Exercise 2: Microservice in Go

Table of Contents

1. After Tutorial

After finishing the tutorial, my understanding of the project is the following:

1.1. model.go

The `model.go` contains the definition of our `Product` model and provides basic `CRUD` operations for storing, updating and deleting products from an `SQL` database.

Listing 1. model.go

```go
// model.go

package main

import (
    "database/sql"
)



type product struct {
    ID    int     `json:"id"`
    Name  string  `json:"name"`
    Price float64 `json:"price"`
}

func (p *product) getProduct(db *sql.DB) error {
    return db.QueryRow("SELECT name, price FROM products WHERE id=$1",
        p.ID).Scan(&p.Name, &p.Price)
}

func (p *product) updateProduct(db *sql.DB) error {
    _, err :=
        db.Exec("UPDATE products SET name=$1, price=$2 WHERE id=$3",
            p.Name, p.Price, p.ID)
```

```
    return err
}

func (p *product) deleteProduct(db *sql.DB) error {
    _, err := db.Exec("DELETE FROM products WHERE id=$1", p.ID)

    return err
}

func (p *product) createProduct(db *sql.DB) error {
    err := db.QueryRow(
        "INSERT INTO products(name, price) VALUES($1, $2) RETURNING id",
        p.Name, p.Price).Scan(&p.ID)

    if err != nil {
        return err
    }

    return nil
}

func getProducts(db *sql.DB, start, count int) ([]product, error) {
    rows, err := db.Query(
        "SELECT id, name,  price FROM products LIMIT $1 OFFSET $2",
        count, start)

    if err != nil {
        return nil, err
    }

    defer rows.Close()

    products := []product{}

    for rows.Next() {
        var p product
        if err := rows.Scan(&p.ID, &p.Name, &p.Price); err != nil {
            return nil, err
        }
        products = append(products, p)
    }

    return products, nil
}
```

1.2. app.go

The `app.go` file contains the actual logic of the microservice. The `App` struct uses a `mux.Router` from `gorilla/mux` to route traffic to the endpoints defined by `initializeRoutes()`.

The defined routes use the `CRUD` operations defined on the `Prodcut` with the `postgres` database connection established in `Initialize()` and returns the result as Json according to `REST` standards.

Listing 2. app.go

```
// app.go

package main
```

```
import (
    "database/sql"
    "fmt"
    "github.com/golang-jwt/jwt/v5"
    "log"
    "time"

    "encoding/json"
    "net/http"
    "strconv"

    _ "github.com/golang-jwt/jwt/v5"
    "github.com/gorilla/mux"
    _ "github.com/lib/pq"
)

type App struct {
    Router    *mux.Router
    DB        *sql.DB
    JwtSecret string
}

func (a *App) Initialize(user, password, dbname, jwtSecret string) {
    connectionString :=
        fmt.Sprintf("user=%s password=%s dbname=%s sslmode=disable", user, password, dbname)

    var err error
    a.DB, err = sql.Open("postgres", connectionString)
    if err != nil {
        log.Fatal(err)
    }

    a.Router = mux.NewRouter()
    a.JwtSecret = jwtSecret
    a.initializeRoutes()
}

func (a *App) Run(addr string) {
    log.Fatal(http.ListenAndServe(":8010", a.Router))
}

func (a *App) getProduct(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, err := strconv.Atoi(vars["id"])
    if err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid product ID")
        return
    }

    p := product{ID: id}
    if err := p.getProduct(a.DB); err != nil {
        switch err {
        case sql.ErrNoRows:
            respondWithError(w, http.StatusNotFound, "Product not found")
        default:
            respondWithError(w, http.StatusInternalServerError, err.Error())
        }
        return
    }
```

```
    respondWithJSON(w, http.StatusOK, p)
}

func respondWithError(w http.ResponseWriter, code int, message string) {
    respondWithJSON(w, code, map[string]string{"error": message})
}

func respondWithJSON(w http.ResponseWriter, code int, payload interface{}) {
    response, _ := json.Marshal(payload)

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(code)
    w.Write(response)
}

func (a *App) getProducts(w http.ResponseWriter, r *http.Request) {
    count, _ := strconv.Atoi(r.FormValue("count"))
    start, _ := strconv.Atoi(r.FormValue("start"))

    if count > 10 || count < 1 {
        count = 10
    }
    if start < 0 {
        start = 0
    }

    products, err := getProducts(a.DB, start, count)
    if err != nil {
        respondWithError(w, http.StatusInternalServerError, err.Error())
        return
    }

    respondWithJSON(w, http.StatusOK, products)
}

func (a *App) createProduct(w http.ResponseWriter, r *http.Request) {
    var p product
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(&p); err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid request payload")
        return
    }
    defer r.Body.Close()

    if err := p.createProduct(a.DB); err != nil {
        respondWithError(w, http.StatusInternalServerError, err.Error())
        return
    }

    respondWithJSON(w, http.StatusCreated, p)
}

func (a *App) updateProduct(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, err := strconv.Atoi(vars["id"])
    if err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid product ID")
        return
    }
```

```
    var p product
    decoder := json.NewDecoder(r.Body)
    if err := decoder.Decode(&p); err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid resquest payload")
        return
    }
    defer r.Body.Close()
    p.ID = id

    if err := p.updateProduct(a.DB); err != nil {
        respondWithError(w, http.StatusInternalServerError, err.Error())
        return
    }

    respondWithJSON(w, http.StatusOK, p)
}

func (a *App) deleteProduct(w http.ResponseWriter, r *http.Request) {
    vars := mux.Vars(r)
    id, err := strconv.Atoi(vars["id"])
    if err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid Product ID")
        return
    }

    p := product{ID: id}
    if err := p.deleteProduct(a.DB); err != nil {
        respondWithError(w, http.StatusInternalServerError, err.Error())
        return
    }

    respondWithJSON(w, http.StatusOK, map[string]string{"result": "success"})
}

// Healthcheck endpoint that complies with Java Microprofile Health specification
func (a *App) healthCheck(w http.ResponseWriter, r *http.Request) {
    dbErr := a.DB.Ping()
    if dbErr != nil {
        health := map[string]interface{}{
            "status": "DOWN",
            "checks": []map[string]interface{}{
                {
                    "name":   "database",
                    "status": "DOWN",
                },
            },
        }
        respondWithJSON(w, http.StatusServiceUnavailable, health)
        return
    }

    health := map[string]interface{}{
        "status": "UP",
        "checks": []map[string]interface{}{
            {
                "name":   "database",
                "status": "UP",
            },
        },
```

```
    }

    respondWithJSON(w, http.StatusOK, health)
}

// Generate a JWT token
func (a *App) generateJWT(username, role string) (string, error) {
    expirationTime := time.Now().Add(5 * time.Minute) // token expires in 5 minutes

    claims := jwt.MapClaims{
        "exp":      expirationTime.Unix(),
        "iss":      "go-mux",
        "sub":      role,
        "username": username,
    }

    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

    tokenString, err := token.SignedString([]byte(a.JwtSecret))
    if err != nil {
        return "", err
    }

    return tokenString, nil
}

// Middleware to check if the request has a valid JWT token
func (a *App) jwtAuthentication(requiredRoles []string, next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        tokenString := r.Header.Get("Authorization")
        if tokenString == "" {
            respondWithError(w, http.StatusUnauthorized, "No token provided")
            return
        }

        claims := jwt.MapClaims{}
        token, err := jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (interface{}, error) {
            if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
                return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
            }
            return []byte(a.JwtSecret), nil
        })

        if err != nil || !token.Valid {
            respondWithError(w, http.StatusUnauthorized, "Invalid token")
            return
        }

        if !contains(requiredRoles, claims["sub"].(string)) {
            respondWithError(w, http.StatusUnauthorized, "Insufficient permissions")
            return
        }

        next(w, r)
    }
}

// helper function to check if a string is in a slice
func contains(arr []string, str string) bool {
    for _, item := range arr {
```

```
        if item == str {
            return true
        }
    }
    return false
}

func (a *App) generateToken(w http.ResponseWriter, r *http.Request) {
    var creds struct {
        Username string `json:"username"`
        Password string `json:"password"`
    }

    err := json.NewDecoder(r.Body).Decode(&creds)
    if err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid request payload")
        return
    }

    // Replace this with proper authentication logic
    // hardcoded users => could be replaced with users from a database
    if creds.Username == "admin" && creds.Password == "admin_password" {
        token, err := a.generateJWT("admin", "admin")
        if err != nil {
            respondWithError(w, http.StatusInternalServerError, err.Error())
            return
        }
        respondWithJSON(w, http.StatusOK, map[string]string{"token": token})
    } else if creds.Username == "moderator" && creds.Password == "moderator_password" {
        token, err := a.generateJWT("moderator", "moderator")
        if err != nil {
            respondWithError(w, http.StatusInternalServerError, err.Error())
            return
        }
        respondWithJSON(w, http.StatusOK, map[string]string{"token": token})
    } else {
        respondWithError(w, http.StatusUnauthorized, "Invalid credentials")
    }
}

func (a *App) initializeRoutes() {
    a.Router.HandleFunc("/products", a.getProducts).Methods("GET")
    a.Router.HandleFunc("/product/{id:[0-9]+}", a.getProduct).Methods("GET")
    // use jwtAuthentication middleware to protect the following endpoints
    a.Router.HandleFunc("/product", a.jwtAuthentication([]string{"admin", "moderator"}, a.createProduct)).Methods("POST")
    a.Router.HandleFunc("/product/{id:[0-9]+}", a.jwtAuthentication([]string{"admin", "moderator"}, a.updateProduct)).Methods("PUT")
    a.Router.HandleFunc("/product/{id:[0-9]+}", a.jwtAuthentication([]string{"admin"}, a.deleteProduct)).Methods("DELETE")
    // additional endpoints
    a.Router.HandleFunc("/health", a.healthCheck).Methods("GET")
    a.Router.HandleFunc("/token", a.generateToken).Methods("POST")
}
```

1.3. main.go

The `main.go` file acts as the entry point for our go microservice. It reads the required parameters for the `App` from the environment variables and initializes the microservice with them and makes sure the microservice is available on port `8010`.

Listing 3. main.go

```
// main_test.go

package main

import (
    "log"
    "os"
    "testing"

    "bytes"
    "encoding/json"
    "github.com/stretchr/testify/assert"
    "net/http"
    "net/http/httptest"
    "strconv"
)

var a App

var adminToken = "your_admin_jwt_token"
var moderatorToken = "your_moderator_jwt_token"

func TestMain(m *testing.M) {
    a.Initialize(
        os.Getenv("APP_DB_USERNAME"),
        os.Getenv("APP_DB_PASSWORD"),
        os.Getenv("APP_DB_NAME"),
        os.Getenv("APP_JWT_SECRET"))

    ensureTableExists()

    code := m.Run()
    clearTable()
    os.Exit(code)
}

// these need to be executed before all other tests
func TestGenerateAdminTokenSuccess(t *testing.T) {
    payload := []byte(`{"username":"admin","password":"admin_password"}`)

    req, _ := http.NewRequest("POST", "/token", bytes.NewBuffer(payload))
    req.Header.Set("Content-Type", "application/json")
    response := executeRequest(req)

    assert.Equal(t, http.StatusOK, response.Code)

    var responseBody map[string]string
    err := json.Unmarshal(response.Body.Bytes(), &responseBody)
    assert.Nil(t, err)
    assert.NotEmpty(t, responseBody["token"])

    adminToken = responseBody["token"] // Update the global variable with the retrieved token
}

func TestGenerateModeratorTokenSuccess(t *testing.T) {
    payload := []byte(`{"username":"moderator","password":"moderator_password"}`)

    req, _ := http.NewRequest("POST", "/token", bytes.NewBuffer(payload))
```

```
    req.Header.Set("Content-Type", "application/json")
    response := executeRequest(req)

    assert.Equal(t, http.StatusOK, response.Code)

    var responseBody map[string]string
    err := json.Unmarshal(response.Body.Bytes(), &responseBody)
    assert.Nil(t, err)
    assert.NotEmpty(t, responseBody["token"])

    moderatorToken = responseBody["token"] // Update the global variable with the retrieved token
}

func TestGenerateTokenFail(t *testing.T) {
    payload := []byte(`{"username":"admin","password":"wrong_password"}`)

    req, _ := http.NewRequest("POST", "/token", bytes.NewBuffer(payload))
    req.Header.Set("Content-Type", "application/json")
    response := executeRequest(req)

    assert.Equal(t, http.StatusUnauthorized, response.Code)
}

func ensureTableExists() {
    if _, err := a.DB.Exec(tableCreationQuery); err != nil {
        log.Fatal(err)
    }
}

func clearTable() {
    a.DB.Exec("DELETE FROM products")
    a.DB.Exec("ALTER SEQUENCE products_id_seq RESTART WITH 1")
}

const tableCreationQuery = `CREATE TABLE IF NOT EXISTS products
(
    id SERIAL,
    name TEXT NOT NULL,
    price NUMERIC(10,2) NOT NULL DEFAULT 0.00,
    CONSTRAINT products_pkey PRIMARY KEY (id)
)`

func TestEmptyTable(t *testing.T) {
    clearTable()

    req, _ := http.NewRequest("GET", "/products", nil)
    response := executeRequest(req)

    checkResponseCode(t, http.StatusOK, response.Code)

    if body := response.Body.String(); body != "[]" {
        t.Errorf("Expected an empty array. Got %s", body)
    }
}

func executeRequest(req *http.Request) *httptest.ResponseRecorder {
    rr := httptest.NewRecorder()
    a.Router.ServeHTTP(rr, req)

    return rr
```

```
}

func checkResponseCode(t *testing.T, expected, actual int) {
    if expected != actual {
        t.Errorf("Expected response code %d. Got %d\n", expected, actual)
    }
}

func TestGetNonExistentProduct(t *testing.T) {
    clearTable()

    req, _ := http.NewRequest("GET", "/product/11", nil)
    response := executeRequest(req)

    checkResponseCode(t, http.StatusNotFound, response.Code)

    var m map[string]string
    json.Unmarshal(response.Body.Bytes(), &m)
    if m["error"] != "Product not found" {
        t.Errorf("Expected the 'error' key of the response to be set to 'Product not found'. Got '%s'", m["error"])
    }
}

func TestCreateProduct(t *testing.T) {

    clearTable()

    var jsonStr = []byte(`{"name":"test product", "price": 11.22}`)
    req, _ := http.NewRequest("POST", "/product", bytes.NewBuffer(jsonStr))
    req.Header.Set("Authorization", adminToken)
    req.Header.Set("Content-Type", "application/json")

    response := executeRequest(req)
    checkResponseCode(t, http.StatusCreated, response.Code)

    var m map[string]interface{}
    json.Unmarshal(response.Body.Bytes(), &m)

    if m["name"] != "test product" {
        t.Errorf("Expected product name to be 'test product'. Got '%v'", m["name"])
    }

    if m["price"] != 11.22 {
        t.Errorf("Expected product price to be '11.22'. Got '%v'", m["price"])
    }

    // the id is compared to 1.0 because JSON unmarshaling converts numbers to
    // floats, when the target is a map[string]interface{}
    if m["id"] != 1.0 {
        t.Errorf("Expected product ID to be '1'. Got '%v'", m["id"])
    }
}

func TestGetProduct(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
```

```
    checkResponseCode(t, http.StatusOK, response.Code)
}

// main_test.go

func addProducts(count int) {
    if count < 1 {
        count = 1
    }

    for i := 0; i < count; i++ {
        a.DB.Exec("INSERT INTO products(name, price) VALUES($1, $2)", "Product "+strconv.Itoa(i), (i+1.0)*10)
    }
}

func TestUpdateProduct(t *testing.T) {

    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    var originalProduct map[string]interface{}
    json.Unmarshal(response.Body.Bytes(), &originalProduct)

    var jsonStr = []byte(`{"name":"test product - updated name", "price": 11.22}`)
    req, _ = http.NewRequest("PUT", "/product/1", bytes.NewBuffer(jsonStr))
    req.Header.Set("Authorization", adminToken)
    req.Header.Set("Content-Type", "application/json")

    response = executeRequest(req)

    checkResponseCode(t, http.StatusOK, response.Code)

    var m map[string]interface{}
    json.Unmarshal(response.Body.Bytes(), &m)

    if m["id"] != originalProduct["id"] {
        t.Errorf("Expected the id to remain the same (%v). Got %v", originalProduct["id"], m["id"])
    }

    if m["name"] == originalProduct["name"] {
        t.Errorf("Expected the name to change from '%v' to '%v'. Got '%v'", originalProduct["name"], m["name"], m["name"])
    }

    if m["price"] == originalProduct["price"] {
        t.Errorf("Expected the price to change from '%v' to '%v'. Got '%v'", originalProduct["price"], m["price"], m["price"])
    }
}

func TestDeleteProductWithSufficientPermissions(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    req.Header.Set("Authorization", adminToken)
```

```
    response = executeRequest(req)

    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("GET", "/product/1", nil)
    response = executeRequest(req)
    checkResponseCode(t, http.StatusNotFound, response.Code)
}

func TestDeleteProductWithoutToken(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    response = executeRequest(req)

    checkResponseCode(t, http.StatusUnauthorized, response.Code)
}

func TestDeleteProductWithInsufficientPermissions(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    req.Header.Set("Authorization", moderatorToken)
    response = executeRequest(req)

    checkResponseCode(t, http.StatusUnauthorized, response.Code)
}

func TestDeleteProductWithInvalidToken(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    req.Header.Set("Authorization", "invalidToken")
    response = executeRequest(req)

    checkResponseCode(t, http.StatusUnauthorized, response.Code)
}
```

1.4. main_test.go

As with every service, we naturally need tests to ensure our service is working as expected. The `main_test.go` file contains simple tests for the endpoints provided by the service.

Listing 4. main_test.go

```
// main_test.go

package main

import (
    "log"
    "os"
    "testing"

    "bytes"
    "encoding/json"
    "github.com/stretchr/testify/assert"
    "net/http"
    "net/http/httptest"
    "strconv"
)

var a App

var adminToken = "your_admin_jwt_token"
var moderatorToken = "your_moderator_jwt_token"

func TestMain(m *testing.M) {
    a.Initialize(
        os.Getenv("APP_DB_USERNAME"),
        os.Getenv("APP_DB_PASSWORD"),
        os.Getenv("APP_DB_NAME"),
        os.Getenv("APP_JWT_SECRET"))

    ensureTableExists()

    code := m.Run()
    clearTable()
    os.Exit(code)
}

// these need to be executed before all other tests
func TestGenerateAdminTokenSuccess(t *testing.T) {
    payload := []byte(`{"username":"admin","password":"admin_password"}`)

    req, _ := http.NewRequest("POST", "/token", bytes.NewBuffer(payload))
    req.Header.Set("Content-Type", "application/json")
    response := executeRequest(req)

    assert.Equal(t, http.StatusOK, response.Code)

    var responseBody map[string]string
    err := json.Unmarshal(response.Body.Bytes(), &responseBody)
    assert.Nil(t, err)
    assert.NotEmpty(t, responseBody["token"])

    adminToken = responseBody["token"] // Update the global variable with the retrieved token
}

func TestGenerateModeratorTokenSuccess(t *testing.T) {
    payload := []byte(`{"username":"moderator","password":"moderator_password"}`)

    req, _ := http.NewRequest("POST", "/token", bytes.NewBuffer(payload))
```

```
    req.Header.Set("Content-Type", "application/json")
    response := executeRequest(req)

    assert.Equal(t, http.StatusOK, response.Code)

    var responseBody map[string]string
    err := json.Unmarshal(response.Body.Bytes(), &responseBody)
    assert.Nil(t, err)
    assert.NotEmpty(t, responseBody["token"])

    moderatorToken = responseBody["token"] // Update the global variable with the retrieved token
}

func TestGenerateTokenFail(t *testing.T) {
    payload := []byte(`{"username":"admin","password":"wrong_password"}`)

    req, _ := http.NewRequest("POST", "/token", bytes.NewBuffer(payload))
    req.Header.Set("Content-Type", "application/json")
    response := executeRequest(req)

    assert.Equal(t, http.StatusUnauthorized, response.Code)
}

func ensureTableExists() {
    if _, err := a.DB.Exec(tableCreationQuery); err != nil {
        log.Fatal(err)
    }
}

func clearTable() {
    a.DB.Exec("DELETE FROM products")
    a.DB.Exec("ALTER SEQUENCE products_id_seq RESTART WITH 1")
}

const tableCreationQuery = `CREATE TABLE IF NOT EXISTS products
(
    id SERIAL,
    name TEXT NOT NULL,
    price NUMERIC(10,2) NOT NULL DEFAULT 0.00,
    CONSTRAINT products_pkey PRIMARY KEY (id)
)`

func TestEmptyTable(t *testing.T) {
    clearTable()

    req, _ := http.NewRequest("GET", "/products", nil)
    response := executeRequest(req)

    checkResponseCode(t, http.StatusOK, response.Code)

    if body := response.Body.String(); body != "[]" {
        t.Errorf("Expected an empty array. Got %s", body)
    }
}

func executeRequest(req *http.Request) *httptest.ResponseRecorder {
    rr := httptest.NewRecorder()
    a.Router.ServeHTTP(rr, req)

    return rr
```

```
}

func checkResponseCode(t *testing.T, expected, actual int) {
    if expected != actual {
        t.Errorf("Expected response code %d. Got %d\n", expected, actual)
    }
}

func TestGetNonExistentProduct(t *testing.T) {
    clearTable()

    req, _ := http.NewRequest("GET", "/product/11", nil)
    response := executeRequest(req)

    checkResponseCode(t, http.StatusNotFound, response.Code)

    var m map[string]string
    json.Unmarshal(response.Body.Bytes(), &m)
    if m["error"] != "Product not found" {
        t.Errorf("Expected the 'error' key of the response to be set to 'Product not found'. Got '%s'", m["error"])
    }
}

func TestCreateProduct(t *testing.T) {

    clearTable()

    var jsonStr = []byte(`{"name":"test product", "price": 11.22}`)
    req, _ := http.NewRequest("POST", "/product", bytes.NewBuffer(jsonStr))
    req.Header.Set("Authorization", adminToken)
    req.Header.Set("Content-Type", "application/json")

    response := executeRequest(req)
    checkResponseCode(t, http.StatusCreated, response.Code)

    var m map[string]interface{}
    json.Unmarshal(response.Body.Bytes(), &m)

    if m["name"] != "test product" {
        t.Errorf("Expected product name to be 'test product'. Got '%v'", m["name"])
    }

    if m["price"] != 11.22 {
        t.Errorf("Expected product price to be '11.22'. Got '%v'", m["price"])
    }

    // the id is compared to 1.0 because JSON unmarshaling converts numbers to
    // floats, when the target is a map[string]interface{}
    if m["id"] != 1.0 {
        t.Errorf("Expected product ID to be '1'. Got '%v'", m["id"])
    }
}

func TestGetProduct(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
```

```
    checkResponseCode(t, http.StatusOK, response.Code)
}

// main_test.go

func addProducts(count int) {
    if count < 1 {
        count = 1
    }

    for i := 0; i < count; i++ {
        a.DB.Exec("INSERT INTO products(name, price) VALUES($1, $2)", "Product "+strconv.Itoa(i), (i+1.0)*10)
    }
}

func TestUpdateProduct(t *testing.T) {

    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    var originalProduct map[string]interface{}
    json.Unmarshal(response.Body.Bytes(), &originalProduct)

    var jsonStr = []byte(`{"name":"test product - updated name", "price": 11.22}`)
    req, _ = http.NewRequest("PUT", "/product/1", bytes.NewBuffer(jsonStr))
    req.Header.Set("Authorization", adminToken)
    req.Header.Set("Content-Type", "application/json")

    response = executeRequest(req)

    checkResponseCode(t, http.StatusOK, response.Code)

    var m map[string]interface{}
    json.Unmarshal(response.Body.Bytes(), &m)

    if m["id"] != originalProduct["id"] {
        t.Errorf("Expected the id to remain the same (%v). Got %v", originalProduct["id"], m["id"])
    }

    if m["name"] == originalProduct["name"] {
        t.Errorf("Expected the name to change from '%v' to '%v'. Got '%v'", originalProduct["name"], m["name"], m["name"])
    }

    if m["price"] == originalProduct["price"] {
        t.Errorf("Expected the price to change from '%v' to '%v'. Got '%v'", originalProduct["price"], m["price"], m["price"])
    }
}

func TestDeleteProductWithSufficientPermissions(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    req.Header.Set("Authorization", adminToken)
```

```
    response = executeRequest(req)

    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("GET", "/product/1", nil)
    response = executeRequest(req)
    checkResponseCode(t, http.StatusNotFound, response.Code)
}

func TestDeleteProductWithoutToken(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    response = executeRequest(req)

    checkResponseCode(t, http.StatusUnauthorized, response.Code)
}

func TestDeleteProductWithInsufficientPermissions(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    req.Header.Set("Authorization", moderatorToken)
    response = executeRequest(req)

    checkResponseCode(t, http.StatusUnauthorized, response.Code)
}

func TestDeleteProductWithInvalidToken(t *testing.T) {
    clearTable()
    addProducts(1)

    req, _ := http.NewRequest("GET", "/product/1", nil)
    response := executeRequest(req)
    checkResponseCode(t, http.StatusOK, response.Code)

    req, _ = http.NewRequest("DELETE", "/product/1", nil)
    req.Header.Set("Authorization", "invalidToken")
    response = executeRequest(req)

    checkResponseCode(t, http.StatusUnauthorized, response.Code)
}
```

2. Add-ons

For the exercise, I added 2 additional features to the tutorial. A `health` endpoint that tells us whether the service is up and running healthy and JWT authentication for the creation, update and delete endpoints.

2.1. Health

The `health` endpoint is really simple and returns information on the status of the microservice. Since our microservice is so simple, it just checks whether it can ping the database and return answers that fit the Health Mricoprofile specification.

Listing 5. app.go

```
...
// Healthcheck endpoint that complies with Java Microprofile Health specification
func (a *App) healthCheck(w http.ResponseWriter, r *http.Request) {
    dbErr := a.DB.Ping()
    if dbErr != nil {
        health := map[string]interface{}{
            "status": "DOWN",
            "checks": []map[string]interface{}{
                {
                    "name":   "database",
                    "status": "DOWN",
                },
            },
        }
        respondWithJSON(w, http.StatusServiceUnavailable, health)
        return
    }

    health := map[string]interface{}{
        "status": "UP",
        "checks": []map[string]interface{}{
            {
                "name":   "database",
                "status": "UP",
            },
        },
    }

    respondWithJSON(w, http.StatusOK, health)
}
...
    a.Router.HandleFunc("/health", a.healthCheck).Methods("GET")
...
```

The Microprofile specification says the response should follow the following schema:

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "type": "object",
 "properties": {
   "status": {
     "type": "string"
   },
   "checks": {
     "type": "array",
     "items": {
       "type": "object",
       "properties": {
         "name": {
           "type": "string"
         },
```

```
        "status": {
          "type": "string"
        },
        "data": {
          "type": "object",
          "patternProperties": {
            "[a-zA-Z_]*": {
              "type": [
                "string",
                "boolean",
                "number"
              ]
            }
          },
          "additionalProperties": false
        }
      },
      "required": [
        "name",
        "status"
      ]
    }
  }
 },
 "required": [
   "status",
   "checks"
 ],
 "additionalProperties": false
}
```

More info on the specification can be found here.

2.1.1. JWT Authentication

To secure the endpoints created in the tutorial I added JWT Authentication as a feature. For the authentication, the `/token` is provided so user can request a JWT Token. To make it more interesting, the token can have different roles that define for which endpoint the permissions suffice. I used a middleware (`jwtAuthentication`) to intercept the endpoints that are should be guarded with Authentication. If no token or a token with invalid permissions is present in the `Authorization` header, the request will be denied by `401 Unauthorized`.

I did not implement users, so I used hardcoded credentials to be able to request 2 different kinds of JWT Tokens, a Admin and a Moderator token. The admin token can be used for all endpoints. The Moderator token will only work for creating and updating.

Which roles are needed for an endpoint is handled in `initializeRoutes()`. For generating JWT tokens, I used a symmetric approach that should not be used in production that uses a secret passed to the application. For generating, parsing and validating JWT Tokens I used the `github.com/golang-jwt/jwt/v5` library.

Listing 6. app.go

```
...
```

```go
// Generate a JWT token
func (a *App) generateJWT(username, role string) (string, error) {
    expirationTime := time.Now().Add(5 * time.Minute) // token expires in 5 minutes

    claims := jwt.MapClaims{
        "exp":      expirationTime.Unix(),
        "iss":      "go-mux",
        "sub":      role,
        "username": username,
    }

    token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)

    tokenString, err := token.SignedString([]byte(a.JwtSecret))
    if err != nil {
        return "", err
    }

    return tokenString, nil
}

// Middleware to check if the request has a valid JWT token
func (a *App) jwtAuthentication(requiredRoles []string, next http.HandlerFunc) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        tokenString := r.Header.Get("Authorization")
        if tokenString == "" {
            respondWithError(w, http.StatusUnauthorized, "No token provided")
            return
        }

        claims := jwt.MapClaims{}
        token, err := jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (interface{}, error) {
            if _, ok := token.Method.(*jwt.SigningMethodHMAC); !ok {
                return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])
            }
            return []byte(a.JwtSecret), nil
        })

        if err != nil || !token.Valid {
            respondWithError(w, http.StatusUnauthorized, "Invalid token")
            return
        }

        if !contains(requiredRoles, claims["sub"].(string)) {
            respondWithError(w, http.StatusUnauthorized, "Insufficient permissions")
            return
        }

        next(w, r)
    }
}

// helper function to check if a string is in a slice
func contains(arr []string, str string) bool {
    for _, item := range arr {
        if item == str {
            return true
        }
    }
    return false
```

```
}

func (a *App) generateToken(w http.ResponseWriter, r *http.Request) {
    var creds struct {
        Username string `json:"username"`
        Password string `json:"password"`
    }

    err := json.NewDecoder(r.Body).Decode(&creds)
    if err != nil {
        respondWithError(w, http.StatusBadRequest, "Invalid request payload")
        return
    }

    // Replace this with proper authentication logic
    // hardcoded users => could be replaced with users from a database
    if creds.Username == "admin" && creds.Password == "admin_password" {
        token, err := a.generateJWT("admin", "admin")
        if err != nil {
            respondWithError(w, http.StatusInternalServerError, err.Error())
            return
        }
        respondWithJSON(w, http.StatusOK, map[string]string{"token": token})
    } else if creds.Username == "moderator" && creds.Password == "moderator_password" {
        token, err := a.generateJWT("moderator", "moderator")
        if err != nil {
            respondWithError(w, http.StatusInternalServerError, err.Error())
            return
        }
        respondWithJSON(w, http.StatusOK, map[string]string{"token": token})
    } else {
        respondWithError(w, http.StatusUnauthorized, "Invalid credentials")
    }
}

func (a *App) initializeRoutes() {
    a.Router.HandleFunc("/products", a.getProducts).Methods("GET")
    a.Router.HandleFunc("/product/{id:[0-9]+}", a.getProduct).Methods("GET")
    // use jwtAuthentication middleware to protect the following endpoints
    a.Router.HandleFunc("/product", a.jwtAuthentication([]string{"admin", "moderator"}, a.createProduct)).Methods("POST")
    a.Router.HandleFunc("/product/{id:[0-9]+}", a.jwtAuthentication([]string{"admin", "moderator"}, a.updateProduct)).Methods("PUT")
    a.Router.HandleFunc("/product/{id:[0-9]+}", a.jwtAuthentication([]string{"admin"}, a.deleteProduct)).Methods("DELETE")
    // additional endpoints
    a.Router.HandleFunc("/health", a.healthCheck).Methods("GET")
    a.Router.HandleFunc("/token", a.generateToken).Methods("POST")
}
```

Of course, I also added and updated the tests accordingly. (take a look at the previous sections)

2.2. Setup

For running this application you need to have docker installed and fire up a postgres database with this command:

```
docker run -it -p 5432:5432 -e POSTGRES_HOST_AUTH_METHOD=trust -d postgres
```

Following this, you should set up the following environment variables:

```
export APP_JWT_SECRET=postgres
export APP_DB_USERNAME=postgres
export APP_DB_PASSWORD=
```

```
export APP_DB_NAME=postgres
```

The test can be run via:

```
go test -v
```