Biographical Affidavit

Technical Documentation

Group 1:
Hipolito Bautista
Justin Chuc
Michael Gomez
Rene Allen
Dennis Arnold
Jahmur Lopez

CMPS4131
Manual Medina
April 8, 2023

Table of Contents

Documentation

File Directory Tree

```
Backend -> cmd -> api -> context.go
                       -> errors.go
                       -> handlers.go
                       -> healthCheck.go
                       -> helpers.go
                       -> main.go
                       -> middleware.go
                       -> routes.go
                       -> server.go
        -> internal -> data -> form.go
                            -> models.go
                            -> publicUsers.go
                    -> jsonlog -> jsonlog
                    -> mailer ->
                    -> validator -> validator.go
        -> migrations -> 000001_create_public_user_table.down.sql
                       -> 000001_create_public_user_table.up.sql
                       -> 000002_create_form_table.down.sql
                       -> 000002_create_form_table.up.sql
                       -> 000003_demo_insert_public_user.down.sql
                       -> 000003_demo_insert_public_user.up.sql
go.mod
go.sum
```

Above is the file directory tree of the demonstration program code.

How to start the backend of BIOAFF

The current version of the system submitted with this document consists of only the system's backend, which is structured like an API. To run the application, use the following command:

```
go run ./cmd/api
```

Start-up flags for BIOAFF's backend

Setting the port

By default, the web server of the API will be set to port:4000; however, if you wish to change the port. Use the --port flag when you start the backend.
Example:

```
go run ./cmd/api --port=5000
```

Changing code env

The following instruction doesn't affect the system besides denoting how the system is beings used. They're three environments that can be used: development, staging, and production. They can be set using the --env flag.
Example:

```
go run ./cmd/api --env="staging"
```

Setting which database DSN the system will use

The system utilizes databases; therefore, it'll need to connect to one. The system uses PostgreSQL as its DBMS. To set which database the system will use, the --db-dsn flag can be used.
Example:

```
go run ./cmd/api --db-dsn="BIOAFF_DB_DSN"
```

Setting the maximum number of open connections to the database

The system, by default, will allow for 25 open connections to its database; the number of connections can be changed using the --db-max-open-conns flag.
Example:

```
go run ./cmd/api --db-max-open-conns=40
```

Setting the maximum number of idle connections to the database

The system, by default, will allow for 25 idle connections to its database; the number of connections can be changed using the --db-idle-conns flag.
Example:

```
go run ./cmd/api --db-idle-conns=20
```

Changing the limit on requests per second

The system, by default, will allow for two requests to be received and processed at a given second. If you wish to change that limit, use the --limiter-rps flag
Example:

```
go run ./cmd/api --limiter-rps=4
```

Changing the limit of rate limit bursts

The system, by default, will allow for four requests in a burst to be processed. If you wish to change that limit, use the --limiter-burst flag
Example:

```
go run ./cmd/api --limiter-burst=8
```

Toggling the limiter

The system be the default, will use the limiter to conserve resources and prevent errors in its default state; however,, this can be turned off by setting it to false using the --limiter-enabled flag Example:

```
go run ./cmd/api --limiter-enabled=false
```

File Breakdowns

context.go

This file is currently empty; it only contains a user-defined type of “user

errors.go

This file contains all errors that the system can encounter. Once an error occurs, the system’s logger will ensure that the events preceding the error will log. The functions are simple so that the programmer can log what type of errors a code snippet might produce. Should a new error response need to be created, it can be done like this:

```go
func (app *application) exampleErrorResponse(w http.ResponseWriter, r
*http.Request){
      message := "message describing the cause of the error"
      app.errorResponse(w, r, http.StatusForbidden, message)
}
```

handlers.go

Since the system is constructed like an API, there it’ll need handlers for all the requests the system will need to process. Handlers are written to process a request and will always end by returning the header to the router, which is called the handler in the first place. Handlers are complicated therefore, there’s only one handler in the system at this moment.

```go
// submitFormHandler() - tries to create a form based on information
supplied
func (app *application) submitFormHandler(w http.ResponseWriter, r
*http.Request) {
      //Our target decode destination
      var input struct {
            PublicUser_ID          int64  `json:"puid"`
```

```go
        Status                 string `json:"status"`
        Archive                bool   `json:"archive"`
        Fullname               string `json:"fullname"`
        Othernames             string `json:"othername"`
        Has_Changed_Name       bool   `json:"changed_name"`
        SocialSecurity_Number  int    `json:"ssnumber"`
        SocialSecurity_Date    string `json:"ssdate"`
        SocialSecurity_Country string `json:"sscountry"`
        Passport_Number        string `json:"passport_number"`
        Passport_Date          string `json:"passport_date"`
        Passport_Country       string `json:"passport_country"`
        DOB                    string `json:"dob"`
        Place_of_Birth         string `json:"place_of_birth"`
        Nationality            string `json:"nationality"`
        Acquired_Nationality   string
`json:"acquired_nationality"`
        Spouse_Name            string `json:"spouse_name"`
        Address                string `json:"address"`
        Phone_Number           string `json:"phone"`
        Fax_Number             string `json:"fax"`
        Residential_Email      string `json:"residential_rmail"`
    }

    //initializing a new json.Decoder instance
    err := app.readJSON(w, r, &input)
    if err != nil {
        app.badRequestResponse(w, r, err)
        return
    }

    //copying over the values from input to the new form
    form := &data.Form{
        PublicUser_ID:          input.PublicUser_ID,
        Status:                 input.Status,
        Archive:                input.Archive,
        Fullname:               input.Fullname,
        Othernames:             input.Othernames,
        Has_Changed_Name:       input.Has_Changed_Name,
        SocialSecurity_Number:  input.SocialSecurity_Number,
            SocialSecurity_Date:    input.SocialSecurity_Date,
            SocialSecurity_Country: input.SocialSecurity_Country,
            Passport_Number:        input.Passport_Number,
            Passport_Date:          input.Passport_Date,
            Passport_Country:       input.Passport_Country,
            DOB:                    input.DOB,
            Place_of_Birth:         input.Place_of_Birth,
            Nationality:            input.Nationality,
            Acquired_Nationality:   input.Acquired_Nationality,
            Spouse_Name:            input.Spouse_Name,
            Address:                input.Address,
            Phone_Number:           input.Phone_Number,
            Fax_Number:             input.Fax_Number,
            Residential_Email:      input.Residential_Email,
      }

      //no validation for now

      //creating the form
      err = app.models.Forms.Insert(form)
      if err != nil {
            app.serverErrorResponse(w, r, err)
      }

      //creating a location header for the newly created
resource/form
      headers := make(http.Header)
      headers.Set("Location", fmt.Sprintf("/v1/form/%d", form.ID))

      //writing jSON response with 201 - Created status code with the
body
      //being the form data and the header being the headers map
      err = app.writeJSON(w, http.StatusCreated, envelope{"form":
form}, headers)
      if err != nil {
            app.serverErrorResponse(w, r, err)
      }
}
```

Observing the code snippet of the only handler. Information is parsed from the request's json header in order to supply values for the handler to able to use them.

healthCheck.go

This file is also a handler, but it's separated from the handler.go file because this one is only used to check if the system is running and can receive requests. It'll respond with the status of the web server.

helpers.go

This file contains functions used throughout the system. The main purpose of these functions is to either assist with json processing or other system functionality.

main.go

This file contains all the configuration information, start-up configurations, packages, and most importantly connecting to the database and starting the web server.

middleware.go

Since the system utilizes a web server for its endpoints, it's important to ensure the validity of the requests that the system is receiving. Therefore this file contains all the functions that run in-between endpoints of the system.

- recoverPanic prevents the client from trying to load something after the database has been closed
- rateLimit limits the number of requests that can be process by the system at any given moment
- enableCORs is ensure that authentication can be enforced on the clients of the system.

routes.go

Since the system is constructed like an API, it needs endpoints for its clients. All endpoints or routes are coded in this file. The middleware functions are also implemented inside this file since they need to operate before the request can even be processed by the system.

server.go

This file contains the webserver and its startup code where it receives its configuration settings from the main.go

form.go

Contains the model used to represent the database entity that is the form, for the system to use. It also contains the CRUD functions for it's respective model, to interact with the database.

publicUsers.go

Contains the model used to represent the database entity that is the public users, for the system to use. It also contains the CRUD functions for its respective model, to interact with the database.

models.go

Wraps up all the models into one package so that the rest of the system can use the models via a connection in main.go.

jsonlog.go

Contains the logger used throughout the system, utilizing json. All events are logged onto a map inside the system. The moment a logged event occurs it'll be displayed in the CLI for the admin. It also logs the unwinding of the system upon a fatal/normal error in the system.

validator.go

Contains all the validation functions used to check information received from requests to the system.

Migrations Folder

The migrations folder contains all the necessary SQL files for the database to be ready to receive information from the system.