**Ex. 3 – Kubernetes**

1. We will use the Kubernetes sandbox now at <https://learning.oreilly.com/scenarios/kubernetes-containers-go/9781492090144/?utm_medium=email&utm_source=platform+b2b&utm_campaign=engagement&utm_content=whats+new+interactive+20210726>
2. Click on Start Scenario.
3. **Step 1:** The virtual machine starts a Kubernetes cluster in Step 1. Read through the first screen. Click on the first image with the kubectl command. Is this like a Docker command in some ways?
4. The exercise mentions the Helm package manager. Do you know of other package managers? Think of **apt-get** manager for Linux that can install packages and perform other tasks in Linux Debian systems. The **pom** file in Maven can be thought of as a package manager.
5. Do not be concerned with Kubernetes dashboard now. We will use the Command Line Interface (CLI). Click on Continue to move to Step 2.
6. **Step 2:** Click on the git command to clone a repository. Do you remember using this git command before? Type git clone –help if you need help with this command.
7. Read the **README.md** file. This is standard practice to include this file in a git repository. What are some ways we can view this file? (Remember Linux! Can use cat, ccat, any editor.) So this is definitely fun; we are reviewing git, Linux, Docker later, and some programming although we will see the application is in a language called Go rather than Java. But don’t worry, this will not hold us back.
8. Then you see && to change directories also.
9. Type the tree command. Do you see the **Dockerfile**? You can look at it now but we will check it for sure in the next step. Change the PS1 primary

command prompt to PS1='$(pwd)>' and see what you get for the prompt now. Remember this if you completed the previous exercise.

1. Do you recall the env command in Linux? What does it output? Yes, outputs environmental variables for the system. The go env command just displays the environmental variables for Go. Click continue to move to Step 3.
2. **Step 3:** Click on the black code links with ls and ccat. View the main.go file. Notice main.go is a small file. View the file. If you do not know the language Go that is fine. You are not responsible for it. But do you see similarities between Java and Go and how we would write this simple file in Java?
3. Find the difference between cat and ccat commands by typing ccat –help.
4. Click on go run main.go link to see program output as expected. Please read the section on containers carefully for this step. This is the main part of the exercise.
5. Check the **Dockerfile**. Do you remember the commands in blue in the **Dockerfile**? If not please review. What is the weird looking type in yellow after the FROM command - golang:1.14.2-buster? (an image). *The Docker build command to create an image is different than the go build command to create a binary.*
6. Go through the commands and run them here. You see the image with name 'hello'. Notice the problems with the container in the exercise as explained in this exercise. Please read carefully. Is there a similar problem with the container we created with the Java app in a previous exercise?
7. After the build command run command docker images | grep golang && docker images | grep hello. Do you see two lines with these images and information about them?
8. Click on Continue to move to Step 4.
9. **Step 4:** Click on the first command to see the files in the basic directory and run the go files by clicking on the command on the left frame as before. Then check the code in main.go now for the weather application. Notice some of the program structure of Go, but again this is not critical to this exercise.
10. Read the section carefully on the Container. The key is three FROM statements now in the **Dockerfile**. See the **Dockerfile**. Notice only the third stage runs and the first two are thrown away. Do not be concerned about REST, certificates, or some other details here, although you have hit lightly on these concepts. We will discuss **REST** more in the next step. You may have seen the concept of certificates when you go to some sites that are protected and you see it may not be a trusted certificate.
11. Build the application with the images and see the reduction in size from the hello to basic image - basic image is about 1% of the size of the hello image.
12. See the very good advice and best practices in the paragraph with the blue line on the left side and the next paragraph in bold.
13. Run the basic application as suggested. Then press Continue to move to Step 5.
14. **Step 5:** Let's look at the files with the 'go' file extension files to relate what we know and get some information. There is a short description of these files in the section Microservice App Container. The exercise mentions an in memory model for data. This means we are not really using a database as we did to persist or store data but just a Go data structure. We talked about various data structures in Java. We can actually use in an in memory database like Hypersonic and store data.
15. a. main.go Note the GET, POST, PATCH, DELETE http methods. See the lines such as;

ideaGroup.GET("/", GetAll)

ideaGroup.GET("/:id", Get)

router.Run(":8080")

The first parameter in these http methods is just the path in our URL

or address we type in a browser window. The "/" is the route of our web app. The "/:id" depicts an address that ends in /22 for example.

The second parameter for the same methods is the Go function we will see in file **routes.go**. These are really CRUD methods or functions.

Do you remember those from SQL earlier? This means Create, Read, Update, Delete - same as before!

The Run method shows the application will run on port 8080.

b. **routes. go** These are the CRUD functions as noted above. Notice the functions can handle any error conditions such as blank text.

See the import "net/http"? We don't have to know Go to figure out this is like an import in Java.

So then we can see http in the import used as in http.StatusOK. This is just the 200 response code we can check when everything goes correctly

and according to what we want when we want information from a site. The response code http.StatusNotFound (code 404) means somewhere in our application we have an address of a Web page we coded or some link that cannot be found from our application. Unfortunately all of us

have received or will receive a response code like this and it means we need to debug our code. This file, routes.go, can be thought of

as a controller in a Web application, or a file/class that handles incoming requests from clients or users and sends back responses.

This is what we see and expect from a web application.

1. c. **model.go** This file depicts the model or sometimes this is called a service layer type class. These methods are called by the CRUD methods

in **routes.go** to add behavior and additional functionality to the application CRUD and other methods.

1. What is **curl**? Notice **curl** is used here to test addresses on a server without actually using a browser. You may eventually use tools

like **Postman** for this also. When you click on go run \*.go in this step you see the last line of the terminal says that the server is listening and

serving HTTP on :8080. We specified port 8080 in main.go. This terminal window that hangs is called the server console.

Open a new terminal by clicking on the + button at the top of the terminal. This will be terminal 2. Use this terminal to enter commands.

Click on the first curl command in Step 5.

1. You may recall or you can check in ideaGroup := router.Group("ideas/v1") is part of the Web address URL - same as curl command:

http://localhost:8080/ideas/v1. We trace the GET method (type of http method when we click a link or type an address in a browser and enter for example). We then see ideaGroup.GET("/", GetAll) so control passes to GetAll function in routes.go. This function is

func GetAll(context \*gin.Context) {

context.JSON(http.StatusOK, ideas)

}

1. Then we hit the 'ideas' function in model.go and we see the message expected in the server console window. Look at the server console terminal in terminal 1. Note the green background column of 200 (Status OK) for the curl commands. Also notice the GET and POST methods as specified in the application. You also see the part of the address above as "/ideas/v1/" in the last column. Click on all the curl commands on the left after clicking to make sure you are in terminal 2 and see if you can predict what http method will run. Notice the server console in terminal 1 shows a different background color for each http method. How can you predict from a curl command that it is POST method? How do you know from curl that is GET command? How about DELETE and TRACE here? You can always get more information on curl by typing curl --help in the terminal. Note the --data option and this should help you with one of the questions above (will be POST operation). GET operation is default so if we don't use it in CURL the operation is GET. The other operations will be specified in curl.
2. Stop the server in terminal 1 by pressing **^C** (Control + c).
3. Now build the restafarian image. Check to see it is present. Run it with docker run. Proceed to run other commands in terminal 2 since terminal 1 is the server console. You can see the same results in the server console but the background for the http methods is not a pretty color now! (not important of course.) Do you notice when you enter a new topic and idea as depicted in the curl command this is a POST (like sending information to server) rather than GET (retrieving information)?
4. Stop the server in terminal 1 by pressing **^C** (Control + c).
5. Click on Continue to go to Step 6.
6. **Step 6:** This VM exercise is a Katacoda application and this instance is running a Kubernetes cluster as described. Run the commands in this step. Do you understand the Docker commands build and push in the Push to Registry section? Note the run command for **kubectl**. Then you see the status 'Running'. What is this similar to what you may have seen before? (any service running after you start a service in Linux). Run the commands in section Call REST API. Note you get similar results from application from before the restafarian service was mentioned.
7. So now you are finished with the exercise. There was a lot of information here and you are not expected to understand all of it. We went into more detail than some of the description but did not cover Kubernetes fully in certain areas that the exercises showed for some areas. But we did

not want to or could not cover everything. You should have gained confidence in containerization, applying knowledge of Git, Linux, CRUD, some basic Web programming and concepts, relating a language like Go to Java, packaging a microservice. You know more than you think and

can apply much of your knowledge logically to different concepts that have similar ideas to what you know now. You will be doing things like investigating concepts you have not seen before but can relate them to previous knowledge. The introduction of topics happens constantly as nobody knows close to everything. Technology is a massive subject!