

# ***Módulo del proyecto DAW***

**Gestor**

David Jiménez González

# Índice

Objetivos	Pág 3
Recursos hardware y software	Pág 5
Fases	Pág 10
Desarrollo de las fases	Pág 17
Bibliografía	Pág 36
Preguntas relacionadas con el módulo de FCT	Pág 37

# 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 contarán con páginas. 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 trafico 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 NodeJs. Para la gestión de las rutas usaremos la librería ExpressJs.

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 encargaran 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 tareas 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.

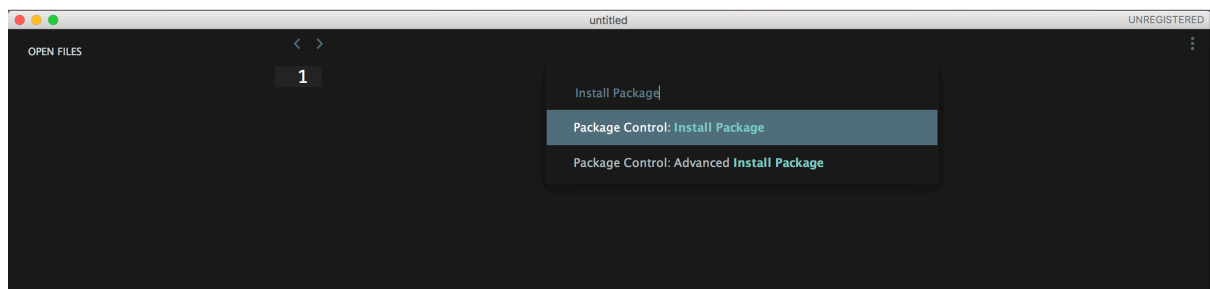
Para facilitar el manejo del state global contamos con ImmutableJs, una librería para crear estructura de datos inmutables haciendo así que cada cambien en el state se refleje en la vista haciendo que este vuelva a renderizarse.

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.

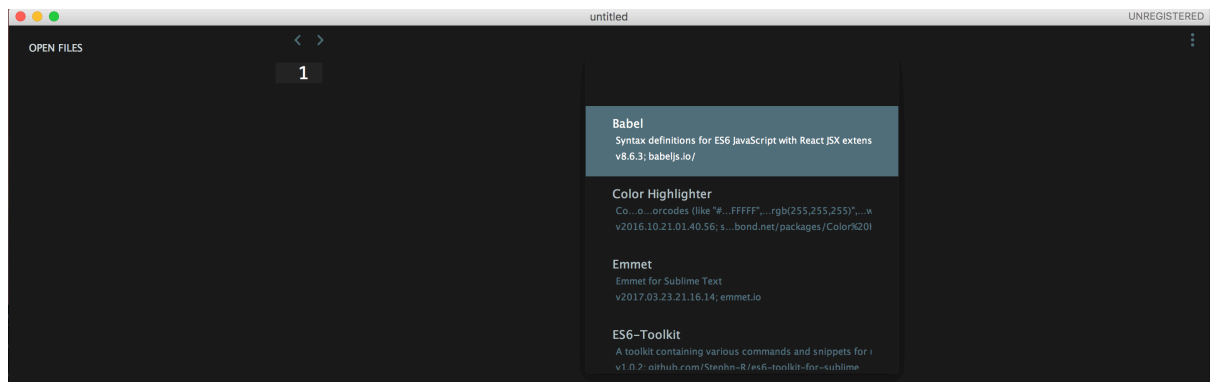
## *Editor de código*

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 pseudo 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 hablara 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 pseudo lenguaje que compila a css. Nos ofrece mayores funcionalidad con menos código.

Materializze: Es un paquete que nos permite atajos para esta librería. Se hablara 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 MONGODB 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
```



```
MacBook-Pro-de-David:~ davidjimenezgonzalez$ sudo mongod
Password:
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] MongoDB starting : pid=1026 port=27017 dbpath=/data/db 64-bit host=MacBook-Pro-de-David.local
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] db version v3.4.3
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] git version: f07437fb5a6cca07c10bafa78365456eb1d6d5e1
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.2k 26 Jan 2017
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] allocator: system
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] modules: none
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] build environment:
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] distarch: x86_64
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] target_arch: x86_64
2017-05-30T20:18:12.003+0200 I CONTROL [initandlisten] options: {}
2017-05-30T20:18:12.004+0200 I - [initandlisten] Detected data files in /data/db created by the 'wiredTiger' storage engine, so setting the active storage engine to 'wiredTiger'.
2017-05-30T20:18:12.004+0200 I STORAGE [initandlisten] wiredtiger_open config: create,cache_size=3584M,session_max=20000,eviction=(threads_min=4,threads_max=4),config_base=false,statistics=(fast),log=(enabled=true,archive=true,path=journal,compressor=snappy),file_manager=(close_idle_time=100000),checkpoint=(wait=60,log_size=2GB),statistics_log=(wait=0),
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten]
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten] ** WARNING: Access control is not enabled for the database.
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten] ** Read and write access to data and configuration is unrestricted.
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten] ** WARNING: You are running this process as the root user, which is not recommended.
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten]
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten]
2017-05-30T20:18:12.874+0200 I CONTROL [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
2017-05-30T20:18:12.901+0200 I FTDC [initandlisten] Initializing full-time diagnostic data capture with directory '/data/db/diagnostic.data'
2017-05-30T20:18:12.902+0200 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 Del Proyecto

## Dependencias

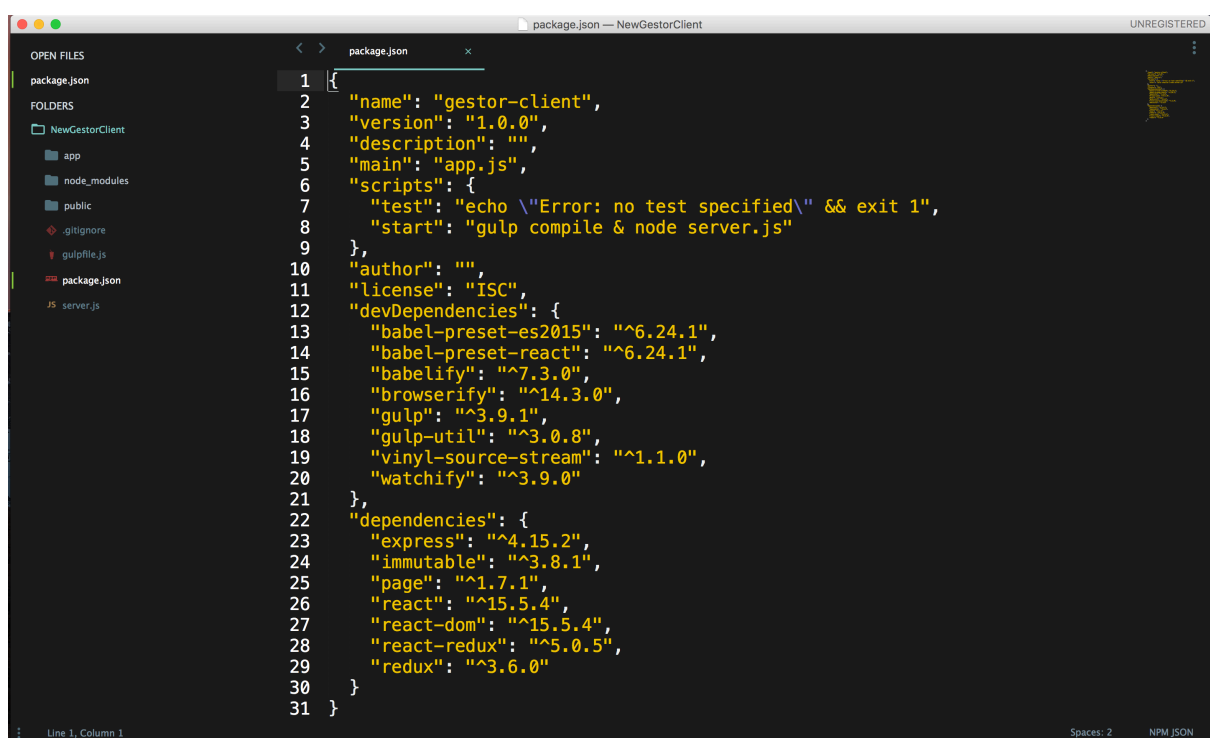
Node cuenta con un sistema de gestión de dependencias llamado npm, el cual se instala automáticamente con él. Todas las dependencias se pueden instalar en 2 ámbitos: global y local.

Por una parte las dependencias globales nos son de utilidad dado que podemos llamarlas desde cualquier parte del sistema. Por otro las dependencias locales solo son visibles para el proyecto.

Npm corre sobre Node de forma global en nuestro sistema. Bastará con abrir una terminal en mac o linux o el cmd en windows, posicionarnos en la carpeta donde tenemos el package.json y escribir el siguiente comando

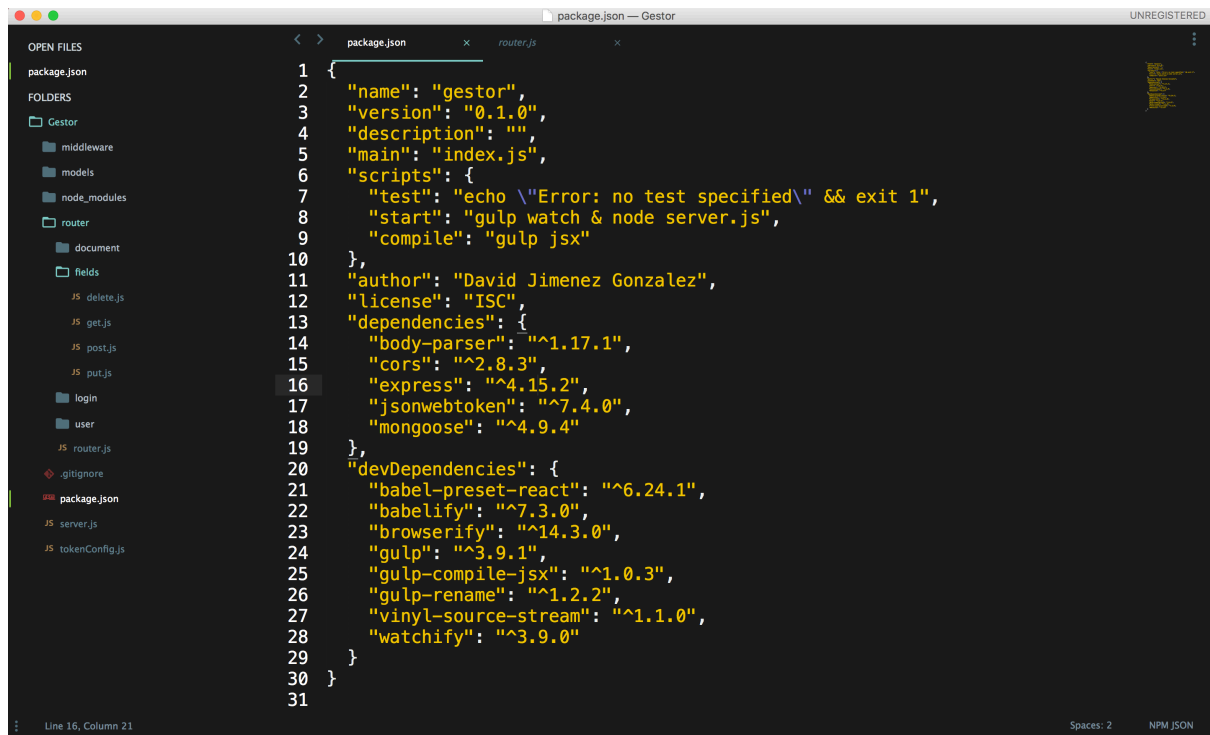
```
npm install - - save {nombre de la dependencia}
```

Esto añadirá la dependencia al archivo package.json. Si quitamos el - - save se instalara de forma global. Para instalar solo para desarrollo bastara con poner -dev al final.



```
1 {
2   "name": "gestor-client",
3   "version": "1.0.0",
4   "description": "",
5   "main": "app.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1",
8     "start": "gulp compile & node server.js"
9   },
10  "author": "",
11  "license": "ISC",
12  "devDependencies": {
13    "babel-preset-es2015": "^6.24.1",
14    "babel-preset-react": "^6.24.1",
15    "babelify": "^7.3.0",
16    "browserify": "^14.3.0",
17    "gulp": "^3.9.1",
18    "gulp-util": "^3.0.8",
19    "vinyl-source-stream": "^1.1.0",
20    "watchify": "^3.9.0"
21  },
22  "dependencies": {
23    "express": "^4.15.2",
24    "immutable": "^3.8.1",
25    "page": "^1.7.1",
26    "react": "^15.5.4",
27    "react-dom": "^15.5.4",
28    "react-redux": "^5.0.5",
29    "redux": "^3.6.0"
30  }
31 }
```

En esta imagen podemos ver las demencias instaladas en el cliente.



```
1 {
2   "name": "gestor",
3   "version": "0.1.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\\"Error: no test specified\\\" && exit 1",
8     "start": "gulp watch & node server.js",
9     "compile": "gulp jsx"
10  },
11  "author": "David Jimenez Gonzalez",
12  "license": "ISC",
13  "dependencies": {
14    "body-parser": "^1.17.1",
15    "cors": "^2.8.3",
16    "express": "^4.15.2",
17    "jsonwebtoken": "^7.4.0",
18    "mongoose": "^4.9.4"
19  },
20  "devDependencies": {
21    "babel-preset-react": "^6.24.1",
22    "babelify": "^7.3.0",
23    "browserify": "^14.3.0",
24    "gulp": "^3.9.1",
25    "gulp-compile-jsx": "^1.0.3",
26    "gulp-rename": "^1.2.2",
27    "vinyl-source-stream": "^1.1.0",
28    "watchify": "^3.9.0"
29  }
30 }
31 }
```

Aquí vemos las dependencias del servidor rest.

Podemos agrupar las dependencias en tres categorías: desarrollo, servidor rest y servidor cliente.

**Desarrollo:** estas dependencias estarán tanto en el servidor de la api rest como en el servidor para el cliente. Por un lado tenemos vinyl-source-stream, babel-preset-react, babelify, browserify y watchify. Estas librerías tienen el único propósito de transpilar el código. Además gracias a watchify podemos observar los cambios y transpilar mientras escribimos el código.

**Servidor REST:** por una parte tenemos las dependencias necesarias para parsear las peticiones a json, dado que express trabaja con objetos puros de javascript.

Gracias a body-parser y cors podemos convertir las cabeceras y las peticiones a objetos planos json para que su manejo sea más sencillo.

Por otro lado tenemos express, la librería encargada de levantar el servidor. Esta librería nos brinda un enrutador, que será con el cual gestionaremos las rutas.

Por último tenemos mongoose, la librería encargada de gestionar las peticiones a la base de datos. Mongoose nos provee de un schema. Dado que la base de datos es no relacionan, esto se hace indispensable. Además también aporta un modelo sobre el cual podremos hacer peticiones a la base de datos y ejecutar queries.

**Servidor Cliente:** en esta parte tenemos 4 tipos de dependencias. Cada una de ellas se encarga de una parte específica.

Por un lado tenemos todas las dependencias referentes a react y redux. Se explicará más adelante que es React, Redux y qué función tendrán en el proyecto.

Page es un enrutador para el cliente parecido a la funcionalidad de express. Se explicará más adelante su funcionamiento y configuración.

Immutable Js es una librería para crear estructuras de información inmutables. Esto nos será muy útil al crear el state global y manejarlo. Veremos ejemplos de su funcionamiento más adelante.

Por último tenemos express, al igual que el servidor rest se encargará de levantar y gestionarlo.

## *Estructura de carpetas*

Para albergar todo el proyecto crearemos una carpeta en el directorio raíz del sistema, o donde más nos convenga, que albergará los archivos de la aplicación. Dentro de esta crearemos 2 subcarpetas, una llamada servidor y otra cliente. Cada una de estas contendrá los archivos necesarios para el funcionamiento de la aplicación.

Dentro de la carpeta servidor crearemos 3 más router, models y middleware. Se explicará su función y contenido más adelante.

Dentro de la carpeta cliente crearemos una carpeta llamada app donde ira todo el código fuente de la aplicación cliente.

Cada una de estas 2 carpetas contarán con estos archivos:

**Package.json:** Archivo de configuración del servidor en formato json. En el se define en nombre, versión, autor, dependencias, punto de entrada y scripts.

Las dependencias tienen 2 ámbitos, desarrollo y producción. Esto es importante para que a la hora de hacer de-los no sea necesario alojar todas las dependencias de desarrollo y aligerar el almacenamiento.

El punto de entrada define el archivo que arrancará el servidor. Tendrá que estar siempre en el raíz del servidor.

El el punto de scripts se definen los comandos que podremos correr para manejar el servidor. Hablaremos mas adelante de esto en la parte de testigo y transpiración.

**Server.js:** Este será nuestro archivo de arranque del servidor. Por convención lo llamaremos Server.js, pero puede tener cualquier nombre siempre que se defina en el packaje.json.

**Gulpfile.js:** Gulp es una libreria para automatización de tareas. La instalaremos como dependencia mas adelante.

Esta libreria nos aportara 2 utilidades: testing y transpilación. Hablaremos de estos 2 apartados mas adelante.

*Transpilación: Consiste en convertir un lenguaje a una versión anterior. Dado que aun no todos los navegadores soportan las ultimas versiones de javascript el uso de estas herramientas se hace indispensable para convertir nuestro código de es6 y 7 a es5. Es son las siglas de EcmaScript, el estándar de javascript.*

**.gitignore:** Este archivo nos servirá para definir que carpetas no serán enviados al hacer un commit en git. Git será el control de versiones que usaremos para subir los archivos al servidor.

## *Creación del Servidor REST*

En esta fase desarrollaremos el software necesario para el funcionamiento del servidor REST. Este servidor correrá bajo NodeJs y Express.

Este servidor tendrá solo un cometido, atender las peticiones de consulta y actualización sobre la base de datos. Para ello definiremos una serie de rutas basadas en los verbos http que serán los puntos de entrada y salida de información del servidor.

Gracias a Express contamos con un router que nos facilita la vida al crear los middelwares necesarios para atender estas peticiones. Un middleware no es mas que una función que recibe 3 paramentos: request, response y next.

En el parametro request tenemos toda la información de la petición recibida. El parámetro response hace referencia a la respuesta con la que vamos a responder y next llama al siguiente middleware definido para esa ruta.

La estructura de cada ruta es la siguiente:

```
router."verbo" ('ruta', [middleware])
```

Tambien contaremos con un sistema de autenticación basado en token, siguiendo el estándar Auth0. Esto lo implementaremos a través de un middleware con el que contarán todas aquellas rutas que lo necesiten.

Aparte contaremos con 2 funcionalidades mas, creación y validación de los tokens.

## *Creación de los modelos de la Base de Datos*

Gracias a la librería Moongoose no vamos a interactuar directamente con la base de datos. Lo que vamos a desarrollar es una serie de modelos que harán de dao para gestionar la base de datos.

Cada modelo cuenta siempre con un Schema, que es la definición de los datos que va a contener ese modelo. Gracias a esto podremos hacer validaciones y controlar que vamos a ingresar y como en la base de datos.

Los schemas solo soportan una serie de tipos de datos. Todos los tipos primitivos: Number, String, Boolean... Arrays y ObjectId. Los ObjectId no son mas que una referencia hacia otro modelo. Estas referencias siempre serán débiles.

## *Creación del servidor Cliente*

El servidor cliente será desarrollado con NodeJs y Express. Este servidor se encargará únicamente de servir la aplicación cliente y todos los archivos estáticos de la aplicación y los generados por los usuarios.

Gracias a express bastara con exactamente 12 lineas de código para que el servidor funcione correctamente.

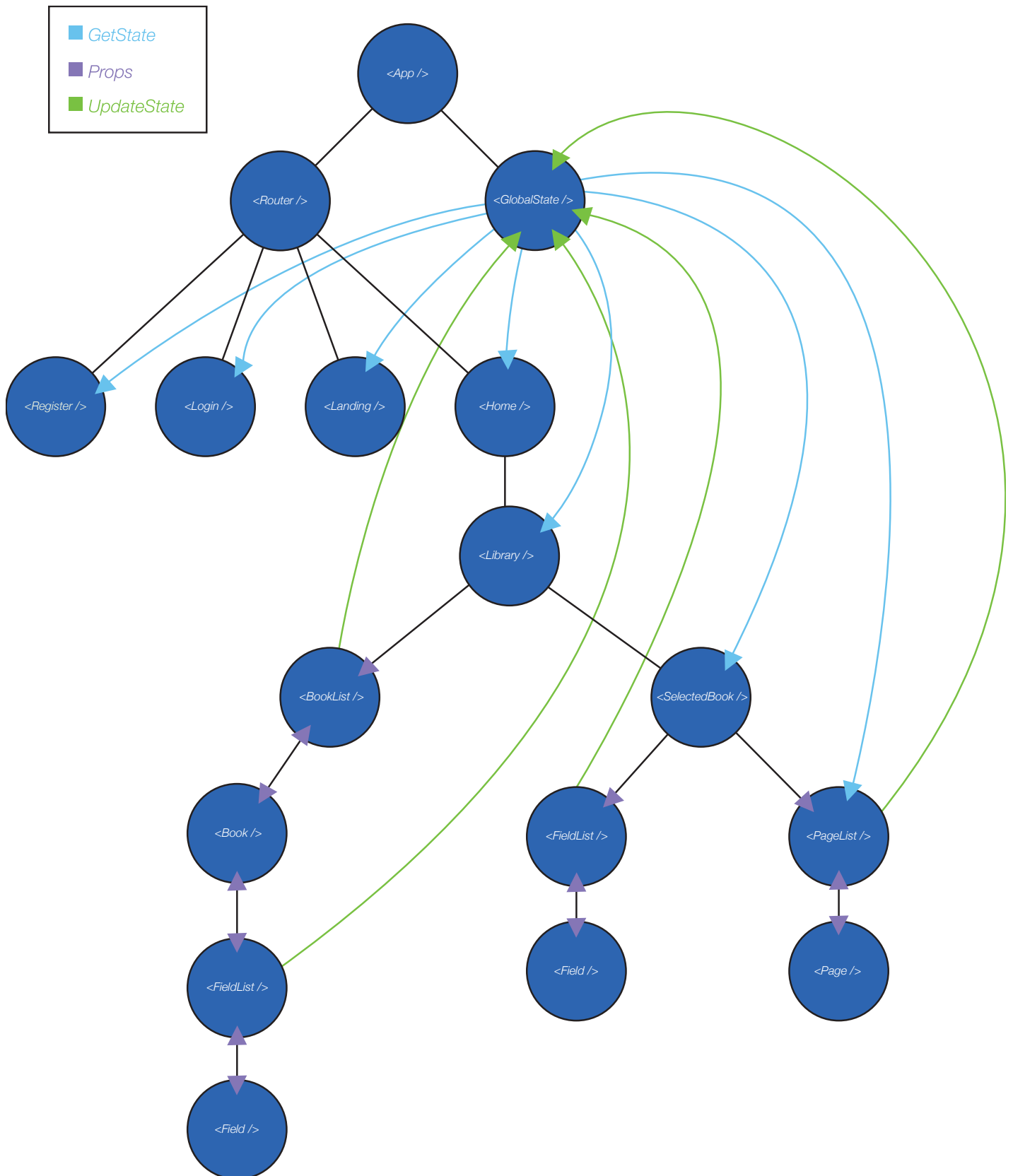
Por otra parte el servidor contará con un sistema de transpilado de archivos, que no será de mucha utilidad cuando estemos desarrollando la aplicación cliente.

## *Creación de la aplicación Cliente*

La aplicación cliente estará desarrollada enteramente en Javascript. Utilizaremos ReactJs para generar y manejar las vistas. Redux e ImmutableJs para el state global y PageJs para simular la navegación.

Para gestionar los estilos usaremos Materialize, una librería de google basada en el diseño material que nos provee de grillas, colores, modas...

## Arquitectura App Cliente





# Desarrollo de las fases

## *Estructura de carpetas*

Comenzaremos por crear la estructura de carpetas de todo el proyecto. Creamos una carpeta en la raíz del sistema llamada gestor. Esta será desde ahora la raíz del proyecto.

Dentro de ella crearemos 2 carpetas mas: Server y Client. La carpeta server contendrá los archivos del servidores rest, y la carpeta client contendrá tanto el servidor cliente como la aplicación cliente en si. Dentro de la carpeta server tendremos 4 mas.

Middleware contendrá una serie de funciones globales que nos serán de utilidad en varios puntos. Por el momento esta carpeta contendrá solo 2 archivos: checkToken.js y serializePassword.js. Estos archivos se encargaran de validar los tokens y encryptar la contraseña.

Models contendrá todos los modelos de mongoose para la interacción con la base de datos. Crearemos 5 archivos: Book, Field, Library, Page y User.

Dado que mongodb es una base de datos no relacionan la estructura es un poco diferente a los que estamos acostumbrados. Cada uno de estos modelos representará un documento en la base de datos. Un documento no es mas que un contenedor de registros definido por un schema. Todos estos modelos estarán relaciones por su ObjectId, un identificador único que crea mongo.

## *Instalación de dependencias*

Para instalar las dependencias bastara con abrir la consola del sistema en el que estemos desarrollando la aplicación y posicionarnos en la carpeta que contenga el archivo package.json y ejecutar el comando npm install. Esto creara una carpeta llamada node\_modules donde se guardaran todos los archivos de las dependencias.

## Base de Datos

Instalación y configuración de MongoDB. Instalamos la base de datos a nivel global en el ordenador. Configuración de un usuario client para monitorizar las peticiones recibidas y loquear los errores.

## Modelos en el Servidor

Creados los modelos necesarios para manejar la información de la base de datos. estos modelos están desarrollados bajo Mongoose. Cada modelo necesita un schema que define la estructura de datos que almacenaran. Mongoose nos provee de los métodos necesarios para interactuar con esta información, véase save, update, find, findById...

### Ejemplo modelo Library:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
module.exports = mongoose.model('Library', new Schema({
  user: {
    type: String,
    required: true
  },
  name: {
    type: String,
    required: true
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
  books: {
    type: Array,
    default: []
  }
}));
```

Creamos diferentes modelos para cada parte de la estructura de datos:

Library, Book, Field, Page y User.

Alguno de estos modelos cuentan con funciones llamadas Middleware para procesar la información antes de insertarla en la bd o al sacarla de ella.

Ejemplo de middleware para serializar la contraseña:

```
module.exports = function (password) {  
    return password.serialize();  
}
```

## Ruter Servidor

Definimos las rutas para las peticiones rest en un fichero llamado router.js. Este será el encargado de gestionar tanto las peticiones como los estados del usuario, validación de login...

Ejemplo de una ruta para las peticiones get de validación de tokens:

```
router.get('/validateToken', (req, res, next) => {  
    var tokenConfig = require(__root + 'tokenConfig');  
    //Comprobamos que existe el token  
    if (req.headers.token){  
        jwt.verify(req.headers.token, tokenConfig.secret, (err, decoded) => {  
            if(err){  
                console.log('--> Token Expirado (409)');  
                res.status(409).send("Token expirado!");  
                next();  
            }  
            console.log('--> Token Correcto! (200)');  
            res.status(200).send('ok');  
        });  
    }  
});
```

Cada ruta cuenta con su middleware para separar el código y que sea mas sostenible:

## Ejemplo middleware ruta get /documents:

```
router.get('/documents', checkToken, require('./document/get'));
```

Observamos 2 middleware encadenados en esta ruta. checkToken es llamado antes de ejecutar la petición en si. Recibe 3 parámetros (req, res, next), y se encarga de comprobar si la petición tiene permiso para proseguir. Si no tiene permiso devolverá un 401. Si es correcto, generara una variable en req llamada user que contendrá el id del usuario y llamara a next(), que ejecutara el siguiente middleware.

```
module.exports = function (req, res, next) {  
  
    var documentModel = require(__root + 'models/document');  
  
    documentModel.find({  
        user: req.user.id  
    }, 'name fields', (err, documents) => {  
  
        if(err) return res.status(500).send(err)  
  
        res.status(200).send(documents);  
  
    });  
  
}
```

Este middleware de encargara de buscar en la base de datos todos los documentos del usuario, paseado atreves del token y guardado en req.user en el middleware anterior.

Si hay algún error devolverá 500 y si todo es correcto devolverá 200 y un json con todos los documentos que posee el usuario.

## Cliente

Configuramos el archivo server.js, que se encargara de levantar el servidor y servir todos los archivos estáticos existentes en /public.

```

var express = require('express');
var server = express();
server.use(express.static('public'));
server.get('*', (req, res) => {
    res.sendFile(__dirname + '/public/index.html');
});
server.listen(3000, function() {
    console.log('Listening on port 3000...');
});

```

## *package.js*

En este archivo se define todo el proyecto. Por un lado tenemos las dependencias del proyecto en modo producción:

```

"dependencies": {
  "express": "^4.15.2",
  "immutable": "^3.8.1",
  "page": "^1.7.1",
  "react": "^15.5.4",
  "react-dom": "^15.5.4",
  "react-redux": "^5.0.5",
  "redux": "^3.6.0"
}

```

Y en modo desarrollo:

```

"devDependencies": {
  "babel-preset-es2015": "^6.24.1",
  "babel-preset-react": "^6.24.1",
  "babelify": "^7.3.0",
  "browserify": "^14.3.0",
  "gulp": "^3.9.1",
  "gulp-util": "^3.0.8",
  "vinyl-source-stream": "^1.1.0",
  "watchify": "^3.9.0"
}

```

Este archivo también define los comandos básicos para manipular el proyecto y definir su author, licencias, nombre...

```
"name": "gestor-client",
"version": "1.0.0",
"description": "",
"main": "app.js",
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "gulp compile & node server.js"
},
"author": "",
"license": "ISC",
```

Este archivo lo contendrá tanto el servidor de rest como el servidor cliente.

### *gulpfile.js*

Este es el archivo encargado de transpirar el código de es6,7,8 a es5. Lee el archivo de entra app.js y todas sus dependencias y crea un nuevo archivo en public con el mismo nombre que será en que se sirve al cliente.

## Ejemplo de un task de gulp para el compilado:

```
gulp.task('compile', function () {  
  //Definimos los archivos  
  var bundle = browserify({  
    entries: './app/app.js',  
    debug: true  
  });  
  //Compilamos la primera vez  
  compile(bundle);  
  //Observamos el archivos cuando cambie  
  watchify(bundle).on('update', function () {  
    compile(bundle);  
  });  
});  
  
/Compilamos a es5  
function compile (bundle) {  
  console.log('Compilando ...');  
  console.time('Compilado!')  
  bundle  
    .transform("babelify", {presets: ["es2015", "react"]})  
    .bundle()  
    .on('error', gutil.log)  
    .on('end', () => {console.timeEnd('Compilado!');})  
    .pipe(source('app.js'))  
    .pipe(gulp.dest('./public/'))  
}
```

Mediante `console.time('Compilado!')` controlado el tiempo que tarda en realizar la transpiración. Destacar que las primeras transpilaciones las realiza en torno a los 3000ms pero a medida que vamos realizando cambios este tiempo se incrementa exponencialmente hasta llegar a ser inviable, lo que provoca tener que reiniciar el servidor a menudo mientras desarrollamos. Aun no se el motivo.

*/app*

La raíz de archivos del cliente. Aquí tenemos un archivo principal llamado `app.js` que es el componente raíz de todo el proyecto. Es el encargado de renderizar los demás componentes y cargar el state global de toda la aplicación.

## Actions

Aquí están definidas todas las acciones a través de las cuales se actualizará el state global de la aplicación. Está definido en 4 secciones: Create, Edit, Remove y Select.

Una acción consiste en una función plana que recibe una serie de parámetros y genera un mensaje en formato flux. Estos mensajes no son más que un objeto json puro. Cada mensaje necesita tener 2 atributos: type y data.

Type define el tipo de acción a realizar y data es la información referente a dicho mensaje.

Ejemplo de Action para crear un nuevo libro:

```
import GlobalStore from '../GlobalStore';
const CreateBook = (book) => {
  GlobalStore.dispatch({
    type: 'CREATE_BOOK',
    data: {
      book: book
    }
  });
}
export default CreateBook;
```

Mediante el método dispatch de GlobalStore se envía esta acción a los reducers que decidirán que hacer con el.

## Reducers

Un reducer no es más que una función pura que recibe 2 parámetros: state y action y genera un nuevo state.

Para hacer una separación lógica de todos los reducers de la aplicación estos están divididos en varios archivos.



Por una parte tenemos una carpeta que define las acciones que competen a dicho reducir (Create, Edit, Remove y Select).

Cada una de estas carpetas contiene un reducer con el mismo nombre de la carpeta que engloba todos los demás. Este reducer no es mas que una función que recibe state y catión, y mediante un switch llama a sus hijos y le envía la información necesaria para actualizar el state.

### Ejemplo de reducer puro:

```
import Immutable from 'immutable';

const CreateBook = (books = Immutable.List(), action) => {
  if(action.type === 'CREATE_BOOK'){
    return books.push(action.data.book);
  }
  return books;
}

export default CreateBook;
```

## Ejemplo de reducer padre:

```
import CreateBook from './CreateBook';
import CreateField from './CreateField';
import CreatePage from './CreatePage';
const Create = (books = Immutable.List(), action) => {
  switch (action.type) {
    case 'CREATE_BOOK':
      return CreateBook(books, action);
    break;
    case 'CREATE_FIELD':
      let newFields = CreateField(books.get(action.data.bookIndex).fields, action);
      books.get(action.data.bookIndex).fields = newFields;
      return books;
    break;
    case 'CREATE_PAGE':
      let newPages = CreatePage(books.get(action.data.bookIndex).pages, action);
      books.get(action.data.bookIndex).pages = newPages;
      return books;
    break;
    default:
      return books;
  }
}
export default Create;
```

Todos estos reducer se agrupan en uno final llamado RootReducer, que es el encargado de guiar la información hacia sus hijos, y mediante la función `combineReducers()` juntar todos en uno.

## RootReducer:

```
import { combineReducers } from 'redux';
import Immutable from 'immutable';
import Create from './Create/Create';
import Edit from './Edit/Edit';
import Remove from './Remove/Remove';
import SelectBook from './Select/SelectBook';
import SelectField from './Select/SelectField';
import SelectPage from './Select/SelectPage';

const RootReducer = combineReducers({
  books: (books = Immutable.List(), action) => {
    switch (action.type) {
      case 'CREATE_BOOK':
      case 'CREATE_FIELD':
      case 'CREATE_PAGE':
        return Create (books, action);
      break;
      case 'EDIT_BOOK':
      case 'EDIT_FIELD':
      case 'EDIT_PAGE':
        return Edit (books, action);
      break;
      case 'REMOVE_BOOK':
      case 'REMOVE_FIELD':
      case 'REMOVE_PAGE':
        return Remove (books, action);
      break;
      default:
        return books;
    }
  },
  selectedBook: (selectedBook = null, action) => {
    if(action.type === 'SELECT_BOOK') {
      return SelectBook (selectedBook = null, action);
    }
    return selectedBook;
  },
  selectedField: (selectedField = null, action) => {
    if(action.type === 'SELECT_FIELD'){
      return SelectField (selectedField = null, action);
    }
  }
});
```

```

        return selectedField;
    },
    selectedPage: (selectedPage = null, action) => {
        if(action.type === 'SELECT_PAGE') {
            return SelectPage (selectedPage = null, action)
        }
        return selectedPage;
    }
});

export default RootReducer;

```

## GlobalStore.js

En este punto es donde se une toda la parte del estado global. Este archivo se encarga de crear el state inicial de la aplicación, inyectar los reducers y exportarlo para hacerlo accesible a toda la aplicación.

```

import { createStore } from 'redux';
import RootReducer from './Reducers/RootReducer';

const store = createStore(RootReducer);

export default store;

```

## Components

En esta carpeta se definen todos los componentes referentes a la aplicación. Diferenciamos 3 tipos: Router, Componente Lógico, Componente Visualización.

### <Router />

En primer lugar el router se encarga de controlar las rutas para que el navegador no tenga que refrescar la página y simular una navegación.

Este componente no renderiza nada, pero cuando es montado ( `componentWillMount()` ) configura la navegación gracias a la librería `page.js`.

## Ejemplo de rutas en el cliente:

```
Page('/', () => {  
    self.props.navigate(<Landing></Landing>);  
});  
Page('/login', () => {  
    self.props.navigate(<Login></Login>);  
});  
Page('/register', () => {  
    self.props.navigate(<Register></Register>);  
});  
Page('/home', isLogged, () => {  
    self.props.navigate(<Library></Library>);  
});  
Page('/home/*', isLogged, () => {  
    self.props.navigate(<Library></Library>);  
});  
Page('/*', () => {  
    self.props.navigate(<NotFound></NotFound>);  
});  
Page();
```

Cada ruta no es mas que un storing que define el path y una función que devuelve un componente.

## Componentes Lógicos

Un componente lógico no es mas que una conexión del state global con sus hijos.

Para ello tenemos 2 funciones: `mapDispatchToProps()` y `mapStateToProps()`.

`mapDispatchToProps()` se encarga de enviar los mensaje al estado para que los reducers hagan su función. Esta no sera necesaria en nuestra aplicación dado que las acciones se encargan de enviarse a si mismas.

`mapStateToProps()` se encarga de conectar una parte del estado con las props del componente. Esto es fundamental para que cuando el state se actualiza, los props necesarios se re-rendericen con la nueva información. Además sirve para delimitar que parte del state es visible en cada componente.

### Ejemplo de componente logico <Library />

```
import React from 'react';
import { connect } from 'react-redux';
import BookList from './BookList';
import SelectedBook from './SelectedBook';
import DropDownBooks from './DropDownBooks';
import DropDownFields from './DropDownFields';

class Library extends React.Component {

  constructor(props) {
    super(props);
  }

  render () {

    return (<div className="row">
      <div className="col s3">
        <DropDownBooks />
        <BookList />
        {
          this.props.state.selectedBook ? <DropDownFields /> : null
        }
      </div>
    </div>);
  }
}
```

```

        {
            this.props.state.selectedBook ? <SelectedBook /> : null
        }
    </div>
    <div className="col s9">

        </div>
    </div>
)

}

}

const mapStateToProps = (state) => {
    return {
        state: state
    }
}

export default connect(mapStateToProps)(Library);

```

## Componente Visualización

Estos componentes se encargan unidamente de renderizar lo que reciben por las props. Estos componentes también contienen los eventos del usuario.

### Ejemplo de componente de visualización

```
import React from 'react';
import { connect } from 'react-redux';
import SelectField from '../Actions/Select/SelectField';

class Field extends React.Component {

  constructor(props) {
    super(props);

    this.selectField = this.selectField.bind(this);
  }

  selectField (e) {
    e.preventDefault();
    e.stopPropagation();
    SelectField (this.props.id, this.props.selectedBook.bookIndex);
  }

  render () {

    let classNames = 'collection-item';
    if(this.props.selectedField && this.props.selectedField.fieldIndex == this.props.id) {
      classNames += ' active';
    }

    return (<li className={classNames} onClick={this.selectField}>{this.props.name}</li>);

  }

}

const mapStateToProps = (state) => {
  return {
    selectedField: state.selectedField,
```



```

        selectedBook: state.selectedBook
      }
    }
  }

  export default connect(mapStateToProps)(Field);

```

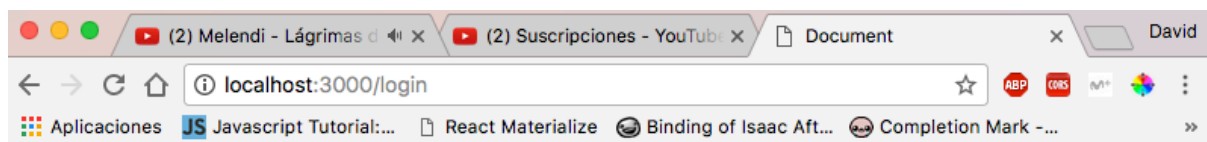
Notese que estos componentes son los encargados, en la mayoría de casos de crear las acciones. Por ejemplo este componente en concreto crear la acción seleccionar campo cuando se hace click en el.

## Css

Para manejar los estilos de la aplicación usaremos Materializze. Una librería parecida a bootstrap basada en el diseño Material propuesto por google. Se utiliza tanto para los estilos generales como para los botones, dropdowns, modal...

## Formulario Login

Vista para realizar el login en la aplicación.



# Login



Nombre



Contraseña

ACCEDER

REGISTRAR

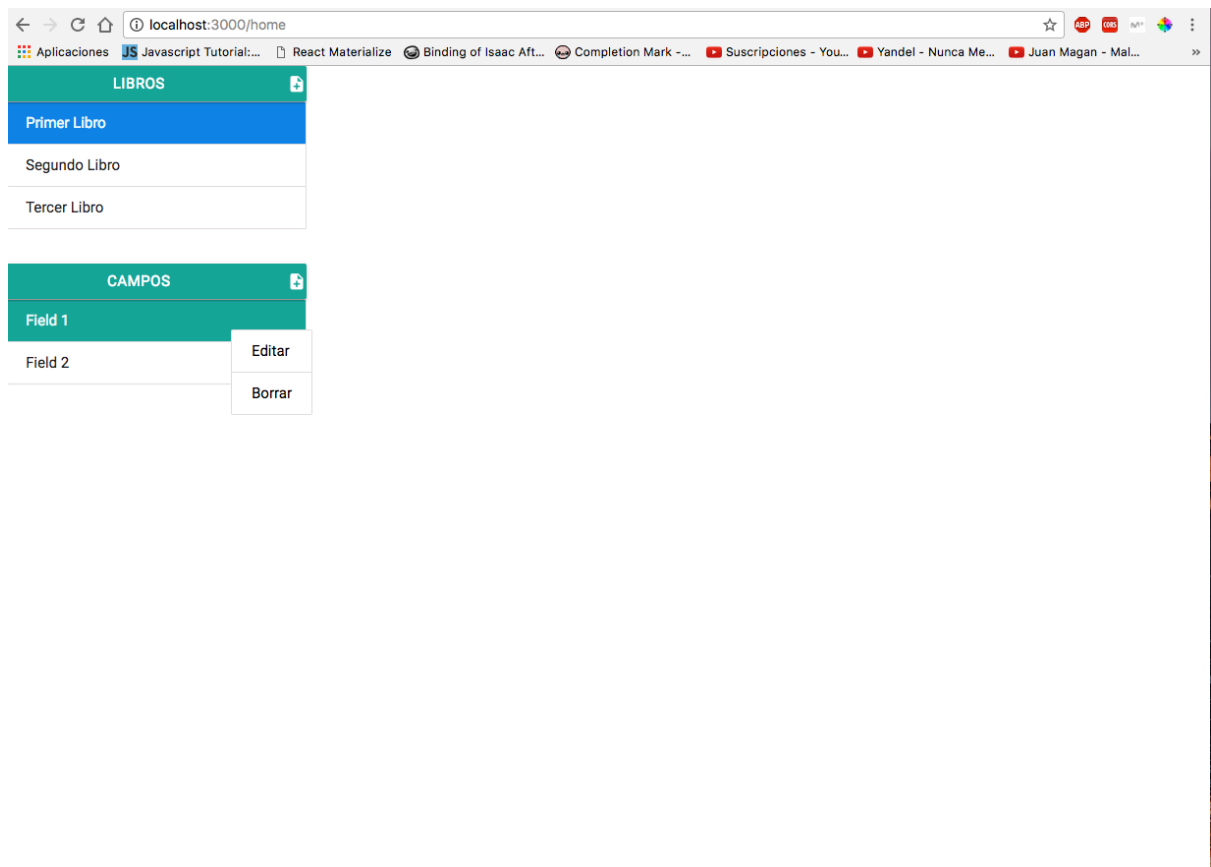
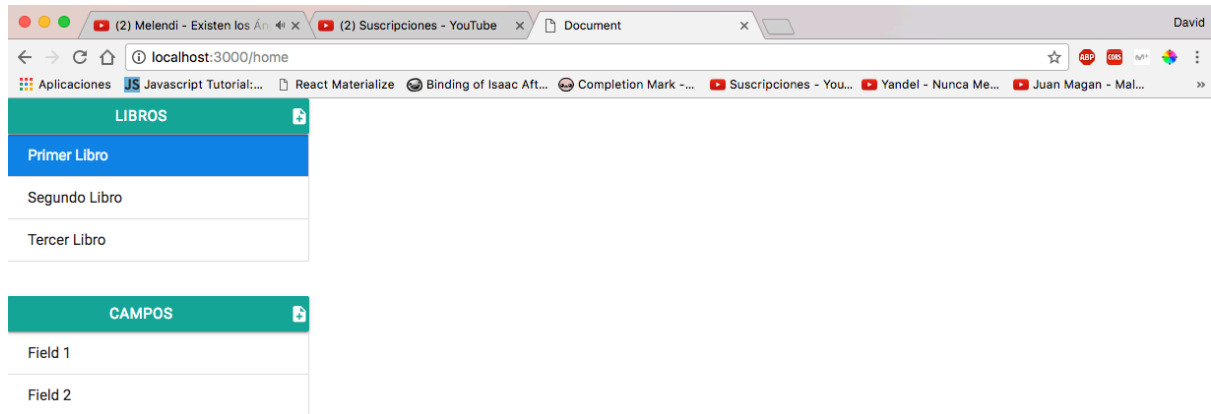
## Formulario Registro Usuario

The screenshot shows a web browser window with the address bar displaying `localhost:3000/register`. The page title is "Registro". The form consists of five input fields, each with an icon to its left: a person icon for "Nombre", another person icon for "Apellidos", an envelope icon for "Email", a padlock icon for "Contraseña", and another padlock icon for "Repita la Contraseña". Below the fields are two teal buttons labeled "REGISTRAR" and "ACCEDER". A gray rectangular box with the text "350 x 150" is positioned to the right of the form fields.

Vista para realizar el registro de usuarios

# Home

Vista de la librería del usuario.



# 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/immutable-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 Elastic 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...