Week 4

This week, we will learn how to configure Jenkins. In particular, how to use agents with Jenkins.

Agents are the execution environment to run work in the pipeline. They may be a physical server or docker container running in a kubernetes pod. Last week, the agent was the same as the Jenkins master. Running on the master is unsafe- if the pipeline was to crash, so would the master. It also does not scale. If many people were to run the pipeline at once the master's CPU would be under pressure.

We will set up agents to be separate pods. That pod could be scheduled on any machine within the kubernetes cluster. If you were running kubernetes in a more realistic setting with multiple servers, rather than your laptop, the pod could be scheduled on any one of those servers. Jenkins could then scale up to support more users.

This week's configuration will also set up one form of persistent storage, so you will not lose your pipelines in certain failure cases.

The agent pods will run an image with the tools to compile Java. We will build a pipeline that uses those tools. The code will be given to you (you do not need to write Java code in this course!), but your pipeline must compile it whenever a change is made in GitHub. There will be a second stage to test the code.

The labs will be demoed in the chat session and recorded (as will be done every week!)

Lab 1: setting up jenkins with multiple agents

This section creates "agents" (separate pods) to run pipelines. Chapter 3 of the textbook shows another way to accomplish this using helm. The two methods achieve the same results but working through the blog is instrumental in learning what is happening.

First tear down last week’s Jenkins and build a new one to work with agents and kubernetes. By deleting the namespace, everything in it (the deployment and its pods) will also be deleted.

```
kubectl delete all --all -n jenkins
```

Pull or clone week4's files from the umlS23 repository

```
git pull https://github.com/dlambrig/umlS23.git
```

We are interested in week4’s yaml files describing the deployment and service.

Create a namespace, then the pvc, then then deployment. **Note we start everything in devops-tools namespace**

```
kubectl create namespace devops-tools
kubectl apply -f jenkins-pvc.yaml -n devops-tools
kubectl apply -f jenkins-full-deployment.yaml -n devops-tools
```

This will start up Jenkins. Once it is running, follow instructions here *with class customization*. In the instructions, follow “Jenkins master and agents on the *same* kubernetes cluster”. Use the yaml file in the github repository.

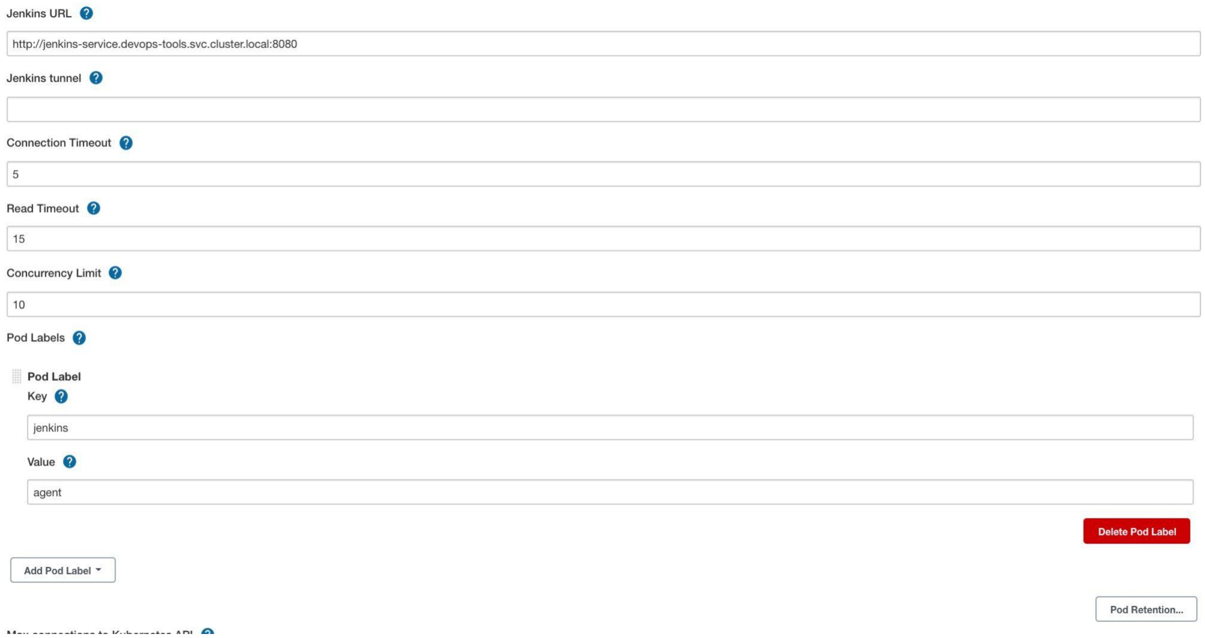
Notes:

- The instructions show how to run “Jenkins master *outside* the cluster and agents *in* the cluster”. In this lab’s setup, both the jenkins master and agents run *inside* Kubernetes.
- Debugging mistakes:
  - If you get stuck, tear everything down and start from the beginning.
  - Looking at jenkins logs is helpful for diagnosis.
  - Setting pod retention to "always" in the kubernetes configuration will prevent agents from being deleted on failure, allowing diagnosis. (this will be demoed)
- Apply the yaml files from the umlS23 git repository (not the one from the blog!) as we won't use a certificate key or service account.

First, do steps 1 and 2 "Jenkins Kubernetes Plugin Configuration" per scenario“. When you reach step 3, do the section "Jenkins server running inside the same Kubernetes cluster”. Then move to step 4.

Step 3 configures the "cloud" jenkins will connect with (kubernetes in this case). Make sure "test connection" works before continuing.

The namespace used in this demo is devops-tools.



Jenkins URL: http://jenkins-service.devops-tools.svc.cluster.local:8080

Jenkins tunnel:

Connection Timeout: 5

Read Timeout: 15

Concurrency Limit: 10

Pod Labels

Pod Label
Key: jenkins
Value: agent

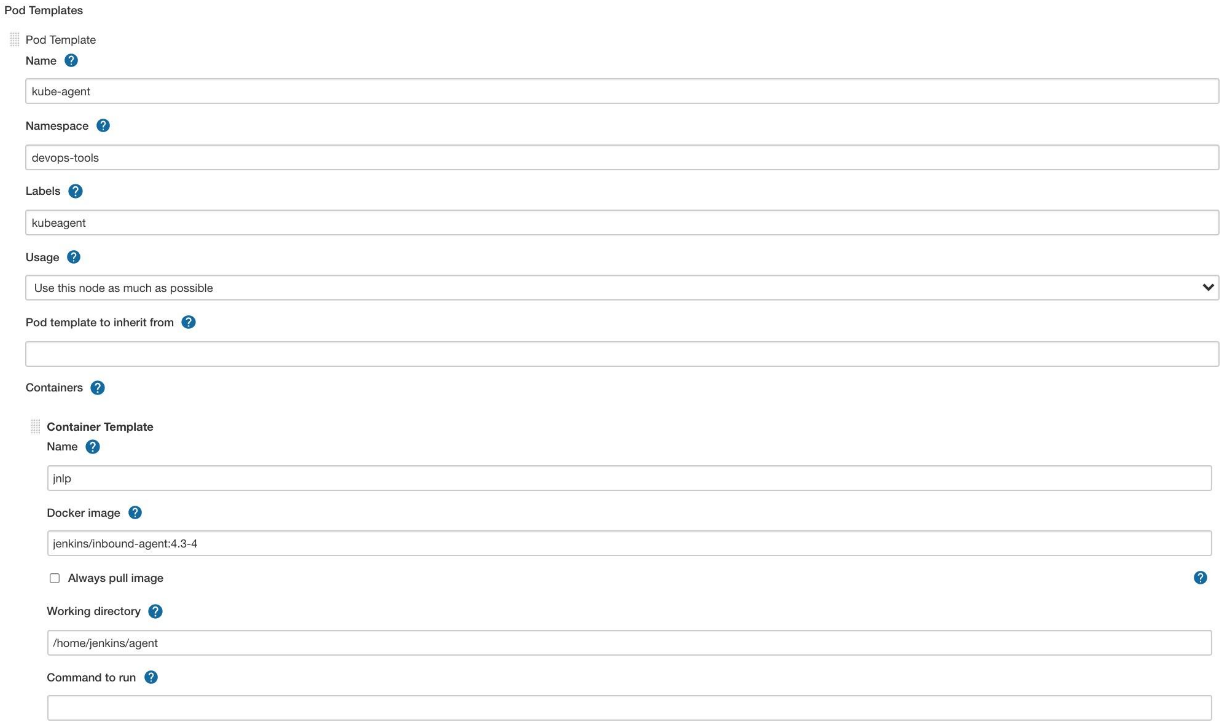
In step 4 configure the "jenkins URL": http://jenkins-service.devops-tools.svc.cluster.local:8080.

This is how other pods will access jenkins. Other pods do not know the IP address of the pod running jenkins, so this DNS name is used.

Our goal is to spawn agents (kubernetes pods) that will run our pipeline. Have a single "label" that will be associated with each pod. The label's value is "agent".

On Step 5, you create the "pod template". Important parts are "name", "namespace" and "labels".

The demo done Sunday used "jenkins/inbound-agent:latest" rather than "jenkins/inbound-agent:4.3-4".



Pod Templates

Pod Template

Name: kube-agent

Namespace: devops-tools

Labels: kubeagent

Usage: Use this node as much as possible

Pod template to inherit from:

Containers

Container Template

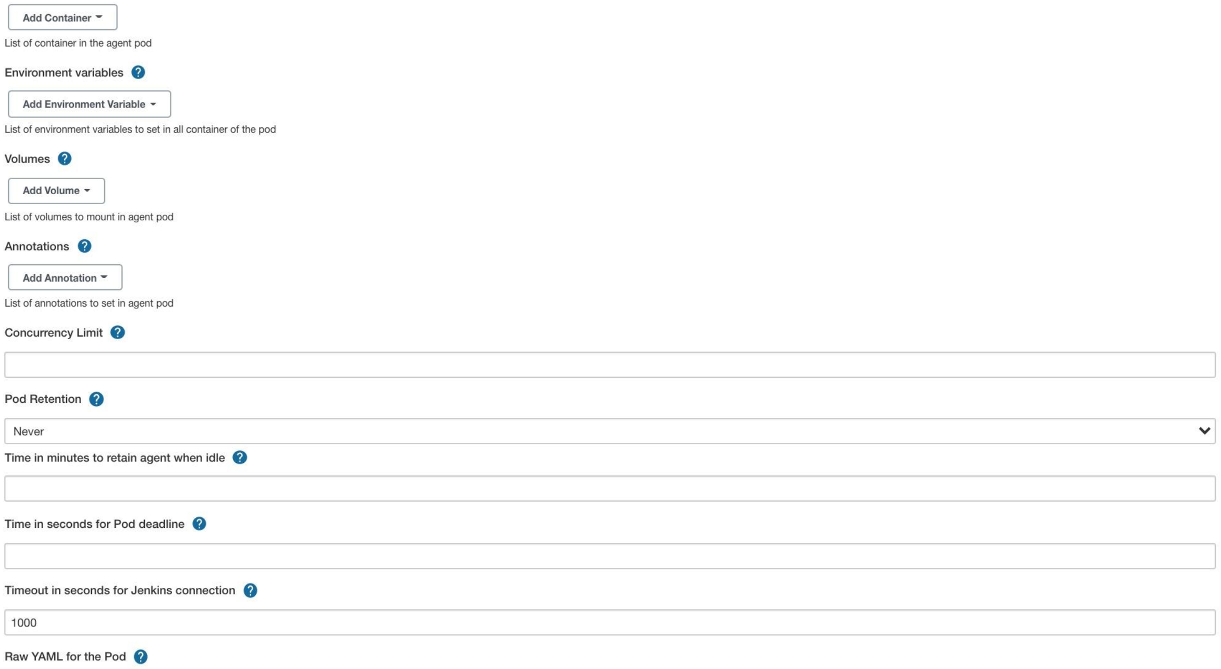
Name: jnlp

Docker image: jenkins/inbound-agent:4.3-4

☐ Always pull image

Working directory: /home/jenkins/agent

Command to run:



List of container in the agent pod

Environment variables

List of environment variables to set in all container of the pod

Volumes

List of volumes to mount in agent pod

Annotations

List of annotations to set in agent pod

Concurrency Limit:

Pod Retention: Never

Time in minutes to retain agent when idle:

Time in seconds for Pod deadline:

Timeout in seconds for Jenkins connection: 1000

Raw YAML for the Pod

By the end of step 5, you have configured jenkins to run inside kubernetes and to spawn off pods to run pipelines.

In step 6, you create a simple "freestyle" pipeline which you should then be able to build. Check “console output” to verify the output is as expected, as described in the blog. For that simple pipeline, the console output should look like this:

```
Building remotely on kube-agent-rjjpk (kubeagent) in workspace
/home/jenkins/agent/workspace/test
[test] $ /bin/sh -xe /tmp/jenkins3564413985139030409.sh
+ echo test
test
Finished: SUCCESS
```

Once this works, you are ready for a more complex pipeline in the next lab.

Lab 2 Compiling Java Code

This section describes how to compile Java code in a github repository.

1. An agent will be spun up that contains the "maven" Java build tool.
2. Java code in a github repository shall be downloaded into the agent.
3. The maven build tool will compile the Java code.

Notice that because we use containers, all Java code is downloaded automatically. You will not need to install anything on your PC.

Here is the Jenkinsfile. The syntax is from the kubernetes plugin.

```
podTemplate(containers: [
    containerTemplate(
        name: 'maven',
        image: 'maven:3.8.1-jdk-8',
        command: 'sleep',
        args: '30d'
        ),
  ]) {

    node(POD_LABEL) {
        stage('Get a Maven project') {
            git
'https://github.com/dlambrig/simple-java-maven-app.git'
            container('maven') {
                stage('Build a Maven project') {
                    sh '''
                    echo "maven build"
                    mvn -B -DskipTests clean package
                    '''
                    }
            }
        }

    }
  }
```

A podTemplate describes the container image that will run in the agent. The container() section defines commands to run inside the container.

Now trying building the project, in the same manner as last week.

Feel free to download the blue ocean plugin, as described in the Jenkins documentation. Once downloaded, click “blue ocean” on the main Jenkins page.

The blue ocean interface will allow you to observe the containers as they are being created. As you wait for the long process to complete, it is useful to know the machine is actually working (as opposed to being stuck).

The blue ocean interface will tell you when progress is made from the build stage to the test phase in a very nice display.