Extending your Pipeline with Shared Libraries, Global Functions and External Code

Brent Laster

Jenkins World

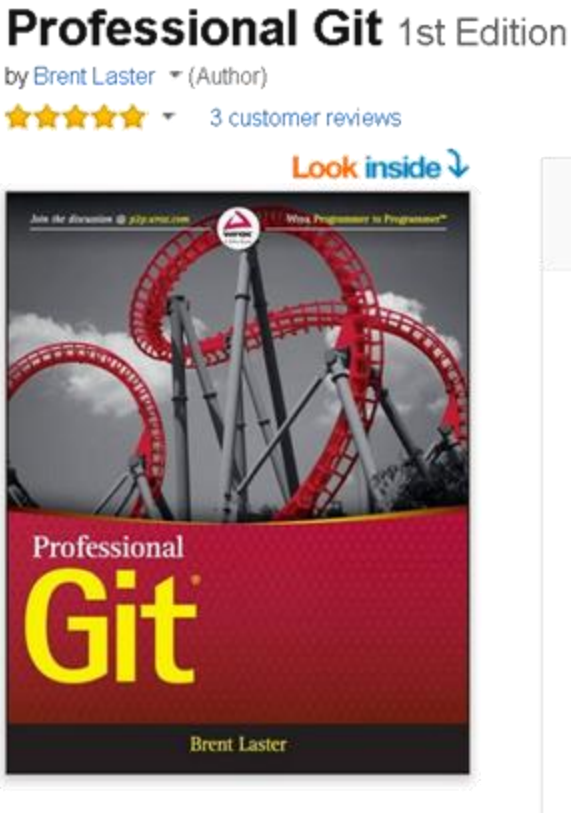
A global DevOps event

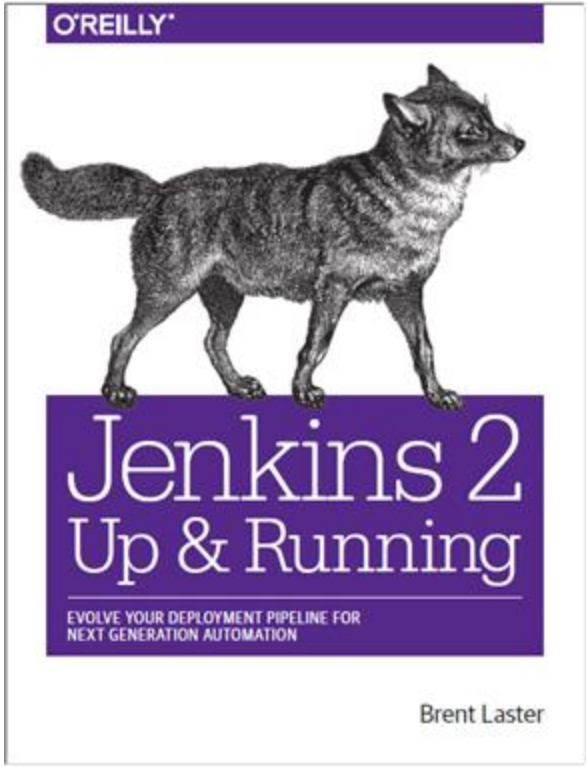
2017

About me

- Senior Manager, R&D at SAS in Cary, NC
- Global Trainer and Speaker
- Git, Gerrit, Gradle, Jenkins, Pipelines
- Author - NFJS magazine, Professional Git book, Jenkins 2 book
- Safari online training Git and Jenkins
- LinkedIn https://www.linkedin.com/in/brentlaster
- Twitter @BrentCLaster

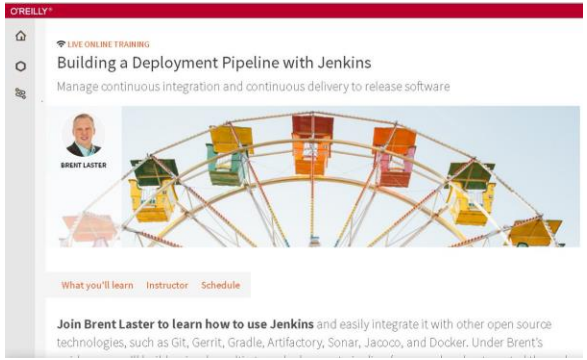
Books

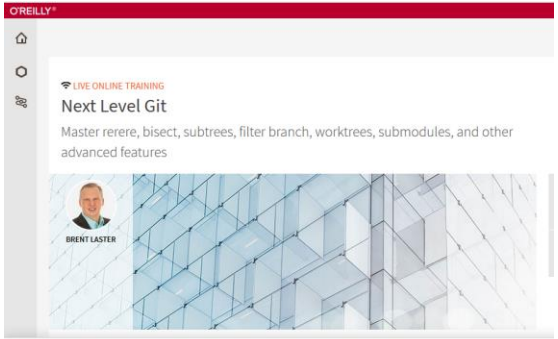

Professional Git 1st Edition
by Brent Laster (Author)
3 customer reviews
Look inside
Professional
Git
Brent Laster

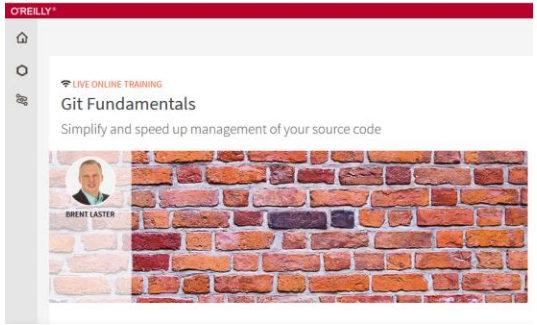

O'REILLY
Jenkins 2
Up & Running
EVOLVE YOUR DEPLOYMENT PIPELINE FOR
NEXT GENERATION AUTOMATION
Brent Laster

Training


O'REILLY
LIVE ONLINE TRAINING
Migrating Jenkins Environments to Jenkins 2
Achieve greater consistency, security, robustness, and maintainability
BRENT LASTER


O'REILLY
LIVE ONLINE TRAINING
Building a Deployment Pipeline with Jenkins
Manage continuous integration and continuous delivery to release software
BRENT LASTER
What you'll learn
Instructor
Schedule
Join Brent Laster to learn how to use Jenkins and easily integrate it with other open source technologies, such as Git, Gerrit, Gradle, Artifactory, Sonar, Jacoco, and Docker. Under Brent's


O'REILLY
LIVE ONLINE TRAINING
Next Level Git
Master rerere, bisect, subtrees, filter branch, worktrees, submodules, and other advanced features
BRENT LASTER


O'REILLY
LIVE ONLINE TRAINING
Git Fundamentals
Simplify and speed up management of your source code
BRENT LASTER

Starting Point - Jenkins Pipelines


```
node('worker_node1') { // scripted pipeline
    stage('Source') { // Get code
        // Get code from the source repository
        git url: 'http://github.com/brentlaster/greetings.git',
            branch: 'demo'
    }
    stage('Compile') { // Compile and do unit testing
        // Run Gradle
        sh 'gradle clean compileJava test'
    }
}
```

```
pipeline {  // declarative pipeline
    agent {label 'worker_node1'}
    stages {
        stage('Source') { // Get code
            steps { // Get code from the source repository
                git branch:'demo',
                  url:'http://github.com/brentlaster/greetings.git'
            }
        }
        stage('Compile') { // Compile and do unit testing
            steps {  // Run Gradle
                sh 'gradle clean compileJava test'
            }
        }
    }
}
```

What is a Shared Library?

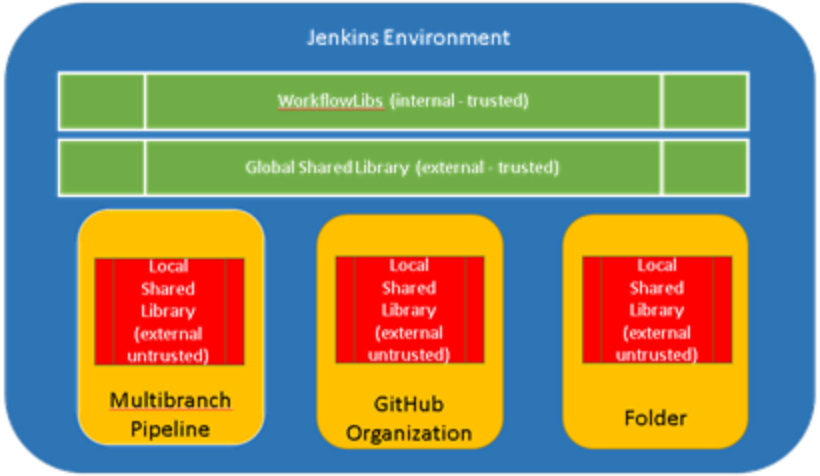
- Functions defined within a specific structure understood by Jenkins
- Stored in a source control system
- Automatically downloaded by Jenkins and made available to your pipeline on import
- Hosted models: Internal and external
- Access levels: Trusted vs. untrusted
- Scopes: global, folder, multi-branch, GitHub organization

Why use Shared Libraries?

- Centralizing functions
- Sharing common code
- Code reuse
- Abstracting out / masking complexity
- Writing code in different syntax, language

Shared Libraries in Jenkins

- Jenkins 2 has many different types of items that can be created.
- For a subset of those types, shared libraries can be defined that only apply to particular items.
- Folders, Multibranch Pipelines, and Github Organization items can each have their own "local" shared pipeline libraries.
- Allows for more dedicated, type-oriented functions to be available to those types and only to those types.
- These types of libraries are considered “untrusted” and run in the “Groovy Sandbox.”

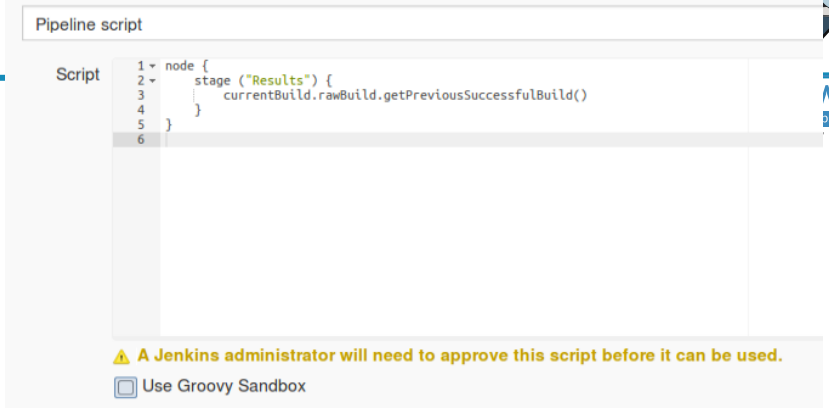


Access Levels: Trusted vs. Untrusted Libraries

- Shared libraries in Jenkins can be in one of two forms: trusted or untrusted.
- Trusted Libraries are ones that can call/use any methods in Java, the Jenkins API, Jenkins plugins, the Groovy language, etc.
  - With great power comes great responsibility - control access!
- Untrusted code is code that is restricted in what it can invoke and use.
  - Not allowed access to call same methods or use internal objects
  - Runs in the Groovy Sandbox
    - » Monitored for calling/using any methods outside of Sandbox's "safe list". If attempted, halted and must have method approved.

Script Approval in General

- Certain scripts/methods require approval
- Global list of approved actions/operations maintained by Jenkins
- Scripts created by administrators are automatically approved (and added to approved list)
- When non-admin saves a script, check is done to see if actions/operations are approved
- If not, request for approval for unapproved pieces is added to a queue in Jenkins

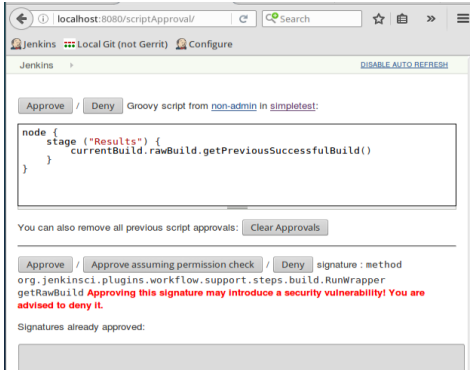


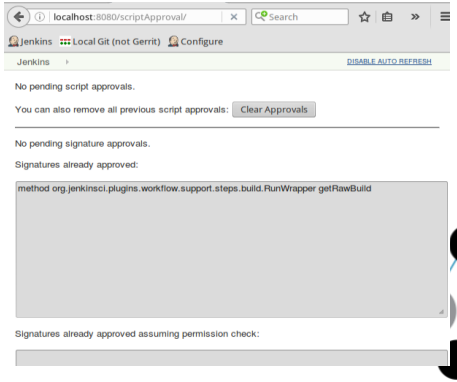
Managed files
e.g. settings.xml for maven, central managed scripts, custom files, ...

In-process Script Approval
Allows a Jenkins administrator to review proposed scripts (written e.g. in Groovy) which run inside the Jenkins process and so could bypass security restrictions. **1 scripts pending approval. 1 signatures pending approval.**

Prepare for Shutdown
Stops executing new builds, so that the system can be eventually shut down safely.

- Jenkins admin can then go to Manage Jenkins->In-process Script Approval and, if they are ok, click Approve
- Trying to run an unapproved script will fail, typically with a message indicating that it needs approval





Groovy Sandboxing

- Approval of every non-administrator script can be challenging and unworkable for teams
- Another option is Groovy Sandboxing
- Whitelist of approved methods (not considered security risks or dangerous) is maintained
- When running in Groovy Sandbox, if script only uses whitelisted, approved methods, can run regardless of user
- If unapproved method is found when run, added to queue for approval by administrator
- Done via Manage Jenkins->In-process Script Approval
- For example, if called to getJobs
- "Approve assuming permissions check" permits running as actual user, not as system call; limits operation to user permissions (for example finding job info)

Pipeline script

Script

```
node {
    stage ("Results") {
        currentBuild.rawBuild.getPreviousSuccessfulBuild()
    }
}
```

☑ Use Groovy Sandbox

```
[Pipeline] node
Running on master in /var/lib/jenkins/jobs/simpletest/workspace
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Results)
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
org.jenkinsci.plugins.scriptsecurity.sandbox.RejectedAccessException: Scripts not permitted to
use method org.jenkinsci.plugins.workflow.support.steps.build.RunWrapper getRawBuild
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.whitelists.StaticWhitelist.rejectMethod(StaticWhiteli
st.java:176)
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxInterceptor$6.reject(SandboxInterceptor
.java:243)
        at
org.jenkinsci.plugins.scriptsecurity.sandbox.groovy.SandboxInterceptor.onGetProperty(SandboxInterc
eptor.java:363)
        at org.kohsuke.groovy.sandbox.impl.Checker$4.call(Checker.java:241)
```

In-process Script Approval

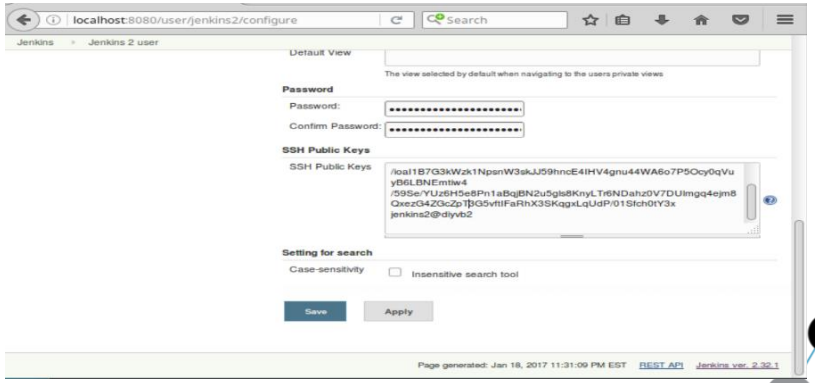
Allows a Jenkins administrator to review proposed scripts (written e.g. in Groovy) which run inside the Jenkins process and so could bypass security restrictions. **1 signatures pending approval.**

Hosted models: Internal vs. External

- Has to do with where the SCM containing the library code is hosted
- Internal - Jenkins includes an internal Git repository that can be leveraged to store internal libraries or for testing purposes.
  - Any content put in this library is trusted for all scripts.
  - But anyone pushing to it has to have the Overall/Runscripts permissions.
- Internal Git repository has a specific name - **workflowLibs.git**.
  - It can be used with Git either through ssh access or through http access.

Accessing Internal Repo via SSH

- Requirements
  - Specify the SSHD port in Jenkins via the Manage Jenkins, Configure Global Security. Use a high number here to avoid needing to use a privileged port.
  - In http://<jenkins url>/user/<userid>/configure, add the user's public ssh key in the SSH Public Keys field on that page.



- Usage

```
git clone ssh://<userid>@<system-name>:<port>/workflowLibs.git
```

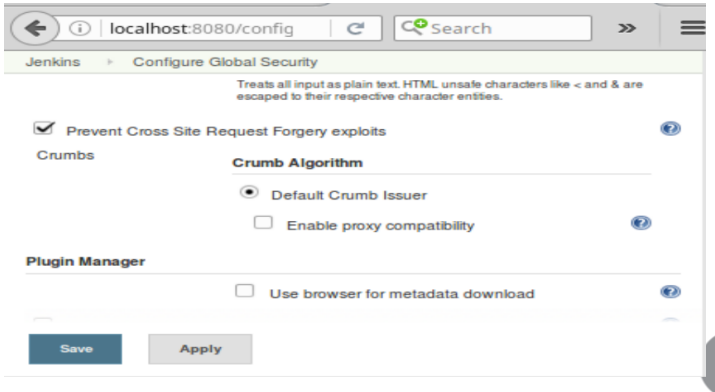
Accessing Internal Repo via http

- Assuming your local Jenkins system is running on localhost on port 8080, you can clone the repository with the command below.

git clone http://localhost:8080/workflowLibs.git

- Potential Issue

Error: RPC failed; HTTP 403 curl 22 The requested URL returned error: 403 No valid crumb was included in the request
fatal: The remote end hung up unexpectedly</p>



  - Solutions
    - » If logged out, log back in
    - » May need to disable the "prevent cross-site forgery attacks" in the Jenkins security settings (temporarily at least).

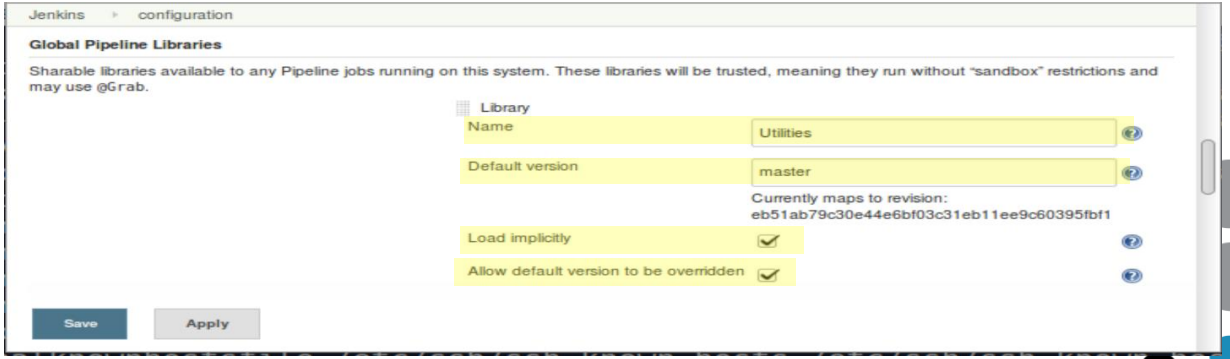
Initializing Internal Repo

- After cloning, library will be empty initially
- To start using it, initialize branch

```
cd workflowLibs
git checkout -b master
```

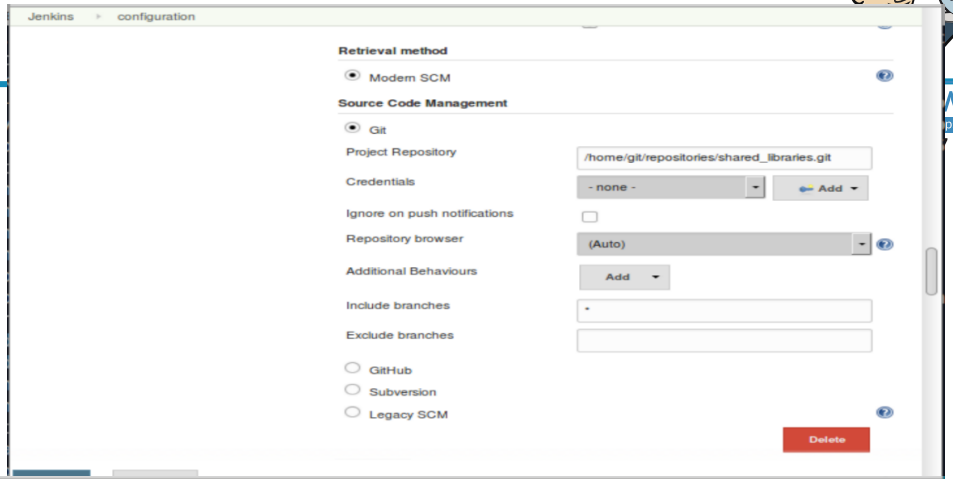
External Libraries

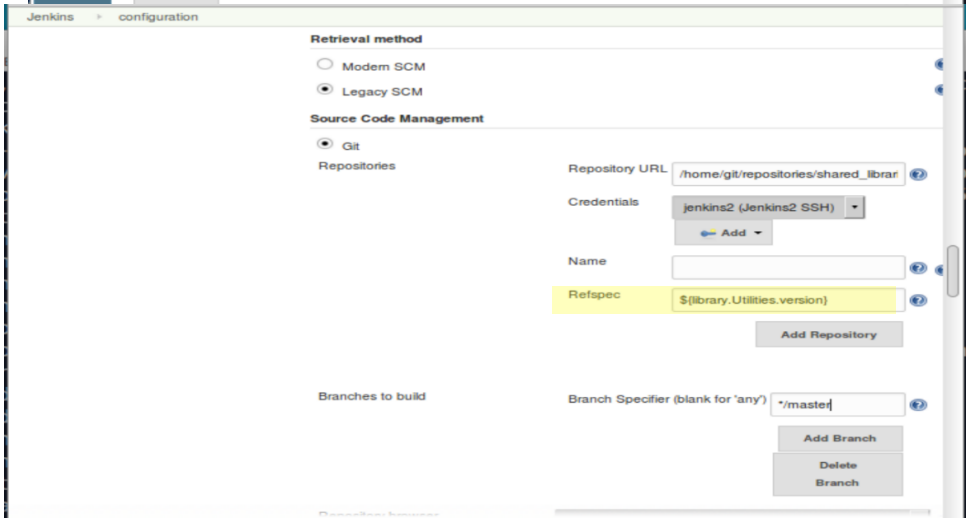
- To define an external library (one stored in a source repository separate from Jenkins) you need to provide a couple of pieces of information:
  - A name for the library (this will be used in your scripts to access it)
  - A version (default version is required if Load Implicitly is used)
  - Option for loading implicitly
  - Option for allowing default version to be overridden
  - A way to find / retrieve it from source control



External Libraries - Telling Jenkins how to load it from source control

- Two options
  - Modern SCM
    - » SCM plugin that has been updated with a new API - to handle pulling a named version
    - » Currently valid for Git, SVN, mercurial, TFS, Perforce
  - Legacy SCM
    - » Use if SCM plugin hasn't been updated
    - » Recommended to include string ${library.<your library name>.version) in specification
    - » String should get expanded to pick up particular version of content
    - » Can use any branch or tag
    - » Best to use fully qualified reference: /refs/tags/<tag.





How Jenkins makes libraries available

- Downloads at start of running job
- Example shows internal library download and external library download

Console Output

```
Started by user Jenkins 2 user
Loading library Utilities@master
 > git rev-parse --is-inside-work-tree # timeout=10
Setting origin to /home/git/repositories/shared_libraries.git
 > git config remote.origin.url /home/git/repositories/shared_libraries.git # timeout=10
Fetching origin...
Fetching upstream changes from origin
 > git --version # timeout=10
 > git fetch --tags --progress origin +refs/heads/*:refs/remotes/origin/*
 > git rev-parse master^{commit} # timeout=10
 > git rev-parse origin/master^{commit} # timeout=10
 > git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url /home/git/repositories/shared_libraries.git # timeout=10
Fetching upstream changes from /home/git/repositories/shared_libraries.git
 > git --version # timeout=10
 > git fetch --tags --progress /home/git/repositories/shared_libraries.git +refs/heads/*:refs/remotes/origin/*
Checking out Revision a0818784940689a1395c46bbc96dc22c4ba5d756 (master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f a0818784940689a1395c46bbc96dc22c4ba5d756
 > git rev-list a0818784940689a1395c46bbc96dc22c4ba5d756 # timeout=10
Loading library workflowLibs@master
 > git rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git config remote.origin.url http://localhost:8080/workflowLibs.git # timeout=10
Fetching upstream changes from http://localhost:8080/workflowLibs.git
 > git --version # timeout=10
 > git fetch --tags --progress http://localhost:8080/workflowLibs.git +refs/heads/*:refs/remotes/origin/*
 > git rev-parse refs/remotes/origin/master^{commit} # timeout=10
 > git rev-parse refs/remotes/origin/origin/master^{commit} # timeout=10
Checking out Revision ddba30eb3e70a8455b4548880aafa1900b1d2ec0 (refs/remotes/origin/master)
 > git config core.sparsecheckout # timeout=10
 > git checkout -f ddba30eb3e70a8455b4548880aafa1900b1d2ec0
 > git rev-list a2e7ce9f3c67688360ebb502a0574bbb7ffaa8ba # timeout=10
[Pipeline] node
```

Loading libraries into your script

- Internal library
  - If content, **workflowLibs** global internal library loaded automatically.
- External libraries
  - Implicit loading ("load implicitly" flag set)
    - » All loaded each time automatically
    - » No annotation needed
  - Explicit loading (scripted syntax)
    - » Requires annotation
      - » Loading default version : @Library('libname')…
      - » Loading explicit version: @Library('libname@version')…
      - » @Library annotation prepares the "classpath" of the script prior to compilation
    - » Imports (optional)
      - » Specify additional import statement for methods, classes, variables : import <items>
      - » If you have annotation and don't have import, specify a "_" after @Library('libname') so annotation has something to link to: @Library('libname')_
    - » Declarative syntax has its own "libraries" section (import used to be supported but no longer)
    - » New 'library' step as of 2.7 (from Shared Groovy Libraries plugin)
      - » Simple syntax for loading library on the fly
        - – Picks up everything from vars area
        - – More explicit syntax if trying to call method in src area since script is already compiled
        - – Allows use of parameters, variables (any interpolated value) for version

Loading Libraries Examples


```
// Load the default version of a library, all content
@Library('myLib')_

// Override the default version and load a specific version of a
library
@Library('yourLib@2.0')_

// Accessing multiple libraries with one statement
@Library(['myLib', 'yourLib@master'])_

// importing specific methods
import static org.demo.Utilities.*

// Annotation with import
@Library('myLib@1.0')
import static org.demo.Utilities
```

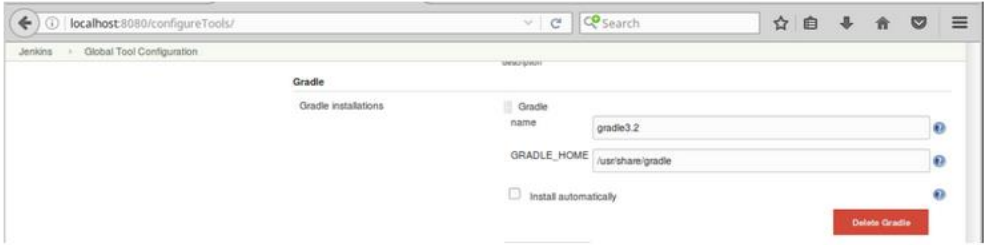
```
// Declarative syntax

pipeline {
    agent any
    libraries {
      lib("mylib@master")
      lib("alib")
    }
    stages {
      ...

// New library step – as of 2.7
//  From Shared Groovy Libraries Plugin
//  access anything from vars/ area
library ‘my-shared-lib’
library ‘my-shared-lib@version’
library ‘my-shared-lib@$LIB_VERSION

// using static class methods from src/ area
library(‘my-shared-lib’).org.demo.pipe.Utils2.aStaticMethod()
```

Sample Library Routine

- Simple routine to invoke gradle build with timestamps

```
timestamps {
    <path-to-gradle-home>/bin/gradle <tasks>
}
```

- **timestamps** is a Jenkins Pipeline DSL step .
- **timestamps** closure here simply tells Jenkins to add timestamps to the console output for this part of our pipeline (the gradle build step)
- Can use gradle version as pointed to by tool configuration in Jenkins



- Actual code

```
timestamps {
    sh "${tool 'gradle3.2'}/bin/gradle ${tasks}"
}
```

Shared Library Structure

- Repository Structure

```
.
├── resources
├── src
│   └── org
│       └── foo
│           └── Utilities.groovy
└── vars
    └── mailUser.groovy
```

  - src directory is added to classpath when pipeline is executed
  - vars directory is for global variables or scripts accessible from pipeline scripts; can have corresponding txt file with documentation
  - resources directory can have non-groovy resources that get loaded from a “libraryResource” step in external libraries

src

- Intended to be setup with groovy files in the standard Java directory structure (i.e. src/org/foo/bar.groovy).
- Added to the classpath when pipelines are executed.
- Any Groovy code valid to use here.
- Generally invoke some kind of pipeline processing using the available pipeline steps.
- Several options for how to implement the step calls within the library and how to invoke them from the script.

src example 1

Jenkins World
A global DevOps event
2017

- You can create a simple method, not enclosed by a class
- Can invoke pipeline steps

*buildUtils.groovy

```
// org/demo/buildUtils.groovy
package org.demo
def timedGradleBuild(tasks) {
    timestamps {
        sh "${tool 'gradle3.2'}/bin/gradle ${tasks}"
    }
}
```

- Stored in library, loaded implicitly
- This can be invoked in a pipeline via

```
@Library('Utilities')_
node('worker_node1') { // scripted pipeline
    stage('Source') { // Get code
        // Get code from the source repository
        git url: 'http://github.com/brentlaster/greetings.git',
            branch: 'demo'
    }
    stage('Compile') { // Compile and do unit testing
        // Run Gradle
        def myUtils = new org.demo.buildUtils()
        myUtils.timedGradleBuild 'clean build'
    }
}
```

src example 2

- Create an enclosing class (facilitates things like defining a superclass)
- Get access to all of the DSL steps by passing the steps object to a method
  - Passed in a constructor or in a method of the class
  - Since enclosed in a class, the class must implement Serializable to support saving the state if the pipeline is stopped or restarted.
  - Also works for environment variables
- Invoked as shown below

BuildUtils2.groovy

```
// org/demo/BuildUtils2
package org.demo
class BuildUtils2 implements Serializable {
    def env
    def steps
    BuildUtils2(env, steps) {
        this.env = env
        this.steps = steps
    }
    def timedGradleBuild(tasks) {
      def gradleHome = steps.tool 'gradle3.2'
      steps.sh "echo Building for ${env.BUILD_TAG}"
      steps.timestamps {
          steps.sh "${steps.tool 'gradle3.2'}/bin/gradle ${tasks}"
      }
    }
}
```

```
@Library('Utilities')
import org.demo.BuildUtils2

def bldtools = new BuildUtils2(env, steps)
node ('worker_node1') {
    stage('Source') { // Get code
        // Get code from the source repository
        git url: 'http://github.com/brentlaster/greetings.git',
            branch: 'demo'
    }
    stage('Compile') { // Compile and do unit testing
        // Run Gradle
        bldtools.timedGradleBuild("clean build")
    }
}
```

src example 3

- Can also use static method and pass in script object - already has access to everything

BuildUtils3.groovy

```
// org.demo.BuildUtils3
package org.demo
class BuildUtils3 {
   static def timedGradleBuild(script, tasks) {
     def gradleHome = script.tool 'gradle3.2'
     script.sh "echo Building for ${script.env.BUILD_TAG}"
     script.timestamps {
        script.sh "${script.tool 'gradle3.2'}/bin/gradle ${tasks}"
     }
   }
}
```

- Can be invoked similarly - note “import static”

```
@Library('Utilities') import static org.demo.BuildUtils3.*
node ('worker_node1') {
    stage('Source') { // Get code
        // Get code from the source repository
        git url: 'http://github.com/brentlaster/greetings.git',
            branch: 'demo'
    }
    stage('Compile') { // Compile and do unit testing
        // Run Gradle
        timedGradleBuild this, 'clean build'
    }
}
```

vars

- Area for hosting scripts that define variables that you want to access in the pipeline
- Scripts are instantiated as singletons when accessed
- Basename should be a valid Groovy identifier
- Can have a <basename>.txt file that contains help or other documentation for the variable. This file can use HTML or Markdown.

vars example 1 - set of methods

- Can define set of methods in a file
- "cmd" and "cmdOut" here are not fields. These are objects created on demand
- Use it in script

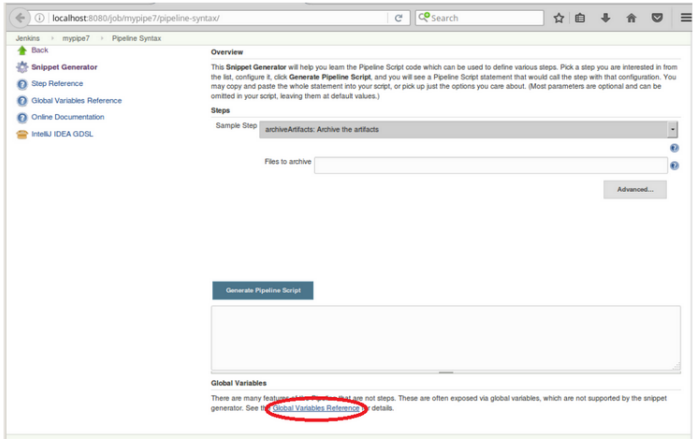
```
// vars/timedCommand.groovy
def setCommand(commandToRun) {
    cmd = commandToRun
}

def getCommand() {
    cmd
}

def runCommand() {
   timestamps {
      cmdOut = sh (script:"${cmd}", returnStdout:true).trim()
   }
}

def getOutput() {
   cmdOut
}
```

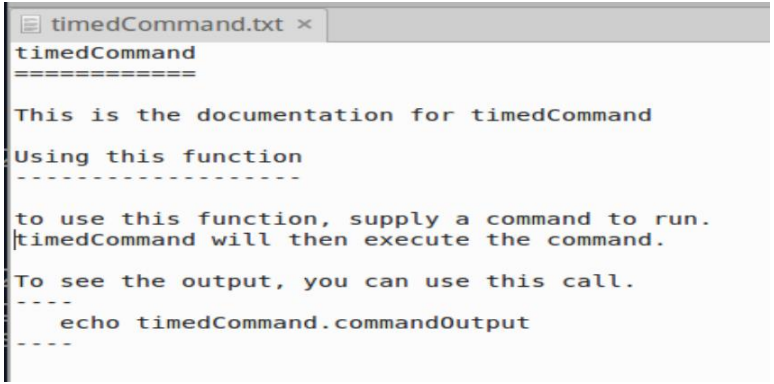
```
node {
      timedCommand.cmd = 'ls -la'
      echo timedCommand.cmd
      timedCommand.runCommand()
      echo timedCommand.getOutput()
}
```

```
[Pipeline] node
Running on worker_node2 in /home/jenkins2/worker_node
/jw17-vars-example-1
[Pipeline] {
[Pipeline] echo
ls -la
[Pipeline] timestamps
[Pipeline] {
[Pipeline] sh
17:20:44 [jw17-vars-example-1] Running shell script
17:20:44 + ls -la
[Pipeline] }
[Pipeline] // timestamps
[Pipeline] echo
total 8
drwxrwxr-x 2 jenkins2 jenkins2 4096 Aug 24 17:20 .
drwxrwxr-x 7 jenkins2 jenkins2 4096 Aug 24 17:20 ..
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

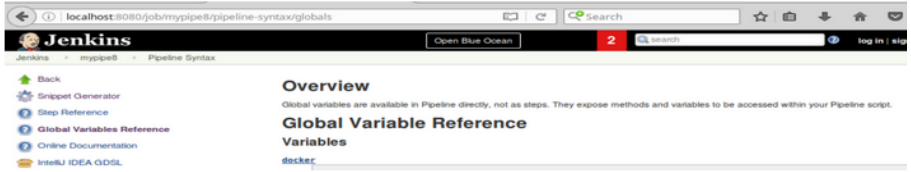
vars - automatic documentation

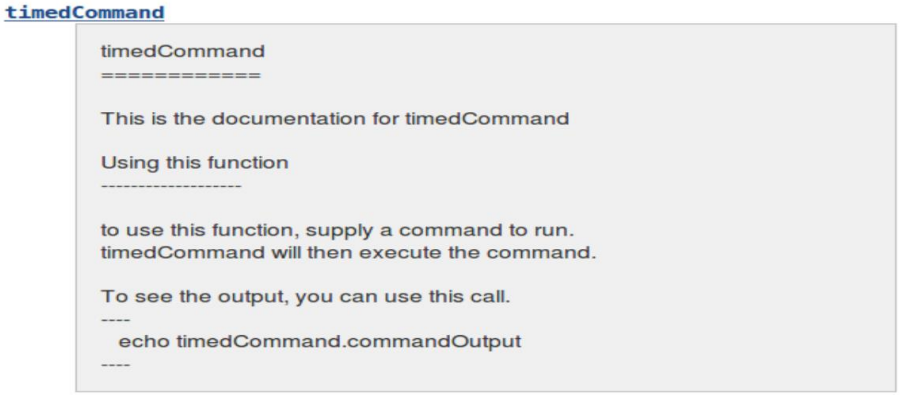
- Create corresponding .txt file >





After run, access Global Variables reference





- Documentation is present >

vars example 2 - Creating global variables that act like steps

- use “call” structure to create function like DSL step

```
// vars/timedCommand2

def call (String cmd, String logFilePath) {
   timestamps {
      cmdOutput = sh (script:"${cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile file: "${logFilePath}", text: "${cmdOutput}"
}
```

```
[Pipeline] node
Running on worker_node3 in /home/jenkins2/worker_node
[Pipeline] {
[Pipeline] timestamps
[Pipeline] {
[Pipeline] sh
18:07:37 [jw17-vars-example-2] Running shell script
18:07:38 + ls -la
[Pipeline] }
[Pipeline] // timestamps
[Pipeline] echo
total 8
drwxrwxr-x 2 jenkins2 jenkins2 4096 Aug 24 18:07 .
drwxrwxr-x 6 jenkins2 jenkins2 4096 Aug 24 18:07 ..
[Pipeline] writeFile
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

- use in pipeline script

```
timedCommand2 'ls -la', 'listing.log'
```

vars example 3 - Passing a closure

- Can define method to take closure and do some operation; useful to do things like run on a node >
- Invoke from script as below

```
@Library('Utilities')_
node {
  timedCommandLinux {
      echo "Start"
      sleep 5
      echo "End"
  }
}
```

```
// vars/timedCommandLinux.groovy

def call(Closure commands) {
   node('linux') {
       timestamps {
           commands()
       }
   }
}
```

```
[Pipeline] node
Running on worker_node1 in /h
[Pipeline] {
[Pipeline] node
Running on linux in /home/jen
[Pipeline] {
[Pipeline] timestamps
[Pipeline] {
[Pipeline] echo
18:31:26 Start
[Pipeline] sleep
18:31:26 Sleeping for 5 sec
[Pipeline] echo
18:31:31 End
[Pipeline] }
[Pipeline] // timestamps
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

vars example 4 - DSL call with named parameters

- uses map (settings = [:])
- delegate allows using the values passed in in our mapping
- use only valid pipeline steps!
- call with simple syntax (named parameters)

```
// vars/timedCommand4.groovy

def call(body) {
    // collect assignments passed in into our mapping
    def settings = [:]
    body.resolveStrategy = Closure.DELEGATE_FIRST
    body.delegate = settings
    body()

    // now, time the commands
   timestamps {
      cmdOutput = sh (script:"${settings.cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile file: "${settings.logFilePath}", text: "${cmdOutput}"
}
```

```
@Library('Utilities')_
node {
  timedCommand4 {
      cmd = 'ls -la'
      logFilePath = 'log.out'
  }
}
```

resources

- Files in this directory can be non-groovy
- Can be loaded via the libraryResource step in an external library; loaded as a string
- Intended to allow external libraries to load up additional non-groovy files they may need
- Examples: datafile (xml, json, etc.)
- Syntax: `def datafile = libraryResource 'org/conf/data/lib/datafile.ext'`
- libraryResource can also be used to load up any resource needed in a script (requires caution!)

```
def myExternalScript = libraryResource 'externalCommands.sh'
sh myLatestScript
```

- can be useful to separate out non-pipeline code or programmatically specify different files to load based on conditions

Using 3rd-party libraries

- Shared libraries can also make use of third-party libraries using the @Grab annotation
- @Grab annotation provided through the Grape dependency manager that is built in to Groovy
- Allows you to pull in any dependency from a Maven repository, such as Maven Central
- Can be done from trusted libraries – doesn't work in Groovy Sandbox
- Invocation

```
node {
        timedCommand5("sleep 10")
}
```

- Output

```
[Pipeline] node
Running on worker in /home/jenkins2/worker_node1/workspace/mypipe11
[Pipeline] {
[Pipeline] echo
The process took 10.009 seconds.

[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

```
// vars/timedCommand5.groovy

@Grab('org.apache.commons:commons-lang3:3.4+')
import org.apache.commons.lang.time.StopWatch

def call(String cmdToRun) {
      def sw = new StopWatch()
      def proc = "$cmdToRun".execute()
      sw.start()
      proc.waitFor()
      sw.stop()
      println( "The process took ${(sw.getTime
()/1000).toString()} seconds.\n")
}
```

Loading Code Directly

- Similar to shared libraries, but not loaded from source control

```
def call(String cmd, String logFilePath) {
   timestamps {
      cmdOutput = sh (script:"${cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile file: "${logFilePath}", text: "${cmdOutput}"
}
return this;
```

- Note “return this” – necessary to make sure correct scope is returned for load
- Example use: Load and invoke

```
node {
   def myProc = load '/home/diyuser2/timedCommand2.groovy'
   myProc 'ls -la', 'command.log'
}
```

Declarative Pipelines and Library Directive

- Declarative syntax has “libraries” directive
- lib statement defines libraries and versions to load similar to @Library annotation
- Functions must be callable in declarative syntax
- Usually works better to use global variables to avoid having to try to use something from classpath
- Per Cloudbees: “Unless you need to create one class with a bunch of static methods it is easier to use global vars in the /vars directory instead of classes in the /src directory.”

```
pipeline {  // declarative pipeline
    agent {label 'worker_node1'}
    libraries {
       lib('Utilities')
    }
    stages {
        stage('Source') { // Get code
            steps { // Get code from the source repository
                git branch:'demo',
                  url:'http://github.com/brentlaster/greetings.git'
            }
        }
        stage('Compile') { // Compile and do unit testing
           steps {  // Run Gradle
             timedCommand2 'gradle clean compileJava test','build.log'
           }
        }
    }
}
```

Library step

- New as of 2.7 with Pipeline Shared Groovy Libraries plugin
- No annotation or section necessary
- Simple syntax for vars area = library('name')
- For src area (scripted pipeline only) can fully qualify name of static method to call = library('name').<lib path>.<static method>
- Can also use computed value for version = library "name@VARIABLE_VALUE"

```
stage('Compile') { // Compile and do unit testing
        // Run Gradle
        library('Utilities')
        timedCommand2 'gradle clean build', 'build.log'
    }
```

```
stage('Compile') { // Compile and do unit testing
  steps {  // Run Gradle
    library 'Utilities'
    timedCommand2 'gradle clean compileJava test','build.log'
  }
}
```

```
node ('worker_node1') {
    stage('Source') { // Get code
        // Get code from the source repository
        git url: 'http://github.com/brentlaster/greetings.git',
            branch: 'demo'
    }
    stage('Compile') { // Compile and do unit testing
        // Run Gradle
        library('Utilities').org.demo.BuildUtils3.timedGradleBuild \
          this, 'clean build'
    }
}
```

Loading code from external SCM

- Requires Remote Loader Plugin*
- Provides fileLoader DSL

fileLoader

Provides methods for loading Pipeline objects from remote sources. More info about available methods and their parameters: the plugin's Wiki page

Available methods

The variable provides following methods:

- **fromGit(String libPath, String repository, String branch, String credentialsId, String labelExpression)** - loading of a single Groovy file from the specified Git repository
- **withGit(String repository, String branch, String credentialsId, String labelExpression)** - wrapper closure for multiple files loading from a same Git repo
- **fromSVN(String libPath, String repository, String credentialsId, String labelExpression)** - loading of a single Groovy file from the specified SVN repository
- **withSVN(String repository, String credentialsId, String labelExpression)** - wrapper closure for multiple files loading from a same SVN repo
- **load(String libPath)** - loading of an object from a Groovy file specified by the relative path. Also can be used within `withGit()` closure to load multiple objects at once

brentlaster / utilities

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki

Branch: master — utilities / jenkins / pipeline / timedCommand.groovy

sasbcl fix capitalization

2 contributors

9 lines (8 sloc) | 231 Bytes

```
public timedCommand(String cmd, String logFilePath) {
   timestamps {
      cmdOutput = sh (script:"${cmd}", returnStdout:true).trim()
   }
   echo cmdOutput
   writeFile file: "${logFilePath}", text: "${cmdOutput}"
}
return this;
```

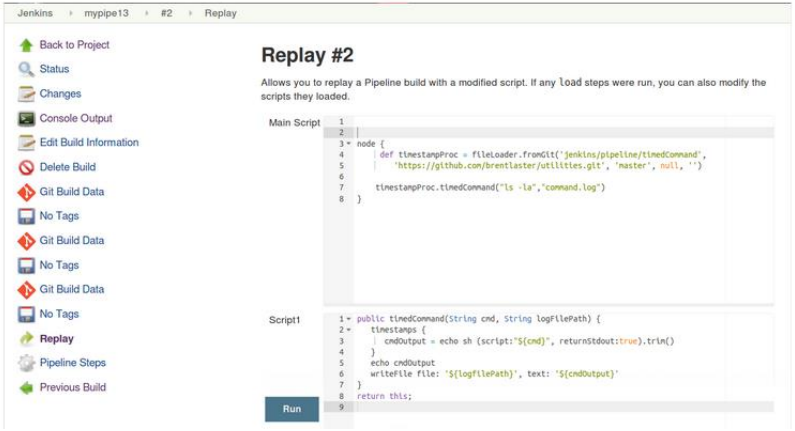
- Invoke as below from script

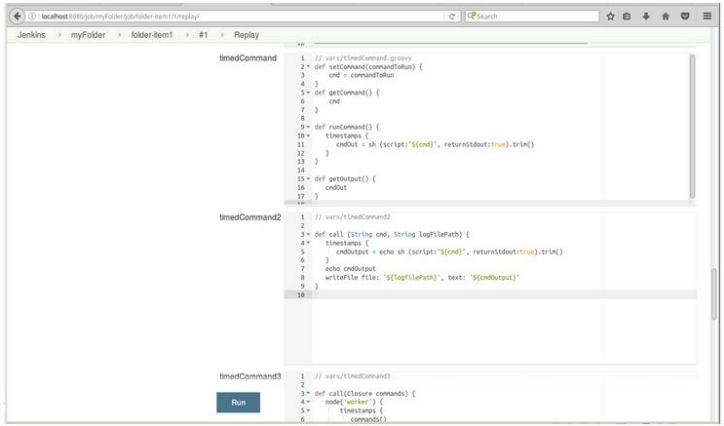
```
def timestampProc = fileLoader.fromGit('jenkins/pipeline/timedCommand'
  |'https://github.com/brentlaster/utilities.git', 'master', null, '')

timestampProc.timedCommand("ls -la","command.log")
```

- Written prior to shared-libraries and not guaranteed to be maintained.

Replay for Libraries

- Only available for untrusted libraries
- Can be invoked after successful run





- If multiple libraries used and untrusted, all are loaded
- Reminder: shared libraries at scope of folder, multibranch, Github organization are untrusted

That’s all - thanks!

@BrentCLaster
Jenkins 2 Up and Running – O’Reilly – Spring 2018

Jenkins World

A global DevOps event

2017