Clean Code Development

1. Detailed comments

During application development, it is especially important to write a lot of comments so that other developers can better understand your project.(cmd/server/main/config.go#L24-L52)

```go
// Init function
func (app *Application) Init(file string) {
    // Import config file
    if err := godotenv.Load(file); err != nil {
        log.Fatal(err) // Check the error
    }
    // Init Config class
    config := &Config{
        connection_string:  os.Getenv("CONNECTION_STRING"), // Set connection string
        addr:   os.Getenv("ADDR"), // Set address

    }
    // Add Config to the Application class
    app.Config = config
    // Init Controller
    app.Controller = &controllers.Controller{
        JWT_secret: []byte(os.Getenv("JWT")), // Set JWT key
    }
    // Init DB connection
    db := app.Controller.DBModel.ConnectDB(app.Config.connection_string)
    // Add DB model class
    dbmodel := &models.DBModel{
        DB: db, // Set Db connection
        Key: os.Getenv("KEY"), // Set encryption key
    }
    // Add Controller to the Application class
    app.Controller.DBModel = dbmodel
    // Init DB migration
    app.Controller.DBModel.InitDB()
}
```

2. Clear function names

When developing an application, it is important to give unambiguous names to functions to make the inputs and outputs easier to understand. (cmd/server/models/folder.go#L43)

```go
// Get Folders by owner and parrent folder function
func (m *DBModel) Get_Folders_By_Owner_and_Parrent_Folder(owner_id int, parrent_folder_id int) ([]Folder){
```

3. Class Design

Using OOP and Classes allows you to make your code more structured. (cmd/server/main/config.go#L12-L23)

```
// Class Application
type Application struct {
    Controller *controllers.Controller
    Config *Config
}
// Subclass Config
type Config struct {
    connection_string string
    addr string
    key string

}
```

4. Strong typing

Strong typing allows you to have better control over the input and output data, and also avoid problems with the application aborting because a function received or returned the wrong data type as input.(cmd/server/controllers/middleware.go#L30)

```
// Verify token function
func (c *Controller) VerifyToken(token_string string ) (error, *Claims) {
```

5. Error handler

Error handlers allow you to constantly monitor possible errors and indicate the exact location and reason for their occurrence. (cmd/server/controllers/middleware.go#L47-L63)

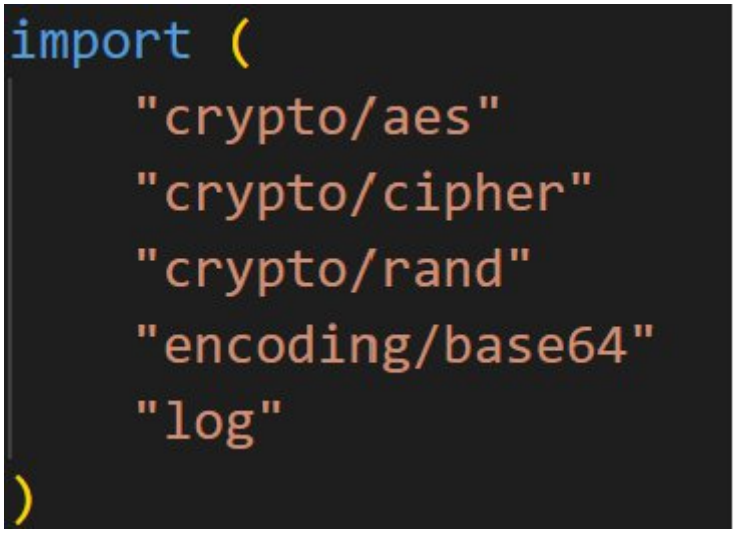
```
if token_string == "" {
    w.WriteHeader(http.StatusUnauthorized) // Set unauthorized status
    return
}
token_string = strings.TrimPrefix(token_string, "Bearer ")
err, claim:= c.VerifyToken(token_string) //Verify token
if err != nil {
    w.WriteHeader(http.StatusUnauthorized) // Set unauthorized status
    return
}

if claim.Role == "Administrator" || claim.Role == "User"{ //check role
    next.ServeHTTP(w, r)
    return
}

w.WriteHeader(http.StatusUnauthorized) // Set unauthorized status
```

6. Using standard packages

The use of standard components allows you to reduce the size of the program, as well as avoid anomalous incorrect work. (cmd/server/models/crypto.go#L4-L7)



7. Clear variables names

When developing an application, it is important to give variables unambiguous names to make the input and output data easier to understand. (cmd/server/models/model.go#L11-L45)

```go
// Role structure
type Role struct {
    ID int `json:"id"`
    Name string `json:"name"`
}
// User structure
type User struct {
    ID int `json:"id"`
    UserName string `json:"username"`
    Password string `json:"password"`
    Role_ID int `json:"role_id"`
    Role *Role `json:"role"`
}
// Folder structure
type Folder struct {
    ID int `json:"id"`
    Name string `json:"name"`
    Owner_ID int `json:"owner_id"`
    Owner *User `json:"owner"`
    Parrent_Folder_ID sql.NullInt64 `json:"parrent_folder_id"`
    Parrent_Folder *Folder `json:"parrent_folder"`
}
// Secret structure
type Secret struct {
    ID int `json:"id"`
    Name string `json:"name"`
    Username string `json:"username"`
    Secret string `json:"secret"`
    Link string `json:"link"`
    Description string `json:"description"`
    Owner_ID int `json:"owner_id"`
    Owner *User `json:"owner"`
    Folder_ID int `json:"folder_id"`
    Folder *Folder `json:"folder"`
}
```

8. Formating

Formatting strings improves the readability of the code and also makes it easier to understand. (cmd/server/controllers/secret.go#L47-L49)

```
    secrets := c.DBModel.Get_Secrets_By_Owner_and_Folder(
        c.DBModel.Get_User_By_UserName(claim.UserName).ID, // Get owner id
        c.DBModel.Get_Folder_By_Name(secret.Folder.Name).ID) // Get folder id
```

9. Using env

Using the environment allows you to avoid storing variables in the code. This is important both for safety reasons and for making modifications easier. (cmd/server/configs/app.env#L1-L4)

```
1   ADDR="0.0.0.0:8080"
2   CONNECTION_STRING="user=smuser password=smuser dbname=sm sslmode=disable"
3   KEY="N1PCdw3M2B1TfJhoaY2mL736p2vCUc47"
4   JWT="secret_jwt_key"
```

10. Testing

Testing is one of the most important parts of writing clean code. Without full testing, the project cannot be considered complete. (cmd/server/main/main_test.go#L13-L38)

```
// Create token test
func Test_Create_Token(t *testing.T) {
    // Init User test structure
    type Test struct {
        username string
        role string
    }
    test := Test{"user", "User"} // Init test data
    app := Application{} //Init application class
    app.Init("../configs/app.env") //Use env variables

    token = app.Controller.CreateToken(test.username, test.role) // Create token
    if token == ""{
        t.Errorf("Error in token generating.")
    }
}

// Verify token test
func Test_Verify_Token(t *testing.T) {
    app := Application{} //Init application class
    app.Init("../configs/app.env") //Use env variables

    if err, _ := app.Controller.VerifyToken(token); err != nil{ //Verify token
        t.Errorf("Error in token verifing.")
    }
}
```