Mongo express angular Node

Instalando Mongo DB

1. Descargas la versión para tu OSX, en mi caso (mongodb-osx-x86\_64-3.4.10). Lo extraes a tu carpeta de preferencia (Ejemplo : /Sandbox/mongoDB
2. Creas los directorios donde vivirá la información  "    sudo mkdir -p /data/db     "
3. Ingresas a la caperta "bin" donde extraiste mongo   "/Sandbox/mongoDB/bin"
4. Ejecutas el demonio para iniciar MongoDB    "   sudo ./mongod   "
5. Iniciar nodemon para estar siempre a la escucha: npm start, según el package.json  
   \_\_\_\_\_

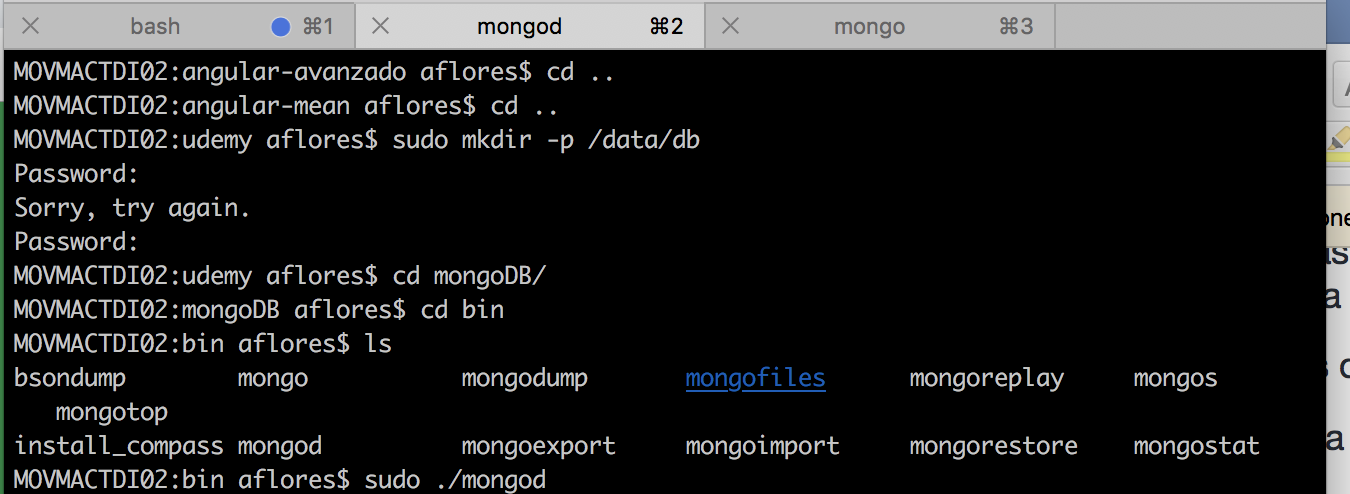
\*El demonio de Mongo siempre debe de estar corriendo para que puedas ejecutar comandos.

\*Entonces en una ventana de la terminal ejecutas el demonio "sudo ./mongod"  (la "d"  es de "demonio"), no lo cierres.

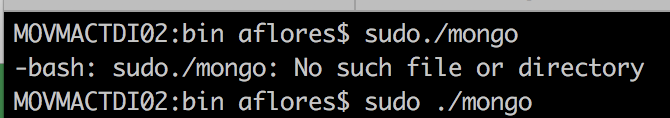
\*En otra ventana de Terminal entras al mismo directorio y ejecutas "sudo ./mongo" . Tienes que usar "   ./   " para ejecutar comandos

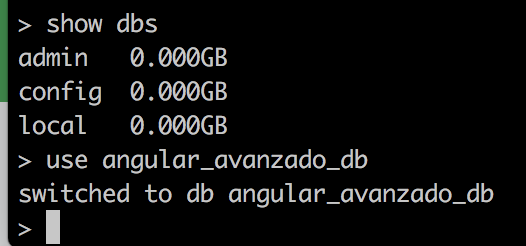
Pagina de descarga  
<https://www.mongodb.com/dr/fastdl.mongodb.org/osx/mongodb-osx-ssl-x86_64-4.0.4.tgz/download>

Creando la carpeta para las base de datos, esta ubicación puede variar según sistema operativo.



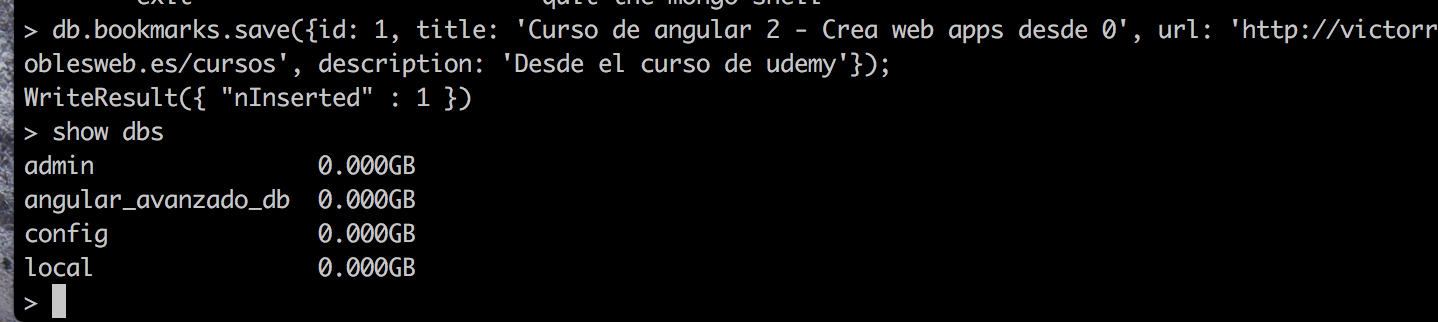
Ahora para ejecutar consultas en mongo

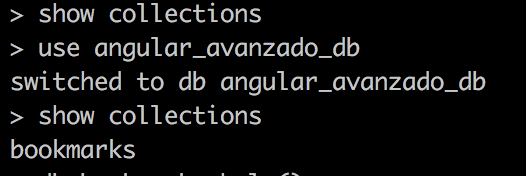


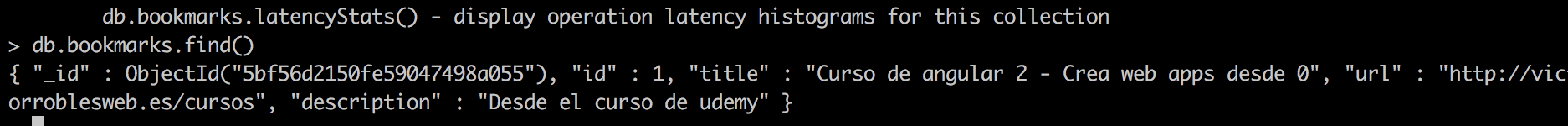
Creando una nueva base de datos:  


Luego dentro de la base de datos se crea una colección de objetos:

Donde declaramos el objeto y se pueden introducir los datos manualmente que tienen sus atributos.



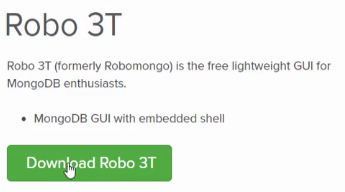
Para mostrar el contenido de la bd   




Administrar mongo db con GUI, RoboMongo y Robo3t

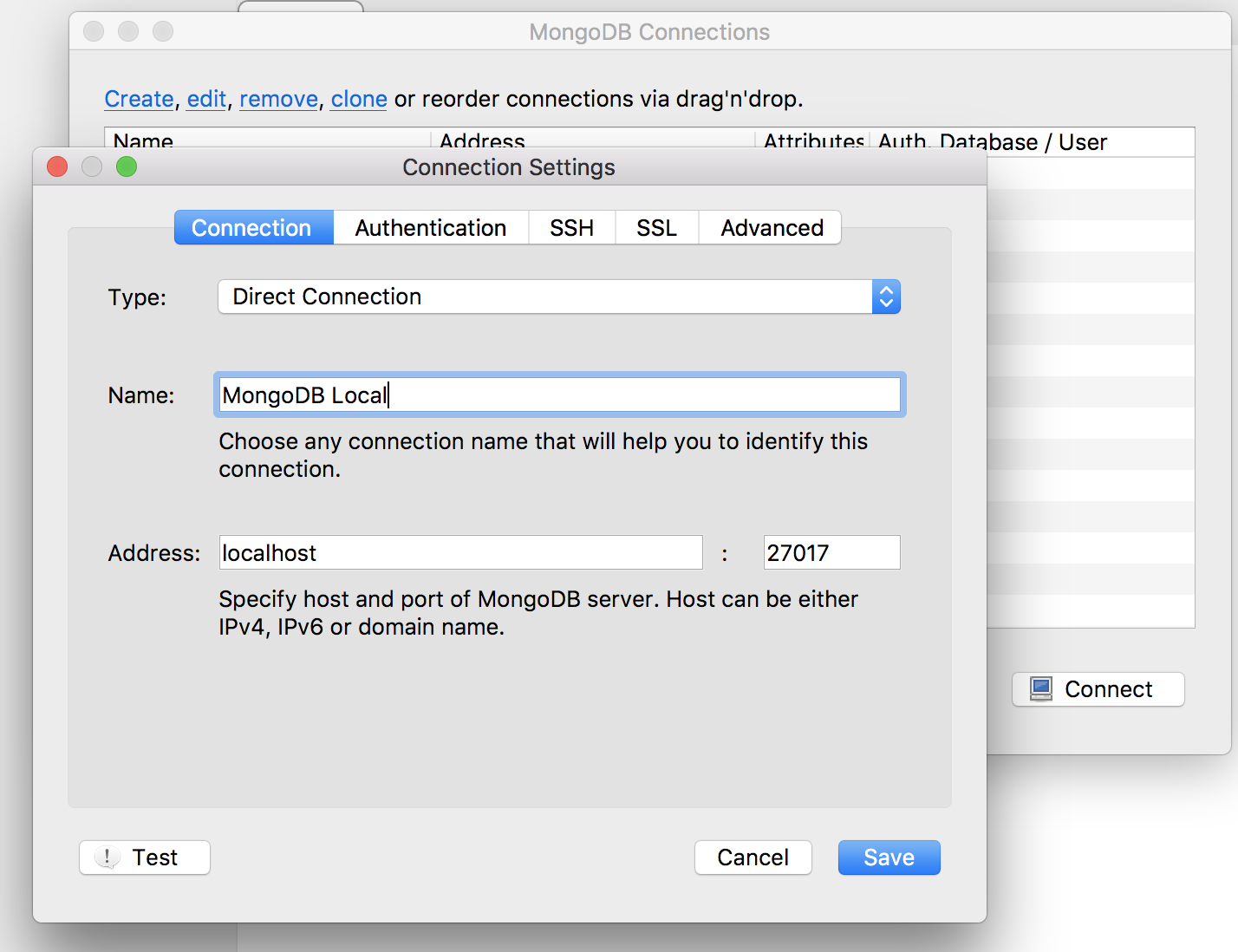
Con esto podremos ver los datos almacenados en mongo db en una forma grafica mas amigable.

https://robomongo.org/



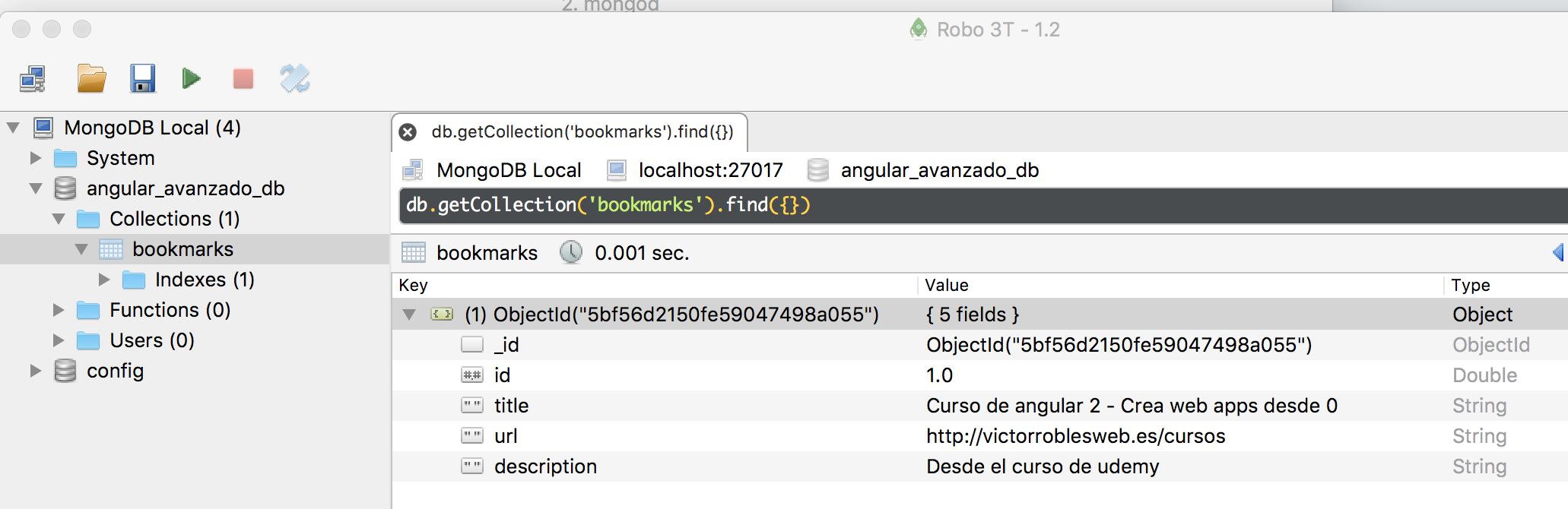
Luego de la instalación solo poner el nombre y ahora hay que establecer una coneccion

Click en créate



Luego eleginmos conectar y podemos visualizar la base de datos existente en ese puerto.

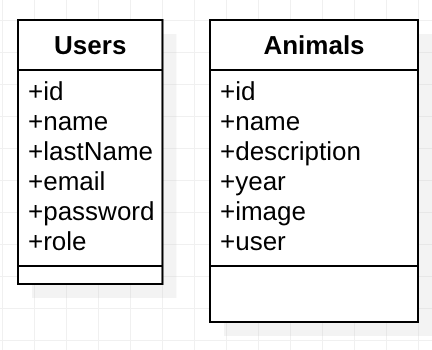
Hacuendo click en la colección (tabla), podemos ver su contenido y en el espacio gris se ve la consulta que se realizo para mostrar los datos.



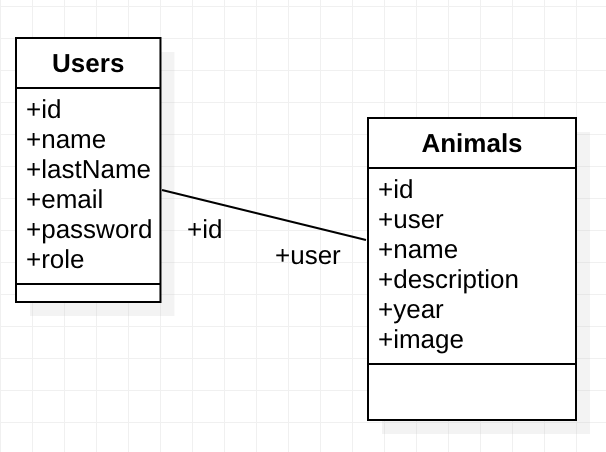
Diseño de esquema de Base de Datos:

Existe un programa que ayuda a realizar esquemas de entidad relación, uml etc.

starUML



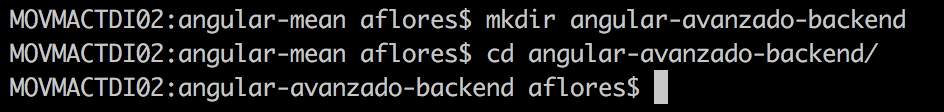
En donde el campo user referencia a otro documento de la colección user, y referencia al id del usuario que ha creado el animal.



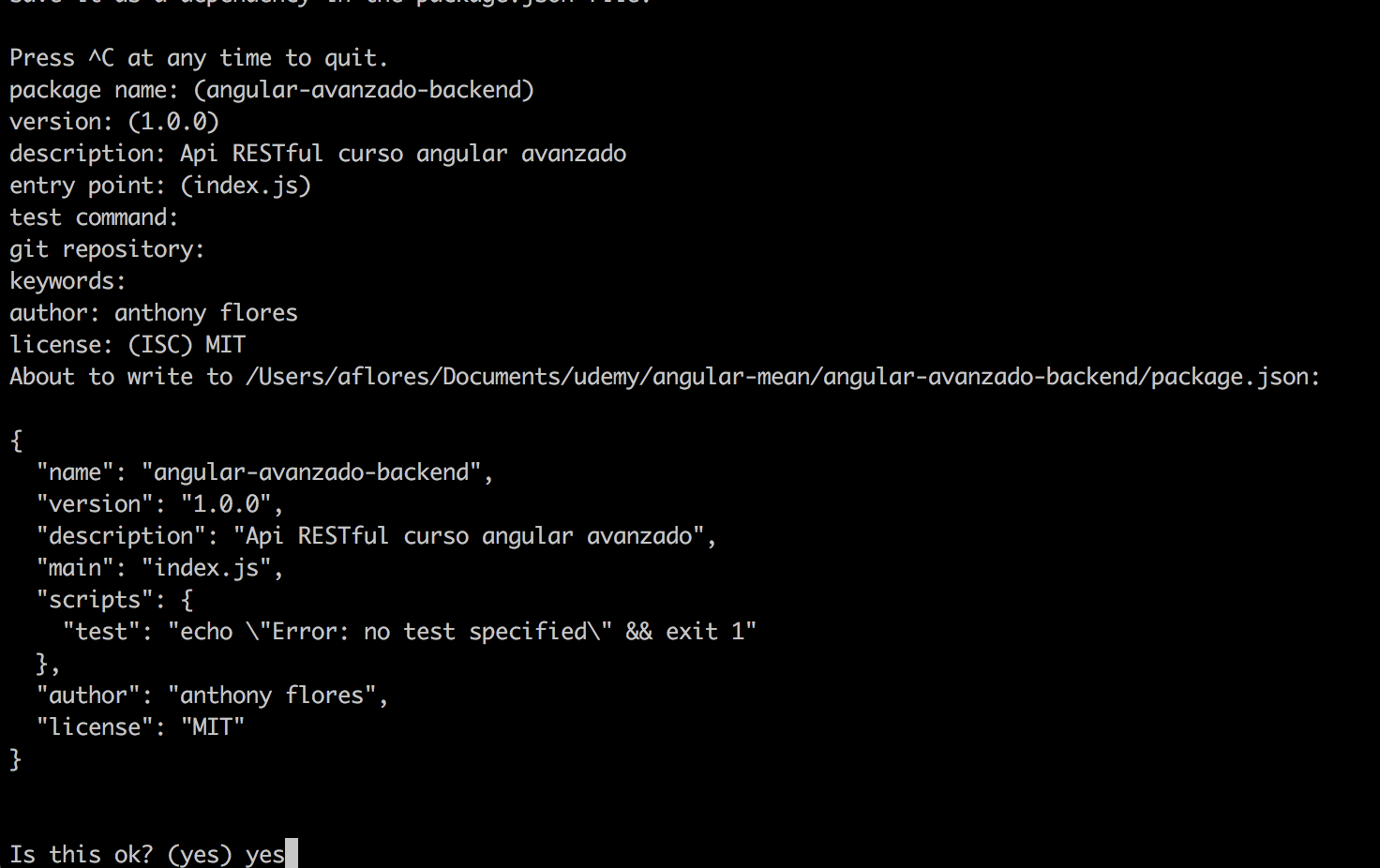
Creando el API

Instalando dependencias:

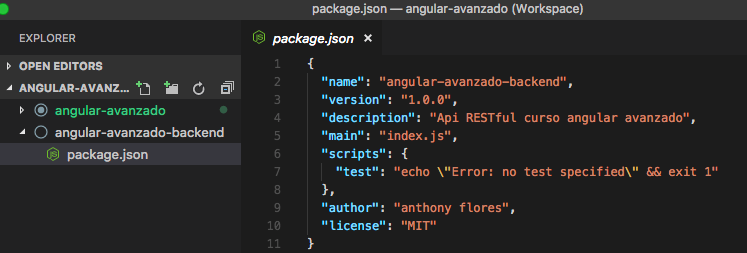
Regresando un nivel fuera del proyecto en línea de comando



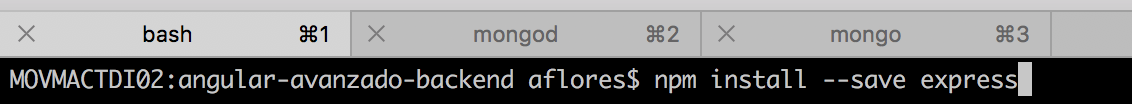
luego se corre el comando npm init lo que crea un proyecto njs



Se añade al proyecto del editor de código



Ahora es necesario instalar el express dentro del proyecto backend



Luego el paquete de cifrado de contraseña:



bodyparser, Para cuando se envie una petición con json, lo convierte a objeto d javascript.



paquete para subir ficheros, connect multiparty



Generacion de token para autenticación del api



Formateo de fechas con la librería momento



Instalando orm mongoose, extrae mongo db e implementa una interfaz que ayuda a trabajar con mongodb



Dependencia de desarrollo:

**Nodemon**, mientras se desarrolla se reinicia el servidor cada vez que se guarda un cambio

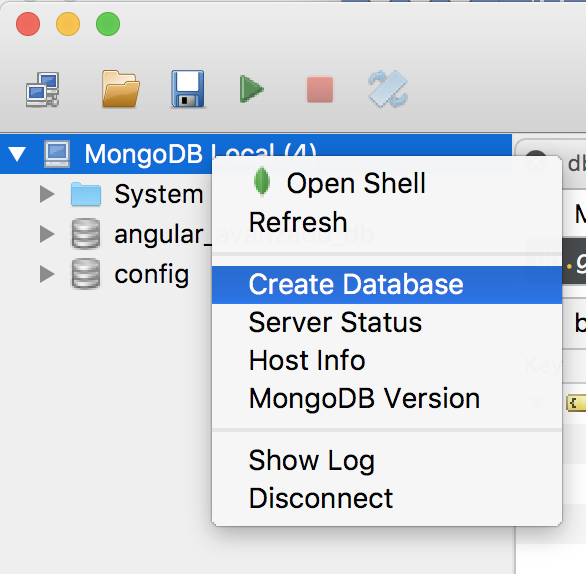
Al poner un –dev, indica que este paquete solo será utilizado en el ambiente local de desarrollo no en producción.



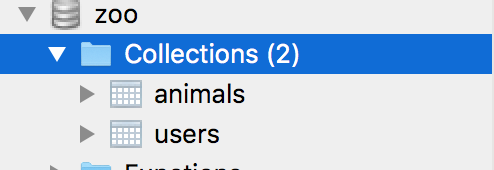


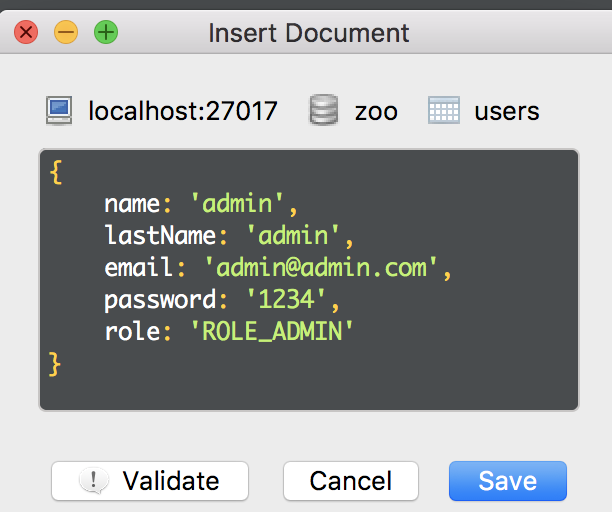
Creando la base de datos

Dentro de robomongo:



Luego se crean las colecciones (tablas) y dentro se hace click y la opción insert document para crear un objeto dentro.

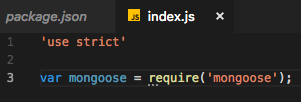




Conectandose a la bdd desde nodejs

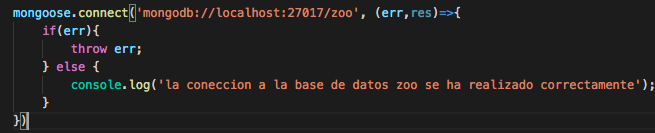
Ahora en nuestro proyecto de backend en la raíz se crea un archivo index.js el cual será el punto de inicio de la aplicación y tendrá la creación del servidor, coneccion a bd etc.

Luego de importar paquetes podemos utilizarlos, en este caso declaramos una varible mongoose y para asignarle valor se utiliza require, que busca el paquete instalado esta importación y la asigna.



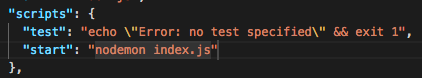
Ahora podemos utilizar esta variable para realizar la coneccion, teniendo entre los parámetros la url siendo el localhost desde el puerto que utiliza mongo y el segundo parámetro una función flecha





Al hacer esto aun no correrá ya que no se manda a ejecutar este archivo.

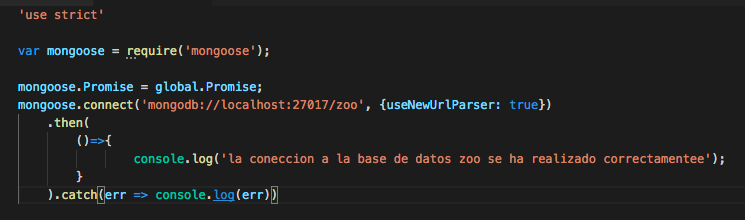
En el package json editamos los scripts y se agrega uno mas:

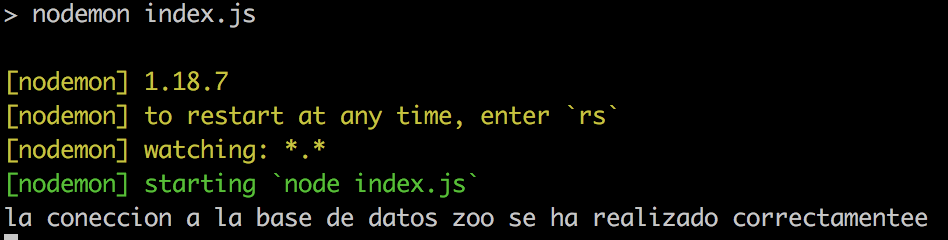


luego se corre npm start



En este momento lanza un error de deprecated, para solucionarlo hay que reestructurar el código utilizando un nuevo parser y estructura:



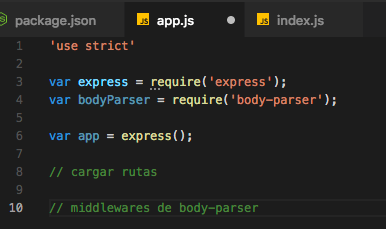


**Creando servidor web con Nodejs y Express**

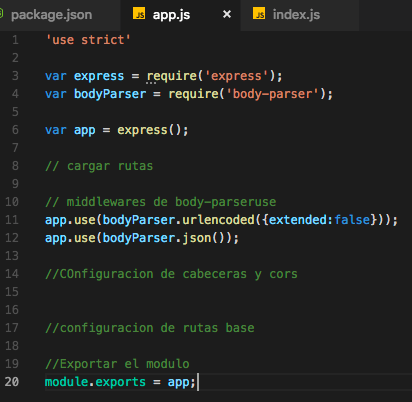
Es el motor de la aplicación se encargara de recibir peticiones http, crear controladores, tener disponibles rutas.

Se crea otro archivo app.js y se realizan las importaciones y ejecuciones

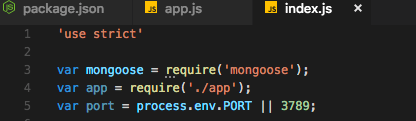
Los middlewares son ejecuciones que se realizan cuando se realizan peticiones http. Antes de devolver el json pedido ya se ha ejecutado el middleware.



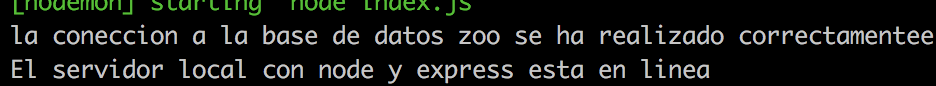
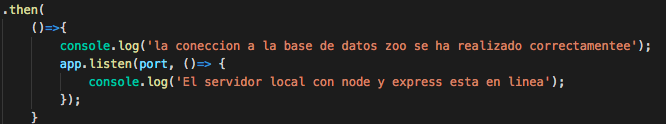




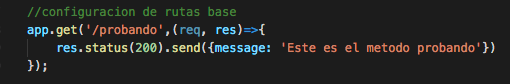
Ahora dentro del index.js hay que importarlo:

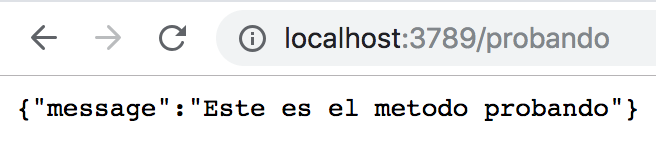


Luego dentro del método then, se ejecuta



Dentro del app js se pueden crear los métodos a los cuales se puede acceder desde el puerto:





El cliente RESTful y como probar el api

Utilizando postman podemos probar las peticiones al servicio y se puede instalar como una extensión de Chrome.

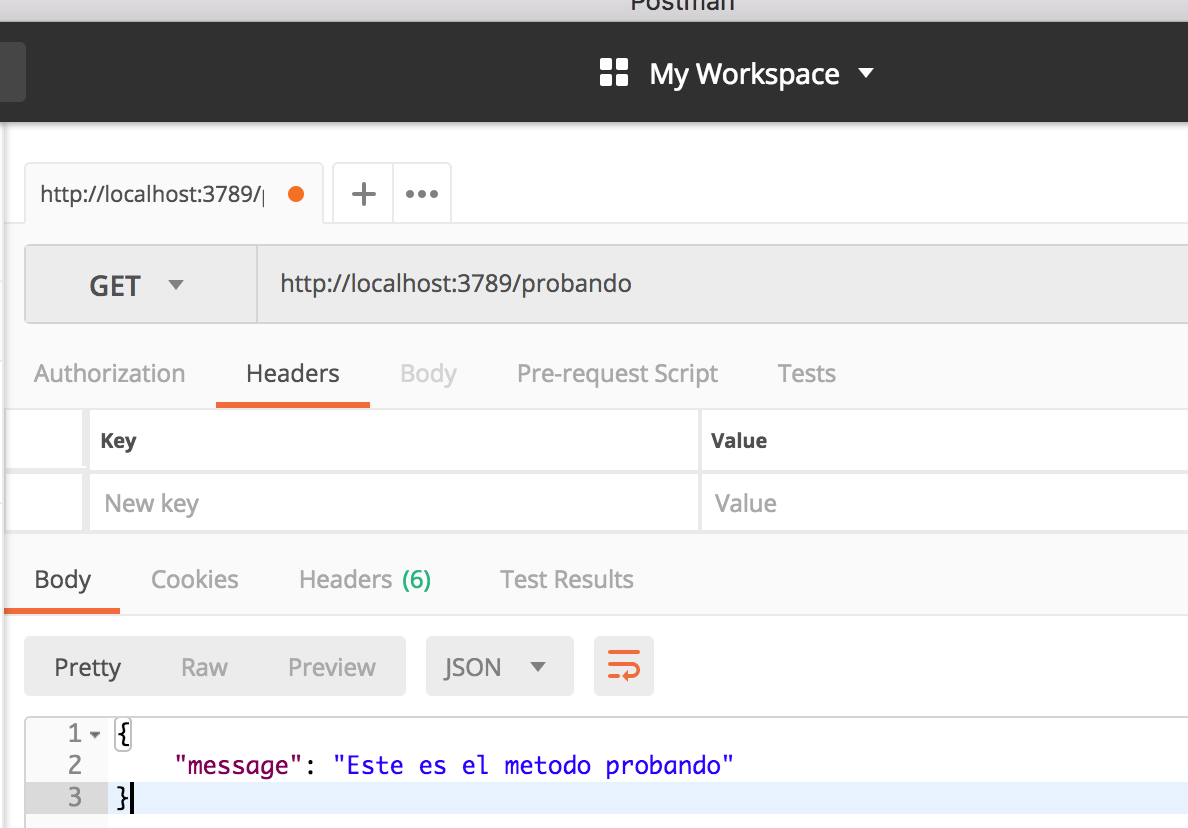


Diagrama de Back y BD

Mongoose

27017

3789

BD

Back

**Crear modelos y entidades**

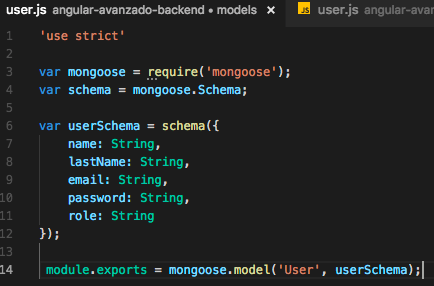
Representaciones de las entidades de la base de datos.

Dentro del poyecto de backend se crea la carpeta models y dentro el archivo del modelo

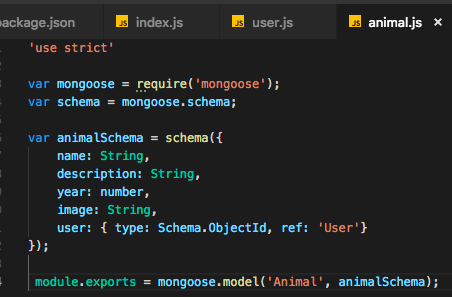
Se importa mongoose y se declara un esquema desde la propiedad de mongoose

Se crea la estructura del modelo y luego se importa con la expresión module export

Mongoose.model, que recibe 2 parametros, el nombre del modelo que luego será pluralizado por mongo, y la estructura creada.

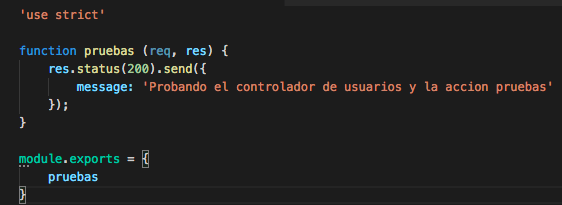


ahora se crea de igual forma el modelo de animal, en el cual su diferencia es la relación que tendrá con la otra colección. Para relacionarla en el campo que se relaciona lo declramos del tipo schema object y pasándole la referencia que se declaró en ese modelo.



**Creando un controlador y las rutas:**

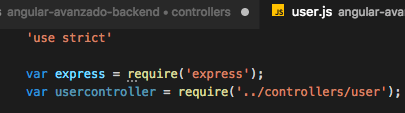
En la raíz del proyecto backend se crea la carpeta controllers, luego se crea una función que recibe dos parámetros, el req y el resp. Con el resp accedemos a la respuesta del estatus y con send podemos enviar un objeto respuesta:



ahora hay que crear un archivo de configuracion de rutas para cada controlador:

Se crea una carpeta routes en la raíz del proyecto back y dentro se crea el archivo user.routes.js

Donde luego se crea una instancia de express y del userController creado previamente con el require.



Ahora necesitamos utilizar el router de express para implementar el api



**EXPLICACION DE CONTROLLERS Y ROUTES**

|  |  |
| --- | --- |
| **Controllers** | **Routes** |
| **Tienen las funciones del back que reciben parámetros para comparar, etc y luego devolver con un res.status().send({}).** | **Provee las rutas para acceder a las funciones. Tienen parámetro de ruta, el “controlador”.”función” y se puede valer de un middleware. Ej: autenticación.** |

Ahora es necesario cargar el archivo de configuración de rutas en la configuración principal del proyecto.

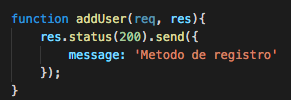
Dentro del app.js en la sección de cargar rutas se declara la nueva ruta importando desde el archivo perteneciente al modelo.

Luego la configuración de la ruta se declara con el app.use, con 2 parametros, el primero que se hara referencia en el url y el segundo nuestra ruta previamente cargada.



Controlador – ruta – llamado al servicio(app).

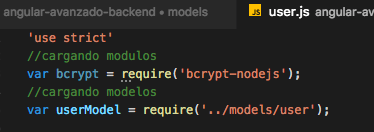
**Registro de usuarios:**

En el controller user se añade la función que por ahora tendrá la respueta  


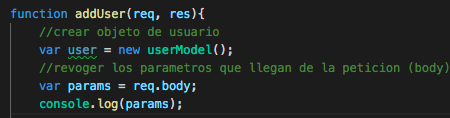
Luego en el routes se añade su llamado con un post por que se podrá añadir.



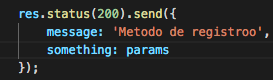
Ahora para añadirle la funcionalidad desde el controlador, hay que cargar el model de usuario. Se importa desde el inicio del archivo user en controller

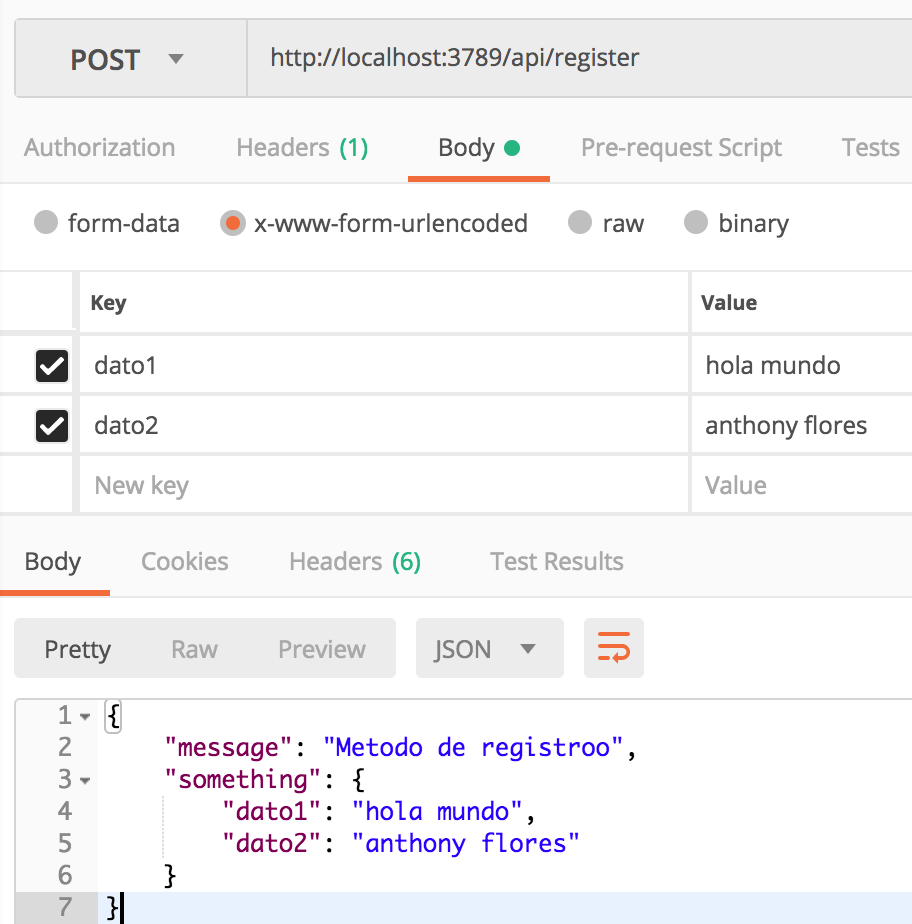


Luego dentro de la función se crea una instancia de nuevo usuario, luego se crea una variable que recibe los parámetros recibidos, para probar los valores desde postman en la sección de body, seleccionando x-www-form-urlencoded

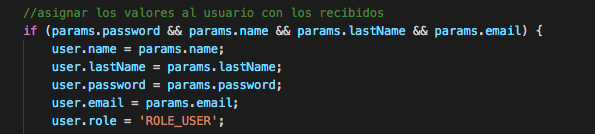


Se introducen los parámetros de prueba para luego poder ser visualizados en la respuesta:

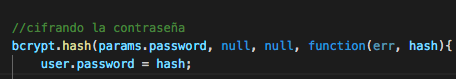




Ahora con los parámetros recibidos podemos asignarlo al objeto que tenemos de userModel



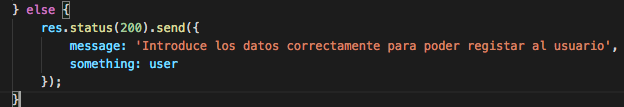
La contraseña recibida hay que cifrarla para eso utilizamos la variable bcrypt importada



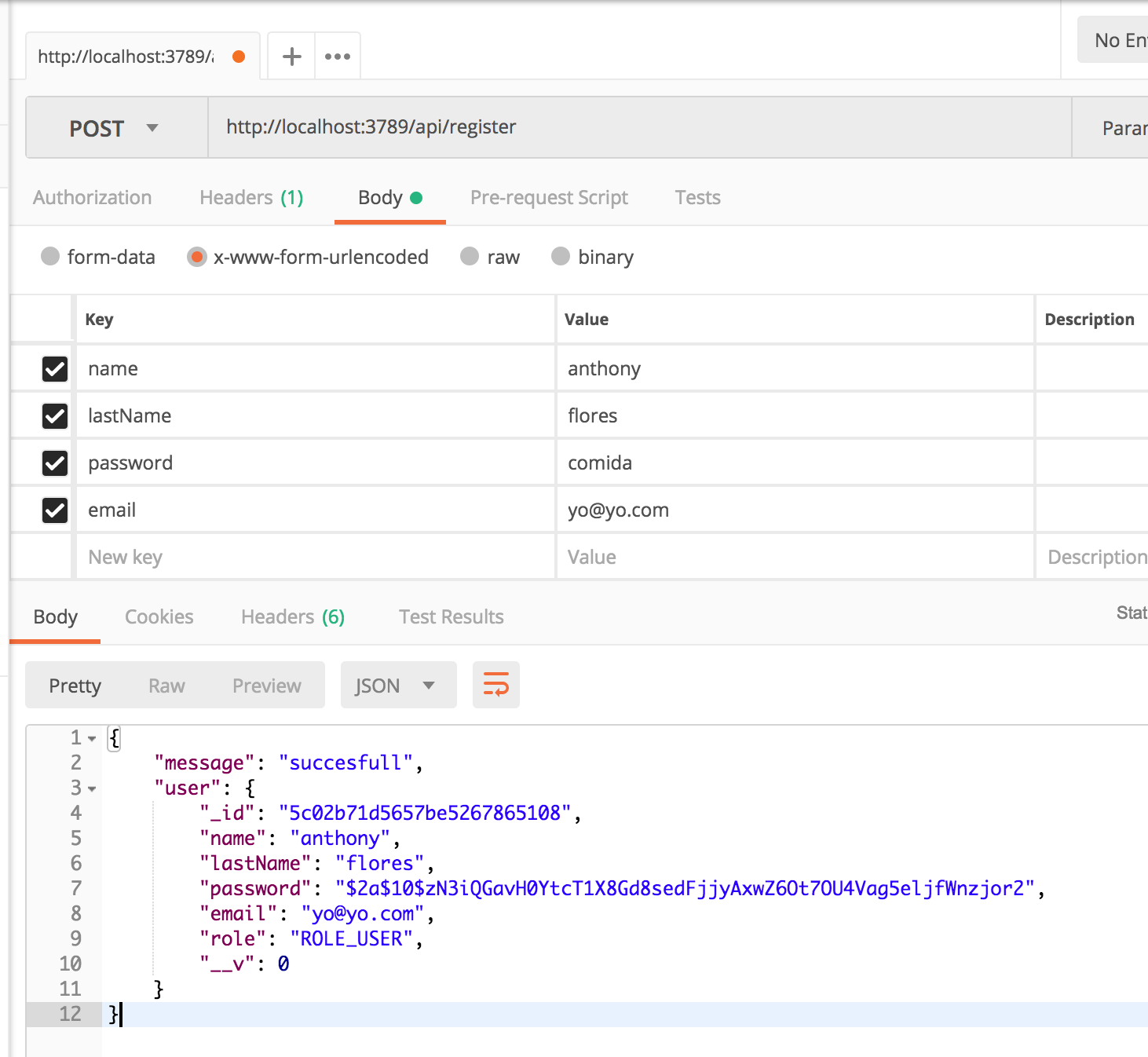
luego guardar el usuario en la base de datos manejando los tipos de errores



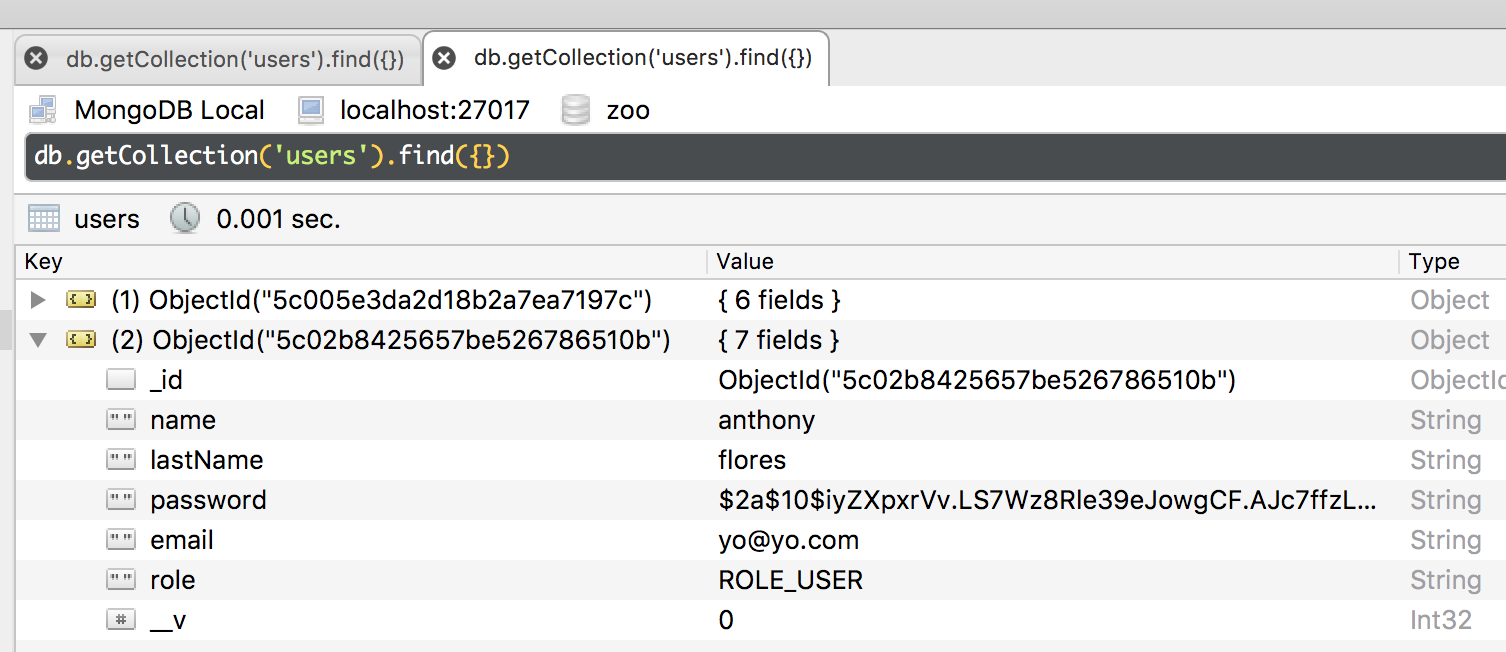
Ahora añadiendo un else cuando no se han introducido todos los datos



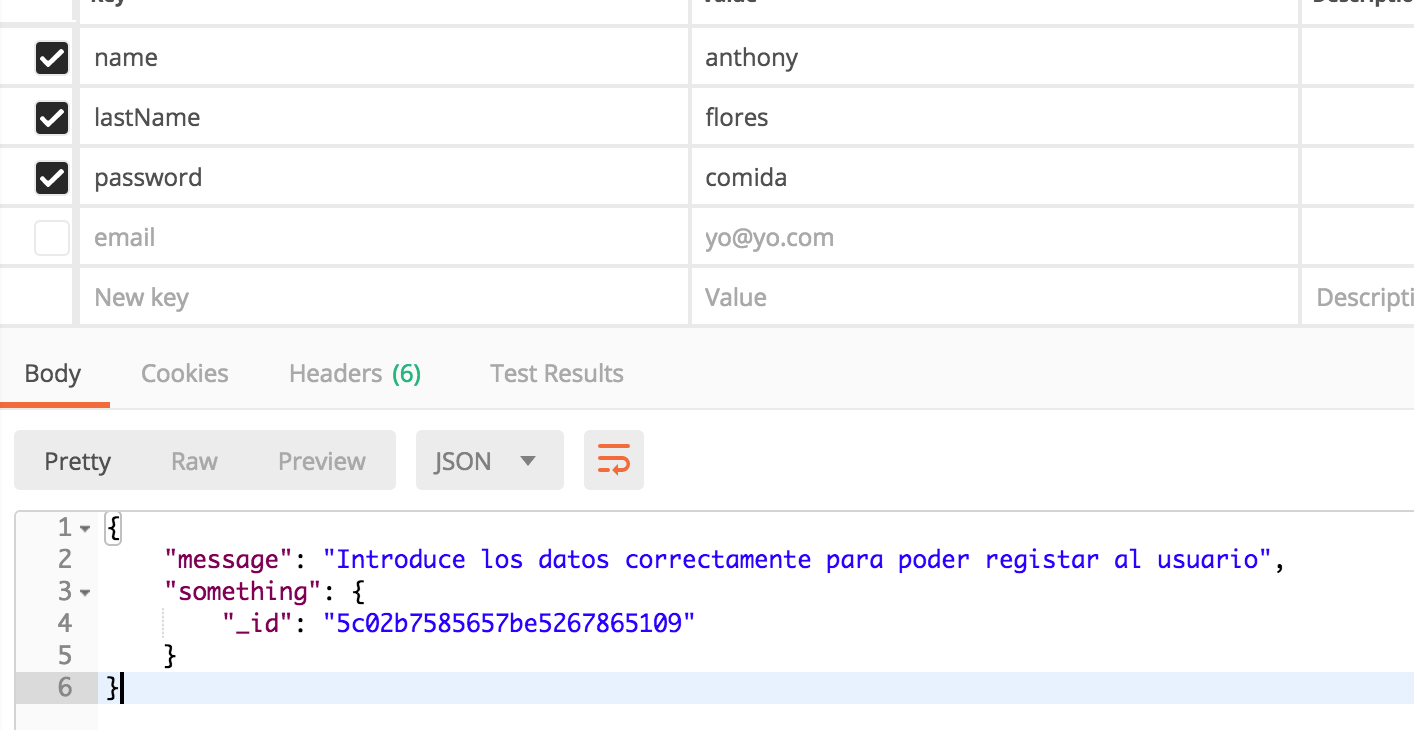
Probando en postman



Y si se visualiza en mongo vemos como se añade a la base de datos

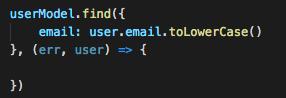


Dependiendo de los parámetros enviados entra en las validaciones y muestra el mensaje esperado



Controlar usuarios duplicados

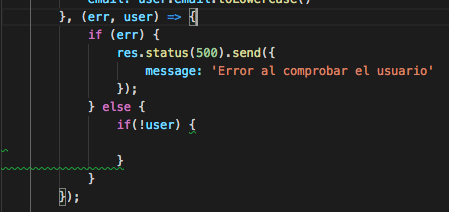
Utilizando el userModel podemos buscar dentro de la base de datos si existe el objeto que se introduce.



Correccion de método



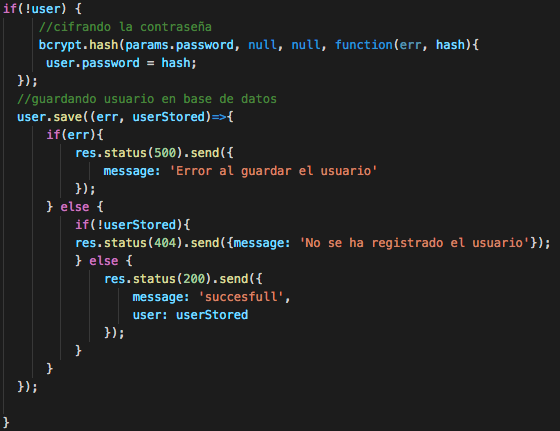
Verificando el si existe algún error enviar un mensaje, y cuando si debería guardar el usuario, osea cuando viene una respuesta vacia de usuario podemos agregarlo



Cambiar el nombre del user de fuera para que no haga conflicto con el user local que se utiliza.

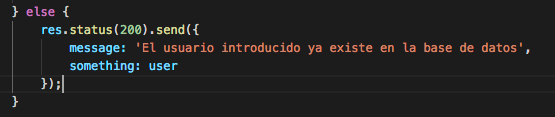


dentro de la validación de ¡user



Cambiar la comparación de user por

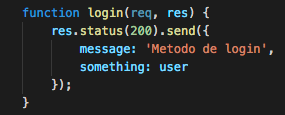


De otro modo al encontrar un usuario mandar un mensaje

Login

Buscar si el usuario existe, comprobar contraseña:

Se crea un método en el controller-user, se exporta:



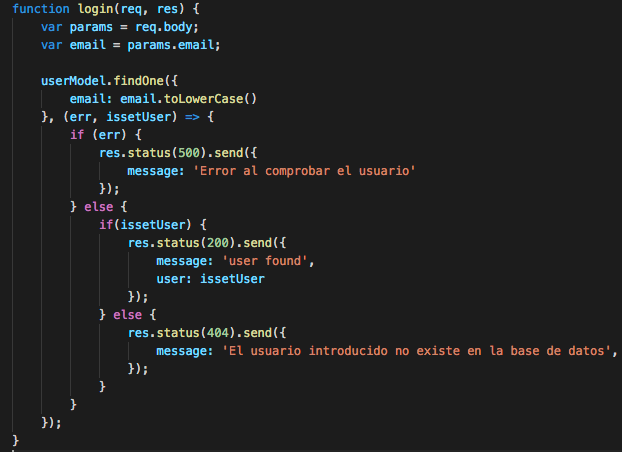
y se integra en el archivo de ruta



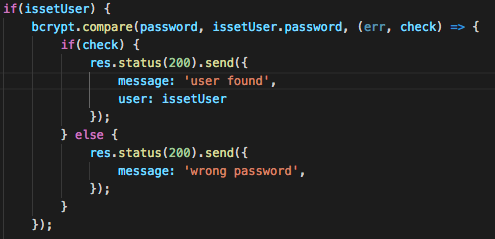
ahora en el método del controller verificar que el usuario existe utilizando los parámetros que llegan:

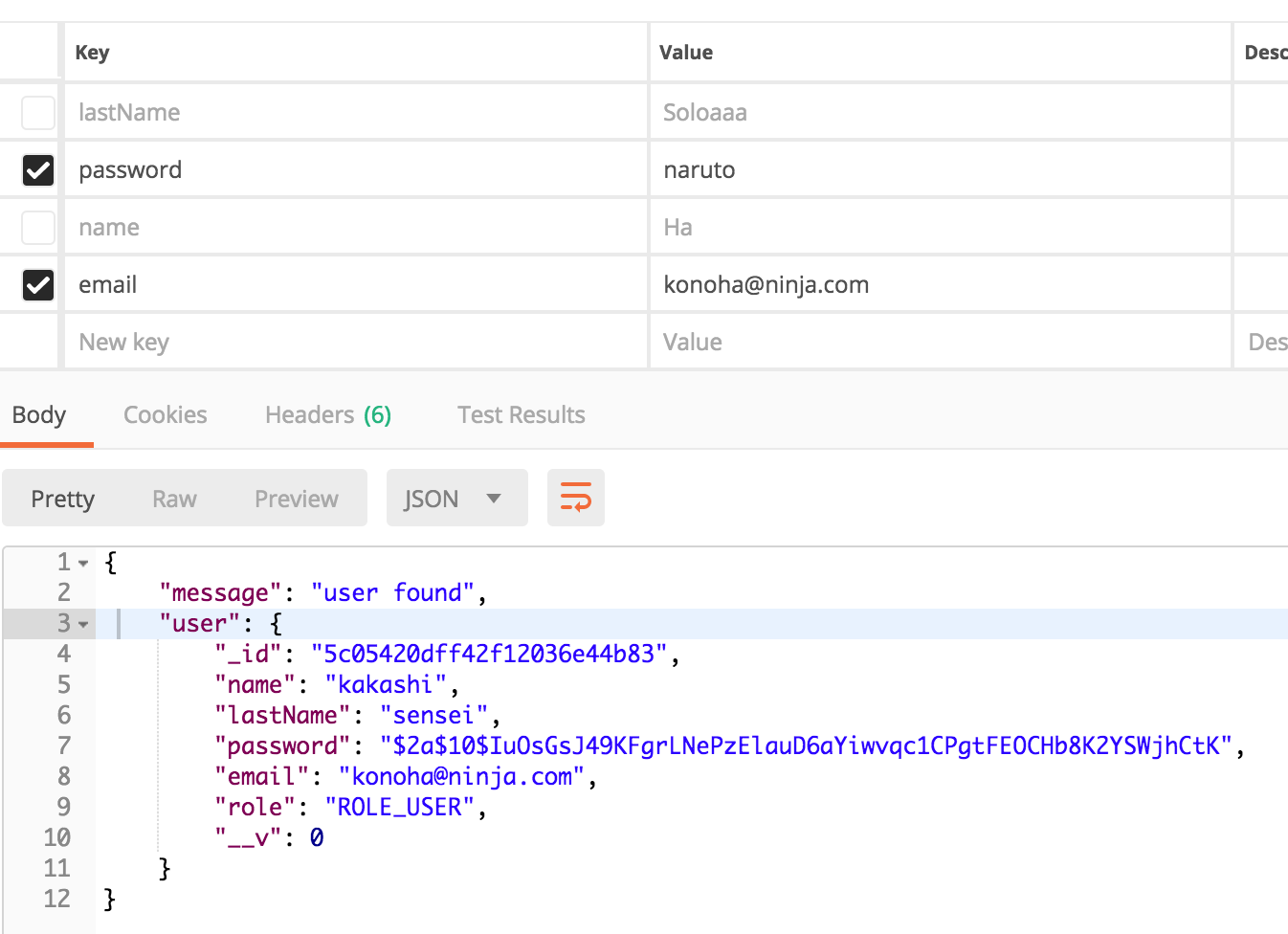


y ahora hacer un find del email



Comparando la contraseña introducida utilizando bcrypt.compare que recibe 2 parametros en el caso de que este check, se encontró el usuario, de otro modo, la contraseña fue mal introducida:





Generando un token con jwt

Este token será utilizado al hacer una petición que se deba utilizar alguna autenticación del usuario.

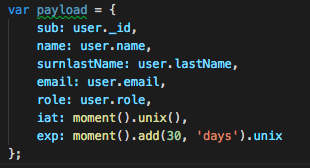
Es necesario crear un servicio, creando una carpeta services en la raíz del proyecto y dentro un archivo jwt.js

Hay que importar el modulo de jwt y moment.

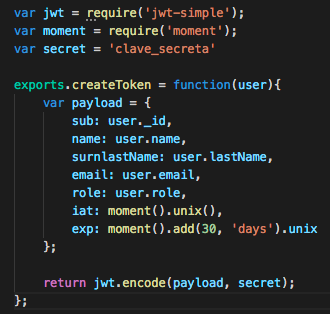
Luego exportamos una función createToken que recibirá como parámetro el Usuario que se esta logueando. Luego generar un payload que es el objeto con que jwt genera el cifrado.



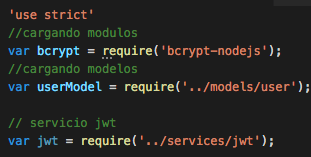
Este objeto recibe la información del usuario mas la fecha de creación y expiración asignada por momento unix y add respectivamente.



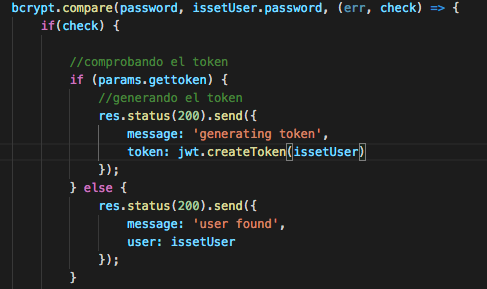
Para generar el token se utiliza la función enconde de jwt pasándole como parámetro el objeto payload y una clave secreta que es una cadena con la “clave”



Luego para poder utilizar el servicio en el archivo user ubicado en controllers importamos la función creada asignándola a una variable.



Ahora en la función de login al momento de hacer check si existe el usuario, se hace otra validación, si existe en los parámetros recibidos la propiedad getToken estaría utilizando la nueva función del servicio.



Middleware de autenticación:

Se puede utilizar el token generado para “proteger” algún método, osea permitir su uso solamente si el usuario esta logeado, osea si tiene token.

Podemos proteger la función en el caso de prueba.

Creando carpeta middlewares

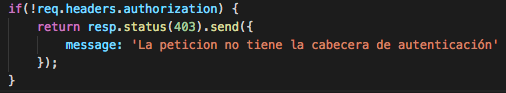


se importa jwt, momento, y se crea la variable secret.

En la función a exportar recibimos req, res y next la cual nos madará a la siguiente función una vez se ejecute el middleware correctamente.



dentro de la función hay que verificar que se este recibiendo la cabecera con la autenticación, de otro modo salir mandando el mensaje.



Si no, se le asigna a la variable token lo que trae el req autorization y se reemplaza si existe en ella comillas simples o dobles por nada.



Despues de obtener el token se hace un try catch de excepciones validando si al hacer decode del token, utilizando la clave secreta, este contiene el sub (id) y la fecha de expiración aun no se ha cumplido, comparando el valor que tiene el token con la fecha actual.



Una vez teniendo el payload con el objeto completo del usuario, se setea una nueva propiedad dentro de req cual será user, y esta se podrá utilizar en todos los métodos del controlador, asignándole el valor actual del payload valido.



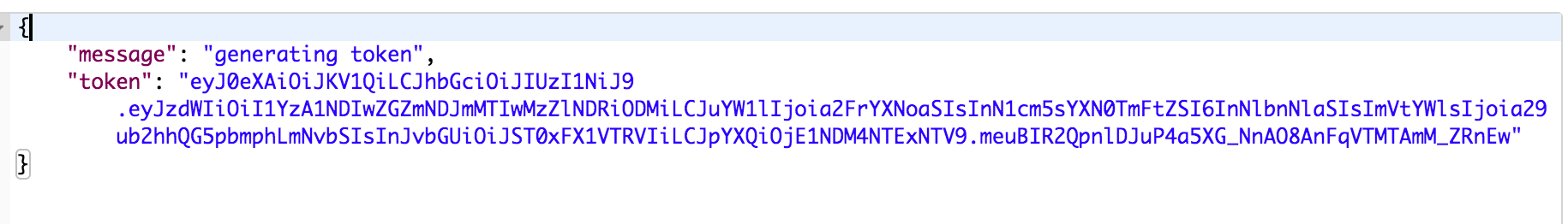
Para utilizar el middleware en las rutas que se requieran hay que ir a user de rutas y cargar el middleware. Luego anteponerlo al servicio que queremos proteger con el middleware.



ahora para poder utilizar este servicio debemos mandar un req con el token que se ha generado anteriormente y sea valido para recibir una respuesta.

En este momento se genera al momento del login recibimos el token.

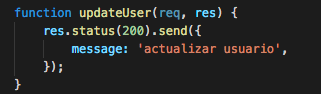
Esta es la llave para poder acceder a las funciones que protejamos con el middleware.





Actualizar usuario:

Se crea la función en user de controller



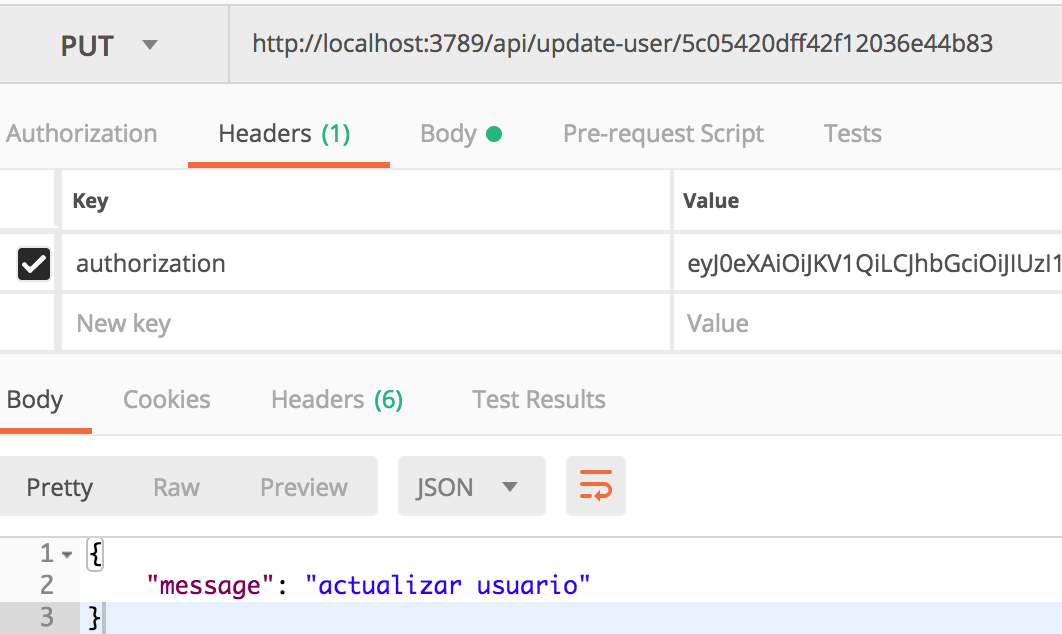
se añade a las exportaciones



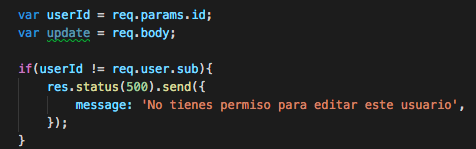
y se añade a las rutas por put ya que será para editar



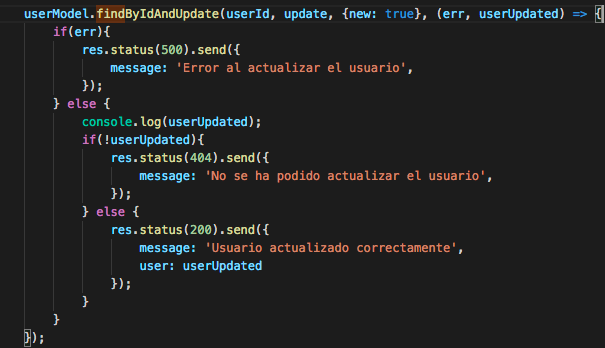
Probando con el postman, pasándole el id de un usuario existente por el url y el token

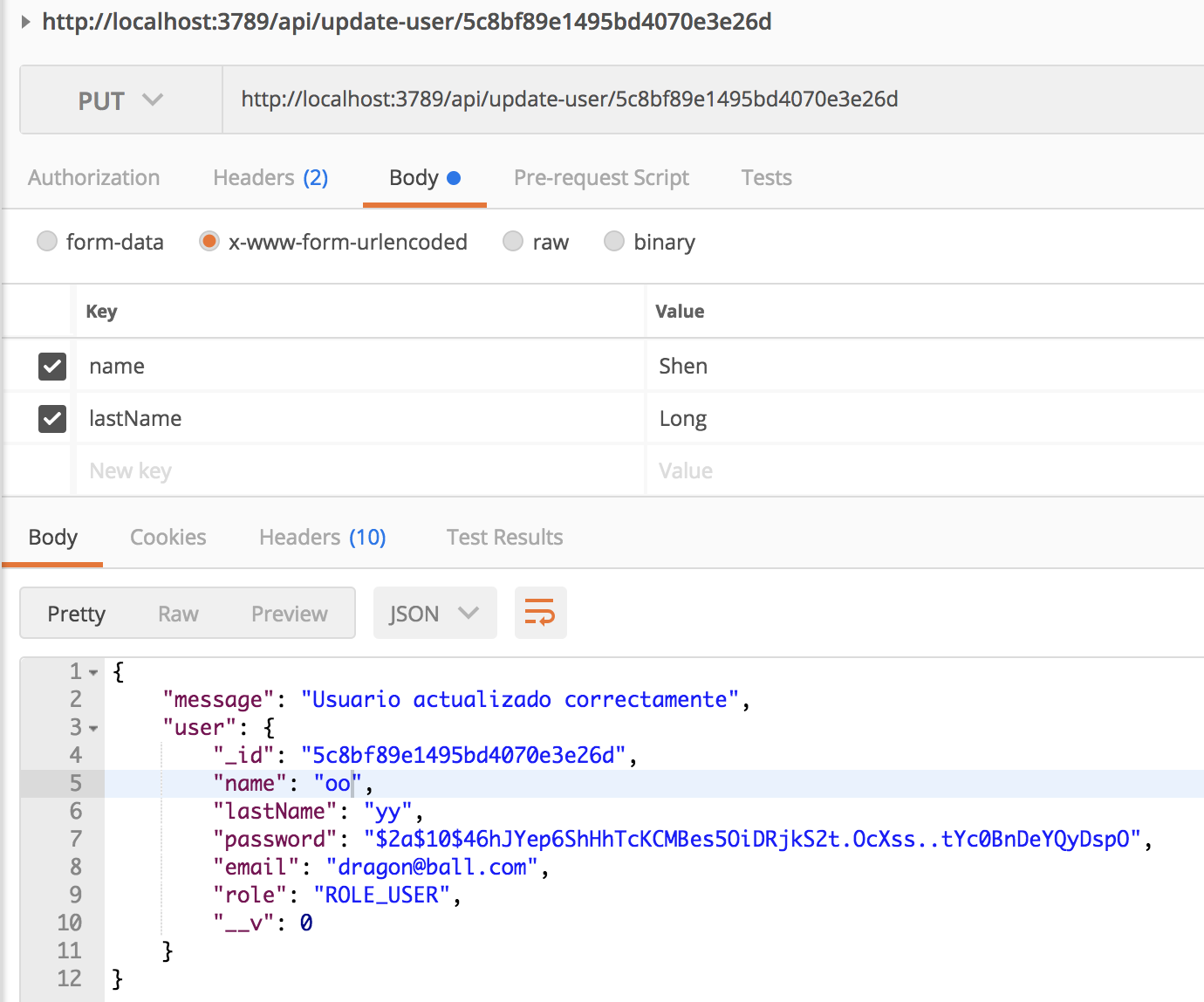


En la función, al comparar el id recibido por el url y el id de la respuesta del body, si no son iguales mandar un mensaje de error.



Para modificar el usuario hay que utilizar la función de buscar y actualizar





**Avatar de usuario:**

Al inicio instalamos el connect multiparty que nos permitirá utilizar un middleware para subir archivos.

Desde el user.routes se importa el multiparty y se crea una variable que tendrá una ruta manejada por multiparty



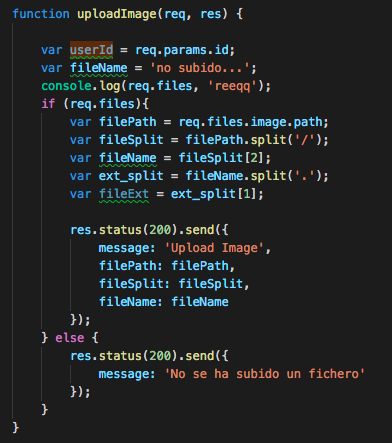
y luego se crean los directerios empezando desde la raíz.



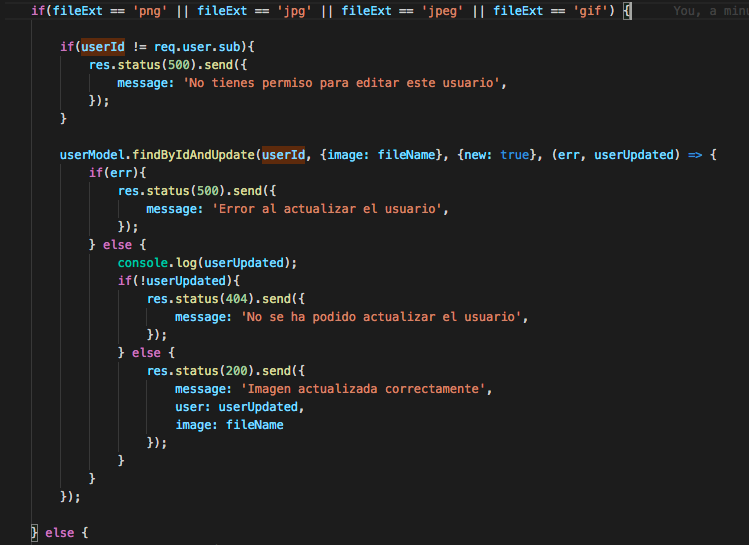
ahora creando el método en user.controller

se obtiene el id que proviene del url, luego se comprueba si existe req.files

y dentro se asignan las variables del path, nombre de archivo y extensión para validar luego



Ahora se añaden las validaciones según la extensión del archivo y luego con los permisos que se tienen según el id



El modelo actualmente no tiene la imagen del usuario por eso responde

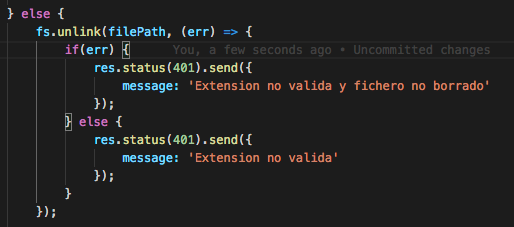


Solo añadir en el modelo user un campo image tipo String

Para filtrar guardar o no los archivos:

En el mismo user.controller importar el modulo de fileSystem





Devolver imágenes de usuario

Una función que devolverá un fichero en la url, que al recorrer y mostrar cualquier usuario habrá una url para cargar la imagen.

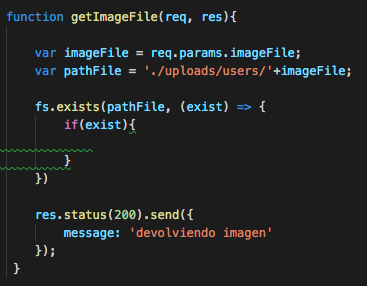
Exportarlo en el controller y en el routes declararlo

Donde recibirá como parámetro en el url la imagen con su extension



Se declara una variable que recibe el image file recibido en los parámetros, luego se declara el path sabiendo la ubicación del archivo en el proyecto y agregándole el nombre del archivo

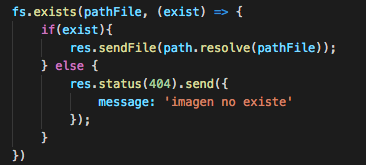
Se utiliza la función del fileSystem verificando si existe este archivo.



Para poder acceder directamente a las rutas de archivos hay que importar el modulo path



y se puede utilizar el path ahora para verificar la existencia

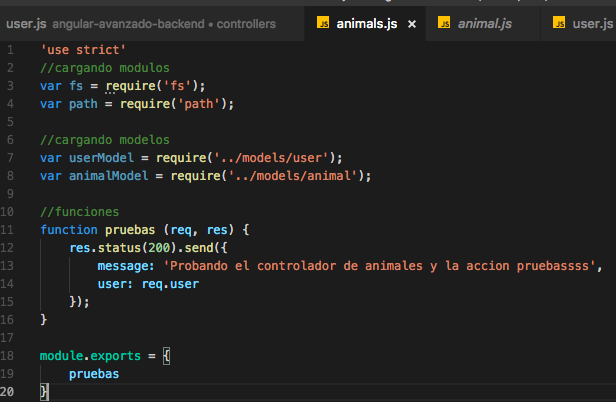


Listar cuidadores

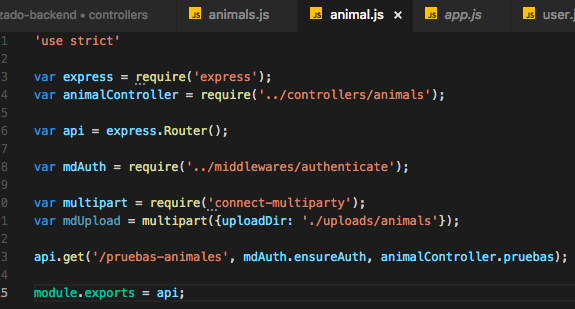
Utilizando el userModel y la función find, nuestro query es el objeto dentro, asi buscamos los que tengan como rol admin y luego se ejecuta y verificamos si devuelve un error y si al retornar users esta vacio.



Controlador de animales

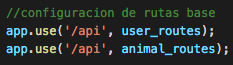


luego se configura la ruta de animals para que utilice el controlador de animals y declarando las rutas de sus funciones



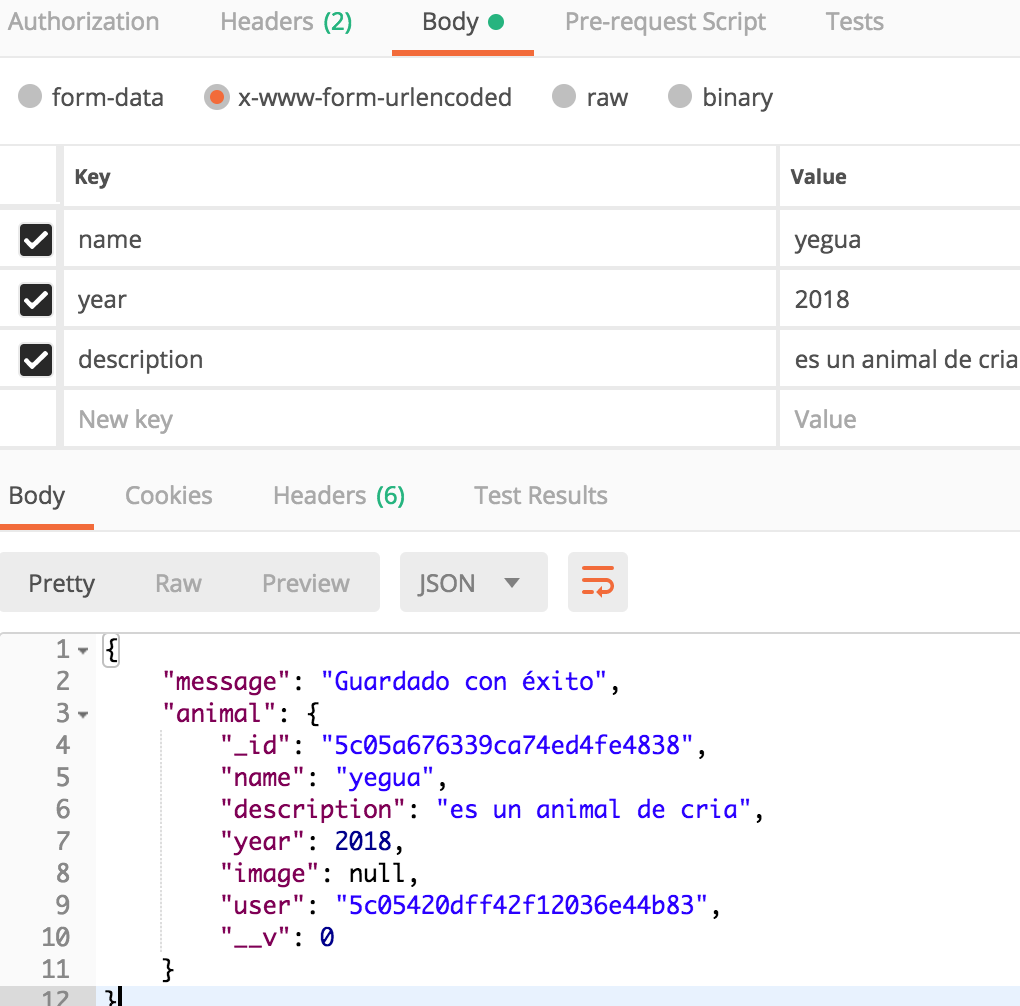
Ahora dentro del app.js



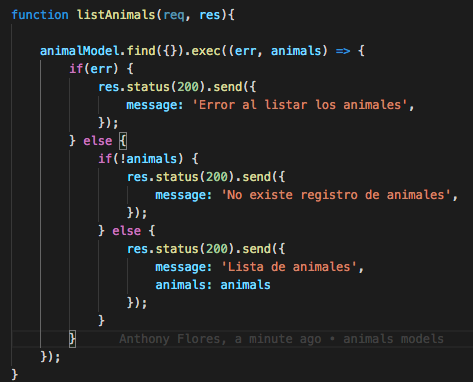


Guardando animales





Devolver listado de animales:



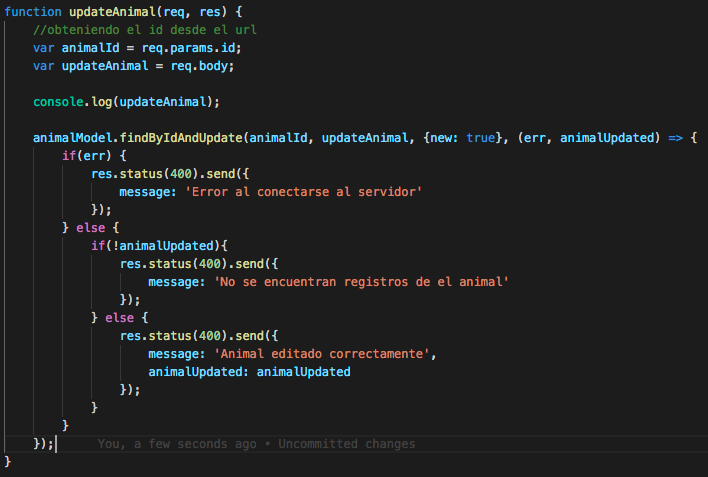
Al utilizar el método populate path user, obtenemos el objeto del usuario que ha introducido este animal, por lo que será su cuidador



Devolver el objeto json de un animal concreto



Actualizar un animal

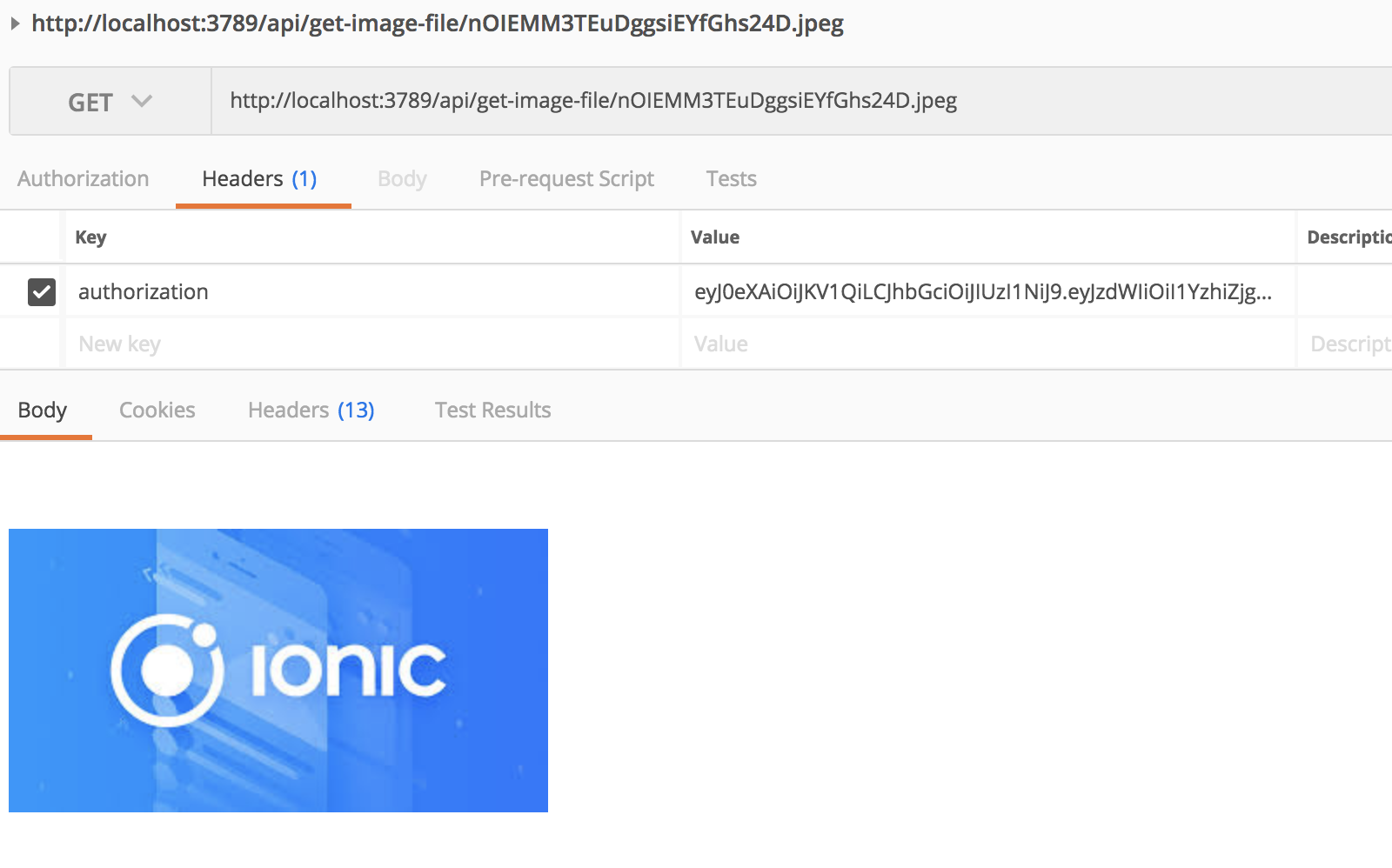


Subir imagen y mostrar imagen.



Obteniendo la imagen



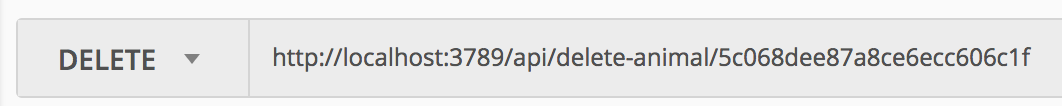


Borrar animales de la bdd





y en postman

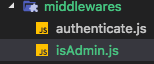


**Middleware para el rol de administrador**

Proteger las rutas en función si es publica o privada

Privadas tipo post crear animal, solo puede acceder un usuario admin

Se crea el middleware esAdmin



recibiendo en el req el usuario y accediendo a su rol se hace la comparación. Si es admin este sigue con el siguiente servicio.



Ahora en el animal.routes se importa el middleware



Ahora anteponiendo el middleware antes de la petición al servicio



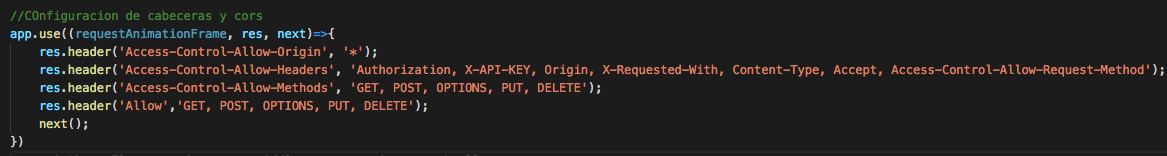
Si intentamos loguearnos utilizando el token de un usuario que no tiene permisos de admin,



Configurar cabecera HTTP y acceso CORS

Cors es el acceso cruzado entre dominios, hay que configurar las cabeceras para permitir peticiones Ajax de un dominio a otro.

Es necesario crear un middleware directamente en el app.js



Con esto ya esta listo el backend api.