



**Argentina
programa
4.0**



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

***primero
la gente***

CLASE 14: JSON Web y Token

Seguridad en aplicaciones BackEnd

Agenda de hoy

- A. El concepto de CRUD
- B. Cómo desarrollar un CRUD con MongoDB y Express
 - a. Lectura de datos
 - b. Grabar un nuevo recurso
 - c. Modificar un recurso existente
 - d. Eliminar un recurso
- C. Testear nuestra API Restful



Qué es un token

En el contexto de la tecnología, un token es una especie de "*credencial*" utilizada para identificarnos y poder acceder a ciertos recursos o realizar ciertas acciones. Podemos pensar en él como una llave especial que nos permite abrir una puerta.

Cuando hablamos de tokens en la web, nos referimos a tokens de autenticación. Estos tokens son utilizados para demostrar que somos quien decimos ser y así obtener acceso a servicios o información protegida.



Qué es un token

Seguramente, en algún momento, tuvimos la oportunidad de ver y/o interactuar con los llaveros que solían entregar bancos a sus clientes, para generar un token de seguridad que nos permitiese confirmar determinadas transacciones en la plataforma de banca online.

Estos token eran generalmente numéricos, y tenían un número que cambiaba al azar cada vez que pulsábamos el botón. Dicho número está relacionado con datos personales nuestros que permitían generar un valor numérico unívoco pero no tan libremente expresado.

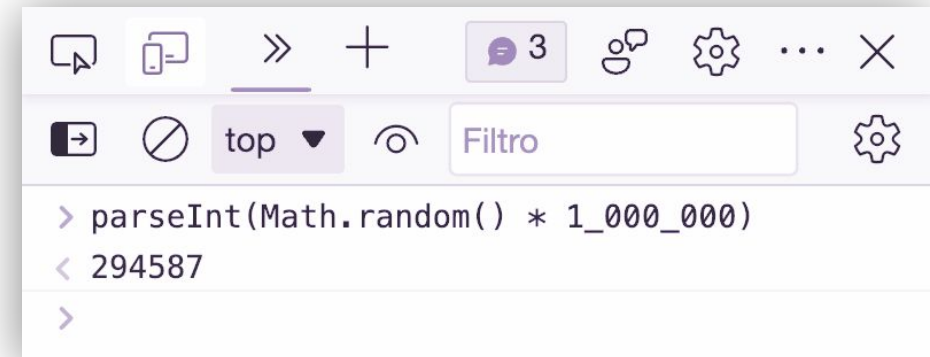


Qué es un token



Hoy, muchos de esos dispositivos electrónicos se reemplazaron por apps móviles que generan esos token numéricos.

A diferencia del token electrónico, las apps utilizan un valor numérico al azar generado por el servidor. El usuario debe ingresarlo en la app de online banking para realizar una operación, y dicho número se constata en el backend para verificar que sea el mismo que fuera generado por este último.



Qué es un token

Por lo tanto, esto representa que, en el mundo digital, un token de autenticación es básicamente un código o una cadena de caracteres único que te identifica de forma segura en un sistema o aplicación.

Al proporcionar este token al sistema, ya sea mediante un nombre de usuario y contraseña, una huella digital o algún otro método de autenticación, podemos demostrar que tenemos permiso para acceder a ciertos recursos o realizar ciertas acciones.



Qué es un token

Una vez que el sistema verifica y valida nuestro token de autenticación, nos permite acceder a áreas restringidas, como ser una cuenta personal, nuestros mensajes privados u otras funciones especiales. El token actúa como una credencial digital que nos identifica y nos permite realizar acciones autorizadas dentro de un sistema.

Es importante destacar que, los tokens, están diseñados para ser únicos y difíciles de falsificar. Además, suelen tener una fecha de expiración para garantizar que no se puedan utilizar de forma indefinida.



Cómo generar un Token

Cómo generar un token

Existen diferentes formas de generar tokens, dependiendo de los lenguajes de programación.

Específicamente en el universo JavaScript, utilizamos dos posibles formas de implementar un mecanismo de este estilo.

- mediante números al azar
- utilizando tokens de identificadores únicos

Números al azar

Números al azar

El objeto **Math** en JS cuenta con un método denominado **random()**. Este nos retorna, cada vez que lo invocamos, un número aleatorio entre **0** y **1** (*el 1 no está incluido*).

Posee una importante cantidad de decimales que varían, lo cual hacen que el número sea casi imposible que se repita.

```
top ▼ Filtro
> Math.random()
< 0.7108248709966316
> Math.random()
< 0.7545980655934001
> Math.random()
< 0.5297614276851403
> Math.random()
< 0.5865992086916769
> Math.random()
< 0.9487836227460738
>
```

Números al azar

```
> Math.random() * 1_000_000  
< 361945.29774438625  
  
> Math.random() * 1_000_000  
< 808235.6902956907
```

Si lo multiplicamos por un número entero de varias cifras, conseguiremos un valor numérico entero más alto (*básicamente corremos la coma de lugar*).

Y si encerramos el resultado en la función **parseInt()**, finalmente obtendremos un número libre de decimales, listo para ser aprovechado donde necesitemos.

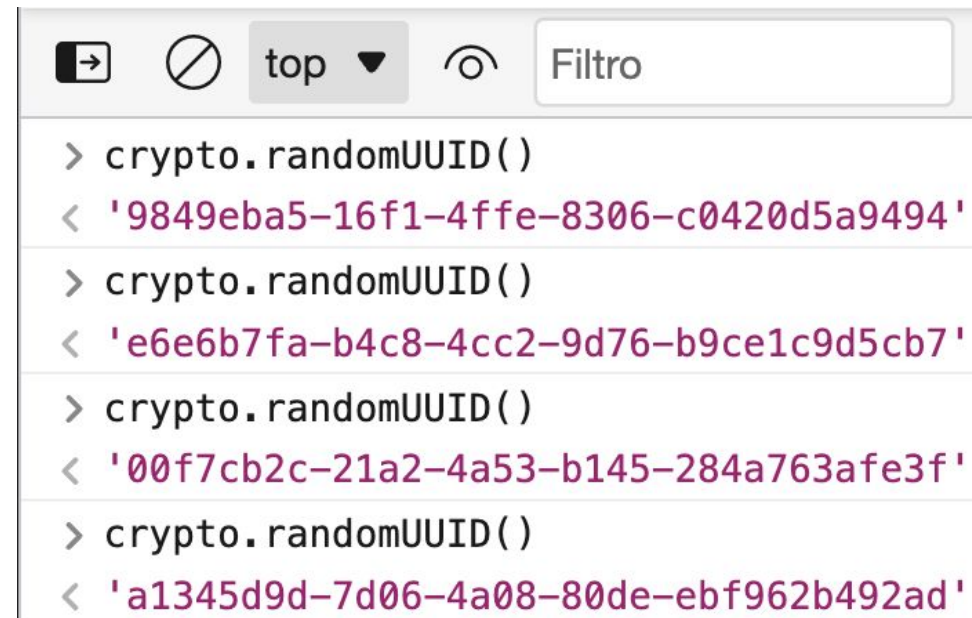
```
parseInt(Math.random() * 1_000_000)  
526683  
  
parseInt(Math.random() * 1_000_000)  
609499
```

Tokens de identificadores únicos

Tokens de identificadores únicos

La biblioteca **crypto**, hoy parte del lenguaje JS, es un objeto que posee un método denominado **randomUUID()**, el cual permite generar un identificador único aleatorio en formato **UUID**.
(*Universally Unique Identifier*)

Este identificador consta de 32 caracteres hexadecimales divididos en cinco grupos y separados por guiones.



```
> crypto.randomUUID()  
< '9849eba5-16f1-4ffe-8306-c0420d5a9494'  
  
> crypto.randomUUID()  
< 'e6e6b7fa-b4c8-4cc2-9d76-b9ce1c9d5cb7'  
  
> crypto.randomUUID()  
< '00f7cb2c-21a2-4a53-b145-284a763afe3f'  
  
> crypto.randomUUID()  
< 'a1345d9d-7d06-4a08-80de-ebf962b492ad'
```

Tokens de identificadores únicos

Este es un mecanismo efectivo a la hora de generar tokens. Si necesitamos eliminar sus guiones, el método de string **replaceAll()** nos permite quitarlos o reemplazarlos. Si queremos acortarlo, podemos incluso reducir su cantidad de caracteres.



```
> crypto.randomUUID()  
< 'b060c329-be8c-4b97-aef2-e2fae7012fee'  
  
> const token = 'b060c329-be8c-4b97-aef2-e2fae7012fee'  
  
> token.replaceAll("-", "")  
< 'b060c329be8c4b97aef2e2fae7012fee'
```


Tokens de identificadores únicos

Existe una posibilidad casi nula de que un token de este tipo se repita.

De acuerdo al total de de letras y números utilizados por este, multiplicado entre sí, y multiplicados por el total de la población mundial, la probabilidad de repetir un mismo **UUID** es de 1 en **3,84e21**.



JSON Web Token

JSON Web Token

JSON Web Token, (JWT), es un estándar abierto de Internet ([RFC 7519](#)) y se utiliza para **transmitir información de manera segura entre dos partes** en forma de un token en formato JSON.

Estos token son utilizados comúnmente para la autenticación y autorización en aplicaciones web y servicios API.



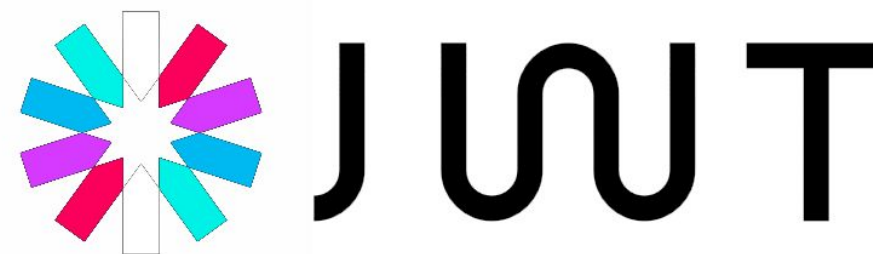
JSON
↗

JSON Web Token

El cuerpo o estructura de un JWT está
compuesto por tres partes separadas
por puntos (".")

Estas partes son:

- **cabecera** (header)
- **cuerpo** (payload)
- **firma** (signature)

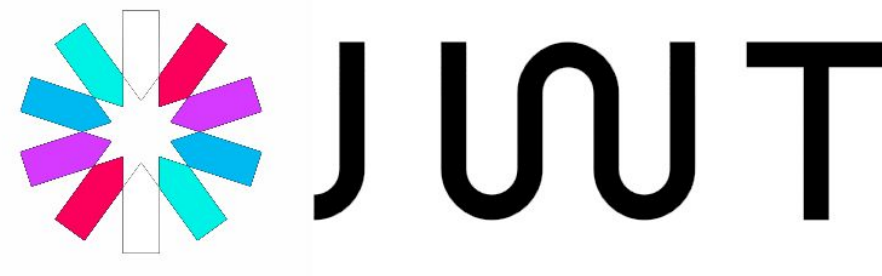


JSON
↗

JSON Web Token

La cabecera (header), es la primera parte del JWT y contiene información sobre el tipo de token y el algoritmo de firma utilizado.

Por lo general, consta de dos partes: el tipo de token, que es "JWT", y el algoritmo de firma utilizado, como por ejemplo "HS256" o "RS256".



JSON
↗

JSON Web Token

El **cuerpo (payload)**, es la segunda parte del JWT y contiene la información adicional que se quiere transmitir, como ser: datos de usuario, permisos, o cualquier otro dato relevante.

El contenido del cuerpo está codificado en **Base64Url** para que sea legible y transmisible en forma de texto.



JSON
↗

JSON Web Token

La **firma (signature)**, es la tercera y última parte del JWT. Se utiliza para verificar que el mensaje no ha sido alterado durante la transmisión y para asegurar la autenticidad del remitente.

La firma se calcula utilizando la cabecera codificada en Base64Url, el cuerpo codificado en Base64Url, una clave secreta conocida solo por el servidor emisor y el algoritmo de firma especificado en la cabecera.



JSON
↗

JSON Web Token

El proceso típico de uso de JWT en una aplicación web es el siguiente:



Autenticación

El usuario inicia sesión en la aplicación con sus credenciales.

El servidor de autenticación las verifica y, si son válidas, genera un JWT con la información de autenticación del usuario.



Autorización

El cliente (*generalmente un navegador web*) recibe el JWT y lo almacena, por ejemplo, en una cookie o en WebStorage.

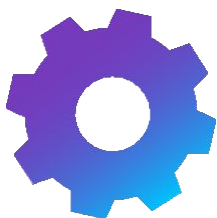
Luego, el cliente incluye el JWT en las cabeceras de las solicitudes a las API protegidas.



Validación

Los servidores de la API protegida validan el JWT recibido en cada solicitud para asegurarse de que sea válido, no haya sido alterado, y que esté firmado correctamente.

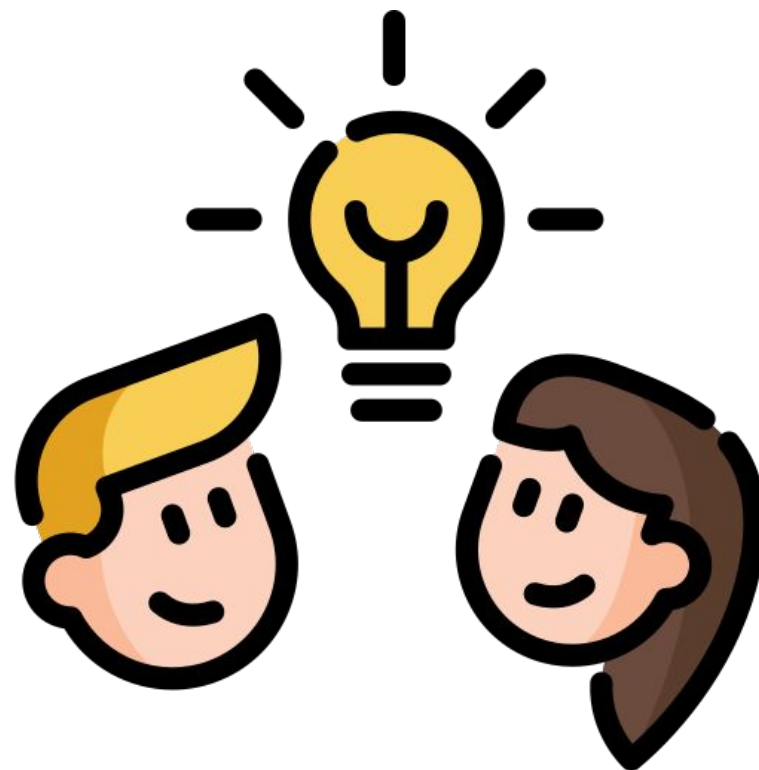
Si el JWT es válido, el servidor de la API autoriza y responde con los datos solicitados.



JSON Web Token

Una de las ventajas de usar JWT es que son autocontenidos, lo que significa que llevan consigo la información necesaria para verificar su autenticidad, lo que evita la necesidad de consultar una base de datos o almacenar información adicional en el servidor.

Sin embargo, también es importante tener en cuenta que los JWT deben ser almacenados y transmitidos de manera segura, ya que cualquier persona que tenga acceso a un JWT válido puede acceder a los recursos protegidos.

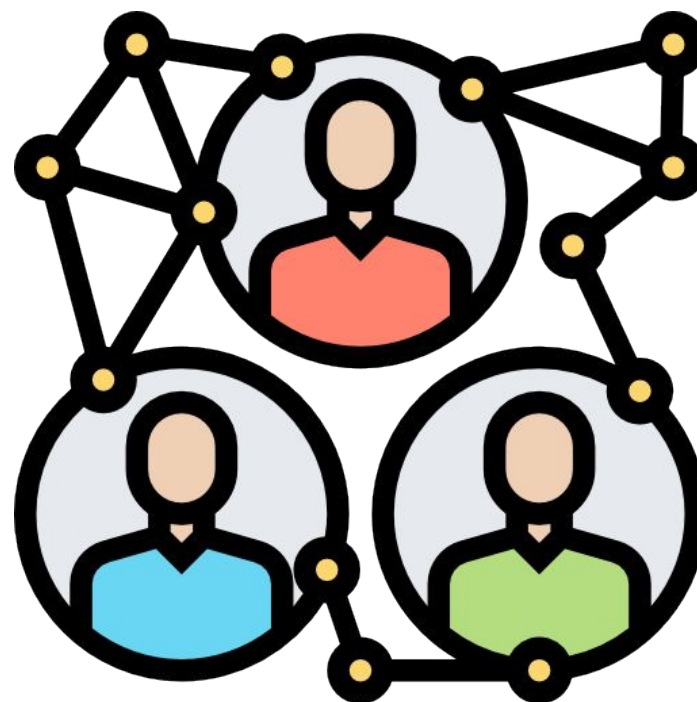


Cómo securizar nuestros desarrollos backend

Cómo securizar nuestros desarrollos backend

Vamos a integrar el módulo **jsonwebtoken** en esta aplicación backend, para entender cómo trabajar con éste. Para acelerar su construcción, la profe te compartirá una aplicación con las bases que utilizaremos en el ejemplo.

Deberás inicializar sus dependencias y crear el archivo **.env** necesario para que la aplicación funcione correctamente.



Cómo securizar nuestros desarrollos backend

Descargado el proyecto, abre el mismo con VS CODE y en la ventana **Terminal**, ejecuta el comando para que se instalen todas sus dependencias:

- *express*
- *dotenv*
- *jsonwebtoken*
- *nodemon*

```
/> npm install
```



```
.env
1 SECRET_KEY=halt-and-catch-fire-es-una-buena-clave-secreta
2 PORT=3008
```

Crea a continuación el archivo **.env** en donde definiremos dos variables de entorno.

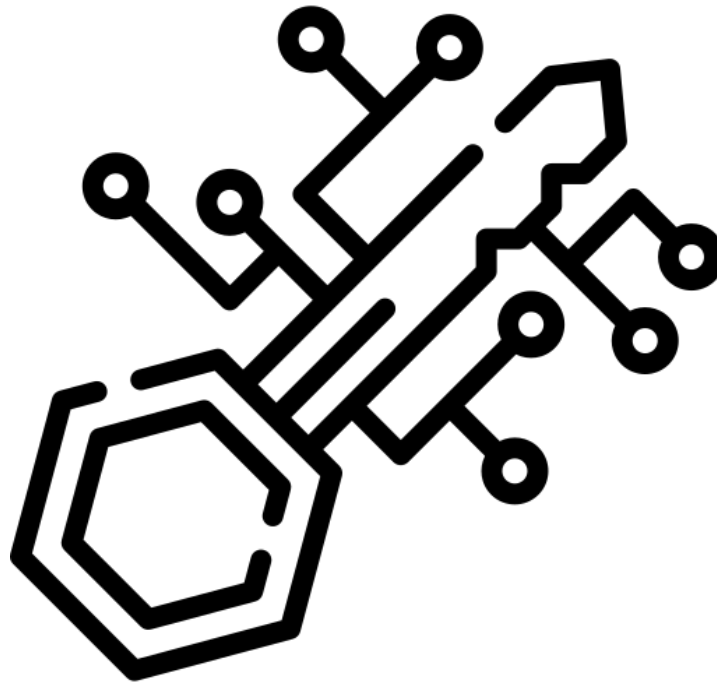
- **SECRET_KEY**
- **PORT**

En **PORT** definimos su valor habitual; el número de puerto.

Cómo securizar nuestros desarrollos backend

La variable de entorno, `SECRET_KEY`, nos servirá para definir parte del mecanismo de seguridad de JWT.

Esta “secret key” formará parte del proceso de autenticación, autorización y validación, que vimos inicialmente cuando hablamos de JWT.



Analizamos nuestra aplicación backend

Analizamos nuestra aplicación backend

JSON Web Token está instanciado en nuestra aplicación como **jwt**. A través de este accederemos a los métodos necesarios de autenticación y validación.

```
JWT

const express = require('express');
const app = express();
const jwt = require('jsonwebtoken');
require('dotenv').config();
const PORT = process.env.PORT || 3008;
const secretKey = process.env.SECRET_KEY;
const userToValidate = {username: 'Cameron', password: 'H@lt-And_Catch?F1re'}
```


Analizamos nuestra aplicación backend

La constante **secretKey** la utilizaremos como parámetro en los métodos de **jwt**.

```
JWT

const express = require('express');
const app = express();
const jwt = require('jsonwebtoken');
require('dotenv').config();
const PORT = process.env.PORT || 3008;
const secretKey = process.env.SECRET_KEY;
const userToValidate = {username: 'Cameron', password: 'H@lt-And_Catch?F1re'}
```

Analizamos nuestra aplicación backend

La constante **userToValidate** será nuestro usuario de pruebas, contra el cual validamos el registro inicial en nuestra aplicación backend.

```
const express = require('express');
const app = express();
const jwt = require('jsonwebtoken');
require('dotenv').config();
const PORT = process.env.PORT || 3008;
const secretKey = process.env.SECRET_KEY;
const userToValidate = {username: 'Cameron', password: 'H@lt-And_Catch?F1re'}
```

En aplicaciones reales, esta información esta almacenada en una base de datos, y debemos ir allí a buscarla para hacer una validación completa. **Nosotras la simularemos a través de este objeto literal.**

**Pongamos en marcha la
aplicación backend**

Primera etapa: Autenticación

Esta ruta será la que utilizaremos para identificarnos por primera vez en la aplicación backend, y generar el JWT correspondiente.



JWT

```
//LOGIN DE USUARIO. SE UTILIZARÁ PARA GENERAR SU JWT  
app.post('/login', (req, res) => {  
  
  })
```

Pongamos en marcha la aplicación backend

Trabajemos sobre el método **post()**. Lo primero que debemos hacer, es obtener los datos que llegan en el cuerpo de **request**. Básicamente será el **usuario** y contraseña que busca generar el token. Estos datos deben ser comparados contra nuestro objeto literal modelo al cual llamamos **userToValidate**.

```
JWT

const username = req.body.username
const password = req.body.password
console.log(`Datos recibidos: Usuario: ${username}, Password: ${password}`)
```

Pongamos en marcha la aplicación backend

Validamos si el usuario y contraseña recibidos coinciden con los esperados. De coincidir, invocamos a **jwt.sign()**. Este método espera el parámetro **username**, la **clave secreta** que creamos, y el tiempo de expiración del token. Definimos todo a través de los parámetros correspondientes.

```
JWT

if (username === userToValidate.username && password === userToValidate.password) {
  const token = jwt.sign({ username: username }, secretKey, { expiresIn: '1h' })
  res.json({ token: token })
} else {
  res.status(401).json({ error: 'Credenciales inválidas' });
}
```

Pongamos en marcha la aplicación backend

```
serToValidate.password) {  
  ey, { expiresIn: '1h' })
```

```
});
```

La propiedad **expiresIn** define el tiempo en el cual vencerá el token generado. Podemos definir un valor numérico específico, el cual es interpretado como segundos.

También podemos definir un valor en formato string, como el del ejemplo.

- “7d” //7 días
- “21h” //2 horas
- “4.5 hrs” //4 horas y media
- “2y” //2 años



Pongamos en marcha la aplicación backend

El método **.sign()** retornará un **token** generado con los parámetros informados. Este token debe ser retornado en una estructura del tipo JSON, al cliente que se identificó. Si las credenciales son inválidas, retornamos un código de estado **401 - Unauthorized**.

```
JWT

if (username === userToValidate.username && password === userToValidate.password) {
  const token = jwt.sign({ username: username }, secretKey, { expiresIn: '1h' })
  res.json({ token: token })
} else {
  res.status(401).json({ error: 'Credenciales inválidas' });
}
```



MiddleWare de verificación de token

Segunda etapa: Autorización

Esta será la función que validará el token recibido, y lo decodificará para saber si el mismo es válido, inválido o si no ha sido proporcionado.

```
JWT  
  
function verifyToken(req, res, next) {  
  
}
```

MiddleWare de verificación de token

A través de la constante **token**, recibimos el token que el cliente envía cuando quiere acceder a contenido protegido.

Este token se recupera del objeto **request**, en el array **headers**.

Lo almacenamos bajo la clave **'authorization'**.

```
JWT
const token = req.headers['authorization'] || null
if (token) {
  jwt.verify(token, secretKey, (err, decoded) => {
    err ? res.status(401).json({ error: 'Token inválido' })
      : req.decoded = decoded
    next()
  })
} else {
  res.status(401).json({ error: 'Token no proporcionado' })
}
```

MiddleWare de verificación de token

El método **jwt.verify()** nos permite validar el **token** recibido, constatando el mismo contra la clave secreta. Esta validación retorna el token decodificado, y un posible error si es que ocurre el mismo.

De haber un error, notificamos con el código de estado **401**.

```
JWT

const token = req.headers['authorization'] || null
if (token) {
  jwt.verify(token, secretKey, (err, decoded) => {
    err ? res.status(401).json({ error: 'Token inválido' })
      : req.decoded = decoded
    next()
  })
} else {
  res.status(401).json({ error: 'Token no proporcionado' })
}
```

Si todo fue bien, tendremos el token decodificado, desde donde luego obtendremos el **username**.

Definir la lógica para la ruta protegida

Tercera etapa: Validación

Esta será nuestra ruta protegida. Solo veremos su contenido si la validación del token a través de la función **verifyToken()** es aprobada; caso contrario, responderemos con un código de error **401**.

```
JWT  
app.get('/rutaprotegida', verifyToken, (req, res) => {  
  })
```

Definir la lógica para la ruta protegida

Nuestra ruta protegida será definida mediante el método **GET** convencional, con la diferencia que ahora recibe un parámetro adicional, el cual corresponde a **verifyToken**. Esta función validará si se puede o no mostrar el contenido de esta ruta, dado que validará el token que debe recibir este endpoint en el header de la petición del cliente.

```
JWT
app.get('/rutaprotegida', verifyToken, (req, res, next) => {
  const username = req.decoded.username;
  res.json({ mensaje: `Hola, ${username}! Esta ruta está protegida.` })
  next()
})
```

Definir la lógica para la ruta protegida

Si no pasa la validación, nunca veremos el contenido que entrega esta ruta de forma predeterminada. Finalmente, con los datos protegidos, ya podemos definir en esta ruta, si enviamos datos sensibles como puede ser un JSON con un array de datos privados, o cualquier otra cosa que necesitemos entregarle a los clientes.

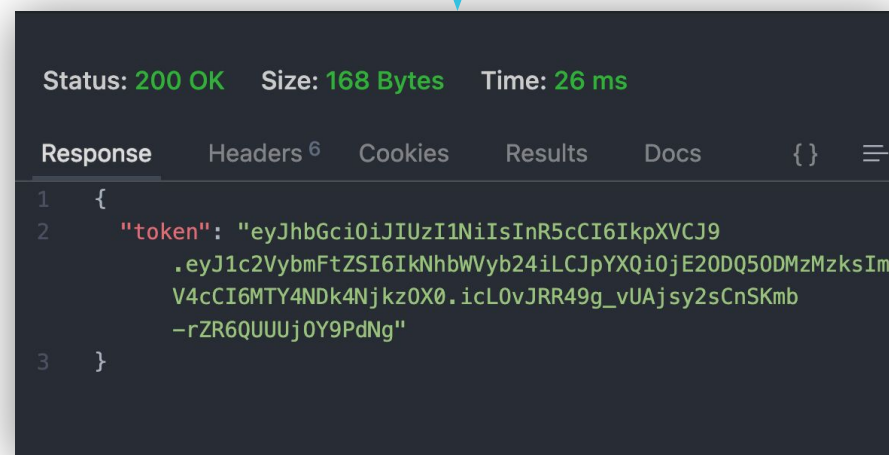
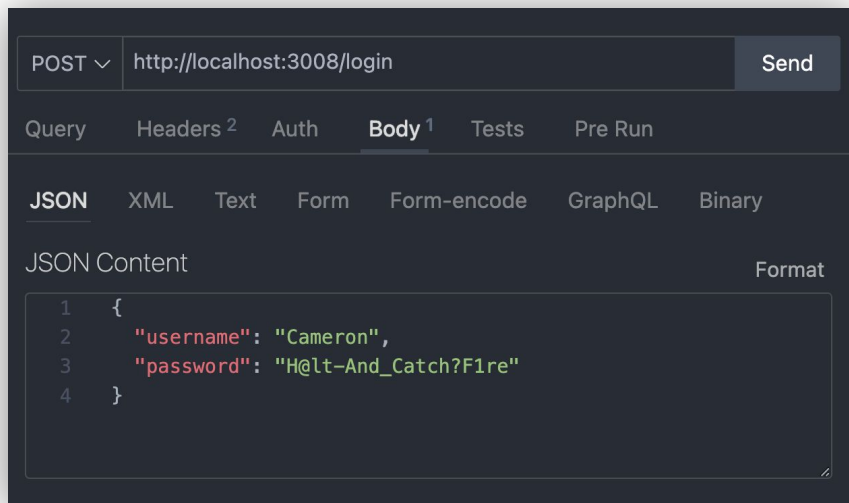
```
JWT

app.get('/rutaprotegida', verifyToken, (req, res, next) => {
  const username = req.decoded.username;
  res.json({ mensaje: `Hola, ${username}! Esta ruta está protegida.` })
  next()
})
```


Probar JWT

Probar JWT

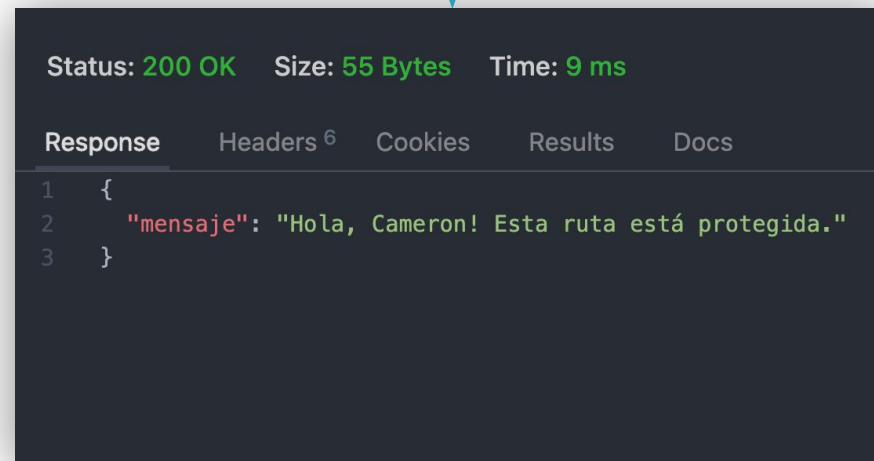
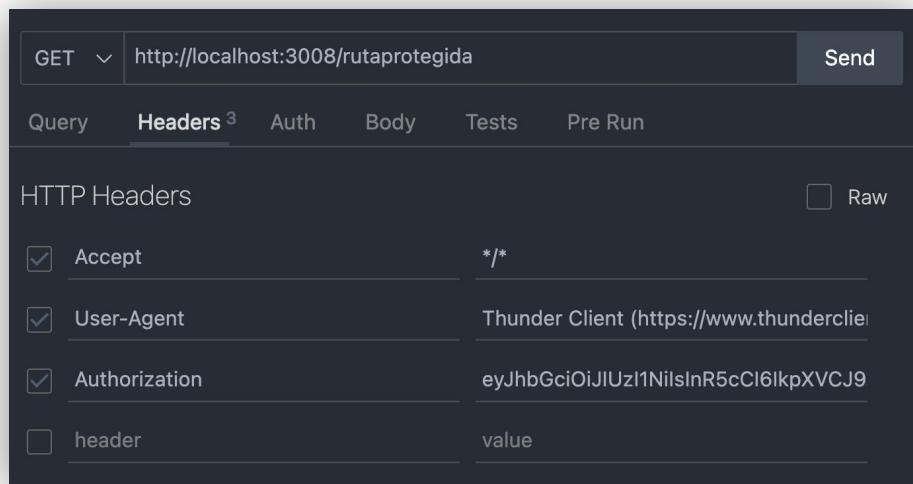
Testeamos nuestra aplicación a través de un cliente **HTTP**. Definimos la ruta **/login** con el método **POST** y, en el cuerpo de la petición, definimos el usuario y contraseña necesario.



Si pasamos la validación enviando un usuario y contraseña válidos, la aplicación generará un token y lo retornará. El cliente HTTP debe almacenar el mismo para luego poder peticionar sobre rutas protegidas.

Probar JWT

Obtenido el token, podemos intentar acceder a la ruta protegida. Para ello, definimos el endpoint **/rutaprotegida** y, en el encabezado de la petición, agregamos el parámetro **Authorization** y su valor; el token recibido previamente.



Al enviar la petición, si pasamos la validación correspondiente, obtendremos un código de estado **200** y los datos que nos envíe el endpoint como respuesta.

Probar JWT

Si intentamos ingresar a una ruta protegida sin un token informado, veremos el código de estado **401** y un mensaje de que **no se ha proporcionado el token**.

```
Status: 401 Unauthorized Size: 34 Bytes Time: 10 ms

Response Headers 6 Cookies Results Docs
1 {
2   "error": "Token no proporcionado"
3 }
```

```
Status: 401 Unauthorized Size: 27 Bytes Time: 22 ms

Response Headers 6 Cookies Results Docs
1 {
2   "error": "Token inválido"
3 }
```

Ante la petición de una ruta protegida enviando un **token incorrecto o vencido**, recibiremos un error **401**, y un mensaje asociado sobre la invalidez del token en cuestión.

Sección práctica

De acuerdo al ejemplo realizado por la profe, flexibilizaremos su funcionalidad para poder identificar diferentes usuarios, generar diferentes tokens, y servir información a todo usuario que disponga de la llave correspondiente.



Prácticas

Aprovechemos la base del token que hemos creado hasta aquí, y agreguemos en esta un array de objetos que disponga de, al menos 4 usuario con diferentes contraseñas.

Este array reemplaza a nuestro objeto básico llamado **userToValidate**, utilizado en el ejemplo anterior. En la lógica del código, previo validar el usuario, debemos buscar si el mismo existe en el array que creamos.

Integraremos para esto el método **array.find()**.

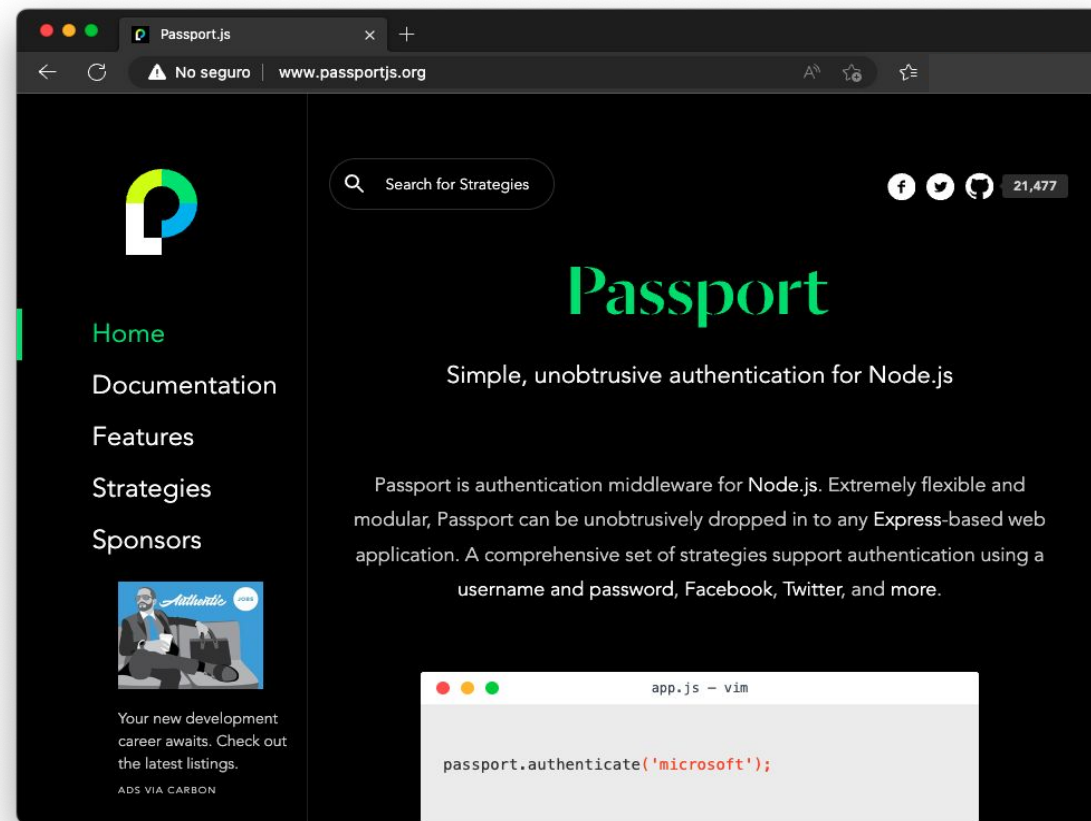
Por último, cuando el usuario recibido haya pasado la validación correspondiente, debemos servir a través de **/rutaprotegida**, algún array de objetos en formato JSON. Puede ser nuestro clásico **array frutas**, algún otro array que tengas con datos de tu interés, o puedes también recuperar nuestro **array trailerflix**. Puede ser en formato array de objetos JS o a través de la lectura de un archivo .JSON.

Otras herramientas de autenticación

Otras herramientas de autenticación

[Passport.js](#) es una biblioteca de autenticación para aplicaciones web. Actúa como un intermediario entre tu aplicación y los diferentes métodos de autenticación, como ser: iniciar sesión con una cuenta de Google, Microsoft, LinkedIn o con un usuario y contraseña.

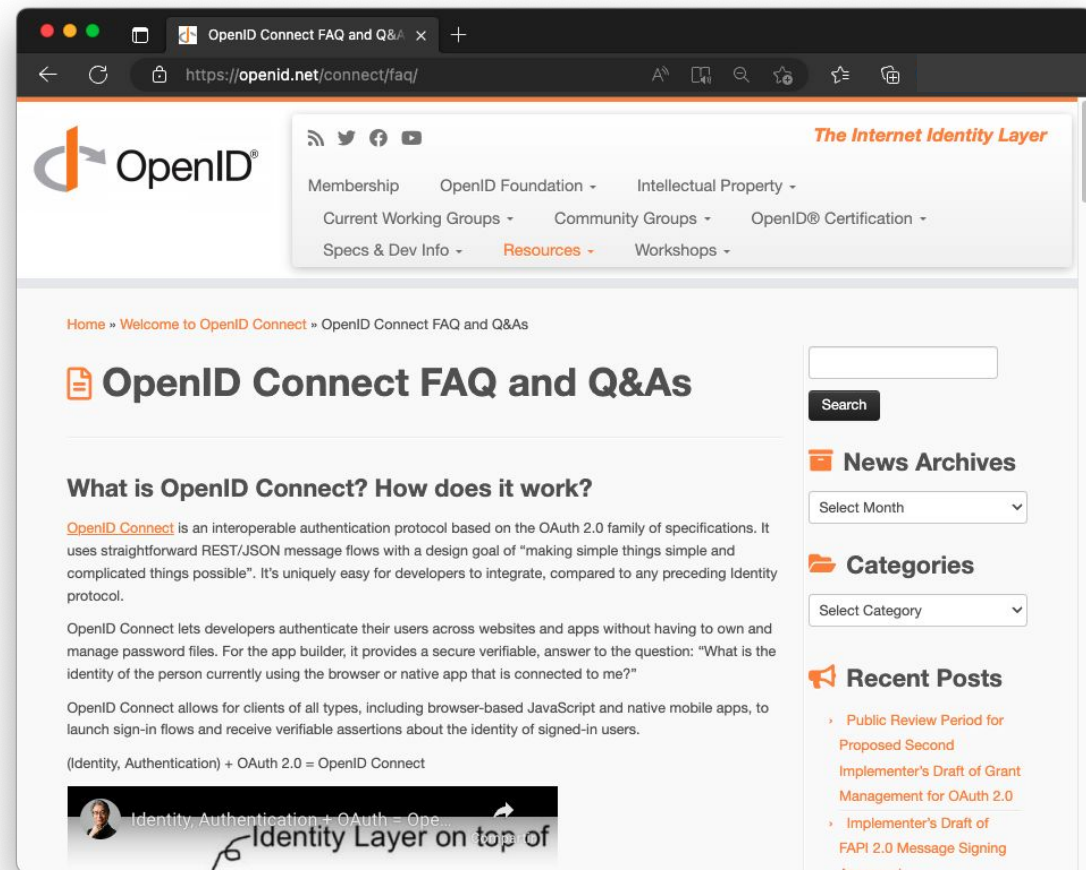
Passport.js se encarga de manejar los detalles complicados y tediosos de la autenticación, como verificar las credenciales, proteger las rutas de tu aplicación y mantener la sesión del usuario.



Otras herramientas de autenticación

OpenID es un estándar de autenticación en línea que te permite iniciar sesión en múltiples sitios web utilizando una única identidad digital.

En lugar de crear una cuenta separada en cada sitio web, puedes utilizar tu cuenta existente de un proveedor de identidad confiable, como Google o Facebook, para iniciar sesión en otros sitios web compatibles con OpenID.

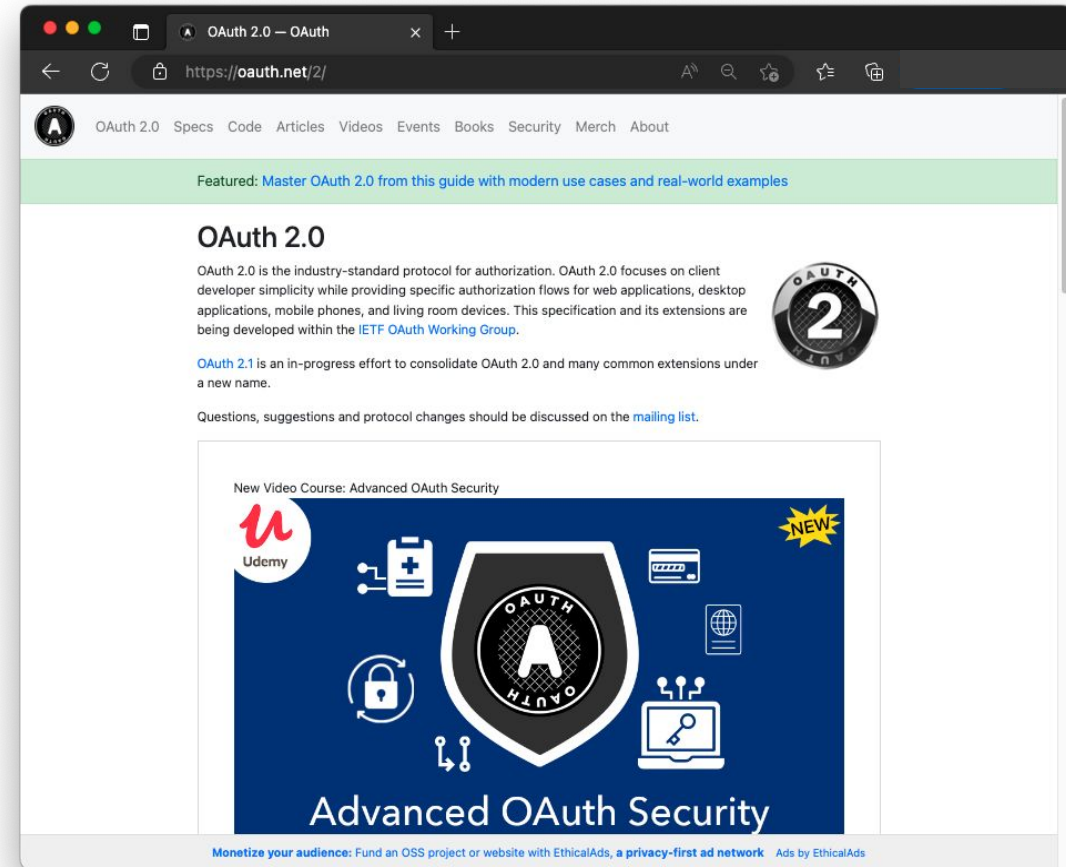




Otras herramientas de autenticación

OAuth2 es un protocolo de autorización que te permite compartir de manera segura tu información personal entre diferentes servicios en línea sin tener que proporcionar tu nombre de usuario y contraseña en cada uno de ellos.

Funciona como un "*permiso temporal*" que otorgas a un servicio para acceder a ciertos datos en otro servicio en tu nombre.



Encriptación

Encriptación

Todos los mecanismos que se utilizan en el proceso de autenticación y validación, tienen o se relacionan en algún punto con el mundo de la encriptación.

Y para entender a la encriptación, podemos pensar en que sería algo parecido a poner tu mensaje en un sobre sellado que solo tú y la persona a la que se lo envías pueden abrir.

Es una forma de convertir tu mensaje (*la contraseña en nuestro caso*) en algo ilegible para los demás, a menos que tengan la "*llave*" correcta para desbloquearlo.

0	1	1	0
1	0	0	1
1	0	1	0

Encriptación

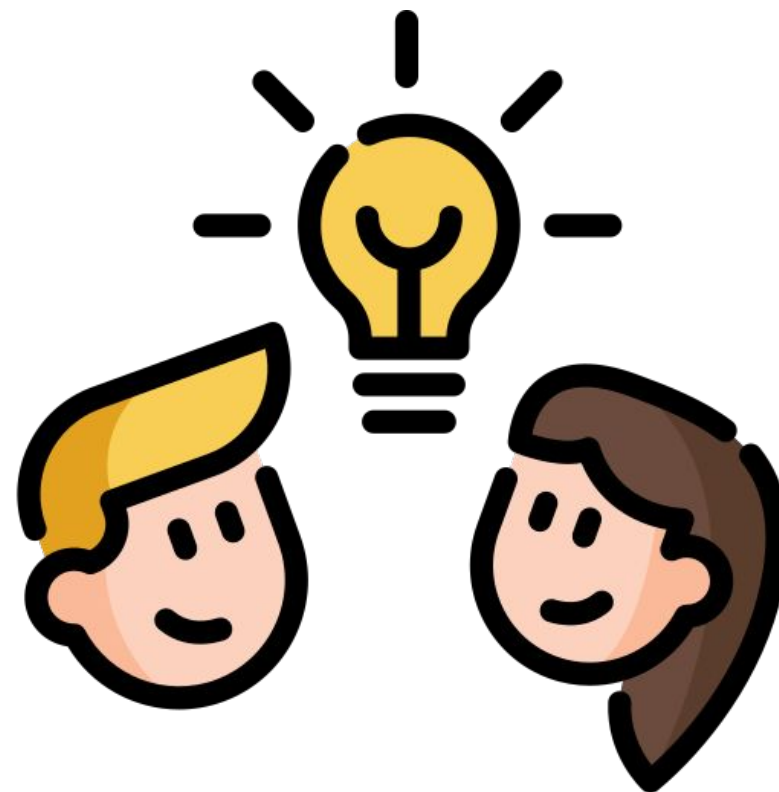
JWT (JSON Web Token) en sí mismo no especifica un mecanismo de encriptación, sin embargo, admite diferentes algoritmos de encriptación para proteger el contenido del token, principalmente a través del uso de **JWE (JSON Web Encryption)**.

En el contexto de JWT, la encriptación se aplica principalmente al contenido del token, es decir, la carga útil (*payload*). Esto permite ocultar la información contenida en el token a terceros no autorizados.

0	1	1	0
1	0	0	1
1	0	1	0

Encriptación

Te compartimos un [artículo de ciberseguridad](#), que cuenta un poco más en detalle la relación entre JWT, JWE, y otros tantos actores importantes en el mundo de la autenticación.



Encriptación

Cuando se utiliza la encriptación en JWT un algoritmo de encriptación simétrica o asimétrica es implementado para cifrar la carga útil.

Los algoritmos comunes usados en JWT:

- **AES** (*Advanced Encryption Standard*)
- **RSA** (*Rivest-Shamir-Adleman*)

0	1	1	0
1	0	0	1
1	0	1	0

Encriptación

AES

Algoritmo de encriptación simétrica
(*Advanced Encryption Standard*)

Utilizan una clave compartida para cifrar y descifrar el contenido del token. La misma clave se utiliza para la encriptación y la desencriptación.

0	1	1	0
1	0	0	1
1	0	1	0

Encriptación

RSA

Algoritmo de encriptación asimétrica
(Rivest-Shamir-Adleman)

Utilizan un par de claves: una pública para la encriptación y una clave privada para la descryptación. La pública es compartida con los destinatarios autorizados para que puedan descifrar el contenido del token.

0	1	1	0
1	0	0	1
1	0	1	0

¡Muchas gracias!



Ministerio de Economía
Argentina

Secretaría de
Economía del Conocimiento

*primero
la gente*