

## Encryption and verification of passwords

One last thing we need to do to complete setup of our project is specifying functions for hashing passwords before saving and verification of passwords.

We use bcrypt again. 😊

For hashing, `bcrypt.GenerateFromPassword` ([ ] byte password, bcrypt.DefaultCost) It accepts byte slice of the password and default cost and provides slices of the hashed password and an error.

For verification, `bcrypt.CompareHashAndPassword` ([ ] byte(password), [ ] byte(hashed password)) It accepts byte slice of password and the hashed password. Then return an error. In case error is there, we need to find out if error occurred due to mismatch or for other reasons. So, we use `bcrypt.ErrMismatchHashAndPassword`. It also returns an error. In case, both errors match, error was due to password mismatch, otherwise for any other reason.

## Setting Handlers / controllers

Now that we are ready with our set up, we can march forward to setting up our handlers for performing actions. Then we will setup routes.

We use the same modular pattern that we used in the `query.go` file. We again define an abstraction called `GoApp` which will be a struct containing the `config.Config` instance and `dbrepo.Interfaces` instance. These two instances will help in maintaining modularity. All the controllers will be methods of this `GoApp` struct. Also, each of these methods will return an anonymous fn of type `gin.HandlerFunc`. Now, all these controllers are called in API's routes defined in `routes.go` file. This `routes.go` file will return a function `Routes`.




which will take gin.Engine() (we will provide ~~gin.Engine~~ our gin.Engine that we developed in main.go to it) and reference of goApp. Then, this function will define all the routes on this ~~gin.Engine~~ router by specifying ~~the~~ and calling the specific controller on the goApp's instance reference. In this file, at first we include the logger and necessary middleware by `u.Use(gin.Logger())`, `gin.Recovery()` and store it somewhere and use it as the router. We will ~~also~~ also set up this file to store the cookies. We'll use gin's sessions for it. Install `github.com/gin-contrib/sessions`. `u.Use(sessions.Sessions("session", cookieData))`. `u.Use(cookie.NewStore(1, byte("go-app")))` sets up a new cookie store. `u.Use(sessions.Sessions("session", cookieData))` use the method directly adds middlewares to the routers.

### Handling Signup request

The user will provide the necessary details, like name, email, etc. for sign up. This information will be in json format and can be accessed using the `gin.Context` in `handler.go`.

Prepare a ~~method~~ method `Signup` for the user. In return, it will return `gin.HandlerFunc` type function. In this func we'll provide the `gin.Context`.

First thing to do inside this fn is creating a variable of the model's User struct type. Then bind this User to the incoming context's data using `ShouldBindJSON`. If data is invalid, it ~~will~~ returns an error. Use validator to validate this data. Set up `user.CreateAt` and `user.UpdateAt`. Call `Hash` of `encrypt.go` to encrypt the password before saving.

If everything is correct, call the `InsertUser` function of `goApp.DB` struct.  Well we need to create a method with name `InsertUser` in the ~~DB~~ interface and define it in the `query.go` file as a method of `goAppDB` struct.



Now, Database operations are to be handled in the query.go file.

~~we will create~~

For sign up, we have to use InsertUser fn of query.go. InsertUser fn takes the user data of type model's user and returns three values  $\rightarrow$  bool, status, err.   
  $\downarrow$   $\downarrow$   $\downarrow$    
 false 1 for registration successful  $\rightarrow$  for handling errors   
 handle internal server error 2 for already exists

Inside the InsertUser function, create a context to set timeout. Verify mail for email format using regexp. Must compile ("email" format). It return true or false. If true proceed.

Now, we need to check if this user already exists, for that we first prepare a filter variable (of type bson.D that is a slice (ordered list) of K-V pairs) with key & value specified (we used email here). Now, the result of the query should be known. So, we use bson.M type that represents map.

Now, if user's details are not found, we hit

User(g.DB, "user").findOne(ctx, filter).Decode(&res)

$\downarrow$    
 func of database that takes out the collection from db

$\downarrow$    
 It decodes the raw bson to bson map

Working on the login route

for login, we accept some data from the user and find a doc matching the credentials. We have to get user. But things don't end here. We need to create jwt token after login so that user is



authenticated at all points, where authorization is reqd.

Now, here we will set up the 'auth.go' file to provide two methods → (i) Generate access and Refresh Token (ii) Verify Token

Generate func. takes the email id and id and generate tokens for us. Before implementing it, we need to create a struct `GoAppClaims` with a ~~standard~~ standard struct called `RegisteredClaims` and additional fields for email & id of the user. We also need a access/refresh token secret (store it in .env and extract it).

Create a struct `GoAppClaims` to provide a blueprint of data that we want to sign in token. ~~and new token (access and refresh)~~

Then create instance of it with accurate `RegisteredClaims` struct (Here info. is specified like issuer, issuedAt and expiry data).

Create first token using `jwt.NewWithClaims` (signing method of jwt, data storing format). Signing (Byte/secret key)

Refresh Token don't contain those info, so just the `jwt.RegisteredClaims` struct is reqd. to be defined.

This fn return the token and error. So, return the token.

Then `Parse` fn is defined that checks the access token and returns the data in `GoAppClaims` format.

Here the MVP is `jwt.ParseWithClaims` (token, & `GoAppClaims`),  
func(t jwt.Token) (interface{}, error) {  
return [] byte(secretkey), nil

7/

~~In Controller, we~~ In the query, we'll define 2 more functions verify user and update info. Verification will be using a very simple filter of email.



updateInfo is more interesting. It will take the token and ID as input. On verification, we will get the user, so we use the id and generate new tokens to provide to updateInfo function.

update variable 'mould' be created to specify the update query.

```
update = bson.D({$set: "token": bson.D({key: "token", Value: tk["t1"]}), $set: "new-token": Value: tk["t2"]})
```

Then perform the update →

```
- err := User(g.DB, "user").UpdateOne(ctx, filter, update)
```

Now, in handler,

do find the information coming from the user. Verify it with the available email. → It will give us response and password. Use encrypt. Verify (user.Password, password) (response["password"]). (spring)

if the user is not found, return 404. If found, verify the password. If correct, generate a new token and return it. If incorrect, return 401.

if the user is not found, return 404. If found, verify the password. If correct, generate a new token and return it. If incorrect, return 401.