WA2894 Booz Allen Hamilton Tech Excellence Modern Software Development Program - Week 3

Student Labs

Web Age Solutions Inc.

Table of Contents

Environment

In this course we will use the following VM images:

- **Labs 1 to 4: PROJECT_VM_2020-07-14**
- **Labs 5 to 7: VM_MSD_REL_1_0**

Lab 1 - Getting Started with Kubernetes

Kubernetes is an open-source container orchestration solution. It is used for automating deployment, scaling, and management of containerized applications. In this lab, you will explore the basics of Kubernetes. You will use Minikube that allows you to create a Kubernetes environment with ease.

The following labs requires a VM named 'PROJECT_VM_2020-07-14', version could be higher but not smaller.

Close other VMs if running and double-click the shortcut on the desktop to launch the VM. Use osboxes.org for the user name and password.

This Lab requires the internet connection to work.

Part 1 - Setting the Stage

__1. Open a new Terminal window by clicking **Applications** > **Terminal**.

__2. Switch the logged-in user to **root**:

```
sudo -i
```

When prompted for the logged-in user's password, enter **osboxes.org**

__3. Enter the following command:

```
whoami
```

You should see that you are **root** now.

```
root
```

Note: Now that you are **root** on your Lab server, beware of system-wrecking consequences of issuing a wrong command.

Part 2 - Start the Cluster

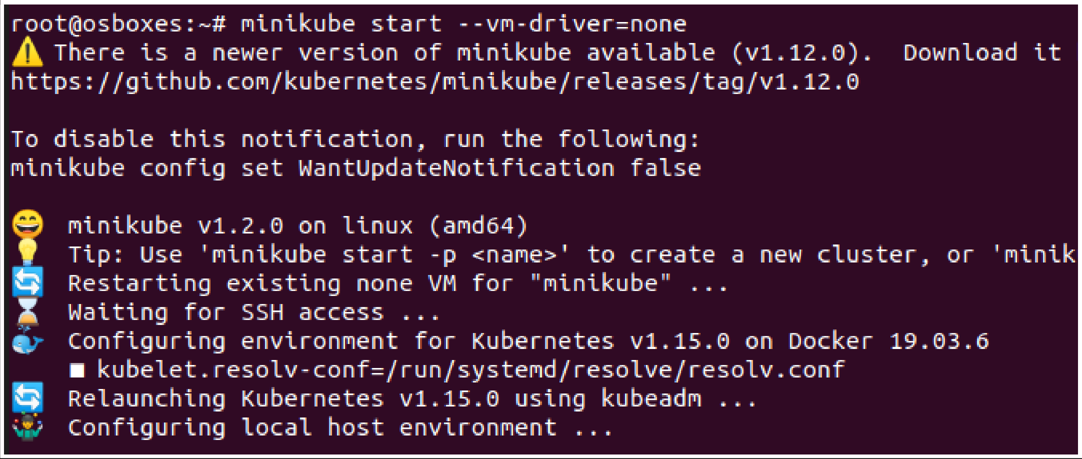
In this part, you will start the Kubernetes cluster and interact with it in various ways.

__1. Run the following command to start a cluster:

```
minikube start --vm-driver=none
```

It will take between 5-10 minutes for Minikube to start up. In case if you see "apiserver timed out" error message, run the command (minikube start) again 2 or 3 times.

Wait until it switches back to the terminal.

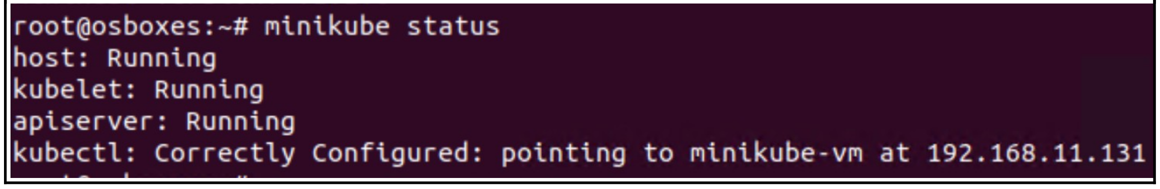


```
root@osboxes:~# minikube start --vm-driver=none
⚠️ There is a newer version of minikube available (v1.12.0).  Download it
https://github.com/kubernetes/minikube/releases/tag/v1.12.0

To disable this notification, run the following:
minikube config set WantUpdateNotification false

😄  minikube v1.2.0 on linux (amd64)
💡  Tip: Use 'minikube start -p <name>' to create a new cluster, or 'minik
🔄  Restarting existing none VM for "minikube" ...
⌛  Waiting for SSH access ...
🐳  Configuring environment for Kubernetes v1.15.0 on Docker 19.03.6
    ■ kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
🔄  Relaunching Kubernetes v1.15.0 using kubeadm ...
🤹  Configuring local host environment ...
```

__2. Run the following command to verify the Kubernetes cluster is running.

```
minikube status
```

Notice it shows messages like this:



```
root@osboxes:~# minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.11.131
```

__3. Verify you can execute kubectl and also obtain Kubernetes version:

```
kubectl version
```

Notice it lists a major and minor version in JSON format. Don't worry if you see a connection message.

__4. Get Kubernetes cluster information:

```
kubectl cluster-info
```

Notice it displays the IP address and port where the Kubernetes master is running. Don't worry if you see a connection message.

__5. Run the following command to obtain the cluster IP address:

```
minikube ip
```

Notice it shows the IP address of our cluster.

Part 3 - View Kubernetes Dashboard

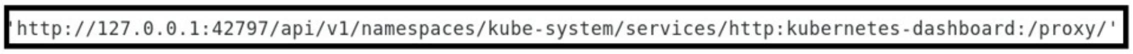
In this part, you will view the Kubernetes dashboard and interact with it.

__1. In the terminal, run following command to view the dashboard URL:

```
minikube dashboard
```

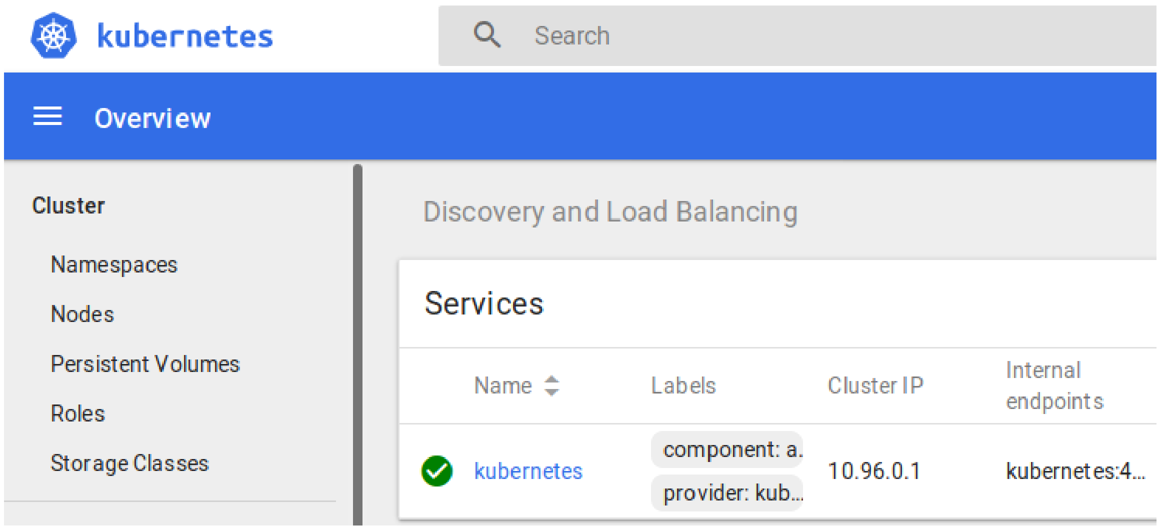
This command will give you the error that Firefox can’t be launched in root mode. This is expected.

__2. Just make a note of the URL which will show up as part of the Minikube dashboard and enter in Firefox manually. The URL looks like this:



```
'http://127.0.0.1:42797/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/'
```

__3. It will open the Firefox web browser and show the Kubernetes page. It may look different than below:



Services

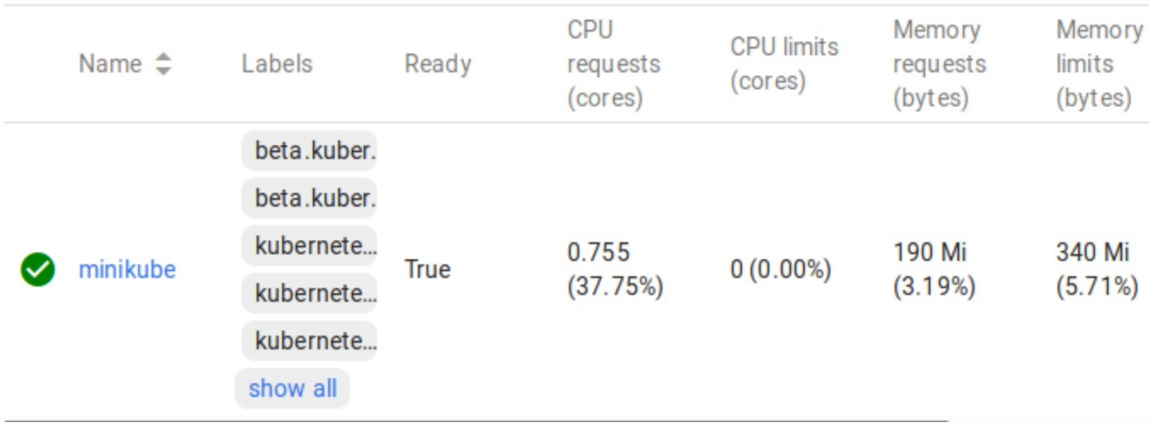
Name	Labels	Cluster IP	Internal endpoints
✓ kubernetes	component: a. provider: kub...	10.96.0.1	kubernetes:4...

Notice that the Terminal is running, if you close the Terminal or terminate the command the browser will close, so keep it open.

If you get an error that Firefox is not supported then click on the link to open the URL.

__4. On the left side of the dashboard, click **Nodes**.

Notice the page looks like this: (minikube is the name of your node. You also noticed this name in VirtualBox as a VM instance).



Name	Labels	Ready	CPU requests (cores)	CPU limits (cores)	Memory requests (bytes)	Memory limits (bytes)
✓ minikube	beta.kuber. beta.kuber. kubernete... kubernete... kubernete... show all	True	0.755 (37.75%)	0 (0.00%)	190 Mi (3.19%)	340 Mi (5.71%)

__5. Open a new terminal window and switch to root:

```
sudo -i
```

__6. Enter **osboxes.org** as the password, if prompted.

__7. In the terminal, run following command to view nodes:

```
kubectl get nodes
```

Notice you can see the node in the terminal. It's the same information you saw on the dashboard earlier in this part of the lab.

In case if you encounter any error message while obtaining the nodes, run the following command to fix it.

kubectl config use-context minikube

This command tells kubectl what cluster to connect to.

Part 4 - Create a Container

In this part, you will create a container and run it in the node/cluster i.e. you will run a workload in the cluster.

__1. Run the following command:

```
kubectl create deployment my-web-server --image=nginx
```

It uses a nginx image to create a container named my-web-server.

In case if the Docker image doesn't get downloaded properly, delete the cached data located under ~/.minikube/cache and retry.

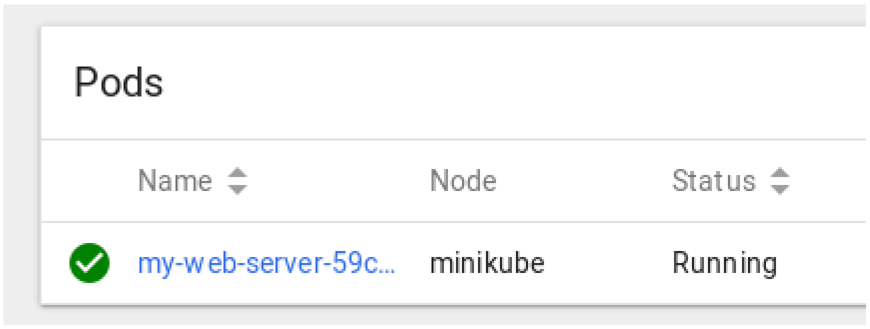
__2. Run the following command to view the pod list and make a note of the full my-web-server pod name.

```
kubectl get pods
```

Notice it shows my-web-server pod. Under READY, 0/1 means there's one container running the Nginx service and it's still created. You might have to wait for a minute or two to see the "running" status.

__3. Switch to the Firefox window, where the dashboard is open, and click **Pods** on the left side of the page.

When the pod is created completely, it would show up like this:



Pods

	Name	Node	Status
✓	my-web-server-59c...	minikube	Running

Part 5 - View Logs and Details

In this part, you will view logs and details in various ways.

__1. On the dashboard, click **my-web-server-####** pod.

Notice it shows various pod details, such as creation date, status, and other events.

__2. Make a note of the pod name under **Details** on the dashboard.

It would be something like this: my-web-server-3913049308-qmcwt

__3. In the terminal, run following command to view pod details:

```
kubectl describe pod my-web-server-####
```

Replace #### with your pod Id.

Notice you get to view similar details in the terminal as you saw in the previous steps of this part.

__4. Run the following command to view pod logs.

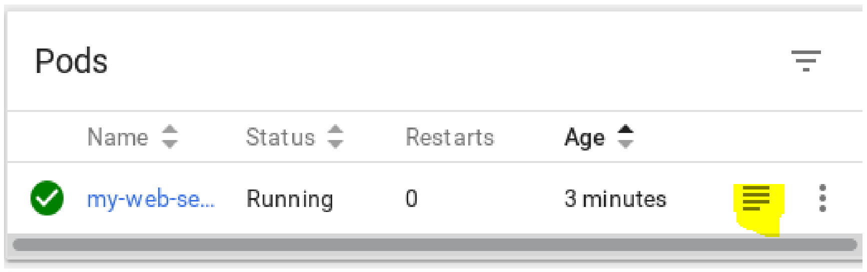
```
kubectl logs my-web-server-####
```

__5. Run the following command to view node details:

```
kubectl describe node minikube
```

Notice it shows the node details.

__6. On the dashboard, click **Pods** on the left side of the page, then click the **Logs** icon next to the pod.



my-web-se... Running 0 3 minutes

Notice it opens another tab and shows the pod log.

__7. Close the tab where the log is displayed and go back to the tab where the dashboard is open.

Part 6 - Expose a Service

In this part, you will expose Nginx service, which you deployed in the previous parts of this lab.

__1. In the terminal, run following command:

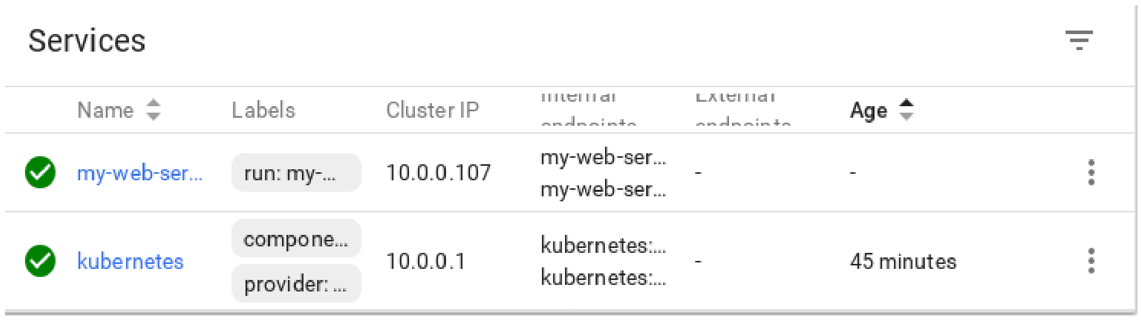
```
kubectl expose deployment my-web-server --type=NodePort --port 80
```

Notice it shows a message that service has been exposed.

Make sure there is a **double dash** before *type=NodePort*

__2. In the Firefox web browser tab where the dashboard is open, click **Services** on the left side of the page.

Notice the page looks like this:



Services

	Name	Labels	Cluster IP	Internal endpoints	External endpoints	Age
✓	my-web-ser...	run: my-...	10.0.0.107	my-web-ser... my-web-ser...	-	-
✓	kubernetes	compone... provider: ...	10.0.0.1	kubernetes:... kubernetes:...	-	45 minutes

Note: The IP address is the internal IP address. You will access the public IP address later in the lab.

__3. In the terminal run following command to view the exposed services:

```
kubectl get services
```

__4. View my-web-server service details:

```
kubectl describe service my-web-server
```

__5. In the terminal, run following command to access the service's public IP address:

```
minikube service my-web-server --url=true
```

Notice it shows IP address like this: http://192.168.99.100:31421

Note. that the URL port may be different.

Troubleshooting. If you don't get a URL then delete the service by running:

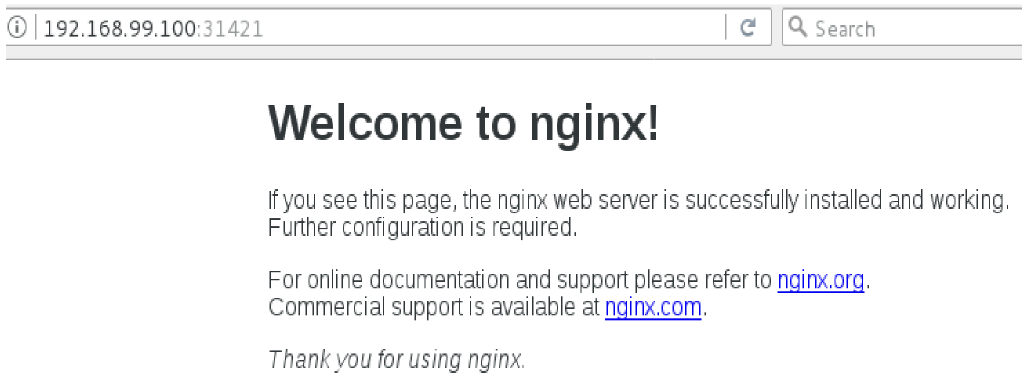
kubectl delete service my-web-server

and then start Part 7 again.

__6. Ctrl+Click the URL in the terminal. It will launch the page in the Firefox web browser.

Alternatively, you can type in the URL manually in Firefox.

Notice it shows the familiar-looking Nginx home page.



192.168.99.100:31421

Search

Welcome to nginx!

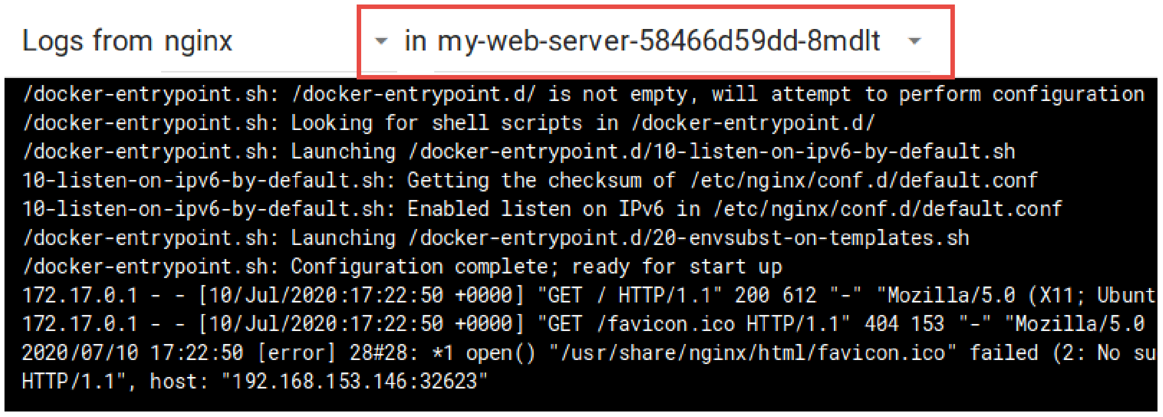
If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

__7. On the dashboard, click **Services**, then click your service, and click the Log icon next to the service.

__8. Make a note of the service name, as shown in the below.



Logs from nginx ▾ in my-web-server-58466d59dd-8mdlt ▾

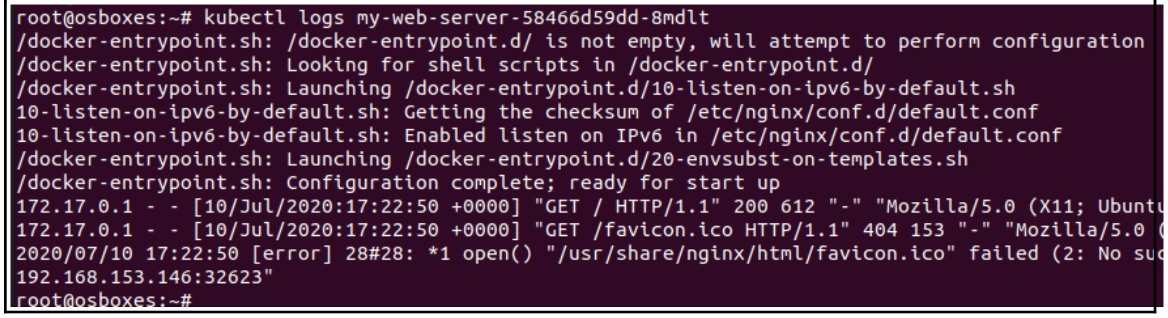
```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
172.17.0.1 - - [10/Jul/2020:17:22:50 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (X11; Ubunt
172.17.0.1 - - [10/Jul/2020:17:22:50 +0000] "GET /favicon.ico HTTP/1.1" 404 153 "-" "Mozilla/5.0
2020/07/10 17:22:50 [error] 28#28: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No su
HTTP/1.1", host: "192.168.153.146:32623"
```

__9. In the terminal run following command:

```
kubectl logs my-web-server-#####
```

Replace ##### with your service name.

Notice it shows the service access log in the terminal like this:



```
root@osboxes:~# kubectl logs my-web-server-58466d59dd-8mdlt
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
172.17.0.1 - - [10/Jul/2020:17:22:50 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (X11; Ubunt
172.17.0.1 - - [10/Jul/2020:17:22:50 +0000] "GET /favicon.ico HTTP/1.1" 404 153 "-" "Mozilla/5.0 (
2020/07/10 17:22:50 [error] 28#28: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No su
192.168.153.146:32623"
root@osboxes:~#
```

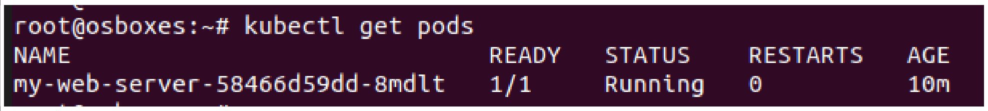
Part 7 - Scaling the Services

In the previous parts of the lab, you exposed a service. There was a single instance of the service, with one Pod that was provisioned on a single node. In this part, you will scale the service by having 3 Pods.

__1. In the terminal, run following command:

```
kubectl get pods
```

Notice it shows result like this:



```
root@osboxes:~# kubectl get pods
NAME                            READY   STATUS    RESTARTS   AGE
my-web-server-58466d59dd-8mdlt   1/1     Running   0          10m
```

Notice there's a single instance running right now.

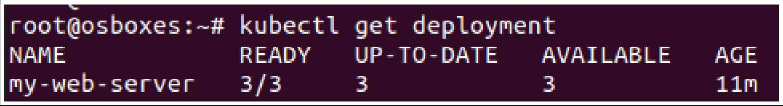
__2. Run the following command to scale the service:

```
kubectl scale --replicas=3 deployment/my-web-server
```

__3. Run the following command to get the service instance count:

```
kubectl get deployment
```

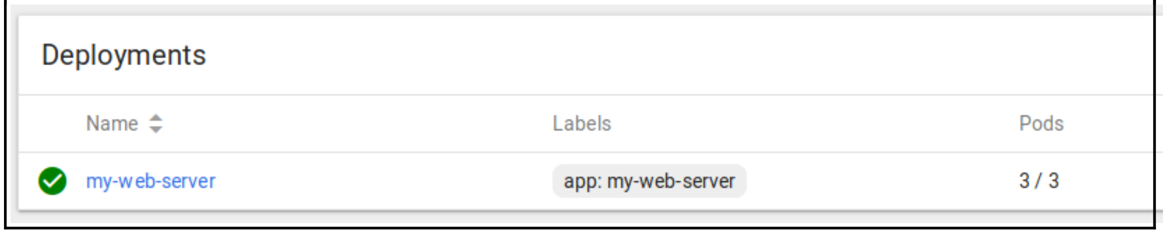
Notice it shows result like this:



```
root@osboxes:~# kubectl get deployment
NAME            READY   UP-TO-DATE   AVAILABLE   AGE
my-web-server   3/3     3            3           11m
```

__4. On the dashboard page, in the web browser, click **Deployments** on the left side of the page.

Notice it shows 3/3 Pods.

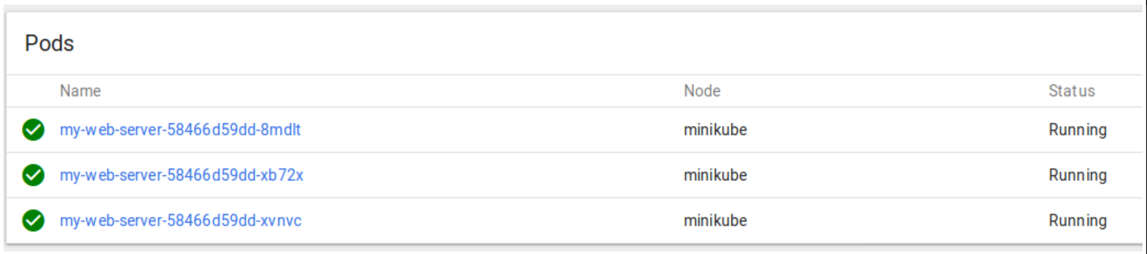


Deployments

Name	Labels	Pods
✓ my-web-server	app: my-web-server	3 / 3

__5. On the dashboard page, click **Services** on the left side of the page, then click **my-web-server** service.

Notice it shows Pods like these:

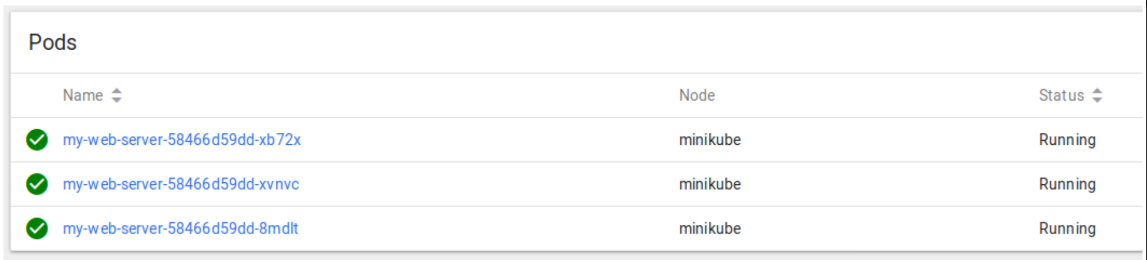


Pods

Name	Node	Status
✓ my-web-server-58466d59dd-8mdlt	minikube	Running
✓ my-web-server-58466d59dd-xb72x	minikube	Running
✓ my-web-server-58466d59dd-xvnvc	minikube	Running

__6. On left side of the page, click **Pods**.

Notice it shows Pod list like this:



Pods

Name	Node	Status
✓ my-web-server-58466d59dd-xb72x	minikube	Running
✓ my-web-server-58466d59dd-xvnvc	minikube	Running
✓ my-web-server-58466d59dd-8mdlt	minikube	Running

Part 8 - Stop and Delete the Cluster

In this part, you will stop and delete the cluster you created in this lab.

__1. In the terminal, run following command to stop the cluster:

```
minikube stop
```

__2. Run the following command to delete the cluster:

```
minikube delete
```

__3. Type exit in the Terminal and then close it.

__4. Close all open browser windows.

__5. In the Terminal that were running minikube, hit CTRL + C.

__6. Type exit in the Terminal and then close it.

Part 9 - Review

In this lab, you learned the basics of Kubernetes with minikube and kubectl.

Lab 2 - Working with Logs in Kubernetes

In this lab, you will view logs by using kubectl and also by using Fluentd, Elastic search, and Kibana. Although Kubernetes logs all messages written to standard out, if a pod is evicted from the cluster, all logs related to that pod are lost. Therefore, to write logs outside a container/pod, you will use fluentd, Elasticsearch, and Kibana. Fluentd is a logging agent that runs on each Kubernetes node and collects logs. Elasticsearch will index the contents of logs and Kibana will let you query the logs.

Part 1 - Setting the Stage

__1. Open a new Terminal window.

__2. Switch the logged-in user to **root**:

```
sudo -i
```

When prompted for the logged-in user's password, enter **osboxes.org**

__3. Enter the following command:

```
whoami
```

You should see that you are **root** now.

```
root
```

Note: Now that you are **root** on your Lab server, beware of system-wrecking consequences of issuing a wrong command.

__4. Switch to the home directory by entering the following commands:

```
cd ..
cd home/osboxes/LabFiles/logging
```

__5. Make sure you can see the content of the directory:

```
ls
```

Part 2 - Start-Up Kubernetes

In this part, you will start up the Kubernetes cluster. The lab setup comes with MiniKube preinstalled.

__1. Start MiniKube:

```
minikube start --vm-driver="none"
```

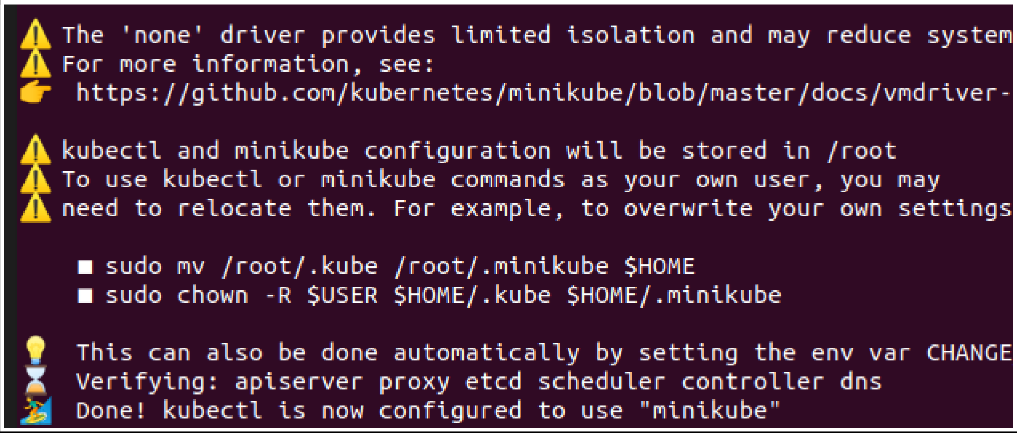
Wait for the service to start up. It can take up to 5 minutes. If it fails, try again. It’s a very resource-intensive service and takes time.

If you repeatedly get the apiserver time out error message (ensure you have tried it at least 2-3 times), delete the old cluster by running the following command:

minikube delete

..and then run the command in step 2 to recreate the cluster.

Ensure the output looks like this:

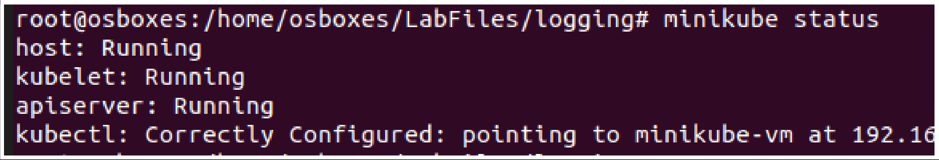


```
⚠️ The 'none' driver provides limited isolation and may reduce system
⚠️ For more information, see:
👉  https://github.com/kubernetes/minikube/blob/master/docs/vmdriver-

⚠️ kubectl and minikube configuration will be stored in /root
⚠️ To use kubectl or minikube commands as your own user, you may
⚠️ need to relocate them. For example, to overwrite your own settings

    ▪ sudo mv /root/.kube /root/.minikube $HOME
    ▪ sudo chown -R $USER $HOME/.kube $HOME/.minikube

💡  This can also be done automatically by setting the env var CHANGE
⌛  Verifying: apiserver proxy etcd scheduler controller dns
🏄  Done! kubectl is now configured to use "minikube"
```

__2. Get MiniKube status:

```
minikube status
```

Ensure your output looks like this. Your IP address might be different.



```
root@osboxes:/home/osboxes/LabFiles/logging# minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.16
```

Part 3 - Deploy Elastic Search

In this part, you will deploy Elastic search portion of the Elastic Stack.

__1. Run the following command to create a namespace to organize the pods for logging:

```
kubectl create namespace logging
```

__2. Deploy Elastic Search component of the Elastic Stack:

```
kubectl create -f kubernetes/elastic.yaml -n logging
```

__3. Verify the pods are created:

```
kubectl get pods -n logging
```

__4. If the pod(s) show 0/1, wait for a few minutes and run the command again. It must show 1/1 and Running status before you proceed to the next step.

__5. Verify the service(s) were created:

```
kubectl get service -n logging
```

Part 4 - Deploy Kibana

In this part, you will deploy Kibana and verify it got deployed.

__1. Run the following command to deploy Kibana:

```
kubectl create -f kubernetes/kibana.yaml -n logging
```

__2. Verify the pods are created:

```
kubectl get pods -n logging
```

__3. If the pod(s) show 0/1, wait for a few minutes and run the command again. It must show 1/1 and Running status before you proceed to the next step.

__4. Verify the service(s) were created:

```
kubectl get service -n logging
```

Part 5 - Deploy Fluentd

In this part, you will deploy a Fluentd logging agent to each node in the Kubernetes cluster, which will collect each container's log files running on that node. Fluentd will be deployed as a DaemonSet.

__1. Run the following command to configure RBAC (role-based access control) permissions so that Fluentd can access the logs:

```
kubectl create -f kubernetes/fluentd-rbac.yaml
```

The above command creates a ClusterRole which grants get, list, and watch permissions on pods and namespace objects. The ClusterRoleBinding then binds the ClusterRole to the ServiceAccount within the kube-system namespace.

__2. Deploy fluentd as a DaemonSet so it can run on each Kubernetes node:

```
kubectl create -f kubernetes/fluentd-daemonset.yaml
```

If you're running Kubernetes as a single node with Minikube, this will create a single Fluentd pod in the kube-system namespace.

__3. Verify fluentd is deployed and running:

```
kubectl get pods -n kube-system
```

Ensure fluentd is listed in the list. Make a note of the pod name. You will use it in the next step. Make sure fluentd shows 1/1.

__4. Take a note of the logs:

```
kubectl logs <fluentd-pod_name> -n kube-system | grep "Connection
opened"
```

```
[Enter this command in one line]
```

Use the fluentd pod name you noted down in the previous step of this part.

Ensure it shows a message like this:

Connection opened to Elasticsearch cluster =>

  {:host=>"elasticsearch.logging", :port=>9200, :scheme=>"http"}

Part 6 - Connect to Kibana

__1. Obtain Kubernetes cluster IP address:

```
minikube ip
```

Make a note of the IP address. You will use it later in this part.

__2. Obtain port number assigned to Kibana service:

```
kubectl get services -n logging | grep kibana
```

Make a note of the port number in front of /TCP. e.g. in the following example, the port number is 32683

```
kibana        NodePort   10.106.226.34    <none>       5601:32683/TCP   74s
```

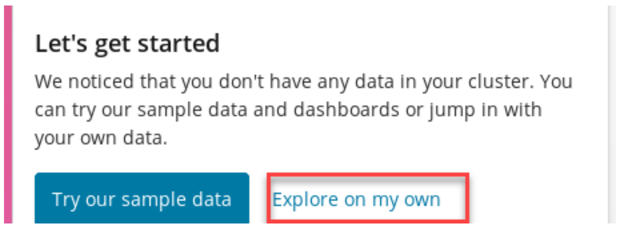
__3. Open Firefox web browser in the VM and enter the following URL:

```
http://<MINIKUBE_IP>:<KIBANA_PORT>
```

Note: Ensure you use the values you noted in the previous steps of this part.

Ensure Kibana web page shows up.

__4. Click **Explore my own**.



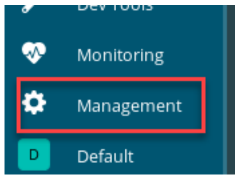
Let's get started

We noticed that you don't have any data in your cluster. You can try our sample data and dashboards or jump in with your own data.

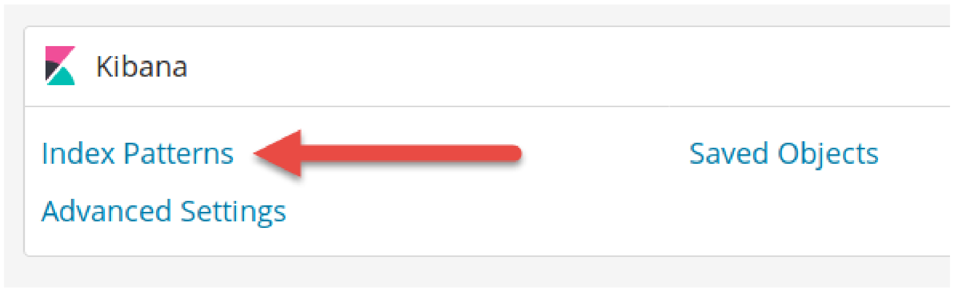
Part 7 - Configure Kibana

In this part, you will configure Kibana so it can get logs from fluentd.

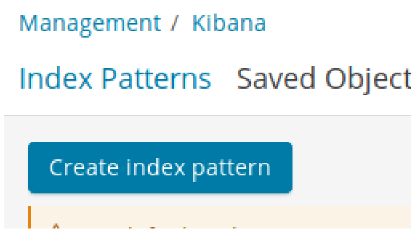
__1. In Firefox web browser where fluentd page is open, click **Management** on the left-hand side navigation.



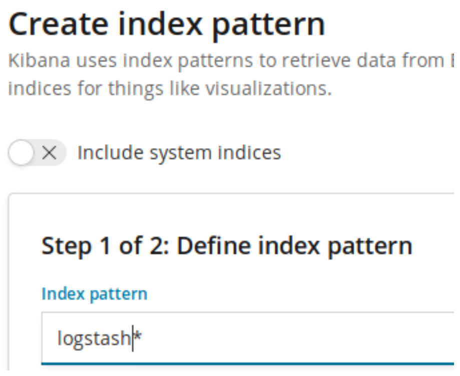
__2. Under **Kibana**, click **Index Patterns**.



__3. Click **Create Index Pattern** button.



__4. In **Index pattern** field, type in **logstash***



Create index pattern

Kibana uses index patterns to retrieve data from I
indices for things like visualizations.

Step 1 of 2: Define index pattern

Index pattern

logstash*

__5. Click **Next step** button.

__6. In **Time Filter filed name** drop-down, select **@timestamp**

__7. Click **Create index pattern** button.

Note: It might show you a spinning wheel. Wait for about 5 seconds and proceed to the next part.

Part 8 - Deploy a Sample App and View Log using kubectl

In this part, you will build a Docker image with a custom node application that logs a message each second. You will deploy the image to Kubernetes cluster and view logs by using a basic technique.

__1. Back in the Terminal, view the source code of a sample node application:

```
gedit sample-app/index.js
```

Notice the code uses a logging package to log messages.

__2. Close the editor.

__3. Build a Docker image that contains your custom node application:

```
docker build -t fluentd-node-sample:latest -f sample-app/Dockerfile
sample-app
```

```
[Enter this command in one line]
```

__4. Deploy the image to Kubernetes:

```
kubectl create -f kubernetes/node-deployment.yaml
```

__5. Get pod list and make a note of the pod name:

```
kubectl get pods
```

__6. View logs using kubectl:

```
kubectl logs <pod_name>
```

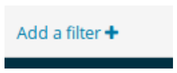
Note: Use the pod name which you noted down in the previous step of this lab.

__7. Verify the log shows the message log test.

Part 9 - View Logs using Kibana

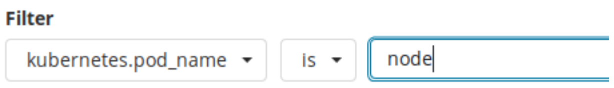
__1. Back in the browser, click **Discover** on the left-hand side navigation.

__2. Click **Add a filter** hyperlink.



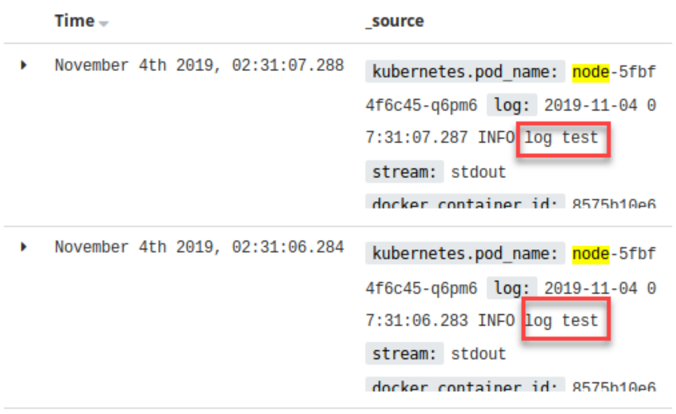
__3. Add a filter with the following values:

```
Filter: kubernetes.pod_name
Operator: is
Value: node
```



__4. Click **Save** button.

__5. Ensure “log test” shows up in the logs.


```
Time          _source
November 4th 2019, 02:31:07.288   kubernetes.pod_name: node-5fbf
                                  4f6c45-q6pm6 log: 2019-11-04 0
                                  7:31:07.287 INFO log test
                                  stream: stdout
                                  docker container id: 8575b10e6
November 4th 2019, 02:31:06.284   kubernetes.pod_name: node-5fbf
                                  4f6c45-q6pm6 log: 2019-11-04 0
                                  7:31:06.283 INFO log test
                                  stream: stdout
                                  docker container id: 8575b10e6
```

Part 10 - Clean-Up

__1. Stop Kubernetes:

```
minikube stop
```

__2. Delete the Kubernetes cluster:

```
minikube delete
```

__3. Close the terminal window.

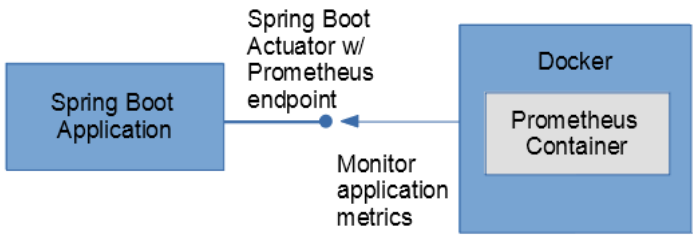
__4. Close the browser.

Part 11 - Review

In this lab, you explored logging using kubectl, fluentd, elasticsearch, and kibana.

Lab 3 - Using Prometheus for Monitoring

In this lab, you will use Prometheus to monitor a Spring Boot application. You will configure the Spring Boot application to enable metrics gathering via Spring Boot Actuators and expose Prometheus endpoint. Then, you will run Prometheus in a Docker container and configure it to monitor your Spring Boot application.


Spring Boot Application
Spring Boot Actuator w/ Prometheus endpoint
Monitor application metrics
Docker
Prometheus Container

Part 1 - Setting the Stage

In this part, you will create a directory and copy an existing project to it.

__1. Open a new Terminal window.

__2. Create a directory where you will copy the Spring Boot application:

```
mkdir -p ~/LabWorks
```

__3. Switch to the directory:

```
cd ~/LabWorks
```

__4. Unzip the existing application:

```
unzip ~/LabFiles/employees-rest-service
```

Note: If it gives you permission denied error, add sudo in front of the command.

__5. Switch to the project directory:

```
cd employees-rest-service
```

__6. Open the build.gradle file:

```
gedit build.gradle
```

__7. Verify you see this version:

```
dependencies {
    compile group: 'org.springframework.boot', name: 'spring-boot-
starter-web', version:'2.1.8.RELEASE'
    …
}
```

__8. Close the editor.

Part 2 - Explore an Existing Spring Boot Application

In this part, you will explore an existing Spring Boot application.

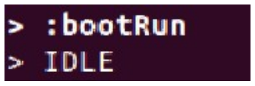
__1. Verify Gradle version:

```
gradle --version
```

__2. Build and run the sample project:

```
gradle clean build bootRun
```

Wait until the application is started.



```
> :bootRun
> IDLE
```

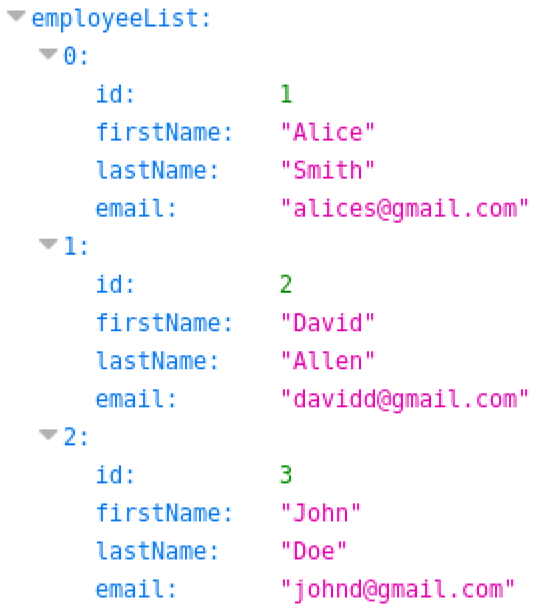
__3. Open Firefox web browser.

__4. Navigate to the following URL:

```
http://localhost:8080/employees/
```

Note: Don't forget to type in the / at the end.

The URL accesses Customers services implemented in the sample application. The service returns data in JSON format and it looks like this:



```
▼ employeeList:
  ▼ 0:
      id:          1
      firstName:   "Alice"
      lastName:    "Smith"
      email:       "alices@gmail.com"
  ▼ 1:
      id:          2
      firstName:   "David"
      lastName:    "Allen"
      email:       "davidd@gmail.com"
  ▼ 2:
      id:          3
      firstName:   "John"
      lastName:    "Doe"
      email:       "johnd@gmail.com"
```

__5. In the terminal where Spring Boot application is running, press **Ctrl+C** to stop the application.

__6. Using gedit or vi/nano text editors, examine the code in the following files:

```
src/main/java/com/webage/rest/model/Employee.java
src/main/java/com/webage/rest/model/Employees.java
src/main/java/com/webage/rest/dao/EmployeeDAO.java
src/main/java/com/webage/rest/controller/EmployeeController.java
```

Note:

model folder contains the models (Employee and Employees).

dao folder contains Data Access Object which populates the data.

controller folder contains the REST service(s).

Part 3 - Add Prometheus Dependencies to the Project

In this part, you will modify build.gradle and add Prometheus dependencies to the project.

__1. Open build.gradle:

```
gedit build.gradle
```

__2. Inside the dependencies block near the bottom, add **Actuator** dependency after the existing dependencies: (Note: It’s a single statement)

```
compile group: 'org.springframework.boot', name: 'spring-boot-starter-
actuator', version: '2.1.8.RELEASE'
```

To enable Spring Boot Actuator, you add the spring-boot-actuator dependency. Actuator dependency enables various features, such as monitoring app, gathering metrics, understanding traffic or the state of database.

__3. Inside the dependencies block near the bottom, add **Prometheus** dependency after the existing dependencies: (Note: It’s a single statement)

```
compile group: 'io.micrometer', name: 'micrometer-registry-prometheus',
version:'1.3.2'
```

To integrate actuator with Prometheus, you need to add the micrometer-registry-prometheus dependency. Spring Boot uses Micrometer, an application metrics facade to integrate actuator metrics with external monitoring systems. It supports several monitoring systems like Netflix Atlas, AWS Cloudwatch, Datadog, InfluxData, SignalFx, Graphite, Wavefront, Prometheus etc.

__4. Save and close build.gradle.

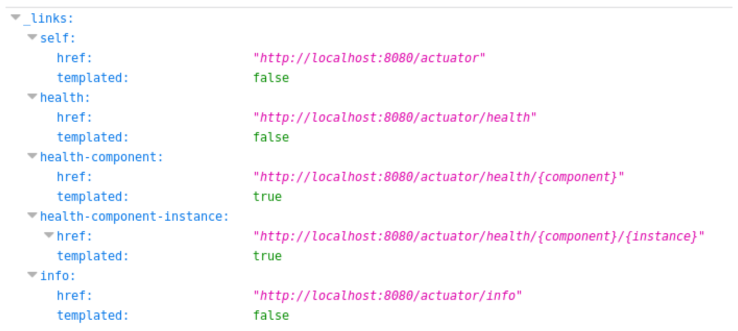
__5. In the terminal, run the following command to start up your application:

```
gradle clean build bootRun
```

__6. In the web browser window, type the following URL:

```
http://localhost:8080/actuator
```

Notice the default endpoints exposed over the HTTP protocol look like this:



```
_links:
  self:
    href:           "http://localhost:8080/actuator"
    templated:      false
  health:
    href:           "http://localhost:8080/actuator/health"
    templated:      false
  health-component:
    href:           "http://localhost:8080/actuator/health/{component}"
    templated:      true
  health-component-instance:
    href:           "http://localhost:8080/actuator/health/{component}/{instance}"
    templated:      true
  info:
    href:           "http://localhost:8080/actuator/info"
    templated:      false
```

Prometheus endpoints are not exposed by default.

__7. In the terminal, press **Ctrl+C** to stop your application.

Part 4 - Modify Spring Boot Application Properties

In this part, you will modify Spring Boot application properties to enable monitoring of various actuators.

__1. Create a directory where you will store application properties:

```
mkdir src/main/resources
```

__2. Create application properties file:

```
gedit src/main/resources/application.properties
```

__3. Enter the following content in the editor:

```
management.security.enabled=false
management.metrics.export.prometheus.enabled: true

management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=

management.endpoints.jmx.exposure.include=*
management.endpoints.jmx.exposure.exclude=
```

These lines enable Prometheus metrics that you can view over HTTP and also JMX.

__4. Save the file and exit gedit.

__5. Rebuild and run your application:

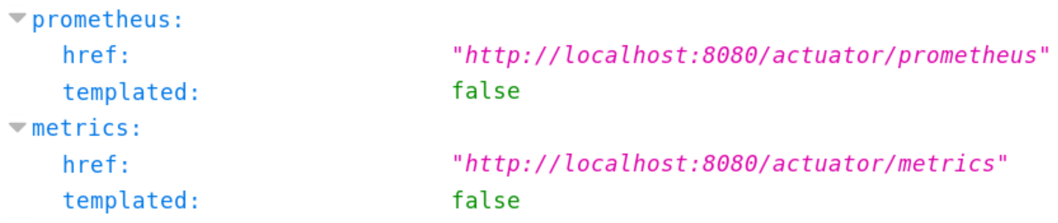
```
gradle clean build bootRun
```

__6. In the web browser, enter the following URL or refresh the browser:

```
http://localhost:8080/actuator
```

Notice there's a huge list of endpoints available.

__7. Browse through the list and notice Prometheus is also available.

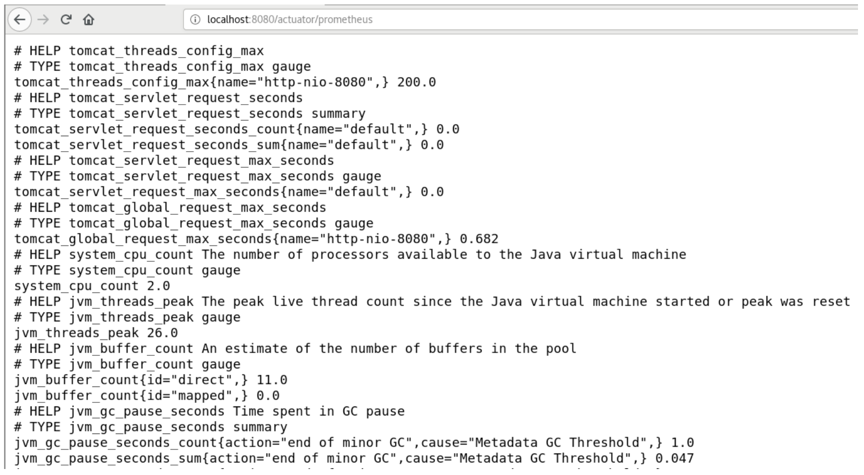


```
▼ prometheus:
    href:                  "http://localhost:8080/actuator/prometheus"
    templated:             false
▼ metrics:
    href:                  "http://localhost:8080/actuator/metrics"
    templated:             false
```

__8. In the web browser, enter the following URL:

```
http://localhost:8080/actuator/prometheus
```

Notice there are various metrics available which can be monitored. You will do so later in this lab.



localhost:8080/actuator/prometheus

```
# HELP tomcat_threads_config_max
# TYPE tomcat_threads_config_max gauge
tomcat_threads_config_max{name="http-nio-8080",} 200.0
# HELP tomcat_servlet_request_seconds
# TYPE tomcat_servlet_request_seconds summary
tomcat_servlet_request_seconds_count{name="default",} 0.0
tomcat_servlet_request_seconds_sum{name="default",} 0.0
# HELP tomcat_servlet_request_max_seconds
# TYPE tomcat_servlet_request_max_seconds gauge
tomcat_servlet_request_max_seconds{name="default",} 0.0
# HELP tomcat_global_request_max_seconds
# TYPE tomcat_global_request_max_seconds gauge
tomcat_global_request_max_seconds{name="http-nio-8080",} 0.682
# HELP system_cpu_count The number of processors available to the Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count 2.0
# HELP jvm_threads_peak The peak live thread count since the Java virtual machine started or peak was reset
# TYPE jvm_threads_peak gauge
jvm_threads_peak 26.0
# HELP jvm_buffer_count An estimate of the number of buffers in the pool
# TYPE jvm_buffer_count gauge
jvm_buffer_count{id="direct",} 11.0
jvm_buffer_count{id="mapped",} 0.0
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count{action="end of minor GC",cause="Metadata GC Threshold",} 1.0
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="Metadata GC Threshold",} 0.047
```

Part 5 - Create a Prometheus Configuration File

In this part, you will configure Prometheus to scrape metrics data from Spring Boot Actuator's /prometheus endpoint.

__1. Open a new Terminal window.

__2. Before you create Prometheus configuration file, option the host IP address. Run the following command and make a note of your host's IP address:

```
ifconfig | grep netmask | grep broadcast
```

```
(Note: It's the value next to inet. Do NOT use the broadcast IP
address)
```

__3. Create a folder to save Prometheus configure file:

```
mkdir ~/LabWorks/prometheus
```

__4. Use gedit to create a file named prometheus.yml:

```
gedit ~/LabWorks/prometheus/prometheus.yml
```

__5. Enter the following contents to the file:

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
    - targets: ['127.0.0.1:9090']

  - job_name: 'spring-actuator'
    metrics_path: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
    - targets: ['HOST_IP:8080']
```

```
[Don't forget to substitute the HOST_IP from an above step. Also,
ensure you indent the code properly]
```

Note: These lines specify the following:

* Prometheus will run on port 9090

* It's going to allow you to monitor Spring Boot actuator at /actuator/prometheus

* The application you want to run is located at HOST_IP:8080

__6. Save the file and exit gedit.

Part 6 - Run Prometheus and Monitor your Application

In this part, you will run Prometheus. Prometheus is available in one of the docker images, which is preloaded on your machines.

__1. Run the following command to start up Prometheus in a container:

```
sudo docker run -d --name=prometheus -p 9090:9090 -v
~/LabWorks/prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
prom/prometheus --config.file=/etc/prometheus/prometheus.yml
```

```
[Enter this command in one line]
```

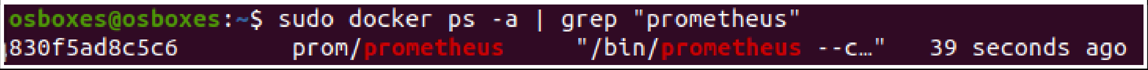
__2. If you are prompted for a password, enter **osboxes.org**

Wait until the process is completed.

__3. Ensure there are no errors in the terminal, wait for a few seconds, and run the following command to verify Prometheus container has started:

```
sudo docker ps -a | grep "prometheus"
```

It should show you Prometheus container like this:

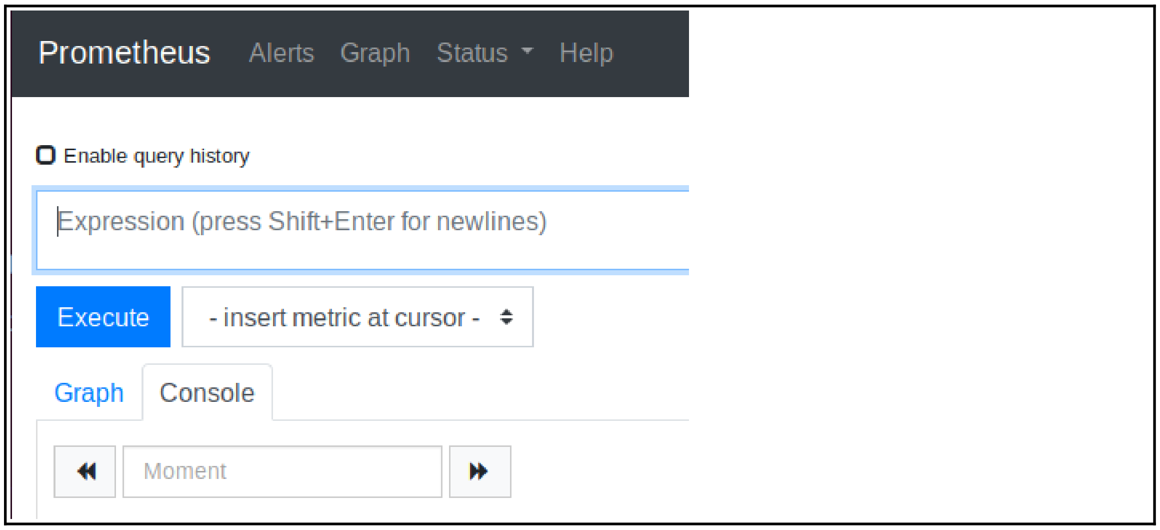


```
osboxes@osboxes:~$ sudo docker ps -a | grep "prometheus"
830f5ad8c5c6          prom/prometheus      "/bin/prometheus --c…"   39 seconds ago
```

__4. In the web browser window, open a new tab, and enter the following URL:

```
http://localhost:9090
```

It should look like this:


Prometheus
Alerts
Graph
Status
Help
Enable query history
Expression (press Shift+Enter for newlines)
Execute
- insert metric at cursor -
Graph
Console
Moment

__5. In Expression textbox, enter the following:

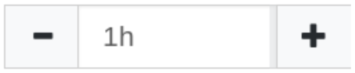
```
process_uptime_seconds
```

Note: It's one of the metrics which is exposed over /actuator/prometheus endpoint. It shows you how long the application has been running. You can navigate to **http://localhost:8080/actuator/prometheus** and search for it. There are also several other metrics which can be monitored.

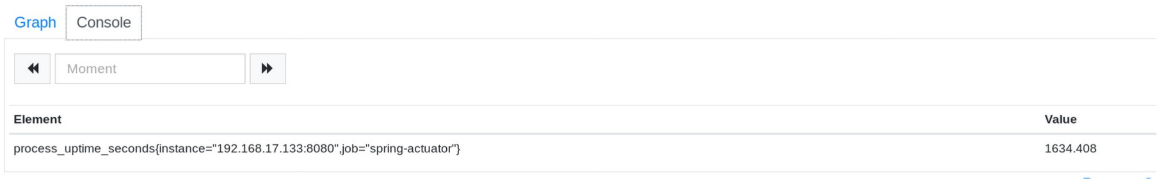
__6. Press **Execute** button.

__7. Click the **Graph** tab.

It can take up to a minute to show the updated result. Refresh the page periodically until you see the graph. Notice a graph shows up under Graph tab. You can customize the time-period by pressing + and – buttons.



__8. Click **Console** tab to see the text version of the metrics.



Element	Value
process_uptime_seconds{instance="192.168.17.133:8080",job="spring-actuator"}	1634.408

__9. Switch back to Graph tab.

__10. In Expression text box, enter the following, and press Execute:

```
system_cpu_usage
```

__11. It will show you CPU usage.

__12. Open a tab in the web browser, and enter the URL:

```
http://localhost:8080/employees/
```

__13. Refresh the page a few times.

__14. In Expression text box, enter the following, and press Execute:

```
http_server_requests_seconds_max{method="GET",status="200",uri="/
employees/"}
```

```
[Enter this command in one line]
```

It will show REST service response latency. It can be helpful to find slow APIs/REST services.

Part 7 - Clean-Up

In this part, you will stop Prometheus and your application.

__1. Run the following command to stop Prometheus container:

```
sudo docker stop prometheus
```

If prompted for a password, enter osboxes.org

__2. Remove Prometheus container:

```
sudo docker rm prometheus
```

If prompted for a password, enter osboxes.org

__3. Switch to the terminal where Spring Boot application is running and press Ctrl+C to stop the application.

__4. Close all Terminal windows.

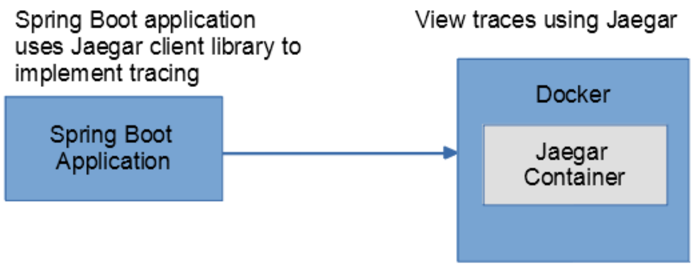
__5. Close web browser tabs where Prometheus is running.

Part 8 - Review

In this lab, you used Prometheus to monitor a Spring Boot application.

Lab 4 - Using Jaeger for Tracing

In this lab, you will implement instrumentation using Jaeger. You will modify an existing Spring Boot application to add tracing to it.


Spring Boot application uses Jaegar client library to implement tracing
View traces using Jaegar
Spring Boot Application
Docker
Jaegar Container

Part 1 - Setting the Stage

NOTE: If you have already completed the Prometheus lab, you can skip Part 1 & 2 and move on to Part 3 of this lab.

In this part, you will create a directory and copy an existing project to it.

__1. Open a new Terminal window.

__2. Create a directory where you will copy the Spring Boot application:

```
mkdir -p ~/LabWorks
```

__3. Switch to the directory:

```
cd ~/LabWorks
```

__4. Unzip the existing application:

```
unzip ~/LabFiles/employees-rest-service
```

Note: If it gives you permission denied error, add sudo in front of the command.

__5. Switch to the project directory:

```
cd employees-rest-service
```

Part 2 - Explore an Existing Spring Boot Application

If you have already completed the Prometheus lab, you can skip Part 1 & 2 and move on to Part 3 of this lab.

In this part, you will explore an existing Spring Boot application.

__1. Verify Gradle version:

```
gradle --version
```

__2. Build and run the sample project:

```
gradle clean build bootRun
```

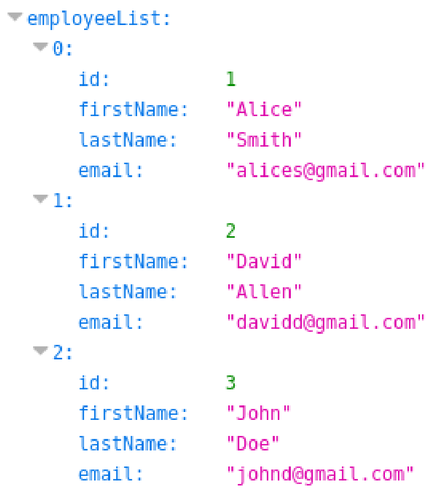
__3. From **Application** menu, open Firefox web browser.

__4. Navigate to the following URL:

```
http://localhost:8080/employees/
```

Note: Don't forget to type in the / at the end.

The URL accesses Customers services implemented in the sample application. The service returns data in JSON format and it looks like this:



```
employeeList:
  0:
    id:          1
    firstName:   "Alice"
    lastName:    "Smith"
    email:       "alices@gmail.com"
  1:
    id:          2
    firstName:   "David"
    lastName:    "Allen"
    email:       "davidd@gmail.com"
  2:
    id:          3
    firstName:   "John"
    lastName:    "Doe"
    email:       "johnd@gmail.com"
```

__5. In the terminal where Spring Boot application is running, press Ctrl+C to stop the application.

__6. Using gedit or vi/nano text editors, examine the code in the following files:

```
src/main/java/com/webage/rest/model/Employee.java
src/main/java/com/webage/rest/model/Employees.java
src/main/java/com/webage/rest/dao/EmployeeDAO.java
src/main/java/com/webage/rest/conroller/EmployeeController.java
```

Note:

model folder contains the models (Employee and Employees)

dao folder contains Data Access Object which populates the data.

controller folder contains the REST service(s)

Part 3 - Add Jaeger Dependency to the Project and Configure It

In this part, you will modify build.gradle and add Jaeger dependency to the project and configure a bean in the main application file which returns a tracing instance.

__1. Open a Terminal window and switch to the project folder:

```
cd ~/LabWorks/employees-rest-service
```

__2. Open the project's main application file:

```
gedit src/main/java/com/webage/rest/SpringBootDemoApplication.java
```

__3. After the existing import statements, add the following imports:

```
import io.jaegertracing.Configuration;
import io.jaegertracing.internal.JaegerTracer;
import org.springframework.context.annotation.Bean;
```

__4. Inside SpringBootDemoApplication class, after the main function, add the following code:

```
@Bean
public static JaegerTracer getTracer() {
   Configuration.SamplerConfiguration samplerConfig =
Configuration.SamplerConfiguration.fromEnv().withType("const").withPara
m(1);
   Configuration.ReporterConfiguration reporterConfig =
Configuration.ReporterConfiguration.fromEnv().withLogSpans(true);
   Configuration config = new Configuration("jaeger
tutorial").withSampler(samplerConfig).withReporter(reporterConfig);
   return config.getTracer();
}
```

Note: These lines configure a bean which returns a JaegerTracing instance which will be used for tracing purpose. Also note, your application will traces will show up as “Jaeger tutorial” in Jaeger. You can customize this name to whatever application name you want.

__5. Save the file and exit gedit.

__6. Open build.gradle:

```
gedit build.gradle
```

__7. In the dependencies tag, add a new dependency:

```
compile group: 'io.jaegertracing', name: 'jaeger-client', version:
'0.35.5'
```

```
[Enter this command in one line]
```

__8. Save and close the file.

__9. Run the following command to build your application:

```
gradle clean build
```

__10. Ensure there are no errors. If there any, resolve them before proceeding to the next part of this lab.

Part 4 - Modify the Existing REST Service

In this part, you will modify the existing REST service/controller and write code to trace messages.

__1. Open the REST service/controller file:

```
gedit src/main/java/com/webage/rest/controller/EmployeeController.java
```

__2. After the existing import statements, add the following imports:

```
import io.opentracing.Span;
import io.opentracing.Tracer;
```

__3. Inside EmployeeController class, add the following as the first statement:

```
@Autowired
private Tracer tracer;
```

__4. Inside **getEmployees** method, delete the existing return statement, and add the following code:

```
Span span = tracer.buildSpan("get employees").start();
span.setTag("http.status_code", 201);
Employees data = employeeDao.getAllEmployees();
span.finish();
return data;
```

Note: Your getEmployees method should look like this:

```
public Employees getEmployees()
{
    Span span = tracer.buildSpan("get employees").start();
    span.setTag("http.status_code", 201);
    Employees data = employeeDao.getAllEmployees();
    span.finish();
     return data;
}
```

Note: These lines use the JaegerTracing object, which you configured in the main application file, to obtain a Span object. Span allows you to start the trace, optionally add some additional data in the form tags, and stop the trace.

__5. Save the file and close gedit.

__6. Run the following command to build your application:

```
gradle clean build
```

__7. Ensure there are no errors. If there any, resolve them before proceeding to the next part of this lab.

Part 5 - Verify Tracing in Jaeger

In this part, you will run your application, start Jaeger, and verify tracing is working properly.

__1. Run the following command to start up your Spring Boot application:

```
gradle bootRun
```

__2. Open a web browser and navigate to the following URL:

```
http://localhost:8080/employees/
```

Note: It’s the same data you saw previously. Next, you will start Jaeger and view the traces.

__3. Open a new Terminal window.

__4. Run the following command to run Jaeger in a Docker container:

```
sudo docker run -d --name Jaeger --rm -it --network=host jaegertracing/
all-in-one:1.12.0


[Enter this command in one line]
```

__5. Enter **osboxes.org** as password if prompt.

__6. Ensure container is up and running:

```
sudo docker ps -a | grep "Jaeger"
```

__7. Enter **osboxes.org** as password, if prompted.

__8. Refresh the following page:

```
http://localhost:8080/employees/
```

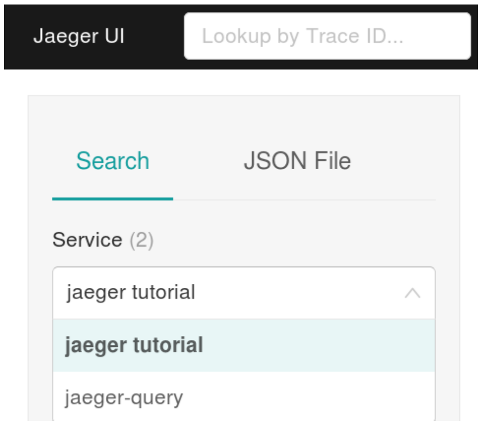
__9. In the web browser window, open a new tab, and navigate to the following URL:

```
http://localhost:16686
```

__10. Refresh the page in the tab where **http://localhost:8080/employees/** page is open.

__11. On the **Jaeger** web console, refresh the page.

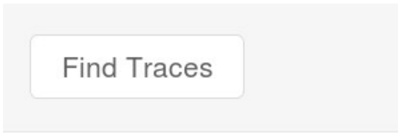
__12. Click **Service** drop-down and select **Jaeger tutorial**:



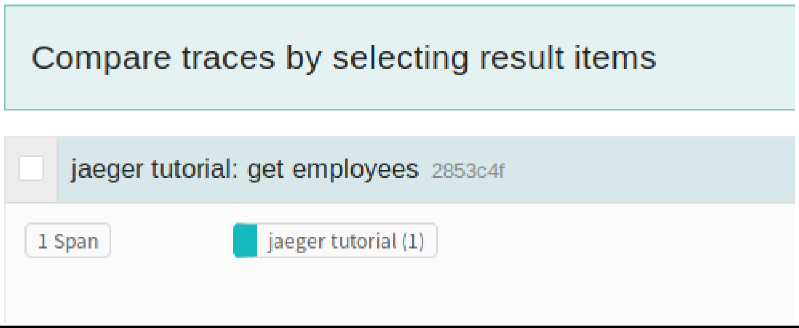
Note: If Service drop-down is blank, refresh the page in the tab where **http://localhost:8080/employees/** page is open, and then refresh Jaeger web console. You might have to wait for a minute before it shows up in the drop down.

Jaeger tutorial is your custom application. Jaeger query is available OOB as part of Jaeger. It allows you to make REST API calls to query Jaeger in case if you don't want to use the web console.

__13. Scroll down and click **Find Traces** button:

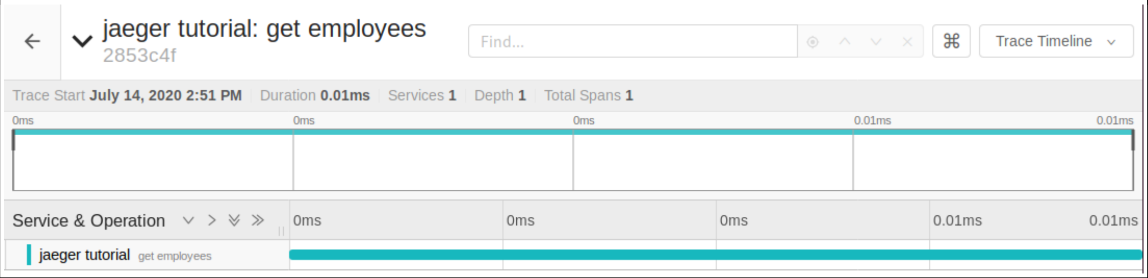


Notice the trace looks like this:


Compare traces by selecting result items
jaeger tutorial: get employees 2853c4f
1 Span
jaeger tutorial (1)

__14. Click the trace '**Jaeger tutorial**' to view details.

It should show up like this:



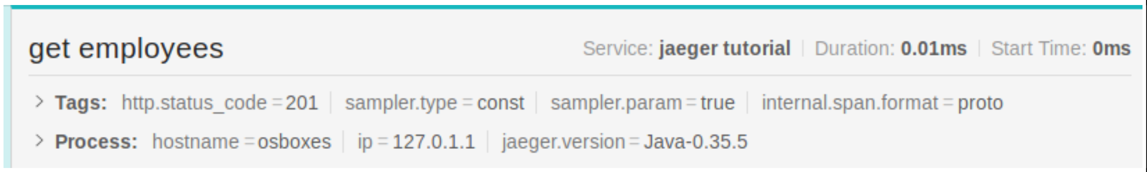
jaeger tutorial: get employees 2853c4f

Trace Start **July 14, 2020 2:51 PM** | Duration **0.01ms** | Services **1** | Depth **1** | Total Spans **1**

Service & Operation — jaeger tutorial get employees

__15. In **Service & Operation** section, click **Jaeger tutorial** again.

Notice it looks like this:



get employees

Service: **jaeger tutorial** | Duration: **0.01ms** | Start Time: **0ms**

Tags: http.status_code = 201 | sampler.type = const | sampler.param = true | internal.span.format = proto

Process: hostname = osboxes | ip = 127.0.1.1 | jaeger.version = Java-0.35.5

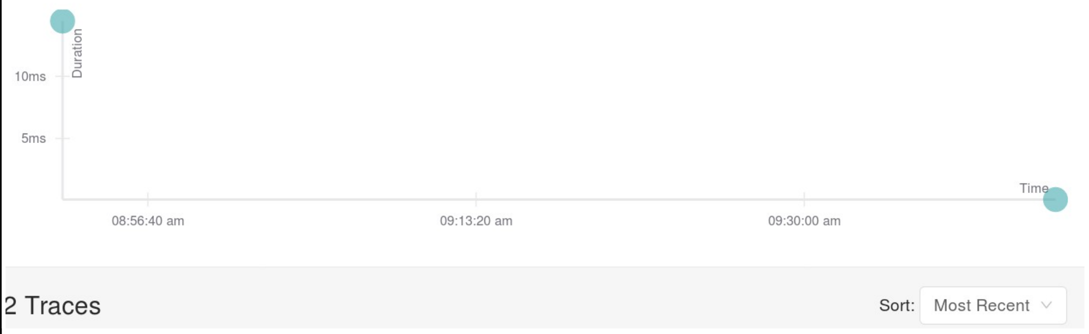
Notice Tags is showing your custom label http.status_code with value 201.

__16. Refresh the **http://localhost:8080/employees/** page.

__17. Go back to the Jaeger web console main page.

__18. Click **Find Traces**.

Notice the traces show up like this:



2 Traces

Note. Your values in the graph may be different.

Part 6 - Clean-Up

In this part, you will stop Prometheus and your application.

__1. Run the following command to stop Jaeger container:

```
sudo docker stop Jaeger
```

If prompted for a password, enter osboxes.org

__2. Switch to the terminal where Spring Boot application is running and press Ctrl+C to stop the application.

__3. Close the Terminal windows.

__4. Close web browser tabs where Jaeger is running.

Part 7 - Review

In this lab, you implemented instrumentation using Jaeger.

Lab 5 - Configure Tools in Jenkins

In this lab you will verify that Jenkins Continuous Integration is already installed and you will configure it.

<u>**At the end of this lab you will be able to:**</u>

1. Verify Jenkins is running
2. Configure tools in Jenkins

Part 1 - Configure Jenkins

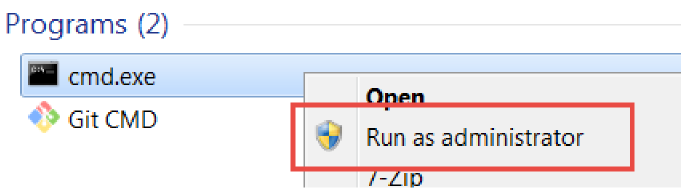
In the following Labs you will work in the 'VM_MSD_REL_1_0' image. You should find a shortcut on the desktop named 'VM_MSD_REL_1_0', double click on it to start this VM if it is not already running and make sure to close all other running VMs to free memory. Login with user wasadmin with password as wasadmin.

If you don't find this VM then the software was installed directly on the computer. For questions regarding the VM to use, please contact your instructor.

After the Jenkins installation, you can configure few other settings to complete the installation before creating jobs.

In this part, you will set JDK HOME and Gradle Installation directory.

__1. Open a command prompt window as an administrator.



__2. If jenkins was installed then change to Jenkins installation folder:

```
cd C:\Program Files (x86)\Jenkins
```

__3. Verify Jenkins is started:

```
jenkins status
```

__4. If it is not started then , enter this command:

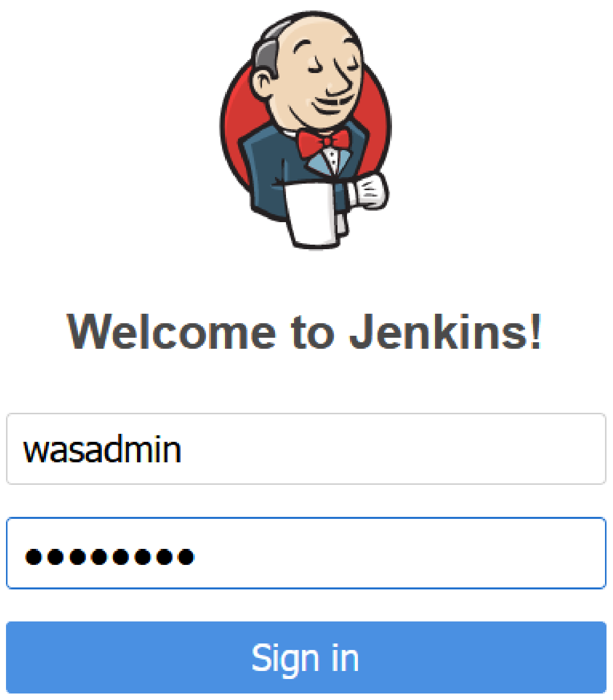
```
jenkins start
```

__5. Close the window.

__6. To connect to Jenkins, open Chrome and enter the following URL.

```
http://localhost:8080/
```

__7. Enter **wasadmin** as user and password and click **Sign in**.

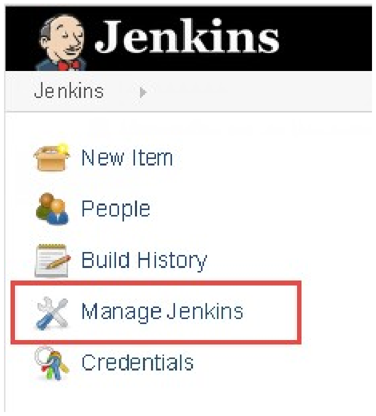

Welcome to Jenkins!

wasadmin

●●●●●●●●

__8. Don't save the password if prompt or select Never Remember password for this site.

__9. Click on the **Manage Jenkins** link.



Don't worry about any warning.

__10. Click **Global Tool Configuration**.

__11. Scroll down and find the JDK section, Click **Add JDK**.

__12. Enter **OracleJDK** for JDK name.

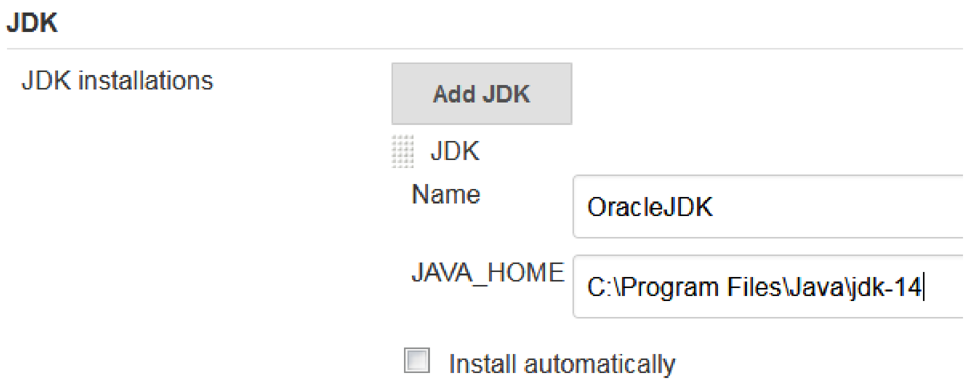
__13. Uncheck the 'Install automatically' option.

__14. Enter **JAVA_HOME** value as:

```
C:\Program Files\Java\jdk-14
```

Note. You may need to use another path if Java was installed in a different folder like **C:\ Program Files\Java\jdk-14**, contact your instructor or search for the right path and use it as JAVA_HOME.

__15. Verify your settings look as below:

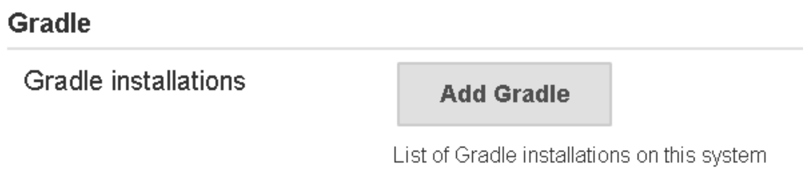


Name: OracleJDK

JAVA_HOME: C:\Program Files\Java\jdk-14

☐ Install automatically

__16. In the Gradle section, click **Add Gradle**.



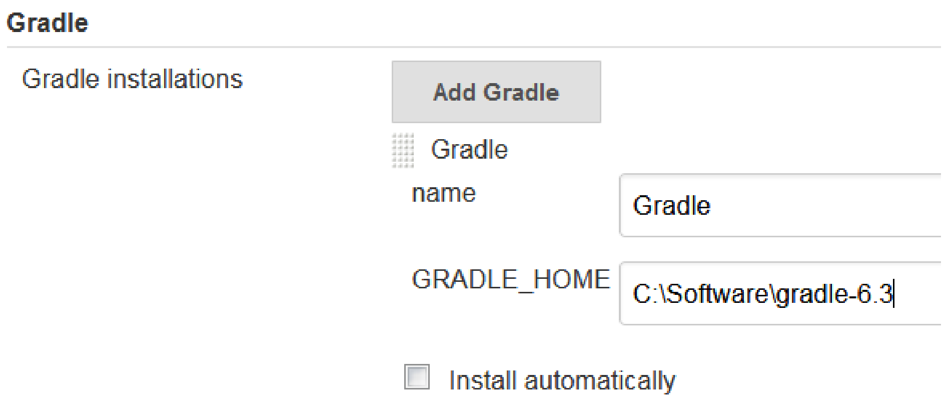
__17. Enter **Gradle** for Gradle name.

__18. Uncheck the 'Install automatically' option.

__19. Enter the following for GRADLE_HOME. Make sure this folder is correct:

```
C:\Software\gradle-6.3
```

__20. Verify your settings look as below:



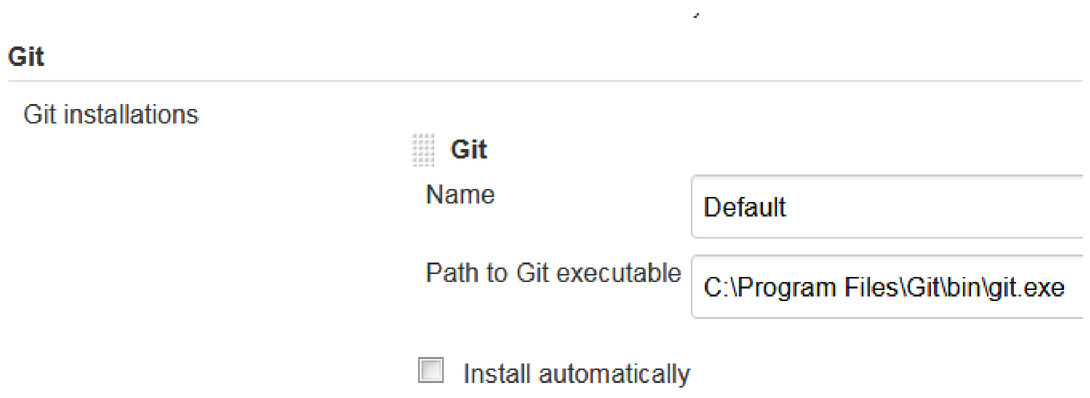
name: Gradle

GRADLE_HOME: C:\Software\gradle-6.3

☐ Install automatically

__21. Scroll and find **Git**, you may see an error regarding the git path. Enter the following path and then hit the tab key (Make sure the path is valid or find the right path):

```
C:\Program Files\Git\bin\git.exe
```



Git

Git installations

Git

Name: Default

Path to Git executable: C:\Program Files\Git\bin\git.exe

☐ Install automatically

__22. Click **Save**.

Part 2 - Review

In this lab you configured the Jenkins Continuous Integration Server.

Lab 6 - Create a Jenkins Job

In this lab you will create and build a job in Jenkins.

Jenkins freestyle projects allow you to configure just about any sort of build job, they are highly flexible and very configurable.

__At the end of this lab you will be able to:__

1. Create a Jenkins Job that accesses a Git repository.

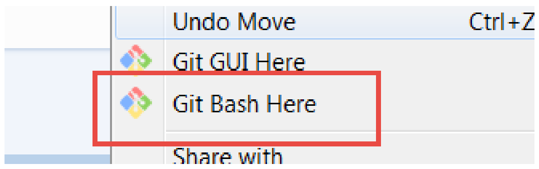
Part 1 - Create a Git Repository

As a distributed version control system, Git works by moving changes between different repositories. Any repository apart from the one you're currently working in is called a "remote" repository. Git doesn't differentiate between remote repositories that reside on different machines and remote repositories on the same machine. They're all remote repositories as far as Git is concerned. In this lab, we're going to start from a source tree, create a local repository in the source tree, and then clone it to a local repository. Then we'll create a Jenkins job that pulls the source files from that remote repository. Finally, we'll make some changes to the original files, commit them and push them to the repository, showing that Jenkins automatically picks up the changes.

__1. Using File Explorer, navigate to the folder:

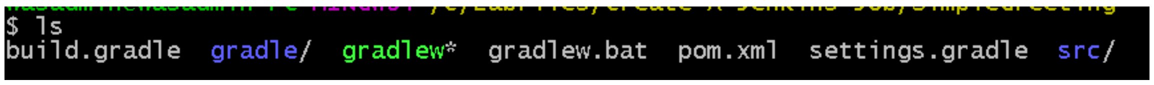
```
C:\LabFiles\Create A Jenkins Job_GRADLE\SimpleGreeting
```

__2. Right click in the empty area and select **Git Bash Here**. The Git command prompt will open.



__3. Enter the following command:

```
ls
```



```
$ ls
build.gradle   gradle/   gradlew*   gradlew.bat   pom.xml   settings.gradle   src/
```

__4. Enter the following lines. Press enter after each line:

```
git config --global user.email "wasadmin@webagesolutions.com"
git config --global user.name "Bob Smith"
```

The lines above are actually part of the initial configuration of Git. Because of Git's distributed nature, the user's identity is included with every commit as part of the commit data. So we have to tell Git who we are before we'll be able to commit any code.

__5. Enter the following lines to actually create the Git repository:

```
git init
git add .
git commit -m "Initial Commit"
```

The above lines create a git repository in the current directory (which will be **C:\LabFiles\Create A Jenkins Job_GRADLE\SimpleGreeting**), add all the files to the current commit set (or 'index' in git parlance), then actually performs the commit.

__6. Enter the following, to create a folder called **repos** under the C:\Software folder.

```
mkdir /c/Software/repos
```

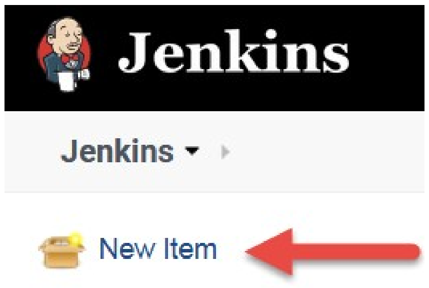
__7. Enter the following to clone the current Git repository into a new remote repository.

```
git clone --bar . /c/Software/repos/SimpleGreeting.git
```

At this point, we have a "remote" Git repository in the folder **C:\Software\repos\SimpleGreeting.git**. Jenkins will be quite happy to pull the source files for a job from this repo.

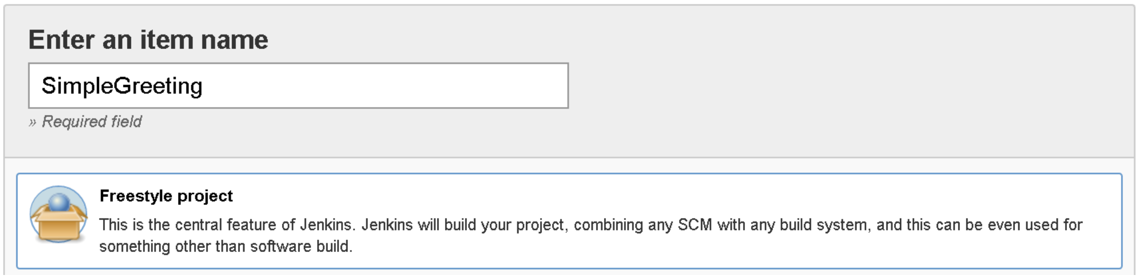
Part 2 - Create the Jenkins Job

__1. Go to the Jenkins home and click on the **New Item** link.



__2. Enter **SimpleGreeting** for the project name.

__3. Select **Freestyle project** as the project type.



Enter an item name

SimpleGreeting

» Required field

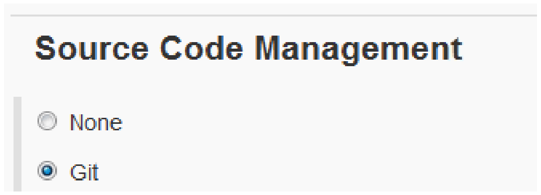
Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

__4. Click **OK**, to add a new job.

After the job is created, you will be on the job configuration page.

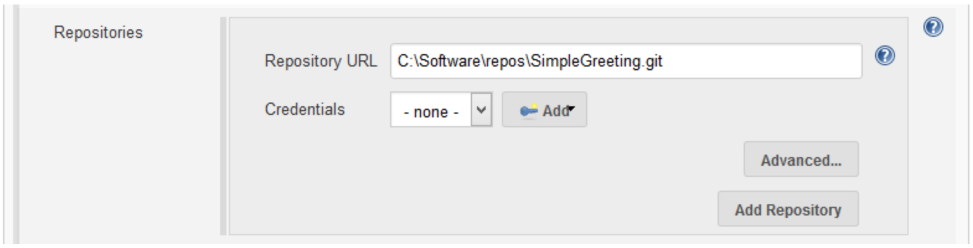
__5. Scroll down to the **Source Code Management** section and then select **Git**.



Source Code Management

None [✓ Git]

__6. Under Repositories, enter **C:\Software\repos\SimpleGreeting.git** and press tab key.

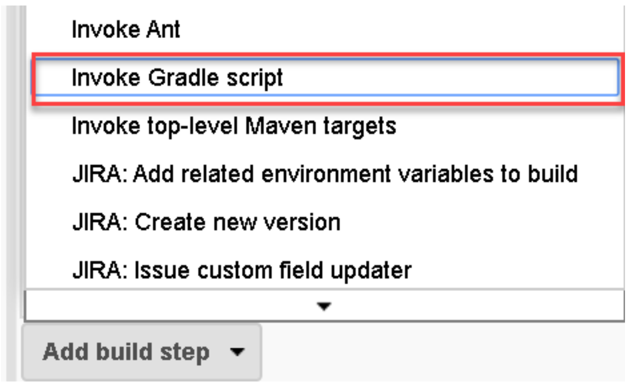


Repositories

Repository URL: C:\Software\repos\SimpleGreeting.git

Credentials: - none -

__7. In **Build** section, click **Add build step > Invoke Gradle script**

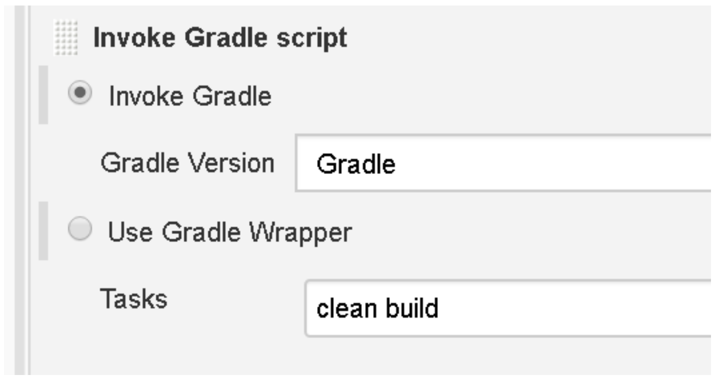

Invoke Ant
Invoke Gradle script
Invoke top-level Maven targets
JIRA: Add related environment variables to build
JIRA: Create new version
JIRA: Issue custom field updater
Add build step

__8. Ensure **Invoke Gradle** radio button is selected.

__9. In **Gradle Version**, select **Gradle**

__10. In **Tasks**, enter **clean build**

__11. The **Invoke Gradle script** configuration should look like this:



Invoke Gradle script

● Invoke Gradle

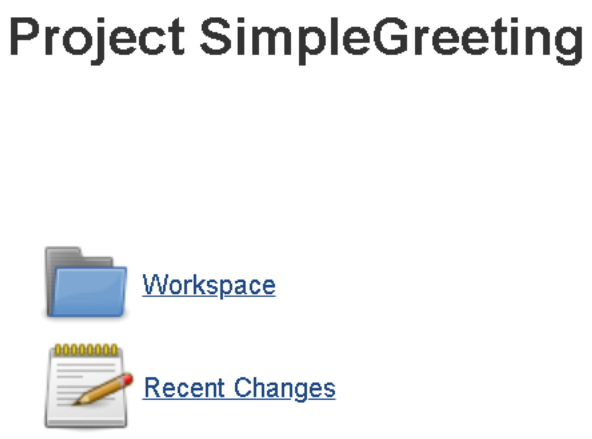
Gradle Version: Gradle

○ Use Gradle Wrapper

Tasks: clean build

__12. Click **Save**.

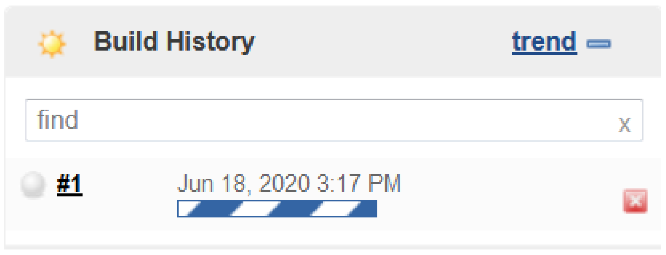
__13. You will see the Job screen. Click **Workspace**.



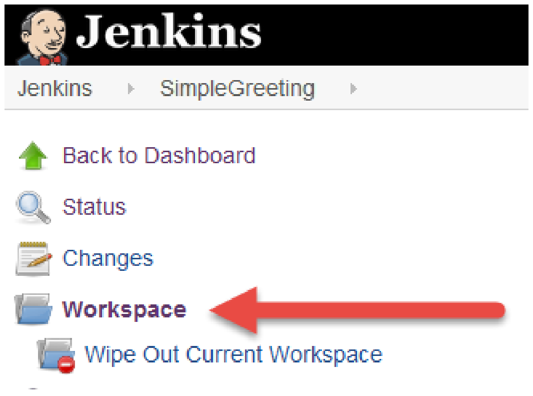
You may see *Error: no workspace*. It's OK for now.

__14. Click **Build Now**.

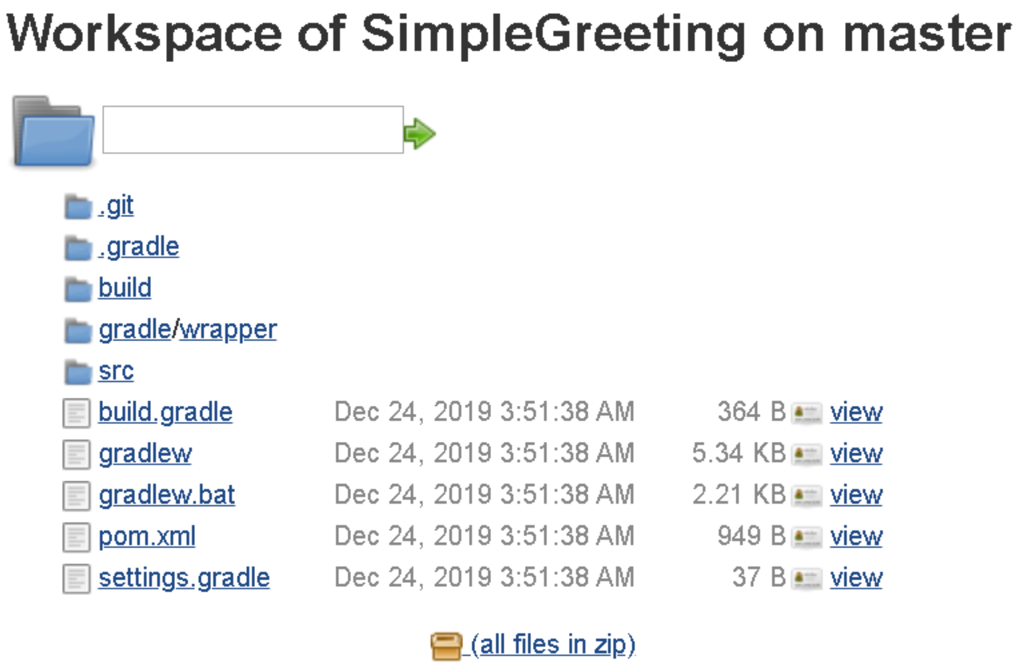
You should see the build in progress in the **Build History** area.



__15. After a few seconds the build will complete, the progress bar will stop. Click on **Workspace**.



You will see that the directory is populated with the source code for our project.



Workspace of SimpleGreeting on master

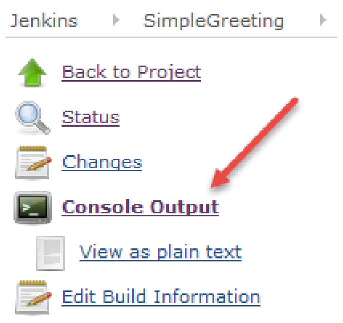
.git
.gradle
build
gradle/wrapper
src

build.gradle	Dec 24, 2019 3:51:38 AM	364 B	view
gradlew	Dec 24, 2019 3:51:38 AM	5.34 KB	view
gradlew.bat	Dec 24, 2019 3:51:38 AM	2.21 KB	view
pom.xml	Dec 24, 2019 3:51:38 AM	949 B	view
settings.gradle	Dec 24, 2019 3:51:38 AM	37 B	view

(all files in zip)

__16. Find the **Build History** box, and click on the 'time' value for the most recent build. You should see that the build was successful.

__17. Click the **Console Output** from the left menu.



__18. At the end of the console you will also see the build success and successful build finish.

```
BUILD SUCCESSFUL in 11s
5 actionable tasks: 5 executed
Build step 'Invoke Gradle script' changed build result to SUCCESS
Finished: SUCCESS
```

You have created a project and built it successfully.

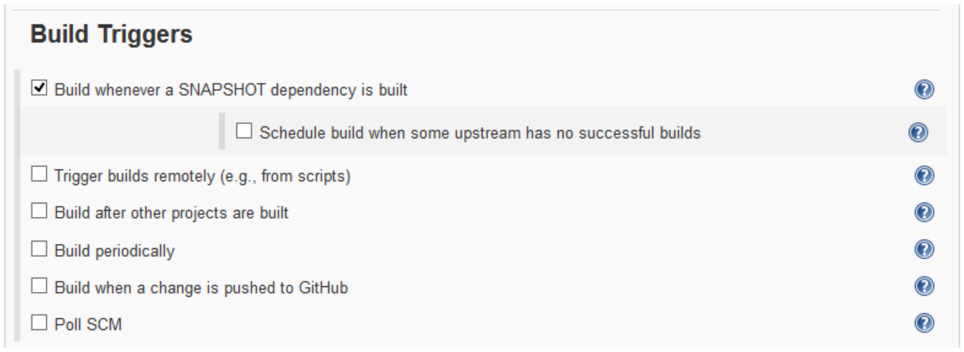
Part 3 - Enable Polling on the Repository

So far, we have created a Jenkins job that pulls a fresh copy of the source tree prior to building. But we triggered the build manually. In most cases, we would like to have the build triggered automatically whenever a developer makes changes to the source code in the version control system.

__1. In the Jenkins web application, navigate to the **SimpleGreeting** project. You can probably find the project in the breadcrumb trail near the top of the window. Alternately, go to the Jenkins home page and then click on the project.

__2. Click the **Configure** link.

__3. Scroll down to find the **Build Triggers** section.

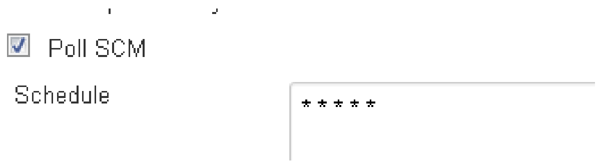


Build Triggers

- [✓] Build whenever a SNAPSHOT dependency is built
  - ☐ Schedule build when some upstream has no successful builds
- ☐ Trigger builds remotely (e.g., from scripts)
- ☐ Build after other projects are built
- ☐ Build periodically
- ☐ Build when a change is pushed to GitHub
- ☐ Poll SCM

__4. Click on the check box next to **Poll SCM**

__5. Enter '* * * * *' into the **Schedule** text box. [Make sure there is a space between each *]



[✓] Poll SCM

Schedule `* * * * *`

Note: The above schedule sets up a poll every minute. In a production scenario, that's a higher frequency than we need, and it can cause unnecessary load on the repository server and on the Jenkins server. You'll probably want to use a more reasonable schedule - perhaps every 15 minutes. That would be 'H/15 * * * *' in the schedule box.

__6. In **Post-build Actions** section, click **Add post-build action** and select **Publish JUnit test result report**.

This will allow you to graphically view unit test results.

__7. In **Test reports XMLs**, enter the following:

```
build\test-results\test\TEST-*.xml
```

Note: By default, Gradle stores test results in the folder mentioned above. In this case, the actual file name is TEST-com.simple.TestGreeting.xml but you can use * wildcard to specify the file name.

__8. Click **Save**.

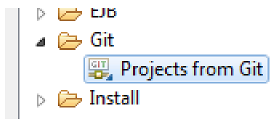
Part 4 - Import the Project into Eclipse

In order to make changes to the source code, we'll clone a copy of the Git repository into an Eclipse project.

__1. Start Eclipse by running **C:\Software\eclipse\eclipse.exe** and use **C:\Workspace** as Workspace.

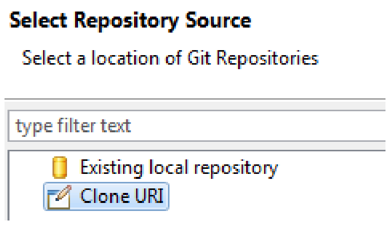
__2. From the main menu, select **File → Import…**

__3. Select **Git → Projects from Git**.



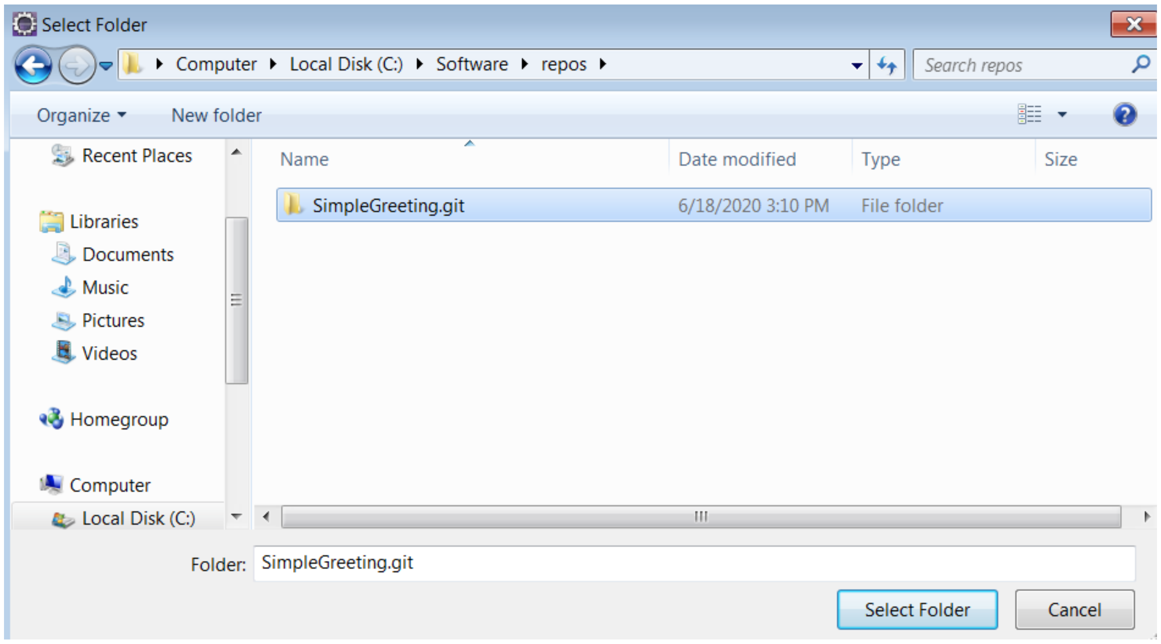
__4. Click **Next**.

__5. Select **Clone URI** and then click **Next**.


Select Repository Source
Select a location of Git Repositories
type filter text
Existing local repository
Clone URI

You might think that 'Existing local repository' would be the right choice, since we're cloning from a folder on the same machine. Eclipse, however, expects a "local repository" to be a working directory, not a bare repository. On the other hand, Jenkins will complain if we try to get source code from a repository with a working copy. So the correct thing is to have Jenkins pull from a bare repository, and use **Clone URI** to have Eclipse import the project from the bare repository.

__6. Click on **Local File...** and then navigate to **C:\Software\repos\SimpleGreeting.git**

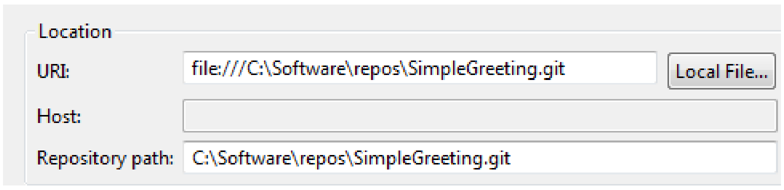


Name	Date modified	Type	Size
SimpleGreeting.git	6/18/2020 3:10 PM	File folder	

Folder: SimpleGreeting.git

__7. Click **Select Folder**.

__8. Back in the **Import Projects** dialog, click **Next**.

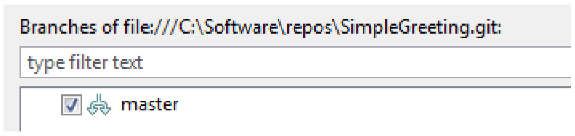

Location

URI: file:///C:\Software\repos\SimpleGreeting.git

Host:

Repository path: C:\Software\repos\SimpleGreeting.git

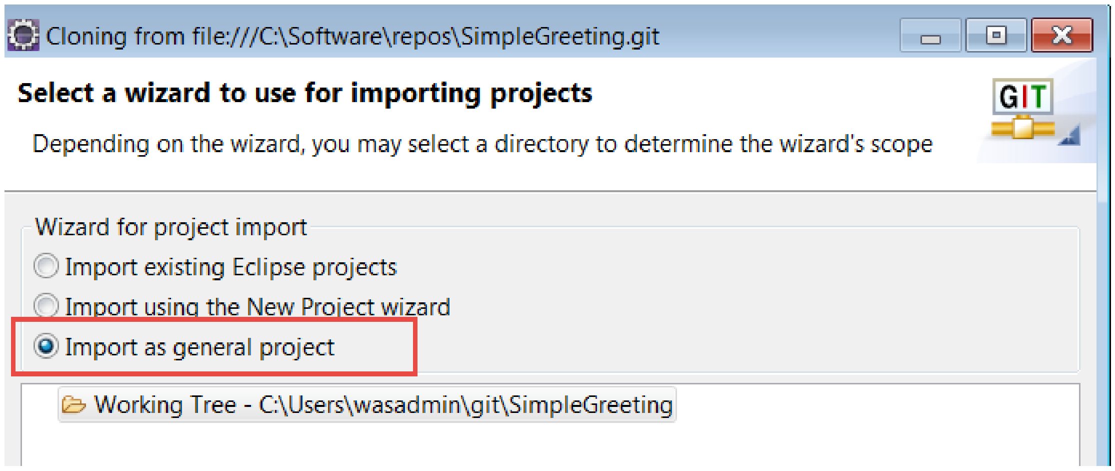
__9. Click **Next** to accept the default 'master' branch.


Branches of file:///C:\Software\repos\SimpleGreeting.git:

[✓] master

__10. In the **Local Destination** pane, leave the defaults and click **Next**.

__11. Select **Import as a General Project** and click **Next**.



Cloning from file:///C:\Software\repos\SimpleGreeting.git

Select a wizard to use for importing projects

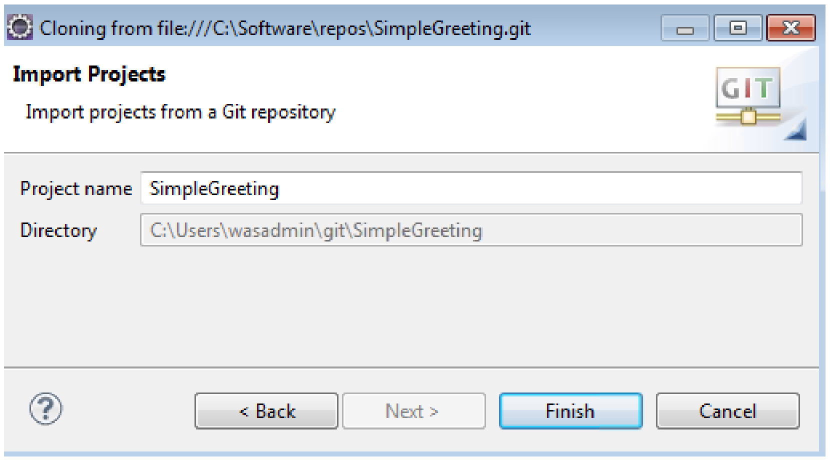
Depending on the wizard, you may select a directory to determine the wizard's scope

Wizard for project import

- ○ Import existing Eclipse projects
- ○ Import using the New Project wizard
- ● Import as general project

Working Tree - C:\Users\wasadmin\git\SimpleGreeting

__12. Click **Finish**.



Cloning from file:///C:\Software\repos\SimpleGreeting.git

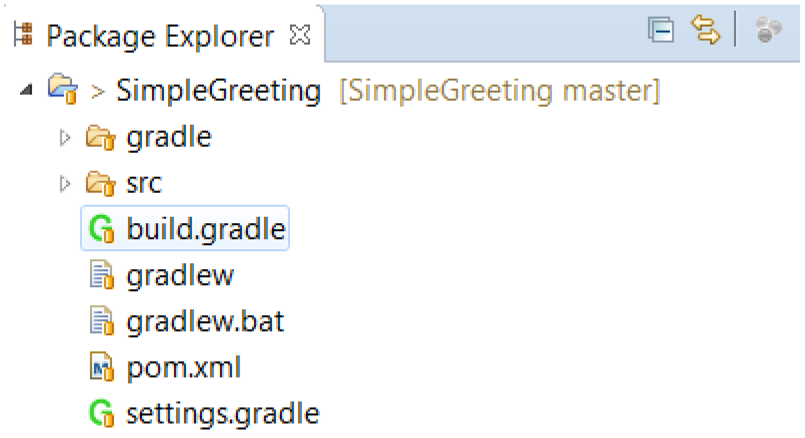
Import Projects

Import projects from a Git repository

Project name: SimpleGreeting

Directory: C:\Users\wasadmin\git\SimpleGreeting

__13. You should see the new project in the **Project Explorer**, expand it.



In real-world, after this step, we should convert our project in Eclipse so it understands Gradle project layout. However, it's not required to understand Jenkins continuous integration so we will leave the project layout as it is.

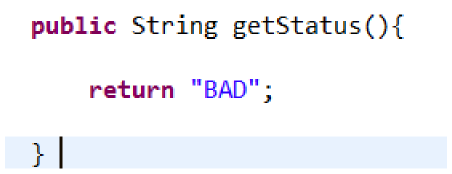
Part 5 - Make Changes and Trigger a Build

The project that we used as a sample consists of a basic "Hello World" style application, and a unit test for that application. In this section, we'll alter the core application so it fails the test, and then we'll see how that failure appears in Jenkins.

__1. In the **Project Explorer**, expand the **src/main/java/com/simple** tree node to reveal the **Greeting.java** file.

__2. Double-click on **Greeting.java** to open the file.

__3. Find the line that says 'return "GOOD";'. Edit the line to read 'return "BAD";'



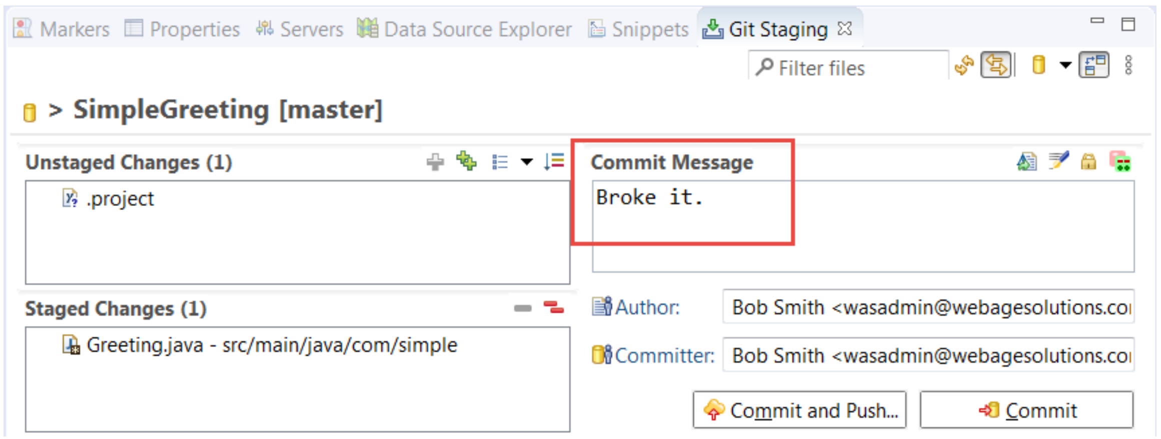
```
public String getStatus(){

    return "BAD";

}
```

__4. Save the file by pressing Ctrl-S or selecting **File** → **Save**.

Now we've edited the local file. The way Git works is that we'll first 'commit' the file to the local workspace repository, and then we'll 'push' the changes to the upstream repository. That's the same repository that Jenkins is reading from. Eclipse has a short-cut button that will commit and push at the same time.

__5. Right-click on **SimpleGreeting** in the **Project Explorer** and then select **Team** → **Commit.**

__6. Eclipse will open the **Git Staging** tab. Enter a few words as a commit message, and then click **Commit and Push**.



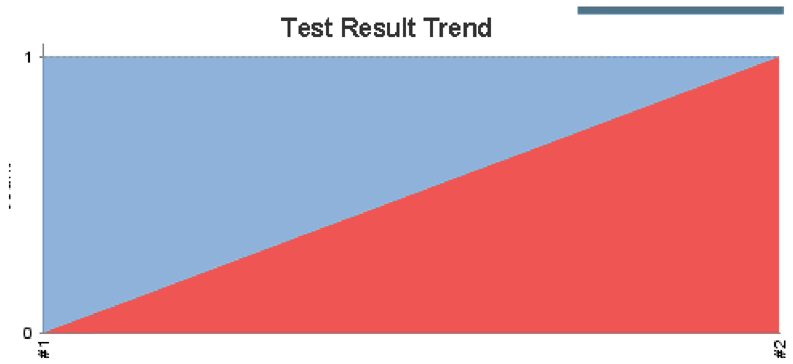
```
Broke it.
```

Author: Bob Smith <wasadmin@webagesolutions.co
Committer: Bob Smith <wasadmin@webagesolutions.co

__7. Click **Close** in the status dialog that pops up.

__8. Now, flip back to the web browser window that we had Jenkins running in. If you happen to have closed it, open a new browser window and navigate to **http://localhost:8080/SimpleGreeting**. After a few seconds, you should see a new build start up. You can launch a new build if it's taking too long.

__9. This build should fail. Refresh the page and you should see that there is now a 'Test Result Trend' graph that shows we have a new test failure.

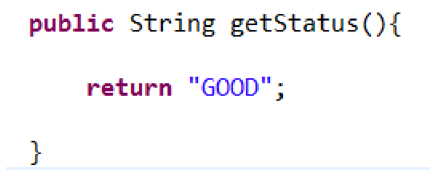


Make sure you are using Chrome, when writing this Lab, Mozilla was not working properly.

What happened is that we pushed the source code change to the Git repository that Jenkins is reading from. Jenkins is continually polling the repository to look for changes. When it saw that a new commit had been performed, Jenkins checked out a fresh copy of the source code and performed a build. Since Gradle automatically runs the unit tests as part of a build, the unit test was run. It failed, and the failure results were logged.

Part 6 - Fix the Unit Test Failure

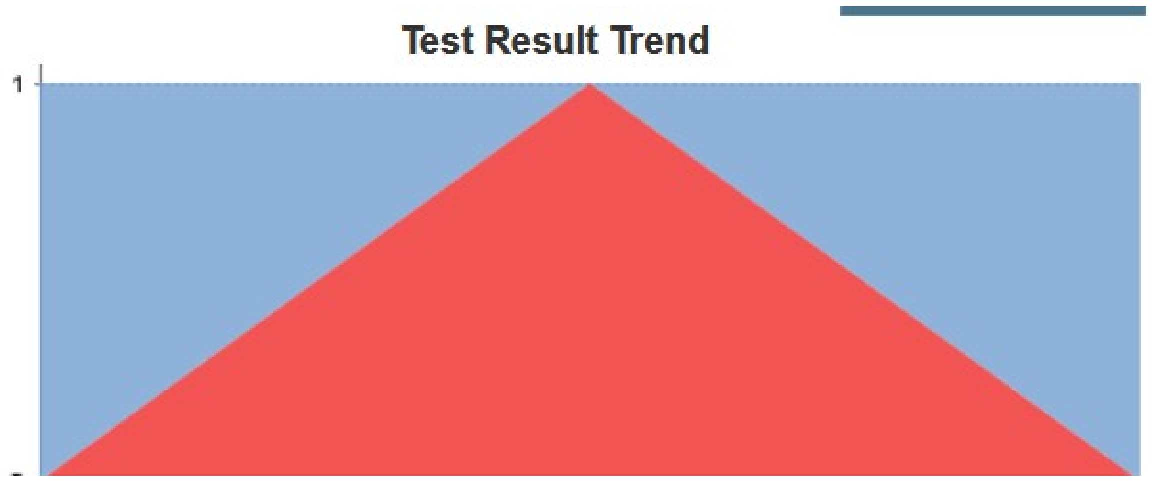
__1. Back in eclipse, edit the file **Greeting.java** so that the class once again returns 'GOOD'.



```
public String getStatus(){

    return "GOOD";

}
```

__2. As above, save, commit and push the change.

__3. Watch the Jenkins web browser window. After a minute or two you should see the build start automatically or you can click **Build now**, this time the build will pass, when the build is done then refresh the page.



__4. Close all.

Part 7 - Review

In this lab you learned

- How to Set-up a set of distributed Git repositories
- How to create a Jenkins Job that reads from a Git repository
- How to configure Jenkins to build automatically on source code changes.

Lab 7 - Create a Pipeline

In this lab you will explore the Pipeline functionality.

At the end of this lab you will be able to:

1. Create a simple pipeline
2. Use a 'Jenkinsfile' in your project
3. Use manual input steps in a pipeline

Part 1 - Create a Simple Pipeline

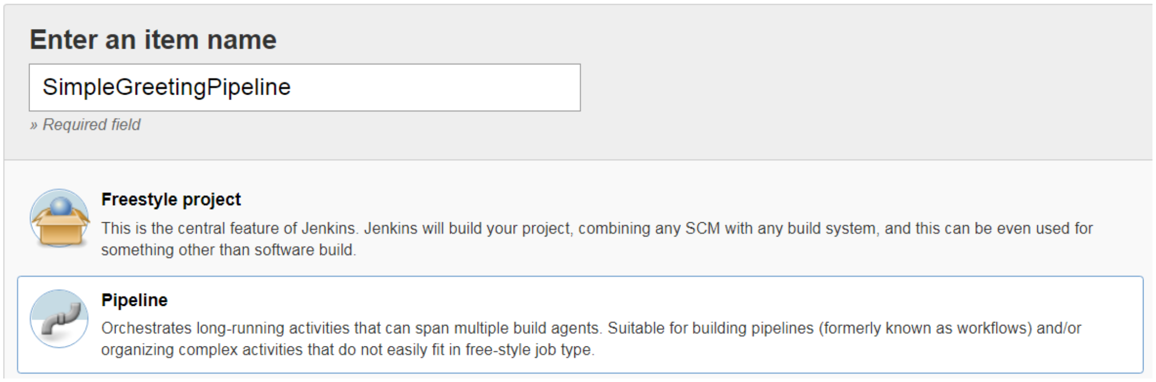
We can create a pipeline job that includes the pipeline script in the job configuration, or the pipeline script can be put into a 'Jenkinsfile' that's checked-in to version control.

To get a taste of the pipeline, we'll start off with a very simple pipeline defined in the job configuration.

Prerequisite: We need to have a project in source control to check-out and build. For this example, we're using the 'SimpleGreeting' project that we previously cloned to a 'Git' repository at 'C:\Software\repos\SimpleGreeting.git'

__1. In Jenkins, click on the **New Item** link.

__2. Enter '**SimpleGreetingPipeline**' as the new item name, and select '**Pipeline**' as the item type.



Enter an item name

SimpleGreetingPipeline

» Required field

Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

Pipeline

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

__3. When the input looks as above, click on **OK** to create the new item.

__4. Scroll down to the **Pipeline** section and enter the following in the **Script** text window.

```
node {
    stage('Checkout') {
      git url: 'C:\\Software\\repos\\SimpleGreeting.git'
    }

    stage('Gradle build') {
      bat 'gradle build'
    }
}
```

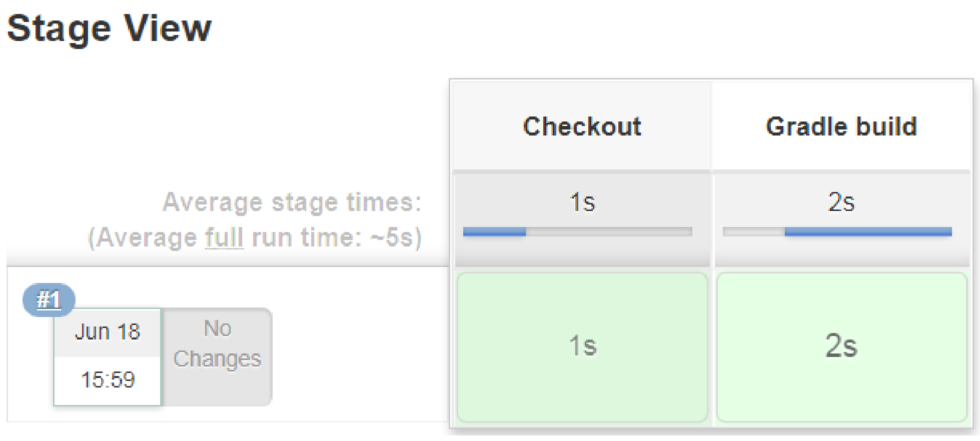
This pipeline is divided into two stages. First, we checkout the project from our 'git' repository. Then we use the 'bat' command to run 'gradle build' as a Windows batch file.

All of the above is wrapped inside the 'node' command, to indicate that we want to run these commands in the context of a workspace running on one of Jenkins execution agents (or the master node if no agents are available).

__5. Click on **Save** to save the changes and return to the project page.

__6. Click on **Build Now** to start up a pipeline instance.

__7. After a few moments, you should see the **Stage View** appear, and successive stages will appear as the build proceeds, until all three stages are completed.



Part 2 - Pipeline Definition in a 'Jenkinsfile'

For simple pipelines or experimentation, it's convenient to define the pipeline script in the web interface. But one of the common themes of modern software development is "If it isn't in version control, it didn't happen". The pipeline definition is no different, especially as you build more and more complex pipelines.

You can define the pipeline in a special file that is checked out from version control. There are several advantages to doing this. First, of course, is that the script is version-controlled. Second, we can edit the script with the editor or IDE of our choice before checking it in to version control. In addition, we can employ the same kind of "SCM Polling" that we would use in a more traditional Jenkins job.

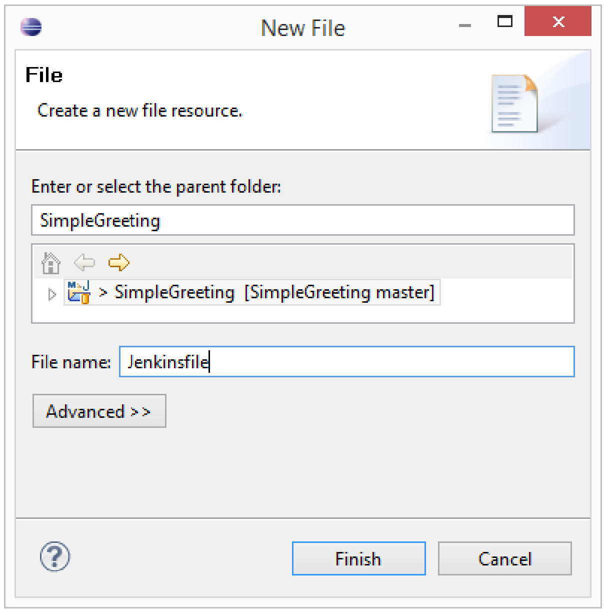
In the following steps, we'll create a Jenkinsfile and create a pipeline job that uses it.

__1. Open the Eclipse editor.

If this lab is completed in the normal sequence, you should have the 'SimpleGreeting' project already in Eclipse's workspace. If not, check out the project from version control (consult your instructor for directions if necessary).

__2. In the **Project Explorer**, right-click on the root node of the **SimpleGreeting** project, and then select **New → File**.

__3. Enter '**Jenkinsfile**' as the file name.


New File

File

Create a new file resource.

Enter or select the parent folder:

SimpleGreeting

> SimpleGreeting [SimpleGreeting master]

File name: Jenkinsfile

__4. Click **Finish** to create the new file.

__5. You may see a compatibility error, just click OK.

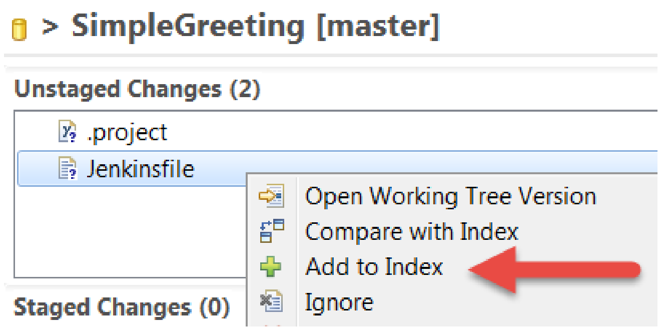
__6. Enter the following text into the new file (Note: this is the same script that we used above, so you could copy/paste it from the Jenkins Web UI if you want to avoid some typing):

```
node {
    stage('Checkout') {
      git url: 'C:\\Software\\repos\\SimpleGreeting.git'
    }

    stage('Gradle build') {
      bat 'gradle build'
    }
}
```

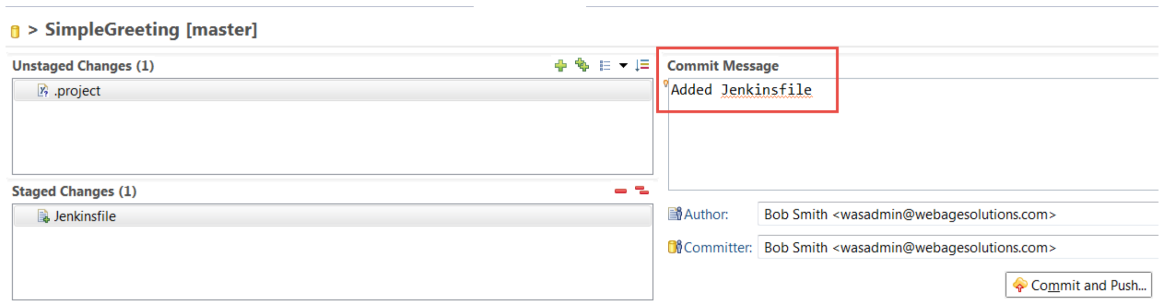
__7. Save the Jenkinsfile by selecting **File** → **Save** from the main menu, or by hitting Ctrl-S.

__8. In the **Project Explorer**, right-click on the **SimpleGreeting** node, and then select **Team** → **Commit...**

__9. Eclipse will open the **Git Staging** tab. Right click Jenkinsfile and click **Add to Index**.



__10. The file will be now available in the Staged Changed section. Add a comment and click **Commit and Push** then click Close to dismiss the next window.



```
Added Jenkinsfile
```

Author: Bob Smith <wasadmin@webagesolutions.com>

Committer: Bob Smith <wasadmin@webagesolutions.com>

Now we have a Jenkinsfile in our project, to define the pipeline. Next, we need to create a Jenkins job to use that pipeline.

__11. In the Jenkins user interface, navigate to the root page, and then click on **New Item**.

__12. Enter '**SimpleGreetingPipelineFromGit**' as the name of the new item.

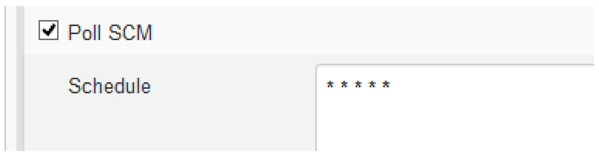
__13. Select **Pipeline** as the item type.

__14. Click **OK** to create the new item.

__15. Scroll down to the **Build Triggers** section.

__16. Click on **Poll SCM**

__17. Enter '*** * * * ***' as the polling schedule. This entry will cause Jenkins to poll once per minute.



☑ Poll SCM

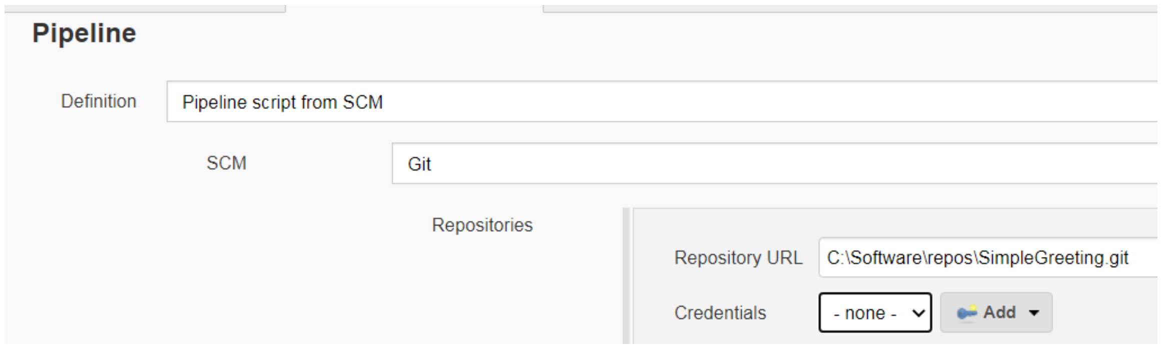
Schedule: * * * * *

__18. Scroll down to the **Pipeline** section, and change the **Definition** entry to '**Pipeline Script from SCM**'.

__19. Enter the following:

SCM:	**Git**
Repository URL:	**C:\Software\repos\SimpleGreeting.git**
	(Press the tab key)

__20. The **Pipeline** section should look similar to:


Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories

Repository URL: C:\Software\repos\SimpleGreeting.git

Credentials: - none -

__21. Click **Save** to save the new configuration.

__22. Click **Build Now** to launch the pipeline.

__23. You should see the pipeline execute, similar to the previous section.

Part 3 - Try out a Failing Build

The pipeline that we've defined so far appears to work perfectly. But we haven't tested it with a build that fails. In the following steps, we'll insert a test failure and see what happens to our pipeline.

__1. In Eclipse, go to the **Project Explorer** and locate the file 'Greeting.java'. It will be under **src/main/java** in the package 'com.simple'.

__2. Open 'Greeting.java'.

__3. Locate the line that reads 'return "GOOD";'. Change it to read 'return "BAD";'

```
public String getStatus(){
    return "BAD";
}
```

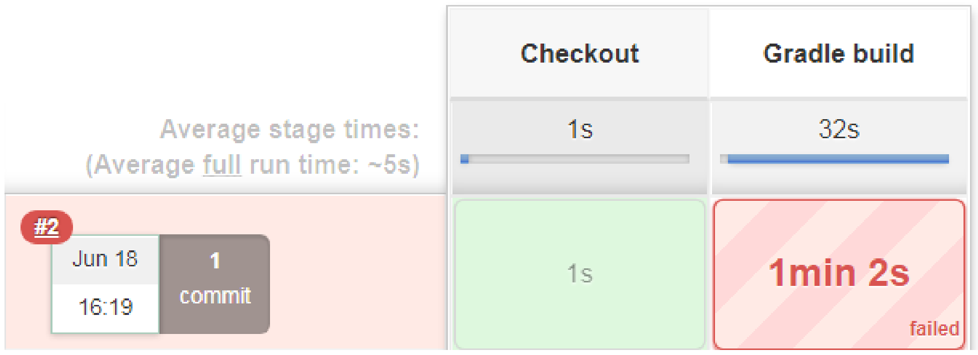
__4. Save the file.

__5. In the **Project Explorer**, right-click on **Greeting.java** and then select **Team** → **Commit**... (This is a shortcut for committing a single file).

__6. Enter an appropriate commit message and then click **Commit and Push**, then click **Close** in the results box.

__7. Switch back to Jenkins.

__8. In a minute or so, you should see a build launched automatically. Jenkins has picked up the change in the 'Git' repository and initiated a build. If nothing happens then click **Build Now**.



This time, the results are a little different. The 'Gradle Build' stage is showing a failure.

What's happened is that the unit tests have failed and Gradle exited with a non-zero result code because of the failure. As a result, the rest of the pipeline was canceled. In case if you want to perform a build without running unit tests, you can add *-x test* to *gradle build.* This way Gradle will continue with the build even if the tests fail.

__9. Go back to Eclipse and open the '**Jenkinsfile**' if necessary.

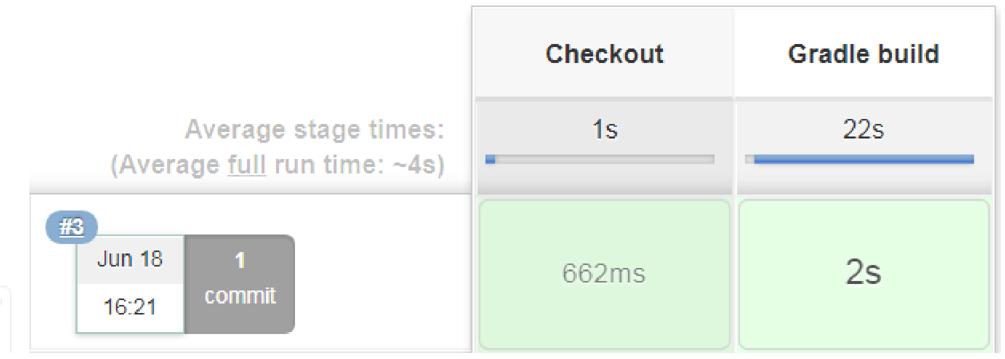
__10. Alter the 'bat "gradle" line to read as follows:

```
bat 'gradle build -x test'
```

__11. Save the 'Jenkinsfile'.

__12. Commit and push the changes using the same technique as above.

__13. After a minute or so, you should see a new Pipeline instance launched. If nothing happens then click **Build Now**.



This time, the pipeline runs to completion.

Part 4 - Add a Manual Approval Step

One of the interesting features of the Pipeline functionality is that we can include manual steps. This is very useful when we're implementing a continuous deployment pipeline. For example, we can include a manual approval step (or any other data collection) for cases like 'User Acceptance Testing' that might not be fully automated.

In the steps below, we'll add a manual step before a simulated deployment.

__1. Go to Eclipse and open the '**Jenkinsfile**' if necessary.

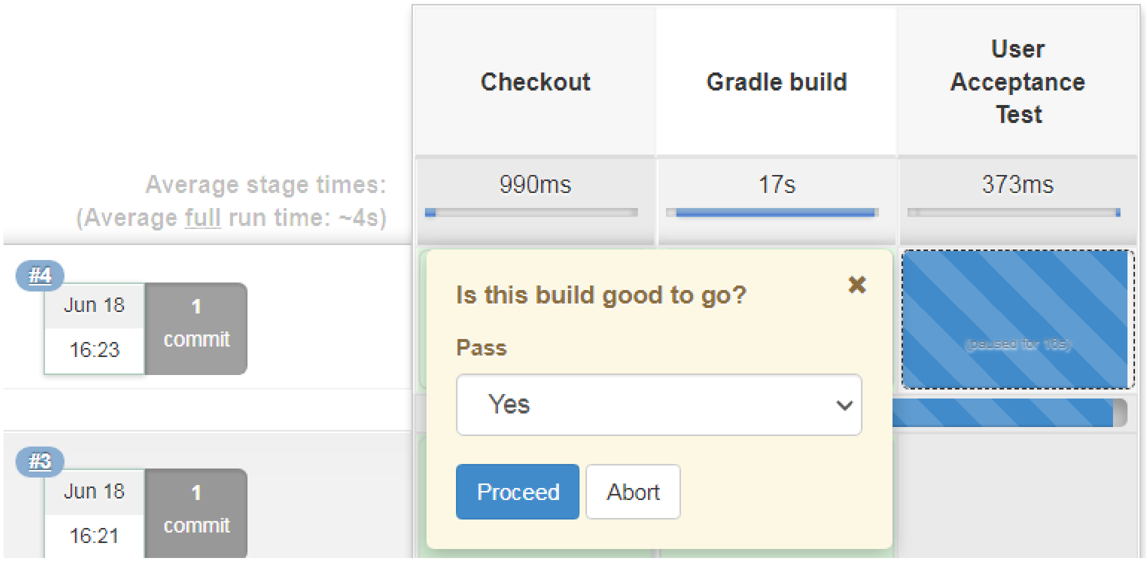
__2. Add the following to the end of the file before the **node** block closing }:

```
stage('User Acceptance Test') {

  def response= input message: 'Is this build good to go?',
   parameters: [choice(choices: 'Yes\nNo',
   description: '', name: 'Pass')]

  if(response=="Yes") {
    stage('Deploy') {
      bat 'gradle build -x test'
    }
  }
}
```

This portion of the script creates a new stage called 'User Acceptance Test', then executes an 'input' operation to gather input from the user. If the result is 'Yes', the script executes a deploy operation. (In this case, we're repeating the 'gradle build' that we did previously. Only because we don't actually have a deployment repository setup)

__3. Save and commit 'Jenkinsfile' as previously.

__4. When the pipeline executes, watch for a "paused" stage called 'User Acceptance Test". Move your mouse over this step, you'll be able to select "Yes" or "No".



	Checkout	Gradle build	User Acceptance Test
Average stage times: (Average full run time: ~4s)	990ms	17s	373ms
#4 Jun 18 16:23 1 commit			(paused for 16s)
#3 Jun 18 16:21 1 commit			

Is this build good to go?

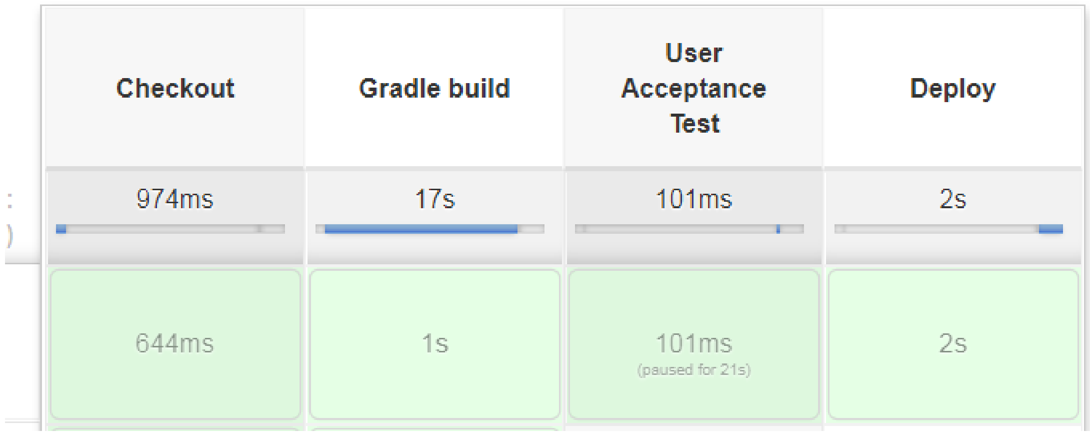
Pass

Yes

Proceed Abort

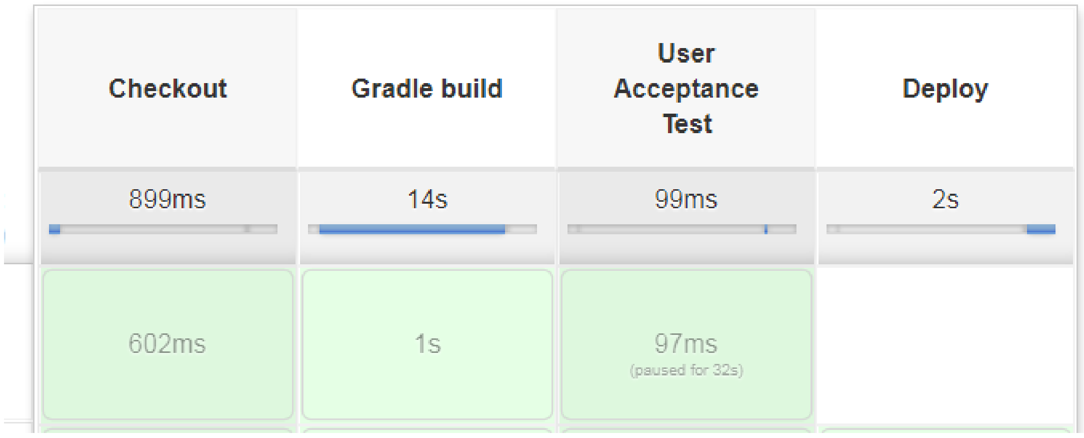
__5. Select **Yes** and click **Proceed**.

You should see the job run to completion.



Checkout	Gradle build	User Acceptance Test	Deploy
974ms	17s	101ms	2s
644ms	1s	101ms (paused for 21s)	2s

__6. Click **Build Now** to run the pipeline again but this time, select **No** on the 'User Acceptance Test' and click **Proceed**, you'll see that the pipeline exits early and doesn't run the 'deploy' stage.



Checkout	Gradle build	User Acceptance Test	Deploy
899ms	14s	99ms	2s
602ms	1s	97ms (paused for 32s)	

What's happened is that the final 'Deploy' stage was only executed when we indicated that the 'User Acceptance Test' had passed.

__7. Close all.

Part 5 - Review

In this lab, we explored the Pipeline functionality in Jenkins. We built a simple pipeline in the Jenkins web UI, and then used a 'Jenkinsfile' in the project. Lastly, we explored how to gather user input, and then take different build actions based on that user input.