- acend gmbh

Setup

Setup instructions

This training depends on `oc`, the OpenShift command-line interface.

You have the choice of either using OpenShift's web terminal or installing `oc` locally.

If you prefer to not install anything on your computer, follow the instructions on the *1. Web terminal* page.

The *2. Local usage* chapter explains how to install `oc` for the respective operating system.

Also have a look at the *3. Other ways to work with OpenShift*, which is, however, totally optional.

> **Warning**
> In case you've already installed `oc`, please make sure you have an up-to-date version.

- acend gmbh

1. Web terminal

Using OpenShift's web terminal might be more convenient for you as it doesn't require you to install `oc` locally on your computer.

> **Note**
> If you do change your mind, head right over to *2. Local usage*.

Task 1.1: Login on the web console

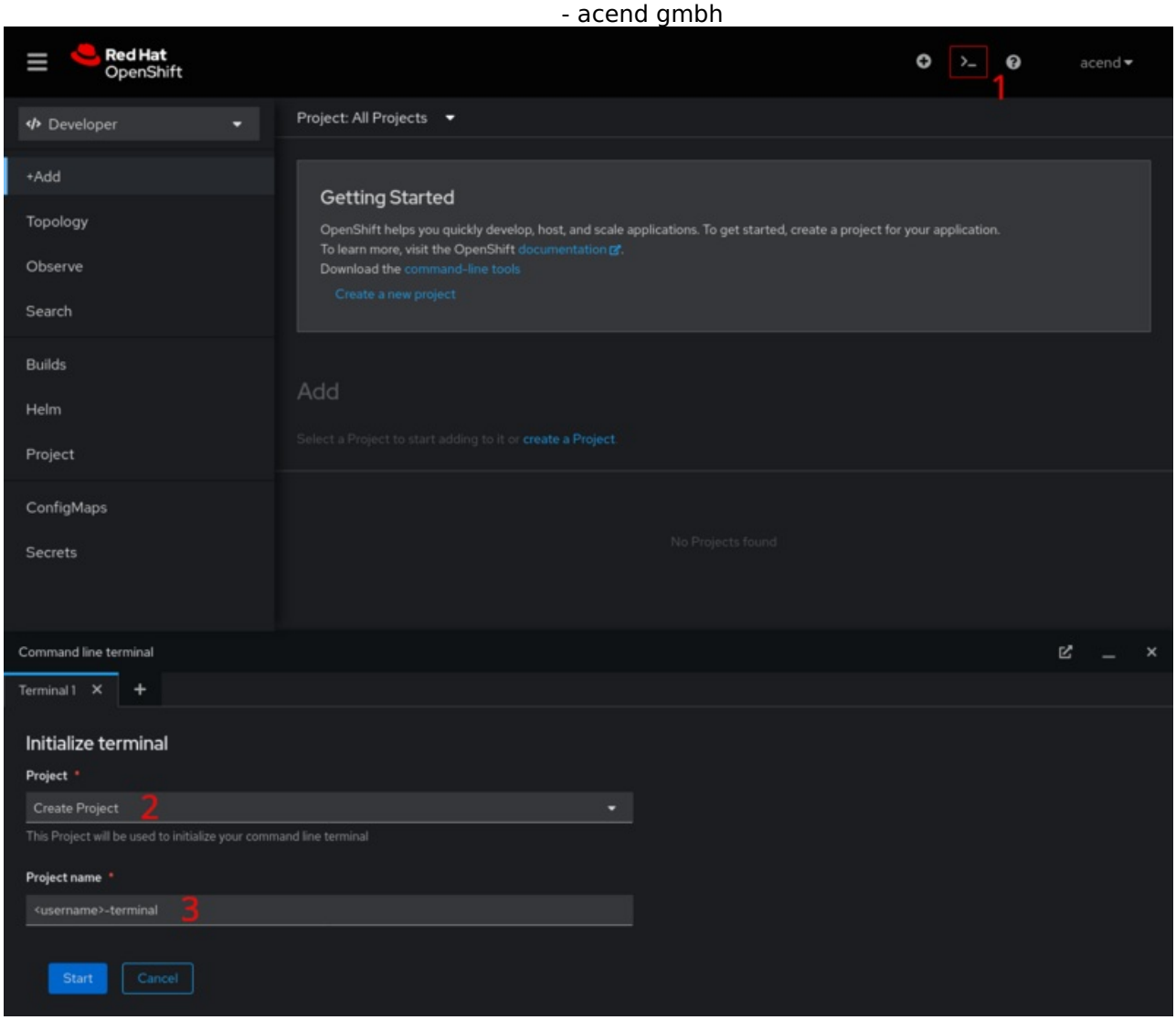
First of all, open your browser. Then, log in on OpenShift's web console using the URL and credentials provided by your trainer.

Task 1.2: Initialize terminal

> **Warning**
> Make sure to create a dedicated project for the web terminal!

In OpenShift's web console:

1. Click on the terminal icon on the upper right
2. Choose to create a new project
3. Name your project `<username>-terminal` where `<username>` is the username given to you during this training
4. Click **Start**



Task 1.3: Verification

After the initial setup, you're presented with a web terminal. Tools like `oc` are already installed and you're also already logged in.

You can check this by executing the following command:

```
oc whoami
```

You're now ready to go!

> **Warning**
> The terminal project is only meant to be used for the web terminal resources. Always check that you do not use the terminal namespace for the other labs!

- acend gmbh

Next steps

If you’re interested, have a look at the *3. Other ways to work with OpenShift*, which is however totally optional.

When you’re ready to go, head on over to the labs and begin with the training!

2. Local usage

Please follow the instructions on the *2.1. cli installation* page to install `oc`.

If you already have successfully installed `oc`, please verify that your installed version is current. Then, head over to *2.2. Console login* to log in.

2.1. cli installation

The `oc` command is the command-line interface to work with one or several OpenShift clusters.

The client is written in Go and you can run the single binary on the following operating systems:

- *2.1.1. Windows*
- *2.1.2. macOS*
- *2.1.3. Linux*

2.2. Console login

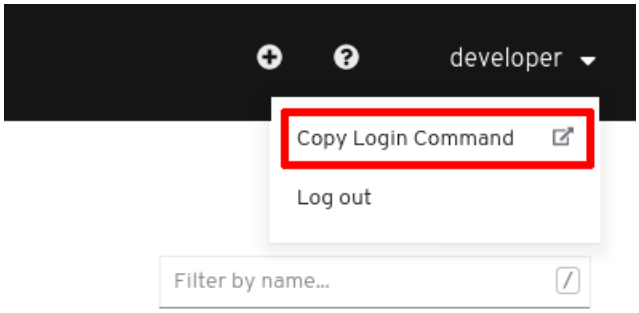
Task 2.2.1: Login on the web console

First of all, open your browser. Then, log in on OpenShift's web console using the URL and credentials provided by your trainer.

Task 2.2.2: Login on the command line

In order to log in on the command line, copy the login command from the web console.

To do that, open the Web Console and click on your username that you see at the top right, then choose **Copy Login Command**.



A new tab or window will open in your browser.

> **Note**
> You might need to log in again.

The page now displays a link **Display token**. Click on it and copy the command under **Log in with this token**.

Now paste the copied command on the command line.

Task 2.2.3: Verify login

If you now execute `oc version` you should see something like this (your output may vary):

```
Client Version: 4.11.2
Kustomize Version: v4.5.4
Kubernetes Version: v1.24.0+dc5a2fd
```

First steps with oc

The `oc` binary has many subcommands. Invoke `oc --help` (or simply `-h`) to get a list of all subcommands; `oc <subcommand> --help` gives you detailed help about a subcommand.

- acend gmbh

Next steps

If you’re interested, have a look at the *3. Other ways to work with OpenShift*, which is however totally optional.

When you’re ready to go, head on over to the labs and begin with the training!

3. Other ways to work with OpenShift

Other ways to work with OpenShift

If you don’t have access to a running OpenShift development environment (anymore), there are several options to get one.

- OpenShift Developer Sandbox : 30 days of no-cost access to a shared cluster on OpenShift
- OpenShift Local : A local OpenShift environmennt running on your machine
- OKD single node installation : OKD (OpenShift community edition) single node installation

Next steps

When you’re ready to go, head on over to the labs and begin with the training!

- acend gmbh

Labs

The purpose of these labs is to convey OpenShift basics by providing hands-on tasks for people. OpenShift will allow you to deploy and deliver your software packaged as containers in an easy, straightforward way.

Goals of these labs:

- Help you get started with this modern technology
- Explain the basic concepts to you
- Show you how to deploy your first applications on Kubernetes

Additional Docs

- OpenShift Docs

Additional Tutorials

- OpenShift Interactive Learning Portal

- acend gmbh

1. Introduction

In this lab, we will introduce the core concepts of OpenShift.

All explanations and resources used in this lab give only a quick and not detailed overview. As OpenShift is based on Kubernetes, its concepts also apply to OpenShift which you can find in the official Kubernetes documentation .

Core concepts

With the open source software OpenShift, you get a platform to build and deploy your application in a container as well as operate it at the same time. Therefore, OpenShift is also called a *Container Platform*, or the term *Container-as-a-Service* (CaaS) is used.

Depending on the configuration the term *Platform-as-a-Service* (PaaS) works as well.

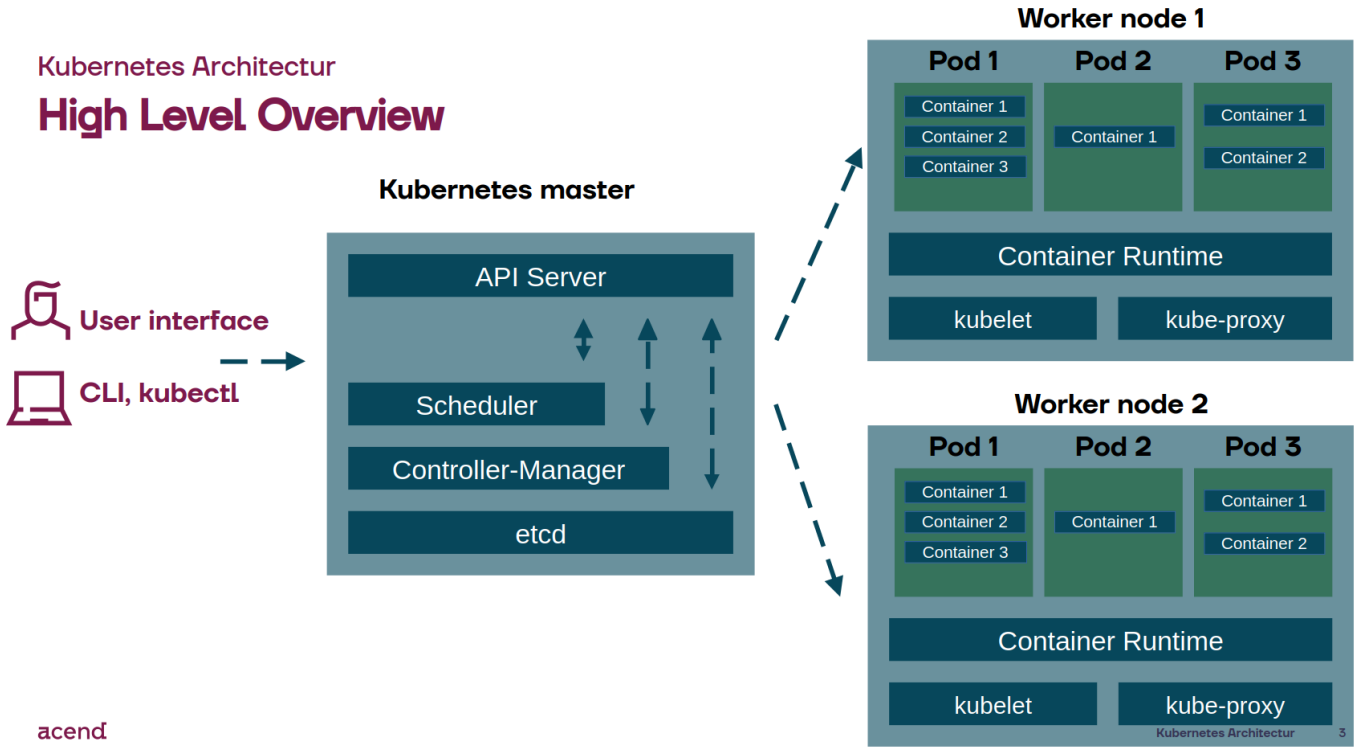
Container engine

OpenShift's underlying container engine is CRI-O . Earlier releases used Docker .

Docker was originally created to help developers test their applications in their continuous integration environments. Nowadays, system admins also use it. CRI-O doesn't exist as long as Docker does. It is a "lightweight container runtime for Kubernetes" and is fully OCI-compliant .

Overview

OpenShift basically consists of control plane and worker nodes.



Control plane and worker nodes

The control plane components are the *API server*, the *scheduler* and the *controller manager*. The API server itself represents the management interface. The scheduler and the controller manager decide how applications should be deployed on the cluster. Additionally, the state and configuration of the cluster itself are controlled in the control plane components.

Worker nodes are also known as compute nodes, application nodes or minions, and are responsible for running the container workload (applications). The *control plane* for the worker nodes is implemented in the control plane components. The hosts running these components were historically called masters.

Containers and images

The smallest entities in Kubernetes and OpenShift are Pods, which resemble your containerized application.

Using container virtualization, processes on a Linux system can be isolated up to a level where only the predefined resources are available. Several containers can run on the same system without “seeing” each other (files, process IDs, network). One container should contain one application (web server, database, cache, etc.). It should be at least one part of the application, e.g. when running a multi-service middleware. In a container itself any process can be started that runs natively on your operating system.

Containers are based on images. An image represents the file tree, which includes the binary, shared libraries and other files which are needed to run your application.

A container image is typically built from a `Containerfile` or `Dockerfile`, which is a text file filled with instructions. The end result is a hierarchically layered binary construct. Depending on the backend, the implementation uses overlay or copy-on-write (COW) mechanisms to represent the image.

Layer example for a Tomcat application:

1. Base image (Alpine)
2. Install Java
3. Install Tomcat
4. Install App

The pre-built images under version control can be saved in an image registry and can then be used by the container platform.

Namespaces and Projects

Namespaces in Kubernetes represent a logical segregation of unique names for entities (Pods, Services, Deployments, ConfigMaps, etc.).

In OpenShift, users do not directly create Namespaces, they create Projects. A Project is a Namespace with additional annotations.

> **Note**
> OpenShift’s concept of a Project does not coincide with Rancher’s.

Permissions and roles can be bound on a per-project basis. This way, a user can control his own resources inside a Project.

> **Note**
> Some resources are valid cluster-wise and cannot be set and controlled on a namespace basis.

Pods

A Pod is the smallest entity in Kubernetes and OpenShift.

It represents one instance of your running application process. The Pod consists of at least one container which contains your application. The application ports from inside the Pod are exposed via Services.

Services

A service represents a static endpoint for your application in the Pod. As a Pod and its IP address typically are considered dynamic, the IP address of the Service does not change when changing the application inside the Pod. If you scale up your Pods, you have an automatic internal load balancing towards all Pod IP addresses.

There are different kinds of Services:

- `ClusterIP` : Default virtual IP address range
- `NodePort` : Same as `ClusterIP` plus open ports on the nodes
- `LoadBalancer` : An external load balancer is created, only works in cloud environments, e.g. AWS ELB
- `ExternalName` : A DNS entry is created, also only works in cloud environments

A Service is unique inside a Namespace.

Deployment

Have a look at the official documentation .

Volume

Have a look at the official documentation .

Job

Have a look at the official documentation .

History

There is a official Kubernetes Documentary available on Youtube.

- Kubernetes: The Documentary [PART 1]
- Kubernetes: The Documentary [PART 2]

Inspired by the open source success of Docker in 2013 and seeing the need for innovation in the area of large-scale cloud computing, a handful of forward-thinking Google engineers set to work on the container orchestrator that would come to be known as Kubernetes– this new tool would forever change the way the internet is built.

These engineers overcome technical challenges, resistance to open source from within, naysayers, and intense competition from other big players in the industry.

Most engineers know about “The Container Orchestrator Wars’’ but most people would not be able to explain exactly what happened, and why it was Kubernetes that ultimately came out on top.

There is no topic more relevant to the current open source landscape. This film captures the story directly from the people who lived it, featuring interviews with prominent engineers from Google, Red Hat, Twitter

- acend gmbh

and others.

1.1. YAML

YAML Ain't Markup Language (YAML) is a human-readable data-serialization language. YAML is not a programming language. It is mostly used for storing configuration information.

> **Note**
> Data serialization is the process of converting data objects, or object states present in complex data structures, into a stream of bytes for storage, transfer, and distribution in a form that can allow recovery of its original structure.

As you will see a lot of YAML in our Kubernetes basics course, we want to make sure you can read and write YAML. If you are not yet familiar with YAML, this introduction is waiting for you. Otherwise, feel free to skip it or come back later if you meet some less familiar YAML stuff.

This introduction is based on the YAML Tutorial from cloudbees.com .

For more information and the full spec have a look at https://yaml.org/

A simple file

Let's look at a YAML file for an overview:

```yaml
---
foo: "foo is not bar"
bar: "bar is not foo"
pi: 3.14159
awesome: true
kubernetes-birth-year: 2015
cloud-native:
  - scalable
  - dynamic
  - cloud
  - container
kubernetes:
  version: "1.22.0"
  deployed: true
  applications:
    - name: "My App"
      location: "public cloud"
```

The file starts with three dashes. These dashes indicate the start of a new YAML document. YAML supports multiple documents, and compliant parsers will recognize each set of dashes as the beginning of a new one.

Then we see the construct that makes up most of a typical YAML document: a key-value pair. `foo` is a key that points to a string value: `foo is not bar`

YAML knows four different data types:

- `foo` & `bar` are strings.
- `pi` is a floating-point number
- `awesome` is a boolean
- `kubernetes-birth-year` is an integer

- acend gmbh

You can enclose strings in single or double-quotes or no quotes at all. YAML recognizes unquoted numerals as integers or floating point.

The `cloud-native` item is an array with four elements, each denoted by an opening dash. The elements in `cloud-native` are indented with two spaces. Indentation is how YAML denotes nesting. The number of spaces can vary from file to file, but tabs are not allowed.

Finally, `kubernetes` is a dictionary that contains a string `version`, a boolean `deployed` and an array `applications` where the item of the array contains two `strings`.

YAML supports nesting of key-values, and mixing types.

Indentation and Whitespace

Whitespace is part of YAML's formatting. Unless otherwise indicated, newlines indicate the end of a field. You structure a YAML document with indentation. The indentation level can be one or more spaces. The specification forbids tabs because tools treat them differently.

Comments

Comments begin with a pound sign. They can appear after a document value or take up an entire line.

```
---
# This is a full line comment
foo: bar # this is a comment, too
```

YAML data types

Values in YAML's key-value pairs are scalar. They act like the scalar types in languages like Perl, Javascript, and Python. It's usually good enough to enclose strings in quotes, leave numbers unquoted, and let the parser figure it out. But that's only the tip of the iceberg. YAML is capable of a great deal more.

Key-Value Pairs and Dictionaries

The key-value is YAML's basic building block. Every item in a YAML document is a member of at least one dictionary. The key is always a string. The value is a scalar so that it can be any datatype. So, as we've already seen, the value can be a string, a number, or another dictionary.

Numeric types

YAML recognizes numeric types. We saw floating point and integers above. YAML supports several other numeric types. An integer can be decimal, hexadecimal, or octal.

```
---
foo: 12345
bar: 0x12d4
plop: 023332
```

YAML supports both fixed and exponential floating point numbers.

```
---
foo: 1230.15
bar: 12.3015e+05
```

Finally, we can represent not-a-number (NAN) or infinity.

```
---
foo: .inf
bar: -.Inf
plop: .NAN
```

Foo is infinity. Bar is negative infinity, and plop is NAN.

Strings

YAML strings are Unicode. In most situations, you don't have to specify them in quotes.

```
---
foo: this is a normal string
```

But if we want escape sequences handled, we need to use double quotes.

```
---
foo: "this is not a normal string\n"
bar: this is not a normal string\n
```

YAML processes the first value as ending with a carriage return and linefeed. Since the second value is not quoted, YAML treats the `\n` as two characters.

```
foo: this is not a normal string
bar: this is not a normal string\n
```

YAML will not escape strings with single quotes, but the single quotes do avoid having string contents interpreted as document formatting. String values can span more than one line. With the fold (greater than) character, you can specify a string in a block.

```
bar: >
  this is not a normal string it
  spans more than
  one line
  see?
```

But it's interpreted without the newlines: `bar : this is not a normal string it spans more than one line see?`

The block (pipe) character has a similar function, but YAML interprets the field exactly as is.

```
---
bar: |
  this is not a normal string it
  spans more than
  one line
  see?
```

So, we see the newlines where they are in the document.

```
bar : this is not a normal string it
spans more than
one line
see?
```

Nulls

You enter nulls with a tilde or the unquoted null string literal.

```
---
foo: ~
bar: null
```

Booleans

YAML indicates boolean values with the keywords True, On and Yes for true. False is indicated with False, Off, or No.

```
---
foo: True
bar: False
light: On
TV: Off
```

Arrays

You can specify arrays or lists on a single line.

```
---
items: [ 1, 2, 3, 4, 5 ]
names: [ "one", "two", "three", "four" ]
```

Or, you can put them on multiple lines.

```
---
items:
  - 1
  - 2
  - 3
  - 4
  - 5
names:
  - "one"
  - "two"
  - "three"
  - "four"
```

The multiple line format is useful for lists that contain complex objects instead of scalars.

```
---
items:
  - things:
      thing1: huey
      things2: dewey
      thing3: louie
  - other things:
      key: value
```

An array can contain any valid YAML value. The values in a list do not have to be the same type.

Dictionaries

We covered dictionaries above, but there's more to them. Like arrays, you can put dictionaries inline. We saw this format above.

```
---
foo: { thing1: huey, thing2: louie, thing3: dewey }
```

We've seen them span lines before.

```
---
foo: bar
bar: foo
```

And, of course, they can be nested and hold any value.

```
---
foo:
  bar:
    - bar
    - rab
    - plop
```

2. First steps

In this lab, we will interact with the OpenShift cluster for the first time.

> **Warning**
> Please make sure you completed *Setup* before you continue with this lab.

Projects

As a first step on the cluster, we are going to create a new Project.

A Project is a logical design used in OpenShift to organize and separate your applications, Deployments, Pods, Ingresses, Services, etc. on a top-level basis. Authorized users inside a Project are able to manage those resources. Project names have to be unique in your cluster.

Task 2.2: Create a Project

Create a new Project in the lab environment. The `oc help` output can help you figure out the right command.

> **Note**
> Please choose an identifying name for your Project, e.g. your initials or name as a prefix. We are going to use `<namespace>` as a placeholder for your created Project.

Solution

To create a new Project on your cluster use the following command:

```
oc new-project <namespace>
```

> **Note**
> In order to declare what Project to use, you have several possibilities:
>
> - Some prefer to explicitly select the Project for each `oc` command by adding `--namespace <namespace>` or `-n <namespace>`
> - By using the following command, you can switch into another Project instead of specifying it for each `oc` command
>
> ```
> oc project <namespace>
> ```

Task 2.3: Discover the OpenShift web console

Discover the different menu entries in the two views, the **Developer** and the **Administrator** view.

Display all existing Pods in the previously created Project with `oc` (there shouldn't yet be any):

```
oc get pod --namespace <namespace>
```

> **Note**
> With the command `oc get` you can display all kinds of resources.

3. Deploying a container image

In this lab, we are going to deploy our first container image and look at the concepts of Pods, Services, and Deployments.

Task 3.1: Start and stop a single Pod

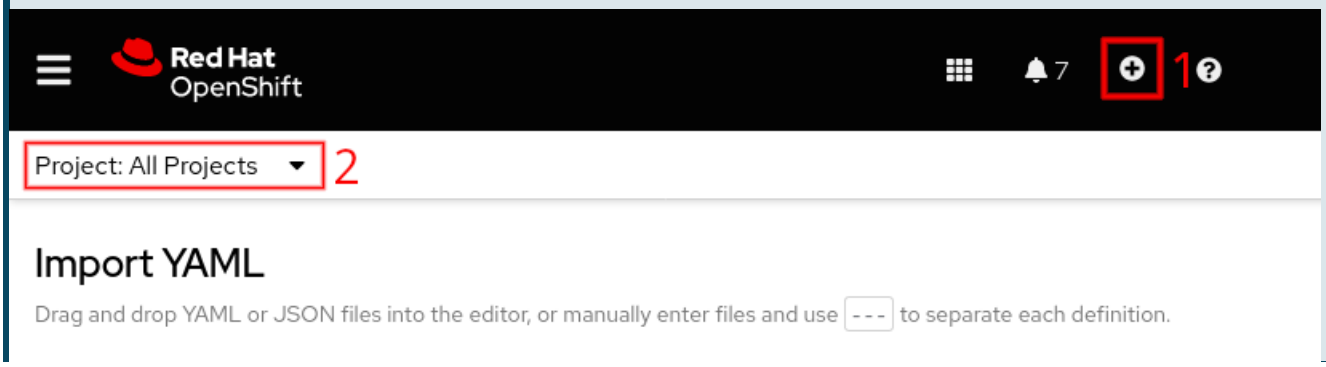
After we've familiarized ourselves with the platform, we are going to have a look at deploying a pre-built container image from Quay.io or any other public container registry.

In OpenShift we have used the `<project>` identifier to select the correct project. Please use the same identifier in the context `<namespace>` to do the same for all upcoming labs. Ask your trainer if you want more information on that.

First, we are going to directly start a new Pod. For this we have to define our Kubernetes Pod resource definition. Create a new file `pod_awesome-app.yaml` with the content below.

Note

Alternatively, you can create the Pod definition on the web console. Simply click on the **plus sign button** on the upper right (1), make sure you've selected the correct **Project** (2) and paste the content.




```
apiVersion: v1
kind: Pod
metadata:
  name: awesome-app
spec:
  containers:
  - image: quay.io/acend/example-web-go:latest
    imagePullPolicy: Always
    name: awesome-app
    resources:
      limits:
        cpu: 20m
        memory: 32Mi
      requests:
        cpu: 10m
        memory: 16Mi
```

Note

If you used the web console to import the Pod's YAML definition, don't execute the following command.

Now we can apply this with:

```
oc apply -f pod_awesome-app.yaml --namespace <namespace>
```

The output should be:

```
pod/awesome-app created
```

Use `oc get pods --namespace <namespace>` in order to show the running Pod:

```
oc get pods --namespace <namespace>
```

Which gives you an output similar to this:

```
NAME          READY   STATUS    RESTARTS   AGE
awesome-app   1/1     Running   0          1m24s
```

Have a look at your awesome-app Pod inside the OpenShift web console.

Now delete the newly created Pod:

```
oc delete pod awesome-app --namespace <namespace>
```

Task 3.2: Create a Deployment

In some use cases it can make sense to start a single Pod. But this has its downsides and is not really a common practice. Let’s look at another concept which is tightly coupled with the Pod: the so-called *Deployment*. A Deployment ensures that a Pod is monitored and checks that the number of running Pods corresponds to the number of requested Pods.

To create a new Deployment we first define our Deployment in a new file `deployment_example-web-go.yaml` with the content below.

> **Note**
> You could, of course, again import the YAML on the web console as described above.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: example-web-go
  name: example-web-go
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-web-go
  template:
    metadata:
      labels:
        app: example-web-go
    spec:
      containers:
        - image: quay.io/acend/example-web-go:latest
          name: example-web-go
          resources:
            requests:
              cpu: 10m
              memory: 16Mi
            limits:
              cpu: 20m
              memory: 32Mi
```

And with this we create our Deployment inside our already created namespace:

> **Note**
> If you used the web console to import the Deployment's YAML definition, don't execute the following command.

```
oc apply -f deployment_example-web-go.yaml --namespace <namespace>
```

The output should be:

```
deployment.apps/example-web-go created
```

We're using a simple sample application written in Go, which you can find built as an image on Quay.io or as source code on GitHub .

OpenShift creates the defined and necessary resources, pulls the container image (in this case from Quay.io) and deploys the Pod.

Use the command `oc get` with the `-w` parameter in order to get the requested resources and afterward watch for changes.

> **Note**
> The `oc get -w` command will never end unless you terminate it with `CTRL-c`.

```
oc get pods -w --namespace <namespace>
```

> **Note**
> Instead of using the `-w` parameter you can also use the `watch` command which should be available on most Linux distributions:

```
watch oc get pods --namespace <namespace>
```

This process can last for some time depending on your internet connection and if the image is already available locally.

> **Note**
> If you want to create your own container images and use them with OpenShift, you definitely should have a look at these best practices and apply them. This image creation guide may be for OpenShift, however it also applies to Kubernetes and other container platforms.

Creating Kubernetes resources

There are two fundamentally different ways to create Kubernetes resources. You've already seen one way: Writing the resource's definition in YAML (or JSON) and then applying it on the cluster using `oc apply`.

The other variant is to use helper commands. These are more straightforward: You don't have to copy a YAML definition from somewhere else and then adapt it. However, the result is the same. The helper commands just simplify the process of creating the YAML definitions.

As an example, let's look at creating above deployment, this time using a helper command instead. If you already created the Deployment using above YAML definition, you don't have to execute this command:

```
oc create deployment example-web-go --image=quay.io/acend/example-web-go:latest --namespace <namespace>
```

It's important to know that these helper commands exist. However, in a world where GitOps concepts have an ever-increasing presence, the idea is not to constantly create these resources with helper commands. Instead, we save the resources' YAML definitions in a Git repository and leave the creation and management of those resources to a tool.

Task 3.3: Viewing the created resources

Display the created Deployment using the following command:

```
oc get deployments --namespace <namespace>
```

A Deployment defines the following facts:

- Update strategy: How application updates should be executed and how the Pods are exchanged
- Containers
  - Which image should be deployed
  - Environment configuration for Pods
  - ImagePullPolicy

- The number of Pods/Replicas that should be deployed

By using the `-o` (or `--output`) parameter we get a lot more information about the deployment itself. You can choose between YAML and JSON formatting by indicating `-o yaml` or `-o json`. In this training we are going to use YAML, but please feel free to replace `yaml` with `json` if you prefer.

```
oc get deployment example-web-go -o yaml --namespace <namespace>
```

After the image has been pulled, OpenShift deploys a Pod according to the Deployment:

```
oc get pods --namespace <namespace>
```

which gives you an output similar to this:

```
NAME                              READY   STATUS    RESTARTS   AGE
example-web-go-69b658f647-xnm94   1/1     Running   0          39s
```

The Deployment defines that one replica should be deployed — which is running as we can see in the output. This Pod is not yet reachable from outside the cluster.

Task 3.4: Verify the Deployment in the OpenShift web console

Try to display the logs from the example application in the OpenShift web console.

Task 3.5: Build the image yourself

Up until now, we've used pre-built images from Quay.io. OpenShift offers the ability to build images on the cluster itself using different strategies :

- Docker build strategy
- Source-to-image build strategy
- Custom build strategy
- Pipeline build strategy

We are going to use the Docker build strategy. It expects:

> [...] a repository with a Dockerfile and all required artifacts in it to produce a runnable image.

All of these requirements are already fulfilled in the source code repository on GitHub , so let's build the image!

> **Note**
> Have a look at OpenShift's documentation to learn more about the other available build strategies.

First we clean up the already existing Deployment:

```
oc delete deployment example-web-go --namespace <namespace>
```

We are now ready to create the build and deployment, all in one command:

```
oc new-app --name example-web-go --labels app=example-web-go --context-dir go/ --strategy docker https://github.com/ace
nd/awesome-apps.git --namespace <namespace>
```

Let’s watch the image’s build process:

```
oc logs bc/example-web-go --follow --namespace <namespace>
```

The message `Push successful` signifies the image’s successful build and push to OpenShift’s internal image.

In the above command you discovered a new resource type `bc` which is the abbreviation for *BuildConfig*. A BuildConfig defines how a container image has to be built.

A *Build* resource represents the build process itself based upon the BuildConfig’s definition. A build takes place in a Pod on OpenShift, so instead of referencing the BuildConfig in our `oc logs` command, we could have used the build Pod’s log output. However, referencing the BuildConfig has the advantage that it can be reused each time a build is run. A build Pod changes its name with every build.

Have a look at the new Deployment created by the `oc new-app` command:

```
oc get deployment example-web-go -o yaml --namespace <namespace>
```

It looks the same as before with the only essential exception that it uses the image we just built instead of the pre-built image from Quay.io:

```
    ...
    spec:
      containers:
      - image: image-registry.openshift-image-registry.svc:5000/<namespace>/awesome-app@sha256:4cd671273a837453464f7264
afe845b299297ebe032f940fd005cf9c40d1e76c
      ...
```

4. Exposing a service

In this lab, we are going to make the freshly deployed application from the last lab available online.

Task 4.1: Create a ClusterIP Service

The command `oc apply -f deployment_example-web-go.yaml` from the last lab creates a Deployment but no Service. A OpenShift Service is an abstract way to expose an application running on a set of Pods as a network service. For some parts of your application (for example, frontends) you may want to expose a Service to an external IP address which is outside your cluster.

OpenShift `ServiceTypes` allow you to specify what kind of Service you want. The default is `ClusterIP` .

`Type` values and their behaviors are:

- `ClusterIP` : Exposes the Service on a cluster-internal IP. Choosing this value only makes the Service reachable from within the cluster. This is the default ServiceType.
- `NodePort` : Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service from outside the cluster, by requesting <NodeIP>:<NodePort>.
- `LoadBalancer` : Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- `ExternalName` : Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

You can also use Ingress to expose your Service. Ingress is not a Service type, but it acts as the entry point for your cluster. Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Route resource. A Route may be configured to give Services externally reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An Ingress controller is responsible for fulfilling the route, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

In order to create a Route, we first need to create a Service of type ClusterIP .

To create the Service add a new file `svc-web-go.yaml` with the following content:

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: example-web-go
  name: example-web-go
spec:
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
  selector:
    app: example-web-go
  type: ClusterIP
```

And then apply the file with:

```
oc apply -f svc-web-go.yaml --namespace <namespace>
```

There is also am imperative command to create a service and expose your application which can be used instead of the yaml file with the `oc apply ...` command

```
oc expose deployment example-web-go --type=ClusterIP --name=example-web-go --port=5000 --target-port=5000 --namespace <
namespace>
```

You will get the error message reading `Error from server (AlreadyExists): services "example-web-go" already exists` here. This is because the `oc new-app` command you executed during lab 3 already created a service. This is the default behavior of `oc new-app` while `oc create deployment` doesn't have this functionality.

As a consequence, the `oc expose` command above doesn't add anything new but it demonstrates how to easily create a service based on a deployment.

Let's have a more detailed look at our Service:

```
oc get services --namespace <namespace>
```

Which gives you an output similar to this:

```
NAME             TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
example-web-go   ClusterIP  10.43.91.62   <none>        5000/TCP
```

Note

Service IP (CLUSTER-IP) addresses stay the same for the duration of the Service's lifespan.

By executing the following command:

```
oc get service example-web-go -o yaml --namespace <namespace>
```

You get additional information:

```
apiVersion: v1
kind: Service
metadata:
  ...
  labels:
    app: example-web-go
  managedFields:
    ...
  name: example-web-go
  namespace: <namespace>
  ...
spec:
  clusterIP: 10.43.91.62
  externalTrafficPolicy: Cluster
  ports:
  - port: 5000
    protocol: TCP
    targetPort: 5000
  selector:
    app: example-web-go
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

The Service's `selector` defines which Pods are being used as Endpoints. This happens based on labels. Look at the configuration of Service and Pod in order to find out what maps to what:

```
oc get service example-web-go -o yaml --namespace <namespace>
```

```
...
  selector:
    app: example-web-go
...
```

With the following command you get details from the Pod:

> **Note**
> First, get all Pod names from your namespace with (`oc get pods --namespace <namespace>`) and then replace <pod> in the following command. If you have installed and configured the bash completion, you can also press the TAB key for autocompletion of the Pod's name.

```
oc get pod <pod> -o yaml --namespace <namespace>
```

Let's have a look at the label section of the Pod and verify that the Service selector matches the Pod's labels:

```
...
  labels:
    app: example-web-go
...
```

This link between Service and Pod can also be displayed in an easier fashion with the `oc describe` command:

```
oc describe service example-web-go --namespace <namespace>
```

```
Name:                     example-web-go
Namespace:                example-ns
Labels:                   app=example-web-go
Annotations:              <none>
Selector:                 app=example-web-go
Type:                     ClusterIP
IP:                       10.39.240.212
Port:                     <unset>  5000/TCP
TargetPort:               5000/TCP
Endpoints:                10.36.0.8:5000
Session Affinity:         None
External Traffic Policy:  Cluster
Events:
  Type    Reason          Age    From          Message
  ----    ------          ----   ----          -------
```

The `Endpoints` show the IP addresses of all currently matched Pods.

Task 4.2: Expose the Service

With the ClusterIP Service ready, we can now create the Route resource.

```
oc create route edge example-web-go --service example-web-go --namespace <namespace>
```

The output should be:

```
route.route.openshift.io/example-web-go created
```

We are now able to access our app via the freshly created route at `https://example-web-go-<namespace>.<appdomain>`

Find your actual app URL by looking at your route (HOST/PORT):

```
oc get route --namespace <namespace>
```

Browse to the URL and check the output of your app.

Note

If the site doesn't load, check if you are using the http:// , not the https:// protocol, which might be the default in your browser.

Note

The `<appdomain>` is the default domain under which your applications will be accessible and is provided by your trainer. You can also use `oc get route example-web-go` to see the exact value of the exposed route.

Task 4.4: For fast learners

Have a closer look at the resources created in your namespace `<namespace>` with the following commands and try to understand them:

```
oc describe namespace <namespace>
```

```
oc get all --namespace <namespace>
```

```
oc describe <resource> <name> --namespace <namespace>
```

```
oc get <resource> <name> -o yaml --namespace <namespace>
```

5. Scaling

In this lab, we are going to show you how to scale applications on OpenShift. Furthermore, we show you how OpenShift makes sure that the number of requested Pods is up and running and how an application can tell the platform that it is ready to receive requests.

> **Note**
> This lab does not depend on previous labs. You can start with an empty Namespace.

Task 5.1: Scale the example application

Create a new Deployment in your Namespace. So again, lets define the Deployment using YAML in a file `deployment_example-web-app.yaml` with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: example-web-app
  name: example-web-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-web-app
  template:
    metadata:
      labels:
        app: example-web-app
    spec:
      containers:
      - image: quay.io/appuio/example-spring-boot:latest
        name: example-web-app
        resources:
          limits:
            cpu: 750m
            memory: 512Mi
          requests:
            cpu: 50m
            memory: 128Mi
```

and then apply with:

```
oc apply -f deployment_example-web-app.yaml --namespace <namespace>
```

If we want to scale our example application, we have to tell the Deployment that we want to have three running replicas instead of one. Let’s have a closer look at the existing ReplicaSet:

```
oc get replicasets --namespace <namespace>
```

Which will give you an output similar to this:

```
NAME                              DESIRED   CURRENT   READY   AGE
example-web-app-86d9d584f8        1         1         1       110s
```

Or for even more details:

```
oc get replicaset <replicaset> -o yaml --namespace <namespace>
```

The ReplicaSet shows how many instances of a Pod are desired, current and ready.

Now we scale our application to three replicas:

```
oc scale deployment example-web-app --replicas=3 --namespace <namespace>
```

Check the number of desired, current and ready replicas:

```
oc get replicasets --namespace <namespace>
```

```
NAME                              DESIRED   CURRENT   READY   AGE
example-web-app-86d9d584f8        3         3         3       4m33s
```

Look at how many Pods there are:

```
oc get pods --namespace <namespace>
```

Which gives you an output similar to this:

```
NAME                                READY   STATUS    RESTARTS   AGE
example-web-app-86d9d584f8-7vjcj    1/1     Running   0          5m2s
example-web-app-86d9d584f8-hbvlv    1/1     Running   0          31s
example-web-app-86d9d584f8-qg499    1/1     Running   0          31s
```

Note

OpenShift supports horizontal and vertical autoscaling .

As we changed the number of replicas with the `oc scale deployment` command, the `example-web-app` Deployment now differs from your local `deployment_example-web-app.yaml` file. Change your local `deployment_example-web-app.yaml` file to match the current number of replicas and update the value `replicas` to `3` :

```
[...]
metadata:
  labels:
    app: example-web-app
  name: example-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example-web-app
[...]
```

Check for uninterruptible Deployments

Now we expose our application to the internet by creating a service and a route.

First the service:

```
oc expose deployment example-web-app --name="example-web-app" --port=8080 --namespace <namespace>
```

Then the route:

```
oc create route edge example-web-app --port 5000 --service example-web-app --namespace <namespace>
```

Let’s look at our Service. We should see all three corresponding Endpoints:

```
oc describe service example-web-app --namespace <namespace>
```

```
Name:              example-web-app
Namespace:         acend-test
Labels:            app=example-web-app
Annotations:       <none>
Selector:          app=example-web-app
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                172.30.89.44
IPs:               172.30.89.44
Port:              <unset>  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.125.4.70:8080,10.126.4.137:8080,10.126.4.138:8080
Session Affinity:  None
Events:            <none>
```

Scaling of Pods is fast as OpenShift simply creates new containers.

You can check the availability of your Service while you scale the number of replicas up and down in your browser: `http://<route hostname>` .

> **Note**
> You can find out the route’s hostname by looking at the output of `oc get route` .

Now, execute the corresponding loop command for your operating system in another console.

Linux:

```
URL=$(oc get routes example-web-app -o go-template="{{ .spec.host }}" --namespace <namespace>)
while true; do sleep 1; curl -s https://${URL}/pod/; date "+ TIME: %H:%M:%S,%3N"; done
```

Windows PowerShell:

```
while(1) {
  Start-Sleep -s 1
  Invoke-RestMethod https://<URL>/pod/
  Get-Date -Uformat "+ TIME: %H:%M:%S,%3N"
}
```

Scale from 3 replicas to 1. The output shows which Pod is still alive and is responding to requests:

```
example-web-app-86d9d584f8-7vjcj TIME: 17:33:07,289
example-web-app-86d9d584f8-7vjcj TIME: 17:33:08,357
example-web-app-86d9d584f8-hbvlv TIME: 17:33:09,423
example-web-app-86d9d584f8-7vjcj TIME: 17:33:10,494
example-web-app-86d9d584f8-qg499 TIME: 17:33:11,559
example-web-app-86d9d584f8-hbvlv TIME: 17:33:12,629
example-web-app-86d9d584f8-qg499 TIME: 17:33:13,695
example-web-app-86d9d584f8-hbvlv TIME: 17:33:14,771
example-web-app-86d9d584f8-hbvlv TIME: 17:33:15,840
example-web-app-86d9d584f8-7vjcj TIME: 17:33:16,912
example-web-app-86d9d584f8-7vjcj TIME: 17:33:17,980
example-web-app-86d9d584f8-7vjcj TIME: 17:33:19,051
example-web-app-86d9d584f8-7vjcj TIME: 17:33:20,119
example-web-app-86d9d584f8-7vjcj TIME: 17:33:21,182
example-web-app-86d9d584f8-7vjcj TIME: 17:33:22,248
example-web-app-86d9d584f8-7vjcj TIME: 17:33:23,313
example-web-app-86d9d584f8-7vjcj TIME: 17:33:24,377
example-web-app-86d9d584f8-7vjcj TIME: 17:33:25,445
example-web-app-86d9d584f8-7vjcj TIME: 17:33:26,513
```

The requests get distributed amongst the three Pods. As soon as you scale down to one Pod, there should be only one remaining Pod that responds.

Let’s make another test: What happens if you start a new Deployment while our request generator is still running?

```
oc rollout restart deployment example-web-app --namespace <namespace>
```

During a short period we won’t get a response:

```
example-spring-boot-2-73aln TIME: 16:48:25,251
example-spring-boot-2-73aln TIME: 16:48:26,305
example-spring-boot-2-73aln TIME: 16:48:27,400
example-spring-boot-2-73aln TIME: 16:48:28,463
example-spring-boot-2-73aln TIME: 16:48:29,507
<html><body><h1>503 Service Unavailable</h1>
No server is available to handle this request.
</body></html>
 TIME: 16:48:33,562
<html><body><h1>503 Service Unavailable</h1>
No server is available to handle this request.
</body></html>
 TIME: 16:48:34,601
 ...
example-spring-boot-3-tjdkj TIME: 16:49:20,114
example-spring-boot-3-tjdkj TIME: 16:49:21,181
example-spring-boot-3-tjdkj TIME: 16:49:22,231
```

It is even possible that the Service gets down, and the routing layer responds with the status code 503 as can be seen in the example output above.

In the following chapter we are going to look at how a Service can be configured to be highly available.

Uninterruptible Deployments

The rolling update strategy makes it possible to deploy Pods without interruption. The rolling update strategy means that the new version of an application gets deployed and started. As soon as the application says it is ready, OpenShift forwards requests to the new instead of the old version of the Pod, and the old Pod gets terminated.

Additionally, container health checks help OpenShift to precisely determine what state the application is in.

Basically, there are two different kinds of checks that can be implemented:

- Liveness probes are used to find out if an application is still running
- Readiness probes tell us if the application is ready to receive requests (which is especially relevant for the above-mentioned rolling updates)

These probes can be implemented as HTTP checks, container execution checks (the execution of a command or script inside a container) or TCP socket checks.

In our example, we want the application to tell OpenShift that it is ready for requests with an appropriate readiness probe.

Our example application has a health check context named health: `http://localhost:9000/health`. This port is not exposed by a service. It is only accessible inside the cluster.

Task 5.2: Availability during deployment

Define the readiness probe on the Deployment using the following command:

```
oc set probe deploy/example-web-app --readiness --get-url=http://:9000/health --initial-delay-seconds=10 --timeout-seco
nds=1 --namespace <namespace>
```

The command above results in the following `readinessProbe` snippet being inserted into the Deployment:

```
...
containers:
  - image: quay.io/appuio/example-spring-boot:latest
    imagePullPolicy: Always
    name: example-web-app
    readinessProbe:
      httpGet:
        path: /health
        port: 9000
        scheme: HTTP
      initialDelaySeconds: 10
      timeoutSeconds: 1
...
```

We are now going to verify that a redeployment of the application does not lead to an interruption.

Set up the loop again to periodically check the application's response (you don't have to set the `$URL` variable again if it is still defined):

```
URL=$(oc get routes example-web-app -o go-template="{{ .spec.host }}" --namespace <namespace>)
while true; do sleep 1; curl -s https://${URL}/pod/; date "+ TIME: %H:%M:%S,%3N"; done
```

Windows PowerShell:

```
while(1) {
  Start-Sleep -s 1
  Invoke-RestMethod https://<URL>/pod/
  Get-Date -Uformat "+ TIME: %H:%M:%S,%3N"
}
```

Restart your Deployment with:

```
oc rollout restart deployment example-web-app --namespace <namespace>
```

Self-healing

Via the Deployment definition we told OpenShift how many replicas we want. So what happens if we simply delete a Pod?

Look for a running Pod (status `RUNNING`) that you can bear to kill via `oc get pods`.

Show all Pods and watch for changes:

```
oc get pods -w --namespace <namespace>
```

Now delete a Pod (in another terminal) with the following command:

```
oc delete pod <pod> --namespace <namespace>
```

Observe how OpenShift instantly creates a new Pod in order to fulfill the desired number of running instances.

- acend gmbh

6. Troubleshooting

This lab helps you troubleshoot your application and shows you some tools to make troubleshooting easier.

Logging into a container

Running containers should be treated as immutable infrastructure and should therefore not be modified. However, there are some use cases in which you have to log into your running container. Debugging and analyzing is one example for this.

Task 6.1: Shell into Pod

With OpenShift you can open a remote shell into a Pod without installing SSH by using the command `oc rsh`. The command can also be used to execute any command in a Pod.

> **Note**
> If you're using Git Bash on Windows, you need to append the command with `winpty`.

Choose a Pod with `oc get pods --namespace <namespace>` and execute the following command:

```
oc rsh --namespace <namespace> <pod>
```

You now have a running shell session inside the container in which you can execute every binary available, e.g.:

```
pwd
```

```
/home/default
```

With `exit` or `CTRL+d` you can leave the container and close the connection:

```
exit
```

Task 6.2: Single commands

Single commands inside a container can also be executed with `oc rsh`:

```
oc rsh --namespace <namespace> <pod> <command>
```

Example:

```
oc rsh --namespace acend-test example-web-app-8b465c687-t9g7b env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
HOSTNAME=example-web-app-8b465c687-t9g7b
NSS_SDB_USE_CACHE=no
KUBERNETES_PORT_443_TCP=tcp://172.30.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
EXAMPLE_WEB_APP_PORT_5000_TCP_PORT=5000
...
```

The debug command

One of the disadvantages of using the `oc rsh` command is that it depends on the container to actually run. If the Pod can't even start, this is a problem but also where the `oc debug` command comes in. The `oc debug` command starts an interactive shell using the definition of a Deployment, Pod, DaemonSet, Job or even an ImageStreamTag. In OpenShift 4 it can also be used to open a shell on a Node to analyze it.

The quick way of using it is `oc debug RESOURCE/NAME` but have a good look at its help page. There are some very interesting parameters like `--as-root` that give you (depending on your permissions on the cluster) a very powerful means of debugging a Pod.

Watching log files

Log files of a Pod can be shown with the following command:

```
oc logs <pod> --namespace <namespace>
```

The parameter `-f` allows you to follow the log file (same as `tail -f`). With this, log files are streamed and new entries are shown immediately.

When a Pod is in state `CrashLoopBackOff` it means that although multiple attempts have been made, no container inside the Pod could be started successfully. Now even though no container might be running at the moment the `oc logs` command is executed, there is a way to view the logs the application might have generated. This is achieved using the `-p` or `--previous` parameter.

> **Note**
> This command will only work on pods that had container restarts. You can check the `RESTARTS` column in the `oc get pods` output if this is the case.

```
oc logs -p <pod> --namespace <namespace>
```

Task 6.3: Port forwarding

OpenShift allows you to forward arbitrary ports to your development workstation. This allows you to access admin consoles, databases, etc., even when they are not exposed externally. Port forwarding is handled by the OpenShift control plane nodes and therefore tunneled from the client via HTTPS. This allows you to access the OpenShift platform even when there are restrictive firewalls or proxies between your workstation and OpenShift.

Get the name of the Pod:

```
oc get pod --namespace <namespace>
```

Then execute the port forwarding command using the Pod's name:

> **Note**
> Best run this command in a separate shell, or in the background by adding a "&" at the end of the command.

```
oc port-forward <pod> 9000:9000 --namespace <namespace>
```

Don't forget to change the Pod name to your own installation. If configured, you can use auto-completion.

The output of the command should look like this:

```
Forwarding from 127.0.0.1:9000 -> 9000
Forwarding from [::1]:9000 -> 9000
```

> **Note**
> Use the additional parameter `--address <IP address>` (where `<IP address>` refers to a NIC's IP address from your local workstation) if you want to access the forwarded port from outside your own local workstation.

Now the health endpoint is available at: http://localhost:9000/ .

We could not access this endpoint before because it is only exposed inside the cluster.

The application probe endpoint is now available with the following link: http://localhost:9000/health . Or try a `curl` command:

```
curl localhost:9000/health
```

With the same concept you can access databases from your local workstation or connect your local development environment via remote debugging to your application in the Pod.

This documentation page offers some more details about port forwarding.

> **Note**
> The `oc port-forward` process runs as long as it is not terminated by the user. So when done, stop it with `CTRL-c`.

Events

OpenShift maintains an event log with high-level information on what's going on in the cluster. It's possible that everything looks okay at first but somehow something seems stuck. Make sure to have a look at the

events because they can give you more information if something is not working as expected.

Use the following command to list the events in chronological order:

```
oc get events --sort-by=.metadata.creationTimestamp --namespace <namespace>
```

Dry-run

To help verify changes, you can use the optional `oc` flag `--dry-run=client -o yaml` to see the rendered YAML definition of your Kubernetes objects, without sending it to the API.

The following `oc` subcommands support this flag (non-final list):

- `apply`
- `create`
- `expose`
- `patch`
- `replace`
- `run`
- `set`

For example, we can use the `--dry-run=client` flag to create a template for our Deployment:

```
oc create deployment example-web-app --image=quay.io/appuio/example-spring-boot:latest --namespace acend-test --dry-run
=client -o yaml
```

The result is the following YAML output:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: example-web-app
  name: example-web-app
  namespace: acend-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-web-app
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: example-web-app
    spec:
      containers:
        - image: quay.io/appuio/example-spring-boot:latest
          name: example-web
          resources: {}
status: {}
```

`oc` API requests

If you want to see the HTTP requests `oc` sends to the Kubernetes API in detail, you can use the optional flag `--v=10`.

For example, to see the API request for creating a deployment:

```
oc create deployment test-deployment --image=quay.io/appuio/example-spring-boot:latest --namespace <namespace> --replic
as=0 --v=10
```

The resulting output looks like this:

```
I1114 15:31:13.605759   85289 request.go:1073] Request Body: {"kind":"Deployment","apiVersion":"apps/v1","metadata":{"n
ame":"test-deployment","namespace":"acend-test","creationTimestamp":null,"labels":{"app":"test-deployment"}},"spec":{"r
eplicas":0,"selector":{"matchLabels":{"app":"test-deployment"}},"template":{"metadata":{"creationTimestamp":null,"label
s":{"app":"test-deployment"}},"spec":{"containers":[{"name":"example-web","image":"quay.io/appuio/example-spring-boot:l
atest","resources":{}}]}},"strategy":{}},"status":{}}
I1114 15:31:13.605817   85289 round_trippers.go:466] curl -v -XPOST  -H "Accept: application/json, */*" -H "Content-Typ
e: application/json" -H "User-Agent: oc/4.11.0 (linux/amd64) kubernetes/262ac9c" -H "Authorization: Bearer <masked>" 'h
ttps://api.ocp-staging.cloudscale.puzzle.ch:6443/apis/apps/v1/namespaces/acend-test/deployments?fieldManager=kubectl-cr
eate&fieldValidation=Ignore'
I1114 15:31:13.607320   85289 round_trippers.go:495] HTTP Trace: DNS Lookup for api.ocp-staging.cloudscale.puzzle.ch re
solved to [{5.102.150.82 }]
I1114 15:31:13.611279   85289 round_trippers.go:510] HTTP Trace: Dial to tcp:5.102.150.82:6443 succeed
I1114 15:31:13.675096   85289 round_trippers.go:553] POST https://api.ocp-staging.cloudscale.puzzle.ch:6443/apis/apps/v
1/namespaces/acend-test/deployments?fieldManager=kubectl-create&fieldValidation=Ignore 201 Created in 69 milliseconds
I1114 15:31:13.675120   85289 round_trippers.go:570] HTTP Statistics: DNSLookup 1 ms Dial 3 ms TLSHandshake 35 ms Serve
rProcessing 27 ms Duration 69 ms
I1114 15:31:13.675137   85289 round_trippers.go:577] Response Headers:
I1114 15:31:13.675151   85289 round_trippers.go:580]     Audit-Id: 509255b1-ee23-479a-be56-dfc3ab073864
I1114 15:31:13.675164   85289 round_trippers.go:580]     Cache-Control: no-cache, private
I1114 15:31:13.675181   85289 round_trippers.go:580]     Content-Type: application/json
I1114 15:31:13.675200   85289 round_trippers.go:580]     X-Kubernetes-Pf-Flowschema-Uid: e3e152ee-768c-43c5-b350-bb3cbf
806147
I1114 15:31:13.675215   85289 round_trippers.go:580]     X-Kubernetes-Pf-Prioritylevel-Uid: 47f392da-68d1-4e43-9d77-ff5
f7b7ecd2e
I1114 15:31:13.675230   85289 round_trippers.go:580]     Content-Length: 1739
I1114 15:31:13.675244   85289 round_trippers.go:580]     Date: Mon, 14 Nov 2022 14:31:13 GMT
I1114 15:31:13.676116   85289 request.go:1073] Response Body: {"kind":"Deployment","apiVersion":"apps/v1","metadata":{"
name":"test-deployment","namespace":"acend-test","uid":"a6985d28-3caa-451f-a648-4c7cde3b51ac","resourceVersion":"206938
5577","generation":1,"creationTimestamp":"2022-11-14T14:31:13Z","labels":{"app":"test-deployment"},"managedFields":[{"m
anager":"kubectl-create","operation":"Update","apiVersion":"apps/v1","time":"2022-11-14T14:31:13Z","fieldsType":"Fields
V1","fieldsV1":{"f:metadata":{"f:labels":{".":{},"f:app":{}}},"f:spec":{"f:progressDeadlineSeconds":{},"f:replicas":{},
"f:revisionHistoryLimit":{},"f:selector":{},"f:strategy":{"f:rollingUpdate":{".":{},"f:maxSurge":{},"f:maxUnavailable":
{}},"f:type":{}},"f:template":{"f:metadata":{"f:labels":{".":{},"f:app":{}}},"f:spec":{"f:containers":{"k:{\"name\":\"e
xample-web\"}":{".":{},"f:image":{},"f:imagePullPolicy":{},"f:name":{},"f:resources":{},"f:terminationMessagePath":{},"
f:terminationMessagePolicy":{}}},"f:dnsPolicy":{},"f:restartPolicy":{},"f:schedulerName":{},"f:securityContext":{},"f:t
erminationGracePeriodSeconds":{}}}}}]},"spec":{"replicas":0,"selector":{"matchLabels":{"app":"test-deployment"}},"temp
late":{"metadata":{"creationTimestamp":null,"labels":{"app":"test-deployment"}},"spec":{"containers":[{"name":"example-
web","image":"quay.io/appuio/example-spring-boot:latest","resources":{},"terminationMessagePath":"/dev/termination-log"
,"terminationMessagePolicy":"File","imagePullPolicy":"Always"}],"restartPolicy":"Always","terminationGracePeriodSeconds
":30,"dnsPolicy":"ClusterFirst","securityContext":{},"schedulerName":"default-scheduler"}},"strategy":{"type":"RollingU
pdate","rollingUpdate":{"maxUnavailable":"25%","maxSurge":"25%"}},"revisionHistoryLimit":10,"progressDeadlineSeconds":6
00},"status":{}}
deployment.apps/test-deployment created
```

As you can see, the output conveniently contains the corresponding `curl` commands which we could use in our own code, tools, pipelines etc.

> **Note**
> If you created the deployment to see the output, you can delete it again as it’s not used anywhere else

(which is also the reason why the replicas are set to `0`):

```
oc delete deploy/test-deployment --namespace <namespace>
```

7. Attaching a database

Numerous applications are stateful in some way and want to save data persistently, be it in a database, as files on a filesystem or in an object store. In this lab, we are going to create a MariaDB database and configure our application to store its data in it.

> **Warning**
> Please make sure you completed labs *2. First steps* and *5. Scaling* before you continue with this lab.

Task 7.1: Instantiate a MariaDB database

We are going to use an OpenShift template to create the database. This can be done by using the CLI.

We are going to instantiate the MariaDB Template from the `openshift` Project. Before we can do that, we need to know what parameters the Template expects. Let's find out:

```
oc process --parameters openshift//mariadb-ephemeral
```

```
NAME                     DESCRIPTION                                                 GENERATOR           V
ALUE
MEMORY_LIMIT             Maximum amount of memory the container can use.                                 5
12Mi
NAMESPACE                The OpenShift Namespace where the ImageStream resides.                          o
penshift
DATABASE_SERVICE_NAME    The name of the OpenShift Service exposed for the database.                     m
ariadb
MYSQL_USER               Username for MariaDB user that will be used for accessing the database.  expression          u
ser[A-Z0-9]{3}
MYSQL_PASSWORD           Password for the MariaDB connection user.                        expression          [
a-zA-Z0-9]{16}
MYSQL_ROOT_PASSWORD      Password for the MariaDB root user.                              expression          [
a-zA-Z0-9]{16}
MYSQL_DATABASE           Name of the MariaDB database accessed.                                          s
ampledb
MARIADB_VERSION          Version of MariaDB image to be used (10.2 or latest).                           1
0.2
```

As you might already see, each of the parameters has a default value ("VALUE" column). Also, the parameters `MYSQL_USER`, `MYSQL_PASSWORD` and `MYSQL_ROOT_PASSWORD` are going to be generated ("GENERATOR" is set to `expression` and "VALUE" contains a regular expression). This means we don't necessarily have to overwrite any of them so let's simply use those defaults:

```
oc process openshift//mariadb-ephemeral -pMYSQL_DATABASE=acend_exampledb  | oc apply --namespace=<namespace> -f -
```

The output should be:

```
secret/mariadb created
service/mariadb created
deploymentconfig.apps.openshift.io/mariadb created
```

Task 7.2: Inspection

What just happened is that you instantiated an OpenShift Template that creates multiple resources using the (default) values as parameters. Let's have a look at the resources that have just been created by looking at the Template's definition:

```
oc get templates -n openshift mariadb-ephemeral -o yaml
```

The Template's content reveals a Secret, a Service and a DeploymentConfig.

The Secret contains the database name, user, password, and the root password. However, these values will neither be shown with `oc get` nor with `oc describe` :

```
oc get secret mariadb --output yaml --namespace <namespace>
```

```
apiVersion: v1
data:
  database-name: YWNlbmQtZXhhbXBsZS1kYg==
  database-password: bXlzcWxwYXNzd29yZA==
  database-root-password: bXlzcWxyb290cGFzc3dvcmQ=
  database-user: YWNlbmRfdXNlcg==
kind: Secret
metadata:
  ...
type: Opaque
```

The reason is that all the values in the `.data` section are base64 encoded. Even though we cannot see the true values, they can easily be decoded:

```
echo "YWNlbmQtZXhhbXBsZS1kYg==" | base64 -d
```

> **Note**
> There's also the `oc extract` command which can be used to extract the content of Secrets and ConfigMaps into a local directory. Use `oc extract --help` to see how it works.

> **Note**
> By default, Secrets are not encrypted!
>
> However, both OpenShift and Kubernetes (1.13 and later) offer the capability to encrypt data in etcd.
>
> Another option would be the use of a secrets management solution like Vault by HashiCorp .

The interesting thing about Secrets is that they can be reused, e.g., in different Deployments. We could extract all the plaintext values from the Secret and put them as environment variables into the Deployments, but it's way easier to instead simply refer to its values inside the Deployment (as in this lab) like this:

```
...
spec:
  template:
    spec:
      containers:
      - name: mariadb
        env:
        - name: MYSQL_USER
          valueFrom:
            secretKeyRef:
              key: database-user
              name: mariadb
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              key: database-password
              name: mariadb
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              key: database-root-password
              name: mariadb
        - name: MYSQL_DATABASE
          valueFrom:
            secretKeyRef:
              key: database-name
              name: mariadb
...
```

Above lines are an excerpt of the MariaDB Deployment. Most parts have been cut out to focus on the relevant lines: The references to the `mariadb` Secret. As you can see, instead of directly defining environment variables you can refer to a specific key inside a Secret. We are going to make further use of this concept for our Python application.

Task 7.3: Attach the database to the application

By default, our `example-web-app` application uses an SQLite memory database.

However, this can be changed by defining the following environment variable to use the newly created MariaDB database:

```
#SPRING_DATASOURCE_URL=jdbc:mysql://<host>/<database>
SPRING_DATASOURCE_URL=jdbc:mysql://mariadb/acend_exampledb
```

The connection string our `example-web-app` application uses to connect to our new MariaDB, is a concatenated string from the values of the `mariadb` Secret.

For the actual MariaDB host, you can either use the MariaDB Service's ClusterIP or DNS name as the address. All Services and Pods can be resolved by DNS using their name.

Add the environment variables by directly editing the Deployment:

```
oc edit deployment example-web-app --namespace <namespace>
```

```
      ...
      containers:
      - image: quay.io/appuio/example-spring-boot:latest
        imagePullPolicy: Always
        name: example-web-app
        ...
        env:
        - name: SPRING_DATASOURCE_DATABASE_NAME
          valueFrom:
            secretKeyRef:
              key: database-name
              name: mariadb
        - name: SPRING_DATASOURCE_USERNAME
          valueFrom:
            secretKeyRef:
              key: database-user
              name: mariadb
        - name: SPRING_DATASOURCE_PASSWORD
          valueFrom:
            secretKeyRef:
              key: database-password
              name: mariadb
        - name: SPRING_DATASOURCE_DRIVER_CLASS_NAME
          value: com.mysql.cj.jdbc.Driver
        - name: SPRING_DATASOURCE_URL
          value: jdbc:mysql://mariadb/$(SPRING_DATASOURCE_DATABASE_NAME)?autoReconnect=true
        ...
```

The environment can also be checked with the `set env` command and the `--list` parameter:

```
oc set env deploy/example-web-app --list --namespace <namespace>
```

This will show the environment as follows:

```
# deployments/example-web-app, container example-web-app
# SPRING_DATASOURCE_DATABASE_NAME from secret mariadb, key database-name
# SPRING_DATASOURCE_USERNAME from secret mariadb, key database-user
# SPRING_DATASOURCE_PASSWORD from secret mariadb, key database-password
SPRING_DATASOURCE_DRIVER_CLASS_NAME=com.mysql.cj.jdbc.Driver
SPRING_DATASOURCE_URL=jdbc:mysql://mariadb/$(SPRING_DATASOURCE_DATABASE_NAME)?autoReconnect=true
```

Warning

Do not proceed with the lab before all example-web-app pods are restarted successfully.

The change of the deployment definition (environment change) triggers a new rollout and all example-web-app pods will be restarted. The application will not be connected to the database until all pods are restarted successfully.

In order to find out if the change worked we can either look at the container's logs (`oc logs <pod>`) or we could register some "Hellos" in the application, delete the Pod, wait for the new Pod to be started and check if they are still there.

Note

This does not work if we delete the database Pod as its data is not yet persisted.

Task 7.4: Manual database connection

As described in *6. Troubleshooting* we can log into a Pod with `oc rsh <pod>` .

Show all Pods:

```
oc get pods --namespace <namespace>
```

Which gives you an output similar to this:

```
NAME                                READY   STATUS      RESTARTS   AGE
example-web-app-574544fd68-qfkcm    1/1     Running     0          2m20s
mariadb-f845ccdb7-hf2x5             1/1     Running     0          31m
mariadb-1-deploy                    0/1     Completed   0          11m
```

Log into the MariaDB Pod:

> **Note**
> As mentioned in *6. Troubleshooting*, remember to append the command with `winpty` if you're using Git Bash on Windows.

```
oc rsh --namespace <namespace> <mariadb-pod-name>
```

You are now able to connect to the database and display the data. Login with:

```
mysql -u$MYSQL_USER -p$MYSQL_PASSWORD -h$MARIADB_SERVICE_HOST $MYSQL_DATABASE
```

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 52810
Server version: 10.2.22-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [acend_exampledb]>
```

Show all tables with:

```
show tables;
```

Show any entered "Hellos" with:

```
select * from hello;
```

Task 7.5: Import a database dump

Our task is now to import this dump.sql into the MariaDB database running as a Pod. Use the `mysql` command line utility to do this. Make sure the database is empty beforehand. You could also delete and recreate the database.

> **Note**
> You can also copy local files into a Pod using `oc cp`. Be aware that the `tar` binary has to be present inside the container and on your operating system in order for this to work! Install `tar` on UNIX systems with e.g. your package manager, on Windows there's e.g. cwRsync . If you cannot install `tar` on your host, there's also the possibility of logging into the Pod and using `curl -O <url>`.

Solution

This is how you copy the database dump into the MariaDB Pod.

Download the dump.sql or get it with curl:

```
curl -O https://raw.githubusercontent.com/acend/kubernetes-basics-training/main/content/en/docs/attaching-a-database/dump.sql
```

Copy the dump into the MariaDB Pod:

```
oc cp ./dump.sql <podname>:/tmp/ --namespace <namespace>
```

This is how you log into the MariaDB Pod:

```
oc rsh --namespace <namespace> <podname>
```

This command shows how to drop the whole database:

```
mysql -u$MYSQL_USER -p$MYSQL_PASSWORD -h$MARIADB_SERVICE_HOST $MYSQL_DATABASE
```

```
drop database `acend_exampledb`;
create database `acend_exampledb`;
exit
```

Import a dump:

```
mysql -u$MYSQL_USER -p$MYSQL_PASSWORD -h$MARIADB_SERVICE_HOST $MYSQL_DATABASE < /tmp/dump.sql
```

Check your app to see the imported “Hellos”.

Note

You can find your app URL by looking at your route:

```
oc get route --namespace <namespace>
```

Note

A database dump can be created as follows:

```
oc rsh --namespace <namespace> <podname>
```

```
mysqldump --user=$MYSQL_USER --password=$MYSQL_PASSWORD -h$MARIADB_SERVICE_HOST $MYSQL_DATABASE > /tmp/dump.sql
```

```
oc cp <podname>:/tmp/dump.sql /tmp/dump.sql
```

8. Persistent storage

By default, data in containers is not persistent as was the case e.g. in *7. Attaching a database*. This means that the data written in a container is lost as soon as it does not exist anymore. We want to prevent this from happening. One possible solution to this problem is to use persistent storage.

Request storage

Attaching persistent storage to a Pod happens in two steps. The first step includes the creation of a so-called *PersistentVolumeClaim* (PVC) in our namespace. This claim defines amongst other things what size we would like to get.

The PersistentVolumeClaim only represents a request but not the storage itself. It is automatically going to be bound to a *PersistentVolume* by OpenShift, one that has at least the requested size. If only volumes exist that have a bigger size than was requested, one of these volumes is going to be used. The claim will automatically be updated with the new size. If there are only smaller volumes available, the claim cannot be fulfilled as long as no volume with the exact same or larger size is created.

Attaching a volume to a Pod

In a second step, the PVC from before is going to be attached to the Pod. In *5. Scaling* we used `oc set` to add a readiness probe to the Deployment. We are now going to do the same and insert the PersistentVolume.

Task 8.1: Add a PersistentVolume

The `oc set volume` command makes it possible to create a PVC and attach it to a Deployment in one fell swoop:

> **Note**
> If you are using Windows, your shell might assume that it has to use the POSIX-to-Windows path conversion for the mount path `/var/lib/mysql` . PowerShell is known to not do this while, e.g., Git Bash does.
>
> Prepend your command with `MSYS_NO_PATHCONV=1` if the resulting mount path was mistakenly converted.

```
oc set volume dc/mariadb --add --name=mariadb-data --claim-name=mariadb-data --type persistentVolumeClaim --mount-path=
/var/lib/mysql --claim-size=1G --overwrite --namespace <namespace>
```

With the instruction above we create a PVC named `mariadb-data` of 1Gi in size, attach it to the DeploymentConfig `mariadb` and mount it at `/var/lib/mysql` . This is where the MariaDB process writes its data by default so after we make this change, the database will not even notice that it is writing in a PersistentVolume.

> **Note**
> Because we just changed the DeploymentConfig with the `oc set` command, a new Pod was automatically redeployed. This unfortunately also means that we just lost the data we inserted before.

We need to redeploy the application pod, our application automatically creates the database schema at

startup time. Wait for the database pod to be started fully before restarting the application pod.

If you want to force a redeployment of a Pod, you can use this:

```
oc rollout restart deployment example-web-app --namespace <namespace>
```

Using the command `oc get persistentvolumeclaim` or `oc get pvc` , we can display the freshly created PersistentVolumeClaim:

```
oc get pvc --namespace <namespace>
```

Which gives you an output similar to this:

```
NAME            STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mariadb-data    Bound    pvc-2cb78deb-d157-11e8-a406-42010a840034   1Gi        RWO            standard       11s
```

The two columns `STATUS` and `VOLUME` show us that our claim has been bound to the PersistentVolume `pvc-2cb78deb-d157-11e8-a406-42010a840034` .

Error case

If the container is not able to start it is the right moment to debug it! Check the logs from the container and search for the error.

```
oc logs mariadb-f845ccdb7-hf2x5 --namespace <namespace>
```

> **Note**
> If the container won't start because the data directory already has files in it, use the `oc debug` command mentioned in *7. Attaching a database* to check its content and remove it if necessary.

Task 8.2: Persistence check

Restore data

Repeat the task to import a database dump .

Test

Scale your MariaDB Pod to 0 replicas and back to 1. Observe that the new Pod didn't loose any data.

9. Additional concepts

OpenShift does not only know Pods, Deployments, Services, etc. There are various other kinds of resources. In the next few labs, we are going to have a look at some of them.

9.1. ConfigMaps

Similar to environment variables, *ConfigMaps* allow you to separate the configuration for an application from the image. Pods can access those variables at runtime which allows maximum portability for applications running in containers. In this lab, you will learn how to create and use ConfigMaps.

ConfigMap creation

A ConfigMap can be created using the `oc create configmap` command as follows:

```
oc create configmap <name> <data-source> --namespace <namespace>
```

Where the `<data-source>` can be a file, directory, or command line input.

Task 9.1.1: Java properties as ConfigMap

A classic example for ConfigMaps are properties files of Java applications which can't be configured with environment variables.

First, create a file called `java.properties` with the following content:

```
key=value
key2=value2
```

Now you can create a ConfigMap based on that file:

```
oc create configmap javaconfiguration --from-file=./java.properties --namespace <namespace>
```

Verify that the ConfigMap was created successfully:

```
oc get configmaps --namespace <namespace>
```

```
NAME                DATA   AGE
javaconfiguration   1      7s
```

Have a look at its content:

```
oc get configmap javaconfiguration -o yaml --namespace <namespace>
```

Which should yield output similar to this one:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: javaconfiguration
data:
  java.properties: |
    key=value
    key2=value2
```

Task 9.1.2: Attach the ConfigMap to a container

Next, we want to make a ConfigMap accessible for a container. There are basically the following possibilities to achieve this :

- ConfigMap properties as environment variables in a Deployment
- Command line arguments via environment variables
- Mounted as volumes in the container

In this example, we want the file to be mounted as a volume inside the container.

As in *8. Persistent storage*, we can use the `oc set volume` command to achieve this:

> **Note**
> If you are using Windows and your shell uses the POSIX-to-Windows path conversion, remember to prepend your command with `MSYS_NO_PATHCONV=1` if the resulting mount path was mistakenly converted.

```
oc set volume deploy/example-web-app --add --configmap-name=javaconfiguration --mount-path=/etc/config --name=config-vo
lume --type configmap --namespace <namespace>
```

> **Note**
> This task doesn't have any effect on the example application inside the container. It is for demonstration purposes only.

This results in the addition of the following parts to the Deployment (check with `oc get deploy example-web-app -o yaml`):

```
...
  volumeMounts:
  - mountPath: /etc/config
    name: config-volume
...
volumes:
- configMap:
    defaultMode: 420
    name: javaconfiguration
  name: config-volume
...
```

This means that the container should now be able to access the ConfigMap's content in `/etc/config/java.properties` . Let's check:

```
oc exec <pod> --namespace <namespace> -- cat /etc/config/java.properties
```

> **Note**
> On Windows, you can use Git Bash with `winpty oc exec -it <pod> --namespace <namespace> -- cat //etc/config/java.properties`.

```
key=value
key2=value2
```

Like this, the property file can be read and used by the application inside the container. The image stays portable to other environments.

Task 9.1.3: ConfigMap environment variables

Use a ConfigMap by populating environment variables into the container instead of a file.

9.2. ResourceQuotas and LimitRanges

In this lab, we are going to look at ResourceQuotas and LimitRanges. As OpenShift users, we are most certainly going to encounter the limiting effects that ResourceQuotas and LimitRanges impose.

> **Warning**
> For this lab to work it is vital that you use the namespace `<username>-quota` !

ResourceQuotas

ResourceQuotas among other things limit the amount of resources Pods can use in a Namespace. They can also be used to limit the total number of a certain resource type in a Project. In more detail, there are these kinds of quotas:

- *Compute ResourceQuotas* can be used to limit the amount of memory and CPU
- *Storage ResourceQuotas* can be used to limit the total amount of storage and the number of PersistentVolumeClaims, generally or specific to a StorageClass
- *Object count quotas* can be used to limit the number of a certain resource type such as Services, Pods or Secrets

Defining ResourceQuotas makes sense when the cluster administrators want to have better control over consumed resources. A typical use case are public offerings where users pay for a certain guaranteed amount of resources which must not be exceeded.

In order to check for defined quotas in your Namespace, simply see if there are any of type ResourceQuota:

```
oc get resourcequota --namespace <namespace>-quota
```

To show in detail what kinds of limits the quota imposes:

```
oc describe resourcequota <quota-name> --namespace <namespace>-quota
```

For more details, have look into OpenShift's documentation about resource quotas .

Requests and limits

As we've already seen, compute ResourceQuotas limit the amount of memory and CPU we can use in a Project. Only defining a ResourceQuota, however is not going to have an effect on Pods that don't define the amount of resources they want to use. This is where the concept of limits and requests comes into play.

Limits and requests on a Pod, or rather on a container in a Pod, define how much memory and CPU this container wants to consume at least (request) and at most (limit). Requests mean that the container will be guaranteed to get at least this amount of resources, limits represent the upper boundary which cannot be crossed. Defining these values helps OpenShift in determining on which Node to schedule the Pod because it knows how many resources should be available for it.

> **Note**

Containers using more CPU time than what their limit allows will be throttled. Containers using more memory than what they are allowed to use will be killed.

Defining limits and requests on a Pod that has one container looks like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: lr-demo
  namespace: lr-example
spec:
  containers:
  - name: lr-demo-ctr
    image: docker.io/nginxinc/nginx-unprivileged:latest
    resources:
      limits:
        memory: "200Mi"
        cpu: "700m"
      requests:
        memory: "200Mi"
        cpu: "700m"
```

You can see the familiar binary unit “Mi” is used for the memory value. Other binary (“Gi”, “Ki”, ...) or decimal units (“M”, “G”, “K”, ...) can be used as well.

The CPU value is denoted as “m”. “m” stands for *millicpu* or sometimes also referred to as *millicores* where `"1000m"` is equal to one core/vCPU/hyperthread.

Quality of service

Setting limits and requests on containers has yet another effect: It might change the Pod’s *Quality of Service* class. There are three such *QoS* classes:

- *Guaranteed*
- *Burstable*
- *BestEffort*

The Guaranteed QoS class is applied to Pods that define both limits and requests for both memory and CPU resources on all their containers. The most important part is that each request has the same value as the limit. Pods that belong to this QoS class will never be killed by the scheduler because of resources running out on a Node.

Note

If a container only defines its limits, OpenShift automatically assigns a request that matches the limit.

The Burstable QoS class means that limits and requests on a container are set, but they are different. It is enough to define limits and requests on one container of a Pod even though there might be more, and it also only has to define limits and requests on memory or CPU, not necessarily both.

The BestEffort QoS class applies to Pods that do not define any limits and requests at all on any containers. As its class name suggests, these are the kinds of Pods that will be killed by the scheduler first if a Node runs out of memory or CPU. As you might have already guessed by now, if there are no BestEffort QoS Pods, the scheduler will begin to kill Pods belonging to the class of *Burstable*. A Node hosting only Pods of class Guaranteed will (theoretically) never run out of resources.

LimitRanges

As you now know what limits and requests are, we can come back to the statement made above:

> As we've already seen, compute ResourceQuotas limit the amount of memory and CPU we can use in a Namespace. Only defining a ResourceQuota, however is not going to have an effect on Pods that don't define the amount of resources they want to use. This is where the concept of limits and requests comes into play.

So, if a cluster administrator wanted to make sure that every Pod in the cluster counted against the compute ResourceQuota, the administrator would have to have a way of defining some kind of default limits and requests that were applied if none were defined in the containers. This is exactly what *LimitRanges* are for.

Quoting the Kubernetes documentation , LimitRanges can be used to:

- Enforce minimum and maximum compute resource usage per Pod or container in a Namespace
- Enforce minimum and maximum storage requests per PersistentVolumeClaim in a Namespace
- Enforce a ratio between request and limit for a resource in a Namespace
- Set default request/limit for compute resources in a Namespace and automatically inject them to containers at runtime

If for example a container did not define any requests or limits and there was a LimitRange defining the default values, these default values would be used when deploying said container. However, as soon as limits or requests were defined, the default values would no longer be applied.

The possibility of enforcing minimum and maximum resources and defining ResourceQuotas per Namespace allows for many combinations of resource control.

Task 9.2.1: Namespace

> **Warning**
> Remember to use the namespace `<username>-quota` , otherwise this lab will not work!

Analyse the LimitRange in your Namespace (there has to be one, if not you are using the wrong Namespace):

```
oc describe limitrange --namespace <namespace>-quota
```

The command above should output this (name and Namespace will vary):

```
Name:       ce01a1b6-a162-479d-847c-4821255cc6db
Namespace:  eltony-quota-lab
Type        Resource  Min  Max  Default Request  Default Limit  Max Limit/Request Ratio
----        --------  ---  ---  ---------------  -------------  -----------------------
Container   memory    -    -    16Mi             32Mi           -
Container   cpu       -    -    10m              100m           -
```

Check for the ResourceQuota in your Namespace (there has to be one, if not you are using the wrong Namespace):

```
oc describe quota --namespace <namespace>-quota
```

The command above will produce an output similar to the following (name and namespace may vary)

```
Name:            lab-quota
Namespace:       eltony-quota-lab
Resource         Used  Hard
--------         ----  ----
requests.cpu     0     100m
requests.memory  0     100Mi
```

Task 9.2.2: Default memory limit

Create a Pod using the stress image:

```
apiVersion: v1
kind: Pod
metadata:
  name: stress2much
spec:
  containers:
    - command:
        - stress
        - --vm
        - "1"
        - --vm-bytes
        - 85M
        - --vm-hang
        - "1"
      image: quay.io/acend/stress:latest
      imagePullPolicy: Always
      name: stress
```

Apply this resource with:

```
oc apply -f pod_stress2much.yaml --namespace <namespace>-quota
```

> **Note**
> You have to actively terminate the following command pressing `CTRL+c` on your keyboard.

Watch the Pod's creation with:

```
oc get pods --watch --namespace <namespace>-quota
```

You should see something like the following:

```
NAME          READY   STATUS              RESTARTS   AGE
stress2much   0/1     ContainerCreating   0          1s
stress2much   0/1     ContainerCreating   0          2s
stress2much   0/1     OOMKilled           0          5s
stress2much   1/1     Running             1          7s
stress2much   0/1     OOMKilled           1          9s
stress2much   0/1     CrashLoopBackOff    1          20s
```

The `stress2much` Pod was OOM (out of memory) killed. We can see this in the `STATUS` field. Another way to find out why a Pod was killed is by checking its status. Output the Pod's YAML definition:

```
oc get pod stress2much --output yaml --namespace <namespace>-quota
```

Near the end of the output you can find the relevant status part:

```
  containerStatuses:
  - containerID: docker://da2473f1c8ccdffbb824d03689e9fe738ed689853e9c2643c37f206d10f93a73
    image: quay.io/acend/stress:latest
    lastState:
      terminated:
        ...
        reason: OOMKilled
        ...
```

So let's look at the numbers to verify the container really had too little memory. We started the `stress` command using the parameter `--vm-bytes 85M` which means the process wants to allocate 85 megabytes of memory. Again looking at the Pod's YAML definition with:

```
oc get pod stress2much --output yaml --namespace <namespace>-quota
```

reveals the following values:

```
...
    resources:
      limits:
        cpu: 100m
        memory: 32Mi
      requests:
        cpu: 10m
        memory: 16Mi
...
```

These are the values from the LimitRange, and the defined limit of 32 MiB of memory prevents the `stress` process of ever allocating the desired 85 MB.

Let's fix this by recreating the Pod and explicitly setting the memory request to 85 MB.

First, delete the `stress2much` pod with:

```
oc delete pod stress2much --namespace <namespace>-quota
```

Then create a new Pod where the requests and limits are set:

```
apiVersion: v1
kind: Pod
metadata:
  name: stress
spec:
  containers:
    - command:
        - stress
        - --vm
        - "1"
        - --vm-bytes
        - 85M
        - --vm-hang
        - "1"
      image: quay.io/acend/stress:latest
      imagePullPolicy: Always
      name: stress
      resources:
        limits:
          cpu: 100m
          memory: 100Mi
        requests:
          cpu: 10m
          memory: 85Mi
```

And apply this again with:

```
oc apply -f pod_stress.yaml --namespace <namespace>-quota
```

> **Note**
> Remember, if you only set the limit, the request will be set to the same value.

You should now see that the Pod is successfully running:

```
NAME     READY   STATUS    RESTARTS   AGE
stress   1/1     Running   0          25s
```

Task 9.2.3: Hitting the quota

Create another Pod, again using the `stress` image. This time our application is less demanding and only needs 10 MB of memory (`--vm-bytes 10M`):

Create a new Pod resource with:

```
apiVersion: v1
kind: Pod
metadata:
  name: overbooked
spec:
  containers:
    - command:
        - stress
        - --vm
        - "1"
        - --vm-bytes
        - 10M
        - --vm-hang
        - "1"
      image: quay.io/acend/stress:latest
      imagePullPolicy: Always
      name: overbooked
```

```
oc apply -f pod_overbooked.yaml --namespace <namespace>-quota
```

We are immediately confronted with an error message:

```
Error from server (Forbidden): pods "overbooked" is forbidden: exceeded quota: lab-quota, requested: memory=16Mi, used:
 memory=85Mi, limited: memory=100Mi
```

The default request value of 16 MiB of memory that was automatically set on the Pod lets us hit the quota which in turn prevents us from creating the Pod.

Let’s have a closer look at the quota with:

```
oc get quota --output yaml --namespace <namespace>-quota
```

which should output the following YAML definition:

```
...
  status:
    hard:
      cpu: 100m
      memory: 100Mi
    used:
      cpu: 20m
      memory: 80Mi
...
```

The most interesting part is the quota’s status which reveals that we cannot use more than 100 MiB of memory and that 80 MiB are already used.

Fortunately, our application can live with less memory than what the LimitRange sets. Let’s set the request to the remaining 10 MiB:

```
apiVersion: v1
kind: Pod
metadata:
  name: overbooked
spec:
  containers:
    - command:
        - stress
        - --vm
        - "1"
        - --vm-bytes
        - 10M
        - --vm-hang
        - "1"
      image: quay.io/acend/stress:latest
      imagePullPolicy: Always
      name: overbooked
      resources:
        limits:
          cpu: 100m
          memory: 50Mi
        requests:
          cpu: 10m
          memory: 10Mi
```

And apply with:

```
oc apply -f pod_overbooked.yaml --namespace <namespace>-quota
```

Even though the limits of both Pods combined overstretch the quota, the requests do not and so the Pods are allowed to run.

9.3. Init containers

A Pod can have multiple containers running apps within it, but it can also have one or more *init containers*, which are run before the app container is started.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

Check out the Init Containers documentation for more details.

Task 9.3.1: Add an init container

In *7. Attaching a database* you created the `example-web-app` application. In this task, you are going to add an init container which checks if the MariaDB database is ready to be used before actually starting your example application.

Edit your existing `example-web-app` Deployment by changing your local `deployment_example-web-app.yaml` . Add the init container into the existing Deployment (same indentation level as containers):

```
...
    spec:
      initContainers:
        - name: wait-for-db
          image: docker.io/busybox:1.28
          command:
            [
              "sh",
              "-c",
              "until nslookup mariadb.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local;
 do echo waiting for mydb; sleep 2; done",
            ]
...
```

And then apply again with:

```
oc apply -f deployment_example-web-app.yaml --namespace <namespace>
```

> **Note**
> This obviously only checks if there is a DNS Record for your MariaDB Service and not if the database is ready. But you get the idea, right?

Let's see what has changed by analyzing your newly created `example-web-app` Pod with the following command (use `oc get pod` or auto-completion to get the Pod name):

```
oc describe pod <pod> --namespace <namespace>
```

You see the new init container with the name `wait-for-db` :

```
...
Init Containers:
  wait-for-db:
    Container ID:  docker://77e6e309c88cfe62d03ed97e8fae20704bbf547a1e717a8f699ba79d9879cca2
    Image:         busybox
    Image ID:      docker-pullable://busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47
    Port:          <none>
    Host Port:     <none>
    Command:
      sh
      -c
      until nslookup mariadb.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo
waiting for mydb; sleep 2; done
    State:          Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Tue, 10 Nov 2020 21:00:24 +0100
      Finished:     Tue, 10 Nov 2020 21:02:52 +0100
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-xz2b7 (ro)
...
```

The init container has the `State: Terminated` and an `Exit Code: 0` which means it was successful. That's what we wanted, the init container was successfully executed before our main application.

You can also check the logs of the init container with:

```
oc logs -c wait-for-db <pod> --namespace <namespace>
```

Which should give you something similar to:

```
Server:    10.43.0.10
Address 1: 10.43.0.10 kube-dns.kube-system.svc.cluster.local

Name:      mariadb.acend-test.svc.cluster.local
Address 1: 10.43.243.105 mariadb.acend-test.svc.cluster.local
```

Deployment hooks on OpenShift

A similar concept are the so-called pre and post deployment hooks. Those hooks basically give the possibility to execute Pods before and after a deployment is in progress.

Check out the official documentation for further information.

- acend gmbh

10. Helm

Helm is a Cloud Native Foundation project to define, install and manage applications in Kubernetes.

tl;dr

Helm is a Package Manager for Kubernetes

- package multiple K8s resources into a single logical deployment unit
- … but it’s not just a Package Manager

Helm is a Deployment Management for Kubernetes

- do a repeatable deployment
- manage dependencies: reuse and share
- manage multiple configurations
- update, rollback and test application deployments

10.1. Helm overview

Ok, let’s start with Helm. First, you have to understand the following 3 Helm concepts: **Chart**, **Repository** and **Release**.

A **Chart** is a Helm package. It contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. Think of it like the Kubernetes equivalent of a Homebrew formula, an Apt dpkg, or a Yum RPM file.

A **Repository** is the place where charts can be collected and shared. It’s like Perl’s CPAN archive or the Fedora Package Database, but for Kubernetes packages.

A **Release** is an instance of a chart running in a Kubernetes cluster. One chart can often be installed many times in the same cluster. Each time it is installed, a new release is created. Consider a MySQL chart. If you want two databases running in your cluster, you can install that chart twice. Each one will have its own release, which will in turn have its own release name.

With these concepts in mind, we can now explain Helm like this:

> Helm installs charts into Kubernetes, creating a new release for each installation. To find new charts, you can search Helm chart repositories.

- acend gmbh

10.2. CLI installation

This guide shows you how to install the `helm` CLI tool. `helm` can be installed either from source or from pre-built binary releases. We are going to use the pre-built releases. `helm` binaries can be found on Helm's release page for the usual variety of operating systems.

> **Warning**
> If you do this training in our acend web based environment, no installation is required.

Task 10.2.1: Install CLI

Install the CLI for your **Operating System**

1. Download the latest release
2. Unpack it (e.g. `tar -zxvf <filename>`)
3. Copy to the correct location
   - Linux: Find the `helm` binary in the unpacked directory and move it to its desired destination (e.g. `mv linux-amd64/helm ~/.local/bin/`)
     - The desired destination should be listed in your $PATH environment variable (`echo $PATH`)
   - macOS: Find the `helm` binary in the unpacked directory and move it to its desired destination (e.g. `mv darwin-amd64/helm ~/bin/`)
     - The desired destination should be listed in your $PATH environment variable (`echo $PATH`)
   - Windows: Find the `helm` binary in the unpacked directory and move it to its desired destination
     - The desired destination should be listed in your $PATH environment variable (`echo $PATH`)

Task 10.2.2: Verify

To verify, run the following command and check if `Version` is what you expected:

```
helm version
```

The output is similar to this:

```
version.BuildInfo{Version:"v3.10.1", GitCommit:"9f88ccb6aee40b9a0535fcc7efea6055e1ef72c9", GitTreeState:"clean", GoVers
ion:"go1.18.7"}
```

From here on you should be able to run the client.

10.3. Create a chart

In this lab we are going to create our very first Helm chart and deploy it.

Task 10.3.1: Create Chart

First, let's create our chart. Open your favorite terminal and make sure you're in the workspace for this lab, e.g. `cd ~/<workspace-kubernetes-training>` :

```
helm create mychart
```

You will now find a `mychart` directory with the newly created chart. It already is a valid and fully functional chart which deploys an nginx instance. Have a look at the generated files and their content. For an explanation of the files, visit the Helm Developer Documentation . In a later section you'll find all the information about Helm templates.

The default image freshly created chart deploys is a simple nginx image listening on port `80` .

Since OpenShift doesn't allow to run containers as root by default, we need to change the default image to an unprivileged one (`docker.io/nginxinc/nginx-unprivileged`) and also change the containerPort to `8080` .

Change the image in the `mychart/values.yaml`

```
...
image:
  repository: docker.io/nginxinc/nginx-unprivileged
  pullPolicy: IfNotPresent
  # Overrides the image tag whose default is the chart appVersion.
  tag: "latest"
...
```

And then change the port in the `mychart/values.yaml`

```
...
service:
  type: ClusterIP
  port: 8080
...
```

Task 10.3.2: Install Release

Before actually deploying our generated chart, we can check the (to be) generated Kubernetes resources with the following command:

```
helm install --dry-run --debug --namespace <namespace> myfirstrelease ./mychart
```

Finally, the following command creates a new release and deploys the application:

```
helm install --namespace <namespace> myfirstrelease ./mychart
```

With `oc get pods --namespace <namespace>` you should see a new Pod:

```
NAME                                    READY   STATUS    RESTARTS   AGE
myfirstrelease-mychart-6d4956b75-ng8x4   1/1     Running   0          2m21s
```

You can list the newly created Helm release with the following command:

```
helm ls --namespace <namespace>
```

Task 10.3.3: Expose Application

Our freshly deployed nginx is not yet accessible from outside the OpenShift cluster. To expose it, we have to make sure a so called ingress resource will be deployed as well.

Also make sure the application is accessible via TLS.

A look into the file `templates/ingress.yaml` reveals that the rendering of the ingress and its values is configurable through values(`values.yaml`):

```
{{- if .Values.ingress.enabled -}}
{{- $fullName := include "mychart.fullname" . -}}
{{- $svcPort := .Values.service.port -}}
{{- if and .Values.ingress.className (not (semverCompare ">=1.18-0" .Capabilities.KubeVersion.GitVersion)) }}
  {{- if not (hasKey .Values.ingress.annotations "kubernetes.io/ingress.class") }}
  {{- $_ := set .Values.ingress.annotations "kubernetes.io/ingress.class" .Values.ingress.className}}
  {{- end }}
{{- end }}
{{- if semverCompare ">=1.19-0" .Capabilities.KubeVersion.GitVersion -}}
apiVersion: networking.k8s.io/v1
{{- else if semverCompare ">=1.14-0" .Capabilities.KubeVersion.GitVersion -}}
apiVersion: networking.k8s.io/v1beta1
{{- else -}}
apiVersion: extensions/v1beta1
{{- end }}
kind: Ingress
metadata:
  name: {{ $fullName }}
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
  {{- with .Values.ingress.annotations }}
  annotations:
    {{- toYaml . | nindent 4 }}
  {{- end }}
spec:
  {{- if and .Values.ingress.className (semverCompare ">=1.18-0" .Capabilities.KubeVersion.GitVersion) }}
  ingressClassName: {{ .Values.ingress.className }}
  {{- end }}
  {{- if .Values.ingress.tls }}
  tls:
    {{- range .Values.ingress.tls }}
    - hosts:
        {{- range .hosts }}
        - {{ . | quote }}
        {{- end }}
      secretName: {{ .secretName }}
    {{- end }}
  {{- end }}
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ .host | quote }}
      http:
        paths:
          {{- range .paths }}
          - path: {{ .path }}
            {{- if and .pathType (semverCompare ">=1.18-0" $.Capabilities.KubeVersion.GitVersion) }}
            pathType: {{ .pathType }}
            {{- end }}
            backend:
              {{- if semverCompare ">=1.19-0" $.Capabilities.KubeVersion.GitVersion }}
              service:
                name: {{ $fullName }}
                port:
                  number: {{ $svcPort }}
              {{- else }}
              serviceName: {{ $fullName }}
              servicePort: {{ $svcPort }}
              {{- end }}
          {{- end }}
    {{- end }}
{{- end }}
```

Thus, we need to change this value inside our `mychart/values.yaml` file. This is also where we enable the TLS part:

Note

Make sure to replace the `namespace` and `appdomain` accordingly.

```
[...]
ingress:
  enabled: true
  className: ""
  # as we learned in previous labs, OpenShift uses Routes instead of Ingresses
  # to let OpenShift automatically generate the corresponding Route, we need the following annotation. more information
:
  # https://docs.openshift.com/container-platform/latest/networking/routes/route-configuration.html#nw-ingress-creating
-a-route-via-an-ingress_route-configuration
  annotations:
    route.openshift.io/termination: "edge"
  hosts:
    - host: mychart-<namespace>.<appdomain>
      paths:
        - path: /
          pathType: ImplementationSpecific
[...]
```

> **Note**
> Make sure to set the proper value as hostname. `appdomain` will be provided by the trainer.

Apply the change by upgrading our release:

```
helm upgrade --namespace <namespace> myfirstrelease ./mychart
```

This will result in something similar to:

```
Release "myfirstrelease" has been upgraded. Happy Helming!
NAME: myfirstrelease
LAST DEPLOYED: Wed Dec  2 14:44:42 2020
NAMESPACE: <namespace>
STATUS: deployed
REVISION: 2
NOTES:
1. Get the application URL by running these commands:
  https://<namespace>.<appdomain>/
```

Check whether the ingress was successfully deployed by accessing the URL `https://mychart-<namespace>.<appdomain>/`

Task 10.3.4: Overwrite value using commandline param

An alternative way to set or overwrite values for charts we want to deploy is the `--set name=value` parameter. This parameter can be used when installing a chart as well as upgrading.

Update the replica count of your nginx Deployment to 2 using `--set name=value`

Solution

```
helm upgrade --namespace <namespace> --set replicaCount=2 myfirstrelease ./mychart
```

Values that have been set using `--set` can be reset by helm upgrade with `--reset-values` .

Task 10.3.5: Values

Have a look at the `values.yaml` file in your chart and study all the possible configuration params introduced in a freshly created chart.

Task 10.3.6: Remove release

To remove an application, simply remove the Helm release with the following command:

```
helm uninstall myfirstrelease --namespace <namespace>
```

Do this with our deployed release. With `oc get pods --namespace <namespace>` you should no longer see your application Pod.

10.4. Complex example

In this extended lab, we are going to deploy an existing, more complex application with a Helm chart from the Artifact Hub.

Artifact Hub

Check out Artifact Hub where you'll find a huge number of different Helm charts. For this lab, we'll use the WordPress chart by Bitnami , a publishing platform for building blogs and websites.

WordPress

As this WordPress Helm chart is published in Bitnami's Helm repository, we're first going to add it to our local repo list:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Let's check if that worked:

```
helm repo list
```

```
NAME    URL
bitnami https://charts.bitnami.com/bitnami
```

Now look at the available configuration for this Helm chart. Usually you can find it in the `values.yaml` or in the chart's readme file. You can also check it on its Artifact Hub page .

We are going to override some of the values. For that purpose, create a new `values.yaml` file locally on your workstation (e.g. `~/<workspace>/values.yaml`) with the following content:

```
---
persistence:
  size: 1Gi
service:
  type: ClusterIP
updateStrategy:
  type: Recreate

podSecurityContext:
  enabled: false
containerSecurityContext:
  enabled: false

ingress:
  enabled: true
  hostname: wordpress-<namespace>.<appdomain>
  extraTls:
  - hosts:
      - wordpress-<namespace>.<appdomain>


mariadb:
  primary:
    persistence:
      size: 1Gi

    podSecurityContext:
      enabled: false
    containerSecurityContext:
      enabled: false
```

> **Note**
> Make sure to set the proper value as hostname. `appdomain` will be provided by the trainer.

If you look inside the Chart.yaml file of the WordPress chart, you'll see a dependency to the MariaDB Helm chart . All the MariaDB values are used by this dependent Helm chart and the chart is automatically deployed when installing WordPress.

The `Chart.yaml` file allows us to define dependencies on other charts. In our Wordpress chart we use the `Chart.yaml` to add a `mariadb` to store the WordPress data in.

```
dependencies:
  - condition: mariadb.enabled
    name: mariadb
    repository: https://charts.bitnami.com/bitnami
    version: 9.x.x
```

Helm's best practices suggest to use version ranges instead of a fixed version whenever possible. The suggested default therefore is patch-level version match:

```
version: ~3.5.7
```

This is e.g. equivalent to `>= 3.5.7, < 3.6.0` Check this SemVer readme chapter for more information on version ranges.

> **Note**
> For more details on how to manage **dependencies**, check out the Helm Dependencies Documentation .

Subcharts are an alternative way to define dependencies within a chart: A chart may contain another chart (inside of its `charts/` directory) upon which it depends. As a result, when installing the chart, it will install all of its dependencies from the `charts/` directory.

We are now going to deploy the application in a specific version (which is not the latest release on purpose). Also note that we define our custom `values.yaml` file with the `-f` parameter:

```
helm install wordpress bitnami/wordpress -f values.yaml --namespace <namespace>
```

Look for the newly created resources with `helm ls` and `oc get deploy,pod,ingress,pvc`:

```
helm ls --namespace <namespace>
```

which gives you:

```
NAME      NAMESPACE       REVISION  UPDATED                                 STATUS    CHART             APP VERSION
wordpress <namespace>         1     2021-03-25 14:27:38.231722961 +0100 CET     deployed  wordpress-10.7.1  5.7.0
```

and

```
oc get deploy,pod,ingress,pvc --namespace <namespace>
```

which gives you:

```
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/wordpress   1/1     1            1           2m6s

NAME                             READY   STATUS    RESTARTS   AGE
pod/wordpress-6bf6df9c5d-w4fpx   1/1     Running   0          2m6s
pod/wordpress-mariadb-0          1/1     Running   0          2m6s

NAME                            HOSTS                                          ADDRESS       PORTS   AGE
ingress.extensions/wordpress    wordpress-<namespace>.<appdomain>              10.100.1.10   80      2m6s

NAME                                          STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS            AGE
persistentvolumeclaim/data-wordpress-mariadb-0   Bound    pvc-859fe3b4-b598-4f86-b7ed-a3a183f700fd   1Gi        RWO
        cloudscale-volume-ssd   2m6s
persistentvolumeclaim/wordpress                  Bound    pvc-83ebf739-0b0e-45a2-936e-e925141a0d35   1Gi        RWO
        cloudscale-volume-ssd   2m7s
```

In order to check the values used in a given release, execute:

```
helm get values wordpress --namespace <namespace>
```

which gives you:

```
USER-SUPPLIED VALUES:
containerSecurityContext:
  enabled: false
ingress:
  enabled: true
  hostname: wordpress-<namespace>.<appdomain>
  extraTls:
  - hosts:
      - wordpress-<namespace>.<appdomain>
mariadb:
  primary:
    containerSecurityContext:
      enabled: false
    persistence:
      size: 1Gi
    podSecurityContext:
      enabled: false
persistence:
  size: 1Gi
podSecurityContext:
  enabled: false
service:
  type: ClusterIP
updateStrategy:
  type: Recreate
```

As soon as all deployments are ready (meaning pods `wordpress` and `mariadb` are running) you can open the application with the URL from your Ingress resource defined in `values.yaml` .

Upgrade

We are now going to upgrade the application to a newer Helm chart version. When we installed the Chart, a couple of secrets were needed during this process. In order to do the upgrade of the Chart now, we need to provide those secrets to the upgrade command, to be sure no sensitive data will be overwritten:

- wordpressPassword
- mariadb.auth.rootPassword
- mariadb.auth.password

> **Note**
> This is specific to the wordpress Bitami Chart, and might be different when installing other Charts.

Use the following commands to gather the secrets and store them in environment variables. Make sure to replace `<namespace>` with your current value.

```
export WORDPRESS_PASSWORD=$(oc get secret wordpress -o jsonpath="{.data.wordpress-password}" --namespace <namespace> |
base64 --decode)
```

```
export MARIADB_ROOT_PASSWORD=$(oc get secret wordpress-mariadb -o jsonpath="{.data.mariadb-root-password}" --namespace
<namespace> | base64 --decode)
```

```
export MARIADB_PASSWORD=$(oc get secret wordpress-mariadb -o jsonpath="{.data.mariadb-password}" --namespace <namespace
> | base64 --decode)
```

Then do the upgrade with the following command:

```
helm upgrade -f values.yaml --set wordpressPassword=$WORDPRESS_PASSWORD --set mariadb.auth.rootPassword=$MARIADB_ROOT_P
ASSWORD --set mariadb.auth.password=$MARIADB_PASSWORD wordpress bitnami/wordpress --namespace <namespace>
```

And then observe the changes in your WordPress and MariaDB Apps

Cleanup

```
helm uninstall wordpress --namespace <namespace>
```

Additional Task

Study the Helm best practices as an optional and additional task.

11. Kustomize

Kustomize is a tool to manage YAML configurations for Kubernetes objects in a declarative and reusable manner. In this lab, we will use Kustomize to deploy the same app for two different environments.

Installation

Kustomize can be used in two different ways:

- As a standalone `kustomize` binary, downloadable from kubernetes.io
- With the parameter `--kustomize` or `-k` in certain `oc` subcommands such as `apply` or `create`

> **Note**
> You might get a different behaviour depending on which variant you use. The reason for this is that the version built into `oc` is usually older than the standalone binary.

Usage

The main purpose of Kustomize is to build configurations from a predefined file structure (which will be introduced in the next section):

```
kustomize build <dir>
```

The same can be achieved with `oc` :

```
oc kustomize <dir>
```

The next step is to apply this configuration to the OpenShift cluster:

```
kustomize build <dir> | oc apply -f -
```

Or in one `oc` command with the parameter `-k` instead of `-f` :

```
oc apply -k <dir>
```

Task 11.1: Prepare a Kustomize config

We are going to deploy a simple application:

- The Deployment starts an application based on nginx
- A Service exposes the Deployment

- acend gmbh

- The application will be deployed for two different example environments, integration and production

Kustomize allows inheriting Kubernetes configurations. We are going to use this to create a base configuration and then override it for the different environments. Note that Kustomize does not use templating. Instead, smart patch and extension mechanisms are used on plain YAML manifests to keep things as simple as possible.

Get the example config

Find the needed resource files inside the folder `content/en/docs/kustomize/kustomize` of the techlab github repository. Clone the repository or get the content as zip

Change to the folder `content/en/docs/kustomize/kustomize` to execute the kustomize commands.

> **Note**
> Commands for git checkout and folder switch:

```
git clone https://github.com/acend/kubernetes-basics-training.git
cd kubernetes-basics-training/content/en/docs/kustomize/kustomize/
```

File structure

The structure of a Kustomize configuration typically looks like this:

```
.
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    ├── production
    │   ├── deployment-patch.yaml
    │   ├── kustomization.yaml
    │   └── service-patch.yaml
    └── staging
        ├── deployment-patch.yaml
        ├── kustomization.yaml
        └── service-patch.yaml
```

Base

Let’s have a look at the `base` directory first which contains the base configuration. There’s a `deployment.yaml` with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kustomize-app
spec:
  selector:
    matchLabels:
      app: kustomize-app
  template:
    metadata:
      labels:
        app: kustomize-app
    spec:
      containers:
        - name: kustomize-app
          image: quay.io/acend/example-web-go
          env:
            - name: APPLICATION_NAME
              value: app-base
          command:
            - sh
            - -c
            - |-
              set -e
              /bin/echo "My name is $APPLICATION_NAME"
              /usr/local/bin/go
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
```

There's also a Service for our Deployment in the corresponding `base/service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: kustomize-app
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    app: kustomize-app
```

And there's an additional `base/kustomization.yaml` which is used to configure Kustomize:

```
resources:
  - service.yaml
  - deployment.yaml
```

It references the previous manifests `service.yaml` and `deployment.yaml` and makes them part of our base configuration.

Overlays

Now let's have a look at the other directory which is called `overlays`. It contains two subdirectories `staging` and `production` which both contain a `kustomization.yaml` with almost the same content.

`overlays/staging/kustomization.yaml`:

```
nameSuffix: -staging
bases:
  - ../../base
patchesStrategicMerge:
  - deployment-patch.yaml
  - service-patch.yaml
```

`overlays/production/kustomization.yaml` :

```
nameSuffix: -production
bases:
  - ../../base
patchesStrategicMerge:
  - deployment-patch.yaml
  - service-patch.yaml
```

Only the first key `nameSuffix` differs.

In both cases, the `kustomization.yaml` references our base configuration. However, the two directories contain two different `deployment-patch.yaml` files which patch the `deployment.yaml` from our base configuration.

`overlays/staging/deployment-patch.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kustomize-app
spec:
  selector:
    matchLabels:
      app: kustomize-app-staging
  template:
    metadata:
      labels:
        app: kustomize-app-staging
    spec:
      containers:
        - name: kustomize-app
          env:
            - name: APPLICATION_NAME
              value: kustomize-app-staging
```

`overlays/production/deployment-patch.yaml` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kustomize-app
spec:
  selector:
    matchLabels:
      app: kustomize-app-production
  template:
    metadata:
      labels:
        app: kustomize-app-production
    spec:
      containers:
        - name: kustomize-app
          env:
            - name: APPLICATION_NAME
              value: kustomize-app-production
```

The main difference here is that the environment variable `APPLICATION_NAME` is set differently. The `app` label also differs because we are going to deploy both Deployments into the same Namespace.

The same applies to our Service. It also comes in two customizations so that it matches the corresponding Deployment in the same Namespace.

`overlays/staging/service-patch.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: kustomize-app
spec:
  selector:
    app: kustomize-app-staging
```

`overlays/production/service-patch.yaml` :

```
apiVersion: v1
kind: Service
metadata:
  name: kustomize-app
spec:
  selector:
    app: kustomize-app-production
```

> **Note**
> All files mentioned above are also directly accessible from GitHub .

Prepare the files as described above in a local directory of your choice.

Task 11.2: Deploy with Kustomize

We are now ready to deploy both apps for the two different environments. For simplicity, we will use the same Namespace.

```
oc apply -k overlays/staging --namespace <namespace>
```

```
service/kustomize-app-staging created
deployment.apps/kustomize-app-staging created
```

```
oc apply -k overlays/production --namespace <namespace>
```

```
service/kustomize-app-production created
deployment.apps/kustomize-app-production created
```

As you can see, we now have two deployments and services deployed. Both of them use the same base configuration. However, they have a specific configuration on their own as well.

Let’s verify this. Our app writes a corresponding log entry that we can use for analysis:

```
oc get pods --namespace <namespace>
```

```
NAME                                    READY   STATUS    RESTARTS   AGE
kustomize-app-production-74c7bdb7d-8cccd   1/1     Running   0          2m1s
kustomize-app-staging-7967885d5b-qp6l8     1/1     Running   0          5m33s
```

```
oc logs kustomize-app-staging-7967885d5b-qp6l8
```

```
My name is kustomize-app-staging
```

```
oc logs kustomize-app-production-74c7bdb7d-8cccd
```

```
My name is kustomize-app-production
```

Further information

Kustomize has more features of which we just covered a couple. Please refer to the docs for more information.

- Kustomize documentation: https://kubernetes-sigs.github.io/kustomize/
- API reference: https://kubernetes-sigs.github.io/kustomize/api-reference/
- Another `kustomization.yaml` reference: https://kubectl.docs.kubernetes.io/pages/reference/kustomize.html

- Examples: https://github.com/kubernetes-sigs/kustomize/tree/master/examples