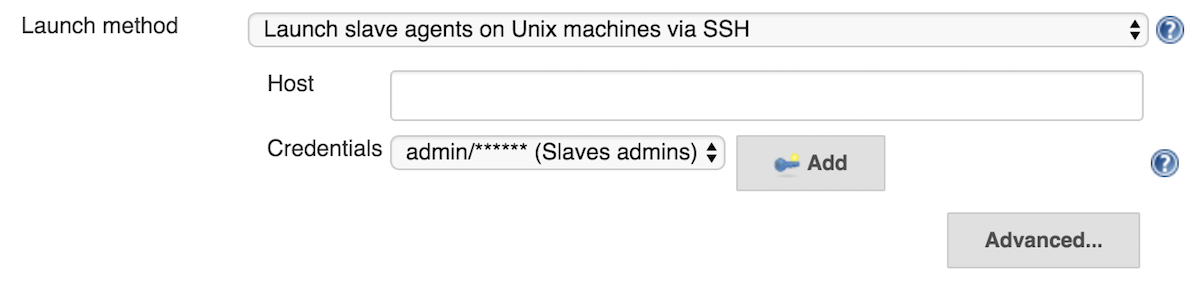
* Needs a bi-directional byte stream
  + No shared file system, no network topology constraints, etc
* Agent-to-Master Access Control is enabled by default
  + Protects the master from malicious agents
* Very flexible in how this gets set up
  + See Master/Agent Connectors

## MASTER-AGENT CONNECTORS

* An Agent can be launched using any of these master/agent connectors:
  + SSH issued from master
    - Supported on all Linux and MacOS systems
    - Windows 10, Windows Server 2012, and Windows Server 2016 offer an optional SSH server
  + Agent Managed as Windows Service
  + JNLP (Java Web Start)
  + Custom script run on Master

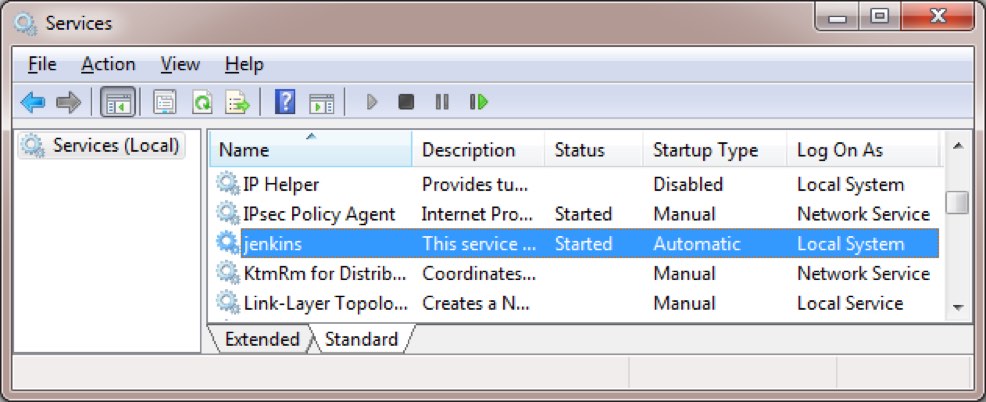
## MASTER-AGENT CONNECTORS: SSH

* Communication between Jenkins and Unix agents can use SSH
* SSHD only needs to be installed on agents
  + Jenkins can install Java and other tools
  + Agent’s JAR file is managed by Jenkins Master
* Just need a hostname and credentials to connect
  + Master’s SSH Public Key must be in remote node’s authorized keys



## MASTER-AGENT CONNECTORS: WINDOWS SERVICE

* To connect Jenkins to Windows agent, DCOM is an option
  + Just need admin user name and password
  + No manual intervention
  + Works even from Unix masters



## MASTER-AGENT CONNECTORS: JNLP

* Agent is launched as a single command with the slave.jar file and signals itself to the master
  + For cases when the master cannot “see” agents (firewall, subnets)
  + A separate socket connection is made (TCP or HTTP-tunneling)
  + The TCP port must be defined on the **Manage Jenkins → Configure Global Security** page
* Once started, can be installed as a service (Unix Daemon, systemd, …​)
  + Can also be run **headlessly** via jar (instead of Java Web Start with GUI)

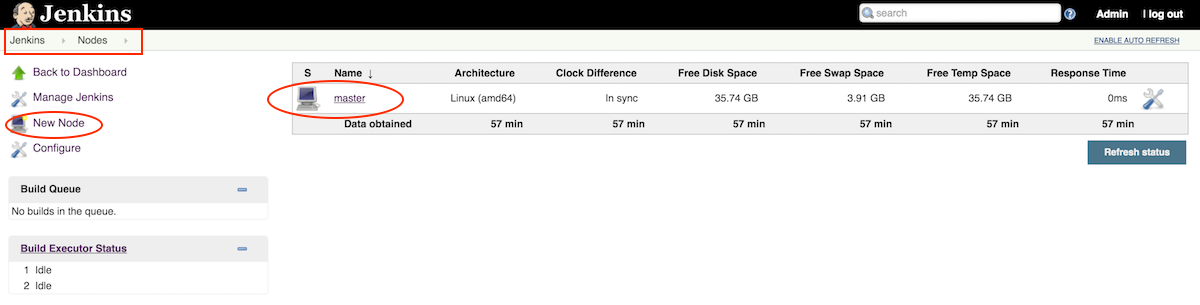


## MASTER-AGENT CONNECTORS: SCRIPT ON MASTER

* Launch agent with script on master: Customizable way of launching agent.
  + Responsibility of the script to :
    - Fetch slave.jar
    - Establish connection by launching remote agent
    - Do anything else related to your custom needs

## HOW TO: CREATE AGENT

* Access the Node Management page:
  + From Jenkins Management Page:
  + Browse to Manage Nodes
* Jenkins Master is already (and **always**) registered as a "Node"
* You can create, delete, edit nodes from there



## BENEFITS: SCALABILITY

* Masters can use resources to orchestrate, allowing one master  
  to orchestrate 100s to 1,000s of builds a day.
* When build demand increases, scale up by merely pointing Jenkins  
  at another agent node, rather than creating an entirely new master from scratch.

## BENEFITS: SECURITY

* Jobs that run on a master have full permissions to all Jenkins resources on the master node, potentially allowing a malicious user to access private information from other builds or the system itself.
  + Running jobs on separate and isolated machines mitigates this risk.
* For even greater security, use Cloud Agents that are created for each build then destroyed when that build is over.
  + If a malicious agent taints the node/executor, no other build is affected.

## INSTALLED TOOLS PROBLEM

* Tools location problem: Are JDK, Maven, Gradle and other CI tools installed on **all** your agents?
  + Are you really that good at provisioning machines?
  + Huge Nodes in that case
  + With versioning problems (JDKs 6, 7, 8, 9 …​)
* From this, we consider 3 patterns of Agents:
  + Dedicated Agents
  + Fungible Agents
  + Cloud Agents

## SOLUTION 1: DEDICATED AGENT

* Default case; having dedicated agents is required
  + Specialized hardware / operating systems
  + Segregation by teams / groups
  + Specialized software
* Jobs can be tied to specific Agents using **labels**
  + Use the Restrict where this project can run Project Configuration Option
  + Pattern and logical combination can be applied
  + Agent Usage must be tuned:
    - Use it as much as possible?
    - Restrict only when requested by a label restriction?

## SOLUTION 2: FUNGIBLE AGENTS

* A Fungible Agent can be substituted by any other Agent
  + Notice, when creating Node, how it is called **"Dumb"** ? :o)
* Builds know they are running in "empty" environment:
  + Consider installing required tools themselves
  + Use Pre-Build Step scripts, other Jobs, Pipeline Libraries
  + A plugin is available for this: [Custom Tools Installation](https://wiki.jenkins.io/display/JENKINS/Custom+Tools+Plugin)

## SOLUTION 3: CLOUD AGENTS

* **Throwable** Agent in a Cloud, to efficiently use resources
  1. Start a Cloud-based Agent
  2. Run the build inside this agent
  3. Tear-down the agent
* **Cloud** can be a public cloud like Amazon EC2, MS Azure.
  1. Can also be any private platform supported by **Apache jclouds** (Apache CloudStack, OpenStack, VMware…​)
* Can be combined with the "Bring you own Agent" pattern
  1. Jenkins Users can specify their own cloud templates with their **own** tools
* This is **easy** to do with container technologies like Docker:
  1. Quick start of container and immutable templates; no waiting!

## MONITORING NODES

* Jenkins has a tool named **NodeMonitor**
* It runs healthchecks against **Nodes**
  + CPU and memory usage
  + Checking disk space
  + More can be added by plugins
* When a node is down or in bad shape, Jenkins stops launching builds until it comes up again
  + Alerting can be triggered for admins

## WHAT DID WE LEARN ?

* Jenkins can distribute its build workload across a farm of **Agents**
* A Jenkins good practice is: **Never build on master**
  + Simple Security
  + Scale build workloads
  + Separate concerns
  + Specialized builds
* Be careful: Distributing builds adds some overhead for sysadmin
  + Road to DevOps: Jenkins users will efficiently learn their tooling
* Highly tunable concepts:
  + Scheduling, monitoring, restrictions, resource usages, provisioning

## GOING FURTHER

* Some recommended readings on this subject. Note that some of these documents use the obsolete "slave" terminology to refer to an agent.
  + [Distributed builds](https://wiki.jenkins.io/display/JENKINS/Distributed+builds) is an updated version of Kosuke Kawaguchi’s seminal and comprehensive paper
  + [Distributed Builds Architecture](https://jenkins.io/doc/book/architecting-for-scale/#distributed-builds-architecture)
  + [Step by step guide to set up master and agent machines on Windows](https://wiki.jenkins.io/display/JENKINS/Step+by+step+guide+to+set+up+master+and+agent+machines+on+Windows)
  + [Swarm Plugin](https://wiki.jenkins.io/display/JENKINS/Swarm+Plugin)
  + [Custom Tools Plugin](https://wiki.jenkins.io/display/JENKINS/Custom+Tools+Plugin)
  + [Job Restrictions Plugin](https://wiki.jenkins.io/display/JENKINS/Job+Restrictions+Plugin)
  + [Docker Slaves Plugin](https://wiki.jenkins.io/display/JENKINS/Docker+Slaves+Plugin)
* If you are interested in really large Jenkins clusters, you may enjoy the [So you want to build the world’s largest Jenkins cluster?](https://www.cloudbees.com/sites/default/files/2016-jenkins-world-soyouwanttobuildtheworldslargestjenkinscluster_final.pdf) presentation from the 2016 Jenkins World.

## LAB EXERCISE

[Distributed Jenkins Builds](https://s3.amazonaws.com/cloudbees-training-materials/training-admin-fundamentals/master/selfpaced/labs/Distributed_Jenkins_Builds.html)

## SECURITY OVERVIEW

## WHY DO SECURITY ?

* Protect your information
* Avoid executing malicious code

## SECURE YOUR INFORMATION

* Your organization has information that is used to create value
* Information is valuable and you must ensure the following:
  + Confidentiality
  + Integrity
  + Availability
* Implementing security practices protects your information

## AVOID EXECUTING MALICIOUS CODE

* Your Jenkins environment is also a fully distributed build system
  + Each network connection is a potential point of entry
* A bad actor could access your environment to launch a DDoS attack or a bot or other mischief
  + The code that runs your builds can be perverted to run anything
* Users could inadvertently download malware from a web site if the site is not protected
  + This malware could then infect Jenkins in some way

## WHAT IS SECURITY ?

* Security is the set of practices and tools to fight and prevent **threats**
* Major principles for security
  + **Know the system:** The more you understand about how your system works,  
    the more you are prepared to protect the integrity of your system
  + **Least privilege:**
    - Give people the privileges required to do their jobs but do not give everyone  
      permission to do everything
    - Do not open ports that are not required for your work
  + **Defense in Depth:** Systems are layered. Put security on all layers.
  + **Prevention is good. Detection is better:** Monitor your Jenkins installation  
    constantly to detect signs of a security breech quickly

## HOW TO DO SECURITY? LEAST PRIVILEGE

* Handling "Least privileges" concepts helps you manage the **AAA** concepts:
  + Authentication — Control who can access the system
  + Authorization — Control what each user can do on the system
  + Accounting — Monitor your system to ensure that only valid processes are executing

## HOW JENKINS EXECUTES A PIPELINE

* A simple overview of how Jenkins executes a Pipeline helps us understand the security considerations:
  + By default, the Pipeline executes with the full privileges of the Jenkins administrator
    - You can configure Jenkins to execute Pipelines with fewer privileges
  + All of the Pipeline logic, the Groovy conditionals, loops, etc execute on the master
  + Creates a workspace for each build that runs
    - Stores files created for that build
* The Pipeline calls steps that run on agents

## WHAT AGENTS DO

* The Pipeline calls a series of steps, each of which is a script or command that does the real work and mostly executes using an executor on an agent
  + Agent writes some files to the local node
  + Agent sends data back to the master
  + Agent can also request information from the master

## HOW JENKINS CAN DO MISCHIEF

* Jenkins and the Pipelines it runs must be able to do almost anything
  + A malicious Pipeline could reconfigure the Jenkins instance, delete files, launch malware attacks
* A trusted user could visit an infected web site and accidentally introduce malicious code into the Jenkins instance

## AGENTS AND SECURITY

* Agents that execute on the Jenkins master may be able to access Jenkins  
  configuration files and the workspaces of other builds
* An agent may be able to request information that belongs to other teams  
  or organizations from the master
  + This is not generally a concern for small Jenkins environments where  
    all agents are "trusted" to the same degree that the master is trusted  
    and all users have the same level of access to all configured processes
* An agent can write malicious code to the local disk so that the node is tainted
  + For maximum security, run all builds on ephemeral agents in the cloud  
    that are destroyed at the end of the build job

## IMPLICATIONS OF BUILDING ON MASTER

* Do not run builds on the Jenkins master in production environments
  + A Pipeline that performs administrative tasks such as backup may run on the master
    - Be sure to label the executor and only allow it to be used by jobs that use that label
  + Build jobs can run on the master node for demonstration purposes  
    in non-production environments
* A build job that uses an agent that runs on the master node has access to Jenkins  
  master files and configuration, which poses a security risk

## SECURE ACCESS TO EXTERNAL RESOURCES

* Pipelines may need trusted access to external resources such as a Nexus, Artifactory or Elasticsearch database
* To avoid having to code usernames and passwords in each Pipeline and share the information with everyone each time you change them, you can set up Credentials
  + Defined by the administrator
  + Called from the Pipeline code when accessing the external resource
* The administrator can easily change the password frequently to reduce the time that "stolen" credential information can be used

## AUTHENTICATION

* Authentication is the set of tools and procedures that **identify** a user with enough confidence
  + When police check your ID, it is authentication
  + Using a login and a password is an authentication
  + Biometrics are also a type of authentication
  + 2FA, which stands for "Two Factor Authentication", is a stronger authentication scheme

## SET UP AUTHENTICATION

* Setting Security Realms tells Jenkins which "referential" to use for authentication
  + Security Realm is a dedicated database for user and passwords
* 4 kinds of Realms supported out of the box, covered below:
  + Jenkins User Database
  + Unix user/group Database
  + Servlet Container
  + External LDAP
* New Realms types can be added by other plugins
  + E.g. ActiveDirectory
* Users who are not authenticated are always bound to the **"anonymous"** special user

## SECURITY REALM

* Defines the security implementation used to establish the identity of users
  + Jenkins supports many different authentication systems through security realms
* Only one security realm can be active at a time
* Additional security implementations can be added with plugins
  + For example, the SAML 2.0 Single Sign On facility
* By default, users and groups come from the Jenkins internal user database
  + Smaller, more informal installations can use this internal database
  + Enterprise installations usually use a corporate service (LDAP, AD, or Unix), which allows users to log into Jenkins with their usual username and password.

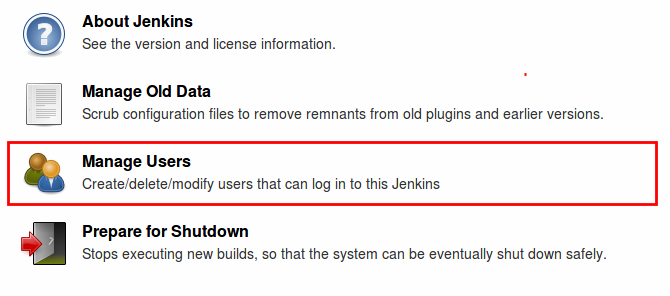
## CONFIGURE SECURITY REALM

* Choose the security realm used to authenticate users:



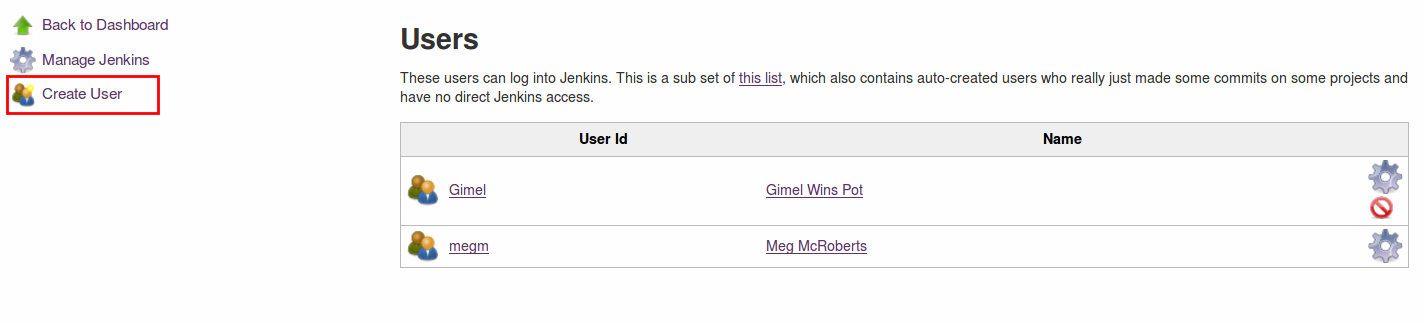
## MANAGE USERS IN JENKINS USER DATABASE

* Use the "Manage Users" screen to create new users in the Jenkins User Database and to view and modify information about existing users:
* It is not necessary to populate this database if you are using another security realm
  + Users can log in with their regular user ID and password



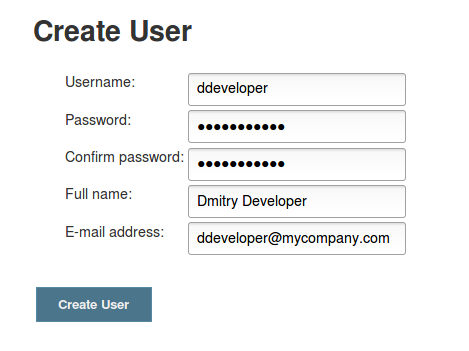
## CREATE NEW USER

* Click the "Create User" link in the left frame to add a new user:



## USER INFORMATION

* Fill in the form that is displayed



* Click the "Create User" button to create the new user

## NEW USER IS CREATED

* Jenkins displays the revised list of all users in the Jenkins user database, which now includes the new user you created:



* Click the flywheel to the right of each user to view or modify configuration information  
  for that user

## MODIFY USER INFORMATION

* Click the flywheel to the right of each user to view details about that user. On that screen, you can:
  + Provide a "Description" that tells other users more about who you are and what you do
  + Supply SSH Public Keys
  + Modify the configured email address and password

## AUTHORIZATION

## WHAT IS AUTHORIZATION?

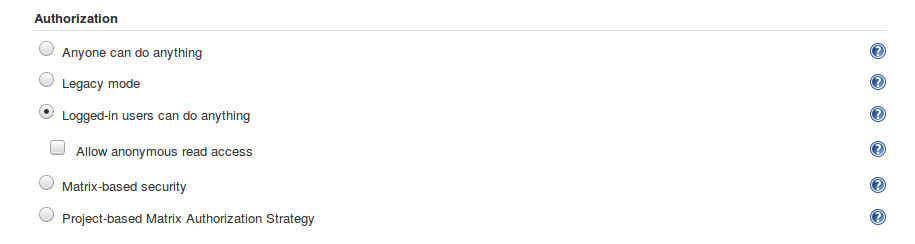
* Once a user is authenticated, need to determine the actions this user should have the right to do
* **Authorization** always occurs in the context of authentication
* Authorization grants access rights to:
  + **Resource:** Task, object and/or action to manipulate
  + **Role:** A set of rights grouped together by commodity
  + **Requester:** User or group that has roles assigned and wants to manipulate resources

## SECURITY MATRIX

* Use a Security Matrix to populate and visualize the authorizations:
  + **Matrix-based security** assigns global privileges to users and groups
  + **Project-based Matrix Authorization Strategy** assigns permissions  
    to users and groups based on the project(s) they access
  + **Role strategy** assigns permissions to specific roles;
    - It is available as an open source plugin but not discussed in this class

## CONFIGURE AUTHORIZATION

* Configure authorization — who is allowed to do what on Jenkins



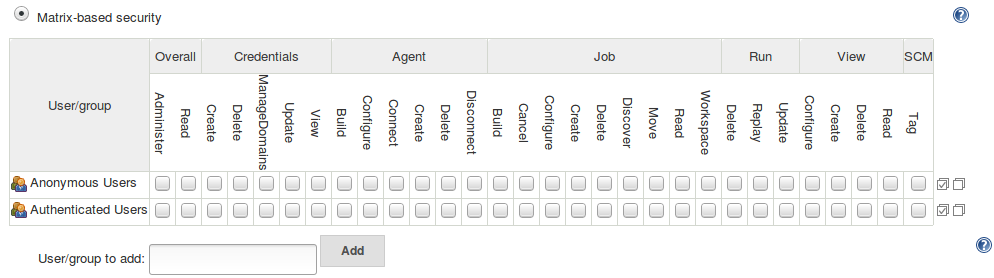
* The default "Logged-in users can do anything" is seldom a good idea!
* One of the matrix based schemes is typically used

## MATRIX-BASED SCHEMES

* Choose one of the following schemes:
  + Matrix-Based Security (Per User)
    - Define actions allowed for each user or group globally
  + Project-based Matrix Authorization Strategy
    - Define actions allowed by project

## MATRIX-BASED SECURITY PER USER

* The following screen is displayed when you choose "Matrix-based security":



## ABOUT THE SECURITY MATRIX

* You can add users and/or groups to this matrix
* Hover the cursor over each "Privilege" to display information about what that privilege entails
* Click the appropriate boxes to define the access that is allowed for each defined user or group

## ROWS

* Each row in the table represents a user or group
* Two rows are defined by default:
  + **anonymous** — permissions granted to all unauthenticated users  
    who access the Jenkins environment
  + **authenticated** — permissions granted to all authenticated users  
    who access the environment
* To add a user or group to this matrix, type the name into  
  the "User/group to add:" box + and click "Add"
* Permissions granted are additive

## PERMISSION GROUPINGS

* Permissions are grouped:
  + Overall
  + Credentials
  + Agent
  + Job
  + Run
  + View
  + SCM
* Other plugins you install could add groupings
* The [Matrix-based security](https://wiki.jenkins.io/display/JENKINS/Matrix-based+security) wiki gives more detailed descriptions of many of these permissions

## OVERALL PERMISSIONS

* The first group of permissions in the matrix are the "Overall" permissions
  + Several of these are at least as powerful as "administer"
    - For historical reasons, "administer" can also perform the actions associated  
      with these permissions
  + Users with overall permissions have the ability to do great harm to your cluster,  
    either accidentally or deliberately
* Consider carefully how you allocate these permissions

## UNDERSTAND OVERALL PERMISSIONS

* **Administer** - Make system-wide configuration changes
  + Perform highly sensitive operations that amount to full local system access (within the scope granted by the underlying operating system)
  + In versions 5.25 (November 2018) and later, Overall/Administrator does **not** imply the Overall/RunScripts, Overall/UploadPlugins, and Overall/Configure UpdateCenter permissions
* **Read** - View almost all pages within Jenkins
* **RunScripts** - Run Groovy scripts using the Groovy console or Groovy CLI command
* **UploadPlugins** - Upload arbitrary plugins
  + Can execute arbitrary code in the context of any Jenkins internal user
* **ConfigureUpdateCenter** - Configure update sites and proxy settings
  + Can control the update site metadata and plugin files downloaded y the Jenkins plugin manager
  + Can be used to execute arbitrary code

## MORE ABOUT OVERALL/RUNSCRIPTS

* Users with Overall/RunScripts permissions can use the **Script Console** to run  
  a Groovy script in the Groovy Sandbox without getting administrator approval
  + Is useful for troubleshooting and diagnostics
  + Each method call, object construction, and field access is checked against  
    a whitelist of approved operations
  + If the script attempts to use an unapproved operation, it is killed
  + The unapproved operation is added to an approval queue which an administrator  
    can review and, if appropriate, can add the operation to the whitelist

## CAVEATS FOR OVERALL/RUNSCRIPTS PERMISSIONS

* Users with the "Overall/RunsScripts" permission can execute arbitrary code  
  in the context of any Jenkins internal user, including the internal SYSTEM user
* [Jenkins Script Console](https://wiki.jenkins.io/display/JENKINS/Jenkins+Script+Console) includes valuable security guidelines

## CONSIDERATIONS FOR SETTING PERMISSIONS

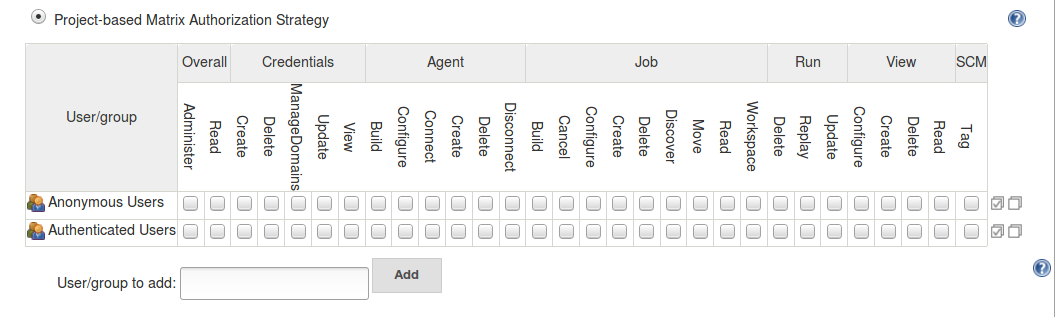
* Security principle of **Least Privilege**
* Trade-off between maximum security and avoiding bottlenecks in your CI/CD flow
  + The fewer people who have a permission, the less chance of a security breach
  + Ensure that someone is always available to do critical tasks such as take an agent offline
    - A particular challenge with groups spread across different timezones
  + Are tasks covered adequately if someone is sick or takes vacation?

## DOING SOME WORK

* Grant appropriate permissions to "Anonymous Users" and "Authenticated Users"
* Add the other users that are in the "Authentication" section
  + Discuss what sorts of permissions each needs
  + Skip Credentials for now; we’ll do that in the Credentials section that follows
  + Set up those permissions
* Add a user who is not in the Jenkins User Database
  + Discuss what happens
* Show how to define a group
  + Add a group and set permissions

## PROJECT-BASED MATRIX AUTHORIZATION STRATEGY

* Use the project-based matrix to define authorization by project rather than users:



## IMPLEMENTING PROJECT-BASED MATRIX AUTHORIZATION STRATEGY

* Additional access control lists (ACLs) can be defined separately for each project
  + Users and groups can be granted access only to the specified projects
* ACLs defined for project-based matrix authorization are additive
  + Access grants defined here are combined with project-specific ACLs
  + Managing inheritance options is complicated

## ACCOUNTING

* Occurs in the context of a user who is authenticated and authorized.
* Measures resources used or consumed by the user during access.
  + This can be amount of data, compute resources or system time.
* Enforces limits when they are defined to protect system.
* Related to system measurement, capacity planning and feedback loops

## LOGS ON THE SYSTEM

* When running jenkins.war manually with java -jar jenkins.war,  
  all logging information is output to stdout by default.

## LOGS ON LINUX SYSTEMS

* By default, logs are available in /var/log/jenkins/jenkins.log,
  + Log location can be customized
    - In /etc/default/jenkins (for \*.deb)
    - In /etc/sysconfig/jenkins (for \*.rpm)

## LOGS ON WINDOWS SYSTEMS

* By default, logs are available in:
  + %JENKINS\_HOME%/jenkins.out
  + %JENKINS\_HOME%/jenkins.err
* The log location is customized in %JENKINS\_HOME%/jenkins.xml

## LOGS ON DOCKER

* If you run Jenkins inside Docker as a detached container:
  + Use docker logs containerId to collect Jenkins logs

## LOGS ON JENKINS GUI

* Go to **Manage Jenkins → System Log**
* Create a custom log recorder
  + Helps you group relevant logs together while filtering out the noise

## MONITORING SERVER LOAD

* Jenkins provides built-in monitoring of server activity
* Go to **Manage Jenkins → Load Statistics**
  + Displays a graph of the server load time for the master node
  + Graph keeps track of four metrics:
    - Number of online executors
    - Number of busy executors
    - Number of available executors
    - Queue length

## MONITORING

* Many open source plugins are available for monitoring
* Some of the popular ones are:
  + Monitoring
  + Disk Usage
  + Build Monitor

## MONITORING PLUGIN

* The Monitoring plugin provides monitoring of Jenkins with JavaMelody
  + Charts of CPU, memory, system load average, HTTP response time
  + Details of HTTP sessions, errors and logs, actions for GC, heap dump, invalid session(s)
* The Disk Usage plugin shows project-wide details for all jobs and all workspaces
  + It also displays Disk Usage Trend
* The Build Monitor Plugin provides a detailed view of the status of selected Jenkins jobs
  + It also shows the names of people who might be responsible for "breaking the build".

## AUDITING

* Many open source plugins are available for Auditing
* Some of the popular ones are:
  + Audit Trail
  + Job Configuration History

## AUDIT TRAIL PLUGIN

* Keeps a log of users who performed particular Jenkins operations, such as configuring jobs
* This plugin adds an Audit Trail section in the main Jenkins configuration page
* Many configuration options are supported
  + Save output audit logs in rolling files
  + Send audit logs to a Syslog server
  + Output audit logs in stdout or stderr. Primarily intended for debugging purposes

## VIEW JOB CONFIGURATION HISTORY PLUGIN

* Saves a copy of the configuration file of a job **(config.xml)** for each change made and of the system configuration
* Provides an overview page of all changes
* The overview page only lists changes made to the system configuration (for performance reasons)
  + Use links to view either all job configuration histories or just the deleted jobs or all kinds of configuration history entries together
    - This may take some time to load, If your instance is running many jobs

## WHAT DID WE LEARN

* Security on Jenkins is a vast topic, but many resources are available
* Start by enabling authentication and authorization
  + Numerous authentication realms, extensible by plugins
  + Few authorization strategies, but powerful ones
  + Many threats can be leveraged by accounting and Jenkins OOB settings

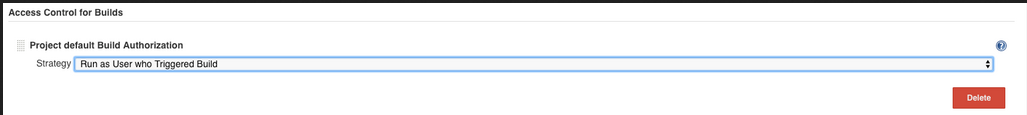
## GLOBAL SECURITY SETTINGS

## ABOUT SECURITY SETTINGS

* Global Security Settings close off intrusion paths for the Jenkins instance
* By default, Jenkins is installed with restrictive settings
  + This reduces the attack surface of the Jenkins cluster
  + You may need to relax these settings to run your jobs
* Recommend that you reduce protection only when necessary

## ACCESS CONTROL FOR BUILDS

* By default, all Pipelines/jobs run with administrator privileges
  + Any Pipeline can reconfigure Jenkins, delete/modify files, etc
* Set "Access Control for Builds" to limit the permissions used for builds:



## OTHER GLOBAL SECURITY SETTINGS

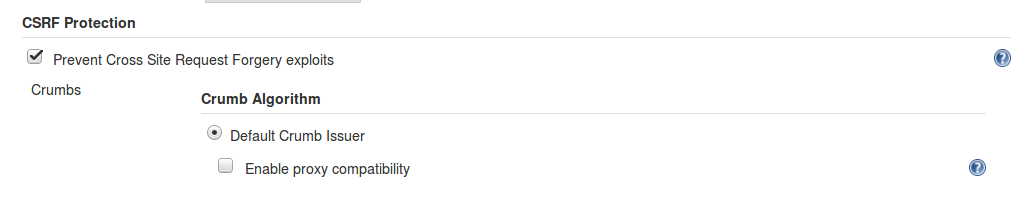
* The "Global Security Settings" screen also includes fields to configure or enable various features that have security ramifications
* By default, Jenkins is installed in "locked-down" mode
  + Features that can have security ramifications are turned off
  + You may need to enable some of these features to get the functionality you require
  + In most cases, plugins and practices are available to reduce the security vulnerability of these features

## CSRF (CROSS-SITE REQUEST FORGERY)

* CSRF is an exploit that enables an unauthorized third party to perform requests  
  against a web application by impersonating another, authenticated user
* In a Jenkins environment, a CSRF attack could allow a malicious actor to delete projects,  
  alter builds, or modify the system configuration of the Jenkins instance
* Also sometimes called XSRF
* See the OWASP Foundation’s [Cross-Site Request Forgery](https://owasp.org/www-community/attacks/csrf) document for more information about CSRF exploits

## CSRF PROTECTION

* Jenkins enables CSRF Protection by default; we strongly recommend that you keep it enabled:



* When enabled, Jenkins checks for a CSRF token (or "crumb") on any request that might change data in the Jenkins environment
  + This includes any form submission and calls to the remote API, including those that use "Basic" authentication

## ISSUES WITH CSRF PROTECTION

* CSRF protection might result in challenges for your Jenkins builds:
  + Accessing Jenkins through a poorly-configured reverse proxy may result in the CSRF HTTP header being stripped from requests, causing protected actions to fail
    - See [Jenkins behind an NGINX reverse proxy](https://wiki.jenkins.io/display/JENKINS/Jenkins+behind+an+NGinX+reverse+proxy) for information about correcting the proxy configuration.
    - Clicking "Enable proxy compatibility" may allow you to run your build without disabling CSRF protection.
  + Outdated plugins that have not been tested with CSRF protection enabled may not function properly
  + In Jenkins 2.95 and earlier, CSRF protection made use of the remote API difficult. Since 2.96, using the remote API with username/API token authentication no longer requires that a valid CSRF protection crumb be provided.
    - You must use an API token rather than a password
* It may be necessary to disable CSRF protection if any of these interfere with your builds

## JNLP (JAVA NETWORK LAUNCH PROTOCOL)

* JNLP (Java Network Launch Protocol) can be used to launch an application on a client desktop by using resources that are hosted on a remote web server
  + Used primarily by Windows based agents
  + These agents could instead be run using SSH
* Jenkins uses a TCP port to communicate with agents launched via JNLP.
  + By default, this port is chosen randomly to avoid collisions

## CONFIGURING JNLP ACCESS

* Use the "Agents" field to configure JNLP access



* Recommend leaving this disabled unless you have JNLP agents
  + If your Windows agents use SSH rather than JNLP, leave the TCP port for JNLP agents disabled
  + You can then configure your Firewall appropriately
* To enable JNLP access, click either the "Fixed" or "Random" buttons
  + The more secure option is to choose "Fixed" and define a TCP port to use for JNLP

## AGENT-TO-MASTER ACCESS CONTROL

* Agent-to-Master Access Control filters the commands that agents can send to the Jenkins master
  + It is enabled by default:



* **Strongly recommend** that this protection be enabled
  + Especially for a large enterprise deployment that takes agents from other teams
  + Click on the link to tweak the rules if necessary

## TWEAK THE AGENT-TO-MASTER RULES

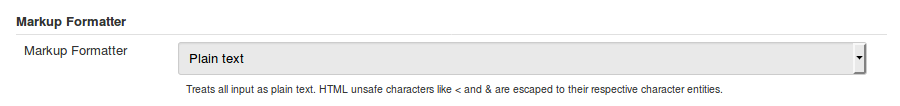
* If builds are failing because of the Agent-to-Master Access Control:
  + First, be sure that you have upgraded to the latest version of all plugins
    - Older versions of plugins may not work correctly with this feature
  + Whitelist the specific commands that you need
  + Add allow/deny rules to refine the list of files on the master and the type of access that is allowed for the agent.
  + See the [Slave To Master Access Control](https://wiki.jenkins.io/display/JENKINS/Slave+To+Master+Access+Control) documentation for more information.

## MARKUP FORMATTING

* Jenkins allows user input in some text areas and configuration fields
  + Examples include job descriptions, user descriptions, view descriptions and build descriptions
* HTML formatting in these text areas can lead to users inadvertently (or maliciously) inserting unsafe HTML and/or JavaScript
  + This could be used for Cross-site Scripting (XSS) attacks on web pages
  + XSS enables attackers to inject client-side scripts into web pages that are viewed by others

## CONTROL HTML FORMATTING

* Use the "Markup Formatter" field to control how HTML code is rendered on your Jenkins instance:



* By default, this is set to "Plain text", which causes all HTML formatting to be ignored. This is the safest option but users are deprived of the aesthetic advantages of formatted text
* You can choose the "Safe HTML" option, which allows you to use a subset of HTML markup such as hyperlinks, bold and italic characters.
* Other plugins can be installed to support additional options for handling HTML markup
  + If such plugins are installed, they are included on the dropdown menu for this field

## CONTENT SECURITY POLICY (CSP)

* The Content Security Policy (CSP) header is applied to static files served by Jenkins
* Set to a very restrictive default set of permissions to protect Jenkins users from malicious HTML and JavaScript files in workspaces, /userContent or archived artifacts
* May need to relax the default rules for some popular, useful plugins
* Details in [Configuring Content Security Policy](https://wiki.jenkins.io/display/JENKINS/Configuring+Content+Security+Policy)

## MANAGING CREDENTIALS

## ABOUT CREDENTIALS

* Credentials are used to get trusted access to other resources without having to share the actual passwords, et cetera, with all users.
  + Jenkins itself uses credentials to secure internal communication
  + Administrators can define credentials that Pipelines use to access secured resources
* The canonical documentation about credentials is [Credentials API Plugin documentation](https://github.com/jenkinsci/credentials-plugin/tree/master/docs), written and maintained by Stephen Connolly, the author of the Credentials plugin.

## EXAMPLES OF CREDENTIAL USE

* Jenkins uses credentials to get trusted access to resources such as:
  + SCM services for getting and pushing code
  + Remote Secured services (such as secured LDAP plugins or the Elasticsearch analytics engine)
* Pipelines use Credentials to access resources such as
  + Artifact servers (such as Nexus and Artifactory) for fetching dependencies (or deploying new ones)
  + Secured environments to which it is deploying code

## WHY USE CREDENTIALS

* Using Credentials:
  + Provides centralized credentials management
  + Treats credentials as Jenkins Resources
  + Provides an API consumable by plugins and other Jenkins Resources
  + Plugins may add additional credential types:
    - GitHub/Google/Bitbucket OAuth
    - "SSH agents" plugin
* Authorization matrix defines who can create, delete, update, and view credentials

## HOW CREDENTIALS ARE RELATED TO RESOURCES

* Implemented through plugins for a type of external resource
  + The plugin defines the credentials that are supported for that resource
  + Examples include the plugins to access SCMs and artifact servers like Nexus and Artifactory
* Each plugin defines the credential types it supports and may not support all credential types
  + For example, the Git Plugin (and Git Client Plugin) can only use username/password credentials and private key credentials
  + This means that secret text, certificates, or other credential types are not proposed when using the Git plugin

## HOW CREDENTIALS ARE USED IN A PIPELINE

* Pipeline needs to deploy results to an external resource  
  such as Nexus, Artifactory, or Elasticsearch.
* Access to that external resource is protected by a username and password
* Sharing that username/password with 20 or 100 users is undesireable
  + It is a security risk to have multiple people share a password
  + If the password is changed, all those people must be informed, which is a hassle
* Instead, the administrator can define a credential that knows the username and password for the external resource
  + Defined under "Configure → Credentials" on the Jenkins dashboard
* Pipelines can then use either the environment or withCredentials directive to supply the username and password and access the secured resource
  + Users who run the Pipeline do not know what the password is

## LOWEST LEVEL PRINCIPLE

* Limiting the number of projects that can access a Credential enhances its security
* Credentials should be defined at the lowest possible level
  + Credentials defined for the master are available to all Pipelines run by that master
  + Credentials defined for a Folder are only available to Pipelines run from that Folder.
* The screens used to add and manage credentials for masters and folders are identical except for their location

## HOW CREDENTIALS ARE STORED

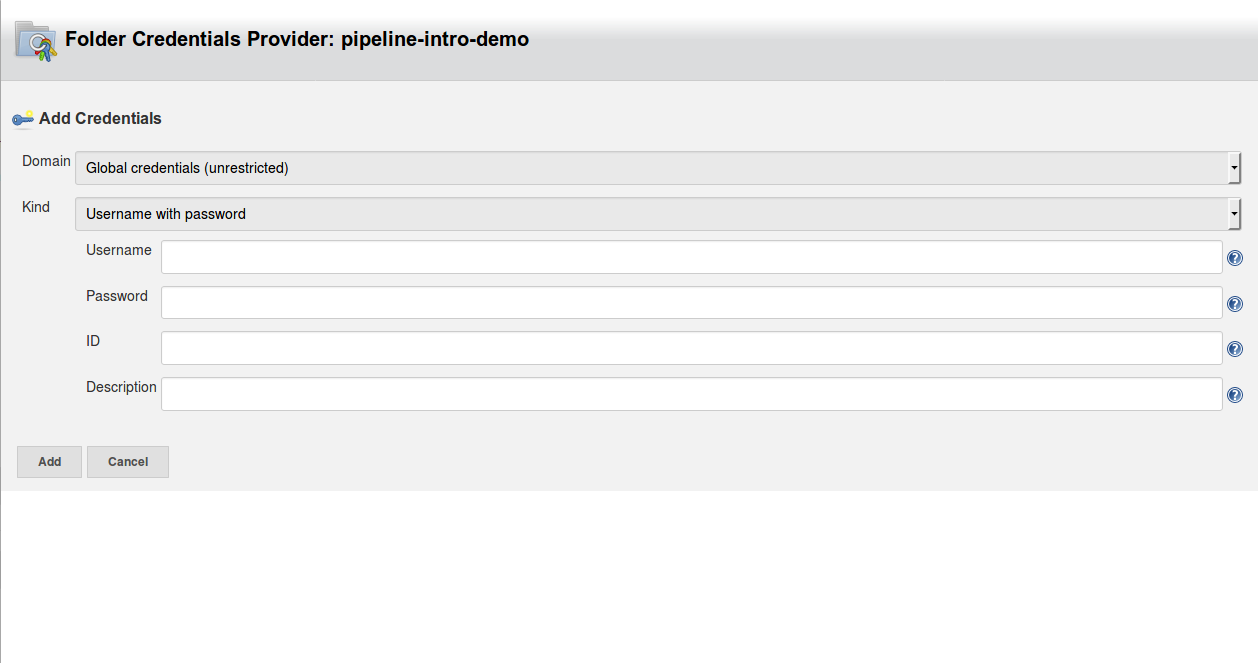
* Credentials are stored in an encrypted form on the Jenkins Master
  + Credentials are encrypted using a key derived from the master key
  + All credentials that you create are stored as type Secret
    - For users who have appropriate (configure) permission, the secret fields are round-tripped in their encrypted form
    - For other users, only a placeholder is used
* Keys are tied to and encrypted by the instance ID
  + This means they can not be migrated or copied to a new instance

## PROTECTING SECRETS

* Keys to decrypt secrets are stored in the $JENKINS\_HOME/secrets/ directory
  + This directory has restrictive Linux file system permissions and should be excluded from backups
  + Keys should not be stored in the SCM where their contents could be breached
  + A disk crash or data corruption for your application could cause the loss of all the encrypted information for your instance
  + You should **store the secret in some other location** with restricted access; it is small and never changes after it is created; you can manually re-add it when restoring from backup

## ADD CREDENTIALS

* Credentials can be added by any authorized user who has the **Credentials → Create** permission
* To create a credential, fill in the following form under "Configure" on the Web UI



## NOTES ABOUT ADDING CREDENTIALS (1/2)

* Assign an ID that can be used to access the credential. The Pipeline uses this ID to apply these credentials
  + We recommend that you use a string that is reasonably easy to remember and type
  + If you do not assign an ID, Jenkins assigns an ID that is a GUID
  + IDs should be 40 characters or less
  + Only use alpha-numerical characters plus the separator characters - , \_ .
  + IDs are case sensitive

## NOTES ABOUT ADDING CREDENTIALS (2/2)

* Scope defines the places where this credential can be used
  + System: Credential is available only to the object with which the credential is associated. Examples include email authorization or node connection, where the Jenkins instance itself is using the credential
  + Global: Credential is available to the object with which the credential is associated plus all objects that are children of that object. Most Credentials that are called in Pipelines are defined as Global.

## CREDENTIALS TYPE

* The Jenkins Credentials Type defines the behavior. It must match what the target object expects:
  + Secret Text type:
    - MYVARNAME contains the path of the file that contains the Secret Text
  + Standard username and password type:
    - MYVARNAME is set to <username>:<password> (conjoined name and password string)
    - MYVARNAME\_USR is set to just the username
    - MYVARNAME\_PSW is set to just the password

## MORE ABOUT CREDENTIAL TYPES

* Each credential type supported for the environment directive:
  + Must be supported in the credentials binding plugin
  + Must have a special handler for Declarative Pipeline
* Standard Declarative syntax currently supports the following credentials:
  + Username and Password (plugin type StandardUsernamePasswordCredentials)
  + Secret file (FileCredentials)
  + Secret text (StringCredentials)
* Use the sshagent step for SSH credentials, which withCredentials does not support

## EXAMPLE CODE

...

stage('Deploy Reports')

steps {

...

withCredentials(bindings: [string(credentialsId: 'elastic-access-key', variable: 'ELASTIC\_ACCESS\_KEY')]) {

// Environment Variable available in the remote shell

sh 'env | grep ELASTIC\_ACCESS\_KEY'

sh 'echo ${ELASTIC\_ACCESS\_KEY} > secret-file.txt'

// Groovy Variable

sh "echo ${ELASTIC\_ACCESS\_KEY} > secret-file.txt"

}

...

}

## NOTES ABOUT EXAMPLE CODE

* Credentials are implemented in a Pipeline using the withCredentials() method
* Uses the credential that is defined with the ID of elastic-access-key
* withCredentials binds that to a local Pipeline variable called ELASTIC\_ACCESS\_KEY
* The deploy-token step uses the ELASTIC\_ACCESS\_KEY local Pipeline variable to establish the credentials that it needs
* Anyone who can run this Pipeline gets the necessary credentials to run deploy-token

## FOR FURTHER READING

* [Using Credentials](https://jenkins.io/doc/book/using/using-credentials/)
* [Storing Secrets](https://jenkins.io/doc/developer/security/secrets/)

## SECURITY BEST PRACTICES

## JENKINS SECURITY TEAM

* The Jenkins security team works constantly to improve the general security of Jenkins
  + See [Jenkins Security](https://jenkins.io/security/)
* Security advisories and updates are released as needed
* Notifications are posted to the jenkinsci-advisories Google group
  + We recommend that you subscribe to this group to receive timely notifications
  + When an applicable security update is released, install it as soon as you can!

## GOING FURTHER

* Some recommended readings on this subject:
  + [Quick and Simple Security](https://wiki.jenkins.io/display/JENKINS/Quick+and+Simple+Security)
  + [Script Security Plugin](https://wiki.jenkins.io/display/JENKINS/Script+Security+Plugin)
  + [Matrix-based security](https://wiki.jenkins.io/display/JENKINS/Matrix-based+security)
  + [Best Practices for Setting up Jenkins Auditing and Compliance](https://www.previous.cloudbees.com/blog/best-practices-setting-jenkins-auditing-and-compliance)
  + [Audit Trail Plugin](https://wiki.jenkins.io/display/JENKINS/Audit+Trail+Plugin)
  + [JobConfigHistory Plugin](https://wiki.jenkins.io/display/JENKINS/JobConfigHistory+Plugin)
  + [Job Restrictions Plugin](https://wiki.jenkins.io/display/JENKINS/Job+Restrictions+Plugin)
  + [Disable security](https://wiki.jenkins.io/display/JENKINS/Disable+security)
  + [PAM Authentication Plugin](https://wiki.jenkins.io/display/JENKINS/PAM+Authentication+Plugin)

## ORGANIZE BUILDS

## WHY ORGANIZE YOUR BUILDS

* As the number of build projects grows, you’ll need some organization

## FOLDERS

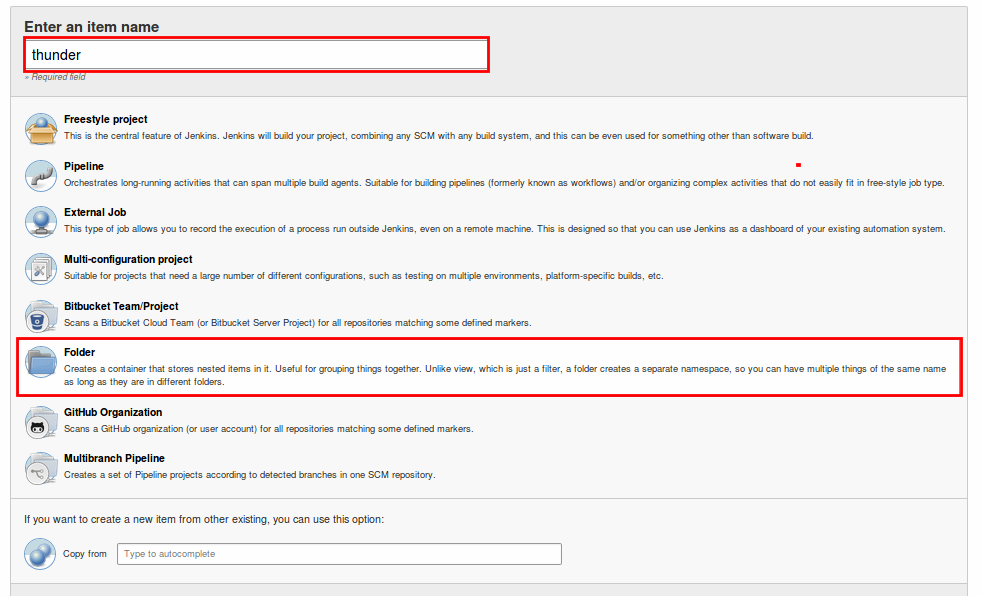
* Use **Folders** to organize a large number of builds around taxonomies such as projects or departments
* Folders can be nested like file system folders
  + Folder 1
    - Folder 2
      * Folder 3 … Folder n
* Creates namespace
  + "build" job in Folder **A** is different than "build" job in Folder **B**

## ABOUT FOLDERS

* One can create a new folder and move existing Pipelines into that folder
* You can clone an existing folder, leaving the children intact
  + Useful to seed new projects
* Folders can define properties that are only visible to jobs inside them
  + For example, assign Credentials to a Folder
    - Only Pipelines in that Folder can use those Credentials
    - This improves security
  + Simplifies branch and Pipeline management

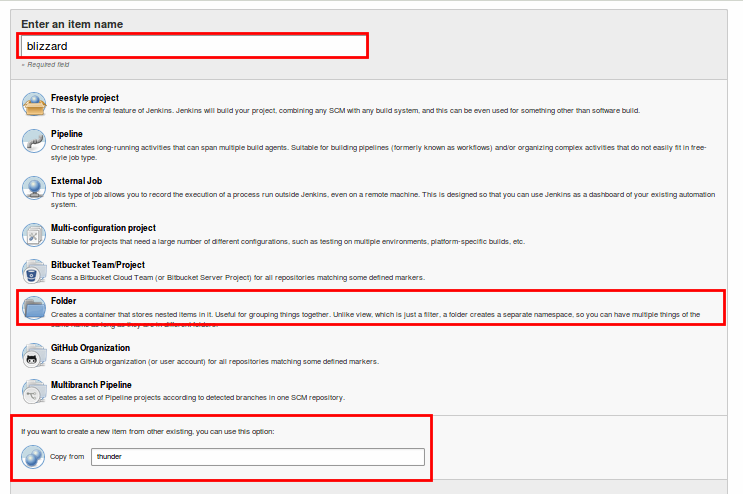
## CREATE A FOLDER

* Click on **New Item**, then click on **Folder** and type the name for the folder in the box:



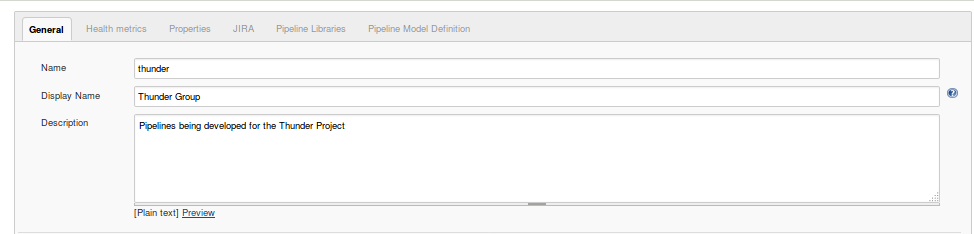
## CLONE A FOLDER

* To clone an existing folder, fill in the "Copy from" box at the bottom of the screen:



## CONFIGURE A FOLDER

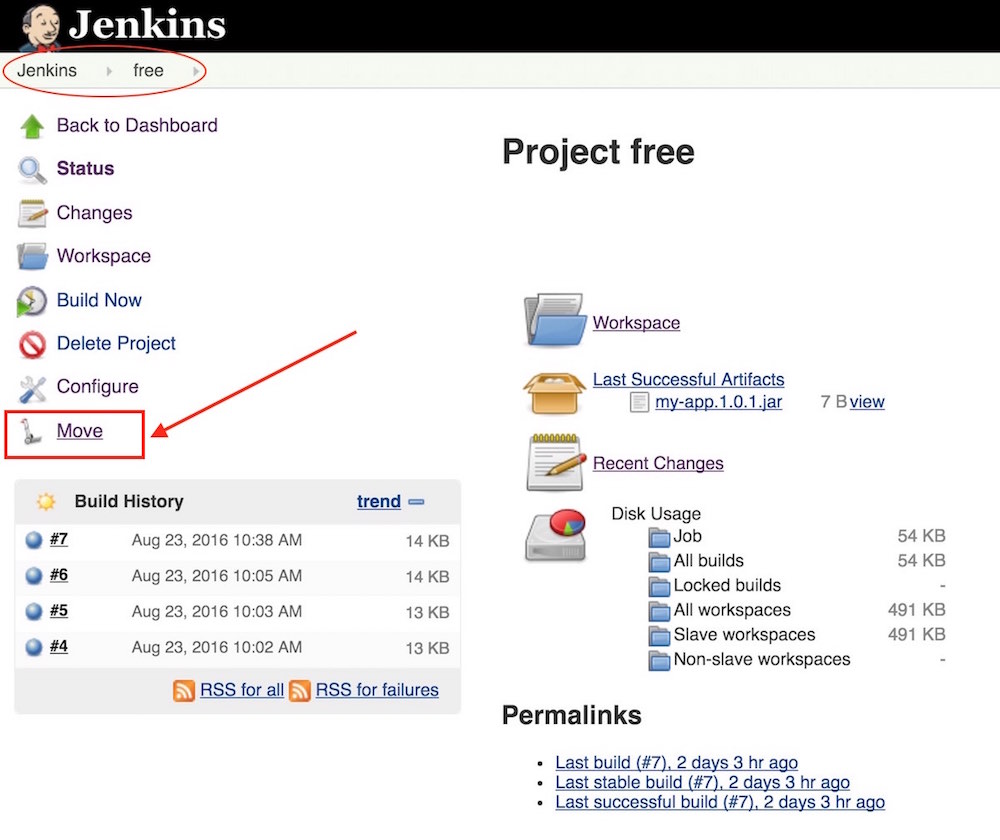
* You can (optionally) supply a "Display Name" for this folder and a description:

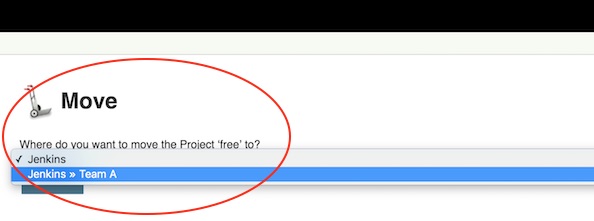


* Other configuration options are available here but are beyond the scope of this class

## MOVE A JOB TO A FOLDER

* Go back to the Jenkins dashboard and click on a Pipeline project you want to move to this folder; click "Move" in the left frame:





* Select the Folder to which you want to move the Pipeline

## GOING FURTHER

* Some recommended readings on this subject:
  + [CloudBees Folders Plugin](https://wiki.jenkins.io/display/JENKINS/CloudBees+Folders+Plugin)
  + User’s Guide for the [Folders Plugin](https://go.cloudbees.com/docs/plugins/folder/)

## MONITOR JENKINS

## WHY MONITOR?

* Ensure that the Jenkins instance is healthy
* Must understand how resources are being used before one can take actions to improve the health of the instance

## LOGS

* Define and differentiate:
  + Build Logs
  + System Logs

## METRICS

* Out-of-the-box metrics from inside Jenkins
* Other Metrics types:
  + build time
  + agent related

## METRIC AGGREGATORS

* Metrics exist but must be collected
* A Metric Aggregator service is recommended
  + Not covered in this class

## BACKUP AND RESTORE

## BACKUPS ARE IMPORTANT!

* Disaster recovery
* Recover an older configuration
  + An accidental configuration change may not be discovered until long after it was made
* Recover a file that is corrupted or deleted accidentally
* Your backup strategy should include validation of each backup
  + You do not want to learn that your backup is no good when you need it!

## BACKUP TOOLS: FILESYSTEM SNAPSHOTS

* Filesystem snapshots provide maximum consistency for backups
  + Run faster than live backups, reducing the possibility of copying different data at different time points
  + Supported by Linux Logical Volume Manager (LVM), Solaris ZFS (which also supports incremental backups), and other file system architecture
  + Some separate storage devices also let you create snapshots at the storage level

## BACKUP TOOLS: PLUGINS

* Several plugins are available for backup
  + Go to **Manage Jenkins → Manage Plugins → Available** and search for "backup"
  + None of the open source plugins are being maintained
* You can try these plugins but you may have problems with them

## BACKUP TOOLS: WRITE A SHELL SCRIPT

* You can write your own shell script that copies the appropriate files and directories to a backup location
* Use cron to schedule when the backup script runs
* Create a directory such as /mnt/backup; be sure that you have write permissions
  + Consider creating /mnt/backups as a separate filesystem with its own mount point
    - A subdirectory to /var is another option; you may need to use **sudo** for the restore operation
    - Backing up to /tmp is not advised because /tmp may be cleaned on reboot
  + Create a unique identifier for each backup (use a timestamp, for example) to ensure that today’s backup does not overwrite yesterday’s backup
* Writing files to a local file system is the fastest way to take the backup
  + Consider copying the completed backup to a remote backup server or device for long term storage

## BACK UP $JENKINS\_HOME

* All files are located under the **$JENKINS\_HOME** directory
* You can back up the entire **$JENKINS\_HOME** directory
  + This preserves everything
  + To restore the system, just copy the entire backup to the new system
  + Includes a number of files that do not really need to be backed up
* Selecting specific directories and files to back up yields smaller backups
  + May require a greater effort to restore a system
* You may want to back up different directories on different schedules

## BACKUP CONFIGURATION FILES

* Configuration files are stored directly in the **$JENKINS\_HOME** directory
  + **./config.xml** is the main Jenkins configuration files
  + Other configuration files have the **.xml** suffix
* Specify **$JENKINS\_HOME/\*.xml** to back up all configuration files
* Configuration files can also be stored in an SCM repository
  + This keeps copies of all previous versions of each file that can be retrieved using standard SCM facilities

## BACK UP ./JOBS SUBDIRECTORY

* **$JENKINS\_HOME/jobs** contains information related to all the jobs you create in Jenkins
  + **./builds** — Contains build records
  + **./builds/archive** — Contains archived artifacts
    - Back this up if it is important to retain these artifacts long term
    - These can be very large and may make your backups very large
  + **./workspace** — Contains files checked out from the SCM
    - Usually not necessary to back them up; you can perform a clean checkout after restoring the system
  + **./plugins/\*.hpi** — Plugin packages with specific versions used on your system
  + **./plugins/\*.jpi** — Plugin packages with specific versions used on your system

## WHAT DOES NOT NEED TO BE BACKED UP

* The following files and directories do not usually need to be backed up:
  + **./war** — Exploded war file
    - To restore a system, download the latest war file
  + **./cache** — Downloaded tools
    - To restore a system, download the current version of the tools
  + **./tools** — Extracted tools
    - To restore a system, extract the tools again
  + **./plugins/xxx** — Subdirectories of installed plugins
    - Will be automatically populated on next restart
* Alternatively, you can make an infrequent backup of just these files if you want to be able to easily restore the same versions of the system and all downloaded tools

## SIMPLE VALIDATION OF A BACKUP

* A simple way to validate a full backup is to restore it to a temporary location:
  + Create a directory for the test validation (such as **/mnt/backup-test**) and restore the backup to that directory
  + Set $JENKINS\_HOME to point to this directory, specifying a random HTTP port so you do not collide with the real Jenkins instance:

export JENKINS\_HOME=/mnt/backup-test

* + Now execute the restored Jenkins instance:

java-jar jenkins.war ---httpPort=9999

## WHAT DID WE LEARN

* Backups are critical for Disaster Recovery
* Always set up a backup policy
  + What configurations and records need to be saved from the master?
  + How often should backups be taken?
  + Where should backups be stored?
* Validate your backups
  + you should periodically check whether your backups are intact and can be used to meet your recovery objectives
* Taking backups is a **Jenkins best practice**

## GOING FURTHERWHY AUTOMATE JENKINS

* Scripting allows you to automate and standardize tasks:
  + Routine tasks
  + Bulk updates
  + Troubleshooting
  + and more
* Provides consistency and efficiency
  + For different staff members
  + At different times

## Some recommended readings on this subject:

* + [Why Smart, Efficient Backup and Restore Techniques are Essential with Jenkins Production Server](https://www.previous.cloudbees.com/blog/why-smart-efficient-backup-and-restore-techniques-are-essential-jenkins-production-server)
  + [Backup Plugin](https://wiki.jenkins.io/display/JENKINS/Backup+Plugin)
  + [thinBackup Plugin](https://wiki.jenkins.io/display/JENKINS/thinBackup)

## AUTOMATE JENKINS

## WHY AUTOMATE JENKINS

* Scripting allows you to automate and standardize tasks:
  + Routine tasks
  + Bulk updates
  + Troubleshooting
  + and more
* Provides consistency and efficiency
  + For different staff members
  + At different times

## WHAT IS JENKINS CLI ?

* Jenkins Command Line Interface (CLI) is a Java application
  + Provided as jenkins-cli.jar file, downloadable from Jenkins
  + Executed in a JVM
* This application is a client that reaches your Jenkins instance
  + Provides sub-commands to execute remote operations
  + Does not have to run on the same machine as Jenkins

## HOW IT WORKS ?

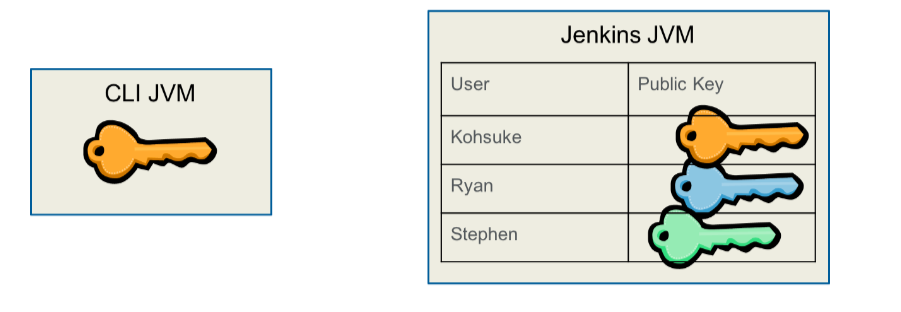
* CLI makes an HTTP call to Jenkins, discovers the port
  + This is the port used for the JNLP Agent
  + CLI attempts to open TCP/IP connection
* If it fails, it falls back to HTTP-based connection
  + Slow and wastes effort, but works with reverse proxy



* Command execution mostly happens on master
  + jenkins-cli.jar file needs to be downloaded locally
  + Therefore, one CLI jar can be used to talk to different masters

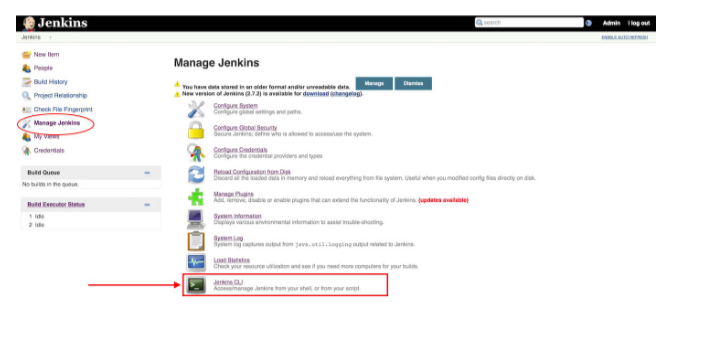
## HOW TO: CLI AUTHENTICATION

* By SSH public key
  + Register your public key to Jenkins under your user account
  + Hide your private key in ~/.ssh



## HOW TO: ACCESS CLI HOMEPAGE

* Access CLI home page from the Manage Jenkins page:

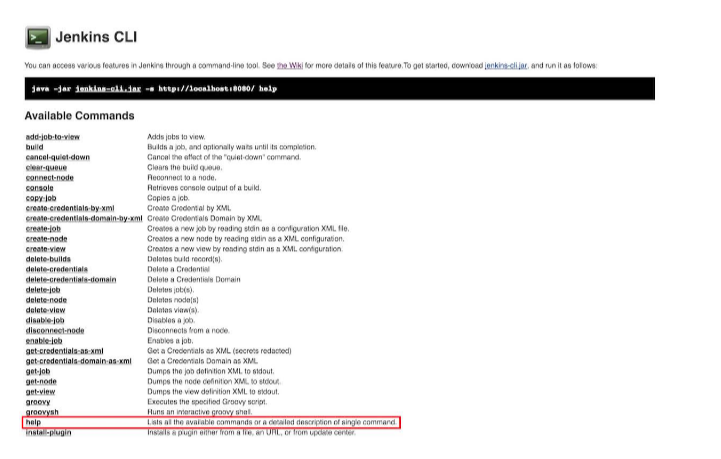


## HOW TO: USING CLI



* **-s** option: specifies the URL to connect to Jenkins
* JENKINS\_URL environment variable to use instead

## EXISTING CLI COMMANDS



* Plugins can contribute commands; this list can grow

## CLI SHOW CASE - BUILD COMMAND

* Equivalent of clicking “build now”
* Can take parameters:
  + **-c**: run polling first and build only if there is a change
  + **-s**: do not just schedule, but wait until the build is completed
  + **-p**: specify build parameters
  + **-v**: report console output as well

## CLI SHOW CASE - CRUD OF JOBS

* create-job, get-job, update-job, delete-job
* Reads/writes XML representation of the job
  + use stdin/stdout of CLI process
* Can be used to script job creation / update

## CLI SHOW CASE - GROOVYSH

* Use the interactive Apache Groovy shell to manipulate the in-memory state of Jenkins
* Equivalent to the script console, but interactive



## MORE ABOUT APACHE GROOVY IN CLI

* Use **groovy** command to run a pre-authored script file
* See <https://github.com/jenkinsci/jenkins-scripts> for scripts other people wrote

## JENKINS REMOTE ACCESS API

## WHY USE JENKINS API ?

* You may need to integrate Jenkins with another application
* Machine-to-machine communications is required when a service talks to the Jenkins instance
  + Not GUI

## WHAT IS THE JENKINS API ?

* A Remote access API that is machine consumable
* A REST-like API, allowing remote resources manipulation
* No single entrypoint, but per resource /api URL
* Projects API: http://<JENKINS\_URL>/job/api
* A multi-flavored data serialization:
  + XML
  + JSON with JSONP support
  + Python

## HOW TO: SIMPLE JSON API EXAMPLE

* This example will launch a parameterized build using the API
* We are using the JSON flavor
* This is done using a command-line tool named **curl**, that can issue HTTP requests
* An example Jenkins configuration:
  + Main URL is http://${JENKINS\_URL}
  + Project name is ${MY\_PROJECT}
  + The project is protected with a TOKEN: ${TOKEN}
  + Build requires 2 parameters:
    - parameter name **id** which will be exported as ${ID} (or %ID on windows) when the build starts
    - parameter name **verbosity** which will be exported as ${verbosity} (or %verbosity% on windows) when the build starts



## MORE ABOUT THE JENKINS API

* Jenkins API clients are available for various programming languages:
  + Ruby: <https://rubygems.org/gems/jenkins_api_client>
  + Python: <https://pypi.python.org/pypi/python-jenkins/>
  + JavaScript: <https://www.npmjs.com/package/jenkins-api>

## WHAT DID WE LEARN ?

* Automating Jenkins can be done using its CLI
* An alternative to running Groovy with the GUI
* The CLI is a JAR file that can be executed remotely to run Jenkins operations
* The API is REST-like and can be used to integrate Jenkins with other services

## GOING FURTHER

* Some recommended readings on this subject:
  + [Jenkins CLI](https://wiki.jenkins.io/display/JENKINS/Jenkins+CLI)
  + [Remote access API](https://wiki.jenkins.io/display/JENKINS/Remote+access+API)