

# *Módulo del proyecto DAW*

**Gestor**

David Jiménez González

# Índice

Objetivos	Pág 4
Recursos hardware y software	Pág 6
Servidor	Pág 6
Transpilado	Pág 6
Gestión de dependencias	Pág 7
Gestor de tareas	Pág 7
Editor de código	Pág 8
MongoDb	Pág 9
Node Js	Pág 10
Fases	Pág 11
Preparación del entorno de desarrollo	Pág 11
Instalación de dependencias	Pág 11
Servidor	Pág 12
Configuración general	Pág 12
Controladores	Pág 12
Servicios	Pág 12
Cliente	Pág 13
Modulos	Pág 13
Modelos	Pág 13
Servicios	Pág 13
Directivas	Pág 14
Componentes	Pág 14
Desarrollo de las fases	Pág 15
Preparación del entorno de desarrollo	Pág 15
Instalación de dependencias	Pág 17
Servidor	Pág 24
Configuración general	Pág 24
Controladores	Pág 25
Servicios	Pág 26
Cliente	Pág 27
Modulos	Pág 29

Modelos	Pág 30
Servicios	Pág 32
Directivas	Pág 34
Componentes	Pág 34
Visualización de la aplicación	Pág 37
Bibliografía	Pág 41
Preguntas relacionadas con el módulo de FCT	Pág 42

# Objetivos

Desarrollar una plataforma para la gestión de información. El sistema será capaz de almacenar, clasificar y gestionar dicha información. Para que esto sea posible, la información necesitará algún tipo de estructura.

## Biblioteca

La primera separación lógica de información será la biblioteca. Cada usuario será dueño de una librería. Esto no es mas que una referencia a un documento en la base de datos.

## Libros

Cada librería contendrá una serie de libros. Cada libro será una estructura de datos definida por el usuario. Para ello cada libro contendrá una serie de campos.

## Campos

Los campos definen el esquema básico del libro. El usuario podrá decidir que campos contiene cada libro. Para que la información tenga una estructura y se pueda gestionar los posibles campos estarán limitados a unos tipos definidos previamente.

Los tipos básicos de los campos serán: Number, String, Date, Boolean, Array, File y Libro.

Los tipos básicos solo podrán contener sus correspondientes valores. En cambio los tipos file podrán almacenar cualquier tipo de fichero, véase audio, imagen, video...

Los campos de tipo libro serán una referencia hacia otro libro creado previamente o a un campo específico de este.

## Páginas

Para contener los datos de cada libro, estos contaran con paginas. Cada una de estas guarda un registro definido por los campos.

# Recursos hardware y software

El proyecto contará con 3 servidores. Uno albergará la base de datos, otro será el encargado de gestionar un servicio rest y por último tenemos un servidor que se encargará de servir toda la aplicación cliente.

Este sistema estará montado en un entorno mac, pero puede correr en cualquier sistema operativo y con un hardware mínimo. Dado que el tráfico esperado va a ser mínimo el deploy final se hará en un servidor compartido en la plataforma Heroku. Elegimos esta plataforma dado la sencillez que tiene desplegar una aplicación de estas características en su sistema.

## Servidor

Dado que el servidor tendrá una arquitectura REST y toda la aplicación será realizada en javascript, el servidor será Node.js. Para la gestión de las rutas usaremos la librería Express.js.

Una serie de librerías serán añadidas para la gestión de algunas partes del servidor. Para el manejo de sesiones utilizaremos ExpressSession, una librería que añade esto al modulo de express que ya poseemos.

Para validar las peticiones de usuario usaremos tokens basados en Auth0. La librería encargada de gestionar dichos token será JsonWebToken. Cors, BodyParser y Multer se encargarán de parsear las peticiones que lleguen al servidor a una estructura json para su futuro procesamiento.

## Transpilado

Dado que los navegadores actuales no soportan es6 en adelante, el uso de un transpilador resulta esencial. Para este propósito utilizaremos Babelify, una librería basada en Babel, que se integra con Browserify, y nos gestiona todas las

dependencias en el código dando como resultado un solo archivo final en es5, minificado y preparado para correr en cualquier navegador.

## Gestión De Dependencias

Para este punto usaremos Browserify, una librería que se encarga de gestionar los import, require.. en la aplicación cliente.

## Gestor De Tareas

Este punto será gestionado por gulp. Gracias a NPM podemos correrlo en el servidor y programar una serie de tareas automatizadas. Por una parte tendremos un comando que se encargara de llamar a Browserify, y gracias a Babelify, compilar el código en tiempo de ejecución para hacer mas fácil el desarrollo. Programaremos también otro tipo de tares tales como poblar la db, pasar de un ambiente a otro (producción a desarrollo y viceversa), así como el lanzamiento del servidor.

## Cliente

El cliente será una Single Page App. Para la gestión de las rutas usaremos una librería llamada Page. Esta librería estructura las rutas de la misma forma que express, pero en el cliente. También provee una serie de funcionalidades, tales como midelwares, re direcciones, window state...

Toda la app estará diseñada con el patrón flux. Para integrarlo utilizaremos Redux, una librería que nos provee de un state global para toda la aplicación, facilitando así la gestión de toda la información.

La vista estará definida por componentes. Para ello usaremos ReactJs, una librería para crearlos y gestionarlos.

Para la integración del state global con los componentes usaremos ReacRedux, una librería que se encarga de comunicar el state global con el state de cada componente, convirtiendo estos en reactivos.

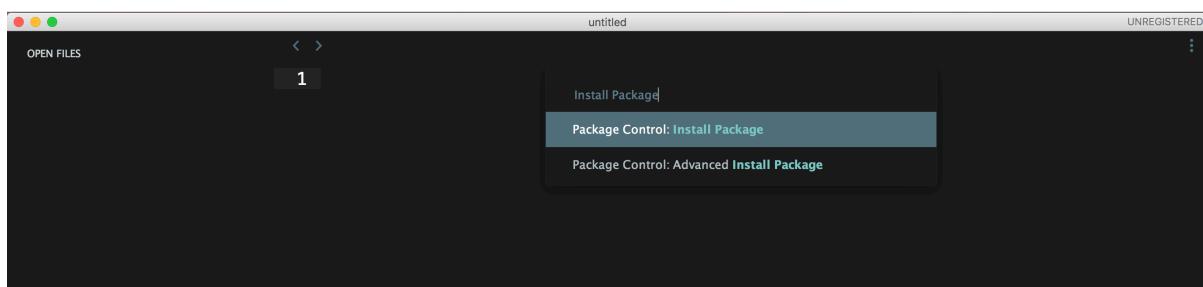
Para facilitar el manejo del state global contamos con ImmutableJs, una librería para crear estructura de datos inmutables haciendo así que cada cambio en el state se refleje en la vista haciendo que este vuelva a renderizarse.

Para dar estilos a la aplicación usaremos Materialize, una librería basada en Material Design propuesto por google. Esta librería nos provee tanto un sistema de rejilla parecido al de Bootstrap, así como una serie de funcionalidad tales como tarjetas, modal, buttons, inputs... definidos previamente.

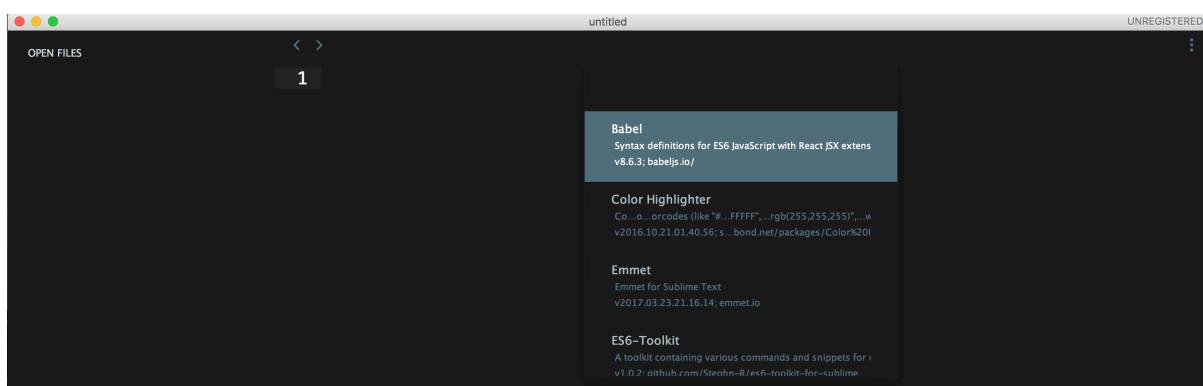
## Editor De Código

Para desarrollar este proyecto usaremos SublimeText 3. Se puede desarrollar con cualquier otro editor de código que soporte Javascript, dado que será el único lenguaje que usaremos. Bastará con descargarlo desde su página web e instalarlo en nuestro sistema.

Para mayor comodidad instalaremos una serie de plugins. Para instalar un plugin en Sublime lo iniciaremos y pulsaremos Ctrl + Mayús + P en windows y linux o Cmd + Mayús + P en mac. Accederemos a un menú e instaremos Install Package.



Elegimos la primera opción y escribimos el nombre del paquete a instalar. En este caso instalaremos Babel, un editor de código para Jsx. Explicaremos mas adelante que es este seudo lenguaje cuando desarrollemos la parte del cliente.



Bastara con dar un intro para instalarlo. Seguidamente instalaremos estos paquetes:

Emmet: Nos proporciona mejor manejo para html y atajos para creación de etiquetas.

ES6-toolKit: Nos proporciona atajos para manejo del estándar de javascript ecmaScript 6. Se hablará mas adelante de este estándar.

JSX: Otra herramienta que amplia las utilidades del paquete anteriormente instalado Babel.

Less: Herramienta para compilación de archivos less. Less es un seudo lenguaje que compila a css. Nos ofrece mayores funcionalidad con menos código.

Materializze: Es un paquete que nos permite atajos para esta librería. Se hablará mas adelante en que consiste y que funciona tendrá.

Color: Resalta los colores en el código. Si escribimos por ejemplo rgba(0,0,0,.5) nos resaltara este texto de esta manera:

```
rgba(0,0,0,.5)
```

## Mongodb

Lo primero que necesitaremos será la base de datos. MongoDb cuenta con un instalador para cualquier sistema operativo descargable desde su página web. Bastará con descargar y ejecutar el instalador. Dado que este tipo de servicios cuentan con una seguridad elevada, en entornos mac y linux tendremos que arrancarlo con permisos de administrador.

La instalación se realiza de forma global. El único requisito que presenta la instalación es crear una carpeta en la raíz del sistema llamada data, que será donde se guarde toda la información de la db.

Mongo utiliza un sistema de almacenamiento basado en bson, un lenguaje parecido a json pero binarizado. Esto nos permite llevarnos toda la base de datos solo con hacer una copia del archivo storage.bson, que se encuentra en la carpeta data/db.

Para iniciar este servicio bastará con abrir una consola y ejecutar el comando MONGOD con permisos de administrador. Con esto ya tenemos el servicio corriendo en el puerto 27017. Por defecto usaremos este puerto, pero es modificable si tenemos algún otro servicio que este haciendo uso de el.

Al iniciar el servicio deberíamos ver algo parecido a esto en la consola. En ella se muestra el puerto, el path donde se guardarán los datos (dbpath=/data/db), información del sistema y por último un mensaje como este:

```
I NETWORK [thread1] waiting for connections on port 27017
```

Este mensaje nos informa que el servicio esta iniciado correctamente y esta preparado para trabajar.

## Nodejs

Los servidores estarán bajo la plataforma NodeJs. Este servicio corre en cualquier tipo de sistema y necesita muy pocos recursos. Es por ello que es ideal para nuestra aplicación, dado que también es programable en javascript, el lenguaje con el que estará desarrollada toda la aplicación.

Lo primero que tenemos que hacer es instalar este servicio de forma global. Note cuenta también con un instalador para cualquier sistema operativo descargable desde su pagina web. Bastará con ejecutar el instalador y seguir los pasos.

# Fases

## Preparación Del Entorno De Desarrollo

Para codificar este proyecto vamos a usar SublimeText3. Este programa nos permite trabajar con typescript de forma nativa, avisando de los errores y facilitando la escritura de código.

Para la base de datos usaremos MongoDb, una base de datos no relacional, muy integrada en el entorno javascript.

Aparte de esto solo necesitaremos la terminal de Mac para manejar las acciones referentes al proyecto, tales como transpirar, copiar ficheros, iniciar el proyecto... Todo esto lo haremos gracias a npm, el gestor de paquetes de node. Esta librería nos permite también definir una serie de scripts para correrlos directamente desde la consola.

## Instalación De Dependencias

La primera dependencia que vamos a necesitar es TypeScript. Este seudolenguaje nos permite typar Javascript. Ademas nos permite trabajar con imports de forma nativa y validación de objetos, al igual que cualquier otro lenguaje basado en poo.

Para el servidor vamos a necesitar Nodejs, una librería escrita en c que nos permite hacer servidores muy ligeros con solo unas pocas líneas de código. Para hacer mas ágil el desarrollo añadiremos express, una librería para el manejo de rutas, peticiones http, sesiones... Todas estas funcionalidad están de forma nativa en node, pero express nos provee una interfaz mas amigable para ponerlas en marcha.

La comunicación con la base de datos será gestionada por Moongoose, una librería para node que nos facilita la creación de schemas, modelos y llamadas a la base de datos. La funcionalidad de esta librería consiste en crear un schema que define un modelo, y mediante este nos comunicaremos con la base de datos, siendo esto similar al patrón dao.

Para desarrollar el cliente usaremos Angular IO, la segunda versión de angular. Esta librería nos facilita mucho la vida ya que nos provee de prácticamente todo lo necesario para el desarrollo del cliente. Angular IO, a diferencia de la versión anterior esta basado en componentes. Un componente no es mas que un paquete lógico que mezcla html, js y css. Estos componentes son totalmente reutilizables. Ademas de la modularización, gracias a los componentes, angular nos provee de un router, directivas predefinidas y un sistema de ajax para la gestión de las peticiones al servidor.

Por último necesitaremos bootstrap tanto para los estilos css generales de la página, como para añadir ciertas funcionalidad como cards, panels, modals...

## Servidor

### Configuración General

Configuraremos el servidor y definiremos la conexión a la base de datos, la ruta del servidor y la clave secreta para los tokens.

### Controladores

Los controladores definen las acciones acciones y las asocian a las rutas. Esto en node no es mas que definir una ruta y asignarle una función a ejecutar. Estas funciones reciben 3 parámetros, req, res y next. La primera define la petición, la segunda define la respuesta y por ultimo next llama a la siguiente función dado que una ruta puede tener mas de una función asociada.

## Servicios

Los servicios no son mas que un conjunto de funciones que interactúan con la base de datos. Los utilizaremos para separar los controladores de los modelos.

## Cliente

El cliente esta desarrollado completamente con Angular. Angular se estructura en modelos que a su vez contienen componentes, servicios y directivas.

Por otra parte definiremos modelos en typescript para proveer de una estructura a la información que maneja la aplicación.

## Modulos

Un contenedor que nos permite modular la aplicación y hacerla escalable y sostenible. En el se definen las sus dependencias y como se renderizará.

## Modelos

Los modelos no son mas que clases escritas en typescript que definen la estructura de la información. También definen métodos para poder interactuar con ella.

## Servicios

Los servicios son los encargados de la comunicación con el servidor. En ellos se definen las peticiones http y serán invocados para sincronizar la información con la base de datos.

## Directivas

Son componentes auxiliares que se pueden invocar desde cualquier parte de la aplicación y proveen de una funcionalidad específica a esta.

## Componentes

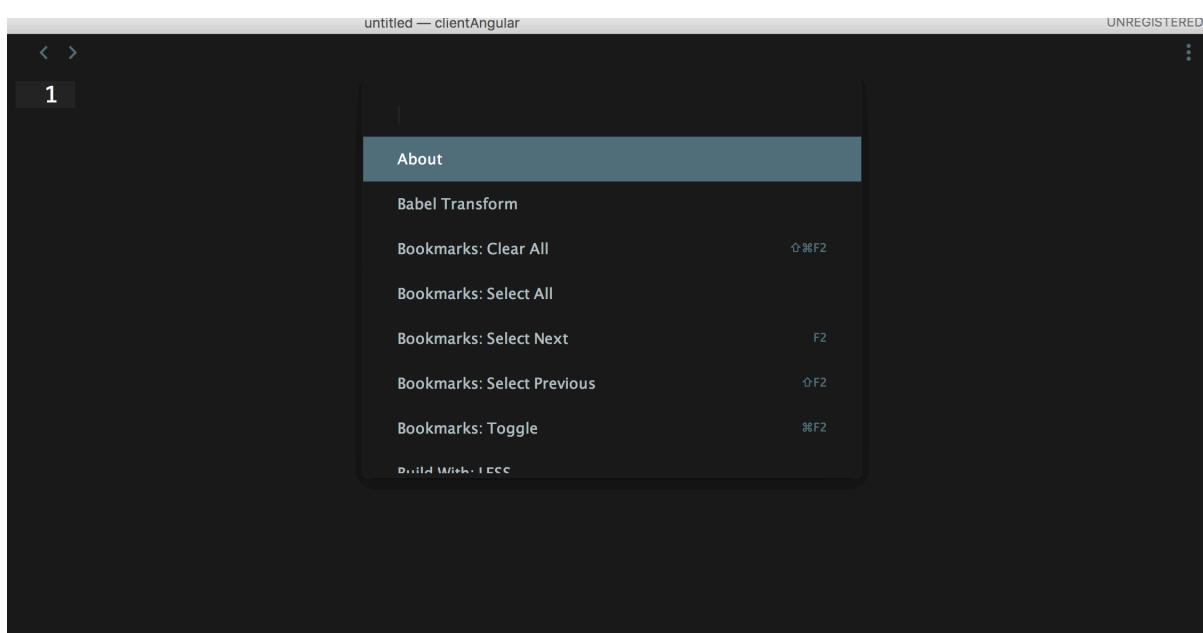
La base sobre la que se renderizan las vistas, se interactua con el usuario... Los componentes proveen la funcionalidad a la aplicación, muestran la información y dan estilos a la vista.

# Desarrollo de las fases

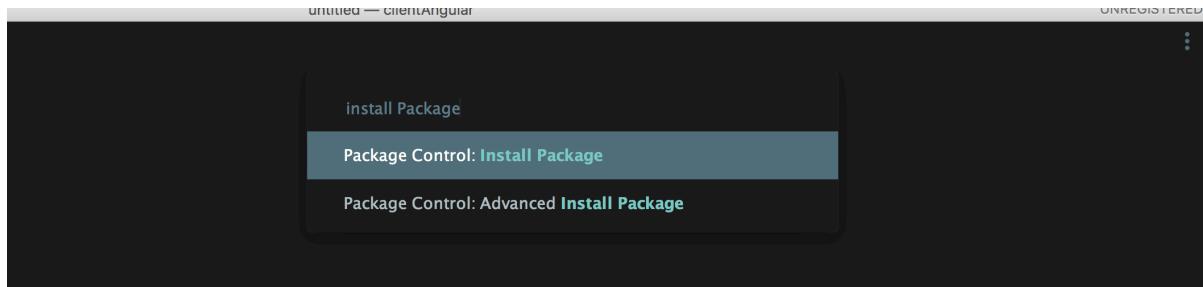
## Preparación Del Entorno De Desarrollo

Lo primero que vamos hacer es descargar SublimeText de su página oficial. Aunque estemos desarrollando el proyecto en entorno mac, sublime corre en cualquier sistema operativo. Bastará con descargar el instalador desde su web e instalarlo. Para facilitar el desarrollo vamos a instalar una serie de paquetes. El primero será TypeScript. Este paquete nos ayudara a visualizar errores y nos provee de atajos de teclado para la escritura de código. Otro paquete interesante será Html5, que nos provee atajos para la escritura de estructuras html5 predefinidas. Por ultimo instalaremos less, un paquete para el manejo de estilos css.

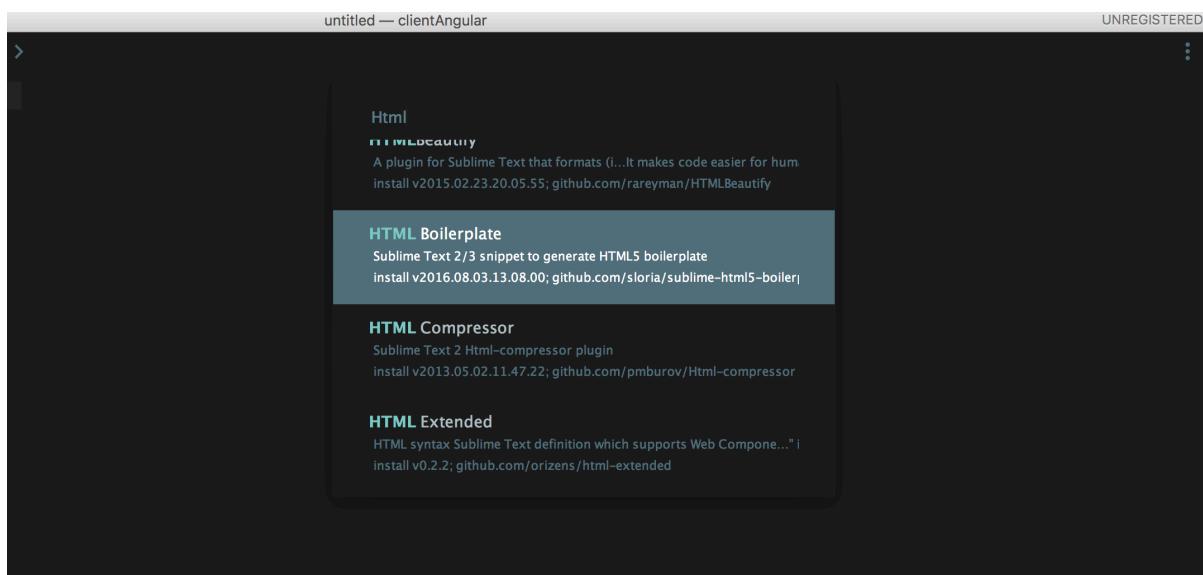
Estos paquetes no son esenciales para el desarrollo pero nos ayudaran bastante. Para instalar cualquier paquete en Sublime bastará con abrir el programa y presionar cmd + mayas + p.



Este comando abrirá una consola donde escribiremos install pakages y pulsaremos intro.



En la consola escribiremos el paquete que queremos descargar y con pulsar intro lo instalaremos.



# Instalación De Dependencias

Lo primero que vamos a descargar e instalar es NodeJs. Podemos descargarlo desde su web oficial. Descargaremos un instalador y bastará con instalarlo. El instalador de node nos instala también su gestor de paquetes NPM.



Node.js® es un entorno de ejecución para JavaScript construido con el [motor de JavaScript V8 de Chrome](#). Node.js usa un modelo de operaciones E/S sin bloqueo y orientado a eventos, que lo hace liviano y eficiente. El ecosistema de paquetes de Node.js, [npm](#), es el ecosistema mas grande de librerías de código abierto en el mundo.

[Descargar para macOS \(x64\)](#)

**v6.11.0 LTS**

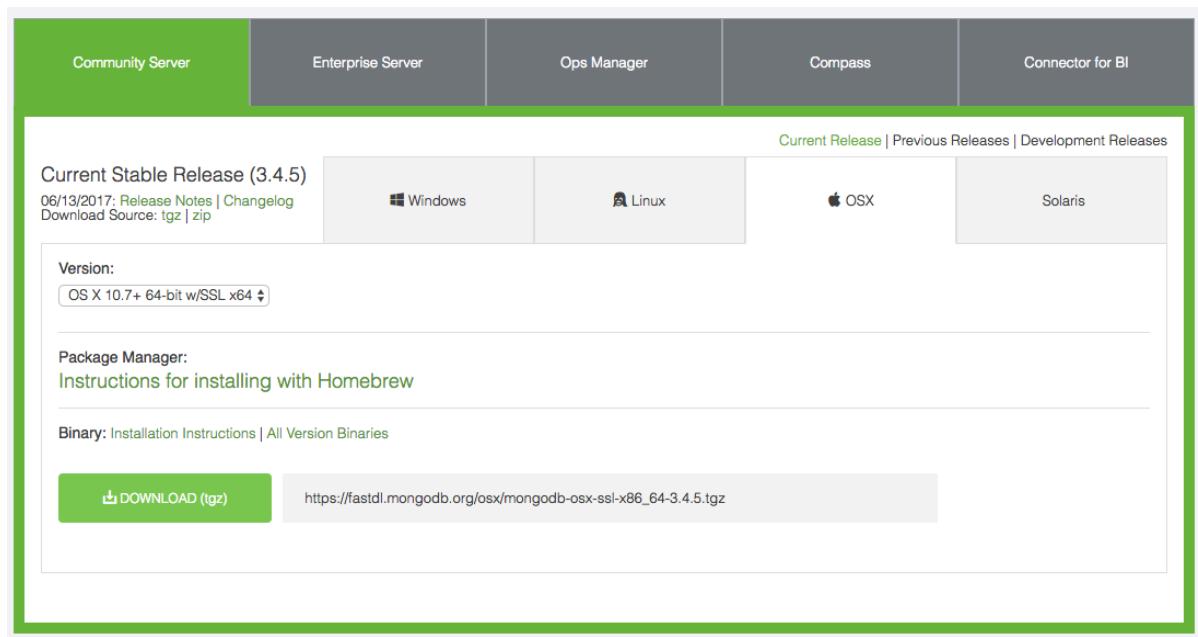
Recomendado para la mayoría

**v8.1.1 Actual**

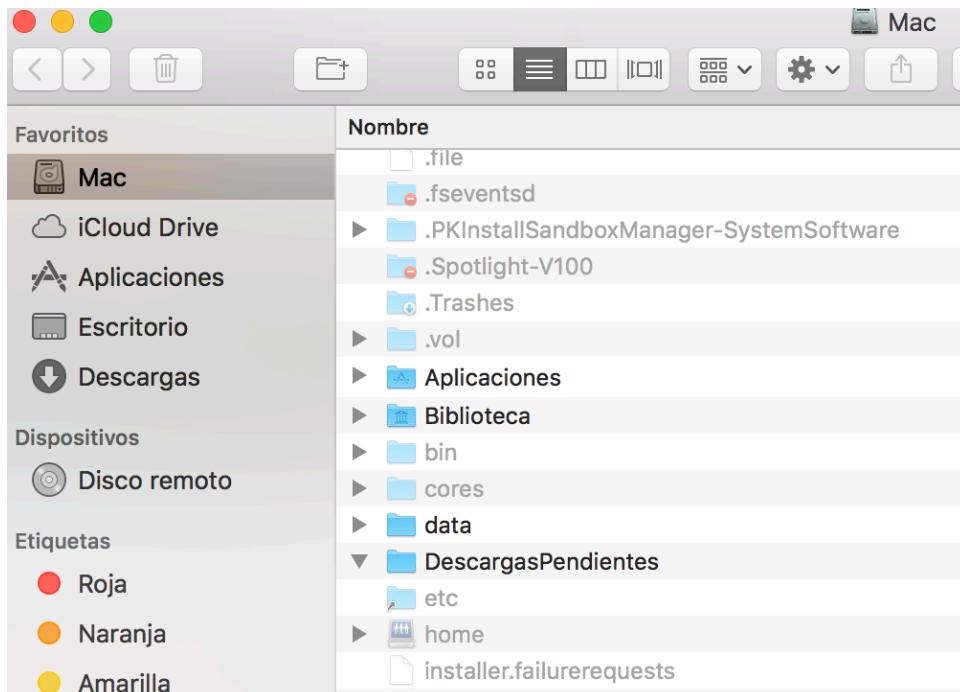
Últimas características

[Otras Descargas](#) | [Cambios](#) | [Documentación del API](#)      [Otras Descargas](#) | [Cambios](#) | [Documentación del API](#)

Tenemos que instalar también Moongo Db. Al igual que note cuanta con su instalador.



Una vez descargado e instalado tenemos que hacer una ultima cosa. Dentro de la raíz del sistema tenemos que crear una carpeta llamada Data que será donde mongo guardara sus archivos.



Por último vamos a instalar TypeScript. Esta librería se instala bajo node de forma global en el sistema. Para ello abriremos una consola e insertaremos el siguiente comando:

```
npm install -g typescript
```

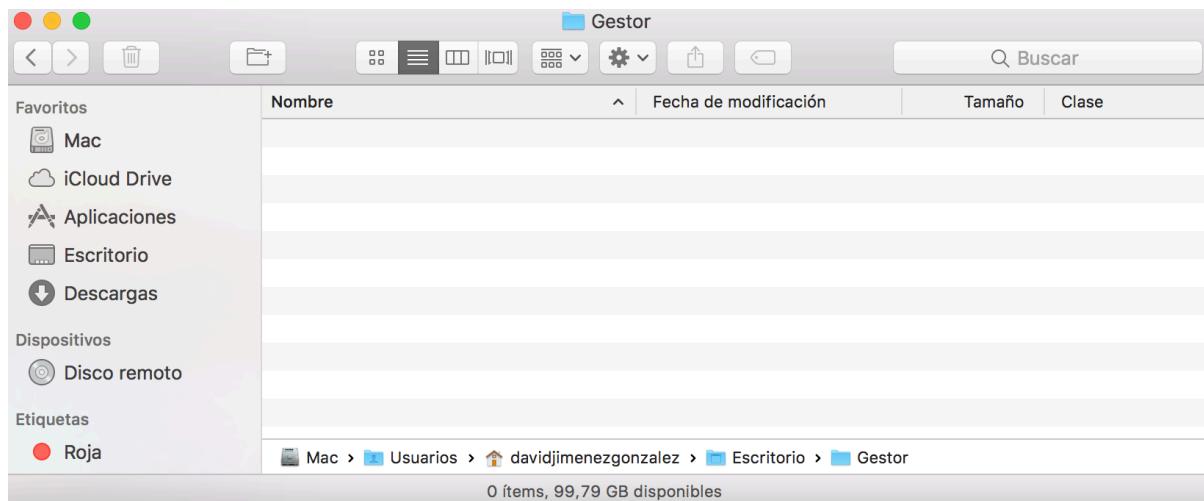


A screenshot of a macOS terminal window. The title bar says "davidjimenezgonzalez — -bash — 80x24". The window shows the command "Last login: Wed Jun 14 10:43:31 on ttys003" followed by "MacBook-Pro-de-David:~ davidjimenezgonzalez\$ npm install -g typescript". The cursor is at the end of the command.

Esto iniciará el proceso de instalación. TypeScript ademas nos provee de un transpilador para convertir el código a javascript puro. Para ello cuenta con un comando llamado tsc. No usaremos este comando de forma pura, sino que lo invocaremos al iniciar el proyecto y lo pondremos a trabajar para que visualize los cambios y transpile en tiempo real.

Para el resto de dependencias usaremos npm. Npm cuanta con un archivo llamado package.json donde configuraremos los comandos del proyecto, las dependencias y los datos del proyecto tales como nombre, descripcion, versión...

Lo primero será crear una carpeta en cualquier lugar del sistema. Llamaremos a esta carpeta Gestor, pero puede tener cualquier nombre.



Seguidamente abriremos la terminal del sistema y nos posicionaremos en dicha carpeta.

```
Gestor — -bash — 80x24
Last login: Wed Jun 14 10:43:31 on ttys003
MacBook-Pro-de-David:~ davidjimenezgonzalez$ cd desktop/gestor
MacBook-Pro-de-David:gestor davidjimenezgonzalez$ ls
MacBook-Pro-de-David:gestor davidjimenezgonzalez$
```

A screenshot of a terminal window. The title bar says "Gestor — -bash — 80x24". The terminal output shows the user's last login information, then they change their directory to "desktop/gestor", list the contents of that directory, and finally end the session. The terminal window has a light gray background with black text.

Una vez sistuados escribiremos el comando npm init. Este comando iniciar un script para la creación del archivo package.json. Bastará con rellenar los campos básicos que nos pide.

```
MacBook-Pro-de-David:gestor davidjimenezgonzalez$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
[name: (Gestor)
Sorry, name can no longer contain capital letters.
[name: (Gestor) gestor
[version: (1.0.0)
[description: Proyecto Daw Gestor
[entry point: (index.js) server.js
[test command:
[git repository:
[keywords:
[author:
[license: (ISC)
About to write to /Users/davidjimenezgonzalez/Desktop/Gestor/package.json:

{
  "name": "gestor",
  "version": "1.0.0",
  "description": "Proyecto Daw Gestor",
  "main": "server.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes) yes
MacBook-Pro-de-David:gestor davidjimenezgonzalez$
```

Una vez creado el archivo contendrá los campos básicos, que iremos ampliando a continuación.

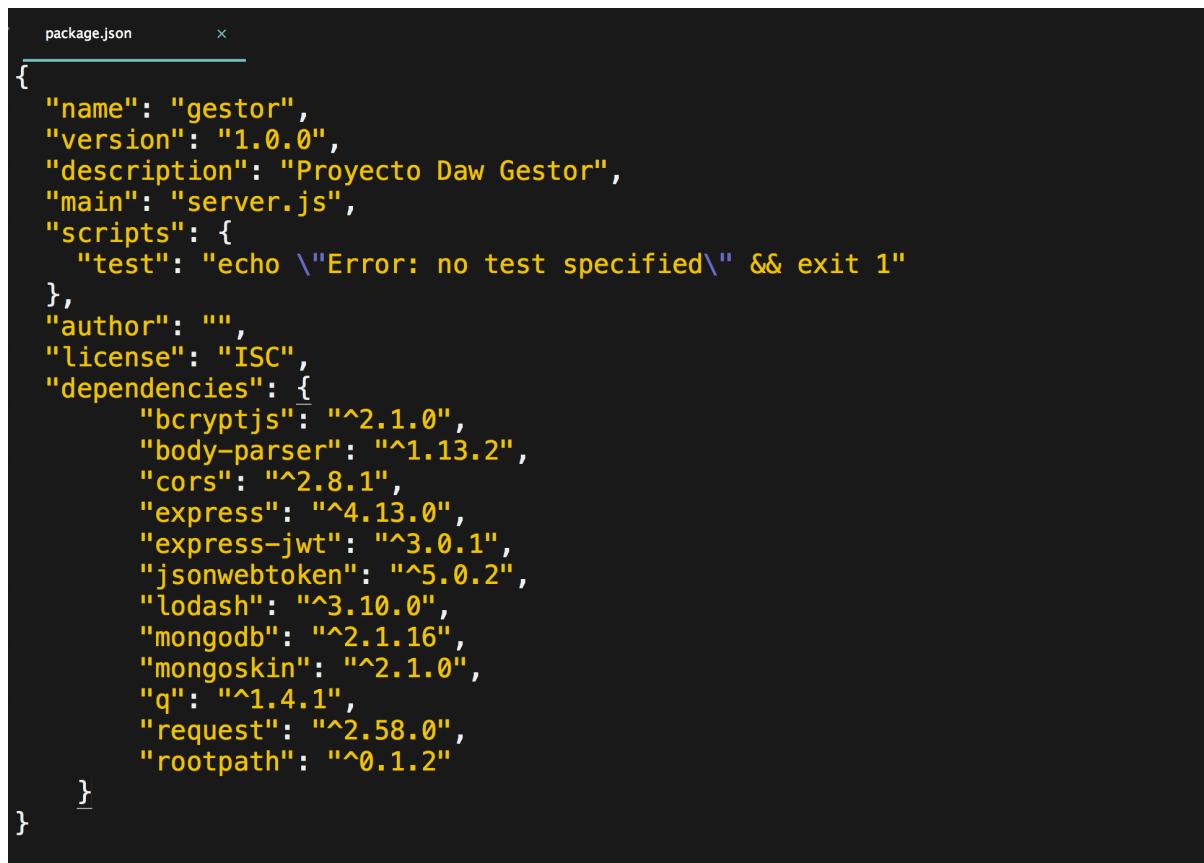


```
package.json
{
  "name": "gestor",
  "version": "1.0.0",
  "description": "Proyecto Daw Gestor",
  "main": "server.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Lo primero será configurar las dependencias.

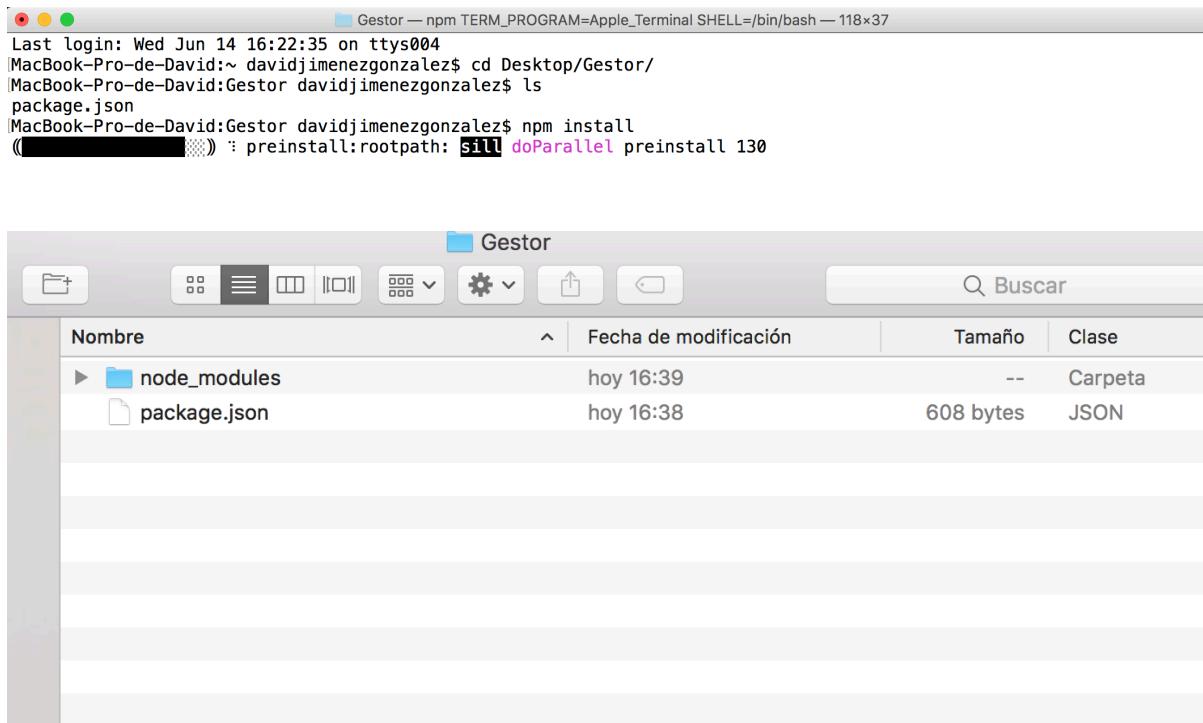
```
"bcryptjs": "^2.1.0", //parseado de peticiones
"body-parser": "^1.13.2", //parseado de peticiones
"cors": "^2.8.1", //Cors para el servidor
"express": "^4.13.0", //Libreria para el manejo del server
"express-jwt": "^3.0.1", //Tokens para el servidor
"jsonwebtoken": "^5.0.2", //Tokens para el servidor
"lodash": "^3.10.0", //Gestor para las dependencias internas
"mongodb": "^2.1.16", //Libreria para la base de datos
"mongoskin": "^2.1.0", //Libreria para la base de datos
"q": "^1.4.1", //Peticiones http asíncronas
"request": "^2.58.0", //Parseado de peticiones
"rootpath": "^0.1.2" //Libreria para las rutas
```

Una vez añadidas las dependencias el archivo quedará así



```
package.json
{
  "name": "gestor",
  "version": "1.0.0",
  "description": "Proyecto DAW Gestor",
  "main": "server.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "bcryptjs": "^2.1.0",
    "body-parser": "^1.13.2",
    "cors": "^2.8.1",
    "express": "^4.13.0",
    "express-jwt": "^3.0.1",
    "jsonwebtoken": "^5.0.2",
    "lodash": "^3.10.0",
    "mongodb": "^2.1.16",
    "mongoskin": "^2.1.0",
    "q": "^1.4.1",
    "request": "^2.58.0",
    "rootpath": "^0.1.2"
  }
}
```

Bastará con escribir npm install en la consola para instalarlas. Esto creará una carpeta en nuestro directorio llamada node\_modules donde se guardarán los archivos necesarios.



The screenshot shows a Mac OS X desktop environment. At the top, there is a menu bar with 'Gestor' selected. Below it is a Terminal window titled 'Gestor — npm TERM\_PROGRAM=Apple\_Terminal SHELL=/bin/bash — 118x37'. The terminal history shows:

```
Last login: Wed Jun 14 16:22:35 on ttys004
MacBook-Pro-de-David:~ davidjimenezgonzalez$ cd Desktop/Gestor/
MacBook-Pro-de-David:Gestor davidjimenezgonzalez$ ls
package.json
MacBook-Pro-de-David:Gestor davidjimenezgonzalez$ npm install
(██████████) : preinstall:rootpath: sill doParallel preinstall 130
```

Below the Terminal is a Finder window titled 'Gestor'. The contents pane shows a folder named 'node\_modules' and a file named 'package.json'. The table view shows the following details:

Nombre	Fecha de modificación	Tamaño	Clase
node_modules	hoy 16:39	--	Carpeta
package.json	hoy 16:38	608 bytes	JSON

Para el cliente crearemos otra carpeta diferente. Angular nos provee un quickstart para realizar su instalación más rápidamente. Cabe decir que es lo mismo que acabamos de hacer, pero con dependencias y comandos ya predefinidos.

Los comando a ejecutar serán los siguientes. Estos comandos tendrán que ir en orden.

```
npm install -g @angular/cli //Instala el gestor de dependencias de angular
```

```
ng new gestorclient //Crea la carpeta para el cliente e instal las dependencias
```

```
ng serve —open //Inicia el servidor del cliente para desarrollo
```

# Servidor

## Configuración General

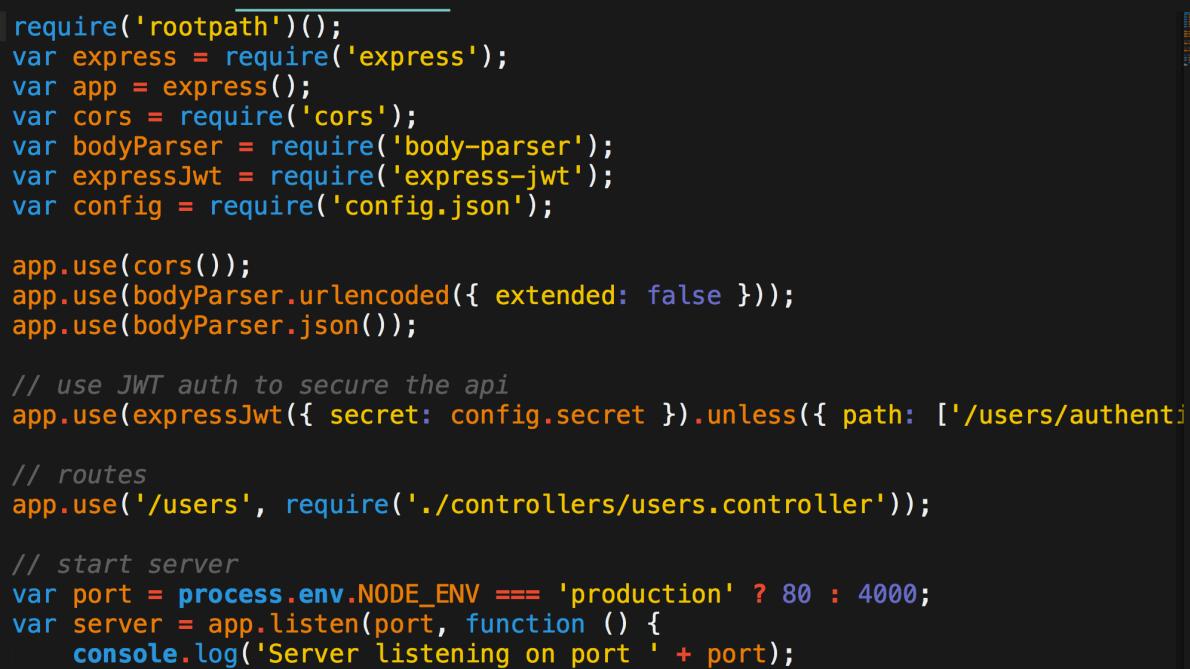
Crearemos el archivo config.js donde definiremos la ruta para la conexión a la base de datos, la ruta donde se inicia el servidor y la clave para los tokens.



```
> config.json x

{
  "connectionString": "mongodb://localhost:27017/gestor",
  "apiUrl": "http://localhost:4000",
  "secret": "gestoraaa123bbb"
}
```

Crearemos el archivo server.js donde configuraremos el servidor. En este archivo de configura tanto las peticiones como los tokes, así como se va levantar el servidor.



```
require('rootpath')();
var express = require('express');
var app = express();
var cors = require('cors');
var bodyParser = require('body-parser');
var expressJwt = require('express-jwt');
var config = require('config.json');

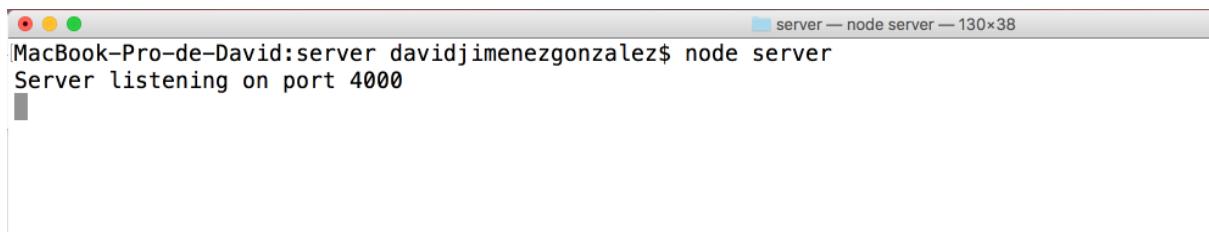
app.use(cors());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());

// use JWT auth to secure the api
app.use(expressJwt({ secret: config.secret }).unless({ path: ['/users/authenticate'] }));

// routes
app.use('/users', require('./controllers/users.controller'));

// start server
var port = process.env.NODE_ENV === 'production' ? 80 : 4000;
var server = app.listen(port, function () {
  console.log('Server listening on port ' + port);
});
```

Una vez hecho esto bastará con posicionarnos en la carpeta donde esta el servidor y ejecutar node server. Este comando iniciar el servidor.



A screenshot of a terminal window titled "server — node server — 130x38". The window shows the command "MacBook-Pro-de-David:server davidjimenezgonzalez\$ node server" and the output "Server listening on port 4000".

## Controladores

Crearemos una carpeta llamada controller donde albergaremos los controladores. La aplicación contará con 2 controladores uno para los usuarios y otro para las librerías de los usuarios.

```
users.controller.js  x

var config = require('config.json');
var express = require('express');
var router = express.Router();
var userService = require('services/user.service');

// routes
router.post('/authenticate', authenticate);
router.post('/register', register);
router.get('/', getAll);
router.get('/current', getCurrent);
router.put('/:id', update);
router.delete('/:id', _delete);

module.exports = router;

function authenticate(req, res) {
  userService.authenticate(req.body.username, req.body.password)
    .then(function (user) {
      if (user) {
        // authentication successful
        res.send(user);
      } else {
        // authentication failed
        res.status(401).send('Username or password is incorrect');
      }
    })
    .catch(function (err) {
      res.status(400).send(err);
    });
}
```

## Servicios

Crearemos una carpeta llamada services donde crearemos dos archivos. Uno se llamará user y otro library. Estos definirán como se comunica el server con la base de datos.

```
user.service.js      x

var config = require('config.json');
var _ = require('lodash');
var jwt = require('jsonwebtoken');
var bcrypt = require('bcryptjs');
var Q = require('q');
var mongo = require('mongoskin');
var db = mongo.db(config.connectionString, { native_parser: true });
db.bind('users');

var service = {};

service.authenticate = authenticate;
service.getAll = getAll;
service.getById = getById;
service.create = create;
service.update = update;
service.delete = _delete;

module.exports = service;

function authenticate(username, password) {
  var deferred = Q.defer();

  db.users.findOne({ username: username }, function (err, user) {
    if (err) deferred.reject(err.name + ': ' + err.message);

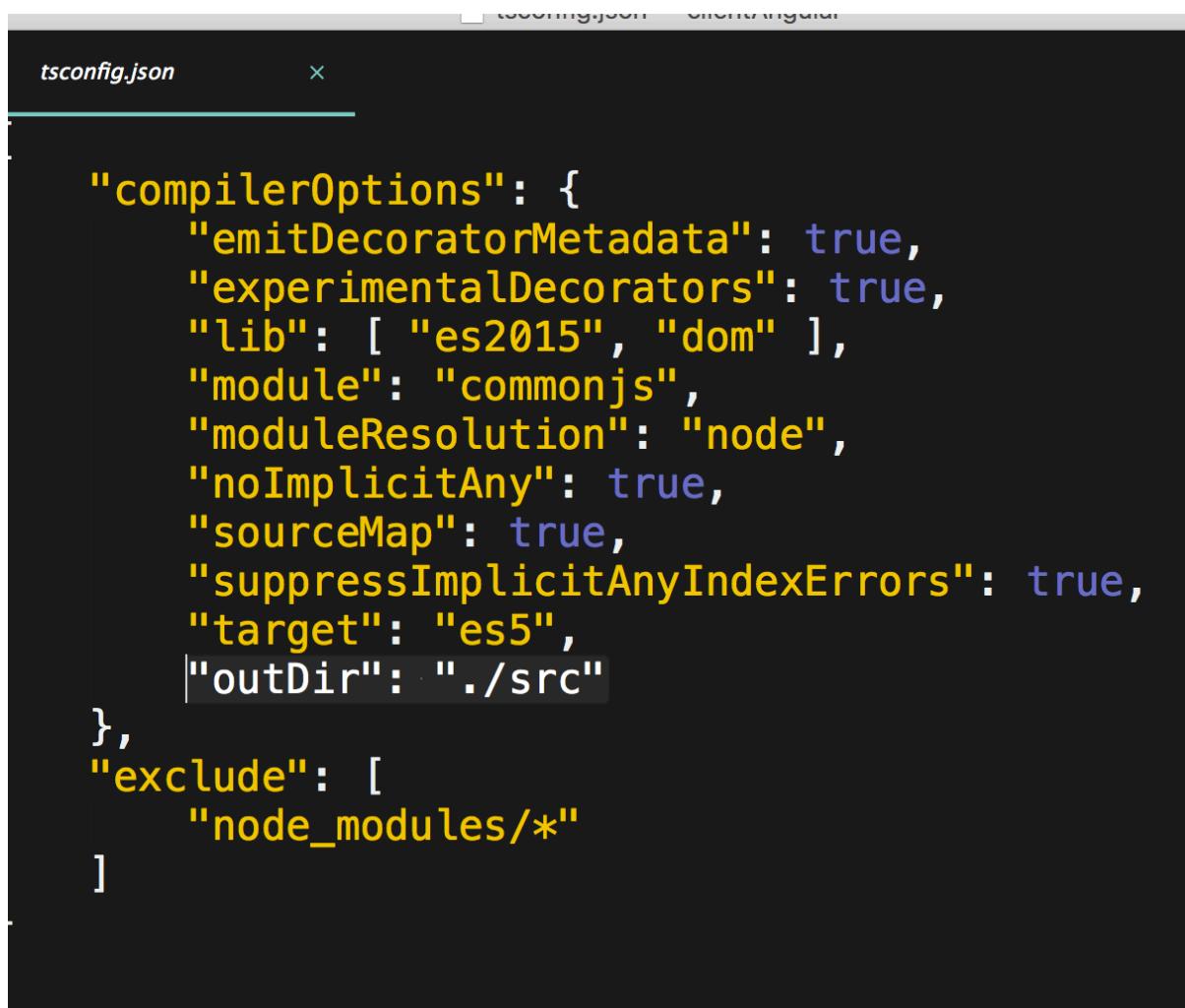
    if (user && bcrypt.compareSync(password, user.hash)) {
      // authentication successful
      deferred.resolve({
        id: user.id,
        token: jwt.sign(user, config.secret, {
          expiresIn: 60 * 60
        })
      });
    } else {
      deferred.reject('Authentication failed');
    }
  });
}
```

## Cliente

Una vez instalado el quickstart de angular tenemos que configurar varias cosas. Por una lado angular hace la transpiración de archivos en la misma carpeta donde estas los fuente. Estos puede resultar muy engorroso a la hora de codificar dado que una carpeta puede llegar a contener demasiados archivos.

Para solucionar esto vamos a modificar el archivo tsconfig.json para que los archivos transitados se guarden en otra carpeta. Bastará con añadir esta linea de código:

```
"outDir": "./src"
```



The screenshot shows a code editor window with the file 'tsconfig.json' open. The code is written in JSON and includes the following configuration:

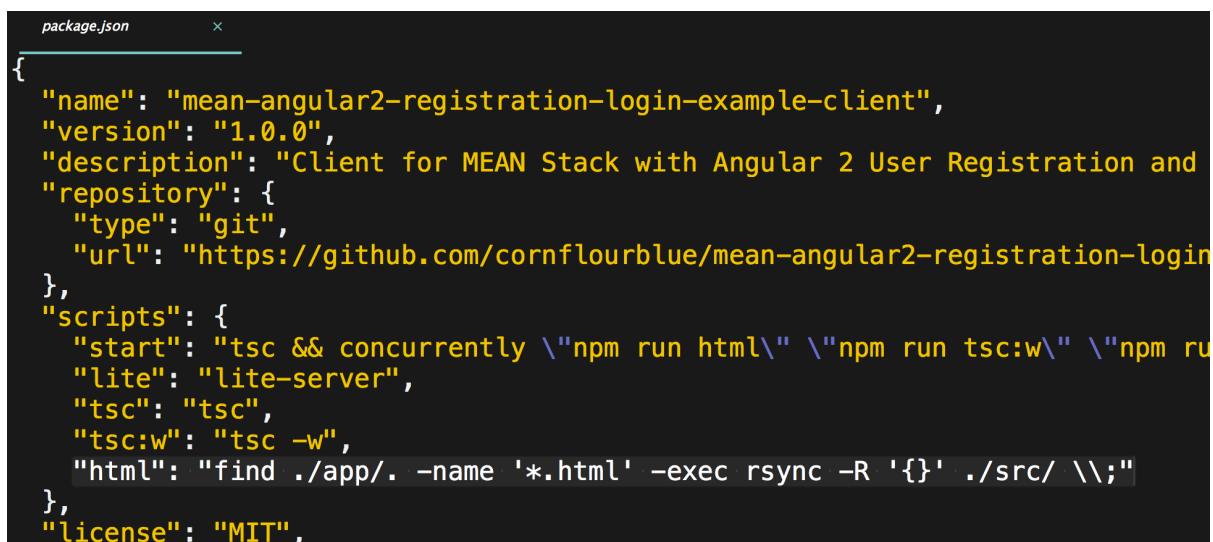
```
"compilerOptions": {
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [ "es2015", "dom" ],
    "module": "commonjs",
    "moduleResolution": "node",
    "noImplicitAny": true,
    "sourceMap": true,
    "suppressImplicitAnyIndexErrors": true,
    "target": "es5",
    "outDir": "./src"
},
"exclude": [
    "node_modules/*"
]
```

Esto genera un problema, dado que typescript solo transpila los archivos .ts a .js pero no los html ni los css. Para ello vamos a configurar un comando para que lo haga por nosotros.

Abriremos el archivo package.json y añadiremos esta linea en los scripts:

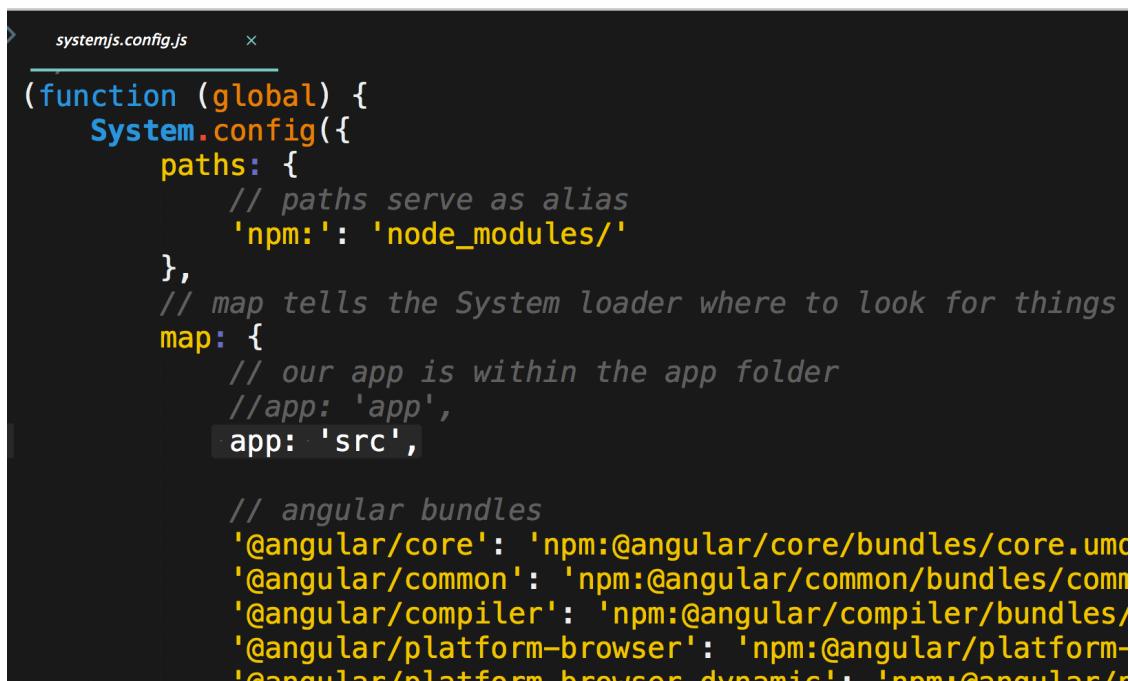
```
"html": "find ./app/. -name '*.html' -exec rsync -R '{}' ./src/ \\;"
```

Esta linea solo funciona en mac, si estamos en otro entorno abra que modificar el comando en función del sistema.



```
package.json
{
  "name": "mean-angular2-registration-login-example-client",
  "version": "1.0.0",
  "description": "Client for MEAN Stack with Angular 2 User Registration and Login Example Client",
  "repository": {
    "type": "git",
    "url": "https://github.com/cornflourblue/mean-angular2-registration-login-example-client"
  },
  "scripts": {
    "start": "tsc && concurrently \"npm run html\" \"npm run tsc:w\" \"npm run lite\"",
    "lite": "lite-server",
    "tsc": "tsc",
    "tsc:w": "tsc -w",
    "html": "find ./app/. -name '*.html' -exec rsync -R '{}' ./src/ \\;"
  },
  "license": "MIT",
}
```

Por último tendremos que informar al gestor de demencias system donde tiene que buscar los nuevos archivos. Para ello bastará con modificar el map del archivo system.config.js



```
systemjs.config.js
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      //app: 'app',
      app: 'src',

      // angular bundles
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
      '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-browser.umd.js',
      '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-dynamic/bundles/platform-browser-dynamic.umd.js'
    }
  });
})(this);
```

Con esto ya estará todo configurado y bastara con ejecutar npm start para iniciar el servidor del cliente y se apliquen los cambios. Pese a ello esto tiene un inconveniente, cada vez que modifiquemos un archivo html o css tendremos que ejecutar npm run html para copiar dichos archivos.

## Modulos

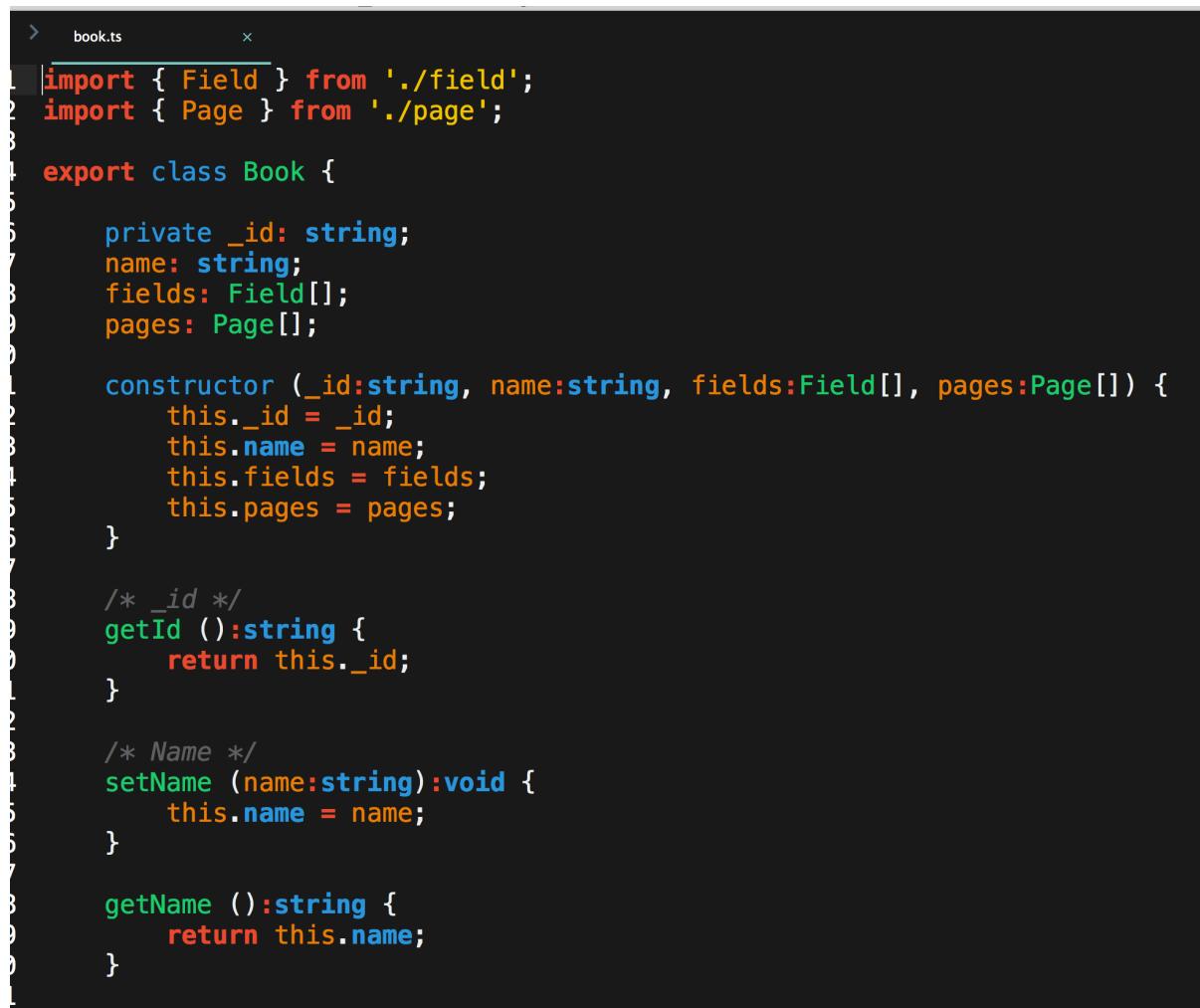
Dado la envergadura del proyecto, con un único modulo será mas que suficiente para el proyecto. Cabe decir que si este creciera seria bueno separar el código en varios modelos que encargan de cosas específicas.

El quickstart de angular ya nos crea el archivo app.module.ts y configura todo por lo que de momento no tendremos que tocar nada.

## Modelos

Vamos a crear los modelos necesarios para la aplicación. Lo primero será crear una carpeta dentro de app llamada \_models. El guion bajo es un estándar de angular.

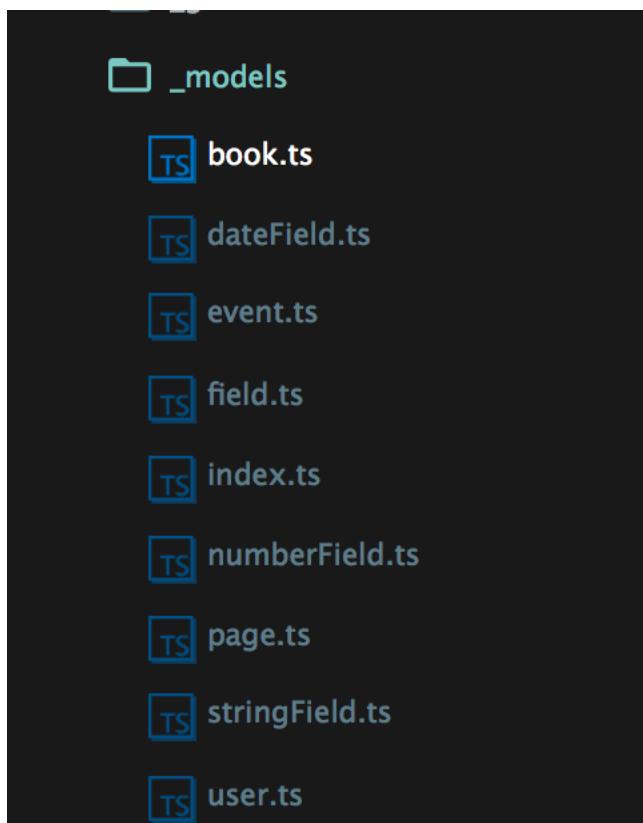
Dentro de esta carpeta crearemos un archivo por cada modelo que queramos añadir. Los modelos no son mas que un clase o una interfaz que define una estructura de datos.



```
> book.ts <

1 import { Field } from './field';
2 import { Page } from './page';
3
4 export class Book {
5
6     private _id: string;
7     name: string;
8     fields: Field[];
9     pages: Page[];
10
11     constructor (_id:string, name:string, fields:Field[], pages:Page[]) {
12         this._id = _id;
13         this.name = name;
14         this.fields = fields;
15         this.pages = pages;
16     }
17
18     /* _id */
19     getId ():string {
20         return this._id;
21     }
22
23     /* Name */
24     setName (name:string):void {
25         this.name = name;
26     }
27
28     getName ():string {
29         return this.name;
30     }
31 }
```

Crearemos los siguientes modelos:



## Servicios

Vamos a crear los servicios necesarios para que la aplicación se comunique con el servidor y se sincronice con este.

Lo primero será crear una carpeta dentro de app llamada \_services. Dentro definiremos los servicios necesarios.

```
book.service.ts x

import { Injectable } from '@angular/core';
import { Http, Headers, RequestOptions, Response } from '@angular/http';

import { AppConfig } from '../app.config';
import { User, Book, NumberField, DateField, StringField, Page } from '../'

@Injectable()
export class BookService {

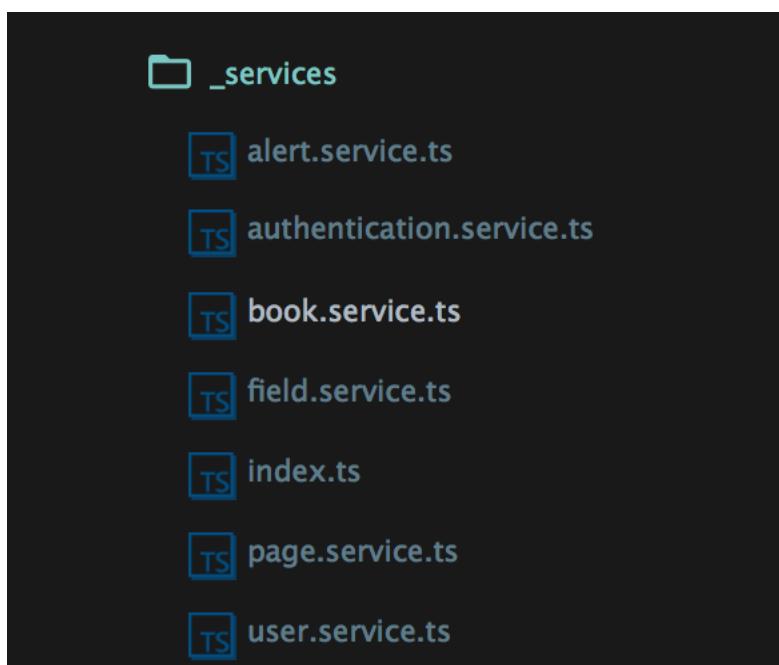
  constructor(private http: Http, private config: AppConfig) {
  }

  getAllBooks ():Book[] { //Revisar
    let book = new Book(
      '123',
      'Mi primer LIBro',
      [new NumberField('123', 'Mi primer field', 123)],
      [new Page('123',
        [
          new NumberField('123', 'Number', 123),
          new StringField('123', 'String', "hola"),
          new DateField('123', 'Date', new Date())
        ]
      ),
      new Page('123',
        [
          new NumberField('123', 'Number', 123),
          new StringField('123', 'String', "hola"),
          new DateField('123', 'Date', new Date())
        ]
      )
    );
  }
}
```

Cabe decir que los servicios hay que añadirlos al modulo para poder usarlos en cualquier parte de este.

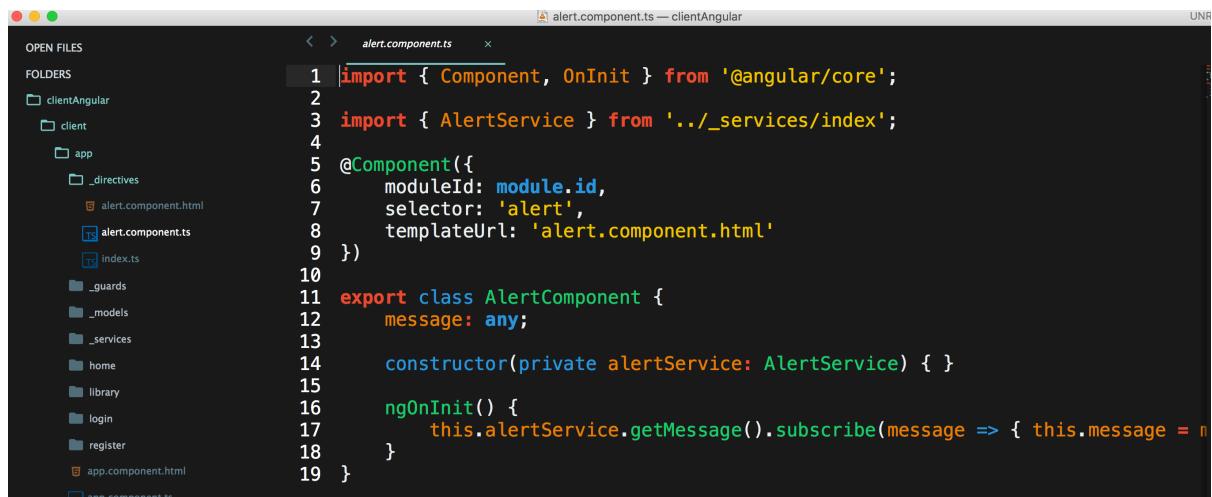
```
  ],
  providers: [
    AppConfig,
    AuthGuard,
    AlertService,
    AuthenticationService,
    UserService,
    BookService,
    PageService,
    FieldService
  ],
  bootstrap: [AppComponent]
```

Estos serán todos los servicios necesarios:



## Directivas

Las directivas estarán alojadas en la carpeta \_directives. De momento solo tendremos una directiva para las alertas globales.



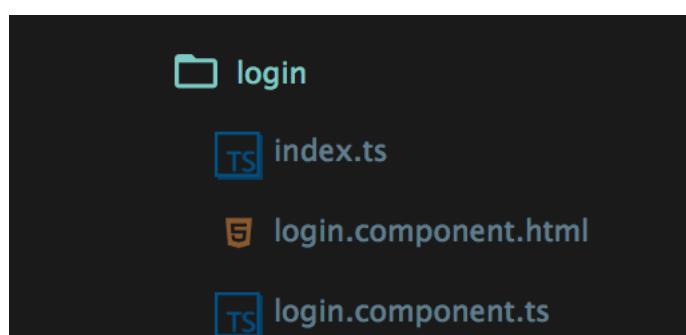
```
import { Component, OnInit } from '@angular/core';
import { AlertService } from '../_services/index';
@Component({
  moduleId: module.id,
  selector: 'alert',
  templateUrl: 'alert.component.html'
})
export class AlertComponent {
  message: any;
  constructor(private alertService: AlertService) { }
  ngOnInit() {
    this.alertService.getMessage().subscribe(message => { this.message = message });
  }
}
```

## Componentes

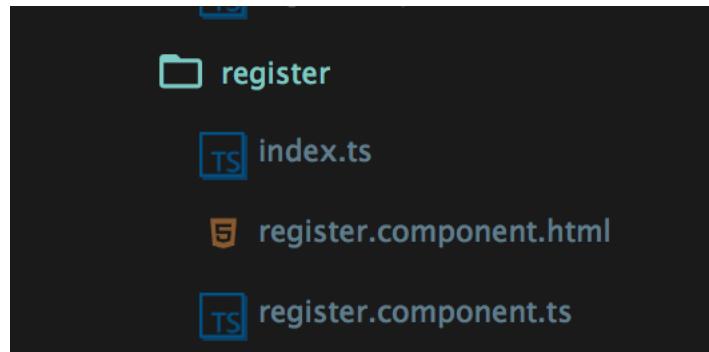
Dada la importancia de los componentes estos no tendrán una carpeta global, sino que tendrán una carpeta para cada uno de ellos.

Tendremos 3 componentes base que definirán las rutas de la aplicación cliente.

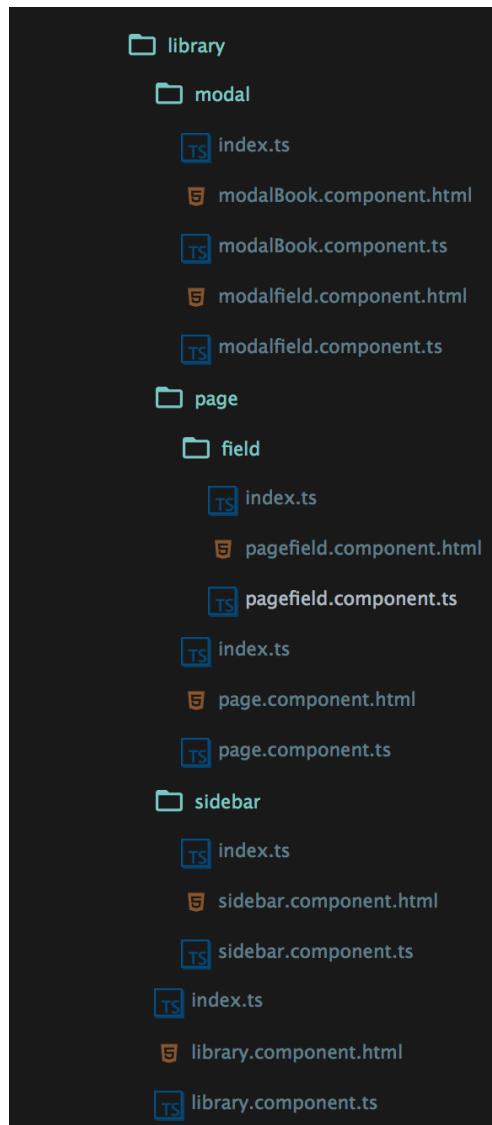
Login: Define el componente para hacer login en la aplicación. Este componente genera el token al loguearse y valida el login.



Register: Define el componente para registro de usuarios.



Library: El componente raíz de la aplicación. Este componente engloba otros y define la aplicación en si una vez que nos hemos logueado.



En este componente se guardara el modelo de la libreria. Dentro de el tenemos un componente para la barra izquierda y otro para la visualización de las páginas así como otro componente para los modal que se usara para crear las paginas los libros y los campos.

```
> library.component.ts x

import { Component, OnInit, Input } from '@angular/core';
import { User, Book, Field, Page, Event } from '../_models/index';
import { BookService, FieldService } from '../_services/index';

@Component({
  moduleId: module.id,
  templateUrl: 'library.component.html',
  selector: 'library-component'
})

export class LibraryComponent implements OnInit {

  private bookService: BookService;
  private currentUser: User;
  private books: Book[];
  private selectedBook: Book;
  private selectedField: Field;
  private showModalBook: Book;
  private showModalField: Field;
  private selectedPageIndex: number;

  constructor(bookService: BookService) {
    this.bookService = bookService;
    this.currentUser = JSON.parse(localStorage.getItem('currentUser'));
  }

  ngOnInit() {
    this.loadAllBooks();
  }
}
```

# Visualización de la aplicación

localhost:3000/login

Login

Username

Password

Login    Register

localhost:3000/login

Username or password is incorrect

Login

Username

pepe

Password

\*\*\*

Login    Register

localhost:3000/login

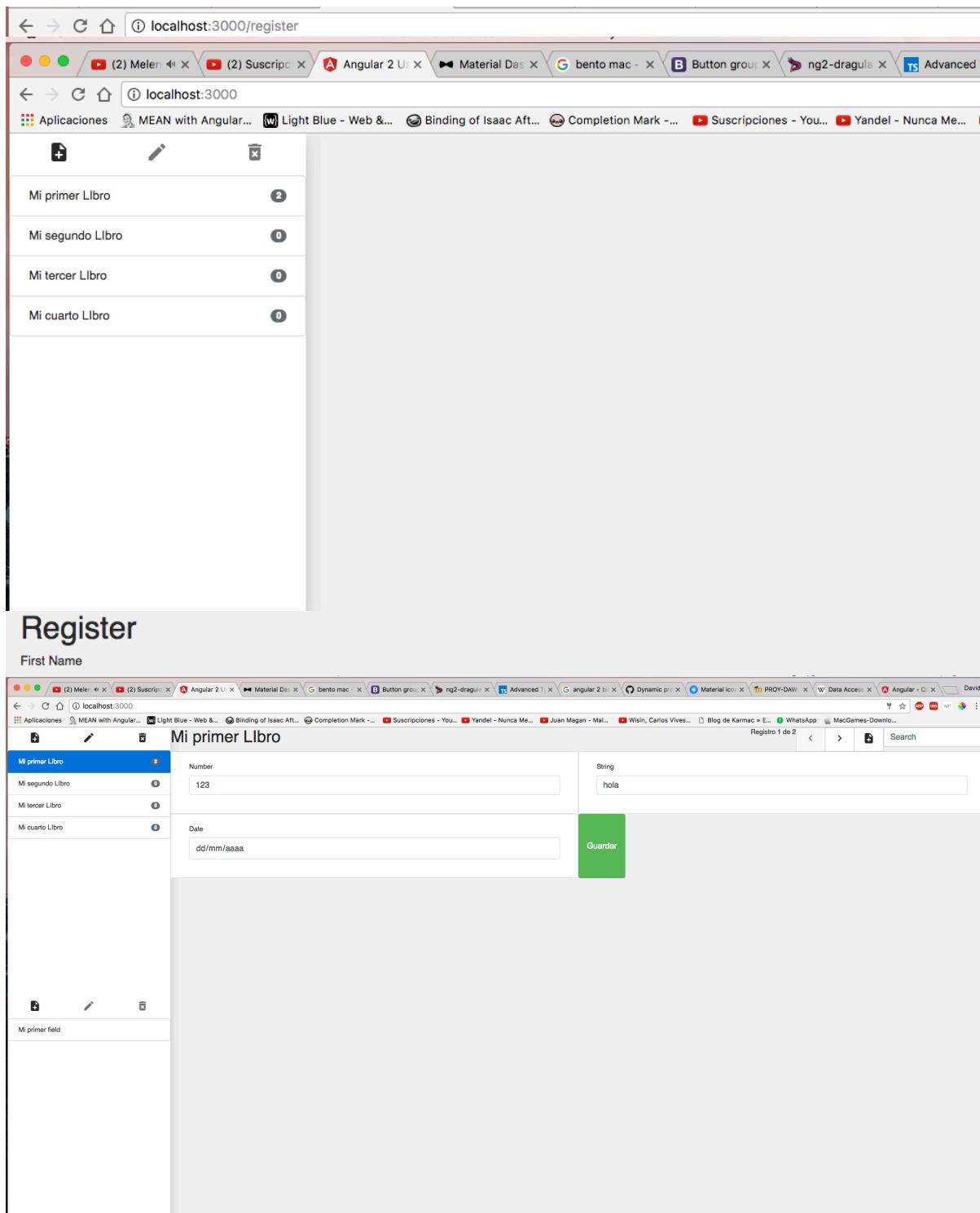
Registration successful

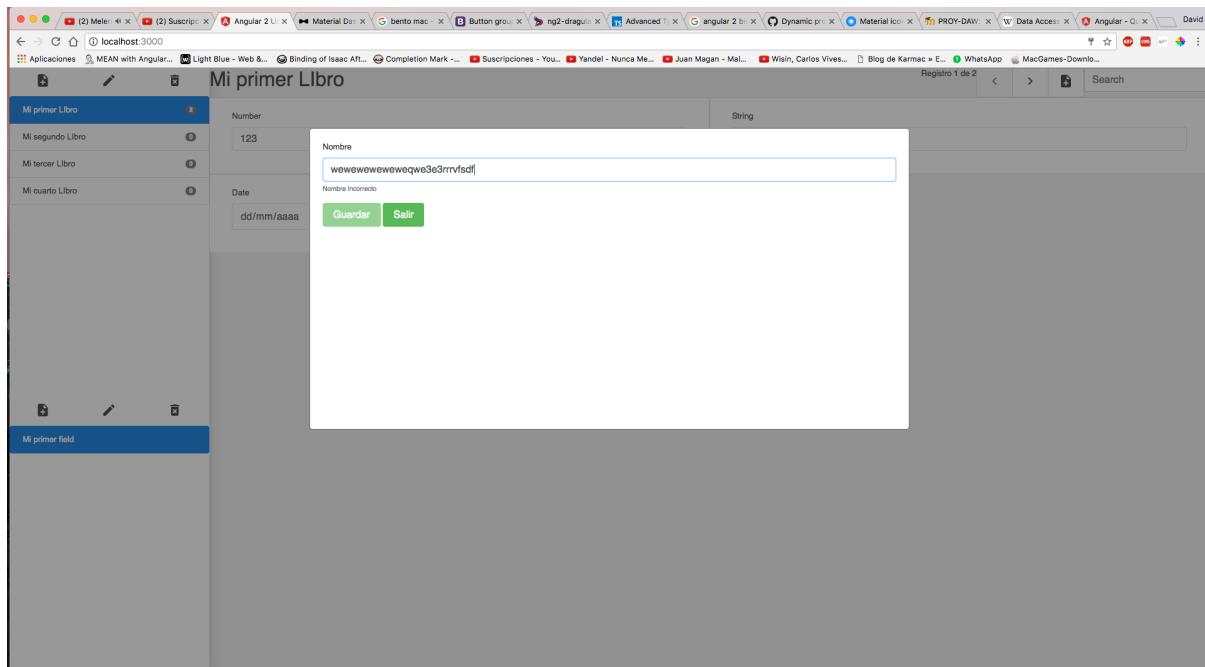
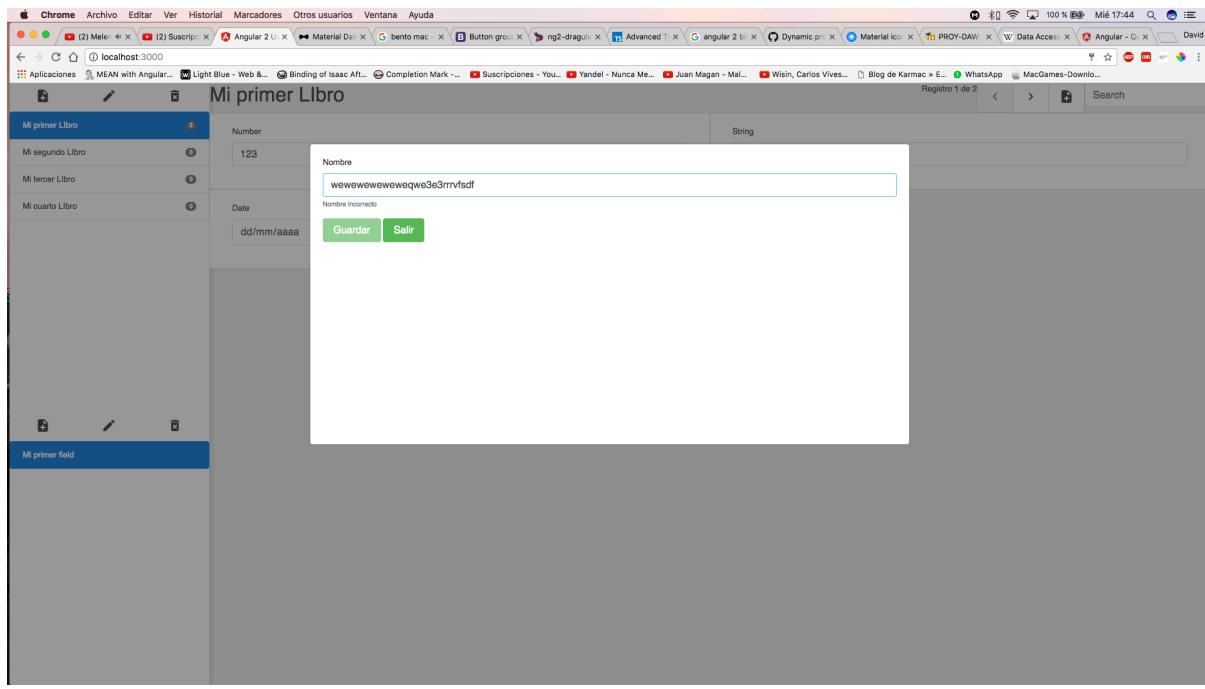
Login

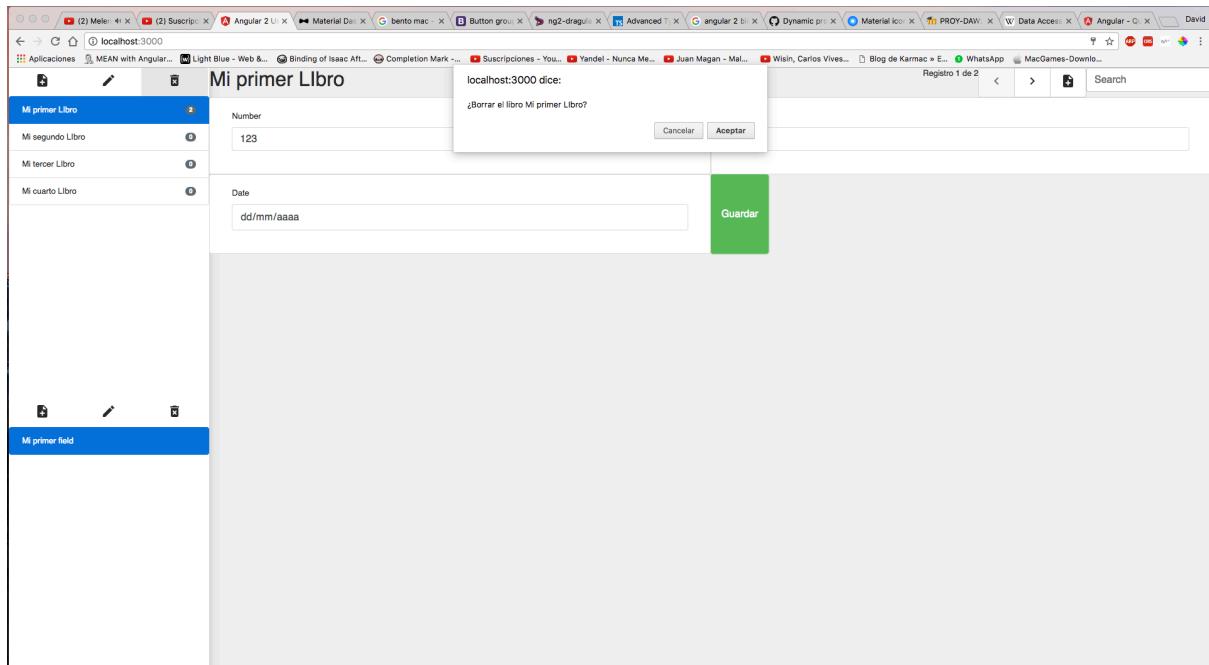
Username

Password

Login    Register







# Bibliografia

<http://expressjs.com/es/>

<https://nodejs.org/es/>

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

<https://facebook.github.io/react/>

<http://redux.js.org/>

<http://materializecss.com/>

<https://facebook.github.io/imutable-js/>

<http://gulpjs.com/>

<https://babeljs.io/>

<http://browserify.org/>

<https://github.com/substack/watchify>

<https://visionmedia.github.io/page.js/>

<https://github.com/auth0/node-jsonwebtoken>

<http://mongoosejs.com/>

<https://www.npmjs.com/package/cors>

# Preguntas relacionadas con el módulo de FCT

Bases de datos

¿Se utiliza Access en la empresa? Si es así ¿qué tipo de tareas se realizan con dicho software?

Si, pero solo Excel para general informes.

¿Qué otros SGBD relacionales utilizan? ¿qué tipo de tareas se realizan?

Sql Lite, para almacenamiento temporal de información antes de ser transmitida a Eslastic Search.

Construyes

Sistemas Operativos

¿Qué Sistemas Operativos utilizan (Windows, Linux, otros)? Indica también la versión y/o distribución.

Windows 10 y Linux bajo una distribución de Ubuntu.

¿Qué tareas relacionadas con los Sistemas Operativos has realizado?

Gestión de los servidores.

Has configurado algún servicio de red ¿Cuál?

No

Lenguajes de marcas

¿Utilizan XML (XML Schema, XPath, XQuery)?

Si

¿Actualizan la página web de la empresa con alguna base de datos utilizando XML?

No

Programación en entorno servidor (PHP, JSP, ASP .NET...)

¿Qué entorno de desarrollo se utiliza?: Visual Studio, NetBeans, Eclipse,...  
Eclipse y Visual Estudio

En caso de emplear Visual Studio utilizan el patrón de diseño MVC o Web Forms? Y ¿qué lenguaje de programación?: PHP, Visual Basic, C#, Java....  
Java y Visual Basic.

En el caso de programar en PHP, ¿realizan POO? ¿Con algún Framework:  
Laravel, Symfony, Yii, Zend..?

Solo se maneja java para el servidor.

¿Se trabaja con o sobre algún gestor de contenidos tipo Wordpress  
(desarrollo de plugins o de temas), Prestashop.....como usuario, administrador o  
desarrollador?

Se utiliza RedMine para la gestión de incidencias y proyectos con plugins  
reinstalados.

¿Qué servidores utilizan: web, FTP, DNS,...?  
Web, DNS y FTP

¿Qué servidores de aplicaciones utilizan?

Programación en entorno cliente

¿Utilizan JavaScript? Con qué entorno se trabaja?  
Si, toda la aplicación cliente esta escrita en AngularJS. Con Eclipse.

Se utilizan librerías como JQuery con Javascript. ¿Cuáles?

Las 2 librerías principal sen AngularJs y Jquery. También se usa un serie de  
librerías para la gestión de timepicker, slider...