VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Duong Tuan Hiep

TRANSFORMING MONOLITH PHP SERVICES TO INDUSTRIAL RESTFUL SERVICES

Technology and Communication
2020

VAASA UNIVERSITY OF APPLIED SCIENCES
Degree Program in Information Technology

ABSTRACT

Author	Duong Hiep
Title	Transforming Monolith PHP Services to Industrial RESTful Services
Year	2020
Language	English
Pages	29
Name of Supervisor	Smail Menani

The thesis was done as a part of the work of rewriting Fliq's Backend with a modern programming language with the aim of modernizing their current system in 2020.

The task included the re-design of the current API to conform with RESTful Architecture standard and implementing these services in Golang, eliminating existing bugs inside the system and set a baseline for future services.

The thesis detailed the new design for the API using API Specification Language RAML, how those services would be implemented in Golang, common patterns when coding Golang and the workaround for those problems.

Keywords Backend, Golang, Web Services, RAML

CONTENTS

LIST OF FIGURES AND TABLES

1 INTRODUCTION

As technology is increasingly evolving, the need to adapt and improve previous software to extends its capabilities to meet the current requirements and needs become an important task in every software service companies. Fliq’s solution for managing warehouses, factories and transportation, which was written mostly in PHP, has proven to be a reliable and powerful solution for customer companies. However, as the demand in performance and stability increases, the old technologies used in producing the software are not capable for further optimizing and upscaling thus leading to the decision of re-creating the software in a new, modern language.

The new back end is required to have the same functionality of the previous version in PHP and at the same time eliminate a majority of issues that exist inside the old system. The new backend code should be easier to understand and well formatted so that new engineers can get familiar with the system easier. Another important requirement for the new back end is to have the document and the specification along with the code base itself. On the other hand, the performance of the new backend should be significantly better than its predecessor as well as the possibility to further scale up the system horizontally and vertically. Additionally, the new product should also run with less resources and platform independent. With the above requirement, the new languages of choice and its related document should be chosen wisely. After considering all the possibilities, a decision was made to design the new back end APIs with RAML (Restful API Modelling Language) while the language that will implements the API Specification is Golang.

2 TECHNOLOGIES USED

The new technologies to be used in the new backends must satisfy several requirements, ranging from performance to codebase maintainability.

Firstly, the new backend should be significantly faster than its predecessor was. This means that refactoring the current PHP codebase will not be sufficient as it will still has the same performance overhead cost of having PHP script running as a codebase. PHP is a scripting language, which means that in will need an additional server to read and understand the language. Hence, the new backend must be written in a language that allow the program to act as a standalone server without the need of an additional webserver to assist.

Secondly, a great issue with the current PHP codebase is that it has been written by many developers throughout the years. This leads to the codebase being too large to be refactored effectively and at the same time it allows many parts of the code to be outdated and idempotent. The new codebase should be maintainable and easier to understand.

Lastly, the new backend, while having the same functionalities as the old code base, should have all the APIs created in a unified, standardized manners. Previously, in PHP, all the URLs are registered simply by function declaration. This leads to different developers creating APIs with different standard naming convention and functionalities. This requires an additional form of documentation, specifically an API specification document where the functionality of each services is clearly defined.

2.1 Golang and Gin Framework

Go is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing and CSP-style concurrency. The language is often referred to as "Golang" because of its domain name, golang.org, but the proper name is Go. /1/

Writing the new backend with Golang can be a wise start for the new backend. The syntax for the language is very strict while at the same time requires very little additional notations. This helps greatly with code readability. On the other hand, the performance for Golang is one of the fastest out of the most common 4 languages for service-side development. /2/

Gin is a high-performance micro-framework that can be used to build web applications. It allows developers to create middleware that can be plugged into one or more request handlers or groups of request handlers.

Gin’s performance has been proven to be one of the fastest in the middleware solution inside Golang /3/. Additionally, the framework also contains shortcuts for a great number of repetitive tasks and method which are commonly used which will also reduce the side of the codebase and help increase the readability.

2.2 API Specification Language – RAML

An API’s design is a solid blueprint on what the developer API wants to achieve and gives a comprehensive overview on all the endpoints and CRUD operations associated with each of them. An effective API design can greatly help in implementation and prevent complicated configurations, adherence to naming schema within classes, and a host of other issues that can cost significantly later as the software mature. The design process will also help designers think through exactly how data will be distributed and how core product will work.

RESTful API Modelling Language (RAML) is a YAML-based language for describing RESTful APIs. It provides all the information necessary to describe RESTful or practically RESTful APIs. Although designed with RESTful APIs in mind, RAML is capable of describing APIs that do not obey all constraints of REST (hence the description "practically RESTful"). It encourages reuse, enables discovery and pattern sharing and aims for merit-based emergence of best practices. /4/

RAML was chosen to be the language to handle the designing, documenting the APIs because of multiple reasons. Firstly, RAML is one of the most common documentation languages that are currently available as an open-source language. Writing the new API specification in a common language means that there is a greater number of third-party developers, in Fliq's case the customer company developers, will understand the languages. Additionally, the minimal annotations inside RAML will also reduce the new learner difficulties when picking up the language. Lastly, the fact that RAML allows functions, traits reusable means that multiple services, common services can have similar functionalities while only need it to be defined once.

2.3 Rest Architecture

Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations. Other kinds of Web services, such as SOAP Web services, expose their own arbitrary sets of operations.

With the task of redesigning the API, it is also a great opportunity to standardize the current API existed inside Fliq's API infrastructure to conform with REST Architecture as this will help significantly in giving the system as a whole an uniformed standards that future API created can be based up on.

3 API SPECIFICATION

The new API is re-designed to conform to the rules of the RESTful API architecture. Firstly, the task is to identify the resources that needed to be exposed for accessing and manipulating. Developers can then apply RESTful naming convention to these services. Secondly the aim is to define the common traits and behaviours that these services need to have, from which developers can define it clearly and then can reuse for all the services and future services. The third task is to plug these traits and behaviours on services that need to support the functionalities.

3.1 Resource Maming

In RESTful architecture style, APIs use Uniform Resource Identifiers (URIs) to address resources, the entity that the API support. By implementing knowledge about REST Architecture design. An example of the “notes” services can be modified and supports the following endpoints:

- GET: /{BASE-URI}/tp/notes
- GET: /{BASE-URI}/tp/notes/{id}
- POST: /{BASE-URI}/tp/notes
- PUT: /{BASE-URI}/tp/notes
- DELETE: /{BASE-URI}/tp/notes/{id}

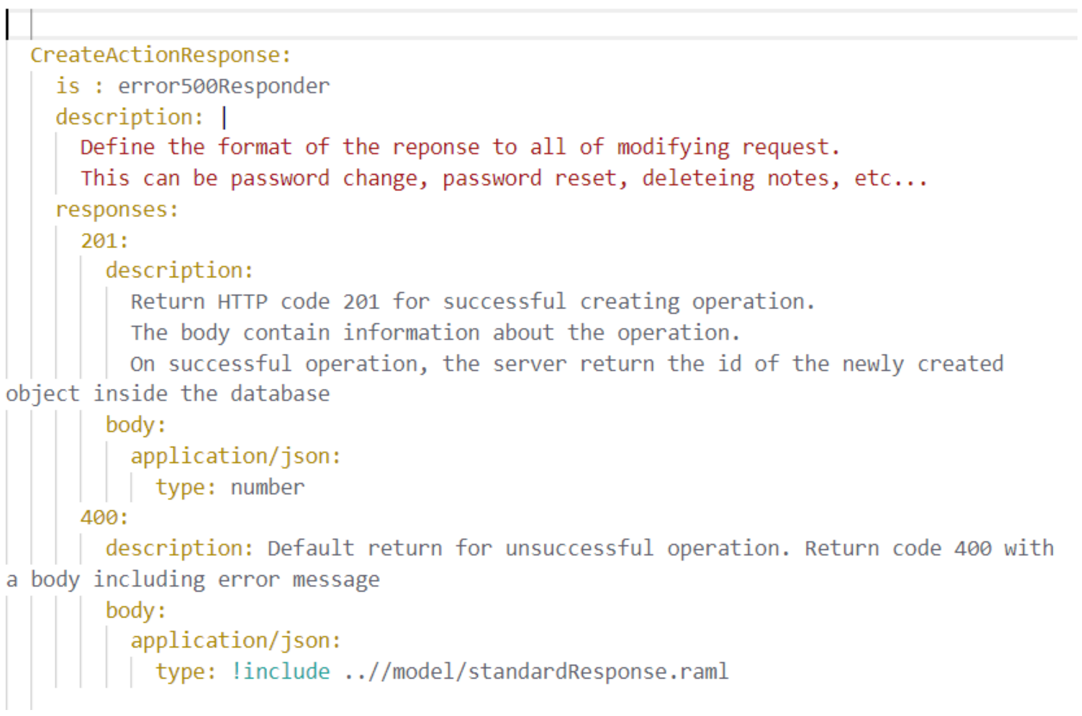
The notes endpoints support retrieving a collection of notes in the services. It can support additional function to present the notes collection to clients such as filtering, sorting and ordering ability. Notes/{id} supports retrieving a specific note with the “id” specified by the URI parameter {id}. Creating a new note resource can be done with notes endpoint that has the HTTP Method POST while modifying the note can be done with the same endpoint but with HTTP Method PUT. The delete functionality can be implemented with endpoint notes/{id} where the “id” of the notes can be specified on the URI itself.

By separating the previously created “update_note” into three different endpoints for each of the function it supports: create, update and delete, this API becomes more coherent since each end point will only serve a singular purpose. The design also makes uses of additional HTTP methods: POST, PUT and DELETE. These methods help any external users understand the functionality of the Notes Services since these methods are the standard defined by RFC and are widely used in modern webservice.

3.2 Define Common Traits and Behaviors

Taking the Notes services a step further, the Note service can also support additional functionality when retrieving a collection of record (/tp/notes), have a standardized way of responding to create, update and delete.

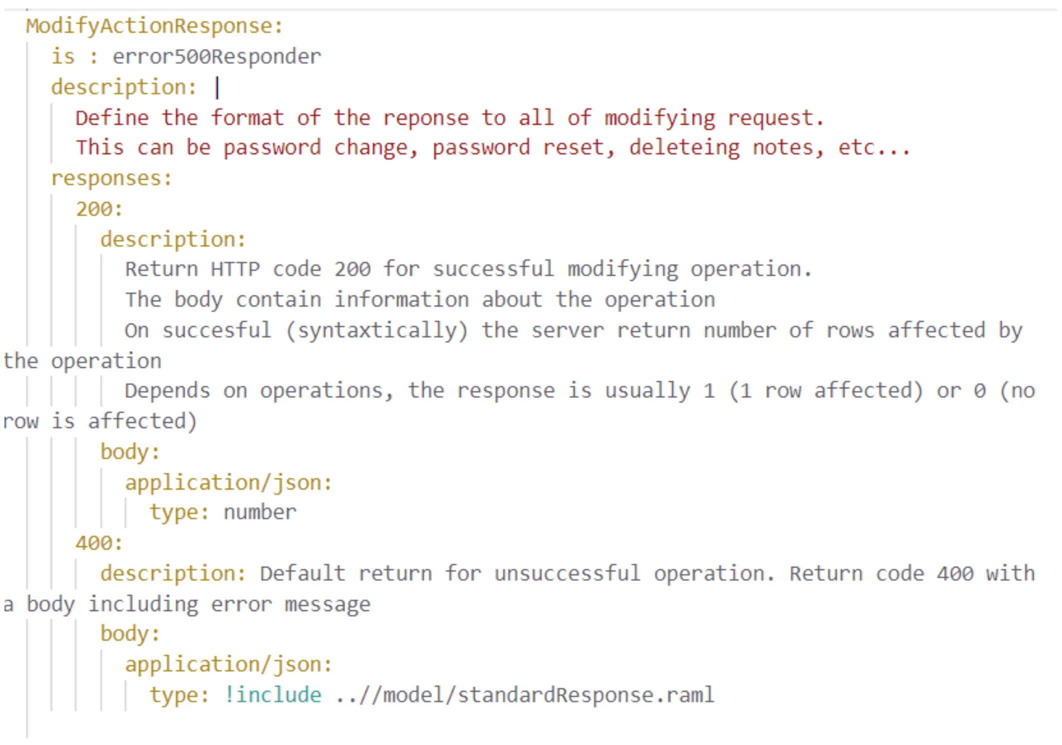
A create operation return data should contains information related to the operation. One of the most common practices is to include the ID of the newly created resource in the response. A create behavior can be defined as follows:



```
 CreateActionResponse:
   is : error500Responder
   description: |
     Define the format of the reponse to all of modifying request.
     This can be password change, password reset, deleteing notes, etc...
   responses:
     201:
       description:
         Return HTTP code 201 for successful creating operation.
         The body contain information about the operation.
         On successful operation, the server return the id of the newly created
object inside the database
       body:
         application/json:
           type: number
     400:
       description: Default return for unsuccessful operation. Return code 400 with
a body including error message
       body:
         application/json:
           type: !include ..//model/standardResponse.raml
```

Figure 1. Create action response defined in RAML

Other standard behaviours include modify and delete operation response. From the client perspective, these operations will alter a group or a single record. The return data should contain information that will reflect the outcome of the request. In the “notes” services case, it is only possible to alter a single record at a time, which designers can convey by integers and a standard message.

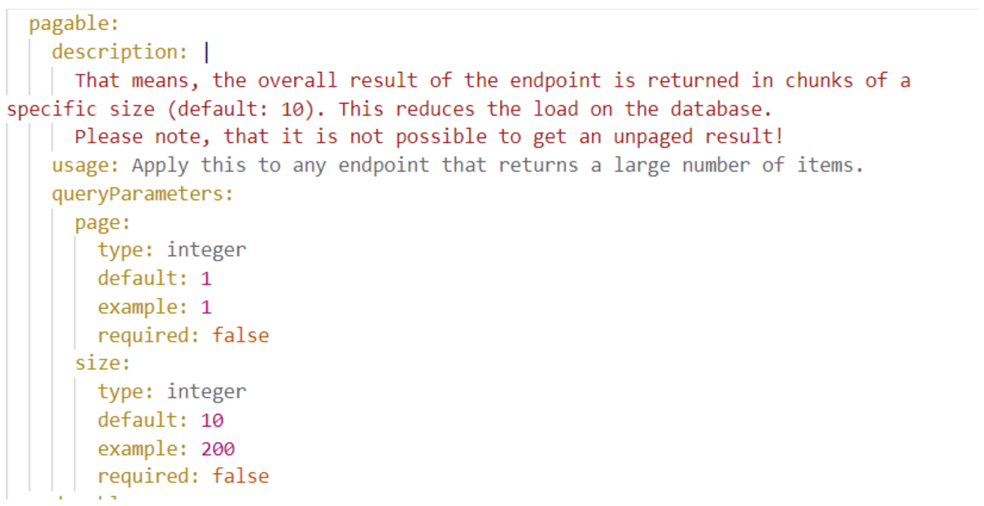


```
ModifyActionResponse:
  is : error500Responder
  description: |
    Define the format of the reponse to all of modifying request.
    This can be password change, password reset, deleteing notes, etc...
  responses:
    200:
      description:
        Return HTTP code 200 for successful modifying operation.
        The body contain information about the operation
        On succesful (syntaxtically) the server return number of rows affected by
the operation
        Depends on operations, the response is usually 1 (1 row affected) or 0 (no
row is affected)
      body:
        application/json:
          type: number
    400:
      description: Default return for unsuccessful operation. Return code 400 with
a body including error message
      body:
        application/json:
          type: !include ..//model/standardResponse.raml
```

Figure 2. Modification and deletion traits defined in RAML

Common functionalities often seen in services that return a collection of resources are the ability to Sort, Order and to Filter based on certain well – defined parameters. Using RAML as the API specification language, designers can also create and construct the structure of all the “traits” that he wants the API to support.

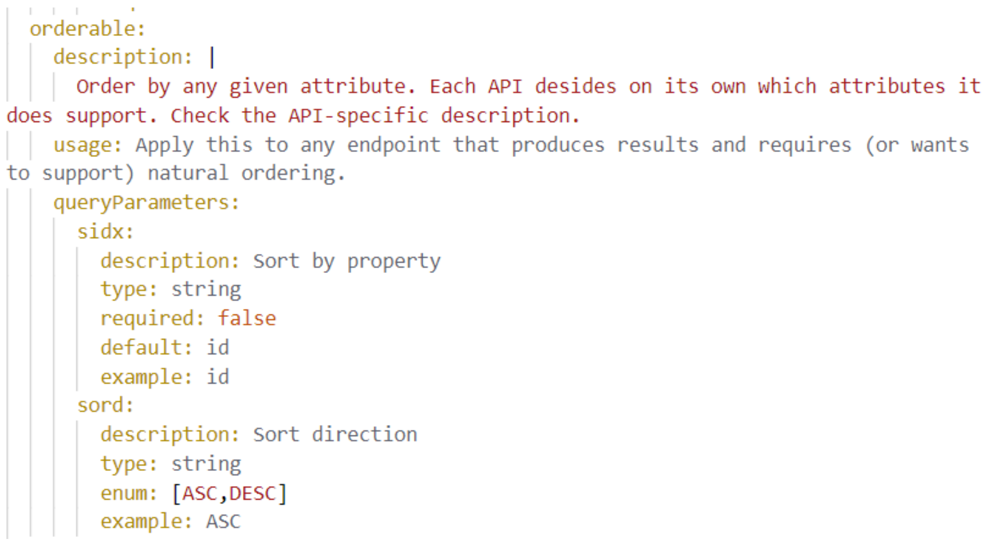
A designer can create a trait called “pagable” to define the rules and functionality of the paging mechanism of the endpoints. He can also define traits called “orderable” and “filterable” to define ordering and filtering capabilities, respectively.



```
pagable:
  description: |
    That means, the overall result of the endpoint is returned in chunks of a
specific size (default: 10). This reduces the load on the database.
    Please note, that it is not possible to get an unpaged result!
  usage: Apply this to any endpoint that returns a large number of items.
  queryParameters:
    page:
      type: integer
      default: 1
      example: 1
      required: false
    size:
      type: integer
      default: 10
      example: 200
      required: false
```

Figure 3. "Pagable" trait defined in RAML

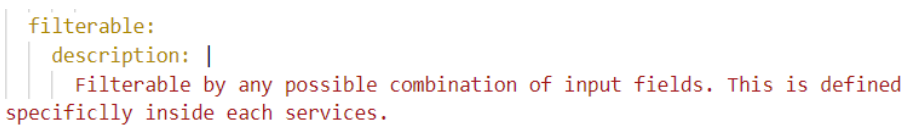
The "pagable" trait defines the ability of the paging mechanism in the service. Performing paging will require developer to have a "page" parameter to specify the number of the page and a "size" parameter to specify the number of items on each page.



```
orderable:
  description: |
    Order by any given attribute. Each API desides on its own which attributes it
does support. Check the API-specific description.
  usage: Apply this to any endpoint that produces results and requires (or wants
to support) natural ordering.
  queryParameters:
    sidx:
      description: Sort by property
      type: string
      required: false
      default: id
      example: id
    sord:
      description: Sort direction
      type: string
      enum: [ASC,DESC]
      example: ASC
```

Figure 4. "Orderable" trait defined in RAML

The orderable requires two parameters "sidx" and "sord". "sidx" to specify the name of the parameter should be used as a sorting target and "sord" to specify which order.



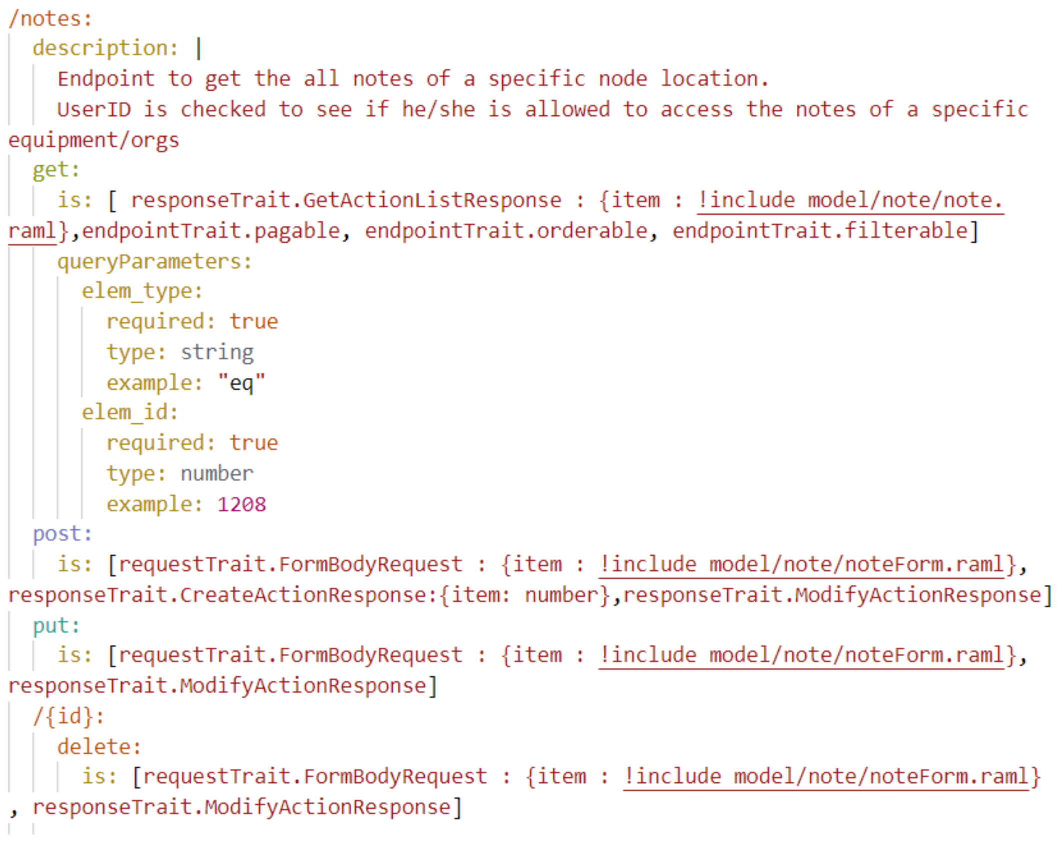
```
  filterable:
    description: |
      Filterable by any possible combination of input fields. This is defined
specificlly inside each services.
```

Figure 5. “Filterable” trait defined in RAML

A filterable trait can be defined loosely so that it allows other services to have different attributes filtered.

3.3 Plugging the Common Traits and Behaviors

A completed design of the API inside API specification would look as the follows:



```
/notes:
  description: |
    Endpoint to get the all notes of a specific node location.
    UserID is checked to see if he/she is allowed to access the notes of a specific
equipment/orgs
  get:
    is: [ responseTrait.GetActionListResponse : {item : !include model/note/note.
raml},endpointTrait.pagable, endpointTrait.orderable, endpointTrait.filterable]
    queryParameters:
      elem_type:
        required: true
        type: string
        example: "eq"
      elem_id:
        required: true
        type: number
        example: 1208
  post:
    is: [requestTrait.FormBodyRequest : {item : !include model/note/noteForm.raml},
responseTrait.CreateActionResponse:{item: number},responseTrait.ModifyActionResponse]
  put:
    is: [requestTrait.FormBodyRequest : {item : !include model/note/noteForm.raml},
responseTrait.ModifyActionResponse]
  /{id}:
    delete:
      is: [requestTrait.FormBodyRequest : {item : !include model/note/noteForm.raml}
, responseTrait.ModifyActionResponse]
```

Figure 6. Notes service defined in RAML

4 ARCHITECTURE

4.1 Overall Application Structure

Structuring the system the right way is a very important step at the beginning of the services. This is even more crucial as in Golang the files, folders structure also represents the inner working structure of the application.

On the high-level design, the backend services would look as the following figures:

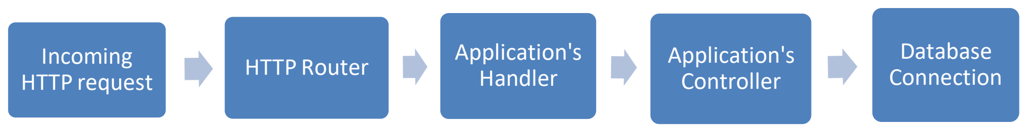


Figure 7. Logic flow of a request inside the service

4.2 Application Folder & Packages Structure

The previous application structure can be translated into a packages structure as follows:

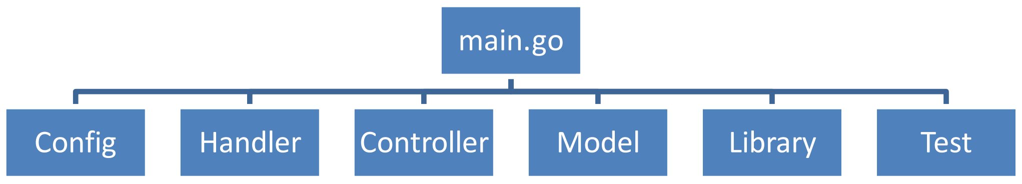


Figure 8. Folder structure of the application

• Main.go: This is the starting point of the application, where all the endpoint will be declared as well as the configuration for those endpoint and services.

- Config: Contains all the configuration of the application. This configuration includes: Database configuration, session configuration, user’s account configuration, and endpoint configuration

- Handler: The first contact point of business logic inside the application. When the routers successfully route a request, it will invoke a specific part of the handlers. We will do some general checking and overall logic here before passing it to the Controller

- Controller: Controller package is where most of the logic of the application will be handled. It is also where the communication between the services and the database is done.

- Model: Model contains all the structures of the object (class) of our application

- Library: contains all the external packages and packages that can be independently re-used in other application.

- Test: contains all the tests related to our application

5 IMPLEMENTATION

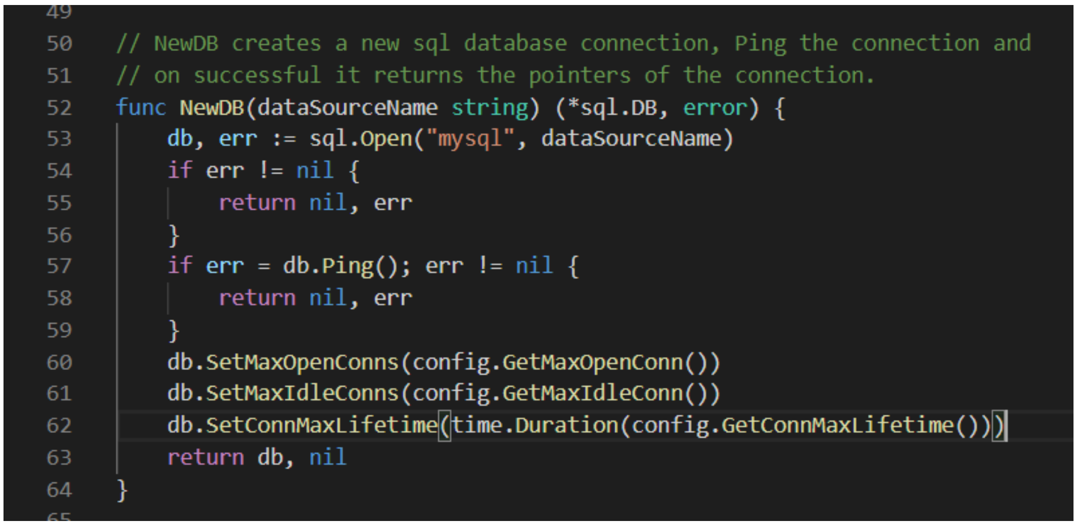
5.1 Application Configuration

The first things that should be taken into consideration in the application is the configuration. Maintaining a well-run application requires that the configuration should be made correctly as it will have great impacts on the application.

5.1.1 Database Configuration

The very first thing in the configuration is the Database – related configuration. The connection of the database and the backend service is handled by the driver of the "sql" standard library package in Golang.

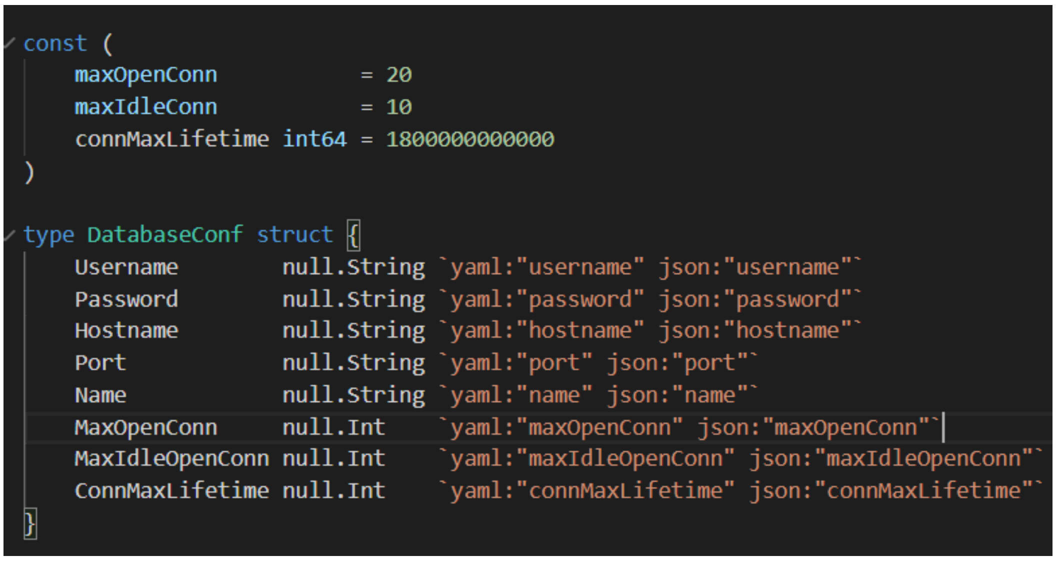
Configuration that needs to be taken care of includes the data source name, maximum number of open connections, maximum number of idle connections and maximum lifetime of each connection.



```go
// NewDB creates a new sql database connection, Ping the connection and
// on successful it returns the pointers of the connection.
func NewDB(dataSourceName string) (*sql.DB, error) {
    db, err := sql.Open("mysql", dataSourceName)
    if err != nil {
        return nil, err
    }
    if err = db.Ping(); err != nil {
        return nil, err
    }
    db.SetMaxOpenConns(config.GetMaxOpenConn())
    db.SetMaxIdleConns(config.GetMaxIdleConn())
    db.SetConnMaxLifetime(time.Duration(config.GetConnMaxLifetime()))
    return db, nil
}
```

Figure 9. Database configuration function

As a requirement, database configuration should also be portable as will be situation where configuring it externally (outside of the code of the application) is needed and therefore it will also be put inside the configuration package.

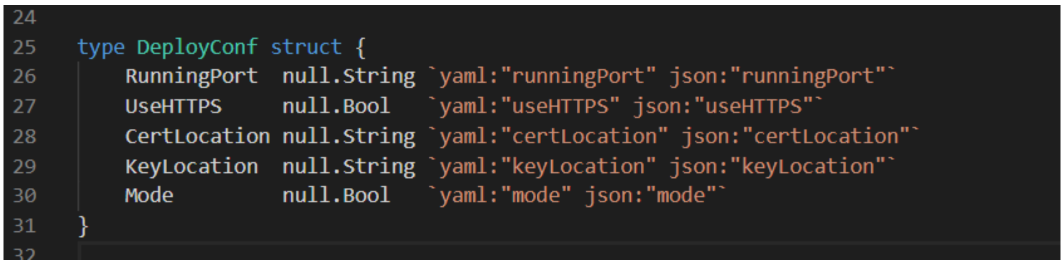


```go
const (
    maxOpenConn            = 20
    maxIdleConn            = 10
    connMaxLifetime int64 = 1800000000000
)

type DatabaseConf struct {
    Username        null.String `yaml:"username" json:"username"`
    Password        null.String `yaml:"password" json:"password"`
    Hostname        null.String `yaml:"hostname" json:"hostname"`
    Port            null.String `yaml:"port" json:"port"`
    Name            null.String `yaml:"name" json:"name"`
    MaxOpenConn     null.Int    `yaml:"maxOpenConn" json:"maxOpenConn"`
    MaxIdleOpenConn null.Int    `yaml:"maxIdleOpenConn" json:"maxIdleOpenConn"`
    ConnMaxLifetime null.Int    `yaml:"connMaxLifetime" json:"connMaxLifetime"`
}
```

Figure 10. Database configuration and config struct

5.1.2 Gin Middleware Configuration

Gin middleware is a highly efficient and highly functional middleware that are currently available in Go. It being very developer-friendly, users of the client will not need too much configuration other than the running mode.



```go
type DeployConf struct {
    RunningPort  null.String `yaml:"runningPort" json:"runningPort"`
    UseHTTPS     null.Bool   `yaml:"useHTTPS" json:"useHTTPS"`
    CertLocation null.String `yaml:"certLocation" json:"certLocation"`
    KeyLocation  null.String `yaml:"keyLocation" json:"keyLocation"`
    Mode         null.Bool   `yaml:"mode" json:"mode"`
}
```

Figure 11. Deployment configuration inside the application

The few configuration needed is the deployment configuration and its running mode. This gives developers very few problems as they can focus on other elements of the applications.

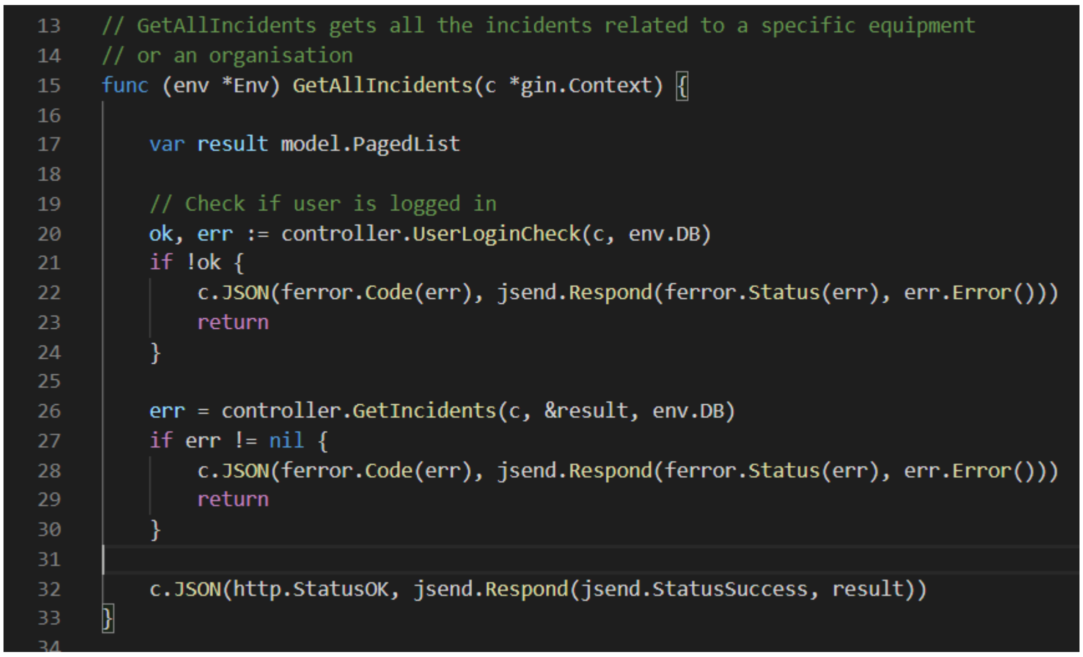
5.2 Implementation of the Services

After managing the configuration, developers should next look at ways to implement a service inside the system. Readers can look as an example of the “incident” services to see, which patterns are used and the exact implementation of the services.

5.2.1 Creating the Handler

The handler will be the first contact point of the coming request with the business logic. This is where the services do the validation of the request. This validation can be checking if the request is coming from the valid sources (the authorized user). It is done by checking the cookies coming with the request and finding the users that the cookies belong to. If the process is successful, the user’s metadata will be added to the current processing context and passed to the controller to handle the actual request.

In addition to the mentioned functions, these layers define the error handling as well as structure the response from the controller layer underneath to present it to the client in a standardized format.



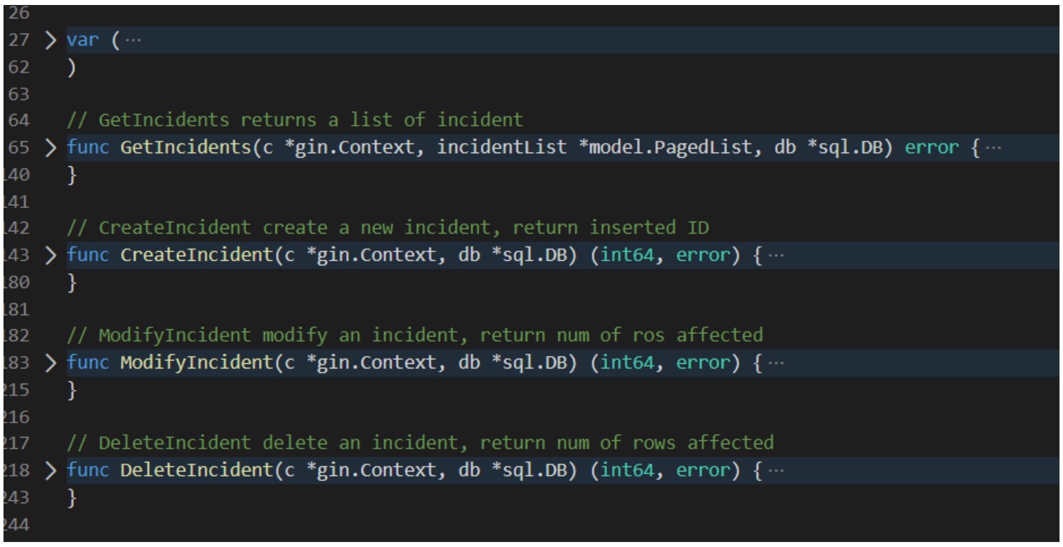
```go
// GetAllIncidents gets all the incidents related to a specific equipment
// or an organisation
func (env *Env) GetAllIncidents(c *gin.Context) {

    var result model.PagedList

    // Check if user is logged in
    ok, err := controller.UserLoginCheck(c, env.DB)
    if !ok {
        c.JSON(ferror.Code(err), jsend.Respond(ferror.Status(err), err.Error()))
        return
    }

    err = controller.GetIncidents(c, &result, env.DB)
    if err != nil {
        c.JSON(ferror.Code(err), jsend.Respond(ferror.Status(err), err.Error()))
        return
    }

    c.JSON(http.StatusOK, jsend.Respond(jsend.StatusSuccess, result))
}
```

Figure 12. Controller file of “GetAllIncident”

The function begins with creating the return model for the data. This return model is a standardized return model for all the services inside the applications. After creating the return object, the handler then checks for the validity of the request by invoking “UserLoginCheck”. If this fails, an error message will be sent as the response. On successful checking, the context is then passed to the controller where the business logic of “GetIncidents” is handled.

5.2.2 Creating the Controller

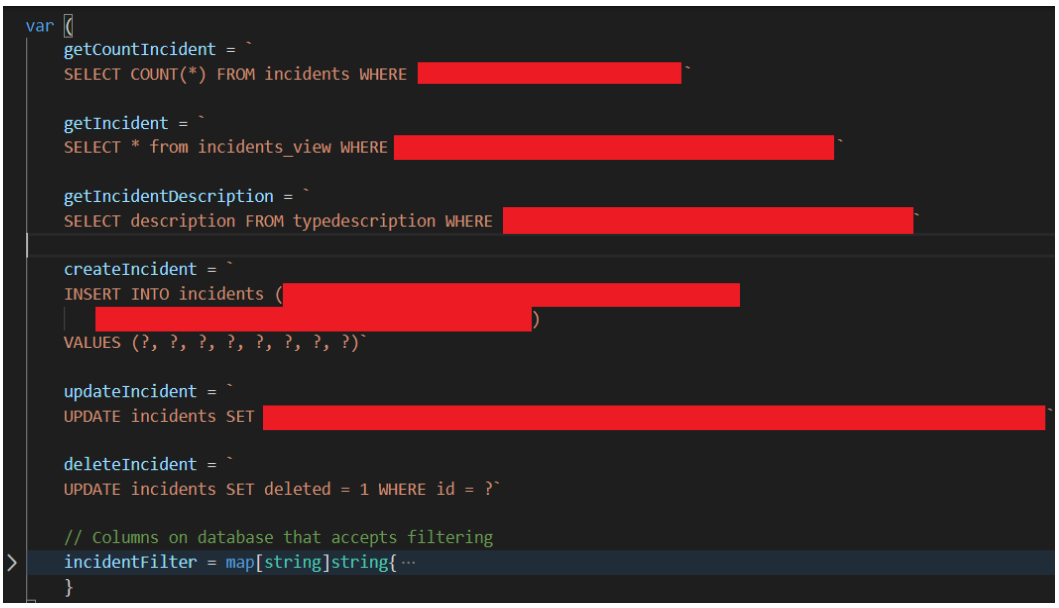
The controller is where the logic of the application is a handled. On this layer, the communication between the services and the database take places and the representation of the data is decided.



```go
var (···
)

// GetIncidents returns a list of incident
func GetIncidents(c *gin.Context, incidentList *model.PagedList, db *sql.DB) error {···
}

// CreateIncident create a new incident, return inserted ID
func CreateIncident(c *gin.Context, db *sql.DB) (int64, error) {···
}

// ModifyIncident modify an incident, return num of ros affected
func ModifyIncident(c *gin.Context, db *sql.DB) (int64, error) {···
}

// DeleteIncident delete an incident, return num of rows affected
func DeleteIncident(c *gin.Context, db *sql.DB) (int64, error) {···
}
```

Figure 13. Controller file overall structure

A typical controller file for a service will contains two main elements: the used local variable declared inside “var” block and the actual functions that handles the business logic.



```
var (
    getCountIncident = `
    SELECT COUNT(*) FROM incidents WHERE [redacted]`

    getIncident = `
    SELECT * from incidents_view WHERE [redacted]`

    getIncidentDescription = `
    SELECT description FROM typedescription WHERE [redacted]`

    createIncident = `
    INSERT INTO incidents ([redacted]
        [redacted])
    VALUES (?, ?, ?, ?, ?, ?, ?, ?)`

    updateIncident = `
    UPDATE incidents SET [redacted]`

    deleteIncident = `
    UPDATE incidents SET deleted = 1 WHERE id = ?`

    // Columns on database that accepts filtering
    incidentFilter = map[string]string{···
)
```

Figure 14. Var block declaration

A good pattern to separate an SQL command from the Go code is to explicitly declare them as type String variable and store them at a concentrated place. This helps increase the readability of the Go code in the application since it will be purely a Go code.

Another good pattern to use “var” block is to create a HashMap of fields that are available to be sorted as well as filtered against. This creates a “white-list” for each of the services and therefore reduces the risk of SQL injection as malformed and wrong request parameters will be ignored. /5/

The second part is the function where the logic is handled. The following figures details the logic flow of “getIncident” functions.

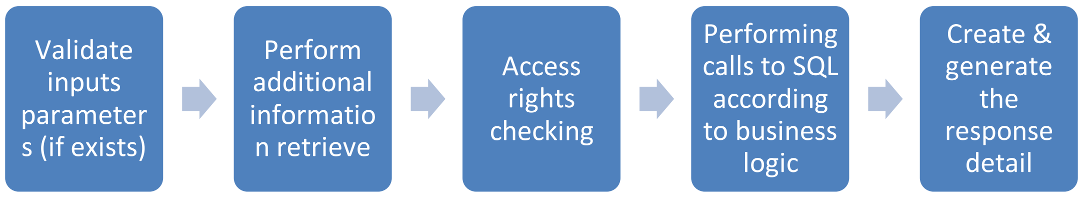
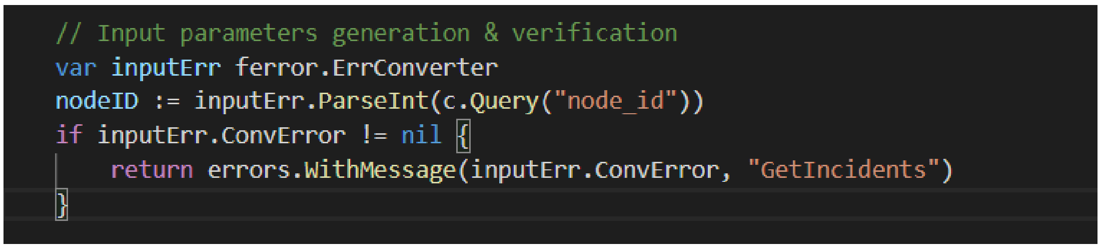

Validate inputs parameters (if exists)
Perform additional information retrieve
Access rights checking
Performing calls to SQL according to business logic
Create & generate the response detail

Figure 15. Logic flow inside controller function

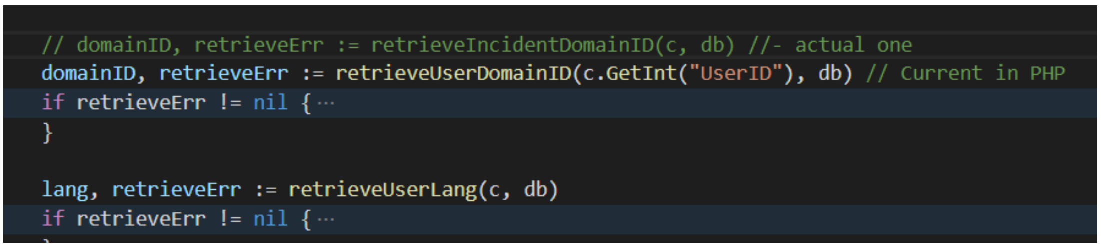
In the first step, input parameters required to perform the action will be validated and type checked. This ensures that a wrong type input will not be executed as SQL commands.



```go
// Input parameters generation & verification
var inputErr ferror.ErrConverter
nodeID := inputErr.ParseInt(c.Query("node_id"))
if inputErr.ConvError != nil {
    return errors.WithMessage(inputErr.ConvError, "GetIncidents")
}
```

Figure 16. input validation

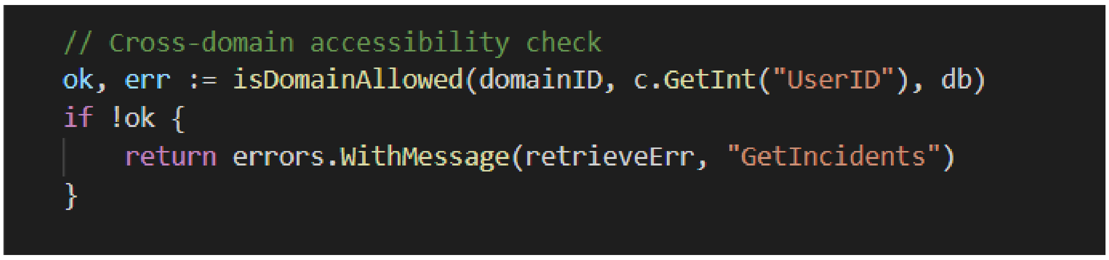
The following steps are to retrieve the required information needed to execute the services.



```go
// domainID, retrieveErr := retrieveIncidentDomainID(c, db) //- actual one
domainID, retrieveErr := retrieveUserDomainID(c.GetInt("UserID"), db) // Current in PHP
if retrieveErr != nil {···
}

lang, retrieveErr := retrieveUserLang(c, db)
if retrieveErr != nil {···
```

Figure 17. Retrieving additional information for the request

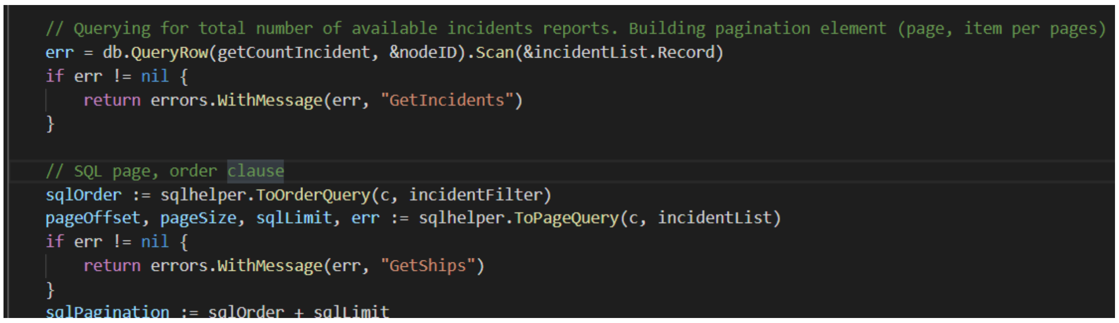
On the access right checking, the services will be checked if the owner of the processing request has the right to perform/access the information. At this stage, the user is authenticated but not yet authorized by the services to perform his request.



```go
// Cross-domain accessibility check
ok, err := isDomainAllowed(domainID, c.GetInt("UserID"), db)
if !ok {
    return errors.WithMessage(retrieveErr, "GetIncidents")
}
```

Figure 18. Domain check - user authorisation check

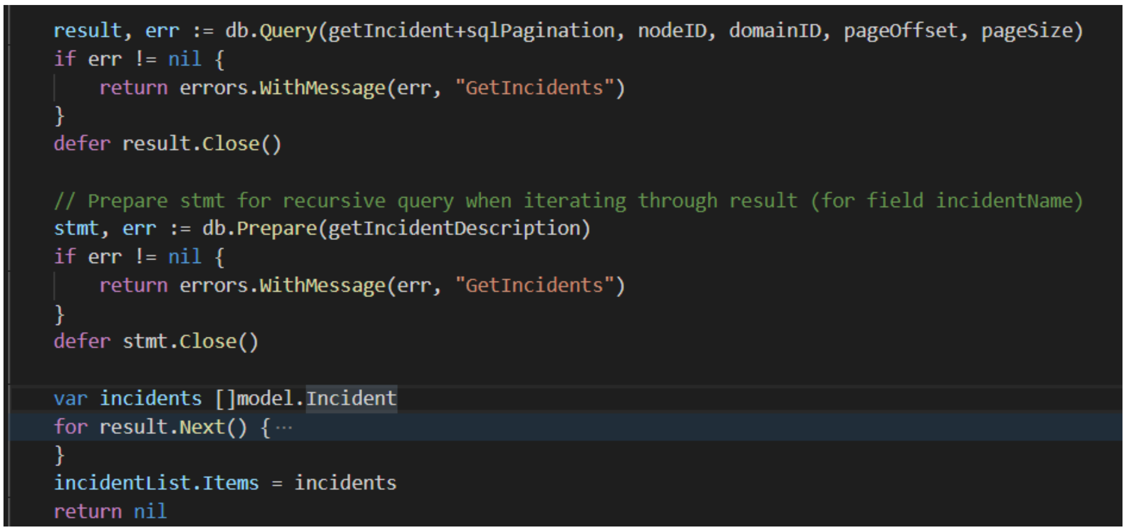
In addition to retrieve the required information, this is also the step where the filters, sorting and paging functionality added to the services. With the aim of increasing maintainability and keeping the code concise, it is advised that the implementation of these functionalities should be written separately and will be called by the main controller if needed. The following figure shows how the paging, sorting and filtering function called inside a controller.



```go
// Querying for total number of available incidents reports. Building pagination element (page, item per pages)
err = db.QueryRow(getCountIncident, &nodeID).Scan(&incidentList.Record)
if err != nil {
    return errors.WithMessage(err, "GetIncidents")
}

// SQL page, order clause
sqlOrder := sqlhelper.ToOrderQuery(c, incidentFilter)
pageOffset, pageSize, sqlLimit, err := sqlhelper.ToPageQuery(c, incidentList)
if err != nil {
    return errors.WithMessage(err, "GetShips")
}
sqlPagination := sqlOrder + sqlLimit
```

Figure 19. Sorting, filtering implementation

The following steps are to execute the SQL command and assemble the needed resource for the final outputs.



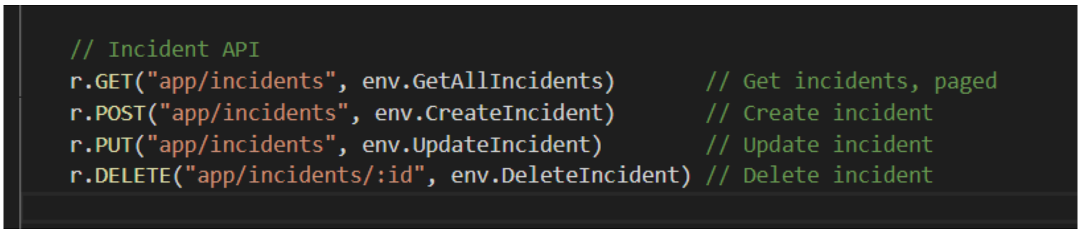
```go
result, err := db.Query(getIncident+sqlPagination, nodeID, domainID, pageOffset, pageSize)
if err != nil {
    return errors.WithMessage(err, "GetIncidents")
}
defer result.Close()

// Prepare stmt for recursive query when iterating through result (for field incidentName)
stmt, err := db.Prepare(getIncidentDescription)
if err != nil {
    return errors.WithMessage(err, "GetIncidents")
}
defer stmt.Close()

var incidents []model.Incident
for result.Next() {···
}
incidentList.Items = incidents
return nil
```

Figure 20. Querying for result & assemble response

After the response is generated from the controller, it is then passed back to the handler where the final modification to the response is made.

5.2.3 Register the Endpoint at “main.go”

The final step after having the controller and handler set up is to register the endpoints in “main.go”. This process is made very simple thanks to Gin framework as each endpoint will only need a single line.



```go
// Incident API
r.GET("app/incidents", env.GetAllIncidents)       // Get incidents, paged
r.POST("app/incidents", env.CreateIncident)       // Create incident
r.PUT("app/incidents", env.UpdateIncident)        // Update incident
r.DELETE("app/incidents/:id", env.DeleteIncident) // Delete incident
```

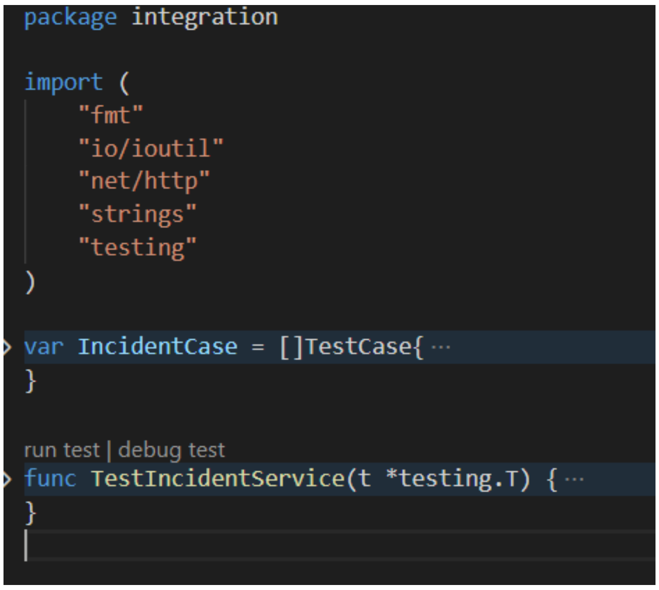
Figure 21. Endpoint declaration

6 TESTING

Package testing provides support for the automated testing of Go packages. It is intended to be used in concert with the “go test” command, which automates the execution of any function of the form “func TestXxx(*testing.T)” where Xxx does not start with a lowercase letter. The function name serves to identify the test routine. /6/

Within these functions, the developer can use the Error, Fail or related methods to signal failure.

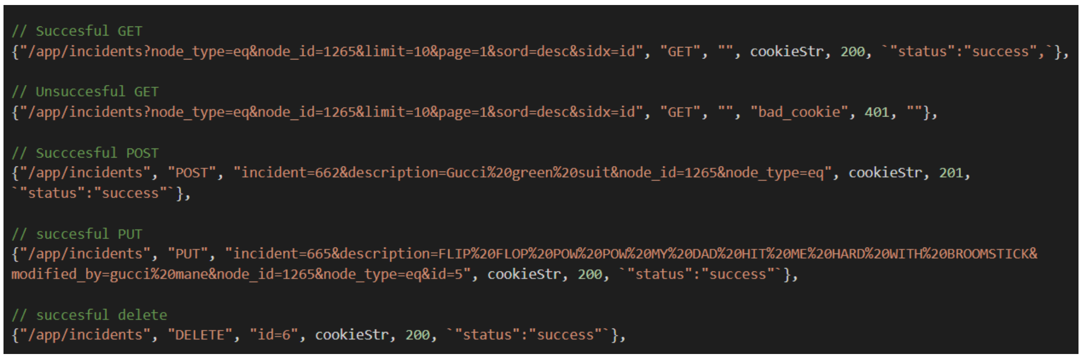
Writing a new test suite requires a file needs to be created whose name ends “_test.go” that contains the “TestXxx” functions as described here. The file is put in the same package as the one being tested. The file will be excluded from the regular package builds but will be included when the “go test” command is run. The structure of the table test on Incident services will look as follows:



```
package integration

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "strings"
    "testing"
)

var IncidentCase = []TestCase{ ···
}

run test | debug test
func TestIncidentService(t *testing.T) { ···
}
```

Figure 22. Testing file structure

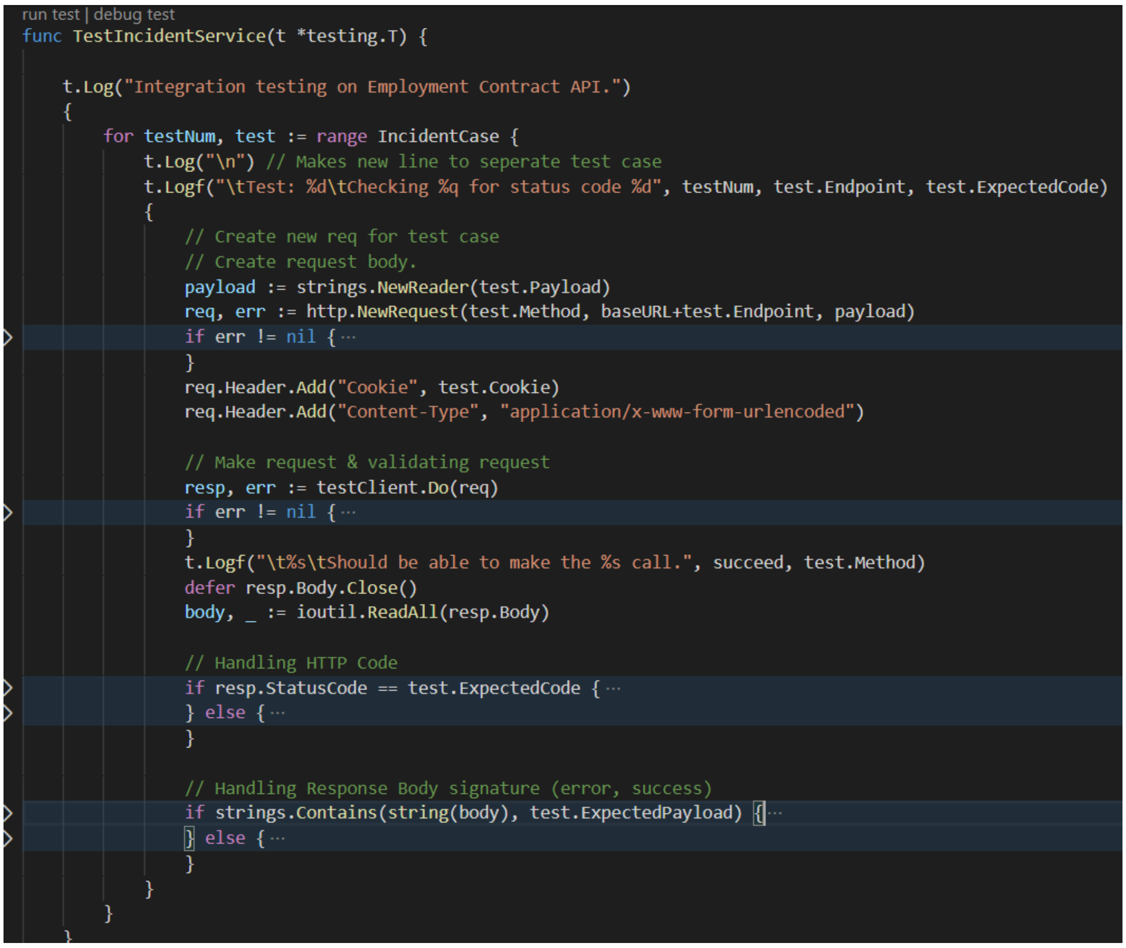
Inside the “IncidentCase” block, there are testcases that contain the request, the requests additional parameters and the signature of a successful operations.



```
// Succesful GET
{"/app/incidents?node_type=eq&node_id=1265&limit=10&page=1&sord=desc&sidx=id", "GET", "", cookieStr, 200, `"status":"success",`},

// Unsuccesful GET
{"/app/incidents?node_type=eq&node_id=1265&limit=10&page=1&sord=desc&sidx=id", "GET", "", "bad_cookie", 401, ""},

// Succcesful POST
{"/app/incidents", "POST", "incident=662&description=Gucci%20green%20suit&node_id=1265&node_type=eq", cookieStr, 201,
`"status":"success"`},

// succesful PUT
{"/app/incidents", "PUT", "incident=665&description=FLIP%20FLOP%20POW%20POW%20MY%20DAD%20HIT%20ME%20HARD%20WITH%20BROOMSTICK&
modified_by=gucci%20mane&node_id=1265&node_type=eq&id=5", cookieStr, 200, `"status":"success"`},

// succesful delete
{"/app/incidents", "DELETE", "id=6", cookieStr, 200, `"status":"success"`},
```

Figure 23. A series of test inside table test for incident service

The testing function will then create a client, make a call with provided information in each test case and then compare the actual result of the calls with the provided signatures. If the expected data matches the output the test cases are successful. If not, one of the break points put at every step will be triggered and we will get the detailed explanation.



```
run test | debug test
func TestIncidentService(t *testing.T) {

    t.Log("Integration testing on Employment Contract API.")
    {
        for testNum, test := range IncidentCase {
            t.Log("\n") // Makes new line to seperate test case
            t.Logf("\tTest: %d\tChecking %q for status code %d", testNum, test.Endpoint, test.ExpectedCode)
            {
                // Create new req for test case
                // Create request body.
                payload := strings.NewReader(test.Payload)
                req, err := http.NewRequest(test.Method, baseURL+test.Endpoint, payload)
                if err != nil {···
                }
                req.Header.Add("Cookie", test.Cookie)
                req.Header.Add("Content-Type", "application/x-www-form-urlencoded")

                // Make request & validating request
                resp, err := testClient.Do(req)
                if err != nil {···
                }
                t.Logf("\t%s\tShould be able to make the %s call.", succeed, test.Method)
                defer resp.Body.Close()
                body, _ := ioutil.ReadAll(resp.Body)

                // Handling HTTP Code
                if resp.StatusCode == test.ExpectedCode {···
                } else {···
                }

                // Handling Response Body signature (error, success)
                if strings.Contains(string(body), test.ExpectedPayload) {···
                } else {···
                }
            }
        }
    }
```

Figure 24. Example of the test function of Incident service

7 CONCLUSIONS

The primary aim of the thesis was to create a baseline for the new backend written in Go. The services that were selected in PHP to be rewritten in Go has been documented and has its traits as well as behaviors modified in RAML. The signatures of those APIs have been changed to conform with RESTful standard when developing APIs.

Regarding the implementation, the RAML specification files modelling the PHP services has been implemented successfully in Go. Additionally, an overall folders structure and the application structure has been established in Go, creating a guideline for future developers to follow the implementing tasks.

REFERENCES

/1/: Ryazanov, Mikhail. 2020. Go Programming Language Wikipedia. Accessed 5/2020 https://en.wikipedia.org/wiki/Go_(programming_language)

/2/: Peabody, Brad. 2020. Server-side I/O Performance: Node vs. PHP vs. Java vs. Go. Accessed 5/2020. https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go

/3/: Diyan, Alexey. 2020. Golang Web Framework Comparison. Accessed 5/2020 https://github.com/diyan/go-web-framework-comparison/blob/master/README.md

/4/: RAML developers. Introduction to RAML. Accessed 5/2020. https://raml.org/

/5/: Shan, X.D.S. Representational state transfer. Accessed 5/2020. https://en.wikipedia.org/wiki/Representational_state_transfer

/6/: Manico, J. & Saad, E., OWASP, Web Service Security Cheat Sheet. Accessed 5/2020 https://cheatsheetseries.owasp.org/cheatsheets/Web_Service_Security_Cheat_Sheet.html

/7/: Golang Developer Teams, Golang Orgs. Package testing document. Accessed 5/2020. https://golang.org/pkg/testing