***Notas del Curso.***

**Stack de seguridad moderno.**

Anteriormente las compañías se comunicaban mediante un [intranet](https://es.wikipedia.org/wiki/Intranet), un [intranet](https://es.wikipedia.org/wiki/Intranet) a diferencia del [internet](https://es.wikipedia.org/wiki/Internet) es una red privada que funciona dentro de las compañías, en esta red había protocolos como [SOAP](https://es.wikipedia.org/wiki/Simple_Object_Access_Protocol), [SAML](https://es.wikipedia.org/wiki/Security_Assertion_Markup_Language), [WS-Federation](https://en.wikipedia.org/wiki/WS-Federation), pero esos protocolos se quedaron muy cortos cuando llegó la **revolución mobile**, además tecnologías como [HTML5](https://es.wikipedia.org/wiki/HTML5), empezaron a necesitar otra serie de cosas y conceptos como la autenticación y la autorización, también necesitaban una evolución, además el auge de los [microservicios](https://docs.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices) y la necesidad de tener múltiples clientes, hicieron la creación de un nuevo STACK, esté stack se compone generalmente de 3 protocolos: [JSON Web Tokens](https://es.wikipedia.org/wiki/JSON_Web_Token), [OAuth 2.0](https://es.wikipedia.org/wiki/OAuth#OAuth_2.0) [o](https://www.digitalocean.com/community/tutorials/una-introduccion-a-oauth-2-es) , y [OpenID Connect](https://es.wikipedia.org/wiki/OpenID_Connect).

* **JSON Web Tokens**: Son un estándar de la industria abierto que nos permite comunicarnos entre 2 clientes de un lado a otro de una manera más segura.
* **OAuth 2.0**: Es un estándar de la industria que permite implementar autorización, hay que tener mucho cuidado en **No confudir autorización con autenticación**. Precisamente una de las diferencias de ***open autorización 1.0(OAuth 1.0)*** y su versión 2 fue la necesidad de adaptarse a estas nuevas tecnologías mobile.
* **OpenID Connect**: Es una capa de autenticación que funciona por encima de ***OAuth 2.0***



# ¿Qué es la autenticación y la autorización ?

# La *autenticación* sirve para verificar la identidad de un usuario, verificar si el usuario existe y si los datos que está colocando son correctos.



La ***autorización*** es la acción de permitir a un usuario acceso limitado a nuestro recursos.

Analogía:

*"Los carros modernos suelen tener 2 llaves, una llave que sirve para conducir y 1 llave que sirve para el pallet parking. El pallet parking es un servicio que tienen algunos restaurantes donde llegas con tu carro, donde te recibe alguien y lo estaciona por ti,* ***estas llaves tienen permisos limitados****, solo permiten encender el carro a cierta velocidad, no alejarse de cierta área."*

En los sistemas pasa algo muy similar, nosotros a veces otorgamos permisos de solo lectura y escritura, es nuestra aplicación nosotros vamos a otorgar una serie de permisos, unos permisos que son del usuario final que son de lectura y escritura sobre ciertas colecciones, pero también vamos a otorgar otros permisos administrativos, y esto lo vamos a hacer manejando unos tokens que vamos a otorgarle a nuestro servidor.

**Introducción a las sesiones.**

Cuando tu visitas un sitio web se crea una petición http. [Http](https://es.wikipedia.org/wiki/Protocolo_de_transferencia_de_hipertexto) es un protocolo que no tienen estado esto quiere decir que diferentes direcciones http nunca comparten información entre si, así que la manera de poder compartir está información en peticiones http es mediante el uso de una sesión. Cuando visitas un sitio web por primera vez se crea una sesión, no es necesario que estés autenticado para que esta sesión sea creada.

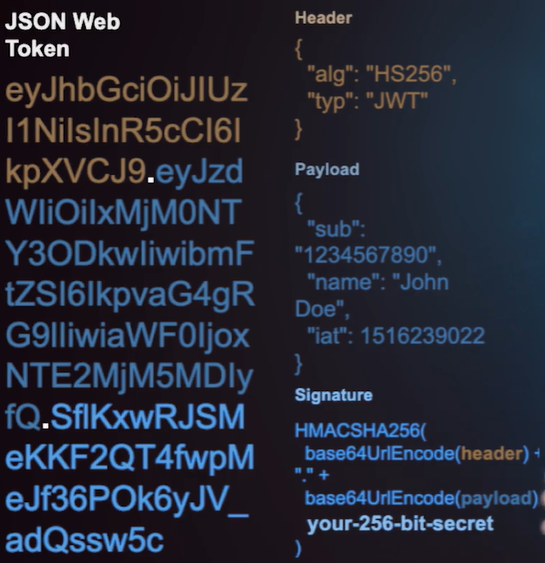
Supongamos que vas a un sitio a buscar vuelos, cuando tu entras al sitio se te crea una sesión y a menudo que vas haciendo búsqueda a esos vuelos, se van guardando tus preferencias de búsqueda en esta sesión, luego está sesión genera un ID que se almaceno en una [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)). La [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) es un archivo que se almacena en tu navegador, para que cuando tu cierres el navegador la [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) permanezca con el id de la sesión, así la próxima vez que vuelvas esté ID de la sesión que permanece en la [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) se relaciona con la sesión que estaba previamente abierta y así puede cargar tus preferencias en los vuelos que estabas buscando.

Es por eso que muchas veces aunque nosotros no iniciemos sesión podemos ver que nuestras preferencias está ahí, también cuando hay un proceso de autenticación, la sesión se almacena directamente y se relaciona con tu usuario, por seguridad la sesión debería terminar ciertos minutos después de que hay un inactividad, sin embargo dependiendo el mecanismo que estés usando podrías tener sesiones por días o incluso por meses **cookiesSession** y [express-session](https://www.npmjs.com/package/express-session) son librerías que nos permiten implementar todo el tema de sesiones en express, la diferencia más grande es que **cookiesSession** nos permite almacenar la sesión en la [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)), mientras que **express-session** nos permite almacenar la sesión en la memoria en el lado del servidor.

A la hora de escalar la sesión es muy importante utilizar bases de datos en memoria como Redis, eso es una ventaja que tiene JWT, pues JWT no tiene estado y por lo tanto no necesita memoria, pero más adelante vamos a ver cuáles son las diferentes ventajas y desventajas sobre JWT y Sesiones.

**Anatomía de un JWT**

**JWT** es un estándar de la industria que nos permite manejar demandas de información entre dos clientes.



En la columna de la izquierda puedes ver cómo es un JWT mientras que en la columna de la derecha puedes ver un JWT decodificado.

Un JWT consta de 3 partes: **Header, Payload y Signature**, generalmente divididas por un punto.

* **Header**: tiene 2 atributos, el tipo que en este caso siempre debería ser JWT y el algoritmo de encriptación de la firma, el algoritmo de encriptación de la firma **puede ser síncrono o asincrono**. Recordemos que los algoritmos ***asíncronos*** **usan 2 llaves** de encriptación; una llave **privada** y una llave **pública**, donde la llave pública se usa para encriptar y la llave privada se usa para desencriptar. Los algoritmos de encriptación ***síncronos*** en cambio usan la misma llave para desencriptar e encriptar, ambos son seguros de usar pero depende donde los uses.

Los **algoritmos asíncronos** deben usarse donde hay partes públicas que nos pueden poner en riesgo accediendo a nuestra llave, como por ejemplo en un protocolo **HTTP**, mientras que los **algoritmos síncronos** solo deben usarse en sistemas como el **backend.**

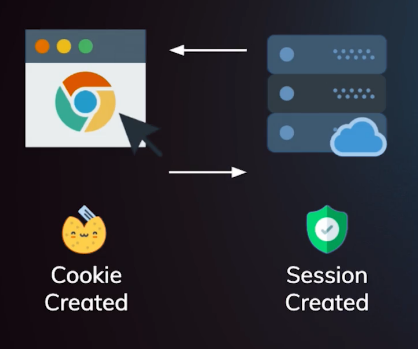
* **Payload**: Es donde guardamos toda la información de nuestro usuario, incluso todos los scopes de autorización, esté payload se compone de algo llamado los **claims**, los claims son generalmente representados por 3 letras para mantener el JWT muy pequeño, hay diferentes tipos de claims.

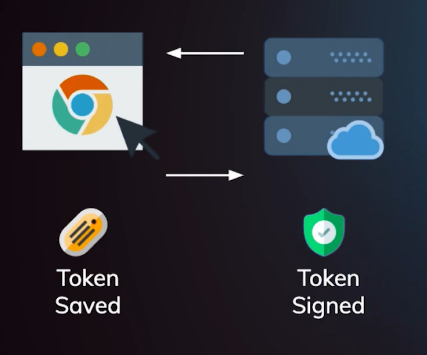
Nosotros en la página donde está el estándar podemos ver en la sección 4.1 lo que se llama los Registered Claim Names. Estos son claims específicos que tienen una definición propia y debe respetarse.

También podemos usar los Public Claim Names, estos pueden usarse entre diferentes aplicaciones y ya están también definidos, mientras que los **Private Claim Names**, son los que tu defines para tu aplicación.

* **Signature**: La tercera parte del JWT que es la firma y es lo que hace muy poderoso el JWT está compuesto por el **header codificado** más el **payload codificado**, ha esto se le aplica el algoritmo de encriptación por su puesto usando un secret. En el caso del algoritmo H256 debemos usar un string de 256 bits de longitud.

# Autenticación tradicional vs JWT

En la autenticación tradicional cuando sucede un proceso de autenticación se crea una sesión, el id de esta sesión se almacena en una [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) que es enviada al navegador. Recordemos que las [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica))s no se llaman [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica))s por las galletas de chocolate, sino que se llaman [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica))s **por las galletas de la fortuna que tienen mensajes**, apartir de ahí todos los request tienen la [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) que tiene almacena el id de la sesión y está es usada para verificar la sesión previamente activa, uno de los problemas que tiene es esté browser es por ejemplo: clientes como las **Single Pages Apps**, no pueden refrescar o no pueden refrescar todas las veces entonces no pudieron saber si hubo cambios en la sesión. Otro problema es que por definición las Rest API no deberían tener estado, al usar sesiones estamos generando estado y esto contradice esté principio, otro problema es que en arquitecturas modernas que usan por ejemplo microservicios, la sesión que solo existe en una máquina no fluye durante los otros clientes, entonces es un poco difícil de escalar, y otro problema es que por ejemplo el control de acceso siempre requiere que vayamos a base de datos, finalmente controlar el uso de memoria también puede ser un problema, ya que cada cliente que se conecta genera una sesión generando más consumo de memoria.



En la autenticación con JWT al suceder el proceso de autenticación se firma un token, apartir de ahí el token es enviado al cliente y esté deber ser almacenado en memoria o en una [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)), todos los request de aquí en adelante llevan esté token, una de las ventajas es que una aplicación como una **Single Pages App** ya no requiere del backend para saber si el usuario está autenticado, lo otro es que el backend puede recibir múltiples request de múltiples clientes y lo único que le interesa es saber si el token está bien firmado, finalmente es el cliente quien sabe que permisos tienen y no tiene que ir hasta base de datos para saber si tiene estos permisos.

# Firmando y Verificando nuestro JWT

Para firmar nuestro token utilizaremos un paquete de node llamado jsonwebtoken y al usarlo en nuestro código se verá de esta manera:



El primer atributo que recibe es el payload o sea los datos que guardaremos en ese token. De segundo atributo recibe una clave secreta con la cual será firmado y finalmente podremos pasarle opciones si es nuestro caso.

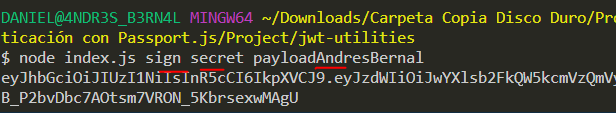
Para verificar nuestro token lo haremos de la siguiente manera:



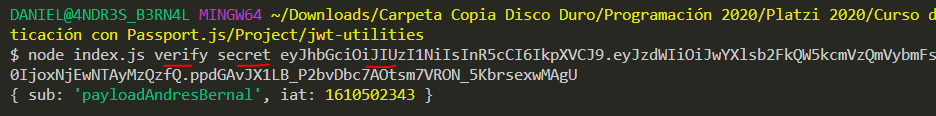
Como primer atributo recibiremos el token, de segundo atributo el secreto de la firma y como tercer argumento (opcional) recibiremos el token decodificado.

* Vamos a inicializar nuestro proyecto con npm init -y
* Crearemos el archivo index.js
* Vamos a instalar los paquetes necesarios con npm i jsonwebtoken
* En el index.js vamos a hacer toda la lógica de nuestra aplicación

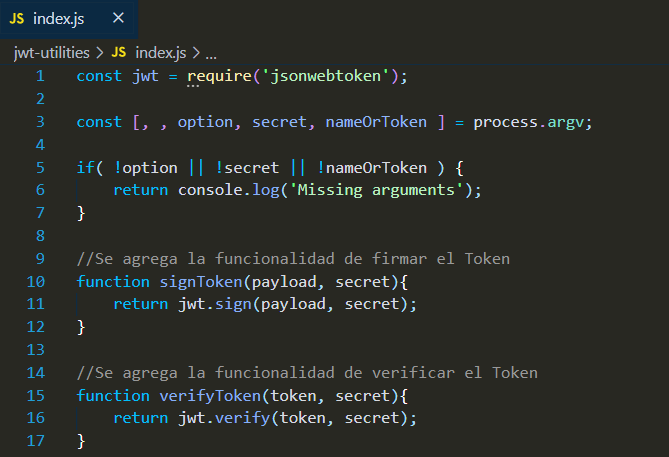
Para generar Token:

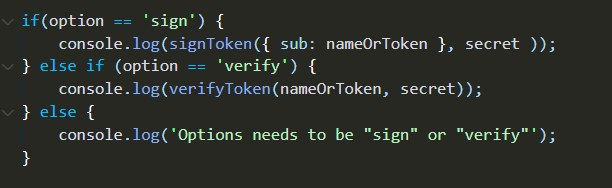


Para verificar Token:



**En el código.**





# Server-side vs Client-side sessions.

# Sesiones del lado del servidor vs sesiones del lado del cliente

## ***¿Qué es una sesión?***

En términos generales una sesión es una manera de preservar un estado deseado.

## ***¿Qué es una sesión del lado del servidor?***

La sesión en el lado del servidor suele ser una pieza de información que se guarda en memoria o en una base de datos y esta permite hacerle seguimiento a la información de autenticación, con el fin de identificar al usuario y determinar cuál es el estado de autenticación. Mantener la sesión de esta manera en el lado del servidor es lo que se considera “stateful”, es decir que maneja un estado.

## ***¿Qué es una sesión del lado del cliente?***

Las SPA (Single-page apps) requieren una manera de saber si el usuario esta autenticado o no. Pero esto no se puede hacer de una manera tradicional porque suelen ser muy desacopladas con el backend y no suelen refrescar la página como lo hacen las aplicaciones renderizadas en el servidor.

JWT (JSON Web Token) es un mecanismo de autenticación sin estado, lo que conocemos como “stateless”. Lo que significa que no hay una sesión que exista del lado del servidor.

La manera como se comporta la sesión del lado del cliente es:

1. Cuando el usuario hace “login” agregamos una bandera para indicar que lo está.
2. En cualquier punto de la aplicación verificamos la expiración del token.
3. Si el token expira, cambiamos la bandera para indicar que el usuario no está logueado.
4. Se suele chequear cuando la ruta cambia.
5. Si el token expiró lo redireccionamos a la ruta de “login” y actualizamos el estado como “logout”.
6. Se actualiza la UI para mostrar que el usuario ha cerrado la sesión.

**Mas sobre Sesiones y cookies:**

[***https://programacionymas.com/blog/jwt-vs-cookies-y-sesiones***](https://programacionymas.com/blog/jwt-vs-cookies-y-sesiones)

# Buenas prácticas con JSON Web token.

En los últimos años se ha criticado fuertemente el uso de JSON Web Tokens como buena práctica de seguridad. La realidad es que muchas compañías hoy en día los usan sin ningún problema siguiendo unas buenas prácticas de seguridad, que aseguran su uso sin ningún inconveniente.

A continuación listaremos unos consejos que se deben tener en cuenta:

## **Evitar almacenar información sensible**

Debido a que los JSON Web tokens son decodificables es posible visualizar la información del payload, por lo que ningún tipo de información sensible debe ser expuesto como contraseñas, keys, etc. Tampoco debería agregarse información confidencial del usuario como su número de identificación o información médica, ya que como hablamos anteriormente, los hackers pueden usar esta información para hacer ingeniería social.

## **Mantener su peso lo más liviano posible**

Suele tenerse la tentación de guardar toda la información del perfil en el payload del JWT, pero esto no debería hacerse ya que necesitamos que el JWT sea lo más pequeño posible debido a que al enviarse con todos los request estamos consumiendo parte del ancho de banda.

## **Establecer un tiempo de expiración corto**

Debido a que los tokens pueden ser robados si no se toman las medidas correctas de almacenamiento seguro, es muy importante que estos tengan unas expiración corta, el tiempo recomendado es desde 15 minutos hasta un maximo de 2 horas.

## **Tratar los JWT como tokens opacos**

Aunque los tokens se pueden decodificar, deben tratarse como tokens opacos, es decir como si no tuviesen ningún valor legible. Esto es porque desde el lado del cliente no tenemos manera de verificar si la firma es correcta, así que si confiamos en la información decodificada del token, alguien podría introducir un token invalido con otra información a propósito. Lo mejor, es siempre enviar el token del lado del servidor y hacer las verificaciones allí.

## **¿Dónde guardar los tokens?**

Cuando estamos trabajando con SPA (Single Page apps) debemos evitar almacenar los tokens en Local Storage o Session Storage. Estos deben ser almacenados en memoria o en una Cookie, pero solo de manera segura y con el flag **httpOnly**, esto quiere decir que la cookie debe venir del lado del servidor con el token almacenado. Más información: <https://auth0.com/docs/security/store-tokens#single-page-apps>

## **Silent authenticacion vs Refresh tokens**

Debido a que es riesgoso almacenar tokens del lado del cliente, no se deberían usar Refresh Tokens cuando se trabaja solo con una SPA. Lo que se debe implementar es Silent Authentication, para ello se debe seguir el siguiente flujo:

1. La SPA obtiene un access token al hacer login o mediante cualquier flujo de OAuth.
2. Cuando el token expira el API retornara un error 401.
3. En este momento se debe detectar el error y hacer un request para obtener de nuevo un access token.
4. Si nuestro backend server tiene una sesión valida (Se puede usar una cookie) entonces respondemos con un nuevo access token.

Más información:

* <https://auth0.com/docs/api-auth/tutorials/silent-authentication>
* <https://auth0.com/docs/tokens/refresh-token/current>

Hay que tener en cuenta que para implementar Silent authentication y Refresh tokens, se requiere tener un tipo de sesión valida del lado del servidor por lo que en una SPA es posible que sea necesario una especie de backend-proxy, ya que la sesión no debería convivir en el lado del API server.

En el paso 2, si se está usando alguna librería para manejo de estado como redux, se puede implementar un middleware que detecte este error y proceda con el paso 3.

**Pagina para desencriptar un token:**

[**https://jwt.io/**](https://jwt.io/)

**¿Qué son las cookies y cómo implementar el manejo de sesión?**

Una *cookie* es un archivo creado por un sitio web que tiene pequeños pedazos de datos almacenados en él. Su propósito es identificar al usuario mediante el almacenamiento de su historial.

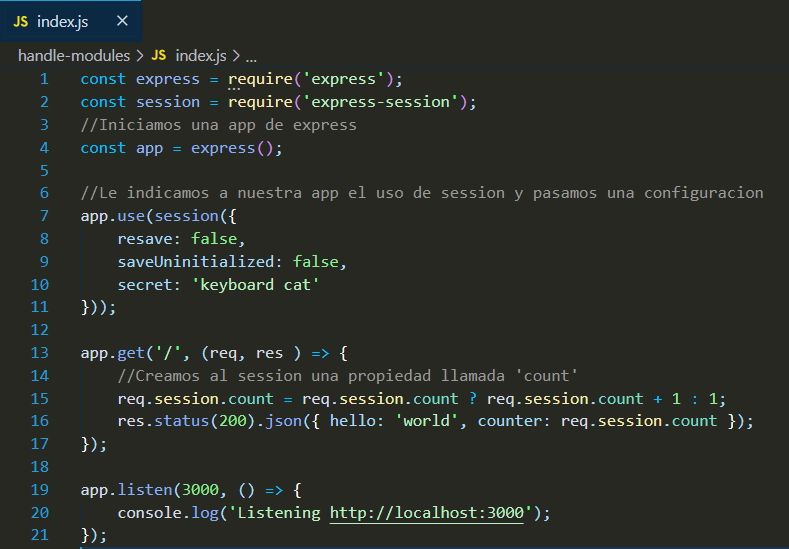
Las *cookie session* son cookies que tienen un corto periodo de vida ya que son removidas cuando el navegador o la pestaña se cierran.

Las *persistent cookies* se usan generalmente para guardar información de interés para el usuario.

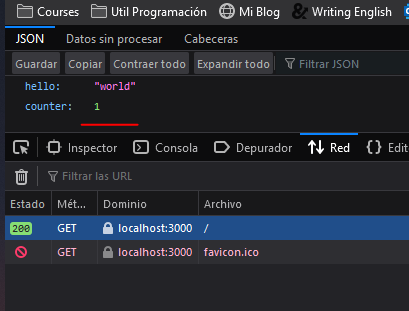
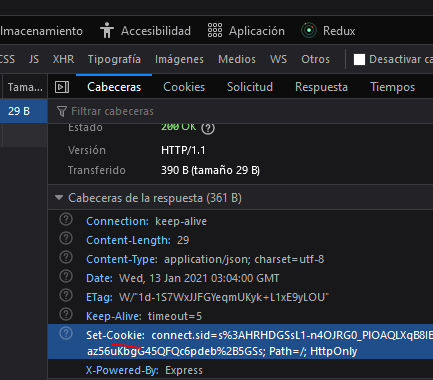
Las *secure cookies* almacenan datos de forma cifradas para que terceros no puedan tener acceso a ellas, se suelen usar en conexiones **HTTPS** (Conexiones seguras).

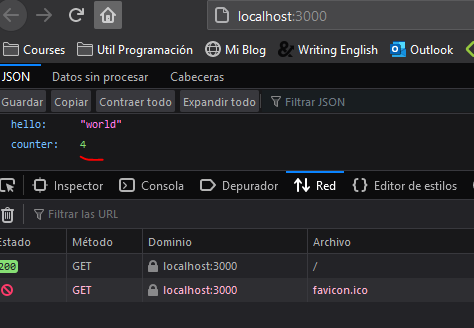
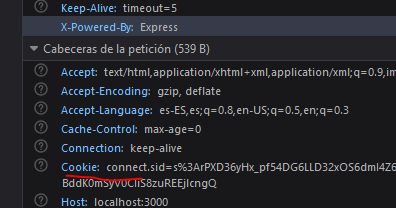
Hay leyes sobre cookies que debes seguir al pie de la letra:

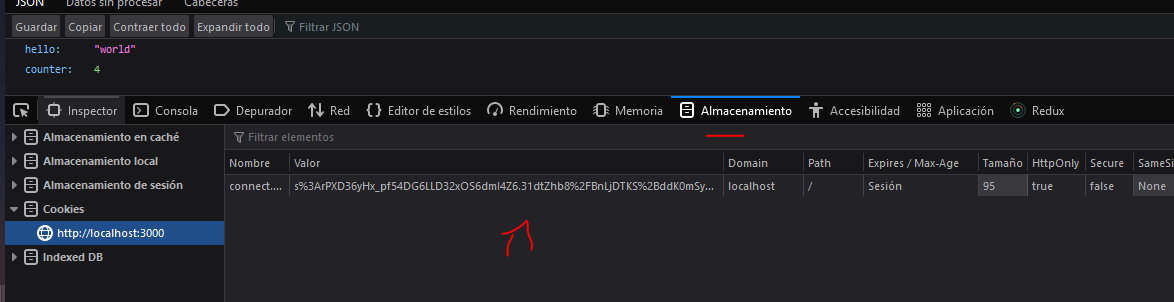
1. Avisarle al usuario que estás haciendo uso de cookies en tu sitio para guardar información
2. Es necesario que el usuario de su consentimiento para manejar cookies en tu sitio.



Podemos observar que cuando inicia el servidor, en primero momento se crea la cookie:

Y cuando volvemos y recargamos la cookie ya va existir y va ser enviada en el request para su manipulación:

Podemos ver y eliminar las cokies en Almacenamiento / Storage:

## **Cookies vs Session Storage vs Local Storage**

El **Local Storage**: tiene un almacenamiento de máximo 5 MB, la información almacenada aquí no se va con cada request que hacemos al servidor, esto nos ayuda a reducir la información entre cliente y servidor, la información que esta almacenada en el local storage persiste aunque cerremos la ventana de nuestro navegador, esto quiere decir que cuando volvemos a nuestro navegador la información la vamos a encontrar ahí.

El **Session Storage**: es similar al Local Storage solo que **la información está disponible por tab o por window**, esto quiere decir que **apenas cerremos un tab o un window la información deja de persistir** al igual solo la información que almacenamos en cada tab está disponible en ese mismo tab.

Las **cookies** solo tienen un almacenamiento de 4KB, lo interesante de las cookies es que si se les puede establecer un tiempo de expiración.

Para usar el Local Storage o Session Storage lo tendríamos que hacer programáticamente con Javascript.

Una de las desventajas que tienen las cookies es que por cada petición que se haga al servidor sea de imágenes, html, etc :

**Las cookies van adjuntas a la petición esto ocasiona un gran consumo de datos cada vez que se hacen las peticiones**.

Finalmente una de sus ventajas es que las cookies se pueden hacer seguras mediante un flag llamado httpOnly eso permite que la información de la [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) solo sea accedida y modificada en el servidor.

Finalmente te preguntaras: ¿Cuándo debemos usar el uno o el otro?

* Si la información no es sensible podemos almacenarla en Local Storage o en Session Storage.
* Si la información es medianamente sensible como por ejemplo: nombres de usuario o algunos términos que puedan identificar al usuario, lo más recomendado es usar el Session Storage.
* Finalmente si la información es muy sensible como contraseñas o JSON Web Tokens, lo más recomendado es almacenarlo en una [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) pero siempre teniendo en cuenta el flag httpOnly.

Vamos a ver unos ejemplo de cómo se comporta Local Storage y Session Storage en el navegador.

La manera en cómo podemos acceder a la API de WebStorage y SessionStorage es mediante la palabra sessionStorage está tienen 2 métodos, uno que es setItem donde podemos especificar con que **key** queremos almacenar nuestra información, en esté ejemplo vamos a almacenar en el key 'hello' otro string.



Muy similar al Local Storage mediante la palabra localStorage



Si nos fijamos en la aplicación en el Local Storage podemos ver que está aquí la palabra hello world, mientras que en el sessionStorage está la palabra 'hello', 'temporary world'. Si nosotros quisiéramos acceder a esta información lo podemos hacer mediante sessionStorage.getItem('hello'); es parecido con el Local Storage con localStorages.getItem('hello');.

Esta es la manera en cómo podemos usar Session Storage y Local Storage en el navegador.

**Challenge**: Discutir si fueramos a implementar autenticación que opción entre cookies y Web Storage sería ideal.

Usaría cookies para guardar la información sensible, y Web Storage para almacenar otra información no sensible. Además la web storage se crea desde el lado del cliente y no del lado de servidor. cookies definitivamente.

Otros apuntes:

[***https://github.com/JasanHdz/passportjs/blob/master/notes/notes.md***](https://github.com/JasanHdz/passportjs/blob/master/notes/notes.md)

# Configuración de Passport.js

**Passport.js** es un middleware para express que nos permite implementar estrategias de autenticación de una manera rápida y simple.

En está clase vamos a explorar la configuración inicial de passport.js. **PassportJS es un middleware para express que nos permite implementar distintas estrategias de autenticación de una manera muy fácil y simple**.

1. Primero vamos a instalar nuestras dependencias npm i passport jsonwebtoken passport-http passport-jwt

**passport-http:** para implementar la estrategia basic de passport **passport-jwt:** para implementar la estrategia de JWT

Otra librería que necesitamos instalar pero esta vez en modo desarrollo es una librería que se llama chalk que lo que permite es pintar colores en la terminal. npm i -D chalk

Lo otro que necesitamos para nuestros scripts que vamos a ejecutar más adelante y es donde precisamente vamos a utilizar nuestra librería chalk, es agregar en nuestro archivo .env y .env.example las siguientes variables de entorno:

Por defecto vamos a agregar los siguientes valores, igual ustedes pueden agregar algún otro.

**.env**

# USERS

DEFAULT\_ADMIN\_PASSWORD=root

DEFAULT\_USER\_PASSWORD=secret

Estás variables de entorno van a ser necesarias para cuando estemos insertando nuestros usuarios iniciales, no estemos definiendo un password por defecto en el código, si no que sea un password diferente para cada aplicación que estemos creando, en este caso vamos a generar un password por defecto para el administrador y un password por defecto para estos usuarios iniciales.

Lo otro que vamos a necesitar es definir el **secret** que vamos a necesitar para firmar nuestros JWT, **esté secret se debería sacar de una página que se llama**: [keygen.io](https://keygen.io/). Está página lo que nos permite es definir un key de base 264 bits, nosotros vamos a buscar nuestro string de 256 bits, lo copiamos y en nuestro código lo agregamos.

# AUTH

AUTH\_JWT\_SECRET=FC6xnhTSEKlt0mNHjw4fucpzqg2e5M9B

Otra cosa que vamos a definir es cuales van a ser sus API\_KEYS, recuerden que nuestros API\_KEY es lo que nos va a permitir definir que cuando haya un login con alguno de nuestro clientes, es decir el **Render-Server o nuestro cliente Administrativo**, le otorgue los permisos. En este caso esos API\_KEYS van a ser generados aleatoriamente apartir de nuestros scripts, teniendo en cuenta estas variables de entorno nos aseguramos de tenerlas en nuestro archivo .env.

# API KEY

PUBLIC\_API\_KEY\_TOKEN=

ADMIN\_API\_KEY\_TOKEN=

Nos aseguramos que nuestro archivo .env.example no tenga ningún valor porque esto es la referencia que va a usar el desarrollador para llenar su archivo .env, procedemos a copiar está misma configuración en nuestro archivo de configuración de variables de entorno.

**Config/index.js**

defaultAdminPassword: process.env.DEFAULT\_ADMIN\_PASSWORD,

defaultUserPassword: process.env.DEFAULT\_USER\_PASSWORD,

authJwtSecret: process.env.AUTH\_JWT\_SECRET,

publicApiKeyToken: process.env.PUBLIC\_API\_KEY\_TOKEN,

adminApiKeyToken: process.env.ADMIN\_API\_KEY\_TOKEN

La estrategia que ocupe en VS code es agregar el cursos al final de la linea, copie y pegue y use una librería para convertir los strings a camel Case.

Teniendo en cuenta estos archivo de configuración lo que voy a hacer es copiar unos scripts que yo cree previamente, los voy a pegar en mi API Server.

Script para agregar películas usando nuestros mocks:

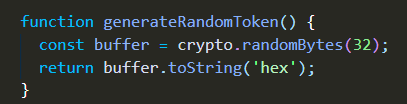


Los API\_KEYS los necesitamos en nuestras variables de entorno, por lo que una vez creados nuestros api keys en la base de datos de mongo **(en este caso)**, procedemos a copiarlos e incluirlos en nuestro archivo .env

De esta manera ya hemos creado una serie de películas, unos usuarios y los **API TOKENS** necesarios donde vamos a hacer uso de nuestra estrategia de autenticación con passport.js

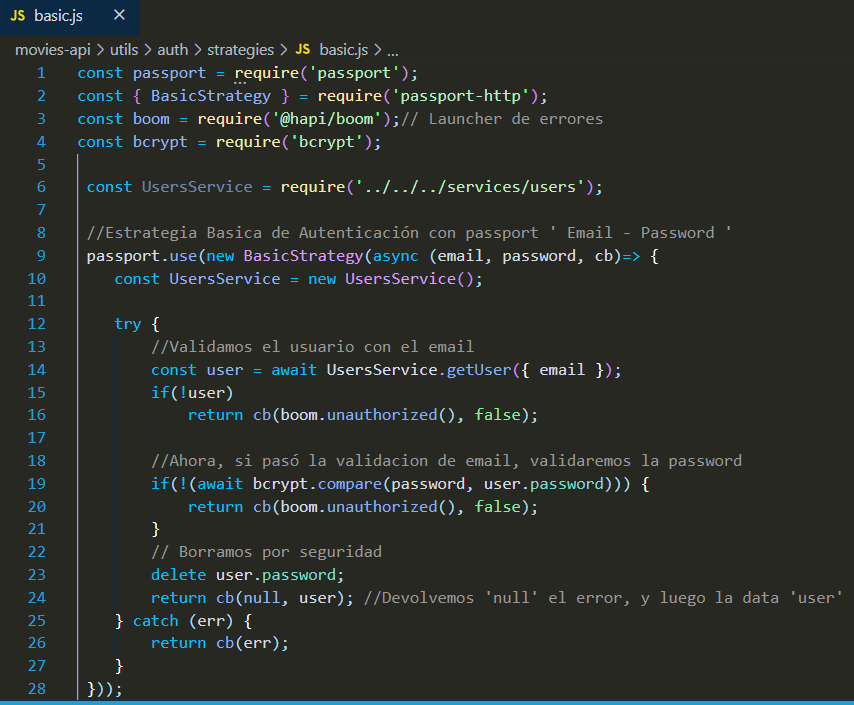
Generar un token:





## **Implementación de BasicStrategy con Passport.js**

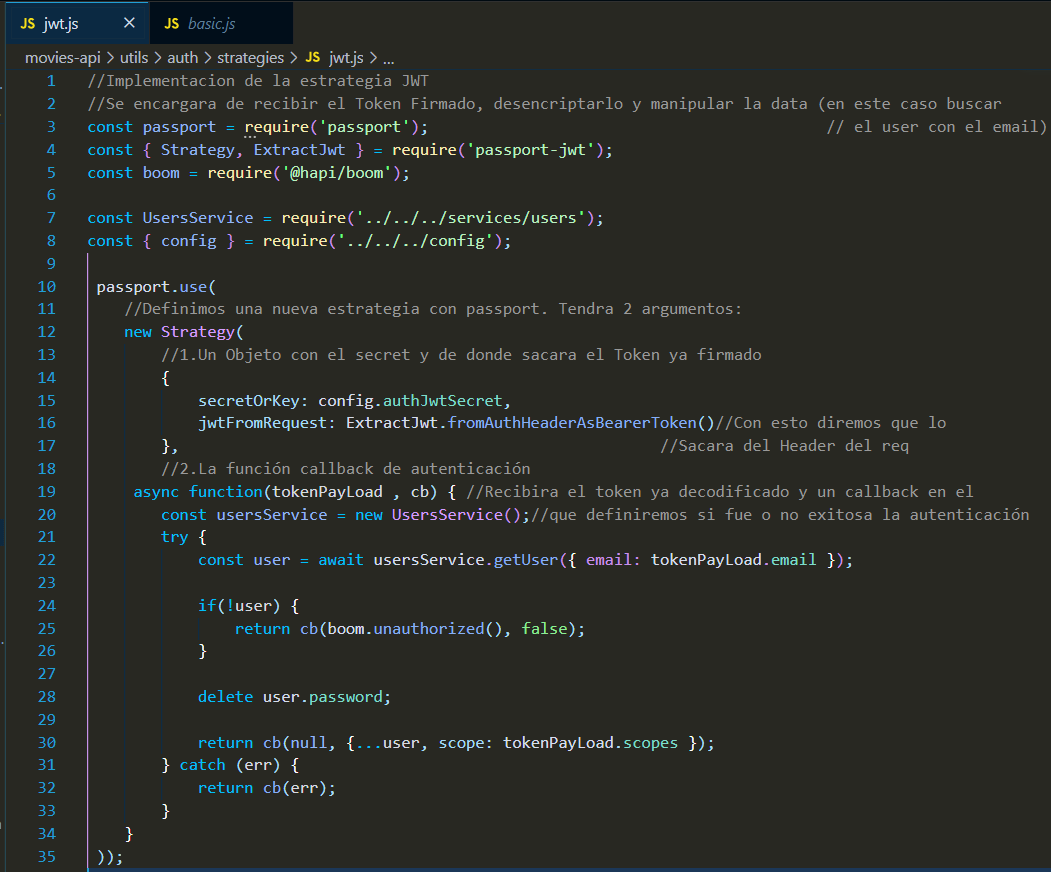
En esta clase aprenderás como implementar estrategias de autenticación haciendo uso de Passport.js. Las estrategias de autenticación nos sirven para determinar cómo nos vamos a autenticar, haciendo uso de estas estrategias en las diferentes rutas y así definir de donde saldrá el usuario que vamos a usar de ahí en adelante, vamos a verlo como se hace en el código.

1. Vamos a crear una nueva carpeta dentro de utils llamada auth/strategies/basic.js, aquí es donde vamos a implementar nuestra estrategia de tipo Basic.

Con esto ya tenemos implementada nuestra estrategia Basic, esto quiere decir que cuando se la agreguemos como middleware a una ruta, si hacen una petición de una autenticación basic, va a poder extraer a través de email y password, el usuario y apartir de ahí definir quien está autenticado en nuestra aplicación.

# Implementación de JWT con Passport.js

Vamos a implementar ahora la estrategia de JWT donde recibiremos un JWT y apartir del token decodificado buscaremos al usuario.

1. Creamos un nuevo archivo llamado jwt.js dentro de nuestra carpeta de strategies

# En profundidad..

En esta ocasión vamos a implementar la estrategia de JWT, en donde recibiremos en Json web Token y a partir de **Token** decodificado buscaremos y autenticaremos el usuario.

* Lo primero es asegurarnos que nuestro **AUTH\_JWT\_SECRET** este dentro de nuestro archivo **.env**, recordemos que podemos generarlo [aquí](https://keygen.io/) en la opción de **WEP 256-bit Key**
* Lo agregamos también a nuestro archivo de configuración.

authJwtSecret: process.env.AUTH\_JWT\_SECRET,

* Importamos dentro de nuestro **utils/auth/jwt.js** lo siguiente

const passport = require('passport');

const { Strategy, ExtractJwt } = require('passport-jwt');

const boom = require('@hapi/boom');

const UsersService = require('../../services/users');

const { config } = require('../../config/index');

Comenzamos a definir nuestra estrategia, la cual tiene  dos parámetros

* 1. La estrategia a usar
  2. la función callback de autenticación

passport.use(

    new Strategy(

{},

    async() => {}

));

Esta estrategia recibe como constructor dos parámetros:

1. **secretOrKey:** que es el mismo **AUTH\_JWT\_SECRET** que tenemos ahora en nuestra configuración
2. **jwtFromRequest:** que es el lugar de donde se espera obtener el JWT es aquí donde tenemos que usar **ExtractJwt** para definir que nuestro token vendrá en el Header como un **Bearer Token**

    new Strategy({

        secretOrKey: config.authJwtSecret,

        jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()

    })

La función callback va  a recibir dos parámetros:

    async (tokenPayload, callback) =>{}

1. el payload del **token** ya decodificado y
2. otro callback que es donde vamos a definir finalmente si la autenticación fue exitosa o no

### **Dentro de nuestra función callback**

* Comenzamos implementando los servicios del usuario para poder conectarnos y consultar la base de datos

    const usersService = new UsersService();

* Abrimos nuestro **try** e intentamos traer nuestro usuario con el email que sacamos del tokenPayload

    try {

      const user = await usersService.getUsers({ email: tokenPayload.email })

    }

* si **no** obtenemos el usuario con ese email respondemos con un **boom.unauthorized**

    try {

      const user = await usersService.getUsers({ email: tokenPayload.email })

      if(!user) {

        return callback(boom.unauthorized(), false);

    }

* Si efectivamente obtenemos el usuario eliminamos su contraseña ya que no la necesitamos más dentro del flujo de la aplicación.

      delete user.password;

* Por último devolvemos el usuario en adición con los **scopes** del **token**

**Scopes:** Los permisos con los que contará el usuario, por ejemplo si el usuario es administrador tendrá estos scopes:

    const adminScopes = [

      'signin:auth',

      'signup:auth',

      'read:movies',

      'create:movies',

      'update:movies',

      'delete:movies',

      'read:user-movies',

      'create:user-movies',

      'delete:user-movies'

    ];

* Si caemos en el **catch** vamos a retornar el **callback** con el error correspondiente.

catch (error) {

      return callback(error)

    }

### **Toda nuestra estrategia quedaría así:**

passport.use(

  new Strategy({

    secretOrKey: config.authJwtSecret,

    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()

  }),

  async (tokenPayload, callback) => {

    const usersService = new UsersService();

    try {

      const user = await usersService.getUsers({ email: tokenPayload.email })

      if(!user) {

        return callback(boom.unauthorized(), false);

      }

      delete user.password;

      callback(null, {...user, scopes: tokenPayload.scopes});

    } catch (error) {

      return callback(error)

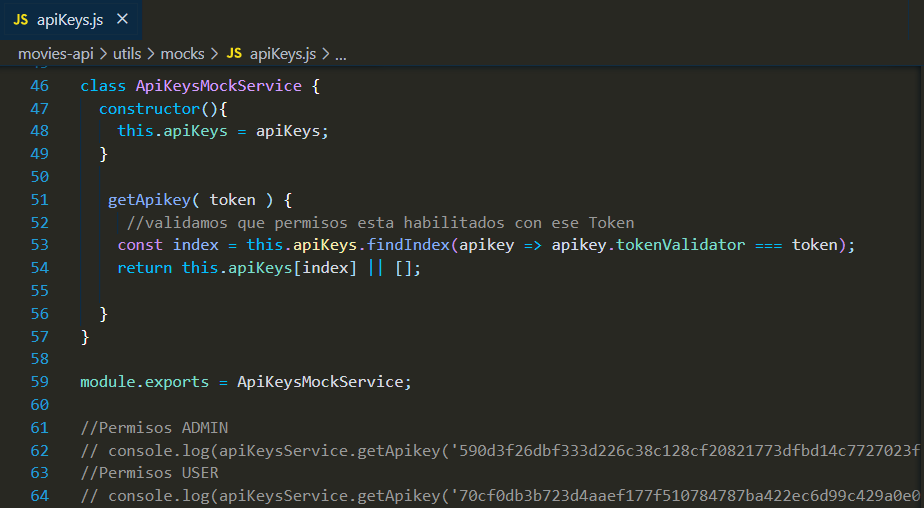
    }

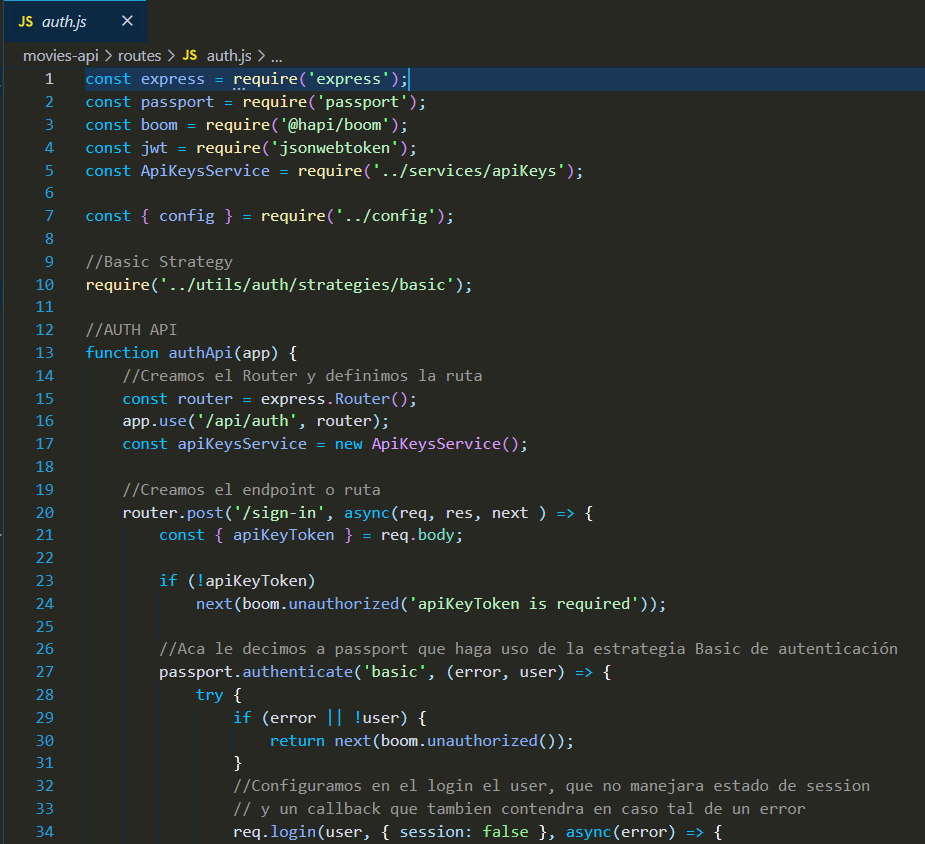
  }

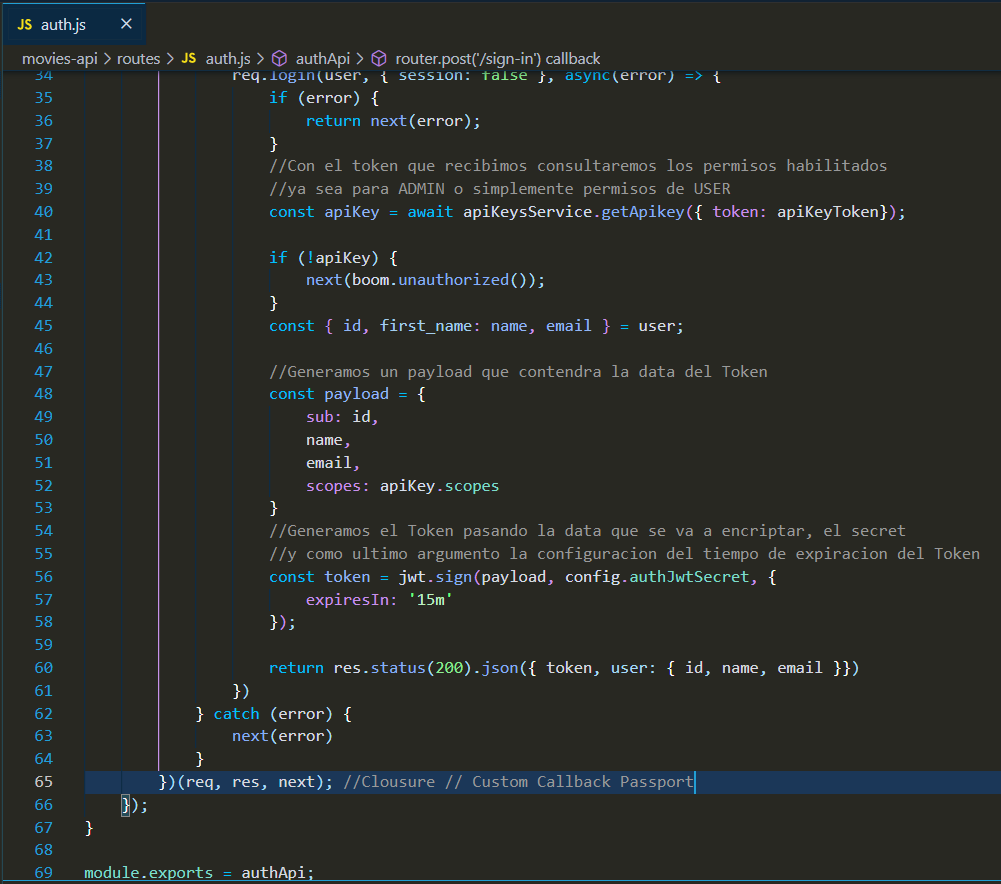
);

# Implementación de nuestro Sign-in

Ya que hemos implementado nuestras estrategias, ahora vamos a implementar la ruta de Sign In.

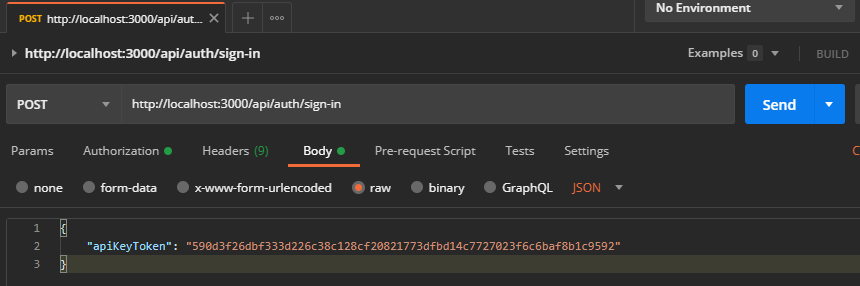
1. Antes de implementar nuestra ruta debemos crear un nuevo servicio, llamado apiKeys.js, esté servicio nos va a permitir que apartir de un API-Key-Token enviada desde el cliente podamos obtener los permisos según corresponda para luego firmar un JWT con los scopes correspondientes de acuerdo al API Token que nosotros enviemos.
2. Ahora creamos una nueva ruta que se va a llamar auth.js en la que vamos a manejar todo lo relacionado a la autenticación y del Login.

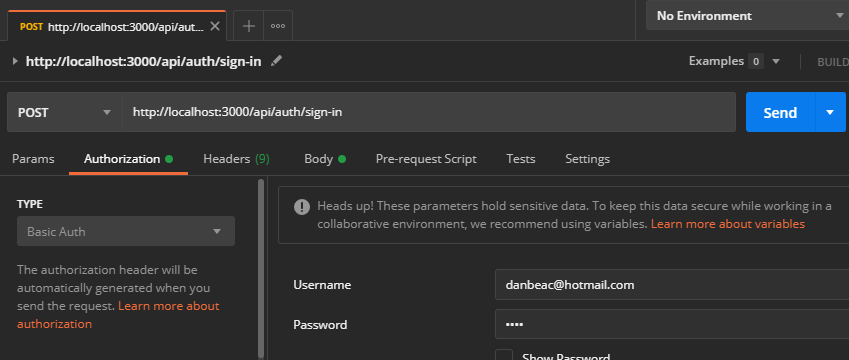


Ahora nos vamos a nuestro archivo index.js y agregamos la nueva ruta que acabamos de crear.

1. Levantamos el servidor y toca ir a probar nuestra ruta de signIn usando Postman.

Vamos a hacer un request de tipo POST a la ruta localhost:3000/api/auth/sign-in/ en el vamos a hacer una Authorization de tipo Basic Auth. Y en el body tenemos que enviar el API\_TOKEN el cual copiamos de nuestro archivo .env.





<http://www.passportjs.org/docs/login/>

**Para tener en cuenta:**

En este caso estamos usando un Clousure en nuestro **passport.authenticate, (req, res, next)** es decir que va recordar las elementos que como *middleware* vamos a tener que usar dentro de nuestro authenticate.

Authenticate normalmente se usa como un middleware y por defecto express le inyectará req, res, y next pero cuando termina de hacer todo el proceso tiene un comportamiento por default.

Cuando importamos nuestro ***Basic Strategy*** del ***require('../utils/auth/strategies/basic');***

Estamos llamando simplemente todo el código escrito allí, implementando la estrategia, en ese archivo que va quedar configurada la ruta de Autenticación.

# Protegiendo nuestras rutas con Passport.js

En esté modulo vamos a aprender cómo podemos proteger nuestras rutas de la API, haciendo uso de Passport.js.

Lo que vamos a hacer es que vamos a proteger nuestras rutas de movies y userMovies. Para eso vamos a importar nuestra estrategia de JWT, es bastante fácil poder proteger nuestras rutas, lo único que tenemos que hacer es hacer uso de passport y hacemos passport.autenticate en el vamos a definir que la estrategia que vamos a ocupar es jwt y definimos que la sessión va a ser false

nuestra ruta movies quedaría de la siguiente manera:

Cuando importamos nuestra Strategy de JWT

require('../utils/auth/strategies/jwt');

Es donde estamos recibiendo un JWT que viene del authorization header, vamos a obtener el email del usuario y apartir del vamos a buscarlo y devolverlo, en esta ocasión no hay necesidad de implementar un Custom Callback, entonces lo único que hay que hacer es llamar a passport.authenticate definir la estrategia y esto funciona como un middleware. Apartir de ahí lo que vamos a hacer es agregar está misma linea a los otros Endpoints.

Con esto ya tenemos protegidas todas nuestras rutas, esto quiere decir que la única manera de acceder a ellas es si tenemos un JWT valido.

Vamos a hacer lo mismo para nuestras rutas de userMovies

// JWT Strategy

require('../utils/auth/strategies/jwt');

Después copiamos la misma linea de:

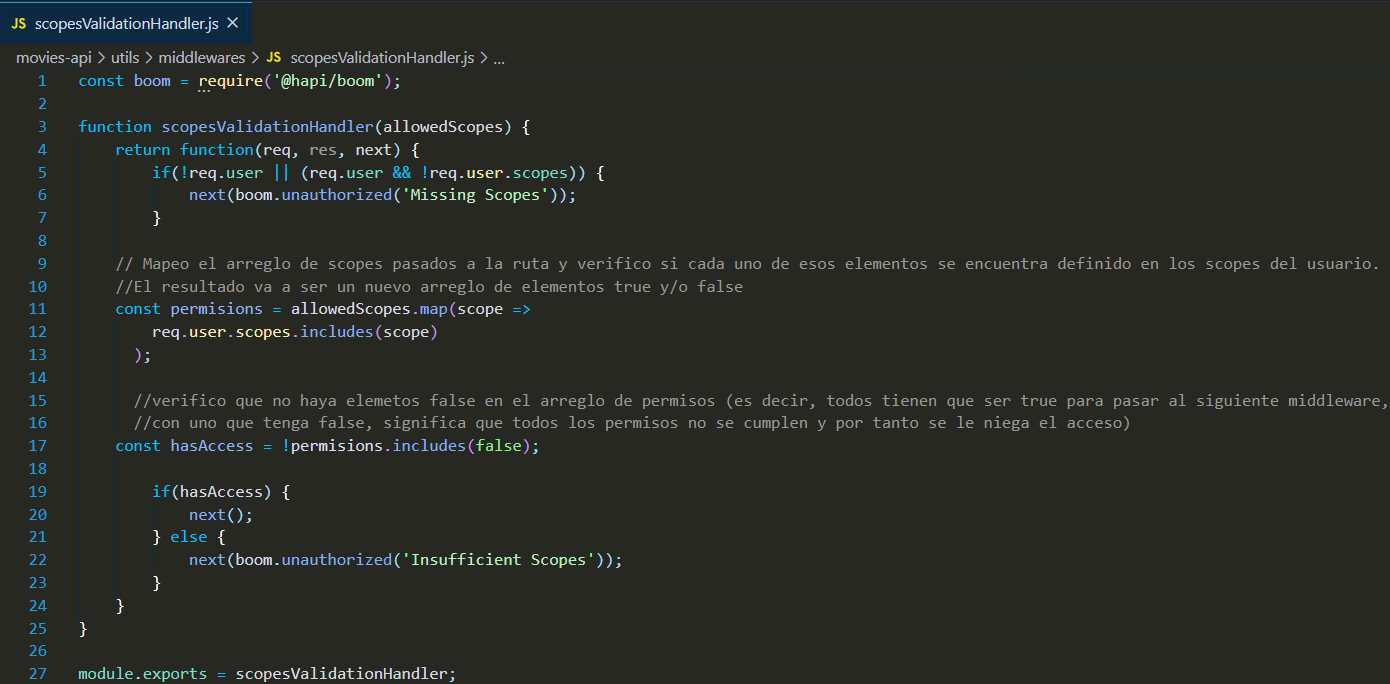
passport.authenticate('jwt', {session: false}),

Que nos permite proteger las otras rutas.

Ahora levantamos nuestro servidor en modo desarrollo y vamos a postman. Si ahora hacemos un GET movies desde postman ahora nos dice que no estamos autorizados, ahora tenemos que hacer sign-in para que nos devuelva un token y lo que debemos de hacer es que cada vez que queramos llamar nuestras rutas que ya están protegidas, **debemos de ir al tab de autorización**, definir que vamos a autorizarnos con **Bearer Token** y agregar nuestro **Access Token** en el campo Token, de esta manera si ahora llamo las rutas de las peliculas ahora si nos devuelve datos, y lo mismo pasa con las demás rutas a las que autenticamos usando passport.js

Con esto hemos cerrado prácticamente todo el ciclo de autenticación, ya tenemos todo el flujo de hacer sign-in y tenemos todo el flujo de hacer sign-up y finalmente tenemos protegidas nuestras rutas de la API.

# Middleware para el manejo de scopes

En esta clase crearemos un middleware que nos permitirá validar los scopes de las rutas.

Ahora debo definir los scopes de las diferentes rutas, después de que verifico con el **middleware** de que está autenticado, entonces indico cuál es el scope necesario, para el de getMovies el scopes necesario es [read:movies], es decir si el usuario tiene el scope de [read:movies] podrá leer las peliculas, para el getMovieId necesita el mismo scope, para el createMovie necesita el scope de ['create:movies'], entonces fíjense que si tenemos unos scopes públicos no vamos a poder consumir nuestra ruta de crear peliculas, luego del put o updateMovies necesitaríamos el scope de ['update:movies'] y para el endpoint de deleteMovies necesitaríamos el scope de ['deleted:movies'].

Tanto passport.authenticate, scopesValidationHandler y validationHandler son middlewares.

**El propósito de los middlewares es filtrar e intervenir el código** request que viene **desde la ruta o endpoint hasta que ya hacemos su funcionalidad** y tratamos los suficientes datos, entonces esto nos permite en este caso: validar la autenticación, después validar que tenga los permisos necesarios, y por último validar que los datos esté correctos.

Haremos lo mismo para nuestros datos de userMovies.

ejemplos de la implementación:

scopesValidationHandler(['read:movies']),

scopesValidationHandler(['create:movies']),

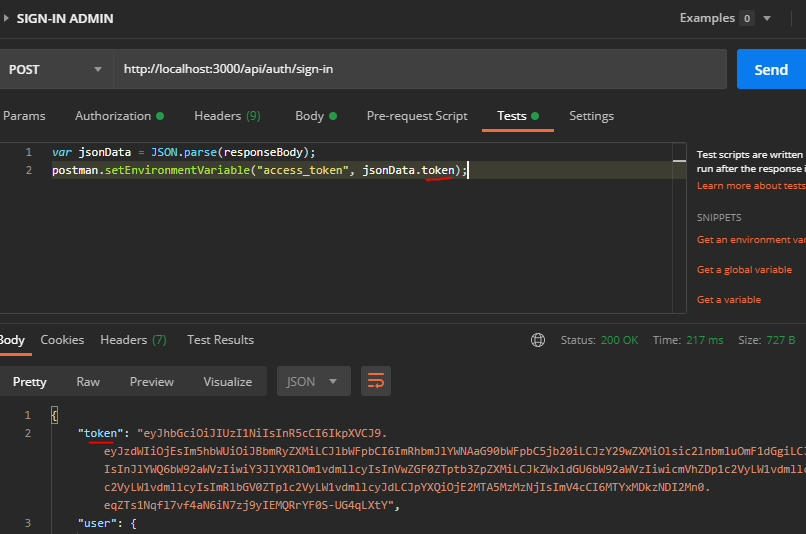
scopesValidationHandler(['update:movies']),

scopesValidationHandler(['delete:movies']),

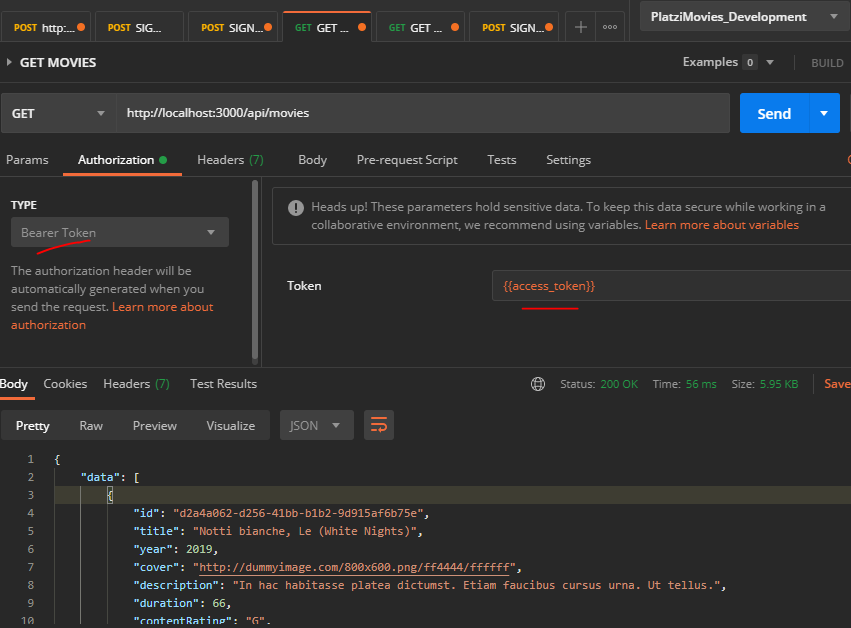
Ahora levantamos el servidor en modo desarrollo y nos dirigiremos a postman a verificar que esto esté funcionando correctamente. Antes de hacer request a nuestras diferentes rutas y verificar que nuestras variables de entorno para verificar que estemos ocupado la variable de entorno correcta.

**Ahora para validar en Postman:**

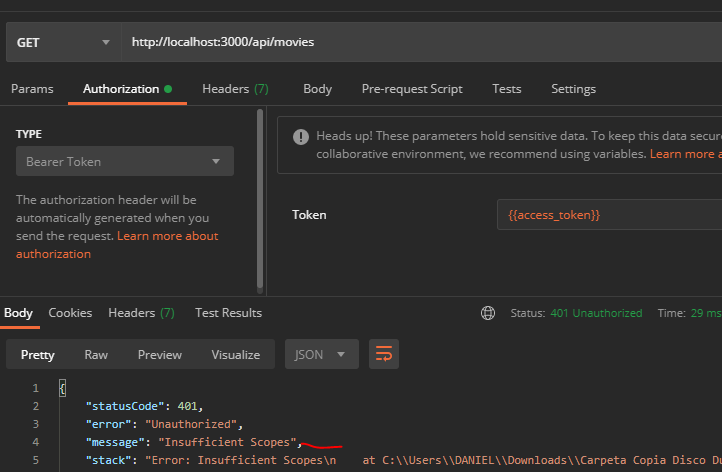
Ejecutamos primero nuestro Sign-In con un test que nos almacene en una variable de entorno el token que nos devuelve firmado.



Para luego hacer los request que están protegidos (ej. get/movies) con los permisos que se almacenan en el Token. Llamamos ahora esta variable de entorno en la autenticación como un Bearer Token.



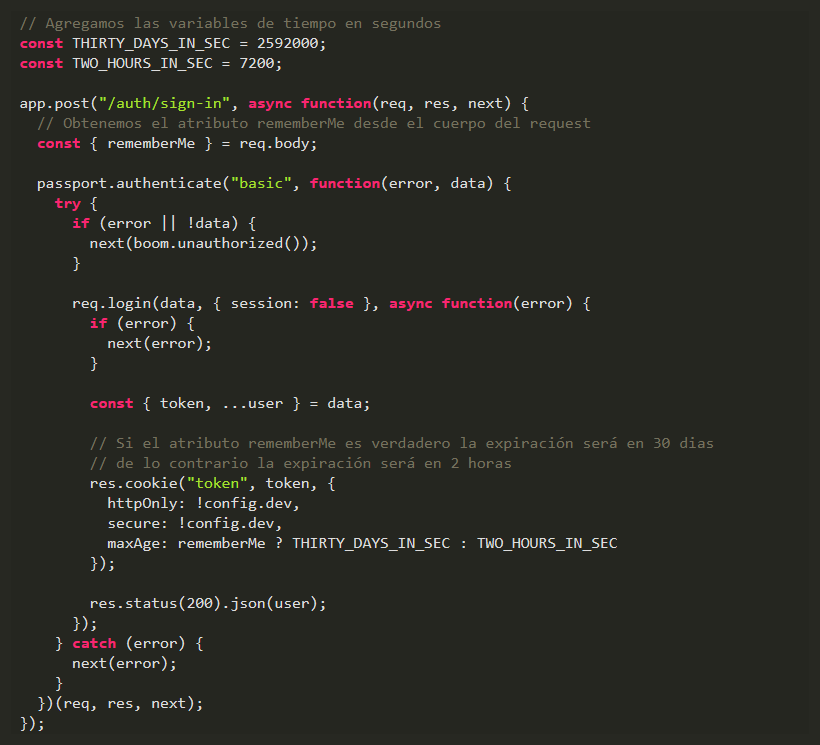
En caso de no tener el permiso.



# Implementando recordar sesión.

Generalmente cuando queremos implementar la opción de recordar sesión para Express mediante passport, lo que hacemos es extender la expiración de la Cookie.

En nuestra ruta de sign-in de nuestro render server hacemos las siguientes modificaciones:



# Basic Strategy vs Authorization Bearer

# 

**Authorization Basic**: Se utiliza un usuario y contraseña y se usa generalmente para el primer acceso a un sistema sign-in y este tipo de autorización nos puede devolver un **JSON WEB TOKEN**.

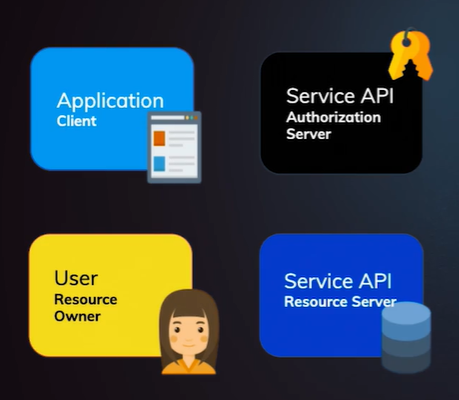
**Authorization Bearer**: Se utiliza un JWT con el cual el sistema valida nuestra identidad y los niveles de autorización que tenemos dentro del sistema, estos datos se ponen el el payload del JWT.

Como se vio en el curso estos dos tipos de autorización funcionan bien juntos, primero utilizas Basic para obtener tu token y después utilizas el token para no depender de una sesión del lado del servidor y poder manejar las rutas y los accesos con el token.

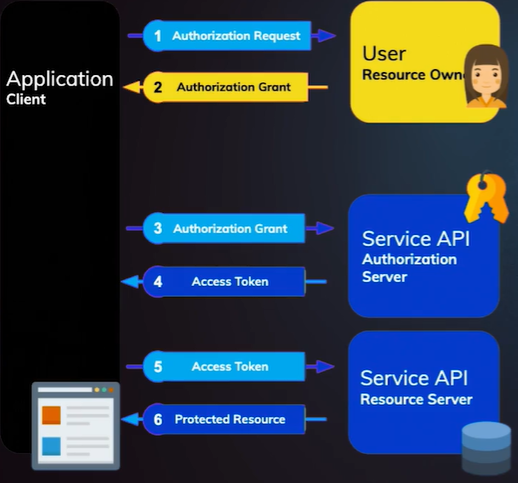
## **¿Qué es OAuth 2.0?**

Estos estándares nos permiten implementar cosas como autenticación con redes sociales.

* [OAuth 2.0](https://github.com/JasanHdz/passportjs/blob/master/notes): Es un estándar de la industria que nos permite implementar autorización.

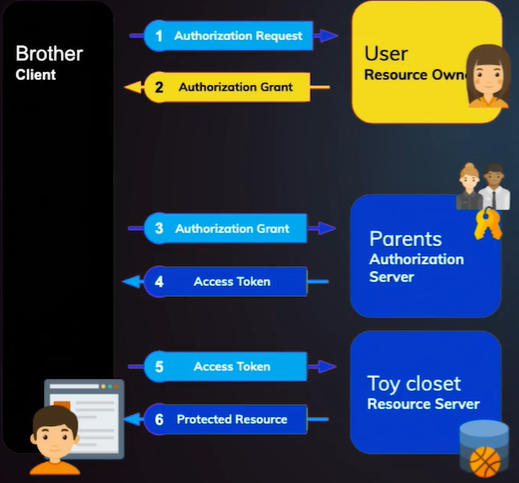
[](https://github.com/JasanHdz/passportjs/blob/master/notes/assets/oauth.png)

Lo más importante para aprender a implementar OAuth es entender cuáles son los roles envueltos en los flujos, en este caso el **primer rol** sería el usuario **Resource Owner**, en este caso podrías ser tú, luego tenemos el **Resource Server** que sería una API donde están tus recursos o tus datos, luego tenemos el application o el **Client** quien es el que va intentar acceder a estos recursos en nombre del usuario y finalmente tenemos el **Authorization Server** qué es quien se encarga de verificar la identidad del usuario.

[](https://github.com/JasanHdz/passportjs/blob/master/notes/assets/appclient-oauth.png)

El flujo empieza cuando la aplicación quiere hacer una **Authorization Request**, entonces tu como usuario tienes que permitirle a la aplicación acceder a tus recursos, esto lo hace mediante una **Authorization Grant**. La aplicación con esté Authorization Grant va al **Authorization Server**, el verifica que los datos son correctos y te crea un Access Token, el **Access Token** puede ser un **token** como cualquiera o podría ser un JWT, apartir de ahí la aplicación con ese Access Token puede hacer cualquier petición y obtener recursos en tu nombre, entonces el **Resource Server** que sería la **API**, lo que haría sería devolver los recursos protegidos, gracias a que tú le enviaste un Access Token.

Aquí tenemos un ejemplo del mundo real:

[](https://github.com/JasanHdz/passportjs/blob/master/notes/assets/oauth-exmaple.png)

"supongamos que el cliente es tu hermano, tu hermano quiere acceder a la pelota que está en el closet, el closet sería el ***Resource Server***, pero tú no te encuentras en casa, sin embargo tus padres se encuentran en casa, entonces el hermano lo que haría sería pedirte una autorización, tú lo que podrías enviar como ***Authorization Grant*** sería una carta donde tu hermano está permitido en obtener la pelota, tu hermano con esa carta lo que haría sería ir con el ***Authorization Server*** que serían tus padres, tus padres verifican que sea una carta con tu letra, y les darían las llaves en este caso sería el ***Access Token*** ahora tu hermano con las llaves puede acceder al closet y obtener la pelota".

## **¿Qué es OpenID Connect?**

Open ID Connect: **es una capa de autenticación que funciona sobre la capa de Oauth**. Lo que sucedió es que las compañías que estaban implementando Oauth para autenticar, estaban teniendo problemas de seguridad. **Facebook** hace tiempo tuvo problemas donde podían suplantar la identidad porque estaban usando Access Token para hacer todo el proceso de Autenticación, entonces **Facebook tuvo que hacer unas soluciones sobre esa capa de Oauth** y lo que pasó es que las otras empresas entonces también tuvieron que empezar a implementar esos fixes.

OpenID Connect se trata de generar uno estándar así no todas las personas no tienen que hacer su propia versión de autenticación sobre Oauth.

Las diferencias que tiene sobre Oauth es que **los access token se usan exclusivamente para los llamados a la API es decir para obtener los recursos**, y entra un concepto llamado **IdToken**, es un nuevo token que nos permite verificar si el usuario está autenticado y nos permite también obtener la información del usuario, básicamente Open Id Connect también define unos claims y unos scopes definidos para este **IdToken** y debemos también implementar un endpoint llamado **userInfo** donde enviamos el idToken y podemos obtener la información del usuario.

**Open Id Connect** también define como debemos hacer uso del manejo de sesión es decir cómo se debe hacer logout como se implementan cosas como single signInOut, etc.

El flujo es más o menos el siguiente: se hace un request a /authorizate y esté nos genera un IdToken, este IdToken debe tener definidos los scopes de openId y profile, con este IdToken entonces ya sabemos que el usuario está autenticado y finalmente podemos hacer un request a /user-info y obtener la información del usuario.

# ¿Qué es OAuth 2.0 y OpenID Connect?

**OAuth 2.0** es un estándar de la industria que nos permite implementar autorización.

**OpenID Connect** es una capa de autenticación que funciona sobre OAuth con métodos ya construidos.

# Cómo crear un proyecto en Google API para hacer autenticación con 0Auth 2.0.

Con el fin de poder usar Google como método de autenticación es necesario crear un nuevo proyecto dentro de la consola de desarrolladores de Google.

1. Nos dirigimos a <https://console.developers.google.com> y nos autenticamos con nuestra cuenta de Google.
2. En la parte superior izquierda seleccionamos la organización que queremos usar (Debe haber una por defecto) y hacemos click en **Create Project**.
3. Luego nos vamos al sidebar izquierdo y seleccionamos **Credentials > Create Credentials > OAuth client ID**
4. Nos aseguramos de elegir Web Application como el tipo de aplicación.
5. Luego establecemos el nombre del cliente que en nuestro caso será SSR Server, el **Authorized JavaScript origins**: http://localhost:8000 y el **Authorized Redirect URLs** http://localhost:8000/auth/google-oauth/callback. Cuando hagamos despliegue de nuestra aplicación lo ideal es crear otro cliente y remplazar las URLs por las URLs correspondientes de producción.
6. El **Application Name** del **Consent Screen** será Platzi Videos.
7. Al finalizar la creación copiamos el **Client ID** y **Client secret** que serán usados como GOOGLE\_CLIENT\_ID y GOOGLE\_CLIENT\_SECRET respectivamente.

**Useful Information:**

<https://heynode.com/tutorial/process-user-login-form-expressjs>

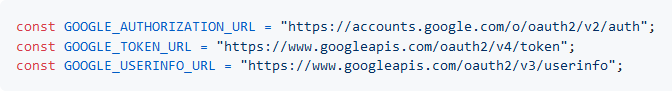
# Implementando 0Auth2.0 con Google.

Recuerda que de la lectura anterior debemos obtener el GOOGLE\_CLIENT\_ID y GOOGLE\_CLIENT\_SECRET.

Estos valores los vamos a poner en nuestro documento de variables de entorno, aparte de ahí también tenemos que agregarlos en nuestro archivo de configuración.

Lo otro que necesitamos hacer es instalar nuestra librería: passport-oauth, luego en nuestro Render Server, vamos a crear una nueva estrategia dentro de nuestro directorio ssr-server/utils/auth/statregies llamado oauth.js, aquí vamos a determinar nuestra estrategia de Oauth2.0.

En el archivo vamos a copiar unas URLS que hacen parte del OAuth de Google, estás url son las siguientes:



**To copy:**

const GOOGLE\_AUTHORIZATION\_URL = "https://accounts.google.com/o/oauth2/v2/auth";

const GOOGLE\_TOKEN\_URL = "https://www.googleapis.com/oauth2/v4/token";

const GOOGLE\_USERINFO\_URL = "https://www.googleapis.com/oauth2/v3/userinfo";

¿Como hacer las configuraciones de OpenIDConnect y OAuth2 para Google?

* <https://developers.google.com/identity/protocols/OpenIDConnect>
* <https://developers.google.com/identity/protocols/OAuth2>
* <https://developers.google.com/identity/protocols/OAuth2WebServer>

Estas urls de nuestro documento las sacamos de los enlaces de arriba.

Ahora lo que vamos a hacer es generar nuestra estrategia de Oauth.

# Revisar el código de realizado en *ssr-server/index.js* y *ssr-server/../utils/strategies/outh.js*

# Implementando Sign Provider en nuestra API.

En esté modulo vamos a implementar nuestra ruta para que cualquier proveedor tercero como por ejemplo la autenticación con Google puede hacer uso de nuestra API.

Lo que vamos a hacer es que en nuestros servicios de usuarios, esto es en: movies-api/services/ vamos a crear un nuevo método que se va a llamar getOrCreateUser. Esté método recibe un usuario y apartir de este método vamos a determinar que si no existe lo cree y si existe lo traiga.

Para poder tener nuestra ruta debemos hacer una modificación en nuestro schema de users.js.

# 

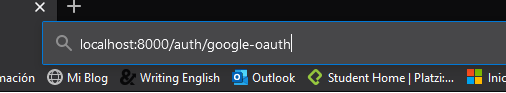
Porque ahora el userSchema lo necesitamos solo para hacer un nuevo schema que se va ha llamar createProviderUserSchema

Con esto ya podemos implementar nuestra ruta, y vamos a la ruta de autenticación. Necesitamos agregar nuestro nuevo createProviderUserSchema y al final vamos a agregar una nueva ruta post.

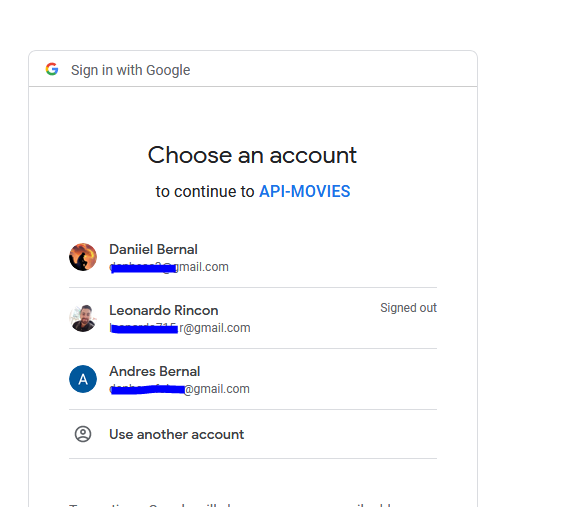


Con esto ya tenemos implementando toda nuestra ruta de Sign Provider, ahora vamos a revisar que todo esté funcionando correctamente, primero levantamos nuestros 2 Servidores, el de API y el del SSR y dirigirnos a la URL http://localhost:8000/auth/google-oauth, hacemos la prueba, después nos pide acceder a una cuenta de Google, si todo sale bien se va ha ir el callback y me va ha devolver el usuario, no solo eso si no que en nuestras cookies de la aplicación vamos a tener la [cookie](https://es.wikipedia.org/wiki/Cookie_(inform%C3%A1tica)) con el token.

Exploremos si el JWT es el token que esperamos, nos vamos a [jwt.io](https://github.com/JasanHdz/passportjs/blob/master/notes/jwt.io), copiamos el token para que sea decodificado y efectivamente acá tenemos nuestro usuario, email y los scopes que le definimos a nuestro usuario. Con esto hemos terminado la implementación haciendo la autenticación con Google en nuestro render server.



Esto me va a redirigir allí:



Y luego nos retornara un Json con la data del User.

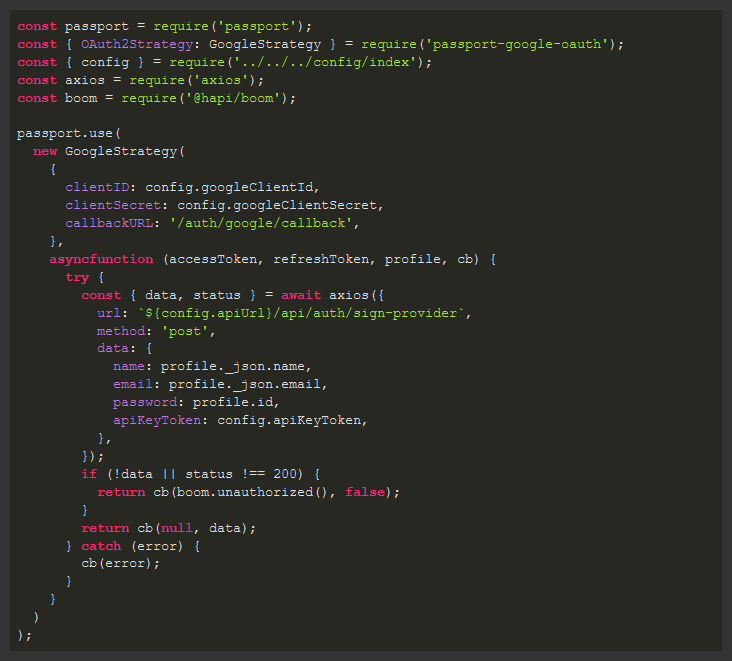
# Autenticación con Google usando OpenID Connect.

En el curso vimos cómo se podía implementar Google Authentication usando la estrategia de OAuth directamente. En esta lectura vamos a explorar como hacer la autenticación usando la estrategia directa de Google.

Lo primero es instalar nuestras dependencias:



Luego creamos una nueva estrategia llamada google dentro de utils/auth/strategies/google.js:



Teniendo nuestra estrategia de **Google** ya podemos agregar las dos nuevas rutas de autenticación.



Con esto tenemos nuestra implementación de autenticación con Google pero mucho más sencilla.

# No olvidar tener configuradas las entradas en Google API:

# 

# Cómo crear una cuenta de desarrollador con Twitter

Con el fin de poder usar Twitter como método de autenticación es necesario crear una cuenta de desarrollador de Twitter. Es necesario postularse para que la cuenta de desarrollador pueda ser utilizada y suele tomar hasta 24 horas de aprobación. Para aplicar a la cuenta es necesario hacer dirigirse a <https://developer.twitter.com/en/apply-for-access.html> y hacer click en el botón Apply for a developer account.

Cuando la cuenta ha sido aprobada, entonces procederemos a crear una aplicación para usarla en nuestro método de autenticación, para ellos debemos:

1. Nos dirigimos a la lista de nuestras aplicaciones en <https://developer.twitter.com/en/apps>
2. Allí creamos una nueva app haciendo click en el botón **Create an app**.
3. Llenamos los correspondientes campos como **App Name** y **Website URL**.
4. Las URLs de **Terms of Service URL** y **Privacy policy URL** son necesarias para poder solicitar el email del usuario.
5. Como callback URL podemos usar http://localhost:8000/auth/twitter/callback. Lo ideal es que cuando salimos a producción creamos una aplicación diferente y esta vez usaremos la URL de producción.
6. Marcamos como **Enabled** que nuestra app va a ser usada para **Sign in with Twitter**.
7. Hacemos click en **Create**, accedemos a los detalles de la app creada y en el tab de **Permissions**, y luego en **Additional permissions** marcamos **Request email address from users** y guardamos.
8. Nos vamos al tab de **Keys and tokens** y copiamos los **Consumer API Keys** que son los que usaremos como TWITTER\_CONSUMER\_KEY y TWITTER\_CONSUMER\_SECRET respectivamente.

# Autenticación con Twitter.

De la lectura anterior debemos copiar el consumer\_key y el consumer\_secret de twitter, es importante copiarlos en nuestro archivo .env y además también debemos crear un string de session, ya que esté middleware de passport con twitter necesita tener una session activa, a mi me gusta usar una página que se llama [keygen.io](https://github.com/JasanHdz/passportjs/blob/master/notes/keygen.io) donde podemos copiar un SHA KEY de 256 bits, lo copiamos y lo incluimos en nuestro secret de session también es importante leerlos desde nuestro archivo de configuración para después poder consumirlos de nuestra estrategia.

Ahora vamos a crear una nueva estrategia que se va a llamar twitter, creamos la estrategia twitter.js.

# 

Con nuestra estrategia implementada ahora podemos ir a nuestro servidor y lo primero que debemos incluir es unas librerías que necesitamos express-session, luego entonces haríamos en el app.use una implementación de la session y en este caso el secret que nos va ha codificar la session sería el config.sessionSecret.

Lo otro que tenemos que hacer, es hacer uso de passport.initialize() para que inicialice la session y luego de passport-session, debemos tener todo esto requerido porque está librería de twitter requiere que tengamos una session activa. Ahora lo que tenemos que hacer es incluir nuestra estrategia de twitter y como hicimos la autenticación con google es simplemente implementar las rutas de autenticación.

# 

# Ahora lo que tenemos que hacer es levantar nuestros 2 servidores y hacer las pruebas necesarias, si aún no estamos autorizados por twitter tenemos que esperar a que nos autorize para que funcione correctamente.

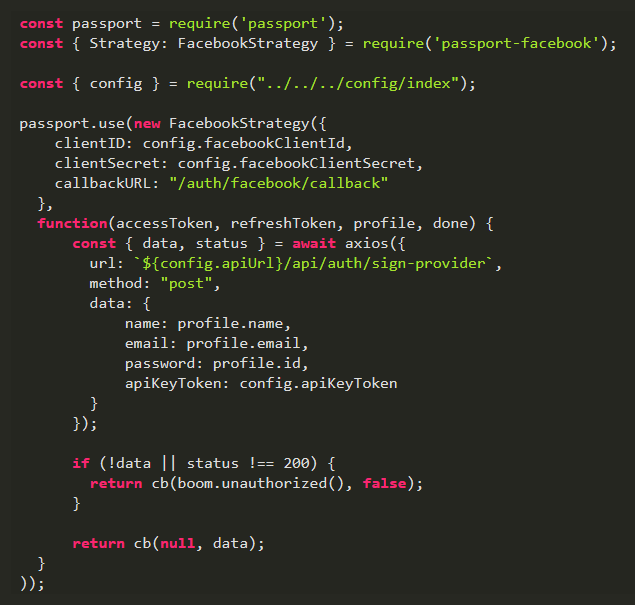
# Autenticación con Facebook.

A continuación veremos cómo podemos implementar la autenticación haciendo uso de la estrategia para Facebook. Es importante primero tener una cuenta de Facebook para desarrolladores: <https://developers.facebook.com/> y crear una app/project como lo hemos hecho anteriormente para Google y Twitter. Podemos seguir las instrucciones aquí: <https://developers.facebook.com/docs/apps>, para obtener nuestros FACEBOOK\_CLIENT\_ID y FACEBOOK\_CLIENT\_SECRET respectivamente.

Lo primero es instalar nuestras dependencias:



Luego creamos una nueva estrategia llamada facebook dentro de utils/auth/strategies/facebook.js:



Teniendo nuestra estrategia de **Facebook** ya podemos agregar las dos nuevas rutas de autenticación.



Con esto tenemos nuestra implementación de autenticación con Facebook.

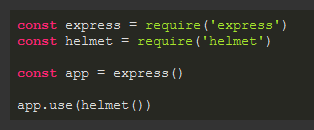
# Seguridad con Helmet y npm audit

En esta clase aprenderemos a ponerle seguridad a nuestra aplicación.

**Helmet** es un middleware que nos sirve para darle seguridad a las headers de nuestra aplicación.

Lo podremos utilizar de la siguiente manera con la configuración por defecto:

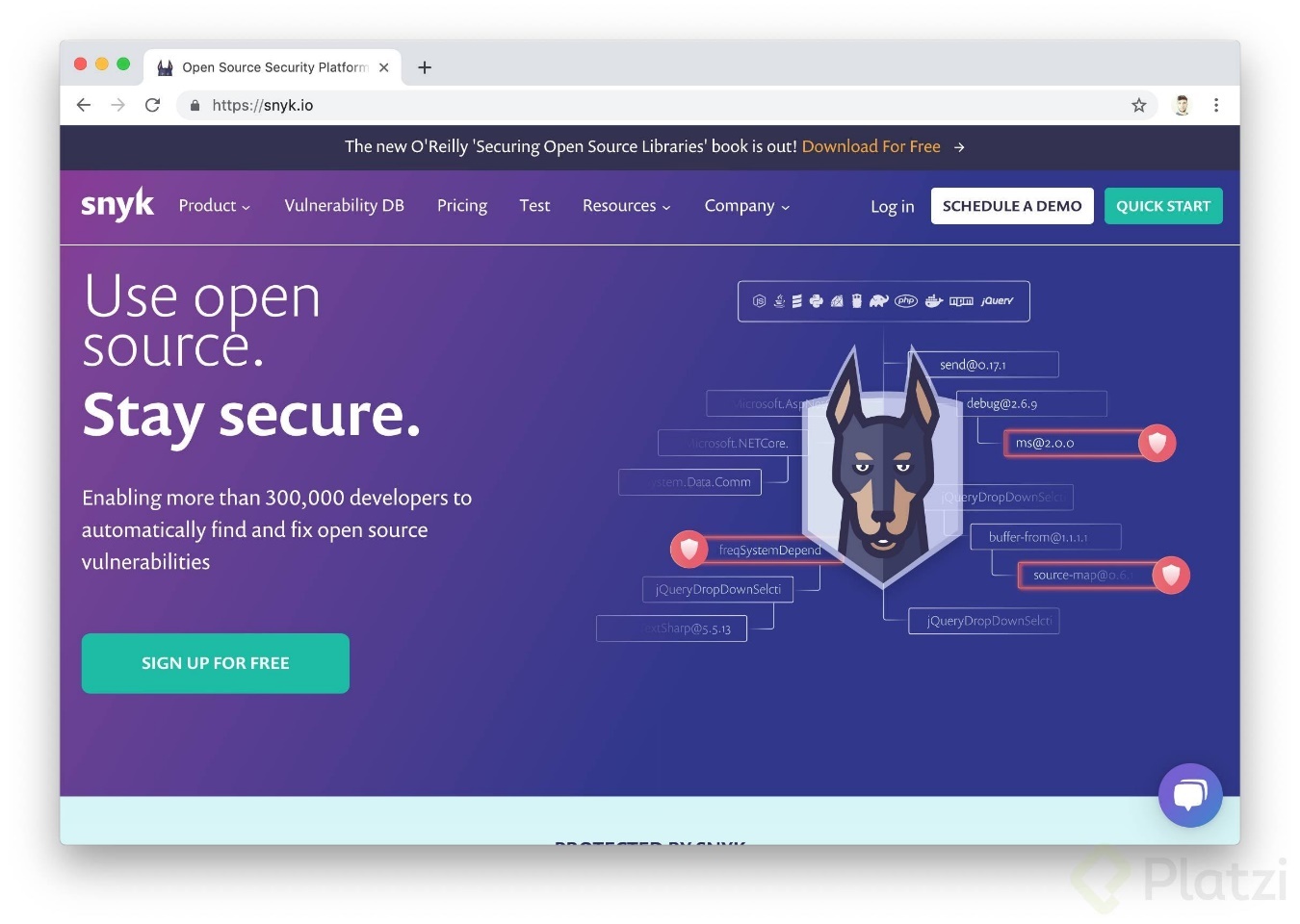
*npm i helmet*



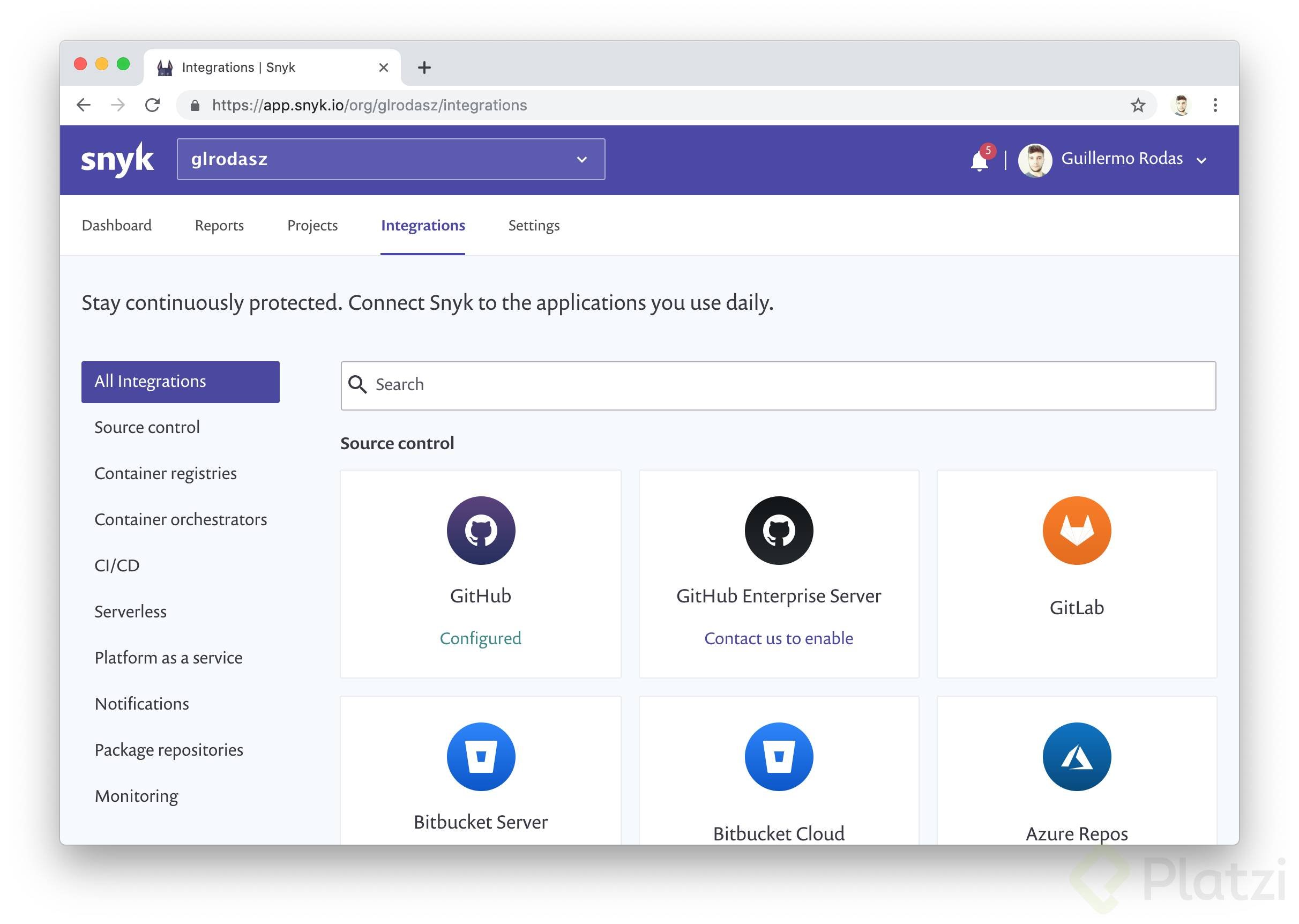
También aprenderemos a utilizar npm audit para ver cuáles son las vulnerabilidades de nuestra aplicación

# Automatizar el chequeo de vulnerabilidades con Snyk.

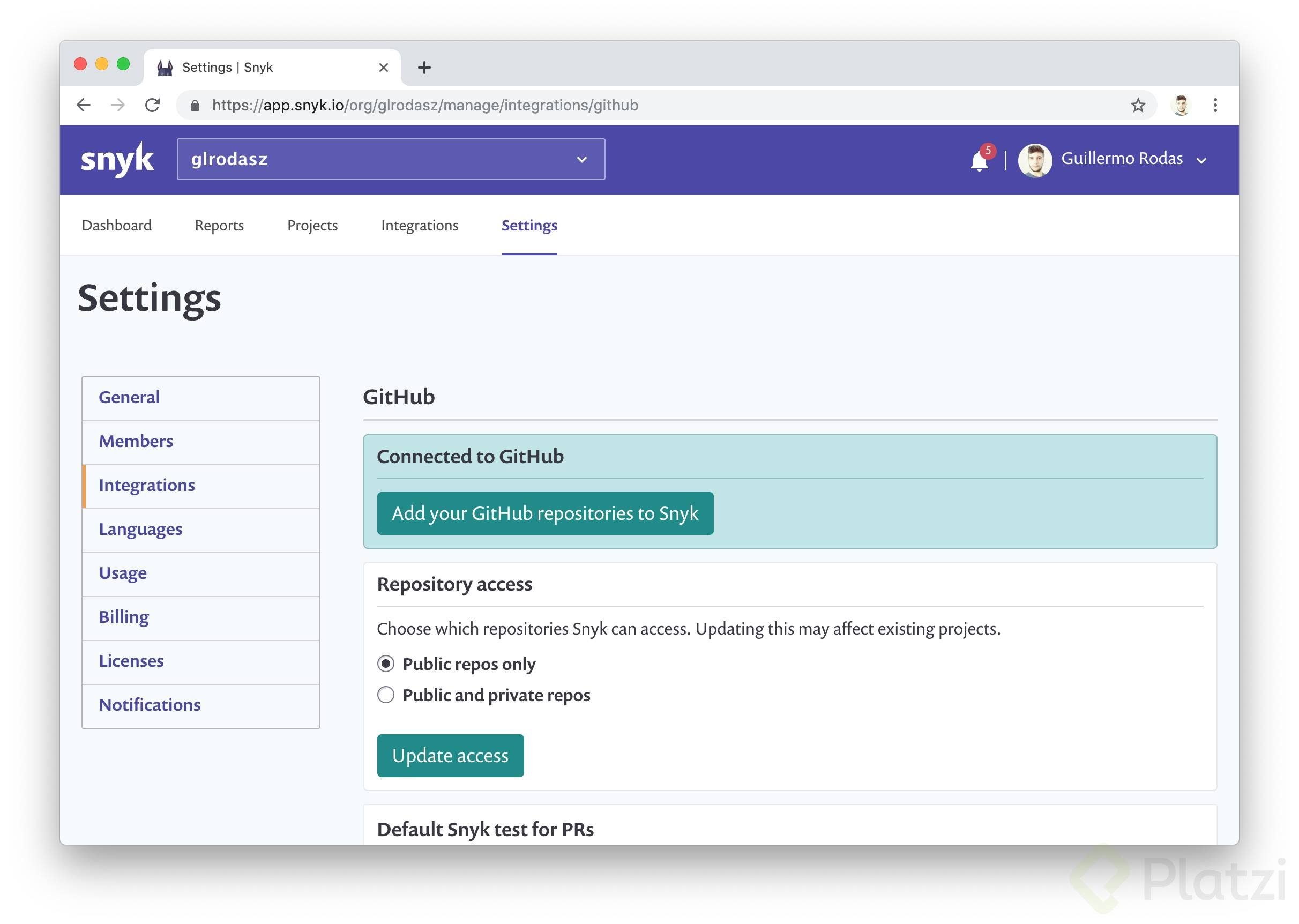
Para usar snyk lo primero es crear una cuenta en <https://app.snyk.io/signup> para agilizar el proceso recomiendo usar la cuenta de GitHub.



Después de la creación de la cuenta nos redireccionara a las integraciones o podemos ir directamente mediante el link https://app.snyk.io/org/<usuario>/integrations.



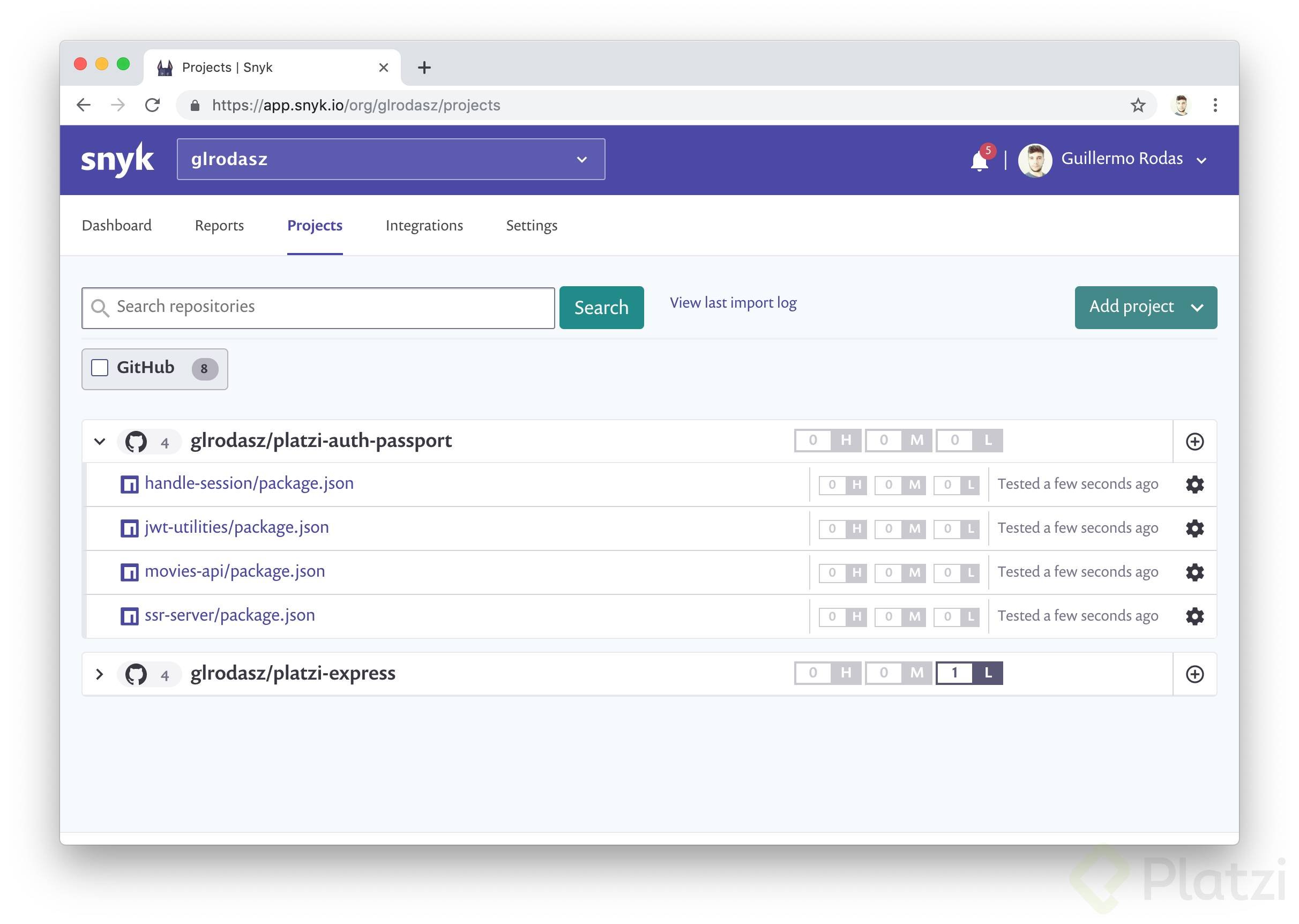
Seleccionamos la integración con GitHub (ó con el servicio más adecuado para nuestro proyecto) y allí nos aseguramos de conectar nuestra cuenta con GitHub, otorgando los permisos necesarios.



Por último necesitamos agregar el repositorio de nuestro proyecto para que haga la revisión automática de vulnerabilidades en cada Pull request, mediante el botón Add your GitHub repositories to Snyk.

Allí buscaremos el repositorio de nuestro proyecto, lo seleccionamos y le damos en el botón Add 1 selected repository to Snyk.

Cuando termine la integración podemos dirigirnos al dashboard de nuestros proyectos haciendo click en la pestaña Projects o mediante el link https://app.snyk.io/org/<usuario></usuario>/projects y verificar el estado de nuestras dependencias:



En lo posible debemos evitar tener vulnerabilidades High (H) o Medium (M) para corregirlas le damos click en View report and fix.

Tener en cuenta que algunas vulnerabilidades no tienen solución en el momento por lo que toca estar pendiente de un posible fix o cambiar de librería.

## **Qué es OWASP y buenas prácticas de seguridad**

**OWASP**: son las siglas del **Open Web Application Security Proyect**, es una organización que se encarga de velar por las buenas prácticas de seguridad a nivel mundial. Hay un documento que se llama el **Top Ten OWASP** y **lista los 10 riesgos de seguridad más comunes**.

Hablemos de las primeras 3:

**Inyección**: es un ataque que describe como hacer sql-inyection, esté ataque determina que hay input que no están protegidos podemos usar código SQL en nuestros inputs directamente para afectar la base de datos

**Broken Autentication**: Está relaciona con todo el tema de autenticación, explica en no fomentar o banear personas que intentan autenticar muchas veces o usar contraseñas muy débiles. Por ejemplo te puedes dirigir a [howsecureismypassword](https://howsecureismypassword.net/) y probar que tan seguro es tu password.

Si yo ingreso el password 123456 me dice que el password será crakeado instantáneamente y esto es porque este password hace parte de los top 5 de los password más usados, la mejor manera de construir un password seguro es que tenga muchos caracteres incluso caracteres especiales.

**Sensitive Data Exposure**: Aunque tu no lo creas revelar información sensible al usuario puede ser factor para que los hackers hagan algo llamado **ingeniería social**, si ellos tienen tu información personal se pueden hacer pasar por identidades gubernamentales o por cualquier otra identidad que se haga creer que es tu banco, seguridad social, etc.

Por eso es muy importante mantener la información segura y encriptada. La [GDPR] que significa **General Data Protection Regulation** que te obliga a que la información de tus usuarios sea guardada de manera segura, es muy importante seguir esta ley porque si no te pueden multar.

Si quieres seguir profundizando en este tema te recomiendo el curso de OWASP que está en platzi.

Top Ten de seguridad de OWASP <https://www.owasp.org/images/5/5e/OWASP-Top-10-2017-es.pdf>

## **Buenas prácticas de seguridad**

* **Usar un gestor de contraseñas**: las contraseñas nunca se deben de repetir además los gestores de contraseñas nos indican si las contraseñas tienen buena seguridad, si están repetidas y las genera por nosotros, recuerda que las contraseñas son como la ropa interior. Debes cambiarlas regularmente, no las debes dejar en cima del escritorio, y nunca se las prestes a nadie.

[](https://github.com/JasanHdz/passportjs/blob/master/notes/assets/passwords.png)

* **Usar multi-factor auth**: o autenticación en 2 pasos, esto trata además de que introduzcas la contraseña en tu cuenta, debes además generar otro método de autenticación, esté puede ser reconocimiento facial, la huella, o hay algo muy interesante llamado ue keys: son una llave que necesita ser introducida a tu computador para poder acceder a tu cuenta.
* **Seguridad en tu entorno**: Si tu dejas tu computador al aire libre en un lugar inseguro fácilmente también te van a poder hackear, de nada sirve que tengas la puerta más segura con todos los cerrojos si dejas una ventana abierta de tu casa.
* **Mantén actualizadas tus aplicaciones y SO**: como vimos en clases anteriores las vulnerabilidades de los paquetes suceden de un momento para otro, es muy importante entonces mantener actualizadas las aplicaciones y el sistema operativo porque ahí es donde se corrigen todas estas vulnerabilidades, además como lo mencionamos debes mantener actualizadas las dependencias, no solo de un proyecto sino de todos tus proyectos.
* **Mantente informado**: twitter es una muy buena fuente de información acerca de cuáles son los ataques de seguridad, el hashtag #secure #cibersecure te puede ayudar a identificar qué es lo que está sucediendo, investiga que blogs de seguridad existen, ve a eventos, y si vas a salir con una aplicación a producción te recomiendo que contrates a una compañía experta en seguridad que te haga una auditoria de tu aplicación.

Importants Questions and Answers.