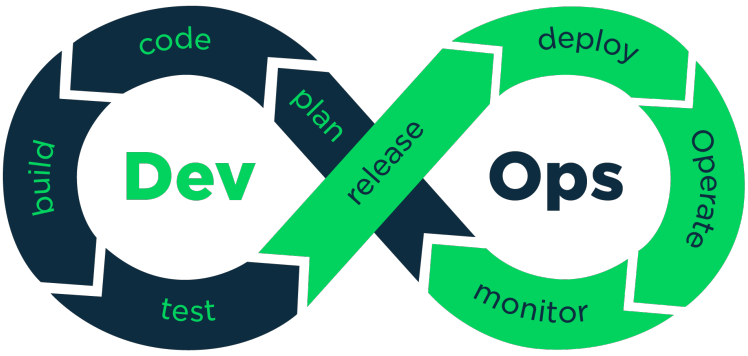
B5 - Advanced DevOps

B-DOP-500

Bernstein - Bootstrap

Grab the baton and become an amazing conductor



1.2.1

Bernstein - Bootstrap

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

As seen in the previous projects, using containers is an amazing way to ship code and to ease the deployment of applications and services, by removing the host infrastructure question burden.

However, during the previous projects, you only worked with single-host deployments, meaning that all containers you had to run were run on the same machine.
While being an easy infrastructure to set up, it also comes with a set of cons, such as:

- If the host goes down, the entire application will be unavailable.
- If the load is increasing, the scalability is very limited.

Fortunately, a solution to manage the deployment of containers across a set of hosts (also known as a *cluster*) exists and is in great use in the DevOps world: it is the notion of *orchestration*.

Managing the how, when, where and what to run is part of an orchestrator duties.

During this bootstrap, you are going to become familiar with one of the most popular orchestration platform: *Kubernetes*.

Step 0 – *Prelude to Kubernetes in YAML major, Op. 42*

A little quiz to start with

Before starting to use Kubernetes, it is important to gain some knowledge and to either acquire or clarify important points. Try to answer those questions with your friends:

- Where is Kubernetes?
- What is Kubernetes?
- Why is Kubernetes?
- What is a cluster?
- What is a node?
- What is a pod?
- In what way(s) are Kubernetes and Docker related?
- What is a Kubernetes namespace?

Do not skip this part, you need to be able to answer the above questions to be at ease with using Kubernetes afterwards.

Setup time

Installing a full Kubernetes cluster can be difficult at first glance, and it will not be needed for this bootstrap.

We will use a single node for this bootstrap, for the sake of simplicity, but keep in mind that **you will need a multi-node cluster for the project itself**.

Here is what will be needed:

- *Oracle VM VirtualBox* or *VMware*;
- *Minikube*, a simple tool to start a single-node Kubernetes cluster locally;
- *Docker*;
- *kubectl*, a command-line tool for sending commands to Kubernetes.

You can make your life easier with `kubectl completion bash`.

Your Mini Crewmate Minikube

Minikube starts a VM into VirtualBox (by default) and automatically installs a Kubernetes cluster inside. Nice. Try to figure out which 2-word commands allow you to do each of these actions:

1. *Start* Minikube.
2. Get the *status* of Minikube.
3. Get the *IP* address of Minikube.
4. *Stop* Minikube.
5. *Delete* the Minikube cluster.

Now that you know the five basics commands of Minikube, let us quickly configure `kubectl` (only needed the first time):

```
Terminal
~/B-DOP-500> kubectl config current-context
~/B-DOP-500> kubectl config use-context minikube
```

Do you `get` it?

`kubectl` allows you to `get` information on the different components of your cluster, including:

- nodes;
- namespaces;
- and pods.

Two-and-a-half quick questions:

- Which namespaces are automatically created on a fresh new cluster? What are their purpose?
- How to get pods from a specific namespace with `kubectl`?

Step 1 – Pod patrol

Create a simple pod named `hello-world` that runs a Docker container named `hello-world` with the following Docker image: `samber/hello-world-nodejs`.

Put its definition in a `hello-world.pod.yaml` file at the root of your repository.

You can test your pod with the following command:

```
Terminal
~/B-DOP-500> kubectl apply -f hello-world.pod.yaml
```

Then wait a few seconds for the deployment.

Now that the pod is deployed, find the command adapted to each of the following actions:

- Listing existing pods in the cluster.
- Printing more details about the pod.
- Fetching and following the pod container's logs.
- Executing the 'date' command inside the running container.
- Deleting the pod.

Step 2 – Environment variables for constant ecology

As you have understood with the previous projects, environment variables are the key to flexible, easy, and customizable deployments.

Update your `hello-world.pod.yaml` file and add the `PORT=8080` environment variable to the container.

Delete the pod and apply again the new configuration.

Now, check if it worked. You can either:

- execute `env` inside container;
- or execute `kubectl describe pod <pod-name>`.

Step 3 – Exposing ports, so that boats can dock(er)

The small application you are working with is a web server. Being able to access it would be nice, would not it?

Update your `hello-world.pod.yaml` file and ask Kubernetes to expose the port 8080.

Re-apply your configuration by… deleting and applying `hello-world.pod.yaml` again. ;)

The port has been exposed only inside the pod. To test your configuration, execute:

Terminal

```
~/B-DOP-500> kubectl port-forward [pod-name] 8080:8080
~/B-DOP-500> curl localhost:8080/
```

`kubectl port-forward` command built a tunnel between your computer and the Kubernetes cluster.

Now, can you print full configuration of the pod with `kubectl describe`?
Do you see the exposed port?
Do you see the pod's IP address?

Step 4 – Internal DNS for external comfort

Thanks to the previous step, other services can request the HTTP server on `<pod-ip>:8080`.

However, if the container dies, IP address will change. Big sad.
But there is a solution: *Domain Name Systems* (a.k.a. *DNS*)!

Create a Kubernetes service of type `ClusterIP` linked to the `hello-world` app exposing port 8080 to the same port on the cluster.

Put its definition in a `hello-world.service.yaml` file at the root of your repository.

Apply the service's configuration (you already know the command ;)).

If you print the created services' information, you should see the pod's IP in the `Endpoints` field.

Terminal

```
~/B-DOP-500> kubectl describe service [service-name] | grep Endpoints
```

Now, you should be able to ping your service from everywhere **within** the Kubernetes cluster:

```
Terminal
~/B-DOP-500> ping [service-name].[namespace].svc.cluster.local
```

The `ping` command should resolve the domain name (i.e.: display the IP address associated to it), but it might not actually receive a response. This is fine, the only thing that matters here is that the domain name can be resolved, thanks to the internal DNS.

Step 5 – Volumes (no joke here, because you do not joke about volumes)

Create a 512 MB *PersistentVolume* and attach it via a *PersistentVolumeClaim* to the `hello-world` container.

Put the definitions of the PersistentVolume and the PersistentVolumeClaim in a `hello-world.volume.yaml` file at the root of your repository.

Step 6 – Pods have evolved into deployments!

Pods are nice, but *deployments* are the real boss stuff.

Deployments offer more features, such as:

- replication policies;
- deployment history (and rollback!);
- and auto-scaling rules.

The real question is: can you convert your pod into a deployment?

Prove that you can by putting its new definition in a `hello-world.deployment.yaml` file at the root of your repository.

Step 8 – Wait, where is the step 7?

Thank you student, but the step 7 is in another document; namely, the project's document itself!

If you have reached this step and succeeded at the previous tasks, you are ready to grab the baton and to start conducting your very own symphony of containers.

Best of luck! :)