

ESTRATEGIAS DE SEGURIDAD

Práctica Final: Copias de seguridad cifradas con Golang
+ Javascript



Carlos Ascó Rico

Arturo Juan Martínez Sánchez

Contenido del archivo comprimido	3
Servidor	3
BBDD	3
Carpeta backups	4
Cliente back-end	4
Carpeta recover	4
Archivos nombre_usuario.p	4
Cliente front-end	5
myui.go	5
Comunicación con el back-end	6
Manual de usuario	8
Register	8
Login	8
Menu	9
Upload	9
Download	10
Share	10
Stop Share	11
Periodicity	11
Stop Periodicity	12
Documentación sobre la implementación	13
Servidor	13
main.go	13
user.go	15
token.go	16
Cliente	17
main.go	17
myui.go	17
periodical.go	18
auxiliarFunctions.go	19
compress.go	19
user.go	20

Contenido del archivo comprimido

El proyecto podemos dividirlo en tres partes. Una es el servidor, otra es el back-end del cliente, y otra es el front-end del cliente. Así que lo explicaremos por partes:

Servidor

Está compuesto por 3 archivos de código, además de los certificados para hacer uso de SSL, la carpeta backups y la bbdd, que almacena los backup de los clientes.

BBDD

Lo que hace el servidor al principio del todo es abrir el fichero **bbdd** que contiene toda la información de los usuarios, sus claves pública y privada (cifrada previamente en el cliente), la password que le llega al servidor (que recordamos que no es la misma que escribe el cliente), la sal generada aleatoriamente en el servidor para ralentizar ataques por fuerza bruta, y una lista de los ficheros propios y compartidos con el mismo usuario:

```
type user struct {
    Username           string `json:"name"`
    PasswordHashed     []byte `json:"pass"`
    Salt               []byte `json:"salt"`
    PubKey              string `json:"pubkey"`
    PrivKey             string `json:"privkey"`
    Files               []file `json:"files"`
    SharedFilesWithMe []file `json:"shared_files_with_me"`
}
```

Este struct está en el fichero user.go del que hablaremos luego, pero lo mostramos ya que sirve para ver qué es lo que se guarda en el archivo **bbdd**. Este archivo se encuentra cifrado en el servidor, con la contraseña del administrador del servidor, que es introducida al principio de la ejecución del servidor, para que nadie ajeno, pueda coger la base de datos y leerla, al menos sin saber la contraseña del administrador.

```
arturo@arturo: [7 files]
~/cuarto/segundo cuatri/ES/P1/backup-system/server -> go run .
Enter admin password:
Password accepted
```

Carpeta backups

En esta ruta se almacenarán las copias de seguridad de los usuarios organizadas cada una por carpetas con el nombre de los mismos. Cada copia de seguridad estará encriptada con la clave del usuario. Su nombre vendrá predefinido por :

- Tipo de copia de seguridad: manual/periodical.
- Nombre del archivo
- Timestamp de la copia de seguridad.

Ejemplo:

`periodical;Archivo1;2020-05-17 18:49:30.282268236 +0200 CEST m=+2260.447392257`

Las carpetas se crearán automáticamente cuando se suba la primera copia de seguridad del usuario.

Cliente back-end

Carpeta recover

En esta carpeta se encuentra en la carpeta “client” podremos obtener cada una de las copias de seguridad descargadas por el cliente. Si lo que hemos descargado es un archivo será este el que se obtenga al igual que si es una carpeta será esta y todo su contenido el que obtendremos.

Archivos nombre_usuario.p

Este fichero podrá encontrarse en la carpeta “client” por usuarios que hayan creado periodicidades para sus copias de seguridad. Esta información estará encriptada para que ningún otro usuario pueda acceder a la información en la que se guardará cada una de las copias de seguridad junto a su periodicidad y el tiempo restante para empezar con la siguiente copia (información necesaria para ejecutar las periodicidades una vez cerrada la aplicación).

Cliente front-end

myui.go

Para poder realizar la comunicación entre Back-end(Golang) y el Front-end (Html+js) usaremos la librería Lorca. Esta nos proporciona una serie métodos que usaremos para poder llamar a código en Go desde las diferentes vistas.

Cada una de las vistas irán enlazadas con un método creado en myui.go ya que cada una de estas deberá estar unida al código que lancemos desde el Front-end.

```
func (myui *MyUI) chargeView(filePath string) {
    data, err := ioutil.ReadFile(filePath)
    if err != nil {
        panic(err)
    }

    err = myui.ui.Load("data:text/html," + url.PathEscape(string(data)))
    if err != nil {
        panic(err)
    }
}
```

Como podemos ver a continuación esta sería la manera correcta de enlazar cada una de las vistas a nuestro código del Back-end

```
<body>
  <!-- UI layout -->

  <div class="form">
    <div>
      <h2 class="titlemenu">MENU</h2>
      <a onclick="chargeView('www/index.html')" class="iconmenu">&#128282;</a>
    </div>

    <div><h1>Select the action</h1></div><br>
    <div id="outer">
      <div class="inner"><button autofocus onclick="chargeDirectoryFirst('www/...'>
      <div class="inner"><button onclick="chargeView('www/download.html')" class="
    </div>
    <div id="outer">
      <div class="inner"><button onclick="chargeView('www/share.html')" class=
      <div class="inner"><button onclick="chargeView('www/stopshare.html')" cla
    </div>
    <div id="outer">
      <div class="inner"><button onclick="chargeDirectoryFirst('www/periodicity
      <div class="inner"><button onclick="chargeView('www/stopperiodicity.html
    </div>
  </div>
</body>
```

MENU

END

Select the action

Upload Download

Share Stop Share

Periodicity Stop Periodicity

Comunicación con el back-end

Aun así para que podamos acceder a los métodos de cambio de vistas u otros como métodos para recuperar información del sistema o del servidor necesitaremos unir ambos por un túnel que nos facilita el método Bind:

```
myui.chargeView("./www/index.html")
ui.Bind("SignIn", myui.u.SignIn)
ui.Bind("SignUp", myui.u.SignUp)
ui.Bind("EncryptFile", myui.u.EncryptFile)
ui.Bind("DecryptFile", myui.u.DecryptFile)
ui.Bind("chargeView", myui.chargeView)
ui.Bind("SendBackUpToServer", myui.u.SendBackUpToServer)
ui.Bind("RecoverBackUp", myui.u.RecoverBackUp)
ui.Bind("chargeDirectoryFirst", myui.chargeDirectoryFirst)
ui.Bind("chargeDirectory", chargeDirectory)
ui.Bind("ListFiles", myui.u.ListFiles)
ui.Bind("ShareFileWith", myui.u.ShareFileWith)
ui.Bind("GetSharedFiles", myui.u.GetSharedFiles)
ui.Bind("StopSharingFile", myui.u.StopSharingFile)
ui.Bind("AddPeriodicity", myui.u.AddPeriodicity)
ui.Bind("DeletePeriodicity", myui.u.DeletePeriodicity)
```

Por ejemplo para realizar una búsqueda de los ficheros compartidos del usuario necesitaremos llamar a la función **GetSharedFiles** propia del Back-end desde la vista **Stop Sharing** para poder buscar las copias compartidas (Al llamar al método desde nuestra vista deberemos hacerlo de manera asíncrona):

```
catch(error){
    document.getElementById("infoText").in
}
});

async function callChargeFiles(){
    response = await GetSharedFiles();
    var files = response.Msg.split(',');

    if (files == "") {
        select.add(new Option(String.fromCharCode(0x2013)));
        submit.style = "display:none";
    }else{
        var fileIcon = String.fromCharCode(0x2606);
        for(index in files){
            var datos = files[index].split(";");
```

STOP SHARE

BACK END

Select the file to stop sharing

File: p Archivo1 2020-05-17 12:35:56 ▼

Stop sharing

Esta llamada se resolverá en el Back-end pudiendo así obtener la información que necesitamos(en este caso una lista de los archivos compartidos)

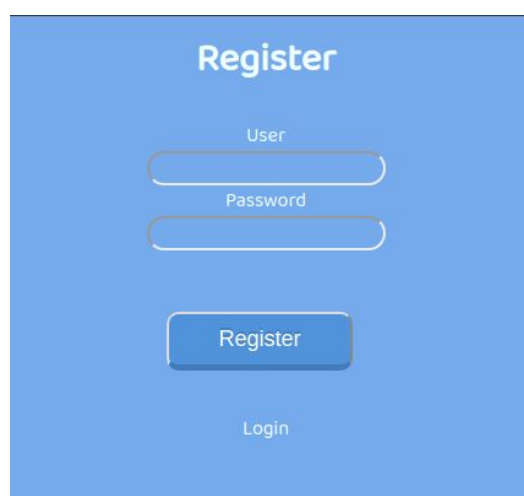
```
user.go x
client > user.go > {} main > (*user).GetSharedFiles
525 func (u *user) GetSharedFiles() (resp, error) {
526     response := resp{}
527     req, err := http.NewRequest("GET", "https://localhost:9043/share", nil)
528     req.Header.Add("token", u.token)
529     res, err := u.httpClient.Do(req)
530     if err != nil {
531         return response, err
532     }
533     defer res.Body.Close()
534
535     //Unmarshal the response to a resp struct
536     body, err := ioutil.ReadAll(res.Body)
537     if err != nil {
538         return response, err
539     }
540     json.Unmarshal(body, &response)
541
542     return response, nil
543 }
```


Manual de usuario

En este manual de usuario vamos a explicar todas las funcionalidades de nuestra aplicación. Para ello explicaremos las funcionalidades de cada una de las vistas que iremos encontrándonos y cómo usarlas correctamente.

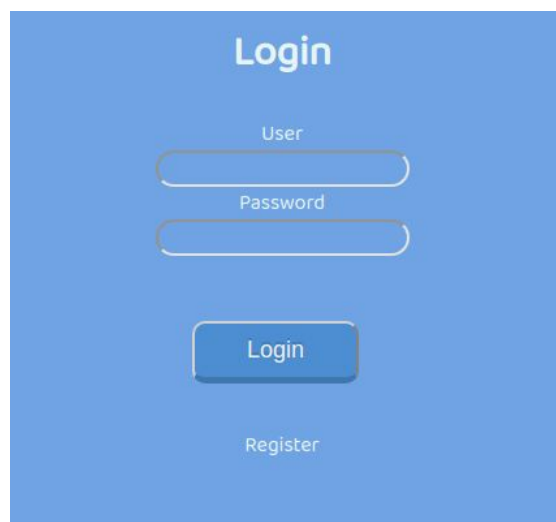
1. Register

En esta vista podremos darnos de alta en la aplicación. Para darnos de alta tendremos que introducir el nombre de usuario y su contraseña. El usuario tiene que tener mínimo **6 caracteres** y la contraseña tener **8 caracteres** en los que debería incluir **al menos 1 número y 1 letra mayúscula**. Una vez rellenados estos valores pulsaremos click en el botón “Register” para darnos de alta. Seguidamente iremos a la vista “**Menu**”


The image shows a 'Register' form on a blue background. At the top, the word 'Register' is written in white. Below it are two input fields: the first is labeled 'User' and the second is labeled 'Password'. Both fields have a light blue border and rounded corners. Below the input fields is a blue button with the word 'Register' in white. At the bottom of the form, the word 'Login' is written in a smaller, lighter blue font.

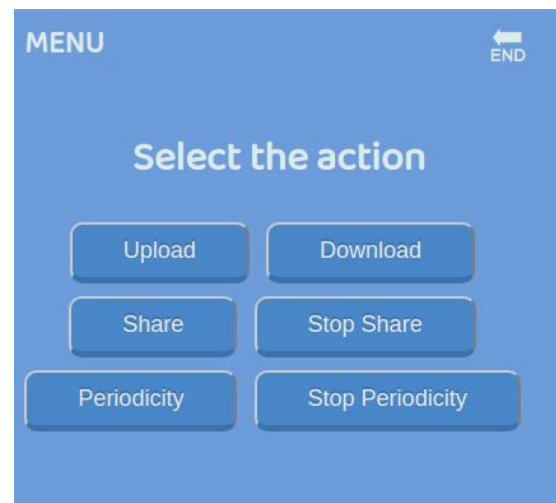
2. Login

En esta vista podremos acceder a la aplicación si estamos dados de alta (vista “**Register**”) anteriormente. Los pasos serían los mismo que para la vista anterior: Rellenar los campos User y Password y pulsar esta vez el botón “**Login**” para acceder a la vista “**Ménu**”

The image shows a 'Login' form on a blue background. At the top, the word 'Login' is written in white. Below it are two input fields: the first is labeled 'User' and the second is labeled 'Password'. Both fields have a light blue border and rounded corners. Below the input fields is a blue button with the word 'Login' in white. At the bottom of the form, the word 'Register' is written in a smaller, lighter blue font.

3. Menu

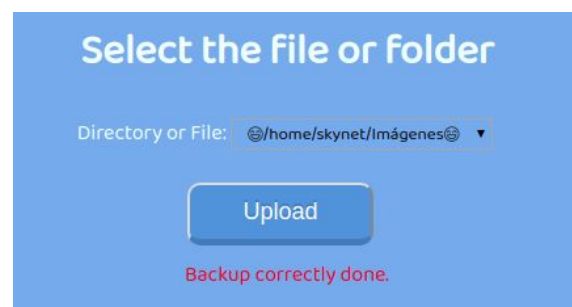
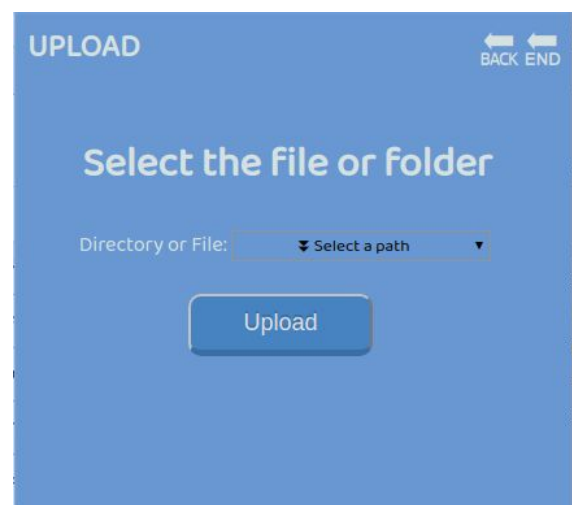
Esta vista será nuestro portal de acceso a todas las funcionalidades en las que podremos clicar en las 6 disponibles y también podremos volver al menú de Login clicando en el botón “ END” de la esquina superior derecha de la vista. (Botón que se comparte en todas las vistas mostradas a continuación)



4. Upload

En esta vista podremos subir al servidor un archivo (pudiendo ser también una carpeta) al servidor. Para ello tendremos que buscar el archivo mediante el menú desplegable indicando con la etiqueta “Directory or File” en el que iremos navegando por el raíz de nuestro equipo hasta seleccionar el archivo a subir.

Una vez seleccionado pulsaremos el botón “Upload” para realizar la subida. Obtendremos un mensaje de confirmación de la acción.



Todas las subidas generadas con esta vista irán etiquetadas con “m” de “manual”

También podremos pulsar el botón “ BACK” para volver a la vista “Menu” (Botón que se comparte en todas las vistas mostradas a continuación)

5. Download

En esta vista podremos descargarnos nuestras copias de seguridad desde el servidor. Para ello tendremos que seleccionar el archivo queramos descargar de una lista de todas nuestras copias disponibles y una vez seleccionada pulsar el botón **“Download”**. Estos archivos serán fácilmente identificables gracias al siguiente patrón:

Tipo subida (m -> Manual, p->Periodical + nombre archivo + Fecha y Hora subida.

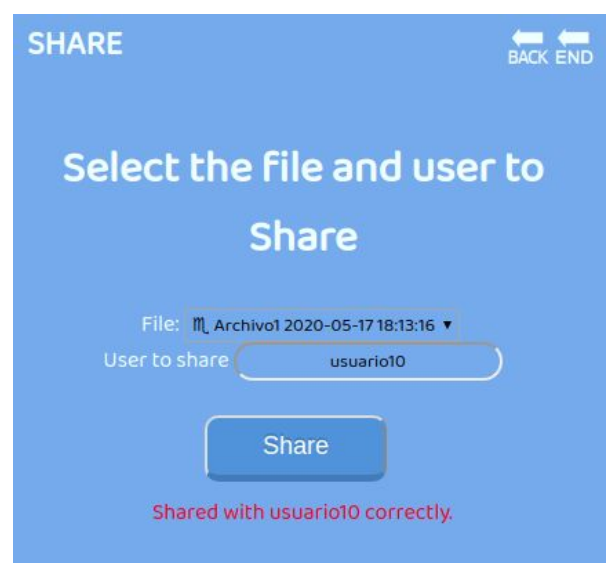
En el ejemplo de la imagen mostramos la subida de la vista “Upload” mostrada anteriormente en el que se puede ver que es una subida **manual** de la carpeta **Imágenes** la fecha de hoy **2020-05-17** a las **12:26:39**

Al realizar la descarga deberíamos buscarla en la carpeta **recover**.



6. Share

En la siguiente vista podremos seleccionar un archivo subido anteriormente y compartirlo con otro usuario dado de alta en el servidor de esta manera podrá descargarse nuestras copias de seguridad. Para ello solo tendremos que seleccionar la copia deseada y indicar el nombre del usuario con el que se quiere compartir el archivo y seguidamente pulsar el botón **“Share”**.



De esta manera podremos comprobar que el usuario indicado puede descargar el archivo desde la vista “Download”



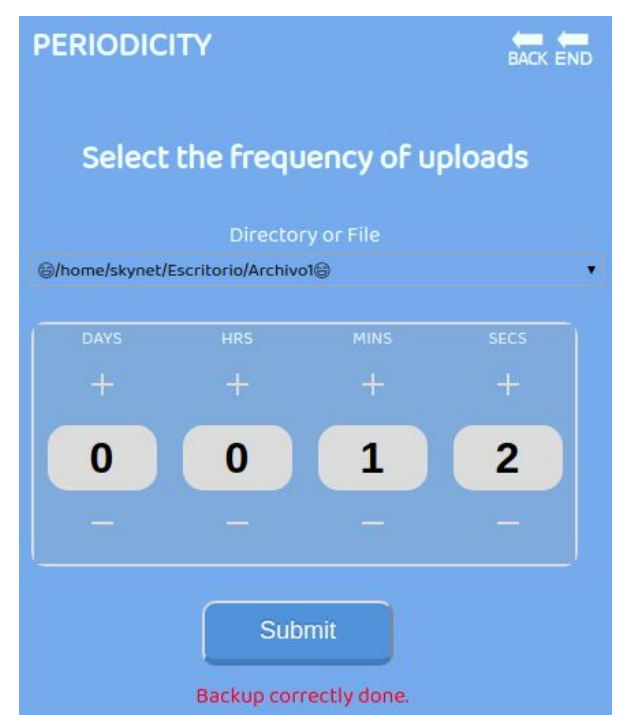
7. Stop Share

En esta vista podremos dejar de compartir un archivo compartido anteriormente con un usuario. Para ello tendremos que seleccionar el archivo y pulsar el botón “Stop sharing”.



8. Periodicity

En esta vista podremos subir archivos como en la vista “Download” pero además podremos añadirles una periodicidad con la que se hará automáticamente la copia de seguridad del archivo seleccionado. Para ello solo tendremos que indicar el número de días, horas, minutos ,y segundos que ha de esperar la aplicación hasta hacer la siguiente copia.



Las copias de seguridad realizadas con esta vista estarán indicadas con el tipo “p” de “Periodical”

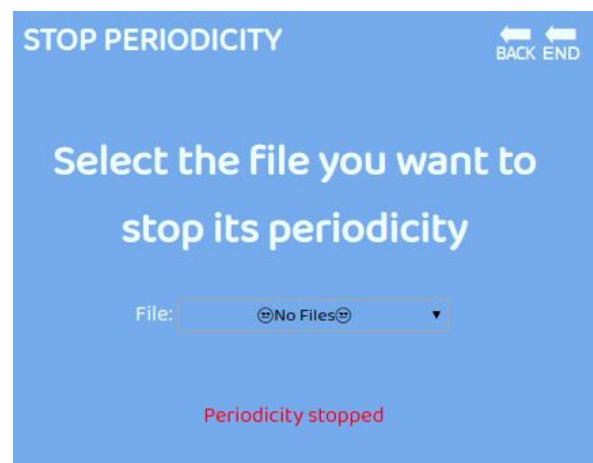


9. Stop Periodicity

En esta vista podremos cancelar una periodicidad, de esta manera cancelaremos la subida del archivo seleccionado y ya no haremos mas copias periódicas de este.



Para ello tendremos que seleccionar el archivo del cual estamos haciendo la periodicidad (este estará indicado con el nombre del archivo y su periodicidad) y pulsar el botón “Stop”.



Documentación sobre la implementación

Los archivos de código se distinguen por la terminación `.go`, que es la terminación de los archivos que contienen código del lenguaje Go utilizado en nuestra práctica.

Servidor

Todas las funciones del servidor residen en la carpeta **server**, y las explicaremos distinguiendo qué hace cada fichero:

`main.go`

En el servidor hemos hecho uso de librerías que trae Go por defecto como la de “net/http”, “io/ioutil” o “encoding/json” entre muchas otras.

Para la comunicación entre cliente y servidor quisimos usar TLS, para una comunicación cifrada, pero el manejo de buffers demás lo hacía algo complicado, pero por suerte, la librería “net/http” trae una función “ListenAndServeTLS()” que nos permite utilizar el protocolo HTTP, bastante más sencillo para la comunicación y con el cifrado TLS incorporado.

Así que una vez el administrador ha introducido la contraseña de cifrado de la BBDD, el servidor se pone a la escucha, con un previo enlace de los handlers que ahora hablaremos

```
http.HandleFunc("/register", registerHandler)
http.HandleFunc("/login", loginHandler)
http.HandleFunc("/backup", backupHandler)
http.HandleFunc("/share", shareHandler)
http.HandleFunc("/keys", keysHandler)
http.HandleFunc("/keyfile", keyfileHandler)

err = http.ListenAndServeTLS(":9043", "certificates/server.crt", "certificates/server.key", nil)
if err != nil {
    panic(err)
}
```

Todos estos métodos se encuentran en el archivo `server/main.go`, y sus nombres son bastante descriptivos, pero haremos una breve descripción de lo que hace cada uno (hay que aclarar que en todos los métodos se comprueba previamente que la petición lleve un token, cosa que explicaremos posteriormente):

- **registerHandler:** recoge los datos de registro (usuario, password, claves pública y privada) mediante un post-form, lo almacena en la bbdd en caso de ser todo válido, y devuelve información descriptiva.
- **loginHandler:** recibe los datos de un usuario mediante un post-form y devuelve información descriptiva como si el usuario no existe o las credenciales son inválidas.
- **backupHandler:** acepta sólo métodos GET y POST.
 - Si es un GET, y el cuerpo va vacío, devuelve todos los back up del usuario listados en un string, en cambio si el cuerpo tiene contenido, está pidiendo un back up, y en la respuesta irá el contenido del back up, en caso de que exista el mismo
 - Si es un POST, el cliente está pidiendo almacenar un back up en el servidor, por lo que en los headers de la petición habrá información como la clave de encriptación del fichero (encriptada con clave pública) y en el body, el contenido del back up.
- **shareHandler:** acepta sólo métodos GET, DELETE y POST.
 - Si es un método GET, devuelve un string con los archivos que el usuario ha compartido listados.
 - Si es un DELETE, recibe la nueva clave de encriptación y el nombre del fichero mediante los headers, y a continuación deja de compartir ese archivo con todo el mundo.
 - Si es un POST, recibe el nombre del fichero, el del amigo al que le quiere compartir y la clave de encriptación (encriptada con la clave pública del amigo) en los headers, y el servidor almacena esa información entre los archivos compartidos del amigo.
- **keysHandler:** este manejador es muy simple, sólo devuelve las clave pública y privada en el header de respuesta, en caso de que el header de la petición sea la cadena “me”, si es otra cadena, busca entre todos los usuarios de la BBDD y le devuelve sólo su clave pública.
- **keyfileHandler:** este manejador también es bastante simple, recibe el nombre del back up en los headers, y devuelve la clave de encriptación (cifrada con la clave pública del usuario).

Como aclaración, para saber quién está haciendo la petición, se usa un token que siempre viaja encriptado mediante TLS en las peticiones HTTP en el header, esto permite, no sólo identificar quién hace la petición, sino que también, no tener que enviar siempre el usuario y la contraseña de quien la hace. A excepción del login y el registro, que la información viaja en un post-form, y no existe aún el token, porque lo crea y lo devuelve el propio

handler. Y también hay un estándar de respuesta que es la siguiente estructura:

```
// Response type to communicate with the client
type resp struct {
    Ok bool
    Msg string
}
```

Que es la que se usa en la mayoría de respuestas de los handler, a excepción de cuando se quiere recuperar un back up.

user.go

```
type file struct {
    From      string `json:"from"`
    Name      string `json:"name"`
    Key       string `json:"key"`
    IsShared  bool   `json:"is_shared"`
}

type user struct {
    Username          string `json:"name"`
    PasswordHashed    []byte `json:"pass"`
    Salt              []byte `json:"salt"`
    PubKey            string `json:"pubkey"`
    PrivKey           string `json:"privkey"`
    Files             []file `json:"files"`
    SharedFilesWithMe []file `json:"shared_files_with_me"`
}
```

En este fichero usaremos librerías nativas de Golang como “crypto/sha256”, “bytes”, “crypto/aes”, entre otras. Este fichero contiene funciones, descriptivas por su propio nombre sobre el manejo del usuario en el servidor, aunque también las explicaremos:

- **GetKey:** devuelve la clave encriptación de un fichero (previamente encriptada con clave pública), en caso de existir.
- **MyFiles:** devuelve un string con los back up pertenecientes al usuario y los que le han sido compartidos.
- **GetFullPath:** recibe un nombre de un back up, y en caso de estar entre sus ficheros, devuelve el path del servidor donde se encuentra.
- **SharedFiles:** devuelve un string con los archivos que el usuario está compartiendo, listados.

- **StopSharing:** recibe el nombre del backup que quiere dejar de compartir y la nueva clave de encriptación. Deja de compartir ese archivo con todo el mundo y almacena la nueva clave de encriptación.
- **AddSharedFileWithMe:** simplemente añade el fichero y la clave de encriptación del mismo en el array de ficheros compartidos consigo mismo del usuario.
- **DeleteSharedFileWithMe:** borra el fichero del array de ficheros compartidos consigo mismo.
- **Hash:** esta función no sólo hashea la contraseña y la sal sino que también almacena la contraseña (más sal y posteriormente hasheada) y la sal en el struct de usuario.
- **CompareHash:** recibe una contraseña sin hashear, recupera la sal del usuario y lo hashea todo, en caso de que coincida con la contraseña almacenada del usuario devuelve true, en caso contrario false.
- **EncryptContent:** recibe un array de bytes que es el contenido, y lo devuelve encriptado con aes con la contraseña del usuario como clave. Este método es sólo usado por el usuario administrador.
- **DecryptContent:** exactamente igual que el método anterior, pero para descifrar.

token.go

Este fichero sólo hace uso de la librería “time”, ya que en él lo que vamos a hacer es implementar funciones propias de un token que expira al tiempo. Se compone de un struct Token y otro struct Tokens que simplemente sirve para manejar todos los tokens del servidor. Los nombres de funciones son bastante descriptivas pero las describiremos más a fondo:

- **Exists:** recibe un token y devuelve su posición en el array y true si existe, o -1 y false en caso contrario.
- **Add:** genera un token para ese usuario y lo añade a la lista. Devuelve el string identificativo del token.
- **Delete:** recibe un token y lo borra si existe en el array y devuelve true, si no devuelve false.
- **DeleteExpireds:** mira todo el array y los tokens que tengan una fecha posterior a ahora, los quita del array.
- **Owner:** recibe el identificador del token y devuelve el propietario del mismo.

- **isOutdated:** recibe una fecha y devuelve true si el token ha expirado, en caso contrario false.

Cliente

Los fuentes del cliente se almacenan en la carpeta **client**, y dentro de ella tenemos la carpeta **www** que es la que tiene los html que se usan como interfaz (ya explicado anteriormente).

main.go

Este archivo sólo inicializa la interfaz y hace los links entre el servidor y el cliente. También inicializa el contador único que es usado para evitar conflictos creando copias de seguridad en la función **SendBackUpToServer** que veremos a próximamente.

```
u := user{}
u.backUpIdentifier = 0
u.mutex = &sync.Mutex{}
```

myui.go

```
//MyUI is an struct to be able to charge views from javascript
type MyUI struct {
    ui lorca.UI
    u user
}
```

Contiene funciones para el renderizado de la interfaz, como siempre, son bastante descriptivas pero las explicaremos brevemente:

- **chargeView:** recibe el path donde se encuentra el fichero html de la interfaz y lo muestra en la aplicación.
- **chargeViewDownload:** hace lo mismo que chargeView pero el html recibe más datos porque tienen un desplegable.
- **chargeDirectoryFirst:** hace lo mismo que chargeView pero con las vistas que necesitan moverse entre directorios.
- **chargeDirectory:** recibe un directorio del sistema y devuelve todas las carpetas y archivos que tiene en un array. Sirve para que el cliente pueda encontrar el directorio donde quiere hacer el back up.

periodical.go

```
//Periodical is to manage the periodical back ups
type Periodical struct {
    ID            int
    Path          string
    TimeToUpload  time.Duration
    NextUpload    time.Time
    stopchan      chan struct{}
}
```

Contiene funciones que son propias del struct **user**, pero no están en el fichero **user.go** porque era demasiado extenso y estas funciones se podían separar porque todas ellas eran parte de las periodicidades. Nótese que las funciones que **empiezan por mayúscula** son públicas ya que serán **llamadas desde JavaScript**, y hay algunas con el mismo nombre pero empezando por minúscula y hacen algo totalmente distinto.

- **readPeriodicity**: lee un fichero **nombre_del_usuario.p** que contiene toda la información sobre las periodicidades del usuario. Por supuesto este fichero se descifra porque está cifrado y se unmarshalea (pasa de json a struct) en el struct **Periodical**.
- **loopPeriodicity**: recorre todas las periodicidades del array de **Periodicity** y lanza una go routine (un hilo) por cada una de ellas. Además si el campo **NextUpload** es más antiguo que la fecha actual, pasa a ser fecha actual + cada cuánto se hace una copia de seguridad.
- **isOutdated**: recibe una fecha y devuelve true si es posterior a ahora, en caso contrario false.
- **addPeriodicity**: esta es una de las funciones más importantes en cuanto a la periodicidad, es usada por **loopPeriodicity** para ser usada como go routine. Primero crea un canal con **time.After()** de la librería oficial de go, que devuelve un canal que recibe información cuando ha pasado el tiempo indicado, este tiempo se calcula restando **time.Now()** menos **Periodical.NextUpload**. Entonces se inicia un bucle infinito que sólo para cuando **Periodical.stopchan** es cerrado, y cuando pasa el tiempo mencionado anteriormente se hace el back up y se resetea el tiempo a **Periodical.TimeToUpload**.
- **AddPeriodicity**: esta función es llamada desde JavaScript y recibe un path y cada cuánto se hará la copia de seguridad, se añade al array de copias de seguridad y se escribe cifrado en su fichero de configuración. Devuelve un response con información relevante sobre si ha ido bien el

proceso, similar al que se usa en otras funciones de user.go para mantener la concordancia.

- **deletePeriodicity:** recibe un id que es buscado en el array de Periodical y si existe, lo elimina del array, en caso contrario devuelve un error distinto de nil.
- **DeletePeriodicity:** esta función es llamada desde JavaScript y lo que hace es cerrar el canal de **Periodical.stopchan** y devuelve un response con información relevante sobre la ejecución del método.
- **GetPeriodicity:** es usado en JavaScript. Devuelve un array de **PeriodicalParse** en lugar de **Periodical** a secas porque JavaScript no es capaz de entender el **time.Duration** de **Periodical.TimeToUpload**.

auxiliarFunctions.go

Tiene algunas funciones que no sabíamos dónde meter, son pocas y descriptivas, pero la explicación es la siguiente:

- **RandStringBytes:** recibe un número n y devuelve un array de bytes con caracteres alfanuméricos aleatorios.
- **encode64:** codifica el array de bytes a base64.
- **decode64:** decodifica el string en base64 a array de bytes.

compress.go

Contiene funciones usadas para comprimir y descomprimir.

- **compressData:** recibe un array de bytes y lo devuelve comprimido
- **uncompressData:** recibe un array de bytes comprimido y lo devuelve descomprimido.
- **compressFile:** recibe un path source, que es la carpeta o el archivo que se va a comprimir en zip, y un path target que es dónde se va a generar el archivo.zip y su nombre.
- **uncompressFile:** recibe un path archive que es donde se encuentra el archivo.zip y su nombre, y lo descomprime en el path target.

user.go

```
//user is used to log in the server, send and recover files, etc.
type user struct {
    username            string
    passwordLogInServerHassed []byte
    cipherKey           []byte
    httpClient          *http.Client
    token               string
    pubKey              *rsa.PublicKey
    privKey              *rsa.PrivateKey
    periodicals         []Periodical
    backUpIdentifier    int
    mutex               *sync.Mutex
}
```

Este es el archivo donde reside la mayoría de la lógica del back-end del cliente.

En los métodos que se comunican con el servidor siempre se devuelve un response igual al del servidor y un posible error de ejecución del método. Además en la mayoría se usa envía un string llamado **token** al servidor que sirve como autenticación.

Las funciones son mayormente para comunicarse con el servidor, aunque algunas sirven de pasos intermedios que se realizan varias veces (cifrar, descifrar, etc.):

- **Hash:** recibe un password, lo hashea y lo divide en dos, la mitad será enviada al servidor para ser logeado, y la otra mitad servirá para cifrar la clave privada del cliente.
- **sign:** es una función auxiliar, ya que SignIn y SignUp hacían cosas muy parecidas, decidimos convergerlas en una sola. Recibe un nombre usuario, una password y un comando ("login" ó "register"). Almacena el nombre, llama a la función Hash con la contraseña, crea el cliente http que será utilizado en varias ocasiones posteriormente (el cliente http tiene el flag **InsecureSkipVerify** a **true** porque los certificados del servidor son autofirmados), llama a la función **AuthorizeOnServer** y si el proceso ha ido bien (response.Ok es true) se inicializan los hilos de periodicidades de sesiones anteriores.
- **SignIn:** sirve para loguearse ante el servidor, inicializa las claves pública y privada para luego ser almacenadas en la respuesta del servidor.

- **SignUp:** sirve para registrarse. Genera una clave pública y privada y las manda al servidor para que las guarde (la privada cifrada obviamente en métodos posteriores).
- **LogOut:** cierra todos los go routines (hilos) para evitar conflictos
- **encrypt:** recibe un array de bytes que es el contenido a cifrar, y una clave, en caso de la clave ser nula, se usa la primera mitad de la password del usuario hasheada. Devuelve el contenido cifrado.
- **EncryptFile:** recibe un path y una clave de cifrado, abre ese el archivo del path, lo cifra y lo sobrescribe con el contenido cifrado.
- **decrypt:** recibe un array de bytes que es el contenido a descifrar, y una clave, en caso de la clave ser nula, se usa la primera mitad de la password del usuario hasheada. Devuelve el contenido descifrado.
- **DecryptFile:** recibe un path y una clave de cifrado, abre ese el archivo del path, lo descifra y lo sobrescribe con el contenido descifrado.
- **AuthorizeOnServer:** recibe un comando ("login" ó "register") y genera un post-form con el usuario, la segunda mitad de la password hasheada y las clave pública y privada (cifrada con la primera mitad de la password hasheada) en caso de ser un registro. En caso de ser un login leerá las claves pública y privada (que será descifrada con la primera mitad de la password hasheada).
- **SendBackUpToServer:** recibe un path que es lo que el cliente va a enviar, y un booleano indicando si la copia es periódica o no. Tendrá que comprimir y cifrar lo que vaya a enviar, por lo que se genera un **nombre único**, y para evitar que haya conflictos en el nombre del mismo, se **usa un mutex** que bloquea la ejecución de otros hilos que vayan a usar esas variables hasta que se llame a Unlock. Entonces envía el contenido del back up y elimina el archivo temporal.
- **RecoverBackUp:** recibe un nombre, que será el nombre que tiene el back up en el servidor y le hace la petición al servidor y este devuelve el contenido del mismo. Este contenido se descifra y se descomprime en la carpeta **recover**.
- **downloadFile:** esta función sirve como complemento a la anterior, aunque también se usará en otras como la de **StopSharingFile**. Recibe un nombre de back up (en el servidor) y devuelve su contenido y la clave de cifrado del mismo.
- **ListFiles:** como no podemos conocer de primeras los nombres que le ha puesto el servidor a los back up, este método le pide al servidor los nombres de estos, y los devuelve en el response.

- **ShareFileWith:** este método le pide al servidor la clave de cifrado de un archivo y la clave pública del usuario al que le queremos compartir, entonces descifra con su clave privada la clave de cifrado del fichero, y la vuelve a cifrar con la clave pública del otro usuario.
- **getFriendPubKey:** esta función recibe un nombre de usuario, hace una petición al servidor y devuelve la clave pública de ese usuario.
- **getFileKey:** esta función recibe un nombre de archivo, hace una petición al servidor y devuelve la clave de cifrado del archivo (cifrada con la clave pública del usuario).
- **StopSharingFile:** recibe el nombre del fichero (en el servidor) que queremos dejar de compartir, genera una nueva clave de cifrado del archivo y vuelve a cifrar el back up, entonces cifra la nueva clave del back up con la clave pública y la envía al servidor (la clave de cifrado del archivo cifrada) junto al contenido del back up (cifrado con la nueva clave).
- **GetSharedFiles:** devuelve los ficheros que el usuario ha compartido con otros usuarios.