

# Setup

## Command line tool

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.

## Setup introduction

This training depends on the installation of `helm`. Follow the instructions on the subsequent pages to complete the setup on your platform of choice.

# Installation for Windows

## Installation for Windows

Install the `helm` CLI binary on your system:

1. [Download the latest release](#)
2. Unzip it
3. 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` )

### Note

Windows quick hack: Copy the `helm` binary directly into the folder `C:\Windows`.

If the `$PATH` variable doesn't contain a suitable directory, it can be changed in the advanced system settings:

- [How to set the path and environment variables in Windows](#)

## Verification

Now, [verify your installation](#) .

# Installation for macOS

## Installation for macOS

Install the `helm` CLI binary on your system:

1. [Download the latest release](#)
2. Unpack it (e.g. `tar -zxvf <filename>` )
3. 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` )

## Verification

Now, [verify your installation](#) .

# Installation for Linux

## Installation for Linux

Install the `helm` CLI binary on your system:

1. [Download the latest release](#)
2. Unpack it (e.g. `tar -zxvf <filename>`)
3. 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` )

## Verification

Now, [verify your installation](#) .

# Verify the installation

## Verify the installation

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

```
---
```

The output should be similar to this:

```
version.BuildInfo{Version:"v3.7.0", GitCommit:"eeac83883cb4014fe60267ec6373570374ce770b", GitTreeState:"clean", GoVersion:"go1.16.8"}
```

## First steps with helm

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

## Next steps

When you're ready to go, head on over to the [labs](#) and begin with the training!

# Labs

The purpose of these labs is to convey Helm basics by providing hands-on tasks for people. [Helm](#) is a [Cloud Native Foundation](#) project to define, install and manage applications in Kubernetes and any Kubernetes distribution like OpenShift.

It can be used to package multiple Kubernetes resources into a single logical deployment unit.

But it's not just a package manager.

Helm is a deployment management for Kubernetes. It can:

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

Goals of these labs:

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

## Prerequisites

- We assume you have knowledge about OpenShift and understand the concepts of [Pods](#) , [Deployments](#) , [Services](#) , [Ingress](#) and [Secrets](#) .
- Make sure you have access to the internet from your shell. If required, set appropriate proxy settings in your shell. This is only needed for lab 2 to access the Artifact Hub.

## 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 OpenShift 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 OpenShift 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.

## Getting started

After this short introduction, make sure you have completed [setup](#) . You're then ready to start [with the first lab](#) !

# 1. Getting started

## Task 1.1: Web IDE

The first thing we're going to do is to explore our lab environment and get in touch with the different components.

### Note

You can also use your local installation of the cli tools. Make sure you completed [the setup](#) before you continue with this lab.

### Note

The URL and Credentials to the Web IDE will be provided by the teacher. Use Chrome for the best experience.

Once you're successfully logged into the web IDE open a new Terminal by hitting `CTRL + SHIFT + `` or clicking the Menu button -> Terminal -> new Terminal and check the installed ocversion by executing the following command:

```
---
```

The Web IDE Pod consists of the following tools:

- oc
- kubectl
- kustomize
- helm
- kubectx
- kubens
- tekton cli
- odo
- argocd

The files in the home directory under `/home/project` are stored in a persistence volume, so please make sure to store all your persistence data in this directory.

## Task 1.2: Login

- Open a separate Browser Tab and login to the OpenShift [Webconsole](#) using your `<username>` and `<password>`
- Open the menu behind your username in the top right corner
- Hit Copy Login Command
- Display the Token and copy the command
- Execute the command in your Webshell terminal.

## Task 1.3: Namespace

- acend gmbh

For the following labs, we are going to create a namespace. You can choose any name, we suggest using e.g. your username.

You can create your namespace with:

```
...
```

### Note

We are going to use `<namespace>` as a placeholder for your created project. Each time you see a `<namespace>` somewhere in a command, replace it with your chosen project name.

## Task 1.3: Helm CLI

The Helm [CLI](#) is a self-contained binary written in the Go programming language. The most recent release can be found on the official [GitHub page](#) . In this chapter, you will learn how to get help and how to enable the autocompletion feature.

### Task 1.1: Getting familiar with the CLI

You can get help for each command of the CLI with the flag `--help` .

```
...
```

You will see a list of the available commands and flags. If you prefer to browse the manual in the browser you'll find it in the [online documentation](#) .

## - acend gmbh

Usage:

```
helm [command]
```

Available Commands:

```
completion  generate autocompletion scripts for the specified shell
create      create a new chart with the given name
dependency  manage a chart's dependencies
env         helm client environment information
get        download extended information of a named release
help       Help about any command
history    fetch release history
install    install a chart
lint       examine a chart for possible issues
list       list releases
package    package a chart directory into a chart archive
plugin     install, list, or uninstall Helm plugins
pull       download a chart from a repository and (optionally) unpack it in local directory
repo       add, list, remove, update, and index chart repositories
rollback   roll back a release to a previous revision
search     search for a keyword in charts
show       show information of a chart
status     display the status of the named release
template   locally render templates
test       run tests for a release
uninstall  uninstall a release
upgrade    upgrade a release
verify     verify that a chart at the given path has been signed and is valid
version    print the client version information
```

Flags:

```
--debug          enable verbose output
-h, --help       help for helm
--kube-apiserver string  the address and the port for the Kubernetes API server
--kube-as-group stringArray  group to impersonate for the operation, this flag can be repeated to specify multiple groups.
--kube-as-user string     username to impersonate for the operation
--kube-ca-file string     the certificate authority file for the Kubernetes API server connection
--kube-context string     name of the kubeconfig context to use
--kube-token string       bearer token used for authentication
--kubeconfig string      path to the kubeconfig file
-n, --namespace string   namespace scope for this request
--registry-config string  path to the registry config file (default "/home/bbuehlmann/.config/helm/registry.json")
--repository-cache string  path to the file containing cached repository indexes (default "/home/bbuehlmann/.config/helm/repository")
--repository-config string  path to the file containing repository names and URLs (default "/home/bbuehlmann/.config/helm/repositories.yaml")
```

Use "helm [command] --help" for more information about a command.

You can use the `--help` flag on each command and subcommand of the CLI.

This will print out the documentation for the `install` command. Play around and get familiar with the different commands of the Helm CLI.

## Task 1.1: Autocompletion

### Note

This step is only needed when you're not working with the Web IDE we've provided. The autocompletion is already installed in the Web IDE

- acend gmbh

If you are using the Helm CLI on Linux or Mac OS X you can enable the [autocomplete feature](#) . With autocomplete it's even easier to learn the commands, subcommands and their flags. Last but not least it improves the productivity while using Helm.

The autocomplete feature can be enabled for `bash` , `zsh` and `fish` .

The following example enables autocomplete in the current `bash` :

```
helm completion bash >> ~/.bashrc
```

After typing `helm` you can autocomplete the commands and subcommands with a double tap the tabulator key. This works even for installed releases on the cluster: A double tab after `helm get all` prints out all installed helm releases.

To install autocomplete permanently for `bash` you can use the following command:

```
helm completion bash >> ~/.bashrc
```

This appends the command `source <(helm completion bash)` to the end of file `~/.bashrc` which will be sourced on launch of the `bash` .

## 2. A simple chart

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

### Task 2.1: 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-helm-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 a `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 the 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 it to an unprivileged one ( `nginxinc/nginx-unprivileged:latest` ) and also change the `containerPort` to `8080` .

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

```
image: nginxinc/nginx-unprivileged:latest
```

And then change the `containerPort` in the `mychart/templates/deployment.yaml` :

```
containerPort: 8080
```

### Task 2.2: Install Release

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

```
helm template mychart
```

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

```
helm install mychart
```

#### Note

Use the `helm upgrade -i` command, instead of `helm install` or `helm upgrade` depending on whether the release is already installed or not.



- acend gmbh

```
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:  
   http://mychart-<namespace>.training.openshift.ch/
```

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

### Note

It might take a moment until the app is accessible.

You should see the Nginx welcome page.

# Welcome to nginx!

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

For online documentation and support please refer to [nginx.org](https://nginx.org).  
Commercial support is available at [nginx.com](https://nginx.com).

*Thank you for using nginx.*

## Task 2.4: 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. `--set name=value` 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 Task 2.4

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

Verify the replicaCount with the following command:

- acend gmbh

```
NAME                                READY  STATUS   RESTARTS  AGE
myfirstrelease-mychart-c95cb97d6-g76rc  1/1    Running  0          10m
myfirstrelease-mychart-c95cb97d6-tqztc  1/1    Running  0          8m25s
```

## Task 2.5: Rollback a release

Helm also provides the functionality to roll back a release to a specific revision number. Every change you make to a release by installing, upgrading and even rollback, it will increase the `REVISION`. The deployed revision can be displayed with the following command:

```
NAME          NAMESPACE    REVISION  UPDATED                               STATUS   CHART          APP VERSION
myfirstrelease <namespace>  3         2021-04-14 14:29:58.808282266 +0200 CEST  deployed  mychart-0.1.01.16.0
```

Let's now rollback our release `myfirstrelease` to revision 2.

The replicaCount should be back down to 1 after the rollback. Check if that's true with the following command:

```
NAME                                READY  STATUS   RESTARTS  AGE
myfirstrelease-mychart-c95cb97d6-g76rc  1/1    Running  0          10m
```

## Task 2.6 Explore values.yaml

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 2.7: Remove release

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

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

Pod.

- acend gmbh

## 3. A more complex application

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 search repo bitnami
```

```
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:

```
replicaCount: 1

image:
  repository: bitnami/wordpress
  tag: latest
  pullPolicy: IfNotPresent

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

service:
  type: ClusterIP

ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
  hosts:
    - wordpress.local
  paths:
    - path: /
      pathType: Prefix
  tls: []

resources: {}

nodeSelector: {}

tolerations: []
```

#### Note

Make sure to replace the `<namespace>` and `training.openshift.ch` accordingly.

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.

- acend gmbh

```
helm upgrade --install my-app my-chart --version 3.5.7
```

[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 upgrade --install my-app my-chart --version 3.5.7 --values values.yaml
```

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

```
helm ls
```

which gives you:

```
NAME            REVISION    UPDATED                         STATUS          CHART NAME    CHART VERSION    APP VERSION     DESCRIPTION
my-app          1           2023-01-01 10:00:00             DEPLOYED       my-chart       3.5.7           1.0.0           A Helm chart for Kubernetes
```

and

```
oc get deploy,pod,ingress,pvc
```

which gives you:

```
NAME          READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES
deployment.apps/my-app  1/1     1             1           1m   my-app       my-chart:3.5.7
pod           1/1     1             1           1m   my-app       my-chart:3.5.7
ingress       1/1     1             1           1m   my-app       my-chart:3.5.7
pvc           1/1     1             1           1m   my-app       my-chart:3.5.7
```

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

which gives you:

```
helm upgrade --install wordpress wordpress --namespace <namespace> --values values.yaml
```

As soon as all deployments are ready (meaning pods `wordpress` and `mariadb` are running) you can open the application with the URL `https://wordpress-<namespace>.training.openshift.ch` 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=$(kubectl get secret --namespace <namespace> wordpress-secret -o jsonpath='{.data.wordpress-password}' | base64 --decode)
```

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

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

Then do the upgrade with the following command:

```
helm upgrade --install wordpress wordpress --namespace <namespace> --values values.yaml --set wordpressPassword=$WORDPRESS_PASSWORD --set mariadb.auth.rootPassword=$MARIADB_ROOT_PASSWORD --set mariadb.auth.password=$MARIADB_PASSWORD
```

And then observe the changes in your WordPress and MariaDB Apps

## Cleanup

## Additional Task

Study the Helm [best practices](#) as an optional and additional task.

## 4. Debugging Helm

Debugging templates can be tricky because the rendered templates are sent to the OpenShift API server which may reject the YAML files for reasons other than formatting.

There are a few commands that can help you debug:

- `helm lint` is your go-to tool for verifying that your chart follows best practices.
- `helm install --dry-run --debug --generate-name <chart> --namespace $USER` or `helm template --debug <chart>` : We've seen this trick already. It's a great way to render your templates without applying them to the cluster.
- `helm template -s templates/<template-file> . --debug | cat -n -` render a single file to the standard output
- `helm get manifest <release> --namespace $USER` : This is a good way to see what templates are installed on the server.
- `helm get values <release> --namespace $USER` : This helps you to understand which values are used for a release.

When your YAML is failing to parse but you want to see what is generated, one easy way to retrieve the YAML is to comment out the problem section in the template and then re-run `helm install --dry-run --debug --generate-name <chart> .`

```
apiVersion: v2
# some: problem section
# {{ .Values.foo | quote }}
```

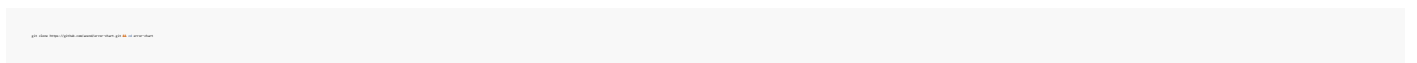
The above will be rendered and returned with the comments intact:

```
apiVersion: v2
# some: problem section
# "bar"
```

This provides a quick way of viewing the generated content without YAML parse errors blocking.

### Task 4.1: Get the chart

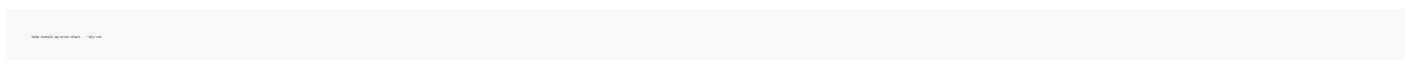
Get the `error-chart` chart by either [downloading the repository's ZIP file](#) or by cloning it from GitHub:



### Task 4.2: Fix the chart

The `error-chart` chart contains some deliberate errors. Try to find all of these errors and fix them, then install the chart using `myrelease` as release name. You can use one or several ways shown above to do so. The goal of this task is a successfully running `myrelease-error-chart` pod in your own namespace.

If you try to install the chart with following command, you will get an error:



### Hints

## YAML

YAML has some strict formatting rules. Check if all files conform to these rules. There is a bad formatting in the `ingress.yaml` template. Use the `helm template -s templates/ingress.yaml . --debug | cat -n -` command to render the ingress template. Be aware that the first two lines ( `---` YAML directive marker and the `# Source: error-chart/templates/ingress.yaml` YAML comment are not considered).

## Kubernetes resource definitions

Find out what resource files are not correct resource definitions.

## Values

Check whether all defined values in `values.yaml` look ok to you. Also check if those values used in the templates reference the correct ones.

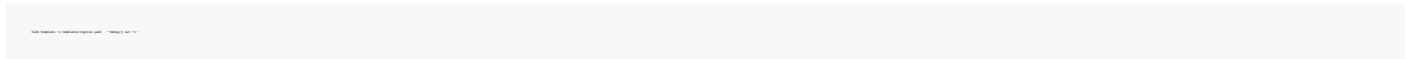
## Solution

### Ingress path

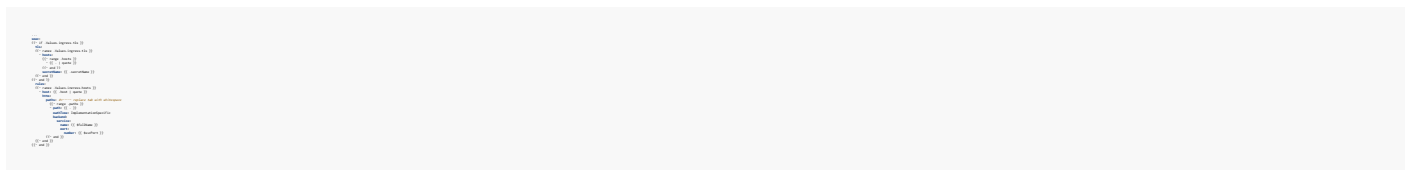
The first error we get when trying to install the chart or when using the `helm lint` command is this:

```
[ERROR] templates/ingress.yaml: unable to parse YAML
error converting YAML to JSON: yaml: line 15: found character that cannot start any token
```

“found character that cannot start any token” probably doesn’t ring a bell so we try to find out what’s wrong with line 15 in file `templates/ingress.yaml`. Beware that line 15 corresponds to line 15 of the rendered file! If you want to know what is on line 15 in the rendered file, you can execute the following command:



Opening the file and going to the appropriate line containing `paths:`, we notice that a tab instead of whitespace characters was used to indent. Replace it with whitespace characters.



### Deployment empty selector

The second error we get when trying to install the chart reads:

```
Error: release myrelease failed: Deployment.apps "myrelease-error-chart" is invalid: spec.selector: Invalid value: v1.LabelSelector{MatchLabels:map[string]string(nil), MatchExpressions:[v1.LabelSelectorRequirement(nil)}: empty selector is invalid for deployment
```

If you look at the deployment template or its rendered form you’ll notice that something’s wrong with the selector:

```
selector:
  matchLabels:
    app.kubernetes.io/name: {{ include "error-chart.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

- acend gmbh

Those two key-value pairs `app.kubernetes.io/name` and `app.kubernetes.io/instance` should be indented by two more whitespaces to look like this:

```
selector:
  matchLabels:
    app.kubernetes.io/name: {{ include "error-chart.name" . }}
    app.kubernetes.io/instance: {{ .Release.Name }}
```

## Ingress host

The host value seems to be incorrect (parts of the url were replaced by placeholders in the following message):

```
Error: release myrelease failed: Ingress.extensions "myrelease-error-chart" is invalid: spec.rules[0].host: Invalid value: "helmttechlab-errorchart-<namespace>.training.openshift.ch": a DNS-1123 subdomain must consist of lower case alphanumeric characters, '-' or '.', and must start and end with an alphanumeric character (e.g. 'example.com', regex used for validation is '[a-z0-9]([-a-z0-9]*[a-z0-9])?(\.[a-z0-9]([-a-z0-9]*[a-z0-9])?)*')
```

If you look closely, you'll notice that the placeholder `<namespace>` is still in the host's value. This does not correspond to hostname conventions. The file `templates/ingress.yaml` reveals that the host's value is defined in the `values.yaml` file. Fix the `host` value.

## Deployment image tag

Even though the chart could successfully be installed the pod has a status of `InvalidImageName`. Looking at the rendered deployment we notice that the image is missing a tag:

```
spec:
  containers:
  - image: 'nginxinc/nginx-unprivileged:'
    imagePullPolicy: IfNotPresent
```

As with the ingress host, the used value seems to be wrong. The ingress template has the following definition of `image`:

```
image: "{{ .Values.image.repository }}:{{ .Values.image.tags }}"
```

So let's have a look at the `values.yaml` file:

```
image:
  repository: nginxinc/nginx-unprivileged
  tag: stable
  pullPolicy: IfNotPresent
```

There's no variable `tags`, instead it's named `tag`. Fix that in the template so that the `image` line reads:

```
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

You might also have to change the `repository` value because Docker Hub might not be accessible from the used environment.

## Compare with solution

You can find the solution in the repository under the `solution` branch. If you want to check if your solution is correct, you can simply compare your workspace against the solution branch.

- acend gmbh

per 01/01/2020

## 5. Go templating

In this lab we are going to learn how to use Go templating in Helm templates.

### Task 5.1: Create a new Helm chart

Let's create a new Helm chart with the name `gotemplatechart` and remove all default templates from the `templates` folder.

#### Solution

```
helm create gotemplatechart
```

#### Warning

Don't forget to adapt the image name to an unprivileged one ( `nginxinc/nginx-unprivileged:latest` ) so OpenShift can run it. See [lab 2](#) for details.

### Task 5.2: Add a ConfigMap template

As the template directory is completely empty, it's time to create our first template. For the purpose of this lab we're going to use a simple ConfigMap template. If you don't exactly understand what a ConfigMap is, consider reading the [OpenShift documentation on how to configure a pod to use a ConfigMap](#) .

Create the template called `gotemplatechart/templates/configmap.yaml` :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Chart.Name }}-cm
data:
  {{ .Chart.Name }}: "value"
```

We can now render the template with the following command:

```
helm template gotemplatechart
```

If everything went ok we can deploy a release of the chart in our namespace:

```
helm install gotemplatechart
```

### Task 5.3: The first Go template directive

As our first Go template directives we are going to add so-called built-in objects: the chart name and version `{{ .Chart.Name }}`-`{{ .Chart.Version }}` .

The template directive is enclosed in double curly braces `{{` and `}}` .

Update the `gotemplatechart/templates/configmap.yaml` :

- acend gmbh

```
helm template gotemplatechart ...
```

Rendering the new template again with `helm template gotemplatechart ...` (see task 2) will therefore result in the following output:

```
helm template gotemplatechart ...
```

The directives `{{ .Chart.Name }}` and `{{ .Chart.Version }}` inject the name and version of the actual chart. But where are those values coming from?

Within those directives data is accessible through a data structure. The leading `.` represents the root of the object structure and is the entrypoint to access data in templates. Built-in objects such as a chart, release, file, template, values and more are therefore accessible in a similar fashion. Check the official [Helm documentation about built-in objects](#) for further and more in-depth information.

The `.Chart` data structure obviously comes from the `Chart.yaml` file and represents the values of this file.

## Task 5.4: Add data from values.yaml

As you could have guessed by now, the `values.yaml` file allows us to configure values and parameters used during the rendering of the templates to replace strings, parameters, functions or even to control whether a part of a template is rendered at all. Let's add a couple more directives to our ConfigMap.

Remove the default content of the `values.yaml` and replace it with the following:

```
values.yaml
```

Now also add your favorite color to be rendered in the ConfigMap as data under the key `myFavoriteColor`. Edit the `gotemplatechart/templates/configmap.yaml` file accordingly.

Use `helm template gotemplatechart ...` again (as in task 2) to see what your rendered Kubernetes resources look like.

### Note

The output should look like this:

```
---
# Source: gotemplatechart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
...
data:
  ...
  myFavoriteColor: blue
```

## Solution Task 5.4



## Task 5.5: Structured data

As mentioned in task 3, data within the built-in objects can be structured and values can be nested.

Update the ConfigMap template so that the following `values.yaml` will result in the same rendered resource as in task 4.



## Solution Task 5.5



### Note

The official [Helm best practices](#) suggest using flat values over nested ones:

In most cases, flat should be favored over nested. The reason for this is that it is simpler for template developers and users.

## Template functions and pipelines

As for now we have learned how to place values unmodified within templates. There are cases where we want to do something with the value before we place it in a resource. With functions and pipelines we can achieve exactly that.

When injecting strings like e.g. our favorite color in a template, we want to quote the strings:

```
{{ .Values.favorite.color }} --> blue
{{ quote .Values.favorite.color }} --> "blue"
```

The quote function therefore adds double quotes around the value. Functions follow the syntax `functionName arg1 arg2 ...`.

Helm has over 60 functions available. Some are defined in the [Go template language](#), others in the [Sprig template library](#).

Similar to Linux pipes known from shell commands, e.g. `ps -aux | grep ps`, you can use pipes in Go templates as well:

```
{{ .Values.favorite.color | upper | quote }} --> "BLUE"
{{ .Values.favorite.color | b64enc | quote }} --> "Ymx1ZQ==" # blue base64 encoded
{{ .Values.favorite.color | repeat 2 | quote }} --> "BLUEBLUE"
{{ .Values.favorite.band | default "Pink Floyd" | quote }} --> "Pink Floyd" # the our values.yaml doesn't consist of th
e favorite --> band value therefore the default value is rendered
{{ printf "%s%s" .Release.Name .Chart.Name | quote }} --> "release-namegotemplatechart"
```

## Task 5.6: Add functions and pipelines

Make the required changes to your ConfigMap template so that it renders to the following output:

```
name: myapp
namespace: default
data:
  mykey: myvalue
```

### Solution Task 5.6

```
name: myapp
namespace: default
data:
  mykey: myvalue
  mykey2: myvalue2
```

## Conditionals

If then else control structures are very common in templating languages like Go Templating and basically look like this

```
if condition {
  content
}
```

### Note: Conditional Operators

With `and`, `or`, `not`, `eq` being functions, conditions therefore look like this `{{ if and .Values.favorite.band (eq .Values.favorite.band "The Rolling Stones") }}` For this condition to be true, the value `.Values.favorite.band` must be set and is set to "The Rolling Stones"

## Task 5.7: Add a condition

Let's add an example condition to our ConfigMap:

- When the favorite drink is water, coke, beer or wine, add a new line to the ConfigMap data part `glass: true`
- When its coffee or tea: `mug: true`

We start with adding the favorite drink to your `values.yaml`

```
favoriteDrink: water
```

Now edit the ConfigMap template accordingly.

### Solution Task 5.7

```
name: myapp
namespace: default
data:
  mykey: myvalue
  mykey2: myvalue2
  mykey3: myvalue3
```

- acend gmbh

This condition is mostly unreadable due to the fact that we need to make sure the spaces for the next key and value set are correct.

Check out [Helm's documentation about controlling whitespace](#) for more details on how to control whitespaces and adapt your ConfigMap.

A more readable version could look like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  my-key: my-value
```

The `with` Operator allows you to set the current scope ( `.` ) to a particular object, in our case the favorite object.

## Loops in Helm templates

The `range` Operator allows you to implement loops in Helm templates. E.g. to iterate over a list of cities:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  my-key: my-value
```

We can implement the template as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  my-key: my-value
```

Check out [Helm's documentation about developing templates](#) for more details on templating with Helm.

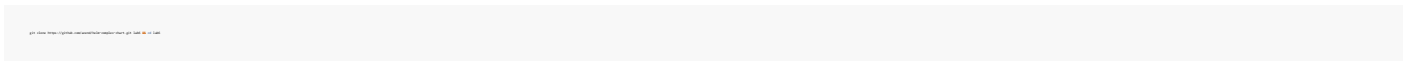
## 6. Your own complex Helm chart

Let's create a bit more complex Chart.

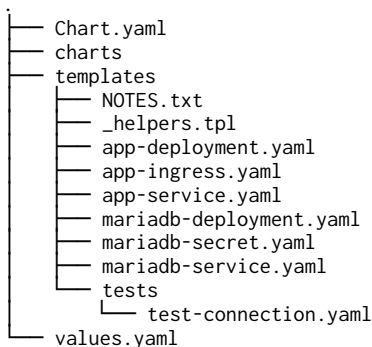
In this Lab we are covering following topics:

- 6.1 Deploy the Chart and add a database to the app
- 6.2 Template the database resources
- 6.3 Deploy the same app with a database dependency instead of your own database deployment
- 6.4 Writing tests for your Helm Charts
- 6.5 Deploy you app with Helm hooks
- 6.6 Prepare and deploy your app for different environments
- 6.7 Publish your Chart

We already prepared a Helm chart skeleton for this, you can clone the Chart with following command:



Let's have a closer look at its directory structure and components. The Chart consists of the following files and folders:



Looking at the `templates/` directory, we notice that there already are a few files:

- `NOTES.txt` : The “help text” for your chart which will be displayed to your users when they run `helm install`
- `app-deployment.yaml` : A basic manifest for creating a Kubernetes deployment
- `app-service.yaml` : A basic manifest for creating a service endpoint for your deployment
- `app-ingress.yaml` : A basic manifest for creating a ingress to expose your app deployment
- `_helpers.tpl` : A place to put template helpers that you can re-use throughout the chart
- `tests/` : A directory to put test files for testing the deployed Helm chart

Furthermore there are some resources (prefixed with `mariadb-`) for a database backend which we're going to use in the next lab section. For now you can ignore them.

### Note

For details on chart templating, check out the [Helm's getting started guide](#) .

## values.yaml





- acend gmbh

Study the [Helm documentation about the Chart.yaml file](#) , then change the description to `My awesome app` and add yourself to the list of maintainers.

## Solution

```
Chart.yaml
name: my-awesome-app
description: My awesome app
maintainers:
- name: your-name
  email: your-email@your-domain.com
version: 0.1.0
```

Continue with the lab “[Deploy your awesome application](#)”.

## 6.1 Deploy your awesome application

Using the generated and modified Helm chart, we are going to deploy our own awesome application.

### Task 6.1.1: Deploy the chart

Let's deploy our awesome application. Therefore we need to adjust ingress configuration in the values file.

```
values.yaml
ingress:
  enabled: true
  host: training.openshift.ch
```

## Solution

### Note

Don't forget to replace `training.openshift.ch` with the value provided by the trainer.

To create a release from our chart, we run the following command within our chart directory:

```
helm install myapp ./
```

This will create a new release with the name `myapp` . If we already had installed a release and wanted to update the existing one, we'd use the following command:

```
helm upgrade myapp ./
```

Check whether the ingress was successfully deployed by accessing the URL `http://helm-complex-chart-  
<namespace>.training.openshift.ch/`

### Task 6.1.2: Connect to the database

In order for the python application to be able to connect to a database, we need to add some environment variables to our deployment. The goal of this task is to allow a user to set them via values. Change your

- acend gmbh

Application Deployment Template in `templates/app-deployment.yaml` and include the new environment variables:

- `MYSQL_DATABASE_NAME` with the value `acenddb`
- `MYSQL_DATABASE_PASSWORD` references the database password value from the secret you created in task 1
- `MYSQL_DATABASE_ROOT_PASSWORD` references the the database root password value from the secret you created in task 1
- `MYSQL_DATABASE_USER` with the value `acend`
- `MYSQL_URI` with the value `mysql://$(MYSQL_DATABASE_USER):$(MYSQL_DATABASE_PASSWORD)@{{ .Release.Name }}-mariadb/$(MYSQL_DATABASE_NAME)`

And enclose all environment variables in a conditional If-statement with following condition: `database.enabled == true`

## Solution Task 6.1.2

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: acend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: acend
  template:
    metadata:
      labels:
        app: acend
    spec:
      containers:
      - name: acend
        image: acend/acend:1.0
        ports:
        - containerPort: 80
        env:
        - name: MYSQL_DATABASE_NAME
          value: acenddb
        - name: MYSQL_DATABASE_PASSWORD
          valueFrom:
            secretKeyRef:
              name: acend-secret
              key: password
        - name: MYSQL_DATABASE_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: acend-secret
              key: root-password
        - name: MYSQL_DATABASE_USER
          value: acend
        - name: MYSQL_URI
          value: mysql://$(MYSQL_DATABASE_USER):$(MYSQL_DATABASE_PASSWORD)@{{ .Release.Name }}-mariadb/$(MYSQL_DATABASE_NAME)
      serviceAccountName: acend
```

Add the following changes in the `values.yaml` to enable the database:

```
database:
  enabled: true
```

Enabling this property will render the environment variable block in the deployment template, as well all the `mariadb-*.yaml` templates. To upgrade your existing release run:

```
helm upgrade --install acend acend --values values.yaml
```

## Task 6.1.3: Check

Check whether the attachment of the new backend worked by either looking at the Pod's logs: In there the application tells you which backend it uses, this should of course be the database. Or simply access the application in your browser, create an entry, re-deploy the application Pod (e.g. by scaling it down and up again) and check if your entry is still there.

## Solution Task 6.1.3

First we need to execute the following command to determine the pod name

```
helm list --output yaml | grep -A 100 acend
```

- acend gmbh

```
2022-01-11 10:10:10.123 [INFO] [main] Application started successfully
```

Next execute the following command to show the logs

```
cat /var/log/your-app.log
```

As you can see in the log output, our application is now connected to the freshly deployed database

Never log sensitive informations like database connection strings which contain the password in plain text!

```
2022-01-11 10:10:10.123 [INFO] [main] Application started successfully
2022-01-11 10:10:10.123 [INFO] [main] Database connection established
2022-01-11 10:10:10.123 [INFO] [main] Application started successfully
```

## Task 6.1.8: Clean up

Uninstall the app

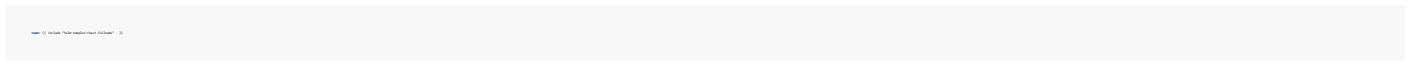
```
rm -rf /var/log/your-app.log
```

## 6.2 Template the database backend

In this lab we are going to templates the resources that are necessary to deploy a MariaDB database as a backend to our `example-web-python` application. Before we start creating those templates we want to have a look at a couple of best practices.

### Resource names

When looking at the app templates of our `mychart` chart, the name of the resource is always defined by a helper function:



Resources within the same Namespace and of the same resource type must have unique names. Since a Helm chart can be instantiated multiple times in different releases, it's best to include the release name as part of the resource name:

```
<releasename>-<chartname>
```

Have a look at the helper function in `mychart/templates/_helpers.tpl` for more details and keep this concept in mind when creating new templates.

### Labels

There are a couple of [recommended Kubernetes labels](#) (that also apply to OpenShift), which help to make interoperability between different tools and concepts easier and define a kind of standard.

We use the following two labels as so-called selector labels. These are labels that are used on services to define which pods belong to them:

- `app.kubernetes.io/name` : Resource name
- `app.kubernetes.io/instance` : Release name

And a couple of additional ones to label our resources with supplemental information:

- `helm.sh/chart` : Chart name, e.g. `mychart-0.1.0`
- `app.kubernetes.io/version` : Chart version, e.g. `1.16.0`
- `app.kubernetes.io/managed-by`: Helm

Similar to the resource name we can also use helpers to generate the necessary labels. There are two helper functions you can use to generate these labels

- `{{- include "helm-complex-chart.selectorLabels" . }}` to generate the so-called selector labels. Primary you can use this function in Service templates under `.spec.selector`
- `{{- include "helm-complex-chart.labels" . }}` to generate the Helm common labels. Mainly used for the `.metadata.labels` in any resource template.

### Task 6.2.1: Add a new helper function

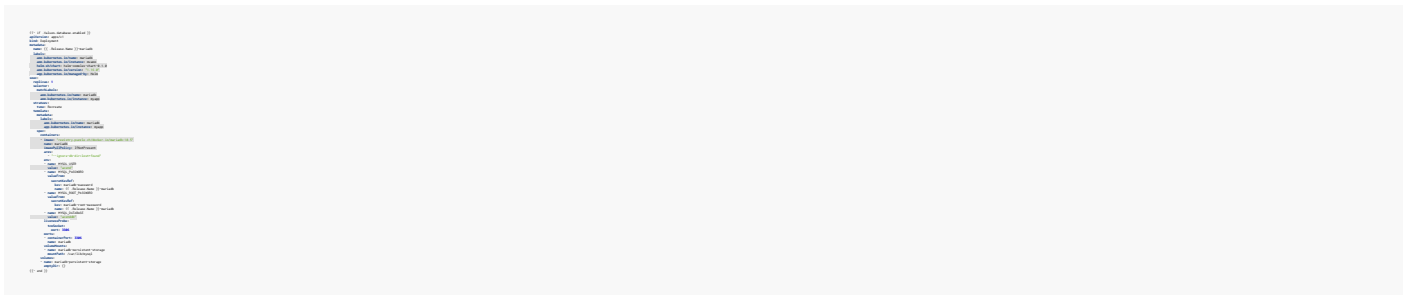
As mentioned above, the `_helpers.tpl` file provides some neat functions to generate the Labels and Selectors for Kubernetes resources. But if we take a closer look, we see that we can not simply use the same labels and selectors. Because we have two different deployments, each with its own service, we need to ensure



- acend gmbh

Let's examine the MariaDB Deployment template first. As we can see, the deployment of the MariaDB is very similar to the App Deployment. Make following changes to create a reusable template

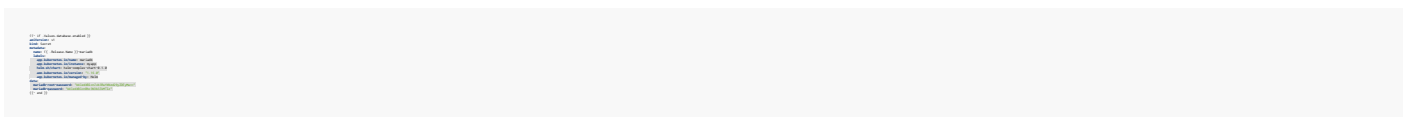
- Replace `metadata.labels` with the template function `helm-complex-chart.labelsDatabase` we created before
- Replace `spec.selector.matchLabels` with the template function we created before
- Replace `spec.template.metadata.labels` with the template function `helm-complex-chart.labelsDatabase` we created before
- Replace `spec.template.spec.containers[0].image` with
- Replace `spec.template.spec.containers[0].imagePullPolicy` with
- Replace under `spec.template.spec.containers[0].env` the values from following keys
  - `MYSQL_USER` value with `.Values.database.databaseuser`
  - `MYSQL_DATABASE` value with `.Values.database.databaseuser`



## Secret

Furthermore you have already a Secret `mariadb-secret.yaml` which contains the passwords for the Database. Make following changes to create a reusable template

- Replace `metadata.labels` with the template function `helm-complex-chart.labelsDatabase` we created before
- Replace under `data` the values from following keys
  - `mariadb-root-password` with **base64 encoded** `.Values.database.databaseuser` value
  - `mariadb-password` with **base64 encoded** `.Values.database.databaseuser` value



When creating the template files, make sure that a user can specify the `mariadb-password` and `mariadb-root-password` from the secret using a variable.

## Service

To connect to the database, we also have a service. Make the following changes to create a reusable template

- Replace `metadata.name` with
- Replace `metadata.labels` with the template function we created before
- Replace `metadata.spec.selector` with the template function we created before



When your changes are ready, upgrade the already deployed release with the new version.

## 6.2.2: Solution

The template file for the MariaDB database `templates/deployment-mariadb.yaml` could look like this.

The following points need to be taken into consideration when creating the template:

- The helper `helm-complex-chart.fullname` will return `release-helm-complex-chart`. Since our first deployment for the `example-web-python` application already uses this name, we have to choose a name for the `mariadb` instead.
  - Let's take `<releasename>-mariadb` instead. As an alternative, we could also alter the full name helper to accept an additional name, which would be different for each deployment.
- The same applies to the label `app.kubernetes.io/name`. We can't therefore use the included `helm-complex-chart.labels`. We could also alter the helper function or in our case simply just add the labels directly.
- In the deployment templates we reference our secrets by again using the full name `<releasename>-mariadb`.

## Deployment



## Secret

The secret `templates/mariadb-secret.yaml` file should look like this:



Note the `| b64enc`, which is a built-in function to encode strings with base64.

## Service

The service at `templates/mariadb-service.yaml` for our MySQL database should look similar to this:

- acend gmbh

```
cat < /dev/null > /dev/null
```

## Values

Then we need to add the following to our `values.yaml` :

```
values:
  - name: acend
    namespace: acend
```

## Install

Finally, to install the release run:

```
helm install --namespace acend acend
```

## Task 6.2.5: Cleanup

If you're happy with the result, clean up your namespace:

```
helm uninstall acend --namespace acend
```

## 6.3 Backend as a Dependency

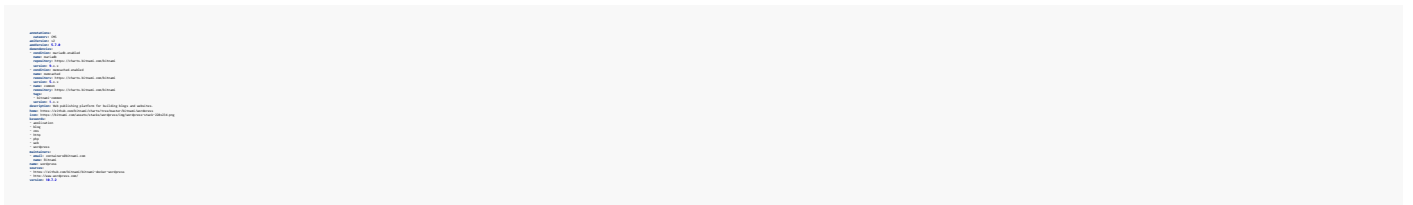
In the previous lab we've manually added a database deployment to our awesome application. Instead of creating your own templates and deployments, we can also integrate other helm charts as dependencies. We have seen this already in lab 3 but are now going to go into more detail in this lab.

### Helm Dependencies

Helm dependencies allow you to integrate other helm charts within your helm chart. It helps to reduce code duplication and to centralize certain functionality and to simply use well maintained helm charts in your deployments.

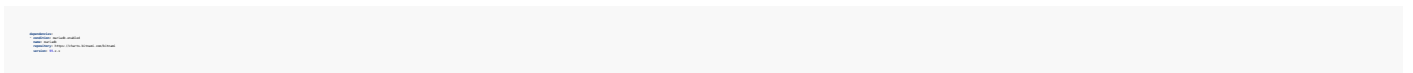
Dependencies are managed in the Chart.yaml, the dependencies itself are stored in the `charts/` directory of your chart.

In lab 3 we had a look at the wordpress chart, which has a mariadb dependency.



### Conditions

Conditions can be used to enable and disable dependencies. For example, if your dev environment has a built-in memory database and only your test and prod environment use the mariadb from the dependency.



Check [this link](#) to learn more about the concept of conditions and tags.

### Choosing Helm Chart as Dependencies

When adding dependencies to your helm charts there are a couple of things you should consider before doing so.

- Be aware of the fact to your helm chart is depending on something you might not have under your control.
- Make sure the helm chart you use as dependency is well maintained, updated regularly and the app version is very close to the actual version of the deployed application.
- Explore the chart, especially when using third party helm charts
  - general architecture
  - Functionality: Does the chart really implement all the features you need.
  - Security: Check the default images, do they run as root, the security context of pod and filesystem, Service accounts
  - Resource Limits and Requests
- Study default values in the values.yaml
- If the chart doesn't match your needs perfectly, consider implementing the functionality yourself.

- acend gmbh

- Reduce the number of dependencies.
- Update your dependencies regularly.

## Task 6.3.1: Use the mariadb bitnami chart as dependency

Let's now replace the mariadb backend, we've manually created in the previous lab with a mariadb chart dependency.

Delete the three templates from the previous lab.

- `templates/service-mariadb.yaml`
- `templates/deployment-mariadb.yaml`
- `templates/secret-mariadb.yaml`

Then we need to add the dependency to the `Chart.yaml`

```
dependencies:
- name: mariadb
  repository: https://charts.bitnami.com/bitnami
  version: 10.1.0
```

Since we already added the bitnami repository in lab 3 ( `helm repo add bitnami https://charts.bitnami.com/bitnami` ), after we added the dependency to the `Chart.yaml`, we can simply run

```
helm dependency update
```

This will download the dependent chart to the `charts/` and update the `Chart.lock` file to the latest matching version according to the version defined in the `Chart.yaml`. The `Chart.lock` contains the exact version of the moment the dependencies were updated, and it's use to recreate that exact version combinations. In order to update the dependency versions of the `Chart.lock` file you can simply run `helm dependency update`.

Next we have to add the configuration of our mariadb dependency to the `values.yaml`. Since the name of our dependency in the `Chart.yaml` is `mariadb`, the configuration of the mariadb must be defined under the `mariadb` root element. Let's add the following configuration right after the database section from lab 6.2

```
mariadb:
  auth:
    database: mariadb
    username: mariadb
    password: mariadb
```

Update your deployment to match the keys in the `values.yaml` to your environment variables defined in the deployment:

- acend gmbh

```
helm upgrade --install mariadb bitnami/mariadb --namespace training --values values.yaml --wait
```

After editing the files we can now install the release.

```
helm upgrade --install mariadb bitnami/mariadb --namespace training --values values.yaml --wait
```

Verify the installation and check whether the new database was deployed.

### Note

The whole deployment will take a while until both pods are ready and deployed.

To check if your deployment is working open the browser and enter the url of your app `https://helm-complex-chart-<namespace>-training.openshift.ch` : Add some new entries and then execute the following command to restart the application pod.

```
helm upgrade --install mariadb bitnami/mariadb --namespace training --values values.yaml --wait
```

After the new pod is created, you can reload the website and you should still see the persisted entries.

## Task 6.3.2: Explore the bitnami mariadb chart

Use the `--dry-run` option or the `template` command to have a look at the new k8s resources introduced by the dependency. Explore the [chart source code](#) and have a look at all the possible [configuration options](#).

## Task 6.3.3: Cleanup

If you're happy with the result, clean up your namespace:

```
helm delete mariadb --namespace training
```



## 6.5 Helm hooks

In the previous lab we learned how to deploy our python example application and connect it to a MariaDB. In this chapter we are going to look at a mechanism of Helm called hooks, which let's chart developers intervene / interact at certain points in a release's life cycle.

### Task 6.5.1: Chart Hooks

At some point when developing helm charts we would like to interact or alter certain behaviour of our deployed resources. For example in this usecase we would like to have test data available in our database when releasing our chart. With the hook functionality of helm we can alter the life cycle and intervene at certain points. Hooks are regular templates specially annotated that causes Helm to utilize them differently. The tests used in the previous chapter are a special case of helm hooks. The following hooks are available for you to alter the behaviour of the chart:

- **pre-install:** Executes after templates are rendered, but before any resources are created in Kubernetes
- **post-install:** Executes after all resources are loaded into Kubernetes
- **pre-delete:** Executes on a deletion request before any resources are deleted from Kubernetes
- **post-delete:** Executes on a deletion request after all of the release's resources have been deleted
- **pre-upgrade:** Executes on an upgrade request after templates are rendered, but before any resources are updated
- **post-upgrade:** Executes on an upgrade request after all resources have been upgraded
- **pre-rollback:** Executes on a rollback request after templates are rendered, but before any resources are rolled back
- **post-rollback:** Executes on a rollback request after all resources have been modified
- **test:** Executes when the Helm test subcommand is invoked

Hooks are controlled and configured via kubernetes annotations. For example to configure the life cycle the hook should be hooked into we use the annotation:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    helm.sh/hook: pre-install,pre-upgrade,pre-rollback
    helm.sh/hook-weight: 5
spec:
  selector:
    app: myapp
  ports:
    - port: 80
```

The resource can even implement multiple hooks:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    helm.sh/hook: pre-install,pre-upgrade,pre-rollback
    helm.sh/hook-weight: 5
    helm.sh/hook-delete-policy: before-hook-runs
spec:
  selector:
    app: myapp
  ports:
    - port: 80
```

When working with multiple hooks hooking into the same life cycle stage, we can alter the execution or runtime behaviour by giving them different weights with the annotation:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    helm.sh/hook: pre-install,pre-upgrade,pre-rollback
    helm.sh/hook-weight: 5
    helm.sh/hook-delete-policy: before-hook-runs
spec:
  selector:
    app: myapp
  ports:
    - port: 80
```

Hook weights can be positive or negative numbers but must be represented as strings. When Helm starts the execution cycle of hooks of a particular Kind it will sort those hooks in ascending order.

### Task 6.5.2: Example hook

- acend gmbh

In this example we would like to populate our database with some test data before finishing our install / upgrade life cycle. We can use a helm hook to inject our logic after installing or upgrading our helm release. Create the following hook `templates/hook.yaml` :

```
apiVersion: batch/v1
kind: Job
metadata:
  annotations:
    "helm.sh/hook": post-install,post-upgrade
    "helm.sh/hook-weight": "5"
  name: post-install-hook
spec:
  template:
    metadata:
      name: post-install-hook
    spec:
      containers:
        - name: post-install-hook
          image: busybox
          command:
            - sleep
            - "10"
      restartPolicy: Never
```

This basic hook creates a kubernetes Job which sleeps for `<.Value.sleepyTime>` or 10 seconds after the installation life cycle of your helm chart.

Notice the `metadata.annotations` block which tells Helm how the hook should interact during the release's life cycle:

```
apiVersion: batch/v1
kind: Job
metadata:
  annotations:
    "helm.sh/hook": post-install,post-upgrade
    "helm.sh/hook-weight": "5"
  name: post-install-hook
spec:
  template:
    metadata:
      name: post-install-hook
    spec:
      containers:
        - name: post-install-hook
          image: busybox
          command:
            - sleep
            - "10"
      restartPolicy: Never
```

Let's install the release and check whether the hook was deployed. First open a second terminal and run the following command:

```
oc get pod
```

The output of the `oc get pod` command should show the deployment of the `post-install-hook` pod.

## Task 6.5.3: Write your own hook

It's time to get our hands dirty! Write two hooks to alter the release's life cycle:

- Create a post-install/post-upgrade hook to create a database table `test` and populate it with data of your favor. In the example solution we create a table `test` with two fields: An autoincrement index `id` and a char field called `name` to store some strings. Ensure this job is running after the job below.
- Create a post-install/post-upgrade hook to drop the table `test` if it exists to have a clean state before releasing. Ensure this job is running before the job above.

To interact with the MariaDB deployed you can use the image `registry.puzzle.ch/docker.io/mariadb:10.5` for example. You can interact and access the database with `$ mysql --host=<hostname> --user=<username> --password=<password> --database=<database> -e <sqlCommand> .`

First Post-install / -upgrade hook:



## 6.6 Deploy your Chart with different environment configurations

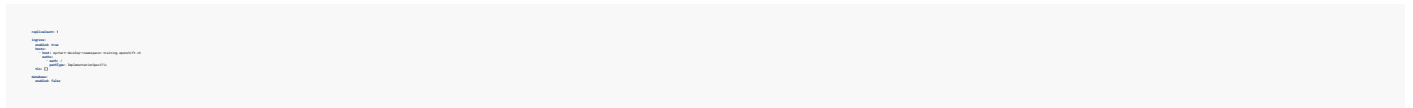
One of the most typical challenges when deploying an application is the handling of different deployment environments.

A typical use case is to deploy your application into a Development and Production environment. Helm is a great solution for handling this challenge. In this lab we are going to show you how to prepare your Helm releases for different environments.

### Task 6.6.1 Create a development release

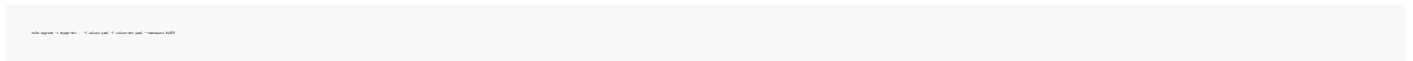
Let us start with a new value file for the Development environment. Create a new file called `values-dev.yaml` and add following content to the file

- Set the number of replicas to `1`
- Change the Ingress configuration to match the following path schema `http://mychart-develop-  
<namespace>.training.openshift.ch/`
- Because we don't need any persistence for the Development environment, disable the database with `database.enabled: false`



You can specify the `'-values'/'-f'` flag multiple times. The priority will be given to the last (right-most) file specified. For example, if both `myvalues.yaml` and `override.yaml` contained a key called `'Test'`, the value set in `override.yaml` would take precedence.

Now install the development release with the following command. First pass the `values.yaml` file which contains all the default values. Then pass the `values-dev.yaml` as second file argument. So the `values-dev.yaml` will overwrite the default Chart values.



### Task 6.6.2 Create a production release

Create for the Production environment a new value files named `values-prod.yaml` and add the following settings to the file

- Due to the higher load in our production environment, we change the number of replicas to `3`
- For production usage we also want to persist the data. To enable the database set `database.enabled` to `true`
- Change the database user under `database.databaseuser` to `acend-prod`
- As a best practice, never use the same password in different environments. Change the password under `database.databasename`
- Change the Ingress configuration to match the following path schema `mychart-production-  
<namespace>.training.openshift.ch`

## Task 6.6.3 Solution

The `values-prod.yaml` should look as follow:

```
values:
  app:
    name: my-app
    namespace: default
    replicas: 1
    image: my-image
    imagePullPolicy: Always
    service:
      type: ClusterIP
    ingress:
      enabled: true
      annotations:
        kubernetes.io/ingress.class: nginx
      hosts:
        - host: my-app.example.com
          paths:
            - path: /
              pathType: Prefix
      tls:
        enabled: true
        secretName: my-app-tls
  database:
    enabled: true
    type: postgres
    version: 12
    image: postgres
    imagePullPolicy: Always
    service:
      type: ClusterIP
    ingress:
      enabled: true
      annotations:
        kubernetes.io/ingress.class: nginx
      hosts:
        - host: my-app.example.com
          paths:
            - path: /db
              pathType: Prefix
      tls:
        enabled: true
        secretName: my-app-tls
  redis:
    enabled: true
    type: redis
    image: redis
    imagePullPolicy: Always
    service:
      type: ClusterIP
    ingress:
      enabled: true
      annotations:
        kubernetes.io/ingress.class: nginx
      hosts:
        - host: my-app.example.com
          paths:
            - path: /redis
              pathType: Prefix
      tls:
        enabled: true
        secretName: my-app-tls
```

Install the production release with the following command. Again pass first the default values with `-f values.yaml` and then the production values with `-d values-prod.yaml`

```
helm upgrade --install my-app my-app --values values.yaml --values values-prod.yaml
```

## Task 6.6.8: Clean up

Uninstall the app

```
helm uninstall my-app
```

## 6.7 Optional: Publish your chart

In this lab we will learn how to publish our Helm chart and make it accessible to the public by using GitHub pages.

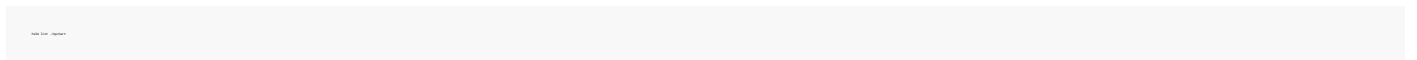
### Note

To work through this lab, you need a personal GitHub account. See [Hosting chart repositories](#) for a lot of different strategies how to serve a Helm repository.

GitHub pages provides an easy way to expose static files over HTTP(S) to the public. Everyone with a GitHub account can use this feature. The files to expose must be located in the `docs/` subdirectory or on a separate branch which can be configured in the repository settings.

### Task 67.1: Linting your chart

Linting the Helm chart and fix the errors before packaging and publishing is a good practice:



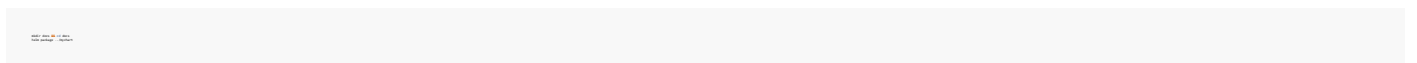
The linter recommends to add an icon to the chart. We can safely ignore this hint :)

```
==> Linting mychart
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

### Task 67.2: Package your chart

Helm chart packages are compressed `tgz` archives. Use the `helm package` command to create such an archive:



This will create an archive with the name `mychart-0.1.0.tgz` inside the `docs` directory. The chart version is defined in `Chart.yaml` and will be appended to the filename.

- acend gmbh

```
> tree
├── docs
│   └── mychart-0.1.0.tgz
├── mychart
│   ├── Chart.yaml
│   ├── charts
│   ├── templates
│   │   ├── deployment-mariadb.yaml
│   │   ├── deployment.yaml
│   │   ├── _helpers.tpl
│   │   ├── hpa.yaml
│   │   ├── ingress.yaml
│   │   ├── NOTES.txt
│   │   ├── secret.yaml
│   │   ├── serviceaccount.yaml
│   │   ├── service.yaml
│   │   └── tests
│   │       └── test-connection.yaml
│   └── values.yaml
```

## Task 67.3: Create a new Github Repository

We will use GitHub pages to publish our packages and make them accessible to the public.

Replace `<GITHUB_USERNAME>` with your personal GitHub account.



```
git init
git add .
git commit -m "initial commit"
git remote add origin https://github.com/<GITHUB_USERNAME>/mycharts.git
git push -u origin main
```

### Note

If you want to access your Git repository over HTTPS use the following url pattern for the origin  
`https://github.com/$USERNAME/mycharts.git`

## Task 67.4: Activate GitHub Pages

On the GitHub website navigate to `Settings -> Pages` of your newly created repository `mycharts`. Select the `main` branch and `/docs` repository and click `Save`.

You will see a message with the URL under which the Chart repository will be exposed:

```
https://<GITHUB_USERNAME>.github.io/mycharts/
```

## Task 67.5: Create an index file

The Helm repository must provide an `index.yaml` file which list all contained packages and provide some metadata about them. Create an index file inside the `docs` directory. Replace `<GITHUB_USERNAME>` with your personal GitHub account.



```
cat > docs/index.yaml
```

This creates an index file with the following content:

- acend gmbh

```
helm repo add mycharts https://github.com/acend/helm-charts
```

## Task 67.6 Verifying your repository

Register your GitHub pages as a new Helm repository:

```
helm repo add mycharts https://github.com/acend/helm-charts
```

Verify that your published chart is found by Helm:

```
helm search mycharts
```

You should the the following output:

NAME	CHART VERSION	APP VERSION	DESCRIPTION
mycharts/mychart	0.1.0	1.16.0	A Helm chart for Kubernetes

## Task 67.7 Installing chart from repository

When your newly created repository is successfully registered you can install the published chart with the following command:

```
helm install mychart mycharts/mychart
```

## Task 67.8: Clean up

Uninstall the app

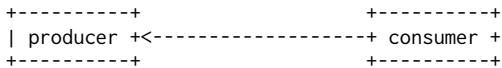
```
helm uninstall mychart
```

The repository can be removed by using

```
helm repo remove mycharts
```

## Optional: 7. Create your own chart

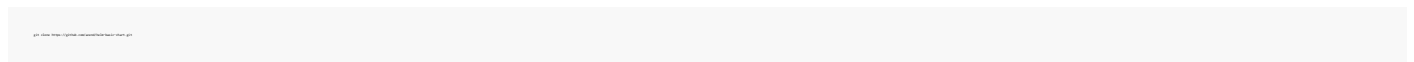
In this section we are going to show you how to modify a Helm chart from an existing Kubernetes deployment. To provide you an easy entry point, we have already prepared a Helm chart skeleton. This chart contains all necessary template files. With this chart, we want to deploy two java microservices, one which produces random data when its REST interface is called. The other microservice consumes then the data and exposes it to its endpoint.



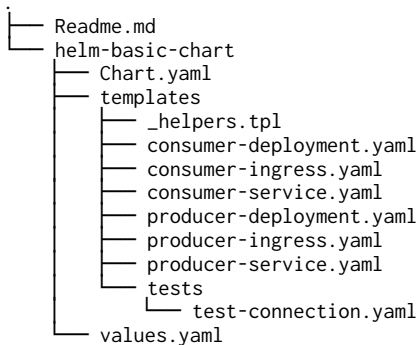
Imagine you're being a java development team embracing the DevOps culture and your job is to deploy the two services you designed to your Kubernetes cluster.

### Task 7.1: Get the chart skeleton

Clone the repository we have prepared for you containing all the needed template files to get you started.



After cloning the chart you have following structure:



The template files are always prefixed with the according application (e.g. `consumer-deployment.yaml` and `producer-deployment.yaml`) and suffixed with the resource they represent. This is a best approach we like to follow in order to keep our files ordered by the applications and for readability purposes.

The setup is quite simple here, for each application we have a deployment, a service and an ingress. If you inspect the files in the folder you will realize that we have a lot of hard coded properties we maybe would like to change in order to bring the applications to multiple environments.

### 7.1 Refactor 1

#### Task 7.1.1: Make the Ingress resource more configurable

- acend gmbh

After cloning the repository and inspecting the templates already created for you, you will notice some potential for improvement. For example the hostname of the applications should not be hard-coded. When deploying to multiple environments you will run into conflicts.

Modify the **consumer and producer** ingress templates and extract following variables to make them configurable:

- Extract `.spec.rules.host` as value
- Extract `.spec.tls.hosts[0]` as value, use the same value as above

## Note

If you want to test the chart locally you can execute following command

```
helm template -s templates/consumer-ingress.yaml ./helm-basic-chart --debug | cat -n -
```

First let us define the new variables in our `values.yaml` file. Replace `<username>` with your username

```
username: <username>
```

Next replace the hard coded values for the host value in our `consumer-ingress.yaml` file.

```
spec:
  rules:
    - host: <host>
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              serviceName: <service>
              servicePort: <port>
```

Replace the same value in our `producer-ingress.yaml` file.

```
spec:
  rules:
    - host: <host>
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              serviceName: <service>
              servicePort: <port>
```

Afterwards we can install our Helm Chart with following command.

```
helm install --name <name> --namespace <namespace> ./helm-basic-chart
```

Verify your deployment! Check if your pods are running and healthy!

```
helm status <name>
```

This should return something like this:

- acend gmbh

```
oc get pods
NAME                                READY   STATUS    RESTARTS   AGE
data-consumer-7686976d88-2wbh5     1/1    Running   0           72s
data-producer-786d6bb688-qpg4c     1/1    Running   0           72s
```

Pay attention to the `1/1` status in the ready section! After checking if our pods are ready and therefore will accept traffic from the service, check if the application is running correctly:

Both applications (data-producer and data-consumer) you just deployed expose an endpoint `/data` which will return a random double data json. Try to verify if your deployment is running correctly by using curl.

If you are struggling with curl you can use the following syntax:

```
curl -s -X GET http://localhost:8080/data
```

When the problem will be a redirect or certificate problem, try the flags `-L` and `-k` to mitigate the error.

```
curl -s -X GET http://localhost:8080/data -L -k
```

If your application returns the data point when consuming the consumers `/data` endpoint, then both applications work.

## Task 7.1.2: Make the deployments more configurable

Not just the ingresses could use some improvements. Also the deployments could be more configurable. In order to keep up to date with the current requirements your task is to adapt the following things:

Producer Deployment:

- Extract the image tag from the `.spec.containers[0].image` on Line 22 field as value
- Extract the `.spec.containers[0].resources` block from line 51 as value `consumer.resources`. Make use of the `toYaml` and the `nindent` function.
- Extract the `.spec.containers[0].env["QUARKUS_LOG_LEVEL"]` on line 26 block as value

Consumer Deployment:

- Extract the `.spec.containers[0].resources` block from line 51 as value `producer.resources`. Make use of the `toYaml` and the `nindent` function.
- Extract the `.spec.containers[0].env["QUARKUS_LOG_LEVEL"]` on line 26 block as value `producer.logLevel`

### Note

Take a look at the official Helm documentation for a list of built in functions.

[Built In Helm functions](#)

producer-deployment.yaml

- acend gmbh

```
consumer-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: consumer
  template:
    metadata:
      labels:
        app: consumer
    spec:
      containers:
      - name: consumer
        image: acendgmbh/consumer:1.0.0
        ports:
        - containerPort: 8080
```

## consumer-deployment.yaml

```
consumer-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: consumer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: consumer
  template:
    metadata:
      labels:
        app: consumer
    spec:
      containers:
      - name: consumer
        image: acendgmbh/consumer:1.0.0
        ports:
        - containerPort: 8080
```

## values.yaml

```
values.yaml
image:
  repository: acendgmbh/consumer
  tag: 1.0.0
```

## Task 7.1.3: Upgrade the chart

Execute following command to update our helm release.

```
helm upgrade --install consumer consumer-chart --values values.yaml
```

Finally, you can visit your application with the URL provided from the Route: `https://consumer-  
<username>.training.openshift.ch/data`

Replace **<username>** with your username or get the URL from your route.

Or you could access the `data` endpoint using curl:

```
curl -X GET https://consumer-username.training.openshift.ch/data
```

When you open the URL you should see the producers data

If you only see `Your new Cloud-Native application is ready!`, then you forgot to append the `/data` path to the URL.

## Task 7.1.4: Prepare another release

At this point we have a configurable Helm chart and a running release. Next we gonna use the cart for another release. We consider to release it into a production environment. therefore we have to adjust some values. First copy the existing `values.yaml` to `values-production.yaml`.



Open the `values-production.yaml` and change following values.

- Debug log level is too high in a production environment, change it to `INFO`
- The resource requirements are usually higher in a production environment than in a development environment. Increase the Memory Limits to `750Mi`
- To avoid DNS collisions we need to change the host to, change it to `producer-<username>-prod.training.openshift.ch` and `consumer-<username>-prod.training.openshift.ch`

### values-production.yaml



## Task 7.1.6: Install and verify production release

Now we have prepared our values file for the production environment. Next we can install the chart again, but with a different name and different values. Execute the Helm install command and pass the new created production values as parameter.



Use the `helm list` command to list all releases in your namespace



You should see following output with de development and the production release

- acend gmbh

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART
myrelease	default	1	2022-05-19 13:26:56.278026261 +0200 CEST	deployed	helm-ba
myrelease-prod	default	1	2022-05-19 13:26:36.570013792 +0200 CEST	deployed	helm-ba

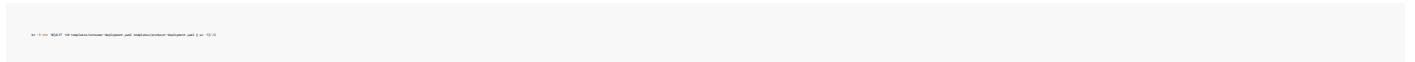
## Task 7.1.7: Cleanup

57 / 63

## 7.2 Refactor 2

If you take a closer look at your Chart you will still recognize some weak spots. For example the producer and consumer will look like a lot of code duplication. We don't like code duplication at all! The only big difference is the "consumer" or "producer" pre- or suffixed everywhere.

**Just for fun:** How much lines of code are actually different?



When considering the differences and how they affect the service, we can easily see the flaw of this Chart. The entire Chart is a duplication. Both services could use the same Chart and just be two instances / releases!

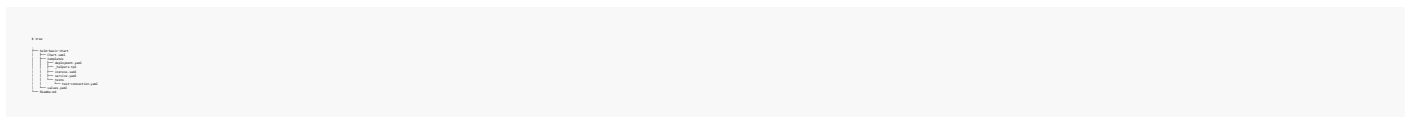
There are now two possibilities achieving the reduction of code duplication here: instantiation or composition.

### 7.2.1 Instantiate the Chart two times

The idea is simple: Instead of having a Chart consisting of two deployments, two services and two ingresses, reduce all resources to one! Eliminate the specifics in the variable names (if you like), if you're lazy you can just remove half of the Chart and continue.

#### Template files

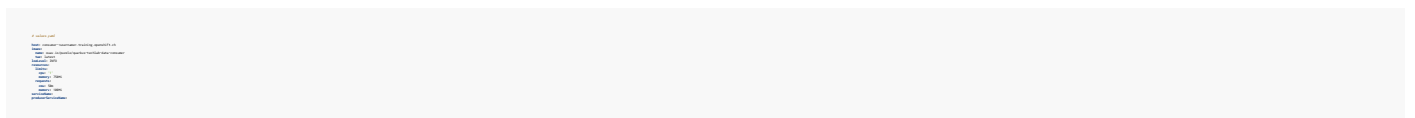
Start of by removing all the resources for the one of the two services and rename them by removing the prefix "producer" or "consumer". After doing so, your Chart's structure should look something like this:



#### Values

Update your variables by removing top most yaml-object "consumer" or "producer". So there is only one configuration for one service left in your `values.yaml` file. Add another value called `serviceName` to your `values.yaml`.

Your `values.yaml` should look like this (might differ if you deleted the consumer or producer part):



#### Deployment

Update your deployment, service and ingress and edit the values accordingly. So your `{{ .Values.producer.image.tag }}` will become `{{ .Values.image.tag }}`.

- acend gmbh

Change the hard-coded occurrences of `data-producer` or `data-consumer` in your templates to `{{ .Values.serviceName }}`. If you fancy you can remove the suffixes `-producer` or `-consumer` in your templates as well.

### deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: data-producer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: data-producer
  template:
    metadata:
      labels:
        app: data-producer
    spec:
      containers:
        - name: data-producer
          image: quay.io/puzzle/quarkus-techlab-data-producer
          ports:
            - containerPort: 8080
```

## Service

### service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: data-producer
spec:
  selector:
    app: data-producer
  ports:
    - port: 8080
```

## Ingress

### ingress.yaml:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: data-producer
spec:
  rules:
    - host: data-producer.training.openshift.ch
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: data-producer
                port:
                  number: 8080
```

## Task 7.2.2 Install release

Install the release with the configuration for the producer. As we learned in previous chapters, we can overwrite values from the `values.yaml` with the help of the `--set variable=value` parameter of the `helm-cli`. Overwrite the values for the producer like the following:

- `host` : `producer-<username>.training.openshift.ch`
- `image.name` : `quay.io/puzzle/quarkus-techlab-data-producer`
- `serviceName` : `producer`

Call the release `data-producer` and install it!

```
helm install data-producer ./data-producer --set host=producer-<username>.training.openshift.ch --set image.name=quay.io/puzzle/quarkus-techlab-data-producer --set serviceName=producer
```

After you installed the producer service you can verify the deployment if you'd like to be sure you did everything right!

Let's do the same thing and deploy the consuming service accordingly. Overwrite the following values for

- acend gmbh

the data-consumer microservice:

- host : consumer-<username>.training.openshift.ch
- image.name : puzzle/quarkus-techlab-data-consumer
- serviceName : data-consumer
- producerServiceName : producer-helm-basic-chart-producer

At the end, verify your two releases again and test if they are still delivering data as they did before!

## Task 7.2.3 Clean up

Uninstall the two releases again to have a fresh ground for the second option!

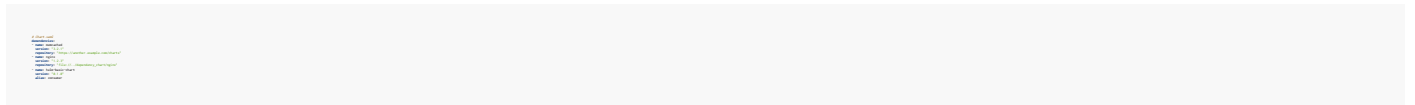
## 7.3 Composition

Instead of instantiating the Chart two times manually by hand, we can also define a composition of two Charts in a single Chart. In other 2 words we create a new Helm Chart and add the Chart we just wrote twice as a dependency to the chart.

In Helm we can make a composition of already existing Charts and own templates and define it as a new Chart. This is a very common use case if we mix self created resources with own templated resources.

How do we achieve this? Maybe first a bit of theoretical input:

We can declare dependencies in the `Chart.yaml` like the following:



In the example above we can see the `dependency` block of the `Chart.yaml`. We define three dependencies in the Chart and add `memcached`, `nginx` and `helm-basic-chart` as a dependency. As you can see the syntax varies a bit, let's check it for a second:

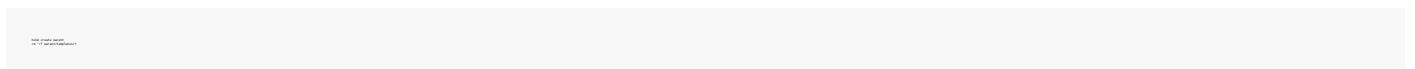
- `name` : The 'name' should be the name of a chart, where that name must match the name in that chart's 'Chart.yaml' file
- `version` : The 'version' field should contain a semantic version or version range
- `repository` : The 'repository' URL should point to a Chart Repository or to a Chart in your local filesystem
- `alias` : The 'alias' of a dependency (this is very handy to import the same Chart twice)

If we add a dependency without the repository defined, Helm will try to find the defined dependency in either your added repositories or the `/charts` directory in your Chart's directory.

So let's get some work done!

### Task 7.3.1 Create a new Chart

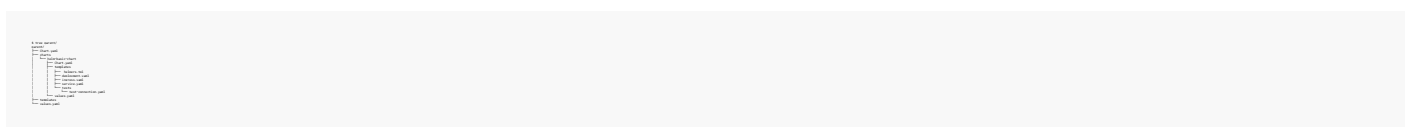
Create a new Chart `parent` and empty the templates folder, since we don't need any additional templates in the Chart other than the dependencies.



### Task 7.3.2 Copy the dependencies

Duplicate the Chart we created in the section before. Copy the duplicated Chart from the section before into your new Chart's `/charts` directory.

Your new Chart's structure should look like this:



### Task 7.3.3 Add the chart as dependency

- acend gmbh

Add two new dependencies to your parent Chart's dependencies with the aliases `producer` and `consumer`. Since we have the `helm-basic-chart` (or the name you chose in the dependency) in the local `/charts` directory, we don't need to add any `repository` in the dependency definition.

```
dependencies:
  - name: nginx
    version: 1.19.0
    repository: https://charts.acend.com
  - name: producer
    version: 1.0.0
    repository: https://charts.acend.com
  - name: consumer
    version: 1.0.0
    repository: https://charts.acend.com
```

## Task 7.3.4 Configure the values

When we add dependencies to your Helm Charts we can configure them in the parent's `values.yaml`. We can override configuration of dependencies in a parent chart by adding configuration in the `values.yaml` prefixed by the dependencies name or alias. For example if we have a dependency `nginx` in your `Chart.yaml` we can override the configuration of the `nginx's` `url` property like the following:

```
nginx:
  url: https://nginx.org
```

It is your task to configure the two dependencies so they will work just as they worked before!

In our example we added two dependencies: `consumer` and `producer`. We can define two blocks of configuration in the parent's `Chart.yaml`:

```
consumer:
  url: https://charts.acend.com
```

```
producer:
  url: https://charts.acend.com
```

## Task 7.3.5 Install the release

Install a Helm Chart release `myrelease` and verify if the two services are running correctly!

```
helm install myrelease
```

## Task 7.3.5 Verify the release

```
helm status myrelease
```

## Task 7.3.6 Clean up

Congratulations! You succeeded in the Chapter and the only thing left is to do some cleanup afterwards!

- acend gmbh

Remove all the installed objects installed during the chapter!

-----