CHAPTER 5
IMPLEMENTATION AND RESULTS

5.1. Implementation

5.1.1. Monolith Architecture

The application needs a database to store the user data and the list of application data, in the monolith application all services need to connect to the database and the database that is used is using MySQL. All services in the monolith application will be separated through packages, including the database function that are in database packages, the function in the packages consists of creating the connection to connect the database and creating tables.

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

The code above is used to connect to the database that located in database packages, variable DB is created in line 1 and will have *sql.DB data type, line 4-11 will be used to read database configuration from the env file using package os and store the configuration to variable. To open the connection to database line 13 will call the sql packages and use the database configuration in a variable that was previously loaded from the env file.

```
1. func Create_table() {
2.   _, 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); ")
3.   if errFeature != nil {
4.        panic(errFeature)
5.   }
6.
7.   _, 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));")
8.   if errUser != nil {
9.        panic(errUser)
10.       }
11.  }
```

After connecting to the database, the application will automatically create the tables for both the user data and the list feature data. The code above is a function to create tables that are located in database packages along with connecting to the database function. Line 2 and 7 is used to create a table for list features data and user data if the tables does not exist, if the tables exist then the SQL query is not executed. Line 3 and line 8 is used to check if there is an error in the query or not.

```
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.  }
```

After the database is connected and the table is created with no problem, the next thing to do is to create a route to access the user service and list features data service. The code above is a function to register the route and to start the HTTP server. Line 4 is used to initiate new routers using gorilla packages, line 5-8 is used to create subrouter to a specific HTTP method such as GET, POST, PUT and DELETE. Line 12-13 is used to register the user login and register service using HTTP method POST, the user service is not using the subrouter variable because the user service is not using middleware. Line 17-27 is used to register the list features data services using the subrouter variables, and in the line 30-33 is used to use the middleware to authenticate the JWT token. Line 35-40 is used to serve the http server in localhost:8080.

```
1. var id = createUserId()
2. if !validation.StringValidation(user.Username, user.Email)
   {
3.   http.Error(w, err.Error(), http.StatusBadRequest)
4.   return
5. }
6. var users = QueryUser(user.Email)
7. if (data_register_user{}) == users {
8.                hashedPassword,          err           :=
   bcrypt.GenerateFromPassword([]byte(user.Password),
   bcrypt.DefaultCost)
9. if len(hashedPassword) != 0 && checkErr(w, r, err) {
10.   stmt,err:=database.DB.Prepare("INSERT    INTO    tblUser
   (user_id,username,user_email,user_password)
   values(?,?,?,?);")
11.  if err == nil {
12.       _, err := stmt.Exec(&id, &user.Username, &user.Email,
   &hashedPassword)
13.            }
14.       }
15.  }
```

The user services consists of 2 services, namely the register user services and user login services. The code above is the main function of register user services, line 1 is used to call createUserId that will create an id for a user using the uuid package and store it to the id variable. Line 2-5 is used to verify the username and email. Line 6 is used to verify if the user already exists or not, by calling the QuerryUser function and if it does not return empty then the user already exists. Line 7 is used to verify if the user is already exists or not, if user not exists the code will execute line 8 where the password will hashed using bcrypt password if its success the code will execute line 10 where database query will be prepared and if its success it will executed and inserted in database in line 12.

```
1. users := QueryUser(data_login.Email)
2. if (data_register_user{}) == users {
3.   fmt.Print(users)
4.   http.Error(w, err.Error(), http.StatusBadRequest)
5.   return
6. }
7. var                    password_tes                    =
   bcrypt.CompareHashAndPassword([]byte(users.Password),
   []byte(data_login.Password))
8. if password_tes == nil {
9.   token, err := CreateToken(&users)
10.       if err != nil {
11.           http.Error(w, err.Error(), 500)
12.       }
13.       tokenString, _ := json.Marshal(variable.M{"token":
   token.AccessToken})
14.       w.Write([]byte(tokenString))
15.  }
```

The code above is the main part of the user login services, same as registering in line 1 it will check if the user exists or not, if the user does not exist line 2-6 will be executed and will return an error. Line is used to compare the hashed password that user input with the hashed password that is already stored in the database, if password matches then line 8-15 will be executed. Line 9 will call CreateToken function that will create the JWT token, if there is no error the token will be turned into []byte data type in line 13 and will return a JWT response.

```
1. checkDuplicate,                  err                  :=
   checkDuplication(data_input.FeatureName,
   data_input.FeatureSegmentation, data_input.FeatureChannel)
2. userCheck, err := checkUser(email.(string))
3. if checkDuplicate && userCheck {
4. // inserting to database
5. statementFeature, err := database.DB.Prepare("INSERT INTO
   tblFeature
   (featureId,featureName,featureDescription,featureLiveDate,
```

```
   featureSegmentation,featureChannel,limitTransaction,limitD
   aily,limitMonthly,created_by)
   values(?,?,?,?,?,?,?,?,?,?);")
6. if err == nil {
7.   _,        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)
8. }
```

The code above is the main part of the insert list features data function, line 1 will check if there is a data with the same name, segmentation and channel, if there is a duplicate the function will return an error, line 2 will check whether the user is already registered or not, the email will get from the JWT token. Line 3 will check if there is no duplicate and the user is eligible, and will execute line 5 where the database will be prepared to insert the data, if there is no error in line 6, the program will execute the query in line 7.

```
1. rows,   err   :=   database.DB.Query("select   featureId,
   featureName,     featureDescription,     featureLiveDate,
   featureSegmentation,   featureChannel,   limitTransaction,
   limitDaily, limitMonthly from tblFeature;")
2. defer rows.Close()
3. var list_feature []data_view
4. for rows.Next() {
5.   var each = data_view{}
6.   var  err  =  rows.Scan(&each.FeatureId,  &each.FeatureName,
   &each.FeatureDescription,          &each.FeatureLiveDate,
   &each.FeatureSegmentation,          &each.FeatureChannel,
   &each.LimitTransaction,              &each.LimitDaily,
   &each.LimitMonthly)
7.   list_feature = append(list_feature, each)
8. }
9. if err = rows.Err(); err != nil {
10.       fmt.Println(err.Error())
11.       return
12.  }
13.  output, err := json.Marshal(list_feature)
14.  w.Header().Set("content-type", "application/json")
15.  w.Write(output)
```

The code above is the main part of the view all list features data function, line 1 is used to query the database to get the list of the list features data. Line 3-8 will loop the list and append it to list, if there is no error when looping, line 13-15 will be executed and will return a response of the list of list features data.

```
1. id := r.Context().Value("id")
2. var list_feature data_view
```

```
3. 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)
4. if dberr == sql.ErrNoRows {
5.   fmt.Print(dberr)
6.   return
7. }
8. output, err := json.Marshal(list_feature)
9. if err != nil {
10.       http.Error(w, err.Error(), 500)
11.       return
12.  }
13.  w.Header().Set("content-type", "application/json")
14.  w.Write(output)
```

The code above is the main part of the view list features by id, line 1 is used to get id from context that will be used to get the data. Line 3 is used to query the data using id that previously get from line 1 and copy the result to listfeature variable that created in line 2. Line 4-7 is used to check if there is any error in query process, if there is no error in process the listfeature variable will be encoded to json in line 8 and will be checked if there is an error in line 9-12. Line 13–14 will write the result in header with application/json content type.

```
1. func DeleteData(w http.ResponseWriter, r *http.Request) {
2.   id := r.Context().Value("id")
3.   var checkData int
4.   err := database.DB.QueryRow("SELECT count(*) FROM
   tblFeature where featureId=?;", id).Scan(&checkData)
5.   if checkData == 0 {
6.       http.Error(w, err.Error(),
   http.StatusInternalServerError)
7.       fmt.Println(err.Error())
8.       return
9.   }
10.      delete, err := database.DB.Prepare("DELETE FROM
   tblfeature where featureId=?;")
11.      if err == nil {
12.           _, err := delete.Exec(id)
13.           if err != nil {
14.                http.Error(w, err.Error(),
   http.StatusInternalServerError)
15.                return
16.           }
17.      }
```

```
18.  }
```

The code above is the main part of delete list features by id, in line 2 the id is obtained from context. Line 3-9 will check if there is a list feature with the same id that was already obtained in line 2. After checking the list line 10-17 will execute the delete function and if there is no error the list will be deleted based on id.

```
1. func UpdateData(w http.ResponseWriter, r *http.Request) {
2.   userInfo := r.Context().Value("userInfo").(jwt.MapClaims)
3.   var email = userInfo["email"]
4.   id := r.Context().Value("id")
5.   b, err := ioutil.ReadAll(r.Body)
6.   var data_input data
7.   err = json.Unmarshal(b, &data_input)
8.   userCheck, err := checkUser(email.(string))
9.   if  userCheck {
10.          statementFeature,          err          :=
   database.DB.Prepare("UPDATE  tblFeature  set  featureName=?,
   featureDescription=?,                  featureLiveDate=?,
   featureSegmentation=?,                  featureChannel=?,
   limitTransaction=?,     limitDaily=?,     limitMonthly=?,
   update_by=?, update_date=? where featureId=?;")
11.          if err == nil {
12.                _,                  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)
13.  }
```

The code above is the main part of the update function that will update the list based on id. Line 2 and 3 will get user email from the JSON web token which was sent together with the body. Line 4 will get the id from context that will be used for searching the list that wants to be updated. Line 5-7 will receive the body and unmarshal it as the body is a JSON data type. Line 8 will check if the user is valid or not, if the user is valid then it will pass the if statement in line 9 and it will execute the update with the id and the body that was already obtained earlier.

5.1.2. Microservice Architecture

To implement the microservice architecture both of the services need to decompose into small components that run independently. Both services will still use Go as a programming language and Go Kit as a framework to build the API services.

Each of the services that use Go Kit as a framework will have 3 layers that consist of a transport layer that will use HTTP as the transport method, endpoint layer that will use a RPC style method and a service layer.

5.1.2.1. User Service

The user service will be rebuilt using the Go kit framework, the functions that are in the user services will be declared in the service layer, as interface and then it will be implemented as function.

```
1. type UserServices interface {
2.   Login(string, string) (string, error)
3.   Register(string, string, string) error
4. }
5. type UserService struct{}
```

The code above is the main part of the services layer that will be used to describe the services in user services and model it as an interface and will have implementation. The login service will have email and password with string data type as a request and will have JWT token with string data type and error data type as return in line 2. Register service will have email, username and password with string data type as a request and will have error data type as return.

```
1.  func (UserService) Login(email, password string) (string,
   error) {
2.   if !validation.EmailValidation(email) {
3.       return "", errors.New("Wrong Email Format")
4.   }
5.   checkUser := database.QueryUser(email, password)
6.   if !checkUser {
7.       return "", errors.New("Wrong Email or passwird")
8.   }
9.   token, err := jwt.CreateToken(email)
10.      return token.AccessToken, nil
11.  }
```

The code above is the implementation of a login service that was previously described as an interface. Line 2 will check the format of the email and will return an error if the format is wrong. Line 5 will be executed to check if the users exists or not if the users does not exist then an error will be returned. After checking the user line 9 will create a JWT token and if there is nothing error the function will return the token and nil.

```
1. func (UserService) Register(username, email, password
   string) error {
2.   var id = createUserId()
3.   if !validation.StringValidation(username, email) {
```

```
4.        return errors.New("Wrong Email or username Format")
5.    }
6.    checkUser := database.CheckUser(email)
7.    if checkUser {
8.        hashedPassword,                err                :=
   bcrypt.GenerateFromPassword([]byte(password),
   bcrypt.DefaultCost)
9.        err  =  database.RegisterUser(id,  username,  email,
   hashedPassword)
10.       }
11.       return nil
12.   }
```

The code above is the implementation of a register service that was previously described as an interface. In line 2 will be executed to create an id for the user using uuid. The username and email that is sent to the function then will be validated in line 3 and will return an error if the format is wrong. In line 6 the email will be used to check if there is the same user that has the same email. If there is no user with the same email then it will execute the if statement block in line 7. In line 8 the password will be hashed and in line 9 the email, username and hashed password will be inserted to the database, if there is no error the function will return nil.

```
1. func    MakeRegisterEndpoint(usv    services.UserServices)
   endpoint.Endpoint {
2.    return  func(ctx  context.Context,  request  interface{})
   (response interface{}, err error) {
3.        req := request.(registerRequest)
4.        err    =    usv.Register(req.Username,    req.Email,
   req.Password)
5.        if err != nil {
6.            return registerResponse{err.Error()}, nil
7.        }
8.        return registerResponse{}, nil
9.    }
10.   }
```

The code above is a register endpoint that takes the user services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to registerRequest struct that has property of username, email and password. The req variable then will be used as a parameter in line 4 that will call the Register function in the services layer and if there is no error the endpoint will return nil.

```
1. func     MakeLoginEndpoint(usv     services.UserServices)
   endpoint.Endpoint {
2.    return  func(ctx  context.Context,  request  interface{})
   (response interface{}, err error) {
3.        req := request.(loginRequest)
4.        v, err := usv.Login(req.Email, req.Password)
5.        if err != nil {
```

```
6.              return registerResponse{err.Error()}, nil
7.          }
8.          return loginResponse{v, ""}, nil
9.      }
10.     }
```

The code above is a login endpoint that takes the user services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to loginRequest struct that has property of email and password. The req variable then will be used as a parameter in line 4 that will call the Register function in the services layer and if there is no error the endpoint will return the token as variable v and nil as the error.

```
1. logger := log.NewLogfmtLogger(os.Stderr)
2. database.Connect_db()
3. defer database.DB.Close()
4. var usv services.UserServices
5. usv = services.UserService{}
6. loginHandler := httptransport.NewServer(
7.   transport.MakeLoginEndpoint(usv),
8.   transport.DecodeLoginRequest,
9.   transport.EncodeResponse,
10.  )
11.  registerHandler := httptransport.NewServer(
12.      transport.MakeRegisterEndpoint(usv),
13.      transport.DecodeRegisterRequest,
14.      transport.EncodeResponse,
15.  )
16.  var err = godotenv.Load(".env")
17.  var port = os.Getenv("SERVICE_USER_PORT")
18.  http.Handle("/login", loginHandler)
19.  http.Handle("/register", registerHandler)
20.  logger.Log("msg", "HTTP", "addr", "8081")
21.  logger.Log("err", http.ListenAndServe(port, nil))
```

The code above is the main part of the transport layer. In line 1 the logger that is used is imported from Go Kit libraries. In line 2 and 3 will be used to open the connection to the database. Line 4 -15 will be used to handle the transport of the endpoints and using it in line 18 and 19 that use net/http libraries to register the path and the handler. In line 16-17 will get the port number from the env file and use it in line 20 and 21 to start the server.

5.1.2.2. List Features Service

The List Features service will also be rebuilt using the Go Kit framework, the functions that are in the list feature services will be declared in the service layer, as interface and then it will be implemented as function.

```
1. type FeatureListServices interface {
2.   GetAllList(context.Context) ([]*ListFeatures, error)
3.   GetList(context.Context, string) (ListFeatures, error)
4.   InsertList(context.Context,   string,   string,   string,
   string, string, string, string, string, string) error
5.   UpdateList(context.Context,   string,   string,   string,
   string, string, string, string, string, string,
   error
6.   DeleteList(context.Context, string) error
7. }
8. type FeatureListService struct{}
9. type inmemService struct {
10.       mtx sync.RWMutex
11.       m   map[string]ListFeatures
12.  }
13.  func NewInmemService() FeatureListServices {
14.       return &inmemService{
15.            m: map[string]ListFeatures{},
16.       }
17.  }
```

The code above is the main part of the services layer that will be used to describe the services in list features services and model it as an interface and will be implemented. All the services will have context as parameters that will get from the endpoint layer. The GetAllList services will return a list of ListFeatures struct and an error. The GetList services will take id with string data type as parameter and will return ListFeatures struct and error. The InsertList services will take email, features name, description, live date, segmentation, channel, limit transaction, limit monthly and limit daily with string data type as parameters and will return error. The UpdateList services will take the same parameter as InsertList services but added with id with string data type and return error. The DeleteList service only takes an id with string data type as parameter and returns an error.

```
1. id := createDataId()
2. idString := id.String()
3. checkDuplicate,                    err                    :=
   database.CheckDuplication(featureName,
   featureSegmentation, featureChannel)
4. if checkDuplicate {
5.   insertDatabase, err := database.DB.Prepare("INSERT INTO
   tblFeature
   (featureId,featureName,featureDescription,featureLiveDate,
   featureSegmentation,featureChannel,limitTransaction,limitD
   aily,limitMonthly,created_by)
   values(?,?,?,?,?,?,?,?,?,?);")
6.   if err == nil {
7.        _, err := insertDatabase.Exec(idString, featureName,
   featureDescription, featureLiveDate, featureSegmentation,
   featureChannel, limitTransaction, limitDaily, limitMonthly,
   email)
```

```
8.   }
9. }
10.  return nil
```

The code above is the implementation of insert data function, as declared in line 1 is used to create an uuid for the feature using createDataId function. Line 3 will be used to verify if there is no duplicate data, and make it into a condition for if statement. Line 5-10 will be executed to insert the data in the database, if there's no error then the function will return as empty.

```
1. rows, err := database.DB.Query("select featureId,
   featureName, featureDescription, featureLiveDate,
   featureSegmentation, featureChannel, limitTransaction,
   limitDaily, limitMonthly from tblFeature;")
2. defer rows.Close()
3. var featureList []*ListFeatures
4. for rows.Next() {
5.   var each = ListFeatures{}
6.   err = rows.Scan(&each.FeatureId, &each.FeatureName,
   &each.FeatureDescription, &each.FeatureLiveDate,
   &each.FeatureSegmentation, &each.FeatureChannel,
   &each.LimitTransaction, &each.LimitMonthly,
   &each.LimitDaily)
7.   featureList = append(featureList, &each)
8. }
9. return featureList, nil
```

The code above is the implementation of the get all list function, as declared in line 1 it will be used for querying all list features in the database and closing the connection in line 2. Line 6 is used to create an empty variable which refers to the list of ListFeatures struct. Line 4-9 will be executed to loop the list from line 4 and append to the empty variable, if its success the list will be returned in line 12 and if it's not it will be returned as an empty list.

```
1. var each ListFeatures
2. 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)
3. if err == sql.ErrNoRows {
4.   return ListFeatures{}, ErrNotFound
5. }
6. return each, nil
```

The code above is the implementation of the get list by id function, as declared line 1 will be executed to create empty variable with ListFeatures struct as data type, line 2-6 will be executed

to query the feature and assign to the empty variable, if there is no error the function will return the query as AllData struct, if there is an error line 18 will be executed to return an empty list.

```
1. duplicate, err := database.UpdateQueryDataFeature(id)
2. year, month, day := time.Now().Date()
3. updateDate := strconv.Itoa(year) + "-" + strconv.Itoa(int(month)) + "-" + strconv.Itoa(day)
4. if duplicate {
5.   insertDatabase, err := database.DB.Prepare("UPDATE tblFeature set featureName=?, featureDescription=?, featureLiveDate=?, featureSegmentation=?, featureChannel=?, limitTransaction=?, limitDaily=?, limitMonthly=?, update_by=?, update_date=? where featureId=?;")
6.   if err == nil {
7.     _, err := insertDatabase.Exec(featureName, featureDescription, featureLiveDate, featureSegmentation, featureChannel, limitTransaction, limitDaily, limitMonthly, email, updateDate, id)
8.   }
9. }
10. return nil
```

The code above is the implementation of the update list function, as declared in line 1 will be executed to check if there is data with the same id as the id that is sent to the function. Line 2-3 will be executed to create a date that will be inserted in the update date column in the database. Line 4-9 will be executed to update the feature data not only updating the data but also inserting some new data such as update by column and update date column. If there is no error while updating, the function will return empty in line 10.

```
1. delete, err := database.DB.Prepare("DELETE FROM tblfeature where featureId = ?;")
2. if err == nil {
3.   _, err := delete.Exec(id)
4.   if err != nil {
5.     return err
6.   }
7. }
8. return nil
```

The code above is the implementation of the delete function, as declared the delete function will be executed in line 1-7 using the id that is sent to the function, if there is no error deleting the data the function will return empty in line 8.

```
1. func MakeInsertListEndpoint(lvs services.FeatureListServices) endpoint.Endpoint {
2.   return func(ctx context.Context, request interface{}) (response interface{}, err error) {
3.     req := request.(insertListRequest)
```

```
4.        e := lvs.InsertList(ctx, req.Email, req.FeatureName,
   req.FeatureDescription,                    req.FeatureLiveDate,
   req.FeatureSegmentation,                    req.FeatureChannel,
   req.LimitTransaction, req.LimitDaily, req.LimitMonthly)
5.        return insertUpdateListResponse{Err: e}, nil
6.    }
7. }
```

The code above is an insert list features service endpoint that takes the list features services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to insertlistRequest struct that has property of features name, description, live date, segmentation, channel, limit transaction, limit monthly and limit daily with string data type. The req variable then will be used as a parameter in line 4 that will call the InsertList function in the services layer and if there is no error the endpoint will return nil.

```
1. func                          MakeGetAllListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
2.    return func(ctx context.Context, request interface{})
   (response interface{}, err error) {
3.        _ = request.(getAllListRequest)
4.        v, e := lvs.GetAllList(ctx)
5.        return getAllListResponse{List: v, Err: e}, nil
6.    }
7. }
```

The code above is a get all list features service endpoint that takes the list features services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to getAllListRequest struct that has empty property. The req variable then will be used as a parameter in line 4 that will call the GetAllList function in the services layer and return the list of list features, if there is no error the endpoint will return nil.

```
1. func MakeGetListEndpoint(lvs services.FeatureListServices)
   endpoint.Endpoint {
2.    return func(ctx context.Context, request interface{})
   (response interface{}, err error) {
3.        req := request.(IdRequest)
4.        v, e := lvs.GetList(ctx, req.Id)
5.        return getListResponse{List: v, Err: e}, nil
6.    }
7. }
```

The code above is a get list features by id service endpoint that takes the list features services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to idRequest struct that has property of id that has string as data type. The req variable then will be used as a parameter in line 4 that will call the GetList

function in the services layer and return the list feature based on the id, if there is no error the endpoint will return nil.

```
1. func                                MakeUpdateListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
2.   return  func(ctx  context.Context,  request  interface{})
   (response interface{}, err error) {
3.       req := request.(updateListRequest)
4.       e  := lvs.UpdateList(ctx,  req.Email,  req.FeatureId,
   req.FeatureName,                    req.FeatureDescription,
   req.FeatureLiveDate,                req.FeatureSegmentation,
   req.FeatureChannel,  req.LimitTransaction,  req.LimitDaily,
   req.LimitMonthly)
5.       return insertUpdateListResponse{Err: e}, nil
6.   }
7. }
```

The code above is an update list features service endpoint that takes the list features services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to updateListRequest struct that has property of features name, description, live date, segmentation, channel, limit transaction, limit monthly and limit daily with string as data type. The req variable then will be used as a parameter in line 4 that will call the updateList function in the services layer and if there is no error the endpoint will return nil.

```
1. func                                MakeDeleteListEndpoint(lvs
   services.FeatureListServices) endpoint.Endpoint {
2.   return  func(ctx  context.Context,  request  interface{})
   (response interface{}, err error) {
3.       req := request.(IdRequest)
4.       e := lvs.DeleteList(ctx, req.Id)
5.       return deleteListResponse{Err: e}, nil
6.   }
7. }
```

The code above is a delete list features service endpoint that takes the list features services interface as parameter and an endpoint function as return. In line 3 variable request that has interface{} as data type will be casted to idRequest struct that has property of id with string data type. The req variable then will be used as a parameter in line 4 that will call the DeleteList function in the services layer and if there is no error the endpoint will return nil.

```
1. r := mux.NewRouter()
2. e := endpoint.MakeServerEndpoints(s)
3. options := []httptransport.ServerOption{
4. httptransport.ServerErrorHandler(transport.NewLogErrorHand
   ler(logger)),
5.   httptransport.ServerErrorEncoder(endpoint.EncodeError),
6. }
```

```
7.  r.Methods("GET").Path("/getlist").Handler(httptransport.Ne
    wServer(
8.    e.GetAllListEndpoint,
9.    endpoint.DecodeGetAllListRequest,
10.       endpoint.EncodeResponse,
11.       options...,
12.   ))
13.   r.Methods("GET").Path("/getlist/{id}").Handler(httptrans
    port.NewServer(
14.       e.GetListEndpoint,
15.       endpoint.DecodeGetListRequest,
16.       endpoint.EncodeResponse,
17.       options...,
18.   ))
19.   r.Methods("POST").Path("/insertlist").Handler(httptranspo
    rt.NewServer(
20.       e.PostListEndpoint,
21.       endpoint.DecodeInsertListRequest,
22.       endpoint.EncodeResponse,
23.       options...,
24.   ))
25.   r.Methods("PUT").Path("/updatelist/{id}").Handler(httptr
    ansport.NewServer(
26.       e.PutListEndpoint,
27.       endpoint.DecodeUpdateListRequest,
28.       endpoint.EncodeResponse,
29.       options...,
30.   ))
31.   r.Methods("DELETE").Path("/deletelist/{id}").Handler(htt
    ptransport.NewServer(
32.       e.DeleteListEndpoint,
33.       endpoint.DecodeDeleteListRequest,
34.       endpoint.EncodeResponse,
35.       options...,
36.   ))
```

The code above is the main part of the transport layer. In line 1 function mux.NewRouter() is used to make a router for the endpoints. In line 2 MakeServerEndpoints function that is in the endpoint layer is called to serve the server endpoints. Line 7-36 is used to transport the endpoints to the path and assign the HTTP method.

5.2. Results

To find out how microservice architecture average response time when there are a lot of users accessing the application compared to monolithic architecture, a performance testing was conducted. There are 3 different test scenarios with various test plan configurations. The testing plan for this project will test both services at the same time, each function that in list features service will be tested together with user services.

5.2.1. First Test Scenario

The first test scenario for this project is to test both architectures on the same environment and same operating system, where the application will share the same computer resources such as operating system, CPU power, memory and other resources. The main goal of this scenario is to compare the effectiveness of the architecture, whether it is effective to use microservice architecture in 1 environment or whether it's better to just use the monolith architecture in the 1 environment.

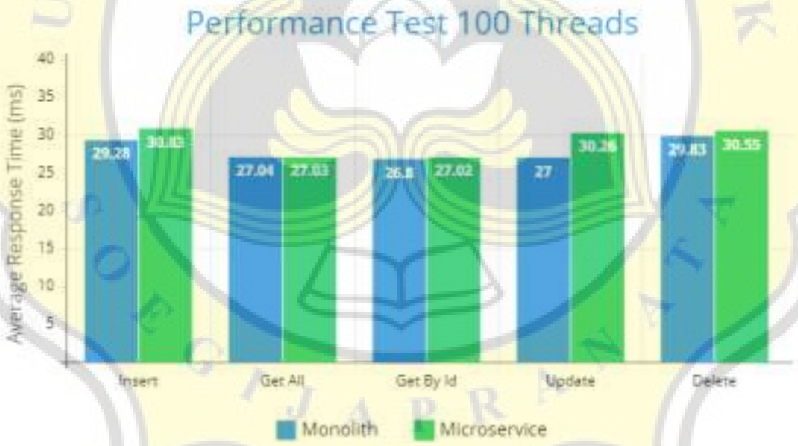

Figure 5.1 First Scenario Test With 100 Threads

The figure 5.1 above shows the average response time from both architectures in the first test scenario with 100 total requests configuration. The results of the test show that the difference between the architectures is not too far. The slowest average response time from monolith architecture is when testing the delete function with 29.83 ms, while from the microservice architecture is when testing the insert function with 30.83 ms. The fastest average response time

from monolith architecture is when testing the get by id function with 26.8 ms, while from the microservice architecture is also when testing the get by id function with 27.02 ms.

Figure 5.2 First Scenario Test With 200 Threads

The figure 5.2 above shows the average response time from both architectures in the first test scenario with 200 total requests configuration. The results of the test show that the difference between the architectures is also not too far compared to the previous test plan configuration. The slowest average response time from monolith architecture is when testing the delete function with 30.16 ms, while from the microservice architecture is when testing the insert function with 31.14 ms. The fastest average response time from monolith architecture is when testing the get by id function with 26.93 ms, while from the microservice architecture is also when testing the get by id function with 27.05 ms.

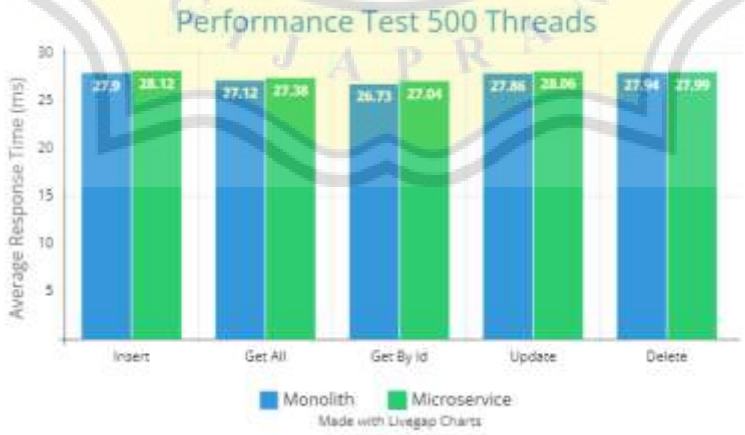

Figure 5.3 First Scenario Test With 500 Threads

The figure 5.3 above shows the average response time from both architectures in the first test scenario with 500 total requests configuration. The slowest average response time from monolith architecture is when testing the delete function with 27.94 ms, while from the microservice architecture is when testing the insert function with 28.12 ms. The fastest average response time from monolith architecture is when testing the get by id function with 26.73 ms, while from the microservice architecture is also when testing the get by id function with 27.04 ms.

Table 5.1. First Scenario Test 100 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	87.84	27.04	26.8	108	59.66	309.34	11	28.12181818
Microservice	92.49	27.03	27.02	121.04	61.1	328.68	11	29.88

Table 5.2. First Scenario Test 200 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	90.03	27.15	26.93	109.72	60.32	314.15	11	28.55909091
Microservice	93.42	27.07	27.05	123.92	61.88	333.34	11	30.30363636

Table 5.3. First Scenario Test 500 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	83.7	27.12	26.73	111.44	59.48	308.47	11	28.04272727
Microservice	84.36	27.38	27.04	112.24	59.98	311	11	28.27272727

From the table 5.1, 5.2 and 5.3 above it can be concluded that in the first test scenarios and in all test plan configurations the monolithic architecture has the lower average response time calculated using weighted arithmetic mean, to add more reference the average response time in this scenario will also be calculated using arithmetic mean without the weighted.

Table 5.4. First Scenario Test 100 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	29.28	27.04	26.8	27	29.83	139.95	5	27.99
Microservice	30.83	27.03	27.02	30.26	30.55	145.69	5	29.138

Table 5.5. First Scenario Test 200 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	30.01	27.15	26.93	27.43	30.16	141.68	5	28.336
Microservice	31.14	27.07	27.05	30.98	30.94	147.18	5	29.436

Table 5.6. First Scenario Test 500 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	27.9	27.12	26.73	27.86	29.74	139.35	5	27.87
Microservice	28.12	27.38	27.04	28.06	29.99	140.59	5	28.118

From the first test scenarios with 100, 200 and 500 total requests configuration that has been conducted, the test results were utilized to determine which architecture has the lower average response time using weighted arithmetic mean and arithmetic mean. Based on the table 5.4, 5.5 and 5.6 above it can be seen that with or without the weighted, the average response time from monolith architecture is lower than microservice architecture.

5.2.2. Second Test Scenario

The second test scenario for this project is to test both architectures on the same environments, both architectures will be tested in virtual machines where the virtual machine will run on its own environment where the application will not share the same computer resources such as operating system, CPU power, memory and other resources, while the monolith architecture will also run in virtual machine. The main goal of this scenario is to simulate the core of

microservice architecture that is distributed computing and to make the comparison fair monolith architecture will also be tested in a virtual machine.

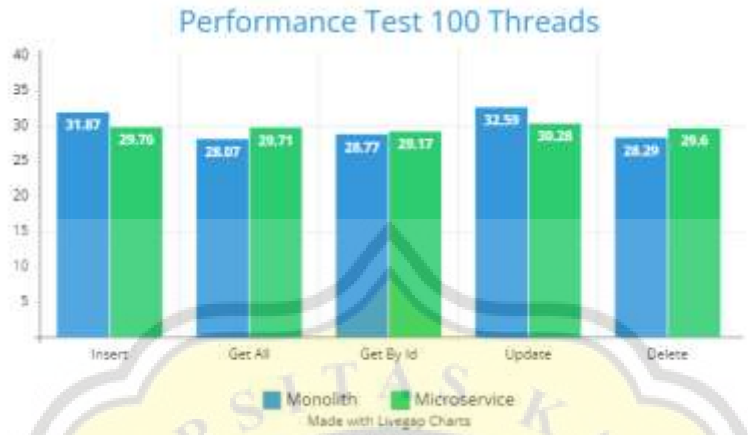

Figure 5.4 Second Test Scenario With 100 Threads

The figure 5.4 above shows the average response time from both architectures in the second test scenario with 100 total requests configuration. The results of the test shows that monolith architecture has lower average response time when testing the get all, get by id and delete function, meanwhile the microservice architecture has lower average response time when testing the insert and update function, which from the HTTP method perspective the POST and PUT function is considered as heavy operation, that in this test the microservice architecture can handle the operation really well. The slowest average response time from monolith architecture is when testing the update function with 32.59 ms while the microservice architecture is also when testing the update function with 30.28 ms. The fastest average response time from monolith architecture is when testing the get all function with 28.07 ms while the microservice architecture is when testing the get by id function with 29.17 ms.

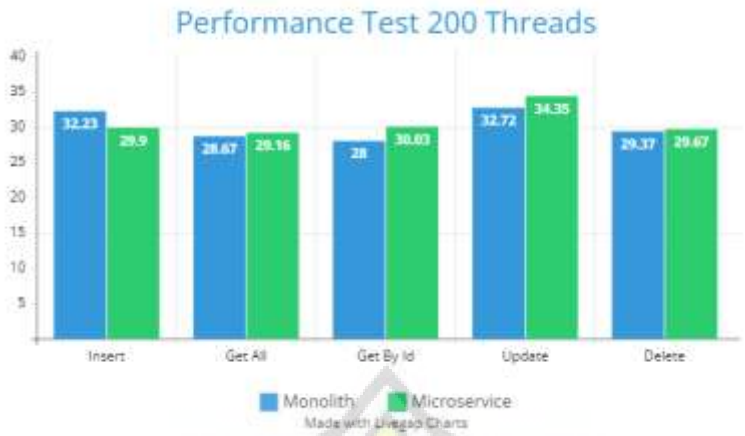

Figure 5.5 Second Test Scenario With 200 Threads

The figure 5.5 above shows the average response time from both architectures in the second test scenario with 200 total requests configuration. The slowest average response time from monolith architecture that is when testing the update function with 32.72 ms while the microservice architecture is also when testing the update function with 34.35 ms and the fastest average response time from monolith architecture is when testing the get by id function with 28 ms and the microservice architecture in get all function with 29.16 ms.

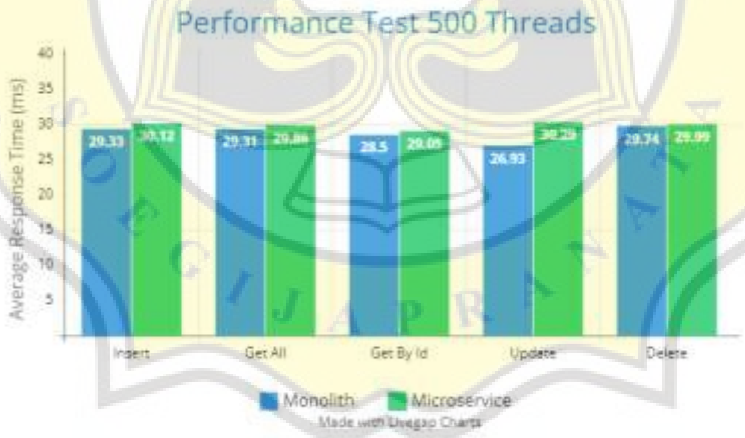

Figure 5.6 Second Test Scenario With 500 Threads

The figure 5.6 above shows the average response time from both architectures in the second test scenario with 500 total requests configuration. The slowest average response time from monolith architecture that is when testing the delete function with 29.74 ms while the microservice architecture is also when testing the update function with 29.99 ms and the fastest average response

time from monolith architecture is when testing the update function with 26.93 ms and the microservice architecture in get by id with 29.05 ms.

Table 5.7. Second Scenario Test 100 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	95.61	28.07	28.77	130.36	56.58	339.39	11	30.85363636
Microservice	89.28	29.71	29.17	121.12	59.2	328.48	11	29.86181818

Table 5.8. Second Scenario Test 200 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	96.69	28.67	28.31	130.88	58.74	343.29	11	31.20818182
Microservice	89.7	29.16	30.03	137.4	59.34	345.63	11	31.42090909

Table 5.9. Second Scenario Test 500 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	87.99	29.31	28.5	107.72	59.48	313	11	28.45454545
Microservice	90.36	29.86	29.05	121.16	59.98	330.41	11	30.03727273

From the table 5.7, 5.8 and 5.9 above it can be seen that in 100 total requests configuration the microservice architecture has lower average response time, however in 200 and 500 total requests configuration monolith architecture is having the lower average response time, to add more reference the average response time in this scenario will also be calculated using arithmetic mean without the weighted.

Table 5.10. Second Scenario Test 100 Total Requests Arithmetic Means Results

Architectures	**Average Response Times**					**SUM**	**Total Tests**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	31.87	28.07	28.77	32.59	28.29	149.59	5	29.918
Microservice	29.76	29.71	29.17	30.28	29.6	148.52	5	29.704

Table 5.11. Second Scenario Test 200 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	32.23	28.67	28.31	32.72	29.37	151.3	5	30.26
Microservice	29.9	29.16	30.03	34.35	29.67	153.11	5	30.622

Table 5.12. Second Scenario Test 500 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	29.33	29.31	28.5	26.93	29.74	143.81	5	28.762
Microservice	30.12	29.86	29.05	30.29	29.99	149.31	5	29.862

From the second test scenario with 100, 200 and 500 total requests configuration that has been conducted, the test results were utilized to determine which architecture has the lower average response time using weighted arithmetic mean and arithmetic mean. Based on the table 5.10, 5.11 and 5.12 above it can be seen that with or without the weighted the average response time from microservice architecture is lower in 100 total requests but higher in 200 and 500 total requests.

5.2.3. Third Test Scenario

The third test scenario for this project is to test both architectures on different environments, the microservice architectures will be tested in virtual machines and the monolith architecture will be tested in 1 environment. The main goal of this scenario is to compare both architectures in their own domain where the microservice architecture will implement the core of its architecture that is distributed computing, and the monolith architecture will be tested in the base operating system.

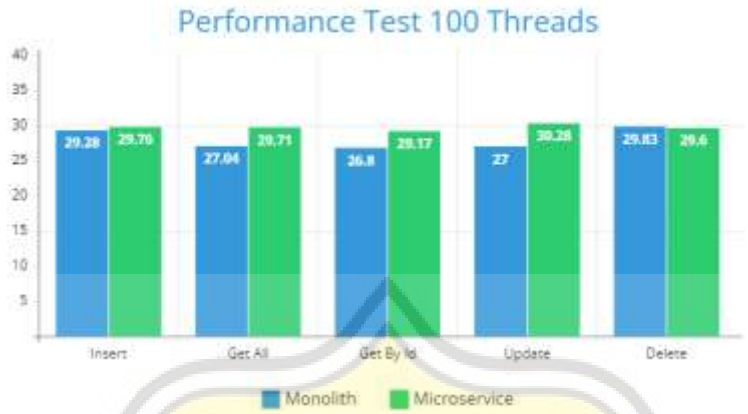

Figure 5.7 Third Test Scenario With 100 Threads

The figure 5.7 above shows the average response time from both architectures in the third test scenario with 100 total requests configuration. The slowest average response time from monolith architecture that is when testing the delete function with 29.83 ms while the microservice architecture is when testing the update function with 30.28 ms and the fastest average response time from monolith architecture is when testing the get by id function with 26.8 ms and the microservice architecture in delete function with 29.6 ms.

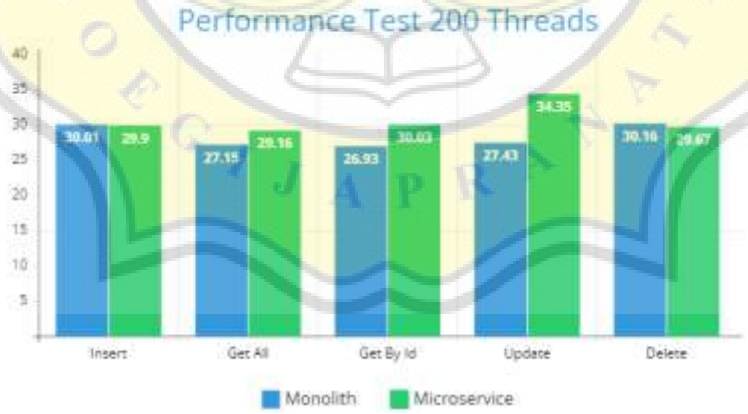

Figure 5.8 Third Test Scenario With 200 Threads

The figure 5.8 above shows the average response time from both architectures in the third test scenario with 200 total requests configuration. The slowest average response time from monolith architecture that is when testing the delete function with 30.16 ms while the microservice

architecture is when testing the update function with 34.35 ms and the fastest average response time from monolith architecture is when testing the get by id function with 26.93 ms and the microservice architecture in delete function with 29.67 ms.

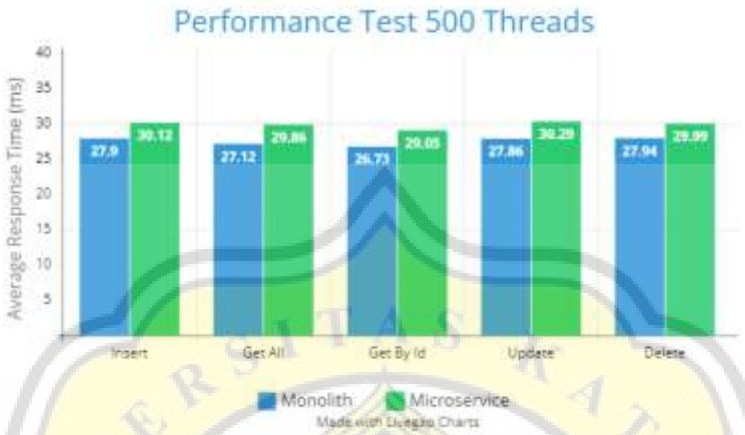

Figure 5.9 Third Test Scenario With 500 Threads

The figure 5.9 above shows the average response time from both architectures in the second test scenario with 500 total requests configuration. The slowest average response time from monolith architecture that is when testing the delete function with 27.94 ms while the microservice architecture is when testing the update function with 30.29 ms and the fastest average response time from monolith architecture is when testing the get by id function with 26.73 ms and the microservice architecture is also in get by id with 29.05 ms.

Table 5.13. Third Scenario Test 100 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	87.84	27.04	26.8	108	59.66	309.34	11	28.12181818
Microservice	89.28	29.71	29.17	121.12	59.2	328.48	11	29.86181818

Table 5.14. Third Scenario Test 200 Total Requests Weighted Arithmetic Means Results

Architectures	**Average Response Times * Weights**					**SUM**	**Total Weight**	**Weighted Arithmetic Means**
	Insert	***Get All***	***Get By Id***	***Update***	***Delete***			
Monolith	90.03	27.15	26.93	109.72	60.32	314.15	11	28.55909091
Microservice	89.7	29.16	30.03	137.4	59.34	345.63	11	31.42090909

Table 5.15. Third Scenario Test 100 Total Requests Weighted Arithmetic Means Results

Architectures	Average Response Times * Weights					SUM	Total Weight	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	83.7	27.12	26.73	111.44	59.48	308.47	11	28.04272727
Microservice	90.36	29.86	29.05	121.16	59.98	330.41	11	30.03727273

From the table 5.13, 5.14 and 5.15 above it can be concluded that in the third test scenarios and in all test plan configurations the monolithic architecture has the lower average response time calculated using weighted arithmetic mean, to add more reference the average response time in this scenario will also be calculated using arithmetic mean without the weighted.

Table 5.16. Third Scenario Test 100 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	29.28	27.04	26.8	27	29.83	139.95	5	27.99
Microservice	29.76	29.71	29.17	30.28	29.6	148.52	5	29.704

Table 5.17. Third Scenario Test 200 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	30.01	27.15	26.93	27.43	30.16	141.68	5	28.336
Microservice	29.9	29.16	30.03	34.35	29.67	153.11	5	30.622

Table 5.18. Third Scenario Test 500 Total Requests Arithmetic Means Results

Architectures	Average Response Times					SUM	Total Tests	Weighted Arithmetic Means
	Insert	*Get All*	*Get By Id*	*Update*	*Delete*			
Monolith	27.9	27.12	26.73	27.86	29.74	139.35	5	27.87
Microservice	30.12	29.86	29.05	30.29	29.99	149.31	5	29.862

From the third test scenarios with 100, 200 and 500 total requests configuration that has been conducted, the test results were utilized to determine which architecture has the lower average response time using weighted arithmetic mean and arithmetic mean. Based on the tables 5.16, 5.17 and 5.18 above it can be seen that with or without the weighted the average response time from monolith architecture is lower than microservice architecture.

5.2.4. Discussion

From the test scenarios that have been conducted it can be concluded that monolith architecture is having a dominantly better performance in 3 different test scenarios, this is proven by how the monolith architecture gets the most lower weighted arithmetic means and arithmetic means results compared to microservice architecture, except in the second test scenarios with 100 total requests configuration somehow the microservice architecture can get lower average response time. From table 5.7 and 5.10 it shows that in the update and insert function where the heaviest weight placed, the microservice managed to get lower results from multiplication of average response time and the weight, despite that the monolith architecture has lower results in get all, get by id and delete function. The role of how big the data in the application is affecting the application performance, but somehow it only increased between the 100 total requests to 200 total requests configuration. The average response time from 500 total requests configuration shows some inconsistency, this anomaly can happen because there are some requests that failed in some operations, so that it affects both architectures test results.

Despite being the traditional model of architecture the impressive performance shown by the monolith architecture can happen because of how the monolith architecture is designed to be a self-contained unit that couples all of its components into a one big code, that in this project the monolith architecture is faster and more effective to use as the architecture of the application, compared to the modern style architecture microservice. Overall despite the gap between the architectures being relatively small and the architecture tested in several environments, the monolith architecture is still managing to have lower average response time, this proves that monolith architecture is an architecture that is still feasible to implement, considering how the monolith architecture is easier to implement.