



ÁGORA

Red social



24 DE JUNIO DE 2019

RUBÉN CASTRO RUIZ

IES Cánovas del Castillo

Grado Superior de Administración de Sistemas Informáticos en Red

Índice

1. Introducción del proyecto.
2. Presentación de Ágora.
3. Hardware usado.
4. Tecnologías usadas.
 - 4.1. Visual Studio Code.
 - 4.2. Web stack MEAN.
 - 4.2.1. Introducción.
 - 4.2.1.1. JavaScript.
 - 4.2.1.2. Diferencias entre LAMP y MEAN.
 - 4.2.1.3. MVC – estructura modelo, vista y controlador-
 - 4.2.2. Mongo DB.
 - 4.2.3. Express.
 - 4.2.3.1. API
 - 4.2.4. Angular.
 - 4.2.4.1. Typescript
 - 4.2.4.2. SPA.
 - 4.2.4.3. Angular CLI.
 - 4.2.5. Node JS.
 - 4.2.5.1. NPM
 - 4.3. Bootstrap.
 - 4.3.1. Bootstrap 4.
 - 4.3.1.1. JQuery
 - 4.3.2. Ng Bootstrap.
 - 4.3.3. FA Font Awesome
 - 4.4. GIT.
 - 4.4.1. GITHUB.
 - 4.5. Postman.
 - 4.6. JWT - JSON WEB TOKEN
5. Explicación Back-End.
 - 5.1. package.json
 - 5.2. Index
 - 5.3. Base de datos
 - 5.4. Rutas
 - 5.5. Controlador
 - 5.6. Modelos
6. Explicación Front-End.
 - 6.1. Angular CLI
 - 6.2. Componentes
 - 6.3. Servicios
 - 6.4. Modelos
 - 6.5. Guardián
 - 6.6. Rutas
7. Propuestas a mejorar o implementar.
8. Bibliografía.

1. Introducción del proyecto.

Me gusta mucha la programación, en mis estudios he aprendido cronológicamente lenguajes: C, SQL, *HTML*, *CSS*, *XML*, PHP y MySQL.

Para este proyecto mi objetivo es aprender el lenguaje JavaScript y sus frameworks. Me llamaba mucho la atención este lenguaje de programación porque está abarcando una gran comunidad y se está expandiendo hacia muchos fines diferentes con sus frameworks, lo que nos da la posibilidad de crear diversas aplicaciones totalmente diferentes con un mismo lenguaje de programación.

Además, es un lenguaje de alto nivel por lo que no será tan complejo aprenderlo en este pequeño tiempo en comparación con otros de bajo nivel.

El reto para aprenderlo, ha sido hacer una aplicación web y me gustaba la idea de crear una red social. Ya que la informática se está expandiendo rápidamente debido a la capacidad de comunicación que nos proporciona.

2. Presentación de Ágora.

Mi idea es crear una red social, su nombre, Ágora, nace de su propio significado en griego. El Ágora en la antigua Grecia era la plaza central donde se reunía el pueblo, un lugar abierto donde se comercializaba, hablaba de política, cultura y en general, donde abrían sus vidas sociales.

Este pasado lo he querido llevar a la actualidad, donde el poder del internet nos permite comunicarnos en un espacio ilimitado.

La pequeña plaza llena de palabras en forma de ondas vibratorias senoidales medidas en Hercios, ahora se ha actualizado a “El Mundo” al completo, lleno de flujotes binarios o “*palabras octales*” que se encienden y apagan en forma de luz en cables de fibra óptica medidos en Bytes.



3. Hardware usado.

El único hardware usado ha sido mi portátil Asus cuyos componentes son:

Un procesador Intel i3 6006U a 2,0 Hz, tarjeta gráfica Nvidia Gforce 920 MX, 8GB de memoria de RAM y 1TB de almacenamiento interno magnético.

4. Tecnologías usadas.

El sistema operativo que he usado, ha sido Microsoft Windows 10 de 64 bits, versión 1803 (compilación 17134.829).

4.1. Visual Studio Code.

He decidido usar este editor de código, desarrollado por Microsoft y escrito en Typescript, JavaScript y CSS siendo una aplicación web progresiva.



La razón de escogerlo en mi caso, es porque esta muy bien enlazado con Typescript (también desarrollado por Microsoft), lo que mejora la rapidez de escritura y nos proporciona funcionalidades muy útiles como por ejemplo con un simple control + click en una función, te lleva a la declaración de ésta, en su propio fichero si existe en el directorio de trabajo, una sección de terminales; o atajos de teclado como comentar todo lo seleccionado, selección múltiple o la posibilidad de instalar plugin; entre muchísimas más funcionalidades.

4.2. Web stack MEAN

4.2.1. Introducción

MEAN son las iniciales de las tecnologías que se usan para hacer una aplicación web con tan solo usar el lenguaje JavaScript. Estas tecnologías son Mongo DB como base de datos NoSQL, Express como framework de Node JS, Angular como framework de Front-End y Node JS como lenguaje de Back-End.

4.2.1.1. JavaScript.

Es un lenguaje de programación que surgió en 1995, para darle dinamismo a las páginas web que hasta entonces solo se utilizaban como transferencia de hipertexto. Actualmente es un dialecto del lenguaje estándar ECMAScript.

JavaScript es un lenguaje multiparadigma, orientado a objetos, interpretado, se ejecuta en el navegador, carece de un buen tipado y es dinámico.



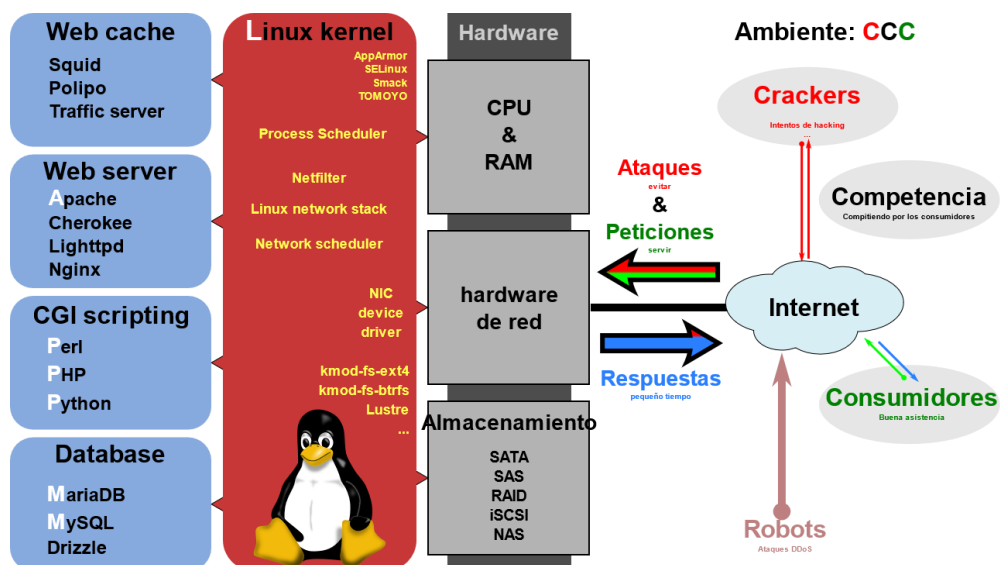


La evolución de JavaScript ha llegado a Node JS, que crea un intérprete fuera del navegador, basado en el motor Chrome V8 como el que usan los navegadores Chromium, Google Chrome y Microsoft Edge. De este modo, se convierte en un lenguaje de propósito general de hecho el editor de texto que he usado existe gracias a él, estando escrito en JavaScript.

4.2.1.2. Diferencias entre LAMP y MEAN

El tradicional web stack **LAMP** son las iniciales de las tecnologías usadas para crear webs, que son las siguientes:

Se basa en un servidor **Linux** también podría ser Windows o Mac, siendo WAMP o MAMP; usando el protocolo HTTP con **Apache** que maneja las peticiones web, un sistema gestor de base de datos como **MySQL**, que puede derivar por ejemplo a María DB donde almacenamos los datos. Y por último el lenguaje de back-End que suele ser **PHP**, pero también podrían usarse otros como Perl o Python, encargado de procesar la petición y mandar una respuesta que podría ser un archivo HTML con PHP embebido.



Para este conjunto deberíamos aprender, por un lado, los lenguajes del lado del servidor, como PHP, MySQL, y la configuración del .htaccess para apache. Por otro lado, la parte del cliente en la que deberíamos aprender HTML, CSS y JavaScript. Serían 5 lenguajes sin contar htaccess totalmente diferentes que aprender para la creación de una web.

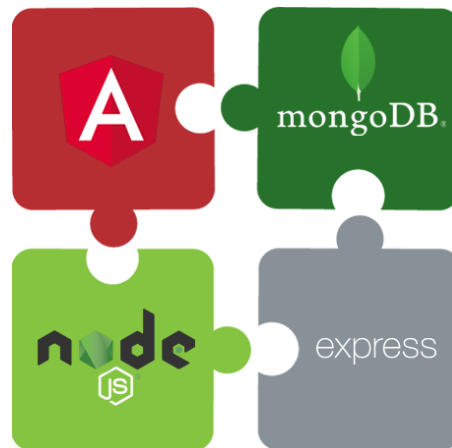
Yo he decidido estructurar mi web app con el web stack **MEAN**, que también debe su nombre a las iniciales de sus tecnologías.

La “M” viene del sistema gestor de base de datos, llamado **Mongo DB**, donde almacenamos la información, es NoSQL y orientado a documentos JSON.

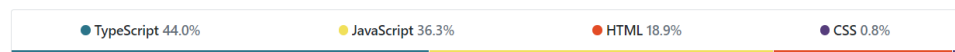
La “E”, es **Express JS**, un framework de Node JS donde controlaremos las peticiones de los clientes pidiendo recursos a Mongo DB, creando una API.

La “A”, es de **Angular**, pero también podría ser React o Vue cambiando el patrón a MERN o MEVN, un framework de Front-End donde crearemos una aplicación web que haga peticiones AJAX a la API y muestre JSON las respuestas como interfaz de usuario.

La “N”, es de **Node JS**, Interprete de JavaScript del lado del servidor con protocolo de HTTP basado en Nginx y sistema de gestión de paquetes NPM.



A diferencia de la metodología LAMP, en esta MEAN, todo está basado en JavaScript, por lo tanto, solo necesitamos aprender un lenguaje además de HTML y CSS. Por otro lado, también mencionar Typescript, un lenguaje que añade tipado estático a JavaScript mejorándolo, el código creado en Typescript posteriormente se compila a JavaScript; Angular está basado en Typescript.



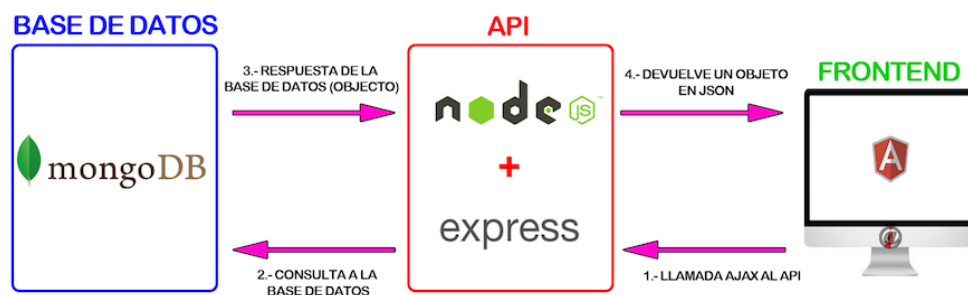
Esta es una captura de GitHub de mi proyecto Ágora, nos muestra el porcentaje de lenguajes usados, el 44% es Typescript, el 36,3% de JavaScript, el 18,9% HTML y el 0,8% de CSS

4.2.1.3. MVC – Modelo, vista, controlador

La arquitectura modelo vista controlador se encarga de organizar la aplicación, separándola en tres partes. El modelo es la base de datos, el controlador es el Back-End y la vista es el Front-End.

De esta forma nosotros le damos unas responsabilidades a cada parte de nuestra aplicación por separado, esto nos permite modificar una parte de la aplicación sin tener que reajustar las demás.

El orden de acciones con esta arquitectura sería como en la siguiente imagen.



El cliente mediante la vista hace peticiones al controlador, según la petición necesitará uno u otros datos que pedirá al modelo, cuando reciba los datos se los devolverá a la vista.

En este proyecto, el cliente mediante un componente de Angular envía una petición a una ruta de la API creada con Express JS, esta ruta, tendrá asociado un controlador que solicitará unos datos mediante una consulta con el ODM Mongoose a la base de datos Mongo DB, si es satisfactoria, responderá con un objeto JSON, este objeto JSON lo devolveremos a la vista, que en mi caso es Angular y también está estructurada con el patrón modelo vista controlador, el servicio de angular le devolverá el objeto JSON a la función que solicitó el cliente y el componente Typescript mostrará el resultado en el componente HTML.

4.2.2. Mongo DB.

Es un sistema gestor de base de datos NoSQL orientado a documentos, de código abierto, multiplataforma y escrito en C++ y desarrollado en 2009.



En lugar de guardar los datos en tablas, como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON, que son archivos JSON binarios con lo que nos asegura mayor rapidez.

Tiene un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil, rápida y escalable.

MongoDB tiene la capacidad de realizar consultas utilizando JavaScript, haciendo que estas sean enviadas directamente a la base de datos para ser ejecutadas.

Para mi proyecto he conectado Node JS a Mongo DB mediante el ODM Mongoose. Y he usado el programa GUI que nos proporciona Mongo, Compass.

Algunas empresas que usan Mongo DB para sus proyectos son: Facebook, Google, Adobe, Cisco, Ebay, SEGA, EA...

4.2.3. Express JS



Express es un framework de Node JS publicado el 16 de noviembre 2010, la última versión se subió el 25 de mayo de 2019. Es muy ligero y flexible, proporciona un conjunto muy robusto de facilidades para crear fácilmente servidores web y recibir peticiones HTTP; por tanto, permite desarrollar API REST de forma muy rápida.

Algunas empresas que usan Express son: Fox Sports, PayPal, Uber e IBM...

4.2.3.1. API

La Interfaz de Programación de Aplicaciones, es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción*.

*La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (característica de caja negra).

En mi proyecto he creado una API con Express a la que accedo desde la aplicación web que he creado en Angular. Mediante la API, puedo hacer CRUD (Create, Read, Update y Delete) en la base de datos.

4.2.4. Angular.

Angular, o Angular (2 – 8), es un framework para aplicaciones web desarrollado en TypeScript, de código abierto, mantenido por Google, se utiliza para crear SPA, aplicaciones web de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador, tiene capacidad de Modelo Vista Controlador (MVC), hace que el desarrollo y las pruebas sean más fáciles.



Angular se basa en clases tipo "Componentes". En dichas clases tenemos propiedades (variables) y métodos (funciones a llamar). Además, usa guardianes para proteger rutas, servicios para comunicarse con la API...

Cada componente es un conjunto de tres archivos, uno HTML, otro CSS y finalmente un TS (También posee un cuarto archivo ".spec.ts", ocupado de hacer pruebas al componente). Cada componente tiene declarado un nombre que podremos usar como etiqueta html para llamarlo.

Angular es la evolución de Angular JS o Angular 1, en esta primera versión estaba basado en JavaScript. Es incompatible ya que en la actualización de la primera a la segunda versión hacen un cambio muy diferente.

La primera versión de Angular se publicó el 14 de septiembre de 2016, la última versión de Angular es Angular 8, publicada el día 28 de mayo 2019. Para mi proyecto he usado Angular 7.2.0, ya que Angular 8 se publicó después.

4.2.4.1. TypeScript.

TypeScript (.TS) es un lenguaje de programación libre y de código abierto desarrollado y mantenido por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade tipos estáticos y objetos



basados en clases. Anders Hejlsberg, diseñador de C# y creador de Delphi y Turbo Pascal, ha trabajado en el desarrollo de TypeScript. TypeScript puede ser usado para desarrollar aplicaciones JavaScript que se ejecutarán en el lado del cliente o del servidor (Node.js).

Todo el código que se crea en los archivos TypeScript debe de ser compilado a JavaScript, ya que los intérpretes solo interpretan el código de JavaScript.

4.2.4.2. SPA – Single Page Application.

El concepto de aplicaciones de una sola página consiste en que el navegador se mantenga sin tener que estar recargándose en cada botón o link que pulsan dentro de una misma web. Este concepto se basa en realizar peticiones AJAX

(Asynchronous JavaScript And XML), la web manda una petición Ajax al servidor y recibe una respuesta, JS modifica el contenido de la web con la respuesta sin tener que recargar la web entera.



Las SPA hace que la UX experiencia de usuario mejore por el aumento de la velocidad, es solo un componente el que se modifica y no la web entera por cada interacción, como las aplicaciones de escritorio.

4.2.4.3. Angular CLI.

Angular posee una Interfaz de Línea de Comandos, esta facilita la escritura de código en Angular, ya que podemos crear proyectos y componentes con comandos muy fáciles. Por ejemplo:



```
...\agora > ng g c inicio
```

Este comando está generando un componente Inicio al proyecto, que son 4 archivos del componente (archivo html, css, ts y spec.ts) más la declaración del componente en el módulo general.

4.2.5. Node JS

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor Chrome V8 de Google. Al contrario que la mayoría del código JavaScript, no se ejecuta en un navegador, sino en el servidor.



Se lanzó por primera vez el 27 de mayo de 2009, su última versión es la 10.15.3 lanzada el 05 de marzo de 2019

4.2.5.1. NPM - Node Package Manager

NPM, es el sistema de gestión de paquetes por defecto para Node JS. Desde el comando npm podemos instalar módulos para usar en nuestro proyecto como JWT



4.3. Bootstrap



Bootstrap es una biblioteca de código abierto para diseño de sitios y aplicaciones web. A diferencia de muchos frameworks web, solo se ocupa del desarrollo front-end.

Bootstrap es el segundo proyecto más destacado en GitHub y es usado por la NASA y la MSNBC entre otras organizaciones

En mi proyecto he usado el cdn de Bootstrap 4 embebido en el html por un lado y Ng-Bootstrap instalado mediante NPM, por otro lado. He intentado usar lo máximo posible Ng-bootstrap pero no está totalmente desarrollado, entonces he tenido también que usar bootstrap 4

4.3.1. Bootstrap 4

Usando Bootstrap el front end de la página se hace más fácil y vistoso desde mi punto de vista, Bootstrap nos proporciona un conjunto de clases y funcionalidades que serían muy costosas de hacer en vanilla pero con una fácil estructura html y unas clases, este framework por ejemplo nos crea carrouseles, headers que se despliegan etc.

El primer lanzamiento fue el 19 de agosto de 2011, la última versión estable es la 4.2.1, lanzada el 21 de diciembre de 2018.



4.3.1.1. JQuery



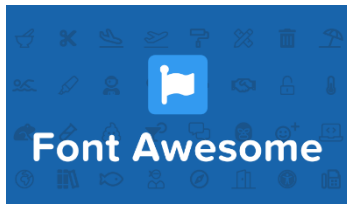
Bootstrap está basado en JQuery, un framework de JavaScript, hay algunas cosas que Bootstrap no puede hacer tan solo con clases, id o atributos data; o que tienen una parte configurable. Es este el momento en el que tenemos que añadir al código un script de JQuery.

4.3.2. Ng-Bootstrap

Ng-Bootstrap es un módulo de Bootstrap diseñado para Angular, eliminando JQuery, además de los demás archivos js que había que añadir como popper.js o el propio Bootstrap.js.



4.3.3. FA Font Awesome



Fuente asombrosa, este es un framework también de front end que nos proporciona iconos a través de clases en las etiquetas `span` o `i`.

4.4. Git

Git es un software de control de versiones diseñado por Linus Torvalds, publicado el 7 de abril de 2005 y siendo la última actualización la versión 2.22.0 publicada el 7 de junio de 2019. Pensando en la eficiencia y la fiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente.

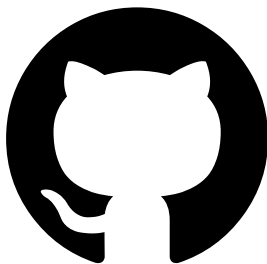


Git se ha convertido en un sistema de control de versiones con funcionalidad plena. Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

He decidido usar Git porque en mi opinión es un software necesario en cualquier proyecto, cualquiera puede perder información fácilmente y de muchas maneras o cambiar un código que a la larga no funcione y queramos recrear el antiguo de nuevo.

Un profesor de sustitución de base de datos en este primer curso, me dijo que trabajaba de programador y los archivos estaban totalmente comentados por cada cambio que hacían, esto es una buena idea, pero genera archivos inmensos que para ver desde cero son imposible. Con Git solucionamos todos estos problemas.

4.4.1. GitHub



GitHub es una plataforma de desarrollo colaborativo, para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora. El software que opera GitHub está escrito en Ruby on Rails. El código de los proyectos alojados en GitHub se almacena típicamente de forma pública, aunque utilizando una cuenta de pago, también permite hospedar repositorios privados.

GitHub fue lanzado el día 8 de febrero de 2008 y el 4 de junio de 2018, Microsoft lo compró por 7.500 millones de dólares.



Ágora está publicado en GitHub en el siguiente link: github.com/rubencastro24/agora

Los comandos para subir el directorio han sido:

```
$ git config --global user.email ruben\_castro\_ruiz@hotmail.es
```

```
$ git config --global user.name "Ruben"
```

Estos comandos configuramos Git

```
$ git init
```

Este comando creamos un repositorio del directorio en Git

```
$ git add ./
```

Este comando añadimos el contenido del directorio al repositorio

```
$ git commit -m "Back-End y Front-End "
```

Este comando guardamos los cambios en el repositorio con un mensaje

```
$ git push -u origin master
```

Este comando envía los cambios que se han hecho en la rama principal de los repertorios remotos

```
$ git remote add origin https://github.com/rubencastro24/agora.git
```

Este comando permite conectar al usuario con el repositorio local a un servidor remoto en mi caso mi cuenta de GitHub.

4.5. Postman



Front-End.

Postman, “cartero” es un software que nos permite enviar una petición a una ruta mediante un método (Get, Post, Put, Delete...), añadiéndole cabeceras, parámetros o contenido body como un formulario. Me ha sido muy útil para desarrollar la API ya que podemos enviarle datos sin tener que crear un

4.6. JWT - JSON WEB TOKEN



JWT es un estándar abierto basado en JSON para la creación de tokens de acceso que permiten la propagación de identidad y privilegios a una API. Un servidor podría generar un token indicando que el usuario tiene acceso y proporcionarlo a un cliente.

El cliente entonces podría utilizar el token para probar que está actuando como un usuario. El token está firmado por la clave del servidor, así que el cliente y el servidor son ambos capaz de verificar que el token es legítimo. Los JSON Web Tokens están diseñados para ser compactos

El token lo enviará el cliente por cabecera http y el servidor comprobará, cada vez que este token es válido. Además, podrá obtener datos como su id.

5. Explicación Back-End.

En este apartado voy a explicar toda la parte del servidor que he programado en Express JS. Los nombres de las variables o funciones los he puesto con el sistema camelCase.



5.1. package.json

Lo primero que debemos hacer como buena práctica al empezar un proyecto es crear un package.json, este lo creamos con el comando:

```
npm init
```

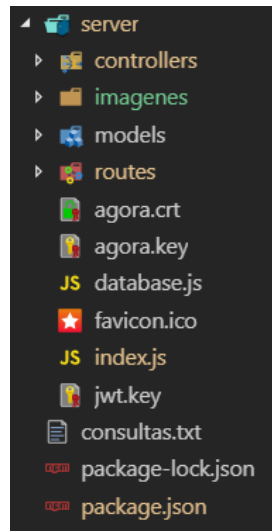
```
packagejson > ...
1 {
2   "name": "agora-server",
3   "version": "1.0.0",
4   "description": "red social",
5   "main": "server/index.js",
6   "scripts": {
7     "start": "node server/index.js",
8     "devStart": "nodemon server/index.js",
9     "test": "echo \\\"Error: no test specified\\\" && exit 1"
10  },
11  "keywords": [],
12  "author": "Rubén Castro",
13  "license": "ISC",
14  "dependencies": {
15    "bcrypt": "^3.0.6",
16    "cors": "^2.8.5",
17    "express": "^4.16.4",
18    "express-jwt": "^5.3.1",
19    "fs": "0.0.1-security",
20    "http": "0.0.0",
21    "https": "^1.0.0",
22    "jsonwebtoken": "^8.5.1",
23    "mongoose": "^5.5.5",
24    "morgan": "^1.9.1",
25    "multer": "^1.4.1"
26  },
27  "devDependencies": {
28    "nodemon": "^1.19.0"
29  },
30  "repository": {
31    "type": "git",
32    "url": "git+https://github.com/rubencastro24/agora.git"
33  },
34  "bugs": {
35    "url": "https://github.com/rubencastro24/agora/issues"
36  },
37  "homepage": "https://github.com/rubencastro24/agora#readme"
38 }
```

Este comando nos creará una estructura JSON con todo el contenido que necesitamos o declaramos para nuestra API.

Usando el comando `npm install [nombre]`, he instalado los módulos que se ven en el objeto `dependencies` `bcrypt`, `cors`, `express`, `express-jwt`, `fs`, `http`, `https`, `jsonwebtoken`, `mongoose`, `Morgan` y `multer`. Por otro lado hay otra `devDependencies`, en esta están los módulos de desarrollo, como `nodemon`. Instalado como `npm install nodemon -D`.

Podemos encontrar otro objeto llamado `script`, en el que he declarado `start` y `devStart`, con estos comandos podemos arrancar el servidor.

Los demás datos son declarativos como el nombre, la descripción... Y las rutas para GitHub.



Este es el directorio del servidor, podemos ver el archivo index.js justo como se declaraba en la propiedad main del package.json.

5.2. Index.

En este archivo index.js es donde configuro toda la parte general del servidor. Para verlo lo vamos a separar en varias partes.

```
server ▸ JS index.js ▸ ...
1  // requerimientos.
2  const express = require('express');
3  const app = express();
4
5  const { mongoose } = require('./database');
6  const http = require('http');
7  const https = require('https');
8
9  const fs = require('fs');
10 const morgan = require('morgan');
11 const cors = require('cors');
12 const jwt = require('jsonwebtoken');
13 const expressJwt = require('express-jwt');
14 const bcrypt = require('bcrypt');
15 const multer = require('multer');
16 const path = require('path');
17
```

Esta es la parte donde requerimos otros archivos o módulos instalados.

En la primera línea pido express y lo guardo en una constante llamada express, en la segunda lo ejecuto y lo guardo en app. App ahora va a tener métodos que nos proporciona express.

En la quinta línea pido mongoose de ./database, que es el archivo de configuración de la base de datos Mongo DB.

En las siguientes líneas pido los módulos que he instalado previamente.


```

18 // Configuración.
19 app.set('httpPort', process.env.HTTP_PUERTO || 80);
20 app.set('httpsPort', process.env.HTTPS_PUERTO || 443);
21 app.set('salt', process.env.SALT || 10);
22 app.set('jwtSecret', process.env.JWT_SECRET || fs.readFileSync(__dirname + '/jwt.key'));
23

```

Aquí establezco algunos datos que se van a utilizar como el puerto http, el https, los salt que serán para la encriptación de las contraseñas y por ultimo la clave secreta de JWT que usaremos para crear las sesiones. Las variables las pido de las variables de entorno, pero si no encuentra valor ahí, se lo doy yo, por ejemplo, http 80, https 443, salt 10 y jwt el contenido del archivo jwt.key. Esto lo hago para que cuando suba mi código, no sepan mi clave privada de jwt, ya que este archivo lo ignoro en la subida.

```

24 // Middlewares.
25 app.use(morgan('dev'));
26 app.use(express.json());
27 app.use(express.urlencoded({extended: false}));
28 app.use(cors({origin: 'http://localhost:4200'}));
29

```

El siguiente paso es configurar los middlewares que son funciones que se ejecutaran entre la petición y la respuesta de ahí su nombre.

Usamos el módulo Morgan que nos permite ver las peticiones que se hacen a nuestro servidor.

```

OPTIONS /api/seguimientos/5cfc54b92988251234121e1b 204 4.742 ms - 0
OPTIONS /api/usuarios/5cfc54b92988251234121e1b 204 0.376 ms - 0
OPTIONS /api/seguidores/5cfc54b92988251234121e1b 204 0.368 ms - 0
OPTIONS /api/seguir/5cfc54b92988251234121e1b 204 0.121 ms - 0
OPTIONS /api/seguimiento/5cfc54b92988251234121e1b 204 0.155 ms - 0
GET /api/usuarios/5cfc54b92988251234121e1b 304 2046.240 ms - -
GET /api/seguir/5cfc54b92988251234121e1b 304 3049.322 ms - -
GET /api/seguimiento/5cfc54b92988251234121e1b 304 3048.404 ms - -
GET /api/seguidores/5cfc54b92988251234121e1b 304 4061.209 ms - -
GET /api/seguimientos/5cfc54b92988251234121e1b 304 4111.319 ms - -
OPTIONS /api/seguimiento/5cfc54b92988251234121e1b 204 0.100 ms - 0
PUT /api/seguimiento/5cfc54b92988251234121e1b 200 121.902 ms - 221
OPTIONS /api/seguimientos/5cfc54b92988251234121e1b 204 0.180 ms - 0
GET /api/seguimiento/5cfc54b92988251234121e1b 200 7.080 ms - 189
OPTIONS /api/seguidores/5cfc54b92988251234121e1b 204 0.216 ms - 0
GET /api/seguimientos/5cfc54b92988251234121e1b 304 11.036 ms - -
GET /api/seguidores/5cfc54b92988251234121e1b 200 9.822 ms - 154

```

Por cada petición nos muestra el método, la ruta, el código de respuesta de estado http, el tiempo tomado en milisegundos y el costo de Bytes.

Express.json() y Express.urlencoded() son para obtener los datos que se reciben json y los parámetros que vienen por la URL.

El último, cors, sirve para que nuestra app Angular pueda conectarse y enviar datos a la API.

```

31 // Rutas.
32 app.use(require('./routes/routes'));
33

```

Aquí llamamos a las rutas, requiriendo un archivo llamado routes dentro de un directorio routes.

```
34 // Arrancar el servidor. Demonio.  
35 var credenciales = {  
36   key: fs.readFileSync(__dirname + '/agora.key').toString(),  
37   cert: fs.readFileSync(__dirname + '/agora.crt').toString()  
38 };  
39  
40 http.createServer(app).listen(app.get('httpPort'), function(){  
41   console.log("Servidor http en el puerto " + app.get('httpPort'));  
42 }  
43 );  
44 https.createServer(credenciales, app).listen(app.get('httpsPort'), function(){  
45   console.log("Servidor https en el puerto " + app.get('httpsPort'));  
46 }  
47 );
```

Por último arrancamos el servidor, para ello usamos el método `createServer()` de `http` y `https` con el parámetro `app`, que es el objeto del servidor `express`, en caso de `https`, también añadiremos las credenciales y lo ponemos en escucha con el método `listen()` añadiendo el parámetro del puerto y hacemos una función que ejecute un `console.log` con el puerto.

Para las credenciales de `https` he usado el programa `OpenSSL`



Y he usado los siguientes comandos:

```
\https>openssl genrsa -des3 -out agora.pem 2048
```

```
\https>openssl req -new -key agora.pem -out agora.csr
```

```
\https>openssl x509 -req -days 365 -in agora.csr -signkey agora.pem -out  
agora.crt
```

```
\https>openssl rsa -in agora.pem -out agora.key
```

Solo necesitaba los dos últimos archivos, el `crt` y el `key` pero para crearlos necesitaba crear los anteriores.

Estos dos archivos los leo con `fs.readFileSync()` y los añado al objeto.

5.3. Base de datos.

El archivo `database` es donde configuramos el ODM `Mongoose` para conectarnos a `Mongo DB`.

```
server > JS databasejs > ...
1  const mongoose = require('mongoose');
2
3  const URI = 'mongodb://localhost/agora'
4
5  mongoose.connect(URI, {useNewUrlParser: true})
6    .then(db => console.log('Base de datos conectada'))
7    .catch(err => console.error(err));
8
9  module.exports = mongoose;
```

Requerimos el módulo ya instalado Mongoose y lo guardamos en una constante llamada mongoose, porque este requerimiento nos devuelve un objeto. En otra constante guardamos la ruta de la base de datos a la que conectarnos con mongoose.

Usando del objeto mongoose el método connect() e introduciendo la ruta de la conexión, la URI ya guardada. Usamos las promesas .then() y .catch(), estas se usan porque la respuesta no es inmediata, entonces se debe esperar a que se establezca la conexión. Si es satisfactoria then ejecutará su la función y mostrará base de datos conectada por consola. En caso contrario, mostrar el error del fallo de conexión. Por ultimo hago un module.exports = mongoose para que otros archivos como index, lo puedan ejecutar.

5.4. Rutas.

En el archivo de rutas creo los end point de la API.

```
server > routes > JS routes.js > ...
1  const express = require('express');
2  const router = express.Router();
3
4  const path = require('path');
5  const multer = require('multer');
6  const imagenesSubidas = multer({
7    dest: path.join(__dirname, '../imagenes/publicaciones')
8  })
9
10 const usuarioControlador = require('../controllers/usuario.controlador');
11 const seguimientosControlador = require('../controllers/seguimientos.controlador');
12 const publicacionesControlador = require('../controllers/publicaciones.controlador');
```

Requiero express y lo ejecuto con el método .Routes() guardándolo en router.

Pido path y multer, path es para las rutas de los directorios y multer es para recibir archivos en las peticiones, en mi caso imágenes para las publicaciones.

Guardo en imagenesSubidas con multer el destino que le doy a los archivos.

Requiero controladores de usuarios, seguimientos y publicaciones.

```
17 /* Ruta de inicio */
18 router.get('/api',(req, res) => {res.json({type:true, data: 'Bienvenido a la API de Ágora'}});
19
```

En esta ruta por Get a /api doy una bienvenida a la API.

```
23 /* Ruta de imagenes */
24 router.get('/api/imagenes/publicaciones/:id', publicacionesControlador.mostrarImagenPublicacion);
25 router.get('/api/imagenes/perfil/:id', publicacionesControlador.mostrarImagenPerfil);
26
```

En estas rutas Get a /api/imagenes/(publicaciones o perfil)/:id hago públicas las imágenes para que se pueda acceder mediante la url. Utilizo el controlador de publicaciones.

```
30 /* Ruta para los controladores de Usuarios */
31 router.post('/api/iniciar-sesion', usuarioControlador.iniciarSesion);
32 router.post('/api/registrarse', usuarioControlador.registrarse);
33 router.get('/api/comprobar-token', usuarioControlador.getComprobarToken);
34 router.get('/api/comprobar-token-bool', usuarioControlador.getComprobarTokenBool);
35
36 /* Middleware */
37 /* comprobador de token y proteger siguientes rutas */
38 router.use(usuarioControlador.comprobarToken);
39
40 router.get('/api/usuarios', usuarioControlador.obtenerUsuarios);
41 router.get('/api/usuarios/buscar/:nick', usuarioControlador.obtenerUsuariosBusqueda);
42 router.get('/api/usuarios/:id', usuarioControlador.obtenerUsuario);
43 router.put('/api/usuarios/:id', usuarioControlador.editarUsuario);
44 router.delete('/api/usuarios/:id', usuarioControlador.borrarUsuario);
45
```

Estas son las rutas de para los usuarios.

Haciendo un post a /api/iniciar-sesión e introduciendo el Nick y la contraseña.

Haciendo un post a /api/registrarse e introduciendo los datos de registro.

Luego hay dos rutas que he creado para desarrollo en las que compruebo token.

Lo siguiente es un middleware que comprobará si tenemos token y es verdadero, de lo contrario no nos dejará pasar a las siguientes rutas porque nos responderá con un error.

Las siguientes tres rutas son get a usuarios y nos permite obtener los usuarios que existen en la base de datos, buscar por Nick y obtener un usuario en concreto por id.

La siguiente ruta es un put a usuarios donde podremos editar los campos del perfil.

Finalmente, un delete, donde se borrará el usuario de la base de datos por id.

Con estas rutas estamos creando un CRUD de usuarios donde podemos crearlos con un registro, listarlos a todos, mediante una búsqueda por Nick o a uno por su id. Podremos editarlo y por último, borrarlo.

```
50 /* Ruta para los controladores de Seguidores */
51 router.put('/api/seguimiento/:id', seguimientosControlador.cambiarSeguimiento);
52
53 router.get('/api/seguimiento/:id', seguimientosControlador.obtenerSeguimiento);
54 router.get('/api/seguidor/:id', seguimientosControlador.obtenerSeguidor);
55
56 router.get('/api/seguidores/:id', seguimientosControlador.obtenerUsuariosSeguidores);
57 router.get('/api/seguimientos/:id', seguimientosControlador.obtenerUsuariosSeguimientos);
```

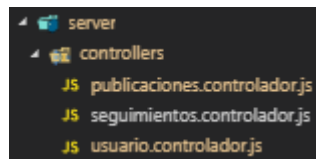
Para los seguimientos he creado 5 rutas, la primera es una put que alternara el seguimiento que un usuario hace a un perfil. Dos siguientes comprueban si el usuario sigue a dicho perfil y si dicho perfil sigue al usuario. Las dos últimas son para mostrar los seguidores totales y los seguimientos totales que hace un usuario.

```
61 /* Ruta para los controladores de Publicaciones */
62 router.post('/api/publicaciones', imagenesSubidas.single('imagen'), publicacionesControlador.hacerPublicacion)
63
```

La ultima ruta es para hacer publicaciones requiere de un middleware de multer en el objeto imagenesSubidas recibe la imagen con el método .single('imagen').

5.5. Controladores.

Los controladores son los que realizan las acciones en cada petición a ruta.



```
server
├── controllers
│   ├── publicaciones.controlador.js
│   ├── seguimientos.controlador.js
│   └── usuario.controlador.js
```

En la carpeta controllers es donde tengo los controladores.

```
server > controllers > JS usuario.controlador.js > ...
1  const UsuariosModelo = require('../models/usuario');
2  const jwt = require('jsonwebtoken');
3  const expressJwt = require('express-jwt');
4  const fs = require('fs');
5  const path = require('path');
6  const jwtSecret = process.env.JWT_SECRET || fs.readFileSync(path.join(__dirname, '../jwt.key'));
7  const usuarioControlador = {};
```

En los archivos controladores tenemos que requerir el modelo para poder hacer peticiones a la base de datos. En este caso, al ser los usuarios, vamos a necesitar requerir jsonwebtoken y express-jwt, aparte de fs y path que los usamos para obtener el contenido de jwt.key que es la clave secreta.

Finalmente creamos un objeto que iremos añadiendo métodos para las rutas.

```
7  usuarioControlador.obtenerUsuarios = async (req, res) => {
8    await UsuariosModelo.find(
9      function (err, usuarios) {
10        if (err) {
11          res.json({
12            type: false,
13            data: "Error: " + err
14          });
15        }
16        else {
17          if (!usuarios) {
18            res.json({
19              type: false,
20              data: "Usuarios no encontrados "
21            });
22          }
23          else {
24            res.json({
25              type: true,
26              usuarios
27            });
28          }
29        }
30      })
31  }
```

Este es el controlador de usuarios que nos muestra todos los usuarios. creamos el método obtenerUsuarios en el objeto usuarioControlador. Tendrá como parámetros req y res, petición y respuesta. Al consultar la base de datos necesitamos ponerle Async. Dentro de la función llamamos al método find() que nos proporciona mongoose para consultar la base de datos con el objeto UsuariosModelo. El await que hay delante es por el Async, al hacer consultas a las bases de datos esta función tardará mas en responder y lo declaramos con esto.

A la consulta no le he puesto ningún parámetro porque quiero todos los resultados.

Esta consulta devuelve una función que tenemos como parámetros err y usuarios. Si devuelve err es que ha habido un error con la base de datos. Si no devuelve usuarios, es que los usuarios no se han encontrado. Si devuelve usuarios, enviamos una respuesta con todos los usuarios.

Este es un controlador fácil.

```
170 usuarioControlador.iniciarSesion = async (req, res) => {
171   req.body.nick = req.body.nick.toLowerCase();
172   const usuarioIniciarSesion = {
173     nick: req.body.nick,
174     pass: req.body.pass
175   }
176   await UsuariosModelo.findOne(usuarioIniciarSesion, function(err, usuario) {
177     if (err) {
178       res.json({
179         type: false,
180         data: "Error: " + err
181       });
182     }
183     else {
184       if (!usuario) {
185         res.json({
186           type: false,
187           data: "nick o contraseña incorrecta."
188         });
189       }
190       else {
191         var token = jwt.sign({idUsuario: usuario._id}, jwtSecret, {expiresIn: '24h'});
192         res.json({
193           type: true,
194           data: usuario,
195           token
196         });
197       }
198     }
199   });
200 }
```

Este es el controlador en el que iniciamos sesión. Tiene una estructura similar, pero más contenido.

Lo primero que hacemos es pasar el Nick que nos han enviado en la petición a minúscula, para que no de problemas de mayúsculas. Ahora crearemos un objeto usuarioIniciarSesion con dos propiedades, Nick y pass, que son los datos que nos ha enviado. Ahora llamo al método UsuariosModelo.findOne(), este encontrara un resultado con los valores que le hemos introducido como parámetro de usuarioIniciarSesion. En caso de que encuentre un resultado, crearemos un token con el id del usuario y la clave secreta y le establecemos como tiempo para caducarse, 24 horas.

El token y el usuario los enviamos como respuesta.

```

202 usuarioControlador.registrarse = async (req, res) => {
203   req.body.nick = req.body.nick.toLowerCase();
204   await UsuariosModelo.findOne({ "nick": req.body.nick }, function(err, usuarioEncontrado) {
205     if (err) {
206       res.json({
207         type: false,
208         data: "Error: " + err
209       });
210     }
211     else {
212       if (usuarioEncontrado) {
213         if (usuarioEncontrado.nick === req.body.nick) {
214           res.json({
215             type: false,
216             data: "El usuario ya existe.",
217             nick: usuarioEncontrado.nick
218           });
219         }
220       }
221       else {
222         var usuarioCrear = new UsuariosModelo();
223         usuarioCrear.nick = req.body.nick;
224         usuarioCrear.pass = req.body.pass;
225         usuarioCrear.nombre = req.body.nombre;
226         usuarioCrear.apellidos = req.body.apellidos;
227         usuarioCrear.fechaNacimiento = new Date(req.body.fechaCumple.año, req.body.fechaCumple.mes - 1, req.body.fechaCumple.dia);
228         usuarioCrear.fechaCreacionUsuario = new Date();
229         usuarioCrear.save();
230         var token = jwt.sign({idUsuario: usuarioCrear._id}, jwtSecret, {expiresIn: '24h'});
231         res.json({
232           type: true,
233           data: usuarioCrear,
234           token
235         });
236       }
237     }
238   });
239 }
240

```

Para el registro, hacemos una consulta del Nick para comprobar si existe o no. Si no existe, entonces creamos un objeto con el modelo, para incorporarlo en la base de datos, e introducimos los datos del registro. Tras llamar al método .save() del objeto creado, se almacenará en la base de datos, además, le creamos un token para tener iniciada la sesión.

```

usuarioControlador.comprobarToken = async (req, res, next) => {
  const token = req.headers['x-access-token'];
  if (token) {
    jwt.verify(token, jwtSecret, (err, tokenDescodificado) => {
      if (err) {
        res.status(401).json({
          type: false,
          data: "autenticación fallida: " + err
        });
      }
      else {
        req.tokenDescodificado = tokenDescodificado;
        next();
      }
    });
  }
  else {
    res.status(401).json({
      type: false,
      data: "Necesitas un token."
    });
  }
}

```

Para el middleware comprobar token, el token lo recibimos por cabecera http, y si existe token lo comprobamos con el método que nos proporciona jsonwebtoken, verify si hay un error le mandamos una

respuesta con estado 401 de no autorizado. Si no existe token también. Si existe token hacemos un next() y continuamos a las siguientes rutas.

```
101 usuarioControlador.editarUsuario = async (req, res) => {
102   const { id } = req.params;
103   const usuario = {
104     nick: req.body.nick,
105     nombre: req.body.nombre,
106     apellidos: req.body.apellidos,
107     descripcion: req.body.descripcion,
108     pass: req.body.pass,
109     fechaNacimiento: new Date(req.body.fechaNacimiento.año, req.body.fechaNacimiento.mes - 1, req.body.fechaNacimiento.día)
110   }
111   console.log(usuario.nick);
112   await UsuariosModelo.findOne({ "nick": usuario.nick }, function (err, nick) {
113     if (err) {
114       res.json({
115         type: false,
116         data: "Error al buscar: " + err
117       });
118     }
119     else {
120       if (nick && nick.nick == usuario.nick && id != nick._id) {
121         res.json({
122           type: false,
123           nick,
124           data: "nick en uso"
125         });
126       }
127       else {
128         UsuariosModelo.findByIdAndUpdate(id, {$set: usuario}, {$new: true}, function (err, editado) {
129           if (err) {
130             res.json({
131               type: false,
132               nick,
133               data: "Error al modificar: " + err
134             });
135           }
136           else {
137             if (!editado) {
138               res.json({
139                 type: false,
140                 editado,
141                 data: "no se pudo editar."
142               });
143             }
144             else {
145               res.json({
146                 type: true,
147                 editado,
148                 data: "usuario editado"
149               });
150             }
151           }
152         });
153       }
154     }
155   });
156 }
```

El controlador para editar el usuario recibe todos los datos. Comprueba si el Nick introducido es el mismo que poseía u otro diferente que no exista, para cambia los datos.

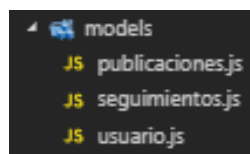
```
seguimientosControlador.cambiarSeguimiento = async (req, res) => {
  const seguimiento = {
    usuario: req.tokenDescodificado.idUsuario,
    sigue: req.params.id
  }
  await SeguimientosModelo.findOne(seguimiento, function(err, sigue) {
    if (err) {
      res.json({
        type: false,
        cambio: false,
        data: "Error: " + err
      });
    }
    else {
      if (sigue) {
        SeguimientosModelo.findByIdAndRemove(sigue._id, function(err, borrado){
          if (err) {
            res.json({
              type: false,
              cambio: false,
              data: "Error borrar seguimiento: " + err
            });
          }
          else {
            if (!borrado) {
              res.json({
                type: true,
                cambio: false,
                data: "Error borrar seguimiento."
              });
            }
            else {
              res.json({
                type: true,
                cambio: true,
                status: 'seguimiento borrado'
              });
            }
          }
        })
      }
    }
  });
}
```

```
    }  
    else {  
      var guardarSeguimiento = new SeguimientosModelo(seguimiento);  
      guardarSeguimiento.fecha = new Date();  
      guardarSeguimiento.save(function (err, creado) {  
        if (err) {  
          res.json({  
            type: false,  
            cambio: false,  
            data: "Error de seguimiento: " + err  
          });  
        }  
        else {  
          if (!creado) {  
            res.json({  
              type: true,  
              cambio: false,  
              data: "Error de seguimiento."  
            });  
          }  
          else {  
            res.json({  
              type: true,  
              cambio: true,  
              status: 'seguimiento creado',  
              seguimiento: creado  
            });  
          }  
        }  
      });  
    }  
  }  
});  
}
```

El controlador de cambio de seguimiento comprueba si el seguimiento existe y lo deja de seguir, sino, lo sigue.

5.6. Modelos.

Los modelos son los archivos donde declaro la estructura de datos con mongoose de cada colección.



```
models  
├── publicaciones.js  
├── seguimientos.js  
└── usuario.js
```

Tenemos los tres modelos que también tenemos de controladores.

```
server > models > JS usuario.js > ...  
1  const mongoose = require('mongoose');  
2  const { Schema } = mongoose;  
3  
4  const UsuarioSchema = new Schema ({  
5    nick: { type: Schema.Types.String, required: true},  
6    pass: { type: Schema.Types.String, required: true},  
7    nombre: { type: Schema.Types.String, required: true},  
8    apellidos: { type: Schema.Types.String, required: true},  
9    descripcion: { type: Schema.Types.String, required: false},  
10   fechaNacimiento: { type: Schema.Types.Date, required: true},  
11   fechaCreacionUsuario: { type: Schema.Types.Date, required: true}  
12  });  
13  
14  module.exports = mongoose.model('usuarios', UsuarioSchema);
```

Este es el modelo de los usuarios en el que creo un objeto UsuarioSchema con Schema de mongoose, en este objeto declaro las propiedades del objeto a crear en mongo. Ponemos el nombre de la propiedad, el tipo de dato y si es requerido.

Finalmente, hacemos un module.export para poder usarlo. Dentro de mongoose.model() el string 'usuarios' será el nombre de la colección en mongo.

```
server > models > JS seguimientos.js > ...
1  const mongoose = require('mongoose');
2  const { Schema } = mongoose;
3
4  const SeguimientosSchema = new Schema ({
5    usuario: { type: Schema.Types.ObjectId, required: true},
6    sigue: { type: Schema.Types.ObjectId, required: true},
7    fecha: { type: Schema.Types.Date, required: true}
8  });
9
10 module.exports = mongoose.model('seguimientos', SeguimientosSchema);
```

Este es el modelo de los seguimientos.

Un usuario sigue a otro en tal fecha.

```
server > models > JS publicaciones.js > ...
1  const mongoose = require('mongoose');
2  const { Schema } = mongoose;
3
4  const publicacionesSchema = new Schema ({
5    usuario: {type: Schema.Types.ObjectId, required: true},
6    imagen: {type: Schema.Types.string, required: true},
7    descripcion: {type: Schema.Types.string, required: false},
8    fechaPublicacion: {type: Schema.Types.Date, required: false},
9    comentarios: {
10     type: [
11       {
12         usuario: {type: Schema.Types.String, required: true},
13         comentario: {type: Schema.Types.String, required: true},
14         fechaComentario: {type: Schema.Types.Date, required: true}
15       }
16     ],
17     required: false
18   },
19   meGusta: {
20     type: [
21       {
22         usuario: {type: Schema.Types.String, require: true},
23         fechaMeGusta: {type: Schema.Types.Date, require: true}
24       }
25     ],
26     required: false
27   }
28 });
29
30 module.exports = mongoose.model('publicaciones', publicacionesSchema);
```

Y este es el modelo de las publicaciones.

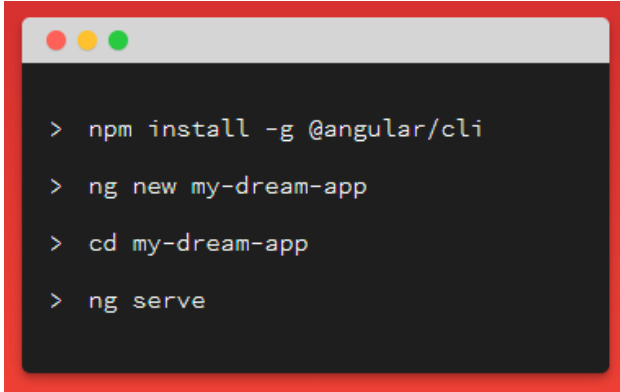
Un usuario hace una publicación, con una imagen almacenada en una ruta del servidor, una descripción y una fecha. En esta imagen un usuario, puede comentar en tal fecha. Y un usuario puede dar me gusta en tal fecha.

6. Explicación de Front-End.

En esta sección explicaré las partes del Front-End que he programado con el framework Angular.

6.1. Angular CLI

Para comenzar el proyecto he instalado Angular CLI.

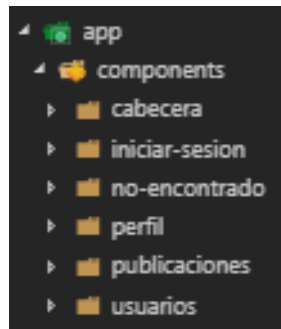
A terminal window with a dark background and a red border. It shows the following commands being executed:

```
> npm install -g @angular/cli
> ng new my-dream-app
> cd my-dream-app
> ng serve
```

Con npm lo instalo de forma global, con ng new creo el proyecto, accedo al directorio y con el comando ng serve, lo compilo para que se abra en el puerto 4200 que es de testing.

6.2. Componentes.

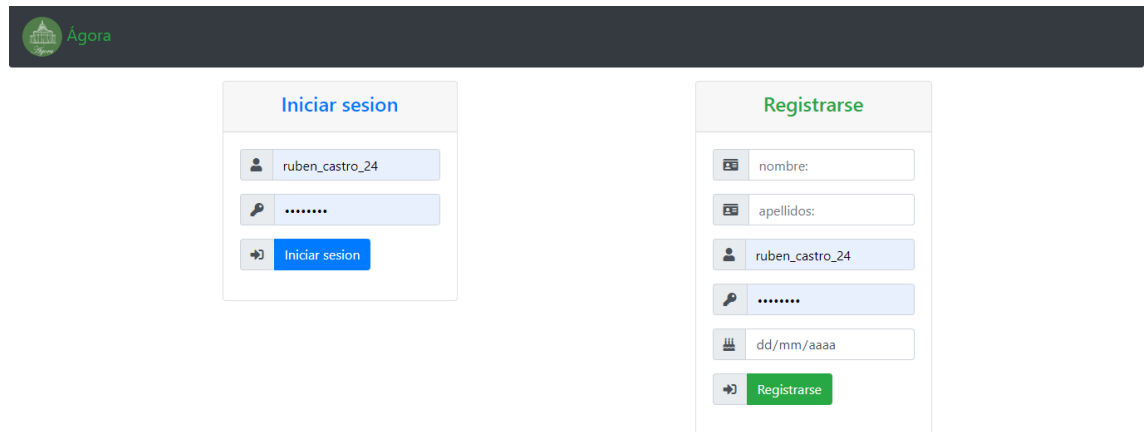
Los componentes son la parte visible que contienen el html, css y ts
Los componentes creados han sido:



Cabecera, en el que muestro un menú de navegación.



Iniciar-sesion, es donde puedo hacer log in y registrarme



The image shows two web forms side-by-side. The left form is titled 'Iniciar sesion' and contains input fields for 'nick' (with the value 'ruben_castro_24') and 'pass' (with masked characters '*****'), followed by an 'Iniciar sesion' button. The right form is titled 'Registrarse' and contains input fields for 'nombre:', 'apellidos:', 'nick' (with the value 'ruben_castro_24'), 'pass' (with masked characters '*****'), and a date field 'dd/mm/aaaa', followed by a 'Registrarse' button.

En su archivo typescript, llamo al servicio que consumirá la API.

```
public iniciarSesion() {  
  if (this.nick && this.pass){  
    this.sesion.iniciarSesion(this.nick, this.pass)  
      .pipe(first())  
      .subscribe(  
        resultado => {  
          if (resultado == true){  
            this.router.navigate(['']);  
          }  
          else {  
            this.errorI = 'Error de autenticación.';  
          }  
        },  
        error => {  
          this.errorI = 'Error de autenticación. ' + error;  
        }  
      );  
  }  
  else {  
    this.errorI = 'Introduce todos los datos'  
  }  
}
```

Al iniciar sesión en nuestra web, necesitamos el Nick y la contraseña, entonces, llamaremos al servicio. Nos subscribimos al método, y esperamos el resultado, si es true, significa que la autenticación es correcta, si no, mostrará el error de autenticación. Y si no introducimos las credenciales, nos mostrará que introduzcamos todos los datos

La Suscripción es del paradigma de programación orientada a Reactivos RxJS. En ésta, nos subscribimos a eventos, que pueden cambiar y esperamos a que cambien para mostrar el cambio.

```
calcularEdad(){
  var fechaSeparada : any[] = this.fecha.split('-');
  this.fechaCumple = {
    dia: fechaSeparada[2],
    mes: fechaSeparada[1],
    año: fechaSeparada[0]
  };
  var fechaHoy : Date = new Date();
  if ( (fechaHoy.getMonth()+1 > this.fechaCumple.mes) ||
    (fechaHoy.getMonth()+1 == this.fechaCumple.mes && fechaHoy.getDate() >= this.fechaCumple.dia) ){
    this.edad = fechaHoy.getFullYear() - this.fechaCumple.año;
  }
  else{
    this.edad = fechaHoy.getFullYear() - this.fechaCumple.año - 1;
  }
}

public registrar() {
  if (this.nombre && this.apellidos && this.nickR && this.passR && this.fecha) {
    this.calcularEdad();
    if (this.edad >= this.limiteEdad){
      let nick = this.nickR;
      let pass = this.passR;

      this.sesion.registrarse(this.nombre, this.apellidos, nick, pass, this.fechaCumple)
        .subscribe(
          resultado => {
            if (resultado == true){
              this.router.navigate(['']);
            }
            else {
              this.errorR = 'El usuario ya existe.';
            }
          },
          error => {
            this.errorR = `Error de registro. ${error}`;
          }
        );
    }
    else {
      this.errorR = `Debes de ser mayor de ${this.limiteEdad} años.`;
    }
  }
  else {
    this.errorR = 'Introduce todos los datos'
  }
}
```

Para el registro necesitaremos ser mayores de edad, por lo tanto, he creado una función para calcular la edad. Cuando nos registramos, si la edad es mayor o igual a 18, e introduzcamos todos los datos, se llamará a el método del servicio para registrarse.

No encontrado, cuando accedes a una ruta no existente:



Error 404


Parece que andas perdido...

¡La pagina que buscas no está disponible!

[Volver a inicio](#)

Perfil es donde puedo ver cada perfil:

Propio:



@ruben_castro_24


X publicaciones 4 seguidores 3 seguimientos

[Subir publicación](#) [Configurar perfil](#)

Rubén Castro Ruiz

Me gusta la programación!

Otro diferente:



@mariamuot

X publicaciones 3 seguidores 2 seguimientos

Te sigue, Le sigues [Dejar de seguir](#)

María Muot Martín

Me encantan los libros!


```
constructor(  
  public usuario$: UsuariosService,  
  public modalService: NgbModal,  
  public sesion$: IniciarSesionService,  
  public seguimientos$: SeguimientosService,  
  public ruta : ActivatedRoute,  
  public router : Router  
) {  
  this.ruta.params  
    .subscribe(  
    params => {  
      this.id = params['id'];  
      this.miId = this.sesion$.miId();  
      this.obtenerUsuario();  
      this.obtenerSeguimientosPerfil();  
      this.obtenerSeguidoresPerfil();  
  
      if (this.id !== this.miId) {  
        this.obtenerSeguimientoPerfil();  
        this.obtenerSeguidorPerfil();  
      }  
    }  
  )  
}
```

En el componente typescript del perfil, en el constructor necesitamos inyectar todos los servicios que vamos a usar, como el de usuarios, iniciar sesión, las rutas, el modal de que muestra los seguidores... Y nos subscribiremos al parámetro de la ruta, que es la id del usuario, si esta ID es diferente a la nuestra saldrá un perfil ajeno, sino, el nuestro. Además, debemos llamar a los métodos de los seguimientos para que se actualicen.

```
seguidoresModal($seguidores) {  
  this.modalService.open(seguidores, {backdropClass: 'bg-secondary' });  
}  
seguimientosModal($seguimientos) {  
  this.modalService.open(seguimientos, {backdropClass: 'bg-secondary' });  
}
```

Para el modal, escribimos la estructura que nos proporciona Ng-bootstrap.



Si pulsamos en el boton de los seguidores, podemos ver cada uno de los seguidores en el Modal de ng-bootstrap.



En el botón de los seguimientos veremos a los usuarios que seguimos.

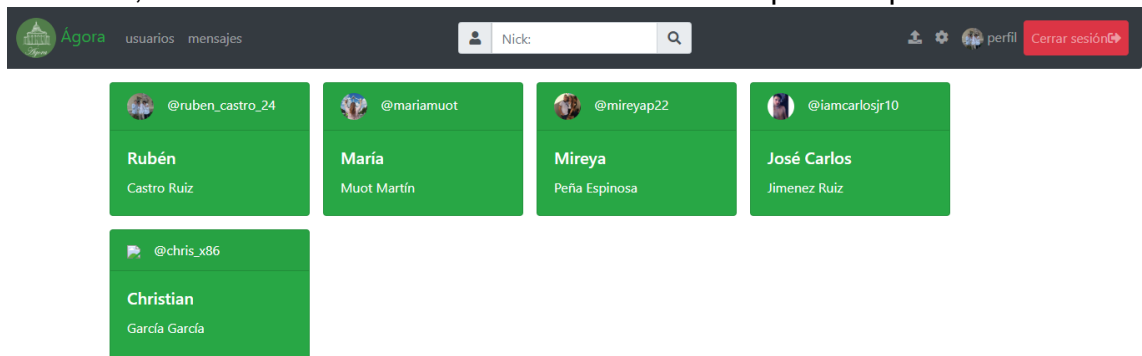
```

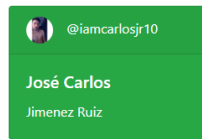
cambiarSeguimientoPerfil() {
  this.seguimientos$.cambiarSeguimiento(this.id)
    .subscribe(
      respuesta => {
        var resultado : any = respuesta;
        if (!resultado.type) {
          console.log("error seguimiento.")
        }
        else {
          if (!resultado.cambio) {
            console.log("cambio de seguimiento nulo.");
          }
          else {
            this.cambioSeguimientoMensaje = resultado.status;
            this.obtenerSeguimientoPerfil();
            this.obtenerSeguimientosPerfil();
            this.obtenerSeguidoresPerfil();
          }
        }
      },
      error => {
        console.log(error);
      }
    );
}

```

La función para cambiar de seguimiento es, una llamada al servicio de seguimientos para cambiar el seguimiento, introduciendo la id. Nos subscribimos, si no obtenemos resultado, mandamos error, de lo contrario, si no se obtiene un cambio mandamos un error de cambio nulo, pero si obtenemos cambio, llamamos a los métodos para cambiar el estado visual.

Usuarios, es donde se muestran todos los usuarios o por búsqueda:





```
obtenerUsuarios() {
  this.usuarios$.obtenerUsuarios()
    .subscribe(
      respuesta => {
        var resultado : any = respuesta;
        if (resultado.type == true){
          this.usuarios = resultado.usuarios as Usuarios[];
        }
        else {
          console.log("no se encuentran usuarios.");
        }
      },
      error => {
        console.log(error);
      }
    );
}

buscarUsuariosQuery() {
  this.usuarios$.obtenerUsuariosBusqueda(this.nick)
    .subscribe(
      respuesta => {
        var resultado : any = respuesta;
        if (resultado.type == true){
          this.usuarios = resultado.usuarios as Usuarios[];
        }
        else {
          console.log("no se encuentran usuarios.");
        }
      },
      error => {
        console.log(error);
      }
    );
}
```

Los métodos son una suscripción en la que si obtenemos respuesta, la almacenamos en una variable, que la mostraremos en el Front-End, sino, mandamos error

Hacer publicación:

Hacer una publicación

 Elige imagen:

Browse

 Descripción:

 Subir publicación

```
public hacerPublicacion() {
  this.error="";
  this.respuesta="";
  if (this.imagen) {
    const formData = new FormData();
    formData.append('imagen', this.form.get('imagen').value);

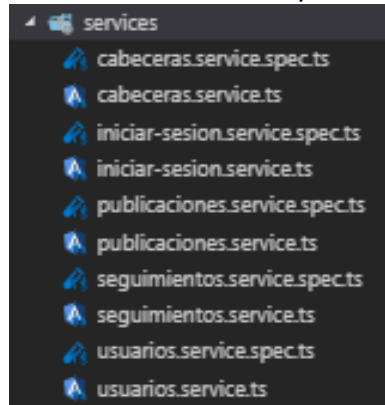
    this.publicaciones$.hacerPublicacion(formData, this.descripcion)
      .subscribe(
        resultado => {
          if (resultado.type == true){
            this.respuesta = 'Publicación realizada con éxito.';
          }
          else {
            this.error = 'Publicación no realizada.';
          }
        },
        error => {
          this.error = `Error de publicación. ${error}`;
        }
      );
  }
  else {
    this.error = 'Selecciona imagen.'
  }
}

onFileChange(event) {
  if (event.target.files.length > 0) {
    const file = event.target.files[0];
    this.form.get('imagen').setValue(file);
  }
}
```

Para hacer una publicación, tenemos que seleccionar una imagen, entonces esta se almacenará en `this.form.get('imagen').value` gracias al cambio de la función de abajo `onFileChange`. Teniendo el archivo imagen, lo enviamos al servidor en la función `hacer publicación` que enviará la imagen al servicio y esta a la api. Nos subscribimos y si obtenemos un resultado `true`, se habrá publicado, sino habrá habido algún error.

6.3. Servicios.

Los servicios son la parte donde envío las peticiones a la API.



He creado casi un servicio por cada componente:

```
cabeceras.service.ts x
agora-app > src > app > services > cabeceras.service.ts
1  import { Injectable } from '@angular/core';
2
3  @Injectable({
4    providedIn: 'root'
5  })
6  export class CabecerasService {
7
8    cabeceras : {};
9
10   constructor() { }
11
12   setHeaders(){
13     this.cabeceras={
14       headers: {
15         "Content-Type": "application/JSON",
16         "x-access-token": localStorage.getItem('token')
17       }
18     };
19   }
20
21   setHeadersPlain(){
22     this.cabeceras={
23       headers: {
24         "x-access-token": localStorage.getItem('token')
25       }
26     };
27   }
28
29 }
```

En el servicio de cabeceras lo que controlo son las cabeceras que envío a la API, aquí es donde controlo el token. Este servicio lo uso es cada uno de los otro servicio, he creado la función setheaders() y setHeadersPlain()

En la primera, añado el content-Type: JSON, este lo envío en todos, menos las publicaciones que el mime es imagen y no texto y en la otra función no añado el content-Type JSON, lo uso para las publicaciones.

```

18
19 public miId(){
20     if (this.token) {
21         var valoresToken : any = this.jwtHelper.decodeToken(this.token);
22         return valoresToken.idUsuario
23     }
24     console.log("error de token");
25 }
26
27 iniciarSesion(nick: string, pass: string): Observable<boolean> {
28     return this.http.post<{token: string}>(this.URL_API + '/api/iniciar-sesion', {nick, pass})
29         .pipe(
30             map(resultado => {
31                 if (resultado.token){
32                     localStorage.setItem('token', resultado.token);
33                     return true;
34                 }
35                 return false;
36             })
37         );
38 }
39
40 registrarse(nombre: string, apellidos: string, nick: string, pass: string, fechaCumple: {}): Observable<boolean> {
41     return this.http.post<{token: string}>(this.URL_API + '/api/registrarse', {nombre, apellidos, nick, pass, fechaCumple})
42         .pipe(
43             map(resultado => {
44                 if (resultado.token){
45                     localStorage.setItem('token', resultado.token);
46                     return true;
47                 }
48                 return false;
49             })
50         );
51 }
52
53 cerrarSesion() {
54     localStorage.removeItem('token');
55 }
56
57 public get sesionIniciada(): boolean {
58     this.token = localStorage.getItem('token');
59     if (this.token){
60         return true
61     }
62     return false
63 }

```

Este es el servicio del inicio de sesión donde tengo funciones para obtener mi id con mi token, iniciar sesión, en la que envió por post las credenciales para loguearme y al recibir el token lo guardo en almacenamiento local. El Registro que es igual que el inicio de sesión solo que envió todos los datos del usuario.

Cerrar sesión, en el que borro el token.

Sesión iniciada, en el que compruebo que tiene token.

El método map() me devuelve el valor de la propiedad token.

```

public hacerPublicacion(imagen, descripcion) {
    this.cabeceras$.setHeadersPlain();

    console.log(imagen)
    return this.http.post<{type: boolean, data: string}>(`${this.URL_API}/publicaciones`, imagen, this.cabeceras$.cabeceras);
}

```

El servicio de publicaciones, donde envío la imagen del formulario por post a la api.

```

public cambiarSeguimiento(id: Seguidores['sigue']) {
  this.cabeceras$.setHeaders();
  return this.http.put(`${this.URL_API}/seguimiento/${id}`, {}, this.cabeceras$.cabeceras);
}

public obtenerSeguimiento(id: Seguidores['sigue']) {
  this.cabeceras$.setHeaders();
  return this.http.get(`${this.URL_API}/seguimiento/${id}`, this.cabeceras$.cabeceras);
}

public obtenerSeguidor(id: Seguidores['sigue']) {
  this.cabeceras$.setHeaders();
  return this.http.get(`${this.URL_API}/seguidor/${id}`, this.cabeceras$.cabeceras);
}

public obtenerSeguidores(id: Seguidores['usuario']) {
  this.cabeceras$.setHeaders();
  return this.http.get(`${this.URL_API}/seguidores/${id}`, this.cabeceras$.cabeceras);
}

public obtenerSeguimientos(id: Seguidores['sigue']) {
  this.cabeceras$.setHeaders();
  return this.http.get(`${this.URL_API}/seguimientos/${id}`, this.cabeceras$.cabeceras);
}

```

Los seguimientos, envío la ruta que el usuario pide por get o put estado en algún perfil o pulsando el botón de seguimiento.

```

public obtenerUsuarios() {
  this.cabeceras$.setHeaders();
  return this.http.get(this.URL_API, this.cabeceras$.cabeceras);
}

public obtenerUsuariosBusqueda(nick: Usuarios['nick']) {
  this.cabeceras$.setHeaders();
  return this.http.get(`${this.URL_API}/buscar/${nick}`, this.cabeceras$.cabeceras);
}

public obtenerUsuario(id: Usuarios['_id']) {
  this.cabeceras$.setHeaders();
  return this.http.get(`${this.URL_API}/${id}`, this.cabeceras$.cabeceras);
}

/* public crearUsuario(usuario: Usuarios) {
  return this.http.post(this.URL_API, usuario, {headers: this.cabeceras});
} */

public editarUsuario(usuario: Usuarios) {
  return this.http.put(this.URL_API + `/${usuario._id}`, usuario);
}

public borrarUsuario(usuario: Usuarios) {
  return this.http.delete(this.URL_API + `/${usuario._id}`);
}

```

El servicio de usuarios en el que envío igualmente la petición a la API

6.4. Modelos.

Los modelos son la parte en la que le damos un tipado a una clase, para luego a un objeto darle el tipo modelo.

```
export class Usuarios {  
  _id: string;  
  nick: string;  
  pass: String;  
  nombre: string;  
  apellidos: string;  
  descripción: string;  
  fechaNacimiento: Date;  
  fechaCreacionUsuario: Date;  
}
```

El modelo de usuarios es el mismo que en el back end pero añadimos `_id`, ya que en el back end lo añade mongo db por nosotros.

```
export class Seguidores {  
  _id: string;  
  usuario: string;  
  sigue: String;  
}
```

El modelo seguidores, igual, id de seguimiento, usuario que hace seguimiento y usuario al que sigue.

```
export class Publicaciones {  
  _id: string;  
  usuario: string;  
  descripción: string;  
  imagen: File;  
  fechaPublicacion: Date;  
  comentarios: [  
    {  
      usuario: string,  
      comentario: string,  
      fechaComentario: Date  
    }  
  ];  
  meGusta: [  
    {  
      usuario: string,  
      fechaMeGusta: Date  
    }  
  ];  
}
```

En las publicaciones, la misma estructura.

6.5. Guardianes.

Los guardianes son quienes nos protegen las rutas, yo tengo un guardián que me defiende las rutas si el cliente no tiene token.


```
agora-app > src > app > guards > iniciar-sesion.guard.ts > ...
1 import { Injectable } from '@angular/core';
2 import { Router, CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
3 import { HttpClient } from '@angular/common/http';
4 import { IniciarSesionService } from '../services/iniciar-sesion.service';
5 import { JwtHelperService } from '@auth0/angular-jwt';
6
7 @Injectable()
8 export class IniciarSesionGuard implements CanActivate {
9   constructor(
10     public router: Router,
11     public http: HttpClient,
12     public iniciarSesionS: IniciarSesionService,
13     public jwtHelper: JwtHelperService
14   ) { }
15
16   readonly URL_API = 'http://localhost:80'
17
18   canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
19     var token : string = localStorage.getItem('token');
20     if (!token) {
21       this.router.navigate(['iniciar-sesion']);
22       return false;
23     }
24     else {
25       if (this.jwtHelper.isTokenExpired(token)){
26         this.iniciarSesionS.cerrarSesion();
27         this.router.navigate(['iniciar-sesion']);
28         return false;
29       }
30       return true;
31     }
32   }
33 }
34 }
```

Requerimos módulos como JwtHelperService, httpClient e Iniciar sesionService.

Los inyectamos en el constructor para usarlos en la clase.

Y creamos la función canActivate() que en caso de no tener token no manda a iniciar sesión, de lo contrario si existe token, pero no es válido también nos envía a iniciar sesión y de lo contrario nos permite estar en la ruta.

6.6. Rutas.

Las rutas en Angular nos mandarán a componentes, de ese modo es cómo funcionan las aplicaciones de una sola página.

```
agora-app > src > app > app-routing.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { IniciarSesionGuard } from '../guards/iniciar-sesion.guard';
5
6  import { UsuariosComponent } from '../components/usuarios/usuarios.component';
7  import { NoEncontradoComponent } from '../components/no-encontrado/no-encontrado.component';
8  import { PerfilComponent } from '../components/perfil/perfil.component';
9  import { IniciarSesionComponent } from '../components/iniciar-sesion/iniciar-sesion.component';
10 import { PublicacionesComponent } from '../components/publicaciones/publicaciones.component';
11
12 const routes: Routes = [
13   { path: 'iniciar-sesion', component: IniciarSesionComponent },
14
15   { path: 'usuarios', component: UsuariosComponent, canActivate: [IniciarSesionGuard] },
16   { path: 'usuarios/buscar/:nick', component: UsuariosComponent, canActivate: [IniciarSesionGuard] },
17   { path: 'usuarios/:id', component: PerfilComponent, canActivate: [IniciarSesionGuard] },
18
19   { path: 'subir-publicacion', component: PublicacionesComponent, canActivate: [IniciarSesionGuard] },
20   { path: 'publicacion/:id', component: PublicacionesComponent, canActivate: [IniciarSesionGuard] },
21
22   { path: '404', component: NoEncontradoComponent },
23   { path: '', redirectTo: '', pathMatch: 'full', canActivate: [IniciarSesionGuard] },
24   { path: '**', /*redirectTo: '404',*/ component: NoEncontradoComponent }
25 ];
26
27 @NgModule({
28   imports: [ RouterModule.forRoot(routes) ],
29   exports: [ RouterModule ]
30 })
31 export class AppRoutingModule { }
32
```

Las rutas las metemos en un array llamado routes, y estas son objetos que contienen el nombre de la ruta, el componente que cargan, y si es necesario protegerla, el canActivate que hemos creado en el Guardián

La ultima ruta es '**' y significa que si la ruta pedida, no es ninguna de las anteriores, cargar esta, que lo manda al componente NoEncontrado.

7. Propuestas a mejorar o implementar.

Como propuestas a mejorar o implementar me gustaría añadir, mensajes privados, acabar las publicaciones y un sistema de notificaciones.

Con esa base ya me gustaría añadir otros frameworks como por ejemplo Socket.io, que me da la posibilidad de aumentar el Ajax de la web, dándonos funcionalidades como chat en directo sin recarga o notificaciones instantáneas.

también añadir ionic, para crear una aplicación web progresiva, que pueda ser instalada en dispositivos móviles.

8. Bibliografía.

https://developer.mozilla.org/es/docs/Web/JavaScript/Introducci%C3%B3n_a_JavaScript_orientado_a_objetos

<https://developer.mozilla.org/es/docs/Web/API/Window/sessionStorage>

https://developer.mozilla.org/es/docs/Web/API/API_de_almacenamiento_web/Usando_la_API_de_almacenamiento_web

<https://medium.com/williambastidasblog/estructura-de-una-api-rest-con-nodejs-express-y-mongodb-cdd97637b18b>

<https://www.mongodb.com/es>

<https://expressjs.com/es/4x/api.html>

<https://angular.io/tutorial>

<https://nodejs-es.github.io/api/modules.html>

<https://www.toptal.com/angular/angular-6-jwt-authentication>

<https://jarroba.com/mean-mongo-express-angular-node-ejemplo-de-aplicacion-web-parte-ii/>

<https://academia-binaria.com/comunicaciones-http-en-Angular/>

<https://code.tutsplus.com/es/tutorials/jwt-authentication-in-angular--cms-32006>

<https://code.tutsplus.com/es/tutorials/token-based-authentication-with-angularjs-nodejs--cms-22543>

<https://danielpozoblog.wordpress.com/2017/11/19/autenticacion-con-nodejs-y-jwt/>

<https://mongoosejs.com/docs/schematypes.html>

<https://charlascylon.com/2013-07-10-tutorial-mongodb-operaciones-de-consulta-avanzadas>

<https://stackoverflow.com/questions/14763721/mongoose-delete-array-element-in-document-and-save>

<https://scotch.io/tutorials/express-file-uploads-with-multer>

<https://malcoded.com/posts/angular-file-upload-component-with-express/>