



Universidad
Politécnica
de Cartagena



GRADO EN INGENIERÍA TELEMÁTICA

**Desarrollo web para la divulgación de
proyectos ecológicos**

Autora: Natalia Pascual M^aDolores

Director: Daniel Pérez Berenguer

Codirector: Mathieu Kessler

Diciembre de 2020

Indice

Capítulo I: Introducción	5
1. Objetivos	6
2. Análisis de requerimientos	7
3. Solución propuesta	9
Capítulo II: Desarrollo del backend	11
1. Instalaciones y configuraciones previas	11
2. Configuración de DB Mongo Atlas	13
3. Conexión entre Node y Mongo	15
4. Configuración CORS	16
5. Variables de entorno	17
6. Modelos de datos	18
7. Middleware	18
8. Seguridad y encriptación	20
9. CRUD	25
10. Subida de archivos al servidor	28
Capítulo III: Desarrollo del frontend	29
1. Estructura y organización del código	29
2. Módulo de rutas	32
3. Formularios reactivos para el login y el registro de usuarios	34
4. Login de Google	36
5. Acceso desde Angular al servicio REST	38
6. Servicios	39
7. Componentes	41
8. Guards	44
9. Pipes	46
10. Gestión de roles	46

11.Sistema de puntuación y favoritos	47
Capítulo IV: Optimización, tests y despliegues	48
1. Lazy load	48
2. Pruebas unitarias	49
3. Generar build de distribución	49
4. Desplegar la aplicación en Netlify	51
Capítulo V: Conclusiones y líneas futuras	52
1. Posibles mejoras	52
2. Conclusiones	53
Capítulo VI: Bibliografía y documentación técnica	55

Capítulo I: Introducción

En la actualidad el calentamiento global es una de las principales preocupaciones de científicos de todo el mundo. Está demostrado que este fenómeno ha sido causado casi exclusivamente por la actividad humana en el planeta, si se sigue la tendencia actual la vida en la tierra dentro de unos años será totalmente imposible

Alguna de las evidencias del rápido aumento de las temperaturas en todo el planeta son:

Calentamiento de los mares y océanos: El mar absorbe gran parte de este calor, y su temperatura ha aumentado 0.33 grados Celsius desde 1969

Disminución del hielo: Las capas de hielo de Groenlandia y la Antártida han disminuido en masa. Los datos del último estudio de la NASA muestran que Groenlandia perdió un promedio de 279 mil millones de toneladas de hielo por año entre 1993 y 2019, mientras que la Antártida perdió 148 mil millones de toneladas de hielo por año

Desaparición de los glaciares: Los glaciares están desapareciendo de casi todas las partes del mundo, incluidos los Alpes, el Himalaya y los Andes

Eventos naturales extremos: Los récords de altas temperaturas en todo el mundo ha ido en aumento, mientras que el récord de bajas temperaturas ha ido disminuyendo desde 1950. También hemos sido testigos del número creciente de lluvias intensas, huracanes e incendios.

Acidificación del mar: Desde el comienzo de la Revolución Industrial, la acidez del mar ha ido en aumento hasta casi un 30%. Este fenómeno es debido a que los humanos emitimos más dióxido de carbono a la atmósfera y debido a esto, el océano absorbe más cantidad.

Estamos en un momento histórico en el que necesitamos un plan de acción por parte de todos los gobiernos a nivel global para frenar el cambio climático y la pérdida de diversidad, de ello depende nuestro

bienestar y futuro.

Aunque el panorama no es muy esperanzador lo cierto es que en los últimos años se han conseguido avances, a veces espectaculares. Por ejemplo, en la cumbre del clima COP21 se firmó el acuerdo de París, En él se especifica un plan de acción a nivel mundial cuyo objetivo es que la temperatura del planeta no suba más de 1,5 grados.

Pero aún así estas medidas están siendo insuficientes, pues siguen existiendo intereses que se resisten a este cambio. Por eso es tan importante el cambio de paradigma en la sociedad.

Cada vez más, las personas buscan a la hora de consumir que sus elecciones sean responsables con el medio ambiente, y gracias a ello, las empresas han tenido que adaptarse para cubrir esas necesidades. Además, se han creado nuevos modelos de negocio para cubrir este mercado de energías renovables y productos sostenibles.

Si la tendencia de la gente es darle importancia a esta problemática, al final a las grandes empresas no les quedará mas remedio que adaptarse a los nuevos tiempos. Por eso es tan importante la concienciación colectiva.

1. Objetivos

Las nuevas tecnologías forman parte del cambio y son una herramienta clave para la transformación de la sociedad. Somos seres sociales que nos gusta compartir información y sentirnos comunicados en todo momento, de ahí el gran éxito de las redes sociales.

Combinando estos dos conceptos nade la idea de crear una aplicación web que permita divulgar los proyectos ecológicos llevados a cabo por estudiantes y centros a los que pertenecen.

Este proyecto pretende impulsar a las personas más jóvenes a tomar conciencia de la situación en la que vivimos y que se involucren de

forma activa en el cambio hacia hábitos más sostenibles. Así como a motivarles a participar en actividades relacionadas con el medio ambiente que organicen en sus colegios o instituciones. Tomando así un papel activo, creando ellos mismos sus propias iniciativas, e influyendo a su vez en su entorno más cercano.

Se pretende dar visibilidad a las propuestas innovadoras que desarrollen tanto alumnos como docentes, motivándoles a llevar a cabo dichos proyectos y competir con otras clases y otros centros por estar entre los más populares.

2. Análisis de requerimientos

La aplicación debe ser accesible desde cualquier dispositivo, responsive y con una interfaz de usuario intuitiva. Debe ser rápida con el tratamiento y visualización de los datos ofreciendo una buena experiencia de usuario. Además, deberá ser escalable para facilitar el desarrollo de nuevas funcionalidades y servicios, además de proporcionar seguridad y encriptación en el tratamiento de datos de los usuarios. Con todo ello se debe elegir un stack tecnológico que se ajuste a estos requerimientos.

Deberá tener las siguientes funcionalidades:

Distintos roles de usuario

- **Administradores:** Serán los que gestionen la aplicación en su totalidad, Podrán dar de alta a las instituciones (centro de secundaria, primaria, universidad, etc.). Tendrán la posibilidad de cambiar el rol del resto de usuarios, modificar información de cualquier usuario, centro y proyecto.
- **Moderadores:** Serán los docentes de cada centro, podrán asignar usuarios a la institución, crear nuevos proyectos, además de editar información de usuarios y proyectos del centro al que pertenecen.
- **Usuarios:** Serán los alumnos de las instituciones, podrán añadir nuevos proyectos en el marco del centro en el que esté estudiando, así como editarlos y borrarlos.

Favoritos

- Cada usuario de la plataforma podrá elegir cuales son sus proyectos favoritos y quedará registrado en la aplicación.

Sistema de puntuaciones

- Los proyectos serán sometidos a votación popular y existirá un ranking de proyectos más votados. Se establecerán unos porcentajes para cada tipo de voto. Cada vez que un usuario añada a sus favoritos un proyecto, la puntuación de este proyecto aumentará un punto si es usuario y en 2 puntos si es moderador o administrador.

Login

- Se podrá acceder a la aplicación registrándose previamente o con el usuario de Google a través de Gmail.

Creación de proyectos

- Los proyectos tendrán título, resumen y descripción, además se podrán añadir imágenes, vídeos de Youtube con su iframe, etc.

Vistas

- La aplicación tendrá una página principal donde se podrán ver todos los centros registrados en la aplicación con sus proyectos mas votados.
- Página interior por cada centro donde aparezcan los proyecto de un centro de manera que se pueda compartir su propia URL.
- Página con el ranking de proyectos mas votados en toda la aplicación
- Para cada usuario una vista de proyectos que ha marcado como favorito

Área de gestión

- Todos los usuarios dispondrán de un panel de control desde donde podrán editar, crear y borrar proyectos. Además los usuarios con roles de administrador y moderador podrán realizar las modificaciones que permita su rol desde este área.

3. Solución propuesta

Tras estudiar los requerimientos y funcionalidades que va a tener la aplicación se ha decidido desarrollarla utilizando el stack MEAN.

Para tomar esta decisión se tuvo en cuenta que este stack de tecnologías se adapta bien a todo tipo de proyectos y arquitecturas, además permite crear aplicaciones web más ligeras y rápidas, la experiencia de usuario es similar a la de usar una aplicación de escritorio. Una característica crucial es que se pueden separar las funcionalidades en módulos, lo que facilita el mantenimiento. Pero, además, permiten atender a muchos más clientes conectados al mismo tiempo sin saturar los recursos del servidor.

Este conjunto de tecnologías es uno de los más usados para desarrollo de aplicaciones web a nivel profesional, MEAN son las siglas de MongoDB, Express, Angular y Node.js. Esto implica que vamos a desarrollar toda la aplicación en todas sus capas con JavaScript.

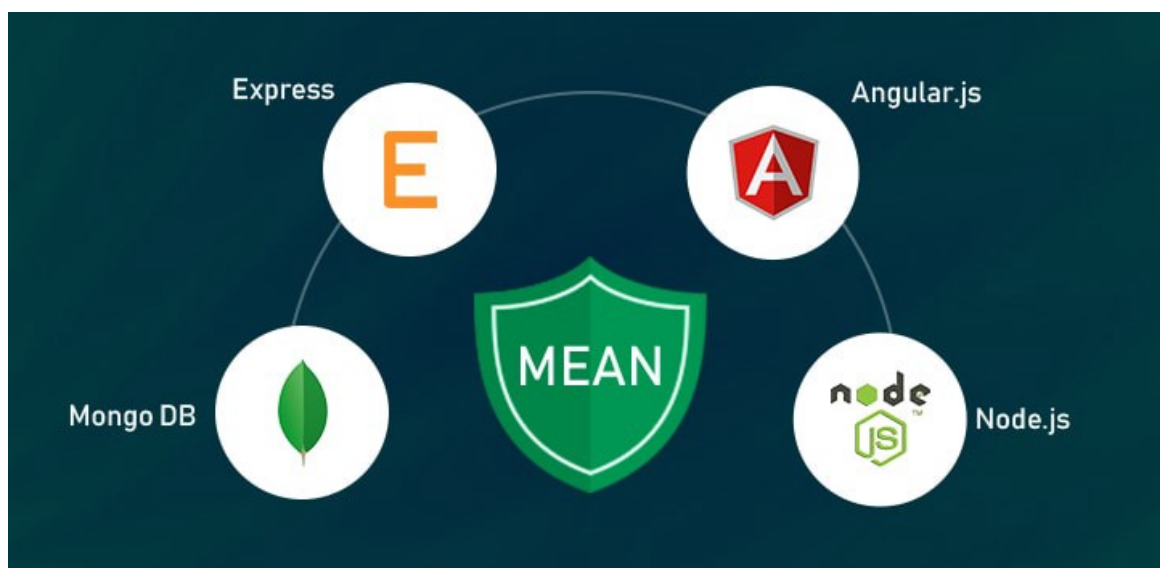


Figura 1: Logotipos MEAN Stack

A continuación se describen las 4 tecnologías que constituyen este entorno:

- **MongoDB:** Es una base de datos de tipo NoSQL, es decir, esta orientada a documentos y es de esquema libre. Entre sus ventajas podemos destacar la alta capacidad de almacenamiento, la escalabilidad y velocidad. De las bases de datos no SQL, es la más sofisticada y utilizada hoy en día.
- **Express:** Es un framework para NodeJS que facilita la creación de la arquitectura REST, implementa los mecanismos necesarios para establecer la comunicación HTTP entre el cliente y el servidor.
- **AngularJS:** Es el framework Javascript que en los últimos años está dominando el mercado. Ahora en el navegador también se utiliza el patrón de diseño MVC (Modelo-Vista-Controlador) y existen muchos frameworks especializados en facilitarnos su uso. De entre todos ellos Angular destaca, ya que está creado y soportado por Google, es gratuito y de código abierto.
- **NodeJS:** Es el lenguaje Javascript sacado de su contexto de ejecución habitual, el navegador. Es el entorno de desarrollo de la capa del servidor, y permite desarrollar rápidamente aplicaciones escalables.

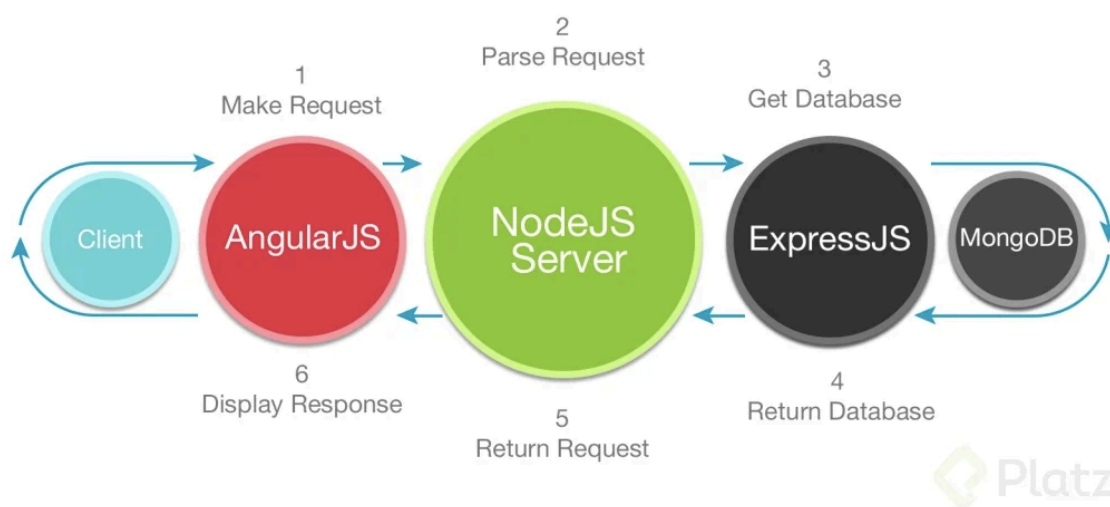


Figura 2: Flujo de datos MEAN stack. Fuente: Platzi

Capítulo II: Desarrollo del backend

El Backend es la parte que no es visible o accesible de manera directa por los usuarios, contiene la lógica de la aplicación que opera con los datos.

En la parte del backend crearemos todos los endpoints que llamaremos desde el cliente, desde él realizaremos todas las consultas necesarias a la base de datos, y en definitiva albergará toda la lógica necesaria para que nuestra aplicación funcione.

1. Instalaciones y configuraciones previas

Para empezar nuestro desarrollo necesitaremos tener node instalado en nuestro ordenador, vamos a utilizar la versión 15.0.1. Al instalar node también se instalará automáticamente npm (Node Package Manager). npm es un gestor de paquetes, lo utilizaremos principalmente para instalar librerías y añadir dependencias de una manera rápida y sencilla.

Hay que tener en cuenta que cuando añadimos nuevos paquetes con npm los instalamos de manera local en nuestro proyecto dentro de la carpeta `node_modules`, pero existe la posibilidad de indicarle que lo instale de manera global en caso necesario.

Algunos módulos vienen instalados por defecto en Node.js y no hará falta usar npm para ello, a estos módulos se les conoce como “módulos nativos”.

el primer paso sería generar un archivo `package.json` es el punto de partida de nuestra aplicación, en el se definen las dependencias que se van a instalar y los scripts y comando necesarios para realizar algunas tareas.

Para crearlo dentro de la carpeta del proyecto ejecutamos en la consola el siguiente comando:

```
npm init -y
```

También necesitaremos instalar express podemos hacerlo desde npm ejecutando el siguiente comando:

```
npm install express --save
```

Añadiendo el flag `--save` hacemos que esta dependencia quede registrada en el `package.json`.

Lo siguiente que necesitaremos es importar el modulo Express e iniciar nuestro servidor de Express, si no especificamos el puerto utilizaría el puerto 4200, que es en el que escucha por defecto angular. En el caso de nuestra aplicación le hemos especificado el puerto 3000 y le hemos añadido una función de callback para que lance un mensaje por consola para saber que el servidor esta desplegado y en que puerto está escuchando.

```
const express = require('express');
```

Figura 3: Importación de express

```
// Lectura y parseo del body  
app.use( express.json() );
```

Figura 4: Iniciando el servidor express

```
app.listen( process.env.PORT, () => {  
  console.log('Servidor corriendo en puerto ' + process.env.PORT );  
});
```

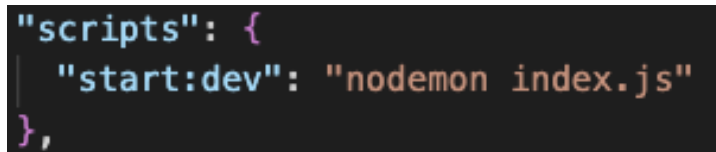
Figura 5: Indicando el puerto en el que escucha en servidor

A continuación instalaremos la librería nodemon, es una utilidad que monitorea los cambios en el código fuente que se está desarrollando y automáticamente reinicia el servidor. Es una herramienta muy útil para desarrollo de aplicaciones en nodejs. Lo instalaremos con el siguiente comando:

```
npm install -g nodemon
```

Lo ejecutamos como administrador y de forma global.

Crearemos un script para iniciar el servidor, lo declaramos dentro de package.json:



```
"scripts": {  
  "start:dev": "nodemon index.js"  
},
```

Figura 6: Comando para iniciar el servidor

Gracias a este script podremos levantar nuestra aplicación servidor ejecutando este comando en el directorio donde se encuentra nuestro proyecto:

```
npm run start:dev
```

2. Configuración de DB Mongo Atlas

Mongo Atlas es un servicio de Mongo en la nube, podemos crear un clúster de MongoDB en cualquier proveedor que elijamos y comenzar a usar este cluster en cuestión de minutos. Para utilizarlo hay que crearse una cuenta de MongoDB Atlas, lo podemos hacer fácilmente cediendo

a su página principal. En la versión gratuita solo se puede crear un cluster que para el inicio de esta aplicación es suficiente.

Mongo hace automáticamente todas las configuraciones necesarias en el cluster y puede tardar unos minutos en estar funcionando.

Antes de utilizar el clúster, usted tendremos que introducir algunos detalles relacionados con la seguridad.

En primer lugar, en la sección de usuarios de MongoDB, debemos crear un nuevo usuario. A continuación, en la sección IP Whitelist, debemos proporcionar una lista de direcciones IP que van a acceder a el cluster. Por ahora, es suficiente proporcionar la dirección IP actual del ordenador. También podemos poner 0.0.0.0 si queremos que se pueda acceder desde cualquier IP.

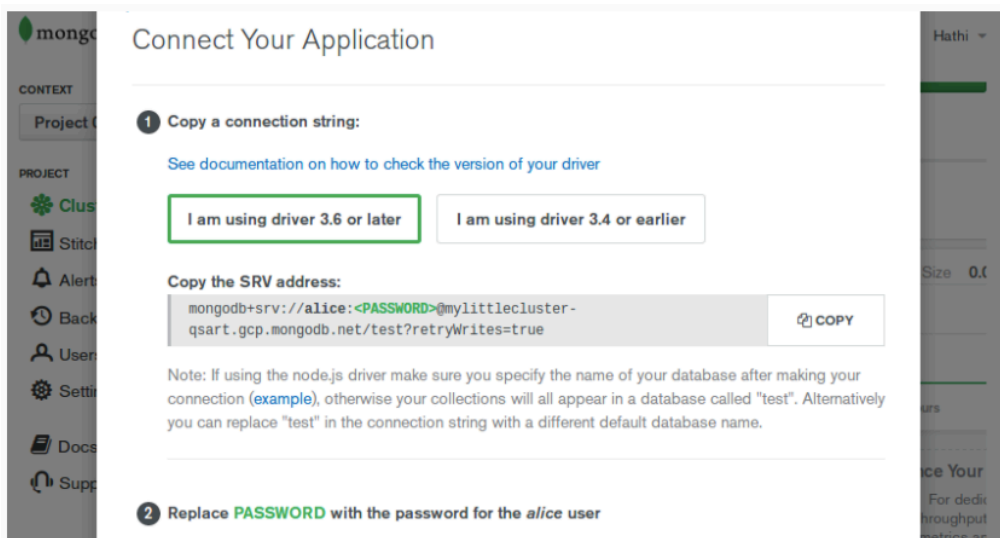


Figura 7: Configuración Mongo Atlas

Quando termine de configurarse el cluster hacemos click en Connect y elegimos la opción de configurar utilizando Mongo compass, continuación copiamos el string de conexión y lo introducimos en mongoDB Compass en el apartado new connection.

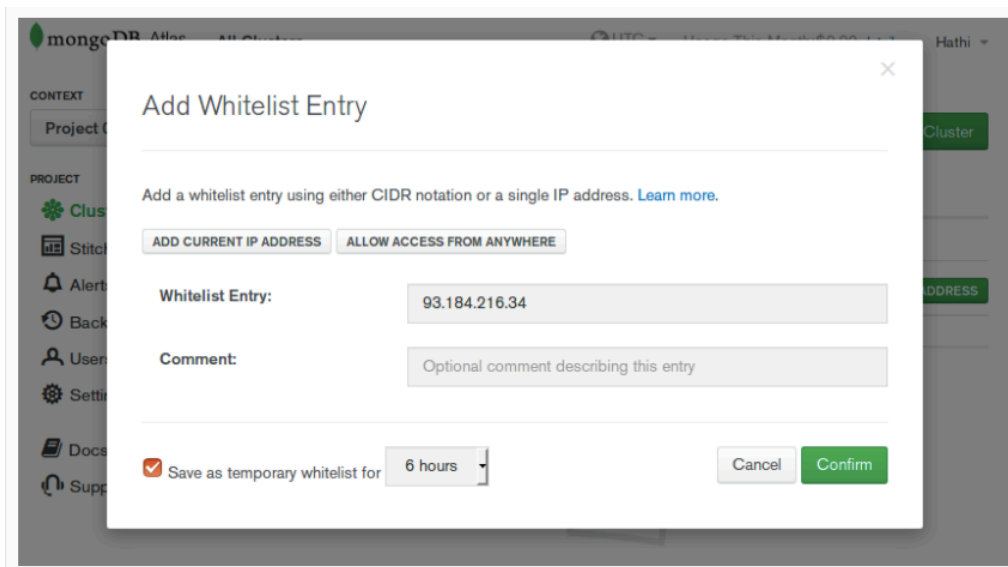


Figura 7: whitelist de mongo

3. Conexión entre Node y Mongo

Para realizar las consultas a la base de datos utilizaremos Mongoose, es paquete de node que permite definir esquemas o modelos con datos fuertemente tipados. Una vez que se define un esquema, Mongoose ofrece la posibilidad de crear un Modelo basado en un esquema específico.

Instalamos con `npm install mongoose`

Creamos una función encargada de hacer la conexión con la base de datos que acabamos de crear, la hacemos asíncrona para que regrese una promesa. Utilizamos `async` y `await` que es una característica propia de node para que funciones asíncronas se comporten como sincronas, a la función le pasamos la cadena de conexión que hemos creado en Mongo atlas.

Cuando hacemos conexiones a base de datos es recomendable utilizar el bloque `try` y `catch`, ya que podría fallar y necesitaremos capturar el error y manejarlo, en este caso si falla la base de datos no podremos hacer nada en nuestra aplicación así que lo único que haremos será

mostrar el error por consola para poder identificarlo y solucionarlo en caso de que algo ocurriera.

```
const dbConnection = async() => {  
  
  try {  
    await mongoose.connect( process.env.DB_CNN , {  
      useNewUrlParser: true,  
      useUnifiedTopology: true,  
      useCreateIndex: true  
    });  
  
    console.log('DB Online');  
  
  } catch (error) {  
    console.log(error);  
    throw new Error('Error a la hora de iniciar la BD ver logs');  
  }  
  
}
```

Figura 8: Conexión entre note y mongo

Esta función necesitamos exportarla para que pueda ser utilizada así que la declararemos como `module.exports`. La importamos en el `index.js` y la invocamos.

4. Configuración CORS

CORS (Cross-origin resource sharing) que traducido al español significa “el intercambio de recursos de origen cruzado”, es un mecanismo a través del cual se puedan solicitar recursos restringidos (como pueden ser por ejemplo, las tipografías o las imágenes) desde un dominio distinto al original. Node tiene disponible una librería para habilitar o deshabilitar CORS con varias opciones configurables.

Instalamos el paquete ejecutando:

```
npm install cors
```

Importamos el modulo en nuestro `index.js` y añadimos la siguiente línea para habilitar las peticiones desde cualquier dirección:

```
App.use(cors())
```

Utilizamos el middleware de express para activar el CORS.

5. Variables de entorno

Queremos configurar variables de entorno para tener centralizadas todas las variables globales, como por ejemplo la cadena de conexión a la base de datos, el puerto o cualquier otra cosa que necesite ser compartida en varios puntos de la aplicación.

También se utilizan para proteger información sensible, ya que estas variables se podrían sobrecribir desde el servidor donde este alojada la aplicación y el programador no tendría acceso a ellas.

Instalamos el paquete `dotenv` (`npm install dotenv`) para poder leer archivos con la extensión `.env`

```
PORT=3000
DB_CNN=mongodb+srv://user:user@cluster0.8lbbq.mongodb.net/test
JWT_SECRET=Holsdj28397kjHd7@asdyui3897k

GOOGLE_ID=200538079333-pr75i7k4obaor8loo6gmakuotdkmv72s.apps.googleusercontent.com
GOOGLE_SECRET=200538079333-pr75i7k4obaor8loo6gmakuotdkmv72s.apps.googleusercontent.com
```

Figura 9: Variables de entorno

Creamos un nuevo archivo llamado `.env` y declaramos las variables:

Para leer estas variables necesitamos importar el `.env` desde cualquier archivo, para acceder a ellas utilizamos `process.env`, este archivo ya existe y es donde node guarda sus variables de entorno, las nuestras se añadirán a todas ellas

6. Modelos de datos

Tendremos 3 tipos de objetos en nuestra base de datos: usuario, proyectos y centro y estos tres tipos de objetos se van a relacionar entre si, ya que por ejemplo, necesitamos saber a que centro pertenece un profesor, o que alumno a regado un proyecto y a que centro pertenece, en la carpeta `modelo`, serán modelos de `mongoose` el cual será el encargado de poner ciertas restricciones a los campos que se van a guardar en la base de datos. El `schema` es la definición de cada uno de los registros que van a estar dentro de una colección, cada elemento se indica el tipo y si es requerido o si tiene que se único

Para implementarlo lo declaramos como `module export`

7. Middleware

Middleware es un concepto muy importante al desarrollar sobre Node.js. Un middleware es una función que se ejecuta antes de que el servidor realice la acción que ha solicitado el cliente en la request.

En nuestro caso vamos a crear un middleware personalizado que compruebe y recopile los posibles errores que se hayan producido en las comprobaciones previas.

Empezaremos creando varias comprobaciones, queremos comprobar que el correo electrónico sea único al registrarlo en la base de datos y que los campos obligatorios vienen en las peticiones al servidor.

```

router.put(('/:id',
  [
    validarJWT,
    check('nombre', 'El nombre es obligatorio').not().isEmpty(),
    check('email', 'El email es obligatorio').isEmail(),
    check('role', 'El role es obligatorio').not().isEmpty(),
    validarCampos,
  ],
  actualizarUsuario
);

```

Figura 10: Middlewares

Para ayudarnos en esta tarea instalaremos el paquete express validator (npm i express-validator), este paquete permite hacer validaciones semiautomáticas en las rutas de nuestros endpoints, de este paquete vamos a utilizar un middleware llamado check, pondremos un check para cada campo que queremos validar, le podemos añadir un mensaje personalizado para que sean más descriptivos, también utilizaremos validationResult (también incluido en express validator), que servirá para añadir a la request todos los errores que sucedieron.

A continuación crearemos un middleware personalizado que podremos utilizar en cualquier endpoint para comprobar si en el request existe algún error de validación de campos antes de realizar cualquier otra acción.

Va a ser muy parecido al resto de controladores pero tendrá un argumento extra (a parte del rest y el send), el argumento next, que es la función a la que llamaremos

si se cumplen todas las condiciones impuestas por el middleware que vamos a crear para que el controlador continúe su ejecución.

```
const salt = bcrypt.genSaltSync();
usuario.password = bcrypt.hashSync( password, salt );
```

Figura 11: Contraseña encriptada

Para utilizar el middleware lo llamamos en la ruta del endpoint antes de realizar cualquier otra operación

8. Seguridad y encriptación

Las contraseñas no deberían guardarse en la base de datos sin encriptar, sería una medida de seguridad básica y no hacerlo implicaría un error grave de seguridad, para encriptarlas vamos a utilizar la librería `bcryptjs`.

La encriptación la haremos en el controlador de usuarios justo antes de guardar en la base de datos, la encriptación será mediante hash de una sola vía. Este tipo de encriptación tiene dos características fundamentales:

1. Es sencillo encriptar el mensaje original con el valor del hash, pero es imposible (en un periodo de tiempo razonable) recrear el mensaje original a través del valor del hash.
2. Es imposible que dos mensajes que se encripten con este método, ya sea con el mismo valor de hash o distinto, generen dos mensajes encriptados idénticos.

Generaremos un número aleatorio para pasarlo como parámetro a `bcrypt` y poder crear el hash para la contraseña:

Otra medida importante de seguridad sería que al obtener los datos del usuario y pasarlo al cliente nunca enviar la contraseña.

Al loguearnos utilizaremos la función `bcrypt.compare`, comparamos la contraseña que ha introducido el usuario y la que hay guardada en la base de datos para ese usuario

```
// Verificar contraseña
const validPassword = bcrypt.compareSync( password, usuarioDB.password );
if ( !validPassword ) {
  return res.status(400).json({
    ok: false,
    msg: 'Contraseña no válida'
  });
}
```

Figura 12: Verificación de la contraseña

A parte de estas medidas de seguridad también generaremos web tokens para mantener de forma pasiva el estado de el usuario en nuestra aplicación.

JWT (JSON Web Token) es un estándar dentro del documento RFC 7519. En él se define uno de los mecanismos más utilizados de autenticación para poder propagar de forma segura, la identidad de un usuario o entidad.

Los datos que queremos compartir están codificados en objetos de tipo JSON, que se pueden añadir dentro del payload o cuerpo el mensaje que va digitalmente firmado.

Los datos que queremos compartir están codificados en objetos.

Un token se compone de 3 partes:

- **Header:** Es el encabezado dónde se puede indicar que algoritmo se utiliza y qué tipo de token se va a enviar. En nuestro caso utilizaremos el algoritmo HS256 y un token tipo JWT.
- **Payload:** Es el cuerpo des mensaje donde pueden aparecer los datos de usuario o cualquier información que queramos añadir.
- **Signature:** una firma, a través de ella podremos verificar si el token es válido

```

const generarJWT = ( uid ) => {
  return new Promise( ( resolve, reject ) => {
    const payload = {
      uid,
    };

    jwt.sign( payload, process.env.JWT_SECRET, {
      expiresIn: '12h'
    }, ( err, token ) => {
      if ( err ) {
        console.log(err);
        reject('No se pudo generar el JWT');
      } else {
        resolve( token );
      }
    });
  });
};
}

```

Figura 13: Generar JWT

Se utilizará autenticación pasiva para no saturar el servidor, esto significa que el token se envía al backend desde el cliente y el servidor verifica si ese token está activo, si ha caducado, etc.

Para desarrollar todo este sistema de verificación empezaremos creando un archivo llamado `jwt.js` la carpeta `helpers`. Vamos a utilizar la librería de `node jsonwebtoken`.

El payload que va a ser visible desde el cliente, por lo tanto, no deberíamos incluir información sensible, en este caso utilizaremos el `uid` de usuario. En el archivo `.env` crearemos una nueva variable de

entorno llamada `JWT_SECRET` que será un string de caracteres inventados que utilizaremos para formar el token.

A esta función que hemos creado en los helpers la llamaremos desde el endpoint de login, la respuesta de este endpoint será el token generado. También la llamaremos al registrar usuario.

Crearemos un nuevo middleware que llamaremos en todos los endpoints que a los que solo se pueda acceder estando loqueado en la aplicación, este nuevo middleware se llamará `validarJWT`, en él leeremos de la cabecera de la request con el fin de comprobar si contiene el parámetro `x-token` (desde el lado del cliente cuando

```
const validarJWT = (req, res, next) => {  
  
  // Leer el Token  
  const token = req.header('x-token');  
  
  if ( !token ) {  
    return res.status(401).json({  
      ok: false,  
      msg: 'No hay token en la petición'  
    });  
  }  
  
  try {  
  
    const { uid } = jwt.verify( token, process.env.JWT_SECRET );  
    req.uid = uid;  
    console.log("valido jwt");  
    console.log(uid)  
  
    next();  
  
  } catch (error) {  
    return res.status(401).json({  
      ok: false,  
      msg: 'Token no válido'  
    });  
  }  
  
}
```

Figura 14: Validar JWT

hagamos las peticiones al servidor deberemos añadir este campo en la cabecera). Utilizamos jwt para verificarlo.

Además de realizar esta verificación, aprovechamos este middleware para incluir el uid en la request, de esta forma las funciones que se ejecuten a continuación dispondrán de esta información.

Google sign in

Para el login de Google vamos a necesitar verificar el token de la misma forma que hemos verificado el token del login normal. Para ello Google tiene sus propias funciones ya definidas.

Como paso previo necesitamos tener una cuenta de google y crear un nuevo proyecto para generar unas credenciales oauth. Cuando las tengamos las guardaremos en las variables de entorno.

Instalaremos la librería google-auth-library y copiamos de la documentación de google la función verify que la utilizaremos en nuestro endpoint de login de google.

Para no cargar el controlador con información innecesaria pondremos esta función en la carpeta helpers.

```
const googleVerify = async( token ) => {  
  
  const ticket = await client.verifyIdToken({  
    idToken: token,  
    audience: process.env.GOOGLE_ID, // Specify the CLIENT_ID of the app that accesses the backend  
    // Or, if multiple clients access the backend:  
    //[CLIENT_ID_1, CLIENT_ID_2, CLIENT_ID_3]  
  });  
  const payload = ticket.getPayload();  
  const { name, email, picture } = payload;  
  
  return { name, email, picture };  
}
```

Figura 15: Verificar el token de Google

```

const { name, email, picture } = await googleVerify( googleToken );
const usuarioDB = await Usuario.findOne({ email });
let usuario;

if ( !usuarioDB ) {
  // si no existe el usuario
  usuario = new Usuario({
    nombre: name,
    email,
    password: '@@',
    img: picture,
    google: true
  });
} else {
  // existe usuario
  usuario = usuarioDB;
  usuario.google = true;
}

```

Figura 16: Guardar contraseña encriptada en la base de datos

En el payload del token de Google estará la información que necesitamos del usuario, el nombre, email e imagen, comprobamos si ya existe un email con ese usuario, y en caso negativo creamos uno nuevo con los datos que hemos obtenido en el payload.

Para saber qué es un usuario loqueado con google podemos añadir un nuevo campo al modelo de usuario.

3. CRUD

CRUD es el acrónimo de “Create, Read, Update and Delete” (Crear, Leer, Actualizar y Borrar). Estas son las cuatro operaciones fundamentales de aplicaciones de base de datos.

Crearemos nuestra API REST par poder realizar todas estas operaciones en la base de datos. Tendremos una serie de rutas que se clasifican según el tipo de objeto con el que operen

Auth

POST /api/login Este endpoint se encarga de foguear al usuario en la aplicación. Recibe como parámetros de entrada el email y la contraseña. Devuelve un token JWT de autenticación con una validez de 12 horas. Responde al cliente con un booleano, el token de autenticación y los datos del usuario.

POST /api/login/google Similar al endpoint anterior, recibe los mismo parámetros y devuelve la misma información, pero para el token de autenticación utiliza las funciones de Google

Búsqueda

GET /api/todo/:termino búsqueda global Se encarga de buscar entre los proyectos y los centros generados en la aplicación según los términos de búsqueda, el parámetro de entrada son los términos de búsqueda y la respuesta un booleano y un json con los resultados

GET /api/todo/coleccion/:tipo/:termino Similar a la búsqueda anterior pero restringiendo solo a un tipo de objeto, pueden ser usuarios, proyectos o centros.

Centros

GET /api/centers Devuelve todos los centros, no tiene parámetros de entrada y devuelve un booleano y un array de centros

POST /api/centers Crea un nuevo centro, recibe como parámetro de entrada el nombre del nuevo centro, devuelve un booleano y un objeto json con la información del centro

PUT /api/centers/:id Edita la información de un centro, recibe como parámetros de entrada el id del centro y la nueva información introducida. Devuelve un booleano y el centro con los datos actualizados

DELETE /api/centers/:id Elimina un centro, recibe como parámetro de entrada el id del centro, devuelve un booleano y un mensaje de éxito

Proyectos

GET /api/project devuelve todos los proyectos. No tiene parámetros de entrada, devuelve un booleano y un array de proyectos

POST /api/project crea un nuevo proyecto, los parámetros de entrada son el título, la descripción y el centro al que pertenece. Devuelve un booleano y el proyecto creado.

PUT /api/project/:id Edita la información de un proyecto, recibe el id del proyecto y los nuevos datos. Devuelve un booleano y el proyecto actualizado

DELETE /api/project/:id Elimina un proyecto, recibe el id y devuelve un booleano y un mensaje de éxito

GET /api/project/byCenter Obtiene todos los proyectos creados por un centro. Recibe como parámetro de entrada el id del centro y devuelve un booleano y un array de proyectos

GET /api/project/favs Obtiene los proyectos favoritos de usuario. No tiene parámetros de entrada y devuelve un booleano y un array de proyectos

GET /api/project/ranking Obtiene todos los proyectos ordenados por puntuación de mayor a menor. No tiene parámetros de entrada y devuelve un booleano y un array de proyectos

Uploads

GET /api/uploads/:tipo/:id Obtiene una imagen. Tiene como parámetro de entrada el tipo de objeto y el id, devuelve un archivo de tipo File.

PUT /api/uploads/:tipo/:id Sube una nueva imagen, los parámetros de entrada son el tipo el id y un archivo de tipo File. Devuelve un booleano, un mensaje de éxito y el nombre del archivo.

Usuarios

GET /api/usuarios Obtiene todos los usuarios registrados en la aplicación, devuelve un booleano y un array de usuarios.

POST /api/usuarios Crea nuevos usuarios, recibe como parámetros de entrada el email y la contraseña. Devuelve un booleano, el token generado y los datos del usuario.

PUT /api/usuarios/:id Modifica los datos de un usuario. Los parámetros de entrada son el id y los nuevos datos. Devuelve un booleano y el usuario actualizado.

DELETE /api/usuarios/:id Elimina un usuario. Los parámetros de entrada son el id de usuario. Devuelve un booleano y un mensaje de éxito.

4. Subida de archivos al servidor

En nuestra aplicación necesitaremos incluir imágenes de los usuarios, proyectos y centros, y almacenarlas en el servidor. Para ello utilizaremos la librería `express-fileupload`. Esta librería es un middleware de `express` que sirve para subir imágenes, cuando termina de procesarlas estarán disponibles en `req.files` para que puedan ser manipuladas por el controlador del endpoint.

Después de instalar este paquete crearemos en la carpeta `uploads` una carpeta por cada tipo de objeto (usuario, proyecto y centro). Ahí es donde se almacenarán las imágenes, estas carpetas tienen que existir previamente porque vamos a indicar en qué ruta tiene que ir cada imagen cuando se suba.

Primero procesaremos la imagen con `express-fileupload`, a continuación extraemos la extensión del archivo y validamos que la extensión este permitida. Vamos a guardar la imagen con un nombre genérico que sea un string de caracteres, para ello utilizaremos la librería `uuid`, que sirve para generar esta cadena que será el nombre del archivo y le añadimos la extensión, le indicamos el path según el tipo de objeto que sea y movemos el archivo a ese directorio.

Una vez terminado este proceso podemos actualizar la base de datos y con el nuevo path de la imagen, para ello creamos una función independiente en la carpeta de helpers con el nombre de `actualizar-imagen`, en la cual buscamos por tipo y por id el elemento al que hemos

```

// Procesar la imagen...
const file = req.files.imagen;

const nombreCortado = file.name.split('.'); // wolverine.1.3.jpg
const extensionArchivo = nombreCortado[ nombreCortado.length - 1 ];

// Validar extension
const extensionesValidas = ['png','jpg','jpeg','gif'];
if ( !extensionesValidas.includes( extensionArchivo ) ) {
  return res.status(400).json({
    ok: false,
    msg: 'No es una extensión permitida'
  });
}

// Generar el nombre del archivo
const nombreArchivo = `${ uuidv4() }.${ extensionArchivo }`;

// Path para guardar la imagen
const path = `./uploads/${ tipo }/${ nombreArchivo }`;

```

Figura 17: Subida de imágenes al servidor

añadido la foto. Es importante comprobar si existe la imagen anteriormente guardada en la base de datos para ese objeto y borrarla para no acumular imágenes innecesarias.

Capítulo III: Desarrollo del frontend

1. Estructura y organización del código

La parte de el cliente esta construida en Angular, por lo que seguirá una estructura típica de este tipo de aplicaciones:

Auth

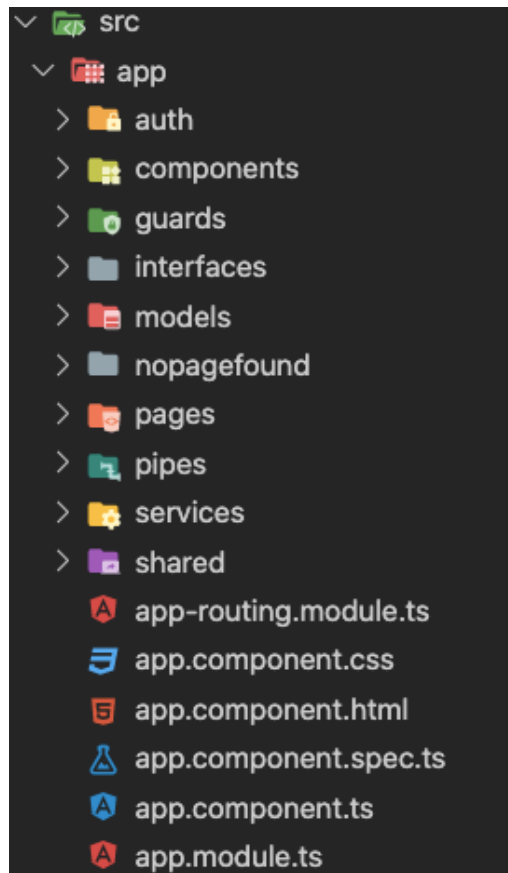


Figura 18: Organización del frontend

En esta carpeta se encuentran los componentes del login y del registro y su respectivo routing

Components

Contiene todos los componentes que podrían reutilizarse desde cualquier punto de la aplicación, en nuestro caso esta el componente de la modal desde donde se suben las imágenes, se utiliza tanto para los usuarios, como para los centros y proyectos

Guards

Contiene el auth guard y el admin guard, se encargan de que el usuario no pueda entrar a ciertas páginas si no está logueado o no tiene permisos.

Interfaces

Contiene login-form y register-form interface, para los formularios reactivos del login y el registro.

Models

Contiene los modelos de datos de usuarios, centros y proyectos.

Nopagefound

Componente que se muestra cuando el usuario ha navegado hasta una url inexistente.

Pages

En esta carpeta están todos los componentes que forman las vistas de la aplicación.

Pipes

Los pipes en Angular nos permiten transformar visualmente la información que se muestra de una variable, en nuestro caso tenemos el image-pipes, que se utiliza para construir la url completa de la imagen para que pueda ser visualizada.

Services

En la carpeta servicios se encuentran todos los servicios que se utilizarán desde cualquier parte de la aplicación.

Shared

Contiene los elementos que estarán presentes en todo la aplicación, como el header y el sidebar

2. Módulo de rutas

En toda aplicación de Angular existe un componente principal que es el llamado “app.component”, este a su vez contendrá todos los componentes creados en la aplicación. El objetivo es definir los objetos que estarán fijos y siempre visibles en la pantalla, como por ejemplo el reader y el sidebar, y tener una parte de la vista que cambie su contenido según el componente vaya a ser mostrado, así se evita recargar la página constantemente y se consigue un efecto de aplicación de escritorio. Todo esto se puede conseguir gracias al módulo de rutas que viene incluido en Angular

```
<!-- Router system -->  
<router-outlet></router-outlet>
```

Figura 19: router outlet

La etiqueta `<router-outlet>` debe aparecer en el `.html` principal de la página. Esta directiva varía su contenido según qué componente este activo. Para indicarle que componente está activo es necesario crear un módulo “app-routing.module” que contiene las rutas que se llamarán para cada componente en particular. Este archivo estará en la raíz de nuestra aplicación, y en él se podrán incluir otros archivos de routing que corresponderán a las rutas hijas de la aplicación. De esta forma el código queda más organizado y en caso de que la aplicación crezca no tendremos una lista interminable de rutas en el archivo principal de routing.

Para hacer uso de esta directiva importamos el módulo “NgModule” y creamos el archivo “app-routing.module.ts” como un módulo más en la aplicación. Además necesitamos los paquetes “RouterModule” y “Routes” importados para poder hacer uso de las rutas correctamente.

```

import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

//Modules
import { PagesRoutingModule } from './pages/pages.routing';
import { AuthRoutingModule } from './auth/auth.routing';

import { NopagefoundComponent } from './nopagefound/nopagefound.component';

const routes: Routes = [
  // path /dashboard PagesRouting
  // path /grafica1 PagesRouting
  // path /progress PagesRouting
  // path /login AuthRouting
  // path /register AuthRouting
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: '**', component: NopagefoundComponent }
];

@NgModule({
  declarations: [],
  imports: [
    RouterModule.forRoot( routes ),
    PagesRoutingModule,
    AuthRoutingModule
  ],
  exports: [ RouterModule ]
})
export class AppRoutingModule { }

```

Figura 19: Creación de rutas

Las rutas se guardarán en un array como el de la imagen, este array contiene objetos con las claves “path” que es la ruta a llamar y “component” que corresponde a el componente se invocará cuando esa ruta este activa. Todos los componentes que aparecen en el array de rutas deben estar previamente importados.

El modulo “app-routing-module” también tiene que estar importado en el módulo principal del proyecto “app.module”.

```

const routes: Routes = [
  {
    path: 'dashboard',
    component: PagesComponent,
    canActivate: [AuthGuard],
    children: [
      { path: '/', component: DashboardComponent, data: {title: "Dashboard"} },
      { path: 'progress', component: ProgressComponent, data: {title: "progress"} },
      { path: 'grafical', component: GraficalComponent, data: {title: "grafica"} },
      { path: 'account-settings', component: AccountSettingsComponent, data: {title: "account settings"} },
      { path: 'search/:termino', component: SearchComponent, data: {title: "Resultados de búsqueda"} },
      { path: 'promises', component: PromisesComponent, data: {title: "promises"} },
      { path: 'rxjs', component: RxjsComponent, data: {title: "rxjs"} },
      { path: 'profile', component: ProfileComponent, data: {title: "Perfil de usuario"} },
      { path: 'users', canActivate: [AdminGuard], component: UsersComponent, data: {title: "Gestión de usuarios"} },
      { path: 'centers', canActivate: [AdminGuard], component: CentersComponent, data: {title: "Gestión de centros"} },
      { path: 'projects', component: ProjectsComponent, data: {title: "Gestión de proyectos"} },
      { path: 'project/:id', component: ProjectComponent, data: {title: "Gestión de proyectos"} },
      { path: 'center/:idCenter', component: CenterPageComponent, data: {title: "Página del centro"} },
      { path: 'center/:idCenter/:idProject', component: ProjectPageComponent, data: {title: "Página del Proyecto"} }
    ]
  }
];

```

Figura 20: Array de rutas

Una vez hecho esto podemos llamar a las rutas de cada componente. Para ello utilizamos la propiedad “routerLink = ‘<ruta>’” en el archivo .html

Cuando navegamos hacia una de estas rutas declaradas, podemos ver que en “<router-outlet>” se muestra la información del componente asociado a esa ruta. Siempre podemos navegar directamente hacia una ruta utilizando “router.navigate” Esto puede ser útil por ejemplo si después de realizar alguna acción queremos devolver al usuario a la página principal. Bastaría con instancias el objeto Router e invocar a ese método dentro de la función que se llama al pulsar el botón “volver”.

3. Formularios reactivos para el login y el registro de usuarios

Gracias al módulo de Angular “FormsModule” podemos controlar los parámetros de nuestros formularios desde el propio componente y automatizar comprobaciones. Como el resto de módulos que hemos

```

public registerForm = this.fb.group({
  nombre: ['', Validators.required],
  email: ['', [Validators.required, Validators.email]],
  password: ['', Validators.required],
  password2: ['', Validators.required],
  terminos: [false, Validators.required],
},{
  validators: this.equalPasswords('password', 'password2')
} )
constructor(private fb: FormBuilder, private usersService: UsersService, private router: Router) { }

```

Figura 21: RegisterForm

utilizado, tenemos que declararlo en el módulo de entrada “app.module”.

Al utilizar este componente tenemos muchas más etiquetas disponibles de las que vienen por defecto en Angular. Con la directiva “[NgModel]” se relaciona el valor de las variables clavadas en el componente.

```

equalPasswords(pass1Name: string, pass2Name: string) {
  //needs return a function
  return (formGroup: FormGroup) =>{
    const pass1Control = formGroup.get(pass1Name)
    const pass2Control = formGroup.get(pass2Name)
    if(pass1Control.value === pass2Control.value) {
      pass2Control.setErrors(null);
    }else {
      pass2Control.setErrors({notEqual: true})
    }
  }
}

```

Figura 22: Validación de contraseña

En nuestra aplicación utilizaremos este tipo de formularios para el login y el registro de usuarios

Para integrarlos en nuestra aplicación incluiremos tanto en el componente del registro como el del login la importación de “FormBuilder, FormGroup, FormControl y Validators”, podemos incluirlos por separados o directamente “@angular/forms”

Para iniciar un formulario reactivo creamos una variable de tipo “FomGroup”, para utilizar tenemos que inyectarla en el controlador como tipo “FormBuilder”.

Una vez que hemos inicializado el formulario tenemos que asociarlo. Lo hacemos mediante la directiva “[FormGroup]” en el elemento html del formulario.

Para acceder al valor de los datos introducidos en el formulario desde el controlador podemos hacerlo utilizando el get del FormGroup. “<formulario>.get(<control>).value”.

Con este tipo de formulario podemos utilizar validadores, con ellos es posible comprobar si se cumplen o no ciertas reglas. En nuestro caso los utilizaremos para asegurarnos de que los campos requeridos se han rellenado y que el email este en el formato adecuado.

4. Login de Google

Para implementar el login de Google primero añadimos el botón en nuestro formulario:

```
<div class="row">
  <div class="col-xs-12 col-sm-12 col-md-12 m-t-10 mb-4 text-center" id="googleButton">
    <div id="my-signin2"></div>
  </div>
</div>
```

Figura 22: Botón de login de Google

Renderizamos el botón con los estilos de Google desde la función ngOnInit de nuestro componente, también añadimos la función que queremos que se llame cuando pulsamos el botón, en nuestro caso será la función startApp:

También tenemos que llamar a la api de google para iniciar la librería de Google Platform. Esta función la añadimos en el servicio de usuarios y la invocamos desde el constructor, tenemos que añadirle nuestro client_id:

```

renderButton() {
  gapi.signin2.render('my-signin2', {
    'scope': 'profile email',
    'width': 240,
    'height': 50,
    'longtitle': true,
    'theme': 'dark'
  });
  this.startApp();
}

```

Figura 23: Función de renderizado

```

googleInit() {
  gapi.load('auth2', () => {
    // Retrieve the singleton for the GoogleAuth library and set up the client.
    this.auth2 = gapi.auth2.init({
      client_id: '200538079333-pr75i7k4obaor8loo6gmakuotdkmv72s.apps.googleusercontent.com',
      cookiepolicy: 'single_host_origin',
    });
  });
}

```

Figura 24: Función GoogleInit

La función startApp iniciará la pasarela de login en Google, y si se ha realizado correctamente llamará a la función attachSignin:

```

startApp = function() {
  gapi.load('auth2', () => {
    // Retrieve the singleton for the GoogleAuth library and set up the client.
    this.auth2 = gapi.auth2.init({
      client_id: '200538079333-pr75i7k4obaor8loo6gmakuotdkmv72s.apps.googleusercontent.com',
      cookiepolicy: 'single_host_origin',
    });
    this.attachSignin(document.getElementById('my-signin2'))
  });
};

```

Figura 25: Función startApp

La función `attachSignin` comprueba si la pasarela de google ha devuelto un usuario válido, en ese caso llama a la función `loginGoogle` que tenemos definida en nuestro servicio de usuarios.

Y finalmente la función `loginGoogle` de nuestro servicio llama al endpoint de google de nuestro servidor para devolvernos el token:

```
attachSignin(element) {
  this.auth2.attachClickHandler(element, {},
    (googleUser) => {
      const id_token = googleUser.getAuthResponse().id_token;
      this.userService.loginGoogle(id_token).subscribe(resp => {
        this.ngZone.run(() => { // para que angular no pierda el ciclo de vida
          this.router.navigateByUrl('')
        })
      })
    }
  );
}, function(error) {
  alert(JSON.stringify(error, undefined, 2));
});
}
```

Figura 26: Función `attachSignin`

```
loginGoogle(token) {
  return this.http.post(`${base_url}/login/google`, {token})
    .pipe(
      tap((response: any) => {
        localStorage.setItem('token', response.token)
        localStorage.setItem('menu', JSON.stringify(response.menu))
      })
    )
}
```

Figura 27: Función `loginGoogle`

5. Acceso desde Angular al servicio REST

Las aplicaciones en la parte del front necesitan comunicarse con el servidor utilizando el protocolo HTTP, para obtener, actualizar datos o

```

createUser(formData: RegisterForm) {
  return this.http.post(`${base_url}/usuarios`, formData)
  .pipe(
    tap((response: any) => {
      localStorage.setItem('token', response.token)
      localStorage.setItem('menu', JSON.stringify(response.menu))
    })
  )
}

```

Figura 28: función createUser

cualquier otro tipo de servicio que ofrezca el backend. Angular ofrece un cliente HTTP simplificado.

Este objeto se encuentra en `@angular/common/http`, para utilizarlo inyectaremos esta dependencia. Declaramos en el “app.module” e importamos el módulo “HttpModule”.

Tras importarlo declaramos una instancia del objeto, las llamadas que se pueden ejecutar desde este objeto son:

```

http.get(<url>)
http.post(<url>, <datos>)
http.put(<url>, <datos>)
http.delete(<url>)

```

Todas estas peticiones deben ser capturadas por el método “subscribe”, que devuelve dos parámetros, “response” y “error”

6. Servicios

El protagonista de las aplicaciones en Angular son los componentes, pero a la larga el componente puede albergar demasiada lógica y ser muy complejo, extenso y difícil de modificar, por ello se considera buena práctica incluir las peticiones HTTP dentro de los servicios.

En resumen, un servicio es un conjunto de funcionalidades que pueden ser consultadas desde cualquier componente. Incluye desde realizar las peticiones REST hasta funciones y variables comunes, o lógica que se quiere ocultar del componente.

Desde Angular-CLI podemos crear servicios fácilmente: “ng generate service <nombre>”. Cuando creamos un servicio desde Angular-CLI no se declaran automáticamente en el módulo principal “app.module”, por lo que tendremos que hacerlo posteriormente a mano.

Para explicar el funcionamiento de un servicios cogeremos de ejemplo el servicio que hemos creado para gestionar usuarios “users.service.ts”. Este servicio se ocupa de realizar todas las peticiones REST que tienen que ver con los usuarios.

Como se muestra en la imagen, la función dentro de nuestro servicio de usuarios encargada de crear nuevos usuarios tiene como parámetro de entrada un objeto de tipo formData, esta función devuelve la respuesta de la petición post, pero antes de devolverla pasa por un pipe. Un pipe es una funcionalidad de rxjs que captura la respuesta obtenida, y nos permite operar con ella y devolverla, es un paso intermedio. En el caso de nuestra función guardaremos en el localStorage del navegador el token y los elementos del menú que verá ese usuario, no devolveremos nada porque no necesitamos ninguna información de esta función, solo saber si ha dado error o no.

Una vez completado el servicio, hay que inyectarlo en el componente donde necesitemos utilizar las funcionalidades que contiene el servicio e invocar a los métodos deseados.

En este caso después de llamar a createUser de nuestro servicio, redirigimos al usuario a la página principal, pues significa que se ha registrado correctamente y puede acceder a la aplicación en sí. En caso de que el servidor nos haya devuelto un error a la hora de crear el

nuevo usuario, se abrirá una ventana modal con el mensaje de error producido.

```
this.userService.createUser(this.registerForm.value).subscribe(response => {
  this.router.navigateByUrl('')
}, (err)=>
{
  Swal.fire('Error', err.error.msg, 'error')
});
```

Figura 29: Llamada a la función createUser

7. Componentes

Como hemos dicho anteriormente, los componentes son el elemento más importante de las aplicaciones de Angular, las aplicaciones se desarrollan en base a un árbol de componentes.

El componente “centros” creado en nuestra aplicación tiene la siguiente forma:

```
@Component({
  selector: 'app-centers',
  templateUrl: './centers.component.html',
  styles: [
  ]
})
```

Figura 30: Creación de un componente

En la imagen se puede ver la etiqueta @component que sirve para indicar que ese archivo es un componente. Primero importamos el módulo “Component” @angular/components

```

constructor(private centerService: CenterService,
  private modalImagenService: ModalImgService,
  private searchsService: SearchsService) {
}

ngOnDestroy(): void {
  this.imgSubs.unsubscribe();
}

ngOnInit(): void {
  this.loadCenters();

  this.imgSubs = this.modalImagenService.newImg
    .pipe(delay(100))
    .subscribe( img => this.loadCenters() );
}

```

Figura 31: Controlador de un componente

Un componente lo forman los siguientes elementos:

Selector: Es el nombre que tendrá la etiqueta o directiva HTML que identifica al componente

Plantilla: Es un archivo de tipo html asociado

Estilos: Es un archivo css asociado a la vista del componente

Dentro del componente declaramos una clase, “CentersComponent” en nuestro caso, en la que incluimos la lógica.

Gracias a Angular-CLI es muy sencillo crear un nuevo componente ejecutando el comando “ng generase component <nombre>”. Así se crean todos los elementos antes descriptor, además se declaran en el módulo “AppModule” de forma automática.

En el constructor de nuestro componente “centers” hemos importado varios servicios que serán utilizados.

En la función ngOnInit se llama a las funciones que se quieren ejecutar cuando el componente se carga, en nuestro caso tenemos “loadCenters” que traerá la lista completa de centros y “imgSubs” que es una subscripción al observable de “newImg” que sirve para que

```

<tr *ngFor="let center of centers">
  <td class="text-center">
    <img [src]="center.img | image:'centers'"
        [alt]="center.name"
        class="w100 cursor"
        (click)="openModal(center)">
  </td>

```

Figura 33: Ejemplo de uso ngFor

cada vez que se cargue una nueva imagen se actualice la lista de

```

loadCenters() {
  this.loading = true;
  this.centerService.loadCenters()
    .subscribe( centers => {
    this.loading = false;
    this.centers= centers;
  })
}

```

Figura 32: Función loadCenters

centros.

El método `ngOnDestroy` se ejecuta cuando se cambia de vista y el componente se destruye. En el caso de nuestro componente, queremos dejar de observar si se ha recibido una nueva imagen.

Un ejemplo de función en el controlador de nuestro componente sería "loadCenters" que llama a su vez al servicio de centros para obtener el listado completo.

La información de los centros la mostramos en el HTML utilizando la directiva `*ngFor`.

```

canActivate(
  next: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): boolean {

  if (this.usersService.user.role === 'ADMIN_ROLE') {
    return true;
  } else {
    this.router.navigateByUrl('/dashboard');
    return false;
  }
}

```

Figura 34: Ejemplo de Guard

8. Guards

Angular viene con una serie de funcionalidades ya construidas que sirven para manejar la autenticación, una de ellas son los guards. Los guarda son interfaces que avisan al objeto router de si el usuario tiene accesible la ruta a la que va a navegar o no. De forma que solo se pueda acceder a ellas si se tiene determinados permisos.

Hay 4 tipo de Guards:

CanActivate: Decide si el usuario puede acceder a una determinada ruta.

CanActivateChild: Decide si el usuario puede entrar en rutas hijas de una página.

CanDeactivate: Comprueba si el usuario que va a abandonar una página puede hacerlo o no, un ejemplo es cuando el usuario tiene cambios sin guardar y se le muestra una modal de confirmación

CanLoad: Sirve para evitar que se carguen determinados módulos si el usuario no tiene permisos.

```

{ path: 'users', canActivate: [AdminGuard], component: UsersComponent, data:

```

Figura 35: Ejemplo de uso de un Guard

```

import { Pipe, PipeTransform } from '@angular/core';
import { environment } from 'src/environments/environment';
const base_url = environment.base_url;

@Pipe({
  name: 'image'
})
export class ImagePipe implements PipeTransform {

  transform(img: string , tipo = "usuarios| centers | projects"): string {
    if(img) {
      if(img.includes('https')) {
        return img;
      }
      return `${base_url}/upload/${tipo}/${img}`;
    } else {
      return `${base_url}/upload/usuarios/no-image`;
    }
  }
}

```

Figura 37: Pipe de imagenes

En nuestro caso solo necesitamos implementar dos guards de tipo "canActivate":

```

let name = 'sheldon';
{{ name | uppercase }}
Result --> SHELDON

```

Figura 36: Caso de uso de un pipe

Los guards se incluyen en los elementos de el objeto que contiene las rutas como un parámetro mas, puede devolver un booleano o una promesa dependiendo del tipo de comprobación que se haga. En la imagen se ve un ejemplo de uso para evitar que los usuarios que no sean administradores no puedan acceder al área de gestión de usuarios

9. Pipes

Los pipes se utilizan para transformar strings, numeros, fechas y otro tipo de datos en la forma en la que se muestran. Son funciones simples que se pueden implementas como expresiones dentro de una plantilla.

Angular tiene pipes por defecto para las transformaciones más comunes. Un ejemplo de pipe por defecto de Angular sería el de uppercase, que transforma el texto en mayúsculas:

Nuestro pipe personalizado recibirá como parámetros de entrada un string con la url de la imagen y el tipo de objeto. Comprueba si la imagen contiene https, si lo contiene significa que la url de la imagen viene de un usuario de google y no hay que hacerle ninguna transformación, en ese caso devolvemos la url tal cual llegó. En caso contrario concatenaremos al string de la imagen los campos necesarios para reconstruir la ruta.

```
<img [src]="center.img | image:'centers'"  
  [alt]="center.name"  
  class="w100 cursor"  
  (click)="openModal(center)">
```

Figura 38: uso del pipe de imágenes

Tenemos que importar en “app.module.ts” nuestro pipe e incluirlo cada vez que queramos incrustar en la vista una imagen.

10. Gestión de roles

Para gestionar los roles en la aplicación tenemos que tener en cuenta tanto las comprobaciones en el servidor como en la parte del cliente.

```

favorite(project:Project) {
  const heartElement = ".heart" + project._id;
  $(".heart" + project._id).toggleClass('is_animating');
  $(".heart" + project._id).toggleClass('heart-fav');
  if($(".heart" + project._id).hasClass('heart-fav')) {
    if(!this.usersService.user.favs) {
      this.usersService.user.favs = [];
    }
    this.usersService.user.favs.push(project._id);
    this.addScore(project);
  }
  else {
    this.usersService.user.favs.splice(this.usersService.user.favs.indexOf(project._id),1)
    this.quitScore(project);
  }
  this.usersService.saveUser(this.usersService.user)
  .subscribe();
}

```

Figura 39: Funcionalidad agregar a favoritos un proyecto

En el servidor gracias a los middlewares podemos comprobar el rol de un usuario antes de hacer el resto de operaciones, y devolver un error si el rol no es el esperado

En la parte del cliente tenemos varias herramientas, por un lado tenemos los guardas que protegen las rutas, el adminGuard lo utilizamos en las rutas que solo están disponibles para moderadores. Además, las opciones del menú que se muestran al usuario las recibimos del servidor, y tiene en cuenta que tipo de usuario es. Y por último tenemos las directivas *ngIf propias de Angular que las utilizamos para ocultar o mostrar elementos según el rol del usuario.

Con todas estas herramientas en conjunto hacemos que la aplicación quede totalmente adaptada al rol del usuario logueado.

11. Sistema de puntuación y favoritos

El sistema de puntuación y de favorito van de la mano ya que un proyecto recibe o pierde puntos cuando un usuario lo añade o lo quita de sus favoritos.

Para añadir un proyecto a favoritos tenemos la siguiente función:

Por un lado tenemos la lógica que controla las animaciones del corazón utilizando jQuery. Comprobamos si el corazón tiene activa la clase que hace que aparezca con color rojo, y en el caso de que así sea llamamos a la función `addScore` y añadimos el proyecto al array de proyectos favoritos del usuario, en caso negativo llamamos a `quitScore` y eliminamos el proyecto del array de favoritos. Finalmente guardamos los cambios en el usuario.

En las funciones de `addScore` y `quitScore` comprobamos el rol de usuario para saber si tenemos que añadir o quitar un punto o dos. Actualizamos la puntuación sumando o restando ese valor y actualizamos la información del proyecto en la base de datos.

```
addScore(project: Project){
  const value = this.userService.user.role === "USER_ROLE" ? 1 : 2;
  if(!project.score){
    project.score = 0;
  }
  project.score = +project.score + value;
  this.projectService.updateProject(project)
    .subscribe();
}

quitScore(project: Project){
  const value = this.userService.user.role === "USER_ROLE" ? 1 : 2;
  if(!project.score){
    project.score = 0;
  }
  project.score = +project.score - value;
  this.projectService.updateProject(project)
    .subscribe();
}
```

Figura 40: Funcionalidad de puntuación

Capítulo IV: Optimización, tests y despliegues

1. Lazy load

Lazy Load cargar de manera dinámica las paginas, se utiliza cuando tenemos muchos modulos y queremos que esos modulos se carguen

bajo demanda, la aplicación ya es eficiente de por sí porque está muy modularizada (mencionar los módulos que hay) pero el módulo más grande es pages y pages routing también contiene mucha información, por eso las rutas hijas de pages las vamos a cargar de manera perezosa con lazy load. Para ello se ha creado un nuevo módulo en pages (child routes module) donde están todas las rutas hijas y en pages routes si van cargando de forma dinámica comprobando antes que el usuario tiene acceso a la página que se va a cargar con los guards

2. Pruebas unitarias

Las pruebas automáticas sirven para probar funcionalidades en específico, normalmente se hacen conforme se van añadiendo nuevas funcionalidades a la aplicación.

Las ventajas de implementar estas pruebas es poder probar el código hecho por otros programadores, detectar errores antes de que sucedan en producción y en definitiva probar cualquier variante que pudiera hacer fallar nuestro código, ayuda a tener un código más limpio.

Las desventajas es que esto no garantiza que no haya errores, escribir las pruebas para toda la aplicación puede ser más largo que desarrollar la aplicación en sí, si estás trabajando solo no son tan útiles.

Así que tan importante es saber hacer pruebas automáticas correctamente como elegir en qué queremos invertir tiempo en probar

En este proyecto se han añadido algunos test unitarios con fines didácticos.

3. Generar build de distribución

Para desplegar la parte del backend simplemente tendremos que añadir un script nuevo que lance la aplicación sin utilizar nodemon, ya que si utilizamos nodemon en un entorno que no sea de desarrollo daría problemas a la hora de volver a levantar el servicio si se cae por algún motivo

```
"scripts": {  
  "start:dev": "nodemon index.js",  
  "start": "node index.js"  
},
```

Figura 41: Scripts de inicio del servidor

El resto simplemente es seguir los pasos que indique el host al que vayamos a subir nuestro servidor. En el apartado siguiente explicamos como lo desplegamos en Netlify.

Para desplegar la parte del cliente tenemos que generar un build de producción. Primero añadimos en las variables de entorno de producción la nueva url, que nos vendrá dada después de desplegar el servidor.

```
export const environment = {  
  production: true,  
  base_url: "http://mission-earth.com/api"  
};
```

Figura 42: Url base en producción

Para generar el build escribimos en la línea de comandos “ng build —prod”. Esto creará una nueva carpeta llamada “dist”, en ella se encuentra nuestra aplicación en versión de distribución.

El último paso sería copiar el contenido de esta carpeta dentro del servidor en la carpeta “public”.

4. Desplegar la aplicación en Netlify

Netlify es una página web en la cual podemos desplegar nuestro proyecto sin necesitar un hosting. Se trata de una plataforma que ofrece un plan gratuito para proyectos básicos o personales, pudiendo mejorar la cuenta en cualquier momento a los planes más avanzados, siendo estos de pago, ofreciendo un mayor ancho de banda para las subidas de proyectos o soporte para nuestro proyecto toda la semana durante todo el día.

En este sitio tenemos las opciones de desplegarlo desde la propia web utilizando una conexión con GitHub o bien podemos trabajar con el propio Netlify CLI. En este caso, se hará uso de la cuenta de GitHub en la que está alojado el proyecto.

Lo primero que podemos ver es un sitio bastante intuitivo, en el que nada más hacer login nos muestra la siguiente información:

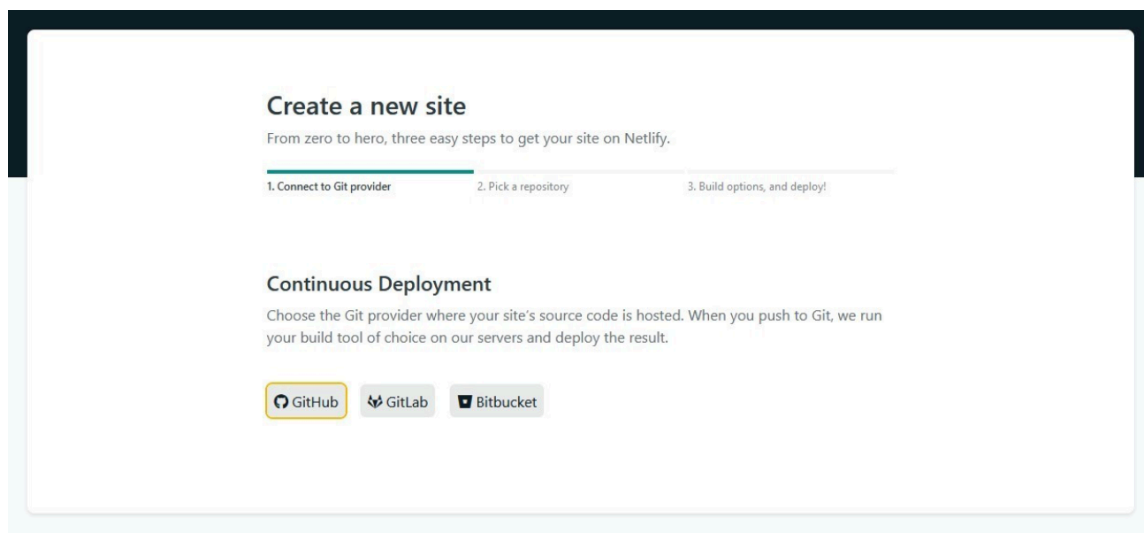


Figura 43: Subir archivos a Netlify

Simplemente elegimos desde que página de Git queremos desplegar el proyecto (en este caso, GitHub).

Tras esperar unos minutos, el sitio web ha sido desplegado y podremos probar que todo funciona.

Es importante cambiar el nombre al proyecto, ya que por defecto Netlify asigna un nombre aleatorio. Además, se puede elegir un dominio propio para mostrar nuestro proyecto en una web propia o bien podemos mostrarlo en la propia plataforma de Netlify.

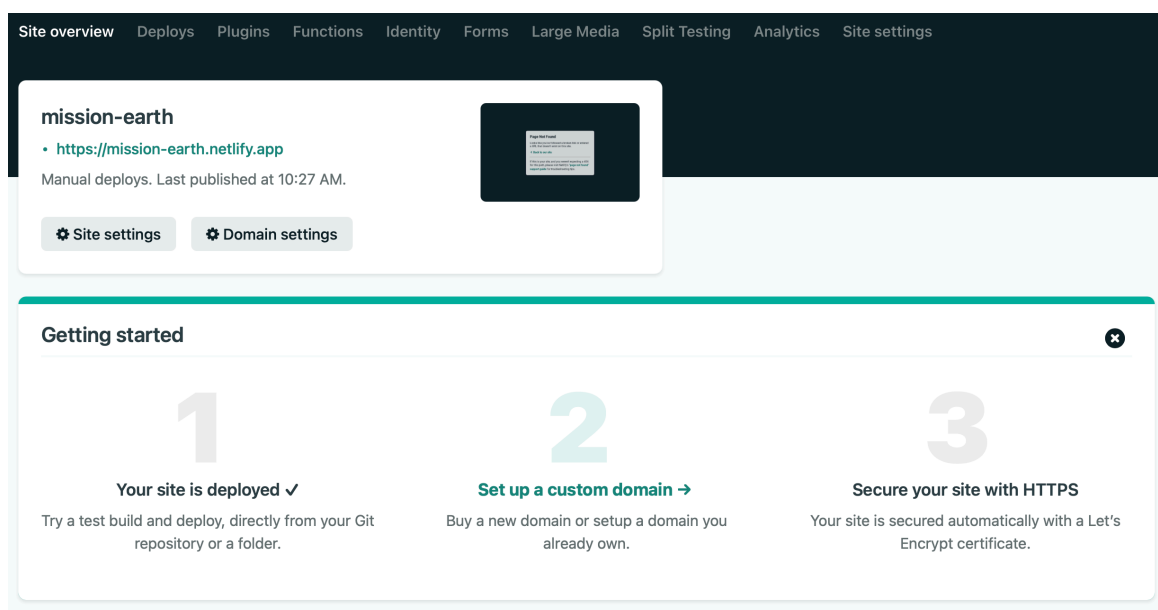


Figura 44: Configuración Netlify

Capítulo V: Conclusiones y líneas futuras

1. Posibles mejoras

La aplicación es completamente funcional pero siempre hay cosas que se pueden añadir y mejorar.

Una desventaja de la aplicación en su estado actual es que solo es accesible desde el navegador, es decir, no está disponible como aplicación para dispositivos móviles. Acceder a la aplicación desde el navegador en el móvil puede ser algo que incomode a los usuarios. Esto podría solucionarse utilizando Ionic.

Ionic está desarrollado con Angular y nació con el propósito de crear aplicaciones móviles híbridas bajo el framework de Angular.

Otra posible mejora sería almacenar las imágenes en AWS o en cualquier otro servicio de almacenamiento en la nube en vez de el servidor, ya que si el número de usuarios crece el servidor podría llegar a saturarse.

Por último, la aplicación podría seguir creciendo con nuevas funcionalidades, como por ejemplo un chat donde pudieran participar todas las personas del mismo centro, un foro, un sistema de comentarios en cada proyecto, etc.

2. Conclusiones

Tras terminar el desarrollo de esta aplicación se han podido comprobar las ventajas de utilizar este stack de tecnologías como herramienta de trabajo para un desarrollador de aplicaciones.

Una de las principales ventajas ha sido la forma fácil de incluir nuevas funcionalidades conforme la aplicación iba creciendo, ha quedado demostrado que si se utilizan correctamente estas tecnologías la escalabilidad está garantizada.

El gestor de paquetes npm es otro de los puntos fuertes de este entorno, ya que es muy sencillo añadir nuevas librerías. En este proyecto hemos incluido muchas de ellas y han facilitado bastante el desarrollo en la parte del backend.

En conclusión se ha podido comprobar que el stack MEAN es una opción muy a tener en cuenta a la hora de iniciarse en el mundo del desarrollo de aplicaciones.

Togo el código se encuentra disponible en Github: <https://github.com/NataliaPmd/mission-earth>

Capítulo VI: Bibliografía y documentación técnica

Todos los recursos en línea se han consultado a inicio de diciembre 2020

National Snow an Ice Data [en línea] Disponible en [National Snow and Ice Data CenterWorld Glacier Monitoring Service](#)

NASA [en línea] Disponible en <https://climate.nasa.gov/evidence/>

WWF [en línea] Disponible en https://www.wwf.es/nuestro_trabajo/clima_y_energia/cumbres_del_clima/

Red Hat [en línea] Disponible en <https://www.redhat.com/es/topics/api/what-is-a-rest-api>

MEAN Stack [en línea] Disponible en <https://platzi.com/blog/que-es-mean-full-stack-javascript/>

Angular-route [en línea] Disponible en https://medium.com/@ryanchenkie_40935/angular-authentication-using-route-guards-bf7a4ca13ae3

Enciclopedia online Wikipedia [en línea] Disponible en <https://es.wikipedia.org/wiki/CRUD>

Angular [en línea] Disponible en: <https://angular.io/>

Angular pipes [en línea] Disponible en: <https://www.acontracorrientech.com/pipes-en-angular-guia-completa/#:~:text=Los%20pipes%20son%20una%20herramienta,s%C3%B3lo%2>

[0lo%20hace%20su%20aspecto.](#)

Seguridad en Angular [en línea] <https://codingpotions.com/angular-seguridad>

Node [en línea] Disponible en: <https://nodejs.org/es/>

MongoDB [en línea] Disponible en: <https://www.mongodb.com/>

Instalación Mongo Atlas [en línea] Disponible en <https://code.tutsplus.com/es/tutorials/create-a-database-cluster-in-the-cloud-with-mongodb-atlas--cms-31840>

Express [en línea] Disponible en: <https://expressjs.com/es/>

Express-validator [en línea] <https://express-validator.github.io/docs/>

Bcrypt [en línea] Disponible en <https://www.npmjs.com/package/bcrypt>

CORS [en línea] Disponible en <https://www.npmjs.com/package/cors>

JWT [en línea] Disponible en <https://jwt.io/>

Express file-upload [en línea] Disponible en <https://www.npmjs.com/package/express-fileupload>

Npm [en línea] Disponible en : <https://www.npmjs.com/>

Sweet alert 2 [en línea] Disponible en: <https://sweetalert2.github.io/>

Middlewares en node [en línea] Disponible en: <https://devcode.la/tutoriales/middlewares-en-nodejs/>

Bootstrap [en línea] Disponible en: <https://getbootstrap.com/>

GitFlow [en línea] Disponible en: <https://cleventy.com/que-es-git-flow-y-como-funciona/#:~:text=Es%20ah%C3%AD%20donde%20entra%20en,a%20los%20lanzamientos%20del%20proyecto.>

Git [en línea]. Disponible en: <https://git-scm.com/>

Stack Overflow [en línea]. Disponible en: <https://stackoverflow.com/>

Introducción a Mongoose [en línea] Disponible en <https://code.tutsplus.com/es/articles/an-introduction-to-mongoose-for-mongodb-and-nodejs--cms-29527>

Json web tokens [en línea] Disponible en <https://openwebinars.net/blog/que-es-json-web-token-y-como-funciona/>