Analysis

**1 Introduction**

With the current situation with the outbreak of Covid-19, many schools and other educational facilities have moved to online education. This comes with several problems, the main one being homework, and the ability to track students that decide to, or cannot complete it. There are several solutions to this issue on the market, two of them being by Apple, with the now discontinued iTunesU, and the more modern Apple Schoolwork. Unfortunately, the latter solution sometimes does not work well from my experience, which leads to confusion and unnecessary delays. I have come up with a more reliable solution, which mimics the main functionality of Apple’s Schoolwork app, named Classwork. Classwork is a web application, which allows a headmaster to manage multiple schools, a teacher to manage subjects and assignments, and students to submit their work to the teacher.

**Complete source code can be found at** [**https://github.com/frogstair/classwork**](https://github.com/frogstair/classwork)

**1.1 Research**

Some research was made before I started working on Classwork. Mainly, I asked students from my school what their complaints for Schoolwork were, and other suggestions. The most popular suggestions were to make it more reliable and faster, so I focused on that. To increase reliability, I looked into ORM (Object-Relational Mapper) systems for my programming language of choice, Go. I picked it because it is considered modern, fast, reliable and concurrent, so its features fully address the problems of Schoolwork. The ORM I picked is called GORM, and it greatly simplifies working with a database, allowing for a more secure and stable operation. The final part of my research consisted of looking for a suitable database management system. In the end I settled for PostgreSQL because it is efficient, concurrent and supports standard SQL queries.

***1.1.1 The Go Programming Language***

*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.*

*From Wikipedia*

I picked Go for my project because I was familiar with it before, so it was a first choice, and because it has many other advantages, such as it being compiled, so it has very high performance, statically typed, so it prevents type errors from occurring, and memory safe, which means it won’t create any unexpected errors. A snippet of Go code looks like this

func main() {

err := godotenv.Load() // Load values from .env

if err != nil { // Check if an error occurred

log.Fatalln("Could not find .env file!")

}

rand.Seed(time.Now().UnixNano()) // Generate a random seed

wg := sync.WaitGroup{} // Create a waitgroup to synchronize goroutines

wg.Add(2)

go run(&wg) // Run function on a new goroutine

wg.Wait() // Wait for the waitgroup to exit

}

***1.1.2 GORM***

*The fantastic ORM library for Go that aims to be developer friendly.*

*From the GORM documentation*

GORM is an Object-Relational Mapper that was created for Go. It automatically builds a safe to use SQL query and tracks relationships between tables. It integrates well with Go as well. A typical use case looks like this

// Retrieve a user by ID and check for errors

err := database.Where("id = ?", userID).First(user).Error

if err != nil {

if util.IsNotFoundErr(err) {

resp.Data = nil

resp.Error = "Invalid user ID"

return 404, resp

}

return 500, "Database error"

}

**1.2 Objectives**

The objectives for the project were pretty clear: create a web application that can host three types of users, each with their own permissions

**Headmaster:**

* Create schools which host students and teachers (one headmaster can manage multiple schools)
* Add teachers and students to the school

**Teacher:**

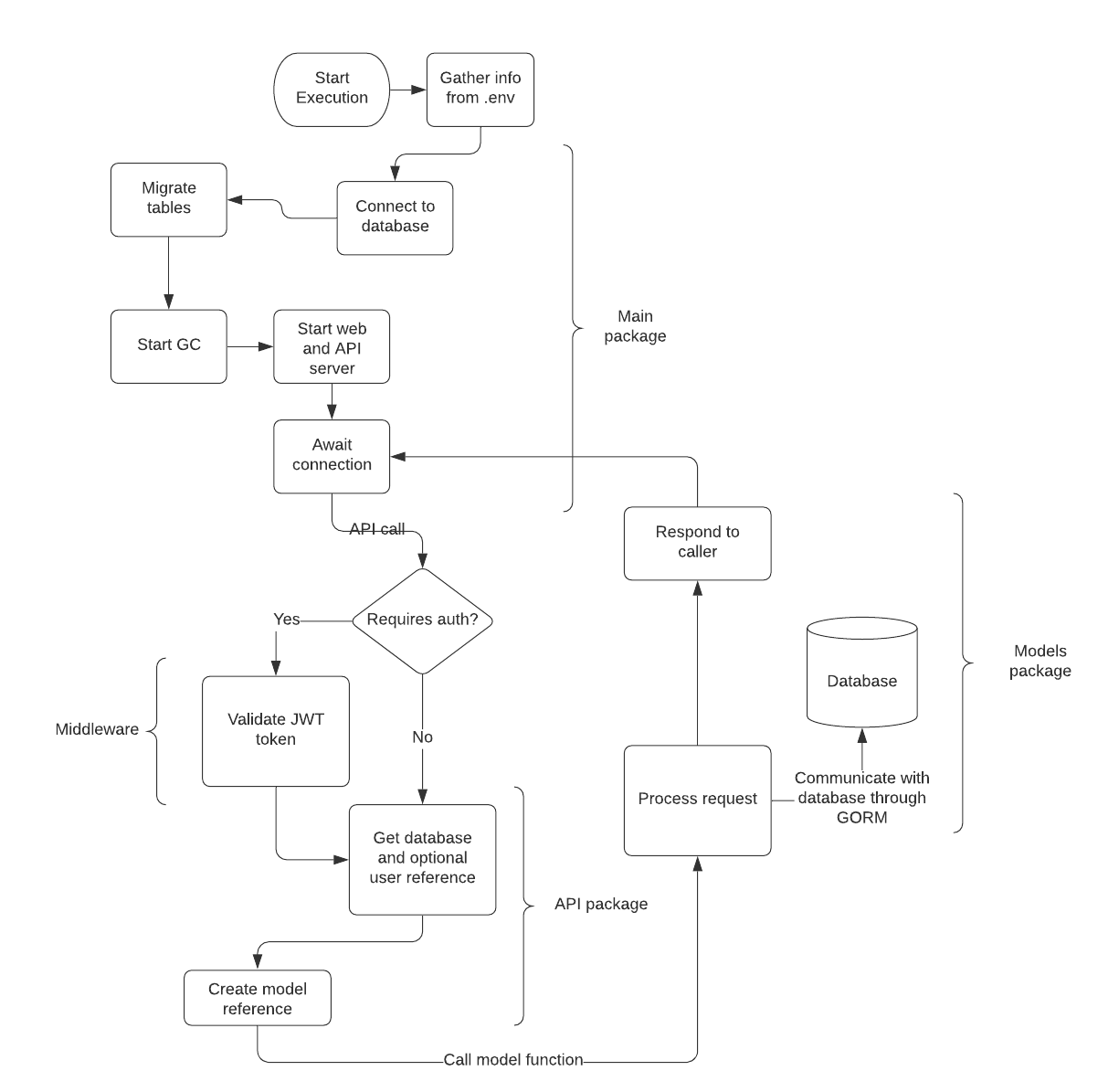
* Create subjects and add students to said subjects (each teacher can manage multiple subjects)
* Create assignments for students to complete

**Student:**

* Upload a file to an assignment

And with all of that must come a web-based user interface written in JavaScript and HTML

**2. General Operation**

[****](https://lucid.app/documents/edit/924f1894-871b-4ffb-883b-45fdeccf2e19/0?callback=close&name=docs&callback_type=back&v=976&s=612)

*Figure 1.*

Figure 1 demonstrates the basic working principle behind every API call, excluding serving HTML and uploading/downloading files, and omitting details like error handling. As a means of communication I will use JSON, which is incredibly common, and Go has a built-in library that helps parse and create JSON from and to Go structs.

**2.1 The Server**

**2.1.1 HTTP/S**

The HTTP/S engine that I decided to use is a framework called Gin, which is considered one of the fastest and most flexible frameworks due to it being written entirely in Go, and it using Go’s native low-level APIs. It also supports a variety of useful features such as middleware, which I will be using to authorize the user and access the database. It’s API is very simple, and can be utilized in many ways, allowing the user to group their endpoints, and handling all the parameters a user might give to the server. I, however, did not implement HTTPS, because it requires owning a domain, to which a certificate should be attached, which I do not have, but it could easily be implemented if such a requirement appears.

The code used to initialize and run the server looks like this:

*File: main.go*

package main

import (

"classwork/api"

"classwork/database"

"classwork/garbage"

m "classwork/middleware"

"classwork/pages"

"log"

"math/rand"

"os"

"sync"

"time"

"github.com/gin-gonic/gin"

"github.com/joho/godotenv"

)

func main() {

err := godotenv.Load() // Load all values from the .env file

if err != nil { // If an error occurred then exit

log.Fatalln("Could not find .env file!")

}

rand.Seed(time.Now().UnixNano()) // Seed random number generator

wg := sync.WaitGroup{} // Create a waitgroup to run everything asynchronously

wg.Add(1) // Only one function will be running asynchronously

go run(&wg) // Run the function

wg.Wait() // Wait for the program to exit

}

func run(wg \*sync.WaitGroup) {

db := database.GetPostgres() // Get a database connection

defer db.Close() // Close the database connection when the function returns

defer wg.Done() // Mark the function as done when the function returns

defer func() { garbage.Quit <- true }() // Quit the GC once the program exits

gin.SetMode(gin.ReleaseMode) // Set gin's mode to release to remove any logs

g := gin.New() // Create a new router

g.Use(gin.Recovery()) // Use the recovery middleware to recover from functions that may have crashed

g.Use(m.Postgres) // Use the postgres middleware to inject the database connection into every function

g.NoRoute(pages.NotFound)

g.NoMethod(pages.NoMethod)

// Create routes to serve all the html pages

// Could have made a smarter system than that

// but it created router conflict

g.GET("/register", pages.ServeRegister)

g.GET("/login", pages.ServeLogin)

g.GET("/login/pass", pages.ServeLoginPassword)

g.GET("/dashboard", pages.ServeDashboard)

g.GET("/school", pages.ServeSchool)

g.GET("/subject", pages.ServeSubject)

g.GET("/assignment", pages.ServeAssignment)

g.Static("/static", "./web/static")

// All the API routes and their handlers

apiGroup := g.Group("/api")

logoutGroup := apiGroup.Group("/logout")

logoutGroup.POST("/", m.ValidateJWT, api.Logout)

logGroup := apiGroup.Group("/login")

logGroup.POST("/", api.Login)

logGroup.POST("/pass", api.GenerateOTC)

regGroup := apiGroup.Group("/register")

regGroup.POST("/", api.Register)

regGroup.GET("/email", api.EmailValid)

dbdGroup := apiGroup.Group("/dashboard")

dbdGroup.GET("/", m.ValidateJWT, api.GetDashboard)

schGroup := apiGroup.Group("/school")

schGroup.POST("/", m.ValidateJWT, api.AddSchool)

schGroup.DELETE("/", m.ValidateJWT, api.DeleteSchool)

schGroup.GET("/", m.ValidateJWT, api.GetSchool)

schGroup.GET("/student", m.ValidateJWT, api.GetStudents)

schGroup.POST("/teacher", m.ValidateJWT, api.AddTeacher)

schGroup.DELETE("/teacher", m.ValidateJWT, api.DeleteTeacher)

schGroup.POST("/student", m.ValidateJWT, api.AddStudent)

schGroup.DELETE("/student", m.ValidateJWT, api.DeleteStudent)

subGroup := schGroup.Group("/subject")

subGroup.POST("/", m.ValidateJWT, api.AddSubject)

subGroup.DELETE("/", m.ValidateJWT, api.DeleteSubject)

subGroup.GET("/", m.ValidateJWT, api.GetSubject)

subGroup.POST("/students", m.ValidateJWT, api.AddStudentSubject)

assgnGroup := subGroup.Group("/assignment")

assgnGroup.POST("/", m.ValidateJWT, api.NewAssignment)

assgnGroup.GET("/", m.ValidateJWT, api.GetAssignment)

assgnGroup.POST("/complete", m.ValidateJWT, api.CompleteAssignment)

g.Use(garbage.AddCollectorToContext)

go garbage.Run()

g.Static("/files", "./files")

fsgroup := g.Group("/files")

fsgroup.POST("/", api.CreateFile)

// Get the address and port from the environment

address, port := os.Getenv("ADDRESS"), os.Getenv("PORT")

// Print a log

log.Printf("Running on %s:%s\n", address, port)

// Run the server

g.Run(address + ":" + port)

}

**2.1.2 The Database**

The database server I decided to use is called PostgreSQL. It has several advantages such as being multithreaded, light on memory usage, fully open source and highly configurable. The database structure is completely decided upon by GORM, so it may not be perfect, and I had to create foreign key relationships myself

The database structure is as follows:

|  |  |
| --- | --- |
| Table name | Description |
| users | User info |
| schools | School info |
| subjects | Subject info |
| assignments | Assignments the teacher has created |
| requests | Upload requests for the assignments |
| request\_uploads | Associates the files that students have uploaded to each request |
| assignment\_files | Files attached to each assignment |
| school\_students | Relate users that are students and schools |
| school\_teachers | Relate users that are teachers and schools |
| subject\_assignments | Relate assignments and subjects |
| subject\_students | Relate subjects and users that are students |
| assignments\_completed | Relates students that completed the assignment to assignments |

When I use GORM, I don’t have to interact with any of these tables, and all the interactions are handled by the library itself.

**Users table**

|  |  |
| --- | --- |
| Column name | Description |
| id | Contains the user’s ID |
| first\_name | User’s first name |
| last\_name | User’s last name |
| email | User’s email |
| password | The hash of the user’s password |
| token | The user’s access token |
| perms | A collection of flags that determine who the user is |
| pass\_set | A flag that determines if the user’s password is set, used to indicate that a user needs to set a password |
| one\_time\_code | A one time code used to set the password for the user |
| school\_id | Indicates to which school a student belongs, otherwise empty. Used to reduce amount of calls to database |

**Schools table**

|  |  |
| --- | --- |
| Column name | Description |
| id | The ID of the school |
| user\_id | The ID of the headmaster that created the school |
| name | School name |

**Subjects table**

|  |  |
| --- | --- |
| Column name | Description |
| id | ID of the subject |
| teacher\_id | ID of the teacher that created the subject |
| school\_id | ID of the school that the subject is in |
| name | Subject name |
| num\_students | Amount of students |

**Assignments**

|  |  |
| --- | --- |
| Column name | Description |
| id | Assignment ID |
| teacher\_id | ID of the teacher that assigned it |
| subject\_id | ID of the subject that the assignment is in |
| name | Name of the assignment |
| text | Assignment description |
| time\_due | Time the assignment is due |
| time\_assigned | Time the assignment was created |

**Connection to the database**

The connection to the database is established at the beginning of the execution of the program, and the function responsible for it will either create a connection to the database, or returns an existing connection.

*File: database/postgres.go*

package database

import (

"fmt"

"log"

"os"

m "classwork/models"

"github.com/jinzhu/gorm"

)

var db \*gorm.DB // Global variable that keeps the database connection

var connected = false // Flag if the connection is established

// GetPostgres initializes the connection to a postgres database or returns an existing connection

func GetPostgres() \*gorm.DB {

if connected { // If the connection is already established then return the connection

return db

}

host := os.Getenv("DB\_ADDR") // Get variables from the environment

port := os.Getenv("DB\_PORT")

role := os.Getenv("DB\_ROLE")

name := os.Getenv("DB\_NAME")

var err error // Variable to contain the error

cstring := fmt.Sprintf("host=%s user=%s dbname=%s sslmode=disable port=%s",

host, role, name, port) // Connection string, formed using a format string

db, err = gorm.Open("postgres", cstring) // Open a "postgres" database connection

// If an error occurred then panic and exit the program

if err != nil {

panic(fmt.Sprintf("\n===========\ncannot establish database connection: \n%s\n===========", err))

}

// Set the flag

connected = true

// Remove unnecessary database logs

db.LogMode(false)

// Print a success log

log.Println("Connected to database")

// Synchronize all the tables in the code with the database

db.AutoMigrate(&m.User{}, &m.School{}, &m.Subject{}, &m.Assignment{}, &m.Request{}, &m.AssignmentFile{}, &m.RequestUpload{})

log.Println("Migrated tables")

// Return the newly established connection

return db

}

// Disconnect closes the database connection

func Disconnect() {

// Close the connection to the database

db.Close()

// Set the flag to false

connected = false

}