Steven Culwell
1001783662

Input Validation Assignment

Building and Running

- First, download and extract the assignment, then cd into it:

```
cd CULWELL_InputValidation_Project
```

- Then build the application with docker:

```
docker build -t secure-go .
```

  When the container is built, all unit tests will be run to make sure everything is working.
- Then run the application with docker:

```
docker run -v ./data:/data -u 1000:1000 -p 8080:8080 secure-go
```

- The application should now be running on port 8080.
- When connecting, use the Bearer Token `a3446bb2-0ae2-4c4d-8bf5-4746f0d6e5b9` for read-only, and `eeef7da7-6c95-4d71-b2d0-3e6924e37462` for read-write. These are defined in the `users.json` file.
- To list all users in the database, send a GET request to the URL:

  http://localhost:8080/PhoneBook/list

  and make sure to pass either the read-only Bearer Token or read-write Bearer Token in the Authorization header.
- To add a user, send a POST request to the URL:

  http://localhost:8080/PhoneBook/add

  and make sure to pass the read-write Bearer Token in the Authorization header. In the body of the request, you should include the name and number of the user to be added, like:

```
{
  "name": "Steven Culwell",
  "phone": "+1(211)321-4312"
}
```

- To delete by name, send a PUT request with the name of the user to delete as a URL encoded string labeled “name”. The URL should look something like:

  http://localhost:8080/PhoneBook/deleteByName?name=someone

  where “someone” is the name of the person to delete. Make sure to include the read-write Bearer Token in the Authorization header.
- To delete by number, send a PUT request with the number of the user to delete as a URL encoded string labeled “number”. The URL should look something like:

  http://localhost:8080/PhoneBook/deleteByNumber?number=num

  where “num” is the number of the person to delete. Make sure to include the read-write Bearer Token in the Authorization header.

Steven Culwell
1001783662

Description of Code

For the HTTP routing, Gorilla's Mux HTTP router was used. This handles each endpoint, including the ADD, LIST, DEL by name, and DEL by number operations, as well as the middleware for authentication.

There are two default users in the system: *read-only* and *read-write*. They are stored in the file `users.json`, which must be present at the root of the project directory. The read-only user has permission to run operations that require the "read" permission, but not operations that require the "write" permission. The only operation that fits this description is the LIST operation. The read-write user has permission to run operations that require read and/or write permissions. This encompasses all operations: LIST, ADD, DEL by name, and DEL by number. Both the read-only and read-write user have two attributes stored: "token" and "permissions". "token" stores a single string, which is used as a bearer token in authentication. "permissions" is a list of strings, which stores each permission granted to that specific user. The supplied `users.json` file is:

```
[
  {
    "name": "read-only",
    "token": "a3446bb2-0ae2-4c4d-8bf5-4746f0d6e5b9",
    "permissions": ["read"]
  },
  {
    "name": "read-write",
    "token": "eeef7da7-6c95-4d71-b2d0-3e6924e37462",
    "permissions": ["read", "write"]
  }
]
```

On application start, all users are loaded by the `auth` package and stored in a struct `Auth` as an array of `User`, which have all required attributes. This package exposes a method called `Middleware`, which can be used to build a middleware function with a set of required permissions.

For each endpoint, a middleware has been put into place. This middleware will be inserted before the operation is called, and is used to check for authentication. There are two middleware's used: one that will allow read operations, and one that will allow both read and/or write operations. Each middleware will check for the *Authorization* header, which should contain a *Bearer Token* in the format:

```
Authorization: Bearer <TOKEN>
```

where `<TOKEN>` is the token used to authenticate the user. The middleware will then check this token against each user, and only allow the router to continue to the requested operation if the associated user's permissions includes the required permissions for this operation. So if a request came in to the ADD endpoint, and the token present is `a3446bb2-0ae2-4c4d-8bf5-4746f0d6e5b9` (read-only's token), the result would be `401: Unauthorized`.

The audit log will be stored inside the `./data/audit.log` file. This file is shared between the docker container and host machine through a volume. This volume is supplied in the `docker run` command through the `-v` flag. This file is always appended to, and not overwritten. Timestamps are

shown for when each operation is called, and the user who issued the operation is shown. All logs are in the format:

```
<Audit> 2024/04/15 08:30:25 [read-only] List all users
```

In this log, “read-only” is the user who issued the operation, the operation was LIST, and it was issued at 2024/04/15 08:30:25.

The SQLite database file is stored inside the `./data/phonebook.db` file. This file is shared between the docker container and host machine through a volume. This file is updated whenever any write operations are made by ADD, DEL by name, or DEL by number. If the database has not been initialized with the proper user table, it will be created once the application starts. All queries, insertions, and deletions on the database use parameterized queries to prevent SQL injection attacks.

The `ValidName` function is used to check whether a given name is allowed. Names which are allowed will return true, and those which are not will return false. Names that are not allowed will cause the DEL by name and ADD operation to return the status `400: Bad Request`. Otherwise, if the name is valid, the DEL by name and ADD operation will return `200: OK`. Valid names follow the rules:

1. All characters must be in the category of Unicode “letter”. This includes accented and non-accented English characters and foreign characters, but not symbols.
2. The format of the names must be “First”, “First Middle”, “First MI.”, “First Last”, “First Middle Last”, “First MI. Last”, “Last, Middle First”, or “Last, MI. First”.
3. First names may be any combination of valid characters defined in (1).
4. Last names may be any combination of valid characters defined in (1), as well as a second valid last name separated by a “-“.
5. Middle names follow the same convention as first names.
6. Middle initials (MI) must be a single valid character as defined in (1).

The `ValidPhone` function is used to check whether a given phone number is allowed. Phone numbers which are allowed will return true, and those which are not will return false. Phone numbers that are not allowed will cause the DEL by number and ADD operation to return the status `400: Bad Request`. Otherwise, if the phone number is valid, the DEL by number and ADD operation will return `200: OK`. Valid phone numbers follow the rules:

1. All non-separator characters must be digits from 0-9.
2. Area codes may be between 2 and 3 digits, with no leading zeros.
3. Phone numbers may be a 5-digit internal extension number like “12345”.
4. Phone numbers may be formatted like “011 +1 (123) 123-1234“.
5. Phone numbers may be formatted like “1-123-123-1234” or “123-123-1234 with either dashes (“-“), periods (“.”), or spaces (“ “) as separators between groups of digits.
6. Phone numbers may be formatted like “12 34 56 78” with either spaces (“ “) or periods (“.”) as separators between groups of digits.
7. Phone numbers may be formatted like “12345 67890” with either spaces (“ “) or periods (“.”) as separators between groups of digits.

Steven Culwell
1001783662

All packages `auth`, `database`, `name`, and `phone` have associated unit tests. The `auth` tests make sure that the `users.json` file is properly loaded, and that users who do not have permission to perform certain operations will receive the appropriate response. The `database` tests make sure that the database is properly loaded from disk and all database operations: `Append`, `List`, `DeleteByName`, and `DeleteByNumber` lead to the appropriate resulting database.

Assumptions

1. It is assumed that, on each request that is expected to succeed, a valid Bearer token that is also present in the `users.json` file is provided in the Authorization header like:

   ```
   Authorization: Bearer <TOKEN>
   ```

2. It is assumed that the end-user has already been provided with a token from an administrator.
3. It is assumed that the existing database does not have invalid names or phone numbers inside before the application is run.
4. It is assumed that port 8080 is not occupied by any other process.
5. It is assumed that foreign characters should be allowed in names.

Pros/Cons of My Approach

Pros

1. The database persists even on separate instances of the same container.
2. Untrusted users cannot access Bearer tokens without one being given to them by a trusted administrator.

Cons

1. There is no signup process, so new users cannot be added without an administrator adding them manually.
2. The audit log may become too large if the server receives too many requests.
3. Any existing invalid names or phone numbers will not be removed, because they are only checked on ADD.
4. The Bearer token can be accessed by anyone who is able to read the request headers.