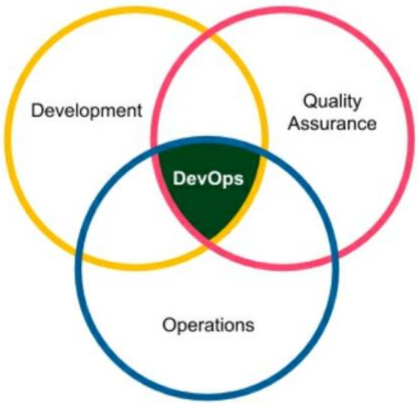
Kubernetes and Containers

DevOps is about making software development and delivery more efficient. Two technologies that give rise to this efficiency are containers and kubernetes. This week's material introduces those tools. Containers are lightweight virtual machines for packaging software. Kubernetes is an environment for running containers.

You should understand the different ways containers help devOps. For example, they can hide the complexity of software versioning and make it easy to manipulate bundles of software.

Kubernetes is an execution environment to run containers. It provides core services of replication, load balancing, and distribution of work over physical servers. It is portable, which means the same kubernetes configurations we run on our laptops can also run on AWS. It is scalable. In this course we will run our software pipelines ("Jenkins") in kubernetes.

This week we will also familiarize ourselves with basic devOpts concepts and responsibilities. We choose a lowest common denominator description: A devOpts practitioner manages tools that automate software testing and deployment. In doing so, the practitioner wears multiple hats: quality assurance (QA), operations, and developer.



Agile business practices are often used to efficiently coordinate work between devOps team members. "Agile" is a subject we will return to later in the course.

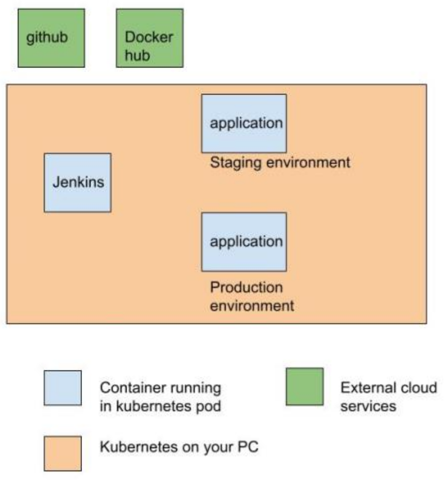
DevOps

DevOps practitioners tend to work in teams which have a variety of responsibilities including testing, operations, and development. A team member should have some skills in each of those areas, but often there are specialists within a team who focus on one area. But everyone works closely together. In this course we will mainly focus on the operations and testing areas.

In this class, the objective of devOps is to perform *continuous integration and delivery*. This means each change to the software (commit) becomes a release candidate, which can then be considered for release to production. We will build the "pipeline" that tests and integrates these changes.

The pipeline is the central tool in this course. All steps to build the software need to be *automated*, and run in the pipeline. The pipeline should run *quickly* so that each commit can be efficiently checked and integrated. It should not ever “stall”. Keeping it *running smoothly* is one of the main tasks of operations.

The diagram below depicts what we will be building over the next few weeks. Notice your kubernetes cluster will run both the jenkins pipeline and the constructed application. Our staging environment and production environment will also run within the kubernetes deployment on your PC. Each of the functionality groups (pipeline, staging, and production environment) may contain many containers. We will use kubernetes namespaces to logically organize functionality.



Containers

A core building block of our pipeline is containers, so we will begin with that.

An analogy to help motivate the benefits of containers may be found in the shipping industry. There are vastly different sizes and quantities of goods to move. There are vastly different transportation options to move goods.

The shipping container was invented to resolve this NxN problem and move goods efficiently. The shipping container is of a standard size. It can handle different quantities of goods. It can be used on different vehicles and vessels.

Similarly, system administrators must manage many applications across many platforms. There is a need for applications to run on any one of the different platforms, without worrying about how it is configured.

Containers go a long way towards realizing this vision. The advantage to containers are

- Containers simplify deployments by including their own dependencies in a single file. This means you do not need to worry about dependencies.
- Containers have very little overhead. Containers share the same operating system and run on “bare metal” hardware. In contrast, a full fledged virtual machine requires its own operating system, and the hardware is emulated. Due to their low overhead, you can probably host many more containers on your physical machine than you could full fledged VMs.

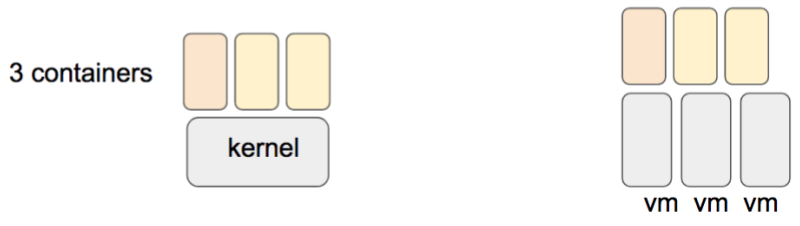
The disadvantage to containers is they cannot host a different OS, unlike real virtual machines. For example, a container could not run a windows application on a Linux machine.

What are containers?

This section describes how containers work.

Operating-system-level virtualization *is a server-virtualization method where the kernel of an operating system allows for multiple isolated user-space instances, instead of just one. Such instances (sometimes called* ***containers****,* ***software containers****, virtualization engines (VE), virtual private servers (VPS), or jails) may look and feel like a real server from the point of view of its owners and users.*

The picture below depicts containers vs virtual machines. Containers share the same OS. In contrast, each VM has its own OS.



Run three apps with containers   Run three apps with virtual machines

Because containers do not have to load an OS, they start up very quickly, and do not use a lot of memory.

	Virtual Machine	Container
Memory Overhead	Hi	Low
Startup time	Slow	Fast
Maintenance	Difficult	Easy
Scaling	10s/server	1000s/server
Primary use	Generic Infrastructure	Single daemon

A container is a combination of several techniques:

1. Containers have a unique file namespace. This means that when you are inside a container, you cannot see any files outside of the namespace. This is accomplished using the chroot tool. It changes the file system root.
2. cgroups . All processes in a container are members of the same process “cgroup”. They live and die together. Resources may be allocated to cgroups. You can freeze a group of processes in a cgroup.
3. Virtual networking is the plumbing to give a unique IP address to a container. The IP address can be externally visible or not depending on your preference.
4. Overlay filesystems are a mechanism to quickly load a lot of files into a container. The files are COW (copy on write). If you write to them, the overlay file system will intercept the write and store it in the overlay file system. This is all done transparently.

Container Orchestration

Once we have containers, we need an execution environment to run them in. The execution environment "orchestrates" the containers and provides many functions for scaling and management. A common scenario is to run a group of container instances on clusters of virtual machines. Kubernetes is the most popular orchestration framework.

Here are some of the capabilities found in Kubernetes:

- Containers and nodes are highly available (HA), meaning restart on failures.
- Work can be “load balanced” across multiple containers.
- “Auto scaling”: New nodes or containers can be created as needed automatically.
- Container code can be updated with rolling updates, meaning not all the containers are updated at once. Instead a few of them are, then there is a pause which gives an opportunity to make sure things worked. If things crash, the containers running the older code are unaffected.

Kubernetes can run in the cloud or on premise. The configuration may largely be the same in both settings which makes things easier to manage. Its all open source, which reduces lock-in to cloud vendors. Applications built within such portable infrastructure are called “cloud native”. They can easily be moved between different clouds or on premise.

Kubernetes Terminology

Do the online interactive tutorials here: “Learn Kubernetes Basics”. Get through modules 1-6.

After completing the demos you should understand these concepts.

Node - worker machine that runs containers
Pod - a group of one or more application containers. They share the same network namespace.
Replica set - maintain a stable set of replica Pods running at any given time
Service - allow your applications to receive traffic. A single IP address can be assigned to a group of pods and remains the same if pods change.
Load balancing - distribute work evenly across multiple pods
Deployment - supervises containers, e.g. restarts them if they die (using replica sets)

Commands (see also this cheat sheet):

kubectl get - list resources
kubectl describe - show detailed information about a resource
kubectl logs - print the logs from a container in a pod
kubectl exec - execute a command on a container in a pod
kubectl run - like docker run, but you can also create replicas of pods
kubectl expose - create a service

Lab 1: Docker

Docker is a company that creates open source software to build and distribute containers. The product we will use manages containers and a manner that is very simple. The company also maintains a "repository" of containers and allows you to upload your own.

Important docker concepts:

Images are like a snapshot of a VM. You can make your own. Containers are created from images. In the next example, we will download on ubunto image.

```
$ docker search debian
..
INDEX       NAME                                            DESCRIPTION
STARS     OFFICIAL   AUTOMATED
docker.io   docker.io/ubuntu                                Ubuntu is a Debian-based
Linux operating s...    7509      [OK]
docker.io   docker.io/debian                                Debian is a Linux
distribution that's comp...   2528      [OK]
..
```

You can download an Image.

```
$ sudo docker pull docker.io/ubuntu
Using default tag: latest
Trying to pull repository docker.io/library/ubuntu ...
latest: Pulling from docker.io/library/ubuntu
d3938036b19c: Pull complete
a9b30c108bda: Pull complete
67de21feec18: Pull complete
817da545be2b: Pull complete
d967c497ce23: Pull complete
Digest: sha256:9ee3b83bcaa383e5e3b657f042f4034c92cdd50c03f73166c145c9ceaea9ba7c
```

And then see what images you have downloaded

```
$ sudo docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED
SIZE
docker.io/ubuntu            latest              c9d990395902        2 days ago
112.9 MB
```

Once you have an image, you can build a container from it.

Next, create a container from an image

```
$ sudo docker run -it --rm --name dan_ubunto c9d990395902 /bin/bash
```

(-it means run interactively, --rm means delete the container when you exit)

This will put you inside the container - in an ubuntu environment. You will see apt-get works, rather than yum.

To break out, but leave it running.

```
[root@950e31bf320c /]# ^P^Q
```

(out of container, list what containers are running)

```
$ docker ps
```

```
root@216324bdb9ca:/# [ec2-user@ip-10-0-0-240 ~]$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS               NAMES
216324bdb9ca        c9d990395902        "/bin/bash"         6 seconds ago       Up 4 seconds
dan_ubunto
```

(re-attach to container, then exit out)

```
$ docker attach dan_ubunto
```

```
[root@950e31bf320c /]# exit
exit
```

Send information to a container at startup

```
$ docker run -it -e "parm1=myparm" --name dan_ubunto c9d990395902 /bin/bash
```

(in container, check environment variable)

```
[root@7acf58927c26 /]# env|grep parm1 && exit
parm1=myparm
exit
```

Next, we will create a docker file. This will allow us to create our own container. It has ubuntu plus the “python” language.

We will create a small python program. Create a file named: uml.py

```
cat << EOF > uml.py
print "Hello UML! This is a python program"
EOF
```

Create a file called “Dockerfile” and populate it like this:

```
cat << EOF > Dockerfile
FROM ubuntu:18.04
RUN apt-get -y update && apt-get -y install python
COPY uml.py .
ENTRYPOINT ["python", "uml.py"]
EOF
```

Next run command

```
docker build -t ubunto-with-python .
```

“Docker images” should show “ubunto-with-python” in the output.

The “FROM” command designates the base image.
The “RUN” command are what commands run inside the container as it is built.
The “COPY” command puts files into the image.
The “ENTRYPOINT” command specifies what executable is run when the container starts.

```
docker run --rm ubuntu-with-python
Hello UML! This is a python program
```

Lab 2: Kubernetes

This is taken from the online O'Reiley demo.

Start a *deployment* on the kubernetes system running on your PC. The deployment concept is described in the textbook. It is a watchdog to ensure the container(s) within a pod are always running.

```
kubectl create deployment http --image=katacoda/docker-http-
server:latest
```

Try running different commands to observe the deployment:

```
kubectl get deployments
kubectl describe deployment http
kubectl get pods --selector run=http
```

A pod is one or more containers.

You can also see the containers running the docker commands.

```
kubectl describe pod | grep Container
Containers:
    Container ID:
docker://f48f3ee741bb80eb618e1deccd1984e173d799fb5a78c4bdc6eed753cc2b4
e62
  ContainersReady   True

docker ps|grep f48f3ee741bb
```

Note on above: if you are using more than one node (such as in the O'Reilly "playground", you should run the docker ps command on the other node.

Lets increase the number of replicas to 3.

```
kubectl scale --replicas=3 deployment/http
kubectl get pods --selector app=http
NAME                    READY   STATUS              RESTARTS   AGE
http-774bb756bb-4xfgz   0/1     ContainerCreating   0          3s
http-774bb756bb-7hqd6   1/1     Running             0          19m
http-774bb756bb-t5b6v   0/1     ContainerCreating   0          3s
```

Try deleting a pod..

```
kubectl delete pod  http-774bb756bb-4xfgz
```

You will see the deployment starts a new one in its place.

```
$  kubectl get pods -o wide
NAME                    READY   STATUS    RESTARTS   AGE   IP
NODE       NOMINATED NODE   READINESS GATES
http-774bb756bb-7hqd6   1/1     Running   0          36m   172.18.0.6
minikube   <none>           <none>
http-774bb756bb-jvkr5   1/1     Running   0          15m   172.18.0.9
minikube   <none>           <none>
http-774bb756bb-t5b6v   1/1     Running   0          17m   172.18.0.7
minikube   <none>           <none>
```

Notice they all have different IP addresses. We need a single IP address to access our webserver. Accomplish this by creating a service resource.

```
kubectl port-forward  deployment/http  :80
Forwarding from 127.0.0.1:58536 -> 80
Forwarding from [::1]:58536 -> 80
Handling connection for 58536
Handling connection for 58536
```

Now a single IP address can access any of the pods, and there will be no effect on the end user if any pod's IP address changes.

You can test it, and view the service resource (be sure to use the right **port**):

```
curl http://172.17.0.13:58536
```

```
kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
http         ClusterIP   10.96.134.64   172.17.0.13   8000/TCP   2m40s
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP    42m
```

You can then tear down all the infrastructure

```
kubectl delete deployment http
```