Autenticación en Nest

La autenticación es una parte esencial de la mayoría de las aplicaciones. Hay muchos enfoques y estrategias diferentes para manejar la autenticación.

Passport es la biblioteca de autenticación de node.js más popular, conocida por la comunidad y utilizada con éxito en muchas aplicaciones de producción. Es sencillo integrar esta biblioteca con una aplicación Nest usando el @nestjs/passportmódulo.

Passport tiene un rico ecosistema de estrategias que implementan varios mecanismos de autenticación.

# Requisitos de autenticación

Desarrollemos nuestros requisitos. Para este caso de uso, los clientes comenzarán por autenticarse con un nombre de usuario y contraseña. Una vez autenticado, el servidor emitirá un JWT que se puede enviar como un token de portador en un encabezado de autorización en solicitudes posteriores para probar la autenticación.

También crearemos una ruta protegida a la que solo puedan acceder las solicitudes que contengan un JWT válido.

Comenzaremos con el primer requisito: autenticar a un usuario. Luego lo ampliaremos emitiendo un JWT.

Finalmente, crearemos una ruta protegida que verifica un JWT válido en la solicitud.

Primero necesitamos instalar los paquetes requeridos. Passport proporciona una estrategia llamada **passport-local** que implementa un mecanismo de autenticación de usuario/contraseña, que se adapta a nuestras necesidades para esta parte de nuestro caso de uso.

$ npm install --save @nestjs/passport passport passport-local

$ npm install --save-dev @types/passport-local

Para cualquier estrategia de Passport que elija, siempre necesitará los paquetes **@nestjs/Passport** y **passport**. Luego, deberá instalar el paquete específico de la estrategia (por ejemplo, **passport-jwt** o **passport-local**) que implementa la estrategia de autenticación particular que está creando.

# Implementación de estrategias de Passport

Primero consideremos cómo funciona Vanilla Passport. En Vanilla Passport, se configura una estrategia proporcionando dos cosas:

1. Un conjunto de opciones que son específicas de esa estrategia. Por ejemplo, en una estrategia JWT, puede proporcionar una semilla para firmar tokens.
2. Una "función callback de verificación", que le dice a Passport cómo interactuar con su tabla de usuarios (donde administra las cuentas de usuario). Aquí, verifica si un usuario existe (y/o crea un nuevo usuario) y si sus credenciales son válidas. La biblioteca Passport espera que esta función de verificacion retorne un usuario completo si la validación tiene éxito, o un valor nulo si falla (la falla se da si no se encuentra el usuario o, en el caso de passport-local, la contraseña no coincide).

Con **@nestjs/passport**, se configura una estrategia de Passport extendiendo la clase **PassportStrategy**. Pasa las opciones de estrategia (elemento 1 arriba) llamando al método **super()** en su subclase, pasando opcionalmente un objeto de opciones. Y uno debe proporcionar la función callback de verificación (elemento 2 anterior) implementando un método **validate()** en la subclase.

Comenzaremos generando un **AuthModule** y en él, un **AuthService**:

$ nest g module auth

$ nest g service auth

a medida que implementemos **AuthService**, nos resultará útil encapsular las operaciones del usuario en un **UsersService**, así que generemos ese módulo y servicio ahora:

$ nest g module users

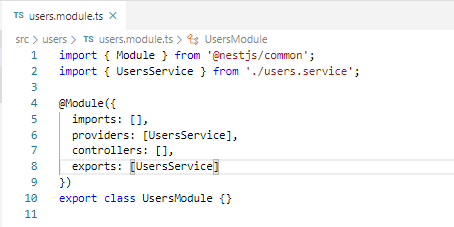
$ nest g service users

reemplace el contenido predeterminado de estos archivos generados como se muestra a continuación. Para nuestra aplicación de muestra, **UsersService** simplemente mantiene una lista de usuarios codificada en memoria y un método de búsqueda para recuperar uno por nombre de usuario.

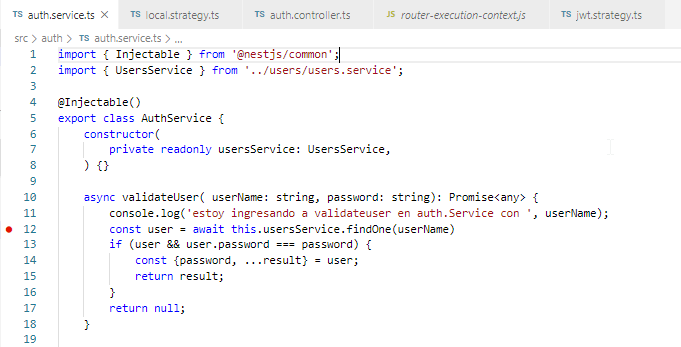
En una aplicación real, aquí es donde construiría su modelo de usuario y capa de persistencia, usando su biblioteca ORM de elección (por ejemplo, TypeORM, Sequelize, Mongoose, etc.).



En el **UsersModule**, el único cambio necesario es agregar **UsersService** al arreglo de exportaciones del decorador **@Module** para que sea visible fuera de este módulo (pronto lo usaremos en nuestro **AuthService**).



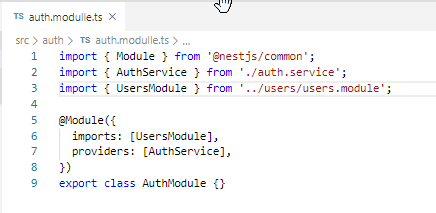
Nuestro **AuthService** tiene el trabajo de recuperar un usuario y verificar la contraseña. Creamos un método **validateUser()** para este propósito. En el siguiente código, usamos un operador de propagación ES6 conveniente para quitar la propiedad de contraseña del objeto de usuario antes de devolverlo. Invocaremos el método **validateUser()**desde nuestra estrategia-local de **Passport** en un momento.



ADVERTENCIA

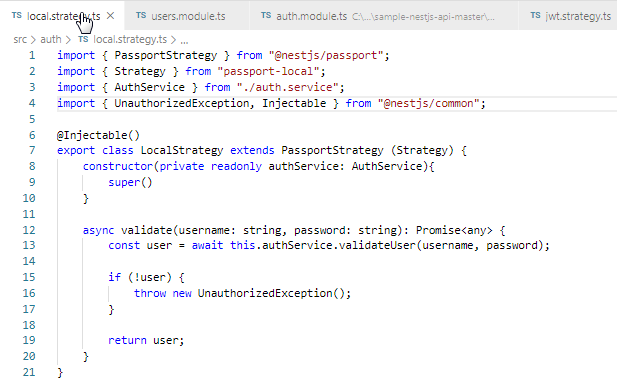
Por supuesto, en una aplicación real, no almacenaríamos una contraseña en texto plano. En su lugar, usaríamos una biblioteca como **bcrypt**, con un algoritmo hash unidireccional. Con ese enfoque, solo almacenaría contraseñas con hash y luego compararía la contraseña almacenada con una versión con hash de la contraseña entrante, por lo que nunca almacenará ni expondrá las contraseñas de usuario en texto sin formato. Para mantener nuestra aplicación de muestra simple, violamos ese mandato absoluto y usamos texto sin formato.

Ahora, actualizamos nuestro **AuthModule** para importar el **UsersModule**.



Implementación de passport-local

Ahora podemos implementar nuestra estrategia de autenticación local de Passport. Creemos un archivo llamado **local.strategy.ts** en la carpeta **/auth** y agregamos el siguiente código:



Hemos seguido la receta descrita anteriormente para todas las estrategias de Passport. En nuestro caso de uso, con passport-local, no hay opciones de configuración, por lo que nuestro constructor simplemente llama **super()**, sin un objeto de opciones.

INSINUACIÓN

Podemos pasar un objeto de opciones en la llamada a **super()** para personalizar el comportamiento de la estrategia de Passport. En este ejemplo, la estrategia local de Passport espera de forma predeterminada las propiedades llamadas username y password en el cuerpo de la solicitud. Pasar un objeto de opciones para especificar diferentes nombres de propiedades, por ejemplo: super({ usernameField: 'email' }). Consulte la documentación de Passport para obtener más información.

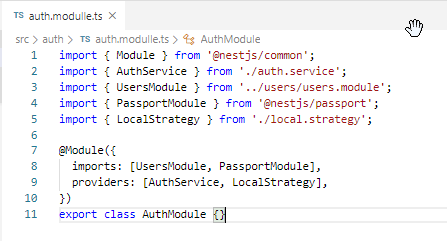
También implementamos el método **validate()**. Para cada estrategia, Passport llamará a la función de verificación (implementada con el método **validate()**en **@nestjs/passport**) utilizando un conjunto de parámetros específico de la estrategia apropiado. Por lo local-estrategia, pasaporte espera un método **validate()** con la siguiente firma:

validate(username: string, password:string): any

La mayor parte del trabajo de validación se realiza en nuestro **AuthService** (con la ayuda de nuestro **UsersService**), por lo que este método es bastante sencillo. El método **validate()** para cualquier estrategia de Passport seguirá un patrón similar, variando solo en los detalles de cómo se representan las credenciales. Si se encuentra un usuario y las credenciales son válidas, se devuelve el usuario para que Passport pueda completar sus tareas (por ejemplo, crear la propiedad **user** en el objeto **Request**), y la canalización de manejo de solicitudes puede continuar. Si no se encuentra, lanzamos una excepción y dejamos que nuestra capa de excepciones la maneje.

Normalmente, la única diferencia significativa en el método **validate()** para cada estrategia es cómo se determina si un usuario existe y es válido. Por ejemplo, en una estrategia **JWT**, según los requisitos, podemos evaluar si el **userId** que se transporta en el **token** decodificado coincide con un registro en nuestra base de datos de usuarios o coincide con una lista de tokens revocados. Por lo tanto, este patrón de sub clasificación e implementación de la validación por estrategia-específica es consistente, elegante y extensible.

Necesitamos configurar nuestro **AuthModule** para usar las funciones de Passport que acabamos de definir. Actualice **auth.module.ts** para verse así:



# Guards Built-in de Passaport

El capítulo de **Guards** (guardias) describe la función principal de **Guards**: determinar si una solicitud será manejada por el manejador de ruta o no. Eso esto es así mientras utilicemos esa capacidad estándar. Sin embargo, en el contexto del uso del módulo **@nestjs/Passport**, también presentaremos una pequeña dificultad que al principio puede ser confusa, así que hablemos de eso ahora. Considere que su aplicación puede existir en dos estados, desde una perspectiva de autenticación:

1. el usuario / cliente no ha iniciado sesión (no está autenticado)
2. el usuario / cliente ha iniciado sesión (está autenticado)

En el primer caso (el usuario no está conectado), necesitamos realizar dos funciones distintas:

* Restringir las rutas a las que puede acceder un usuario no autenticado (es decir, negar el acceso a rutas restringidas). Usaremos para tal efecto guardias en su capacidad familiar, colocando una guardia en las rutas protegidas. Como se puede anticipar, verificaremos la presencia de un **JWT** válido en este punto de guardia, por lo que trabajaremos en este punto de guardia más adelante, una vez que hayamos emitido un **JWT** con éxito.
* Inicie el paso de autenticación en sí mismo cuando un usuario previamente no autenticado intente iniciar sesión. Este es el paso en el que emitiremos un **JWT** a un usuario válido. Pensando en esto por un momento, sabemos que necesitaremos hacer un **POST** con credenciales de nombre de usuario/contraseña para iniciar la autenticación, así que configuraremos una ruta (end point) **POST** **/auth/login** para manejar eso. Esto plantea la pregunta: ¿cómo invocamos exactamente la estrategia **passport-local** en esa ruta?

La respuesta es sencilla: utilizando otro tipo de Guardia, ligeramente diferente. El módulo **@nestjs/passport** nos proporciona un **Guard** integrado que hace esto por nosotros. Este **Guard** invoca la estrategia **Passport** y da inicio a los pasos descritos anteriormente (recuperar credenciales, ejecutar la función de verificación, crear la propiedad **user**, etc.).

El segundo caso enumerado anteriormente (usuario registrado) simplemente se basa en el tipo estándar de Guardia que ya discutimos para permitir el acceso a rutas protegidas para usuarios registrados.

# Ruta de inicio de sesión (login)

Con la estrategia en su lugar, ahora podemos implementar una ruta básica **/auth/login** y aplicar la Guardia incorporada para iniciar el flujo de **passport-local**.

Abra el archivo **auth.controller.ts** y reemplace su contenido con lo siguiente:



Con **@UseGuards(AuthGuard('local'))** estamos utilizando un **AuthGuard** que **@nestjs/passportse** aprovisionó automáticamente para nosotros cuando ampliamos la estrategia de **passports-local**. Analicemos eso. Nuestra estrategia local de Passport tiene un nombre predeterminado de 'local'. Hacemos referencia a ese nombre en el decorador **@UseGuards()** para asociarlo con el código proporcionado por el paquete **passport-local**. Esto se usa para eliminar la ambigüedad de qué estrategia invocar en caso de que tengamos múltiples estrategias de Passport en nuestra aplicación (cada una de las cuales puede proporcionar una estrategia específica **AuthGuard**). Si bien solo tenemos una estrategia de este tipo hasta ahora, en breve agregaremos una segunda, por lo que es necesaria para la desambiguación.

Para probar nuestra ruta, haremos que nuestra ruta **/auth/**login simplemente devuelva el usuario por ahora. Esto también nos permite demostrar otra función de Passport: Passport crea automáticamente un objeto **user**, en función del valor que devolvemos del método **validate()**, y lo asigna al objeto **Request** como **req.user**. Más adelante, reemplazaremos esto con código para crear y devolver en su lugar un **JWT**.

Dado que estas son rutas API, las probaremos usando la biblioteca cURL comúnmente disponible . Puede probar con cualquiera de los objetos **user** codificados en el **UsersService**.

$ # POST to /auth/login

$ curl -X POST http://localhost:3000/auth/login -d '{"username": "john", "password": "changeme"}' -H "Content-Type: application/json"

$ # result -> {"userId":1,"username":"john"}

o podemos probarlo con Postman o Insomnia.

Si bien esto funciona, pasar el nombre de la estrategia directamente a **AuthGuard()** introduce cadenas mágicas en el código base. En su lugar, recomendamos crear su propia clase, como se muestra a continuación:

El archivo a crear sera **auth/local-auth.guard.ts** y el contenido

import { Injectable } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

@Injectable()

export class LocalAuthGuard extends AuthGuard('local') {}

Ahora, podemos actualizar el controlador de la ruta **/auth/login** y usar en su lugar **LocalAuthGuard**:

@UseGuards(LocalAuthGuard)

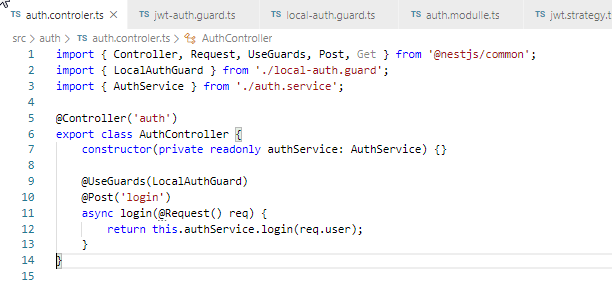
@Post('auth/login')

async login(@Request() req) {

return req.user;

}

Usando este podríamos escribir el **auth.controller.ts** de la siguiente manera



Donde estamos reemplazando

@UseGuards(AuthGuard('local'))

por

  @UseGuards(LocalAuthGuard)

Habiendo definido el **auth.controller.ts**.

# Funcionalidad JWT

Estamos listos para pasar a la parte JWT de nuestro sistema de autenticación. Revisemos y refinemos nuestros requisitos:

1. Permita que los usuarios se autentiquen con nombre de usuario/contraseña, devolviendo un JWT para usar en llamadas posteriores a puntos finales de API protegidos. Estamos en camino de cumplir con este requisito. Para completarlo, necesitaremos escribir el código que emite un JWT.
2. Cree rutas API que estén protegidas en función de la presencia de un JWT válido como token de portador

Necesitaremos instalar un par de paquetes más para cumplir con nuestros requisitos de JWT:

$ npm install --save @nestjs/jwt passport-jwt

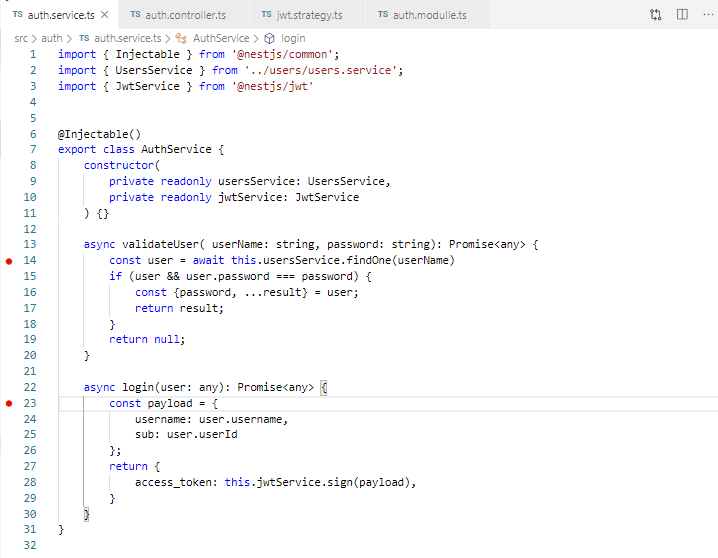
$ npm install --save-dev @types/passport-jwt

El paquete @nestjs/jwt (ver más <https://github.com/nestjs/jwt> ) es un paquete de utilidades que ayuda con la manipulación de JWT. El paquete passport-jwt es el paquete Passport que implementa la estrategia JWT y paquete @types/passport-jwt proporciona las definiciones de tipo de TypeScript.

Echemos un vistazo más de cerca a cómo el **POST** en **/auth/login** maneja una solicitud. Hemos decorado la ruta utilizando el incorporado **AuthGuard** proporcionado por la estrategia de **pasaporte-local**. Esto significa que:

1. El controlador de ruta solo se invocará si el usuario ha sido validado
2. El parámetro **req** contendrá una propiedad **user** (populada por Passport durante el flujo de autenticación de pasaporte local)

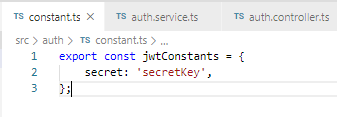
Con esto en mente, ahora finalmente podemos generar un JWT real y devolverlo en esta ruta. Para mantener nuestros servicios perfectamente modularizados, nos encargaremos de generar el **JWT** en **authService**. Abra el archivo **auth.service.ts** en la carpeta **auth**, agregue el método **login()** e importe el **JwtService** como se muestra:



Estamos usando la biblioteca **@nestjs/jwt**, que proporciona una función **sign()** para generar nuestro **JWT** a partir de un subconjunto de las propiedades del objeto **user**, que luego devolvemos como un objeto simple con una sola propiedad **access\_token**. Nota: elegimos un nombre de propiedad de **sub** para mantener nuestro valor **userId** de acuerdo con los estándares de **JWT**. No olvide inyectar el proveedor **JwtService** en **AuthService**.

Ahora necesitamos actualizar **AuthModule** para importar las nuevas dependencias y configurar el **JwtModule**.

Primero, creemos en la carpeta **/auth** el archivo **constants.ts** y agreguemos el siguiente código:



Usaremos esto para compartir nuestra clave entre la firma de JWT y los pasos de verificación.

ADVERTENCIA

No exponga esta clave públicamente . Lo hemos hecho aquí para dejar en claro lo que hace el código, pero en un sistema de producción debe proteger esta clave utilizando las medidas adecuadas, como una secret vault, una variable de entorno o un servicio de configuración.

Ahora, abra **auth.module.ts** en la carpeta **/auth** y actualícelo para que se vea así:



Configuramos el **JwtModule** mediante el uso de **register()**, pasandole un objeto de configuración. Consulte <https://github.com/nestjs/jwt/blob/master/README.md> para obtener más información sobre Nest JwtModule y <https://github.com/auth0/node-jsonwebtoken#usage> para obtener más detalles sobre las opciones de configuración disponibles.

Ahora podemos actualizar la ruta **/auth/login** n el **auth.controller.ts** para devolver un **JWT**.



Sigamos adelante y probemos nuestras rutas usando cURL nuevamente. Puede probar con cualquiera de los objetos **user** codificados (harcode) en el **UsersService**.

$ # POST to /auth/login

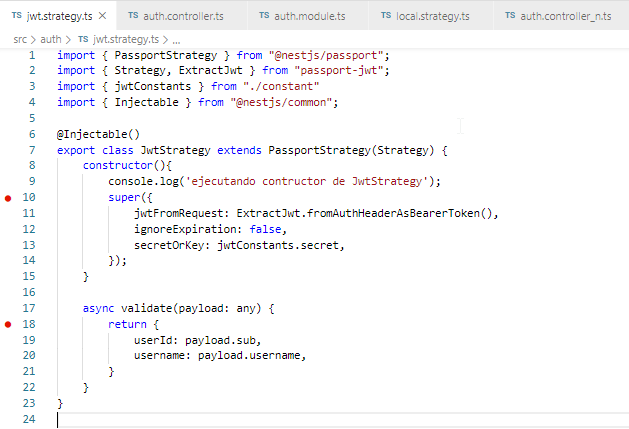
$ curl -X POST http://localhost:3000/auth/login -d '{"username": "john", "password": "changeme"}' -H "Content-Type: application/json"

$ # result -> {"access\_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}

$ # Note: above JWT truncated

Implementación de Passport JWT

Ahora podemos abordar nuestro requisito final: proteger los puntos finales al requerir que un **JWT** válido esté presente en la solicitud. Passport también puede ayudarnos aquí. Proporciona la estrategia **passport-jwt** para asegurar end points RESTful con JSON Web Tokens. Comenzamos creando un archivo llamado **jwt.strategy.ts** en la carpeta **/auth** y agregamos el siguiente código:



Con nuestro **JwtStrategy**, hemos seguido la misma receta descrita anteriormente para todas las estrategias de Passport. Esta estrategia requiere cierta inicialización, por lo que lo hacemos pasando un objeto de opciones en la llamada a **super()** en el constructor. Puede leer más sobre las opciones disponibles <https://github.com/mikenicholson/passport-jwt#configure-strategy> . En nuestro caso, estas opciones son:

**jwtFromRequest**: proporciona el método mediante el cual se extraerá el **JWT** del archivo **Request**. Usaremos el enfoque estándar de proporcionar un token de portador en el encabezado de Autorización de nuestras solicitudes de API. Aquí <https://github.com/mikenicholson/passport-jwt#extracting-the-jwt-from-the-request> se describen otras opciones .

**ignoreExpiration**: solo para ser explícitos, elegimos la configuración predeterminada **false** , que delega la responsabilidad de garantizar que un **JWT** no haya caducado en el módulo Passport. Esto significa que si nuestra ruta se proporciona con un **JWT** caducado, la solicitud será denegada y se enviará una respuesta **401 Unauthorized**. Passport maneja esto de manera conveniente y automática por nosotros.

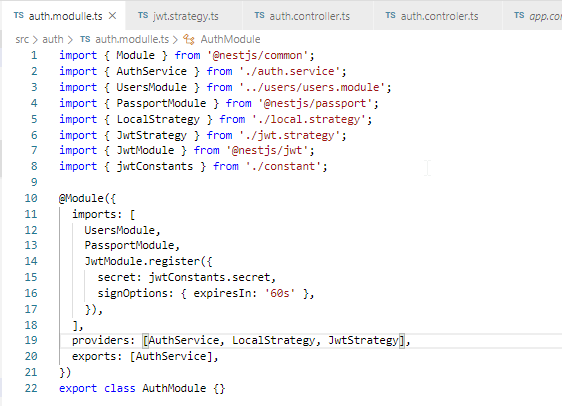
**secretOrKey**: utilizamos la opción conveniente de proporcionar una cifrado simétrico para firmar el token. Otras opciones, como una clave pública codificada en PEM, pueden ser más apropiadas para aplicaciones de producción (consulte aquí <https://github.com/mikenicholson/passport-jwt#extracting-the-jwt-from-the-request> para obtener más información). En cualquier caso, como se advirtió anteriormente, no exponga el cifrado públicamente .

El método **validate()** merece una discusión. Para la estrategia **jwt**, Passport primero verifica la firma del **JWT** y decodifica el JSON. Luego invoca nuestro método **validate()** pasando el JSON decodificado como su único parámetro. Según la forma en que funciona la firma JWT, tenemos la garantía de que estamos recibiendo un token válido que hemos firmado y emitido previamente a un usuario válido.

Como resultado de todo esto, nuestra respuesta a la callback function **validate()** es trivial: simplemente devolvemos un objeto que contiene las propiedades **userId** y **username**. Recuerde nuevamente que **Passport** construirá un objeto **user** basado en el valor de retorno de nuestro método **validate()** y lo adjuntará como una propiedad en el objeto **Request**.

También vale la pena señalar que este enfoque nos deja espacio ('ganchos' por así decirlo) para inyectar otra lógica empresarial en el proceso. Por ejemplo, podríamos hacer una búsqueda en la base de datos en nuestro método **validate()** para extraer más información sobre el usuario, lo que resultaría en un objeto **user** disponible en nuestro Request, más enriquecido. Este es también el lugar donde podemos decidir realizar una validación adicional de tokens, como buscar userId en una lista de tokens revocados, lo que nos permite realizar la revocación de tokens. El modelo que hemos implementado aquí en nuestro código de muestra es un modelo rápido, "JWT sin estado", donde cada llamada a la API se autoriza inmediatamente en función de la presencia de un JWT válido y una pequeña cantidad de información sobre el solicitante (su **userId** y **username**) que está disponible en nuestro canal de solicitudes.

Agregue el nuevo **JwtStrategy** como proveedor en **AuthModule**:



Al importar el mismo cifrado que usamos cuando firmamos el **JWT**, nos aseguramos de que la fase de verificación realizada por **Passport** y la fase de firma realizada en nuestro **AuthService** utilicen un cifrado común.

Finalmente, creamos el archivo **jwt-auth.guard.ts** en la carpeta **/auth** y definimos la clase **JwtAuthGuard** que extiende la incorporada **AuthGuard**:

import { Injectable } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

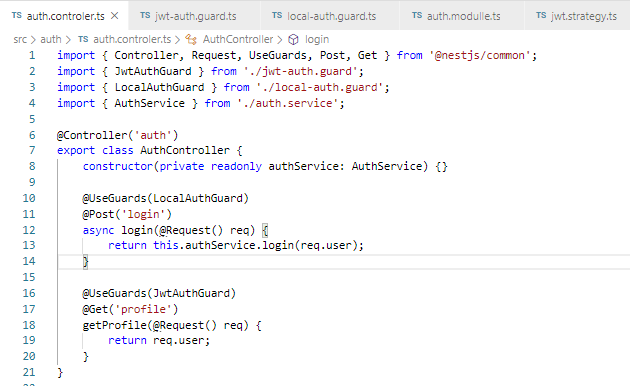
@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

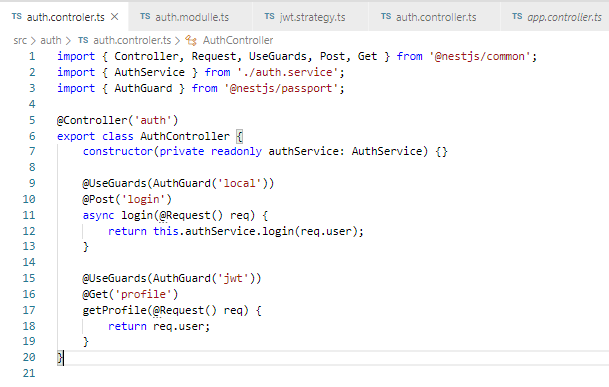
# Implementar guardias de ruta protegida y estrategia JWT

Ahora podemos implementar nuestra ruta protegida y su Guard asociada.

Abra el archivo **auth.controller.ts** y actualícemoslo como se muestra a continuación:



o sin usar **jwt-auth.guard.ts** directamente escribiendo



como lo hicimos con la estrategia local.

Una vez más, estamos aplicando el **AuthGuard** que el módulo **@nestjs/Passport** nos suministró automáticamente cuando configuramos el módulo **passport-jwt**. Este Guard hace referencia por su nombre por defecto, jwt. Cuando se accede a nuestra ruta **GET/profile**, el Guard invocará automáticamente nuestra lógica configurada y personalizada **passport-jwt**, validando el JWT y asignando la propiedad **user** al objeto **Request**.

Asegúrese de que la aplicación se esté ejecutando y pruebe las rutas con cURL.

$ # GET /profile

$ curl http://localhost:3000/profile

$ # result -> {"statusCode":401,"error":"Unauthorized"}

Si ejecutamos el GET/profile si n autenticación previa recibiremos el 401, debiéramos promerpo autenticarnos y lueho antes de 60 msegundos intentar acceder al GET/profile

$ # POST /auth/login

$ curl -X POST http://localhost:3000/auth/login -d '{"username": "john", "password": "changeme"}' -H "Content-Type: application/json"

$ # result -> {"access\_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm... }

$ # GET /profile using access\_token returned from previous step as bearer code

$ curl http://localhost:3000/profile -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."

$ # result -> {"userId":1,"username":"john"}

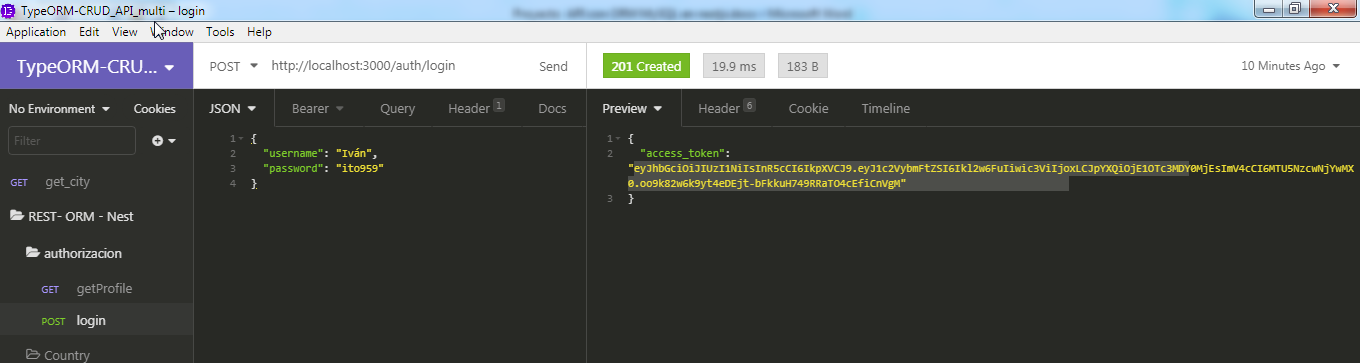
Tenga en cuenta que en el **AuthModule**, configuramos el **JWT** para que tenga una caducidad de 60 seconds. Probablemente sea un vencimiento demasiado corto, y tratar los detalles del vencimiento y la actualización del token está fuera del alcance de este artículo.

Sin embargo, elegimos eso para demostrar una calidad importante de los **JWT** y la estrategia de pasaporte-jwt. Si espera 60 segundos después de la autenticación y antes de intentar una solicitud GET/profile, recibirá una respuesta 401 Unauthorized. Esto se debe a que Passport verifica automáticamente el tiempo de vencimiento del **JWT**, lo que le ahorra la molestia de hacerlo en su aplicación.

De esta manera hemos completado nuestra implementación de autenticación **JWT**. Los clientes de JavaScript (como Angular/React/Vue) y otras aplicaciones de JavaScript ahora pueden autenticarse y comunicarse de forma segura con nuestro servidor API. Puede encontrar una versión completa del código en este capítulo aquí <https://github.com/nestjs/nest/tree/master/sample/19-auth-jwt> .

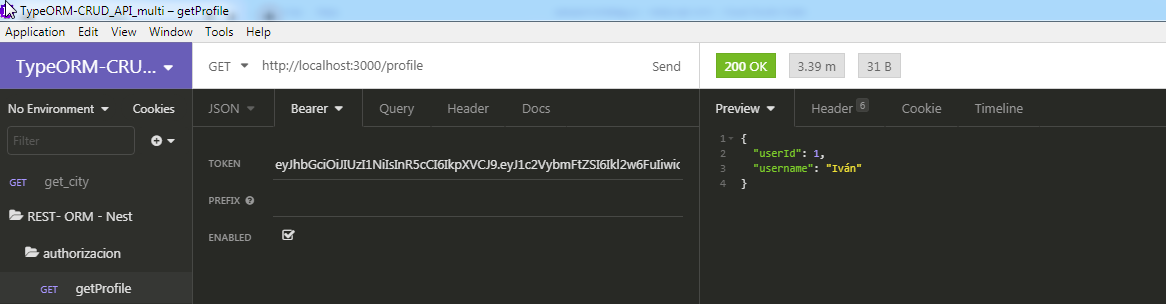
La pruebas de la autenticacion se llevan a cabo desde Postman o insomnia.

Definimos un POST para la ruta <http://localhost:3000/auth/login> y en JSON pasamos u objeto json con el usario y la password. Al enviar la solicitud nos devolverá un jwt sobre la solapa Preview.

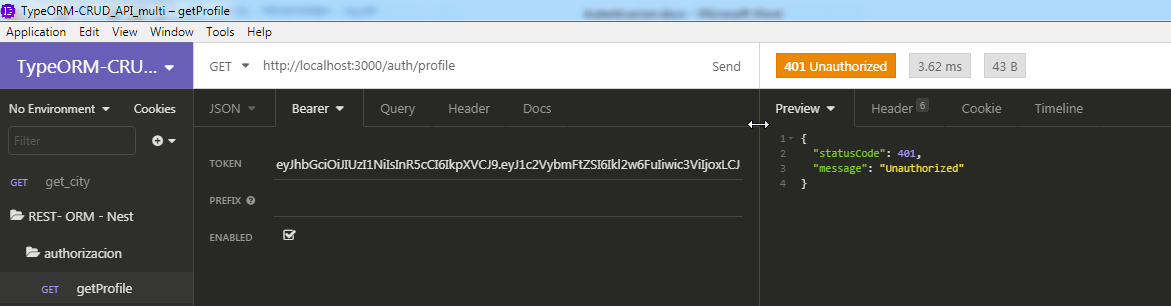


Definimos un GET sobre la ruta <http://localhost:3000/auth/profile> y en la solapa Autorizacion seleccionamos Bearer y pegamos el token que nos respondió el POST, en teoría si aplicacmos el GET dentro de los 60 segs que se especificaron para debiera de retornar algo como

result -> {"userId":1,"username":"Iván"}



o en caso de superar el timeout un error 401 unauthorizated



# Estrategia predeterminada

En nuestro **AuthController**, pasamos el nombre de la estrategia en la función **AuthGuard()**. Necesitamos hacer esto porque hemos introducido dos estrategias de **Passport** (**passport-local** y **passport-jwt**), las cuales proporcionan implementaciones de varios componentes de **Passport**. Pasar el nombre elimina la ambigüedad de la implementación a la que estamos vinculando. Cuando se incluyen múltiples estrategias en una aplicación, podemos declarar una estrategia predeterminada para que ya no tengamos que pasar el nombre en la función **AuthGuard** si usamos esa estrategia predeterminada. A continuación, se explica cómo registrar una estrategia predeterminada al importar el archivo **PassportModule**. Este código iría en el **AuthModule**:

