APPENDIX

1. MONOLITH ARCHITECTURE

IMPORT DATABASE PACKAGE

```
1.   package database
2.
3. import (
4.   "database/sql"
5.   "log"
6.   "os"
7.
8.   _ "github.com/go-sql-driver/mysql"
9.   "github.com/joho/godotenv"
10.  )
```

CONNECT TO DATABASE

```
1. var DB *sql.DB
2. var err error
3.
4. func Connect_db() {
5.
6.   // load the env file
7.   var err = godotenv.Load(".env")
8.   if err != nil {
9.        log.Fatalf("Error loading .env files")
10.       }
11.
12.       var DB_USER = os.Getenv("DB_USER")
13.       var DB_PASSWORD = os.Getenv("DB_PASSWORD")
14.       var DB_NAME = os.Getenv("DB_NAME")
15.       var DB_HOST = os.Getenv("DB_HOST")
16.
17.       // connect to the database
18.       DB,         err         =         sql.Open("mysql",
   DB_USER+":"+DB_PASSWORD+"@tcp("+DB_HOST+")/"+DB_NAME)
19.       if err != nil {
20.            log.Fatalln(err)
21.       }
22.
23.       err = DB.Ping()
24.       if err != nil {
25.            log.Fatalln(err)
26.       }
27.  }
```

ENV CONFIGURATION

```
1. DB_USER=root
2. DB_PASSWORD=
3. DB_NAME=dblistfeature
```

```
4. DB_PORT=3306
5. DB_HOST=127.0.0.1
6. JWT_SIGNATURE_KEY=this_is_secret
7. APPLICATION_NAME=List_Feature_App
```

CREATE TABLE FUNCTION

```
1. package database
2.
3. func Create_table() {
4.   _, errFeature := DB.Exec("CREATE TABLE IF NOT EXISTS
   tblFeature(featureId varchar(36) not null primary key,
   featureName varchar(50), featureDescription varchar(30),
   featureLiveDate varchar(10), featureSegmentation
   varchar(30), featureChannel varchar(30), limitTransaction
   varchar(20), limitDaily varchar(20), limitMonthly
   varchar(20), created_by varchar(50), created_date timestamp
   default CURRENT_TIMESTAMP NOT NULL, update_by varchar(20),
   update_date date); ")
5.   if errFeature != nil {
6.       panic(errFeature)
7.   }
8.
9.   _, errUser := DB.Exec("CREATE TABLE IF NOT EXISTS
   tblUser(user_id varchar(36) not null primary key, username
   varchar(20), user_email varchar(20), user_password
   varchar(100));")
10.      if errUser != nil {
11.          panic(errUser)
12.      }
13.  }
```

IMPORT ROUTES FUNCTION PACKAGE

```
1. package routes
2.
3. import (
4.   "fmt"
5.   "go-monolith/data"
6.   "go-monolith/middleware"
7.   "go-monolith/user"
8.   "net/http"
9.
10.      "github.com/gorilla/mux"
11.  )
```

ROUTES FUNCTION

```
1. func Register() {
2.
3.   // create new router
4.   r := mux.NewRouter()
5.   getR := r.Methods(http.MethodGet).Subrouter()
6.   postR := r.Methods(http.MethodPost).Subrouter()
```

```
7.   deleteR := r.Methods(http.MethodDelete).Subrouter()
8.   putR := r.Methods(http.MethodPut).Subrouter()
9.
10.       // register router without middleware
11.       // user login
12.       r.HandleFunc("/login",
   user.LoginHandler).Methods("POST")
13.       r.HandleFunc("/register",
   user.RegisterHandler).Methods("POST")
14.
15.       // register router with middleware
16.       // view feature data
17.       getR.HandleFunc("/viewData", data.ViewData)
18.       getR.HandleFunc("/viewData/{id}", data.ViewDataById)
19.
20.       // insert feature data
21.       postR.HandleFunc("/insertData", data.InsertData)
22.
23.       // delete feature data
24.       deleteR.HandleFunc("/deleteData/{id}",
   data.DeleteData)
25.
26.       // update feature data
27.       putR.HandleFunc("/updateData/{id}", data.UpdateData)
28.
29.       // middleware
30.       getR.Use(middleware.MiddlewareJWTAuthorization)
31.       postR.Use(middleware.MiddlewareJWTAuthorization)
32.       deleteR.Use(middleware.MiddlewareJWTAuthorization)
33.       putR.Use(middleware.MiddlewareJWTAuthorization)
34.
35.       server := &http.Server{
36.            Handler: r,
37.            Addr:    ":8080",
38.       }
39.       fmt.Println("Server          running          on
   http://localhost:8080")
40.       server.ListenAndServe()
41.  }
```

IMPORT USER SERVICE PACKAGE

```
1. import (
2.   "encoding/json"
3.   "fmt"
4.   "go-monolith/database"
5.   "go-monolith/validation"
6.   "go-monolith/variable"
7.   "io/ioutil"
8.   "net/http"
9.   "time"
10.
11.       "github.com/golang-jwt/jwt"
12.       "github.com/google/uuid"
13.       "golang.org/x/crypto/bcrypt"
14.  )
```

USER SERVICE FUNCTION

```
1. type user_login struct {
2.   Email    string
3.   Password string
4. }
5.
6. type data_register_user struct {
7.   Id       int
8.   Username string
9.   Email    string
10.       Password string
11.  }
12.
13.  type TokenDetails struct {
14.       AccessToken string
15.       AccessUuid  string
16.       AtExpires   int64
17.  }
18.
19.  type MyClaims struct {
20.       jwt.StandardClaims
21.       Username string `json:"username"`
22.       Email    string `json:"email"`
23.  }
24.
25.  func QueryUser(email string) data_register_user {
26.       var users = data_register_user{}
27.       _     =    database.DB.QueryRow(`SELECT    username,
   user_email, user_password FROM tblUser WHERE user_email=?`,
   email).Scan(&users.Username,                  &users.Email,
   &users.Password)
28.       return users
29.  }
30.
31.  func checkErr(w http.ResponseWriter, r *http.Request, err
   error) bool {
32.       if err != nil {
33.            fmt.Println(r.Host + r.URL.Path)
34.            http.Redirect(w, r, r.Host+r.URL.Path, 301)
35.            return false
36.       }
37.       return true
38.  }
39.
40.  func     RegisterHandler(w     http.ResponseWriter,     r
   *http.Request) {
41.       // check if the method is post
42.       if r.Method == "POST" {
43.            // read request from body
44.            b, err := ioutil.ReadAll(r.Body)
45.            defer r.Body.Close()
46.            if err != nil {
47.                 http.Error(w, err.Error(), 500)
48.                 return
```

```
49.           }
50.
51.           // unmarshal the request body
52.           var user data_register_user
53.           err = json.Unmarshal(b, &user)
54.           if err != nil {
55.                http.Error(w, err.Error(), 500)
56.                return
57.           }
58.
59.           // create the id
60.           var id = createUserId()
61.
62.           // validate the username and email
63.           if  !validation.StringValidation(user.Username,
   user.Email) {
64.                http.Error(w,                err.Error(),
   http.StatusBadRequest)
65.                return
66.           }
67.
68.           // Check if the user exists
69.           var users = QueryUser(user.Email)
70.
71.           // check if users in database is not exists
72.           if (data_register_user{}) == users {
73.                // hash the password
74.                hashedPassword,          err          :=
   bcrypt.GenerateFromPassword([]byte(user.Password),
   bcrypt.DefaultCost)
75.
76.                // check if hashed password success and
   check if there is error
77.                if len(hashedPassword) != 0 && checkErr(w,
   r, err) {
78.                     // insert user to database
79.                     stmt,             err             :=
   database.DB.Prepare("INSERT          INTO          tblUser
   (user_id,username,user_email,user_password)
   values(?,?,?,?);")
80.                     if err == nil {
81.                          _,    err    :=    stmt.Exec(&id,
   &user.Username, &user.Email, &hashedPassword)
82.                          if err != nil {
83.                               http.Error(w,   err.Error(),
   http.StatusInternalServerError)
84.                               return
85.                          }
86.                          //variable.Log.Info("Register
   User Success")
87.                          return
88.                     } else {
89.                          variable.Log.Error("Error
   inserting user to database", "error", err)
90.                          http.Error(w,        err.Error(),
   http.StatusInternalServerError)
```

```
91.                         return
92.                     }
93.                 } else {
94.                     variable.Log.Error("error",   "error",
   err)
95.                     http.Error(w,             err.Error(),
   http.StatusInternalServerError)
96.                     return
97.                 }
98.             } else {
99.                 variable.Log.Info("Register User Failed")
100.                http.Error(w,                  err.Error(),
   http.StatusBadRequest)
101.                return
102.            }
103.    }
104. }
105.
106. func LoginHandler(w http.ResponseWriter, r *http.Request)
   {
107.    // check if the method is post
108.    if r.Method == "POST" {
109.        // read request from body
110.        b, err := ioutil.ReadAll(r.Body)
111.        defer r.Body.Close()
112.        if err != nil {
113.            http.Error(w, err.Error(), 500)
114.            return
115.        }
116.
117.        // unmarshal the request body
118.        var data_login user_login
119.        err = json.Unmarshal(b, &data_login)
120.        if err != nil {
121.            http.Error(w, err.Error(), 500)
122.            return
123.        }
124.
125.        // Check if the user exists and get all user data
126.        users := QueryUser(data_login.Email)
127.
128.        // if there is no data return
129.        if (data_register_user{}) == users {
130.            fmt.Print(users)
131.            http.Error(w,                  err.Error(),
   http.StatusBadRequest)
132.            return
133.        }
134.
135.        // compare the password
136.        var               password_tes               =
   bcrypt.CompareHashAndPassword([]byte(users.Password),
   []byte(data_login.Password))
137.        // if password correct create jwt token
138.        if password_tes == nil {
139.
```

```
140.                // create jwt token
141.                token, err := CreateToken(&users)
142.                if err != nil {
143.                     http.Error(w, err.Error(), 500)
144.                }
145.
146.                // marshal the token and send as response
147.                tokenString,                    _                    :=
   json.Marshal(variable.M{"token": token.AccessToken})
148.
149.                w.Write([]byte(tokenString))
150.
151.           } else {
152.                http.Error(w,    "Invalid    username    or
   password", http.StatusBadRequest)
153.                return
154.           }
155.
156.      } else {
157.           http.Error(w,          "Invalid          method",
   http.StatusMethodNotAllowed)
158.           return
159.      }
160. }
161.
162. func       CreateToken(users        *data_register_user)
   (*TokenDetails, error) {
163.
164.      var           LOGIN_EXPIRATION_DURATION            =
   time.Now().Add(time.Minute * 300).Unix()
165.
166.      td := &TokenDetails{}
167.      td.AtExpires = int64(LOGIN_EXPIRATION_DURATION)
168.      td.AccessUuid = uuid.New().String()
169.
170.      claims := MyClaims{
171.           StandardClaims: jwt.StandardClaims{
172.                Issuer:    variable.APPLICATION_NAME,
173.                ExpiresAt: LOGIN_EXPIRATION_DURATION,
174.           },
175.           Username: users.Username,
176.           Email:    users.Email,
177.      }
178.
179.      token := jwt.NewWithClaims(
180.           variable.JWT_SIGNING_METHOD,
181.           claims,
182.      )
183.
184.      signedTokens,                  err                  :=
   token.SignedString(variable.JWT_SIGNATURE_KEY)
185.      if err != nil {
186.           return nil, err
187.      }
188.
189.      td.AccessToken = signedTokens
```

```
190.
191.      return td, nil
192. }
193.
194. func createUserId() uuid.UUID {
195.      // create new uuid
196.      id := uuid.Must(uuid.NewRandom())
197.      return id
198. }
```

LIST FEATURES SERVICE PACKAGE

```
1. import (
2.   "database/sql"
3.   "encoding/json"
4.   "fmt"
5.   "go-monolith/database"
6.   "go-monolith/variable"
7.   "io/ioutil"
8.   "net/http"
9.   "strconv"
10.       "time"
11.
12.       "github.com/golang-jwt/jwt"
13.       "github.com/google/uuid"
14.  )
```

LIST FEATURES SERVICE FUNCTION

```
1.  type data struct {
2.    FeatureName         string `json:"featureName"`
3.    FeatureDescription  string `json:"featureDescription"`
4.    FeatureLiveDate     string `json:"featureLiveDate"`
5.    FeatureSegmentation string `json:"featureSegmentation"`
6.    FeatureChannel      string `json:"featureChannel"`
7.    LimitTransaction    string `json:"limitTransaction"`
8.    LimitDaily          string `json:"limitDaily"`
9.    LimitMonthly        string `json:"limitMonthly"`
10.   }
11.
12.   type data_view struct {
13.       FeatureId           string `json: "featureId"`
14.       FeatureName         string `json:"featureName"`
15.       FeatureDescription                              string
    `json:"featureDescription"`
16.       FeatureLiveDate     string `json:"featureLiveDate"`
17.       FeatureSegmentation                             string
    `json:"featureSegmentation"`
18.       FeatureChannel      string `json:"featureChannel"`
19.       LimitTransaction    string `json:"limitTransaction"`
20.       LimitDaily          string `json:"limitDaily"`
21.       LimitMonthly        string `json:"limitMonthly"`
22.   }
23.
```

```
24.  func checkDuplication(featureName, featureSegmentation,
   featureChannel string) (bool, error) {
25.       // check if there is data duplication
26.       var dataCount int
27.       err := database.DB.QueryRow(`SELECT COUNT(*) from
   tblFeature where featureName=? and featureSegmentation=?
   and featureChannel=?`, featureName, featureSegmentation,
   featureChannel).Scan(&dataCount)
28.
29.       if err == sql.ErrNoRows {
30.            variable.Log.Error("Cant check the duplicate",
   "error", err)
31.            return false, err
32.       }
33.
34.       if dataCount <= 1 {
35.            return true, nil
36.       }
37.
38.       variable.Log.Info("There is a duplicate Data")
39.       return false, err
40.  }
41.
42.  func checkUser(email string) (bool, error) {
43.       // check if there is a user
44.       var usersCount int
45.       err  :=  database.DB.QueryRow(`SELECT  COUNT(email)
   from tblUser where email=?`, email).Scan(&usersCount)
46.
47.       if usersCount <= 1 {
48.            return true, nil
49.       }
50.
51.       variable.Log.Info("There is a duplicate account")
52.       return false, err
53.  }
54.
55.  func InsertData(w http.ResponseWriter, r *http.Request) {
56.       // check the http method
57.       if r.Method == "POST" {
58.
59.            // get user info from middleware
60.            userInfo                                    :=
   r.Context().Value("userInfo").(jwt.MapClaims)
61.            var email = userInfo["email"]
62.
63.            // create id
64.            id := createDataId()
65.
66.            // read request from body
67.            b, err := ioutil.ReadAll(r.Body)
68.            defer r.Body.Close()
69.            if err != nil {
70.                 http.Error(w, err.Error(), 500)
71.                 return
72.            }
```

```
73.
74.           // unmarshal the request body and casting it to
   data
75.           var data_input data
76.           err = json.Unmarshal(b, &data_input)
77.           if err != nil {
78.                http.Error(w, err.Error(), 500)
79.                return
80.           }
81.
82.           //   check   duplication   of   data   by   name,
   segmentation, and channel
83.           checkDuplicate,              err              :=
   checkDuplication(data_input.FeatureName,
   data_input.FeatureSegmentation, data_input.FeatureChannel)
84.           if err != nil {
85.                http.Error(w, err.Error(), 500)
86.                return
87.           }
88.
89.           // check if user exists
90.           userCheck, err := checkUser(email.(string))
91.           if err != nil {
92.                http.Error(w, err.Error(), 500)
93.                return
94.           }
95.
96.           if checkDuplicate && userCheck {
97.                // inserting to database
98.                statementFeature,          err          :=
   database.DB.Prepare("INSERT        INTO        tblFeature
   (featureId,featureName,featureDescription,featureLiveDate,
   featureSegmentation,featureChannel,limitTransaction,limitD
   aily,limitMonthly,created_by)
   values(?,?,?,?,?,?,?,?,?,?);")
99.                if err == nil {
100.                    _, err := statementFeature.Exec(&id,
   &data_input.FeatureName,   &data_input.FeatureDescription,
   &data_input.FeatureLiveDate,
   &data_input.FeatureSegmentation,
   &data_input.FeatureChannel,  &data_input.LimitTransaction,
   &data_input.LimitDaily, &data_input.LimitMonthly, &email)
101.                    if err != nil {
102.                         http.Error(w,        err.Error(),
   http.StatusInternalServerError)
103.                         return
104.                    }
105.               } else {
106.                    http.Error(w,             err.Error(),
   http.StatusInternalServerError)
107.                    return
108.               }
109.          } else {
110.               http.Error(w,                  err.Error(),
   http.StatusInternalServerError)
111.               return
```

```
112.          }
113.      }
114. }
115.
116. func ViewData(w http.ResponseWriter, r *http.Request) {
117.      // check the http method
118.      if r.Method == "GET" {
119.          // query all data
120.          rows,    err    :=    database.DB.Query("select
   featureId,         featureName,         featureDescription,
   featureLiveDate,    featureSegmentation,    featureChannel,
   limitTransaction,     limitDaily,     limitMonthly     from
   tblFeature;")
121.          if err != nil {
122.              fmt.Println(err.Error())
123.              return
124.          }
125.
126.          defer rows.Close()
127.
128.          var list_feature []data_view
129.
130.          // loop the rows
131.          for rows.Next() {
132.              var each = data_view{}
133.              var   err   =   rows.Scan(&each.FeatureId,
   &each.FeatureName,                &each.FeatureDescription,
   &each.FeatureLiveDate,           &each.FeatureSegmentation,
   &each.FeatureChannel,               &each.LimitTransaction,
   &each.LimitDaily, &each.LimitMonthly)
134.
135.              if err != nil {
136.                  fmt.Println(err.Error())
137.                  return
138.              }
139.
140.              list_feature = append(list_feature, each)
141.          }
142.
143.          if err = rows.Err(); err != nil {
144.              fmt.Println(err.Error())
145.              return
146.          }
147.
148.          // marshal the list and write the response
149.          output, err := json.Marshal(list_feature)
150.          if err != nil {
151.              http.Error(w, err.Error(), 500)
152.              return
153.          }
154.
155.          w.Header().Set("content-type",
   "application/json")
156.          w.Write(output)
157.
158.      }
```

```
159.
160. }
161.
162. func ViewDataById(w http.ResponseWriter, r *http.Request)
   {
163.      // get the id from parameter
164.      id := r.Context().Value("id")
165.
166.      var list_feature data_view
167.
168.      // query the data from id
169.      dberr  :=  database.DB.QueryRow("select  featureId,
   featureName,      featureDescription,      featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily,    limitMonthly    from    tblFeature    where
   featureId=?;",          id).Scan(&list_feature.FeatureId,
   &list_feature.FeatureName,
   &list_feature.FeatureDescription,
   &list_feature.FeatureLiveDate,
   &list_feature.FeatureSegmentation,
   &list_feature.FeatureChannel,
   &list_feature.LimitTransaction,  &list_feature.LimitDaily,
   &list_feature.LimitMonthly)
170.      if dberr == sql.ErrNoRows {
171.           fmt.Print(dberr)
172.           return
173.      }
174.
175.      // marshal the data annd write to response
176.      output, err := json.Marshal(list_feature)
177.      if err != nil {
178.           http.Error(w, err.Error(), 500)
179.           return
180.      }
181.
182.      w.Header().Set("content-type", "application/json")
183.      w.Write(output)
184. }
185.
186. func DeleteData(w http.ResponseWriter, r *http.Request) {
187.      id := r.Context().Value("id")
188.
189.      fmt.Print(id)
190.      var checkData int
191.
192.      err  :=  database.DB.QueryRow("SELECT  count(*)  FROM
   tblFeature where featureId=?;", id).Scan(&checkData)
193.      if err == sql.ErrNoRows {
194.           http.Error(w,                      err.Error(),
   http.StatusInternalServerError)
195.           fmt.Println(err.Error())
196.           return
197.      }
198.
199.      if checkData == 0 {
```

```
200.          http.Error(w,                          err.Error(),
   http.StatusInternalServerError)
201.          fmt.Println(err.Error())
202.          return
203.     }
204.
205.     delete,  err  :=  database.DB.Prepare("DELETE  FROM
   tblfeature where featureId=?;")
206.     if err == nil {
207.          _, err := delete.Exec(id)
208.          if err != nil {
209.               http.Error(w,                     err.Error(),
   http.StatusInternalServerError)
210.               return
211.          }
212.     }
213.
214.     variable.Log.Info("Delete Succes")
215. }
216.
217. func UpdateData(w http.ResponseWriter, r *http.Request) {
218.     // check the http method
219.     if r.Method == "PUT" {
220.
221.          // get the email from jwt token
222.          userInfo                                  :=
   r.Context().Value("userInfo").(jwt.MapClaims)
223.          var email = userInfo["email"]
224.
225.          // get the id
226.          id := r.Context().Value("id")
227.
228.          // read request from body
229.          b, err := ioutil.ReadAll(r.Body)
230.          defer r.Body.Close()
231.          if err != nil {
232.               http.Error(w, err.Error(), 500)
233.               return
234.          }
235.
236.          // unmarshal the request body and casting it to
   data
237.          var data_input data
238.          err = json.Unmarshal(b, &data_input)
239.          if err != nil {
240.               http.Error(w, err.Error(), 500)
241.               return
242.          }
243.
244.          // check the duplication
245.          checkDuplicate,             err           :=
   checkDuplication(data_input.FeatureName,
   data_input.FeatureSegmentation, data_input.FeatureChannel)
246.
247.          // check if user exists
248.          userCheck, err := checkUser(email.(string))
```

```
249.          if err != nil {
250.               http.Error(w, err.Error(), 500)
251.               return
252.          }
253.
254.          // create update date
255.          year, month, day := time.Now().Date()
256.          updateDate   :=   strconv.Itoa(year)   +   "-"   +
   strconv.Itoa(int(month)) + "-" + strconv.Itoa(day)
257.
258.          if checkDuplicate && userCheck {
259.               // update the table
260.               statementFeature,          err          :=
   database.DB.Prepare("UPDATE  tblFeature  set  featureName=?,
   featureDescription=?,                    featureLiveDate=?,
   featureSegmentation=?,                    featureChannel=?,
   limitTransaction=?,      limitDaily=?,      limitMonthly=?,
   update_by=?, update_date=? where featureId=?;")
261.               if err == nil {
262.                    _,                  err               :=
   statementFeature.Exec(&data_input.FeatureName,
   &data_input.FeatureDescription,
   &data_input.FeatureLiveDate,
   &data_input.FeatureSegmentation,
   &data_input.FeatureChannel,  &data_input.LimitTransaction,
   &data_input.LimitDaily,  &data_input.LimitMonthly,  email,
   updateDate, id)
263.                    if err != nil {
264.                         http.Error(w,        err.Error(),
   http.StatusInternalServerError)
265.                         return
266.                    }
267.               }
268.
269.          } else {
270.               http.Error(w, err.Error(), 500)
271.               return
272.          }
273.
274.     }
275. }
276.
277. func createDataId() uuid.UUID {
278.     // create new uuid
279.     id := uuid.Must(uuid.NewRandom())
280.     return id
281. }
```

IMPORT MIDDLEWARE PACKAGES

```
1. import (
2.   "context"
3.   "fmt"
4.   "go-monolith/variable"
5.   "net/http"
6.   "strings"
```

```
7.
8.   "github.com/golang-jwt/jwt"
9.   "github.com/gorilla/mux"
10.  )
```

MIDDLEWARE

```
1. type CustomMux struct {
2.   http.ServeMux
3.   middlewares []func(next http.Handler) http.Handler
4. }
5.
6. type MyClaims struct {
7.   jwt.StandardClaims
8.   Username string `json:"username"`
9.   Email    string `json:"email"`
10.  }
11.
12.  func (c *CustomMux) RegisterMiddleware(next func(next
   http.Handler) http.Handler) {
13.       c.middlewares = append(c.middlewares, next)
14.  }
15.
16.  func (c *CustomMux) ServeHTTP(w http.ResponseWriter, r
   *http.Request) {
17.       var current http.Handler = &c.ServeMux
18.
19.       for _, next := range c.middlewares {
20.            current = next(current)
21.       }
22.
23.       current.ServeHTTP(w, r)
24.  }
25.
26.  func    MiddlewareJWTAuthorization(next    http.Handler)
   http.Handler {
27.       return http.HandlerFunc(func(w http.ResponseWriter,
   r *http.Request) {
28.
29.            params := mux.Vars(r)
30.            id, _ := params["id"]
31.
32.            pathParameter := params["parameter"]
33.
34.            authorizationHeader                          :=
   r.Header.Get("Authorization")
35.            if       !strings.Contains(authorizationHeader,
   "Bearer") {
36.                 http.Error(w,       "Invalid       Token",
   http.StatusBadRequest)
37.                 return
38.            }
39.
40.            tokenString                                  :=
   strings.Replace(authorizationHeader, "Bearer ", "", -1)
41.
```

```
42.           token, err := jwt.Parse(tokenString, func(token
   *jwt.Token) (interface{}, error) {
43.               if          method,          ok          :=
   token.Method.(*jwt.SigningMethodHMAC); !ok {
44.                   return nil, fmt.Errorf("Signing Method
   Invalid")
45.               }       else       if       method       !=
   variable.JWT_SIGNING_METHOD {
46.                   return nil, fmt.Errorf("Signing Method
   Invalid")
47.               }
48.
49.               return variable.JWT_SIGNATURE_KEY, nil
50.           })
51.
52.           if err != nil {
53.               http.Error(w,                  err.Error(),
   http.StatusBadRequest)
54.               return
55.           }
56.
57.           claims, ok := token.Claims.(jwt.MapClaims)
58.           if !ok || !token.Valid {
59.               http.Error(w,                  err.Error(),
   http.StatusBadRequest)
60.               return
61.           }
62.
63.           ctx  :=  context.WithValue(context.Background(),
   "userInfo", claims)
64.           ctx1 := context.WithValue(ctx, "id", id)
65.           ctx2    :=    context.WithValue(ctx1,    "name",
   pathParameter)
66.           r = r.WithContext(ctx2)
67.
68.           next.ServeHTTP(w, r)
69.       })
70.  }
```

IMPORT VARIABLE PACKAGES

```
1. import (
2.   "log"
3.   "os"
4.
5.   "github.com/golang-jwt/jwt"
6.   "github.com/hashicorp/go-hclog"
7.   "github.com/joho/godotenv"
8. )
```

VARIABLE PACKAGES

```
1. type M map[string]interface{}
2.
3. func init() {
```

```
4.   var err = godotenv.Load(".env")
5.   if err != nil {
6.        log.Fatalf("Error loading .env files")
7.   }
8. }
9.
10.  var Key = os.Getenv("JWT_SECRET")
11.  var Log = hclog.Default()
12.
13.  var APPLICATION_NAME = "List Features APP"
14.  var JWT_SIGNING_METHOD = jwt.SigningMethodHS256
15.  var                  JWT_SIGNATURE_KEY                    =
     []byte(os.Getenv("JWT_SIGNATURE_KEY"))
16.  var              JWT_REFRESH_SIGNATURE_KEY                =
     []byte("this_is_secret_between_two_of_us")
```

MAIN FUNCTION

```
17.  package main
18.
19.  import (
20.       "go-monolith/database"
21.       "go-monolith/routes"
22.  )
23.
24.  func main() {
25.       database.Connect_db()
26.       database.Create_table()
27.       routes.Register()
28.
29.       defer database.DB.Close()
30.  }
```

2. MICROSERVICE ARCHITECTURE

a. USER SERVICE

IMPORT MAIN PACKAGES

```
1. import (
2.   "go-kit-user/database"
3.   "go-kit-user/jwt"
4.   "go-kit-user/services"
5.   "go-kit-user/transport"
6.   "net/http"
7.   "os"
8.
9.   httptransport "github.com/go-kit/kit/transport/http"
10.       "github.com/go-kit/log"
11.       "github.com/joho/godotenv"
12.  )
```

MAIN FUNCTION

```
1. func main() {
2.
3.   logger := log.NewLogfmtLogger(os.Stderr)
4.
5.   database.Connect_db()
6.   defer database.DB.Close()
7.
8.   var usv services.UserServices
9.   usv = services.UserService{}
10.
11.       var vsc jwt.VerifyServices
12.       vsc = jwt.VerifyService{}
13.
14.       loginHandler := httptransport.NewServer(
15.            transport.MakeLoginEndpoint(usv),
16.            transport.DecodeLoginRequest,
17.            transport.EncodeResponse,
18.       )
19.
20.       registerHandler := httptransport.NewServer(
21.            transport.MakeRegisterEndpoint(usv),
22.            transport.DecodeRegisterRequest,
23.            transport.EncodeResponse,
24.       )
25.
26.       verifyHandler := httptransport.NewServer(
27.            transport.MakeVerifyEndpoint(vsc),
28.            transport.DecodeVerifyRequest,
29.            transport.EncodeResponse,
30.       )
31.
32.       // load port from env
33.       var err = godotenv.Load(".env")
34.       if err != nil {
35.            logger.Log("Error loading .env files", "error",
   err)
36.       }
37.
38.       var port = os.Getenv("SERVICE_USER_PORT")
39.
40.       http.Handle("/login", loginHandler)
41.       http.Handle("/register", registerHandler)
42.       http.Handle("/verify", verifyHandler)
43.
44.       logger.Log("msg", "HTTP", "addr", "8081")
45.       logger.Log("err", http.ListenAndServe(port, nil))
46.  }
```

IMPORT USER TRANSPORT PACKAGES

```
1. import (
2.   "bytes"
3.   "context"
4.   "encoding/json"
5.   "go-kit-user/jwt"
6.   "go-kit-user/services"
```

```
7.   "io/ioutil"
8.   "net/http"
9.
10.       "github.com/go-kit/kit/endpoint"
11.  )
```

USER TRANSPORT FUNCTION

```
1. type registerRequest struct {
2.   Username string `json:"username"`
3.   Email    string `json:"email"`
4.   Password string `json:"password"`
5. }
6.
7. type registerResponse struct {
8.   Err string `json:"err,omitempty"`
9. }
10.
11.  type loginRequest struct {
12.       Email    string `json:"email"`
13.       Password string `json:"password"`
14.  }
15.
16.  type loginResponse struct {
17.       Token string `json:"token"`
18.       Err   string `json:"err,omitempty"`
19.  }
20.
21.  type verifyRequest struct {
22.       Token string `json:"token"`
23.  }
24.
25.  type verifyResponse struct {
26.       Email string `json:"email"`
27.  }
28.
29.  // register endpoint
30.  func   MakeRegisterEndpoint(usv   services.UserServices)
   endpoint.Endpoint {
31.       return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
32.            req := request.(registerRequest)
33.            err  =  usv.Register(req.Username,  req.Email,
   req.Password)
34.            if err != nil {
35.                 return registerResponse{err.Error()}, nil
36.            }
37.            return registerResponse{}, nil
38.       }
39.  }
40.
41.  func    DecodeRegisterRequest(_    context.Context,    r
   *http.Request) (interface{}, error) {
42.       var request registerRequest
43.       if err := json.NewDecoder(r.Body).Decode(&request);
   err != nil {
```

```
44.             return nil, err
45.       }
46.
47.       return request, nil
48.  }
49.
50.  func    DecodeRegisterResponse(_    context.Context,    r
   *http.Response) (interface{}, error) {
51.       var response registerResponse
52.       if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
53.             return nil, err
54.       }
55.
56.       return response, nil
57.  }
58.
59.  // login endpoint
60.  func     MakeLoginEndpoint(usv     services.UserServices)
   endpoint.Endpoint {
61.       return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
62.             req := request.(loginRequest)
63.             v, err := usv.Login(req.Email, req.Password)
64.             if err != nil {
65.                   return registerResponse{err.Error()}, nil
66.             }
67.             return loginResponse{v, ""}, nil
68.       }
69.  }
70.
71.  func     DecodeLoginRequest(_     context.Context,     r
   *http.Request) (interface{}, error) {
72.       var request loginRequest
73.       if err := json.NewDecoder(r.Body).Decode(&request);
   err != nil {
74.             return nil, err
75.       }
76.
77.       return request, nil
78.  }
79.
80.  func    DecodeLoginResponse(_    context.Context,     r
   *http.Response) (interface{}, error) {
81.       var response loginResponse
82.       if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
83.             return nil, err
84.       }
85.
86.       return response, nil
87.  }
88.
89.  func       EncodeResponse(_       context.Context,       w
   http.ResponseWriter, response interface{}) error {
90.       return json.NewEncoder(w).Encode(response)
```

```
91.  }
92.
93.  func  EncodeRequest(_  context.Context,  r  *http.Request,
   request interface{}) error {
94.       var buf bytes.Buffer
95.       if err := json.NewEncoder(&buf).Encode(request); err
   != nil {
96.            return err
97.       }
98.       r.Body = ioutil.NopCloser(&buf)
99.       return nil
100. }
101.
102. func      MakeVerifyEndpoint(jwt      jwt.VerifyServices)
   endpoint.Endpoint {
103.      return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
104.           req := request.(verifyRequest)
105.           v := jwt.VerifyToken(req.Token)
106.           if err != nil {
107.                return registerResponse{err.Error()}, nil
108.           }
109.           return verifyResponse{v}, nil
110.      }
111. }
112.
113. func     DecodeVerifyRequest(_     context.Context,     r
   *http.Request) (interface{}, error) {
114.      var request verifyRequest
115.      if err  :=  json.NewDecoder(r.Body).Decode(&request);
   err != nil {
116.           return nil, err
117.      }
118.
119.      return request, nil
120. }
121.
122. func     DecodeVerifyResponse(_     context.Context,     r
   *http.Response) (interface{}, error) {
123.      var response verifyResponse
124.      if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
125.           return nil, err
126.      }
127.
128.      return response, nil
129. }
```

USER LOGIN FUNCTION

```
1. package services
2.
3. import (
4.   "errors"
5.   "go-kit-user/database"
6.   "go-kit-user/jwt"
```

```
  "go-kit-user/validation"
)

func (UserService) Login(email, password string) (string, error) {
      // validate the username and email
      if !validation.EmailValidation(email) {
           return "", errors.New("Wrong Email Format")
      }

      checkUser := database.QueryUser(email, password)

      if !checkUser {
           return "", errors.New("Wrong Email or passwird")
      }

      // create jwt token
      token, err := jwt.CreateToken(email)
      if err != nil {
           return "", err
      }

      return token.AccessToken, nil
}
```

USER REGISTER FUNCTION

```
package services

import (
  "errors"
  "go-kit-user/database"
  "go-kit-user/validation"

  "golang.org/x/crypto/bcrypt"
)

func (UserService) Register(username, email, password string) error {
      // create user id
      var id = createUserId()

      // validate the username and email
      if !validation.StringValidation(username, email) {
           return errors.New("Wrong Email of Username Format")
      }

      checkUser := database.CheckUser(email)

      if checkUser {
           // hash the password
           hashedPassword, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)
           if err != nil {
```

```
26.                 return err
27.           }
28.
29.           err = database.RegisterUser(id, username, email,
   hashedPassword)
30.           if err != nil {
31.                return err
32.           }
33.      }
34.      return nil
35. }
```

USER MAIN FUNCTION

```
1. package services
2.
3. import (
4.   "github.com/google/uuid"
5. )
6.
7. type UserServices interface {
8.   Login(string, string) (string, error)
9.   Register(string, string, string) error
10.  }
11.
12.  type UserService struct{}
13.
14.  type M map[string]interface{}
15.
16.  func createUserId() uuid.UUID {
17.       // create new uuid
18.       id := uuid.Must(uuid.NewRandom())
19.       return id
20.  }
```

CREATE TOKEN FUNCTION

```
1. package jwt
2.
3. import (
4.   "os"
5.   "time"
6.
7.   "github.com/golang-jwt/jwt"
8.   "github.com/google/uuid"
9.   "github.com/hashicorp/go-hclog"
10.       "github.com/joho/godotenv"
11.  )
12.
13.  type TokenDetails struct {
14.       AccessToken string
15.       AccessUuid  string
16.       AtExpires   int64
17.  }
18.
```

```
19.  type myClaims struct {
20.       jwt.StandardClaims
21.       Email string
22.  }
23.
24.  func CreateToken(email string) (*TokenDetails, error) {
25.
26.       log := hclog.Default()
27.
28.       // read env file
29.       var err = godotenv.Load(".env")
30.       if err != nil {
31.            log.Error("Error loading .env files", "error",
     err)
32.       }
33.
34.       // variable for jwt auth
35.       var APPLICATION_NAME = os.Getenv("APPLICATION_NAME")
36.       var JWT_SIGNING_METHOD = jwt.SigningMethodHS256
37.       var                JWT_SIGNATURE_KEY                =
     []byte(os.Getenv("JWT_SIGNATURE_KEY"))
38.       //var          JWT_REFRESH_SIGNATURE_KEY           =
     []byte("this_is_the_secret")
39.       var           LOGIN_EXPIRATION_DURATION           =
     time.Now().Add(time.Minute * 300).Unix()
40.
41.       // insert expiration duration and create new uuid
42.       td := &TokenDetails{
43.            AtExpires:  int64(LOGIN_EXPIRATION_DURATION),
44.            AccessUuid: uuid.New().String(),
45.       }
46.
47.       // create the claims
48.       claims := myClaims{
49.            StandardClaims: jwt.StandardClaims{
50.                 Issuer:    APPLICATION_NAME,
51.                 ExpiresAt: LOGIN_EXPIRATION_DURATION,
52.            },
53.            Email: email,
54.       }
55.
56.       // signing method
57.       token := jwt.NewWithClaims(
58.            JWT_SIGNING_METHOD, claims,
59.       )
60.
61.       // signing the token
62.       signedTokens,                err                :=
     token.SignedString(JWT_SIGNATURE_KEY)
63.       if err != nil {
64.            log.Error("Cant  signing  the  token",  "error",
     err)
65.            return nil, err
66.       }
67.
68.       td.AccessToken = signedTokens
```

```
69.
70.       return td, nil
71.  }
```

b. LIST FEATURES SERVICE

MAIN FUNCTION

```
1. package main
2.
3. import (
4.   "flag"
5.   "fmt"
6.   "go-kit-listfeatures/database"
7.   "go-kit-listfeatures/services"
8.   "go-kit-listfeatures/transport"
9.   "net/http"
10.       "os"
11.       "os/signal"
12.       "syscall"
13.
14.       "github.com/go-kit/kit/log"
15.       "github.com/joho/godotenv"
16.  )
17.
18.  func main() {
19.
20.       database.Connect_db()
21.
22.       // load port from env
23.       var err = godotenv.Load(".env")
24.       if err != nil {
25.            panic(err)
26.       }
27.
28.       var port = os.Getenv("SERVICE_USER_PORT")
29.
30.       var (
31.            httpAddr = flag.String("http.addr", port, "HTTP
   listen address")
32.       )
33.       flag.Parse()
34.
35.       var logger log.Logger
36.       {
37.            logger = log.NewLogfmtLogger(os.Stderr)
38.            logger       =       log.With(logger,       "ts",
   log.DefaultTimestampUTC)
39.            logger     =     log.With(logger,     "caller",
   log.DefaultCaller)
40.       }
41.
42.       var s services.FeatureListServices
43.       {
44.            s = services.NewInmemService()
```

```
45.       }
46.
47.       var h http.Handler
48.       {
49.            h         =         transport.MakeHTTPHandler(s,
   log.With(logger, "component", "HTTP"))
50.       }
51.
52.       errs := make(chan error)
53.       go func() {
54.            c := make(chan os.Signal, 1)
55.            signal.Notify(c,                 syscall.SIGINT,
   syscall.SIGTERM)
56.            errs <- fmt.Errorf("%s", <-c)
57.       }()
58.
59.       go func() {
60.            logger.Log("transport",     "HTTP",     "addr",
   *httpAddr)
61.            errs <- http.ListenAndServe(*httpAddr, h)
62.       }()
63.
64.       logger.Log("exit", <-errs)
65.  }
```

TRANSPORT FUNCTION

```
1. package transport
2.
3. import (
4.   "errors"
5.   "go-kit-listfeatures/endpoint"
6.   "go-kit-listfeatures/services"
7.   "net/http"
8.
9.   "github.com/go-kit/kit/log"
10.       "github.com/go-kit/kit/transport"
11.       httptransport "github.com/go-kit/kit/transport/http"
12.       "github.com/gorilla/mux"
13.  )
14.
15.  var (
16.       ErrBadRouting  =  errors.New("inconsistent  mapping
   between route and handler")
17.  )
18.
19.  func   MakeHTTPHandler(s   services.FeatureListServices,
   logger log.Logger) http.Handler {
20.       r := mux.NewRouter()
21.       e := endpoint.MakeServerEndpoints(s)
22.       options := []httptransport.ServerOption{
23.
     httptransport.ServerErrorHandler(transport.NewLogErrorHa
   ndler(logger)),
24.
     httptransport.ServerErrorEncoder(endpoint.EncodeError),
```

```
25.       }
26.
27.       r.Methods("GET").Path("/getlist").Handler(httptrans
   port.NewServer(
28.            e.GetAllListEndpoint,
29.            endpoint.DecodeGetAllListRequest,
30.            endpoint.EncodeResponse,
31.            options...,
32.       ))
33.
34.       r.Methods("GET").Path("/getlist/{id}").Handler(http
   transport.NewServer(
35.            e.GetListEndpoint,
36.            endpoint.DecodeGetListRequest,
37.            endpoint.EncodeResponse,
38.            options...,
39.       ))
40.
41.       r.Methods("POST").Path("/insertlist").Handler(httpt
   ransport.NewServer(
42.            e.PostListEndpoint,
43.            endpoint.DecodeInsertListRequest,
44.            endpoint.EncodeResponse,
45.            options...,
46.       ))
47.
48.       r.Methods("PUT").Path("/updatelist/{id}").Handler(h
   ttptransport.NewServer(
49.            e.PutListEndpoint,
50.            endpoint.DecodeUpdateListRequest,
51.            endpoint.EncodeResponse,
52.            options...,
53.       ))
54.
55.       r.Methods("DELETE").Path("/deletelist/{id}").Handle
   r(httptransport.NewServer(
56.            e.DeleteListEndpoint,
57.            endpoint.DecodeDeleteListRequest,
58.            endpoint.EncodeResponse,
59.            options...,
60.       ))
61.       return r
62.  }
```

ENDPOINT FUNCTION

```
1. package endpoint
2.
3. import (
4.   "bytes"
5.   "context"
6.   "encoding/json"
7.   "errors"
8.   "go-kit-listfeatures/middleware"
9.   "go-kit-listfeatures/services"
10.       "io/ioutil"
```

```
11.       "net/http"
12.       "net/url"
13.       "strings"
14.
15.       "github.com/go-kit/kit/endpoint"
16.       httptransport "github.com/go-kit/kit/transport/http"
17.       "github.com/gorilla/mux"
18.  )
19.
20.  var (
21.       ErrBadRouting     = errors.New("inconsistent mapping
   between route and handler")
22.       ErrInconsistentIDs = errors.New("inconsistent IDs")
23.  )
24.
25.  type Endpoints struct {
26.       GetAllListEndpoint endpoint.Endpoint
27.       GetListEndpoint    endpoint.Endpoint
28.       PostListEndpoint   endpoint.Endpoint
29.       PutListEndpoint    endpoint.Endpoint
30.       DeleteListEndpoint endpoint.Endpoint
31.  }
32.
33.  func MakeServerEndpoints(s services.FeatureListServices)
   Endpoints {
34.       return Endpoints{
35.            GetAllListEndpoint: MakeGetAllListEndpoint(s),
36.            GetListEndpoint:    MakeGetListEndpoint(s),
37.            PostListEndpoint:   MakeInsertListEndpoint(s),
38.            PutListEndpoint:    MakeUpdateListEndpoint(s),
39.            DeleteListEndpoint: MakeDeleteListEndpoint(s),
40.       }
41.  }
42.
43.  func  MakeClientEndpoints(instance  string)  (Endpoints,
   error) {
44.       if !strings.HasPrefix(instance, "http") {
45.            instance = "http://" + instance
46.       }
47.       tgt, err := url.Parse(instance)
48.       if err != nil {
49.            return Endpoints{}, err
50.       }
51.       tgt.Path = ""
52.
53.       options := []httptransport.ClientOption{}
54.
55.       return Endpoints{
56.            GetAllListEndpoint:
   httptransport.NewClient("GET",                          tgt,
   EncodeGetListAllRequest,         DecodeGetAllListResponse,
   options...).Endpoint(),
57.            GetListEndpoint:
   httptransport.NewClient("GET",  tgt,  EncodeGetListRequest,
   DecodeGetListResponse, options...).Endpoint(),
```

```
58.           PostListEndpoint:
   httptransport.NewClient("POST",                          tgt,
   EncodeInsertListRequest,          DecodeInsertListResponse,
   options...).Endpoint(),
59.           PutListEndpoint:
   httptransport.NewClient("PUT",                           tgt,
   EncodeUpdateListRequest,          DecodeUpdateListResponse,
   options...).Endpoint(),
60.           DeleteListEndpoint:
   httptransport.NewClient("DELETE",                        tgt,
   EncodeDeleteListRequest,          DecodeDeleteListResponse,
   options...).Endpoint(),
61.      }, nil
62.  }
63.
64.  // get all list req resp
65.  type getAllListRequest struct{}
66.
67.  type getAllListResponse struct {
68.      List []*services.ListFeatures
69.      Err  error `json:"err,omitempty"`
70.  }
71.
72.  func (r getAllListResponse) error() error { return r.Err
   }
73.
74.  // get list req and delete req resp
75.  type IdRequest struct {
76.      Id string
77.  }
78.
79.  type getListResponse struct {
80.      List services.ListFeatures
81.      Err  error `json:"err,omitempty"`
82.  }
83.
84.  func (r getListResponse) error() error { return r.Err }
85.
86.  // insert update req resp
87.  type insertListRequest struct {
88.      Email              string `json:"email"`
89.      FeatureName        string `json:"featureName"`
90.      FeatureDescription                          string
   `json:"featureDescription"`
91.      FeatureLiveDate    string `json:"featureLiveDate"`
92.      FeatureSegmentation                         string
   `json:"featureSegmentation"`
93.      FeatureChannel     string `json:"featureChannel"`
94.      LimitTransaction   string `json:"limitTransaction"`
95.      LimitDaily         string `json:"limitDaily"`
96.      LimitMonthly       string `json:"limitMonthly"`
97.  }
98.
99.  type updateListRequest struct {
100.     Email              string `json:"email"`
101.     FeatureId          string `json:"featureid"`
```

```
102.      FeatureName          string `json:"featureName"`
103.      FeatureDescription                              string
   `json:"featureDescription"`
104.      FeatureLiveDate      string `json:"featureLiveDate"`
105.      FeatureSegmentation                             string
   `json:"featureSegmentation"`
106.      FeatureChannel       string `json:"featureChannel"`
107.      LimitTransaction     string `json:"limitTransaction"`
108.      LimitDaily           string `json:"limitDaily"`
109.      LimitMonthly         string `json:"limitMonthly"`
110. }
111.
112. type insertUpdateListResponse struct {
113.      Err error `json:"err,omitempty"`
114. }
115.
116. func (r insertUpdateListResponse) error() error { return
   r.Err }
117.
118. // delete resp
119. type deleteListResponse struct {
120.      Err error `json:"err,omitempty"`
121. }
122.
123. func (r deleteListResponse) error() error { return r.Err
   }
124.
125. // Get All List implements service
126. func   (e   Endpoints)   GetAllList(ctx   context.Context)
   ([]*services.ListFeatures, error) {
127.      request := getAllListRequest{}
128.      response, err := e.GetAllListEndpoint(ctx, request)
129.      if err != nil {
130.           return []*services.ListFeatures{}, err
131.      }
132.      resp := response.(getAllListResponse)
133.      return resp.List, resp.Err
134. }
135.
136. // Get List implements service
137. func (e Endpoints) GetList(ctx context.Context, id string)
   (services.ListFeatures, error) {
138.      request := IdRequest{Id: id}
139.      response, err := e.GetListEndpoint(ctx, request)
140.      if err != nil {
141.           return services.ListFeatures{}, err
142.      }
143.      resp := response.(getListResponse)
144.      return resp.List, resp.Err
145. }
146.
147. // Post List implements service
148. func   (e   Endpoints)   PostList(ctx   context.Context,
   featureName,      featureDescription,      featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily, limitMonthly string) error {
```

```
149.      request := insertListRequest{
150.           FeatureName:          featureName,
151.           FeatureDescription:   featureDescription,
152.           FeatureLiveDate:      featureLiveDate,
153.           FeatureSegmentation: featureSegmentation,
154.           FeatureChannel:       featureChannel,
155.           LimitTransaction:     limitTransaction,
156.           LimitDaily:           limitDaily,
157.           LimitMonthly:         limitMonthly,
158.      }
159.      response, err := e.PostListEndpoint(ctx, request)
160.      if err != nil {
161.           return err
162.      }
163.      resp := response.(insertUpdateListResponse)
164.      return resp.Err
165. }
166.
167. // Put List implements service
168. func (e Endpoints) PutList(ctx context.Context, id,
     featureName, featureDescription, featureLiveDate,
     featureSegmentation, featureChannel, limitTransaction,
     limitDaily, limitMonthly string) error {
169.      request := updateListRequest{
170.           FeatureId:            id,
171.           FeatureName:          featureName,
172.           FeatureDescription:   featureDescription,
173.           FeatureLiveDate:      featureLiveDate,
174.           FeatureSegmentation: featureSegmentation,
175.           FeatureChannel:       featureChannel,
176.           LimitTransaction:     limitTransaction,
177.           LimitDaily:           limitDaily,
178.           LimitMonthly:         limitMonthly,
179.      }
180.      response, err := e.PutListEndpoint(ctx, request)
181.      if err != nil {
182.           return err
183.      }
184.      resp := response.(insertUpdateListResponse)
185.      return resp.Err
186. }
187.
188. // Delete List Endpoint
189. func (e Endpoints) DeleteList(ctx context.Context, id
     string) error {
190.      request := IdRequest{Id: id}
191.      response, err := e.DeleteListEndpoint(ctx, request)
192.      if err != nil {
193.           return err
194.      }
195.      resp := response.(deleteListResponse)
196.      return resp.Err
197. }
198.
199. // Get All List Endpoint
```

```
200. func                              MakeGetAllListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
201.      return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
202.           _ = request.(getAllListRequest)
203.           v, e := lvs.GetAllList(ctx)
204.           return getAllListResponse{List: v, Err: e}, nil
205.      }
206. }
207.
208. func    DecodeGetAllListRequest(_    context.Context,    r
   *http.Request) (interface{}, error) {
209.      email, err := middleware.VerifyToken(r)
210.      if err != nil {
211.           return nil, err
212.      }
213.      if email == "" {
214.           return nil, err
215.      }
216.      var request getAllListRequest
217.      if  err  :=  json.NewDecoder(r.Body).Decode(&request);
   err != nil {
218.           return nil, err
219.      }
220.
221.      return request, nil
222. }
223.
224. func  EncodeGetListAllRequest(ctx  context.Context,  req
   *http.Request, request interface{}) error {
225.      req.URL.Path = "/getlist/"
226.      return encodeRequest(ctx, req, request)
227. }
228.
229. func   DecodeGetAllListResponse(_   context.Context,   r
   *http.Response) (interface{}, error) {
230.      var response getAllListResponse
231.      if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
232.           return nil, err
233.      }
234.
235.      return response, nil
236. }
237.
238. // Get List Endpoint
239. func                                MakeGetListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
240.      return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
241.           req := request.(IdRequest)
242.           v, e := lvs.GetList(ctx, req.Id)
243.           return getListResponse{List: v, Err: e}, nil
244.      }
245. }
246.
```

```
247. func     DecodeGetListRequest(_     context.Context,     r
   *http.Request) (interface{}, error) {
248.      email, err := middleware.VerifyToken(r)
249.      if err != nil {
250.           return nil, err
251.      }
252.      if email == "" {
253.           return nil, err
254.      }
255.      var request IdRequest
256.      vars := mux.Vars(r)
257.      id, ok := vars["id"]
258.      if !ok {
259.           return nil, ErrBadRouting
260.      }
261.      request = IdRequest{Id: id}
262.      if err := json.NewDecoder(r.Body).Decode(&request);
   err != nil {
263.           return nil, err
264.      }
265.
266.      return request, nil
267. }
268.
269. func   EncodeGetListRequest(ctx   context.Context,   req
   *http.Request, request interface{}) error {
270.      r := request.(IdRequest)
271.      id := url.QueryEscape(r.Id)
272.      req.URL.Path = "/getlistbyid/" + id
273.      return encodeRequest(ctx, req, request)
274. }
275.
276. func    DecodeGetListResponse(_    context.Context,    r
   *http.Response) (interface{}, error) {
277.      var response getListResponse
278.      if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
279.           return nil, err
280.      }
281.
282.      return response, nil
283. }
284.
285. // Insert List Endpoint
286. func                          MakeInsertListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
287.      return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
288.           req := request.(insertListRequest)
289.           e     :=     lvs.InsertList(ctx,     req.Email,
   req.FeatureName,                    req.FeatureDescription,
   req.FeatureLiveDate,                req.FeatureSegmentation,
   req.FeatureChannel,  req.LimitTransaction,  req.LimitDaily,
   req.LimitMonthly)
290.           return insertUpdateListResponse{Err: e}, nil
291.      }
```

```
292. }
293.
294. func   DecodeInsertListRequest(_   context.Context,   r
   *http.Request) (interface{}, error) {
295.      email, err := middleware.VerifyToken(r)
296.      if err != nil {
297.           return nil, err
298.      }
299.      if email == "" {
300.           return nil, err
301.      }
302.      var request insertListRequest
303.      if err := json.NewDecoder(r.Body).Decode(&request);
   err != nil {
304.           return nil, err
305.      }
306.
307.      request.Email = email
308.
309.      return request, nil
310. }
311.
312. func  EncodeInsertListRequest(ctx  context.Context,  req
   *http.Request, request interface{}) error {
313.      req.URL.Path = "/insertlist/"
314.      return encodeRequest(ctx, req, request)
315. }
316.
317. func   DecodeInsertListResponse(_   context.Context,   r
   *http.Response) (interface{}, error) {
318.      var response insertUpdateListResponse
319.      if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
320.           return nil, err
321.      }
322.
323.      return response, nil
324. }
325.
326. // Update List Endpoint
327. func                          MakeUpdateListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
328.      return     func(ctx    context.Context,     request
   interface{}) (response interface{}, err error) {
329.           req := request.(updateListRequest)
330.           e     :=     lvs.UpdateList(ctx,     req.Email,
   req.FeatureId,   req.FeatureName,   req.FeatureDescription,
   req.FeatureLiveDate,              req.FeatureSegmentation,
   req.FeatureChannel,  req.LimitTransaction,  req.LimitDaily,
   req.LimitMonthly)
331.           return insertUpdateListResponse{Err: e}, nil
332.      }
333. }
334.
335. func   DecodeUpdateListRequest(_   context.Context,   r
   *http.Request) (request interface{}, err error) {
```

```
336.      email, err := middleware.VerifyToken(r)
337.      if err != nil {
338.           return nil, err
339.      }
340.      if email == "" {
341.           return nil, err
342.      }
343.      vars := mux.Vars(r)
344.      id, ok := vars["id"]
345.      if !ok {
346.           return nil, ErrBadRouting
347.      }
348.      var req updateListRequest
349.      if err := json.NewDecoder(r.Body).Decode(&req); err
   != nil {
350.           return nil, err
351.      }
352.      return updateListRequest{
353.           Email:              email,
354.           FeatureId:          id,
355.           FeatureName:        req.FeatureName,
356.           FeatureDescription: req.FeatureDescription,
357.           FeatureLiveDate:    req.FeatureLiveDate,
358.           FeatureSegmentation: req.FeatureSegmentation,
359.           FeatureChannel:     req.FeatureChannel,
360.           LimitTransaction:   req.LimitTransaction,
361.           LimitDaily:         req.LimitDaily,
362.           LimitMonthly:       req.LimitMonthly,
363.      }, nil
364. }
365.
366. func  EncodeUpdateListRequest(ctx  context.Context,  req
   *http.Request, request interface{}) error {
367.      r := request.(updateListRequest)
368.      id := url.QueryEscape(r.FeatureId)
369.      req.URL.Path = "/updatelist/" + id
370.      return encodeRequest(ctx, req, request)
371. }
372.
373. func   DecodeUpdateListResponse(_   context.Context,   r
   *http.Response) (interface{}, error) {
374.      var response insertUpdateListResponse
375.      if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
376.           return nil, err
377.      }
378.
379.      return response, nil
380. }
381.
382. // Delete List Endpoint
383. func                          MakeDeleteListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
384.      return     func(ctx     context.Context,     request
   interface{}) (response interface{}, err error) {
385.           req := request.(IdRequest)
```

```
386.          e := lvs.DeleteList(ctx, req.Id)
387.          return deleteListResponse{Err: e}, nil
388.     }
389. }
390.
391. func   DecodeDeleteListRequest(_   context.Context,   r
   *http.Request) (request interface{}, err error) {
392.     email, err := middleware.VerifyToken(r)
393.     if err != nil {
394.          return nil, err
395.     }
396.     if email == "" {
397.          return nil, err
398.     }
399.     vars := mux.Vars(r)
400.     id, ok := vars["id"]
401.     if !ok {
402.          return nil, ErrBadRouting
403.     }
404.     return IdRequest{Id: id}, nil
405. }
406.
407. func  EncodeDeleteListRequest(ctx  context.Context,  req
   *http.Request, request interface{}) error {
408.     r := request.(IdRequest)
409.     id := url.QueryEscape(r.Id)
410.     req.URL.Path = "/deletelist/" + id
411.     return encodeRequest(ctx, req, request)
412. }
413.
414. func   DecodeDeleteListResponse(_   context.Context,   r
   *http.Response) (interface{}, error) {
415.     var response deleteListResponse
416.     if err := json.NewDecoder(r.Body).Decode(&response);
   err != nil {
417.          return nil, err
418.     }
419.
420.     return response, nil
421. }
422.
423. type errorer interface {
424.     error() error
425. }
426.
427. func      EncodeResponse(ctx      context.Context,      w
   http.ResponseWriter, response interface{}) error {
428.     if e, ok := response.(errorer); ok && e.error() !=
   nil {
429.          // Not a Go kit transport error, but a business-
   logic error.
430.          // Provide those as HTTP errors.
431.          EncodeError(ctx, e.error(), w)
432.          return nil
433.     }
```

```
434.      w.Header().Set("Content-Type",    "application/json;
   charset=utf-8")
435.      return json.NewEncoder(w).Encode(response)
436. }
437.
438. func encodeRequest(_ context.Context, req *http.Request,
   request interface{}) error {
439.      var buf bytes.Buffer
440.      err := json.NewEncoder(&buf).Encode(request)
441.      if err != nil {
442.           return err
443.      }
444.      req.Body = ioutil.NopCloser(&buf)
445.      return nil
446. }
447.
448. func   EncodeError(_   context.Context,   err   error,   w
   http.ResponseWriter) {
449.      if err == nil {
450.           panic("encodeError with nil error")
451.      }
452.      w.Header().Set("Content-Type",    "application/json;
   charset=utf-8")
453.      w.WriteHeader(codeFrom(err))
454.      json.NewEncoder(w).Encode(map[string]interface{}{
455.           "error": err.Error(),
456.      })
457. }
458.
459. func codeFrom(err error) int {
460.      switch err {
461.      case services.ErrNotFound:
462.           return http.StatusNotFound
463.      case services.ErrAlreadyExists, ErrInconsistentIDs:
464.           return http.StatusBadRequest
465.      default:
466.           return http.StatusInternalServerError
467.      }
468. }
```

INSERT FUNCTION

```
1. package services
2.
3. import (
4.   "context"
5.   "go-kit-listfeatures/database"
6. )
7.
8. func  (s  *inmemService)  InsertList(ctx  context.Context,
   email,  featureName,  featureDescription,  featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily, limitMonthly string) error {
9.
10.      s.mtx.Lock()
11.      defer s.mtx.Unlock()
```

```
12.
13.       // create uuid
14.       id := createDataId()
15.       idString := id.String()
16.
17.       // check duplication of data by name, segmentation,
   and channel
18.       checkDuplicate,                err                :=
   database.CheckDuplication(featureName,
   featureSegmentation, featureChannel)
19.       if err != nil {
20.            return ErrAlreadyExists
21.       }
22.
23.       if checkDuplicate {
24.            insertDatabase,              err              :=
   database.DB.Prepare("INSERT        INTO        tblFeature
   (featureId,featureName,featureDescription,featureLiveDate,
   featureSegmentation,featureChannel,limitTransaction,limitD
   aily,limitMonthly,created_by)
   values(?,?,?,?,?,?,?,?,?,?);")
25.            if err == nil {
26.                 _,  err  :=  insertDatabase.Exec(idString,
   featureName,      featureDescription,      featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily, limitMonthly, email)
27.                 if err != nil {
28.                      return err
29.                 }
30.            }
31.       }
32.
33.       return nil
34.  }
```

GET LIST FUNCTION

```
1. package services
2.
3. import (
4.   "context"
5.   "database/sql"
6.   "go-kit-listfeatures/database"
7. )
8.
9. func  (s  *inmemService)  GetAllList(ctx  context.Context)
   ([]*ListFeatures, error) {
10.       s.mtx.RLock()
11.       defer s.mtx.RUnlock()
12.
13.       rows,  err  :=  database.DB.Query("select  featureId,
   featureName,      featureDescription,      featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily, limitMonthly from tblFeature;")
14.       if err != nil {
15.            return nil, err
```

```
16.       }
17.       defer rows.Close()
18.
19.       var featureList []*ListFeatures
20.       for rows.Next() {
21.            var each = ListFeatures{}
22.            err        =        rows.Scan(&each.FeatureId,
   &each.FeatureName,               &each.FeatureDescription,
   &each.FeatureLiveDate,          &each.FeatureSegmentation,
   &each.FeatureChannel,              &each.LimitTransaction,
   &each.LimitMonthly, &each.LimitDaily)
23.            if err != nil {
24.                 return []*ListFeatures{}, ErrNotFound
25.            }
26.
27.            featureList = append(featureList, &each)
28.       }
29.
30.       return featureList, nil
31.  }
32.
33.  func (s *inmemService) GetList(ctx context.Context, id
   string) (ListFeatures, error) {
34.       s.mtx.RLock()
35.       defer s.mtx.RUnlock()
36.
37.       var each ListFeatures
38.       err   :=   database.DB.QueryRow("select   featureId,
   featureName,     featureDescription,     featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily,   limitMonthly    from    tblFeature    where
   featureId=?;",                  id).Scan(&each.FeatureId,
   &each.FeatureName,              &each.FeatureDescription,
   &each.FeatureLiveDate,          &each.FeatureSegmentation,
   &each.FeatureChannel,              &each.LimitTransaction,
   &each.LimitMonthly, &each.LimitDaily)
39.       if err == sql.ErrNoRows {
40.            return ListFeatures{}, ErrNotFound
41.       }
42.       return each, nil
43.  }
```

UPDATE FUNCTION

```
1. package services
2.
3. import (
4.   "context"
5.   "go-kit-listfeatures/database"
6.   "strconv"
7.   "time"
8. )
9.
10.  func (s *inmemService) UpdateList(ctx context.Context,
   email,      id,      featureName,      featureDescription,
```

```
   featureLiveDate,    featureSegmentation,    featureChannel,
   limitTransaction, limitDaily, limitMonthly string) error {
11.
12.       s.mtx.Lock()
13.       defer s.mtx.Unlock()
14.
15.       // check if there is a duplicate
16.       duplicate,                    err                    :=
   database.UpdateQueryDataFeature(id)
17.       if err != nil {
18.            return ErrNotFound
19.       }
20.
21.       // create update date variable
22.       year, month, day := time.Now().Date()
23.       updateDate    :=    strconv.Itoa(year)    +    "-"    +
   strconv.Itoa(int(month)) + "-" + strconv.Itoa(day)
24.
25.       if duplicate {
26.            // insert to database
27.            insertDatabase,              err              :=
   database.DB.Prepare("UPDATE  tblFeature  set  featureName=?,
   featureDescription=?,                  featureLiveDate=?,
   featureSegmentation=?,                   featureChannel=?,
   limitTransaction=?,     limitDaily=?,     limitMonthly=?,
   update_by=?, update_date=? where featureId=?;")
28.            if err == nil {
29.                 _, err := insertDatabase.Exec(featureName,
   featureDescription,  featureLiveDate,  featureSegmentation,
   featureChannel, limitTransaction, limitDaily, limitMonthly,
   email, updateDate, id)
30.                 if err != nil {
31.                      return err
32.                 }
33.            }
34.       }
35.       return nil
36.  }
```

DELETE FUNCTION

```
1. package services
2.
3. import (
4.   "context"
5.   "go-kit-listfeatures/database"
6. )
7.
8. func (s *inmemService) DeleteList(ctx context.Context, id
   string) error {
9.   s.mtx.Lock()
10.       defer s.mtx.Unlock()
11.       delete,  err  :=  database.DB.Prepare("DELETE  FROM
   tblfeature where featureId = ?;")
12.       if err == nil {
13.            _, err := delete.Exec(id)
```

```
14.           if err != nil {
15.                return err
16.           }
17.      }
18.
19.      return nil
20. }
```

FEATURE LIST MAIN FUNCTION

```
1. package services
2.
3. import (
4.   "context"
5.   "errors"
6.   "sync"
7.
8.   "github.com/google/uuid"
9. )
10.
11.  type FeatureListServices interface {
12.      GetAllList(context.Context) ([]*ListFeatures, error)
13.      GetList(context.Context,   string)   (ListFeatures,
   error)
14.      InsertList(context.Context, string, string, string,
   string, string, string, string, string, string) error
15.      UpdateList(context.Context, string, string, string,
   string, string, string, string, string, string)
   error
16.      DeleteList(context.Context, string) error
17.  }
18.
19.  type FeatureListService struct{}
20.
21.  type ListFeatures struct {
22.      FeatureId          string `json: "featureId"`
23.      FeatureName        string `json:"featureName"`
24.      FeatureDescription                              string
   `json:"featureDescription"`
25.      FeatureLiveDate    string `json:"featureLiveDate"`
26.      FeatureSegmentation                             string
   `json:"featureSegmentation"`
27.      FeatureChannel     string `json:"featureChannel"`
28.      LimitTransaction   string `json:"limitTransaction"`
29.      LimitDaily         string `json:"limitDaily"`
30.      LimitMonthly       string `json:"limitMonthly"`
31.  }
32.
33.  func createDataId() uuid.UUID {
34.      // create new uuid
35.      id := uuid.Must(uuid.NewRandom())
36.      return id
37.  }
38.
39.  var (
40.      ErrAlreadyExists = errors.New("already exists")
```

```
41.       ErrNotFound       = errors.New("not found")
42.       ErrEmpty          = errors.New("duplicate data")
43.       ErrFailed         = errors.New("acction failed")
44.  )
45.
46.  type inmemService struct {
47.       mtx sync.RWMutex
48.       m   map[string]ListFeatures
49.  }
50.
51.  func NewInmemService() FeatureListServices {
52.       return &inmemService{
53.            m: map[string]ListFeatures{},
54.       }
55.  }
```

MIDDLEWARE FUNCTION

```
1. package middleware
2.
3. import (
4.   "context"
5.   "errors"
6.   "fmt"
7.   "net/http"
8.   "os"
9.   "strings"
10.
11.       "github.com/golang-jwt/jwt"
12.       "github.com/joho/godotenv"
13.  )
14.
15.  type CustomMux struct {
16.       http.ServeMux
17.       middlewares []func(next http.Handler) http.Handler
18.  }
19.
20.  type MyClaims struct {
21.       jwt.StandardClaims
22.       Email string `json:"email"`
23.  }
24.
25.  func JwtMiddleware(next http.Handler) http.Handler {
26.       return http.HandlerFunc(func(w http.ResponseWriter,
   r *http.Request) {
27.
28.            authorizationHeader                    :=
   r.Header.Get("Authorization")
29.            if       !strings.Contains(authorizationHeader,
   "Bearer") {
30.                 http.Error(w,       "Invalid       Token",
   http.StatusBadRequest)
31.                 return
32.            }
33.
34.            key := os.Getenv("JWT_SIGNATURE_KEY")
```

```
35.
36.           tokenString                                        :=
   strings.Replace(authorizationHeader, "Bearer ", "", -1)
37.
38.           token, err := jwt.Parse(tokenString, func(token
   *jwt.Token) (interface{}, error) {
39.                if          method,          ok          :=
   token.Method.(*jwt.SigningMethodHMAC); !ok {
40.                     return nil, fmt.Errorf("Signing Method
   Invalid")
41.                } else if method != jwt.SigningMethodHS256
   {
42.                     return nil, fmt.Errorf("Signing Method
   Invalid")
43.                }
44.
45.                return []byte(key), nil
46.           })
47.
48.           if err != nil {
49.                http.Error(w,                    err.Error(),
   http.StatusBadRequest)
50.                return
51.           }
52.
53.           claims, ok := token.Claims.(jwt.MapClaims)
54.           if !ok || !token.Valid {
55.                http.Error(w,                    err.Error(),
   http.StatusBadRequest)
56.                return
57.           }
58.
59.           ctx  :=  context.WithValue(context.Background(),
   "userInfo", claims)
60.           r = r.WithContext(ctx)
61.
62.           next.ServeHTTP(w, r)
63.      })
64.  }
65.
66.  func VerifyToken(r *http.Request) (string, error) {
67.
68.      authorizationHeader := r.Header.Get("Authorization")
69.      if  !strings.Contains(authorizationHeader,  "Bearer")
   {
70.           return "", ErrorToken
71.      }
72.      // read env file
73.      var err = godotenv.Load(".env")
74.      if err != nil {
75.           panic(err)
76.      }
77.
78.      key := os.Getenv("JWT_SIGNATURE_KEY")
79.
```

```
80.       tokenString := strings.Replace(authorizationHeader,
   "Bearer ", "", -1)
81.
82.       token, err := jwt.Parse(tokenString, func(token
   *jwt.Token) (interface{}, error) {
83.           if method, ok := 
   token.Method.(*jwt.SigningMethodHMAC); !ok {
84.               return nil, fmt.Errorf("Signing Method
   Invalid")
85.           } else if method != jwt.SigningMethodHS256 {
86.               return nil, fmt.Errorf("Signing Method
   Invalid")
87.           }
88.
89.           return []byte(key), nil
90.       })
91.
92.       if err != nil {
93.           return "", ErrorToken
94.       }
95.
96.       claims, ok := token.Claims.(jwt.MapClaims)
97.       if !ok || !token.Valid {
98.           return "", ErrorToken
99.       }
100.
101.      userInfo := claims["Email"]
102.      userInfoString := userInfo.(string)
103.
104.      return userInfoString, nil
105. }
106.
107. var (
108.      ErrorToken = errors.New("invalid token")
109. )
```

Similarity Report

PAPER NAME

TA-18.K1.0014.docx

WORD COUNT

12273 Words

CHARACTER COUNT

68749 Characters

PAGE COUNT

32 Pages

FILE SIZE

418.7KB

SUBMISSION DATE

Jan 9, 2023 10:54 AM GMT+7

REPORT DATE

Jan 9, 2023 10:54 AM GMT+7

10% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

- 7% Internet database
- Crossref database
- 5% Submitted Works database
- 6% Publications database
- Crossref Posted Content database