

CrypterCS

Proyecto grupal de Compresión y Seguridad desarrollado por:

Javier Martín Gómez

Pablo Gutiérrez Gomis

Carlos Díez de Castro

Àngel Torregrosa Calabuig

Índice

1.	Contenido de los archivos comprimidos y enlaces de descarga	3
2.	Tecnologías del proyecto	4
3.	Librerías	5
4.	Manual de usuario	6
2.1.	Ejecución de la aplicación	6
2.2.	Ventanas de la aplicación	7
2.2.1.	Ventana de Login y Registro_____	7
2.2.2.	Ventana de Usuario_____	9
2.3.	Encriptar un archivo	10
2.3.1.	Selección de los archivos	10
2.3.2.	Encriptar	11
2.4.	Desencriptar un archivo	12
2.5.	Compartir un archivo	13
5.	Protocolo de seguridad	16
6.	Detalles de implementación	18
7.	Posibles mejoras	26
8.	Bibliografía	27

Contenido de los archivos comprimidos y enlaces de descarga

Ejecutable

Enlace: [CS_Ejecutable_Final.zip](#)

Para la ejecución de la aplicación, descomprimos el archivo zip. Una vez descomprimido el zip, hacemos doble clic sobre el ejecutable ([crypter-cs.exe](#)) y ya tenemos la aplicación en ejecución.

Código

Enlace: [CS_Codigo_Final.zip](#)

La estructura del proyecto por carpetas es la siguiente:

- **db**: Relativo a la base de datos y su manejo
- **dist y dist_electron**: Código generado al construir el proyecto.
- **files**: Donde se guardan los archivos encriptados y desencriptados, divididos en carpetas organizadas por nombre de usuario.
- **node_modules**: Librerías utilizadas para el desarrollo del proyecto.
- **public**: Archivo padre de la web donde Vue.js se indexa el código que genere.
- **src**: Código desarrollado por nosotros.
 - **assets**: Estilos, imágenes,...
 - **components, pages y layouts**: Componentes de Vue.js. Bloques de código que conforman la interfaz, y algunas funciones relativas a su manejo.
 - **plugins y helpers**: Funciones que ayudan al desarrollo del código en distintas partes del proyecto.
 - **router**: Enrutador para manejar las distintas páginas.
 - **services y store**: Funciones con la lógica del proyecto. Donde además se emplean las librerías utilizadas.

Tecnologías del proyecto

Vue.js con TypeScript

vuejs.org

Vue.js es un framework progresivo, desarrollado en JavaScript, que permite dividir las aplicaciones en bloques con funcionalidades independientes, llamados componentes. Utilizamos este framework para desarrollar la interfaz gráfica de nuestro proyecto de manera más ágil.

TypeScript nos da el tipado de datos sobre JavaScript, para hacer nuestro código más consistente y menos propenso a fallar.

Electron

electronjs.org

Permite el desarrollo de aplicaciones gráficas de escritorio usando componentes del lado del cliente y del servidor originalmente desarrolladas para aplicaciones web: Node.js del lado del servidor y Chromium como interfaz, en nuestro caso la interfaz es Vue.js.



Base de datos

Hemos creado nuestra base de datos a partir de un fichero en plano JSON. Todos los cambios se van actualizando para garantizar la persistencia, y son cargados al iniciar la aplicación.

El funcionamiento está basado en la librería [NeDB](#).

Librerías

CryptoJS

cryptojs.gitbook.io/docs/

Es una librería de algoritmos criptográficos estándar y seguros implementados en JavaScript. Son rápidos y tienen una interfaz simple y consistente.

Node-RSA

github.com/rzcoder/node-rsa

Librería basada en la librería jsbn de Tom Wu (<http://www-cs-students.stanford.edu/~tjw/jsbn/>). No necesita OpenSSL, genera el par de claves y soporta mensajes largos para encriptar y desencriptar.

SHA3

github.com/phusion/node-sha3

Librería de algoritmos hash implementada en Javascript. Incluye principalmente Keccak y SHA3.

ESLint y Prettier

eslint.org y [prettier](https://prettier.io)

Ambas son librerías para estandarizar el código en desarrollos de varias personas, y detectar código sospechoso, confuso o incompatible.

File-system

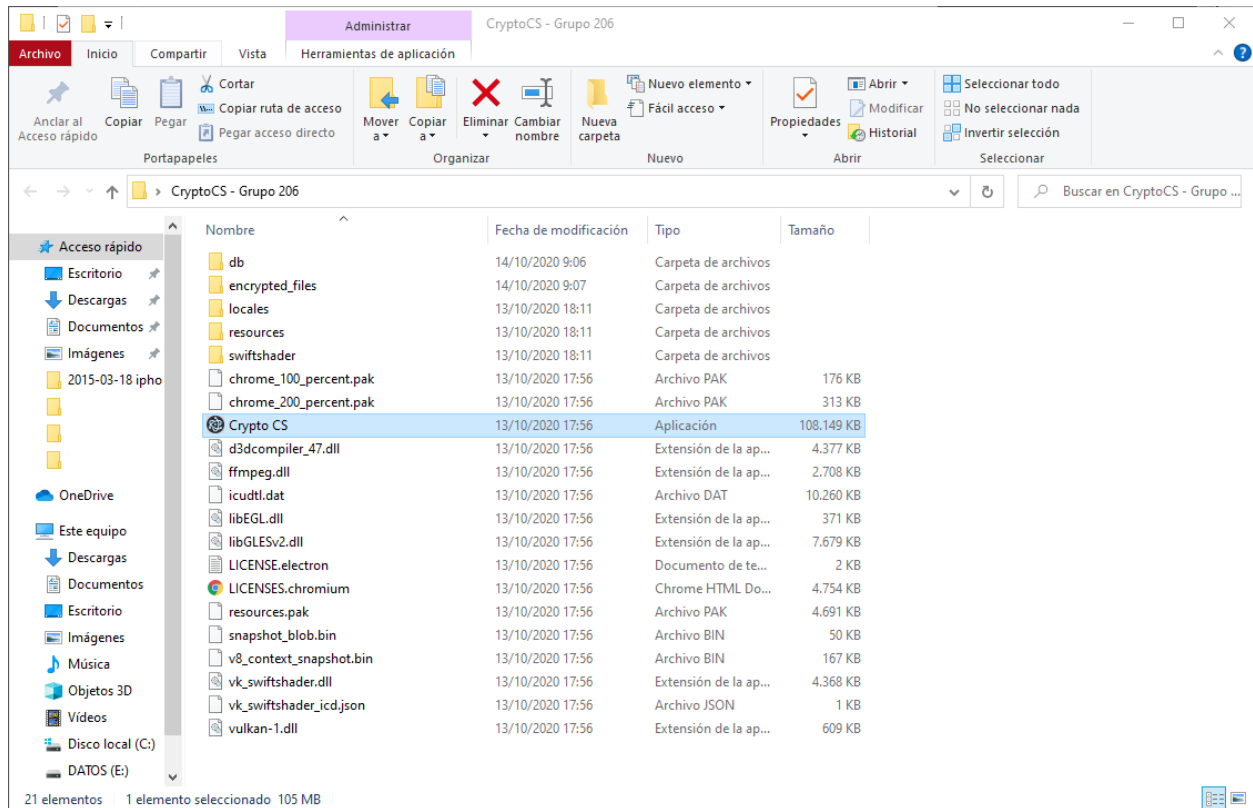
npmjs.com/package/file-system

Para gestionar archivos: Leer, copiar, escribir...

Manual de usuario

Para poder usar la app hemos creado una aplicación de escritorio que corre en Windows. Por tanto a la hora de usar nuestra aplicación no hace falta ningún comando.

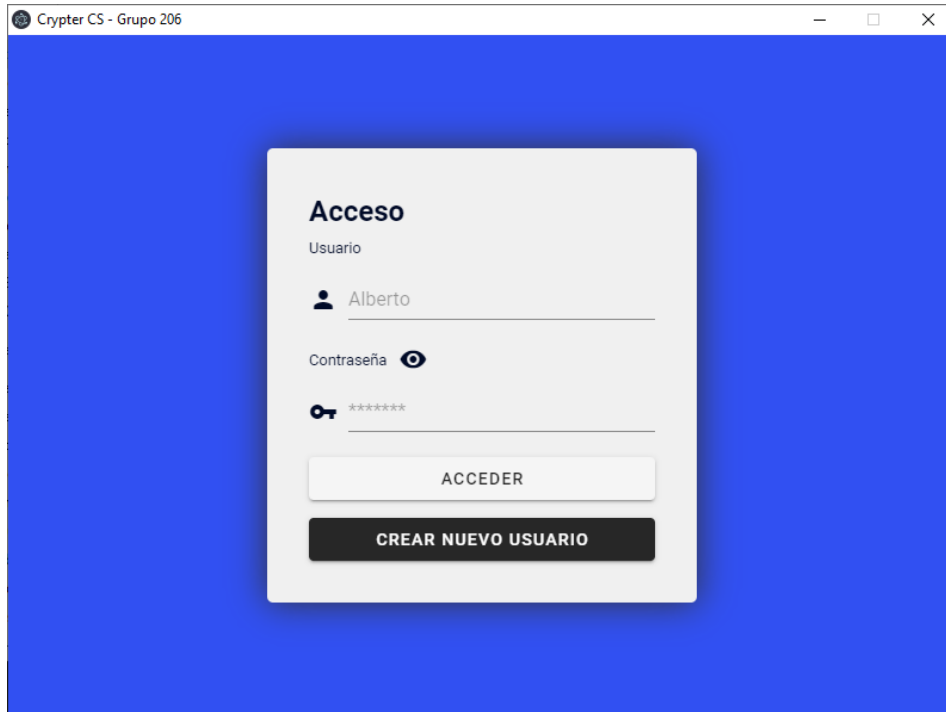
1. Ejecución de la aplicación



Para la ejecución de la aplicación, descomprimos el archivo zip. El archivo ejecutable ,CryptoCS.exe, tiene que estar dentro de la carpeta como mostramos en la imagen anterior, ya que este archivo necesita las dependencias que hay en ella. Una vez descomprimido el zip, hacemos doble clic sobre el ejecutable y ya tenemos la aplicación en ejecución.

2. Ventanas de la aplicación

2.1. Ventana de Login y Registro



The screenshot shows a web application window titled "Crypter CS - Grupo 206". The background is a solid blue color. In the center, there is a white rectangular form with a subtle shadow. The form is titled "Acceso" in bold black text. Below the title, there are two input fields. The first is labeled "Usuario" and contains the text "Alberto". The second is labeled "Contraseña" and contains seven asterisks "*****". To the right of the password field, there is a small eye icon. Below the input fields, there are two buttons. The first button is white with a black border and contains the text "ACCEDER". The second button is solid black with white text and contains the text "CREAR NUEVO USUARIO".

Para crear un usuario válido:

Usuario: mínimo de caracteres 4.

Contraseña: mínimo de caracteres 16 y máximo 26.

Si no cumple estos requisitos, se mostrarán mensajes para avisar al usuario.

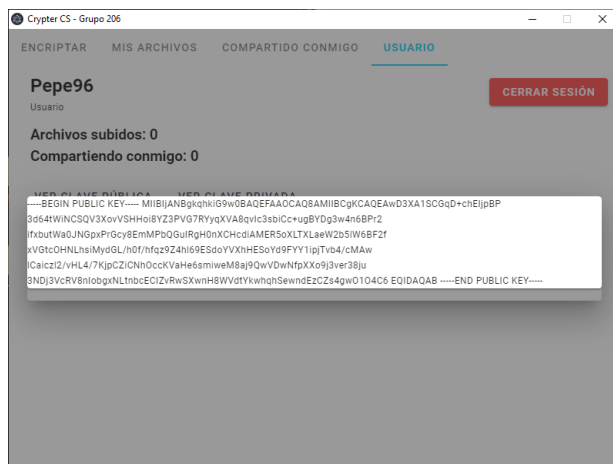
2.2. Ventana de Usuario



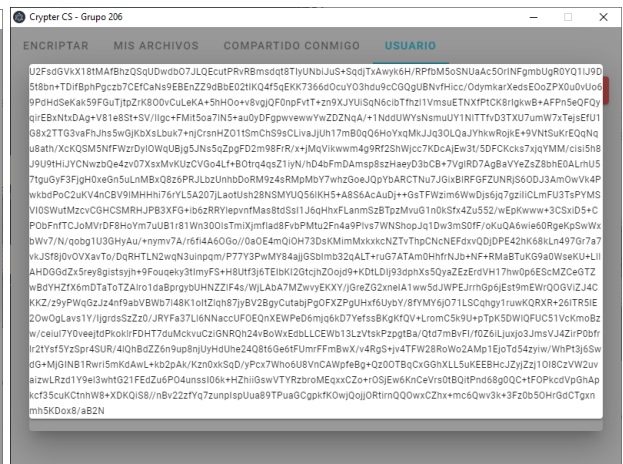
En esta ventana podemos realizar las acciones destinadas a los usuarios como:

- a) Cerrar sesión
- b) Ver nuestra clave pública
- c) Ver nuestra clave privada

a:

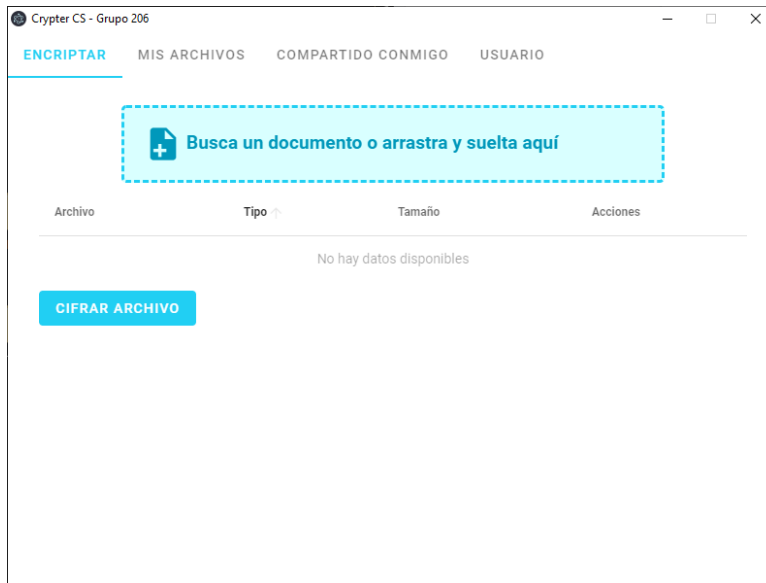


b:

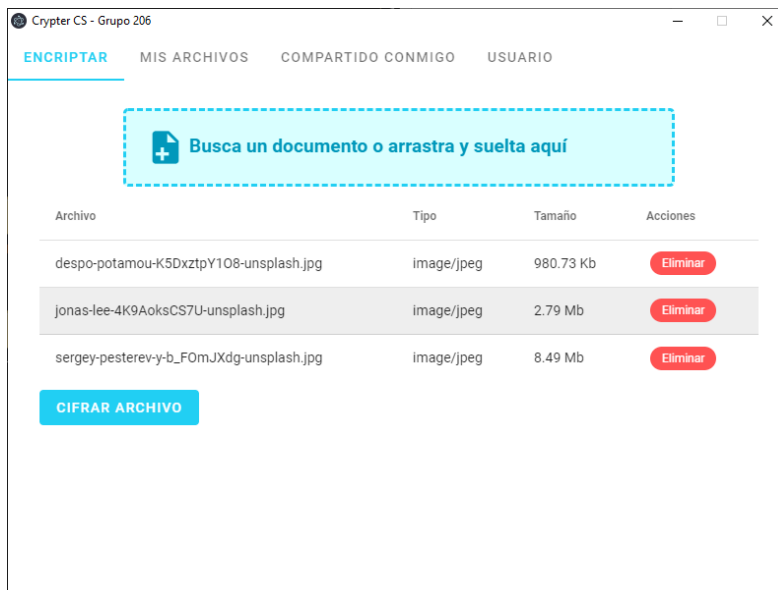


3. Encriptar un archivo

3.1. Selección de los archivos

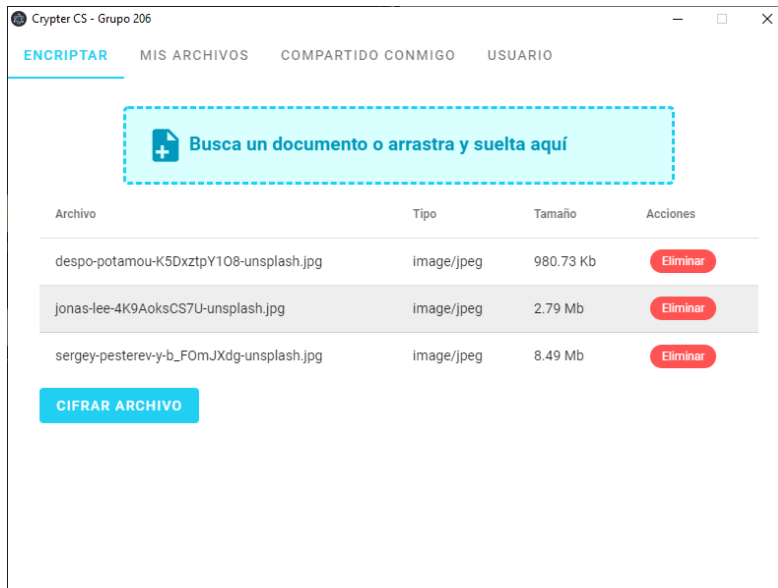


Para la selección de los archivos podemos hacer clic sobre el recuadro azul o arrastrar los archivos que deseemos sobre él, hasta un máximo de 6 simultáneamente.

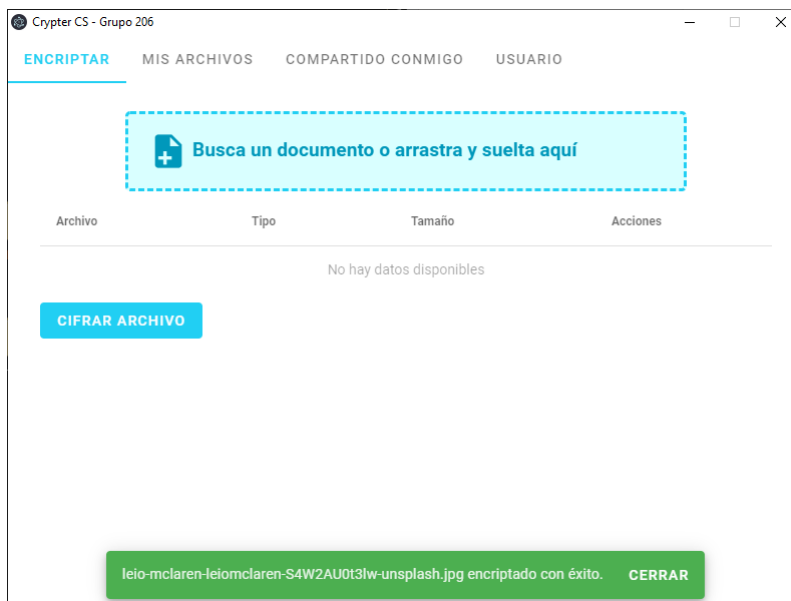


Una vez seleccionados los archivos que queremos encriptar pasamos al siguiente paso. También tenemos la posibilidad de seguir añadiendo archivos o por el contrario quitar alguno que hemos seleccionado por error.

3.2. Encriptar

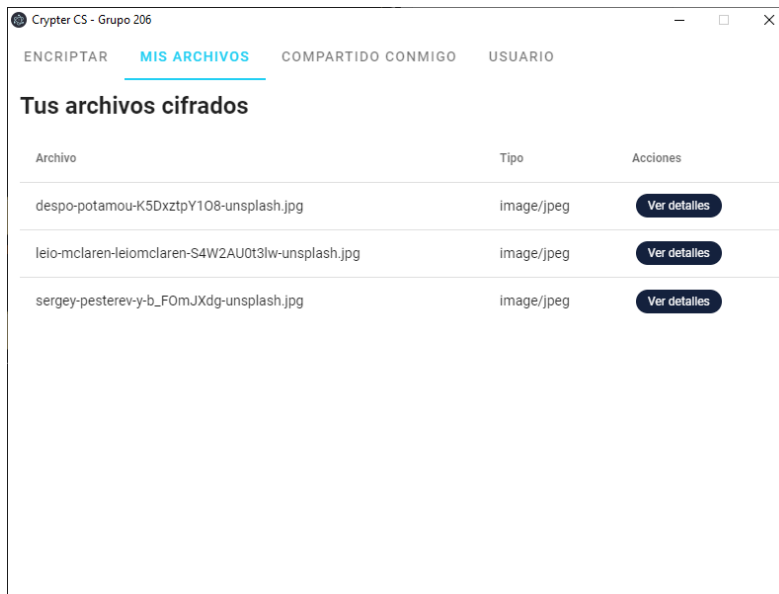


Una vez tenemos todos los archivos que queremos cifrar, hacemos clic sobre el botón azul.



Cuando el proceso se realice correctamente, aparecerá un mensaje verde que indica que se han encriptado los archivos.

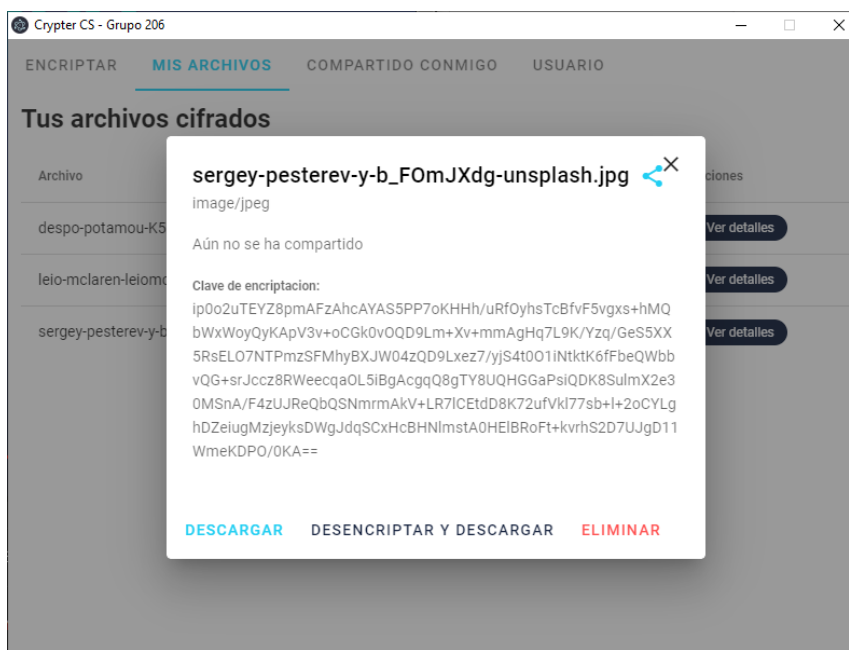
Los archivos cifrados los encontraremos en la ventana de “Mis archivos” de nuestra aplicación como podemos ver en la siguiente imagen.



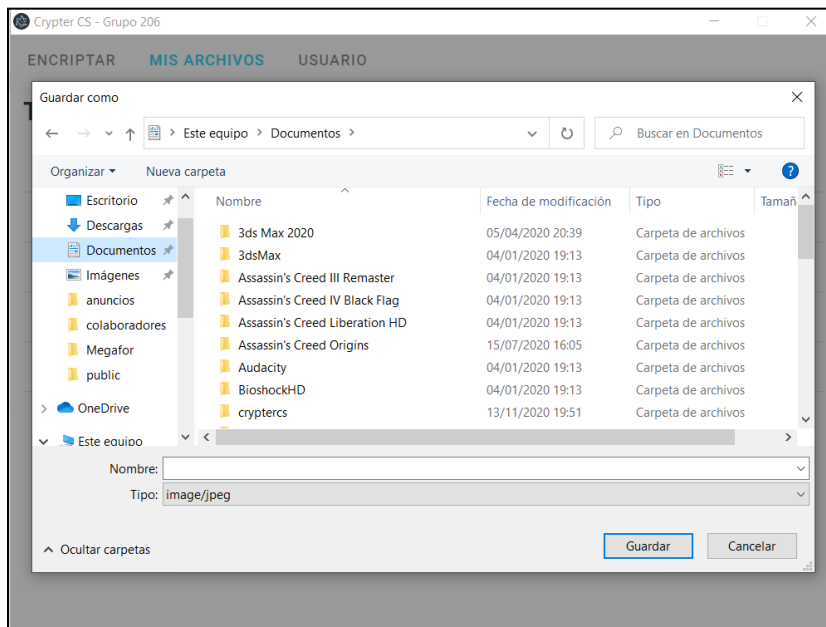
4. Desencriptar un archivo

Una vez entramos en “Mis archivos”, al hacer click en “Ver detalles” en algún archivo, podremos realizar 4 acciones:

- Descargar el archivo encriptado
- Descargar el archivo desencriptado
- Eliminar el archivo encriptado
- Compartir el archivo

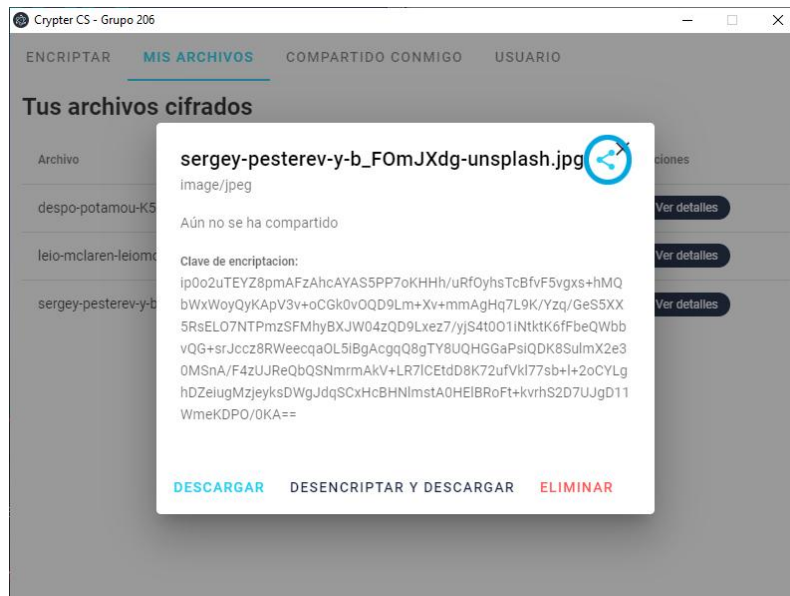


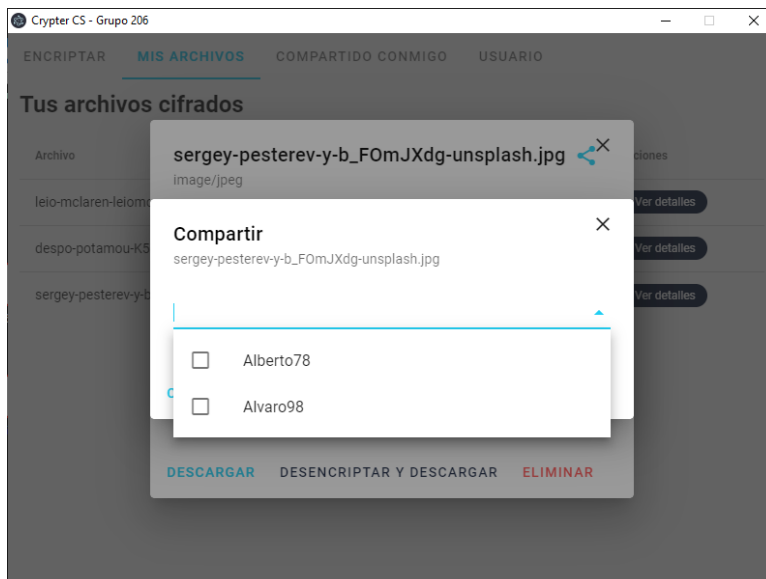
Para desencriptar, hacemos clic en el botón azul claro “Desencriptar y descargar”, y lo guardamos en la ruta que deseemos.



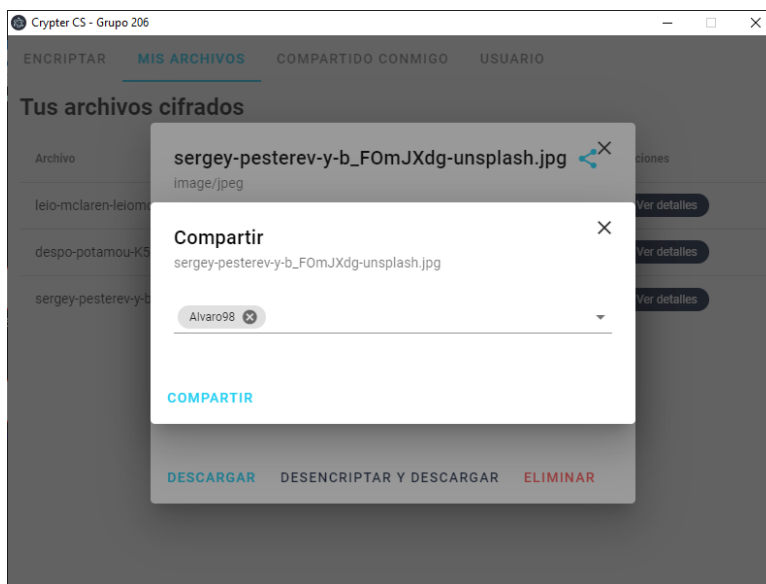
5. Compartir un archivo

Para compartir un archivo, tenemos que acceder a través del botón de **Ver detalles** del archivo. Hacemos click sobre el icono de compartir.

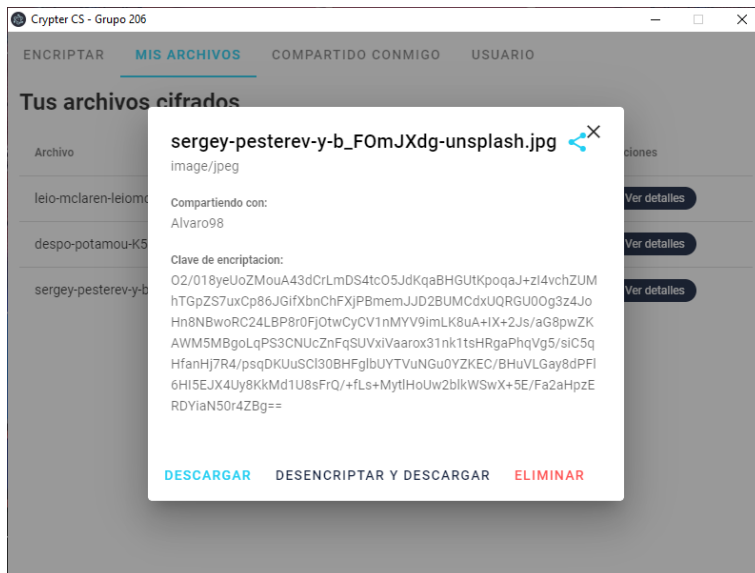




Seleccionamos el usuario o usuarios que deseemos compartir el archivo cifrado.

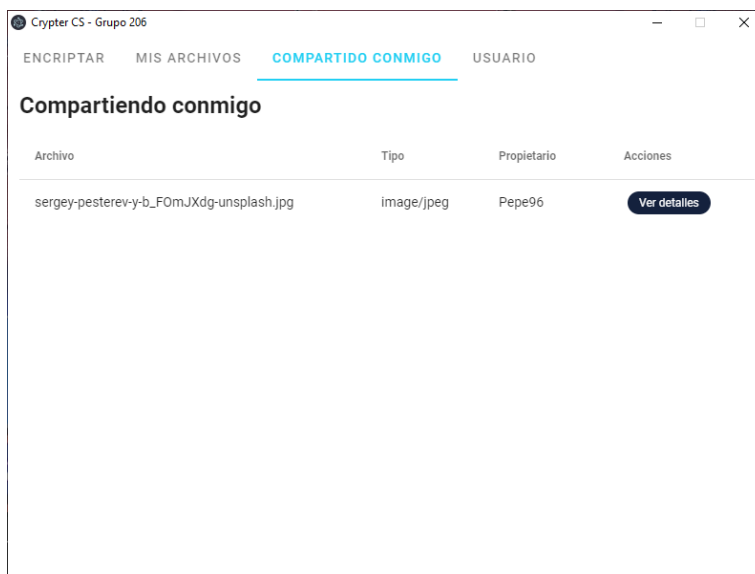


Hacemos click en compartir y el archivo se compartirá con los usuarios seleccionados.



Ahora podemos ver desde detalles del archivo, con quien lo hemos compartido.

El usuario al que se le comparte el archivo, encontrará dicho archivo en la pestaña **Compartido conmigo**, desde allí podrá descargar el archivo encriptado y descriptado.



Protocolo de seguridad

Registro usuario

Para el correcto registro de un usuario **u**, se necesita:

- Un nick, de al menos 4 caracteres y no que esté repetido.
- Una contraseña, de al menos 16 caracteres y máximo 26. p_u
- Un par de claves pública y privada, serán generadas por nuestro programa. K_u^{RSA} / k_u^{RSA}

Pasos a seguir:

- El usuario **u** introduce los datos correctos y solicita el registro.
- Una vez el servidor acepta el registro:
 - Se hace un hash de la contraseña. H_{pu}^{SHA3}
 - Se genera un par de claves pública y privada. K_u^{RSA} / k_u^{RSA}
 - Se encripta la clave privada con la contraseña hasheada. $E_H^{AES}(k_u^{RSA})$
 - Almacenamos todos los datos en la base de datos

De esta forma, guardamos la contraseña de una manera más segura y así ningún administrador de la base de datos puede saber la contraseña de ningún usuario a simple vista, ya que hay usuarios que utilizan las mismas contraseñas en diferentes programas o páginas web.

Subir un archivo

Un usuario **u** desea subir un archivo **f**:

- Se genera una clave AES aleatoria y única para el archivo. k_f
- Se cifra el archivo **f** con la clave generada anteriormente. $E_{kf}^{AES}(f)$
- Se cifra la clave del fichero con la clave pública del usuario. $E_{Ku}^{RSA}(k_f)$
- Se almacenan ambos elementos en la base de datos. $E_{kf}^{AES}(f) \ E_{Ku}^{RSA}(k_f)$

De esta forma, guardamos en la base de datos el archivo **f** cifrado con AES y la clave AES del archivo, se cifra con RSA mediante la clave pública de **u**. Por tanto el servidor no podrá acceder a la información del archivo ni saber la clave AES.

Descargar un archivo encriptado

Un usuario **u** desea descargar un archivo **f**:

- Se obtiene el archivo cifrado. $E_{kf}^{AES}(f)$
- Se obtiene la clave AES del archivo cifrada con la clave pública. $E_{ku}^{RSA}(k_f)$
- Se descifra la clave AES. $D_{ku}^{RSA}[E_{ku}^{RSA}(k_f)] = k_f$
- Una vez descifrada la clave AES, descifra el archivo. $D_{kf}^{AES}[E_{kf}^{AES}(f)] = f$
- Se descargara el archivo en la carpeta que seleccionemos.

De esta forma, desciframos la clave AES con la clave privada del usuario **u** mediante el algoritmo RSA. Una vez descifrada esta clave, podemos descifrar con AES el archivo.

Compartir un archivo encriptado

Un usuario **u** desea compartir un archivo **f**, que ya está subido en el servidor, con otro usuario de la plataforma **v**:

- Se obtiene de la base de datos la clave AES de **f**. $E_{ku}^{RSA}(k_f)$
- Se descifra la clave AES del archivo **f** con la clave privada de **u** y obtenemos la clave de **f**. $D_{ku}^{RSA}[E_{ku}^{RSA}(k_f)] = k_f$
- Se obtiene la clave pública a través del servidor la clave pública del usuario **v** y se cifra la clave AES de **f** con la clave pública de **v**. $E_{kv}^{RSA}(k_f)$
- Se almacenan ambos elementos en la base de datos. $E_{kf}^{AES}(f)$ $E_{kv}^{RSA}(k_f)$

De esta forma, el usuario **v** podrá descargar el archivo **f** cuando desee, ya que con su clave privada será capaz de conseguir la clave AES del archivo **f**.

Detalles de implementación

Registro usuario

```
create(user: Partial<Usuario>): Usuario {
  try {
    const newUser = user

    if (!user.password) throw new Error("Sin contraseña")

    const password = SHA3.encryptSHA3(user.password) as string
    const keys = rsa.generateKeys(password)

    newUser.privateKey = keys.private
    newUser.publicKey = keys.public
    newUser.password = password
    newUser.admin = false

    this.store.dispatch("usuarios/create", user)

    return newUser as Usuario
  } catch (_e) {
    throw new Error("Error en la creación del usuario")
  }
}
```

La función anterior es la encargada de realizar la tarea de crear usuarios en nuestra base de datos. Creamos una variable y la igualamos al “user” pasado por parámetro, que tendrá el nombre y contraseña que le dió el usuario en el formulario. Si no tiene contraseña saltará un error. Hasheamos la contraseña y creamos el par de claves en RSA.

Añadimos a las variables del nuevo usuario las claves privadas y públicas, la contraseña e indicamos si es administrador o no dicho usuario. Metemos al usuario en la base de datos, que gracias al “store”, se actualizará en tiempo real, sin tener que recargar la vista. Luego lo devolvemos, si en algún momento se ha producido un error lo capturamos y lanzamos.

```
function encryptSHA3(payload: string): string {
  const hash = new SHA3(512)
  hash.update(payload)
  return hash.digest("hex")
}
```

Para poder hacer el hash en la función anterior, utilizaremos esta, usando la librería de sha3.

Manejo de usuarios

```
load(users: Usuario[]) {
  this.store.dispatch("usuarios/load", users)
}

delete(id: string) {
  this.store.dispatch("usuarios/delete", id)
}

login(nick: string, password: string) {
  const user = this.get.nick(nick) as Usuario

  if (!user) return false
  if (password !== user.password) return false

  return user
}
```

Aquí vemos 3 funciones diferentes. La primera nos sirve para cargar un conjunto de usuarios en la base de datos, los cuales pasamos por parámetro.

En la segunda sirve para poder borrar usuarios de la base de datos, utilizando para ello el ID del usuario que quisiéramos borrar.

La última función es la que se utilizará para poder hacer login en la aplicación.

Se pasa por parámetro el nick del usuario y la contraseña con los que se intenta hacer login. Comprobamos que haya un usuario con ese nick, de no ser así, devolveremos false, al igual

que si la contraseña que se pasa no coincide con la del usuario. Si todo coincide devolveremos el user.

```
_getUsers() {  
  return this.store.getters["usuarios/all"] ?? []  
}  
  
public get = {  
  all: () => {  
    return this._getUsers()  
  },  
  id: (id: string) => {  
    return this._getUsers().find((user: Usuario) => user.id === id)  
  },  
  nick: (nick: string) => {  
    return this._getUsers().find((user: Usuario) => user.nick === nick)  
  }  
}
```

Estas dos funciones se utilizan para acceder a los usuarios en la base de datos, la primera es la llamada que nos devolverá todos los usuarios. En la segunda, vemos que existen 3 casos.

Si no se indica nada, devolverá todos los usuarios que existan en la base de datos, usando la primera función que estábamos comentando. Si le pasamos un ID, buscará en la base de datos al usuario que coincida con ese ID. Si le pasamos el nick, hará lo mismo pero filtrando por nicks.

```
public static init(store: Store<Usuario[]>) {  
  return this._instance || (this._instance = new this(store))  
}
```

Esta función la usamos para implementar un patrón Singleton.

Encriptado y desencriptado con AES

```
function encryptStringAES(payload: string, key: string) {  
  return CryptoJS.AES.encrypt(payload, key).toString()  
}  
  
function decryptStringAES(payload: string, key: string) {  
  return CryptoJS.AES.decrypt(payload, key).toString(CryptoJS.enc.Utf8)  
}
```

En la práctica anterior, teníamos módulos para encriptar y desencriptar archivos en AES, los cuales mostraremos después, con estos dos nuevos módulos, podremos encriptar y desencriptar la clave privada del usuario en AES, usando la librería de CryptoJS.

```
// Encrypt  
function encrypt(uploadedFile: UploadedFile, key: string): Promise<boolean> {  
  const file = uploadedFile.file  
  const reader = new FileReader()  
  
  return new Promise((resolve, reject) => {  
    reader.onload = async () => {  
      const result: any = reader.result  
      const wordArray = CryptoJS.lib.WordArray.create(result)  
  
      const encrypted = CryptoJS.AES.encrypt(wordArray, key).toString()  
  
      const encryptedFile = new Blob([encrypted], { type: uploadedFile.type })  
  
      const writtenSuccessfully = await filesystem.writeEncryptedFile(  
        uploadedFile.name + ".enc",  
        encryptedFile  
      )  
  
      resolve(writtenSuccessfully)  
    }  
  
    reader.onerror = () => {  
      console.error(reader.error)  
  
      reject("Error convirtiendo el archivo")  
    }  
  
    reader.readAsArrayBuffer(file)  
  })  
}
```

En esta función, como podemos observar, realizamos la lectura del archivo que queramos encriptar, lo transformamos en un “WordArray”, array utilizado en la librería de CryptoJS, realizamos la encriptación con AES y la “key” o clave que usaremos para encriptar el archivo.

Creamos un objeto de tipo “Blob” (binary large object), objeto que se usa en las bases de datos para almacenar datos de gran tamaño y que cambian de forma dinámica.

Si se ha realizado la encriptación correctamente, se devuelve el objeto, si no, devolvemos el mensaje de error y lo mostramos por pantalla.

```
// Decrypt
function decrypt(file: string, fileData: Archivo): Promise<boolean> {
  return new Promise((resolve, reject) => {
    try {
      const decryptedWordArray = CryptoJS.AES.decrypt(file, fileData.password)

      const decryptedFile = new Blob([
        convertWordArrayToUint8Array(decryptedWordArray)
      ])
      const writtenSuccessfully = fileSystem.writeDecryptedFile(
        fileData.nombre,
        decryptedFile
      )

      resolve(writtenSuccessfully)
    } catch (error) {
      reject("Error descriptando el archivo")
    }
  })
}
```

El siguiente método es el de descriptar. Creamos el array con el archivo que vamos a descriptar y la contraseña necesaria para esto. Hacemos la descriptación de CryptoJS, que nos devuelve un WordArray ya descriptado. Creamos un archivo de tipo “Blob” nuevamente con el array de enteros resultante de la función “convertWordArrayToUint8Array”, función que se comentará más adelante, y reescribimos el archivo con esta información, obteniendo así el archivo descriptado.

Nuevamente si ocurre algún fallo en el proceso, devolveremos y mostraremos por pantalla el error.

Claves de RSA

```
import NodeRSA from "node-rsa"
import AES from "@services/crypter"

function generateKeys(password: string) {
  const key = new NodeRSA({ b: 2048 })
  return {
    public: key.exportKey("pkcs8-public-pem"),
    private: AES.encryptStringAES(key.exportKey("pkcs8-private-pem"), password)
  }
}
```

Para poder realizar todo lo referido a RSA, hemos utilizado la librería de node llamada "node-rsa". La importamos en la primera línea para poder utilizarla.

La función "generateKeys" es la encargada de generar los pares de claves, almacenándolas en la constante "key". Devolvemos la clave pública exportada en el formato que hemos indicado, en este caso "pkcs8-public-pem". Y la privada la encriptamos con AES usando la contraseña del usuario.

```
function _importPublicKey(publicKey: string): NodeRSA {
  const key = new NodeRSA({ b: 2048 })
  key.importKey(publicKey, "pkcs8-public-pem")
  return key
}

function _importPrivateKey(privateKey: string): NodeRSA {
  const key = new NodeRSA({ b: 2048 })
  key.importKey(privateKey, "pkcs8-private-pem")
  return key
}
```

Estas dos funciones las utilizamos para poder importar las claves públicas y privadas.

Les pasamos por parámetro la clave que queremos importar, creamos una clave de RSA que nos servirá para importar en ella la clave pública o privada con la que queramos trabajar. Usamos la función de importar que nos facilita la librería de "node-rsa", le damos formato y devolvemos la clave.

Encriptado y desencriptado con RSA

```
function decryptRSA(  
  payload: string,  
  encryptedPrivateKey: string,  
  password: string  
) {  
  const privateKey = AES.decryptStringAES(encryptedPrivateKey, password)  
  const key = _importPrivateKey(privateKey)  
  
  return key.decrypt(payload, "utf8")  
}
```

En esta función, pasamos por parámetro la clave del fichero, la clave privada encriptada y la contraseña. Primero desencriptamos la clave privada usando la contraseña para ello, la importamos en la constante “key” una vez descifrada. Y para finalizar, desciframos la clave del fichero con RSA usando la clave privada de la constante “key”.

```
function encryptRSA(payload: string, publicKey: string) {  
  const key = _importPublicKey(publicKey)  
  return key.encrypt(payload, "base64")  
}
```

La siguiente función es la que se encarga de encriptar con RSA la clave del archivo que queramos cifrar. Le pasamos por parámetro la clave del archivo y la clave pública. Importamos la clave pública en la constante “key” y luego encriptamos la clave del archivo, llamada “payload” con ella. Al final la devolvemos ya encriptada.

Sistema de archivos

```
async shareFile() {
  this.loading = true
  this.archivo.sharedWith = this.selectedNicknames
  const success = await this.createCopiesOfSharedFiles(
    this.selectedNicknames
  )

  if (success) {
    this.snackbarMessage = `${this.archivo.nombre} compartido con éxito.`
    this.snackbar = true
  }
  this.loading = false
},
```

Esta función es la que utilizamos para poder compartir archivos entre usuarios. Lo cargamos y actualizamos su variable de “sharedWith” con los valores del usuario al que le vamos a compartir el archivo. Luego llamamos a la función “createCopiesOfSharedFiles”, que explicaremos después, pasándole por parámetro el nick del usuario. Si todo sale bien y podemos compartir el archivo, mandaremos un mensaje de éxito, después pondremos su variable de loading a false.

```
async createCopiesOfSharedFiles(users: string[]) {
  try {
    for (let i = 0; i < users.length; i++) {
      const user = users[i]

      const isDecrypted = await this.decrypt()

      if (!isDecrypted) throw Error("Can not decrypt")

      await this.encrypt(user)
    }
    return true
  } catch (error) {
    console.error(error)
    return false
  }
}
```

Esta es una función que utilizamos a la hora de compartir archivos entre usuarios. Creamos un bucle for que irá recorriendo el array de usuarios que le pasamos por parámetro. Desencriptamos el archivo que se vaya a compartir, para luego encriptarlo para cada usuario, poniéndolo así en sus archivos encriptados usando su clave. Si surge cualquier error, lo capturamos y mostramos. Devolvemos true si todo ha ido bien, false si ha habido algún error.

```

getFilesInFolderAndDatabase() {
  const allFiles = this.$store.getters["archivos/all"]
  this.files = allFiles
    ? allFiles.filter((file: Archivo) =>
      |   file.sharedWith.includes(this.currentUser)
      | )
      : []
  return this.files
},

```

Con esta función lo que conseguimos es listar todos los archivos compartidos con el usuario. Para esto cargamos todos los archivos en la constante "allFiles", y ahí filtramos todos los archivos que hayan sido compartidos con el usuario actual.

Posibles mejoras

Las posibles mejoras en cuanto a seguridad serían:

- Sistema de doble factor de autenticación: Implementar un sistema de envío de códigos a través de SMS o correo electrónico.
- Sistema de guardado de la password con hash: utilizar los primeros 128 bits para almacenarlos en la base de datos y los restantes 128 para la clave AES del archivo.
- Añadir sal al hash: Reducir la contraseña mínima de 16 caracteres a 12 y rellenar con 4 caracteres aleatorios. De esta manera, se reduciría la vulnerabilidad que tienen muchas de las contraseñas que utilizan los usuarios.

Bibliografía

<p><i>Documentación de CryptoJS</i> https://cryptojs.gitbook.io/docs/</p> <p><i>Git Hub de la librería de CryptoJS</i> https://github.com/brix/crypto-js</p> <p><i>AES 128 con CryptoJS</i> https://stackoverflow.com/questions/28583357/how-to-decrypt-aes-128-in-cryptojs-nodejs-web-browser</p> <p><i>Encriptar imágenes con CryptoJS Ejemplo</i> http://plea.se/me/cryptimage3.html</p> <p><i>Encrypt images in JavaScript</i> https://alicebobandmallory.com/articles/2010/10/14/encrypt-images-in-javascript</p> <p><i>Encriptar y desencriptar en Angular</i> https://blog.maestriajs.com/blog/angular/Encriptar-Desencriptar-Angular/</p> <p><i>Creating a File Encryption App with JavaScript</i> https://tutorialzine.com/2013/11/javascript-file-encrypter</p> <p><i>Writing Your First Electron App Electron</i> https://www.electronjs.org/docs/tutorial/first-app</p> <p><i>SQLite3</i> https://www.npmjs.com/package/sqlite3</p> <p><i>Git Hub RSA</i> https://github.com/rzcoder/node-rsa</p>	<p><i>Instalar CryptoJS mediante npm</i> https://www.npmjs.com/package/crypto-js</p> <p><i>Cómo usar CryptoJS en javascript</i> https://stackoverflow.com/questions/51005488/how-to-use-cryptojs-in-javascript</p> <p><i>Desencriptar imágenes con CryptoJS</i> https://stackoverflow.com/questions/3930762/decrypting-images-using-javascript-within-browser</p> <p><i>Encrypt and upload an image file to IPFS</i> https://medium.com/chaintope-blogchain/encrypt-and-upload-an-image-file-to-ipfs-cac49fa3eb86</p> <p><i>Documentación de Electron</i> https://www.electronjs.org/docs</p> <p><i>Aplicación para encriptar ficheros en Javascript</i> https://programacion.net/articulo/aplicacion_para_encriptar_ficheros_en_javascript_1450</p> <p><i>HashTag Apps Electron</i> https://www.electronjs.org/apps/hashtag</p> <p><i>Instalar Nodejs y npm en Windows 10</i> https://tutobasico.com/instalar-nodejs-y-npm/</p> <p><i>Local Data storage for Electron</i> https://dev.to/ctxhou/local-data-storage-for-electron-2h4p</p> <p><i>Git Hub SHA3</i> https://github.com/phusion/node-sha3</p>
--	---