Backend de fotosgram

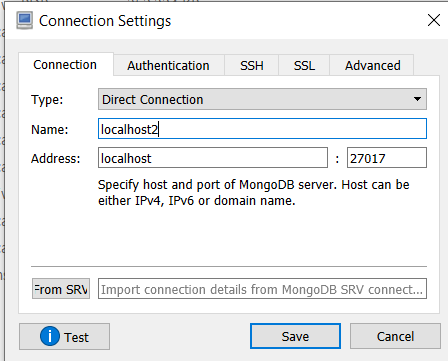
Crear Usuario, Login, Actualizar.

Instalando mongoDb

<https://www.mongodb.com/download-center/community>

instalar y luego abrir y ejecutar  
C:\Program Files\MongoDB\Server\4.0\bin  
mongo.exe

Gestor de base de datos robo mongo  
<https://robomongo.org/>

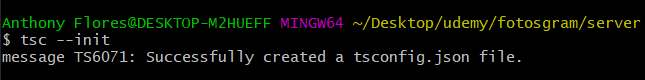


Creamos una carpeta para almacenar el proyecto del servidor, ejecutamos la terminal e iniciamos el proyecto de node con npm init:

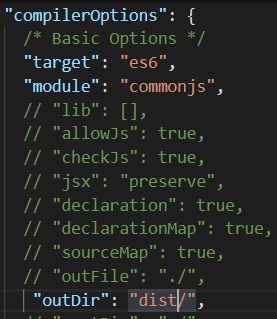
Lo que nos crea un package.json



Ahora creamos un archivo index.ts  
Como prueba escribimos un console log

Corremos el comando tsc –init que crea un archivo de inizializacion de typescript.  


En este archivomodificamos el outDir



Luego en la consola escribimos tsc -w para que entre en modo observador typescript

En otra terminal:

Instalaremos nodemon para estar escuchando los cambios y poder mostrarlos automáticamente.



Y corremos nodemon dist

Instalamos los paquetes necesarios:

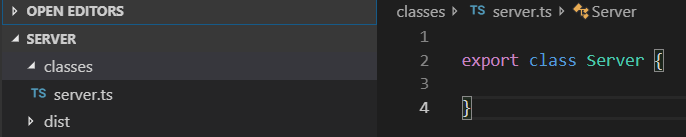
**npm install express:**Crea un servidor web y lo necesario paramontar un servidor rest  
  
**npm install body-parser**Recibir la información de post y transformarla en un objeto de javascript   
  
**npm install cors**Permite hacer peticiones CrossDomain

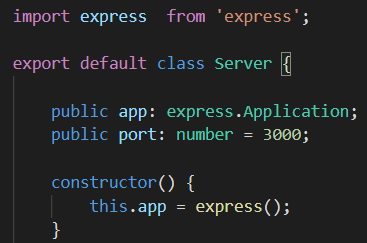
**npm install mongoose**Trabajar con el modelado de datos del lado de node e interactuar con la base de datos. **npm install express-fileupload**Recibir las peticiones de imagenes que se postean  
 **npm install jsonwebtoken**Autenticacion de la aplicación, creando web tokens para mantener de forma pasiva la sesión del usuario. **npm install bcrypt**encriptar las contraseñas de los usuarios.

npm install express body-parser cors mongoose express-fileupload jsonwebtoken bcrypt

Class Server, crea una instancia de express

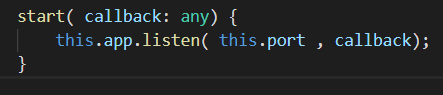
Creamos una carpeta classes y dentro un archivo clase llamado server



Necesitamos instalar express para poder importarlo en la clase  


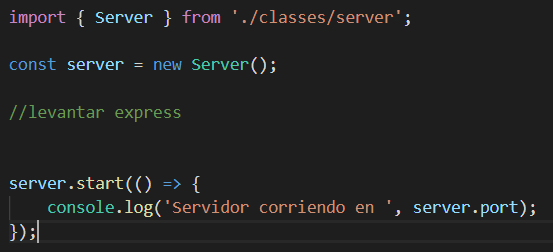
Como es una dependencia de desarrollo no es necesario que este cuando se despliegue a producción y para esto el –save-dev  

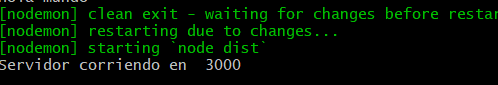

Creamos ahora una función para iniciar el servidor que reciba el puerto y una función callback



Y desde el index.ts

Importamos la clase servidor y creamos una constante que será inicializada con el constructor de la clase, permitiendo ejecutar la función start antes creada y levantando el servidor.



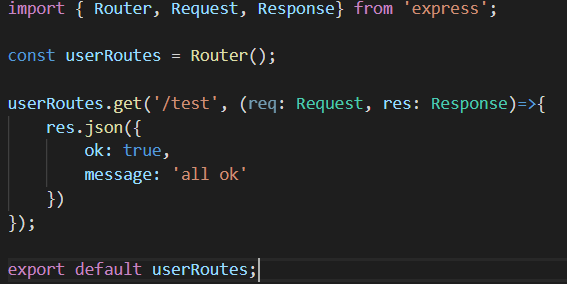


Creando servicio REST en express

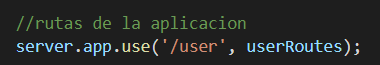
Creamos una carpeta routes y dentro el archivo routes.ts



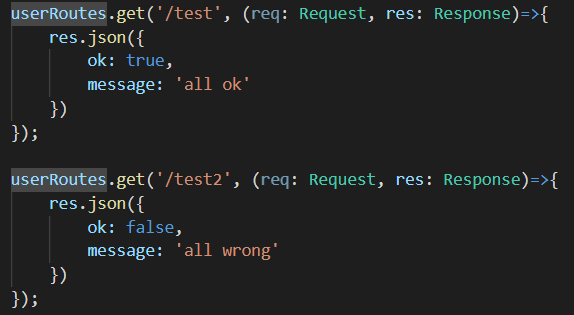
Donde creamos una constante de tipo Routes, y luego creamos la función get, con su dirección como parámetro, y el segundo parámetro la función flecha que recibe un request y responde el response.  
Recordando exportar las funciones creadas.

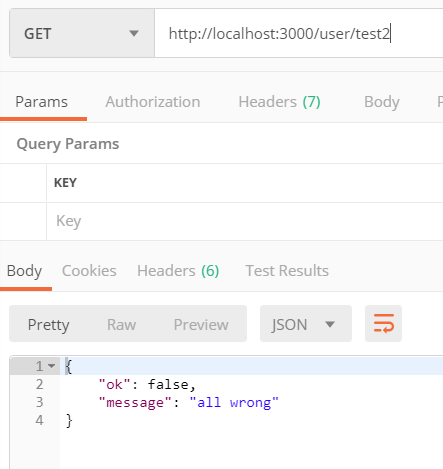


Y en el index.ts empezamos a escuchar la función creada



En donde una ruta puede tener varias funciones



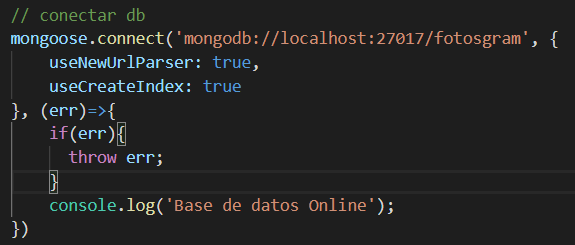


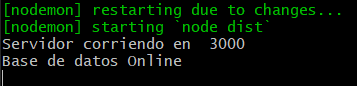
Levantando la base de datos de mongodb y conectar la aplicación.

Tener mongod ejecutado y abrir robo mongo, estar atento al puerto.

En el index debemos importar el mongoose. Pero hay que instalarlo.

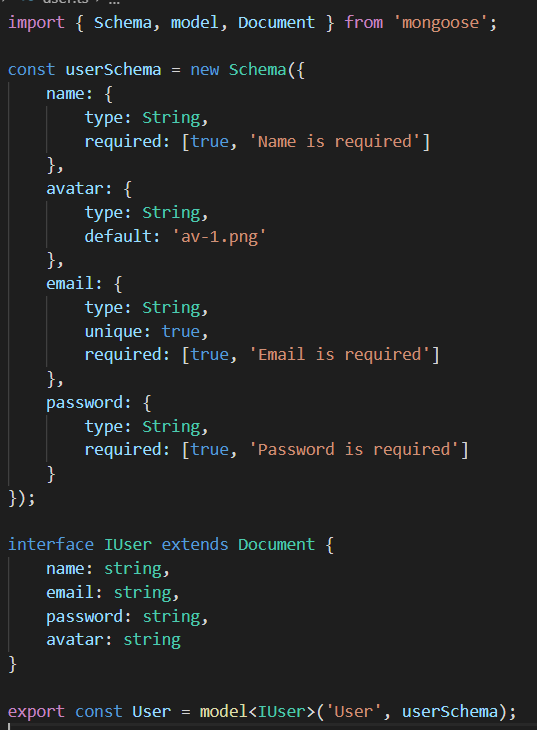




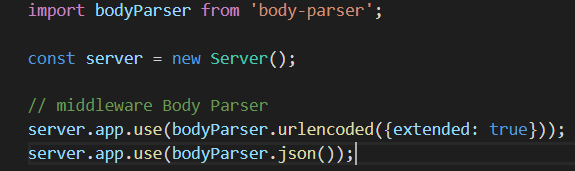


Modelo de base de datos del usuario  
En donde creamos la constante userSchema con el constructor de Schema que es necesario para crear el modelo de la base de datos con que trabaja mongo.  
Declarando luego los atributos y dentro de ellos un objeto que refiere el tipo, si es requerido, su mensaje de error y por si es default un valor.

Creamos una interfaz para poder tipar el objeto.  
y finalmente exportamos la el modelo con la sintaxis final.

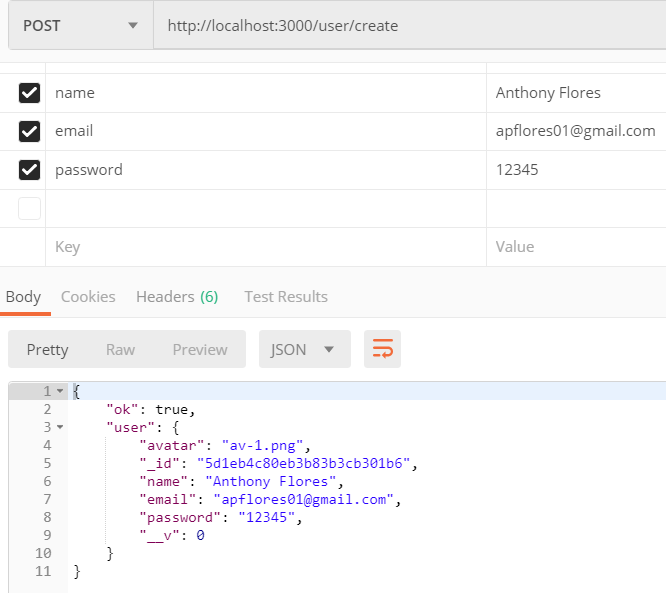


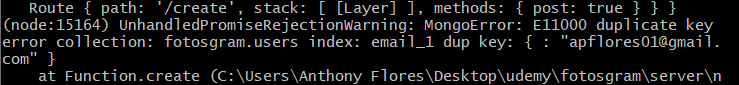
Creando un usuario

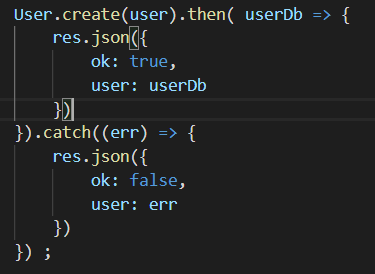
Es necesario pasar por un middleware el server  
Es una función que se ejecuta antes de otras, en este caso siempre se va a ejecutar.  
En el index configuramos el body parser el cual permitirá convertir el request a un objeto json. Utilizando el ---xenconded  


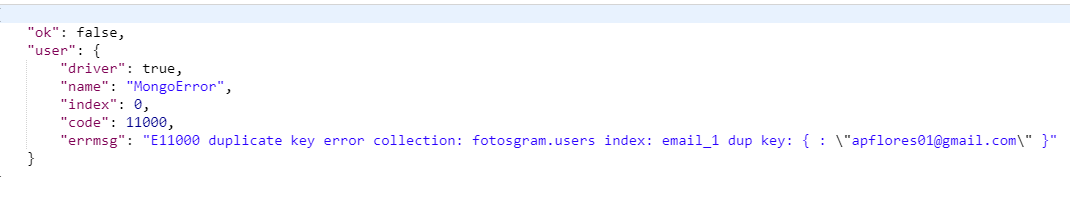
Y creamos la función post de crear el usuario, que crea un objeto user con el request.body y sus propiedades mandadas.  
Para luego con el modelo de la base de datos (importandolo), crear el objeto en la base de datos mediante la función flecha.



  
Y verificando en el robomongo esta creado

Si intentamos crear nuevamente el mismo usuario, nos manda un error ya que no estamos manejando el reject  






Encriptando la contraseña del usuario





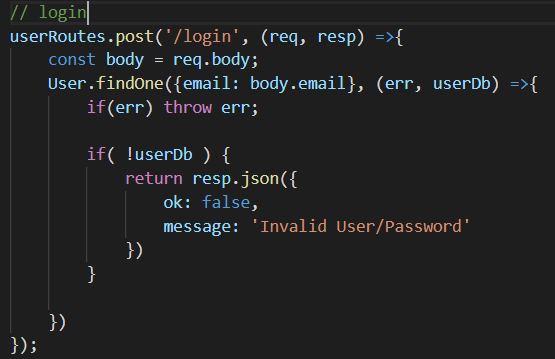
Pasamos la contraseña por el bcrytp.





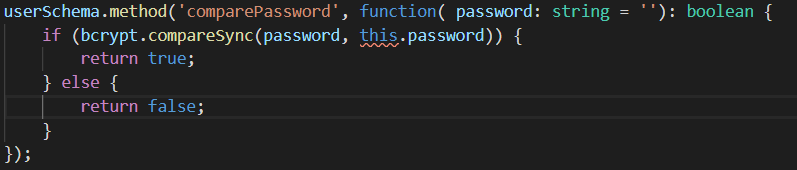
Login de usuario

Se declara la constante body con la respuesta del request.  
Se utiliza el modelo de usuario de la base de datos para buscar un objeto que tenga el correo, si ocurre un error sale, si no existe el usuario, manda de respuesta el objeto con el mensaje debido.



Si encuentra el correo debe comparar la contraseña ingresada con la que esta registrada, esto se hace desde el modelo.

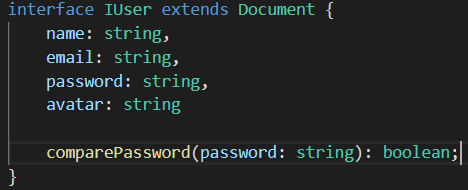
Importando el bcrypt y creando la función (NO FLECHA) que recibe la contraseña y retorna un booleano.



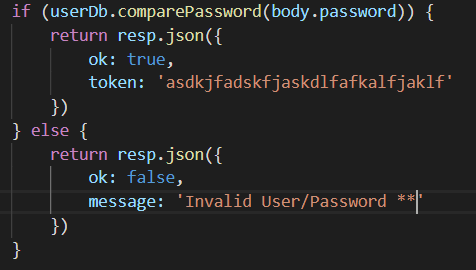
Para quitar el error del this, en el tsconfig descomentamos el noImplicitThis paraque no de error cuando este relacionado a un any…?



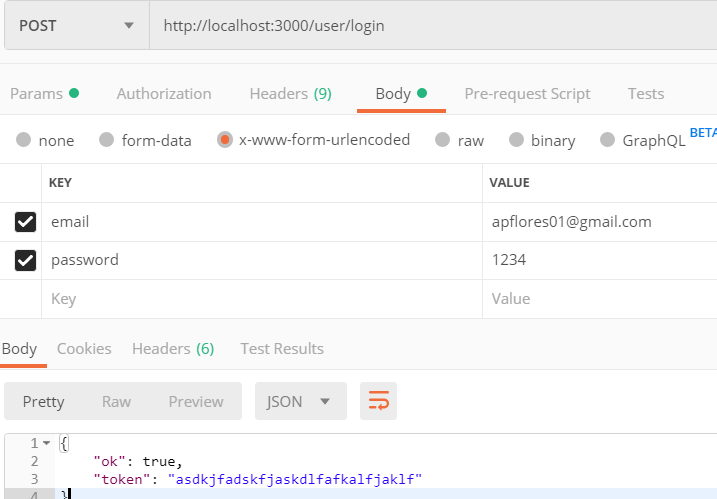
Luego esta función la declaramos en la interfaz, para poder utilizarla fuera.



Y desde el user routes seguimos con la validación de la contraseña utilizando la función que creamos en el modelo.



Recordar enviar en el postman x-www-form-urlencoded

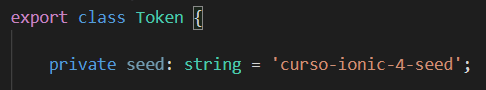


Creando un json Web Token

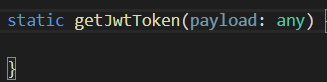
Creamos una clase dentro de la carpeta classes llamada token, y será estatica.  
Por lo que lego no se va a necesitar hacer   
const token = new Token();

Solo: Token.something

En donde crearemos una variable seed la cual será una clave secreta que solo debemos tener para solo nosotros poder abrir y firmar los sockets.

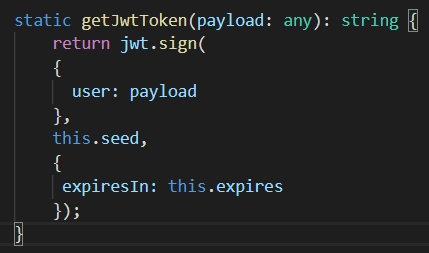


Creamos la función getJwtToken la cual recibe un payload que es donde viene el id del usuario donde se puede comparar con la información almacenada.



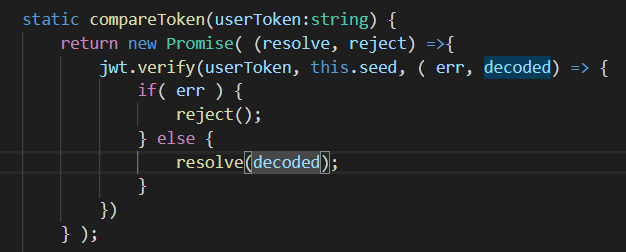
Ahora debemos instalar jsonwebtoken para poder importarlo en la clase  
  


En la función pasamos un objeto de usuario con el payload, la clave secreta (seed) y la expiración del token.



Luego debemos comparar el token que estamos recibiendo mediante la función verify, la cual recibe el token actual del usuario y utiliza la función verify para comprobarlo pasandole el token y el seed.

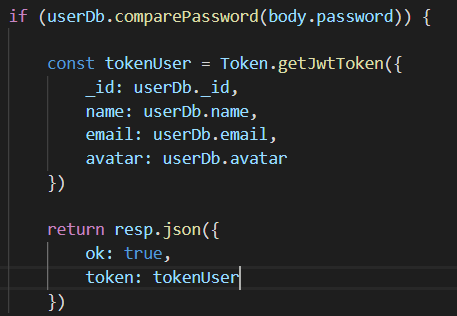
Esta comparación la metemos dentro de una promesa, para evaluar los errores, y en caso de que tenga un error ejecutar la función reject de la promesa, y de ser correcta el resolve.



Llamando a la clase token para generar un token valido luego de iniciar sesión

Desde user.routes en la función login.

Si al comparar la contraseña y esta es correcta, creamos una constante token que es igual a la clase creada de token. Donde llamaremos a la función creada de getjwtToken y pasandole el objeto con la información del usuario.

Y finalmente se le envia en la respuesta el token generado.  


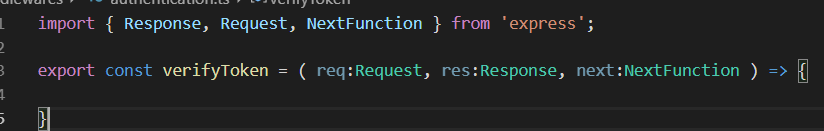
Esto hay que repetirlo cuando creamos un usuario

Validando el token al hacer un llamado a un endpoint

Para ejecutar esta validación de token antes de llamar un endpoint creamos un middleware que será llamado antes del endpoint.

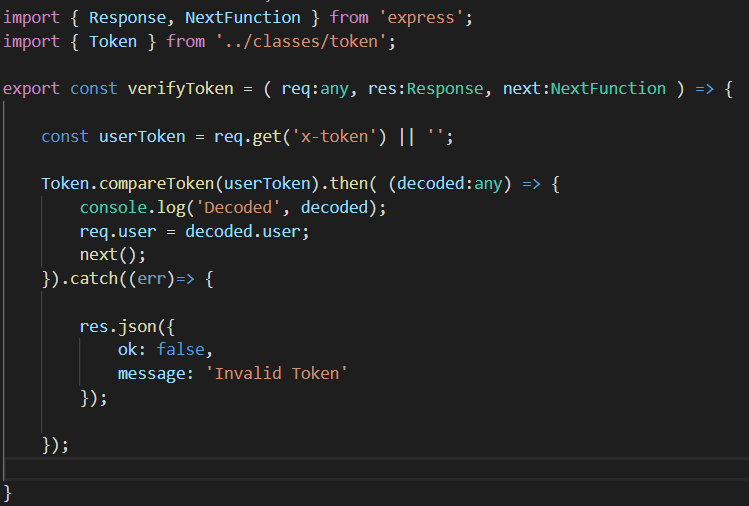
Creamos una carpeta middlewares.

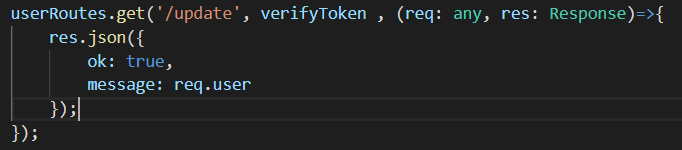


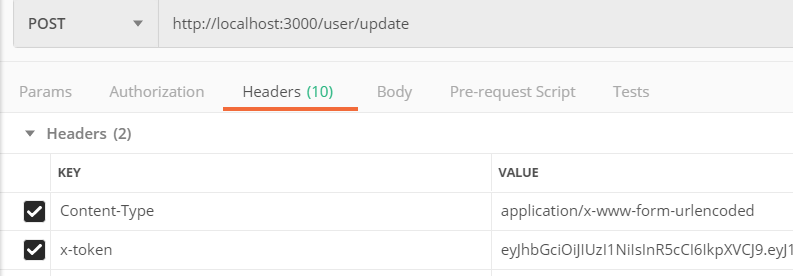


Vamos a recibir el token por los headers con una propiedad personalizada x-token, si esta no existe en el request lo asignaremos como un string vacio.  


Llamamos a la clase token y la función de compare, pasandole como parámetro el valor del token del header recibido en request. En donde al responder asignamos al valor a decoded de tipo any. Luego reasignamos el valor de request user con el valor que nos retorna el decoded para poder utilizarlo en el back y realizar la acción de actualizar.



Y para implementarlo simplemente se añade como parámetro en el endpoint. De ser varios middlewares se hace el segundo parámetro un arreglo y dentro los middlewares.  


Probando en el postman  




Actualizando base de datos

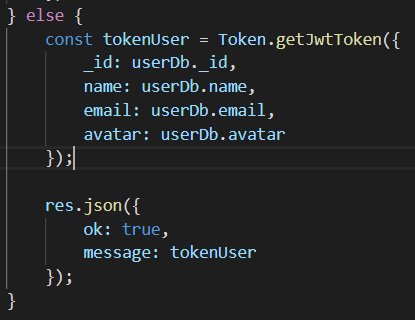
Se crea una constante user que recibe del middleware la respuesta, en este caso el req.user y sus datos, se evalua el body por si existe en la petición.

Luego se accede al modelo de la base de datos para que busque por id y actualice con los parámetros del id del usuario, el objeto user para actualizar , y el objeto new para que la respuesta sea el usuario actualizado.

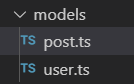
El ultimo parámetro es la función flecha que tiene el error o la respuesta correcta de la bd.  
Si no existe el usuario en la base de datos se responde en el mensaje



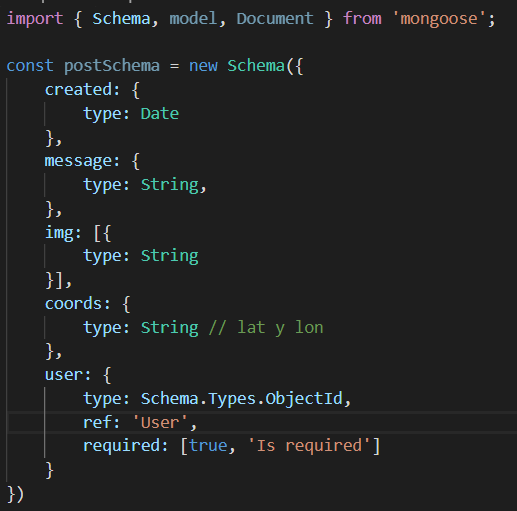
Ahora si encuentra al usuario, se debe generar otro token ya que la información que esta almacenada ha cambiado.



Modelo de base de datos de un post  
Creamos un archivo para el modelo de post.

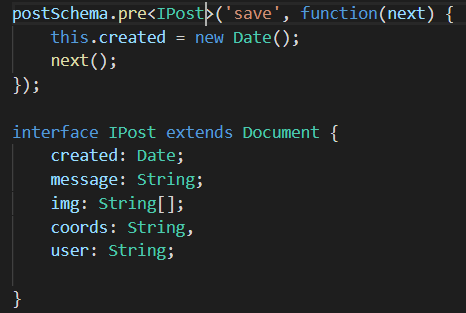


Creamos el esqueleto que contendrá la estructura de un post. Teniendo un usuario como parte y definiéndolo como tipo objeto de esquema y haciendo referencia al modelo de usuario.



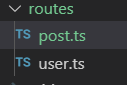


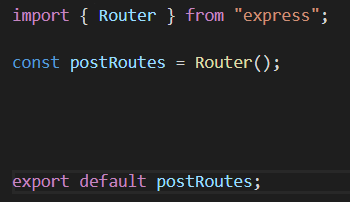
Luego definimos una función que se ejecuta antes de guardar el post para obtener la fecha de creación del post.



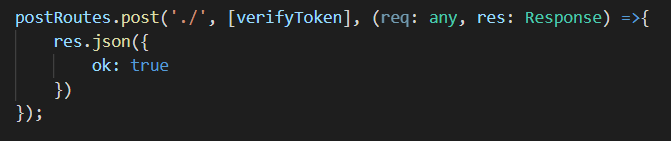
Y finalmente exportamos el modelo  


Servicio para crear un post

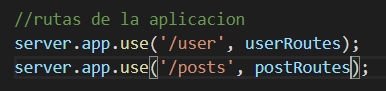
Creamos el archivo de post en routes  




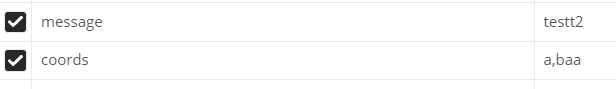
Creamos la estructura del método.



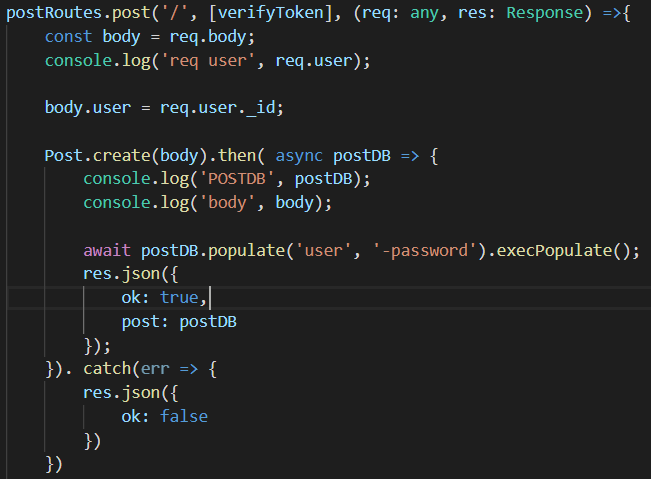
Luego en el index.ts se define la ruta.



Creamos el método completo.

Recibimos el req que ya paso por el verify token lo que hace un decode del token y obtenemos el usuario. El req body es la petición que se hizo en el postman  


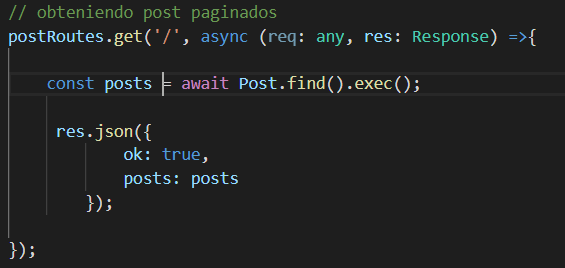
Luego se crea la propiedad user en el body y se le asigna el id del usuario que viene desde el middleware.  
  
Utilizamos el modelo Post creado para crear el registro en la base de datos, haciendo un proceso asíncrono con la respuesta postDB y que espere que se realice el populate de la propiedad user y que no responda la propiedad password.



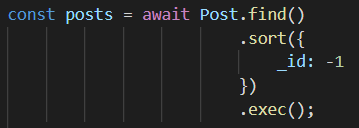
Investigar la relación entre el populate, execPopulate y que la propiedad user quede con los datos del usuario en vez del id de usuario.

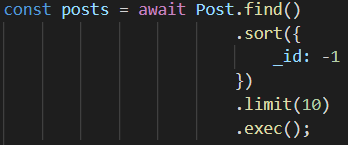
Obteniendo los post de forma paginada.

Creamos 20 registros manualmente para tener datos de prueba para la paginación.  
Para traer todos los registros el método es:  
Donde creamos una constante que recibe el find del modelo de post, que es asíncrono.

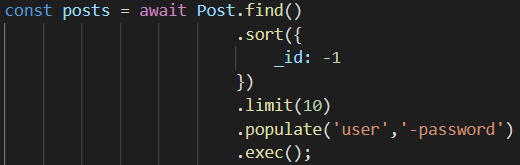


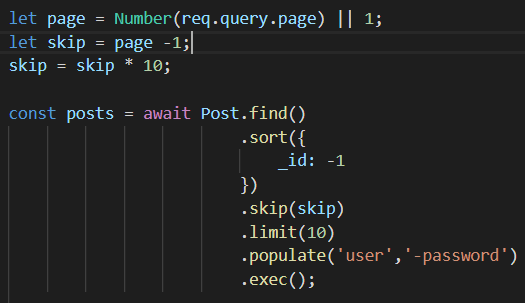
Para mostrar los resultados de manera descendiente



Y para mostrar solo 10 registros  


Mostrando la info del usuario

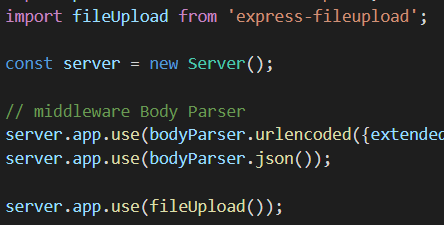


Mostrando por paginación  
Creamos la variable skip que tendrá la pagina enviada desde la aplicación la cual será restada de 1.  
luego se multiplica por 10 para calcular la cantidad de registros a saltar y que muestre los siguientes 10.  


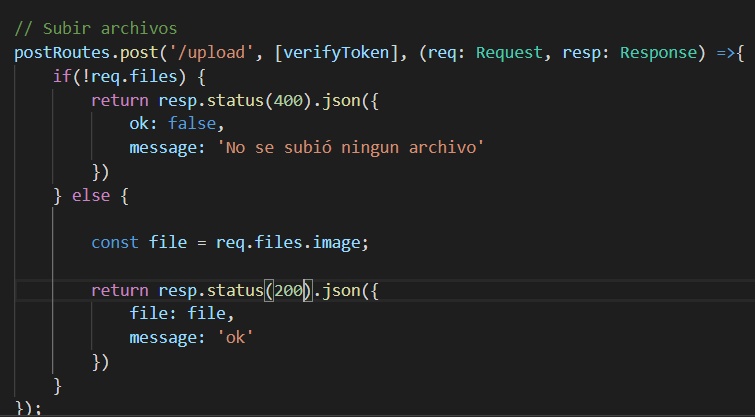
Servicio para subir archivos

En el index.ts

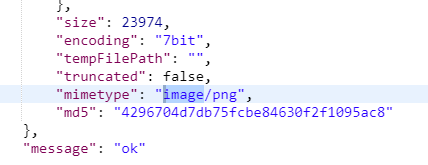
Importamos el fileupload, hacemos la instalación npm necesaria y lo declaramos en el se del server.app.



Y desde el archivo de rutas

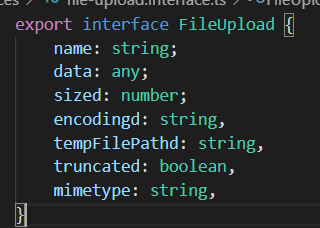


Al subir un archivo en postman nos envia la información de la imagen

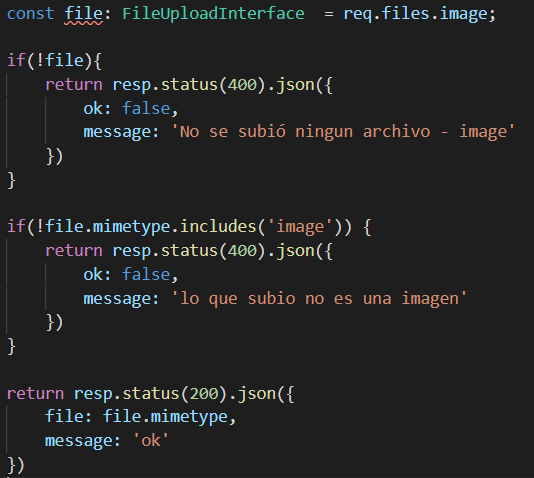


Hay que trabajar con estos atributos.

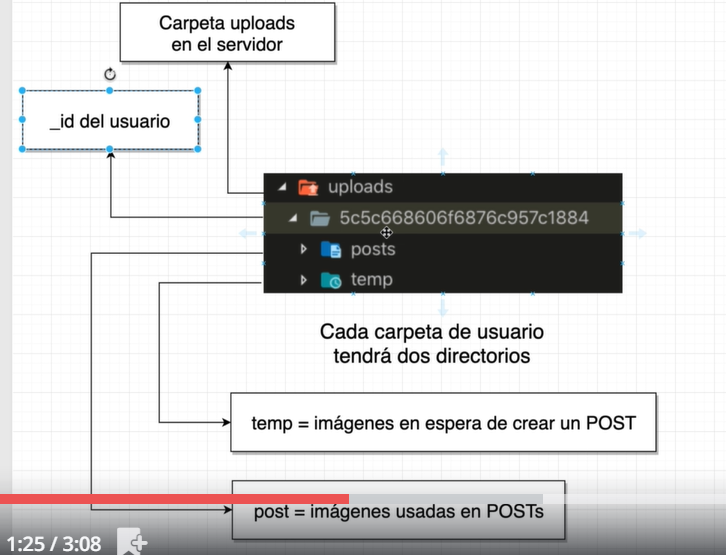
Creamos una carpeta de interfaces



Y dentro de la función en rutas



La estructura de los archivos subidos estarán encabezados en la carpeta uploads con el id del usuario seguido por las carpetas que contienen su info.



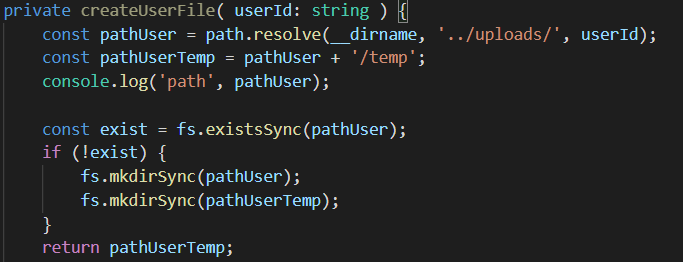
Manejo del FileSystem

Creamos la carpeta uploads y un archivo index, el cual estará vacio, solo para que se transpile la carpeta y sea creada dentro de dist.

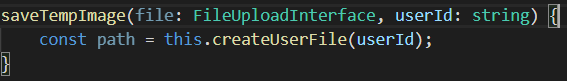


Creamos dentro de la clase una función que se encargue de crear las carpetas.  
Recibiendo el id de usuario, luego importando  
path from path y fs from fs.  
con path y los parametos usados creamos el path en el directorio deseado (\_\_dirname obtiene el directorio actual del proyecto en node).

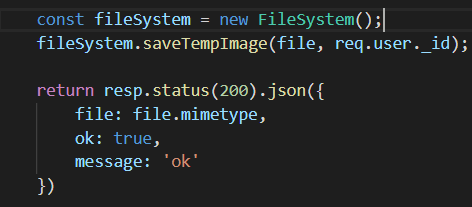
Luego con fs verificamos si existe el path y si no , creamos los directorios.  
retornando finalmente el pathUserTemp.



Creamos la función de guardar la imagen llamando al método de crear el path.

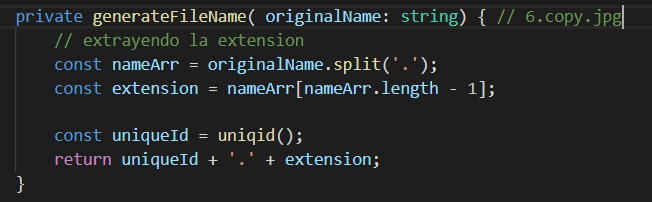


Llamamos al método de guarda imagen desde el routes. Creando un objeto de la clase FileSystem.

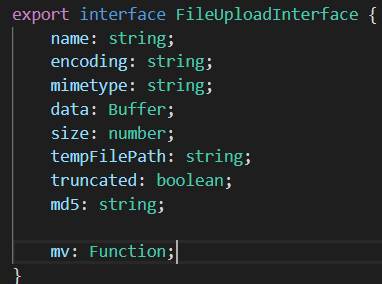


Generar un nombre único a la imagen

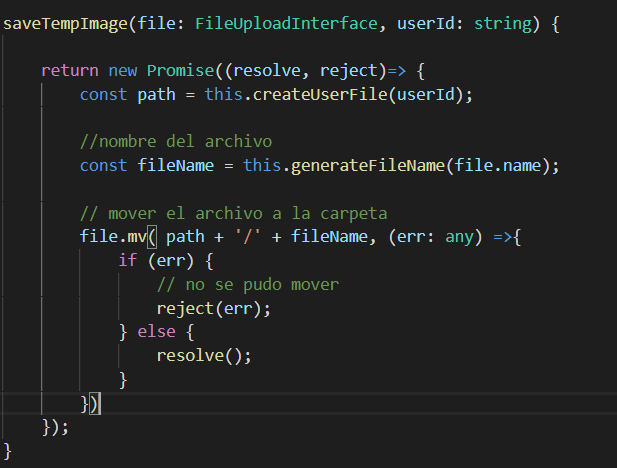
 luego instalar con el type y el –save-dev



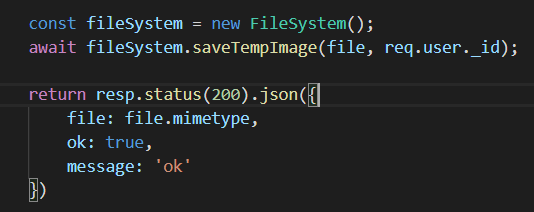
Moviendo el archivo físico a la carpeta creada.  
En la interfaz creamos otro atributo que será una función.



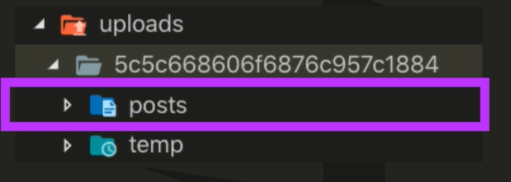
Ahora en el file-system.ts hacemos que la función retorne una promesa para poder hacer el eject y resolve.  
utilizando la función mv creada en la interfaz, le pasamos como parámetro el path con el nombre del archivo, y una función flecha donde se controla la respuesta de la promesa.  
Investigar…como la función mueve el archivo.



Y en el post.ts llamamos a la función con sus parámetros haciendo un await para esperar la respuesta de la promesa.

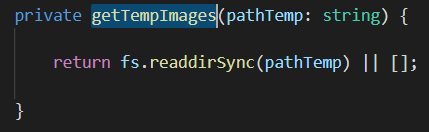


Moviendo la imagen desde temp hacia la carpeta post.



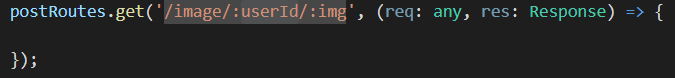
Creamos en el filesystem la función para mover la imagen a la carpeta de post.



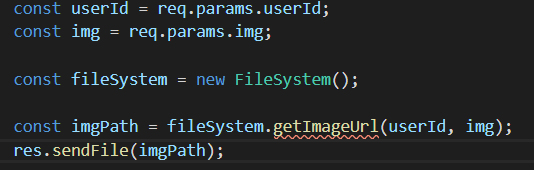


Servicio para mostrar una imagen por url

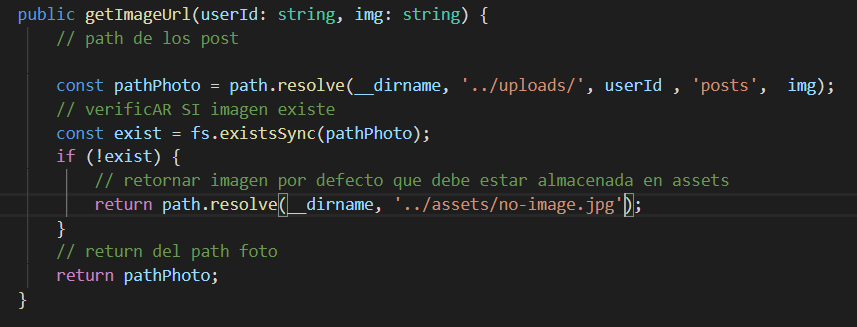
En el routes . post



Creamos las variables que serán recibidas en los parámetros del llamado.  
Luego declaramos el path de la imagen obteniéndolo desde el filesystem con una función que será creada para que busque y obtenga el path. Y finalmente enviando como respuesta la imagen para mostrarla directamente.



Ahora en el fileSystem



**Retornando la información del usuario verificando el token en el user routes**

