

Continuous Delivery and Containerization

Levering Jenkins 2 and Docker to simplify and standardize your pipeline

Revision 1.4 - 11/26/17

Brent Laster

IMPORTANT NOTES:

1. You must have internet access through your VM for some of the labs.
2. If you run into problems, double-check your typing!
3. Highlighted text is a description of what the solution will contain. Feel free to try and come up with the solution before looking at the answers.

Lab 1 starts on the next page!

Lab 1 – Creating a standalone Docker node

Purpose: In this lab, we'll see how to take an existing Docker image for a Jenkins node and wire it up as an agent to run a pipeline script on.

1. Click on the icon to open up Jenkins (or open a browser and go to <http://localhost:8080>). Log into the system with userid = "jenkins2" and password the same.
2. Now we want to setup our Docker image as a Jenkins agent. We have a local Docker registry on the VM with a copy of the standard jenkins ssh-slave image in it. To create a usable container from this, it requires passing in an SSH public key that is accessible to Jenkins.

We will pass that as an argument to the Docker run command by cat'ing the file as part of the command. As well, we will pass in the tool location for our Gradle instance since we have a script that is referencing that.

On the VM, open a terminal session and run the command to start an instance of the container running.

This should use the Docker command line to run the *localhost:5000/jenkinsci/ssh-slave* image, passing in an environment variable setting *JENKINS_SLAVE_SSH_PUBKEY* equal to the results of the *.ssh/jenkins2.pub* file contents. It should also pass in a volume mapping */usr/share/gradle* to the same on the container.

The command is below. If you type this in, pay attention to spelling and the back ticks around `cat .ssh/jenkins2.pub`` (This command assumes you are in the home directory for diyuser2 – the directory that opens up when you start a terminal session.)

```
docker run -e "JENKINS_SLAVE_SSH_PUBKEY=`cat .ssh/jenkins2.pub`" -v /usr/share/gradle:/usr/share/gradle localhost:5000/jenkinsci/ssh-slave
```

3. After you issue the command, you'll see Docker pull the image and start running it. You'll see what the image is doing, ending with it listening on port 22. We now need to get the ip address of the running container. Open a new terminal and locate the container id with the "docker ps" command. Then find the ip address by grepping in the inspect command. See commands below.

```
docker ps
```

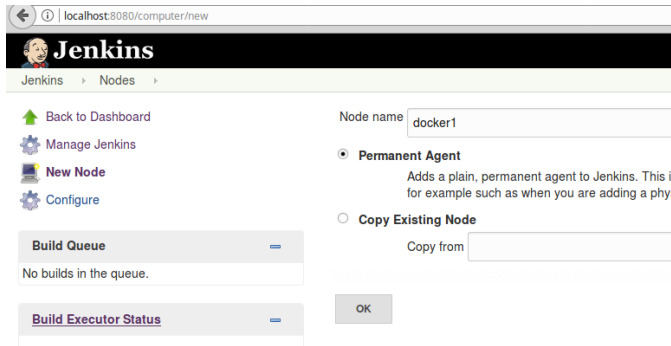
Note the first 3 characters from the "Container Id" field in the line for the "localhost:5000/jenkinsci/ssh-slave" image. Use that in the command below in place of <container-id>.

```
docker inspect <container-id> | grep IPAddr
```

Make a note of the IPAddr value here – you'll need it for the next step.

4. Now we need to configure Jenkins to use this image as a node. Switch back to Jenkins, go to the "Manage Jenkins" section and click on "Manage Nodes".

5. In the upper left part of the Nodes page, click on “**New Node**”. On the next page, fill in the “**Node name**” field with a name and click on “**Permanent Agent**”. Then click the “**OK**” button.



6. There are only a couple of fields on the next screen that we have to change. Others, such as the “**Description**” can be modified as you see fit.

For the “**Remote root directory**” field, enter

/home/jenkins

(This is the home directory on the docker image we’re using.)

For the “**Launch method**” field, select

Launch slave agents via SSH

For the “**Host**” field, enter the ip address you gathered in step 5. Usually that’s

172.17.0.# (where # is some digit)

For the “**Credentials**” field, select the credentials named “**Jenkins-ssh**”.

For the “**Host Key Verification Strategy**”, select the “**Non verifying Verification Strategy**” for simplicity.

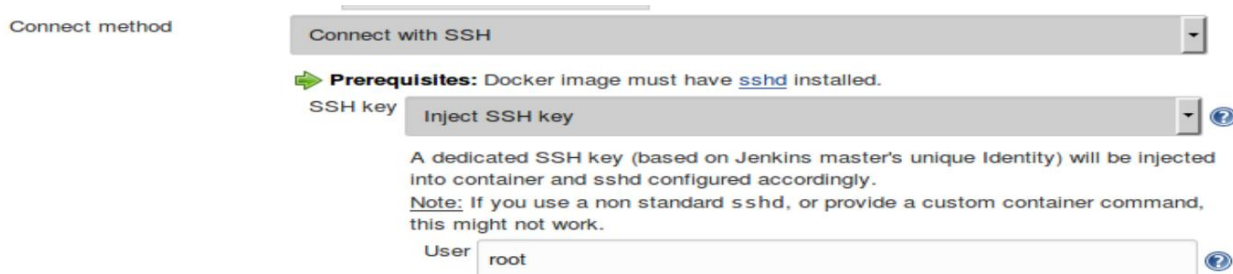
7. Save your changes and your new node should be launched. (If you look at the log, you can see the output from starting the agent.)
8. Now that we have the agent up and running, we can use it in a job. Switch back to Jenkins, select the “**lab1**” job and go to the **Configure** page.
9. Update the node statement in the job to change it to use the name you gave the agent. Save your changes and do a “**Build Now**” to start the job running. It should run to completion.

Lab 2 – Adding a Docker Cloud

Purpose: In this lab, we’ll configure the Docker plugin to use as a cloud – meaning spinning up the agent, using it, and spinning it down.

1. First, we need to configure the global configuration for the cloud. Go to the Jenkins dashboard, then to **Manage Jenkins**, then to **Configure System**.
2. Scroll down to the “**Cloud**” section, then select “**Add a new cloud**” and then select “**Docker**”.

3. Provide a name for the cloud, such as **“docker-cloud-node”**.
4. And for the **Docker URI**, you can use: **“unix:///var/run/docker”**.
5. You can then click on **“Test Connection”** and you should get a response that indicates the version of Docker that is installed.
6. Now that we have the basic configuration done, we need to add a Docker image to use. Click on the **“Docker Agent Templates...”** and then on **“Add Docker Template”**.
7. In the Docker Agent templates area, set a label for our Docker images in the **“Labels”** area. For example, you can add a label of **“docker2”**.
8. In the **“Docker Image”** section, enter the name of our image: **“localhost:5000/jenkinsci/ssh-slave:latest”**
9. Click on the **“Container Settings”** button to expand that area.
10. In the **“Volumes”** section, we need to add the path to our gradle instance so we can have it automatically mounted as read-only. In that section, enter **“/usr/share/gradle:/usr/share/gradle:ro”**.
11. Further down, for **“Connect method”**, select **“Connect with SSH”**. For **“SSH Key”**, select **“Inject SSH key”**. You can leave the user as **“root”**.



The screenshot shows the Jenkins configuration interface for a Docker agent. The 'Connect method' dropdown is set to 'Connect with SSH'. Below it, a green arrow icon points to the text 'Prerequisites: Docker image must have [sshd](#) installed.' The 'SSH key' dropdown is set to 'Inject SSH key'. Below this, there is a text box explaining that a dedicated SSH key will be injected into the container and sshd configured accordingly. A note states: 'Note: If you use a non standard sshd, or provide a custom container command, this might not work.' At the bottom, the 'User' field is set to 'root'.

12. Save your changes.
13. Go back to the dashboard and open up the **“lab2”** project. Go to its configuration page.
14. Change the **“agent any”** line at the top to use the label of the docker node we just setup:

agent { label 'docker2' }

15. Save your changes and select **“Build Now”** to see the agent run.
16. While the build is running, you can switch back to the dashboard and see the new Docker node listed in the nodes list. You can click on the link for it and see it being started and the log.
17. You can also go back to the **“Manage Jenkins”** page and take a look at the Docker **“dashboard”**.



[Manage Nodes](#)

Add, remove, control and monitor the various nodes that Jenkins runs jobs on.



[Docker](#)

Plugin for launching build Agents as Docker containers



[About Jenkins](#)

See the version and license information.

This will give you an overview of the running containers and images.

Lab 3 - Using a Docker agent in a pipeline stage

Purpose: In this lab, we'll see how to use a docker image at the granularity of an individual stage of a pipeline.

1. In Jenkins, open the "lab3" project and click on the "Configure" link. Looking at the code, notice that the "Source" stage is already done for you.
2. We're going to add a new "Build" stage. Start out by adding the framework for the stage. It will go after the "Source" stage and before the closing bracket of the script.

This should be a new stage referenced as 'Build' with an empty *steps* block.

The lines to add are show in bold below.

```

pipeline {
    agent any
    stages {
        stage('Source') {
            steps {
                cleanWs()
                git 'git@diyvb2:repositories/mavenapp.git'
            }
        }
        stage('Build') {

            steps {

                }
            }
        }
    }
}

```

3. Now, since we don't have maven installed on this system, we want to use a Docker image with Maven on it for this stage of our pipeline. There is a small image tagged as "maven:alpine" in our local repository. Add an agent block in the "Build" stage to use that image.

This will be an agent definition that defines a docker agent. The docker agent will take an image parameter corresponding to `'localhost:5000/maven:alpine'` and an args parameter passing in our local maven repo at `/home/diyuser2/.m2` as `/.m2` for a volume.

The lines to add are in bold below.

```
stage('Build') {  
    agent {  
        docker {  
            image 'localhost:5000/maven:alpine'  
            args '-v /home/diyuser2/.m2:/.m2'  
        }  
    }  
    steps {  
  
        }  
    }  
}
```

Notice that we are using an image from our local repository. Also notice that we are passing in the local maven cache from the host system so that we don't have to download the dependencies again.

4. Next, we'll add the actual step to do the Maven build. Add the step below (in bold).

This will be a call to the *shell step* in Jenkins to run *maven* passing in a designated local repository of `/.m2/repository` and calling the *compile* and *package* targets.

```
steps {  
    sh 'mvn -Dmaven.repo.local=/.m2/repository compile package'  
    }  
}
```

Notice that we are passing in the path to the local maven repo that we passed as a volume.

5. Save your changes and **"Build Now"**. Did it build successful or not. Why?
6. To correct the failure, we need to tell Jenkins to have the Docker agent reuse the same workspace as the one where we pulled the source code.

To do this we add a “reuseNode” option in the Docker agent specification setting its argument to true. Add the line in bold below.

```
stage('Build') {  
    agent {  
        docker {  
            image 'localhost:5000/maven:alpine'  
            args '-v /home/diyuser2/.m2:/m2'  
            reuseNode true  
        }  
    }  
}
```

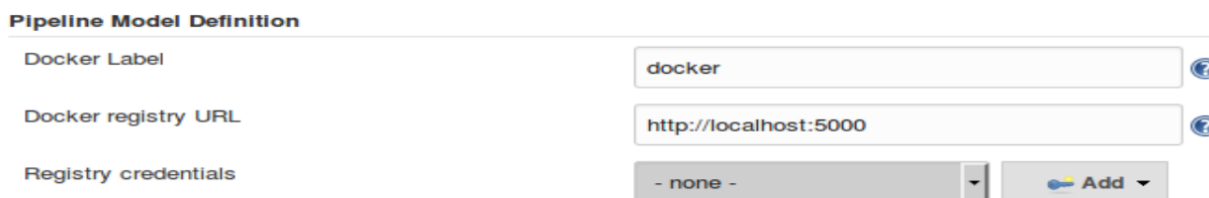
7. Save your changes and execute “**Build Now**” again. This time, the job should run without errors.

Lab 4 – Configuring the Pipeline Model Definition

Purpose: In this lab, we’ll see how to use the Pipeline Model Definition section of the global configuration to set some defaults for Docker use in declarative pipelines.

Within Jenkins, not all agents may be capable of running Docker. We want to have Docker agents used in declarative pipelines run on a node capable of running Docker. Also, we want to setup the default repository to be our local one if we don’t specify.

1. Go back to the list of nodes available in our Jenkins system. Go to the dashboard, then to “**Manage Jenkins**”, and then to “**Manage Nodes**”.
2. For “**worker_node2**” and “**worker_node3**”, click on the “**Configure**” menu option and note what labels are set for both.
3. Now, go back to the “**Configure System**” section of Jenkins. Scroll down until you find the “**Pipeline Model Definition**” section. This is where we will specify the default label for which node to execute Docker agents on (if not overridden). Also, we will specify the default registry.
4. In the **Pipeline Model Definition** section, in the “**Docker Label**” field, enter the label “**docker**” (that was on worker_node3) to indicate that Docker agents should run on that node by default.
5. In the “**Docker Registry**” field, enter <http://localhost:5000> so our local registry will be the default one. You do not need to supply any credentials. After this step, the section should look like the picture below.



Pipeline Model Definition

Docker Label:

Docker registry URL:

Registry credentials: [Add](#)

6. **Save** your changes to the system configuration page.
7. Now, switch back to the Dashboard. We're going to create a new item based on the completed lab3 job. Click on **"New Item"**. Give it a name of **"lab4"**. Scroll to the bottom of the screen and select **"Copy from"**. Enter **"lab3"** as the job to copy. Click **"OK"**.
8. You should now be in the **"lab4"** job and on its configuration page. (If not, open the job and go to the configure page.) In the **"Pipeline"** section, change the **"agent any"** line at the top to use `worker_node 2` by specifying the **"linux"** label in the agent declaration.

agent {label 'linux'}

9. Now, edit the agent specification in the **"Build"** stage to **remove** (or comment out) the **"reuseNode true"** line. Also, modify the name of the image to **remove** the **"localhost:5000"** prefix. Afterwards, the agent specification should look similar to this (changed lines bolded).

```
agent {  
  docker {  
    image 'maven:alpine'  
    args '-v /home/diyuser2/.m2:/m2'  
    // reuseNode true  
  }  
}
```

10. Go to a terminal session and remove the existing `localhost:5000/maven:alpine` image using the Docker `rmi` command.

docker rmi localhost:5000/maven:alpine

11. Save your changes and **"Build Now"**. Notice that the build fails. Why?
12. Open the **Console Log** for the build. Find where the project was running on **worker_node2** at the start. Then, find the build section and notice that it was running on **worker_node3** because it was a Docker agent and had `worker_node3's "docker"` label set as the default. This is intended but also presents a problem – the source was pulled to `worker_node2` but the compile ran on `worker_node3`. We'll fix this in a moment.
13. Also notice that the **maven:alpine** image was pulled from our local registry. If we didn't have the default, then it would have been pulled from the Docker hub.
14. To fix the build failure, we can use the DSL steps **"stash"** and **"unstash"** to transfer files between nodes. To get the syntax for the **"stash"** command, we'll use the **"Snippet Generator"**. Go back to the **"project"** page for **lab4**

and click on the “**Pipeline Syntax**” link on the left. Or you can go to the “**configure**” page and click on the “**Pipeline Syntax**” link at the bottom of the page.

15. You’ll now be on the page for the “**Snippet Generator**”. In the **Steps** field, find the entry for “**stash**”.
16. You’ll now have a template on screen that you can fill in to form the syntax for the step. For the **Name** field, you can provide any name, for example “**src**” (no quotes). For the **Includes** field, you can just put in “******” (no quotes). The two asterisks are shorthand for all of the files and directories.
17. Now click on the “**Generate Pipeline Script**” button. Copy the generated text from the box below and place it **after** the **git** step in the **lab4** pipeline script. That section should now look like this:

```
stage('Source') {
    steps {
        cleanWs()
        git 'git@diyvb2:repositories/mavenapp.git'
        stash includes: '**', name: 'src'
    }
}
```

18. Now we need to “**unstash**” in the steps for the build stage since that runs on a different node.

This is simply a call to the unstash step passing the name of the stash we created above.

Just add the line as shown in bold below.

```
steps {
    unstash 'src'
    sh 'mvn -Dmaven.repo.local=/.m2/repository compile package'
}
```

19. **Save** your changes to **lab4**’s configuration and **build** it again. This time it should build successfully.

Lab 5 – Creating a Docker image in Jenkins

Purpose: In this lab, we’ll see how to use the Docker global variable to create an image and put it into our local repository.

1. In Jenkins, open the “**lab5**” job and go to the “**Configure**” screen. Here we have the node block and the step to retrieve the Dockerfile from source control. (You can look at the Dockerfile using the Dockerfile shortcut on the desktop.)
2. For this Docker image, we are going to add a newer version of Gradle – which has already been downloaded to the VM in **/home/diyuser2/docker-work/gradle-dist**.

This will be a line to use the *shell* step from Jenkins to call *cp* to recursively copy over */home/diyuser2/docker-work/gradle-dist*.

Add the line in bold below to do this.

```
node() {
    stage('Build Image') {
        // download the Dockerfile to build from
        git 'git@diyvb2:repositories/docker-gradle.git'

        // copy over our gradle distribution for the build context
        sh 'cp -R /home/diyuser2/docker-work/gradle-dist .'
    }
}
```

3. Next, we'll use the **"build"** method to tell the pipeline code to build off of the default Dockerfile and with the tag of **"gradle4.3:snapshot"** – by adding the line in bold below.

This should be a call to the build method of the docker global variable with the result assigned to the myImg variable.

```
// build our docker image
myImg = docker.build 'gradle4.3:snapshot'
```

4. Now we need to push the newly created image back to our local repository so we can use it in the next lab.

This is just a call to the push method call on the myImg variable.

Add the push line below to the script.

```
// push it back into the local registry

myImg.push()
}
```

5. If we were to try and run this now, it would attempt to push our image out to the public Docker hub. Since we just want this available locally, we can use the **"withRegistry"** method to point it to the registry.

This should be a call to the withRegistry method on the docker global variable using the local registry url.

Add the lines below in bold.

```
// push it back into the local registry
docker.withRegistry("http://localhost:5000") {
```

```
        myImg.push()  
    }
```

6. Save your changes and build the project.

Lab 6 – Pulling an image and using the inside command

Purpose: In this lab, we'll use the image we built in the last lab and see how to use the Docker “inside” method to startup a container, map volumes, and execute commands easily within it.

1. In Jenkins, open the “lab6” job and go to the “Configure” page.
2. The first thing we want to do is point our “myImg” variable to the desired image.

This will use the docker image method to build the default Dockerfile and produce an image named “localhost:5000/gradle4.3:snapshot”. The result will be pointed to by *myImg*.

Add the line in bold below in the script.

```
node ('worker_node2') {  
    def myImg  
    myImg = docker.image("localhost:5000/gradle4.3:snapshot")
```

3. Now we need to pull that image from the local repository.

This is simply invoking the pull method on *myImg*.

Add the step below.

```
node ('worker_node2') {  
    def myImg  
    myImg = docker.image("localhost:5000/gradle4.3:snapshot")  
    myImg.pull()
```

4. In the next section, we already have our git step to pull the source code. But we want to make this run on the Docker image, not on our current node. Wrap the git step with an “inside” method call (passing in the location of the Git repositories) so it can find the volume and get the code.

This involves invoking the *inside* method on *myImg*, passing in */home/git/repositories* as a volume, and creating a *block ()* around the git command.

```
stage('Get Source') {
```

```

        myImg.inside('-v /home/git/repositories:/home/git/repositories') {
            git '/home/git/repositories/gradle-greetings.git'
        }

```

5. Finally, in the “**Build**” stage, we want to have our commands run in the Docker image not on the current node. Add an “inside” block around those statements as well. (We do not need to pass any additional arguments.)

This is simply a block around the statements with a call to the inside method (no arguments).

```

stage('Build') {
    myImg.inside() {
        sh 'gradle -version'
        sh 'gradle clean build'
    }
}

```

6. Save the changes to the job and do a “Build Now” to see it execute.

Lab 7 – Using containers together

Purpose: Learn how to work with multiple containers and link them together in a pipeline script.

In this lab, we’ll look at how to run containers, gather information from containers, and how to link containers together (like a compose). The architecture here is that we have a database image and a frontend image that we will link together to form the application. We will also have the script print out a link where we can access the running application.

1. Open the “**lab7**” project in Jenkins. Click on the “**Configure**” link.
2. We’ll start out by getting the database container running. Note that there are several environment variables being passed in here. Add the line in bold just after the **withCredentials** block definition (between lines 3 and 4). Note that the step is continued on the next page.

```

node {
    stage('runApp') {
        withCredentials([usernamePassword(credentialsId: 'mysql-admin', passwordVariable:
            'mysqlAdminPass', usernameVariable: 'mysqlAdminUser'), usernamePassword(credentialsId: 'mysql-
            root', passwordVariable: 'mysqlRootPass', usernameVariable: '')]) {
            docker.image('bclaster/ref-db-image').withRun('-e
"MYSQL_USER=${mysqlAdminUser}" -e

```

```
"MYSQL_PASSWORD=${mysqlAdminPass}" -e "MYSQL_DATABASE=registry"
-e "MYSQL_ROOT_PASSWORD=${mysqlRootPass}" -p 3308:3306')
```

```
{
```

- Next, we will leverage the object created by **withRun**, denoted by “**db**” and use it to pass custom arguments to another container (the one for the web frontend). We’ll use the “**id**” property of the container to establish a link. Add the line in bold below:

```
"MYSQL_ROOT_PASSWORD=${mysqlRootPass}" -p 3308:3306')
{
db ->
    docker.image('bclaster/ref-web-image').withRun("-p 8089:8080 --link
${db.id}:mysql")
```

This will start the web frontend container running, linked to the database container and able to reference it as “**mysql**”.

- Now, we want to be able to inspect the web container, find its ip address, and expose it so we can look at our webapp. Doing this is a multi-step process. First, near the bottom of the script, we have a placeholder for a simple function. We’re going to leverage running Docker via the shell to run a docker “**inspect**” command and find the ip address.

Add the line in bold below to the body of the subroutine.

```
def containerInfo(container) {
    sh "docker inspect --format '\n{{.Name}} is available at

"
}
```

- With the function setup, we can pass the object created by the web image’s **withRun()** to it. To do this, add the line below to have the container object invoke the routine with it’s id.

This is just a call to the *containerInfo* method passing in the *web* reference.

```
docker.image('bclaster/ref-web-image').withRun("-p 8089:8080 --link ${db.id}:mysql")
{
web ->
    containerInfo(web)
```

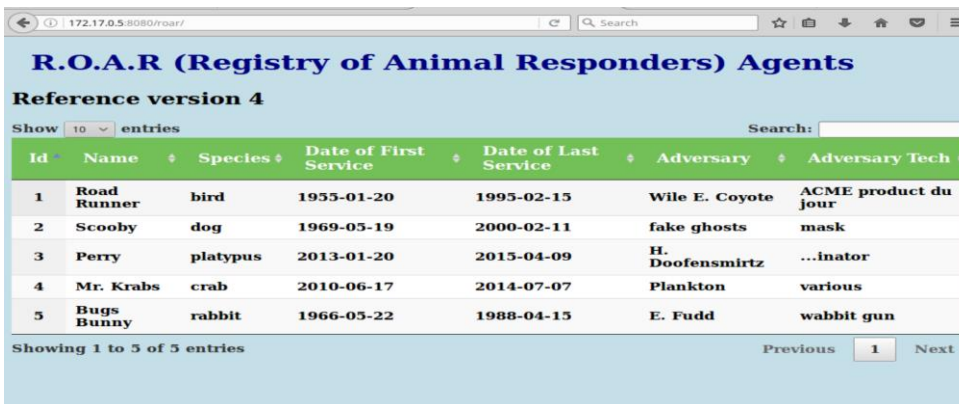
- Save your changes and do a “**Build Now**” on the job. In the console output, you should see the link displayed to the container web front end. It will look similar to the screenshot below.

```
[Pipeline] sh
[lab7] Running shell script
+ docker ps -q -l
+ docker inspect --format
{{.Name}} is available at http://{{.NetworkSettings.IPAddress }}:
d16a08966071

/nostalgic_torvalds is available at http://172.17.0.5:8080/roar

[Pipeline] timeout
Timeout set to expire in 2 min 0 sec
[Pipeline] {
```

You can click on that to see the application running. It will look like this:



Id	Name	Species	Date of First Service	Date of Last Service	Adversary	Adversary Tech
1	Road Runner	bird	1935-01-20	1995-02-13	Wile E. Coyote	ACME product du jour
2	Scooby	dog	1969-05-19	2000-02-11	fake ghosts	mask
3	Perry	platypus	2013-01-20	2015-04-09	H. Doofensmirtz	...inator
4	Mr. Krabs	crab	2010-06-17	2014-07-07	Plankton	various
5	Bugs Bunny	rabbit	1966-05-22	1988-04-13	E. Fudd	wabbit gun

- The timeout statement allows the containers to stay active for 2 minutes so you can have time to go out and look at them. The input statement allows you to cancel out early if you are done.

=====

END OF LABS

=====