**ECGR 4090/5090 Cloud Native Application Architecture**
**Lab 6: REST API with Swagger/OpenAPI**

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, use testing tools, and many other use cases. (From: swagger.io)

The OpenAPI Specification (originally called Swagger specification) was donated to the Linux Foundation under the OpenAPI Initiative in 2015. Swagger tools are the most widely used implementation of OpenAPI specification. In this lab, we will design a REST API using Go Swagger tools. Along the way, we will also organize our code into multiple directories such as build, internal etc.

Create a *rest-api-swagger* directory under your local repo directory
Create the following directories under *rest-api-swagger* directory.
*internal, bin, pkg*
Start with implementing a simple Go web server. Cut and paste the following code under the */internal* directory.

```
package main

import (
    "fmt"
    "html"
    "log"
    "net/http"
)

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, "Hello %q", html.EscapeString(r.URL.Path))
    })
    log.Println("Listening on localhost: 8080")
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

// Test the web server. From another terminal
*$ curl localhost:8080*

As a step towards automating building the code we will use the *make* utility. Create the following Makefile under *rest-api-swagger* directory. (**Note**: Makefile uses tabs for indentation)

```
#------------------------
# Build artifacts
#------------------------
GO   := go
build:
    $(GO) build -o bin/http-go-server internal/main.go
:
```

// build the webserver buildserver
*$ make*

//run the webserver
*$ bin/http-go-server*

From another terminal
*$ curl localhost:8080* // Should get the Hello "/" message

Install go-swagger from source (from https://goswagger.io/install.html)
```
$ dir=$(mktemp -d)
$ git clone https://github.com/go-swagger/go-swagger "$dir"
$ cd "$dir"
$ go install ./cmd/swagger
```

Test
*$ swagger version*

We describe the REST API as a Swagger YAML file. YAML (YAML Ain't Markup Language) is a human readable format for specifying configuration files. YAML is a superset of JSON. Unlike JSON, YAML uses indentation instead of curly braces. It should be noted that only spaces are used for indentation (no tabs).

For a quick introduction to YAML see
https://www.cloudbees.com/blog/yaml-tutorial-everything-you-need-get-started/

Back to our Swagger YAML file - cut and paste the following code under *pkg/swagger* directory in a file *swagger.yml* (create *swagger* directory)

```
---
consumes:
- application/json
info:
  description: HTTP server in Go with Swagger endpoints definition
  title: http-go-server
  version: 0.1.0
produces:
- application/json
schemes:
- http
swagger: "2.0"

paths:
  /healthz:
    get:
      operationId: checkHealth
      produces:
      - text/plain
      responses:
        '200':
          description: OK message
          schema:
            type: string
            enum:
            - OK

  /hello/{user}:
      get:
        description: Returns a greeting to the user!
        parameters:
          - name: user
            in: path
            type: string
            required: true
            description: The name of the user to greet.
        responses:
          200:
            description: Returns the greeting.
            schema:
              type: string
          400:
            description: Invalid characters in "user" were provided.
```

From Swagger docs 2.0 -
Every Swagger specification starts with the Swagger version (note: 3.0 is the latest version). A Swagger version defines the overall structure of an API specification – what you can document and how you document it. Then, you need to specify the API info – title, description (optional), version (API version, not file revision or Swagger version). The base URL for all API calls is defined using schemes, host and basePath. All API paths are relative to the base URL. The consumes and produces sections define the MIME types supported by the API. The paths section defines individual endpoints (paths) in your API, and the HTTP methods (operations) supported by these endpoints. Operations can have parameters that can be passed via URL path *(/users/{userId}*), query string *(/users?role=admin)*, headers (X-CustomHeader: Value) and request body. For each operation, you can define possible status codes, such as 200 OK or 404 Not Found, and schema of the response body. Schemas can be defined inline or referenced from an external definition via *$ref*.
For more see -
https://swagger.io/docs/specification/2-0/basic-structure/

Let's validate the swagger definition

*$ swagger validate pkg/swagger/swagger.yml*

Let's automate this with *make*. Copy and paste the following code in the *Makefile* (Remember to use tabs for indentation).

```
#----------------------
# Target: swagger.validate
#----------------------
.PHONY: swagger.validate

swagger.validate:
    swagger validate pkg/swagger/swagger.yml
```

Point your browser to htttps://editor.swagger.io

Cut and paste the swagger file into the editor on the left panel. On the right panel you will see the end points displayed in a clickable easy to read format. Fix errors (mostly due to indentation) if any.

We now use *go-swagger* to generate Go code. We will use *go generate* and *make* to automate this.
Under *pkg/swagger* create a *gen.go* file, and copy and paste the following code -

```
package swagger
//go:generate rm -rf server
//go:generate mkdir -p server
//go:generate swagger generate server --quiet --target server --name hello-api --spec swagger.yml --exclude-main
```

Generate runs commands described by directives within existing files. Those commands can run any process but the intent is to create or update Go source files.

Go generate scans the file for directives, which are lines of the form,

```
//go:generate command argument...
```

Go generate is never run automatically by *go build*, *go get*, *go test*, and so on. It must be run explicitly.

For more see - https://golang.org/pkg/cmd/go/internal/generate/

Copy and paste the following to the top of Makefile (after the Go := go statement)

```
pkgs  = $(shell GOFLAGS=-mod=vendor $(GO) list ./... | grep -vE -e /vendor/ -e /pkg/swagger/)
pkgDirs = $(shell GOFLAGS=-mod=vendor $(GO) list -f {{.Dir}} ./... | grep -vE -e /vendor/ -e /pkg/swagger/)
DIR_OUT:=/tmp
```

Copy and paste the following to the Makefile. (Remember to use tabs for indentation)

```
#-------------------------
# Code generation
#-------------------------
.PHONY: generate

## Generate go code
generate:
    @echo "==> generating go code"
    GOFLAGS=-mod=vendor $(GO) generate $(pkgs)
```

Run make to use Swagger to generate Go code

*make generate*

Examine pkg/swagger to see the new files generated by go-swagger

Let's edit the internal/main.go to use the REST APIs generated by Swagger. Copy and paste the following code.
**Note:** Change the import paths to that corresponding to your system

```
package main

import (
    "log"

    "github.com/go-openapi/loads"
    "github.com/go-openapi/runtime/middleware"

"gitlab.com/arunravindran/cloudnativecourse/lab6-openapi/restapi-swagger/pkg/swagger/server/restapi"

"gitlab.com/arunravindran/cloudnativecourse/lab6-openapi/restapi-swagger/pkg/swagger/server/restapi/operations"
)

func main() {

    // Initialize Swagger
    swaggerSpec, err := loads.Analyzed(restapi.SwaggerJSON, "")
    if err != nil {
        log.Fatalln(err)
    }

    api := operations.NewHelloAPIAPI(swaggerSpec)
    server := restapi.NewServer(api)

    defer func() {
        if err := server.Shutdown(); err != nil {
            // error handle
            log.Fatalln(err)
        }
    }()

    server.Port = 8080
```

```
    // Start server which listening
    if err := server.Serve(); err != nil {
        log.Fatalln(err)
    }
}
```

Build and run the server -

*$ make build*

*$ bin/http-go-server*

Test from another terminal

*$ curl localhost:8080*

$ *curl localhost:8080/hello*

$ *curl localhost:8080/hello/batman*

Since none of the API endpoints are yet implemented, you should get 404 errors.This is an example of red-green testing, where tests starts initially fail (red), and subsequently you make the tests pass (green)

Copy and paste the implementations of the hello and healthz handlers outside the main()

```
//Health route returns OK
func Health(operations.CheckHealthParams) middleware.Responder {
    return operations.NewCheckHealthOK().WithPayload("OK\n")
}

//GetHelloUser returns Hello + your name
func GetHelloUser(user operations.GetHelloUserParams) middleware.Responder {
    return operations.NewGetHelloUserOK().WithPayload("Hello " + user.User + "!")
}
```

Add the following lines in *main()* to call the handlers

```
// Implement the CheckHealth handler
    api.CheckHealthHandler = operations.CheckHealthHandlerFunc(Health)

// Implement the GetHelloUser handler
    api.GetHelloUserHandler = operations.GetHelloUserHandlerFunc(GetHelloUser)
```

Build and run the server -

*$ make build*

*$ bin/http-go-server*

In a second terminal
*$ curl localhost:8080/hello/batman*

*$ curl localhost:8080/healthz*

This should work!
To recap, we defined the API in a YAML file following OpenAPI 2.0 specifications. The client and server can then use this documentation to implement/use the API. On the server side, we used go-swagger to generate the server Go code including the stubs for the API handlers. We then edited the server main.go file to implement the API handlers.
We tested our implementation using curl.

Be sure to update your Gitllab/Github repository if you haven't already done so.

This lab is adapted from the Dzone blog "How to Write an HTTP REST API Server in Go in Minutes" by Aurelie Vache

**To do -**
Skim through the implementation of a "todo-list" application described in this tutorial from *goswagger.io.* The implementation has other REST operations such as POST and DELETE.
https://goswagger.io/tutorial/todo-list.html