GoForMail

Technical Guide

Student 1: Andre Rafael Cruz da Fonseca **Student ID:** 21460092

Student 2: Sean Albert Dagohoy **Student ID:** 21392656

Project Supervisor: Stephen Blott

CSC1097 - Final Year Project Dublin City University

Last Updated: May 2nd 2025

Table of Contents

1. Introduction

1.1 Overview

GoForMail is a Mailing List Manager, written in Go.

The project is similar to other applications such as GNU/Mailman but features a more modern interface. It was written to serve Redbrick, by connecting to their Postfix instance.

1.2 Problems Encountered

Problem	Solution
Testing in Go is quite limited as you can only mock structs that you created. You can't mock non struct functions.	Ignored unit testing handling for some errors. Additionally, made the app very object oriented to be able to mock more functions.
CLI auto completion does not support native Go flag support	No CLI auto completion was able to be added.
Due to how NextJS handled routing, the statically compiled website would not route correctly when ran with the go app	Needed to add ".html" to every page to allow it to rout correctly
Due to the project being fully client sided and also statically compiled, could not use NextJS' server-side fetching for getting data	Use NextJS' client-side fetching hook useSWR

1.3 Future Work

1.3.1 Better Logging

Audit Logging- Logs could be kept of which user does what actions regarding users, lists and emails. This helps increase accountability for users with access to the app.

General Logging- Logs could be kept for any automatic actions (receiving and

sending mails), as well as any errors, in both std.out (for real time monitoring) and in a text file (in order to access later, or in the event of a crash).

1.3.2 UI Improvements

Dark Mode Toggle- The UI could give users an option of using either dark or light mode. This would appeal to more users' preferences, as well as help with accessibility issues.

Lock Actions When User Has No Perms- By 'graying out' or removing any buttons that the user can't press, it prevents confusion from a user who may not understand why certain actions aren't working.

1.3.3 Better Feedback

Send confirmation emails on queue/send- GoForMail could send an email back to the sender to act as confirmation that the email has been received and will either be queued or sent. Additionally, once an email has been approved the sender could be sent a confirmation email once again. This helps email senders stay in the loop, even if they don't have access to the app's interfaces.

Send emails on password reset- In order to give users who have forgotten their password a chance to reset it, an email could be sent with a reset code.

1.3.4 Better Stability

Reconnect to Database- If the database shuts down while the app is still running, it may cause it to have unpredictable behaviour. Some logic could be implemented to reconnect to the database when it is available again, as well as ensure data is rejected or stored in memory/filesystem while waiting.

Graceful Shutdown- If the app gets shut down in the middle of talking to postfix, talking to the database or fulfilling a REST request, it may cause it to have unpredictable behaviour. To combat this, some logic could be implemented to create a more graceful shutdown.

2. Research

2.1 Learning Go

Learning Go was the main point of research in our project. With DCU not teaching it and neither of us having ever coded using it before, we both got to learn a lot.

Go seemed quite confusing at the beginning, especially the testing. We both came from a background of testing in Java, where everything is a mockable object- including things you don't manage. Due to this, it took a while to adapt to the language's idioms.

However, after spending more time with Go, we came to really enjoy some of its aspects. One example of this is being able to return multiple parameters from a single function- something that has come in very handy very often. Additionally, the ease with which we can start new threads by using goroutines.

2.2 NextJS Static Compilation

With NextJS being a popular framework, NextJS was used for the project. We have never touched nextJS before and we thought the 4th year project would be the perfect time to try and learn NextJS.

While NextJS has amazing features, we found that it can be a little harder to work with when statically compiling the website and serving it on a fileserver compared to when just running NextJS as a node server.

2.3 LMTP & SMTP

Having never learned anything about LMTP & SMTP while also basing our project around the protocols, this research was definitely one of the most important and biggest tasks to tackle.

We found finding useful resources around these protocols to be difficult. It was clear that there weren't too many people who looked into these protocols, especially LMTP, for their own projects. We did eventually find useful resources like the postfix documentation and rfc2033. We also looked into GNU/Mailman's implementation to see if we could draw inspiration from it. Another large part of the research was fidgeting with postfix for ourselves, figuring out how it worked with our own servers at home.

2.4 Authentication Systems

Having only a small amount of knowledge about authentication systems, it was a big learning opportunity to see how authentication is generally handled. In particular, JWTs were completely new to us.

Previously, we had created basic authentication systems. However, these stored passwords in plain text and had non-expiring tokens. Due to this, learning about how to salt and hash passwords safely in Go was great, and learning about self-expiring tokens that don't need to be stored (JWTs) was even better.

2.5 Command-Line Interfaces

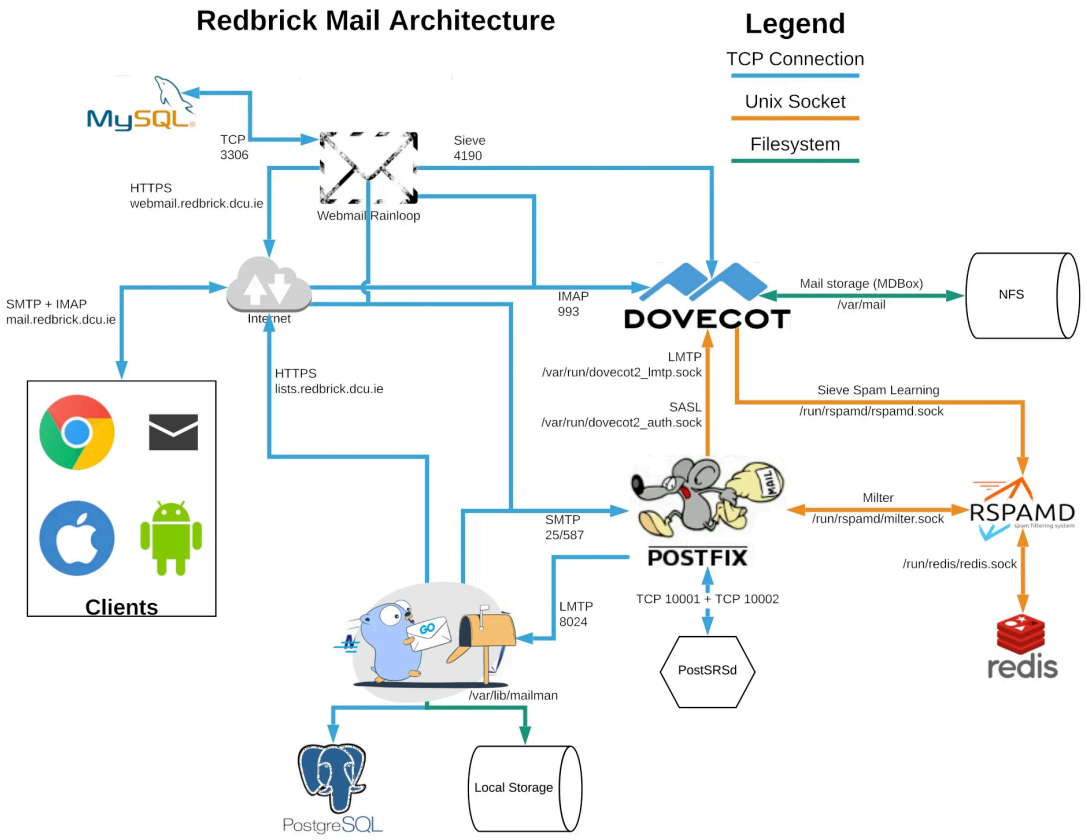
Prior to this project, neither of us had ever written a command line interface- and we didn't even know if it was possible to include it in the same executable as the main app.

This part of the project allowed us to learn more about how Go handles arguments and flags and made us think about how we can reutilise business logic for 2 separate entry points (CLI and REST).

3. Design

3.1 System Architecture

GoForMail acts as a microservice in a chain of other services. A diagram below shows how it connects with Postfix and PostgreSQL in order to create a fully functional system.



GoForMail receives and sends emails through postfix, changing the recipients to the adequate ones for any given list.

3.2 GoForMail - High Level Overview

The main GoForMail app is split into different packages. Each one is responsible for a different role, however none of them does anything by themselves.

DB Repository- The database repository keeps only one connection open to save resources, as well as ensures all statements are prepared to prevent sql injection and other unwanted behaviour.

Service- The service layer is responsible for handling all the business logic of the application. If anything needs to be done, it goes through the service layer.
Mail Controller- The mail controller is responsible for receiving and sending mail. It also periodically checks if more mail needs to be sent.

REST Controller- The REST controller is responsible for receiving and replying to any incoming requests. While it may not know how to process them, it is responsible for translating the request into a task and the result into a valid response.

CLI Controller- The CLI controller is responsible for processing any responses received from the command line. Similarly to the REST Controller, it may not know how to process them but it is responsible for translating a CLI command into a task and printing out its result.

3.3 GoForMail - Low Level Overview

3.3.1 DB Repository

The DB repository is a single struct, which other services are able to call, that spans across 5 Go files.

```
internal
 └──db
     ├──conn.go
     ├──emails.go
     ├──lists.go
     ├──users.go
     └──error.go
```

`conn.go` handles the creation of the struct, as well as the migration of the database. It connects to the database using Go's `database/sql` and `lib/pq`. A big framework wasn't chosen as the goal was to learn more about Go and that is best achieved with less abstraction.

`error.go` does a lot more simpler stuff, it just parses database errors so that other components don't need to worry about database logic.

`emails.go`, `lists.go` and `users.go` all handle database operators requested by other components of the application, for their specific types. This is done through the use of Go's prepared statements. An example is below, for getting a list:

```
db.conn.QueryRow(`
     SELECT name, recipients, mods, approved_senders, locked
     FROM lists WHERE id = $1
     `, id,
).Scan(&list.Name, pq.Array(&list.Recipients),
     pq.Array(&list.Mods),pq.Array(&list.ApprovedSenders),
     &list.Locked);
```

3.3.2 Service

The service layer is a group of structs, one per data type., which other services are able to call, that spans across 5 Go files.

```
internal
 └──service
     ├──auth.go
     ├──emails.go
     ├──lists.go
     └──users.go
```

`auth.go` is a bit different from the rest, as it does not have a corresponding data type. Instead, it is responsible for validating JWT tokens every time someone uses the REST API. It is also responsible for generating JWT tokens when a user logs in. Password hashing is handled by `x/crypto` and token management by `golang-jwt/jwt/v5`.

`emails.go`, `lists.go` and `users.go` all handle middle logic between a controller and the database repository. This means they are responsible for ensuring that operations are valid and transforming any needed data. This means they take in a request object and return a response object. An example can be seen below for some of the users' function definitions:

```
// GetUser returns a response object
func (man *UserManager) GetUser(id int) (*model.UserResponse,
*util.Error)

// CreateUser takes in a request object
func (man *UserManager) CreateUser(user *model.UserRequest) (int,
*util.Error)
```

3.3.3 Mail Controller

The Mail controller is a group of 3 structs. These can be seen as different files below.

```
internal
 └──mail
     ├──mtp.go
     ├──receiver.go
     └──sender.go
```

`mtp.go` handles all the transfer protocol logic. This includes both LMTP and SMTP. Due to the lack of good LMTP libraries, we had to implement it from scratch.

`receiver.go` creates a tcp socket and listens for new connections in a loop. When a connection comes, it uses mtp's logic to receive that email and place it in the database. It then attempts to send it, if the sender is not in the middle of sending.

`sender.go`, loops checking for emails which need to be sent. When it finds some, it starts sending them. Upon sending them all, it checks again if there are any new ones. If there are none left to send, it goes back to waiting.

3.3.4 Rest Controller

The mail controller is a single struct, which other services are able to call, that spans across 5 Go files.

```
internal
 └──db
     ├──controller.go
     ├──emails.go
     ├──lists.go
     ├──users.go
     ├──users.go
     └──auth.go
```

`controller.go` handles the creation of the struct, as well as the assignment of endpoints and listening for requests. This is done through Go's `net/http`. A big framework wasn't chosen as the goal was to learn more about Go and that is best achieved with less abstraction.

`emails.go`, `lists.go`, `users.go` and `auth.go` all handle REST requests and send them to their corresponding business logic in the app. The handling is done by assigning each request method a different function for each endpoint, as seen below.

```
ctrl.mux.HandleFunc("/api/list/", func(w http.ResponseWriter, r
*http.Request) {
     if r.URL.Path != "/api/list/" {
          handleUnknownMethod(w, r)
          return
```

```
	}
	switch r.Method {
	case "GET":
		ctrl.getList(w, r)
	case "POST":
		ctrl.postList(w, r)
	case "PATCH":
		ctrl.patchList(w, r)
	case "DELETE":
		ctrl.deleteList(w, r)
	default:
		handleUnknownMethod(w, r)
	}
})
```

3.3.5 CLI Controller

The mail controller is a single struct, which other services are able to call, that spans across 5 Go files.

```
cmd
 └──goformail
     └──cli
         ├──root.go
         ├──get.go
         ├──create.go
         ├──update.go
         ├──delete.go
         └──approve.go
```

`root.go` handles sending the execution to the correct router based on the command's first arg.

`get.go`, `create.go`, `update.go`, `delete.go` and `approve.go` all handle calling the right part of the code based on the command's arguments and flags. Flags are parsed through Go's `flag` pkg. A big framework wasn't chosen as the goal was to learn more about Go and that is best achieved with less abstraction. Below is an example of the parsing of arguments and flags.

```
if len(args) < 1 {
	fmt.Println(unknownUpdateCommand)
	return
}
```

```
args = append(args[1:], args[0])

perms := stringSlice{}
cmd := flag.NewFlagSet("user", flag.ExitOnError)
email := cmd.String("email", "", "")
password := cmd.String("password", "", "")
cmd.Var(&perms, "permission","")
parseArgs(cmd, args)
if cmd.NArg() != 1 {
      fmt.Println(unknownUpdateCommand)
      return
}

id := convertId(args[len(args)-1])
```

The first arg is thrown onto the end as flag.Parse() ignores any flags after the first arg.

3.4 Web UI - Overview

The web ui for GoForMail was created with NextJS and statically compiled to be served by the go app. It is one way of managing mailing lists, the other being the CLI. Users must log in to be able to manage mailing lists. When users are logged in, they receive a session token which lasts for 72 hours. Once those 72 hours are up, they are kicked out of the dashboards and must log in again. If a user wants to sign up, they must communicate with a known user that has the correct permissions to create the account for them. The reason there is no sign up is because this service is intended only for moderator use, not for the public.

4. Testing

4.1 Adhoc Testing

During development, frequent adhoc testing was performed. In order to confirm that any new features were functional, the person implementing them always checked in a real setting.

Additionally, during any merge requests, the reviewer had to spin up the changes on their machine and test them in order to know that everything worked. This ensured both people tested every feature before merging it to main.

Adhoc testing consisted of sending emails through postfix, interacting with the app through the REST API and CLI, testing integration between the web UI and the app, as well as altering data in the db to create edge cases.

4.2 Unit Testing in Go

Unit testing was set up in the main app in order to ensure features that previously worked did not stop working. These tests aimed to ensure that every functionality of every function was accounted for and to be specific enough that tracking down causes for failure is simple.

These tests are run frequently as every merge request requires them to run on the pipeline before merging with the main branch. This ensured we would always know if a test began failing due to a change in code logic.

The main app's unit tests are written using Go's standard testing library but enhanced using testify. Testify allows for quicker writing of tests (through its assertions module), as well as higher test flexibility (through its mocking library).

Finally, the database query tests use testcontainers (and hence require access to rootless docker). This is due to the lack of a sql mocking library for Go. This means that these unit tests are tested against a real database during their unit tests, adding extra safety.

4.3 Unit Testing in NextJS

Unit testing was set up within the web ui to ensure that components deemed important are rendered properly and that certain components will only render based on certain conditions met. Like within the go unit tests, these tests also aim to ensure that previous features that worked did not stop working.

These tests are run frequently as they run within the pipelines when the branch is pushed, however, we generally run these tests locally first before pushing to the branch to ensure all tests pass.

The web ui uses "vitest" for testing. Once the backend and frontend were connecting, we used useSWR's mocking value to mock fetch requests. This allowed us to test for various conditions such as when the data is still loading, when the data was given but was invalid, and when the data was given and was valid.

There were also some tests done on certain interactions. For example, for dynamic inputs where a user could add another recipient or remove another recipient, Vitest allows you to trigger events. We then checked if what we expected would happen for example we expect a component gets removed when a button is pressed.

4.4 End 2 End Testing

End 2 End testing was planned but never got executed due to lack of time. It would've been similar to the integration tests, but also include running emails through postfix. Instead of using Go's built- in testing library, they would've likely been shell or javascript scripts and had their own pipeline job.