Melanie Wong

CSE 151/L

Final Project

Lab - Golang REST API with Docker

Description

This is a lab that explains how to write a simple REST API in Golang and host using Docker.

Objectives

- Introduction to REST API
- Developing simple endpoints in Golang
- Building and running the application using Docker

Prerequisites

- Install Golang/GO
  - https://golang.org/doc/install
- Install Docker Desktop for Windows and Mac
  - https://www.docker.com/products/docker-desktop

Getting started

Create a new folder called go-rest-api and a main.go file

Commands

- mkdir go-rest-api
- cd go-rest-api
- touch main.go

Test out the app using the main function defined below (copy/paste into main.go)

(Extra info: “fmt” is a golang package that implements formatted I/O)

```go
package main

import "fmt"

func main() {
   fmt.Println("Hello World!")
}
```

Compile and run the code with one line (should print “Hello World!”)

- go run main.go

Introduction to REST API

A RESTful/REST API is an application programming interface (API) used to interact with data in web services development. It uses HTTP requests:

- GET to retrieve data
- POST to create new data
- PUT to update existing data
- DELETE to remove data

Developing simple endpoints in Golang

Set up the HTTP server using Gorilla Mux

(Extra info: Gorilla Mux is a package that implements a request router and dispatcher for matching incoming requests to their respective handler)

Installation command

- go get –u github.com/gorilla/mux

Import packages

(“//” is used to write comments in code)

```
// all go executable need a main package
package main

import (
   "log"      // for logging any errors
   "net/http" // for writing rest api

   "github.com/gorilla/mux"
)
```

Use Gorilla Mux to create endpoint at “/” for each method (default route)

```
func main() {
   r := mux.NewRouter()
   r.HandleFunc("/", get).Methods("GET")
   r.HandleFunc("/", post).Methods("POST")
```

```go
    r.HandleFunc("/", put).Methods("PUT")
    r.HandleFunc("/", delete).Methods("DELETE")
    r.HandleFunc("/", notFound)

    log.Fatal(http.ListenAndServe(":8085", r))
}
```

Create functions to handle each HTTP request method

GET

```go
func get(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "Successfully retrieved - Welcome to the home page!"}`))
}
```

POST

```go
func post(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte(`{"message": "Successfully created!"}`))
}
```

PUT

```go
func put(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "Successfully updated!"}`))
}
```

DELETE

```go
func delete(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "Successfully deleted!"}`))
}
```

Each function accepts a request of corresponding method (GET, POST, PUT, DELETE) as input, and returns a response in JSON format with a corresponding status code (more details below).

If the user requests an endpoint that doesn't exist, return a response indicating that the page is not found.

```
func notFound(w http.ResponseWriter, r *http.Request) {
   w.Header().Set("Content-Type", "application/json")
   w.WriteHeader(http.StatusNotFound)
   w.Write([]byte(`{"message": "not found :("}`))
}
```

Status Codes

Indicates successful completion of requests, or any errors upon receiving and processing requests. The most common ones – also the ones used in this lab – are listed below.

- 200 OK
  - Response from GET, PUT and DELETE
- 201 Created
  - Response from POST
- 404 Not Found
  - Response to broken or "dead" link – server cannot find requested resource

Extra Notes

- The request handling functions above do not actually manipulate any data because data is not used in this lab. However, they make up a good starting point for integrating with a database (e.g. Firebase) for a practical backend storage service!
- The complete code in main.go is provided at the end of this lab.

Building and running the application using Docker

A Dockerfile is used to specify instructions for building and running the API in a container. "Containerize" the application using the commands below.

("#" is used to write comments in a Dockerfile)

```
# Getting latest golang docker image from hub
FROM golang:latest
RUN mkdir /app
ADD . /app/
WORKDIR /app
# build application
RUN go build -o main .
CMD ["/app/main"]
```

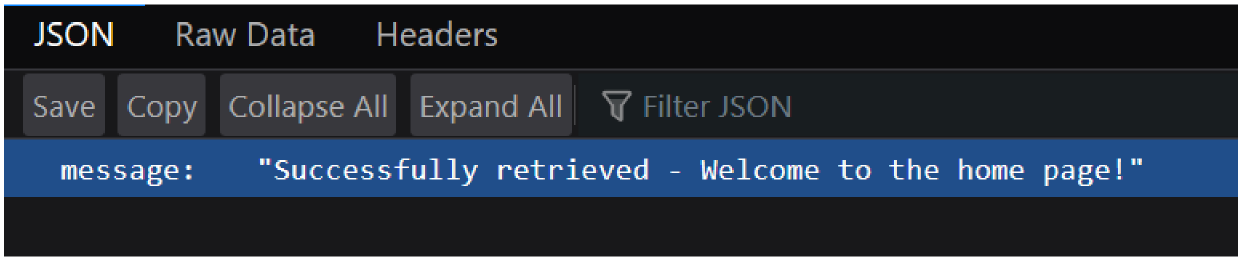
Build an image of the application and run a container

(container = an instance of an image, e.g. can create multiple containers of a single image)

```
# build image
docker build -t go-rest-api .

# run application
docker run -it -p 8085:8085 --name="go-rest-api" go-rest-api
```

Go to localhost:8085 in your favorite web browser (should see response message from the GET request)



```
message:    "Successfully retrieved - Welcome to the home page!"
```

Test the endpoint for all implemented HTTP methods using curl in the command line

```
curl --request GET --write-out "\n%{http_code}\n" http://localhost:8085/
```

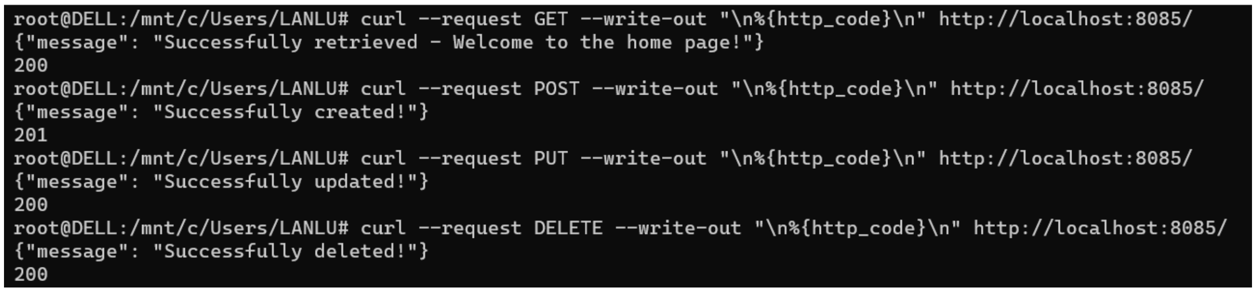
```
curl --request POST --write-out "\n%{http_code}\n" http://localhost:8085/
```

```
curl --request PUT --write-out "\n%{http_code}\n" http://localhost:8085/
```

```
curl --request DELETE --write-out "\n%{http_code}\n" http://localhost:8085/
```

Expected responses for each request

(--write-out “\n%{http_code}\n” simply prints the response status code – useful for debugging)



```
root@DELL:/mnt/c/Users/LANLU# curl --request GET --write-out "\n%{http_code}\n" http://localhost:8085/
{"message": "Successfully retrieved - Welcome to the home page!"}
200
root@DELL:/mnt/c/Users/LANLU# curl --request POST --write-out "\n%{http_code}\n" http://localhost:8085/
{"message": "Successfully created!"}
201
root@DELL:/mnt/c/Users/LANLU# curl --request PUT --write-out "\n%{http_code}\n" http://localhost:8085/
{"message": "Successfully updated!"}
200
root@DELL:/mnt/c/Users/LANLU# curl --request DELETE --write-out "\n%{http_code}\n" http://localhost:8085/
{"message": "Successfully deleted!"}
200
```

Success! ☺

Conclusion

I really enjoyed developing this project because as many people claim, teaching a topic is one of the best ways to learn and thoroughly understand it. Although I have some previous experience in developing REST APIs (in Java and Python) and using Docker, this project gave me motivation to apply my prior knowledge while learning a new programming language.

I learned that including additional background info—not only for the main topic, but also for each specific step throughout the lab—could be extremely helpful for fully understanding the project. Rather than simply including only the commands and/or the source code, I incorporated comments in the code and extra information throughout this lab for any learners interested in the "why" at each step in addition to the "how". Providing extra details behind each tool/technology used in this lab is also intended for making the exercise helpful and easy to understand for learners of any experience level.

References

Similar Golang REST API with docker tutorials

- https://medium.com/the-andela-way/build-a-restful-json-api-with-golang-85a83420c9da
- https://medium.com/@yusufkaratoprak/golang-rest-api-by-docker-7ad7d9b05e22

Explanation of REST API

- https://searchapparchitecture.techtarget.com/definition/RESTful-API

Resources

The complete code for main.go is provided below. The complete directory with the Dockerfile and simple unit testing code can be found on GitHub using the link below.

https://github.com/mwong775/go-rest-api-docker

```go
// all go executable need a main package
package main

import (
    "log"      // for logging any errors
    "net/http" // for writing rest api

    "github.com/gorilla/mux"
)

func get(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "Successfully retrieved - Welcome to the home pag
e!"}`))
}

func post(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte(`{"message": "Successfully created!"}`))
}

func put(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "Successfully updated!"}`))
}

func delete(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    w.Write([]byte(`{"message": "Successfully deleted!"}`))
}

func notFound(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusNotFound)
    w.Write([]byte(`{"message": "not found :("}`))
}

func main() {
    r := mux.NewRouter()
    r.HandleFunc("/", get).Methods("GET")
    r.HandleFunc("/", post).Methods("POST")
    r.HandleFunc("/", put).Methods("PUT")
    r.HandleFunc("/", delete).Methods("DELETE")
    r.HandleFunc("/", notFound)

    log.Fatal(http.ListenAndServe(":8085", r))
}
```