Seguridad en Web

Cordero Hernández,
Marco R.

En sesiones pasadas

- MongoDB
- Mongoose
  - Búsqueda de documentos
  - Update/Delete
  - Relaciones

CONTENIDOS

01	02	03
Trasfondo	Variables de entorno	BCrypt
04	**05**	**06**
JWT - Introducción	JWT - Estructura y uso	JWT + Mongoose

01

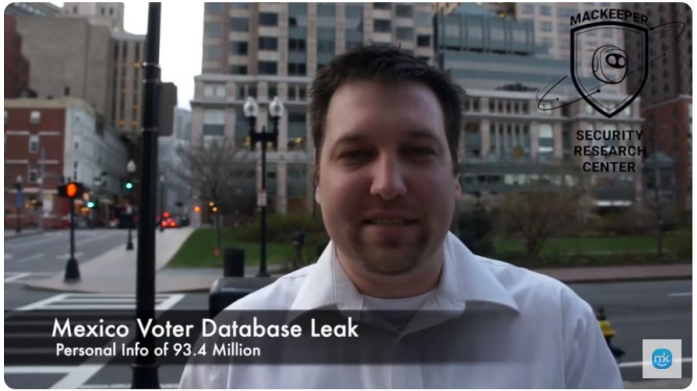
TRASFONDO

Trasfondo

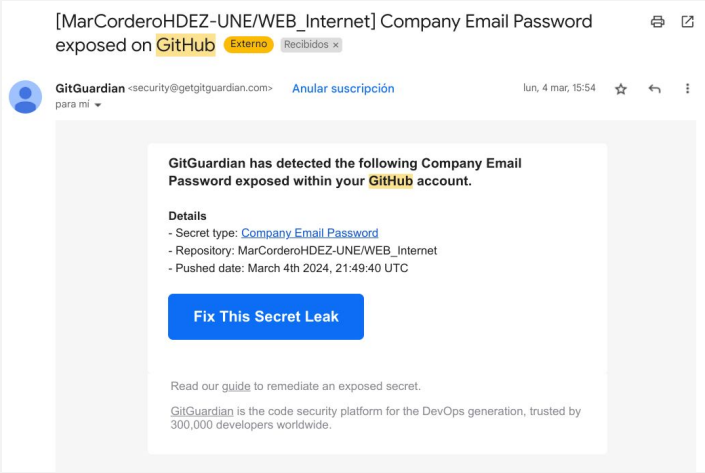
En cualquier desarrollo, el tema de seguridad abarca múltiples vertientes que requieren de equipos enteros para atender incluso solo a algunas cuantas. Es muy sencillo pasar por alto cuestiones aparentemente intrascendentes que luego podrían resultar catastróficas.

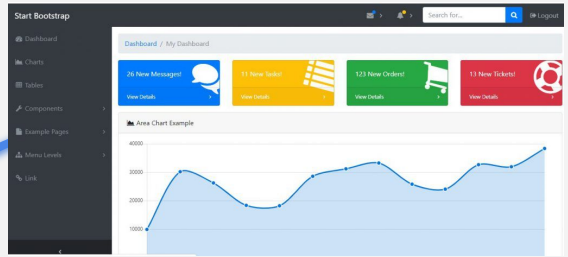
En conjunto del desarrollo móvil, el desarrollo web resulta ser especialmente complejo en cuestión de seguridad, ya que su fácil acceso (navegadores) pone en riesgo tanto al cliente como al servidor e incluso la integridad de una o múltiples empresas.

Trasfondo



Massive Data Breach of Mexican Voter Data



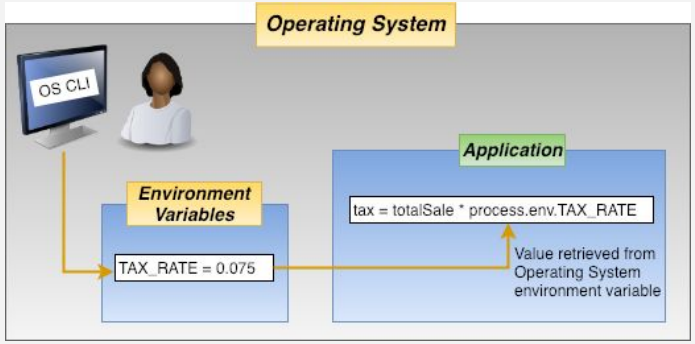


02

VARIABLES DE ENTORNO

Concepto

Las **variables de entorno** definen valores específicos para el *ambiente* sobre el cual se ejecuta una aplicación. Idealmente, se tendrían dos conjuntos de variables: desarrollo y producción.



Las aplicaciones requieren de ciertos parámetros para funcionar correctamente o del modo que se les indique, sin embargo, no siempre se desea exponer este funcionamiento (puerto, cadena de conexión a una API, contraseña de base de datos, etc.)

Creación

Cuando se trabaja de forma colaborativa, usualmente se hace uso de repositorios remotos almacenados en plataformas como *GitHub* o *GitLab*. Al subir el código a alguna de estas, generalmente no se desea exponer las ***variables de entorno*** a ninguna persona, sin embargo, puede que se sí se desee mostrar *cómo* usarlas.

Para atender a esto, es posible crear dos archivos:

- ***.env*** - Contiene los valores reales (variables asignadas)
- ***.env.example*** - Contiene el nombre de las variables sin valor

Ambos son archivos de texto simples

Creación

.env U

Programacion_Internet_WEB > Sesiones > 08 - Seguridad > ses23 > Ejemplos > App > .env

```
PORT=3300
SECRET=1A0b2C9d3E8f4G7h5I6

DB_USER=marcocordero
DB_PWD="MiContraseña"
TARGET_DB="UsuariosDB"
DB_URL="mongodb+srv://<user>:<password>@cluster0.2st7nbj.mongodb.net/<target>?retryWrites=true&w=majority"
# Sustituir <valor> desde JS
```

.env.example U

Programacion_Internet_WEB > Sesiones > 08 - Seguridad > ses23

```
PORT=[Puerto de la aplicación]
SECRET=[Cadena secreta x]

DB_USER=[Usuario de MongoDB]
DB_PWD=[Contraseña de usuario]
TARGET_DB=[Base objetivo en Atlas]
DB_URL=[Cadena de conexión a la base]
```

.gitignore M

Programacion_Internet_WEB > .gitignore

```
# Archivos específicos
*.env
```

Uso desde la aplicación – Método tradicional

Las **variables de entorno** serán usadas desde el *backend*, para ello, es posible instalar el paquete ***dotenv***

- `npm i dotenv`

Dentro de la aplicación, las variables tienen que ser “configuradas” (cargadas) *en cada archivo en donde se quieran usar*.

```
const dotenv = require('dotenv');

// Configurar variables de entorno
dotenv.config();
// Buscará el archivo .env a nivel raíz de la aplicación

// Acceder a variables de entorno
const secret = process.env.SECRET;
console.log(secret);

// Ejemplo de base de datos
let db_url = process.env.DB_URL;
db_url = db_url.replace('<target>', process.env.TARGET_DB);
db_url = db_url.replace('<password>', process.env.DB_PWD);
db_url = db_url.replace('<user>', process.env.DB_USER);
console.log(db_url);
```

Uso desde la aplicación – Nuevos métodos

A partir de **Node 20.6** es posible leer las variables de entorno de manera nativa y apuntando a distinto archivos (no solo .env) con lo siguiente:

- `node –env-file=[archivo (usualmente .env)] archivo.js`

De manera alterna, a partir de **Node 21.7.0** se puede prescindir de lo anterior y directamente dentro del código es posible usar

- ***`process.loadEnvFile(file)`***

Lo cual tendrá el mismo efecto sin tener que instalar paquetes adicionales ni usar parámetros en comandos.

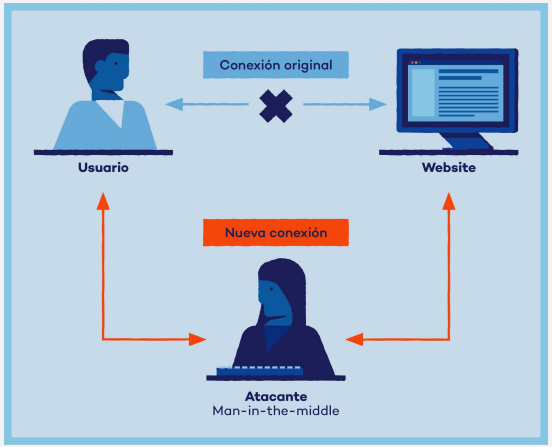
OJO: También se pueden pasar archivos distintos a ***.env*** en estos dos métodos

03

BCrypt

Problema/Necesidad

La lógica de una aplicación usualmente ignora la seguridad en múltiples aspectos operativos. Entre los errores más comunes existe el ***manejo de contraseñas (o cualquier dato sensible) en texto plano***. ¿Qué pasaría si hay un ataque ***MITM***?



https://owasp.org/www-community/vulnerabilities/Password_Plaintext_Storage

Problema/Necesidad

- ¿Qué pasaría si la *base de datos* se ve ***comprometida***?
- ¿Qué pasaría si implemento *mi propio algoritmo de* ***encriptación***?
  - ¿Qué pasaría si no es lo suficientemente ***aleatorio***?
- ¿Qué pasaría si hay una ***filtración*** *de datos*?
- ¿Qué pasaría sí...

Solución (Al menos una posible)

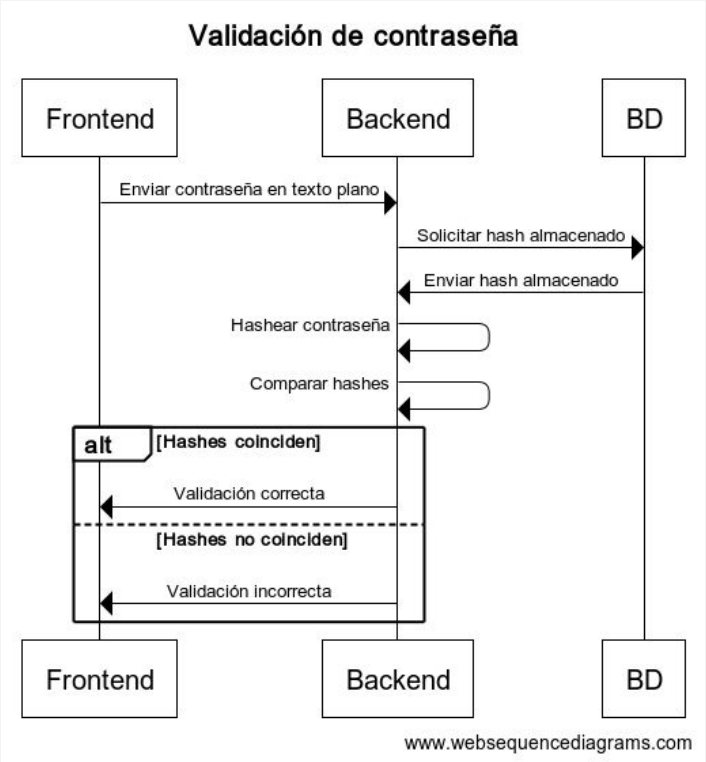
Aún teniendo las mejores medidas de seguridad, un sistema es tan seguro como su eslabón más débil, por lo tanto, si alguien obtiene acceso a nuestros datos, lo más conveniente es que al menos estén *encriptados*. Para esto, es posible usar el paquete **bcrypt**.

```
const bcrypt = require('bcrypt');

// Número de iteraciones
const SALT_ROUNDS = 12; // Usualmente se lee de .env

// Dato a encriptar
let password = 'Mi_Contraseña_En_Texto_Plano';
console.log(`Dato original: ${password}`);

// Encriptar el dato (asíncrono)
bcrypt.hash(password, SALT_ROUNDS, (err, hash) => {
    if (err) {
        console.log('No fue posible encriptar el dato...');
        return;
    }

    console.log(`Dato encriptado de forma asíncrona: ${hash}`);
});

// Encriptar el dato (síncrono)
let hash = bcrypt.hashSync(password, SALT_ROUNDS);
console.log(`Dato encriptado de forma síncrona: ${hash}`);
```

Nota: La encriptación no se limita solo a contraseñas

Validar datos

Cuando un dato se encripta, no puede ser revertido, pero, sí puede ser comparado. Si el texto plano coincide con el hash (usualmente almacenado en una base de datos), se tiene el mismo dato.



Validar datos

```
// Método asíncrono
bcrypt.compare(password, hash, (err, res) => {
    if (err) {
        console.log('No fue posible comparar el dato...');
        return;
    }

    if (res) console.log('Correcto');
    else console.log('Incorrecto');
});

// Ambos son métodos booleanos

// Método síncrono
let match = bcrypt.compareSync(password, hash);
console.log((match ? 'Inc' : 'C') + 'orrecto');
```

04

JWT - INTRODUCCIÓN

Concepto

***J**SON **W**eb **T**oken* es un estándar abierto (RFC 7519) que define de manera compacta *contenido* usando un objeto **JSON** el cual puede ser transmitido sobre la red de manera segura.

La ventaja y principal motivación de este estándar es que la información contenida dentro de los objetos puede ser *verificada* gracias a la presencia de una ***firma digital***.

Aplicaciones

- **Autenticación:** Dentro de un sistema de usuarios y una vez que un usuario se ha logueado, cada solicitud subsecuente debe incluir ***JWT***, lo cual habilita al usuario el acceso a rutas, servicios y recursos que solo son permitidos con su **token** asignado.
- **Intercambio de información:** Permite transmitir información entre distintas partes, debido a que ***JWT*** podría ser firmado usando el sistema de llaves par pública/privada y así poder corroborar el origen de la información. La *firma* se calcula usando el **header** y el **payload** dentro del objeto, de tal manera que se puede comprobar que el contenido no ha sido manipulado.

05

JWT - ESTRUCTURA Y USO

Estructura

- **Header:** Comúnmente consiste del algoritmo de encriptación y el tio del objeto
- **Payload:** Información del objeto (pares llave-valor), también referido como *demandas* (claims). Existen los siguientes tipos:
  - **Registrados:** Demandas predefinidas que permiten la interoperabilidad de contextos en múltiples aplicaciones, ejemplo: iss (issuer/quién lo expide), exp (expiration time/hasta cuándo será válido el objeto), aud (audience/quién debería validar el objeto), ...
  - **Públicos:** Definidos de forma personalizada (se recomienda el uso de demandas ya definidas)
  - **Privados:** Demandas propias de la aplicación

Estructura

- **Firma:** “Une” el header, el payload y una ***clave secreta*** para crear la firma usando el algoritmo definido en el *header*

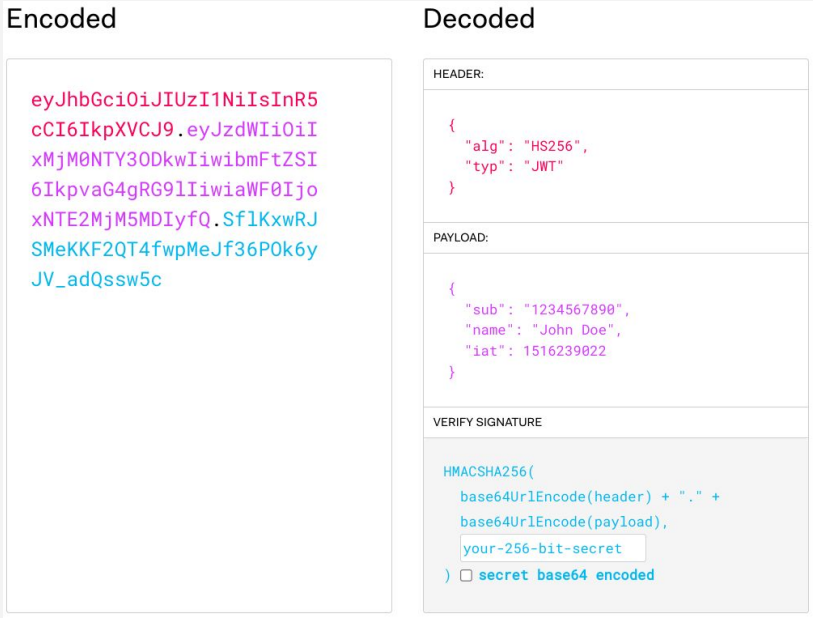
Nota: Aunque no está prohibido, se recomienda ***no poner información secreta en el payload o header*** (como la contraseña). Si la aplicación requiere de un dato secreto, es mejor optar por otras alternativas.

Ejemplo de JWT:

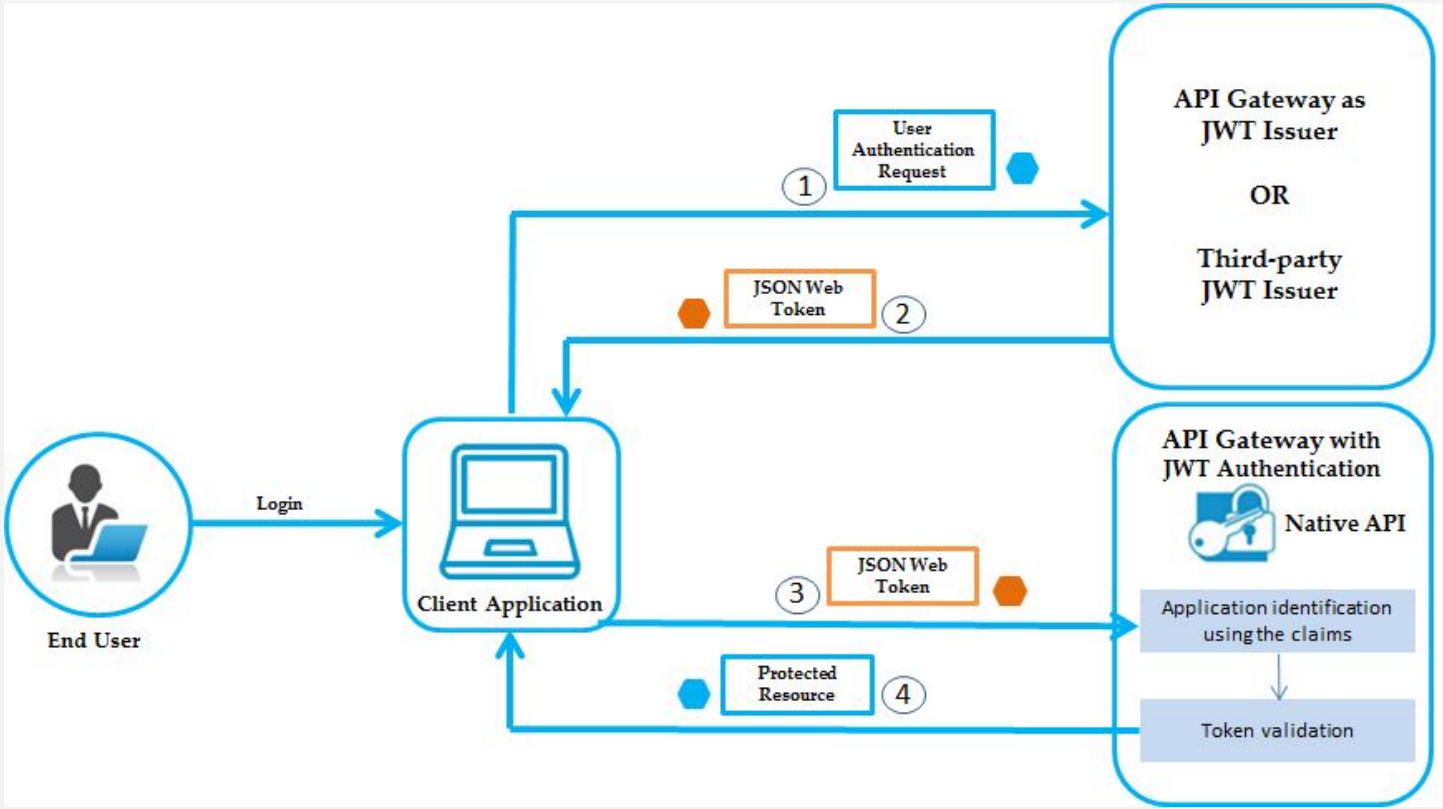
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Ejercicio

Ingresa a jwt.io para que conozcas la manera de crear tokens y cómo se conforma su estructura.



Uso para autenticación (flujo ideal)



Librería *jsonwebtoken*

Para manejar **JWT** dentro de JS (del lado del servidor), se requiere del paquete *jsonwebtoken*

- `npm i jsonwebtoken`

La estructura básica *`jwt.sign(payload, secret, options)`* se utiliza para generar nuevos tokens y existe la versión **síncrona** y **asíncrona**

Librería *jsonwebtoken*

```
const jwt = require('jsonwebtoken');

// Llave privada
const private_key = 'mi_llave_privada';
// Usualmente se lee de .env
// Se recomiendan llaves alfanuméricas
// de al menos 128 caracteres

// Datos
let payload = {
   llave: 'valor'
}

// Síncrono
let token = jwt.sign(payload, private_key);
console.log(token);
// Configuración adicional
token = jwt.sign(payload, private_key, { algorithm: 'HS512' });
console.log(token);

// Asíncrono
jwt.sign(payload, private_key, (err, token) => {
   console.log(token);
});
```

Librería *jsonwebtoken* – métodos adicionales

Dentro de las opciones, se pueden definir lapsos de tiempo para “expirar” el token, es decir, cuando ya no es válido:

```
{ expiresIn: 60*60 }; { expiresIn: '1h' }
```

Para ***verificar*** los datos del token, es posible usar:

- `jwt.verify(jwtToken, secretOrPublicKey[, options, callback])`

Y para ***decodificar*** *sin validar la firma del token* se puede usar:

- `jwt.decode(jwtToken[, options])`
- `let decoded = jwt.decode(jwtToken, { complete: true });`
  `// complete incluirá el header también`

Ejemplo

```
const jwt = require('jsonwebtoken');

let token = jwt.sign({ nombre: 'Marco' }, 'claveSecreta');
jwt.verify(token, 'claveSecreta', (err, decoded) => {
   console.log(decoded.nombre);
});

let decoded = jwt.decode(token);
console.log(decoded.nombre);
```

06

JWT + Mongoose

Creación de esquema

```
let userSchema = mongoose.Schema({
    email: {
        type: String, required: true, trim: true, minlength: 4, unique: true
    },
    password: {
        type: String, required: true, minlegth: 8
    },
    token: {
        type: String, required: true
    },
    acceso: {
        type: String, enum: ['GUEST', 'REGISTERED', 'ADMIN'], required: true
    }
});
```

Validador vinculado al esquema

```
const SECRET_KEY = 'M1_Cl4V3_5eCR3tA';
userSchema.methods.generateToken = () => {
    let user = this;
    let token = jwt.sign({
        _id: user._id.toHexString(),
        acceso: user.acceso
    }, SECRET_KEY,
    { expiresIn: 60*60 }).toString();
    return token;
};
// Puede ser llamado en cualquier momento de la ejecución
```

Validación del token

```
jwt.verify(token, SECRET_KEY, (err, decoded) => {
    if (err) {
        if (err.name == 'TokenExpiredError')
            console.log('El token ya expiró');
        else
            console.log('Error inesperado: ', err);
        return;
    }

    // Manejar acciones subsecuentes
});
// Usado para sesiones de usuario y acceso a recursos
```