



Curso de Back-End con Node.js (Avanzado)

Clase 07



Temario



Temario

- Repaso de Autenticación y Autorización (del curso de Node.js Inicial).
- Autenticación en APIs
- JWT.
- Ejercicio.



Introducción



Introducción

Durante el curso de **Node.js Inicial** se vieron los conceptos de **Autenticación** y **Autorización**, los cuales se repasarán a continuación.

En dicho curso, la Autenticación se vio desde el punto de vista de una *web app* con *server side rendering* mientras que en el presente curso se verá desde el punto de vista de una **API**.

Además, durante el curso de Node.js Inicial se vieron los conceptos de **Cookies** y **Sesiones**, necesarios para brindarle al navegante la “sensación” de estar *logueado* en un sistema. De esta forma, no es necesario solicitar las credenciales de acceso (usuario y contraseña) en cada llamado.

Para lograr todo esto, durante Node.js Inicial se usaron las librerías [Passport.js](#) y [Express Session](#).



Autenticación



Autenticación (1/2)

Es el **proceso** que determina si **alguien** (o algo) **es efectivamente quien dice ser quien es**. En el ámbito web, esto se traduce a un sistema de login de usuarios.

Para implementar un sistema de autenticación en una aplicación se necesita:

- Rutas y vistas que muestran los formularios de **registro** y **login** de usuarios.
- Método encargado de procesar el registro de usuarios.
- Método encargado de validar un email y password.
- Método encargado de “**proteger**” **una ruta**, es decir, inhabilitarla para usuarios que no han hecho login. Por ejemplo, las rutas que empiecen con `/admin` sólo deberían estar disponibles para usuarios logueados.
- Además se debe: **encriptar** (**hashear**) **contraseñas**, gestionar sesiones, crear migraciones para la base de datos...

En fin, no es una tarea sencilla y además se debe repetir para cada proyecto que requiera autenticación. Por suerte hay herramientas que simplifican la tarea.



Autenticación (2/2)

Recordemos que la **seguridad** es uno de los atributos de calidad que solemos buscar en todo sitio web o aplicación, pero según el tipo de aplicación podemos darle más o menos prioridad a la seguridad. Por ejemplo, un banco no requiere el mismo tipo de seguridad que un juego como “El Solitario”.

En caso de ser necesario, se pueden agregar mecanismos adicionales de seguridad como:

- Two-factor Authentication ([link](#)) usando, por ejemplo:
 - Token físico (digital).
 - Token físico (analógico). Ej: “Tarjeta de coordenadas” de Banco Santander.
 - SMS de verificación.
- Cambios periódicos de contraseña (algo muy debatible).
- Control de direcciones IP.

En cualquier caso, siempre usen **HTTPS**.



Autorización



Autorización

Es el proceso que determina **a qué recursos puede acceder** determinado usuario.

Este proceso ocurre luego de que el sistema haya podido autenticar al usuario.


Un típico ejemplo de autorización es definir distintos tipos de **roles** que pueden tener los usuarios de una aplicación como, por ejemplo: lector, editor, administrador, etc. Luego, según el rol del usuario, el sistema determina a qué datos puede acceder y cuáles puede modificar.



¿Cómo guardar contraseñas?



¿Cómo guardar contraseñas? (1/2)

-  Las contraseñas en una base de datos jamás se deben guardar como “texto plano”. ¡Sería un problema de seguridad enorme!
- Las contraseñas deben encriptar o, mejor aún, *hashear*.
- La encriptación tienen un método inverso llamado desenscriptación. Para esto existe una “llave” o “clave” de encriptación y desenscriptación. Si bien guardar una contraseña encriptada es mejor que guardarla como texto plano, el hecho de que se pueda desenscriptar es peligroso.
- Las *funciones de hash* no requieren de una clave y no tienen una función inversa. Una vez que una contraseña es *hasheada*, no se puede volver para atrás.
- Cuando un usuario quiera loguearse a una aplicación, deberá ingresar su contraseña (texto plano). El sistema la *hashear*á y la comparará con el *hash* guardado en la BD.



¿Cómo guardar contraseñas? (2/2)

Existen varias funciones de hash disponibles, algunas son:

- MD5.
- SHA (SHA-1, SHA-256, SHA-512).
- BCrypt.

MD5 es una función de *hash* muy rápida. Es decir, una PC común y corriente puede calcular millones de *hashes* por segundo. Por lo tanto, no es recomendable su uso para contraseñas. En cambio, **BCRYPT** es mucho más complejo y una PC demora mucho más en generar los *hashes*. Además, si la tecnología avanza y las PC se hacen más rápidas, BCrypt se puede configurar (de una forma muy sencilla) para complejizarse mucho más.

👉 La recomendación es usar **BCRYPT** ([link](#)) y hay un [paquete en npm](#) para ello. ⚠ Si ese paquete no les anda (muy probablemente en Windows), prueben con [este otro](#).



Autenticación en APIs



Autenticación en APIs

Así como necesitamos autenticar usuarios para entrar a distintas secciones de un sitio web, también suele ser necesario autenticar usuarios o sistemas que desean acceder a una API.

Hay varias formas de implementar autenticación en APIs. **Algunos** ejemplos son:

- **HTTP Basic Authentication** → Username/Password en los *headers* de cada *request*.
Ej: Twilio.
- **API Key Authentication** → Se usan API Key o Tokens (strings generalmente random y largos) en lugar de un Username/Password.
Ej: Stripe y Sendgrid.
- **OAuth Authentication** → Se usa un `access_token` generado por un sistema tercero. Suele ser la mejor opción para autenticar cuentas de usuario (personales).
Ej: Google, Facebook, Twitter.

👉 En cualquier caso, siempre usen **HTTPS**.

👉 Es probable que necesiten habilitar **CORS** ([link](#)) si es que quieren hacer llamadas AJAX a la API. Para ello pueden usar [este paquete](#) (middleware). Notar que esto en realidad aplica para cualquier API (con o sin autenticación).



JWT



JWT (1/4)

JSON Web Token es un **estándar** abierto ([RFC7519](#)) para crear **tokens de acceso** a una aplicación. Ver [documentación](#). Ver [video](#).

Como dice el nombre, el JWT contiene **datos en formato JSON**. En general están sin encriptar, pero siempre está **firmados** (*signed*).

JWT suele ser útil a la hora de implementar autenticación en una **API** y una comunicación *server-to-server*. Además, podríamos usar el mismo JWT para autenticarnos en dos sistemas dominios diferentes (algo imposible con sesiones y cookies).

En caso de que se quiera realizar una comunicación *browser-to-server*, el token se podría guardar en una cookie, pero no es estrictamente necesario. También se podría guardar en [localStorage](#). En este caso, hay que tener [cuidado](#).



JWT (2/4)

Un JWT se compone por:

- **Header.**
Para especificar el tipo de token y algoritmo.
- **Payload.**
Los datos que queremos guardar en el JWT. Existe [cierto estándar](#) para que el *payload* sea compacto.
- **Signature.**
Para verificar que el JWT es válido (que no haya sido manipulado por nadie más).

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp1hcs0tYSBQw6lyZXoiLCJlbWFPbCI6Im1hcm1hX3B1cmV6QGdtYWlsLmNvbSIsInJvbmGU0iJhZG1pbiIsIm1hdCI6MTUxNjIzOTAyMn0.LIK_0RZK6mcyPN78DTp  
o4T20mJY6n-BMhaIXQ7LDX_M
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

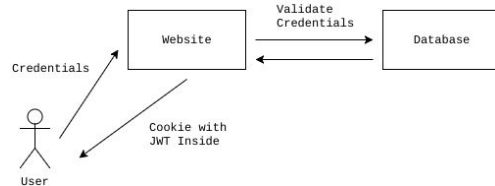
PAYLOAD: DATA

```
{  
  "sub": "1234567890",  
  "name": "María Pérez",  
  "email": "maria.perez@gmail.com",  
  "role": "admin",  
  "iat": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  Un_Pedazo_De_Texto_R  
) ☐ secret base64 encoded
```

JWT (3/4)



Ejemplo de un proceso de autenticación:

1. El cliente ingresa su usuario y contraseña (credenciales).
2. El servidor valida (contra la BD) que las credenciales sean correctas. Si lo son, genera un **JWT** y se lo envía al cliente.
3. De ahora en más, los *requests* realizados por el cliente deberán incluir el **JWT** en los *headers*.
4. Cada vez que el servidor reciba un **JWT**, se deberá verificar que el mismo sea válido. Para esto se deberá verificar que la **firma** sea válida. Esto se puede hacer **sin necesidad de volver a llamar a la BD**. Si la firma es válida, se deja “acceder” al cliente al recurso solicitado.



JWT (4/4)

A partir de ahora, los *requests* realizados por el cliente deberán incluir un *header* de autenticación:

```
Authorization: Bearer <token>
```

Luego, en las rutas “privadas” se deberá **validar que el *token* sea válido**. Para esto se deberá verificar que la **firma** sea válida. Esto se puede hacer **sin necesidad de interactuar con la BD**.

👉 Notar que un problema que pueden tener los **JWT** es la dificultad de hacerlos **expirar** de forma “forzada”, por ejemplo, si a un usuario le roban su *token*. En estos casos sería recomendable llevar un registro en el servidor de los *tokens* generados.



JWT en Node.js



JWT en Node.js (1/3)

La librería más utilizada para gestionar JWTs es `jsonwebtoken` ([link](#)).

```
npm i jsonwebtoken
```

Forma de uso:

```
const jwt = require("jsonwebtoken");  
const token = jwt.sign({ sub: "user123" }, "UnStringMuySecreto");
```

Este token es el que se le envía al cliente.



JWT en Node.js (2/3)

Cuando el servidor recibe un *request* conteniendo un JWT, es necesario validarlo. Para ello se puede utilizar el siguiente código:

```
jwt.verify(token, "UnStringMuySecreto", function (err, decoded) {  
  // Si hubo un error, `err` está definido.  
  // Si está todo OK, `decoded` contiene el payload.  
});
```

Esta validación habría que hacerla en todos los requests a *endpoints* privados.



JWT en Node.js (3/3)

Para agilizar el proceso de validación en Express, se recomienda utilizar el paquete `express-jwt` ([link](#)), que es un *middleware* de autenticación:

```
const { expressjwt: checkJwt } = require("express-jwt");  
  
...  
  
app.get("/ruta-privada", checkJwt({ secret: "UnStringMuySecreto", algorithms: ["HS256"] })), (req, res) => {  
  // ● Si el JWT es válido, el payload del mismo queda disponible aquí adentro vía el objeto `req.auth`.  
  // ● De lo contrario, el middleware responde con un status 401.  
});
```

Si el JWT contiene un atributo (*claim*) llamado `exp`, se validará si el token no está vencido.

PD: Antiguamente no era necesario especificar el *array* de algoritmos. Se hizo obligatorio [recientemente](#).



¿Cómo se le hace llegar
el JWT al cliente?



¿Cómo se le hace llegar el JWT al cliente?

No hay una respuesta única.

De alguna manera, el “dueño” de la API debe hacerle llegar el *token* al cliente que desea acceder a la misma.

Esto puede realizarse a través de un panel de control como es el caso de la API [TheMovieDatabase](#) (ver siguiente diapositiva).

Incluso se podría enviar el *token* por email (aunque no es recomendable).

Otra opción es que la propia API provea un método para obtener un *token*, por ejemplo: **POST** **/tokens** al cual se le deben pasar las credenciales del cliente (ej: *username* y *password*).



Delete Account

eyJhbGciOiJIUzI1NiIsInR5eHQiOiJ3ZGlmcTQyJTktMTU0M2ewq343424zMzQwQWQ5Y2E1Njc3YWRhZCZlZClnNjE677Y329Ti67ljVmwmmODg1YzYwMDIyZWlm544DAAAZnmJmNjsDRfhNyIsinNjb3BlcyI6WyJhcGlfcmlhZCJldCJdLXZKZXJzaW9uIjoxfj0.rolioj-dLDBI3CPDQQNA_9E_w643FmkIJffrUOE55FtEcElu6c

Ejemplo de un flujo de autenticación



Notar que el siguiente diagrama es sólo una referencia.

Por ejemplo, las rutas de la API podrían tener otros nombres y como credencial de acceso se podría usar un *username* en lugar de un *email*.

CLIENTE

1. Cliente se registra como usuario.

POST /users

{fname, lname, email, password, etc...}

{user, token}

2. Cliente solicita un token ≈ "Iniciar sesión".

POST /tokens {email, password}

{user, token}

3. Cliente accede a un recurso privado usando el token obtenido.

GET /private

Header | Authorization: Bearer <token>

{data}

SERVIDOR

Verifica y guarda información en la BD. La contraseña se guarda *hasheada*. Retorna **token**.

Verifica credenciales. Retorna **token**.

Verificar token. Retornar información privada.



Ejercicio

Ejercicio (1/2)

⚠ Si el paquete [bcrypt](#) no les anda, prueben con [bcryptis](#).



- Crear un nuevo proyecto de Node.
- Instalar `express`, `jsonwebtoken`, `express-jwt`, `bcrypt`, `mongoose` y `validator` como dependencias del proyecto. Además, en caso de no tener `nodemon` instalado de forma global, lo pueden instalar como una dependencia de desarrollo en el proyecto.
- Crear tres *endpoints*:
 - a. `[POST] /users`
 - Recibe (en el *body*) un JSON con `firstname`, `lastname`, `email` y `password`.
 - Guarda los datos del usuario en una base de datos MongoDB. Para esto será necesario crear un modelo `User` con Mongoose. Tener en cuenta que el email debe ser único y la contraseña se deberá guardar *hasheada*.
 - Crea un JWT. 🖐 Evaluar los beneficios de guardarlo también en la base de datos. Dentro del token se deberá guardar el `id` del usuario.
 - Retorna un JSON con el usuario creado (sin incluir su contraseña) y su JWT.
 - Si algo falla en el proceso, se deberá responder con los errores pertinentes.

Ejercicio (2/2)



b. [POST] /tokens

- Recibe (en el *body*) un JSON con `email` y `password`.
- Comprueba la existencia del email en la base de datos. En caso de existir, se compara la contraseña en “texto plano” que vino en el *request* con la versión *hasheada* almacenada en la base de datos.
- Si hubo *match*, se crea un *token* JWT para el usuario.
- Retorna un JSON con el `user` encontrado (sin incluir su contraseña) y con el *token* creado.
- Si algo falla en el proceso, se deberá responder con los errores pertinentes.

c. [GET] /private

- Recibe (en los *headers*) un token JWT.
- Validar que el JWT sea válido.
- Si es válido, responder con un JSON que contenga un mensaje “OK” y el usuario propietario del JWT.
- En caso de que el JWT no sea válido o no se encuentre el usuario en la base de datos, responder con un error pertinente.