Reading a file :-

```
data byte, err = ioutil.ReadFile(filename)
```

Starting a web server using Go's Gin framework

- gin.Default()
  - ↳ sets up a gin server instance which is capable of handling http request. Has in-built middlewares for logging and recovery. We need to store this instance in a variable.
- gin.New()
  - ↳ similar to gin.Default() but it has no middlewares.

- Ex → webserver := gin.New()

Now, we can use our webserver to allow creation of Http Get, Post, etc. request handling endpoints (APIs).

Ex:

```
webserver.GET("/home", func(ctx *gin.Context) {
    log.Println("Home Page")
})
```

.GET('home', func(ctx *gin.Context){})

- GET → Type of request
- 'home' → url endpoint
- func → handler fn
- ctx *gin.Context → It takes the context of current request being made. Helps in extracting param, query and post form data

// We've successfully created a route

Now, do err := webserver.Run(Port)

↳ It will the start the server application at the port specified and listen on the routes specified

Understanding Context

The first confusion arises around handling the context. So, we'll do it first.

Context is a way to handle information such as deadlines, cancellation and small pieces of data throughout the program.

When working with backend, user requests usually involve some time delay as it consists of many sub-operations. First use case of context is here only to handle deadlines of requests. We don't want requests to be processed forever. So, we set deadlines.

One way is withTimeout function that terminates the requests after certain time. It provides two outputs → the context and the cancelling fn. Defer this cancel function.

To forcefully cancel a request some time, use withCancel that again provides the context and the cancel fn.

To simulate a goroutine that cancels the context from some time use

```
go func() {
        time.Sleep(time.)
        cancel()
}()
```

To pass data, we can use context's withValue fn. that accepts values to be passed and returns the context.

To extract data from a context, use context.Value("key") (data type)
(name of context)

Gin's context

Gin's context is not same as context package's context. It is more inclined towards handling HTTP request. It provides several functions and ways to collect data from HTTP requests, setting and sending responses, in middlewares and even allows setting new K-V pairs in the context.

Accessing data

(*gin.Context) provides fn to access data.

- let say c *ginContext is provided to any requestHandler of an API.

→ c.Param("param") → to access params
→ c.Query("key") → to " queries
→ c.PostForm("fieldname") → to get form data.

Sending response

c.JSON(statusCode, gin.H{"key": "value"}) → for JSON res
c.String(statusCode, "string data") → for String/plainText res

Middlewares

ginContext is provided to middlewares to access the header or cookies to perform any function and then move to the Next() function.

Setting new k-v pairs

c.Set("key", value)
value, exists := c.Get("key")
↓
bool

Gin provides a method to provide access to the Go's context from Gin's context using c.Request.Context().

Creating servers with Gin and MongoDB atlas

It is important to follow modular structure for setting up any backend web service. It helps in easy debugging and modification of the codebase.
One of the most followed dir structure:-

→ cmd/web
  ↳ main.go → main exec. file
  ↳ middleware.go → it exports the fn req. for middlewares

└ routes.go // Specifying the routes required

└ drivers
  └ driver.go

└ handlers
  └ handlers.go // Similar to controllers (MVC)

└ modules
  └ auth
    └ auth.go // lists and defines the various fns reqd. for Auth.

└ config
  └ config.go // for handling errors, logs, info, validn structure

└ database // for query db
  └ query
    └ db col.go
    └ query.go
  └ db repo.go

└ encrypt
  └ encrypt.go // for encrypting passwd before saving.

└ model
  └ model.go // for defining models

.env // environment variables (more it wherever your main.go file is)
go.mod
go.sum

Designing models

Go to model.go file.

Import mongo-driver/bson/primitive for specifying type of id.

Create structures for the required models. Don't forget to add ID, CreatedAt and UpdatedAt.

First of all, mongo-driver/mongo package should be installed. It is reqd. here for specifying type of _id field.

Connecting MongoDB atlas to our server

First of all setup the database in mongodb atlas to be used by our go server. Store the Mongodb URI in the .env file.

(Set up the config.go file to allow error handling and logging and validation. It could have been much earlier but it's never too late).

Now, in driver.go, write the code to connect to the mongodb atlas. For that, we need to extract the mongodb URI from .env file and provide it to the connection function of driver.go. In main.go, use os.Getenv. Install godotenv package to load our data using godotenv.Load() function. It return error only if it can't load the data. Get your desired data using os.Getenv("Mongodb URI"). (or whatever name is there)

In driver.go file, we need to create a context with some timeout time so that it does not try to connect forever.

then use mongo.Connect(context, opts...)

We will get the instance of the client that we just created. Return this client after testing that it's working using client.Ping(ctx, nil). If no error is encountered Return the client. Use app (an instance of goBasics struct that we made in config.go) to log errors.

options.Client().ApplyURI(URI)

In the main.go, we need to call this conn function of driver.go file. But before that we should set up our Error logger and InfoLogger of appTools of the config.go file. For that, import the config in the main.go file. import log. Then write,

ErrorLogger := log.New(os.Stdout, "", log.LstdFlags|log.Lshortfile|log.LstdFlags)

Similarly, InfoLogger := log.New(os.Stdout, "", log.LstdFlags|log.Lo

After that load MONGOURI from the .env file.
Then call the connection function.
Then also calls a disconnect function (which will obviously be defered) with context.TODO().

Now that connection is established, it's time to start performing CRUD operations. But before that we set up a few more files to have more control over the codebase.
In dbcol.go, define a fn 'User' to get the user collection from our database. It accepts the mongo client that we created and the collection name.
After that we'll set up the query.go page that will allow set up of GoAppTools(config.go) and the mongoclient and it will be used everywhere (I mean whenever we'll have to make a request to the database). For achieving it, we create a struct GoAppDB in query.go with the values specified. (They need to be pointers). We'll then define fns to manipulate the database. These functions should actually be methods of GoAppDB so that all functions have access to App Tools as well as DB client. Then in DBrepo.go define an interface with a list of all DB functions like createAccount, VerifyUser, etc. This interface creation and struct creation may seem to be overwhelming but they are of great use for scalability and modularity purpose. Interface makes sure that any struct implementing all these controllers is a type of interface. This way, if in case, we can switch b/w databases without worrying about other stuffs. Simply create a new struct with functions handling all these controllers and boom. We can select whichever db we want to put data in. Only thing to worry is that struct implementing those functions must implement exact the same functions as the interface. Otherwise the struct won't be considered a type of the interface. With that we get an idea of how useful interface is for scalability.