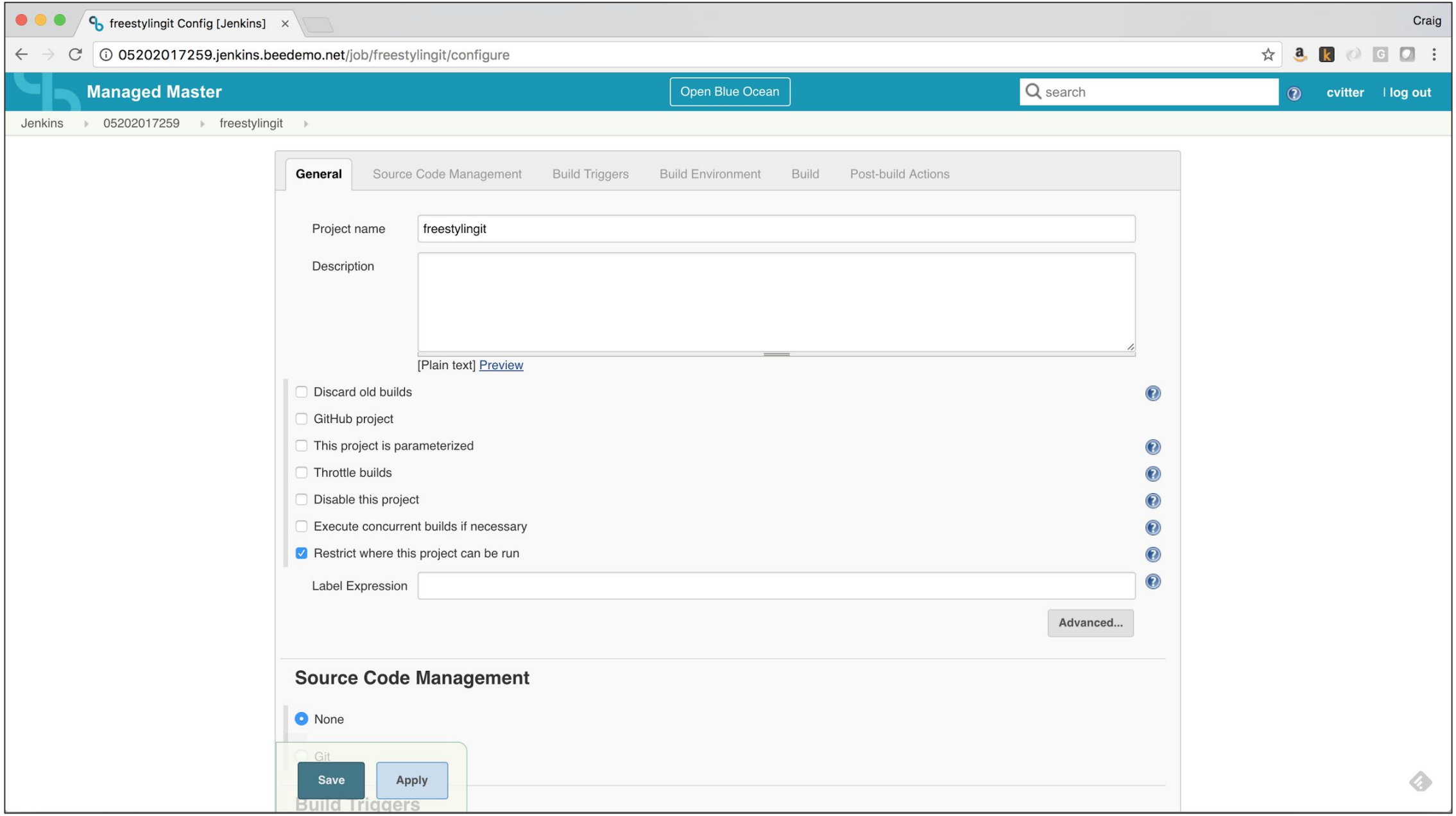
Introduction to Declarative Pipeline

cloudbees

In the beginning there was the Freestyle job....



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)

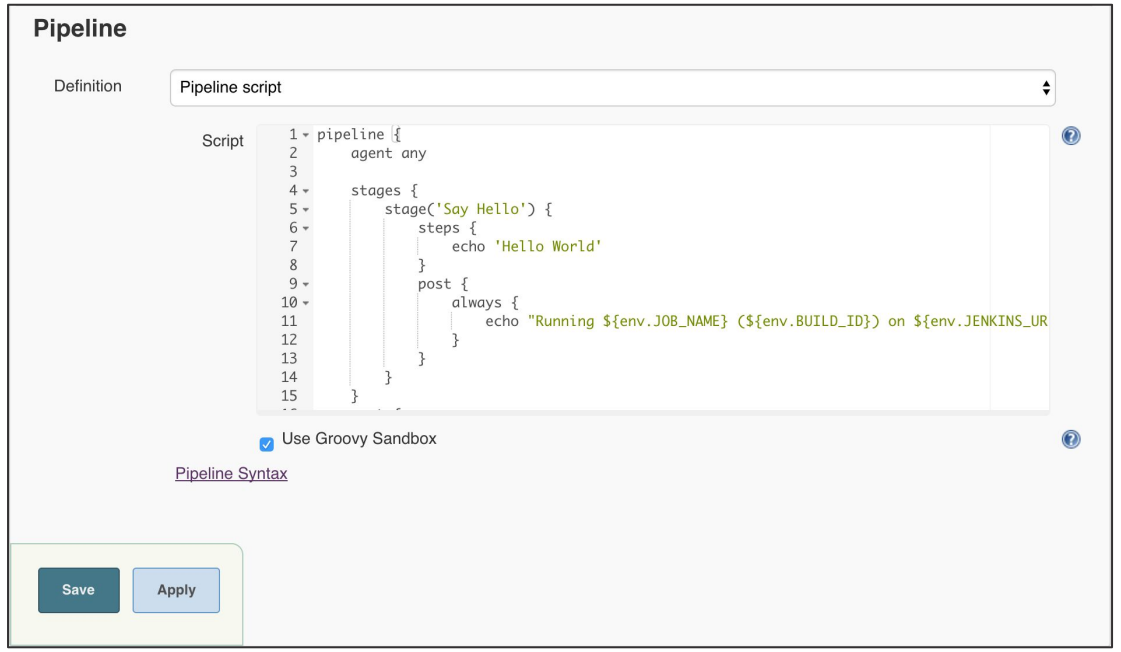
Creating a Pipeline

The Simplest Declarative Jenkins File vs Scripted

```
pipeline {
    agent any

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

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

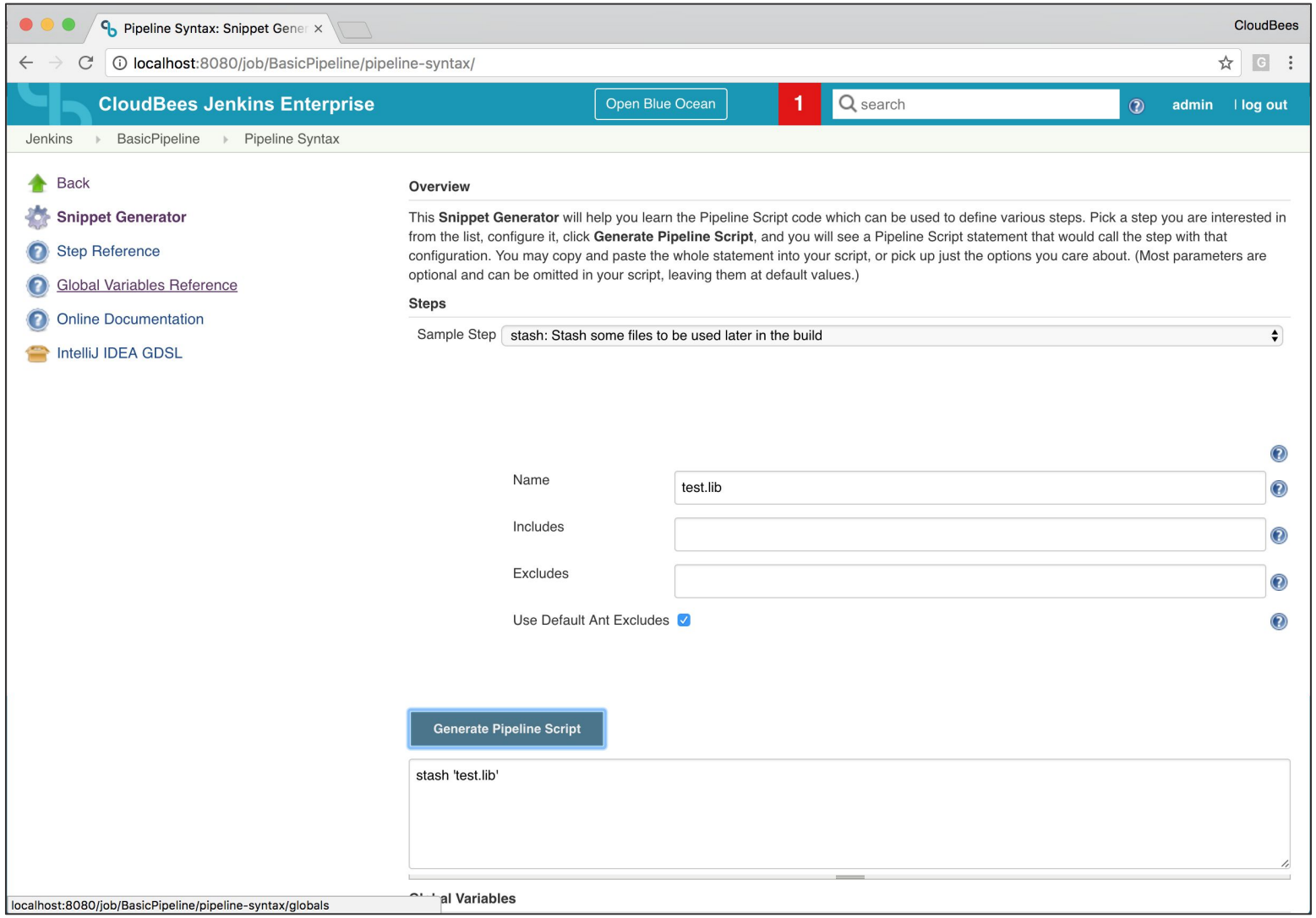


Hands On Exercise 1.1

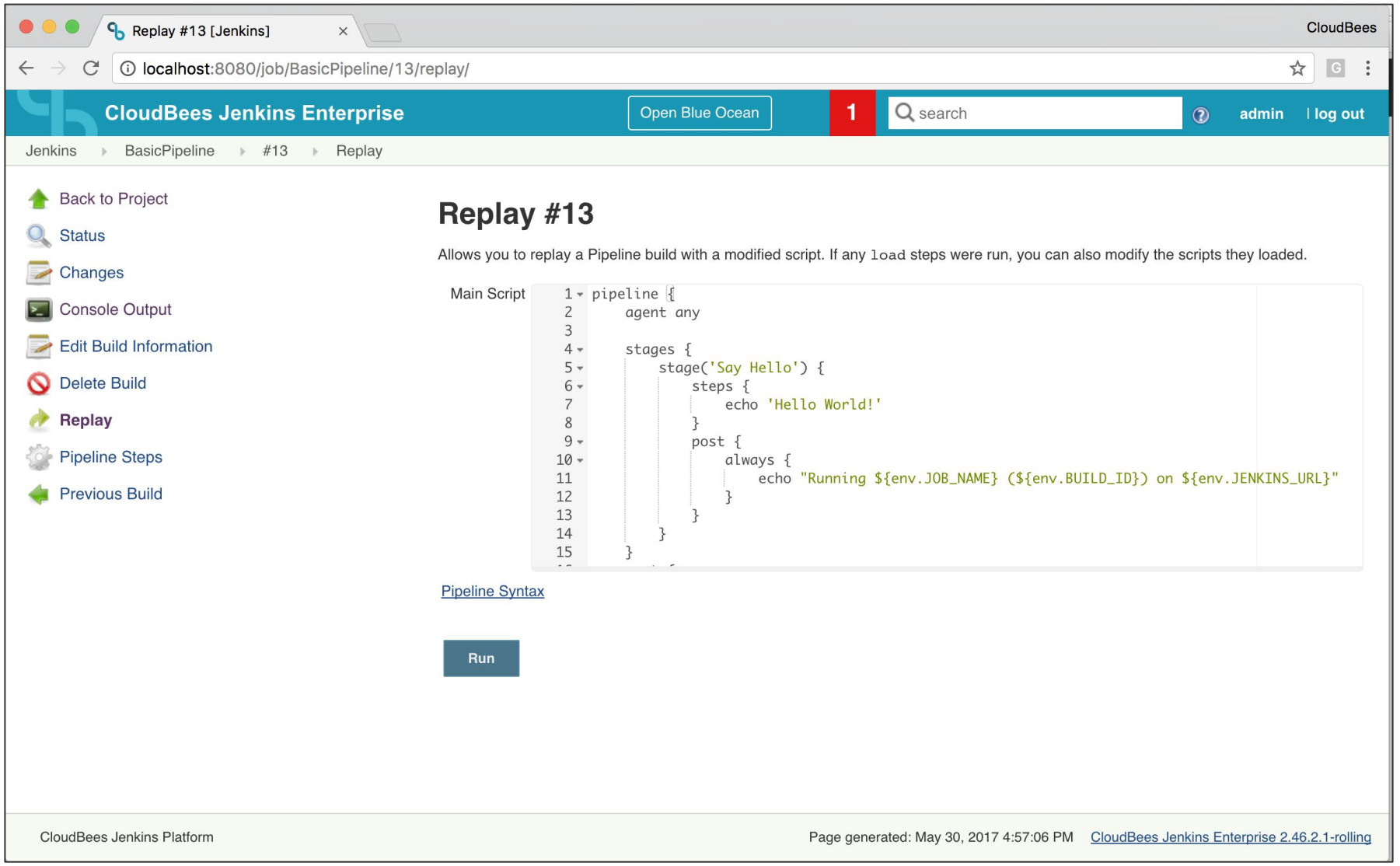
Create a Simple Declarative Pipeline

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

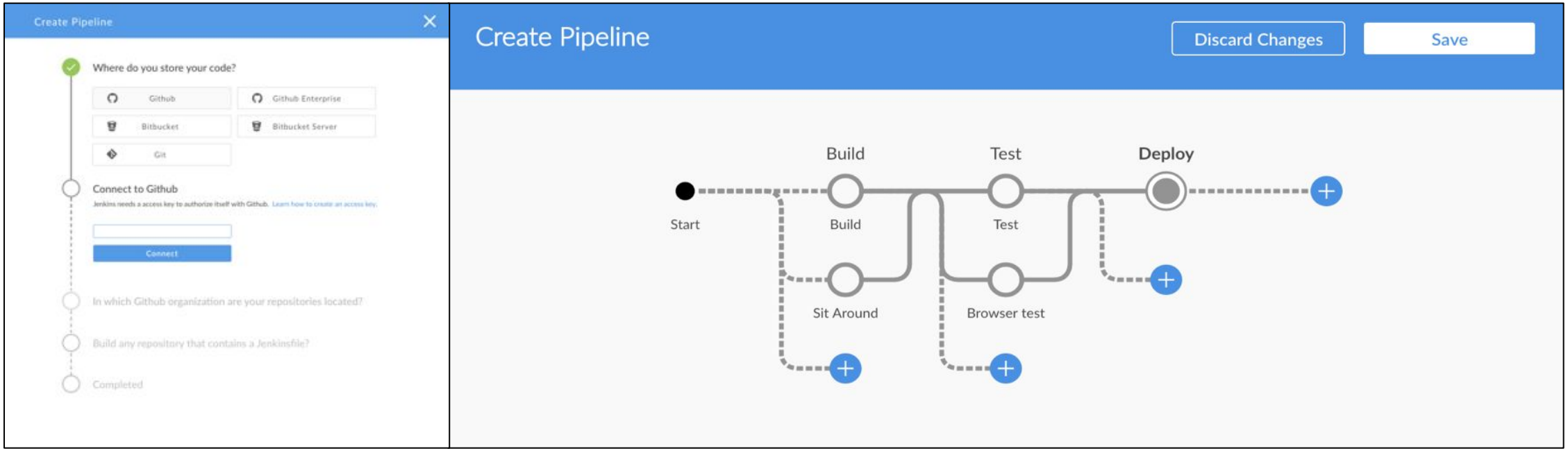
The Jenkins Snippet Generator



Pipeline Replay



Blue Ocean Editor



Beyond Hello World

cloudbees.

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

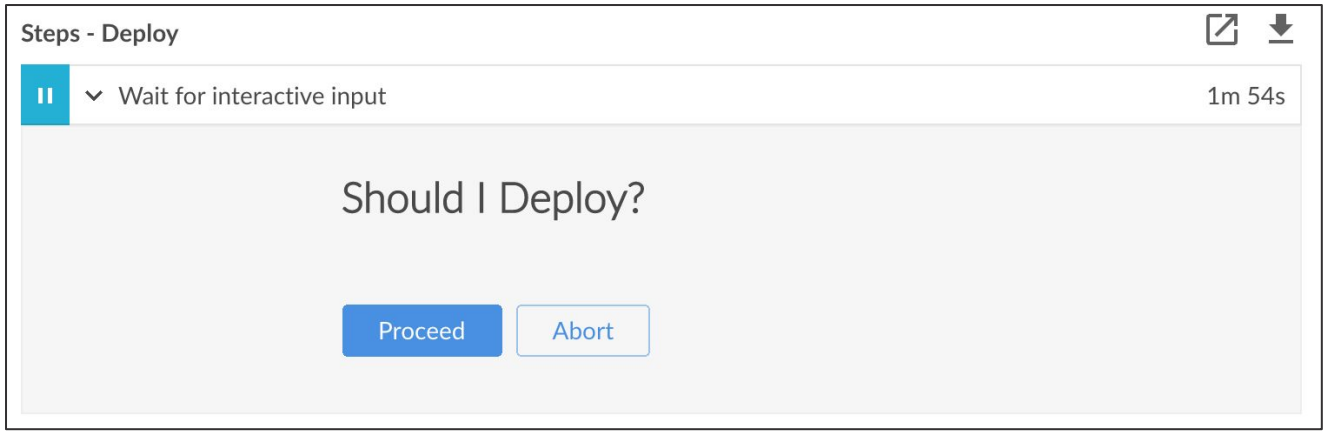
https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-01.md#exercise-14

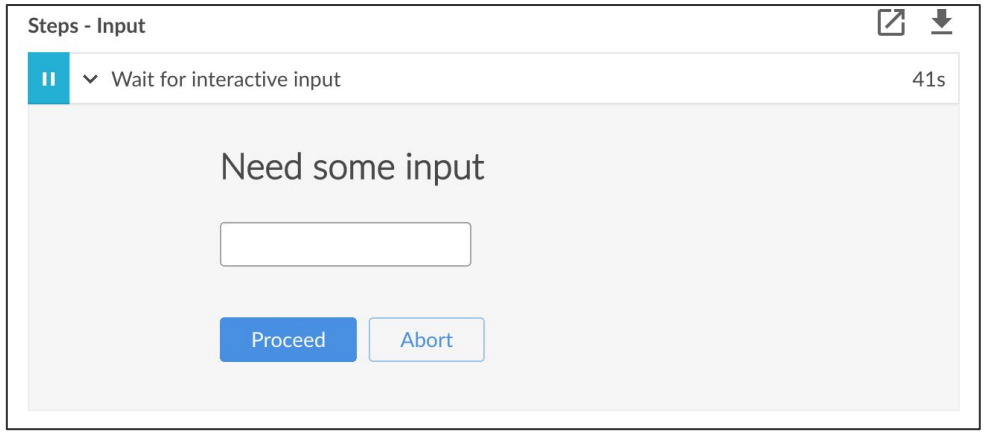
Capturing User Input

```
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}"
    }
}
```





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 { ... }
            }
        }
    }
}
```

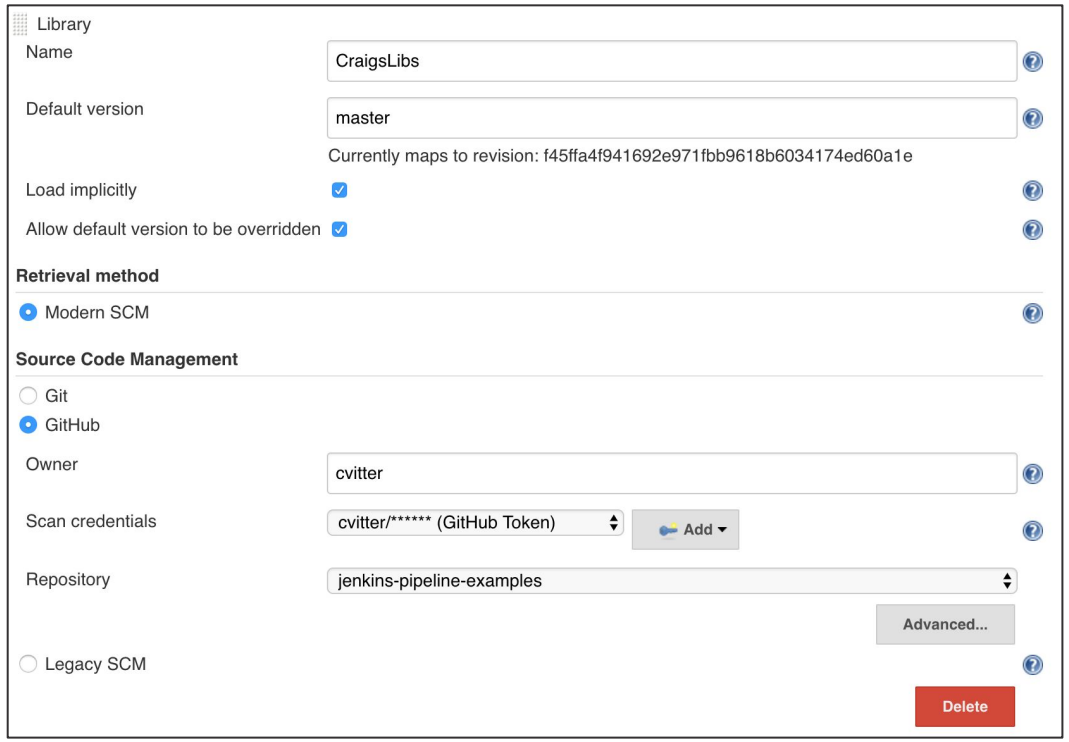
Hands On Exercise 1.6

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!"
}
```




```
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

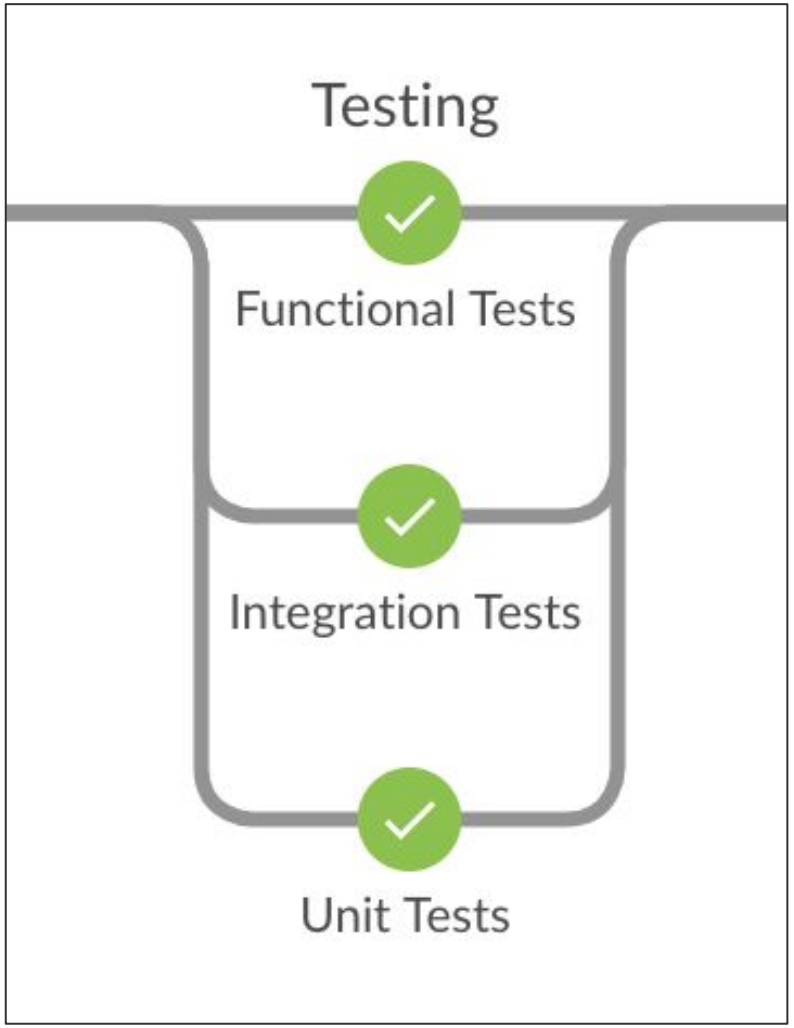
Conditional Flow Control

```
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 on 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

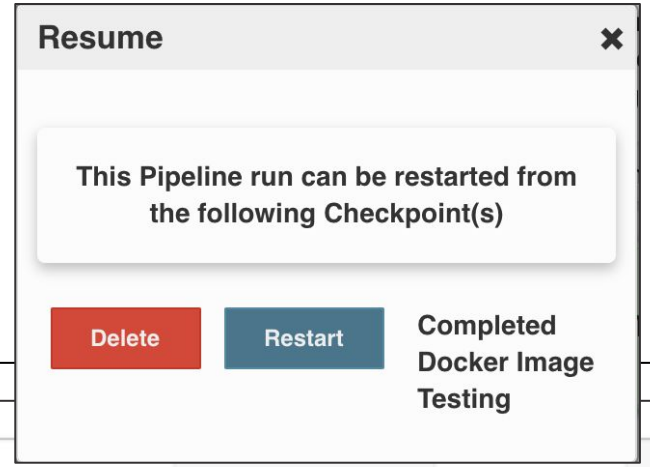
Handling Feature Branches and Pull Requests

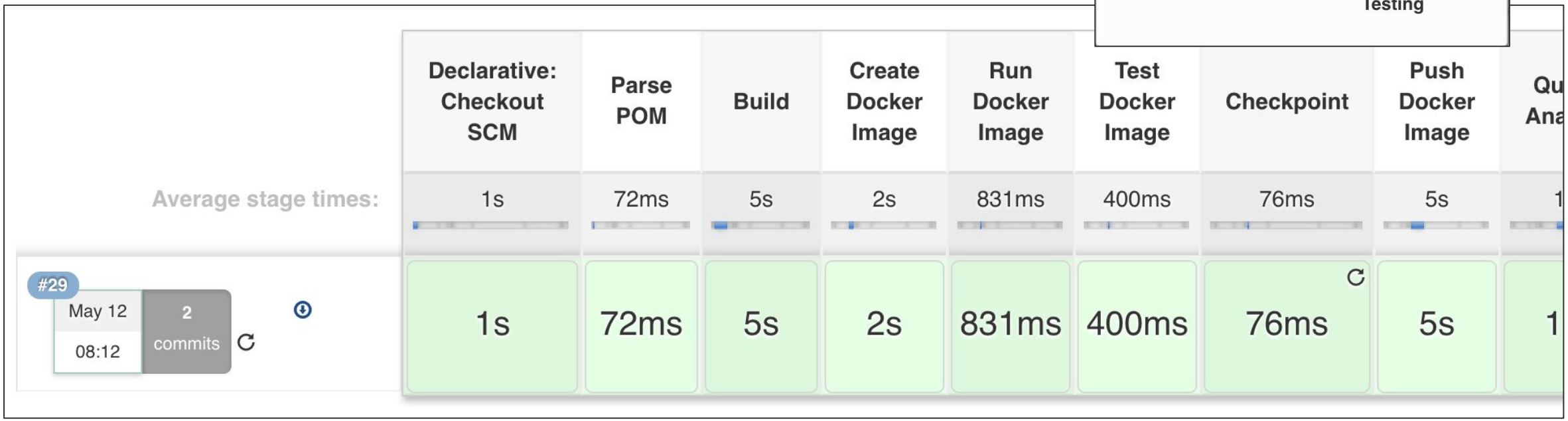
https://github.com/PipelineHandsOn/intro-to-declarative-pipeline/blob/master/Exercise-02.md#exercise-24

Enterprise Only Pipeline Features

Checkpoints*

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



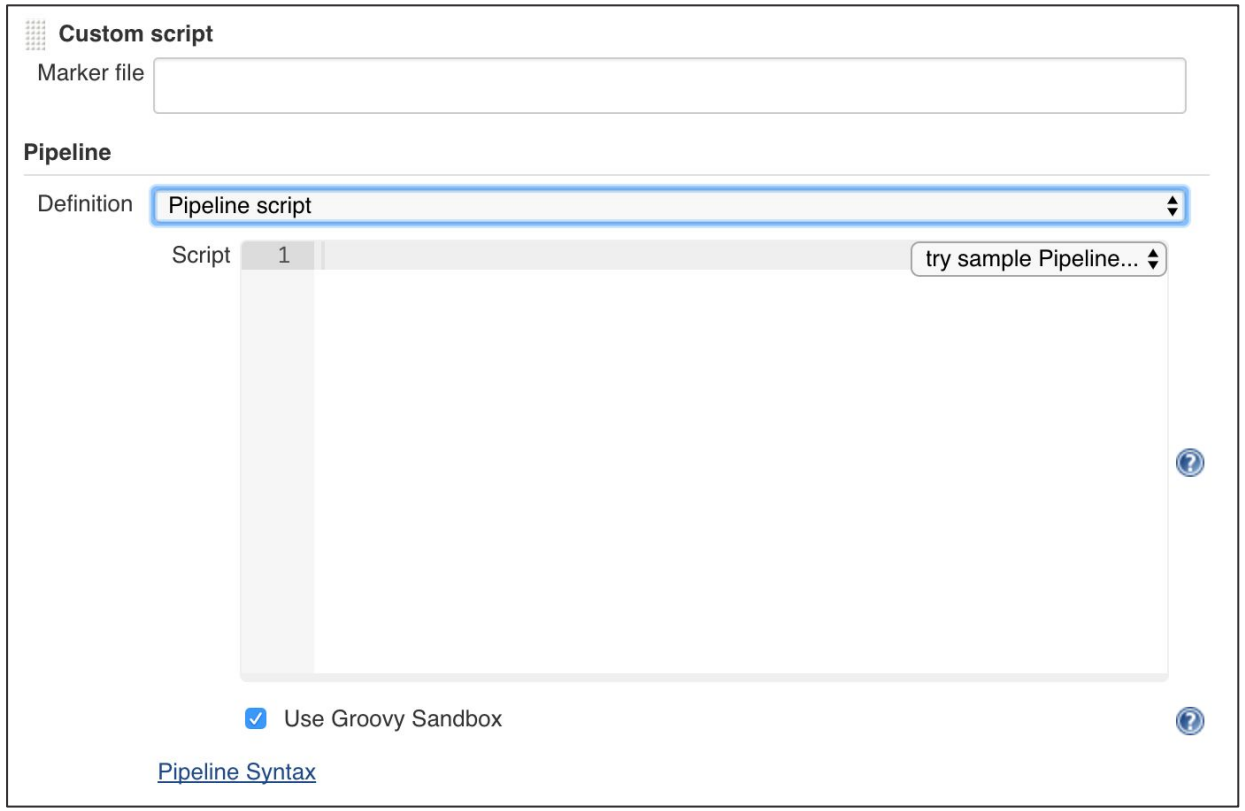


Hands On Exercise 3.1

Create a Checkpoint

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

Custom Markers*



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

cloudbees