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:**

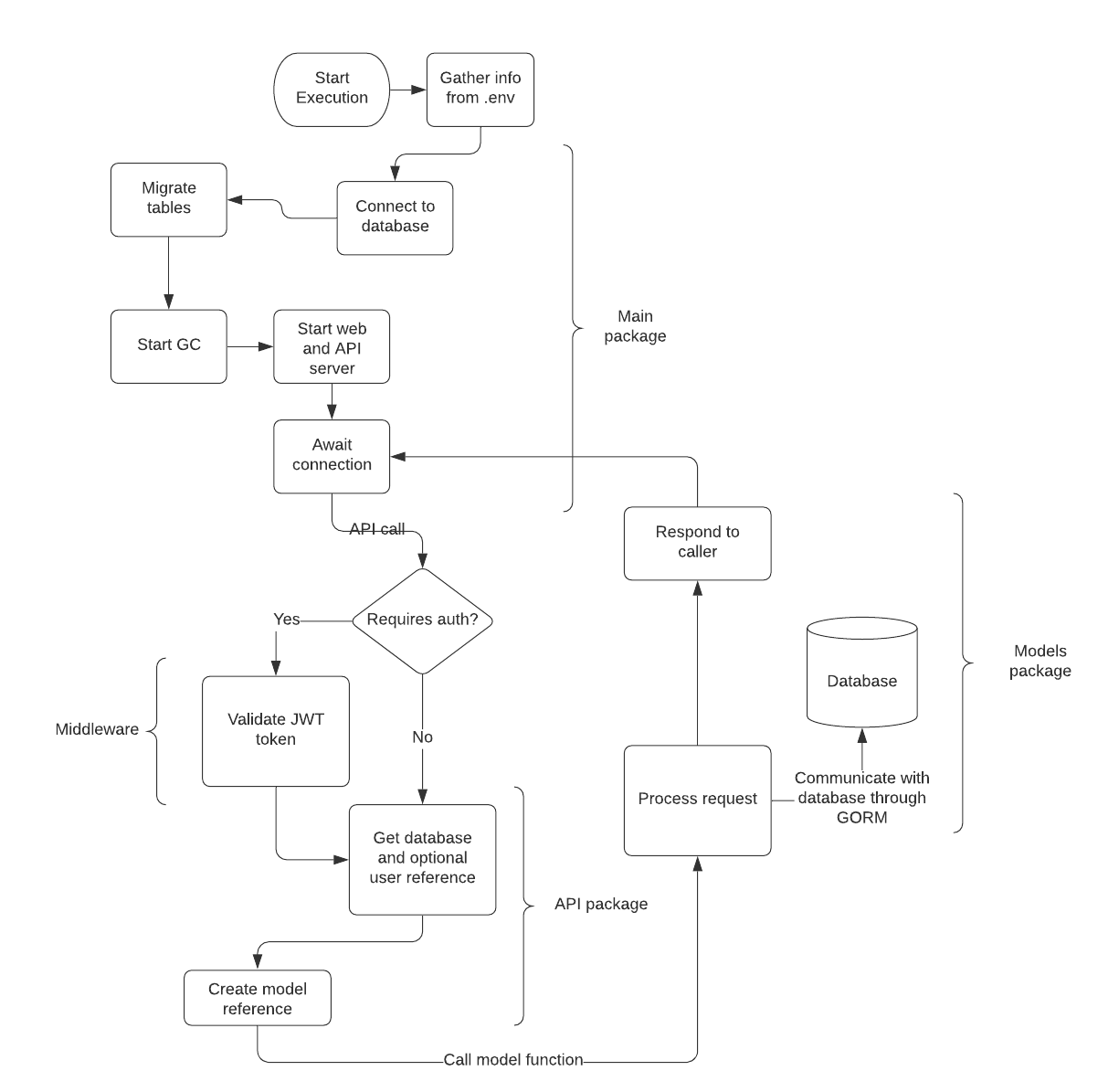
* Create subjects and add students to said subjects (each teacher can manage multiple subjects)
* Create assignments for students to complete

**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 } |

**2.1.4 Authentication**

Authentication is the process of verifying a user’s identity. In Classwork this is done using a JWT (JSON Web Token). JWT is a self-contained key that a user uses to present their claims. For example, a user could claim that he is a headmaster, and request access to a headmaster-only resource. It is the server’s job to validate that key and present the user with information or reject it.

The JWT works the following way:

JWT is represented as an encoded string, separated by fullstops. This is an example of a JWT token that Classwork would generate:

**eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2MTcyMDk4NTYsImlkIjoiMW9JcE5IRTdXT0VtZGxDR3VJc1l1NFI3TEdvIn0.hQQaLpD2koqD1\_o3BXSJtRgfa7tGFVmxwJW6CPw3P-x3PM9Ve9Jr80LCnAqNXa9sHdA0\_f4EqfdmiNJSyO4BBw**

It contains three sections, the first one being **eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9** and delimited by a fullstop. This is called the header, and is actually a Base64 encoded JSON string, which contains the token type, always JWT, and the encryption method, which for Classwork is HS512. The decoded string looks like this: {"alg":"HS512","typ":"JWT"}.

The next section is **eyJleHAiOjE2MTcyMDk4NTYsImlkIjoiMW9JcE5IRTdXT0VtZGxDR3VJc1l1NFI3TEdvIn0.** It contains the claims the client has made. It is the information that can identify the client, such as the User ID. In Classwork the claims made by the client are only the User ID, and when the token expires, i.e becomes invalid. The rest of the information is pulled from the database, so the user doesn’t need to provide their claims. The decoded information looks like this: {"exp":1617209856,"id":"1oIpNHE7WOEmdlCGuIsYu4R7LGo"}.

The third and final section is **hQQaLpD2koqD1\_o3BXSJtRgfa7tGFVmxwJW6CPw3P-x3PM9Ve9Jr80LCnAqNXa9sHdA0\_f4EqfdmiNJSyO4BBw**. In a valid key it contains the same data as the second part, except it is encrypted with the algorithm provided in the header, and a key that is stored on the server. When a server validates this information, this section is decoded and the contents are compared.

The function to validate a user’s token looks like this:

*File: models/tokens.go*

|  |
| --- |
| *// ParseToken parses a token and returns the associated user func ParseToken(tokstr string, db \*gorm.DB) (int, \*util.Response) {   resp := new(util.Response) // Create a response to the user   tok := &Token{} // Allocate memory for a new token  token, err := jwt.ParseWithClaims(tokstr, tok, // ParseWithClaims function parses the token with a function that returns the key  func(t \*jwt.Token) (interface{}, error) { // Since I only store the key in an environment variable, get the key from there  return []byte(os.Getenv("JWT\_SECRET")), nil // Just return it  })   if !token.Valid { // If the token is invalid (i.e invalid syntax)  if err != nil {  resp.Data = nil  resp.Error = "token is invalid"   return 401, resp // Respond with corresponding message and error code  }  }   user := new(User) // Retreive a user from the database  err = db.Where("id = ?", tok.ID).First(user).Error  if err != nil { // Error handling  if util.IsNotFoundErr(err) {  resp.Data = nil  resp.Error = "token does not correspond to user"   return 401, resp  }  log.Printf("Database error: %s\n", err.Error())  return util.DatabaseError(err, resp)  }   if user.Token != tokstr { // In the end, if the tokens dont match  resp.Data = nil  resp.Error = "token does not correspond to user"   return 401, resp  }   resp.Data = user // Return the user variable  resp.Error = ""  return 200, resp // Success }* |

And the function to generate such a token is as follows:

*File: models/tokens.go*

|  |
| --- |
| *// TokenValidity specifies how long a token is valid var TokenValidity = int64(2592000) // 30 days  // Token is the model for a JWT token with custom claims type Token struct {  jwt.StandardClaims // Standard JWT claims (in my case only the `expires` field)   ID string `json:"id"` // The user's ID }  // CreateToken will create a token string for a given ID func CreateToken(id string) string {  token := new(Token) // Create a new token   token.ID = id // Set the ID in the token to the passed ID   token.ExpiresAt = time.Now().Unix() + TokenValidity // Set expiration time (current time + how long a token is valid)   jwtToken := jwt.NewWithClaims(jwt.SigningMethodHS512, token) // Create a new JWT token instance signed with HS512  tokenString, \_ := jwtToken.SignedString([]byte(os.Getenv("JWT\_SECRET"))) // Encrypt the token with the secret from an environment variable   return tokenString // Return the string to be sent to the user }* |

**2.1.5 Environment variables**

Classwork makes use of environment variables to securely store and use variables. They are stored in a .env file in the same directory as the executable, and the first thing the program does is search for that file and import it. In a normal use case, the file would be omitted from programs like git, to prevent security breaches. The # symbol represents a comment and its contents are ignored

*File: .env*

|  |
| --- |
| *PORT=5000 ADDRESS= # empty to allow all connections  DB\_ADDR=127.0.0.1 DB\_PORT=5432 DB\_ROLE=frogstair DB\_NAME=classwork  JWT\_SECRET=3af8cdd15bb6009585c025d4f826dfa06a5fe8943e7d9e1a3b529cfc21baacb2ca41288d7a5d31ccdab5648022ae55bf9880bb829a19c48f95 # MUST REPLACE, KEY IS IN THE OPEN* |

In the main function the I use the godotenv package to read the file

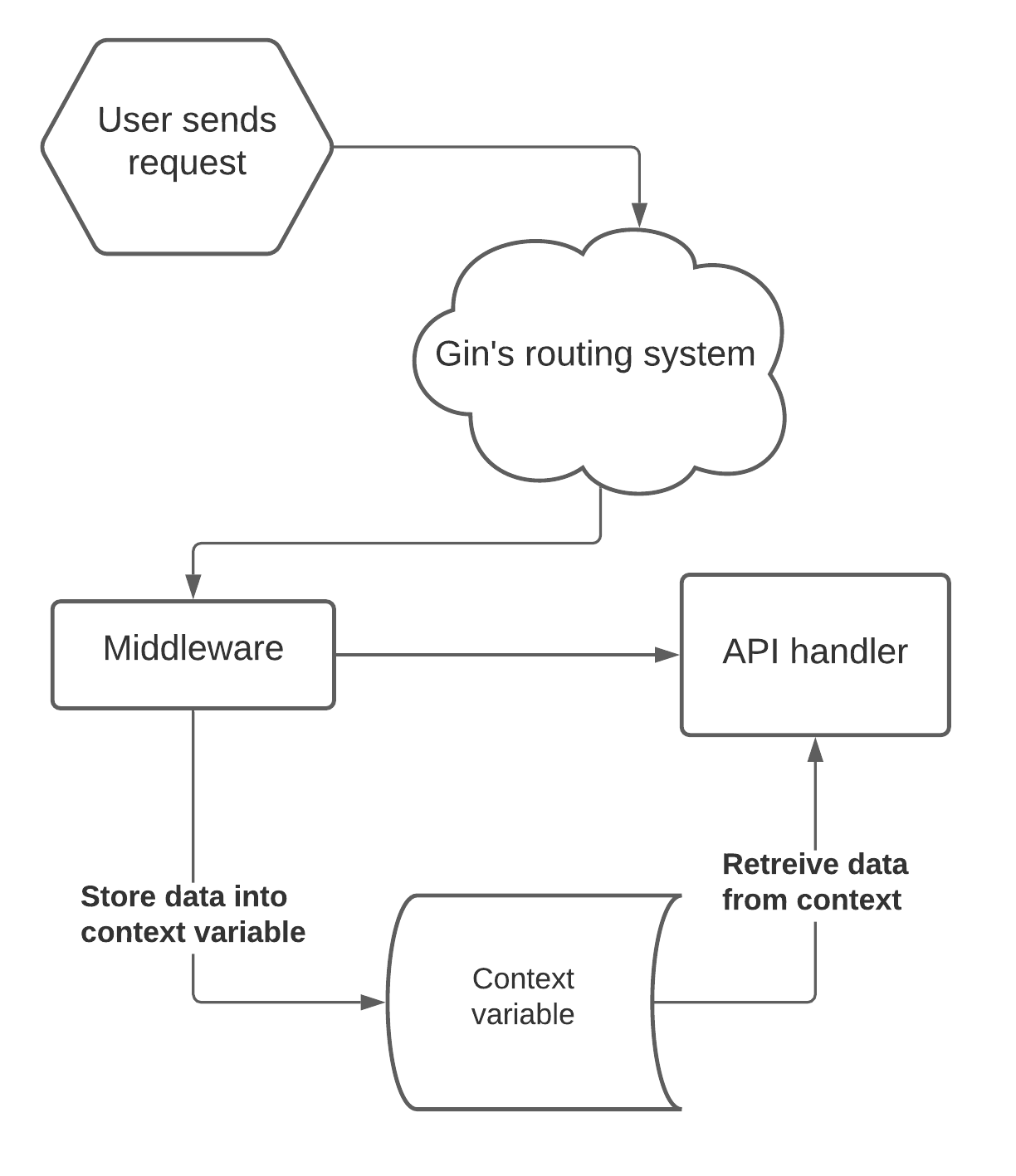
*File: main.go*

|  |
| --- |
| ..... 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!")  } ..... |

The variables specified in the .env file are then available to the whole program through os.GetEnv().

**2.1.6 Middleware**

Classwork uses two middleware functions, one to serve a database connection to another function, and another to validate a JWT token. These middleware functions are run before the actual handlers, saving any necessary data into the context variable, which can then be retrieved by later functions. This is the general operation of the functions:

[](https://lucid.app/documents/edit/70e5db97-a799-474f-8774-e3628b27b8c8/0?callback=close&name=docs&callback_type=back&v=373&s=612)

To use middleware across all routes gin has a useful function Use which does exactly that

*File: main.go*

|  |
| --- |
| *..... 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 .....* |

Middleware can also be injected into individual routes

|  |
| --- |
| dbdGroup := apiGroup.Group("/dashboard")  // Inject the ValidateJWT middleware into the dashboard route dbdGroup.GET("/", m.ValidateJWT, api.GetDashboard) |

The ValidateJWT middleware uses the JWT parsing function from earlier

|  |
| --- |
| // ValidateJWT validates the JWT token and places the user it belongs to in the context func ValidateJWT(c \*gin.Context) {   db, ok := c.Keys["db"].(\*gorm.DB) // Get database variable from context  if !ok { // If not found  c.JSON(500, gin.H{"error": "internal error"})  panic("no database variable in context")  }   tokstr, err := c.Cookie("\_tkn") // Get `\_tkn` from cookie  if err != nil { // If there is an error  c.JSON(400, gin.H{"error": "no token specified"})  c.Abort() // Doesnt run the next function in the chain  return  }   code, data := m.ParseToken(tokstr, db) // Parse token  if code != 200 {  c.JSON(code, gin.H{"error": data.Error})  c.Abort()  return  }   c.Set("usr", data.Data) // Set user context variable  c.Next() } |

And the small database middleware function

|  |
| --- |
| // Postgres attaches a database variable to a given context func Postgres(c \*gin.Context) {  db := database.GetPostgres() // Get existing connection/establish new connection  c.Set("db", db) // Set context variable  c.Next() } |

**3. API and models**

**3.1 /api/login and /api/login/pass**

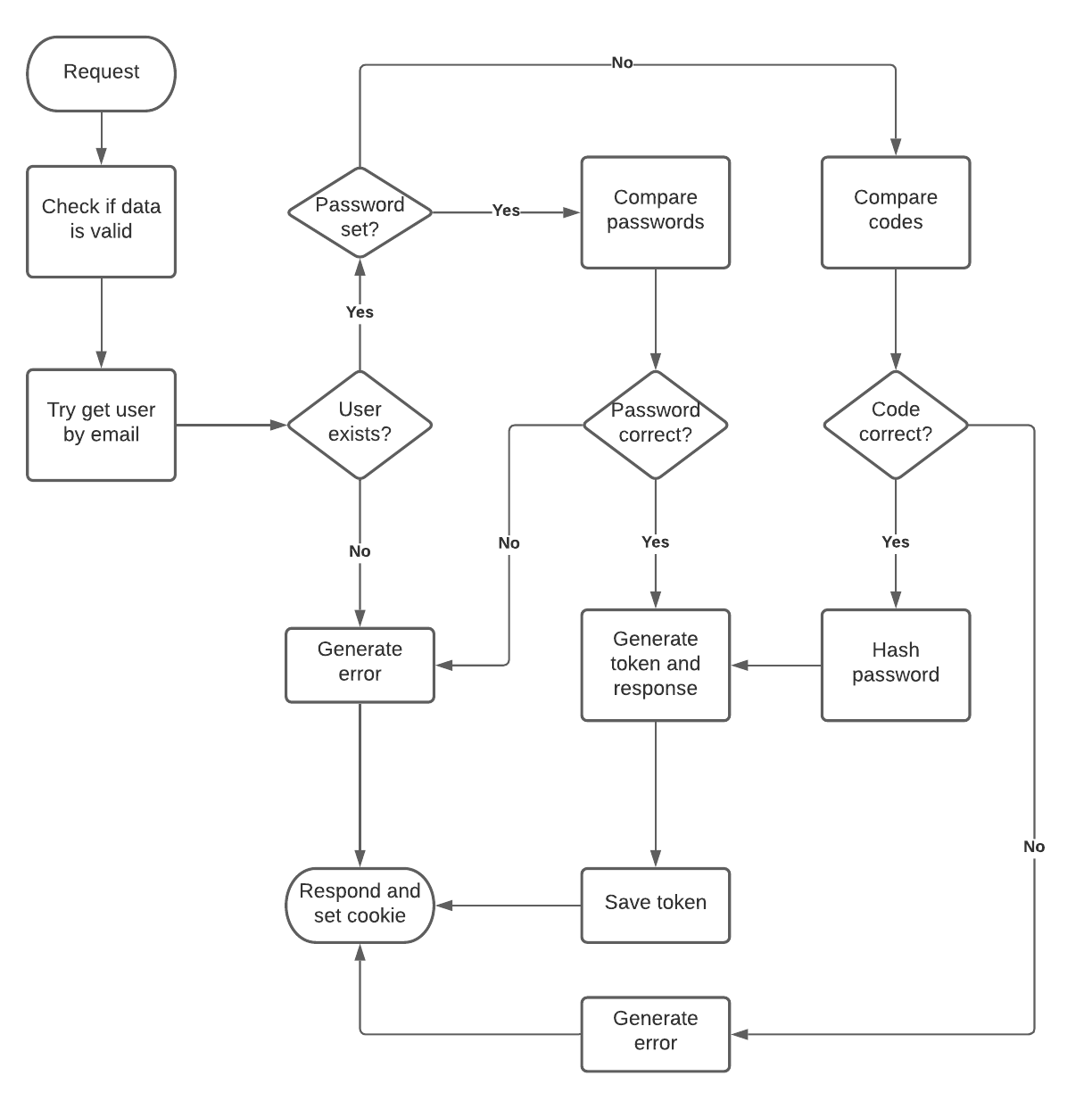
This route is used to login a user. The user must send out the following values in JSON format, which is almost exclusively used in communication with the server

|  |
| --- |
| {  "email": "email@example.com",  "password": "Password123",  "code": ""  } |

The code field is optional, and it’s role will be explained later in this document. Once the user makes a request, the Login function is run, to generate a token

**3.1.1 Login procedure**

The login procedure is described in the image below

[](https://lucid.app/documents/edit/cdee9af1-5875-4efe-8b79-766dbc403d27/0?callback=close&name=docs&callback_type=back&v=1010&s=612)

The code is used to identify users without a password, and is only available once. This only happens when a teacher or a student is added, and it allows them to set their own password the first time they login. This password is then saved and the code is deleted. The code is generated by the server by the /api/login/pass endpoint

The function to generate the code looks like this

|  |
| --- |
| // GenerateOTC creates an OTC for a user if their password is not set func GenerateOTC(c \*gin.Context) {  db, ok := c.Keys["db"].(\*gorm.DB)  if !ok {  c.JSON(500, gin.H{"error": "internal error"})  panic("no database variable in context")  }   otc := new(m.OTCCreate)  err := json.NewDecoder(c.Request.Body).Decode(otc)  if err != nil {  c.JSON(400, gin.H{"error": "invalid data"})  return  }   code, resp := otc.Create(db)  c.JSON(code, resp) } |

The OTCCreate model is a Go struct and contains the following field

|  |
| --- |
| // OTCCreate is the struct to check is a user has a password type OTCCreate struct {  Email string } |

And has a function

|  |
| --- |
| // Create creates an OTC for the user func (o \*OTCCreate) Create(db \*gorm.DB) (int, \*util.Response) {  resp := new(util.Response) // Response placeholder  user := new(User) // User placeholder   err := db.Where("email = ?", o.Email).First(user).Error // Get the user   if err != nil { // Check for errors  if util.IsNotFoundErr(err) { // If user wasn't found  resp.Data = nil  resp.Error = "Invalid user"  return 404, resp  }  return util.DatabaseError(err, resp) // If any other error  }   if user.PassSet { // If the user has already tried to generate the code  resp.Data = nil  resp.Error = "resource gone"  return 410, resp  }   onetimecode := util.RandomCode() // Generate a random code   user.OneTimeCode = onetimecode // Save the code until the user uses it  err = db.Save(user).Error  if err != nil {  return util.DatabaseError(err, resp)  }   resp.Data = onetimecode // Return the code to the user  resp.Error = ""   return 201, resp } |

The login function itself performs the following actions

|  |
| --- |
| // Login creates a token for the user to use for future authentication func Login(c \*gin.Context) {  db, ok := c.Keys["db"].(\*gorm.DB) // Get database variable from context  if !ok { // If an error occurred  c.JSON(500, gin.H{"error": "internal error"}) // Respond to user  panic("no database variable in context") // Crash with a message  }   loginuser := new(m.LoginUser) // Create an instance of a LoginUser  err := json.NewDecoder(c.Request.Body).Decode(loginuser) // Decode the users request  if err != nil { // If there is an error while decoding  c.JSON(400, gin.H{"error": "invalid data"})  return  }   code, data, tok := loginuser.Login(db) // Run the model function  if code == 200 {  c.SetCookie("\_tkn", tok, int(m.TokenValidity), "/api/", "", false, true) // Set a cookie on the user side  }   c.JSON(code, data) // Respond with data } |

The LoginUser model is another Go struct, which contains the following properties

|  |
| --- |
| // LoginUser is the model to create a token for the user type LoginUser struct {  Email string `json:"email"`  Password string `json:"password"`  Code string `json:"code"` } |

And has the following function

|  |
| --- |
| // Login will generate a token for the user func (l \*LoginUser) Login(db \*gorm.DB) (int, \*util.Response, string) {  resp := new(util.Response) // Create response placeholder  l.clean() // Clean the user input (remove trailing spaces and invalid characters)   user := new(User)  err := db.Where("email = ?", l.Email).First(user).Error // Get a user from the database with the entered email   if err != nil { // If an error occurred  if util.IsNotFoundErr(err) { // If no user was found  resp.Data = nil  resp.Error = "Invalid email or password"  return 401, resp, ""  }  \_, resp = util.DatabaseError(err, resp) // Any other errors  return 500, resp, ""  }   if user.PassSet { // If user already has a password  if !util.Compare(user.Password, l.Password) { // Check if password is correct  resp.Data = nil  resp.Error = "Invalid email or password"  return 403, resp, ""  }  } else { // Check the code  if l.Code == user.OneTimeCode { // Create the password if the code is correct  user.PassSet = true  user.Password = util.Hash(l.Password)  user.OneTimeCode = ""  } else { // Otherwise return nothing  resp.Data = nil  resp.Error = "Invatid OTC"  return 401, resp, ""  }  }   user.Token = CreateToken(user.ID) // Create a token for the user  err = db.Save(user).Error // Save the user with the token and password  if err != nil {  \_, resp = util.DatabaseError(err, resp)  return 500, resp, ""  }   resp.Data = true // Place success flag  resp.Error = ""   return 200, resp, user.Token // Respond with the token } |

**3.2 /api/register and /api/register/email**

When a user registers an account, they need to provide the following information

|  |
| --- |
| {  "email": "email@example.com",  "password": "Password123",  "first\_name": "Danila",  "last\_name": "Dudkin" } |

And the register function looks like this

|  |
| --- |
| // Register is used to register a new user func Register(c \*gin.Context) {  db, ok := c.Keys["db"].(\*gorm.DB)  if !ok {  c.JSON(500, gin.H{"error": "internal error"})  panic("no database variable in context")  }   reguser := new(m.RegisterUser)  err := json.NewDecoder(c.Request.Body).Decode(reguser)  if err != nil {  c.JSON(400, gin.H{"error": "invalid data"})  return  }   code, data := reguser.Register(db)  c.JSON(code, data) } |

The RegisterUser model is again a struct

|  |
| --- |
| // RegisterUser is the model to register a new user type RegisterUser struct {  Email string `json:"email"`  Password string `json:"password"`  FirstName string `json:"first\_name"`  LastName string `json:"last\_name"` } |

And performs the registration procedure

|  |
| --- |
| // Register registers the user into the database func (r \*RegisterUser) Register(db \*gorm.DB) (int, \*util.Response) {  resp := new(util.Response) // Response placeholder   r.clean() // Remove trailing whitespace and invalid characters  valid, reason := r.validate() // Validate the input, and if invalid get reason  if !valid {  resp.Data = nil  resp.Error = reason  return 400, resp  }   hashed := util.Hash(r.Password) // Hash the user's password  user := new(User) // Create a user placeholder   user.ID = ksuid.New().String() // Create a GUID and fill in user information  user.Email = r.Email  user.FirstName = r.FirstName  user.LastName = r.LastName  user.Password = hashed  user.Perms = Headmaster  user.PassSet = true   err := db.Create(user).Error // Create the user in the database  if err != nil { // If an error occurred  if util.IsDuplicateErr(err) {  resp.Data = nil  resp.Error = "Email is taken"  return 409, resp  }  return util.DatabaseError(err, resp)  }   userResponse := struct { // Create a response for the user  FirstName string `json:"first\_name"`  LastName string `json:"last\_name"`  ID string `json:"id"`  }{user.FirstName, user.LastName, user.ID}   resp.Data = userResponse  resp.Error = ""   return 201, resp // Respond } |