Diving into Pipeline syntax

Now that we have covered the basic structure of Pipelines, we will dig into Pipeline syntax and the Pipeline steps that allow you to fine-tune the functionality of your Pipeline. Many of these features are available in the Blue Ocean Editor; all of them can be typed into the Blue Ocean code editor or directly into the *Jenkinsfile* if you are working with the command line and the UI.

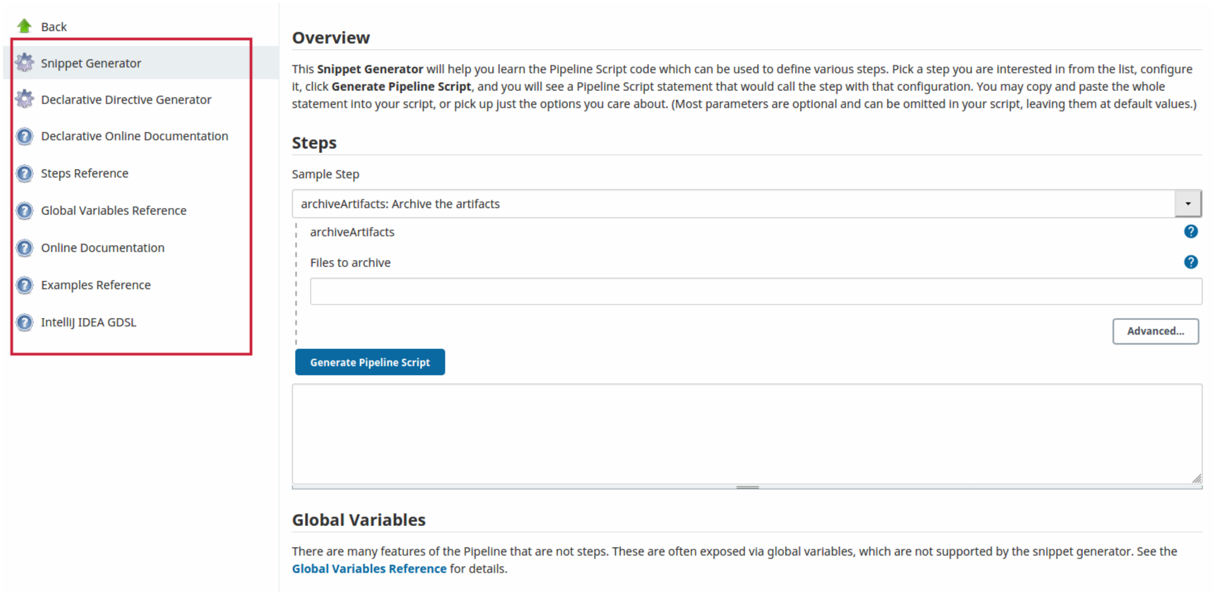
For the rest of the class, you can use Blue Ocean Code Editor, the inline editor on the Jenkins dashboard, or command-line tools and a text editor to do the exercises, and you can alternate between these methods.

Accessing help information

Jenkins provides links to helpful tools and documentation any time you are in the context of a Pipeline. You can access this by doing either of the following:

- From the Jenkins Dashboard, click the arrow to the right of a Pipeline and select **Pipeline Syntax** from the list that is displayed.
- If you are in the context of a Pipeline, click the **Pipeline Syntax** link in the left frame.

This opens a screen with the Snippet Generator (we will explain what this is soon) and a list of useful resources in the left frame.



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: archiveArtifacts: Archive the artifacts

archiveArtifacts

Files to archive

Global Variables

There are many features of the Pipeline that are not steps. These are often exposed via global variables, which are not supported by the snippet generator. See the **Global Variables Reference** for details.

Access help information

Open the **Pipeline Syntax** link using both methods:

- From the Jenkins Dashboard, select **Pipeline Syntax** from the dropdown list displayed for your Pipeline.

- From the Jenkins Dashboard, click the name of your Pipeline and then select **Pipeline Syntax** from the left frame.

As you read the descriptions of the tools and documentation provided, you can click on each piece to familiarize yourself with the structure and information that is required.

Reference documentation

The reference documentation provides complete information about Pipeline syntax and supported steps , which are indispensable.

- The **Declarative Online Documentation** describes the basic syntax of the Declarative Pipeline, including Sections and Directives. This document contains information about flow control, differences between the Pipeline DSL and plain Apache Groovy, and a syntax comparison with Scripted syntax. This is the same information presented in the Pipeline Syntax page of the documentation.
- The Pipeline Steps Reference gives links to descriptions of steps provided by **all** Jenkins plugins. This list is generated automatically; it lists the steps provided by each plugin.

  Right-click on the name of the plugin to open a page that gives full syntax information about each step that is supported by that plugin.

  - The Pipeline Basic Steps plugin is fundamental. This describes the steps that are available to all Jenkins Pipelines by default..
- The **Steps Reference** link in your **Pipeline Syntax** list provides information about **only** the steps that are supported by the plugins that are installed in your instance. This is generated on your system the first time you access it, so expect a slight delay.

  After this list is generated, you can click the icon to the left of each step to display information about the syntax and behavior of that step.

Using the help information in your Pipeline

The easiest way to create code for your Pipeline is to use the Snippet Generator to generate the code. It prompts you for values for any arguments that can be specified. It then generates a code snippet that you can copy and paste into your Pipeline. Similarly, you can use the Declarative Directive Generator to generate a code snippet for any directive you want to use.

You can also copy and paste the code fragments in the reference documents into your Pipeline using a text editor, the inline editor, or the Blue Ocean Code Editor. This gets the exact syntax in place. You can then modify arguments, indentations, and so on.

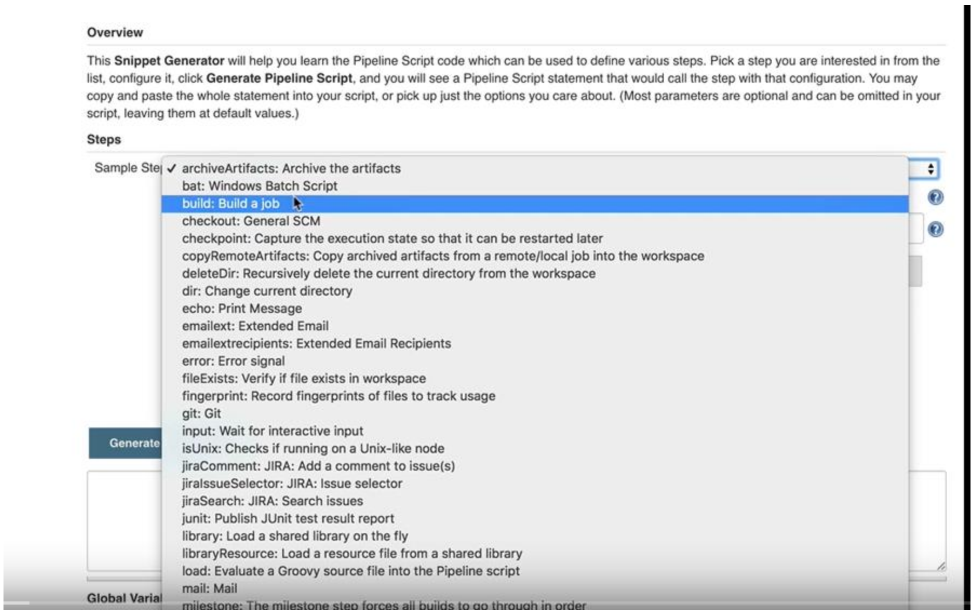
Snippet Generator

The Snippet Generator creates valid code for Scripted Pipeline steps, based on the desired task. You choose the task you want to do (for example, "Archive JUnit-formatted test results" or "Record fingerprints of files to track usage") and it displays the proper code, which you can customize and copy and paste into your *Jenkinsfile*. The Snippet Generator supports Pipeline steps provided by Jenkins and any installed plugins.

To use the Snippet Generator:

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

To select a step, click the "Sample Step" box. A drop-down menu appears with steps you might want to execute. Scroll through the items displayed to select the task you want. Alas, the Snippet Generator does not allow you to search for a task.



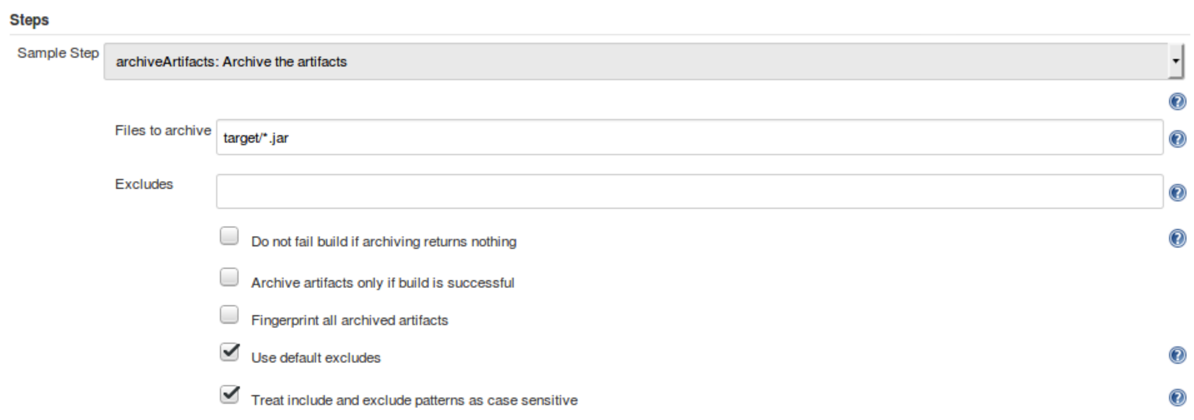
Overview

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 Ste| ✓ archiveArtifacts: Archive the artifacts
bat: Windows Batch Script
build: Build a job
checkout: General SCM
checkpoint: Capture the execution state so that it can be restarted later
copyRemoteArtifacts: Copy archived artifacts from a remote/local job into the workspace
deleteDir: Recursively delete the current directory from the workspace
dir: Change current directory
echo: Print Message
emailext: Extended Email
emailextrecipients: Extended Email Recipients
error: Error signal
fileExists: Verify if file exists in workspace
fingerprint: Record fingerprints of files to track usage
git: Git
input: Wait for interactive input
isUnix: Checks if running on a Unix-like node
jiraComment: JIRA: Add a comment to issue(s)
jiraIssueSelector: JIRA: Issue selector
jiraSearch: JIRA: Search issues
junit: Publish JUnit test result report
library: Load a shared library on the fly
libraryResource: Load a resource file from a shared library
load: Evaluate a Groovy source file into the Pipeline script
mail: Mail
milestone: The milestone step forces all builds to go through in order

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



Steps

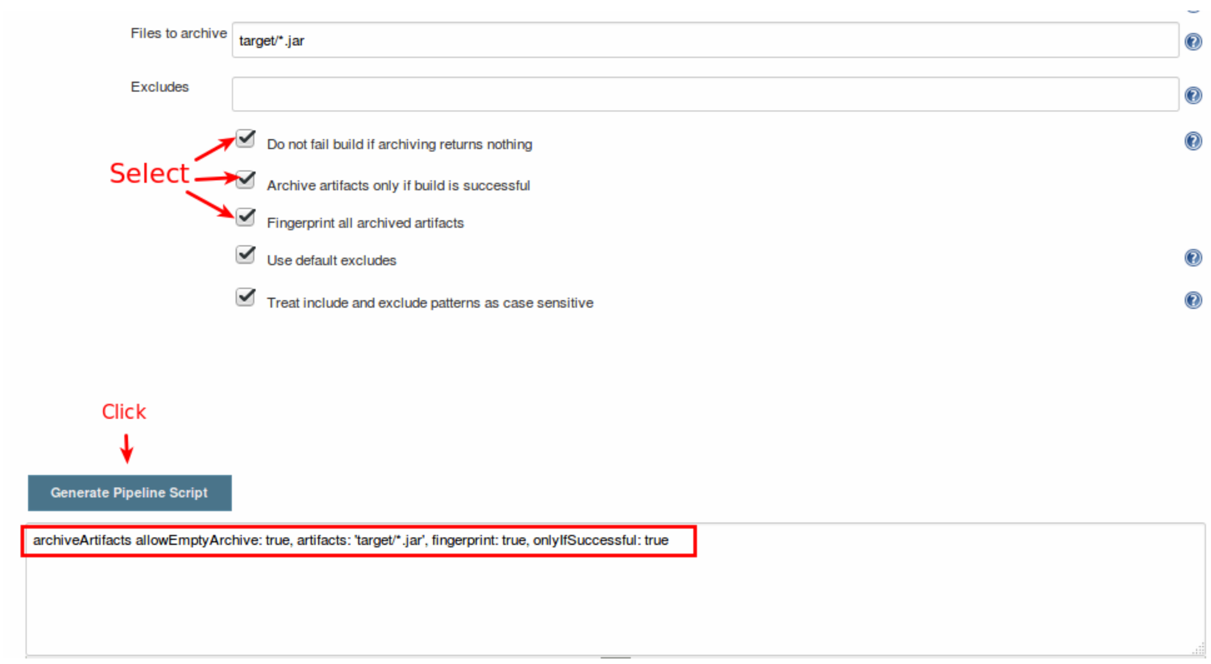
Sample Step: archiveArtifacts: Archive the artifacts

Files to archive: target/*.jar

Excludes:

☐ Do not fail build if archiving returns nothing
☐ Archive artifacts only if build is successful
☐ Fingerprint all archived artifacts
✓ Use default excludes
✓ Treat include and exclude patterns as case sensitive

To see how this works, select the three empty boxes so that all boxes are selected, then click the "Generate Pipeline Script" button; a snippet appears below the button.



Files to archive: target/*.jar

Excludes:

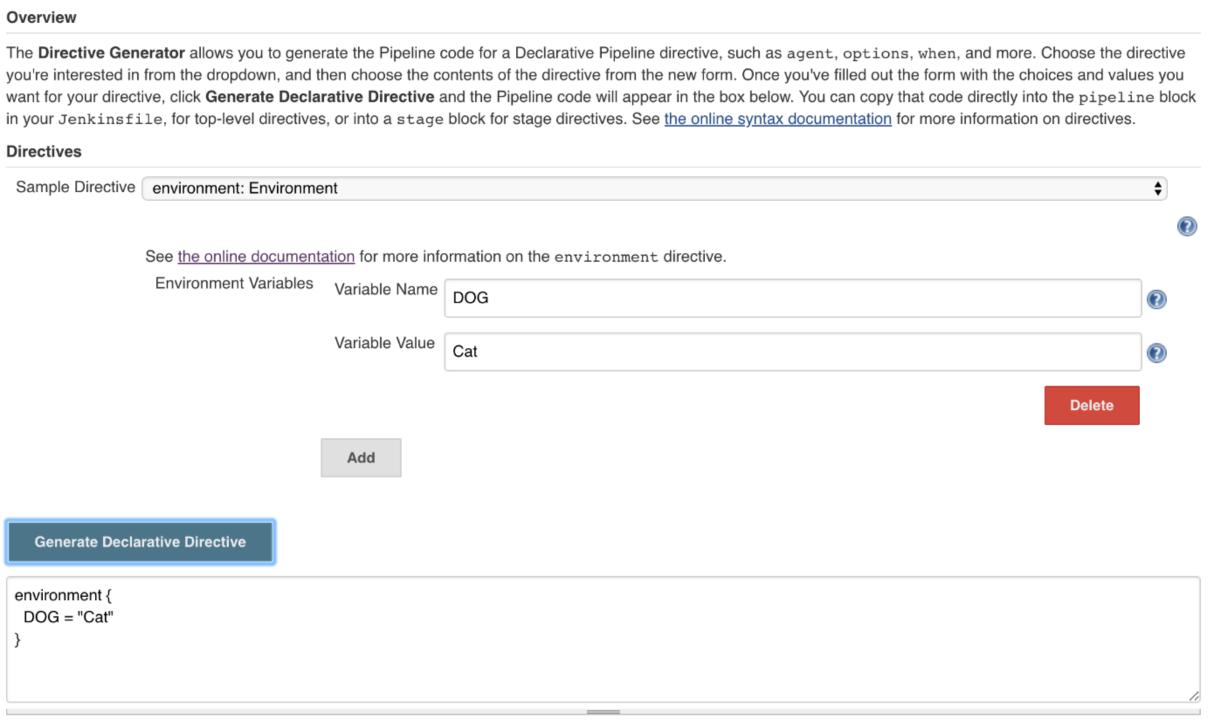
- [x] Do not fail build if archiving returns nothing
- [x] Archive artifacts only if build is successful
- [x] Fingerprint all archived artifacts
- [x] Use default excludes
- [x] Treat include and exclude patterns as case sensitive

```
archiveArtifacts allowEmptyArchive: true, artifacts: 'target/*.jar', fingerprint: true, onlyIfSuccessful: true
```

Copy and paste this snippet into the appropriate part of the Pipeline script, using the inline editor, the *Jenkinsfile* you are editing with a text editor, or the Blue Ocean Code Editor.

Declarative Directive Generator

The Directive Generator is similar to the Snippet Generator except it helps you write Declarative Pipeline directives such as `agent`, `options`, `stage` and more:



Overview

The **Directive Generator** allows you to generate the Pipeline code for a Declarative Pipeline directive, such as `agent`, `options`, `when`, and more. Choose the directive you're interested in from the dropdown, and then choose the contents of the directive from the new form. Once you've filled out the form with the choices and values you want for your directive, click **Generate Declarative Directive** and the Pipeline code will appear in the box below. You can copy that code directly into the `pipeline` block in your `Jenkinsfile`, for top-level directives, or into a `stage` block for stage directives. See the online syntax documentation for more information on directives.

Directives

Sample Directive: environment: Environment

See the online documentation for more information on the `environment` directive.

Environment Variables — Variable Name: DOG — Variable Value: Cat

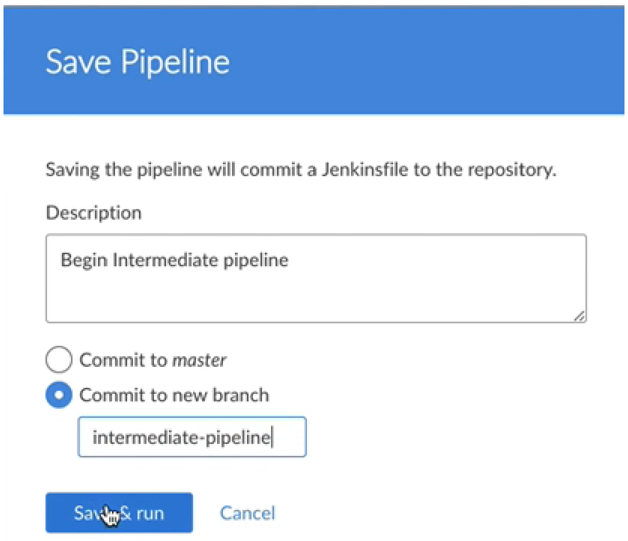
```
environment {
  DOG = "Cat"
}
```

Start a new branch

In the next sections, we will explore some frequently used Pipeline features. Before we begin, we need to set up a new branch called `intermediate pipeline` to use it for the next exercises.

To do this in the Blue Ocean Editor:

- Save the Pipeline to `master`. We want to start with what is in the `master` branch.
- Create the `intermediate-pipeline` branch, which is a new branch for the exercises in this section.



Save Pipeline

Saving the pipeline will commit a Jenkinsfile to the repository.

Description

Begin Intermediate pipeline

( ) Commit to *master*
(●) Commit to new branch

intermediate-pipeline

If you prefer, you can use standard Git commands to set up a new branch that is based on the current contents of the `master` branch.

Everything discussed here can be implemented by typing the code directly into the *Jenkinsfile* or the Blue Ocean Code Editor. Some of the facilities can also be implemented using the Blue Ocean Visual Editor.

Specify agents for our Pipeline

Now we need to modify our Pipeline to use different pre-configured agents in the Pipeline. We recommend that you open your lab environment and implement the steps described in this section to familiarize yourself with the process. After this section, you have a lab exercise to practice what has been discussed in this Lesson.

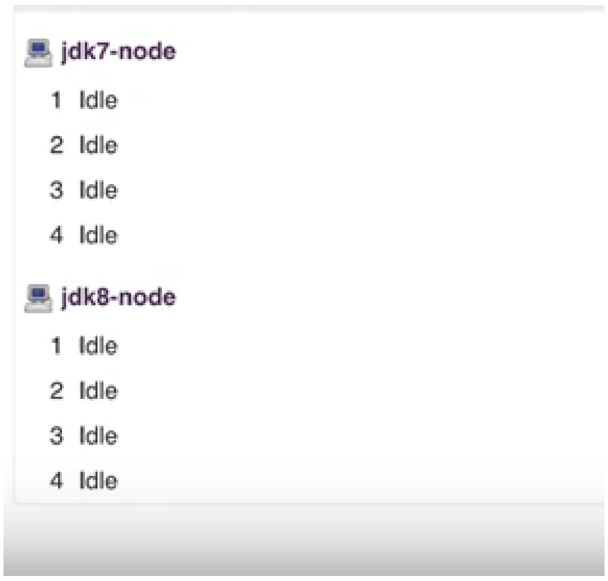
We are building our application to work with both JDK7 and JDK8. This means that we need a JDK7 environment and a JDK8 environment. To implement this:

- See what nodes are available in your lab environment.
- Change the global `agent` to be `agent none`; in other words, there is no global agent
- Specify the appropriate agent 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.

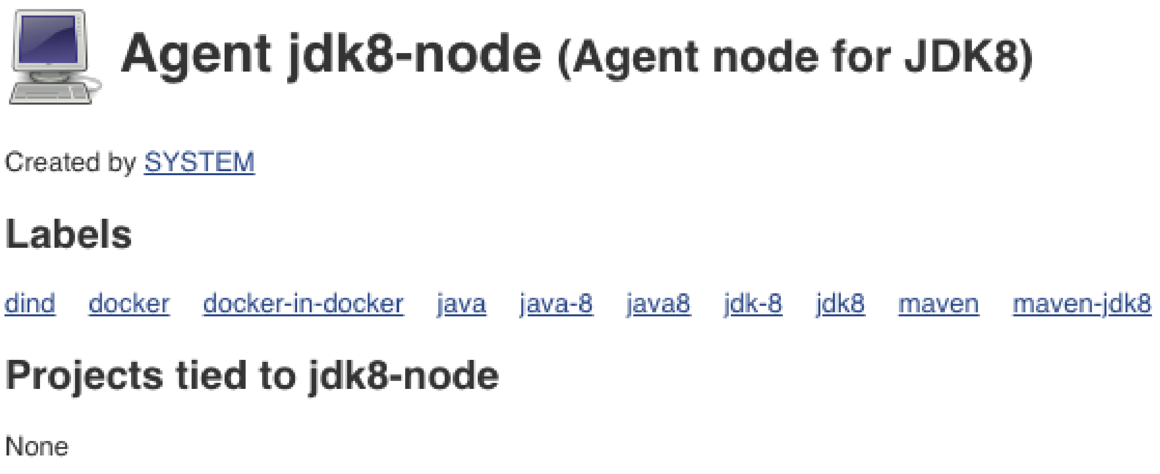
See available nodes

Your lab is pre-configured with nodes that you can use as agents in your Pipeline. You can view these nodes at the bottom of the left frame of the Classic Web UI:


jdk7-node
1 Idle
2 Idle
3 Idle
4 Idle
jdk8-node
1 Idle
2 Idle
3 Idle
4 Idle

In this example, you see that we have two nodes: `jdk7-node` and `jdk8-node`, with four executors for each.

Click on `jdk-8-node` to see details about that node:


Agent jdk8-node (Agent node for JDK8)

Created by SYSTEM

Labels

dind docker docker-in-docker java java-8 java8 jdk-8 jdk8 maven maven-jdk8

Projects tied to jdk8-node

None

Note that multiple labels are defined for this node; using any of these labels as the `agent` parameter specifies this node.

Now click **Nodes** at the top-left of the screen to see how you can then view the details of the other (`jdk7-node`) node.

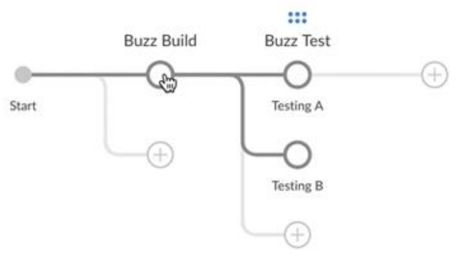
Reset global agent instructions

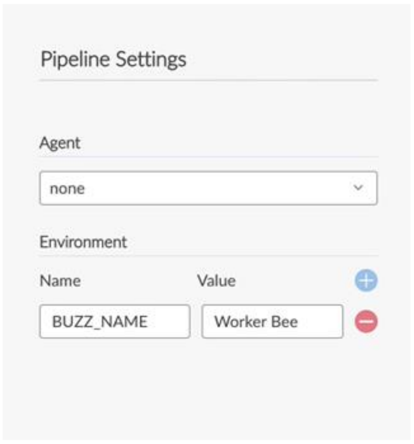
We are going to set up our Pipeline to build and test for both JDK7 and JDK8, which means we need to assign appropriate nodes to each `stage`. First, we will go through the process of doing this in the Blue Ocean Editor. You can then view the resulting *Jenkinsfile* to see the code; you could instead type this code directly into the *Jenkinsfile*.

First, we need to set `agent none` for the entire Pipeline, so we can set specific agents for the individual stages. You can do this by editing the *Jenkinsfile* directly with a text editor or using the Blue Ocean code editor.

To set `agent none` using the Blue Ocean Code Editor:

- From the Blue Ocean editor, in the **intermediate-lab** pipeline, click the edit icon.
- Under **Pipeline Settings** on the right, change the Agent value to **none**:




Pipeline Settings

Agent: none

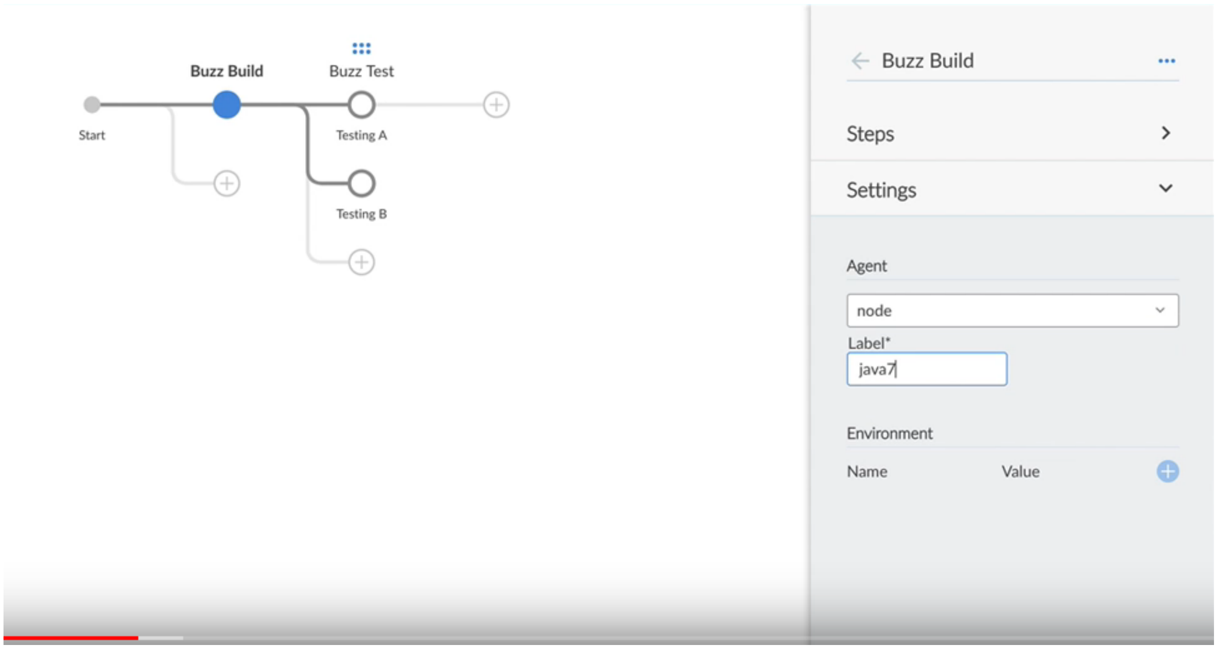
Environment

Name	Value
BUZZ_NAME	Worker Bee

Specify java7 for existing stages

To specify a JDK7 agent for each of the existing stages using the Blue Ocean Visual Editor:

- Select the **Buzz Build** stage
- From the **Settings** menu on the right, under choose **node** from the **Agent** drop-down menu
- Specify the `java7` node under **Label**. Note that you can use any of the labels that are listed for **jdk7-node**.
- Now do the same for the **Testing A** and **Testing B** stages under **Buzz Test**.



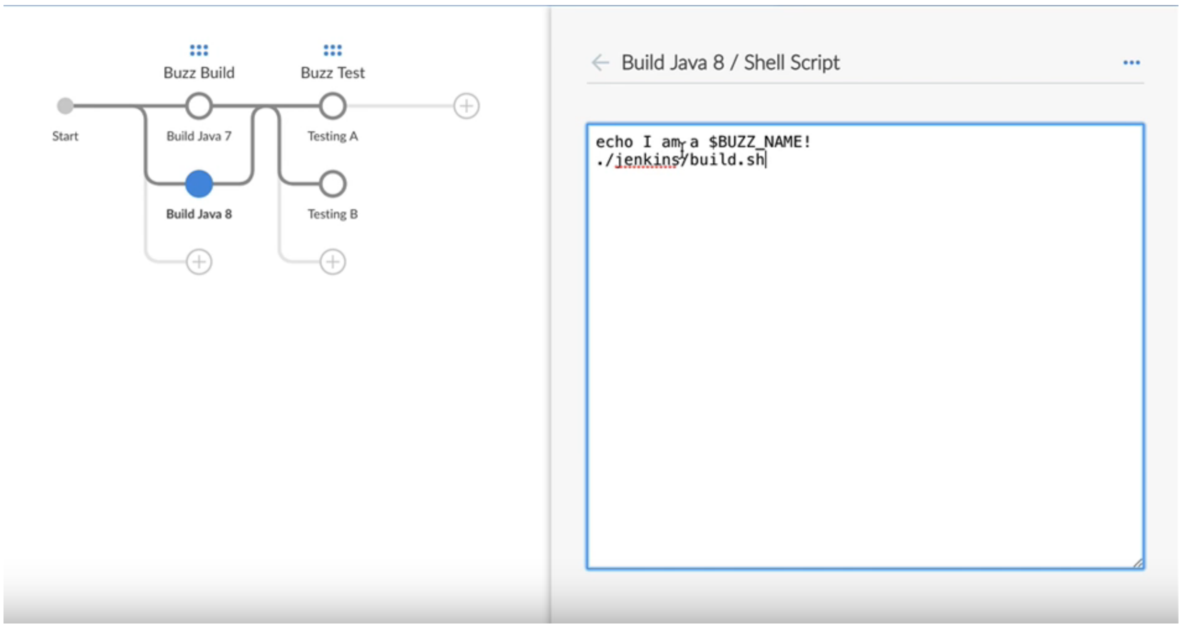
Agent: node
Label*: java7
Environment
Name Value

- Click **Save** and under **Description** write a description of what changed, such as **Added JDK 7 to pipeline**; commit it to **intermediate-pipeline** and click **Save & run**
- Go into the Classic Web UI by clicking the **Go to classic** icon at the top right and open the **Console Output** on the right menu; see that each stage is running on `jdk7-node`

Add JDK8 build

Now we will add the **Build Java 8** stage to execute in parallel to the existing stage under **Buzz Build** and use a JDK8 environment. We want this new stage to do exactly what the existing stage does so we can copy the contents of the existing stages to populate the new stages:

- From the Blue Ocean editor, in the **intermediate-pipeline** branch, click the edit icon.
- Under the **Buzz Build** stage click to add a stage
- Select the newly added stage and name it **Build Java 8** stage, open **Settings**, and choose **node** under the Agent dropdown; under **Label** type `java8`
- Rename the existing **Buzz Build** stage to be **Build Java 7**
- Open the **Build Java 7** stage, open the **Shell Script** 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 the command box
- Create a new **Archive the artifacts** step on **Build Java 8** stage, enter `target/*.jar` in *Artifacts*, and check the **Fingerprint** box.



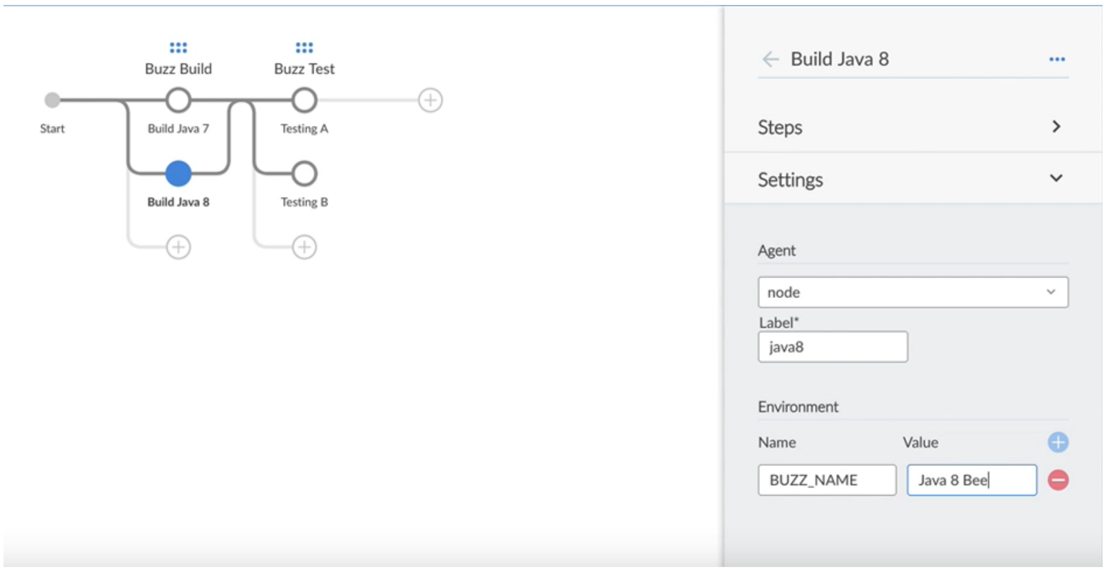
```
echo I am a $BUZZ_NAME!
./jenkins/build.sh
```

You could instead use the Blue Ocean code editor that you can access using `Ctrl+s` (Linux or Windows) or `cmd+s` (macOS) to do this. You would then copy and paste one existing stage and edit the name and the agent for the new stage.

Set stage-specific environment variable

Before we move onto the lab exercise, we will explore using a stage-specific environment variable. Remember that we defined the **BUZZ_NAME** environment variable globally for this Pipeline. A global environment variable can be overriden for an individual stage; to do this just for the **Build Java 8** stage:

- Select **Build Java 8**
- Go into **Settings**
- Under **Environment** click
- Set the value of **Name** to **BUZZ_NAME** and **Value** to **Java 8 Bee**



Agent: node
Label*: java8
Environment
Name: BUZZ_NAME
Value: Java 8 Bee

- Click **Save** to save the Pipeline; use "Add Java8 build" for the Description.
- Go into the Classic Web UI by clicking the **Go to classic** icon at the top right and open the **Console Output** on the right menu; see that stages are running on `jdk8-node`

Implement stash / unstash steps

Use the stash step to save a set of files for use later in the same build, but in another `stage` that executes on a different node/workspace. Use the unstash step to restore the stashed files into the current workspace of the other stage.

By default, stashed files are cleared at the end of the Pipeline run, although the artifact stashed most recently is preserved to support Declarative stage restarting feature. Set the `preserveStashes` job property to preserve more stashes for restarted Pipelines.

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 one of the following options:

- The External Workspace manager plugin
- An external repository manager such as Nexus or Artifactory
- The Artifact Manager on S3 plugin (README)

For our example Pipeline, we will use `stash` and `unstash` to ensure that the JDK 7 build is used by the JDK 7 test stage and the JDK 8 build is used by the JDK 8 test stage.

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 unstash files, call `unstash`, using the `name` you assigned with the `stash` step above. Optionally, use the `dir` step to create a directory where the files will be written.

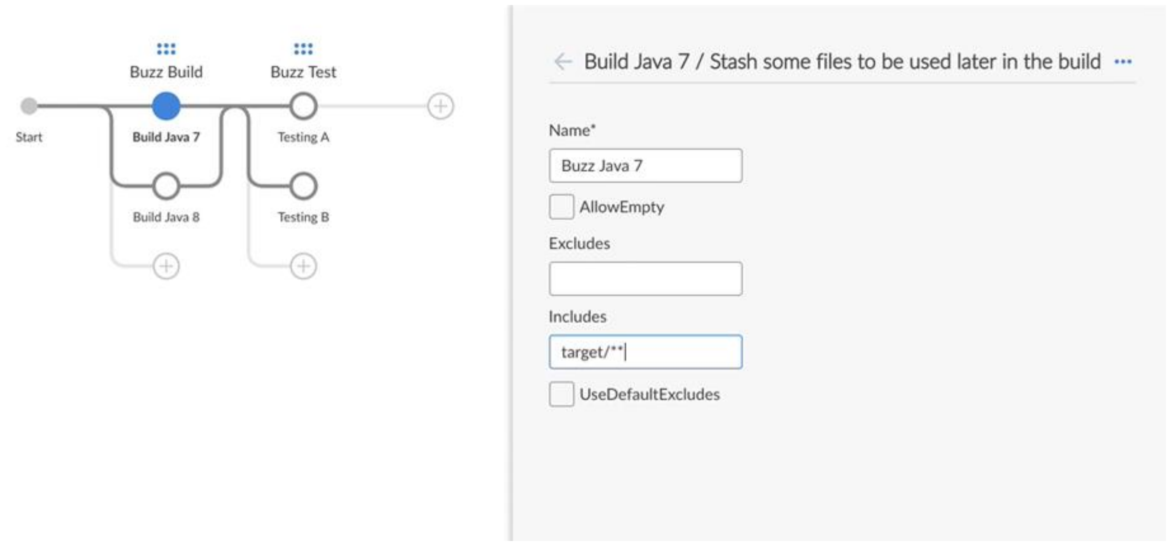
To store files from a different directory (or a subset of the workspace), use the optional `includes` parameter to give the name of the files or directories to store. This accepts a set of Ant-style include patterns. Other optional parameters are documented on the Pipeline: Basic Steps reference page

Implement stash

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.

To implement the `stash` step in the Blue Ocean Editor:

- From the Blue Ocean editor, in the **intermediate-pipeline** branch, click the edit icon.
- Add a step to the **Build Java 7** stage and choose **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.



Build Java 7 / Stash some files to be used later in the build

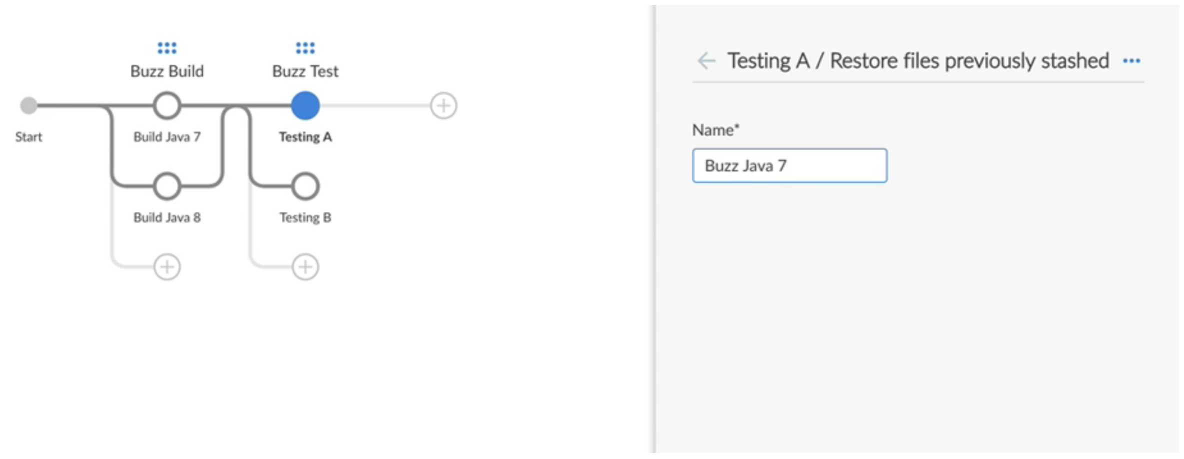
Name*: Buzz Java 7
AllowEmpty
Excludes:
Includes: target/**
UseDefaultExcludes

- Repeat the steps above for **Build Java 8** stage, changing the **Name** to **Buzz Java 8**

Implement unstash

To implement the `unstash` step:

- From the Blue Ocean editor, in the **intermediate-pipeline** branch, click the edit icon.
- Add a step to the **Testing A** stage under **Buzz Test** step and choose **Restore files previously stashed**.
- Use the **Buzz Java 7** name used for the stash we created above.



Testing A / Restore files previously stashed

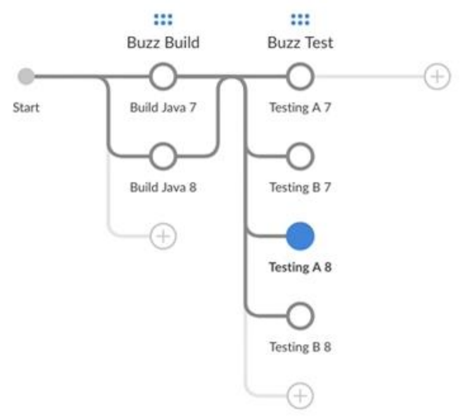
Name*: Buzz Java 7

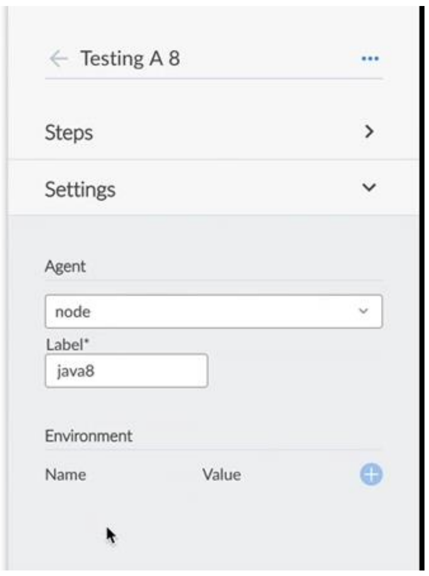
- Follow the same procedure to Add a **Restore file previously stashed** step and **Name** it **Buzz Java 8** in the **Testing B** stage

Move the unstash step

Blue Ocean sets up the Pipeline to run the steps in the order you define them. This works well for the **Build** stages — after all the build steps execute, the results are stashed. However, for the **Test** stages, this means that the `unstash` step would run after the tests that should use it. The easiest way to fix this is:

- Click **Testing A** stage under **Buzz Test**
- In the **Steps** section, click the in the **Restore files previously stashed** step and drag it above the **Shell Script**
- Repeat the click and drag of the **Restore file previously stashed** step for **Testing B**
- Rename the testing stages to be **Testing A 7** and **Testing B 7**




Testing A 8

Steps

Settings

Agent

node

Label*

java8

Environment

Name Value

Replicate testing stages for Java 8

Replicate the same method previously in Java 7 testing for Java 8 testing:

- Add two Stages under **Buzz Test** called **Testing A 8** and **Testing B 8**
- Add a step for both and choose **Restore files previously stashed**.
- Under **Name** use **Buzz Java 8** for both new steps.
- Copy the **Shell Script** contents from **Testing A 7** to a **Shell Script** in **Testing A 8**
- Copy the **Shell Script** contents from **Testing B 7** to a **Shell Script** in **Testing B 8**

Save and run the Pipeline, using **Added Java 8 tests** for the **Description**. Note that the Pipeline runs in just about the same time as it did with two test stages because we have enough agents to run all test stages in parallel.

After the Pipeline run completes successfully, click on the **Tests** tab and see that it now reports that two tests have run successfully. When you exit the editor, your Pipeline visualization shows the four testing stages, running in parallel.

Alternate implementation method

Instead of creating and populating the JDK 8 stage and steps in the Blue Ocean Editor, you could instead use the Blue Ocean Editor to set up stash/unstash for JDK7, then open the code editor and copy the JDK7 code into the appropriate JDK8 sections. After copying the text, you need to edit the **Build Buzz 8** and **Testing B 8** sections to be for JDK 8 rather than JDK 7 and to use a JDK8 agent.

Wait for interactive input

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

Remember that there is a distinction between continuous delivery and continuous deployment. A goal of both is to produce software that could be released to the customer after each modification, but continuous delivery prepares the software and waits for manual approval before deploying whereas continuous deployment automatically publishes/deploys the software.

Jenkins Pipeline provides the `input` step that pauses the entire Pipeline until a manual intervention happens. The `input` step pauses flow execution and allows the user to interact and control the flow of the build. Only a basic **process** or **abort** option is provided; you can optionally request information back.

The `input` step is usually implemented within its own stage. It must run on `agent none` and execute without an *agent* context and its associated workspace so that it doest not block an executor.

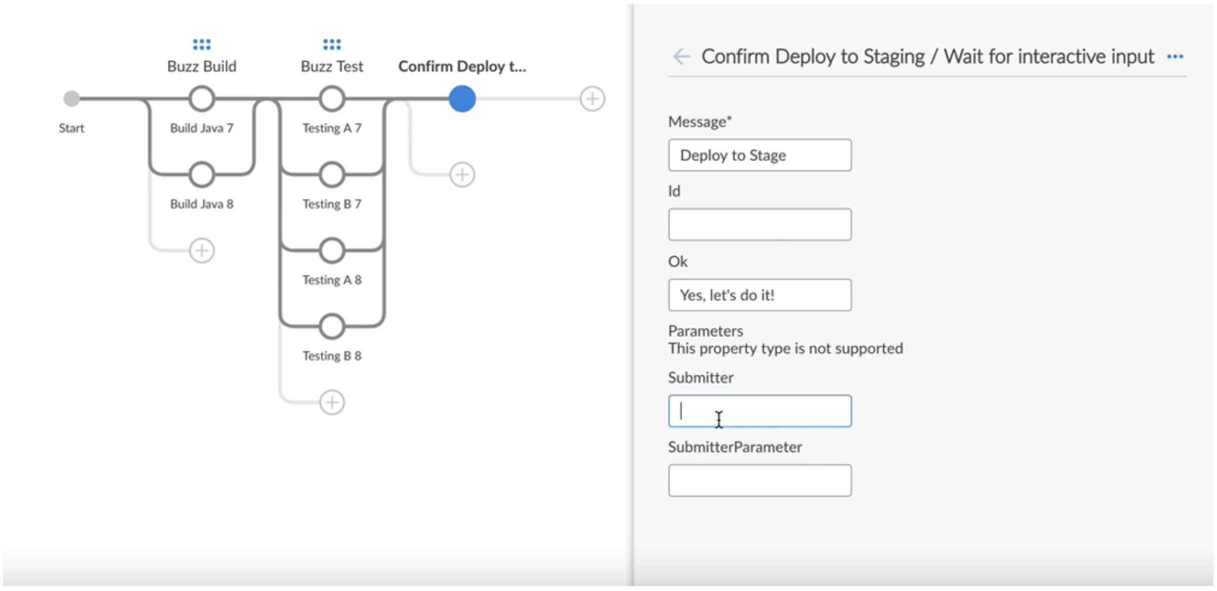
Implement Wait for interactive input step

A common practice is to run a wait-for-input stage after all tests have run successfully, to have someone confirm that the software should be deployed. This is what we will implement for our Pipeline.

Wait for interactive input cannot be implemented in the same stage as the deployment because it must run with `agent none` but the deployment stage requires an *agent* context. So we will create a separate stage that runs after all tests complete successfully but before we call any stages that actually deploy the software.

To implement this functionality using the Blue Ocean Editor:

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



Confirm Deploy to Staging / Wait for interactive input

Message*
Deploy to Stage
Id

Ok
Yes, let's do it!
Parameters
This property type is not supported
Submitter

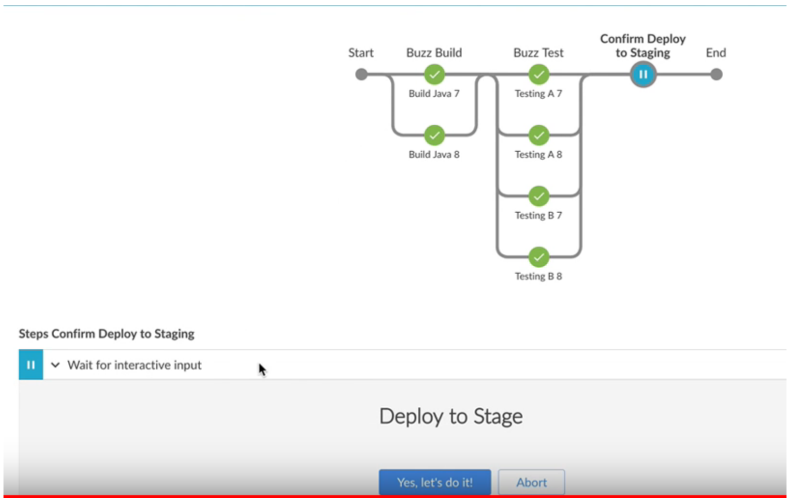
SubmitterParameter

To complete the form:

- Type the text of the question to be printed in the **Message** box; in this example, it is just **Deploy to Stage** but you could instead use a full question such as **Deploy to staging?**
- Type the text to be in the button used to respond in the **OK** box; for example, **Yes** or just **OK**
- In most cases, you want to define specific people who can approve moving forward; use the **Submitter** box to define this.
- Click **Settings** at the bottom of the right frame and set the **Agent** dropbox to **none** for this stage.
- When you have filled out the form, **Save & Run** the Pipeline to **intermediate-pipeline** with description "Deploy dialog added"

What happens when it runs?

The Pipeline run looks like it did before as it executes the build and test stages but, when it executes the **Confirm Deploy to Staging**, it pauses and displays the **Deploy to Stage** box at the bottom of the screen:



Deply to Stage is the text you supplied to the `Message` box; **Yes** is the text you provided in the **OK** box. When someone clicks the **Yes** button, the Pipeline resumes execution, presumably calling the next stage in the Pipeline.

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')
  }
}
```

If you are not using Blue Ocean, you could just type in this text to your pipeline.

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

Now we are ready to add deployment. For our application, imagine pushing a war file or a docker container to a staging server or directly to a production host. The deployment stage(s) could do much more, such as using pre-configured credentials in Jenkins.

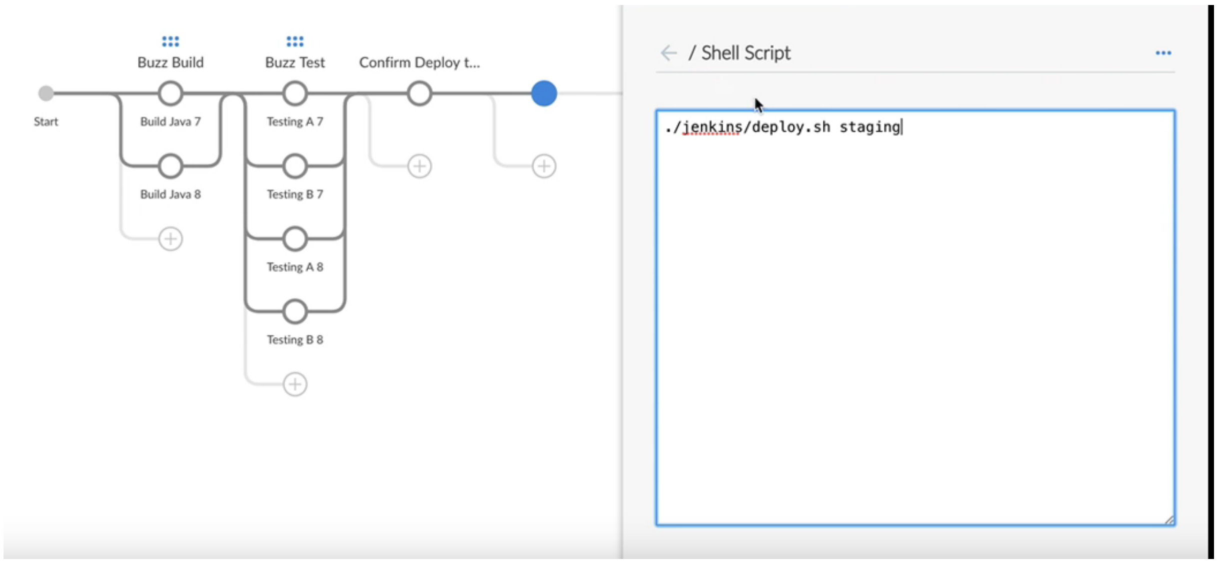
Deployment can have many meanings, such as deploy to AWS/Azure, deploy to a physical data center, upload to an app store, or upload to an internal artifict server. In any event, do not reinvent your deployment system in Jenkins Pipeline. Instead, think of the Pipeline as **glue** that executes and automates the deployent tasks you already have in place.

Implement deploy stage

We are going to add a stage that deploys the JDK 8 software to a staging server. First, we `unstash` the built software we stashed as **Buzz Java 8**, then use a Shell Script step to call the deploy script.

To implement this with the Blue Ocean Editor:

- 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
- Add a step to **Restore files previously stashed** and the Name **Buzz Java 8** as the first step in this stage
- Create a **Shell Script** step and in the field add `./jenkins/deploy.sh staging`



```
./jenkins/deploy.sh staging
```

- Click **Save** and in the Description add "Added deployment stage", commit to **intermediate-pipeline** and click **Save & run**

In the Pipeline progress screen in Blue Ocean, there will be a pause to wait for input; click the **Yes** button to continue execution. When the build completes, look at the completed build.

Do it your self Lab: Multi-environment Pipeline

In this exercise you will:

- Create and run a Pipeline in another feature branch.
- Control the node on which a Pipeline or individual stages run.
- Stash and unstash files from one stage to the next.
- Build and run tests in parallel on two different agent environments.
- Wait for user input before deploying.
- Save a Pipeline to the master branch.

Before you begin this lab, ensure that your Pipeline matches the solution from the previous lab: Create and edit a simple pipeline with parallel stages. Use the Blue Ocean Visual Editor and/or the Blue Ocean Code Editor or a regular text editor to make any necessary changes.

Task: Create and run a Pipeline in another feature branch

1. Edit your pipeline from the `master` branch.
2. Save the pipeline. In the **Save** dialog, provide the description `Create Multi-env Pipeline` and save to a new branch named `multi-env-pipeline`.

Task: Control the node on which a Pipeline runs

Edit your pipeline from the `multi-env-pipeline` branch:

1. In **Pipeline Settings**, set the Pipeline's agent:
   - **Agent**: **node**
   - **Label**: `java7`
2. Save the pipeline. In the **Save** dialog, provide the description `Run pipeline on Java 7` and save to the default branch (`multi-env-pipeline` branch).
3. View the pipeline logs to verify that the pipeline ran on Java 7.

Task: Stash and unstash files from one stage to the next and run stages on Java 8

Edit your Pipeline from the `multi-env-pipeline` branch:

1. Update the Pipeline's agent to **none**.
2. Update each stage's agent to use **node** with the label `java8`.
3. Add a **Stash some files to be used later in the build** step as the *last* step of the Fluffy Build stage. This stashes some files to be used in the test and deploy stages.
   - **Name**: `Java 8`
   - **Includes**: `target/**`

4. Add a **Restore files previously stashed** step as the *first* step of each of the Fluffy Test stages (Backend, Frontend, Performance, and Static) and the Fluffy Deploy stage.
   - **Name**: `Java 8`

     This restores files previously stashed with the name `Java 8`.

5. Save the pipeline. In the **Save** dialog, provide the description `Run stages on Java 8` and save to the default branch (`multi-env-pipeline` branch).
6. View the pipeline logs to verify that various stages ran on Java 8.

Task: Build and run tests in parallel on two different agent environments

Edit your pipeline from the `multi-env-pipeline` branch:

1. Make Fluffy Build a parallel stage with two stages:
   a. Add a Build Java 8 stage.
   b. Rename the Fluffy Build stage Build Java 7.
   c. Add a **Shell Script** step to Build Java 8. Copy and paste the **Shell Script** content in Build Java 7.
2. Update the node on which each of the Build stages runs and stashes files:
   a. Update the Build Java 7 stage to build on the `java7` node and stash its target folder with the name `Java 7`.
   b. Update the Build Java 8 stage to build on the `java8` node and stash its target folder with the name `Java 8`.
   c. Confirm that the Build Java 7 stage has an **Archive the artifacts** step, archiving artifacts to `target/*.jar`. The Build Java 8 stage should not have an *Archive the artifacts* step.
   d. Confirm that **Stash some files to be used later in the build** is the last step in both stages.
3. Create a total of eight stages running in parallel on Java 7 and Java 8 in the Fluffy Test stage:
   a. Rename Backend, Frontend, Performance, and Static to Backend Java 7, Frontend Java 7, Performance Java 7, and Static Java 7.
   b. Make these four stages test and unstash on Java 7. (Update the agent node label and unstash to `java7`.)
   c. Create four corresponding stages (Backend Java 8, Frontend Java 8, Performance Java 8, and Static Java 8) to run and unstash on Java 8.
   d. Copy the steps in each Java 7 test stage and paste them in each corresponding Java 8 test stage.
4. Update the Fluffy Deploy stage to run and unstash on Java 7.
5. Save the pipeline. In the **Save** dialog, provide the description `Build and test Java 7 and 8 in parallel` and save to the default branch (`multi-env-pipeline` branch).
6. View the pipeline logs to verify that each stage ran on the appropriate Java version.

Solution - Part 1

This is a good point to check that your pipeline looks as expected.

Task: Wait for user input before deploying

Edit your pipeline from the `multi-env-pipeline` branch:

1. Add a Confirm Deploy stage with a **Wait for interactive input** step:
   - **Agent**: **none**
   - **Message**: `Okay to deploy to staging?`
   - **Ok**: `Yes`
2. Open the Blue Ocean Code Editor:
   - Move (cut and paste) the code for the Fluffy Deploy stage to be after the Confirm Deploy stage.

     Alternatively, you can use a text editor to move the Fluffy Deploy stage and paste the code back in the Blue Ocean Code Editor.

   - Select **Update** to save the changes from the editor.
3. Save the pipeline. In the **Save** dialog, provide the description `Confirm deploy to staging` and save to the default branch (`multi-env-pipeline` branch).
4. Run the Pipeline and verify that the pipeline waits for input as expected.
5. Try confirming (select **Yes**) or canceling (select **Abort**).

Task: Save a Pipeline to the master branch

1. Edit your pipeline from the `multi-env-pipeline` branch.
2. Save the pipeline. In the **Save** dialog, provide the description `Change to Multi-env Pipeline in master` and save to the `master` branch.

Solution

Jenkinsfile (final)
```
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 Java 8') {
          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: 'Yes')
      }
    }
    stage('Fluffy Deploy') {
      agent {
        node {
          label 'java7'
        }

      }
      steps {
        unstash 'Java 7'
        sh './jenkins/deploy.sh staging'
      }
    }
  }
}
```