



Introduction to Declarative Pipeline

In the beginning there was
the Freestyle job...



freestylingit Config [Jenkins]

Craig

05202017259.jenkins.beedemo.net/job/freestylingit/configure

☆ a k G

Managed Master

Open Blue Ocean

search

cvitter | log out

Jenkins

05202017259

freestylingit

General

Source Code Management

Build Triggers

Build Environment

Build

Post-build Actions

Project name

freestylingit

Description

[Plain text] [Preview](#)

☐ Discard old builds

☐ GitHub project

☐ This project is parameterized

☐ Throttle builds

☐ Disable this project

☐ Execute concurrent builds if necessary

☒ Restrict where this project can be run

Label Expression

Advanced...

Source Code Management

None

Git

Save

Apply

What's Wrong With Freestyle Jobs?

While the Freestyle job type has served the Hudson/Jenkins community well for years it has some major issues including:

- **UI Bound** - The configuration of a job is limited to what can be expressed via the limits of the Jenkins' UI and doesn't allow for building complicated workflows with features like:
 - Control over where builds are executed
 - Flow control (if-then-else, when, try-catch-finally)
 - Ability to run steps in parallel
- **Not Auditable** - The creation and editing of jobs isn't auditable without using additional plugins

Enter Jenkins Pipeline...



What is a Jenkins Pipeline?

Jenkins Pipeline (formerly known as Workflow) was introduced in **2014** and built into Jenkins 2.0 when it was released.

Pipelines are:

- A **Job** type - The configuration of the job and steps to execute are defined in a script (**Groovy** or **Declarative** based with a Domain Specific Language) that can be stored in an external SCM
- **Auditable** - changes can be audited via your SCM
- **Durable** - can keep running even if the master fails
- **Distributable** - pipelines can be run across multiple agents including execution of steps in parallel
- **Pausable** - can wait for user input before proceeding
- **Visualizable** - enables status-at-a-glance dashboards like the built in Pipeline Stage View and Blue Ocean

Why You Should Use Declarative Instead of Scripted

While Declarative Pipelines use the same execution engine as Scripted pipelines Declarative adds the following benefits:

- **Easier to Learn** - the Pipeline DSL (Domain Specific Language) is more approachable than Groovy making it quicker to get started using
- **Docker Pipeline Integration** - ability to execute builds within one or more docker containers is built into Declarative without requiring additional plugins
- **Syntax Checking** - Declarative syntax adds the following types of syntax checking that don't exist for Scripted pipelines:
 - Immediate runtime syntax checking with explicit error messages.
 - API and CLI based file linting
- **Round Trip Visual Editing** - The Blue Ocean pipeline editor can read and write Declarative syntax (but not Scripted)

Creating a Pipeline



The Simplest Declarative Jenkins File vs Scripted

```
pipeline {  
  agent any  
  
  stages {  
    stage('Say Hello') {  
      steps {  
        echo 'Hello World!'  
      }  
    }  
  }  
}
```

```
node {  
  echo 'Hello World!'  
}
```

Pipeline

Definition Pipeline script

Script

```
1 pipeline {  
2   agent any  
3  
4   stages {  
5     stage('Say Hello') {  
6       steps {  
7         echo 'Hello World'  
8       }  
9       post {  
10        always {  
11          echo "Running ${env.JOB_NAME} (${env.BUILD_ID}) on ${env.JENKINS_URL}"  
12        }  
13      }  
14    }  
15  }  
16 }
```

☒ Use Groovy Sandbox

[Pipeline Syntax](#)

Save Apply

Hands On Exercise 1.1

Create a Simple Declarative Pipeline

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md>

The Jenkins Snippet Generator

The screenshot shows the Jenkins Snippet Generator web interface. The browser address bar indicates the URL is `localhost:8080/job/BasicPipeline/pipeline-syntax/`. The Jenkins header shows 'CloudBees Jenkins Enterprise' with a search bar and user links for 'admin' and 'log out'. The breadcrumb trail is 'Jenkins > BasicPipeline > Pipeline Syntax'. The left sidebar contains links: 'Back', 'Snippet Generator' (active), 'Step Reference', 'Global Variables Reference', 'Online Documentation', and 'IntelliJ IDEA GDSL'. The main content area has an 'Overview' section explaining the tool's purpose. Below this is a 'Steps' section with a dropdown menu showing 'stash: Stash some files to be used later in the build'. The configuration section includes fields for 'Name' (filled with 'test.lib'), 'Includes', 'Excludes', and a checked 'Use Default Ant Excludes' checkbox. A 'Generate Pipeline Script' button is present. The output area shows the generated snippet: `stash 'test.lib'`. The bottom status bar shows the path `localhost:8080/job/BasicPipeline/pipeline-syntax/globals` and the 'Global Variables' section header.

CloudBees Jenkins Enterprise

Open Blue Ocean 1 search admin log out

Jenkins > BasicPipeline > Pipeline Syntax

Back

Snippet Generator

Step Reference

Global Variables Reference

Online Documentation

IntelliJ IDEA GDSL

Overview

This **Snippet Generator** will help you learn the Pipeline Script code which can be used to define various steps. Pick a step you are interested in from the list, configure it, click **Generate Pipeline Script**, and you will see a Pipeline Script statement that would call the step with that configuration. You may copy and paste the whole statement into your script, or pick up just the options you care about. (Most parameters are optional and can be omitted in your script, leaving them at default values.)

Steps

Sample Step **stash: Stash some files to be used later in the build**

Name **test.lib**

Includes

Excludes

Use Default Ant Excludes ☒

Generate Pipeline Script

stash 'test.lib'

Global Variables

Pipeline Replay

Replay #13 [Jenkins]

localhost:8080/job/BasicPipeline/13/replay/

CloudBees Jenkins Enterprise

Open Blue Ocean

1

search

admin | log out

Jenkins > BasicPipeline > #13 > Replay

Back to Project

Status

Changes

Console Output

Edit Build Information

Delete Build

Replay

Pipeline Steps

Previous Build

Replay #13

Allows you to replay a Pipeline build with a modified script. If any load steps were run, you can also modify the scripts they loaded.

Main Script

```
1 pipeline {
2   agent any
3
4   stages {
5     stage('Say Hello') {
6       steps {
7         echo 'Hello World!'
8       }
9       post {
10        always {
11          echo "Running ${env.JOB_NAME} (${env.BUILD_ID}) on ${env.JENKINS_URL}"
12        }
13      }
14    }
15  }
16 }
```

[Pipeline Syntax](#)

Run

CloudBees Jenkins Platform

Page generated: May 30, 2017 4:57:06 PM [CloudBees Jenkins Enterprise 2.46.2.1-rolling](#)

Blue Ocean Editor

Create Pipeline

Where do you store your code?

Github

Github Enterprise

Bitbucket

Bitbucket Server

Git

Connect to Github

Jenkins needs a access key to authorize itself with Github. [Learn how to create an access key.](#)

Connect

In which Github organization are your repositories located?

Build any repository that contains a Jenkinsfile?

Completed

Create Pipeline

Discard Changes

Save

Start

Build

Build

Sit Around

+

Test

Test

Browser test

+

Deploy

+

cloudbees

© 2017 CloudBees, Inc. All Rights Reserved.

13

Beyond Hello World



Specifying Agents

```
pipeline {  
  agent any  
  stages { ... }  
}
```

```
pipeline {  
  agent {  
    docker { image 'maven:3.3-jdk-8' }  
  }  
  stages { ... }  
}
```

```
pipeline {  
  agent none  
  stages {  
    stage('Build') {  
      agent any  
      steps {  
        sh 'make'  
        stash includes: '**/target/*.jar', name:  
'app'  
      }  
    }  
    stage('Test') {  
      agent { label 'linux' }  
      steps {  
        unstash 'app'  
        ...  
      }  
    }  
  }  
}
```

Hands On Exercise 1.2

Define a Docker Based Agent

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-12>

Environmental Variables

```
pipeline {
  agent any

  environment {
    A_VALUE = 'Some Value'
  }

  stages {
    stage('Build') {
      steps {
        echo "${A_VALUE}"
        echo "${env.BUILD_ID}"
        echo "${currentBuild.result}"
      }
    }
  }
}
```

Credentials

```
pipeline {
  agent any

  environment {
    SONAR = credentials('sonar')
  }

  stages {
    stage('Build') {
      steps {
        echo "${SONAR_USR}"
        echo "${SONAR_PSW}"
      }
    }
  }
}
```

<http://localhost:8080/job/BasicPipeline/pipeline-syntax/globals>

Hands On Exercise 1.3

Add Environmental Variables

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-13>

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!"
      }
    }
  }
}
```

Hands On Exercise 1.4

Capture Input Parameters

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-14>

Capturing User Input

```
stage('Deploy') {  
    steps {  
        input 'Should I Deploy?'  
    }  
}
```

```
[Pipeline] { (Deploy)  
[Pipeline] input  
Should I Deploy?  
Proceed or Abort  
Approved by admin  
[Pipeline] }
```

```
stage('Input') {  
    steps {  
        script {  
            returnValue = input message:  
                'Need some input',  
                parameters: [string(defaultValue: '',  
                    description: '',  
                    name: 'Give me a value')]  
        }  
        echo "${returnValue}"  
    }  
}
```



Steps - Deploy

Wait for interactive input 1m 54s

Should I Deploy?

Proceed Abort

Steps - Input

Wait for interactive input 41s

Need some input

Proceed Abort

Retries and Timeouts

```
stage('Deploy') {  
    steps {  
        retry(3) {  
            sh './flakey-deploy.sh'  
        }  
  
        timeout(time: 3, unit: 'MINUTES') {  
            sh './health-check.sh'  
        }  
    }  
}
```

```
stage('Deploy') {  
    steps {  
        timeout(time: 3, unit: 'MINUTES') {  
            retry(5) {  
                sh './flakey-deploy.sh'  
            }  
        }  
    }  
}
```

Hands On Exercise 1.5

Capture User Input During Run Time

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-15>

Post Actions

```
pipeline {
  agent any

  stages { ... }
  post {
    always {
      echo 'I always run!'
    }
    success { ... }
    failure { ... }
    aborted { ... }
    unstable { ... }
    changed { ... }
  }
}
```

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
      }
      post {
        always {
          echo 'I always run!'
        }
        success { ... }
      }
    }
  }
}
```


Hands On Exercise 1.6

Handling Post Actions

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-16>

Shared Libraries

```
// Groovy Library located in
// github.com/example/CraigsLibs/vars/helloWorld.groovy
def call(name) {
    echo "Hello ${name}"
    echo "Have a great day!"
}
```

The screenshot shows the Jenkins configuration page for a shared library named 'CraigsLibs'. The 'Name' field is 'CraigsLibs' and the 'Default version' is 'master'. Below this, it states 'Currently maps to revision: f45ffa4f941692e971fbb9618b6034174ed60a1e'. The 'Load implicitly' checkbox is checked. Under 'Retrieval method', 'Modern SCM' is selected. In the 'Source Code Management' section, 'GitHub' is selected as the provider. The 'Owner' is 'cvitter', the 'Scan credentials' is 'cvitter/***** (GitHub Token)', and the 'Repository' is 'jenkins-pipeline-examples'. There are 'Advanced...' and 'Delete' buttons at the bottom right of the configuration area.

```
library 'CraigsLibs'

pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                helloWorld("Bob")
            }
        }
    }
}
```

Hands On Exercise 1.7

Using Shared Libraries

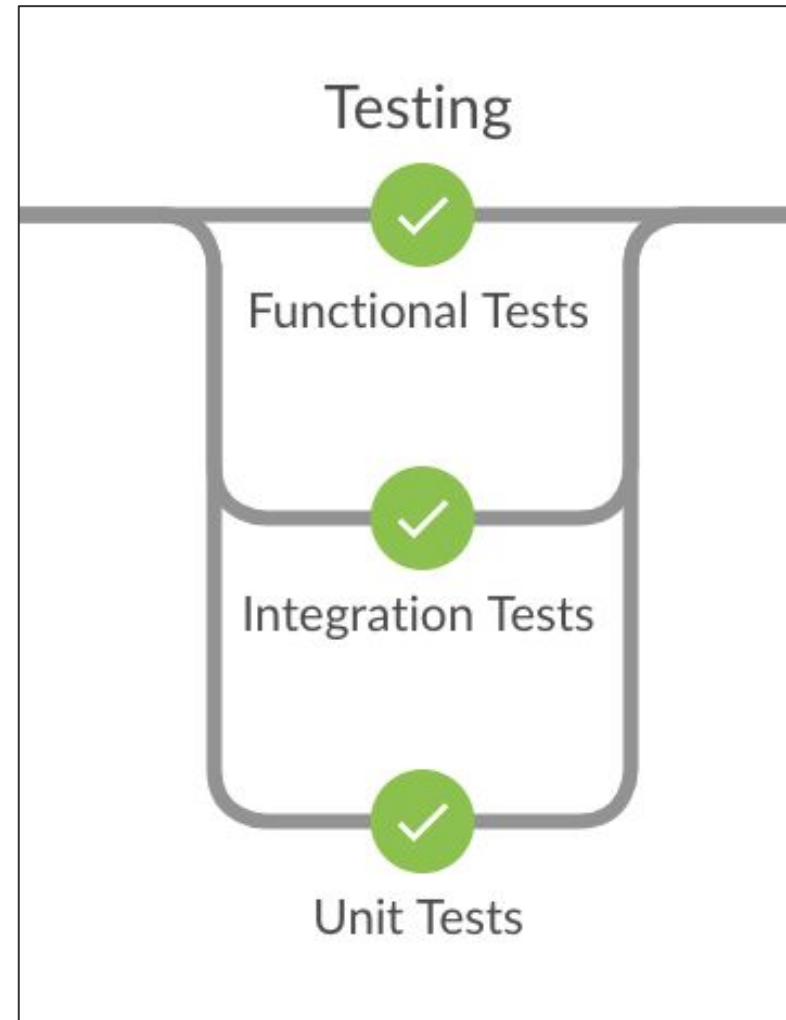
<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-17>

Conditional Flow Control

```
stage('Deploy') {  
    when {  
        expression {  
            currentBuild.result == null || currentBuild.result == 'SUCCESS'  
        }  
    }  
    steps {  
        ...  
    }  
}  
  
stage('Build Master') {  
    when {  
        branch 'master'  
    }  
    steps {  
        ...  
    }  
}
```

Executing Steps in Parallel

```
pipeline {
  agent any
  stages {
    stage("Testing") {
      parallel {
        stage("Unit Tests") {
          agent { docker 'openjdk:7-jdk-alpine' }
          steps {
            sh 'java -version'
          }
        }
        stage("Functional Tests") {
          agent { docker 'openjdk:8-jdk-alpine' }
          steps {
            sh 'java -version'
          }
        }
        stage("Integration Tests") {
          steps {
            sh 'java -version'
          }
        }
      }
    }
  }
}
```



Hands On Exercise 1.8

Executing Parallel Stages

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-18>

Scripted Blocks

```
pipeline {  
  agent any  
  
  environment {  
    APP_VERSION = "0.0.1"  
  }  
  
  stages {  
    stage('Parse POM') {  
      steps {  
        script {  
          pom = readMavenPom file: 'pom.xml'  
          APP_VERSION = pom.version  
        }  
      }  
    }  
  }  
}
```



Building a Multibranch Pipeline



What is a Multibranch Pipeline?

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

A **Github Organization** or **Bitbucket Organization** scans for projects that have a Jenkinsfile and creates a **Multibranch Pipeline** project for each one it finds.

Hands On Exercise 2.1

Fork The sample-rest-server Repo

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-02.md#exercise-21>

Hands On Exercise 2.2

Create a Github Organization Project

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-02.md#exercise-22>

Hands On Exercise 2.3

Add Branch Based Flow Control

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-02.md#exercise-23>

Hands On Exercise 2.4

Handling Feature Branches and Pull Requests

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-02.md#exercise-24>

Enterprise Only Pipeline Features



Checkpoints*

```
stage("Checkpoint") {  
    agent none  
    steps {  
        checkpoint 'Completed Docker Image Testing'  
    }  
}
```

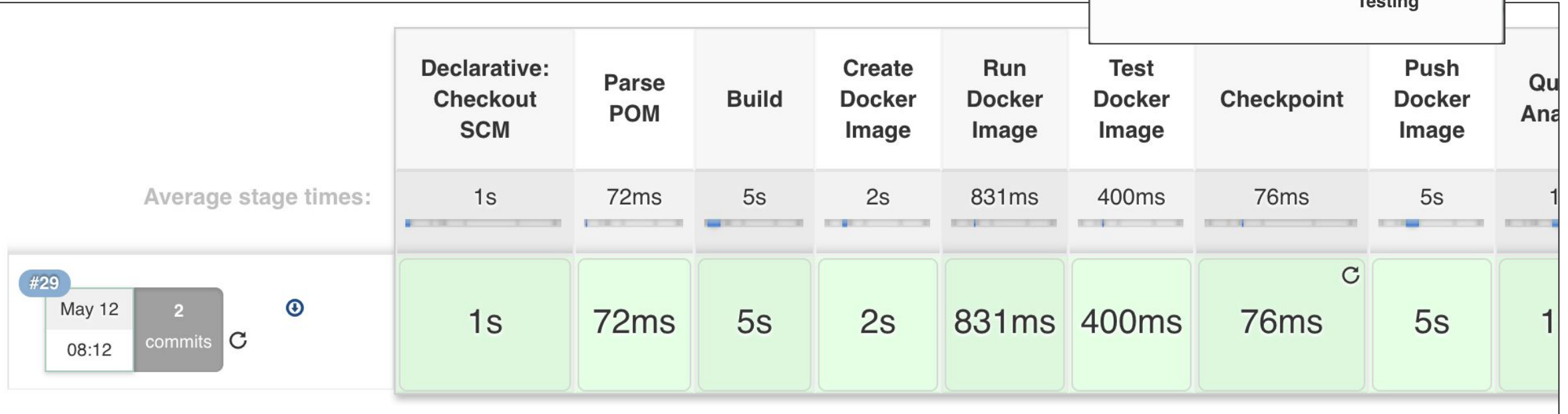
Resume ×

This Pipeline run can be restarted from the following Checkpoint(s)

Delete

Restart

Completed Docker Image Testing



Hands On Exercise 3.1

Create a Checkpoint

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-03.md#exercise-31>

Custom Markers*

Custom script

Marker file

Pipeline

Definition

Pipeline script

Script

1

try sample Pipeline...

☒ Use Groovy Sandbox

[Pipeline Syntax](#)

Custom script

Marker file

Pipeline

Definition

Pipeline script from SCM

SCM

None

Script Path

Jenkinsfile

Lightweight checkout ☒

[Pipeline Syntax](#)

Hands On Exercise 3.2

Use a Custom Marker File

<https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-03.md#exercise-32>

Best Practices



Pipeline Best Practices

A few best practices for creating pipelines in Jenkins:

- **Use a Jenkinsfile** - your pipeline should be treated like code
- **Keep it simple** - limit the amount of logic you use and don't treat declarative like a general purpose programming language (**hint**: every step should be executable from outside of Jenkins)
- **Parallelize your pipeline** - if stages can run in parallel do it to improve execution time
- **Shift important steps to the left of your pipeline** - fail faster
- **Wrap Inputs in Timeouts** - don't leave jobs waiting indefinitely for input blocking executors
- **Prefer Stash to Archiving** - to share files between stages so that you can move execution of stages across multiple agents seamlessly
- **Use Plugins vs custom code** - easier to develop and maintain
- **Prefer external scripts/tools for complex or CPU-expensive processing** - limit processing requirements on the master
- **Use trusted global libraries** - increases reusability/reduces complexity, but beware of requirements for processing scripts on the master

Thank You!

