Project Title

Mailing List Microservices

submitted to the

Department of Master of Computer Applications

in

Partial fulfilment of the requirements

for the

Course: Go Programming

Course Code: MCAE33

By

Student Full Name	**USN**
Ajay Zad	**1MS21MC006**
Anup Kumar	**1MS21MC008**
Madasa Akanksha	**1MS21MC026**

Department of Master of Computer Applications

RAMAIAH INSTITUTE OF TECHNOLOGY

(Autonomous Institute, Affiliated to VTU)
Accredited by National Board of Accreditation & NAAC with ’A+’ Grade MSR Nagar, MSRIT Post, Bangalore-560054 www.msrit.edu

RAMAIAH
Institute of Technology

DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS

CERTIFICATE

This is to certify that the project entitled **Mailing List Microservices** is carried out by

	Student Name	USN
1)	Ajay Zad	1MS21MC006
2)	Anup Kumar	1MS21MC008
3)	Madasa Akanksha	1MS21MC026

Students of 2nd Semester, in partial fulfilment for the Course: Go Programming,

Course code: MCAE33, during the academic year 2022.

Faculty In-charge **Head of the Department**

(Dr. S Seema)

Name of Examiners **Signature with Date**

1.

2.

ABSTRACT

Platform specific software/application have been a major set-back to the integration of heterogenous applications. A mailing list microservices can be used to provide ready-made functionalities to a traditional mailing list application. The data responsible for all processing in the mailing list microservices is managed by the Relational Database Management System. The communication between client and the server is done using gRPC API client and gRPC API Server which allows the client and the server to communicate transparently and develop connected systems. Mailing list can be managed efficiently by transmitting or sending the mail lists in a particular batch format with each batch having limited size to hold the mails in a serial wise order.

1) INTRODUCTION

A microservice typically implements a set of distinct features or functionality. Each micro-service is a mini-application that has its own architecture and business logic. A micro-service architecture patter significantly changes the relationship between the application and the database. Microservices is used to separate the application into manageable chunks or services. Each service has a well-defined boundary in the form of an RPC or a message driven API. Therefore, individual services can be considered to be faster to develop and substantially easier to understand and mange the things. Microservices architecture also lets you scale each service independently. We can deploy the number of instances of each service that satisfy its capacity and availability constraints. When we scale services independently, it helps the increase the availability and the reliability of the entire system.

This technology connects programs to each other across distant points on the global maps, transport large amount of data more efficiently and cheaply than ever before, thereby promoting software portability and reusability in the application that operates over the internet. Mailing list manager have a venerable history on the internet and they are an excellent vehicle for the distributing focused information to the interested receptive public. Within the context of this, a mailing list manager application is developed, and the most basic features of mailing list managers are incorporated into the application. It must be noted that some of the features/functions of the application runs on the servers elsewhere on the internet. When the server is updated, all the clients worldwide see the new capabilities. The application is quite portable so that we can run the same applications on some different type/kind of computer system from elsewhere or from any corner of the world.

1.a) PROBLEM DEFINATION

Managing mailing list using microservices. We have implemented the solution to the given problem using JSON API Server, gRPC API Server, gRPC API client, Protocol buffers, SQLite 3(Database) and CLI.

SYSTEM ANALYSIS

It is a process of collecting and interpreting facts, identifying the problems and decomposition of a system into its components. System analysis is conducted for the purpose of studying a system or its parts in order to identify its objectives. It is a problem-solving technique that improves the system and ensure that all the components of the system work efficiently to accomplish the task. Basically, any analysis specifies what the system to do.

SYSTEM DESIGN

It is a process of planning a new business system or replacing an existing system by defining its components or modules to satisfy the specific requirements. Before planning, you need to understand the old system thoroughly and determine how computers can be used in order to operate efficiently. System design focuses on how to accomplish the objective or task that is given the system/computer.

2) SYSTEM SPECIFICATIONS

Software Requirement Specification

Operating System: Windows 10, macOS

Coding Language: Golang

Database: SQLite 3

Software used: Visual Studio Code

Software Requirement Specification

System: i5 processor

Hard Disk: 500 GB

Processor speed: 1.2 GHz

RAM: 4 to 8 GB

3) FUNCTIONAL REQUIREMENTS

SQLite 3:

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is one of the fastest-growing database engines around, but that's growth in terms of popularity, not anything to do with its size. The source code for SQLite is in the public domain. SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. It is a database, which is zero-configured, which means like other databases you do not need to configure it in your system.

SQLite engine is not a standalone process like other databases, you can link it statically or dynamically as per your requirement with your application. SQLite accesses its storage files directly.

JSON API Server:

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. It is a database, which is zero-configured, which means like other databases you do not need to configure it in your system.

SQLite engine is not a standalone process like other databases, you can link it statically or dynamically as per your requirement with your application. SQLite accesses its storage files directly.

gRPC API Client and Server:

gRPC or Google Remote Procedure Call is a modern open-source high-performance RPC framework that can run in any environment. It can efficiently connect services in and across data centres with pluggable support for load balancing, tracing, health checking and authentication.

RPC stands for **remote procedure calls** which are quite same as the normal function in Python. They are messages that server sends to the remote system to get the task (or routine) completed.

Google's RPC provides the facility of a smooth and efficient communication between the services. It can be utilized in the different ways -

- Generating efficient client libraries.
- Efficiently connecting polyglot services in microservices style architecture.
- Connecting mobile devices, browser client to backend services.

Protocol Buffers:

It is a technique of serialize the data into Binary stream in fast and efficient manner. It is designed as platform-neutral format and abstract data into a language. It is used for the inter-machine communication and RPC (Remote Procedure calls). To get started with the protocol buffer, we should have the knowledge of at-least one programming language to describe the shape of the data. However, the best thing about the protocol buffer is that it supports many programming languages such as Python, Java, C++, JavaScript, C, etc.

Protocol buffers are a defined interface for the serialization of structure data. It establishes the communication to data. It is platform and language independent. Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data - think XML, but smaller, faster, and simpler. You define how you want your data to be structured once.

Goroutines:

A goroutine is a lightweight thread managed by the Go runtime. Unlike the Operating System thread which consumes memory usually in the range of MBs to function properly**,** these goroutines consume memory only in the range of KBs**.** So, it's easy and fast for go runtime to create goroutines than threads for concurrency. Goroutines are not actual threads, these are multiplexed to actual OS threads or even you could assume that these goroutines are a kind of abstract layer over OS threads.

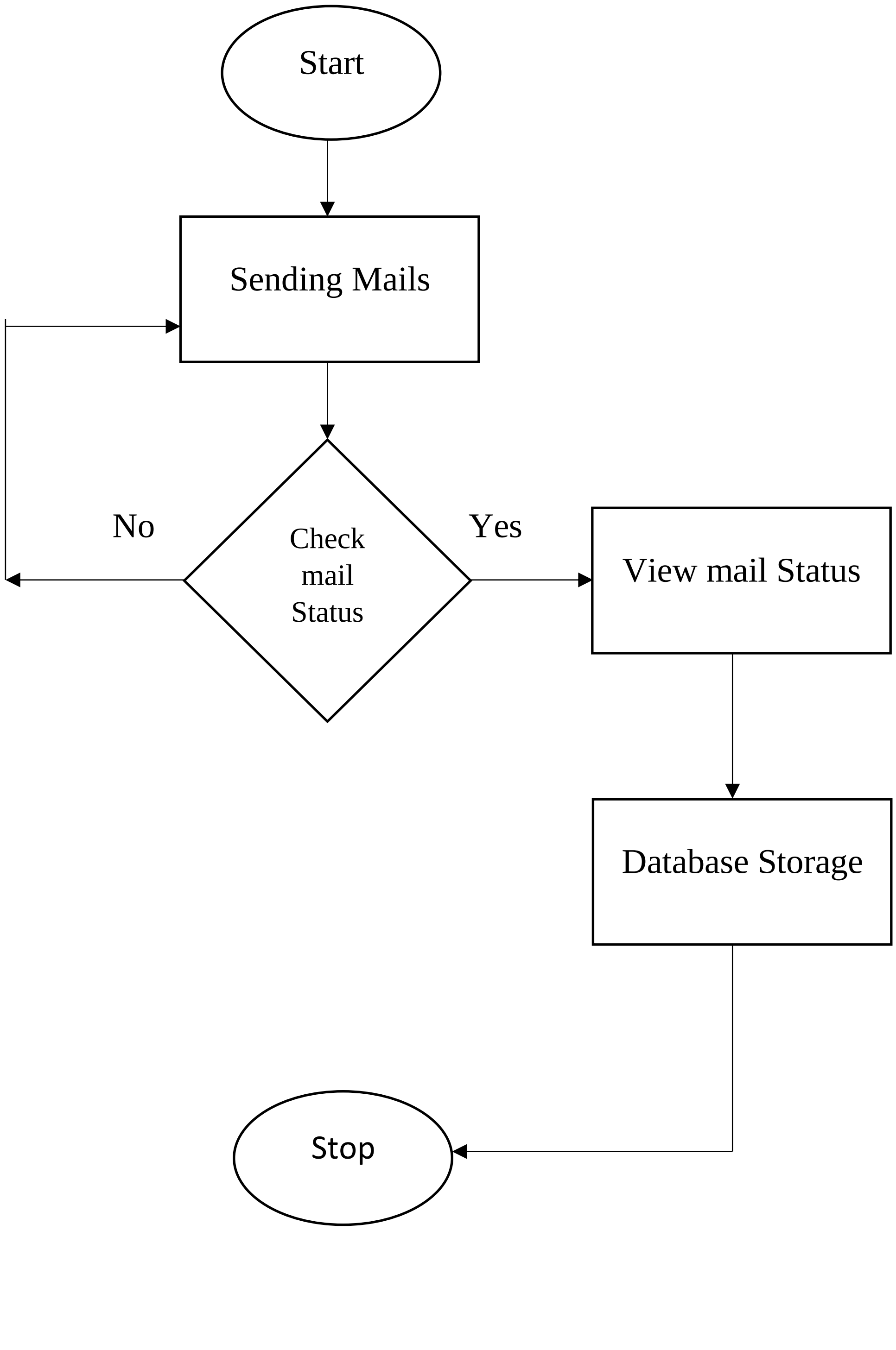
Due to its very small memory footprint, it's easy for go runtime to create thousands of goroutines in a fraction of milliseconds. But creating thousands of OS threads is heavily dependent on memory and the CPU, so it's obviously slow compared with goroutines.

Command Line Interface:

A CLI also called as Command Line Interface. It is an interface that allows the user to perform tasks by issuing commands in successive line and text or command-lines. More over CLI is used to run programs, manage computer files and interact with the computer system. Command-line interfaces are also called command-line user interface, console user interfaces and character user interface. CLI's accept as input commands that are entered by the keyboard, the commands invoked at the command prompt are then run by the computer.

.

4) DESIGN MODEL



5) IMPLEMENTATION

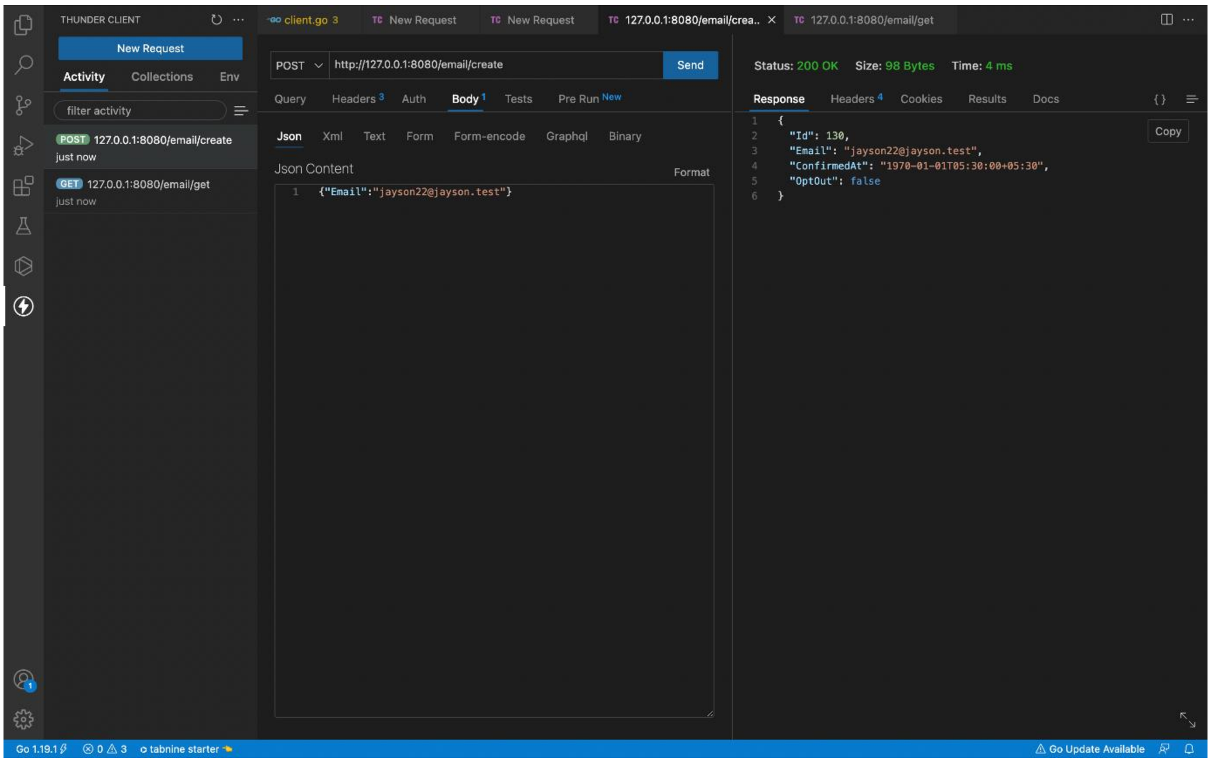
Client.go

```
package main
import (
	"context"
	"log"
	pb "mailinglist/proto"
	"time"
	"github.com/alexflint/go-arg"
	"google.golang.org/grpc"
	"google.golang.org/grpc/credentials/insecure"
)
func logResponse(res *pb.EmailResponse, err error) {
	if err != nil {
		log.Fatalf("  error: %v", err)
	}
	if res.EmailEntry == nil {
		log.Printf("  email not found")
	} else {
		log.Printf("  response: %v", res.EmailEntry)
	}
}
func createEmail(client pb.MailingListServiceClient, addr string) *pb.EmailEntry {
	log.Println("create email")
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	res, err := client.CreateEmail(ctx, &pb.CreateEmailRequest{EmailAddr: addr})
	logResponse(res, err)
	return res.EmailEntry
}
func getEmail(client pb.MailingListServiceClient, addr string) *pb.EmailEntry {
	log.Println("get email")
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	res, err := client.GetEmail(ctx, &pb.GetEmailRequest{EmailAddr: addr})
	logResponse(res, err)
	return res.EmailEntry
}
func getEmailBatch(client pb.MailingListServiceClient, count int, page int) {
	log.Println("get email batch")
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	res, err := client.GetEmailBatch(ctx, &pb.GetEmailBatchRequest{Count: 
int32(count), Page: int32(page)})
	if err != nil {
		log.Fatalf("  error: %v", err)
	}
	log.Println("response:")
	for i := 0; i < len(res.EmailEntries); i++ {
		log.Printf("  item [%v of %v]: %s", i+1, len(res.EmailEntries),
res.EmailEntries[i])
	}
}
func updateEmail(client pb.MailingListServiceClient, entry pb.EmailEntry) *pb.EmailEntry
{
	log.Println("update email")
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	res, err := client.UpdateEmail(ctx, &pb.UpdateEmailRequest{EmailEntry: &entry})
	logResponse(res, err)
	return res.EmailEntry
}
func deleteEmail(client pb.MailingListServiceClient, addr string) *pb.EmailEntry {
	log.Println("delete email")
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	res, err := client.DeleteEmail(ctx, &pb.DeleteEmailRequest{EmailAddr: addr})
	logResponse(res, err)
	return res.EmailEntry
}
var args struct {
	GrpcAddr string `arg:"env:MAILINGLIST_GRPC_ADDR"`
}
func main() {
	arg.MustParse(&args)
	if args.GrpcAddr == "" {
		args.GrpcAddr = ":8081"
	}
	conn, err := grpc.Dial(args.GrpcAddr,
grpc.WithTransportCredentials(insecure.NewCredentials()))
	if err != nil {
		log.Fatalf("did not connect: %v", err)
	}
	defer conn.Close()
	client := pb.NewMailingListServiceClient(conn)
	newEmail := createEmail(client, "main23@GMAILgo .999")
	newEmail.ConfirmedAt = 10000
	updateEmail(client, *newEmail)
	deleteEmail(client, newEmail.Email)
	getEmailBatch(client, 3, 1)
	getEmailBatch(client, 3, 2)
	getEmailBatch(client, 3, 3)
	getEmailBatch(client, 3, 5)
	getEmailBatch(client, 3, 6)
	getEmailBatch(client, 3, 7)
}
```

Grpcapi.go

```
package grpcapi
import (
	"context"
	"database/sql"
	"log"
	"mailinglist/mdb"
	pb "mailinglist/proto"
	"net"
	"time"
	"google.golang.org/grpc"
)
type MailServer struct {
	pb.UnimplementedMailingListServiceServer
	db *sql.DB
}
func pbEntryToMdbEntry(pbEntry *pb.EmailEntry) mdb.EmailEntry {
	t := time.Unix(pbEntry.ConfirmedAt, 0)
	return mdb.EmailEntry{
		Id:          pbEntry.Id,
		Email:       pbEntry.Email,
		ConfirmedAt: &t,
		OptOut:      pbEntry.OptOut,
	}
}
func mdbEntryToPbEntry(mdbEntry *mdb.EmailEntry) pb.EmailEntry {
	return pb.EmailEntry{
		Id:          mdbEntry.Id,
		Email:       mdbEntry.Email,
		ConfirmedAt: mdbEntry.ConfirmedAt.Unix(),
		OptOut:      mdbEntry.OptOut,
	}
}
func emailResponse(db *sql.DB, email string) (*pb.EmailResponse, error) {
	entry, err := mdb.GetEmail(db, email)
	if err != nil {
		return &pb.EmailResponse{}, err
	}
	if entry == nil {
		return &pb.EmailResponse{}, nil
	}
	res := mdbEntryToPbEntry(entry)
	return &pb.EmailResponse{EmailEntry: &res}, nil
}
func (s *MailServer) GetEmail(ctx context.Context, req *pb.GetEmailRequest)
(*pb.EmailResponse, error) {
	log.Printf("gRPC GetEmail: %v\n", req)
	return emailResponse(s.db, req.EmailAddr)
}
func (s *MailServer) GetEmailBatch(ctx context.Context, req *pb.GetEmailBatchRequest)
(*pb.GetEmailBatchResponse, error) {
	log.Printf("gRPC GetEmailBatch: %v\n", req)
	params := mdb.GetEmailBatchQueryParams{
		Page:  int(req.Page),
		Count: int(req.Count),
	}
	mdbEntries, err := mdb.GetEmailBatch(s.db, params)
	if err != nil {
		return &pb.GetEmailBatchResponse{}, err
	}
	pbEntries := make([]*pb.EmailEntry, 0, len(mdbEntries))
	for i := 0; i < len(mdbEntries); i++ {
		entry := mdbEntryToPbEntry(&mdbEntries[i])
		pbEntries = append(pbEntries, &entry)
	}
	return &pb.GetEmailBatchResponse{EmailEntries: pbEntries}, nil
}
func (s *MailServer) CreateEmail(ctx context.Context, req *pb.CreateEmailRequest)
(*pb.EmailResponse, error) {
	log.Printf("gRPC CreateEmail: %v\n", req)
	err := mdb.CreateEmail(s.db, req.EmailAddr)
	if err != nil {
		return &pb.EmailResponse{}, err
	}
	return emailResponse(s.db, req.EmailAddr)
}
func (s *MailServer) UpdateEmail(ctx context.Context, req *pb.UpdateEmailRequest)
(*pb.EmailResponse, error) {
	log.Printf("gRPC UpdateEmail: %v\n", req)
	entry := pbEntryToMdbEntry(req.EmailEntry)
	err := mdb.UpdateEmail(s.db, entry)
	if err != nil {
		return &pb.EmailResponse{}, err
	}
	return emailResponse(s.db, entry.Email)
}
func (s *MailServer) DeleteEmail(ctx context.Context, req *pb.DeleteEmailRequest)
(*pb.EmailResponse, error) {
	log.Printf("gRPC DeleteEmail: %v\n", req)
	err := mdb.DeleteEmail(s.db, req.EmailAddr)
	if err != nil {
		return &pb.EmailResponse{}, err
	}
	return emailResponse(s.db, req.EmailAddr)
}
func Serve(db *sql.DB, bind string) {
	listener, err := net.Listen("tcp", bind)
	if err != nil {
		log.Fatalf("gRPC server error: failure to bind %v\n", bind)
	}
	grpcServer := grpc.NewServer()
	mailServer := MailServer{db: db}
	pb.RegisterMailingListServiceServer(grpcServer, &mailServer)
	log.Printf("gRPC API server listening on %v\n", bind)
	if err := grpcServer.Serve(listener); err != nil {
		log.Fatalf("gRPC server error: %v\n", err)
	}
}
```

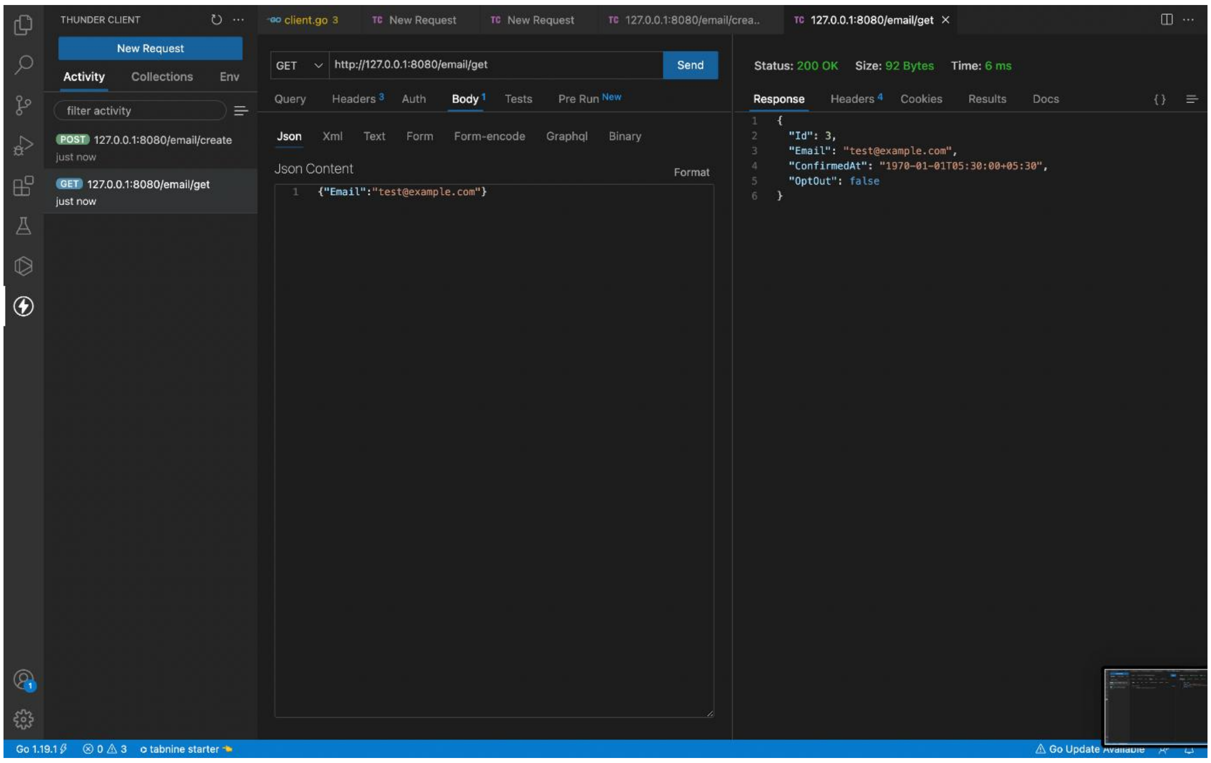
6) PROJECT SCREEN-SHOT


POST http://127.0.0.1:8080/email/create

```
{"Email":"jayson22@jayson.test"}
```

Status: 200 OK Size: 98 Bytes Time: 4 ms

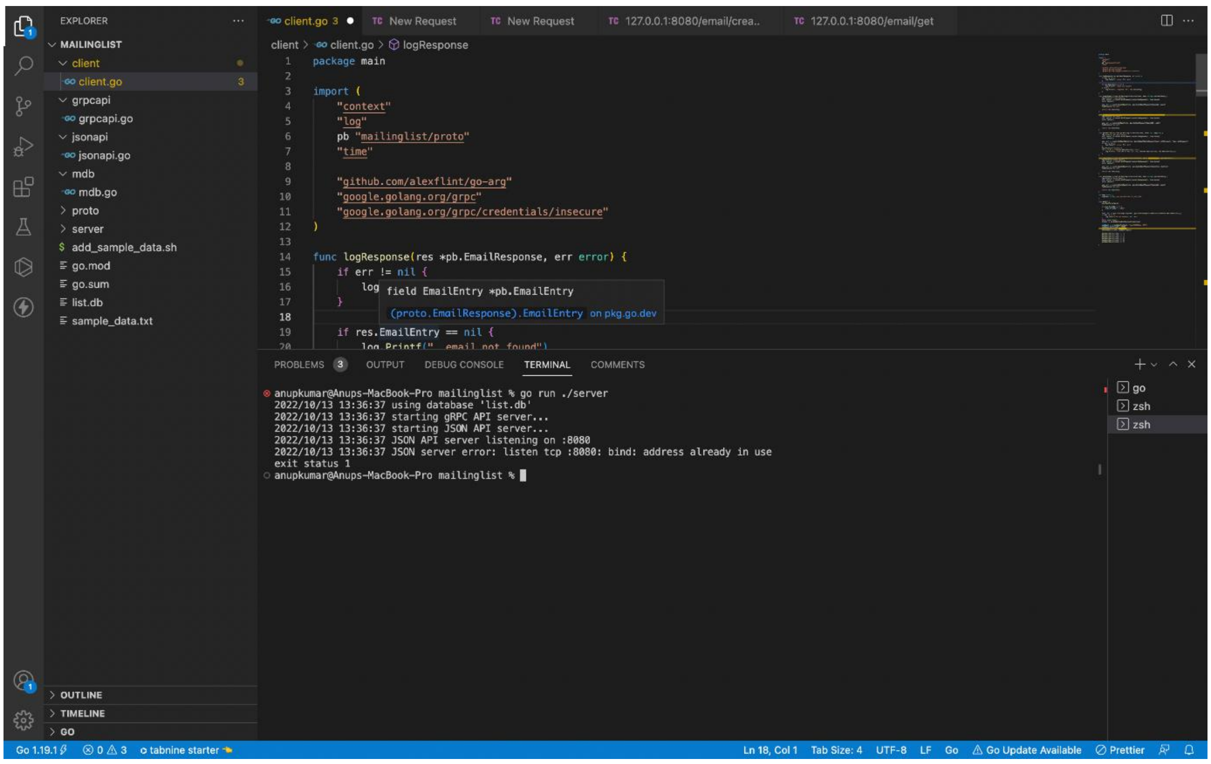
```
{
  "Id": 130,
  "Email": "jayson22@jayson.test",
  "ConfirmedAt": "1970-01-01T05:30:00+05:30",
  "OptOut": false
}
```


GET http://127.0.0.1:8080/email/get

```
{"Email":"test@example.com"}
```

Status: 200 OK Size: 92 Bytes Time: 6 ms

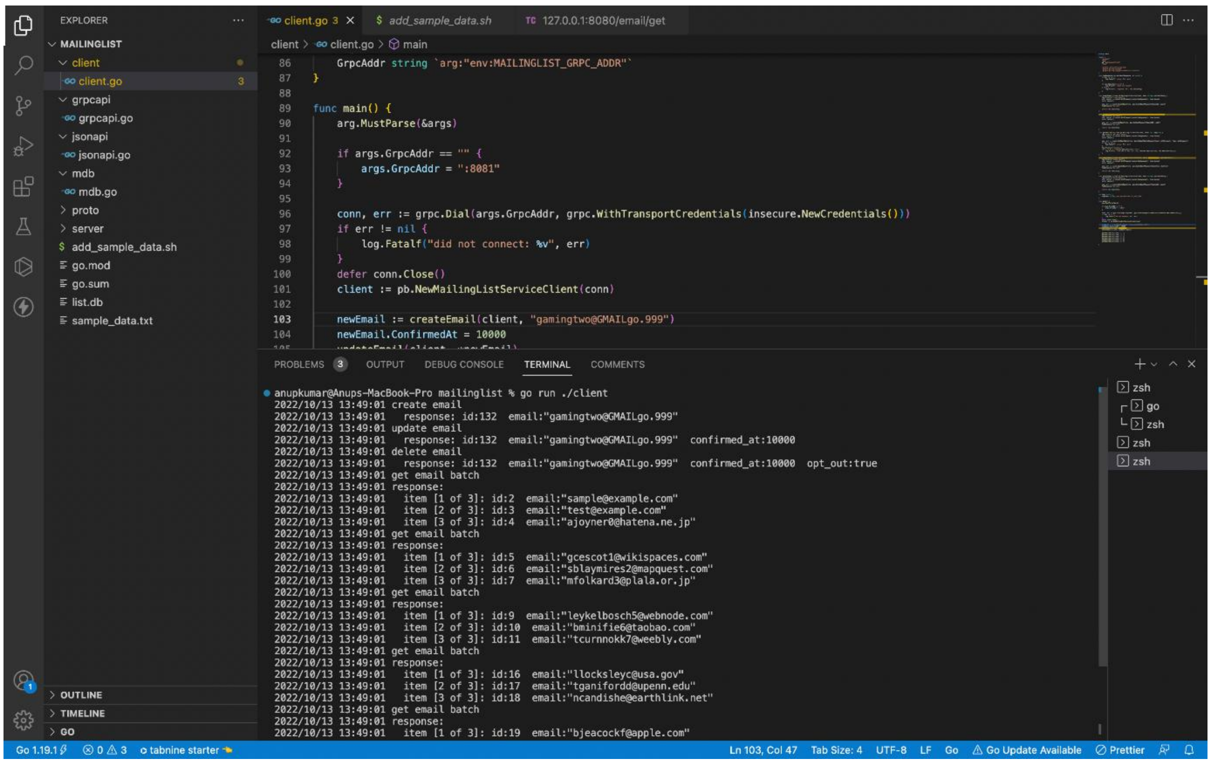
```
{
  "Id": 3,
  "Email": "test@example.com",
  "ConfirmedAt": "1970-01-01T05:30:00+05:30",
  "OptOut": false
}
```



```go
package main

import (
	"context"
	"log"
	pb "mailinglist/proto"
	"time"

	"github.com/alexflint/go-arg"
	"google.golang.org/grpc"
	"google.golang.org/grpc/credentials/insecure"
)

func logResponse(res *pb.EmailResponse, err error) {
	if err != nil {
		log
	}

	if res.EmailEntry == nil {
		log.Printf("  email not found")
```

```
field EmailEntry *pb.EmailEntry
(proto.EmailResponse).EmailEntry on pkg.go.dev
```

```
anupkumar@Anups-MacBook-Pro mailinglist % go run ./server
2022/10/13 13:36:37 using database 'list.db'
2022/10/13 13:36:37 starting gRPC API server...
2022/10/13 13:36:37 starting JSON API server...
2022/10/13 13:36:37 JSON API server listening on :8080
2022/10/13 13:36:37 JSON server error: listen tcp :8080: bind: address already in use
exit status 1
anupkumar@Anups-MacBook-Pro mailinglist %
```



```go
	GrpcAddr string `arg:"env:MAILINGLIST_GRPC_ADDR"`
}

func main() {
	arg.MustParse(&args)

	if args.GrpcAddr == "" {
		args.GrpcAddr = ":8081"
	}

	conn, err := grpc.Dial(args.GrpcAddr, grpc.WithTransportCredentials(insecure.NewCredentials()))
	if err != nil {
		log.Fatalf("did not connect: %v", err)
	}
	defer conn.Close()
	client := pb.NewMailingListServiceClient(conn)

	newEmail := createEmail(client, "gamingtwo@GMAILgo.999")
	newEmail.ConfirmedAt = 10000
```

```
anupkumar@Anups-MacBook-Pro mailinglist % go run ./client
2022/10/13 13:49:01 create email
2022/10/13 13:49:01   response: id:132  email:"gamingtwo@GMAILgo.999"
2022/10/13 13:49:01 update email
2022/10/13 13:49:01   response: id:132  email:"gamingtwo@GMAILgo.999"  confirmed_at:10000
2022/10/13 13:49:01 delete email
2022/10/13 13:49:01   response: id:132  email:"gamingtwo@GMAILgo.999"  confirmed_at:10000  opt_out:true
2022/10/13 13:49:01 get email batch
2022/10/13 13:49:01 response:
2022/10/13 13:49:01   item [1 of 3]: id:2  email:"sample@example.com"
2022/10/13 13:49:01   item [2 of 3]: id:3  email:"test@example.com"
2022/10/13 13:49:01   item [3 of 3]: id:4  email:"ajoyner0@hatena.ne.jp"
2022/10/13 13:49:01 get email batch
2022/10/13 13:49:01 response:
2022/10/13 13:49:01   item [1 of 3]: id:5  email:"gcescot1@wikispaces.com"
2022/10/13 13:49:01   item [2 of 3]: id:6  email:"sblaymires2@mapquest.com"
2022/10/13 13:49:01   item [3 of 3]: id:7  email:"mfolkard3@plala.or.jp"
2022/10/13 13:49:01 get email batch
2022/10/13 13:49:01 response:
2022/10/13 13:49:01   item [1 of 3]: id:9  email:"leykelbosch5@webnode.com"
2022/10/13 13:49:01   item [2 of 3]: id:10  email:"bminifie6@taobao.com"
2022/10/13 13:49:01   item [3 of 3]: id:11  email:"tcurnnokk7@weebly.com"
2022/10/13 13:49:01 get email batch
2022/10/13 13:49:01 response:
2022/10/13 13:49:01   item [1 of 3]: id:16  email:"llocksleyc@usa.gov"
2022/10/13 13:49:01   item [2 of 3]: id:17  email:"tganifordd@upenn.edu"
2022/10/13 13:49:01   item [3 of 3]: id:18  email:"ncandishe@earthlink.net"
2022/10/13 13:49:01 get email batch
2022/10/13 13:49:01 response:
2022/10/13 13:49:01   item [1 of 3]: id:19  email:"bjeacockf@apple.com"
```

CONCLUSION

The project work has been cantered on micro-services and their underlying concepts. Attempts have been made to analysis the strength and weakness of our project work based on micro-services.

The Mailing list manager implemented in this project work has been used as a medium to show how micro-services can be consumed. It can be seen that application which loosely coupled business process can be developed. Application can be developed using by the combination of object oriented programming language and can be seen at the point. The mailing list micro-services was built from scratch and was deployed using a shared hosting server and was consumed by mailing list microservices.

FUTURE ENHANCEMENTS

This project could be hosted over the internet, by creating an appropriate GUI and hosting it over internet in the upcoming future days.