Python Prework

IMC

January 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

I hope you are as excited as we are about you joining IMC, and we're looking forward to having you in our Software Engineering Global Traineeship (colloquially known as Dev School).

Before we start the traineeship together, we want to ensure that you have a solid understanding of certain widely used open-source technologies. We have a limited amount of time available for the traineeship, so we want to be able to use that time for working directly with instructors to learn advanced topics and IMC-specific tooling. Therefore, it is essential that you spend time familiarizing yourself with these technologies prior to the start of the program. The attached documents describe the contents that you should focus on.

While we will provide you with recommended learning resources as a starting point, please note that independent research and exploration are expected from each of you. Self-directed learning is a crucial skill for success at IMC, and this is an opportunity to practice that!

For each technology, we have compiled a set of recommended learning resources that will serve as a great starting point for your studies. We have found these to be good introductory resources for covering fundamental concepts and providing practical examples to help grasp the essentials. If they do not fit your learning style, you are encouraged to find your own resources on Google, YouTube, or elsewhere. If you are already familiar with any material, feel free to skip the specific content.

Additionally, we are providing you with a checklist of concepts for each technology. This checklist will serve as a valuable tool to ensure that you cover the most important topics within each technology. It is highly recommended that you use this checklist as a guideline to gauge your progress and track your learning.

For some topics, we have prepared exercises to work through, and/or an exam to test your knowledge. Any exams are open-book and untimed - once again, the point is to help double-check your understanding and gauge your progress. However, when noted, please submit your answers via email.

It is very important that you are completely comfortable with these basics, as they are mostly taken for granted in the training.

If you have any questions or need further clarification, please don't hesitate to reach out to us. We are more than happy to assist you.

CHAPTER

TWO

GIT

Git is the dominant VCS in the tech community, and is used by all development teams at IMC.

Additionally, most teams use the Gerrit code review tool, which heavily relies on advanced Git features around rewriting history.

2.1 Recommended Reading Material

2.1.1 Pro Git

The following chapters of Pro Git contain the most useful material about using Git.

- 1. Getting Started
- 2. Git Basics
- 3. Git Branching
- 7. Git Tools
  - 7.1 Revision Selection
  - 7.2 Interactive Staging
  - 7.3 Stashing and Cleaning
  - 7.5 Searching
  - 7.6 Rewriting history
  - 7.7 Reset Demystified
  - 7.10 Debugging with Git

Chapters 4-6 cover Git Servers and Workflows that are applicable to working in open source and on Github, but not immediately relevant to work at IMC.

The other sections of Chapter 7 cover more advanced Git tools that are useful, but will not be used in Dev School.

2.2 Recommended Exercises

Learn Git Branching is a good practical exercise for learning about the most useful Git commands.

2.3 Checklist

2.3.1 Essential Concepts

You should have a solid understanding of the following concepts.

- Git Fundamental Elements
  - Commits
  - Branches
  - Tags
  - References
  - Remotes
  - Repositories
- Commit Workflow
  - Working Directory
  - Stage/Index
  - Stash
- Checking out history
- Rewriting history
- `git reset`

2.3.2 Practical Skills

You must be able to do the following:

- Git commit workflow
  - Stage change and create commits
  - See your uncommitted or staged changes
  - Throw away your un-committed changes
  - Stash and unstash changes, stash multiple changes at a time
- Exploring Git History
  - See the history of commits in a repository
  - See the commit message for any given commit?
  - See what changes went into any given commit, or what are the differences between two arbitrary commits
  - Look at the code from a previous state in the repository history
  - Figure out what commit a bug was introduced in

- Branching and tagging
  - Create, switch, merge and delete branches
  - Copy commits from one branch to another
  - Create and delete tags
- Git Remotes
  - Clone from, pull from, push to a remote
- Undoing
  - Undo a `git commit` (without losing your code).
  - Revert a commit so that the code is in the same state as if the commit wasn't done, but the commit is left in history.
  - Delete a commit so that it is gone from git history entirely.
- Rewriting history
  - Modify the commit message of the most recent commit.
  - Modify the commit message of a previous commit
  - Combine two commits into one commit
  - Split one commit into two commits
  - Reorder two commits
  - Delete a commit
  - Change a branch to point to a completely different commit
- Fixing mistakes
  - Undo a `reset`
  - Recover a "lost" commit

2.3.3 Advanced Git Concepts and Tools

You are **not** required to know anything about these.

- Submodules / Subtrees
- Sparse Checkouts
- Hooks
- Git internals
  - Objects: Blobs, Trees, etc.
  - Packfiles
- `git rerere`

2.4 Additional Material

- An Introduction To Git and Github - Introduction to Git, but in video format.
- Think Like (a) Git
- Confusing git terminology

2.5 Exam

The *Git Exam* can be used to evaluate your understanding of git.

2.5.1 Git Exam

Theory

Instructions

Answer the below questions with short (a few sentences) answers. Keep your answers as precise and concise as possible.

Send your answers as an email, with one blank line in between answers to each question, to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work Git theory exam submission".

---

1. What do `git add` and `git commit` do?
2. What is a commit in Git? What are the components of a commit?
3. What is a branch in Git? What is the purpose of branches?
4. What does it mean for a commit to have more than one parent?
5. How do you merge changes from branch A into branch B? What does this mean? What happens to each branch?
6. How do you cherry-pick a change onto branch A? What does this mean? What happens when you do this?
7. How do you rebase changes from branch `A` on top of branch `B`? What does this mean? What happens to each branch?
8. What is a 'merge conflict' in Git? What operations can result in a merge conflict? How are merge conflicts resolved?
9. What is a fast-forward merge?
10. What is a 'remote' in Git? What is 'origin'?
11. What is the difference between `git pull` and `git fetch`?
12. What is the difference between a tag and a branch?
13. What are the most common use cases for tags?
14. What is a "commit-ish"? What are a few examples of a "commit-ish"?
15. What is the difference between `HEAD` and a "head" (such as `refs/heads/master`)?
16. What is "Detached HEAD mode"? How do you get into detached HEAD mode? How do you get out of detached HEAD mode?

17. If you find a bug in code that you know was working at some point in the past, but isn't working now, how would you find the commit where a bug was introduced?
18. How do you check out the code for an arbitrary past commit so that you can debug it?
19. What does rewriting history mean? What is a typical reason you will rewrite history?
20. What is the difference between reverting a commit and deleting it?
21. What is the difference between `git checkout 29f5e54` and `git reset 29f5e54`?
22. How do you delete the most recent commit from a branch?
23. How do you delete an older commit from a branch's history?
24. When should you not rewrite history? What can go wrong when you modify history?
25. If you `git commit --amend` a commit, how can you recover the previous version of that commit?
26. What is the `reflog`? When would you use it?

Practical

Instrucitons

Clone the repository from https://github.com/imc-trading/devschool-git-exam and then complete the following steps.

When you are done, zip up your repository and send it, along with the answers to each question, with a blank line in between each answer, to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work Git practical exam submission".

1. Check out the branch `resume`. Change the commit message from "Add Wok Experience" to "Add Work Experience". What commands did you run?
2. Add this "Work Experience" item in "Resume.md":

```
### The Normal Brand

- Web Developer -- 2019-2021
  - Implemented OAuth login process to support Social Login.
  - Added web analytics
```

Commit it on the branch `resume` with the message "Add Normal Brand item". What command(s) did you run?

3. The commit on the branch `resume` with the message "fill edu" has some issues. Fix it like this: 1. Change the message to "Add content for Education section". 2. The commit has introduced some whitespaces at the end of two lines. Remove those. What command(s) did you run? Explain your actions.
4. The commit with the message "Add empty lines" is not useful. Remove it from the history. What command(s) did you run?
5. There is a branch called `skills`. How can you show the content of the `Resume.md` file on that branch without switching to the branch?
6. On the branch `skills` there is a commit called "Add skills". Add that commit on top of the branch `resume` without switching branches away from `resume`. That should create a conflict. Fix it by incorporating both the previous and the new changes. What commands and actions did you do?

7. In `Resume.md` there is a line `### Illinois State University`. How can you find out what is the last commit that changed/introduced that line? What is the message of that commit? What commands did you run?
8. Create a branch called `letter_of_intent` forking off from the branch `job_applications`. Create a file there called `letter.txt` with the content `I need this job, please!`. Commit this file on the `letter_of_intent` branch with the commit message `Add letter of intent`. Add another commit that adds this line to `letter.txt`: `Sincerely yours, Norman`. Commit it with the message `Sign letter`. What commands did you run?
9. Rebase the `letter_of_intent` onto the `resume` branch without including the commits on the `job_applications` branch. Basically add the commits `Add letter of intent` and `Sign letter` on top of `resume` without changing `resume` and point `letter_of_intent` to the last commit. Do not use `cherry-pick`. What command did you run?
10. Merge the branch `letter_of_intent` into the branch `resume`. What commands did you run? Explain the merge strategy taken.
11. Merge the `job_applications` branch into the branch `resume`. What commands did you run? What merge strategy was taken? What happened to the git history of the `resume` branch?

CHAPTER

THREE

UNIX BASICS

All of our deployment servers are UNIX environments and you will need to be able to interact with them using the command line in order to effectively deploy and manage your apps at IMC.

3.1 Checklist

3.1.1 Essential skills

You must be able to do the following:

- Be familiar with the UNIX command line interface and its basic functionalities.
  - Be able to navigate through a file system, create and remove files and directories. Use the `cd`, `ls`, `mkdir`, `cp`, `mv` and `rm` commands.
- View files using `cat` and `less`
- Edit files using a CLI text editor.
  - We recommend `vim` but `emacs` and `nano` are also available.
    - `vimtutor` is a useful tool for learning.
  - Be able to open, edit and write files from the CLI.

3.1.2 Useful skills

The following skills are extremely useful, but not critical for Dev School.

- Search through files and directories for text patterns with `find` and `grep`
- Redirect output to a new command with the `|` operator.
  - Combine this with `grep`!
- Redirect output to a file with > and >>
- Understand environment variables, `env`, `export`, `echo`
- Monitor a system with `ps`, `top` and `htop`
- Manage processes (`kill`)

3.1.3 Bonus

- Use a CLI editor to create a simple bash script file which prints "Hello World" (`echo`).
- Make it runnable (`chmod +x`) and run it directly from the command line.
- If your script file is called `hello.sh` what is the difference between running `bash hello.sh` and `./hello.sh`? What is a shebang?

CHAPTER

FOUR

DOCKER

Containers streamlines the process of deploying production services by enabling developers to package an application with all its dependencies into a standardized unit for software development. This approach, known as containerization, ensures that the application will run uniformly, regardless of any customized settings that machines might have that could differ from the machine used for writing and testing the code.

Docker is IMC's containerization tool of choice. Docker's lightweight, scalable, and consistent environment is particularly beneficial for creating, deploying, and running applications in a variety of environments, from local development machines to production servers, thereby increasing productivity and reducing overhead.

4.1 Recommended Tutorial

- Docker - Getting Started
  - Parts 1-9

4.2 Checklist

4.2.1 Essential Concepts

You should have a solid understanding of the following concepts. We've included some follow-up questions and related concepts under each point.

- Containers
  - What is a container?
  - Difference between virtual machines and containers
  - Advantages of containerization
  - What is Docker?
- Docker Images
  - What is a Docker image?
  - Dockerfile structure and commands
    - FROM, RUN, CMD, ENTRYPOINT, ENV, COPY, ADD, etc
  - Docker image layers
  - Image tags

- Docker Containers

4.2.2 Useful Concepts

You should have a general familiarity with the following concepts, but deep understanding is not necessary.

- Docker Volumes and Bind Mounts
  - Persistent data and Docker storage options
  - Docker volumes
  - Bind mounts
- Docker Networking
  - Host networking
  - Port binding
- Docker Best Practices
  - Dockerfile best practices
  - Docker development best practices

4.2.3 Practical Skills

You must be able to do the following:

- Build images using a Dockerfile
- Manage and delete images
- Create, start, stop, and remove containers
- Interact with running containers
- Read container logs
- Execute commands in a running container
- Inspect and monitor Docker containers
- Manage and delete containers
- Mount a local directory into a docker container
- Bind ports

4.3 Exam

The *Docker Exam* can be used to evaluate your understanding of Docker.

This exam is not graded - you should it to identify weaknesses in your understanding.

4.3.1 Docker Exam

Instructions

Send your answers as an email, with one blank line in between answers to each question, to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work Docker exam submission".

1. What command(s) do you use to run MariaDB in Docker on your machine and map its port to the port `12345` on your machine? It should contain a database called `prework`.
2. What command(s) do you use on your local machine to open a prompt into the MariaDB server from step 1? You can use any Mysql client.

Clone

Clone this repository from `https://github.com/imc-trading/devschool-docker-exam`

3. How do you run step 1 but use `init.sql` to initialize the database?
4. What command do you use to open bash console in your running MariaDB container?
5. What commands do you use to list containers, list images, stop a container, start a container, remove a container?
6. In `server.py` you have a small Python server implementation. Please create a custom Docker image around it and a Docker compose that starts up this server and the MariaDB container from the steps above. The server should connect to the MariaDB container and it should expose its HTTP port to localhost's port 8082. How do you check that the server works? Note that the server has requirements that must be installed with `pip install -r requirements.txt`

CHAPTER

FIVE

KUBERNETES

Kubernetes, aka K8s, is one of the most popular container orchestration platforms. It was open-sourced by Google, inspired from their internal orchestration system. It is widely used at IMC to deploy software.

5.1 Checklist

5.1.1 Essential Concepts

You should have a solid understanding of the following concepts. We've included some follow-up questions and related concepts under each point.

- Kubernetes Basics
  - What is Kubernetes?
  - What are the benefits to using Kubernetes?
- Kubernetes Clusters
  - Nodes
- Kubernetes Components
  - Pods
  - Deployments
  - Services

5.1.2 Useful Concepts

You should have a general familiarity with the following concepts, but deep understanding is not necessary.

- Storage
  - Volumes
  - Persistent Volumes
- Configuration
  - ConfigMaps

5.1.3 Practical Skills

You must be able to do the following:

- Kubernetes Basics:
  - Deploy a containerized application on a cluster.
  - Update the containerized application with a new software version.
  - Debug the containerized application.
- Basic Kubectl Commands
  - Get info about nodes, pods, deployments, services
  - Create and delete resources
  - Get logs
  - Open an interactive shell on a running pod for debugging

5.2 Tutorial

This *Kubernetes tutorial* covers the skills above.

5.2.1 Tutorial

Prerequisites

- Basic command line knowledge: cd, mkdir
- Higher level understanding of Container technology: Docker. Do the Docker pre-work if not ready.

Introduction

What is Kubernetes?

Kubernetes, aka K8s, is one of the most popular orchestration platforms. It was open-sourced by Google, inspired from their internal orchestration system.

K8s is a container orchestrator. It decides on where and how containerized applications are launched on a server, when to scale them and down the number of application replicas, and what to do when an application stops working.

K8s can run on any kind of server: in an on-premise datacenter, as a cloud service, or a mix of the two. You can install it on your own or use a managed service on a public cloud.

What are containers?

A container bundles an application's binaries and configuration into one unit. You can run multiple containers on a server. An image is a file containing executable code that can be run as a container. A container registry is a database that stores container images. One famous registry is Docker Hub, but companies often have an internal registry as well.

Setting up Kubernetes

Install Docker

Make sure you have a terminal and a text editor ready for use. Make sure you have the Docker engine installed on your machine. If not follow these instructions. Make sure Docker is running:

```
$ docker
```

This should show the list of docker commands.

Install Minikube

We need a K8s cluster. Normally a K8s cluster is installed on at least three machines running in a datacenter. However, for learning purposes we can use a small cluster running entirely on our machine: specifically Minikube. Install minikube as described in the docs.

Let's start the cluster:

```
$ minikube start
```

Get the minikube version to see if it matches the latest stable version:

```
$ minikube update-check
CurrentVersion: v1.32.0
LatestVersion: v1.32.0
```

You can stop the cluster by running:

```
$ minikube stop
```

And you can delete the cluster like this:

```
$ minikube delete
```

Spin up and explore a minikube cluster

If you've deleted the cluster, we'll create a new one:

```
$ minikube start
```

`minikube` is just one way to create a cluster. If you were to use a cloud provider, you'd use whatever alternative tool they provide to create a cluster. No matter how you create a cluster, you will use the K8s CLI tool to connect to the cluster: `kubectl`. You can point it to any cluster. `minikube start` has automatically configured your `kubectl` to connect to the minikube cluster.

Inspect the cluster:

```
$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:32774
CoreDNS is running at https://127.0.0.1:32774/api/v1/namespaces/kube-system/services/
↪kube-dns:dns/proxy
```

That shows the address and port of the cluster control plane. We'll explore this later. If you see an error saying that the connection to the server was refused, it means you don't have a minikube cluster running. Either wait for it to start up or run `minikube start`.

Get info about the cluster nodes with:

```
$ kubectl get nodes
NAME       STATUS   ROLES           AGE     VERSION
minikube   Ready    control-plane   7m10s   v1.28.3
```

You should see just one node, with the role of `control-plane` and the K8s version it's running. Let's look at all the namespaces that get created by default:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   34m
kube-node-lease   Active   34m
kube-public       Active   34m
kube-system       Active   34m
```

Namespaces are a way to isolate and manage applications and services. Now let's look at the pods that run by default on the cluster:

```
$ kubectl get pods -A
NAMESPACE     NAME                               READY   STATUS    RESTARTS      AGE
kube-system   coredns-5dd5756b68-pxfzv           1/1     Running   2 (30m ago)   35m
kube-system   etcd-minikube                      1/1     Running   2 (30m ago)   35m
kube-system   kube-apiserver-minikube            1/1     Running   2 (29m ago)   35m
kube-system   kube-controller-manager-minikube   1/1     Running   2 (30m ago)   35m
kube-system   kube-proxy-2wwf7                   1/1     Running   2 (30m ago)   35m
kube-system   kube-scheduler-minikube            1/1     Running   2 (30m ago)   35m
kube-system   storage-provisioner                1/1     Running   4 (28m ago)   35m
```

The `-A` flag means that we want to see the pods in every namespace. Pods are how containers are run in K8s. The pods above are all in the `kube-system` namespace, meaning that these specific pods are required to run a K8s cluster itself.

Finally, let's see the services that run in this cluster:

```
$ kubectl get services -A
NAMESPACE     NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  ␣
↪   AGE
default       kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP                  ␣
↪   39m
kube-system   kube-dns     ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP␣
↪   39m
```

Services act as load balancers within the cluster and direct network traffic to pods. Don't worry if these look cryptic. We'll cover all these concepts later.

You can pat yourself on the back. You've just spun up a cluster and used `kubectl` to inspect how the cluster is set up by default.

Application deployment

Reading and writing YAML

In the K8s world you will see two big movements being mentioned often: *Infrastructure as Code* and *GitOps*. This means the state of the system can be expressed as code and the changes of that system should be tracked using a code version management system like Git.

In K8s, the code is commonly described in a YAML format, which is a data serialization format, like JSON or XML.

Here is an example of a generic YAML file (not K8s specific):

```
---
 Description
name: Bob Doyle
professions:
  - software engineer
  - coder
  - software developer
  - programmer
  - hacker
previousJobs:
  IMC: 1 year
  IMC trading: 2 years
  titles:
    - team lead
    - software engineer
```

- `---` means that it's the beginning of a document. So you can have multiple documents in one file.
- `# Description` is a comment
- `name:  Bob Doyle` key value pair
- `professions` is list or array of items and each item is preceded by one `-`
- `previousJobs` - is a nested map of key value pairs. On one line you have a `key:` and on the next line you indent and add the next `key:  value` pair.

YAML files end with either `.yaml` or `.yml`. It's easy to make mistakes when writing YAML. Luckily you can validate your YAML with an online service like: https://yamlchecker.com.

Namespaces

Namespaces let you organize and isolate workloads. E.g., if both your prod and dev are running in the same cluster, you can separate those two by creating two namespaces. Let's look at the default ones:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   148m
kube-node-lease   Active   148m
kube-public       Active   148m
kube-system       Active   148m
```

Let's look at a Kubernetes manifest for a namespace:

```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: development
```

The only thing you need to worry about is the `name` field, in this case: `development`. Save this code in a `namespace.yaml` file and let's create this namespace in our K8s cluster:

```
$ kubectl apply -f namespace.yaml
namespace/development created
```

See the newly created namespace by running:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   5h5m
development       Active   46s
kube-node-lease   Active   5h5m
kube-public       Active   5h5m
kube-system       Active   5h5m
```

Since in YAML, we can have multiple documents in a file, we can define multiple K8s resources in a file. Add one more namespace called `production`:

```yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: development
---
apiVersion: v1
kind: Namespace
metadata:
  name: production
```

And apply it again:

```
$ kubectl apply -f namespace.yaml
namespace/development unchanged
namespace/production created
```

As you can see K8s is smart enough to know that it should not create the `development` namespace again, only `production`. Inspect the namespaces again:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   5h8m
development       Active   4m35s
kube-node-lease   Active   5h8m
kube-public       Active   5h8m
kube-system       Active   5h8m
production        Active   14s
```

To delete these namespaces, run:

```
$ kubectl delete -f namespace.yaml
namespace "development" deleted
namespace "production" deleted
```

Deploy an application

First, let's create a simple application. Create a file called `app.py` and paste this code in:

```
from flask import Flask
import os

app = Flask(__name__)


@app.route('/')
def print_env_variables():
    pod_name = os.getenv('POD_NAME', 'Default Pod Name')
    pod_namespace = os.getenv('POD_NAMESPACE', 'Default Pod Namespace')
    pod_id = os.getenv('POD_ID', 'Default Pod ID')

    return f'POD_NAME: {pod_name}, POD_NAMESPACE: {pod_namespace}, POD_ID: {pod_id}'


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=3000)
```

This is a simple Python HTTP server, listening on port 3000, that returns the values of three environment variables: `POD_NAME`, `POD_NAMESPACE`, `POD_ID`.

Now let's create a Docker image that packages this application. Create a `Dockerfile` file with this content:

```
FROM python:3.8-slim
WORKDIR /app
COPY . /app
RUN pip install --trusted-host pypi.python.org Flask
EXPOSE 3000
ENV POD_NAME=default_pod \
    POD_NAMESPACE=default_namespace \
    POD_ID=default_id
CMD ["python", "app.py"]
```

This copies our `app.py` into the image, installs the appropriate Python packages, exposes the port 3000, specifies the expected environment variables, and specifies how to start tha app.

We'll use this Docker image in our minikube K8s cluster. Minikube has its own Docker registry, so before building the image, we need to make sure it gets pushed to the Minikube Docker registry:

```
$ eval $(minikube docker-env)
```

In the same terminal build the Docker image with the name `pod-info-app`:

```
$ docker build -t pod-info-app .
```

Deploying the app on K8s

K8s is designed to make your applications highly available. It does that by creating multiple replicas of your application, so if one fails, the others can still serve your clients.

`Pods` are K8s resources that run your applications and microservices. One way to make sure that your application is highly available is to organize your pods into K8s `deployments`. This is a spec for a K8s deployment (see `kind: Deployment`):

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-info-deployment
  namespace: development
  labels:
    app: pod-info
spec:
  replicas: 3
  selector:
    matchLabels:
      app: pod-info
  template:
    metadata:
      labels:
        app: pod-info
    spec:
      containers:
        - name: pod-info-container
          image: pod-info-app
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 3000
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: POD_ID
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
```

The deployment is called `pod-info-deployment` specifying that the pods should be in the `development` namespace and labeling all the pods in this group with the key value: `app: pod-info`.

Under `spec` we specify that we want 3 replicas of our container to run at the same time. Under `containers` we specify the Docker container we want to run in the pod. Notice that `containers` is a list, so we can have *multiple* containers in a pod.

We define a container called `pod-info-container` which will use the latest version of Docker image we've just

created: `pod-info-app`. Moreover, we direct the traffic for the container to port `3000`. `imagePullPolicy: IfNotPresent` is needed to make Minikube use its own Docker registry instead of going to Docker Hub to fetch the Docker image.

We pass the three environment variables to the Docker container: `POD_NAME`, `POD_NAMESPACE`, and `POD_IP`. These will take their values from the K8s pod metadata: `name`, `namespace`, `status.podIP` respectively.

Save the YAML above in a file called `deployment.yaml`. Make sure you create the `development` namespace as we did above and deploy this K8s deployment:

```
$ kubectl apply -f namespace.yaml
$ kubectl apply -f deployment.yaml
deployment.apps/pod-info-deployment created
```

Now see the deployment in the `development` namespace:

```
$ kubectl get deployments -n development
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
pod-info-deployment   3/3     3            3           5m4s
```

`3/3` means there are three pods ready to be used. Now let's see the pods created:

```
$ kubectl get pods -n development
NAME                                  READY   STATUS    RESTARTS   AGE
pod-info-deployment-84b8474657-hbpbg   1/1     Running   0          6m58s
pod-info-deployment-84b8474657-jgj4z   1/1     Running   0          6m58s
pod-info-deployment-84b8474657-rt9wn   1/1     Running   0          6m58s
```

If you want to see how the K8s deployment ensures three pods running at any time, try deleting one of those pods:

```
$ kubectl -n development delete pod pod-info-deployment-84b8474657-hbpbg && kubectl get␣
↪pods -n development
pod "pod-info-deployment-84b8474657-hbpbg" deleted
NAME                                  READY   STATUS    RESTARTS   AGE
pod-info-deployment-84b8474657-jgj4z   1/1     Running   0          9m25s
pod-info-deployment-84b8474657-mpmjw   1/1     Running   0          31s
pod-info-deployment-84b8474657-rt9wn   1/1     Running   0          9m25s
```

As you can see, it automatically created a new pod instead (the second one).

Great! Now you understand K8s pods and deployments. Yay!

Checking the health of a pod

When a pod gets created, it happens often that it fails creating due to a multitude of reasons:

- The Docker image cannot be found (e.g., due to mis-configuration)
- You could be out of space on your worker node, so the pod cannot be scheduled.
- Or it starts running, but suddenly stops due to some error in your app.

For this you can look into the event log of the pod:

```
$ kubectl -n development describe pod pod-info-deployment-84b8474657-jgj4z
...
Events:
```

(continues on next page)

(continued from previous page)

```
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  17m   default-scheduler  Successfully assigned development/pod-info-
↪deployment-84b8474657-jgj4z to minikube
  Normal  Pulled     17m   kubelet            Container image "pod-info-app" already␣
↪present on machine
  Normal  Created    17m   kubelet            Created container pod-info-container
  Normal  Started    17m   kubelet            Started container pod-info-container
```

There is a lot of useful information here and the events are at the bottom. If the pod has been running for a long time, you'll not see any events, meaning that K8s lets the pod do its own thing because it's healthy.

This is a healthy pod, but let's try with a failing pod. Change your `deployment.yaml` to have a typo in the Docker image, delete the deployment, re-apply it, get the pods, notice the failing pods, and check the events of one of those:

```
$ kubectl delete -f deployment.yaml
$ kubectl apply -f deployment.yaml
$ kubectl -n development get pods
NAME                                  READY   STATUS             RESTARTS   AGE
pod-info-deployment-5c7f779b8b-b5wfc   0/1     ImagePullBackOff   0          51s
pod-info-deployment-5c7f779b8b-csc8j   0/1     ImagePullBackOff   0          51s
pod-info-deployment-5c7f779b8b-gqpgc   0/1     ImagePullBackOff   0          51s
$ kubectl -n development describe pod pod-info-deployment-5c7f779b8b-b5wfc
...
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  86s                default-scheduler  Successfully assigned␣
↪development/pod-info-deployment-5c7f779b8b-b5wfc to minikube
  Normal   Pulling    45s (x2 over 85s)  kubelet            Pulling image "pod-info-app-
↪typo"
  Warning  Failed     20s (x2 over 56s)  kubelet            Failed to pull image "pod-
↪info-app-typo": Error response from daemon: pull access denied for pod-info-app-typo,␣
↪repository does not exist or may require 'docker login': denied: requested access to␣
↪the resource is denied
...
```

This helps you quickly identify where the misconfiguration is. Most issues with pods occur at the beginning of the lifecycle, and now you know exactly how to investigate them.

Now, fix the image in `deployment.yaml` and re-apply.

Check that your application is working

Our pod is now exposing an HTTP server on the port 3000. This port is not yet exposed to the outside world. It's only exposed internally in the cluster, so other pods can access it. In order to try that we'll deploy another pod running a container called `busybox`. Busybox is a minimal Linux docker image that has handy tools like: `cat`, `wget`, etc. It's great for troubleshooting. Let's define another deployment in a file called `busybox.yaml`:

```
---
apiVersion: apps/v1
kind: Deployment
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: busybox
  namespace: default
  labels:
    app: busybox
spec:
  replicas: 1
  selector:
    matchLabels:
      app: busybox
  template:
    metadata:
      labels:
        app: busybox
    spec:
      containers:
        - name: busybox-container
          image: busybox:latest
          # Keep the container running
          command: [ "/bin/sh", "-c", "--" ]
          args: [ "while true; do sleep 30; done;" ]
```

Unlike for our other deployment, we're going to deploy this in the `default` namespace and run only one replica. Let's create it and show the pods:

```
$ kubectl apply -f busybox.yaml
$ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
busybox-7f6c976c4c-p7vhp   1/1     Running   0          39s
```

The last command didn't specify the namespace, meaning that it used the `default` namespace.

Now, we're going to jump into the `busybox` pod and run an HTTP GET request to one of our app pods. For that we first need the IP address of one of our app pods:

```
$ kubectl get pods -n development -o wide
NAME                                    READY   STATUS    RESTARTS   AGE   IP           ␣
↪NODE       NOMINATED NODE   READINESS GATES
pod-info-deployment-84b8474657-7wqbv    1/1     Running   0          18m   10.244.0.31  ␣
↪minikube   <none>           <none>
pod-info-deployment-84b8474657-fm8br    1/1     Running   0          18m   10.244.0.33  ␣
↪minikube   <none>           <none>
pod-info-deployment-84b8474657-xxwbx    1/1     Running   0          18m   10.244.0.32  ␣
↪minikube   <none>           <none>
```

This command shows us extra information, including the IP addresses of our pods. Let's take the first IP address: `10.244.0.31`.

Now let's jump into the `busybox` pod:

```
$ kubectl exec -it busybox-7f6c976c4c-p7vhp -- /bin/sh
/ #
```

This means we want to run the command `/bin/sh` inside the pod, and we use `-it` to get an interactive terminal. Now

let's use `wget` to make the HTTP request to our app pod:

```
/ # wget 10.244.0.31:3000
Connecting to 10.244.0.31:3000 (10.244.0.31:3000)
saving to 'index.html'
index.html           100%␣
↪|********************************************************************************************************************************
↪    95  0:00:00 ETA
'index.html' saved
/ # cat index.html
POD_NAME: pod-info-deployment-84b8474657-7wqbv, POD_NAMESPACE: development, POD_ID: 10.
↪244.0.31
/ # exit
```

This got us an `index.html` file, which we displayed. As you can see we get our environment variables printed. These match the pod information we got when running `kubectl get pods -n development -o wide`.

Yay! Our app is working!

View your application logs

You can also inspect the health of your application by looking at its logs:

```
$ kubectl -n development logs pod-info-deployment-84b8474657-7wqbv
 * Serving Flask app 'app'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a␣
↪production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:3000
 * Running on http://10.244.0.31:3000
Press CTRL+C to quit
127.0.0.1 - - [02/Feb/2024 10:05:31] "GET / HTTP/1.1" 200 -
```

This command outputs the `stdout` of the application running inside the container. If your app is also writing to some file in the container, you can also use `kubectl exec -it <pod name> -- /bin/sh` to jump to your app pod and explore the filesystem.

Complex Application Deployment

Expose your application to the internet with a LoadBalancer

We have the pods accepting traffic from inside the K8s cluster, but how do we make the pods accessible from the internet? The answer is: K8s services.

A K8s service is a load balancer that directs traffic from outside the cluster to the pods. A load balancer has a public and static IP address. The public part means that anyone can access it from the internet. And the static part is important because pods and their IP addresses change frequently, but your service IP needs to remain the same. Let's define a service in a `service.yaml` file:

```
---
apiVersion: v1
kind: Service
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: demo-service
  namespace: development
spec:
  selector:
    app: pod-info
  ports:
    - port: 80
      targetPort: 3000
  type: LoadBalancer
```

Notice `kind: Service`. Its name is `demo-service` and it will be deployed in the `development` namespace. The `selector` is the important part. It tells it that any traffic should be redirected to pods having the label: `app: pod-info`. Looking back at our `deployment.yaml`, we see that all the pods in our deployment had this label:

```
...
kind: Deployment
metadata:
  name: pod-info-deployment
  ...
  labels:
    app: pod-info
...
```

Back to our service, we see that it accepts traffic on the port `80` and it redirects traffic to the port `3000` in the pods. Finally, we are specifying the service type to `LoadBalancer`. This is one of the three types supported. The other ones are `NodePort` and `ClusterIP`.

Note: In IMC we only use `ClusterIP`. `LoadBalancer` and `NodePort` types are usually used in Cloud providers. There is no concept of `EXTERNAL-IP` to internal Kubernetes clusters at IMC. For the sake of this tutorial though, we'll use `LoadBalancer`.

Let's create the service. Since we're using Minikube, in a separate tab run this command and let it run:

```
$ minikube tunnel
```

This makes the Minikube cluster available on the host machine. In the original tab, run:

```
$ kubectl apply -f service.yaml
service/demo-service created
```

Let's find the IP of the service:

```
$ kubectl get service -n development
NAME           TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
demo-service   LoadBalancer   10.99.155.149   127.0.0.1     80:31717/TCP   2m6s
```

Our service has an ip for inside the cluster and one external IP. Notice that the external IP is actually the IP of our localhost. Because Minikube only runs on our machine, not on an actual cloud (e.g., AWS), we're not actually getting an external IP available on the Internet.

Let's try doing an HTTP call to the IP address. You can open it in your browser or from the command line run:

```
$ curl http://127.0.0.1
curl: (7) Failed to connect to 127.0.0.1 port 80 after 3 ms: Couldn't connect to server
```

Oops! It does not work. That is because we are using the port 80, which needs admin rights to open. It would work with a port larger than 1024, e.g., 8080. But don't despair. Go to the `minikube tunnel` tab. That is asking for your admin password. Type it in and try again:

```
$ curl http://127.0.0.1/
POD_NAME: pod-info-deployment-84b8474657-fm8br, POD_NAMESPACE: development, POD_ID: 10.
↪244.0.33
```

Look at that! It works! To see the load balancer in action, run the command above repeatedly. You should get different responses, depending on the pod to which the service sends your traffic to.

Yay! You're a service expert. Congrats!

Add resource requests and limits to your pod

Well configured containers let K8s know how much memory and CPU to reserve on a worker node. Resources refer to the amount of available CPU and memory on the worker node running a pod. If you deploy a pod without specifying the set of resource requests, it can be scheduled on a node that does not have enough processing power or memory and cause the node to fail. Similarly, if you don't specify resource limits, the pod can start using more and more resources till the node runs out of resources and fails. That will affect other pods running on that node, too. See Resource Management for Pods and Containers for more details.

Let's go to our `deployment.yaml` and add resource specification to the pod `spec`:

```
...
spec:
  containers:
    - name: pod-info-container
      image: pod-info-app
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "500m"
...
```

`requests` means: do not schedule the pod unless the node has at least 64Mi of memory and 250m of CPU. `limits` means: stop running this container if it exceeds 128Mi om memory and 500m of CPU (500m means 500 milliCPU, which means 0.5 CPU allocated). But how do you decide what values to put in here? There's no standard answer for this. It depends on your application. The `pod-info` app does not do much, so these small values are fine. Let's redeploy our deployment, inspect the new pods, and describe a pod to see the new limits:

```
$ kubectl -n development get pods
  NAME                                  READY   STATUS             RESTARTS         AGE
  pod-info-deployment-556c97d5fd-46f2d   1/1     Running            0                111s
...
  $ kubectl -n development describe pod pod-info-deployment-556c97d5fd-46f2d
...
Limits:
  cpu: 500m
  memory: 128Mi
Requests:
```

(continues on next page)

(continued from previous page)

```
  cpu: 250m
  memory: 64Mi
...
```

Cleaning up

Now that we've reached the end of this tutorial, feel free to delete all the K8s resources and delete the minikube cluster if you wish. You can delete by using the yaml files in the command `kubectl delete -f <yaml_file>`. Delete `busybox.yaml`, `deployment.yaml`, `service.yaml`, `namespace.yaml`. To delete the minikube cluster: `minikube delete`.

Advanced Topics

Ways to manage K8s pods

Deployments

The most common way to deploy applications on K8s is by using deployments, which allows you to control the number of replicas running. When you are deploying a new version of your application, K8s can keep the old version up and running, roll up the new version, ensure the new pods are running and healthy, and then remove the old pods. This is a no-downtime upgrade, which is one of the most desirable ways to upgrade.

DaemonSets

Another way to deploy apps is using a DaemonSet. It will put one copy of your container on every node running in the cluster, so you cannot directly control how many replicas are running. These are usually used to run processes in the background, for example to collect metrics about the node and the pods running in the node.

Jobs

A job will create one or more pods and run a container inside of them until it has successfully completed its task. An example of a job is a script that runs a backup of your database. After it finishes, the pod gets deleted. This is used for one-off tasks.

Running stateful workloads

Our application above was stateless, i.e., it was fine to restart it at any point and not have to worry about losing any data. What if you want to run an application that uses a database? One option is to connect to an external database service, e.g., one managed by a cloud provider.

Another option is to use a K8s persistent volume. This is a type of data storage that exists in your cluster and remains even after a pod gets deleted. We usually mount it in the filesystem of a pod. For example, we could deploy a SQL database as a pod, but mount its data directory on a persistent volume. So even if you restart the SQL pod, the data is still there. You can use a K8s object called a stateful set, that makes sure that your updated application can communicate with the same volume as the previous pod.

CHAPTER

SIX

HELM

Helm is the package management system for K8s, just like npm is for Node.js and apt-get is for Debian. Without Helm, deploying an application to K8s means copying and pasting lots of yaml files and running kubectl apply -f over and over again to create deployments, services, pods, etc.

Helm allows packaging all the K8s objects needed for an application and installing them with one command. You can also version charts, upgrade them, debug deployments, roll back as needed. Helm allows you to configure a chart, so you can deploy the same application in different environments with different sets of values.

6.1 Checklist

6.1.1 Essential Concepts

You should have a solid understanding of the following concepts. We've included some follow-up questions and related concepts under each point.

- What is Helm and why use it?
- Charts
  - Chart Structure
  - Chart Dependencies
- Repositories
- Releases

6.1.2 Practical Skills

You must be able to do the following:

- Read simple Helm Charts
- Create a simple Helm Chart from scratch
- Deploy a Helm Chart

6.2 Tutorial

This *Helm tutorial* covers the skills above.

6.2.1 Tutorial

Prerequisites

- K8s pre-work:
  - K8s foundations: deployments, services, configmaps, pods.
- Access to a K8s cluster, e.g., minikube
- Know basic Unix commands: mkdir, cd, mv
- A text editor

Introduction to Helm

What is Helm?

Helm is the package management system for K8s, just like npm is for Node.js and apt-get is for Debian. Without Helm, deploying an application to K8s means copying and pasting lots of yaml files and running `kubectl apply -f` over and over again to create deployments, services, pods, etc.

Helm allows packaging all the K8s objects needed for an application and installing them with one command. You can also version charts, upgrade them, debug deployments, roll back as needed. Helm allows you to configure a chart, so you can deploy the same application in different environments with different sets of values.

Many open-source applications, like Mysql or Datadog, have readily available Helm charts that you can customize and easily deploy to your K8s cluster.

Installing Helm

Use the Helm documentation to install Helm on your machine.

Install and Configure a Chart from Helm Hub

Explore the Helm Hub

Go to the Helm Hub and scroll around. Can you recognize any of the apps? For examples, search for the `nginx-ingress` chart. It deploys the nginx ingress controller in your cluster. Open the chart's webpage and look around. The chart has a list of prerequisites, e.g., requires a K8s version that supports the Ingress Controller and at least Helm 3.0. Below we find instructions about how to install and uninstall the chart, and a list of values we can use to configure the chart. We can see the chart versions, the maintainers, and the source code location for the Helm chart.

Install a third party Helm chart

A popular chart to install on K8s clusters is: `kube-state-metrics`. It collects data about the K8s objects in that cluster (nodes, pods, deployments, etc) and monitors resource usage. Search it up in the Helm Hub and pick the one from Bitnami. Bitnami is a company which maintains many high-quality Helm charts for widely used open-source applications.

In order to install this chart, we first need to add the Bitnami chart repo to our Helm, and fetch all the charts from Bitnami. Then we ensure that the repo was properly added.

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
$ helm repo update
$ helm repo list
NAME    URL
bitnami https://charts.bitnami.com/bitnami
```

Now we'll install the chart. Helm will install in the cluster to which your `kubectl` is pointing to. If you started Minikube, your `kubectl` is pointing to the Minikube cluster, thus Helm will install there as well.

First, let's create a new namespace:

```
$ kubectl create namespace metrics
```

Let's install the chart in our new namespace:

```
$ helm install kube-state-metrics bitnami/kube-state-metrics -n metrics
NAME: kube-state-metrics
...
To access kube-state-metrics from outside the cluster execute the following commands:

    echo "URL: http://127.0.0.1:9100/"
    kubectl port-forward --namespace metrics svc/kube-state-metrics 9100:8080
```

An installation of a Helm chart is called a release. The command above means: we installed the chart `kube-state-metrics` from the `bitnami` repo under the release name `kube-state-metrics`. Now do the port forwarding as suggested in the output above.

```
$ kubectl port-forward --namespace metrics svc/kube-state-metrics 9100:8080
```

And now open http://127.0.0.1:9100/ in the browser. That should show a Web page with links to a lot of metrics. These metrics can be sent to a monitoring service, like Prometheus, and get alerts when the cluster is not healthy.

Good stuff! You have deployed a Helm chart.

Inspect a chart in your K8s cluster

Let's list the installed chart:

```
$ helm ls -n metrics
NAME                    NAMESPACE       REVISION        UPDATED                                 ␣
↪       STATUS          CHART                           APP VERSION
kube-state-metrics      metrics         1               2024-02-02 15:41:37.203101 +0100␣
↪CET    deployed        kube-state-metrics-3.11.3       2.10.1
```

Since it's the first time we install this chart, it has the revision 1. If you reinstall it, the revision will be incremented. Let's drill deeper and see what K8s objects have been created by this chart. A simple way is to see all the objects in the namespace, since this is the only thing installed there:

```
$ kubectl get all -n metrics
NAME                                      READY   STATUS    RESTARTS   AGE
pod/kube-state-metrics-86d5bb5bb7-bvm92   1/1     Running   0          8m24s

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/kube-state-metrics     ClusterIP   10.100.80.130   <none>        8080/TCP   8m24s

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/kube-state-metrics      1/1     1            1           8m24s

NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/kube-state-metrics-86d5bb5bb7      1         1         1       8m24s
```

It has created a pod, a service, a deployment, and a replica set that determines the number of pods we're running. Let's look at the pod logs:

```
$ kubectl logs kube-state-metrics-86d5bb5bb7-bvm92 -n metrics
I0202 14:41:37.380005       1 wrapper.go:120] "Starting kube-state-metrics"
...
```

Without Helm, we would've had to create all those yaml files for all these K8s objects. Helm makes this process faster and more reliable.

Try Helm show commands

Normally when installing a Helm chart, you want to customize it. For that you need to inspect it to see what configuration there is. You can read the documentation on the Helm Hub, but if that is missing, you can show the internals of the chart. You can use `helm show`. It has four subcommands, which you can read more about in the `helm show help`: `all`, `chart`, `readme`, `values`. Try running:

```
$ helm show chart
```

That will show a description of the chart, the maintainers, where to find the code, etc. All this is also on the Helm Hub page of the chart. One thing you cannot see in Helm Hub is the values file:

```
$ helm show values bitnami/kube-state-metrics > values.yaml
```

This shows all the values you can change to configure this chart and their default values. Open it by opening a text editor that can highlight yaml files to make it more readable. You can see there things like: the default number of replicas is 1, the service listens on port 8080, there is a liveness probe defined, etc.

If you want to customize your chart, you can change a value in this file and update the chart.

Updating a Helm Chart

As an application releases a new version, likely a new Helm chart version will be released as well. Let's look at the current version of our chart again:

```
$ helm ls -n metrics
NAME                    NAMESPACE       REVISION        UPDATED                                 ␣
↪       STATUS          CHART                           APP VERSION
kube-state-metrics      metrics         1               2024-02-02 15:41:37.203101 +0100␣
↪CET    deployed        kube-state-metrics-3.11.3       2.10.1
```

There are two versions:

- chart version - this is the version of the Helm chart
- app version - the version of the underlying code that gets installed as a pod or set of pods

Sometimes we don't want the latest version of a chart, since it might be incompatible with the rest of our applications. Let's say we want to downgrade to the version `3.11.2`. We can do that with the helm upgrade command:

```
$ helm upgrade kube-state-metrics bitnami/kube-state-metrics --version 3.11.2 -n metrics
```

The command means: I want to upgrade the already installed `kube-state-metrics` chart to the chart `kube-state-metrics` chart from the `bitnami` repo with the version `3.11.2` in the namespace `metrics`. Let's look at the K8s objects:

```
$ kubectl get all -n metrics
NAME                                      READY   STATUS    RESTARTS   AGE
pod/kube-state-metrics-c9849cb4f-x8crw    1/1     Running   0          2m53s

NAME                           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
service/kube-state-metrics     ClusterIP   10.100.80.130   <none>        8080/TCP   65m

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/kube-state-metrics      1/1     1            1           65m

NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/kube-state-metrics-86d5bb5bb7      0         0         0       65m
replicaset.apps/kube-state-metrics-c9849cb4f       1         1         1       2m53s
```

The service and the deployment did not change, whereas a new pod and replica set were created.

Deploy an Application Using Helm

Create a new Helm chart from the command line

So far we learned how to install a third-party chart. But what if we have developed an application and want to deploy it to K8s. These are the steps:

- create a Docker image containing the app
- define a Helm chart template defining the desired K8s object to be deployed

Let's start by generating a template Helm chart from the command line. Let's make a new directory and go there:

```
$ mkdir helm-course && cd helm-course
```

How let's create a chart called `first-chart` and show what was created:

```
$ helm create first-chart
Creating first-chart
$  tree
.
└── first-chart
    ├── Chart.yaml
    ├── charts
    ├── templates
    │   ├── NOTES.txt
    │   ├── _helpers.tpl
    │   ├── deployment.yaml
    │   ├── hpa.yaml
    │   ├── ingress.yaml
    │   ├── service.yaml
    │   ├── serviceaccount.yaml
    │   └── tests
    │       └── test-connection.yaml
    └── values.yaml
```

These boilerplate files are the starting point for us to deploy our application.

Explore the Helm chart directories and files

The file describing the chart is `Chart.yaml`. It has some predefined key values, like API version, chart name and description, chart type `application` (as opposed to `library`), the chart version, and the underlying application version. This metadata can be passed into the Helm templating engine.

The other top level file is `values.yaml`. Here we see lots of key value pairs, that can be used to configure the container(s) running in pod(s) and other K8s objects. To a large extent, it's up to you what you put in here.

The `charts` directory is where you can store sub-charts that are dependencies of the top level chart. Sometimes these are third party charts. Other times they are `library` charts (as we saw in the `type` in `Chart.yaml`).

The most exciting one is the `templates` directory. That contains the `NOTES.txt` file, which will be displayed in the terminal when someone installs the chart. Here is the first time we see the Helm templating system, e.g.:

```
1. Get the application URL by running these commands:
{{- if .Values.ingress.enabled }}
{{- range $host := .Values.ingress.hosts }}
...
```

All the files in the `template` directory are sent through the Helm templating system, where the values from the `values.yaml` are passed in and these templates get converted to familiar K8s objects. The boilerplate Helm chart comes with deployment, ingress, service, and serviceaccount files. In `service.yaml` we can see a familiar K8s service object, but some values are dynamically created with Helm templates. For example:

```
$ cat templates/service.yaml
...
metadata:
  name: {{ include "first-chart.fullname" . }}
```

(continues on next page)

(continued from previous page)

```
...
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
...
```

The value of the name is determined by the value of `first-chart.fullname`. This value is created in the `_helpers.tpl` file. The comment there explains what the method `first-chart.fullname` does:

```
$ cat template/_helper.tpl
...
{{/*
↪
Create a default fully qualified app name.
↪
We truncate at 63 chars because some Kubernetes name fields are limited to this (by the␣
↪DNS naming spec).
If release name contains chart name it will be used as a full name.
↪
*/}}
↪
{{- define "first-chart.fullname" -}}
↪
{{- if .Values.fullnameOverride }}
↪
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" }}
↪
{{- else }}
↪
{{- $name := default .Chart.Name .Values.nameOverride }}
↪
{{- if contains $name .Release.Name }}
↪
{{- .Release.Name | trunc 63 | trimSuffix "-" }}
↪
{{- else }}
↪
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" }}
↪
{{- end }}
↪
{{- end }}
↪
{{- end }}
...
```

Back to `service.yaml`. `type` and `ports` are coming from the `values.yaml` files because they start with `.Values.`. Specifically, they take the values `service.type` (i.e., `ClusterIP`) and `service.port` (i.e., `80`) respectively.

Deploy and update a Kubernetes ConfigMap via Helm

A ConfigMap is a K8s object that stores non-sensitive data used by pods, like port numbers and environment variables. Let us deploy a ConfigMap with Helm. First, check if Helm generates a `templates/configmap.yaml` file.

It does not. It does not mean we cannot add one. The current files in the `templates` directory are nice and all to give us a taste of K8s objects that you can add in a Helm chart, but as a starting point we'll create a smaller chart that only has a ConfigMap. So let's remove all the files from that directory and create a ConfigMap file.

```
$ rm -r templates/*
$ touch templates/cm.yaml
```

Add this to `cm.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: first-chart-configmap
data:
  port: "8080"
```

This ConfigMap is called `first-chart-configmap` and it stores the key value pair of port 8080. Now let's install the current directory with Helm.

```
$ helm install first-chart .
NAME: first-chart
LAST DEPLOYED: Sat Feb  3 16:44:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

It has been installed under the release name `first-chart` in the namespace `default`. Let's confirm that by inspecting the deployed config maps.

```
$ kubectl get cm
NAME                     DATA   AGE
first-chart-configmap    1      76s
$  kubectl describe cm first-chart-configmap
Name:         first-chart-configmap
Namespace:    default
...
Data
====
port:
----
8080
...
```

Great! It's there and it contains the right data. Let's add another key value to our `cm.yaml`: `allowTesting:  "true"`.

```
...
data:
  port: "8080"
  allowTesting: "true"
```

Before we deploy the change to the K8s cluster, we can view the change locally by running:

```
$ helm template first-chart .
---
# Source: first-chart/templates/cm.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: first-chart-configmap
data:
  port: "8080"
  allowTesting: "true"
```

It looks good. So let's upgrade the chart on the K8s cluster:

```
$ helm upgrade first-chart .
Release "first-chart" has been upgraded. Happy Helming!
NAME: first-chart
LAST DEPLOYED: Sat Feb  3 16:51:58 2024
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
```

As you can see the revision has gone up to 2. Let's describe the ConfigMap again to see if our new value was added:

```
$ kubectl describe cm first-chart-configmap
...
Data
====
allowTesting:
----
true
port:
----
8080
...
```

Great! `allowTesting` is there! We've just learned how to deploy a config map with Helm.

Deploy and update a Kubernetes secret via Helm

K8s secrets store sensitive information like passwords and SSH keys. Let's create a `templates/secret.yaml` file with this content:

```
apiVersion: v1
kind: Secret
metadata:
  name: first-secret
type: Opaque
data:
  username:
  password:
```

The `username` we want to use is `admin` and the `password`: `4w572$9sns1&!`. Before we put these in the secret, we have to encode them with `base64`:

```
$ echo -n 'admin' | base64
YWRtaW4=
$ echo -n '4w572$9sns1&!' | base64
NHc1NzIkOXNuczEmIQ==
```

Let's paste those values in `secret.yaml`:

```
...
data:
  username: YWRtaW4=
  password: NHc1NzIkOXNuczEmIQ==
```

Let's process our chart through the Helm template processor and see the K8s objects we get:

```
$ helm template first-chart .
---
# Source: first-chart/templates/secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: first-secret
type: Opaque
data:
  username: YWRtaW4=
  password: NHc1NzIkOXNuczEmIQ==
---
# Source: first-chart/templates/cm.yaml
...
```

Now we can see both the config map and the secret object. Let's deploy the chart and inspect the secret:

```
$ helm upgrade first-chart .
$ kubectl get secrets
NAME                                TYPE                 DATA   AGE
first-secret                        Opaque               2      43s
sh.helm.release.v1.first-chart.v1   helm.sh/release.v1   1      23m
sh.helm.release.v1.first-chart.v2   helm.sh/release.v1   1      15m
sh.helm.release.v1.first-chart.v3   helm.sh/release.v1   1      43s
$ kubectl describe secret first-secret
Name:         first-secret
Namespace:    default
Labels:       app.kubernetes.io/managed-by=Helm
Annotations:  meta.helm.sh/release-name: first-chart
              meta.helm.sh/release-namespace: default

Type:  Opaque

Data
====
password:  13 bytes
username:  5 bytes
```

We don't see the exact secret data, but we see the number of bytes, which is what we expect. Great! Yet another K8s object type deployed by you with Helm.

Roll back a Helm release

Sometimes it happens that you release a version of your application that breaks in an unexpected way. Usually the best course of action is to quickly roll back that version. First you need to decide which version do you want to roll back to: the previous one or something older. You can see the history of your installed helm chart revisions by running:

```
$ helm history first-chart
REVISION        UPDATED                         STATUS          CHART                 ␣
↪APP VERSION     DESCRIPTION
1               Sat Feb  3 16:44:00 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Install complete
2               Sat Feb  3 16:51:58 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Upgrade complete
3               Sat Feb  3 17:07:00 2024        deployed        first-chart-0.1.0     ␣
↪1.16.0          Upgrade complete
```

If you want to roll back to the previous revision, you run:

```
$ helm rollback first-chart
Rollback was a success! Happy Helming!
$ helm history first-chart
REVISION        UPDATED                         STATUS          CHART                 ␣
↪APP VERSION     DESCRIPTION
1               Sat Feb  3 16:44:00 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Install complete
2               Sat Feb  3 16:51:58 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Upgrade complete
3               Sat Feb  3 17:07:00 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Upgrade complete
4               Sat Feb  3 21:36:36 2024        deployed        first-chart-0.1.0     ␣
↪1.16.0          Rollback to 2
```

That succeeded. We can see that in the description of the last revision: `Rollback to 2`. If you want to roll back to a certain revision, run:

```
$ helm rollback first-chart 1
Rollback was a success! Happy Helming!
$ helm history first-chart
REVISION        UPDATED                         STATUS          CHART                 ␣
↪APP VERSION     DESCRIPTION
1               Sat Feb  3 16:44:00 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Install complete
2               Sat Feb  3 16:51:58 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Upgrade complete
3               Sat Feb  3 17:07:00 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Upgrade complete
4               Sat Feb  3 21:36:36 2024        superseded      first-chart-0.1.0     ␣
↪1.16.0          Rollback to 2
5               Sat Feb  3 21:39:21 2024        deployed        first-chart-0.1.0     ␣
↪1.16.0          Rollback to 1
```

Good job! Now you know how to roll back.

Advanced features

Render a ConfigMap value dynamically with Helm templating

Let's learn how we can use the Helm templating system to create dynamic values in the final K8s objects. Let's do that in our ConfigMap. Currently, it has a static name: `first-chart-configmap`. Let's add the chart version number to it. That value is stored in the `Chart.yaml` file and right now that chart is the version 0.1.0. We could hard-code this version in the ConfigMap name, but we want to update the chart version often, and we don't want to keep on changing it manually in the name of the ConfigMap.

This is where Helm templating comes in handy. The templating engine will pull values from the `Chart.yaml` and `values.yaml`. All you have to do is use the templating directive `{{ <something> }}`. To tell Helm to take the value from `Chart.yaml` file, we reference the value with `{{.Chart.<key>}}`, in this case `{{.Chart.Version}}`. Both are capitalized, because that is the convention in Go, the programming language the Helm is written in. So our ConfigMap now looks like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: first-chart-configmap-{{.Chart.Version}}
...
```

Now let's upgrade our chart:

```
$ helm upgrade first-chart .
$ kubectl get cm
NAME                          DATA   AGE
first-chart-configmap-0.1.0   2      14s
```

Yay! We can see the new version in the name of the ConfigMap. Now change the version in `Chart.yaml` to `0.1.1` and upgrade the chart:

```
$ helm upgrade first-chart .
$ kubectl get cm
NAME                          DATA   AGE
first-chart-configmap-0.1.1   2      16s
```

The name of the config map has been updated! Great! Now you know how to pass a chart value to Helm via the templating engine.

Using the values.yaml file

Like `Chart.yaml`, `values.yaml` is a builtin object in Helm. Meaning that the values in these files can automatically be templated using the template directive syntax. The generated `values.yaml` files contains quite a few values predefined, but these are just suggestions, so you can remove or add any values you like. For example if we wanted to have some values to be passed for our staging environment, we could add some data like this to the `values.yaml`: with this content:

```
...
staging:
```

(continues on next page)

(continued from previous page)

```
  sample-key: sample-12345
...
```

If we wanted to access that value in a template, we would use `{{ .Values.staging.sample-key }}`. When you deploy a Helm chart, you can also pass values that will overwrite the values in the `values.yaml` from the chart.

Dynamically render a value with a Helm conditional statement

Like programming languages, the Helm templating system has conditionals, which allows you to render values dynamically, whether a condition is met or not.

Imagine that you have two different environments. Staging is the place where developers can push commits and trigger a set of automated tests to see if the application still works with their changes. And production, where your workloads are handling actual traffic from your users and therefore doesn't need tests to run because the change has already been approved by your CI/CD system. In this scenario we want the value of `allowTesting` to be true for staging and false for production. The if-else statement is a Helm directive and each part of the statement, if, else, and end, begin and end with double curly braces. Let's add those to the ConfigMap:

```
...
data:
  port: "8080"
  {{if eq .Values.env "staging"}}
  allowTesting: "true"
  {{else}}
  allowTesting: "false"
  {{end}}
```

Also let's add to `values.yaml`:

```
...
env: production
...
```

Let's see the change:

```
$ helm template first-chart .
...
---
# Source: first-chart/templates/cm.yaml
...
  allowTesting: "false"
```

Look at that! `allowTesting` got set to false. Now try setting `env: staging` in `values.yaml` and see what happens with `allowTesting`.

Helm going forward

In conclusion, the idea of Helm is pretty simple: you put in `templates` files with any K8s objects you want to be templated. In those you can use the Helm templating system to pass values from `values.yaml` and `Chart.yaml`. If you know how to define a K8s object, you can add it to Helm. We've only tried a minor set of features of the Helm templating systems: values and conditionals. But you can write more complicated code, like loops, or define your own functions in `_helpers.tpl`.

Happy Helming!

6.3 Exam

The *Kubernetes + Helm Exam* can be used to evaluate your understanding of K8s and Helm.

Please complete the task and send us the resulting Helm chart.

6.3.1 Kubernetes + Helm exam

Instructions

Create a Helm chart for the `pod-info-app` application from the K8s pre-work. For the most part this entails copying into the Helm chart's `templates` directory the K8s YAML files you created in the K8s pre-work. Moreover, you'll need to make some things configurable there.

Zip the Helm chart and send it to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work K8s Helm exam submission".

Requirements:

- Use the `exam` namespace.
- Have a deployment that creates a number of replicas for the `pod-info-app` pods. Make the **number of replicas configurable** in `values.yaml` with the key `replicaCount`.
- Add a service. Make the external port configurable with the key `externalPort`.
- Have a `debug` value in `values.yaml`. If `debug: "true"` your chart should deploy a `busybox` deployment.
- Make the name of the deployments end with the version of the chart.

CHAPTER

SEVEN

PYTHON

You will need a solid grasp of the fundamentals of Python before starting Dev School. We will work through more advanced features of the language but expect you to be able to code to a basic level already. If you can complete the attached exercise then you will be at a comfortable level to begin the school.

7.1 Resources

7.1.1 Recommended Exercises

Write code! The best way to get better at a programming language is by writing code.

One way to do this is through Code Katas. There are many lists of Code Katas online. Here is one:

- Code Katas

7.1.2 Recommended Reading Material

- Python 3 Tutorial
- Alternative Tutorial
- Practical Programming concepts in Python (You can skip Chapter 7)

7.2 Exercise

Exercise

7.3 Checklist

7.3.1 Python Language Basics

You should be familiar with the syntax for programming in Python, especially following:

- Essential Syntax and Fundamentals
  - Defining variables and functions
  - Defining classes and inheritance
    - Operator overloading

  - Type hinting (especially if you have not worked with a statically typed language)
  - Operators, control flow
  - `range` and `enumerate`
  - `iter` and `next`
  - Standard library data structures (lists, dicts, sets, tuples) and unpacking syntax
  - f-strings
  - Comprehensions
- Useful items
  - Decorators
  - Dataclasses
  - Context Managers
  - File handling
  - Modules and packages
- Optional
  - Multithreading and multiprocessing
  - Higher order functions (eg passing a function to `map` a collection)
    * lambda functions
    * closures
  - Generators
  - Async python
  - 'Modern' python (3.11) features such as `match case`
- Environments
  - Setting up a `venv` or a conda `env`
  - Jupyter notebooks (try installing and running one locally)

Python Fundamentals Exercise

Bank Transactions Simulator

For this exercise, you will build a bank transactions simulator. Users can create accounts, make deposits, withdrawals, and view transaction history. While you should practice building a small project, it may be useful to experiment with the code in a jupyter notebook first. You may use Google / ChatGPT / any other resources to help if a concept is unfamiliar for you.

Python Basics

1. Create some datamodels to model the bank accounts and transactions, using `@dataclass`. You will need
    - Account (Needs to have an `account_holder`, `account_number`, `balance` and a list of `transactions` (see next line). What datatypes are best to represent each field)?
    - Transaction (these will populate the list held in each account). What types of transaction are there? What fields are needed to represent them and can any be optional?
    - e.g. think about transfers, which move money to or from another account, and deposits/withdrawals.
2. Create `CurrentAccount` and `SavingsAccount` classes that inherit from the `Account` class with additional properties if needed. (e.g. daily limits)
3. Overload the "less than" operator for `Account` class to compare the balance of two accounts.
    - Hint: which dunder methods are used for comparison operators?
4. What is printed when you pass an `account` object to `print`? Add an override to the Account class so that this instead prints a nicely formatted account statement.
    - Hint: which dunder method defines how an object is converted to a string?
    - Multiline f-strings
5. Implement a `Bank` class which holds accounts.
    - What would be a good datastructure use to model this? Consider which fields of an `account` are guaranteed to be unique for each account
    - Pull account(s) information for a given `account_holder`.
    - Add a method which create new accounts for a list of new `account_holders`. First implement this with a loop, then try changing it to a comprehension.
    - Add methods to deposit/withdraw money into an account.
    - Add a method which can transfer money from one account to another, creating the relevant `Transactions` and updating the accounts.
    - What should happen when someone tries to transfer or withdraw more money than they have?
6. When a transaction occurs, write the transaction to a csv file.
    - This should be a single new file each time you create the `Bank`. Try using the `datetime` library to put the current time in the filename.
    - You can use the `open` keyword as a context manager
    - Consider what columns the csv file should have and write these as the first line, when the `Bank` is initialised.
    - Append each transaction to this file as they occur.
    - Hint: You can use `",".join(...)` to construct the line.
7. Write a script which can
    - Generate 100 account names (random strings are fine) and initialise the bank with an account for each.
    - Deposit some money into each of them,
    - Simulate 1000 random transactions between accounts. Choose two accounts at random and pick a random amount. What bounds might make this sensible?
    - Check your log file is the expected length. Make sure you create a new log file each time you run the simulation!

8. The bank wants to find inactive accounts. Use `filter` with a lambda function to find accounts with less than 10 transactions.

CHAPTER

EIGHT

PANDAS

Pandas is the industry standard Python Data Analysis library and is widely used throughout IMC.

8.1 Recommended Reading Material

- Pandas Getting Started Tutorials - These three chapters:
  - Data Type Handled by Pandas
  - Reading/Writing Tabular Data
  - Selecting and Filtering
  - We recommend the rest of the chapters only if your work will involve data analysis on a regular basis.

8.2 Checklist

8.2.1 Pandas

You should be able to do the following using Pandas

- Data Frames
  - Data Selection and Filtering
  - Data Grouping and Aggregation - optional
  - Data Reshaping and Pivot tables - optional
  - Combine data from multiple tables - optional
  - Query time series - optional
- Data Input and Output
  - Read data from external sources such as CSV, Excel, databases, etc.
- Visualizations
  - Generate Charts and Tables - optional

8.3 Exercise

Exercise

8.3.1 Pandas Exercise

This exercise is a continuation of the Bank Transactions Simulator, using Pandas to analyze the simulated transactions.

Pandas

1. Set up your project within a virtual environment, and `pip install pandas`
2. After running the simulation and generating the CSV file with transaction data, perform the following tasks in a notebook, using pandas:
3. Load your CSV file of transactions into a DataFrame.
4. Use `group_by` and `concat` to construct a new dataframe with `account_number` as the index and the following columns
    - number of transactions
    - current balance
    - `min_balance` and `max_balance` over the whole simulation
5. Reproduce the list of `inactive_accounts` from the first part, using Pandas operations on the transactions dataframe. Is the list of transactions enough information to guarantee we find them all?
6. If your bank implementation allows negative balance, find out which accounts went into overdraft at any point. If not, check if anyone's balance dipped below a threshold.
7. Find which account made the most transactions and plot the balance of this account over time.

CHAPTER

NINE

PYCHARM

An IDE is essential to improving your productivity as a Software Engineer. PyCharm is the preferred Python IDE for the vast majority of Python teams at IMC, and the IDE with the most internal support.

9.1 Recommended Reading

- PyCharm - Getting Started

9.2 Recommended Viewing

- Youtube - Be More Productive With IntelliJ IDEA
- Youtube - IntelliJ IDEA Debugger Essentials

9.3 Practical skills

Be able to use the following features:

- Reading Code
  - Code Navigation
  - Find Usages
- Writing Code
  - Inspections
  - Live Templates
  - Local History
  - Code Style and Formatting
- Refactoring
  - Renaming
  - Change Signature
  - Extract/Introduce refactorings
  - Inline

  - Safe Delete
  - Migrate
- Running
  - Running a Python application
  - Running tests
- Debugging
- Venv Integration