IceCream API

Project Summary

A JSON Web Token (JWT) authenticated REST API, to perform CRUD operations on a pre-seeded MongoDB of various icecream products, is developed. The code is completely containerized using Docker images.

Instructions

1. **Setup Docker**
    - Install Docker following the instructions for Windows or for Mac.
2. **Source code**
    - Unzip `icecreamapi.zip` source code into a folder. For example, assume it is unzipped into `C:/goWorkspace/src/icecreamapi` folder.
3. **Build 2 Docker Images** *(requires internet connectivity)*
    - In a bash terminal, navigate to the project folder, i.e., `C:/goWorkspace/src/icecreamapi`.
    - Build a docker image of `icecream` from its Dockerfile by excuting:

    ```
    docker build -t "icecream" .
    ```

    - Navigate to the `seeddata` subdirectory in the project folder, i.e., `C:/goWorkspace/src/icecreamapi/seeddata`, by executing:

    ```
    cd seeddata/
    ```

    - Build a docker image of `seeddata` from its Dockerfile by excuting:

    ```
    docker build -t "seeddata" .
    ```

4. **Run Docker-Compose to Start IceCream Application** *(requires internet connectivity)*
    - In a bash terminal, navigate to the project folder, i.e. `C:/goWorkspace/src/icecreamapi`. Start the application by running docker-compose.

    ```
    docker-compose up
    ```

5. **Functional API Testing**

- Use an API development environment tool such as Postman to test the RESTful API endpoints.
- LOGIN into the `icecream` application by posting `name` and `password` in Postman. Currently, the application only has one authorized user with the following authentication details.

```
POST http://localhost:8080/login
BODY
{
    "name"  : "user1",
    "password" : "1234"
}
```

A JWT with a validity for 5 minutes, will be returned in the http response, such as:

```
{
    "tokenString":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE1Mzk1OTIxNDUsImlzcyI6I
khvbWVCYXNlIn0.r9JlDX_FmoPnRMzf_yox_oCw0Lj0Obyql2Liz8gpE10"
}
```

Please do not use the above token string in your testing as it has exceeded its 5 minutes validity period.

- Copy and paste the JWT string into the `Token` field under `Auth->Bearer Token` in Postman, to be used for subsequent operations.
- POST a new product into the `icecream` database. New products are posted by `name` and thus the field `name` is mandatory.

```
POST http://localhost:8080/product
BODY
{
    "name"                  : "ChocolateLava",
    "image_closed"          : "",
    "image_open"            : "",
    "description"           : "Molten chocolate",
    "story"                 : "Erupting volcano",
    "sourcing_values"       : ["Non-GMO"],
    "ingredients"           : ["koko","sugar"],
    "allergy_info"          : "none",
    "dietary_certifications" : "Halal",
    "productID"             : "9870"
}
```

The database enforces unique product `name` and `productID` indexes. Hence, products with same `name` and/or `productID` cannot be inserted into the database.

- UPDATE a product in the database. Products are updated by `name`, thus the field `name` is mandatory and must match an existing product in the database.

```
PUT http://localhost:8080/product
BODY
{
    "name"                   : "ChocolateLava",
    "image_closed"           : "",
    "image_open"             : "",
    "description"            : "Molten chocolate",
    "story"                  : "Erupting volcano",
    "sourcing_values"        : ["Non-GMO"],
    "ingredients"            : ["koko","sugar, milk, eggs"],
    "allergy_info"           : "contains milk, eggs",
    "dietary_certifications" : "Halal",
    "productID"              : "9870"
}
```

Upon successfully updating the database, a http response is received as:

```
{
    "Result": "Successfully updated"
}
```

- GET a single product from the database. Products are retrieved by `name`, thus the field `name` is mandatory and must match an existing product in the database.

```
GET http://localhost:8080/product/?doc=ChocolateLava
```

The retrieved product is returned in the http response as:

```
{
    "name"                   : "ChocolateLava",
    "image_closed"           : "",
    "image_open"             : "",
    "description"            : "Molten chocolate",
    "story"                  : "Erupting volcano",
    "sourcing_values"        : ["Non-GMO"],
    "ingredients"            : ["koko","sugar, milk, eggs"],
    "allergy_info"           : "contains milk, eggs",
    "dietary_certifications" : "Halal",
    "productID"              : "9870"
}
```

- DELETE a single product from the database. Products are deleted by `name`, thus the field `name` is mandatory and must match an existing product in the database.

```
DELETE http://localhost:8080/product/?doc=ChocolateLava
```

Upon successfully deleting the product, a http response is received as:

```
{
    "Result": "Successfully deleted"
}
```

- GET all products from the database.

```
GET http://localhost:8080/product
```

All products from the databse is returned in the http response in JSON format.

Project Structure

The project structure is as follows:

```
project                          # Assumed to be located at C:/goWorkspace/
└── src                          #
    └── icecreamapi              # Main folder
        ├── seeddata             # To import initial data into MongoDB
        │   ├── Dockerfile       # To build image for initializing MongoDB
        │   └── icecream.json    # Seed data
        ├── vendor               # Folder containing dependencies
        │   ├── credentials      # Dependant package `credentials`
        │   │   ├── jwtoken.go   # Create and authenticate JWT
        │   │   └── login.go     # Hash and compare login passwords
        │   ├── database         # Dependant package `database`
        │   │   ├── connection.go # Generic database connection function
        │   │   └── product.go   # Database CRUD operations
        │   ├── document         # Dependant package `document`
        │   │   └── icecream.go  # `icecream` document to be stored in MongoDB
        │   └── handler          # Dependant package `handler`
        │       └── respond.go   # Generic http response functions
        ├── Docker-compose.yml   # To compose 3 containerized services
        ├── Dockerfile           # Dockerfile to build `icecream` api image
        ├── handlers.go          # Handlers for RESTful operation
        ├── main.go              # Main file of Go code
        └── verify.go            # Verify user login and obtain claims
```

Notes on Solution

1. **Language and Structure**

- The code is written in Golang.
- MongoDB is used as the store database.
- A server with REST endpoints listens at port `localhost:8080`.
- The code is completely containerized for easy deployment, with 3 docker images. Namely,
  - `mongo` - to create Mongo database, this image will be directly pulled via internet from the Docker Hub,
  - `seeddata` - to initialize the database with seed data, and
  - `icecream` - for CRUD operations.

2. **Authentication**

- Only authorized users are able to access the REST endpoints to perform CRUD operations on the database.
- Users are authenticated using JWT mechanism, as it is advantageous for scalability.
- Currently, only one authenticated user is present: `{"name":"user1","password":"1234"}`. Passwords stored in the source code are hashed using bcrypt algorithm.
- Additionally, MongoDB access is protected with an username and password.
  - `--username admin1 --password abcd --authenticationDatabase admin`

3. **Docker Notes**

- When docker-compose is run with Docker-Toolbox, go to `192.168.99.100:8080/` to interact with the application. `192.168.99.100` is the IP address of your docker-machine. Execute `docker-machine ip` to get IP address of your docker-machine.
- To tear down current containers and stored volume: `docker-compose down -v`
- To prune all dangling containers, networks, and build caches: `docker system prune`