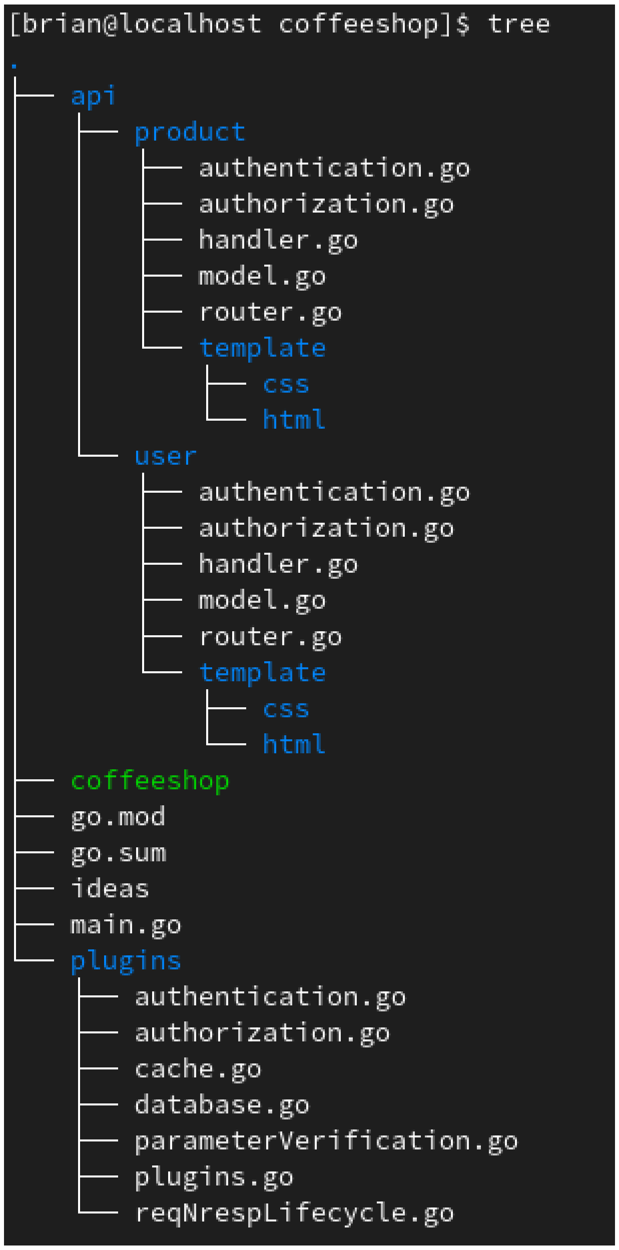
The problem of using framework is that : we see leaves but they hind the forest! To better understand the other frameworks and use them efficiently, we'd better be able to design our own framework from scratch. Sometimes we find that most the existing framework does not satisfy our expectation, and we have to design project based on our own framework!

For this purpose, this is a trial project to design a backend server with golang from scratch using as less framework as possible! If you like the idea, we can design it together, and hopes that it helps some beginners clear out some puzzles.

There are so many frameworks for designing back-end server,written in different programming languages. For instance,C++, PHP, Java, Javascript, Python…,etc. I believe a lot of beginners have confusions about choosing language and frameworks.

As to the choice of frameworks, I like the principle of "simplicity is beauty", I think that the frameworks using model-controller-view is kind of making things complicated, especially the idea of controller. After learning several frameworks, I decide to arrange the project as following:



```
[brian@localhost coffeeshop]$ tree
.
├── api
│   ├── product
│   │   ├── authentication.go
│   │   ├── authorization.go
│   │   ├── handler.go
│   │   ├── model.go
│   │   ├── router.go
│   │   └── template
│   │       ├── css
│   │       └── html
│   └── user
│       ├── authentication.go
│       ├── authorization.go
│       ├── handler.go
│       ├── model.go
│       ├── router.go
│       └── template
│           ├── css
│           └── html
├── coffeeshop
├── go.mod
├── go.sum
├── ideas
├── main.go
└── plugins
    ├── authentication.go
    ├── authorization.go
    ├── cache.go
    ├── database.go
    ├── parameterVerification.go
    ├── plugins.go
    └── reqNrespLifecycle.go
```

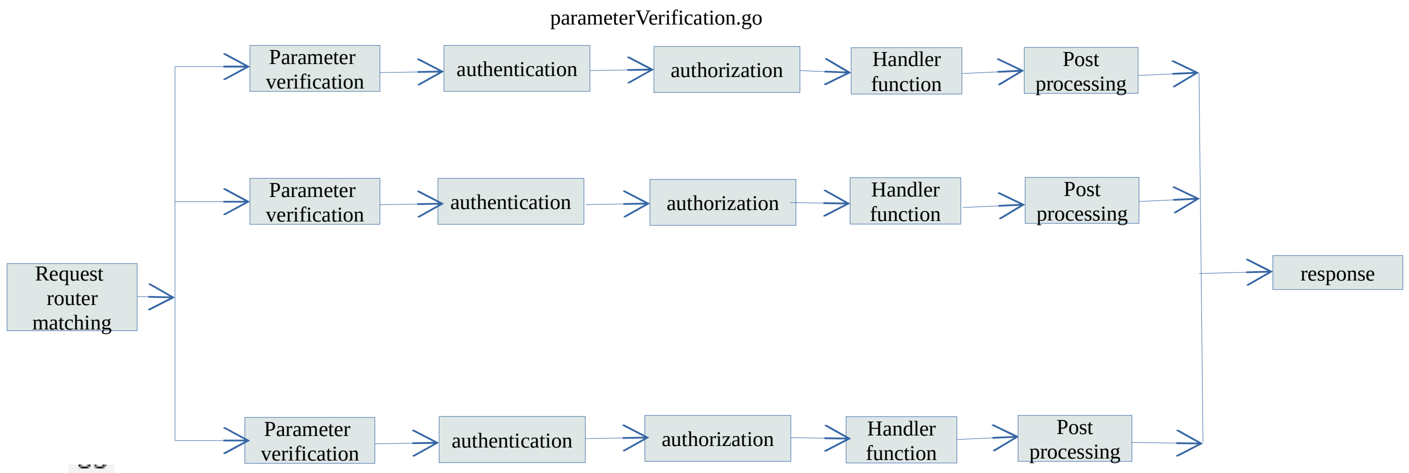
model.go : dealing with database
router.go : connecting request url with corresponding handler functions
handler.go: containing handler functions

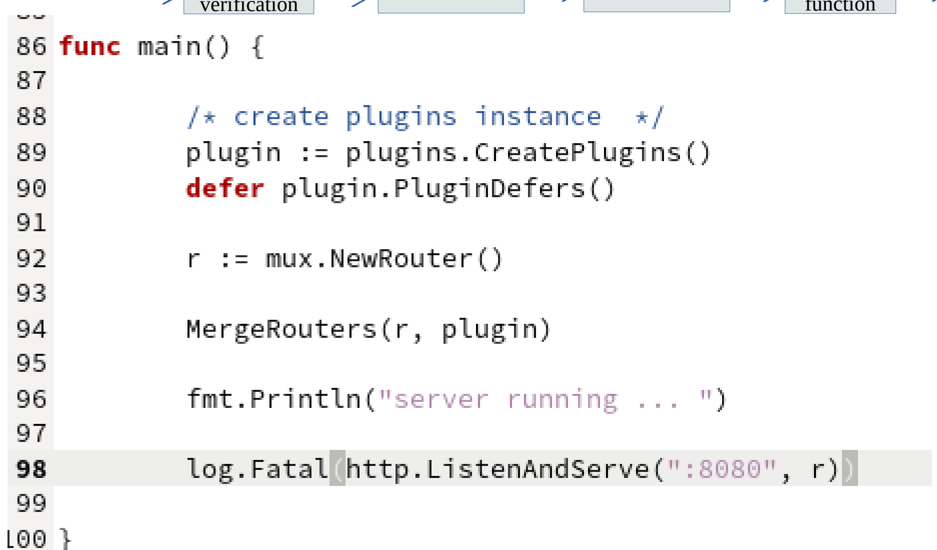
It will be more beneficial to write our own frame. As to programming language, I personally quite like golang and javascript. Golang is more efficient because it is compiled to binary, it executes directly on the native OS, unlike other programming languages, like javascript,java,and PHP, which runs on other “APP” like nodejs. In the old days most people develops backend server with C++, but the development process is quite challenging and time consuming, we have to take care of memory allocation and deletion to make sure the program runs stable. The advantage of using C++ is that it can design extremely heavy server because it runs directly on native OS. But the benefit of using javascript is that the development process is much faster.

Golang makes the development process much faster than C++, and it is very suitable for designing scalable micro webserver . I am kinda of wondering if it could run on embedded devices.

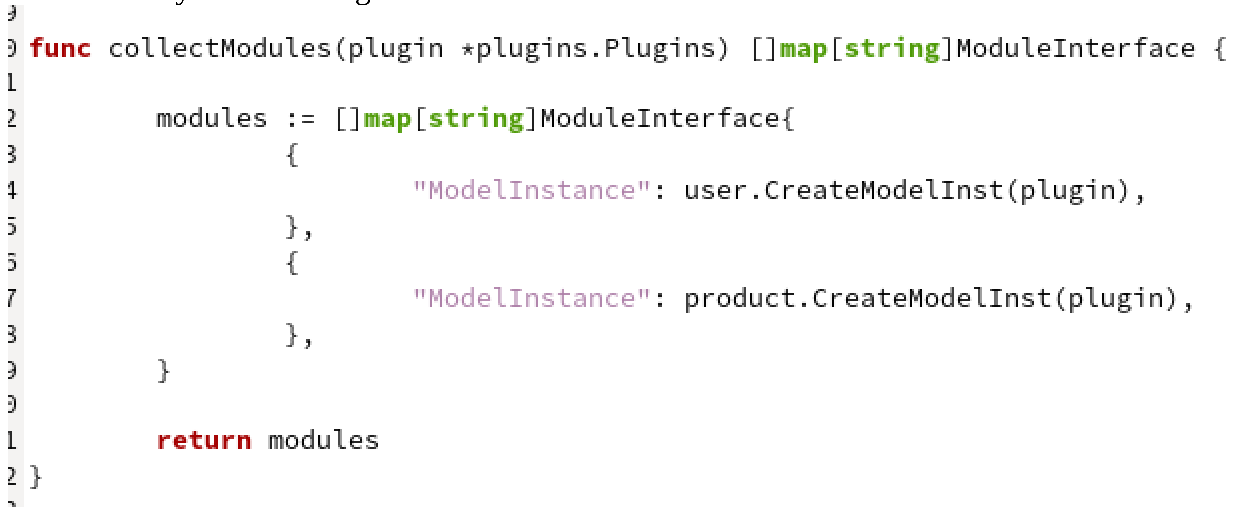
Below is the overall request and response life cycle, make long story short, the core of back-end server is as simple as this life cycle, we can plugin different components into this life cycle, such as authentication, authorization,etc. This components have different names, like middleware, or plugins, here we call it plugins.

Request and response lifecycle



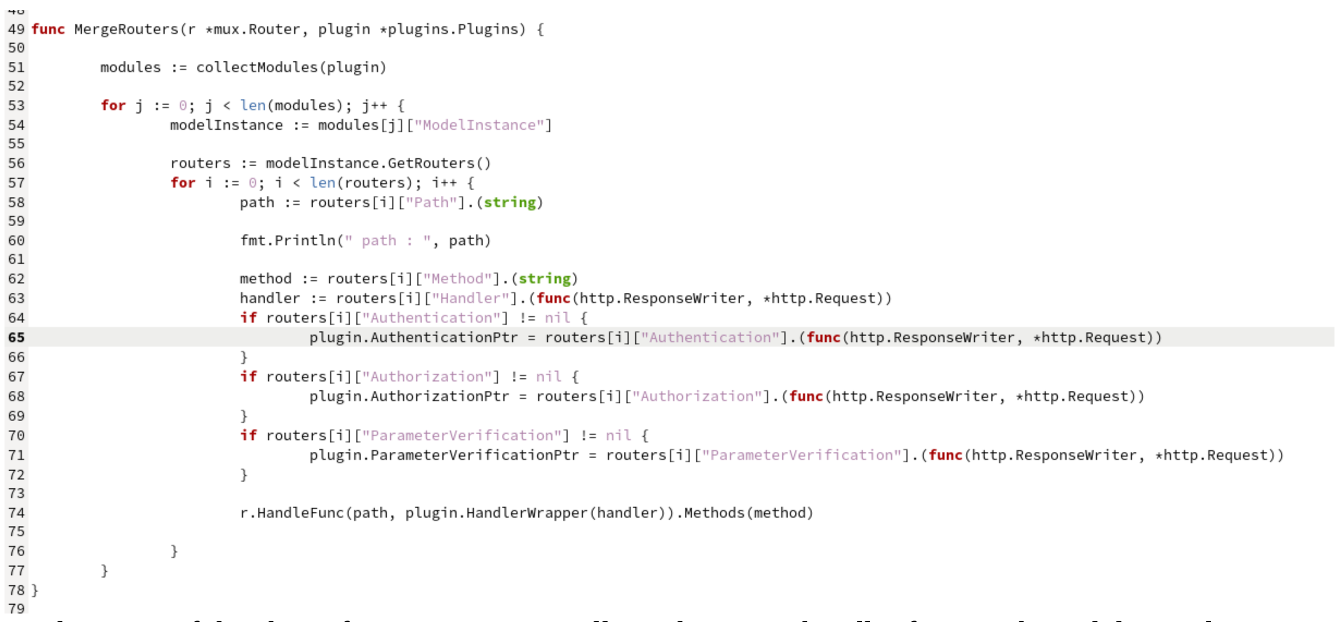

```
func main() {

        /* create plugins instance  */
        plugin := plugins.CreatePlugins()
        defer plugin.PluginDefers()

        r := mux.NewRouter()

        MergeRouters(r, plugin)

        fmt.Println("server running ... ")

        log.Fatal(http.ListenAndServe(":8080", r))

}
```

before we start the server, we need to setup the above router-handler routines.
This is done by the following code section :



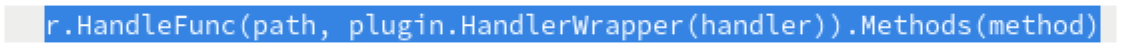
```go
func collectModules(plugin *plugins.Plugins) []map[string]ModuleInterface {

        modules := []map[string]ModuleInterface{
                {
                        "ModelInstance": user.CreateModelInst(plugin),
                },
                {
                        "ModelInstance": product.CreateModelInst(plugin),
                },
        }

        return modules
}
```

because golang is not a script language, we have to manually add module instance to main.go,so the routers know the pointer of the corresponding handler functions. As shown above, modules is an slice which contains the module instances.



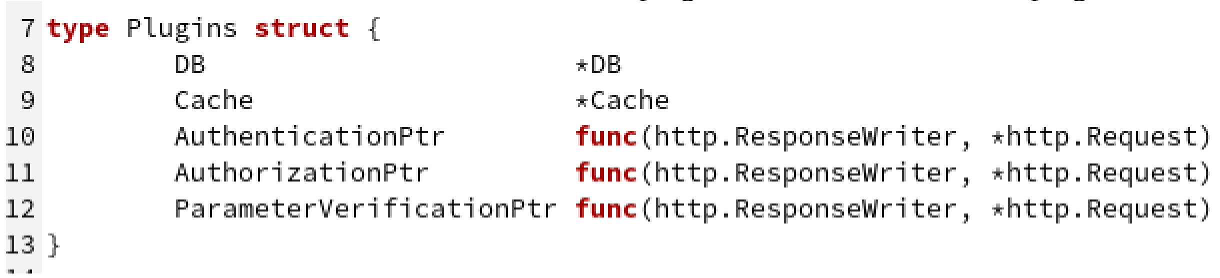
```go
func MergeRouters(r *mux.Router, plugin *plugins.Plugins) {

	modules := collectModules(plugin)

	for j := 0; j < len(modules); j++ {
		modelInstance := modules[j]["ModelInstance"]

		routers := modelInstance.GetRouters()
		for i := 0; i < len(routers); i++ {
			path := routers[i]["Path"].(string)

			fmt.Println(" path : ", path)

			method := routers[i]["Method"].(string)
			handler := routers[i]["Handler"].(func(http.ResponseWriter, *http.Request))
			if routers[i]["Authentication"] != nil {
				plugin.AuthenticationPtr = routers[i]["Authentication"].(func(http.ResponseWriter, *http.Request))
			}
			if routers[i]["Authorization"] != nil {
				plugin.AuthorizationPtr = routers[i]["Authorization"].(func(http.ResponseWriter, *http.Request))
			}
			if routers[i]["ParameterVerification"] != nil {
				plugin.ParameterVerificationPtr = routers[i]["ParameterVerification"].(func(http.ResponseWriter, *http.Request))
			}

			r.HandleFunc(path, plugin.HandlerWrapper(handler)).Methods(method)

		}
	}
}
```

As the name of the above functions says, it collects the router-handler from each modules, and register them to



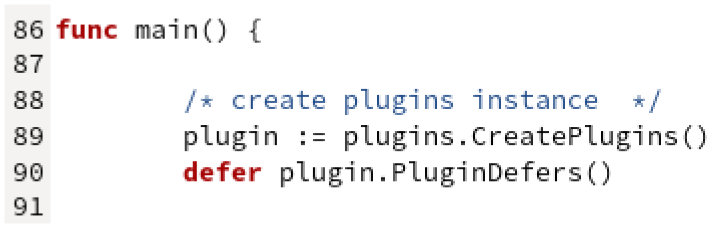
```go
r.HandleFunc(path, plugin.HandlerWrapper(handler)).Methods(method)
```

As to other resources like database, cache, and other plugins, we attach them to the plugin struct,



```go
type Plugins struct {
	DB                       *DB
	Cache                    *Cache
	AuthenticationPtr        func(http.ResponseWriter, *http.Request)
	AuthorizationPtr         func(http.ResponseWriter, *http.Request)
	ParameterVerificationPtr func(http.ResponseWriter, *http.Request)
}
```

and create an instance of the struct in main.go, then pass the plugin instance to each router-handler calls ,so that we can access to the resources in each request handler function. In such a way, we avoid using global variables as much as possible.



```go
func main() {

	/* create plugins instance  */
	plugin := plugins.CreatePlugins()
	defer plugin.PluginDefers()

```

I will try to develop an tutorial on this project, explain in more details. And will try to design an complete project on this.
Happy learning :)