# Documentación Técnica


**ABRIL 10**

**Aaron Palacios Olea**
**Sellit**

SELLIT

# Introducción

## Texto del subtítulo aquí

Esta es la documentación técnica de este proyecto llamado Sellit.

Cabe aclarar que quedan aspectos y mejoras, por falta de relevancia en el plazo estimado, no se han realizado y quedan pendientes de hacerlas en un futuro.

# Tecnologías escogidas

## Frontend

Para este proyecto, en el apartado Frontend he decidido escoger el framework Angular.

Angular es una excelente opción para construir aplicaciones web modernas y escalables por varias razones:

- Potente framework: Angular es un framework potente que proporciona una gran cantidad de herramientas y funcionalidades para construir aplicaciones web complejas.

- Mantenibilidad: La estructura de componentes de Angular hace que el código sea fácil de mantener y escalar.

- Modularidad: Angular permite crear aplicaciones modulares mediante la utilización de módulos, lo que facilita la organización y estructuración del código.

Por estas razones he escogido este framwork.

# Tecnologías escogidas

## Backend

Para este proyecto, en el apartado Backend he decidido escoger NodeJS.

Angular es una excelente opción para construir aplicaciones web modernas y escalables por varias razones:

- Velocidad: Node.js está construido sobre el motor V8 de Google, lo que significa que es extremadamente rápido y eficiente.

- Escalabilidad: Node.js es perfecto para aplicaciones de alta escalabilidad, ya que puede manejar una gran cantidad de solicitudes simultáneas sin afectar el rendimiento.

- Flexibilidad: Node.js puede ser utilizado para una amplia variedad de aplicaciones, desde el desarrollo de aplicaciones web hasta la creación de herramientas de línea de comandos.

- Fácil aprendizaje: Node.js utiliza JavaScript como lenguaje de programación, lo que significa que si ya tienes experiencia con JavaScript en el frontend, es fácil de aprender y comenzar a utilizar en el backend.

- Manejo de datos en tiempo real: Node.js es ideal para aplicaciones en tiempo real, ya que permite la comunicación bidireccional en tiempo real entre el cliente y el servidor.

Por estas razones he escogido NodeJS.

A parte, he implementado la librería express. Esta facilita muchas funcionalidades y reduce la cantidad de código.

# Tecnologías escogidas

## Base de datos

Para este proyecto, he usado MongoDB como DB.

Angular es una excelente opción para construir aplicaciones web modernas y escalables por varias razones:

- Flexibilidad: MongoDB es una base de datos NoSQL, lo que significa que no utiliza una estructura fija de tablas y relaciones como una base de datos relacional. Esto hace que sea muy flexible y adaptable a diferentes tipos de datos y aplicaciones.

- Escalabilidad: MongoDB es muy escalable, lo que significa que puede manejar grandes volúmenes de datos y un alto número de solicitudes de manera efectiva.

- Documentos BSON: MongoDB utiliza un formato de almacenamiento de datos llamado BSON, que es muy similar a JSON. Este formato es muy fácil de leer y escribir para los desarrolladores, lo que facilita el trabajo con la base de datos.

- Alto rendimiento: MongoDB es conocido por su alto rendimiento y velocidad. Debido a su arquitectura de almacenamiento en memoria, puede procesar solicitudes rápidamente y manejar grandes volúmenes de datos sin comprometer la velocidad.

Por estas razones he escogido MongoDB como DB del proyecto.

# Funcionalidades Técnicas

## Backend

### Configuración del servidor

El fichero donde se configura el servidor está en el directorio "/backend/models/server.js".

Este se encarga de implementar todo lo necesario para el funcionamiento del servidor.

- Rutas: Se definen el path de las rutas y el fichero correspondiente a cada una.

```
//Paths
this.usersPath = '/api/users'
this.authPath = '/api/auth'
this.productsPath = '/api/products'
this.imagesPath = '/api/images'
this.salesPath = '/api/sales'
this.offersPath = '/api/offers'
```

```
routes() {
  this.app.use(this.authPath, require('../routes/auth'))
  this.app.use(this.usersPath, require('../routes/user'))
  this.app.use(this.productsPath, require('../routes/product'))
  this.app.use(this.imagesPath, require('../routes/images'))
  this.app.use(this.salesPath, require('../routes/sales'))
  this.app.use(this.offersPath, require('../routes/offers'))
}
```

- Middlewares: Se encarga de activar el Cors y de parsear el Body de todas las peticiones que se reciban.

```
middlewares() {
    //CORS
    this.app.use(cors())

    //Read & Parse body
    this.app.use(express.json())
}
```

- Conexión con DB: Llama a esta función que se encarga de realizar la conexión con la DB. Este fichero se encuentra en el directorio "/backend/db/config.js".

```
async dbConnect(){
    await dbConnection()
}
```

```
const dbConnection = async() => {

    try{
        await mongoose.connect(process.env.MONGODB_CNN, {
            useNewUrlParser: true,
            useUnifiedTopology: true,
        })

        console.log("DB Connected")
    } catch (err){
        console.log(err)
        throw new Error(err)
    }
}
```

- Activa la escucha a todas las peticiones entrantes y define el puerto por el cual estará activo el servidor.

```
  listen() {
    this.app.listen(this.port, () => {
      console.log('Server running on port', this.port)
    })
  }

}
```

Cabe destacar que este es un modelo del servidor.
Realmente este se arranca en este fichero “/backend/index.js”.

```
const server = new Server()
server.listen()
```

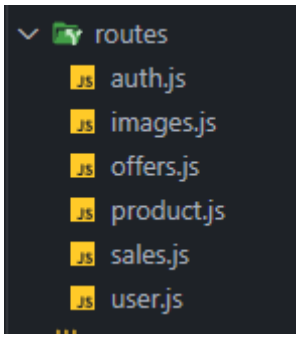
# Funcionalidades Técnicas

## Backend

### Rutas

Todos los ficheros de configuración para las rutas se encuentran en el directorio "/backend/routes".

Dentro podremos ver que cada modalidad tiene su propio fichero, así podemos tener el código más modularizado y organizado.




Si entramos por ejemplo en el de product.js, podemos ver que cada controlador tiene su propio path.

```
router.get('/', productsGet)

router.post('/userProducts', [
  validateJWT
], userProducts)

router.post('/checkUserHaveProduct', checkUserHaveProduct, [
  check('id', 'This is not a valid ID').isMongoId()
])

router.post('/countProducts', [
  // validateJWT
], getProductsCount)
```

A parte, se indica que tipo de petición tiene (GET, POST, PUT…).

Otras funcionalidades que podemos añadirles son los Middlewares.

Podemos utilizar los que ya vienen con la librería express o los personalizados por nosotros:

```
router.post('/userProducts', [
  validateJWT
], userProducts)

router.post('/checkUserHaveProduct', checkUserHaveProduct, [
  check('id', 'This is not a valid ID').isMongoId()
])
```

# Funcionalidades Técnicas

## Backend

### Middlewares

Todos los ficheros que contienen los Middlewares personalizados utilizados en los ficheros routes se encuentran en el directorio “/backend/middlewares”.

Los middlewares son funciones que se ejecutan entre la solicitud de un cliente y la respuesta del servidor en una aplicación Node.js.
Estas funciones pueden tener acceso a la solicitud (req) y la respuesta (res) y se utilizan para realizar una variedad de tareas, como validar datos, autenticar usuarios, administrar sesiones, manipular datos de solicitud y respuesta, y mucho más.

En la página anterior podemos ver como se implemente el Middleware personalizado “validateJWT”, así que veamos el fichero.

```
const validateJWT = async( req, res = response, next ) => { //data from headers
  const token = req.header('x-token')  //personalized name

  if(!token){
    return res.status(401).json({
      msg: 'There is not a token in the request'
    })
  }

  try {
    const { uid } = jwt.verify(token, process.env.SECRETORPRIVATEKEY) //verify token

    //chek if user have the correspond id and get data
    const user = await User.findById( uid )
    if(!user){
      return res.status(401).json({
        msg: 'Invalid token - user not exists in DB'
```

En sí este Middleware se encarga de comprobar que el JWT (JSON Web Token) recibido por una petición entrante es válido.

En caso que no lo sea, se envía una respuesta al cliente indicándole que el token no es válido y por lo tanto, no le enviaría los datos que piden.

```
} catch (error) {
  console.log(error)
  res.status(401).json({
    msg: 'Invalid token'
  })
}
```

Para resumirlo, si una petición entra por routes, primero pasa por el Middleware definido:

```
router.post('/userProducts', [
  validateJWT
], userProducts)
```

Una vez entra, en este caso si el JWT es válido, dará su “OK” y pasará al controlador userProducts:

```
router.post('/userProducts', [
  validateJWT
], userProducts)
```

Si no es válido, entonces devuelve una respuesta al cliente:

```
} catch (error) {
  console.log(error)
  res.status(401).json({
    msg: 'Invalid token'
  })
}
```

# Funcionalidades Técnicas

## Backend

### Controllers

Todos los ficheros que contienen los Controllers utilizados en los ficheros routes se encuentran en el directorio “/backend/controllers”.

Para seguir con el ejemplo, veremos los controladores que se ubican en el fichero “products.js”.

Como podemos ver, este controlador recibe los parámetros pasados por query (por la URL) para realizar una búsqueda.

```
/**
 * Devuelve todos los productos
 */
const productsGet = async(req, res = response) => {

  const { limit/* = 15*/, from = 0 } = req.query

  const [ total, products ] = await Promise.all([
    Product.countDocuments({state: true}),
    Product.find({state: true})
    .skip(Number(from))
    .limit(Number(limit))
  ])

  res.json(
    products
  )
}
```

Podemos ver que recogemos de la URL los parámetros “limit” y “from”.

Esto nos sirve para devolver la cantidad de productos deseada por el usuario.
Si limit=15 y from=0 entonces se devolverán los primeros 15 productos.

En caso que no lleguen los parámetros, se devolverán todos los productos existentes.

También podemos recibir parámetros por Body:

```
/**
 * Devuelve todos los productos del usuario
 */
const userProducts = async(req, res = response) => {
  const { user } = req.body

  const products = await Product.find({user})

  res.json(
    products
  )
}
```

Aquí podemos ver que este controlador se encarga de recibir el usuario creador del producto y con esto hace una búsqueda en la DB.

Este controlador devolverá todos los productos que sean de un usuario (del que se nos pasa por el Body).

# Funcionalidades Técnicas

## Backend

### Seguridad

El tema de la seguridad y privacidad de los datos ha sido un factor a tener en cuenta.

Por ello, a la hora de crear un nuevo usuario, su contraseña es encriptada con Bcrypt, una librería que se encarga de encriptar y desencriptar contraseñas.

También ha sido importante la implementación de JSON Web Tokens para realizar según qué acciones, como por ejemplo modificar productos, borrar ofertas, etc…

# Funcionalidades Técnicas

## Frontend

**Conceptos generales**

### 1.- Tratamiento de datos

A lo largo de toda la aplicación se hacen diferentes llamadas al servidor backend.

En ningún momento se tratan datos de ningún tipo en la parte del cliente, todo se hace en el servidor.
Esto se debe a que el principal objetivo es que el lado cliente sea muy liviano y rápido.

### 2.- Clasificación de componentes

A parte de ello, se han clasificado los componentes ("Vistas") en 2 tipos:

- Graphics: Aquí están los componentes reutilizados y usados en varios componentes. Esto se hace con el fin de optimizar recursos y código.
  Ejemplos de ello serían un navbar, spinner, footer…

- Views: Aquí es donde están las vistas del cliente.
  Hay dos tipos de vistas: **Publicas** y **Privadas**.
  Esto se ha hecho con el fin de proteger aquellas vistas privadas que requieran autenticación.

### 3.- Autenticación de rutas

Como se ha comentado en el apartado anterior, la autenticación de las rutas ha sido muy importante.

Por ello se han implementado "Guards", los cuales sirven para comprobar si el usuario tiene permiso de acceso.
En caso que no lo tenga, será redirigido automáticamente al home del sitio.

## 4.- Servicios

A lo largo de toda la aplicación se usan servicios.

Estos servicios son funcionalidades que son reutilizables en toda la aplicación.
Por tema de escalabilidad, organización y recursos se usan los servicios.

Un ejemplo de ello serían las funciones que están en el fichero “utils.services.ts”:

```
public maxShowText(text:string, index:number){
  if (text.length > index) {
    return text.slice(0, index) + ' ... ';
  }
  return text;
}

public reloadComponent(router: Router) {
  let currentUrl = router.url;
    router.routeReuseStrategy.shouldReuseRoute = () => false;
    router.onSameUrlNavigation = 'reload';
    router.navigate([currentUrl]);
}

public calcularPorcentaje(totalPrice:number, offerPrice:number){
  return Math.ceil(((totalPrice - offerPrice) / totalPrice) * 100);
}
```

Estas son funcionalidades generales que se utilizan seguido en muchos componentes.

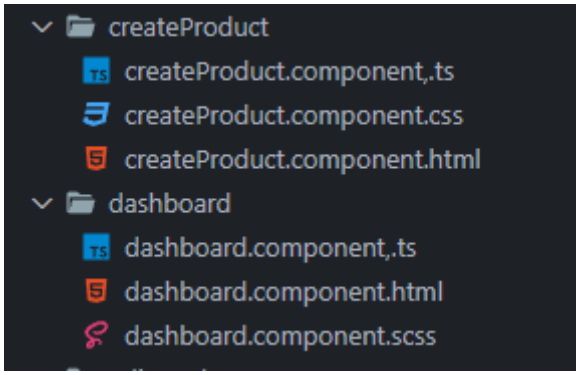
Más adelante se tratará en profundidad los servicios que hay en este proyecto.

## 5.- Hojas de estilo

Por cada componente que lo requiere hay una hoja de estilos (normalmente .css aunque pueden haber .scss).

Esto se hace con el objetivo de modularizar las hojas de estilos.
Se podría decir que estas hojas de estilo están separadas por componentes.

Lo bueno de este procedimiento es que una hoja de estilos de un componente solo influye en su componente respectivamente, y en el resto no.




En los casos en que se necesiten estilos generales para todos los componentes, el fichero de estilos “main” de la aplicación es el que se ubica en “/frontend/src/styles.css”.

Ahí dentro se pueden importar tanto librerías externas como hojas de estilos propias:

```
/* animate.css */
@import '~animate.css/animate.min.css';

/* Importing Bootstrap SCSS file. */
@import 'bootstrap/scss/bootstrap';      Aaron Palacios, hace 3 meses • 2301121624 …
@import '~mdb-ui-kit/css/mdb.min.css';

/* FONTS  */
@import url('https://fonts.googleapis.com/css2?family=Montserrat:wght@800&display=swap');
@import url('https://fonts.googleapis.com/css2?family=Montserrat&display=swap');

/* STYLES */
@import url('./assets/css/effects.scss');
@import url('./assets/css/generics.css');
@import url('./assets/css/home.css');
```

Las hojas de estilo propias se ubican en “/frontend/src/assets/css”.

Estos estilos se aplicarán a todo el proyecto.

### 6.- Recursos en general

El resto de recursos como pueden ser imágenes, svg's o literales (para hacer traducciones) estarán en el directorio "/frontend/src/assets/", el cual es accesible por cualquier componente de la aplicación.

# Funcionalidades Técnicas

## Frontend

**Flujo de datos**

A continuación, voy a explicar el flujo de datos de cada tipo que pasan por la aplicación.

**1.- Datos de autenticación**

Para autenticarse, el usuario tiene 2 opciones:

- Iniciar sesión
- Registrarse

Los dos tienen un formulario donde rellenan los campos necesarios.

Una vez los rellenan, llaman a su respectivo servicio que se ubica en "/frontend/src/app/services/httpServices/user.service.ts".

En concreto cada uno utiliza una función de estas 2:

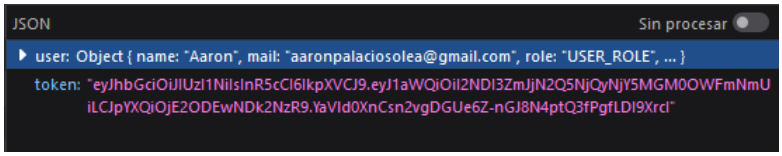
```
private url = 'http://localhost:2022/api/auth';

constructor(private http: HttpClient) { }

register(user: User){
  return this.http.post(`${this.url}/register`, user)
}

login(login: Login){
  return this.http.post(`${this.url}/login`, login)
}
```

Estas funciones se encargan de recibir en un objeto todos los campos del formulario (ya sea el de login o register) y lo envían por petición http al servidor backend, donde este ya realiza las operaciones oportunas.

Si la respuesta que se recibe de backend es satisfactoria, lo que se hace es guardar el objeto que viene en la respuesta de la petición al localStorage:

```
(response:any) => {
  let user = { mail: response.mail, name: response.name, token: response.token}
  this.localStorageService.setItem('userToken', user);
  this.reloadService.reloadComponent.next(true);
  window.history.back(); // Obtiene la URL de la última página visitada
  this.router.navigateByUrl(window.location.pathname); // Navega a la última página visitada
},
```

La respuesta que obtenemos son la información básica del usuario y su JWT:




Ahora, cada vez que se haga una petición al servidor para realizar cambios en productos, ofertas, ver estadísticas y más funciones que requieran autenticación, se enviará el token que hay guardado en el localStorage.

## 2.- Datos de usuario

Si el usuario desea modificar algún dato, entonces necesitará utilizar dos funcionalidades que están en el servicio explicado en el apartado anterior (“/frontend/src/app/services/httpServices/user.service.ts”).

Concretamente estos dos:

```
getUserData(mail: any){
  return this.http.post(`http://localhost:2022/api/users/getUserInfo`, {"mail": mail})
}

editUser(user: any){
  return this.http.put(`http://localhost:2022/api/users`, user)
}
```

Primero tendremos que recoger todos los datos del usuario, por ello se utiliza la función “getUserData” para hacer la petición a back.

El campo que hay que pasarle es el mail del usuario, por lo tanto, la petición le llegará a backend con el mail del usuario en el body.

Una vez que lleguen los datos de vuelta, se rellenarán todos los campos del formulario para que el usuario ya los tenga escritos:

Una vez que el usuario quiere guardar los datos, llama al servicio, concretamente a la función “editUser” y le pasa como parámetro la información del formulario en un objeto:

```
async editUser(){
  this.userHttpService.editUser(this.register) subseribe(
    (response) => {
      window.history.back(); // Obtener la URL de la última página visitada
      this.router.navigateByUrl(window.location.pathname); // Navegar a la última página visitada
      this.notifyToastService.showSuccess("y guardados con éxito.", "Datos editados")
    },
    (error) => { this.notifyToastService.showError("Prueba de nuevo más tarde.", "Ha ocurrido un error e
  );
}
```

```
editUser(user: any){
  return this.http.put(`http://localhost:2022/api/users`, user)
}
```

En general este es el tratamiento de datos en toda la aplicación: recogerlos, enviarlos a backend, recibir la respuesta y mostrarla.

Esto se ha utilizado en el componente “editUser”, el cual está ubicado en “/frontend/sec/app/components/views/private/editUser”.

### 3.- Datos de productos

Hay varios sitios donde se utiliza el siguiente servicio, sobre todo para obtener los productos.

El servicio se ubica en "/frontend/src/app/services/httpServices/products.service.ts".

Uno de estos sitios donde se utiliza este service es en **Marketplace**.
En concreto se utiliza este para obtener todos los productos y mostrarlos:

```
async getProducts(){
  this.productsHttpService.getProducts().subscribe(
    (response) => { this.products = response; },
    (error) => { this.notifyToastService.showError("Prueba de nuevo más tarde.", "Ha ocurrido un error en nuestros se
  );
}
```

```
getProducts() {
  return this.http.get(this.url);
}
```

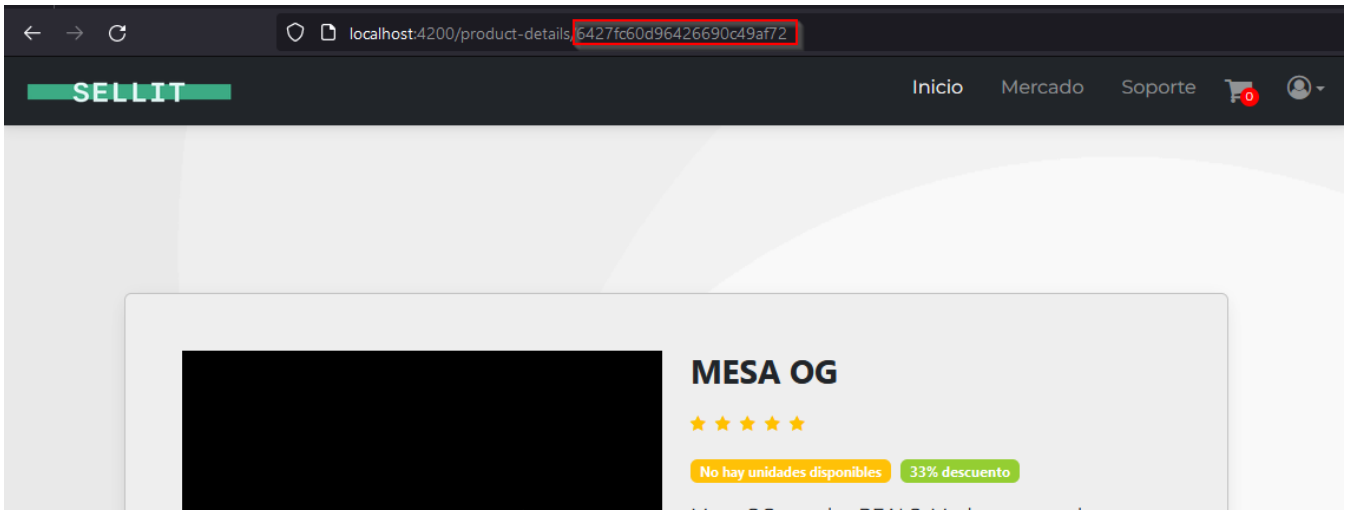
Aquí lo que se hace es realizar la petición y guardar la respuesta de backend (un array con todos los objetos) en una variable llamada products.

Esta luego se muestra en el html recorriéndola con un bucle:

```
<div class="wrapper col-12 col-md-6 col-lg-4" *ngFor="let product of products" routerLink="/product-details/{
  <div class="container">
    <div class="top imgZoom" style="background-image: url('{{product.image}}'); background-repeat: no-repeat;
    </div>
    <div class="bottom">
      <div class="info">
        <div class="details">
          <h1 style="font-size: 30px; width: 200px;">{{this.utilsService.maxShowText(product.title, 10)}}</h1>
          <p *ngIf="product.offerPrice == 0"><a>{{product.price | formatNumber }}</a></p>
          <p *ngIf="product.offerPrice > 0" ><a>{{product.offerPrice | formatNumber }}</a></p>
        </div>
        <div class="addCart">
```

Seré breve con el resto, porque considero que la metodología es exactamente la misma cambiando el tipo de petición.

Estando en la página donde se ven más detalles del producto, para obtener toda la información realizamos una petición con el id del producto a la función/service llamada “getProduct”.




Cuanto se quiera modificar un producto, se rellenará el formulario, y todos los campos se enviarán como objeto al servidor por medio de la función/service “putProduct”.

Lo mismo pasaría al crear un nuevo producto, pero utilizando la función/service “postProduct”.

A la hora de querer ver todos los productos que tiene un usuario, utilizaremos la la función/service llamada “getUserProducts”, donde le pasaremos el id del usuario y su JWT:

```
getUserProducts(user:any, token: any) {
  return this.http.post(`${this.url}/userProducts`, {"user": user}, {
    headers: new HttpHeaders({
      'Authorization': 'my-auth-token',
      'x-token': token
    })
  });
}
```

La respuesta la guardaremos en una variable y se mostrará en el html con un bucle.

### 3.- Resto de datos

El resto de funcionalidades, ya sean los productos, ofertas, ventas y demás se realizan haciendo peticiones, todas exactamente iguales a estas dos mencionadas.

El flujo sería petición del **Componente** al **Service**, y este se encarga de hacer la petición HTTP a **Backend**.
La respuesta vuelve al **Componente** y ahí se tratarían los datos recibidos.

# Funcionalidades Técnicas

## Frontend

### Carrito

El carrito de la compra puede ser utilizado tanto si estás logueado como si no.

Este se guarda en el LocalStorage utilizando el service “/frontend/src/app/services/ /localStorage.service.ts”.

### Imágenes

Todas las imágenes de los productos son almacenadas en un Drive.

Cuando creas o editas un producto, este te da la opción de subir una imagen:


Una vez cargada la imagen, se utiliza el service “/frontend/src/app/services/ /httpServices/uploadFiles.service.ts”.

Este se encarga de enviarlo a backend, y ya allí se almacena la imagen en el drive.

Y lo mismo cuando estamos mirando un producto: la imagen que se muestra viene de Drive.

# Funcionalidades Técnicas

## Frontend

### Formularios reactivos

Para los formularios FormBuilder, FormGroup y Validators de Angular Forms.

Estos permiten crear formularios reactivos, y que cada campo pueda tener las validaciones que queramos.

```
this.registerForm = this.formBuilder.group({
  name: [this.register.name, [Validators.required]],
  mail: [this.register.mail, [Validators.required, Validators.pattern('[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,3}$')]]
  password: [this.register.password, [Validators.required]],
  password2: [this.register.password2, [Validators.required]],
  surname1: [this.register.surname1, [Validators.required]],
  surname2: [this.register.surname2, [Validators.required]],
  telephone: [this.register.telephone, [Validators.required]],
  gender: [this.register.gender, []],
  address: [this.register.address, [Validators.required]],
  zipCode: [this.register.zipCode, [Validators.required, Validators.pattern("[0-9]{5}")]],
  region: [this.register.region, [Validators.required]],
  country: [this.register.country, [Validators.required]],
  years: [this.register.country, [Validators.required]],
  billingAddress: [this.register.billingAddress, [Validators.required]],
  billingZipCode: [this.register.billingZipCode, [Validators.required, Validators.pattern("[0-9]{5}")]],
  billingRegion: [this.register.billingRegion, [Validators.required]],
  billingCountry: [this.register.billingCountry, [Validators.required]]
});
```

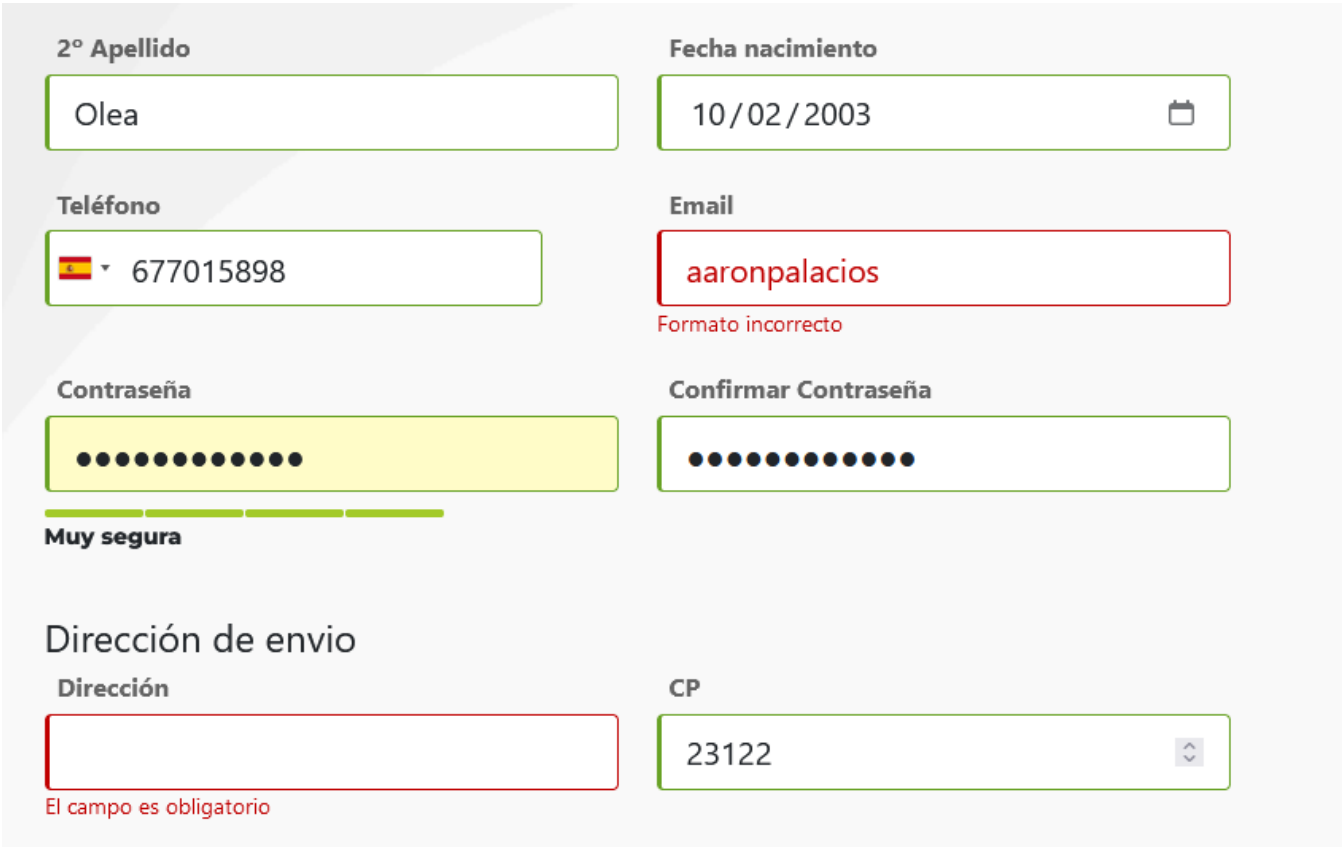
Lo interesante de esto es que puedes crear tus propios validadores para validar los campos.

Y aquí viene el punto más importante, esta función:

```
      billingZipCode: [this.register.billingZipCode, [Validators.required, Validators.pat
      billingRegion: [this.register.billingRegion, [Validators.required]],
      billingCountry: [this.register.billingCountry, [Validators.required]]
    });
  }
  this.service.gestionarValidarErrors(this.registerForm);
}
```

Esta función se encarga de comprobar campo por campo si se cumplen las validaciones que se hayan puesto.

En caso que se cumplan, marcará el campo en verde, y sinó, mostrará un mensaje de error:



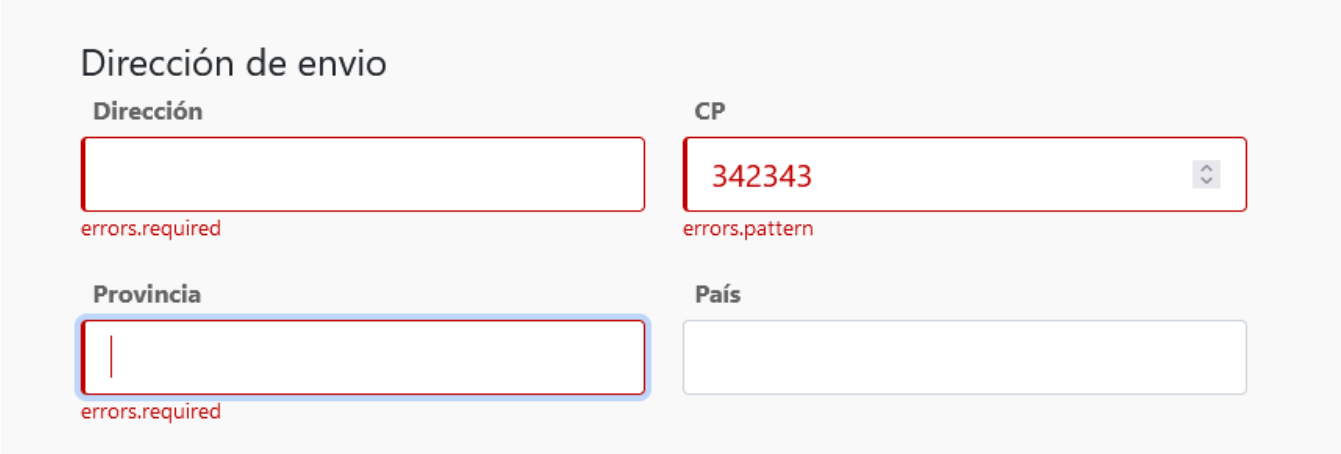
¿Cómo funcionan los mensajes de error?

Tanto si la validación es una que nos ofrece Angular o si es una que hayamos creado nosotros, si no se cumple siempre devuelve un error.

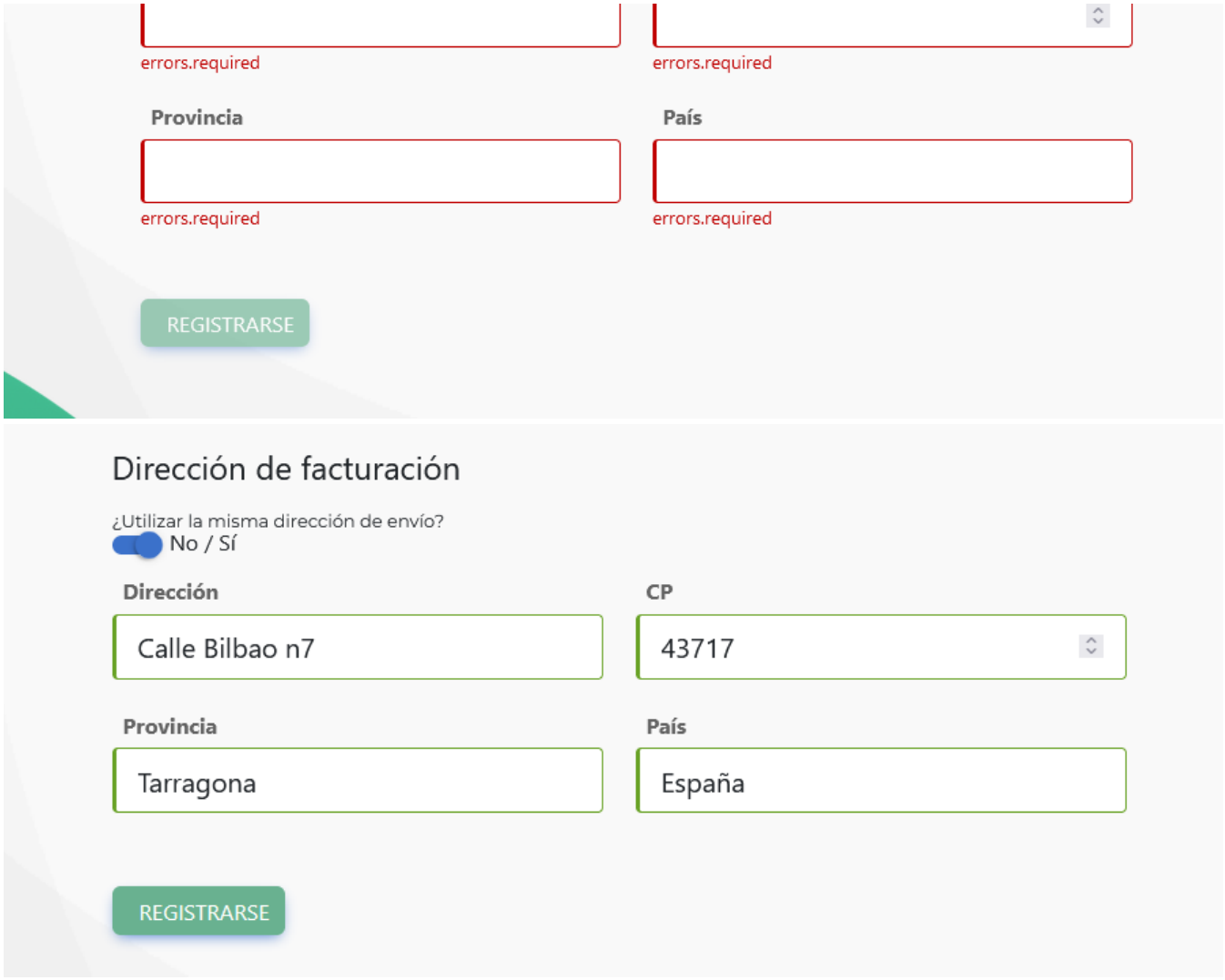
Ahí entran los literales. Con cada código de error, podemos crear un literal y añadirle el mensaje que queramos:

```
"errors": {
  "required": "El campo es obligatorio",
  "email": "Formato incorrecto",
  "ageError": "Edad no permitida",
  "max": "Cantidad máxima sobrepasada",
  "pattern": "Formato incorrecto",
  "number": "El camp tiene que ser numérico",
  "notSamePassword": "Las contraseñas no coinciden",
  "senseMissatge": "",
```

Si no tuviésemos literales, los errores se mostrarían así:



Y para acabar, si alguno de los campos no es válido, el botón submit se deshabilita:



# Tests de Errores

A lo largo de todo el proyecto he utilizado Postman para comprobar que resultados retorna backend.

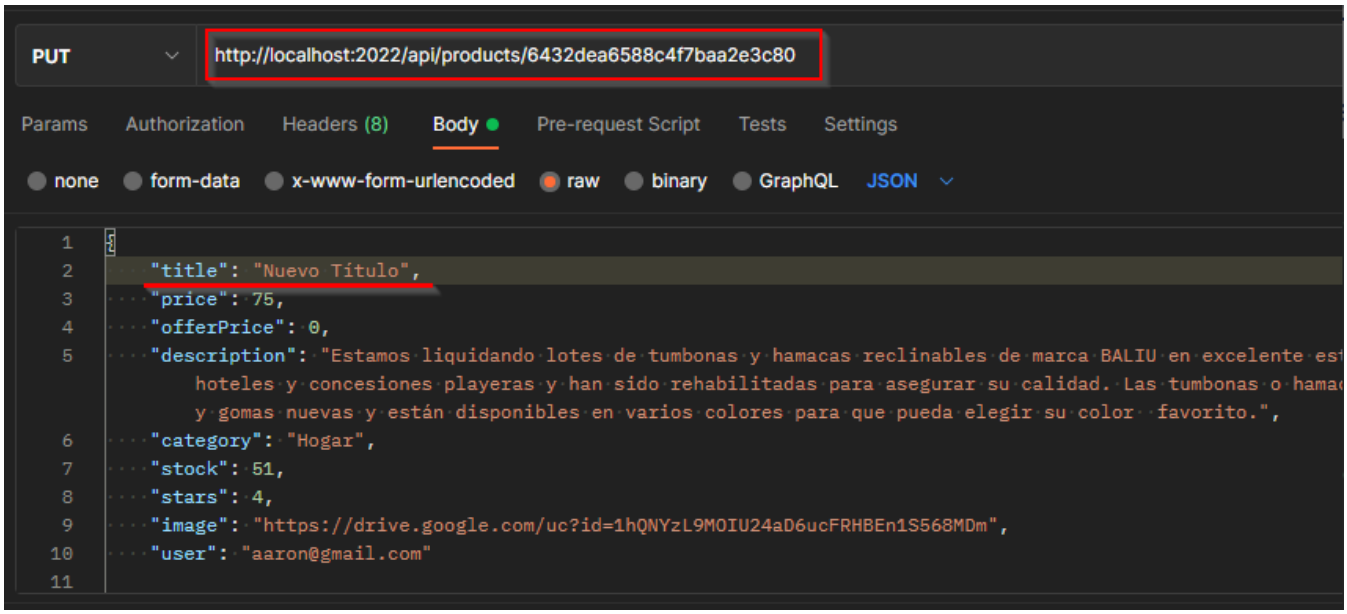
He aquí unas cuantas pruebas:

## 1.- Intentar modificar un producto de un usuario pasando un JWT invalido

Este es el objeto que está guardado en la DB:

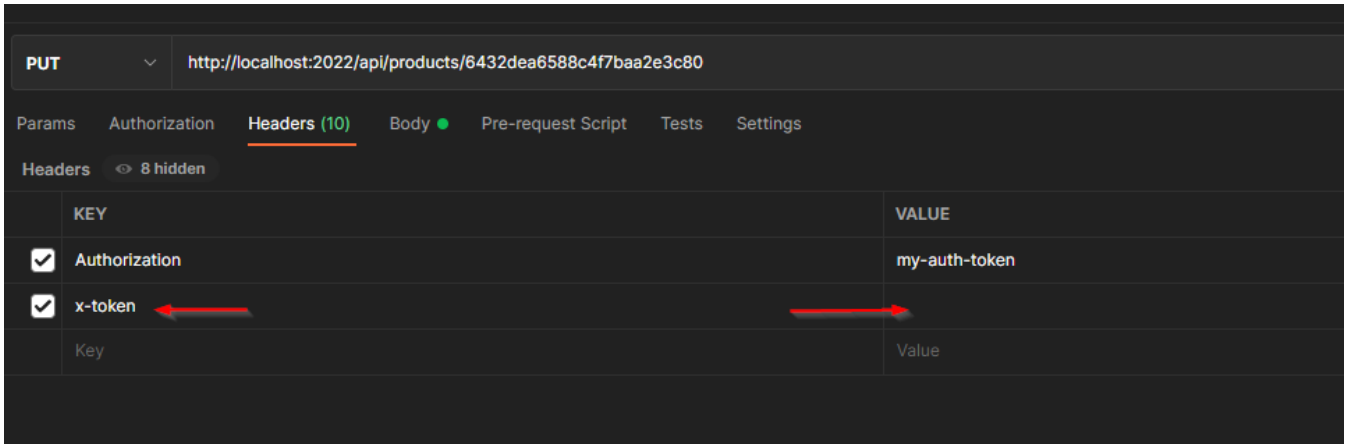
```
1   _id: ObjectId('6432dea6588c4f7baa2e3c80')                                  ObjectId
2   title: "Tumbonas de piscina jardín o playa"                                String
3   price: 75                                                                  Int32
4   offerPrice: 0                                                              Int32
5   description: "Estamos liquidando lotes de tumbonas y hamacas reclinables de" String
6   category: "Hogar"                                                          String
7   stock: 51                                                                  Int32
8   stars: 4                                                                   Int32
9   image: "https://drive.google.com/uc?id=1hQNYzL9MOIU24aD6ucFRHBEn1S568MDm"  String
10  state: true                                                                Boolean
11  user: "aaron@gmail.com"                                                    String
12  __v: 0                                                                     Int32
```

Voy a intentar modificarlo en Postman haciendo un put aquí:




Hay que tener en cuenta que requiere el JWT:

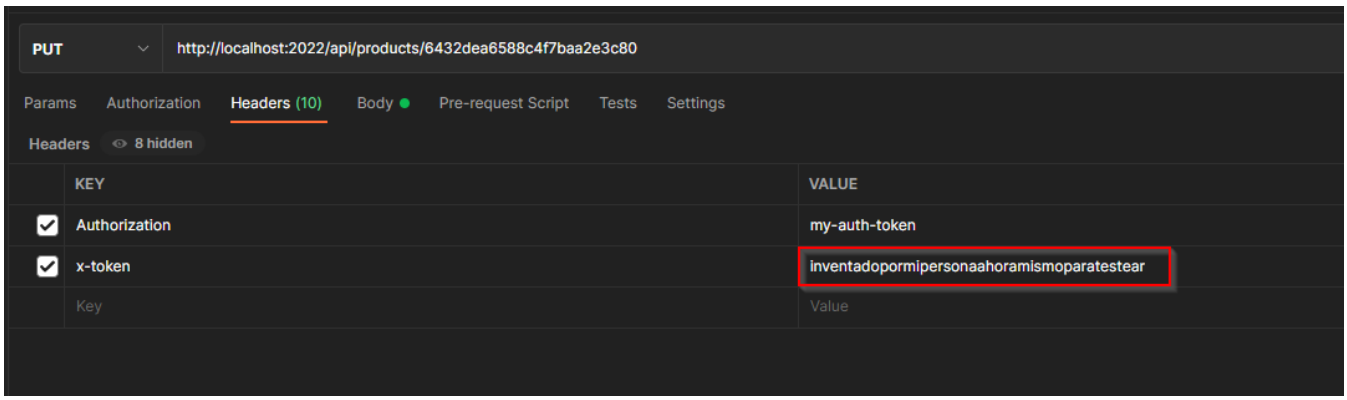

Para ello no le voy a pasar ningún Token por la cabecera:



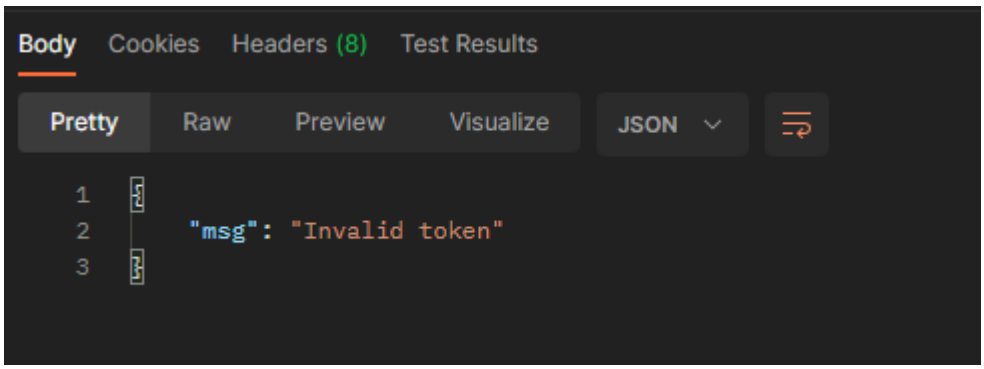
Vamos a ver qué pasa:


Nos indica que no hay ningún token.

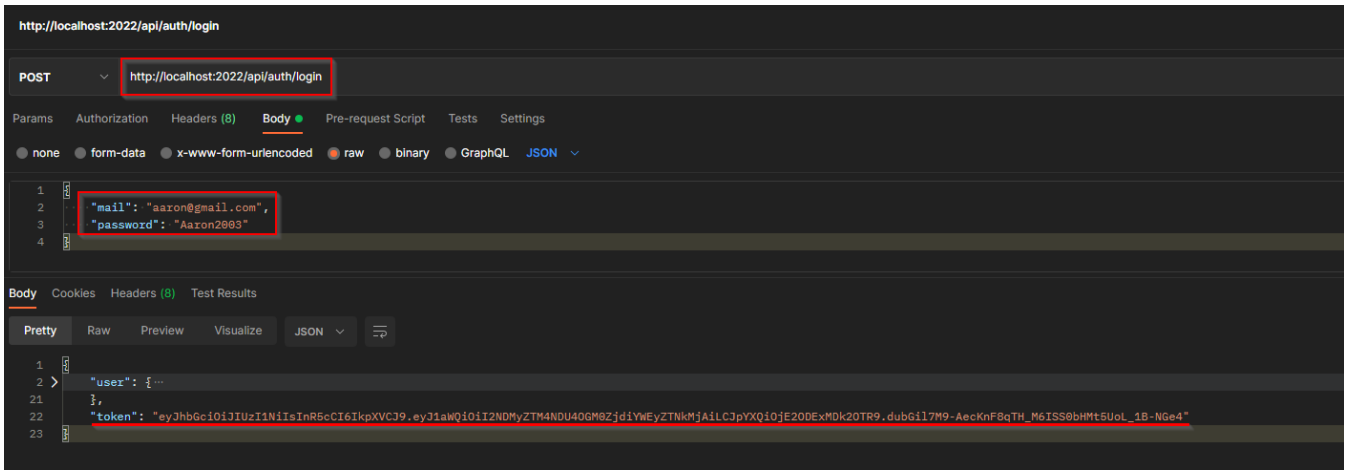
Ahora voy a poner uno inventado:

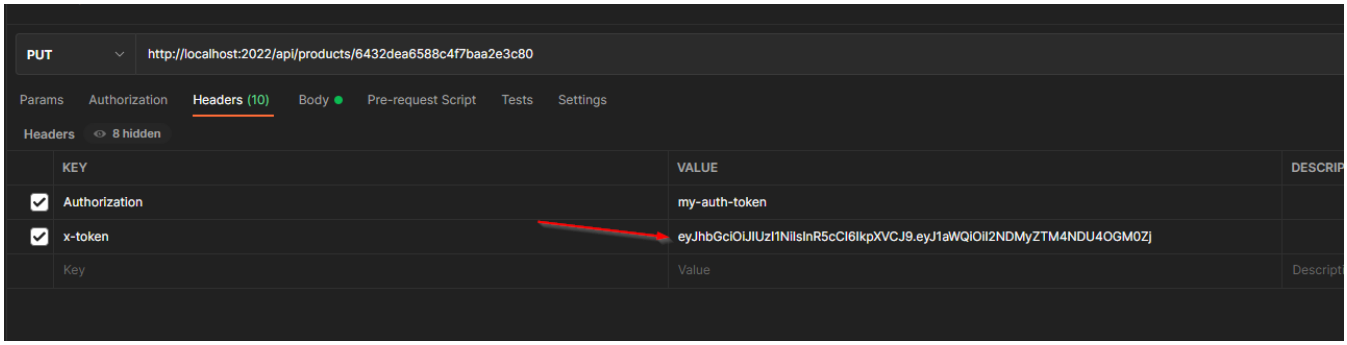



Lo envío y me devuelve esto:

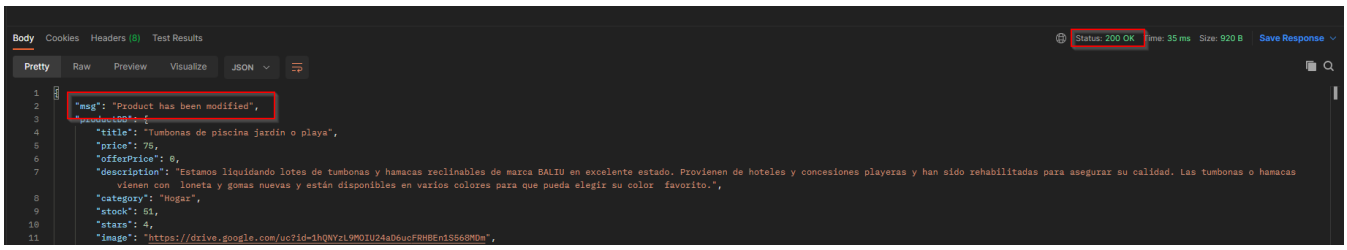


## Ahora voy a hacer login para obtener un token y voy a probar de nuevo:

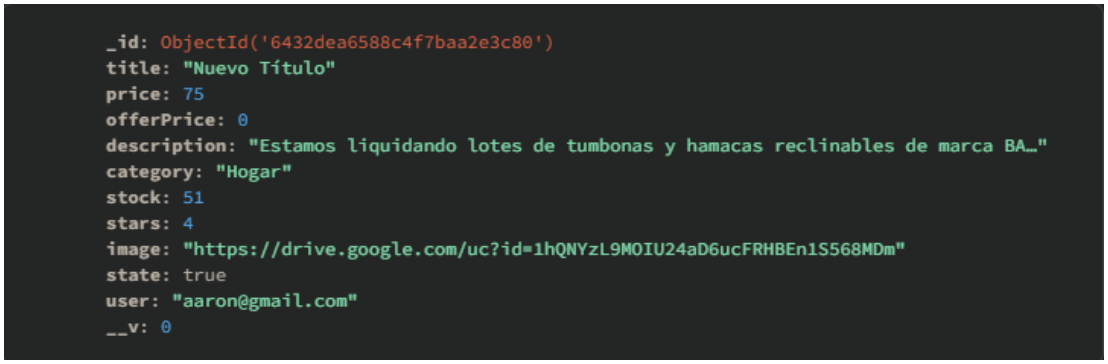



## Hacemos la consulta y ya se modifica:

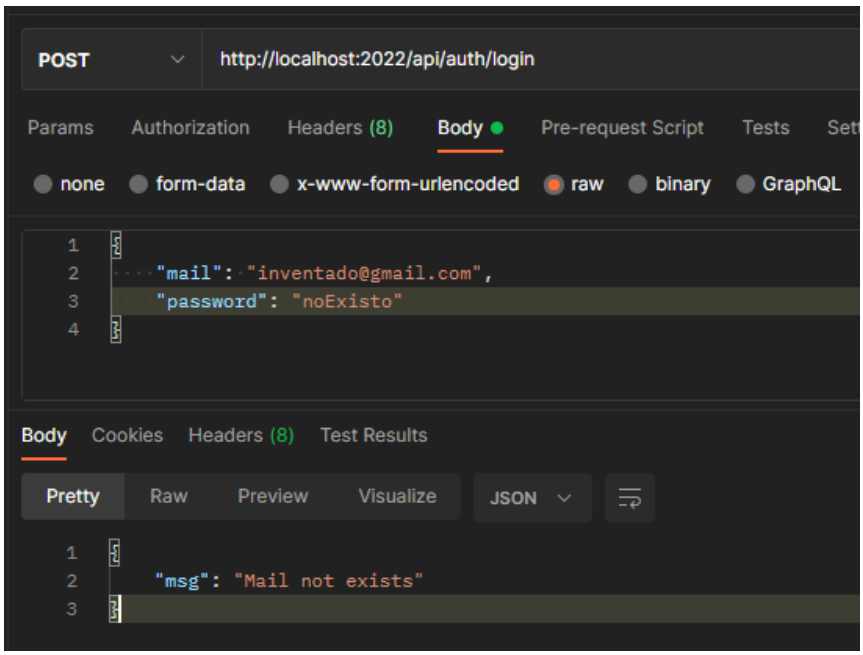



## Chequeamos en la DB:



## 2.- Autenticarse con algún campo inválido

Con mail que no existe:




Con password incorrecta: