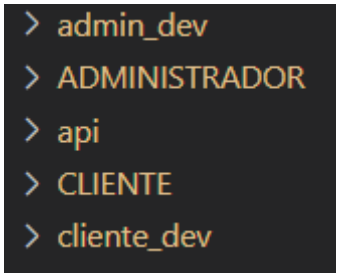
Descripción del proyecto

Proyecto Integrado CSI2.
Alumno: Jesus Diaz Muñoz.
Nombre del proyecto: Cometa Jobs.
Centro: C.D.P Altair.
Ciclo Superior de Desarrollo de aplicaciones web.

Índice de contenido

Estructura del proyecto

El proyecto se divide en 2 partes bien diferenciadas, la API, y el front-end, el cual se subdivide en el **cliente_dev** y **admin_dev.**



admin_dev -> Entorno de desarrollo para el panel de admin.

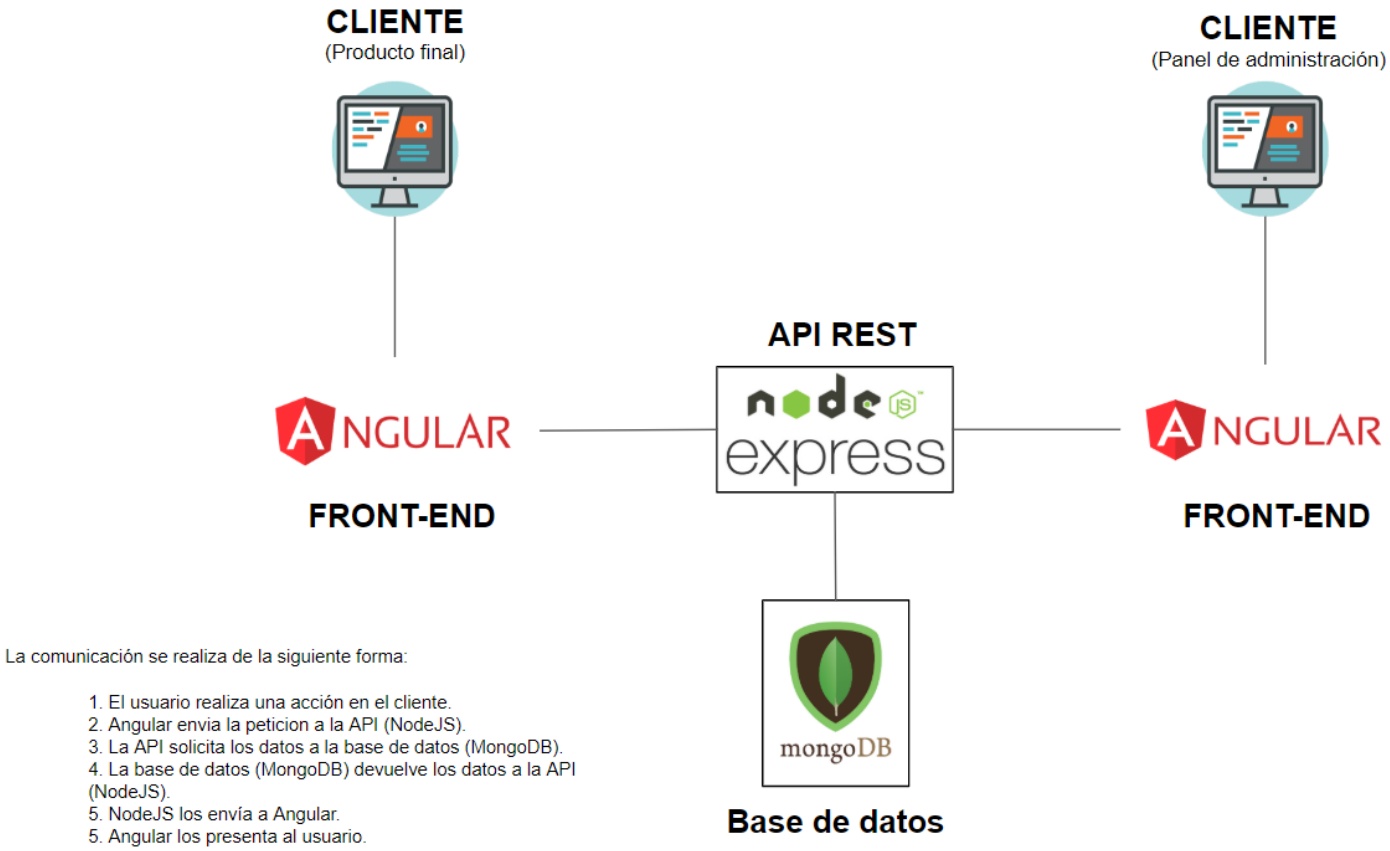
ADMINISTRADOR -> Contiene la compilacion de “admin_dev”.

cliente_dev -> Entorno de desarrollo para el cliente.

CLIENTE -> Contiene la compilacion de “cliente_dev”.

api -> contiene todo el backend.

Esquema de funcionamiento.

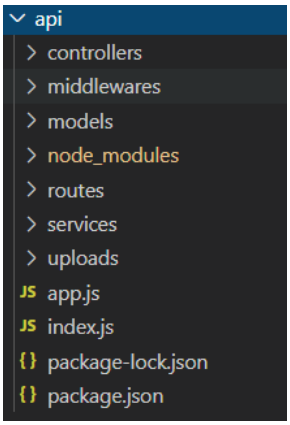


La comunicación se realiza de la siguiente forma:

1. El usuario realiza una acción en el cliente.
2. Angular envia la peticion a la API (NodeJS).
3. La API solicita los datos a la base de datos (MongoDB).
4. La base de datos (MongoDB) devuelve los datos a la API (NodeJS).
5. NodeJS los envía a Angular.
5. Angular los presenta al usuario.

Como podemos observar, nuestros dos front-end, se comunican con el backend (que es la API), y la API, establece una comunicación con la base de datos (MongoDB), que devuelve respuestas JSON a la API, y esta de nuevo a los front-ends, que se encargan de montar todos los datos.

API.



Modelos

No ponemos id ya que JSON los pone automáticamente.

usuario.js

```
var UsuarioSchema = Schema({
    nombre: String,
    login: String,
    password: String,
    movil: String,
    email: String,
    biografia: String,
    acceso: String,
    imagen: String,
    created_at: String
})
```

oferta.js

```
var OfertaSchema = Schema ({
    titulo: String,
    descripcion: String,
    experiencia: String,
    sueldo: String,
    ubicacion: String,
    jornada: {type: Schema.ObjectId, ref: 'Jornada'},
    created_at: String,
    empresa: {type: Schema.ObjectId, ref: 'Usuario'}/
});
```

ubicacion.js

```
var UbicacionSchema = Schema({
    ubicacion: String,
})
```

jornada.js

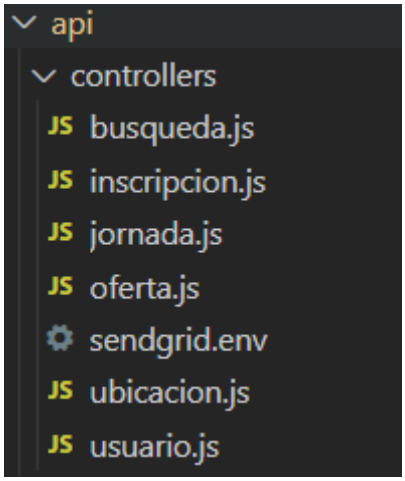
```
var JornadaSchema = Schema ({
    jornada: String,
    color: String

});
```

inscripcion.js

```
var InscripcionSchema = Schema({
    usuario: {type: Schema.ObjectId, ref: 'Usuario'},
    oferta: {type: Schema.ObjectId, ref: 'Oferta'}
})
```

Controladores

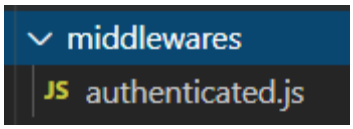
Por cada modelo, tenemos un controlador.



Estos controladores contienen todos los métodos necesarios para interactuar con sus respectivos modelos, los cuales están importados en su correspondiente controlador.

El controlador **búsqueda.js** se basa en el modelo **“oferta”**, pues este controlador solo se encarga de realizar las peticiones de búsqueda de ofertas en la base de datos.

Middlewares



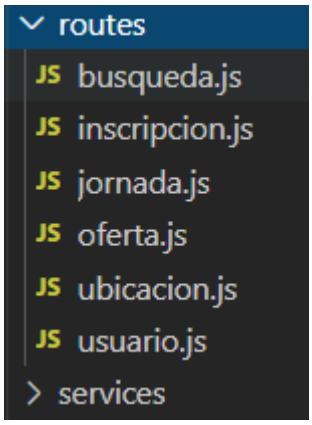
Un breve resumen de lo que es un middleware: es una “función” que se ejecuta entre dos softwares

authenthicated.js se encarga de comprobar si las peticiones que llegan a la API tienen o no el token necesario. Si contiene el token, permite el acceso, y si no, lo deniega.

```
'use strict'

var jwt = require('jwt-simple');
var moment = require('moment');
var secret = 'clave_secreta';

exports.ensureAuth = function(req, res, next){
    if(!req.headers.authorization){
        return res.status(403).send({message: 'La peticion no tiene la cabecera de autenticacion'});
    }

    var token = req.headers.authorization.replace(/['"]+/g, ''); //Quitamos las comillas el token


    try{
        var payload = jwt.decode(token, secret); //Decodificamos el token

        //Verificamos si esta activo o esta expirado
        if(payload.exp <= moment().unix()){
            return res.status(401).send({message: 'El token ha expirado'});
        }

    }catch(ex){
        return res.status(404).send({message: 'El token no es valido'});
    }

    req.user = payload;

    next();
}
```

Rutas

De nuevo, por cada modelo/controlador, tenemos un archivo de rutas.



Estos archivos se encargan de establecer desde que URL se ejecuta cada función del controlador en cuestión. Veamos un ejemplo:

Ruta de: inscripción.js

```
'use strict'

var express = require('express');
var InscripcionController = require('../controllers/inscripcion');
var api = express.Router();
var md_auth = require('../middlewares/authenticated');

api.post('/inscripcion', md_auth.ensureAuth, InscripcionController.saveInscripcion); //Crear i
api.get('/inscripciones', md_auth.ensureAuth, InscripcionController.getInscripciones); //Obten
api.get('/inscripcion/:id', md_auth.ensureAuth, InscripcionController.getInscripcion); //Obten
api.get('/inscritos/:id', md_auth.ensureAuth, InscripcionController.getInscritos); //Obtener l
api.get('/misInscripciones/', md_auth.ensureAuth, InscripcionController.getMisInscripciones);
api.delete('/inscripcion/:id', md_auth.ensureAuth, InscripcionController.deleteInscripcion); /


module.exports = api;
```

Como podemos observar, necesitamos llamar al controlador, a la API, y en este caso nuestro middleware, ya que nuestras funciones (en este caso) requieren comprobación de token

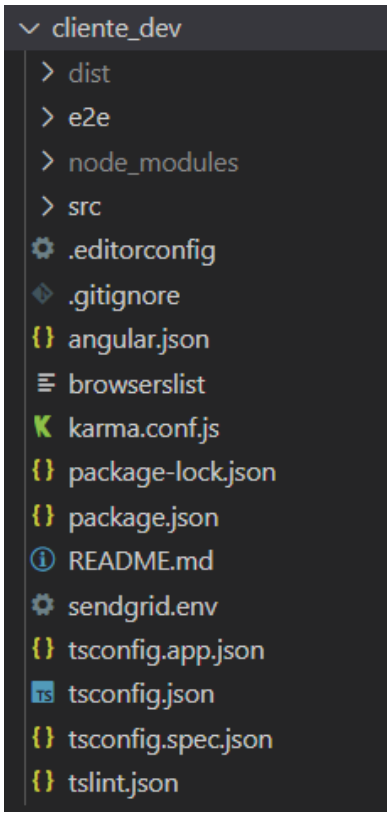
Para establecer las URL, escribimos la URL, seguido del middleware, y, por último, la función del controlador que va a ejecutarse cuando entremos a esa URL.

CLIENTE/ADMIN

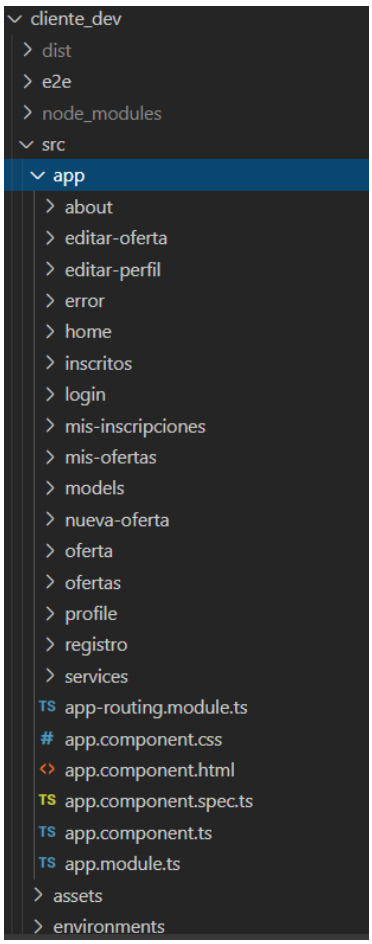
Nota: Son dos proyectos separados, manteniendo la lógica y los servicios, lo único que cambia es el diseño de la interfaz, por tanto, solo se detallara el **cliente_dev**, tomando por tanto el proyecto **admin_dev** la misma descripción.

El motivo de separar el panel de administración de la página de usuario final es la mejor escalabilidad de ambos. Esto se solucionaría metiendo ambos proyectos en uno y aplicando modularización, pero en nuestro caso lo hemos hecho de la forma anteriormente explicada.

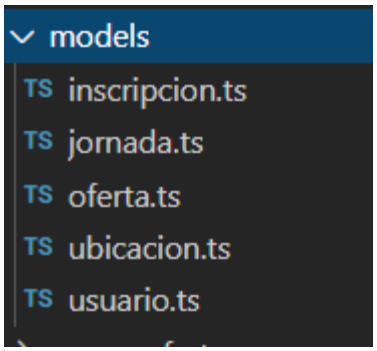
Proyecto de Angular



SRC con los componentes



Modelos



Los modelos tendrán que ser iguales que en la API, pero esta vez sí tendremos que crear un campo ID.

usuarios.ts

```
export class Usuario {

    constructor(
        public _id: string,
        public nombre: string,
        public login: string,
        public password: string,
        public movil: string,
        public email: string,
        public biografia: string,
        public acceso: string,
        public imagen: string,
        public created_at: string,
        public gettoken: string
    ){}
}
```

oferta.ts

```
export class Oferta {

    constructor(
        public _id: string,
        public titulo: string,
        public descripcion: string,
        public experiencia: string,
        public sueldo: string,
        public ubicacion: string,
        public jornada: string,
        public created_at: string,
        public empresa: string
    ){}
}
```

ubicacion.ts

```
export class Ubicacion {

    constructor(
        public _id: string,
        public ubicacion: string

    ){}
}
```

jornada.ts

```
export class Jornada {

    constructor(
        public _id: string,
        public jornada: string,
        public color: string
    ){}
}
```

inscripcion.ts

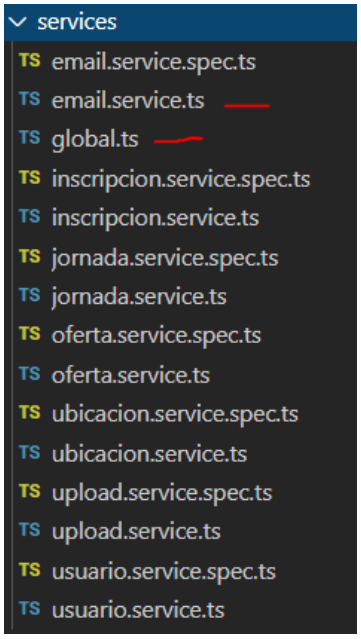
```
export class Inscripcion {

    constructor(
        public _id: string,
        public usuario: string,
        public oferta: string
    ){}
}
```

Servicios

Los servicios, al igual que los controladores en la API, tendremos uno por cada modelo, exceptuando algunos que agregaremos, pero estos no se basan en ningún modelo.

Los servicios tienen la misma función que los controladores. Interactuar con el modelo de dato.

Estos son nuestros servicios. Como hemos dicho, contienen las funciones necesarias para interactuar con el modelo y ofrecer a cada componente las respuestas y que este monte los datos



Vamos a ver en detalle los marcados con una línea roja, dado que estos, no se basan en ningún modelo.

(email.service.ts, global.ts)

email.service.ts

Este servicio se encarga de suministrarnos una función para poder enviar correos electrónicos.

```typescript
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders} from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { GLOBAL } from './global';

@Injectable({
  providedIn: 'root'
})
export class EmailService {

  public url;

  constructor(private _http: HttpClient) {
    this.url = GLOBAL.url;
  }

  sendMail(mail): Observable<any> {
    var params = JSON.stringify(mail); //Pasamos el objeto que nos llega a JSOn
    let headers = new HttpHeaders().set('Content-Type', 'application/json')

    return this._http.post(this.url + 'sendMail', params, { headers: headers });
  }

}
```

global.ts

Este servicio, es muy pequeño, pero de gran importancia, dado que es el que establece la dirección de la API. Todos los servicios anteriores, tienen que llamar a las URL de la API, y sin este archivo, no podrán comunicarse con ella.

```typescript
export var GLOBAL = {
    url: 'https://api.cometajobs.com/api/' //URL de la API (debe de estar corriendo);
}
```

Base de datos

Todos los datos generados y servidos en el proyecto, se almacenan en MongoDB.

MongoDB es una base de datos no relacional. Las estructuras de datos se guardan en formato JSON, por tanto, las respuestas son prácticamente inmediatas.

Las colecciones de la base de datos son generadas automáticamente al usar un modelo de nuestra aplicación. Veamos:

```
'use strict'

//Basandonos en libreria mongoose
var mongoose = require('mongoose');
//Obtenemos el esquema
var Schema = mongoose.Schema;

var JornadaSchema = Schema ({
    jornada: String,
    color: String

});

// Lo exportamos, dandole un nombre al modelo y pasandole el esquema creado
module.exports = mongoose.model('Jornada', JornadaSchema);
```

Arriba tenemos un modelo escrito sobre para MongoDB.

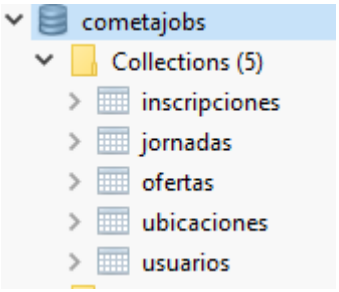
Al nosotros definir el esquema, y luego usarlo en ***mongoose.model('Jornada', JornadaSchema),*** le decimos que cree la colección 'Jornada', con el esquema que le pasamos en el segundo parámetro. El nombre de la colección debemos escribirlo en singular, pues MongoDB se encargará de ponerlo en plural.

¿Que es 'mongoose'?

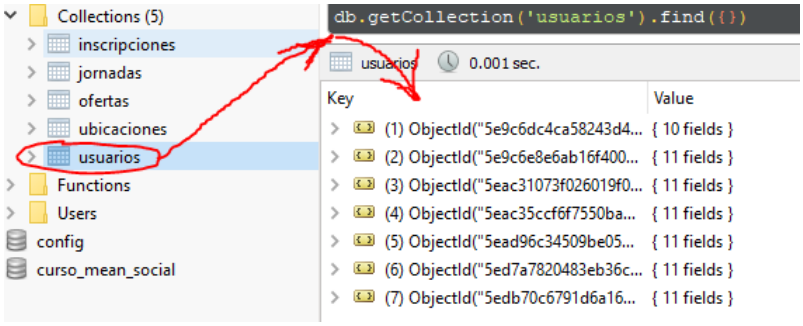
Es una librería para JavaScript que nos permite trabajar con los esquemas de MongoDB.

Veamos cómo queda la base de datos una vez hemos introducido documentos (registros) de todos los modelos de datos.

Colecciones



Documentos de una colección



Documento

Como podemos observar, en el esquema del modelo de dato no teniamos el ID puesto, pero MongoDB lo crea automaticamente.

Si queremos obtener todos los datos de este documento bastaria con acceder a su id, y posteriormente tendriamos los campos guardados en las propiedades. Todo esto viene en una respuesta JSON que devuelve el servidor.

¿Por qué usar MongoDB o una base de datos NoSQL?

- Suelen ser bases de datos mucho más abiertas y flexibles.

-Permiten adaptarse a necesidades de proyectos mucho más fácilmente.

-Ocupan pocos recursos del servidor.

-Las repuestas son más rápidas.

Desventajas

-Baja estandarización.

-Poca documentación.