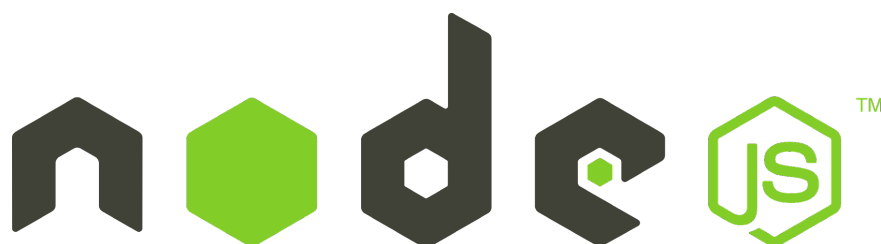


# DESARROLLO DE SERVICIOS WEB CON NODEJS



Email: *[julietalanciotti@gmail.com](mailto:julietalanciotti@gmail.com)*

# Módulo 6 - Archivos y Seguridad

## Manejo de Archivos

### Multer

#### Instalación

### Cargar y Descargar Archivos

#### Paso 1 - Nuevo modelo en la base de datos

#### Paso 2 - Configuración archivo rutas

#### Paso 3 - Rutas de Endpoints

#### Paso 4 - Funcionalidad de los endpoints

#### Subir Archivo

## JWT - Json Web Token

### Estructura de un JWT

#### Header

#### Payload

#### Propiedades

#### Signature

## Implementación de JWT en nuestro proyecto

### Paso 1 - Instalar dependencias

### Paso 2 - Campo password

### Paso 3 - Separar routers principales

### Paso 4 - rutas\_auth()

### Paso 5 - Auth Controller

#### Función LOGIN

#### Función REGISTRARSE

### Paso 6 - SignJWT

### Paso 7 - Rutas de autenticación

### Paso 8 - Autenticar Token

#### Archivo decodeJWT

### Paso 9 - Crear usuario

# Manejo de Archivos

Veamos cómo podemos subir y descargar archivos con nuestra API. Esto nos va a servir por ejemplo si tenemos nuestro propio servidor y nuestro sistema requiere que el usuario suba algún tipo de documentación o imagen.

 Podes descargar el código en el siguiente link:

<https://github.com/JL-Curso-NodeJS/Estructura-Proyecto-4>

## Multer

Multer es un middleware de node.js que permite manejar datos de formularios, y se utiliza principalmente para cargar archivos.

 Documentación: <https://www.npmjs.com/package/multer>

## Instalación

`npm i multer` → para verificar su instalación, podemos ver que en el archivo package.json apareció la nueva dependencia.

## Cargar y Descargar Archivos

### Paso 1 - Nuevo modelo en la base de datos

Antes que nada, necesitaremos crear un nuevo modelo. Nos va a servir para guardar la información de los archivos, y en nuestro caso lo asociaremos a la tabla usuario.

Por lo tanto:

- Un usuario tendrá muchos archivos.
- Un archivo solo pertenece a un usuario.

```
'use strict'

module.exports = (sequelize, DataTypes) => {

  let ArchivoUsuario = sequelize.define('archivo_usuario', {
    id: {
      type: DataTypes.BIGINT,
      autoIncrement: true,
      primaryKey: true,
      allowNull: false
    },
    nombre: { //nombre con el que se identifica al archivo por si sube varios
      type: DataTypes.STRING,
      allowNull: true,
    },
    file: { //nombre del archivo
      type: DataTypes.STRING,
      allowNull: true
    },
    original_name: { //nombre original del archivo cuando se sube
      type: DataTypes.STRING,
      allowNull: true
    },
    createdAt: {
      type: DataTypes.DATE,
      field: 'created_at',
      defaultValue: DataTypes.NOW,
      allowNull: false
    },
    updatedAt: {
      type: DataTypes.DATE,
      field: 'updated_at',
      defaultValue: DataTypes.NOW,
      allowNull: false
    },
    deletedAt: {
      type: DataTypes.DATE,
      field: 'deleted_at'
    }
  }, {
    paranoid: true,
    freezeTableName: true,
  })

  ArchivoUsuario.associate = models => {
    ArchivoUsuario.belongsTo(models.usuario) // un archivo pertenece a un usuario
  }

  return ArchivoUsuario
}
```

Como podemos ver en el código del modelo, esta tabla solo tendrá 3 campos importantes: *nombre*, *file* y *original\_name*. Además, haremos una relación con usuario diciendo que el archivo solo pertenece a un usuario. (**belongsTo**)

Entonces, en el modelo de usuario, haremos la relación con **HasMany**.

- El campo nombre lo utilizaremos por si un usuario decide subir varios archivos, y de esta manera podremos saber más fácil a cuál hace referencia en caso que quiera descargarlo.
- El campo original\_name guardará el nombre original del archivo cuando se sube.
- El campo file, guardará un nombre identificador para guardarlo en nuestro servidor.

Notar que estos dos últimos campos, son recomendados en la documentación oficial de la dependencia

## Paso 2 - Configuración archivo rutas

En nuestro caso, como haremos referencia a que los archivos son de un usuario, modificaremos el archivos de rutas del usuario **usuario.routes.js**

Para poder crear nuestros endpoint, deberemos primero configurar MULTER.

Para esto, agregaremos las siguientes líneas de código:

```
var multer = require('multer')

var upload = multer({ // INSTANCIAMOS MULTER Y LO CONFIGURAMOS
  dest: 'uploads/archivos-usuarios/', //RUTA DONDE SE VAN A SUBIR LOS ARCHIVOS
  limits: { fileSize: globalConstants.MAX_FILE_SIZE } // PESO MAXIMO DEL ARCHIVO 20MB
})
```

1. Importar multer en nuestro archivo
2. Instanciar objeto con la siguiente configuración
  1. *dest*: será donde se guarden nuestros archivos. En nuestro caso haremos un directorio **uploads/archivos-usuarios/**
  2. *limits*: será donde se especifique el tamaño máximo que vamos a aceptar por archivo. En nuestro caso creamos una constante para decir que vamos a aceptar 20MB como máximo.

## Paso 3 - Rutas de Endpoints

Una vez configurado el multer, deberemos crear las rutas a nuestros endpoints.

```
router.post('/subirArchivo', upload.single('jpg'), usuarioController.subirArchivo)

router.post('/descargarArchivo/', usuarioController.descargarArchivo)
```

⚠ Recordar que no es ideal que las rutas indiquen el funcionamiento del endpoint, pero lo haremos ahora para que sea más sencillo entender el funcionamiento.

Notar que estamos haciendo uso del *middleware* que nos brinda multer para permitir que nuestro endpoint acepte un archivo, en este caso un archivo **.jpg**

## Paso 4 - Funcionalidad de los endpoints

### Subir Archivo

```
subirArchivo: async (req, res, next) => {
  try {

    //verifico si existe el usuario
    const usuario = await models.usuario.findOne({
      where: {
        id: req.body.usuarioId
      }
    })
    if (!usuario) return next(errors.UsuarioInexistente)

    // busco el archivo del usuario
    const ar = await models.archivo_usuario.findOne({
      where: {
        usuarioId: req.body.usuarioId,
        nombre: req.body.nombre
      }
    })
    if (!ar) { // si el archivo no existe, lo creo

      const archivo = await models.archivo_usuario.create({
        nombre: req.body.nombre,
        file: req.file ? req.file.filename : null, /
        original_name: req.file ? req.file.originalname : null,
        usuarioId: req.body.usuarioId
      })
    }

    res.json({
      success: true,
      data: {
        message: "archivo cargado"
      }
    })

  } catch (err) {
    return next(err)
  }
},
```



Veamos línea por línea qué hace el siguiente código:

1. El siguiente código verifica si el usuario que viene por el body existe en nuestra base de datos. Si no existe, entonces informaremos un error diciendo “El usuario no existe”. (**ver manejo de errores módulo 5**)

```
const usuario = await models.usuario.findOne({
  where: {
    id: req.body.usuarioId
  }
})
if (!usuario) return next(errors.UsuarioInexistente)
```

2. El siguiente código se encarga de buscar si existe el archivo en la base de datos. Es decir, busca dentro del modelo **archivo\_usuario** alguna fila que coincida con el **usuarioId** y el **nombre** del archivo. Si existe, no haremos nada en este caso.

```
// busco el archivo del usuario
const ar = await models.archivo_usuario.findOne({
  where: {
    usuarioId: req.body.usuarioId,
    nombre: req.body.nombre
  }
})
```

3. En caso de que el archivo no exista, es decir, no está subido, deberemos hacer la carga. Para esto, crearemos una nueva fila en nuestro modelo **archivo\_usuario**. Guardaremos el nombre, el archivo en el campo file (si no

viene este campo se pone null), el nombre original, y el usuario para poder relacionar las tablas.

```
const archivo = await models.archivo_usuario.create({
  nombre: req.body.nombre,
  file: req.file ? req.file.filename : null,
  original_name: req.file ? req.file.originalname : null,
  usuarioId: req.body.usuarioId
})
```

4. Por último, devolvemos una respuesta informando que el archivo se subió correctamente. Si se quiere verificar, después de ejecutar el endpoint, nos debería aparecer un archivo en la carpeta que se nos creó **uploads/archivos-usuarios**

```
res.json({
  success: true,
  data: {
    message: "archivo cargado"
  }
})
```

## Descargar Archivo

```
descargarArchivo: async (req, res, next) => {
  try {

    // verifico si existe el usuario
    const usuario = await models.usuario.findOne({
      where: {
        id: req.body.usuarioId
      }
    })
    if (!usuario) return next(errors.UsuarioInexistente)

    // verifico si existe el archivo
    const archivo = await models.archivo_usuario.findOne({
      where: {
        usuarioId: req.body.usuarioId,
        nombre: req.body.nombre
      }
    })
    if (!archivo) return next(errors.ArchivoInexistente)

    //descarga el archivo
    res.download('uploads/archivos-usuarios/' + archivo.file, archivo.original_name)

  } catch (err) {
    return next(err)
  }
}
```

*Veamos línea por línea qué hace el siguiente código:*

1. Verificamos si existe el usuario. Si no existe, informaremos un error.
2. Verificamos si existe el archivo, pero en este caso, si no existe, deberemos informar un error.
3. En caso de que exista el archivo, le diremos a la respuesta que descargue el archivo que se está buscando, ubicado en la carpeta uploads.



**Si querés probar el código, te dejo el link al repositorio de GitHub:**  
<https://github.com/JL-Curso-NodeJS/Estructura-Proyecto-4>

# JWT - Json Web Token

JWT es un estándar basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios, y así garantizar que sean válidos y seguros.

El caso más común, y el que utilizaremos nosotros, será para manejar la autenticación en aplicaciones web. Para esto, cuando un usuario se quiere autenticar, manda sus datos de inicio de sesión al servidor. Este genera el JWT y se lo manda a la aplicación cliente. Luego, en cada petición, el cliente envía este token, para que el servidor verifique que el usuario está correctamente autenticado y saber quién es.

También se puede usar para transferir cualquier tipo de datos entre servicios de nuestra aplicación y asegurarnos que sean siempre válidos.

1. Autentica utilizando credenciales regulares, como pueden serlo un usuario y contraseña.
2. Se genera una cadena de caracteres con el JWT.
3. Se envía ese token al cliente.
4. El token se almacena del lado del cliente para su posterior uso.
5. El token se manda al servidor en cada petición que realice el cliente.
6. El servidor valida el token y otorga el acceso al recurso.

## Estructura de un JWT

Un JWT consta de tres partes fundamentales: **header.payload.signature**

*eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJlbWFpbCI6Im1bGllbGFsYW5jaW90dGIAZ21haWwuY29tLn0.w0CG2imWYMIH6zQUKn0mfkfXTdmCKCLXMLrzRRFP6g4*

## Header

El header es un string en base64, creado a partir de dos JSON.

Tiene la siguiente forma:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

1. **alg**: Indica el algoritmo utilizado para la firma.

2. **typ**: define el tipo de token, en este caso JWT.

## Payload

El payload también es un string en base64 que está conformado por dos JSON.

Puede tener cualquier propiedad, y será donde se envíe la información importante.

```
{  
  "id": "1",  
  "email": "julietalanciotti@gmail.com"  
}
```

## Propiedades

1. Creador (**iss**) → Identifica a quien creo el **JWT**.
2. Razón (**sub**) → Identifica la razón del **JWT**, se puede usar para limitar su uso a ciertos casos.
3. Audiencia (**aud**) → Identifica a quien se supone que va a recibir el **JWT**. Un ejemplo puede ser web, android o ios. Quien use un **JWT** con este campo debe además de usar el **JWT** enviar el valor definido en esta propiedad de alguna otra forma.
4. Tiempo de expiración (**exp**) → Una fecha que sirva para verificar si el **JWT** esta vencido y obligar al usuario a volver a autenticarse.
5. No antes (**nbf**) → Indica desde qué momento se va a empezar a aceptar un **JWT**.
6. Creado (**iat**) → Indica cuando fue creado el **JWT**.
7. ID (**jti**) → Un identificador único para cada **JWT**.

## Signature

Esta tercera parte toma las anteriores dos y las encripta utilizando un algoritmo, por ejemplo SHA-256.

Se genera utilizando el header y el payload y una key secreta (que solo conoce el servidor).

 Si querés poner en práctica un poco lo que es JWT, podés hacerlo en este link: <https://jwt.io/>

# Implementación de JWT en nuestro proyecto

 Podes descargar el código en el siguiente link:

<https://github.com/JL-Curso-NodeJS/Proyecto-JWT>

## Paso 1 - Instalar dependencias

`npm i bcryptjs` → es una dependencia que nos permite encriptar contraseñas.

`npm i jsonwebtoken` → para utilizar JWT y poder autenticar al usuario.

 Documentación bcrypt: <https://openbase.com/js/bcryptjs/documentation>

Documentación jwt: <https://jwt.io/introduction>

## Paso 2 - Campo password

Ya que vamos a utilizar el modelo de usuario para generar la autenticación, deberemos realizar una migración y agregar el campo password al modelo. De esta manera, haremos un **LOGIN con email y password**.

```
'use strict'
const bcrypt = require('bcryptjs') // para encriptar la contraseña

module.exports = {
  up: (queryInterface, Sequelize) => {
    return Promise.all([
      queryInterface.addColumn('usuario', 'password', {
        type: Sequelize.STRING,
        allowNull: false,
        defaultValue: bcrypt.hashSync('password', 10)
        // Encripto la contraseña por defecto
      }),
    ])
  },
  down: (queryInterface, Sequelize) => {
  }
}
```

En este caso, vamos a asignarle un valor por defecto a nuestro nuevo campo. Pero como NUNCA debemos guardar la contraseña como texto en nuestra base de datos, vamos a encriptarla utilizando la dependencia anteriormente instalada.

```
defaultValue: bcrypt.hashSync('password', 10)
```

Haremos uso de una función de la dependencia, llamada **hashSync**, que lo que hace es encriptar nuestra contraseña a través de un hash.



Recordar agregar el campo password en el archivo **/database/models/usuario.js**



## Paso 3 - Separar routers principales

Recordemos que en el archivo `index.js` principal, nosotros habíamos hecho una función llamada **`configuracionRouter`** y en ella habíamos determinado cuál sería la ruta principal de nuestra api.

Ahora, plantearemos una ruta y una nueva función, pero para todos aquellos endpoints “externos” a la api. Es decir, los que serían para autenticar. Por ejemplo, un login o un registrarse.

```
app.use('/', routerConfig.rutas_auth())
```

Por lo tanto, deberemos implementar la función **`rutas_auth()`**

## Paso 4 - `rutas_auth()`

En el archivo `index.routes.js` nosotros habíamos hecho una función llamada **`rutas_init()`** con su propio router dentro. Esto es porque no es tan seguro que tantos los endpoints internos, como los externos, se encuentren al mismo nivel e incluso utilicen la misma instancia router para manejar las rutas.

Es por eso, que vamos a escribir nuestras rutas de autenticación en una nueva función:

```
const rutas_auth = () => {  
  const router = Router()  
  
  router.use("/auth", authRoutes)  
  
  return router  
}
```

Esta función nos da el pie para darnos cuenta que deberemos crear un nuevo controlador, ya que el controlador del usuario, es para funcionalidades internas de la api.

## Paso 5 - Auth Controller

Vamos a crear un nuevo controlador llamado **auth.controller.js**. Aquí tendremos dos funciones, una será **login**, y la otra la llamaremos **registrarse**.

- La función *login* se encargará de autenticar email y contraseña.
- La función *registrarse* se encargará de crear un nuevo usuario con todos los datos correspondientes, incluyendo su contraseña. (no olvidar que debemos encriptarla)

## Función LOGIN

```
login: async (req, res, next) => {

  // 1. Verifico que el usuario exista solo comparando con el email
  const user = await models.usuario.findOne({
    where: {
      email: req.body.email
    }
  })

  var contraseniaCoincide = false


  if (user) { // Si existe el usuario
    // 2. Verifico que la contraseña sea correcta
    contraseniaCoincide = bcrypt.compareSync(req.body.password, user.password)
    // Comparo la contraseña ingresada con la contraseña de la base de datos
  }

  if (!user || !contraseniaCoincide) {
    return next(errors.CredencialesInvalidas)
  }

  // 3. Si todo está bien, devuelvo el token
  res.json({
    success: true,
    data: {
      token: signJWT(user), // Creo el token con los datos del usuario
      id: user.id,
      nombre: user.nombre,
      apellido: user.apellido,
      email: user.email,
    }
  })
},
```

*Veamos línea por línea qué hace este código:*

1. Verifico si el usuario existe. Vamos a recibir por el body el email y la contraseña. Por lo que buscaremos a este usuario dentro de la base de datos a partir del email.
2. En caso de existir el usuario, deberemos verificar si las contraseñas coinciden. Para esto, utilizamos la función `compareSync` de nuestra dependencia **bcrypt**. Lo que hará esto será comparar la password que viene por el body, con la que está guardada en la base de datos.
3. En caso de no coincidir o no existir el usuario, mostraremos un mensaje de error.

 **Recordar nunca informar qué tipo de error es, ya que daría un indicio a posibles atacantes de poder descubrir si ponen mal el usuario o la contraseña.**

4. Si todo está bien, entonces el usuario podrá hacer login, y como respuesta mostraremos alguno de sus datos (**menos la contraseña**) y además, le devolveremos el token. En el siguiente paso veremos cómo funciona esto.

## Función REGISTRARSE

```
registrarse: async (req, res, next) => {  
  try {  
  
    // encripto la contraseña  
    req.body.password = bcrypt.hashSync(req.body.password, 10)  
  
    // creo el usuario  
    const user = await models.usuario.create(req.body)  
  
    res.json({  
      success: true,  
      data: {  
        id: user.id  
      }  
    })  
  
  } catch (err) {  
    return next(err)  
  }  
}
```

En este caso, únicamente lo que hacemos es primero encriptar la contraseña utilizando la función **hashSync** y luego creamos el usuario en la base de datos. Para informar que se creó bien, únicamente mostramos el id del nuevo usuario.

## Paso 6 - SignJWT

Como mencionamos anteriormente, cuando por fin el usuario puede hacer login, deberemos devolver el token. Para esto necesitamos haberlo creado antes. De esto se va a encargar este nuevo archivo.

Dentro de la carpeta *Middlewares* crearemos un archivo llamado **signJWT.js**

```
const jwt = require('jsonwebtoken') // para crear el token
const globalConstants = require('../const/globalConstants')

module.exports = function (usuario) { // recibe el usuario por parametro

  if (usuario) {

    // Se crea el token con los datos del usuario
    const token = jwt.sign({
      id: usuario.id
    },
    globalConstants.JWT_SECRET, // clave secreta para encriptar el token
    {
      expiresIn: '3000m' // expira en 3 horas
    }
    )
    return token // devuelvo el token
  } else {
    return null // si no hay usuario, devuelvo null
  }
}
```

Cuando explicamos lo que era JWT, hablamos de la estructura que tiene un token y que contiene una firma.

Para crear el token entonces, utilizaremos la función **sign** de la dependencia **jsonwebtoken** instalada anteriormente.

En nuestro token guardaremos el id de nuestro usuario, utilizaremos una clave secreta para encriptar el token, y le pondremos un tiempo límite de validez.

Con este token creado, la función *login* estará completa, y el usuario podrá acceder a nuestra API.

## Paso 7 - Rutas de autenticación

Una vez hecho nuestro controlador, deberemos crear las rutas de autenticación. Por lo tanto, crearemos un archivo llamado **auth.routes.js**, que contendrá las rutas de *login* y de *registrarse*.

```
// RUTAS PARA AUTENTICAR

const router = require("express").Router()
const authController = require('../controllers/auth.controller')

router.post('/login', authController.login)
router.post('/registrarse', authController.registrarse)

module.exports = router;
```

## Paso 8 - Autenticar Token

Hasta el paso 7, el usuario puede loguearse correctamente en nuestra API. Ahora, si quiere utilizar alguno de los endpoints, deberemos decodificar el token para verificar si es correcto y recibir la información.

Para esto, vamos a crear una función llamada **decodeJWT**. Esto lo haremos en un nuevo archivo en la carpeta *middlewares*.

**¿Por qué dentro de los middlewares?** porque esta función se ejecutará **ANTES** que se ejecute el controlador.

## Archivo decodeJWT

```
const jwt = require('jsonwebtoken')
const errors = require('../const/errors')
const models = require('../database/models/index')
const moment = require('moment')
const globalConstants = require('../const/globalConstants')

module.exports = async function (req, res, next) {

  if (req.header('Authorization') && req.header('Authorization').split(' ').length > 1) {
    try {

      // Verifico el token y lo decodifico con la clave secreta para obtener los datos del usuario
      // que lo creó y los guardo en la variable data
      let dataToken = jwt.verify(req.header('Authorization').split(' ')[1],
        globalConstants.JWT_SECRET)

      if (dataToken.exp <= moment().unix())
        return next(errors.SesionExpirada) // Si el token expiró, devuelvo error

      res.locals.token = dataToken

      const usuario = await models.usuario.findOne({
        where: {
          id: dataToken.id
        }
      })
      if (!usuario) return next(errors.UsuarioNoAutorizado)

      res.locals.usuario = usuario //me puedo guardar el usuario en el locals para usarlo en las
      rutas que necesiten el usuario

      next() // Si todo está bien, paso al siguiente middleware o controlador

    } catch (err) {
      return next(errors.SesionExpirada)
    }
  } else {
    return next(errors.UsuarioNoAutorizado)
  }
}
```

Para entender este código tenemos que recordar que el token tiene 3 partes:  
*header.payload.signature*.



1. Comenzaremos preguntando si viene el header `"Authorization"`. Esto va a significar que nos están enviando un token.
2. En una variable llamada `dataToken`, nos guardaremos la información del payload. Recordemos que esta parte del token, era la que contenía en nuestro caso, el id del usuario y el tiempo de expiración.
3. Con esa variable, verificaremos si todavía es valido en cuánto a tiempo. Utilizaremos la función `moment` que nos permite acceder al momento actual. En caso de haber expirado, procederemos a informar un error `"Sesión Expirada"`. → para solucionar esto, el usuario deberá nuevamente hacer login.
4. `res.locals.token = dataToken` → esta instrucción lo que hace es guardarse la información del token de manera local, para poder utilizar en cualquier lugar donde necesitemos devolver por ejemplo el token.
5. Buscamos en la base de datos el usuario que coincida con el ID. En caso de no existir, devolveremos un error `"Usuario no autorizado"`.
6. `res.locals.usuario = usuario` → de igual manera que con el token, nos guardaremos la información que está logueado actualmente en la aplicación.
7. `next()` → si llegamos a este punto, entonces dejaremos que se ejecute la siguiente instrucción. En nuestro caso será el controlador del endpoint al que le hayan pegado.
8. En caso de que haya algún error o no se mande el token, informaremos `"Sesión Expirada"` o `"Usuario no autorizado"`, respectivamente.

Como todo middleware, lo deberemos llamar antes que se ejecute el controlador, pero en este caso, vamos a llamarlo a un nivel más arriba. Es decir, dentro del archivo `index.routes.js` agregaremos el llamado a la función `decodeJWT`.

```
const rutas_init = () => {  
  const router = Router()  
  
  router.use("/usuarios", decodeJWT, usuarioRoutes)  
  
  return router  
}
```

Entonces, cualquier ruta que se quiera acceder de usuario, va a pedir que si o si tenga un token, es decir, que haga login antes.

## Paso 9 - Crear usuario

Por último y no menos importante, hay que tener en cuenta que nosotros antes teníamos un endpoint de crear usuario. En caso de querer conservarlo, deberemos recordar encriptar la contraseña e incluso, agregarle al esquema de creación de usuario, que se espera recibir también una password.

```
crear: async (req, res, next) => {
  try {
    const user = await models.usuario.create(req.body)

    // encriptar contraseña con bcrypt
    user.password = user.cryptPassword(user.password) // encripto la contraseña
    await user.save() // guardo el usuario

    res.json({
      success: true,
      data: {
        id: user.id
      }
    })
  } catch (err) {
    return next(err)
  }
},
```

```
// ARCHIVO PARA ESCRIBIR SCHEMES DE USUARIOS

const Joi = require('joi')

let crearUsuario = Joi.object({
  nombre: Joi.string().required(),
  apellido: Joi.string().required(),
  email: Joi.string().email().optional(),
  edad: Joi.number().optional(),
  password: Joi.string().required(),
})

module.exports = {
  crearUsuario
}
```