Cloud Native Foundations Workshop

Release 0.1.0

Oleksandr Sirenko

Jun 12, 2023

HOW TO USE:

Cloud Native Foundations Workshop is an educational guide for students learning cloud-native application development.

This learning guide relates to SUSE Cloud Native Foundations Scholarship Program from Udacity and covers all hands-on exercises in the Cloud Native Foundations Course.

CHAPTER

ONE

OBJECTIVE

The main goal of this project is to build a knowledge base and help solve technical issues outside the scope of the Cloud Native Foundation Course.

CHAPTER

TWO

CONTENT

The workshop contains exercises and solutions on selected topics that will help you understand the basics of building, deploying, and maintaining cloud-native applications. This is NOT a comprehensive guide, but rather a focused look at a few key topics:

- Flask web development best practices.
- App containerization with Docker.
- Releasing applications to a Kubernetes cluster.
- Automation software development workflows with GitHub Actions.
- Using ArgoCD to build reliable CI/CD pipelines.

All the supporting code is in the GitHub repository.

Check out the *Quick Start* section for further information, including *how to set up your workshop environment*.

CHAPTER

THREE

REQUIRED KNOWLEDGE

- Python 3.
- Flask web development (introductory level).
- Networking (REST, protocols, HTTP methods).
- Git (basic commands, working with remote repositories).
- Linux shell commands.

3.1 Quick Start

3.1.1 Install Tools

1. VS Code (or another IDE of your choice)
2. Python
3. Git

3.1.2 Obtain Data

1. On GitHub, navigate to the workshop repository
2. Fork the workshop repository.
3. On GitHub, navigate to **your fork** of the workshop repository and copy the URL.
4. Clone forked repository to your local machine using git clone command. It will look like this, with your GitHub username instead of `YOUR_USERNAME`:

```
git clone https://github.com/YOUR_USERNAME/cloud-native-foundations
```

See details on how to fork and clone the repository.

3.1.3 Generate Documentation

The documentation is the core of the workshop. You can generate it automatically with a single line of code and use it locally without internet access. Moreover, you can edit and improve documentation, contributing to this project, or build your own knowledge base. To make it possible just follow the instructions:

1. Open the workshop folder with IDE and run terminal
2. Create the virtual environment: `python3 -m venv venv`
3. Activate the virtual environment: `source venv/bin/activate`
4. Install dependencies: `pip install -r requirements.txt`
5. Change the working directory to docs: `cd docs`
6. Generate a local copy of workshop documentation by running `make html`
7. Open the local copy of the documentation in a web browser: `firefox build/html/index.html`

Now you have everything you need to get the most out of this workshop.

3.1.4 Download Documentation

You can download the Cloud Native Foundations Workshop docs in the following formats:

- PDF
- ePub
- Zipped HTML

3.1.5 How To Use Workshop

At the moment, the workshop includes two main sections: exercises and solutions. Follow the instructions in each exercise and try to solve the problems.

Explore the *Learning Resources* to brush up on the current topic, and feel free to check out the solutions.

3.2 Learning Resources

3.2.1 Flask

1. Flask Quick Start
2. Flask URL Route Registrations
3. Flask Logging Documentaion
4. Logging Facility for Python
5. Python Basic Logging Tutorial
6. Python Logging Cookbook
7. Python Advanced Logging Tutorial
8. CS50 2020 - Lecture 9 - Flask
9. Flask Application Video Tutoral from Tech with Tim

10. Learn Flask for Python - freeCodeCamp Video Tutorial

3.2.2 Docker

1. Docker Tutorial for Beginners - 3 Hour Video Course
2. Docker Hub Quickstart
3. Build a Docker Image
4. Pushing a Docker Container Image to Doker Hub
5. How to Build a Containerized Go Application with Docker
6. Docker Image Pipeline for Go
7. Build Python Docker Image
8. How to Serve a Flask App with Amazon Lightsail Containers
9. Docker Cheet Sheet
10. How To Install and Use Docker on Ubuntu 20.04
11. A Beginner-Friendly Introduction to Containers, VMs and Docker

3.2.3 Kubernetes

1. Kubernetes Tutorial for Beginners - 4 Hour Video Course
2. Official Kubernetes Tutorials
3. Kubernetes Hands-on Labs
4. Vagrant Documentation Resources
5. Vagrant Cheat Sheet
6. K3s Lightweight Kubernetes
7. Stopping and starting Kubernetes cluster
8. Organizing Cluster Access Using kubeconfig Files
9. How to Manage Kubernetes With Kubectl
10. Explore kubectl Cheat Sheet
11. Kubernetes Config file
12. Using kubectl to Create a Deployment
13. How to Delete Pods from a Kubernetes Node
14. Use Port Forwarding to Access Applications in a Cluster

3.2.4 GitHub Actions

1. Create Secrets and Configure GitHub Actions
2. GitHub Actions - 30 Min Video Tutorial
3. GitHub and Git Foundations - 12 Video Lessons
4. Get Started with GitHub Actions
5. Publishing Docker Images Approach from GitHub

3.2.5 ArgoCD

1. Getting Started with Argo CD
2. Argo CD Installation Video
3. Guide To GitOps
4. Helm Quickstart Guide
5. Helm Deployment with Argo CD Video Tutorial
6. CI/CD Guides for DevOps Engineers - 8 Video Playlist

3.3 1. Application Endpoints and Logging

In this exercise, you will use a minimal Flask application that looks like this:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
   return "<p>Hello World!</>"

if __name__ == "__main__":
   app.run(host="0.0.0.0")
```

Your tasks are to extend application endpoints and implement the logging functionality. If you are not familiar with Flask, start with the Flask Quickstart documentation and watch the Flask CS50 video tutorial at the top of this page.

3.3.1 Preparation

Note: Make sure you have *installed all essentials (IDE, Python, Git)*

Follow these steps to set up the environment for the exercise:

1. Create the virtual environment: `python3 -m venv venv`
2. Activate the virtual environment: `source venv/bin/activate`

3. Change the working directory: `cd exercises/python-helloworld`
4. Install dependencies: `pip install -r requirements.txt`
5. Run the application: `python app.py`
6. Test the application in a web browser at: http://192.168.1.3:5000/

3.3.2 Exercise 1.1 - Extend Application Endpoints

The endpoint of a web application is simply a URL where a web browser can access the application. So, to extend application endpoints, you need to define new URL routes and create view functions to match those routes. For a better understanding, carefully read two sections of the Flask documentation:

- URL Route Registrations
- Response Objects

Tip: Use `@app.route()` decorator to bind a view function for the application endpoint.

Extend your Python Flask app with `/status` and `/metrics` endpoints, considering the following requirements:

- Both endpoints must return an status code `HTTP 200`
- Both endpoints must return a JSON response e.g. `{"user": "admin"}`
- The `/status` endpoint should return a response similar to this example: `result: OK - healthy`
- The `/metrics` endpoint should return a response similar to this example: `data: {UserCount: 140, UserCountActive: 23}`

3.3.3 Exercise 1.2 - Implement Application Logging

Flask uses standard Python logging module. Messages about your Flask application are logged with app.logger, which takes the same name as app.name. This logger can also be used to log your own messages [3].

Tip: Configure logging for your application as soon as possible when the program starts.

Add a log collection logic for each endpoint in your Flask application according to the following requirements:

- Logs should be kept in a separate file named `app.log`.
- You need to collect logs at the `DEBUG` level.
- Each log must be in the same format and include information about the logging level, actual time, endpoint name, and message.

3.3.4 Additional Resources

1. Flask Quick Start
2. Flask URL Route Registrations
3. Flask Logging Documentaion
4. Logging Facility for Python
5. Python Basic Logging Tutorial
6. Python Logging Cookbook
7. Python Advanced Logging Tutorial
8. CS50 2020 - Lecture 9 - Flask
9. Flask Application Video Tutoral from Tech with Tim
10. Learn Flask for Python - freeCodeCamp Video Tutorial

3.4 2. Docker for Application Packaging

In this exercise, you will use a minimal Go application that looks like this:

```
package main

import (
        "fmt"
        "net/http"
)

func helloWorld(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Sherlock Holmes sat silently...")
}

func main() {
        http.HandleFunc("/", helloWorld)
        http.ListenAndServe(":8080", nil)
}
```

Your tasks are to containerize this Go application and push it to Docker Hub. But before you get started, we highly recommend watching the Docker introductory video at the top of the page and following through with the How to Build a Containerized Go Application with Docker tutorial.

3.4.1 Preparation

Note: This application listens on port `8080`

1. Download and install Docker Engine
2. Install Go: `sudo apt install golang-go`
3. Change the working directory: `cd exercises/go-helloworld`
4. Run Go app using the terminal command: `go run main.go`
5. Check your app in a web browser at: http://127.0.0.1:8080/

3.4.2 Exercise 2.1 - Create Dockerfile

Dockerfile requirements:

- Use an image that is based on the latest stable Go environment.
- Set the proper working directory.
- Copy source code into the image.
- Implement logic to build the application.
- Make sure that app access on a default port `8080`.
- Add the command to start the container.

3.4.3 Exercise 2.2 - Build a Docker Image

Docker image requirements:

- Use the `go-helloworld` name as the image name.
- Tag the image to `YOUR_DOCKERHUB_USERNAME/go-helloworld:v1.0.0`, (don't forget to replace `YOUR_DOCKERHUB_USERNAME` with your Docker Hub username).

3.4.4 Exercise 2.3 - Push a Docker Container to Docker Hub

Note: To push a Docker container image to Docker Hub, you need to have a Docker Hub account

The application image shoud be available on Docker Hub under the `https://hub.docker.com/r/YOUR_DOCKERHUB_USERNAME/go-helloworld`, where `YOUR_DOCKERHUB_USERNAME` is you current user name e.g. https://hub.docker.com/r/osdoc/go-helloworld.

3.4.5 2.4 Pull Image From the Docker Hub

To complete the exercise, do the tasks in the following order:

1. Remove all containers and images from your local computer.
2. Pull the image from the Docker Hub repository.
3. Run the pulled image on your local computer.
4. Verify go-helloworld application at: http://127.0.0.1:8080/

3.4.6 Exercise 2.5 (Optional) - Containerize Python Application

This is an exercise to consolidate the material covered in the section. Your job is to containerize the Flask application that you created in the first lesson.

Follow all these steps to reach your goal:

1. Create Dockerfile.
2. Build a Docker image.
3. Run newly built image as a container.
4. Check the Flask application at http://127.0.0.1:5000/
5. Push the Docker container to Docker Hub.
6. Remove container from the local machine.
7. Pull the container from Docker Hub.
8. Run pulled container.

3.4.7 Additional Resources

1. Docker Tutorial for Beginners - 3 Hour Video Course
2. Docker Hub Quickstart
3. Build a Docker Image
4. Pushing a Docker Container Image to Doker Hub
5. How to Build a Containerized Go Application with Docker
6. Docker Image Pipeline for Go
7. Build Python Docker Image
8. How to Serve a Flask App with Amazon Lightsail Containers
9. Docker Cheet Sheet
10. How To Install and Use Docker on Ubuntu 20.04
11. A Beginner-Friendly Introduction to Containers, VMs and Docker

3.5 3. Kubernetes Cluster

In the previous exercise, you put the Go app into a Docker container and pushed it to the Docker Hub repository. So now you need to deploy this application from your Docker Hub repository to your local Kubernetes cluster.

Note: This exercise is slightly different from the original one - it was modified according to more practical use cases and connect student progress from the previous lessons.

3.5.1 Preparation

Install VirtualBox and Vagrant

1. Install VirtualBox: `sudo apt install virtualbox`
2. Install Vagrant: `sudo apt-get update && sudo apt-get install vagrant`

Up Virtual Machine

1. Change directory to exercises: `cd exercises`
2. Init virtual machine: `vagrant init`
3. Up the virtual machine: `vagrant up`
4. Check the status of virtual machine: `vagrant status`

Useful Vagrant Commands

- Shut down virtual machine forcefully: `vagrant halt`
- Suspend the virtual machine: `vagrant suspend`
- Restart virtual machine: `vagrant up`

Find out more in Vagrant Cheat Sheet

3.5.2 Exercise 3.1 - Create Kubernetes Cluster

- Create k3s Kubernetes cluster on the local machine using Virtual Box and Vagrant.

3.5.3 Exercise 3.2 - Deploy Application to the Kubernetes Cluster

- Run go-helloworld app at the Kunernetes cluster from the Docker Hub.
- Deploy go-helloworld to the Kubernetes cluster from the Docker Hub.

3.5.4 Exercise 3.3 - Define and Deploy Kubernetes Resources

- Deploy the following resources using the kubectl command:
  - **a namespace:**
    - name: `demo`
    - label: `tier: test`
  - **a deployment:**
    - image: `nginx:alpine`
    - name: `nginx-apline`
    - namespace: `demo`
    - replicas: 3
    - labels: `app: nginx`, `tag: alpine`
  - **a service:**
    - expose the above deployment on port `8111`
    - namespace: `demo`
  - **a configmap:**
    - name: `nginx-version`
    - containing key-value pair: `version=alpine`
    - namespace: `demo`

Note: Nginx is one of the public Docker images, that you can access and use for your exercises or testing purposes.

Make sure the following tasks are completed:

You have created a Namespace You have created a Deployment You have created a Service You have created a Configmap

3.5.5 Additional Resourses

1. Kubernetes Tutorial for Beginners - 4 Hour Video Course
2. Official Kubernetes Tutorials
3. Kubernetes Hands-on Labs
4. Vagrant Documentation Resources
5. Vagrant Cheat Sheet
6. K3s Lightweight Kubernetes
7. Stopping and starting Kubernetes cluster
8. Organizing Cluster Access Using kubeconfig Files
9. How to Manage Kubernetes With Kubectl
10. Explore kubectl Cheat Sheet

11. Kubernetes Config file
12. Using kubectl to Create a Deployment
13. How to Delete Pods from a Kubernetes Node
14. Use Port Forwarding to Access Applications in a Cluster

3.6 4. GitHub Actions

3.6.1 Exercise 4.1 Dockerize Application with GitHub Actions

Warning: To securely access the Docker Hub repository, you need to create two secrets in your GitHub account: `DOCKER_HUB_USERNAME` and `DOCKER_HUB_ACCESS_TOKEN` respectively. Read more here.

- Implement CI/CD automation pipeline usind GitHub Actions.

3.6.2 Additiona Resources

1. Create Secrets and Configure GitHub Actions
2. GitHub Actions - 30 Min Video Tutorial
3. GitHub and Git Foundations - 12 Video Lessons
4. Get Started with GitHub Actions
5. Publishing Docker Images Approach from GitHub

3.7 5. Continuous Delivery Fundamentals

In this exercise, you will use Argo CD to automate the delivery of an application to a Kubernetes cluster.

Continuous Delivery (CD) is the ability to get code changes reliably to production environments. This practice should be automated and should enable developers to provide value to consumers efficiently.

3.7.1 Preparation

Prepare your environment by following *Preparation* section in the exercise *3. Kubernetes Cluster*

3.7.2 Exercise 5.1 - Deploy the Application Using Argo CD

1. Install Argo CD by referring to the installation guide.
2. Expose the `argocd-server` using a `NodePort` service on port `30007` on HTTP and `30008` on HTTPS. The YAML manifest for the NodePost service can be found in the tutorial repository.
3. Access the Argo CD UI by going to `https://192.168.50.4.30008` or `https://192.168.50.4.30007`.
4. Insert login credentials by using the credentials guide.
5. Create an Argo CD application named `nginx-alpine` and use the manifests provided in the course repository.

3.7.3 Configuration Managers

For the management of multiple declarative Kubernetes manifests, a templating layer is necessary, especially if the application is replicated across different regions. For this purpose configuration managers, such as Helm and Kustomize, were introduced.

This exercise will focus on creating your first Helm chart to deploy multiple Nginx applications using the same template and multiple input files.

3.7.4 Exercise 5.2 - Create a Helm Chart with Argo CD

Using the manifests provided in the course repository, create a helm chart (Chart.yaml, templates, values.yaml) that will template the following parameters:

- **namespace name**
- **replica count**
- **image:**
  - name
  - tag
  - pull policy
- **resources:**
  - requests for CPU and memory
- **service:**
  - port
  - type (e.g. ClusterIP)
- **configmap data** (e.g. the key-value pair)

The chart details should be as following:

- **name:** nginx-deployment
- **version:** 1.0.0
- **keywords:** nginx

3.7.5 Exercise 5.3 - Create a YAML Values Files

Once the Helm chart is available make sure that a default `values.yaml` file is available. This values file will be used as a default input file for the Helm chart. The `values.yaml` file should have the following specification:

`values.yaml`

- namespace name: demo
- replica count: 3
- image repository: nginx
- image tag: alpine
- image pull policy: IfNotPresent
- resources: CPU 50m and memory 256Mi
- service type: ClusterIP
- service port: 8111
- configmap data: "version: alpine"

Next, create two values files with the following specifications:

`values-staging.yaml`

- namespace name: staging
- replica count: 1
- image repository: nginx
- image tag: 1.18.0
- resources: CPU 50m and memory 128Mi
- configmap data: "version: 1.18.0"

`values-prod.yaml`

- namespace name: prod
- replica count: 2
- image repository: nginx
- image tag: 1.17.0
- resources: CPU 70m and memory 256Mi
- service port: 80
- configmap data: "version: 1.17.0"

3.7.6 Exercise 5.4 - Create Argo CD Applications

Using the values files above (values-prod, values-staging), create two Argo CD applications, `nginx-staging` and `nginx-prod` respectively. These should deploy the nginx Helm Chart referencing each input values files.

3.7.7 Additional Resources

1. Getting Started with Argo CD
2. Argo CD Installation Video
3. Guide To GitOps
4. Helm Quickstart Guide
5. Helm Deployment with Argo CD Video Tutorial
6. CI/CD Guides for DevOps Engineers - 8 Video Playlist

3.8 1. Application Endpoints and Logging

Note: Make sure you have *prepared the environment* for this task.

3.8.1 Solution 1.1 - Extend Application Endpoints

To extend Python Flask application with `/status` and `/metrics` endpoints, follow these steps:

1. Open `app.py` file in `exercises/python-helloworld` folder.
2. Register routes "`/status`" and "`/metrics`" to the app route.
3. Add the logic to those routes, according to the examples below:

```
@app.route("/status")
def status():
    response = app.response_class(
        response=json.dumps({"result": "OK - healthy"}),
        status=200,
        mimetype="application/json"
    )

    return response
```

```
@app.route("/metrics")
def metrics():
    response = app.response_class(
        response=json.dumps(
            {
                "status": "success",
                "code": 0,
                "data": {"UserCount": 140, "UserCountActive": 23}
            }
```

(continues on next page)

(continued from previous page)

```
        ),
        status=200,
        mimetype="application/json"
    )

    return response
```

Watch the video to understand the API Endpoints solution in more detail.

3.8.2 Solution 1.2 - Implement Application Logging

For log messages Flask uses standard Python `logging` library. In Flask it implemented within `app.logger` method, which can be used in the same way and allow you to handle your own masseges. To add basic logging functionality to your Flask application follow the next steps:

1. Open `app.py` file in `exercises/python-helloworld` folder.
2. Import `logging` library.
3. Add log collection logic for each route you want to track:

```
@app.route("/metrics")
def metrics():
    ...
    app.logger.info("Metrics request successfull.")
```

- Add logging configuration to main function to save the logs in a file:

```
if __name__ == "__main__":
    logging.basicConfig(
        filename="app.log",
        level=logging.DEBUG,
        format="%(levelname)s:%(asctime)s:%(name)s:%(message)s",
    )

    app.run(host="0.0.0.0")
```

Watch the video to understand the logging basics in more detail.

Note: In this tutorial, we discover the very basics of logging. The complete solution code is in solutions/python-helloworld/app.py. To improve the app logging facility and create a more sophisticated solution explore the additional materials for the exercise.

3.9 2. Docker for Application Packaging

Note: Make sure you have *prepared the environment* for this task.

3.9.1 Solution 2.1 - Create Dockerfile

1. Create Dockerfile: `touch Dockerfile`
2. Open the Dockerfile.
3. Create layers due to task:

```
# syntax=docker/dockerfile:1

FROM golang:1.16-alpine

WORKDIR /go/src/app

ADD . .

RUN go mod init

RUN go build -o go-helloworld

EXPOSE 8080

CMD ["./go-helloworld"]
```

4. Save changes

3.9.2 Solution 2.2 - Build a Docker Image

1. Make sure you are in the `/exercises/go-helloworld/` directory.
2. Build a Docker image using the prompt command:

```
docker build -t go-helloworld .
```

3.9.3 Solution 2.3 - Push a Docker Container to Docker Hub

Note: To push a Docker container to Docker Hub, you need to have a Docker Hub account

Follow these steps to push a Docker container to Docker Hub:

1. Run a Docker image as a container: `docker run -p 8080:8080 --name go_moriarty -d go-helloworld`
2. Verify if the application at: http://127.0.0.1:8080/

3. Login to the Docker Hub: `docker login -u "YOUR_DOCKERHUB_USERNAME" -p "YOUR_DOCKERHUB_PASSWORD" docker.io`
4. Tag the image: `docker tag go-helloworld YOUR_DOCKERHUB_USERNAME/go-helloworld:v1.0.0`
5. Push the image to the DockerHub repo: `docker push YOUR_DOCKERHUB_USERNAME/go-helloworld:v1.0.0`

Feel free to check the video Docker for Application Packaging

3.9.4 2.4 Pull Image From the Docker Hub

Remove all builds and images:

- Explore the processes: `docker ps -a`
- Stop the container: `docker stop go_moriarty`
- Remove the container: `docker rm go_moriarty`
- Explore the images: `docker images`
- Remove the image from local machine: `docker rmi YOUR_DOCKERHUB_USERNAME/go-helloworld:v1.0.0`

Pull container from your Docker Hub repository:

- Pull the image: `docker pull YOUR_DOCKERHUB_USERNAME/go-helloworld:v1.0.0`
- Run the image as a container: `docker run -p 8080:8080 --name go_pulled -d YOUR_DOCKERHUB_USERNAME/go-helloworld:v1.0.0`
- Verify go-helloworld application at: http://127.0.0.1:8080/

Useful Docker Commands

- Get container name: `docker ps -a`
- Stop the container: `docker stop CONTAINER_NAME`
- Remove the container: `docker rm CONTAINER_NAME`
- Stop all the processes: `docker kill $(docker ps -q)`
- Remove all the processes: `docker rm $(docker ps -a -q)`
- Remove all the images: `docker rmi $(docker images -q)`

Find out more helpful commands in the Docker Cheat Sheet.

3.9.5 Solution 2.5 - Dockerize Python Flask Application

In this step, we have gained a basic understanding of containerizing applications with Docker. So let's take a look at the process of dockerizing a Pyhton Flask application.

Use this recipe to put your Python Flask application in a Docker container and push it to the Docker Hub:

1. Install Docker if not have it yet: `sudo apt-get install docker.io`
2. Activate Python virtual environment: `source venv/bin/activate`
3. Change working directory: `cd exercise/python-helloworld`

4. Create a Dockerfile:

```
# syntax=docker/dockerfile:1

FROM python:3.8

WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt

EXPOSE 5000

CMD [ "python", "app.py" ]
```

3. Run the Docker image: `docker run -p 5000:5000 --name monty_python -d python-helloworld`
4. Verify if the application is available and work properly at: http://127.0.0.1:5000/
5. Build a Docker image: `docker build -t python-helloworld .`
6. Tag application: `docker tag python-helloworld YOUR_DOCKERHUB_USERNAME/python-helloworld:v1.0.0`
7. Login to Docker Hub: `docker login`
8. Push the application to Docker Hub: `docker push YOUR_DOCKERHUB_USERNAME/python-helloworld:v1.0.0`

3.9.6 Common Errors & How to Fix Them

Permission Denied Error

Error: `Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post http://%2Fvar%2Frun%2Fdocker.sock/v1.24/build?buildargs=%7B%7D&cachefrom=%5B%5D&cgroupparent=&cpuperiod=0&cpuquota=0&cpusetcpus=&cpusetmems=&cpushar dial unix /var/run/docker.sock: connect: permission denied`

To fix **Permission Denied Error**, follow the next steps:

1. Create the docker group: `sudo groupadd docker`
2. Add your user to the docker group: `sudo usermod -aG docker ${USER}`
3. Set the superuser: `su -s ${USER}`
4. Check if Docker work correctly: `docker ps -a`
5. Build Docker image: `docker build -t go-helloworld .`

Follow the link to learn more about how to fix **Permission Denied Error**.

File Not Found Error

Error: `go: go.mod file not found in current directory or any parent directory; see 'go help modules'`

This error refers to building a Docker image for the Golang application and means you need to init `go.mod` file. To fix **File Not Found Error**, follow these steps:

1. Open the Dockerfile of your Go application.
2. Add `RUN go mod init` layer before the `RUN go build -o helloworld` layer.
3. Save the changes.
4. Build Docker image using `docker build -t go-helloworld .` command.

After these manipulations, the Dockerfile should look like this:

```
FROM golang:alpine

WORKDIR /go/src/app

ADD . .

RUN go mod init

RUN go build -o helloworld

EXPOSE 8080

CMD ["./helloworld"]
```

3.10 3. Kubernetes Cluster

Note: Make sure you have *prepared the environment* for this task.

3.10.1 Solution 3.1 - Create Kubernetes Cluster

Start Kubernetes Cluster

1. Open Vagrant shell: `vagrant ssh`
2. Get k3s: `curl -sfL https://get.k3s.io | sh`
3. Check nodes: `kubectl get no`

Warning: To stop the Kubernetes cluster, run the command as the root user: `shutdown -h now`

3.10.2 Solution 3.2 - Deploy Application to the Kubernetes Cluster

1. Run the app at a cluster: `kubectl run POD_NAME --image=DOCKER_IMAGE_PATH`
2. Check the pod status using one of the following commands: `kubectl get pods`
3. Deploy the app to the cluster directly from the Docker Hub: `kubectl create deployment DEPLOYMENT_NAME --image=docker.io/DOCKERHUB_USERNAME/DOKER_IMAGE_NAME:TAG`
4. Access the application on localhost: `kubectl port-forward POD_NAME 8080:8080`

Kubeconfig

- K3s stores the kubeconfig file under `/etc/rancher/k3s/k3s.yaml`
- API server - https://127.0.0.1:8080
- Authentication mechanism - username (admin) and password

Useful kubectl Commands

- Get the control plane and add-ons endpoints: `kubectl cluster-info`
- Get all the nodes in the cluster: `kubectl get nodes`
- Get extra details about the nodes: `kubectl get nodes -o wide`
- Get all the configuration details about the node: `kubectl describe node NODE_NAME`
- Get basic pods information: `kubectl get pods`
- Check the detailed information of a particular pod: `kubectl describe pod POD_NAME`
- Delete pod: `kubectl delete pod POD_NAME`

Find out more in kubectl Cheat Sheet

Explore kubectl predefined assets in the video tutorial

3.10.3 Solution 3.3 - Define and Deploy Kubernetes Resources

1. Create the namespace:

```
kubectl create ns demo
```

2. Label the namespace:

```
kubectl label ns demo tier=test
```

3. Create the nginx-alpine deployment:

```
kubectl create deploy nginx-alpine --image=nginx:alpine  --replicas=3 --namespace demo
```

4. Label the deployment:

```
kubectl label deploy nginx-alpine app=nginx tag=alpine --namespace demo
```

5. Expose the nginx-alpine deployment:

```
kubectl expose deployment nginx-alpine --port=8111 --namespace demo
```

6. Create a config map:

```
kubectl create configmap nginx-version --from-literal=version=alpine --namespace demo
```

3.10.4 Common Errors & How to Fix Them

Permission Denied Error

The error message might look like this:

```
Error:  Unable to read /etc/rancher/k3s/k3s.yaml, please start server with
--write-kubeconfig-mode to modify kube config permissions error:  error loading config
file "/etc/rancher/k3s/k3s.yaml":  open /etc/rancher/k3s/k3s.yaml:  permission denied
```

To fix **Permission Denied Error**, run one of these commands:

- `sudo chmod 644 /etc/rancher/k3s/k3s.yaml`: changing access in k3s.yaml
- `sudo su`: setting the superuser permissions

Create Container Error

This type of error related to the status of the pod. The error message might look like this:

```
Error:  Error:  container create failed:  time="2021-04-11T21:03:07Z" level=error
msg="container_linux.go:366:  starting container process caused:  exec:  \
"cluster-kube-scheduler-operator\":  executable file not found in $PATH"
```

To fix **Create Container Error**, use command: `zypper install -t pattern apparmor`

3.11 4. GitHub Actions

Warning: To securely access the Docker Hub repository, you need to create two secrets in your GitHub account: `DOCKER_HUB_USERNAME` and `DOCKER_HUB_ACCESS_TOKEN` respectively. Read more here.

3.11.1 Solution 4.1 Dockerize Application with GitHub Actions

To Dockerize your application using GitHub Actions follow these steps:

1. Open the `docker-build.yaml` file under the `solution/` directory.
2. Edit this file to meet your needs (correct app name, version, etc.).
3. Save changes you have made.

4. On GitHub, open your fork of the workshop repository.
5. Put `docker-build.yaml` to the `.github/workflows` directory to execute.

The `docker-build.yaml` for Python Flask application should look like this:

```yaml
# This is a basic workflow to help you get started with Actions

name: Docker Build and Push

# Controls when the action will run. Triggers the workflow on push or pull request
# events but only for the master branch
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

# A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
      -
        name: Check Out Repo
        uses: actions/checkout@v2
      -
        name: Set up QEMU
        uses: docker/setup-qemu-action@v1
      -
        name: Login to DockerHub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_ACCESS_TOKEN }}
      -
        name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      -
        name: Build and push
        uses: docker/build-push-action@v2
        with:
          context: ./
          file: ./Dockerfile
          platforms: linux/amd64
          push: true
          tags: ${{ secrets.DOCKER_HUB_USERNAME }}/python-helloworld:latest
```

Watch the solution video lesson.

3.12 5. Continuous Delivery Fundamentals

Note: The preparation steps are the same as in the exercise `3. Kubernetes Cluster` Make sure you have *prepared the environment*

3.12.1 Solution 5.1 - Deploy the Application Using ArgoCD

Quick Start with ArgoCD

1. Open Vagrant shell using command:

```
vagrant ssh
```

2. Create namespace:

```
kubectl create namespace argocd
```

3. Install ArgoCD:

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/
↪manifests/install.yaml
```

3. Check ArgoCD pods:

```
kubectl get po -n argocd
```

4. Get all application services:

```
kubectl get svc -n argocd
```

Create and Apply YAML Manifests

Warning: Do not forget to change `GITHUB_USERNAME` inside the `YAML` files below to your GitHub user name!

argocd-server-nodeport.yaml

1. Create a file within the vim shell: `vim argocd-server-nodeport.yaml`
2. Write the manifest:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
  labels:
    app.kubernetes.io/component: server
    app.kubernetes.io/name: argocd-server
    app.kubernetes.io/part-of: argocd
```

(continues on next page)

(continued from previous page)

```
  name: argocd-server-nodeport
  namespace: argocd
spec:
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 8080
    nodePort: 30007
  - name: https
    port: 443
    protocol: TCP
    targetPort: 8080
    nodePort: 30008
  selector:
    app.kubernetes.io/name: argocd-server
  sessionAffinity: None
  type: NodePort
```

3. Escape the vim shell: `esc`
4. Save changes: `:wq`
5. Check the manifest: `cat argocd-server-nodeport.yaml`
6. Apply the manifest: `kubectl apply -f argocd-server-nodeport.yaml`

nginx-alpine.yaml

1. Create a file within the vim shell: `vim nginx-alpine.yaml`
2. Write the manifest:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-alpine
  namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  project: default
  source:
    path: solutions/kubernetes/manifests
    repoURL: https://github.com/GITHUB_USERNAME/cloud-native-foundations
    targetRevision: HEAD
  # Sync policy
  syncPolicy: {}
```

3. Escape the vim shell: `esc`
4. Save changes: `:wq`
5. Check if everything was written correctly: `cat nginx-alpine.yaml`

6. Apply created yaml manifest:: `kubectl apply -f nginx-alpine.yaml`

After preparing all the required manifests, check the application using the command:

```
kubectl get application -n argocd
```

Use ArgoCD in your browser at

- http://192.168.50.4.30007
- http://192.168.50.4.30008

3.12.2 Solutions 5.2 - Create a Helm Chart with ArgoCD

The Helm chart is defined in the `Chart.yaml` file, which contains the API version, name and version of the chart:

```
apiVersion: v1
name: nginx-deployment
description: Install Nginx deployment manifests
keywords:
  - nginx
version: 1.0.0
maintainers:
  - name: GITHUB_USERNAME
```

3.12.3 Solution 5.3 - Create a YAML Values Files

An example of the `values.yaml` file:

```
namespace:
  name: demo
service:
  port: 8111
  type: ClusterIP
image:
  repository: nginx
  tag: alpine
  pullPolicy: IfNotPresent
replicaCount: 3
resources:
  requests:
    cpu: 50m
    memory: 256Mi
configmap:
  data: "version: alpine"
```

The above configuration represents the default parameters of application deployment if it is not overwritten by a different values file.

Below is an example of the `values-prod.yaml` file, which will override the default parameters:

```yaml
namespace:
  name: prod
service:
  port: 80
  type: ClusterIP
image:
  repository: nginx
  tag: 1.17.0
  pullPolicy: IfNotPresent
replicaCount: 2
resources:
  requests:
    cpu: 70m
    memory: 256Mi
configmap:
  data: "version: 1.17.0"
```

3.12.4 Solution 5.4 - Create ArgoCD Applications

ArgoCD application CRD for the `nginx-prod.yaml` deployment:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-prod
  namespace: argocd
spec:
  destination:
    namespace: default
    server: https://kubernetes.default.svc
  project: default
  source:
    helm:
      valueFiles:
        - values-prod.yaml
    path: helm
    repoURL: https://github.com/GITHUB_USERNAME/cloud-native-foundations/tree/main/
↪solutions/argocd
    targetRevision: HEAD
```

Note: The `nginx-staging.yaml`, `values-staging.yaml`, and `nginx-prod.yaml` files can be found in the project repository solutions/helm/nginx-deployment

3.12.5 Common Errors & How to Fix Them

Error Validating Data

Pay attention when copying/pasting to build manifests! The letter `a` on the first line of a created manifest is usually lost after the file is saved. This error turns `apiVersion` into `piVersion` and raises the following error:

```
Error: error: error validating "argocd-server-nodeport.yaml": error validating data:
apiVersion not set; if you choose to ignore these errors, turn validation off with
--validate=false
```

To fix **Error Validating Data**, open the yaml file you working on and correct the typo by turning `piVersion` back to `apiVersion`.