ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

Aplicación de administración con Angular, Node y Express para una aplicación Django

Administration application with Angular, Node and Express for a Django application

Realizado por
Rubén Carreño Villalba
Tutorizado por
Antonio Jesús Nebro Urbaneja
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, Diciembre 2015

Fecha defensa:
El Secretario del Tribunal

Resumen: En este trabajo de fin de grado se ha desarrollado una aplicación de administración que sustituye a las que ofrecen por defecto las aplicaciones creadas con el framework de desarrollo web Django. La aplicación está compuesta por dos partes: un servidor, desarrollado con Node y Express, que ataca a la base de datos MySQL de la aplicación Django (es el nexo de unión entre ambas), y expone una API que es utilizada por la otra parte que compone la aplicación, la parte del cliente. La API es totalmente privada, siendo necesario un token de autenticación válido para poder obtener una respuesta satisfactoria de la misma. La generación del token también es tarea del servidor. El cliente, que es la parte que ve el usuario final, está desarrollada usando el framework Angular. La interfaz de usuario utiliza Bootstrap, por lo que su visualización es correcta en cualquier tipo de dispositivo, tanto de escritorio como móvil. En definitiva, se ha desarrollado una aplicación JavaScript End-to-End, empleando las últimas tecnologías web, mejorando ostensiblemente, las prestaciones que ofrece un panel de administración generado automáticamente por una aplicación Django.

Palabras claves: Django, Angular, Node, Express, MEAN, Panel de Administración, CRUD, MySQL

Abstract: In this project, it has been developed a dashboard that can replace dashboards that are provided by default from an usual web application developed using Django framework. This application has two parts: the first one is the server side, made using Node and Express, that uses the MySQL data base of the Django application (it is the link between the Django application and the dashboard), and it exposes an API that is used by the other part, the client side. The API is totally private, so it is necessary a valid authorization token to get a successful response. This token is also generated in the server (using a non private endpoint). The client side, that is the interface that an user sees, has been developed using the Angular framework. This user interface uses Bootstrap, so it can be visualized perfectly on personal computers or mobile devices. Definitely, it has been made an entire JavaScript end-to-end application, using the latest web technologies, and getting a better approach than a simple Django application could generate.

Keywords: Django, Angular, Node, Express, MEAN, Dashboard, CRUD, MySQL.

ÍNDICE

CAPÍTULO 1

INTRODUCCIÓN

En este primer capítulo se describirán brevemente los aspectos generales del proyecto. En la primera parte se explican los factores que han motivado a desarrollar el proyecto, mientras que durante la segunda parte se hablará de los objetivos que se pretenden cubrir. Tras esto, se mencionarán los medios materiales necesarios para el desarrollo del sistema completo, para finalizar con un pequeño resumen del resto de la memoria.

1.1 Motivación

El mundo de las aplicaciones web es un mundo de constante evolución y actualización, donde aparecen nuevas tecnológicas que ofrecen tanto mejoras visuales que mejoran la experiencia del usuario como de rendimiento.

Este tipo de tecnologías abren un abanico inmenso de posibilidades, y hacen pensar a los analistas y desarrolladores de aplicaciones web que puede haber otras soluciones que mejoren su aplicación, pero que no estaban disponibles cuando definieron la arquitectura y el diseño de sus aplicaciones.

Las aplicaciones desarrolladas con el framework Django tienen la ventaja de que ofrecen una zona de administración que ofrece las acciones básicas para crear, editar y eliminar entidades. El problema radica en que tanto la aplicación principal como la de administración corren bajo el mismo proceso, por lo que una caída del mismo inutilizaría ambas partes. Además la zona de administración no está preparada para su correcta visualización en dispositivos móviles, por lo que dificulta su uso a no ser que el administrador tenga un ordenador a mano.

En este trabajo fin de grado se plantea la realización de una nueva zona de administración de una aplicación Django (PFC de I.T.I. de Sistemas) con las últimas tecnologías web, con el fin de desacoplar la aplicación principal de ésta, haciendo que ambas compartan sólo la base de datos, mejorando la experiencia del usuario tanto en su utilización en ordenadores personales como en dispositivos móviles.

1.2 Objetivo

El objetivo principal de este trabajo fin de grado es el diseño e implementación de una aplicación web, realizada con Node.js y Express para la parte del

servidor, y AngularJS y Bootstrap para la parte del cliente, que permita gestionar la parte de administración de una aplicación web Django, donde se podrán crear, editar, eliminar y consultar las distintas entidades que forman la base de dicha aplicación.

Como se ha comentado líneas atrás, la aplicación tiene 2 zonas bien diferenciadas, el servidor, que es el encargado de exponer una API para el acceso y modificación de datos, y aplicación de usuario, que hace uso de los datos servidos desde la API y los muestra al usuario de una forma amigable.

Ambas partes utilizan tecnologías diferentes, pero con una misma base: JavaScript, utilizando para el intercambio de datos JSON (*JavaScript Object Notation*). La base de datos viene heredada del PFC que amplía este trabajo fin de grado, utilizándose MySQL. Todas estas tecnologías serán explicadas con más detalles en capítulos posteriores.

El objetivo secundario de este trabajo fin de grado es poder facilitar a los administradores de la aplicación Django el uso de la zona de administración en dispositivos móviles.

Con todo esto se propone una evolución en la administración de una aplicación Django, desacoplando el panel de administración de la aplicación Django en sí, aprovechando todas las virtudes que nos ofrecen las nuevas tecnologías, mejorando ostensiblemente su usabilidad.

1.3 Medios materiales

Para realizar el proyecto se han utilizado los siguientes recursos materiales:

Recursos hardware

- Macbook Pro con procesador Intel i7 de doble núcleo a 2,9 GHZ y 8 GB SDRAM DDR3

Recursos software

- Atom como IDE para el desarrollo de la aplicación.
- NodeJs
- ExpressJs
- AngularJs
- Sass
- Bootstrap

1.4 Organización de la memoria

El resto de la memoria se estructura de la siguiente forma:

- *Capítulo 2 – Herramientas utilizadas*. En este capítulo se comentarán las herramientas que se han utilizado para el desarrollo del proyecto, como lenguajes de programación y frameworks.

- *Capítulo 3 – Análisis y arquitectura del sistema*. En este capítulo describe el análisis previo a su desarrollo, y se aportarán diagramas para definir la arquitectura del sistema en general, y de las subaplicaciones en particular.

- *Capítulo 4 – Implementación y diseño de la interfaz*. Se mostraran detalles de la implementación y las diferentes zonas que componen la parte visible de la aplicación.

- *Capítulo 5 – Conclusiones y líneas futuras*. En este capítulo se comentan las conclusiones extraídas del trabajo realizado en el proyecto. Además, se orienta al lector a cuales podrían ser las líneas de trabajo sobre las que podría desarrollarse un trabajo futuro.

- Referencias y bibliografía.

- *Anexo A – Estructura de las aplicaciones*. Se proporciona información detallada sobre la estructura de carpetas y su contenido para una mejor comprensión.

- *Anexo B – Contenido del CD*. En este anexo se explica cuál es el contenido del CD que se entrega de manera adjunta

CAPÍTULO 2

HERRAMIENTAS UTILIZADAS

En este capítulo se describen las herramientas utilizadas para el desarrollo del sistema. Estas descripciones servirán para poner al lector en contexto y dar a conocer algunas de las tecnologías más modernas y poco conocidas que se han usado este proyecto.

Se evitará hablar de tecnologías de sobra conocidas, que son la base de las que describirán, como JavaScript, HTML, CSS o MySQL. Hay muchísima información de estas tecnologías disponibles en diferentes medios, por lo que no se aburrirá al lector con 'parrafadas' sobre ellas que no aportan nada a la comprensión y visualización de este proyecto.

2.1 Entorno de desarrollo – Atom

Es un editor de texto de código abierto desarrollado por *GitHub* disponible para múltiples plataformas como OS X, Linux y Windows. Está desarrollada utilizando tecnologías web, pudiéndose ampliar sus funcionalidades a través de plugins desarrollados con Node.js que pueden ser instalados fácilmente a través de su gestor de paquetes. Esta posibilidad hace que se convierta en un editor de texto muy personal, ya que cada desarrollador elige las características que desea tener en su IDE sin afectar un ápice a su rendimiento.

2.2 Node.js

Se trata de una tecnología relativamente nueva, que fue creada en 2009 por Ryan Dahl. Es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript*[1]*,

asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web.

Node.js*[2]* incorpora una serie de módulos básicos, como un módulo de red, que proporciona una capa para programación de red asíncrona y otros módulos fundamentales, como por ejemplo *Path, FileSystem, Buffer, Timers* y el de propósito más general *Stream*. Ofrece también la posibilidad de utilizar módulos de terceros a través del *Node Package Manger* (*npm*), que permite instalar nuevos módulos de manera global para todas las aplicaciones node del sistema, o como dependencias locales a cada aplicación. Las dependencias locales hacen que las aplicaciones sean portables, además de dejar limpio el directorio de módulos general.

2.2.1 Módulos Node adicionales utilizados

Algunos de los paquetes node adicionales, utilizados para la elaboración de este proyecto son:

- *mysql y sequelize*: Utilidades necesarias para la conexión a la base de datos MySQL, facilitando el acceso a los datos y su modificación.
- *multer*: Para la gestión en el servidor de ficheros binarios.
- *jsonwebtoken*: Generación y verificación de tokens de autenticación.
- *express*-jwt: Middleware para utilizar jwt (se explicara más adelante) con express.
- *pbkdf2*-sha256: para la encriptación de claves de usuario.

Todos estos módulos son dependencias locales al proyecto.

2.3 Express.js

Express.js[3] es un framework web mínimo y flexible para Node.js (siendo el estándar de facto para servidores web de Node.js) que proporciona un conjunto robusto de características para aplicaciones web y móviles. Express proporciona una capa delgada de características fundamentales de aplicaciones web. Facilita la creación de API's gracias la gran variedad de métodos HTTP y middleware que proporciona.

2.4 Angular.js

Es un framework *MVC*[4] de JavaScript para el desarrollo web que permite crear *Single-Page Applications (SPA)*, que se caracterizan por dar la impresión al usuario de que todo sucede en la mima página, sin hacer recargas de la misma.

Angular.js[5] fue creado inicialmente por Google en 2009, pero en la actualidad, además cuenta con una amplía comunidad de desarrolladores dedicados a su mantenimiento y nuevas versiones.

Dentro de una aplicación Angular podemos diferenciar una serie de elementos claves que conforman la base de este framework:

- *Servicios:* son los encargados de proveer los datos. Lo más normal es que estos datos provengan de una API externa.
- *Vistas:* es la parte visible por el usuario. Suelen estar parametrizadas, y todas las vistan tienen asociado un controlador.
- *Controladores:* Sirven los datos y funcionalidades a las vistas asociadas a él. Suelen ser estos los que utilizan los servicios para obtener los datos.
- *Directivas:* Ofrecen elementos nuevos o nuevos comportamientos en elementos ya existentes dentro de las vistas (widgets, validaciones, etc.).

Todas estas partes están íntimamente relacionadas, siendo realmente útil y efectivo en el sistema de inyección de dependencias que ofrece Angular para su correcta conexión.

2.4.1 Módulos adicionales

Angular ofrece la posibilidad de añadir nuevos módulos a su core a través de otros módulos angular, que se pueden utilizar gracias a su sistema de inyección de dependencias mencionado anteriormente. En este proyecto se han utilizado varios módulos adicionales:

- *UI-router:* Sistema de enrutado basado en estados.
- *UI-Bootstrap*: Ofrece múltiples widgets de gran utilidad, como datepicker, timepicker o modal panels.

- *LocalStorage*: Ofrece un servicio para utilizar el almacenamiento local de los navegadores.

- *Toastr*: Ofrece funcionalidades para mostrar mensajes informativos al usuario.

2.5 Bootstrap

Se trata de un framework desarrollado por Twitter en 2011, que contiene plantillas de diseño con tipografías, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basados en HTML y CSS.

Bootstrap esta orientado a facilitar el diseño y elaboración de aplicaciones web utilizando un sistema de cuadrículas que ayuda a conseguir interfaces claras y 'responsives', haciendo que las aplicaciones se vean con diferentes diseños en diferentes dispositivos, dependiendo del espacio del que disponga la aplicación (ordenadores personales, tablets, móviles, etc.), por lo que el programador solo tiene que preocuparse de utilizar las clases css correctamente, olvidándose prácticamente de los quebraderos de cabeza que pueden proporcionar las hojas de estilo.

2.6 Sass

Es un preprocesador CSS que facilita la creación y el mantenimiento de hojas de estilos. Ofrece la posibilidad de crear variables, funciones o la inclusión de otras hojas de estilo Sass, por lo que normalmente, las hojas de estilo Sass suelen ser mas limpias y fácil de entender (y por tanto de mantener) que hojas de estilo convencionales. Cuando son incluidas en la web, es necesario hacer una transformación a hoja de estilos CSS.

Todas estas tecnologías dejan de manifiesto que se ha creado una aplicación JavaScript End-to-End, utilizando las últimas tecnologías en el desarrollo web, tanto en la parte del servidor como en la parte del cliente, que se comunican con objetos que entienden cada una de ellas (JSON), con el fin de ofrecer al usuario una experiencia caracterizada por la rapidez y un diseño claro e intuitivo.

CAPÍTULO 3

ANÁLISIS Y ARQUITECTURA DEL SISTEMA

En este capítulo se expondrá el análisis realizado para la implementación de la aplicación, así como la arquitectura de la misma, ofreciendo diagramas que faciliten la comprensión de la misma.

3.1 Análisis del sistema

Los requisitos del sistema son relativamente claros: una nueva interfaz de administración nueva independiente del proceso Django. Ante este panorama, se piensa en una aplicación End-to-End que ataque a la misma base de datos de la que se sustenta la aplicación Django.

Lo primero que se plantea es la necesidad de recuperar los datos de la base de datos. Lo segundo es mejorar tanto la parte de administración básica de una aplicación Django tanto visualmente como en rendimiento, y lo tercero hacer que su manejo en dispositivos móviles se realice de manera optima.

Analizando todas estas partes, tanto por separado como juntas, se decide utilizar las últimas tecnologías web, para hacer una aplicación JavaScript End-to-End, ya que se satisfice cada una de las necesidades planteadas:

1. Utilizar Node.js y Express para la parte del servidor. El servidor será el encargado de recuperar los datos de la base de datos, gestionar la autenticación (contra la misma base de datos) y exponer una API para que la aplicación del cliente pueda acceder y modificar los datos.

2. Utilizar AngularJs para la parte del cliente. Angular puede comunicarse fácilmente con el servidor, hablando el mismo idioma (JSON), ofreciendo una experiencia de usuario inmejorable implementando una Simple-Page Application.

3. Utilizar Bootstrap para crear una aplicación responsive. Bootstrap ofrece muchas utilidades para conseguir que las aplicaciones web puedan verse de forma correcta tanto en diferentes dispositivos como en diferentes resoluciones.

En capítulos posteriores se darán más datos de algunos detalles de implementación interesantes en cada uno de los casos.

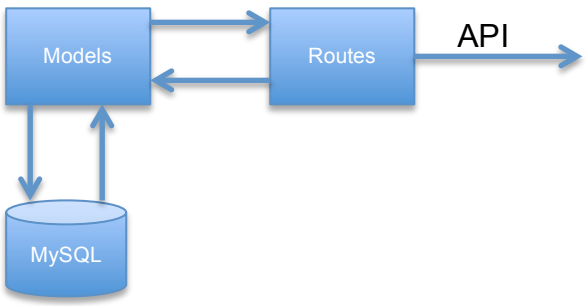
3.2 Arquitectura del sistema

Como se ha comentado en secciones anteriores la aplicación tiene dos sistemas bien diferenciados: servidor y cliente. En esta sección se expondrán las arquitecturas de cada una de ellas así como de la aplicación compete.

3.2.1 Arquitectura del servidor

En el servidor se distinguen principalmente 2 módulos: el módulo donde se definen los modelos correspondientes con la base de datos (*models* en adelante), y el modulo donde se definen los puntos de acceso al servidor para hacer consultas, autenticarse o modificar, crear o eliminar datos (*routes* en adelante).

El modulo models interactúa directamente con la base datos, gracias al paquete node *sequelize,* y el módulo de routes hace uso del módulo models para cubrir las necesidades de las peticiones al servidor. Todas las rutas del sistema ofrecen una interfaz CRUD[6] (Create-Read-Update-Delete).



3.2.2 Arquitectura cliente

En la parte de cliente de la aplicación podemos diferenciar claramente 5 partes: servicios, controladores, directivas, vistas y rutas. A continuación se detallaran las funciones que cumplen cada una de ellas y como interactúan entre si.

- La capa de ***servicios*** es la que se encarga de canalizar los datos entre la parte del servidor y el cliente. Lo hace en ambos sentidos, es decir, se ocupa de recuperar datos del servidor (normalmente con peticiones GET), de enviar nuevos datos desde el cliente al servidor (creación, modificación o borrado de diferentes entidades), y de utilizar los diferentes puntos de acceso que dispone el servidor para la gestión de la autenticación y el acceso seguro a la API del sistema.

- Las ***vistas*** (también llamadas views o partials) son las encargadas de presentar los datos al usuario de una forma amigable. Están escritas en

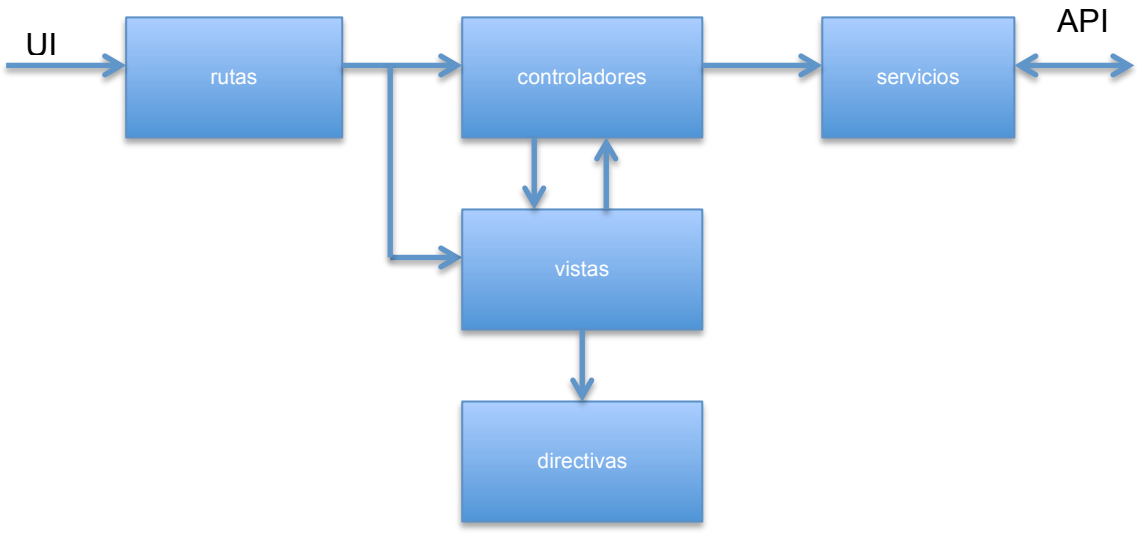
HTML, y hacen usos de distintas clases definidas en Bootstrap para su correcta visualización en diferentes dispositivos y resoluciones.

- Los ***controladores*** o controllers son los encargados de proveer datos a las vistas de una forma en la que estas los entiendan. Se puede decir que son unos intermediarios entre los servicios y las vistas. Por un lado, obtienen datos a través de los servicios, hacen las transformaciones necesarias sobre estos, y los envían a las vistas para que estas los 'pinten'. También son en el intermediario en la otra dirección. En muchas ocasiones las vistas no solo muestran datos, si no que también son las encargadas de ser el medio a través del cual un usuario puede modificar los mismos o crear nuevos datos. Para ello, las vistas envían los datos a los controladores, y estos los mandan a los servicios (con las transformaciones oportunas) para que puedan llegar hasta el servidor.

- Las ***directivas*** añaden nuevas funcionalidades o elementos nuevos a las vistas, extendiendo los elementos básicos HTML.

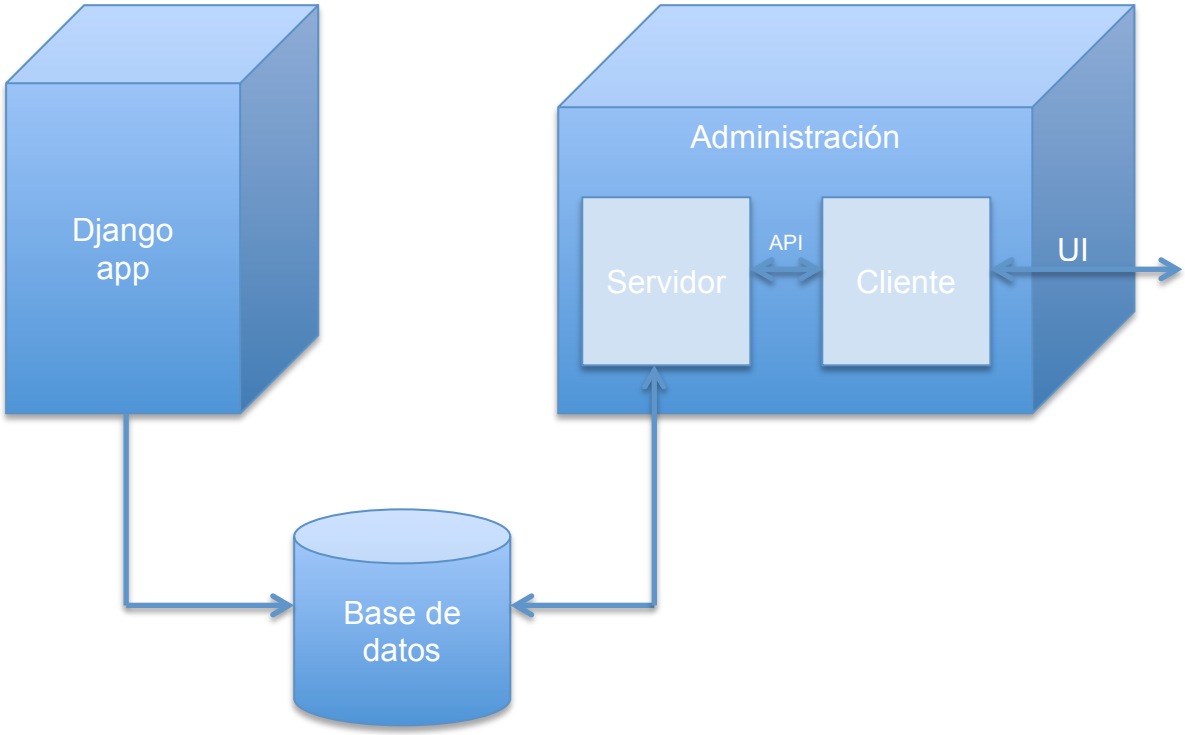
- Las ***rutas*** definen los puntos de acceso del usuario a la aplicación. Cada ruta tiene asociada una vista y un controlador, que hace que estos se relacionen entre sí.

A continuación se escenifica de forma gráfica como interactúan entre sí de las diferentes partes anteriormente descritas:



3.2.3 Arquitectura general

A continuación se mostrará un esquema que presentará la interacción entre las distintas partes de la aplicación, incluyendo la aplicación Django.



CAPÍTULO 4

IMPLEMENTACIÓN E INTERFAZ DE USUARIO

En este capítulo se mostrarán diversos detalles de implementación que el autor piensa que son interesantes y que pueden ayudar a la comprensión de ciertos detalles claves dentro de la aplicación, así como una descripción de la interfaz de usuario.

4.1 Detalles de implementación

A continuación se detallaran algunos detalles relevantes de la implementación del sistema. Se dividirá en 2 secciones, correspondientes a las 2 partes principales del sistema, con el fin de mejorar su claridad y ayudar a la comprensión.

4.1.1 Servidor

En esta sección se mostrará como son los modelos correspondientes a las tablas de la base de datos y de como se registra, algunos detalles de la autenticación y de como se registran las diferentes rutas que son accesibles desde la parte del cliente (API).

Modelos

La conexión con la base de datos (que ya existía con anterioridad) es una de las partes fundamentales de la aplicación, ya que la base de datos es el nexo entre la aplicación Django y la nueva interfaz de administración.

Para realizar esta conexión se utiliza el paquete node '*sequelize'* [7]. Para poder registrar los modelos es necesario crear un modelo por cada tabla existente en la base datos, creándose un ORM [8] que facilita la manipulación y búsqueda de los objetos. La correspondencia entre un modelo y una tabla de la base de datos sería de la siguiente manera:

- Cada tabla se corresponde con un modelo
- Cada instancia de ese modelo u objeto se corresponde con una fila de la tabla.
- Cada atributo o propiedad del objeto se corresponde con una columna de la tabla.

La comunicación con las tablas a través de estos modelos se realiza de forma asíncrona, algo básico en este tipo de aplicaciones. En cada modelo hay que definir el tipo y el nombre de cada columna de la tabla, así como las relaciones existentes con el resto de modelos. En la siguiente imagen se muestra un ejemplo de definición de modelo:

```
module.exports = function(sequelize, DataTypes) {
  var Actividad = sequelize.define("Actividad", {
    titulo: {
      type: DataTypes.STRING(100),
      field: 'titulo'
    },
    descripcion: {
      type: DataTypes.TEXT,
      field: 'descripcion'
    },
    imagen: {
      type: DataTypes.STRING(100),
      field: 'imagen'
    },
    contenido: {
      type: DataTypes.TEXT,
      field: 'contenido'
    }
  }, {
    tableName: 'actividades_actividad',
    timestamps: false,
    classMethods: {
      associate: function(models) {
        Actividad.belongsTo(models.actividadesTipoActividad, { as: 'TipoActividad', foreignKey: 'id' });
        Actividad.hasMany(models.actividadesReserva, { as: 'Reservas', foreignKey: 'actividad_id' });
      }
    }
  });

  return Actividad;
};
```

En la imagen se muestra el ejemplo del modelo de ‘Actividad’, donde se aprecia el tipo y nombre de cada columna, el nombre de la tabla al que corresponde y las diferentes relacione que tiene con otros modelos.

El registro de cada uno de los modelos se realiza de forma automática, ya que está pensado para que, solo creando el modelo, se registre automáticamente, por lo que se facilita su escalabilidad, por si en un futuro se desea ampliar la aplicación. A continuación se muestra el código con el que se registra automáticamente cada modelo:

```
var fs        = require("fs");
var path      = require("path");
var Sequelize = require("sequelize");
var env       = process.env.NODE_ENV || "oficina";
var config    = require(__dirname + '/../config/config.json')[env];
var sequelize = new Sequelize(config.database, config.username, config.password, config);
var db        = {};

fs
  .readdirSync(__dirname)
  .filter(function(file) {
    return (file.indexOf(".") !== 0) && (file !== "index.js");
  })
  .forEach(function(file) {
    var modelDir = path.join(__dirname, file);
    fs
      .readdirSync(modelDir)
      .forEach(function (modelFile) {
        var modelPath = path.join(__dirname, file, modelFile);
        var model = sequelize.import(modelPath);
        var modelName = file + model.name;
        db[modelName] = model;
      });
  });

Object.keys(db).forEach(function(modelName) {
  if ("associate" in db[modelName]) {
    db[modelName].associate(db);
  }
});

db.sequelize = sequelize;
db.Sequelize = Sequelize;

module.exports = db;
```

Autenticación

Dentro de la autenticación podemos diferenciar dos partes: como se autoriza a un usuario a entrar en el sistema, y como se realizan las peticiones a partes privadas de la API (que necesitan autorización).

Login de usuario

Para la gestión de usuarios se utilizan las tablas de usuarios y grupos que crea Django por defecto en cada aplicación (son tablas con el prefijo *auth*). Cada usuario tiene un campo de contraseña en la que se guarda la palabra clave de ese usuario de forma encriptada. Para encriptar esta clave, se utiliza un algoritmo de encriptación. En este caso en particular se utiliza el algoritmo por defecto (*pbkdf2-sha256*). Los pasos que se realizan para saber si las credenciales de usuario son correctas son:

1. Comprobar que en la base de datos existe un usuario con un determinado ‘username’.

2. Si es así, se utiliza el mismo algoritmo de encriptación que se utilizó para guardar la contraseña del usuario, con la contraseña dada.

3. Si al compararse son iguales es que la contraseña es correcta y se autorizará al usuario a entrar en la aplicación. En caso contrario, se devolverá un mensaje de error y el usuario no podrá acceder a ninguna de las secciones que componen el panel de administración.

A continuación se muestra el fragmento de código que realiza las comprobaciones anteriormente mencionadas.

```
var pbkdf2 = require('pbkdf2-sha256');
var Promise = require('promise');
var models = require('../models');
var auth = {};

auth.login = function (user, password) {
  var promise = new Promise(function (resolve, reject) {
    var where = { username: user, isSuperuser: true};

    models.authUser.findAll({where: where}).then(function (users) {
      if (users && users.length === 1) {
        var user = users[0];
        var splitedPassword = user.password.split('$');
        var iterations = parseInt(splitedPassword[1]);
        var salt = splitedPassword[2];
        var hash = splitedPassword[3];
        var candidate = pbkdf2(password, salt, iterations, 64).toString('hex');
        hash = (new Buffer(hash, 'base64')).toString('hex');

        if (candidate.indexOf(hash) === 0) {
          resolve(user);
        }else {
          reject();
        }
      }else {
        reject();
      }
    });
  });

  return promise;
}

module.exports = auth;
```

Autorización para rutas de la API

Una vez el usuario se haya autenticado de forma correcta, ya puede accede a las diferentes zonas de la aplicación de administración. Cada sección dentro esta aplicación necesitará al menos una llamada a la API, situadas en rutas privadas, es decir, que necesitan autorización.

Para que el servidor sepa si un usuario esta autorizado o no se ha empleado una autorización basada en tokens y no en cookies. A este modelo de autorización se le denomina JWT, de las siglas JSON Web Token.

Este modelo se basa en la generación de un token que contiene datos relevantes al usuario junto con un tiempo de expiración. Este token es enviado en la cabecera de cada una de las peticiones de la API. El servidor espera esta cabecera,

comprueba que el token es correcto. Si lo es, permite que se acceda a esa ruta, en caso contrario, devolverá un mensaje de error (401 Unathorized) para informar de que el token utilizado no es correcto. Esta comprobación se realiza a través de un middleware llamado '*express-jwt*'.

Este modelo de autorización de rutas está cada vez más integrado en aplicaciones de este tipo, en detrimento del uso de cookies. Alguna de las razones de esta tendencia son:

- **Cross-domain / CORS:** cookies + CORS[9] no casan bien a través de diferentes dominios. Una solución basada en tokens permite crear llamadas AJAX a cualquier servidor en cualquier dominio, ya que se usa una cabecera HTTP (Authorization) para transmitir la información.

- **Stateless:** No es necesario mantener una sesión almacenada. El token es auto-contenido, por lo que tiene toda la información necesaria del usuario.

- **Autenticación desacoplada**: El token puede ser generado desde cualquier servidor. Tu aplicación solo debe saber el modo de obtenerlo y utilizarlo.

- **Dispositivos móviles**: Para aplicaciones nativas móviles, las cookies no son ideales cuando consumen datos de una API segura, ya que tienen que negociar con contenedores de cookies.

- **Rendimiento**: Buscar si una sesión es válida en la base de datos es más costoso que validar un token parseando su contenido.

Rutas privadas (API)

Cada una de la rutas de la API (que comienzan por /api/{modelo}) es privada, y se registran de manera automática utilizando un módulo genérico creado para exponer rutas CRUD de manera automática. Utilizando este modelo solo es necesario indicar el modelo para el cual se va a crear y la url base que se utilizará para uso. De Nuevo, se consigue una forma genérica de crear puntos de acceso CRUD de un modelo determinado, hacienda la gestión rutas altamente escalable.

En la siguiente imagen se muestra un ejemplo de como registrar las diferentes rutas de modelos asociados a las 'actividades':

```
var crud = require('./crud');
var express = require('express');

var router = express.Router();

crud.service(router, 'actividad', 'actividadesActividad');
crud.service(router, 'reserva', 'actividadesReserva');
crud.service(router, 'tipo', 'actividadesTipoActividad');

module.exports = router;
```

Una vez creadas todas las rutas, solo es necesario asignar esas rutas a su punto de acceso particular. A continuación se muestra una imagen del código que realiza esta función:

```
var express = require('express');
var jwtMiddleware = require('express-jwt');
var secret = require(__dirname + '/../config/secret.json');
var router = express.Router();
var routes = {};

/** ROUTES **/
var authDjango = require('./authdjango');
var auth = require('./auth');
var eventos = require('./eventos');
var actividades = require('./actividades');
var fiestas = require('./fiestas');
var general = require('./general');
var puntoInteres = require('./puntointeres');

/** REGISTRATION **/
routes.register = function (app) {

  app.use(jwtMiddleware({ secret: secret.key}).unless({path: ['/auth/token', '/']}));
  app.use('/auth/token', auth);
  app.use('/api/auth', authDjango);
  app.use('/api/eventos', eventos);
  app.use('/api/actividades', actividades);
  app.use('/api/fiestas', fiestas);
  app.use('/api/general', general);
  app.use('/api/puntosinteres', puntoInteres);

  app.get('/', function(req, res) {
      res.sendFile('index.html', {root: 'app'}); // load the single view file (angular
  });

}

module.exports = routes;
```

4.1.2 Cliente

En esta sección se explicarán brevemente las partes involucradas en la parte de cliente de la aplicación. Como se ha comentado en capítulos anteriores, esta parte está desarrollada con el framework Angular, y a continuación se darán algunos detalles de los servicios, rutas, controladores y vistas, así como de los mecanismos utilizados para la comunicación segura con el servidor.

Servicios

Los servicios son los encargados de conectarse al servidor a través de la API expuesta por este. Cada servicio tiene asociada una url que a la hará las diferentes peticiones. Estas url's están definidas en un módulo llamado *adminConstants,* donde se definen todas las constantes de la aplicación.

Cada servicio pertenece a un módulo propio de servicios llamado *adminServices* y es una dependencia del módulo principal (*adminDjango*).

A continuación se muestra un ejemplo de uno de los servicios creados. Se puede observar como realiza peticiones GET, POST, PUT o DELETE a la misma url, un servicio CRUD:

```
angular.module('adminServices')
    .factory('bannerService', ['$http', 'bannerUrl', function ($http, bannerUrl) {

        var all = function () {
            return $http.get(bannerUrl + 'filter');
        }

        var get = function (id) {
            return $http.get(bannerUrl + id);
        }

        var post = function (banner) {
          var fd = new FormData();
          angular.forEach(banner, function(value, key) {
            fd.append(key, value);
          });
          var config = {
            headers: { 'Content-Type': undefined },
            transformRequest: angular.identity
          };
          return $http.post(bannerUrl, fd, config);
        }

        var put = function (id, banner) {
          var fd = new FormData();
          angular.forEach(banner, function(value, key) {
            fd.append(key, value);
          });
          var config = {
            headers: { 'Content-Type': undefined },
            transformRequest: angular.identity
          };
          return $http.put(bannerUrl + id, fd, config);
        }

        var deleteObject = function (id) {
          return $http.delete(bannerUrl + id);
        }

        return {
            all: all,
            get: get,
            post: post,
            put: put,
            deleteObject: deleteObject
        }
}]);
```

Rutas o estados

Para definir las rutas se utiliza *ui-router[10]*, un gestor de rutas basado en estados. Cada ruta tiene asociada una url, una vista y un controlador. Ciertamente interesante es el parámetro *resolve*, que obligue a la ruta a poseer ciertos datos antes de que se cargue la vista. Muy útil cuando los datos provienen de servidores remotos, como es nuestro caso.

A continuación se muestra una imagen de una de las rutas definidas en el sistema donde se aprecian cada una de las partes definidas previamente:

```
.state('miscelanea.editBanner', {
url: '/banners/edit/:id',
templateUrl: 'partials/miscelanea/banners/form.html',
resolve: {
  banner: function ($stateParams, bannerService) {
    return bannerService.get($stateParams.id).then(function (data) {
      return data.data;
    });
  }
},
controller: 'bannerFormCtrl as BannerForm'
```

Controladores

Los controladores son definidos en el módulo principal de la aplicación (adminDjango), y utilizan los servicios (o dependencias resueltas en las rutas utilizando los servicios) para obtener los datos que será pasados a la vista para que esta los muestre al usuario final. Además de estás dependencias, puede tener otras, como *$state*, utilizada para transitar de un estado a otro (otra ruta diferente, como si fuse una redirección), y *toastr*, utilizada para mostrar mensajes informativos al usuario cuando realiza una acción.

Todos los controladores asociados a rutas, está creados con la notación '*controllerAs*', que permite dar un nombre diferente al controlador al que se le da cuando se define, y que puede ser utilizado posteriormente en la vista para que el código sea más legible, y sepa de donde viene cada variable.

```
angular.module('adminDjango').controller('bannerFormCtrl', BannerFormCtrl);

BannerFormCtrl.$inject = ['$scope', '$state', 'banner', 'bannerService', 'toastr'];

function BannerFormCtrl($scope, $state, banner, bannerService, toastr) {
  var vm = this;
  vm.banner = banner || {};
  vm.sectionName = banner ? 'Modificar banner' : 'Crear banner';
  vm.buttonText = banner ? 'Modificar' : 'Crear';
  vm.setImage = setImage;
  vm.save = save;

  function setImage (image) {
    vm.previewImage = image;
    vm.banner.file = image.image;
  }

  function save () {
    if (banner) {
      bannerService.put(banner.id, vm.banner).then(function () {
        toastr.success("Banner modificado con éxito.");
        $state.go('miscelanea.banners');
      }, function (error) {
        toastr.error('No se ha podido modificar el banner', 'Error');
      });
    } else {
      bannerService.post(vm.banner).then(function (data) {
        toastr.success("Banner creado con éxito.");
        $state.go('miscelanea.banners');
      }, function (error) {
        toastr.error('No se ha podido crear el banner', 'Error');
      });
    }
  }

}
```

Vistas o partials

Están escritas en HTML, pero utilizan algunas de las características que proporciona Angular para que puedan ser utilizadas con diferentes datos. A parte de las directivas (descritas en secciones anteriores), la principal característica de estas es la posibilidad de utilizar funciones y variables definidas en el contexto del controlador. Así, tomando como controlador el mostrado en la imagen anterior, para utilizar un objeto del controlador habría que utilizar la notación *BannerForm.objeto,* y para llamar a una función *BannerForm.nombreFuncion().*

Angular ofrece multitud de características y funcionalidades que pueden ser utilizadas en las vistas y que ahorran al programador de realizar códigos ilegibles y extensos (como herramientas de iteración, directivas de control, etc.). La vista asociada al anterior controlador y ruta sería:

```
<div class="row">
  <div class="col-md-12">
    <button ui-sref="miscelanea.banners" type="button" class="btn btn-w-md btn-gap-v btn-primary">Ir al listado</button>
  </div>
</div>
<div class="row">
  <div class="col-md-12">
      <section class="panel panel-default">
          <div class="panel-heading"><strong><span class="fa fa-th"></span> {{ BannerForm.sectionName }}</strong></div>
          <div class="panel-body">
              <form role="form">
                  <div class="form-group">
                      <label for="banner-title">Título</label>
                      <input ng-model="BannerForm.banner.titulo" type="text" class="form-control" id="banner-title" placeholder="Título">
                  </div>
                  <div class="form-group">
                    <div>
                        <label>Activo</label>
                    </div>
                    <label class="switch switch-success"><input ng-model="BannerForm.banner.activo" type="checkbox" checked><i></i></label>
                  </div>
                  <div class="form-group">
                    <input type="file" title="Elegir imagen" data-ui-file-upload data-image-preview data-vm="BannerForm" class="btn-primary">
                  </div>
                  <div class="form-group">
                    <img ng-show="BannerForm.previewImage" class="img-responsive" ng-src="{{BannerForm.previewImage.url}}" />
                  </div>
                  <div ng-show="!BannerForm.previewImage" ng-if="BannerForm.banner.imagen" class="form-group">
                    <img class="img-responsive" ng-src="{{BannerForm.banner.imagen}}" />
                  </div>
                  <div class="form-group col-md-12">
                    <button type="submit" class="btn btn-success" ng-click="BannerForm.save()">{{ BannerForm.buttonText }}</button>
                  </div>
              </form>
          </div>
      </section>
  </div>
</div>
```

Comunicación segura con el servidor

Para poder utilizar la API del servidor, es necesario enviar en la cabecera de todas las peticiones, un token válido para que el servidor permita el acceso y pueda responder debidamente.

Cuando un usuario hace login de forma correcta, el servidor responde con un token válido, que es almacenado en el Local Storage[11] del navegador.

Cuando un usuario realiza cualquier petición, el sistema incrusta este token en la cabecera de la misma antes de ser enviada al servidor. Esto se realizada mediante los llamados *interceptors[12]* que pueden modificar cualquier petición/respuesta que se haga a servidores remotos.

Un ejemplo de interceptor, definido en la aplicación, es el que se muestra en la siguiente imagen:

```
angular.module('adminDjango').factory('authHttpResponseInterceptor', AuthHttpResponseInterceptor);

AuthHttpResponseInterceptor.$inject = ['$q', '$injector', 'localStorageService'];

function AuthHttpResponseInterceptor ($q, $injector, localStorageService) {
  return {
    request: function (config) {
      config.headers = config.headers || {};

      var token = localStorageService.get('authorizationData');
      if (token) {
        config.headers.Authorization = 'Bearer ' + token;
      }

      return config;
    },

    responseError: function (rejection) {
      if (rejection.status === 401) {
        localStorageService.remove('authorizationData');
        var $state = $injector.get('$state');
        $state.go('login');
      }
      return $q.reject(rejection);
    }
  };
}

angular.module('adminDjango').config(['$httpProvider', function ($httpProvider) {
  //Http Intercpetor to check auth failures for xhr requests
  $httpProvider.interceptors.push('authHttpResponseInterceptor');
}]);
```

4.2 Interfaz de usuario

En esta sección se describirá como está diseñada la interfaz de usuario. Se trata de un diseño claro e intuitivo, de fácil manejo, y que mejora sustancialmente la administración por defecto que ofrece Django.

Se comenzará con una visión general de la interfaz, explicando las diferentes zonas distinguibles en la interfaz, haciendo hincapié en algunos detalles relevantes. Posteriormente se expondrán ejemplos de diferentes listados, formularios o páginas de detalles que están a disposición del usuario.

4.2.1 Estructura general

La interfaz de usuario está compuesta principalmente por 3 partes bien diferenciadas: una cabecera, un menú lateral situado a la izquierda de la pantalla, y una sección principal que es donde se muestra el contenido del panel de administración desarrollado.

A continuación se detallarán alguna de las características que componen cada una de las partes:

Cabecera

Situada en la parte superior de la pantalla, ocupa el tamaño necesario para contener una serie de elementos útiles que muestran información relativa a la sección actual, alertas o un enlace para abandonar la aplicación:

- Título de la sección actual: Situado la izquierda, muestra el nombre de la sección en la que se encuentra en ese momento el usuario.

- Alertas: se trata de unos iconos con un indicador numérico. Uno de ellos indica el número de formularios de contacto que se ha recibido en ese día (icono de un sobre), y el otro (icono campana) muestra el número de reservas que tendrán lugar en menos de 7 días. Se encuentran situadas junto al título de la sección actual, a su derecha.

- Botón de salir: Situado en la derecha de la cabecera, es un botón para salir de la parte autenticada de la aplicación.

Menú lateral izquierdo

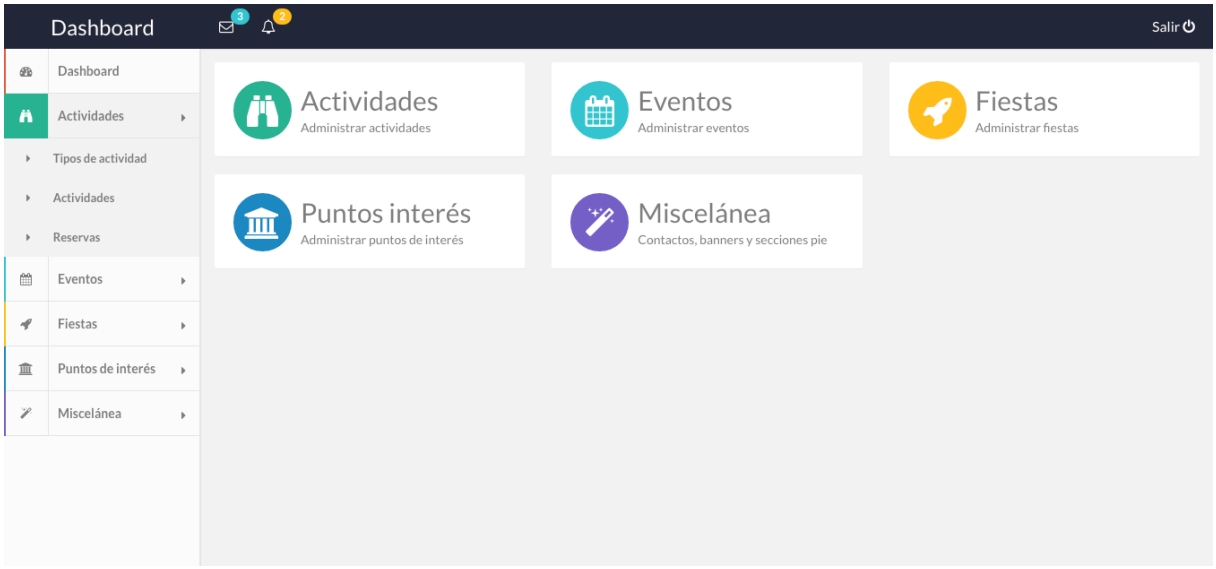
Muestra un listado de las diferentes secciones disponibles en el panel de control. Todos los menús, excepto el menú 'Dashboard', contienen submenús, que se presentan como hijos de su principal en un menú desplegable. Los menús disponibles son:

- Dashboard. Conduce a la página de inicio de la aplicación.
- Actividades. Contiene los submenús: Tipos de actividad, Actividades y Reservas.
- Eventos. Contiene los submenús: Tipos de evento y Eventos.
- Fiestas. Contiene los submenús: Ámbitos de fiesta y Fiestas.
- Puntos de interés. Contiene los submenús: Tipos, Puntos de interés y Olvera.
- Miscelánea. Contiene los submenús: Contactos, Banners y Sección Pie.

Contenido principal

Situado debajo de la cabecera, y la derecha del menú lateral, se trata de la zona de la interfaz donde se mostraran los diferentes listados, formularios de creación y edición de elementos, filtros y páginas de detalle. Es la zona donde el usuario realizará todas las gestiones necesarias para mantener la web a la que pertenece la aplicación de administración.

A continuación se muestra una imagen donde se puede ver claramente todas las partes anteriormente descritas:



4.2.2 Elementos de la interfaz

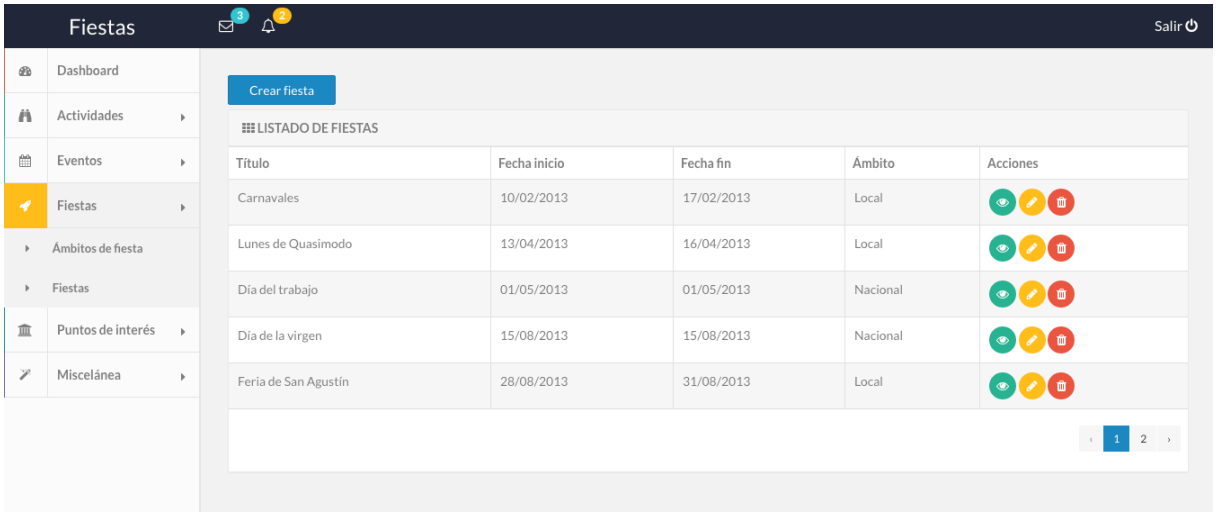
En esta sección se explicaran diferentes elementos que se encuentran en diferentes partes de la aplicación, como formularios, listados, filtros, páginas de detalle de entidades, mensajes de información o ventanas modales.

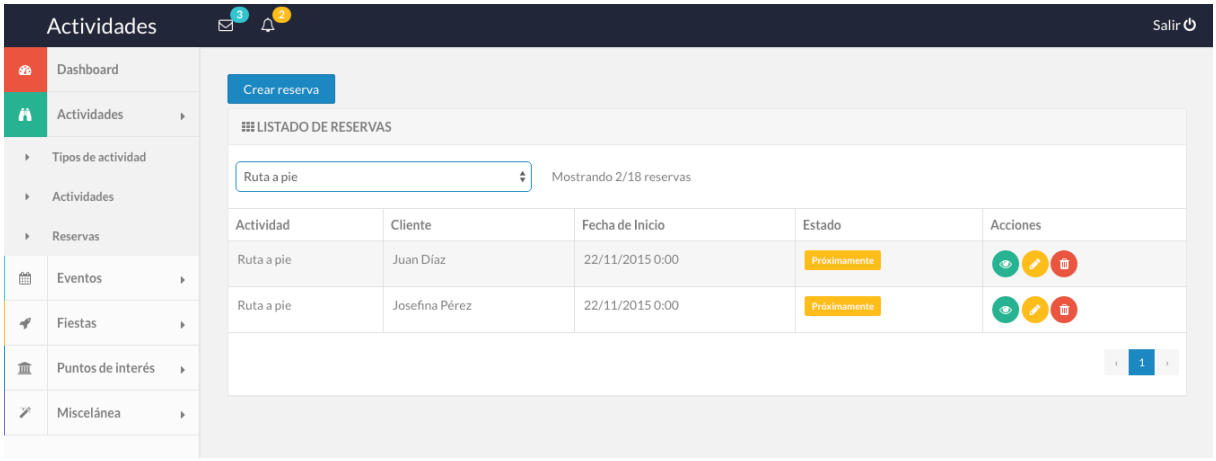
Listados

Es complicado no encontrar una sección que no contenga un listado. Los listados muestran información relevante que hacen que el usuario sepa de un vistazo lo que puede ser esa entidad. Cada listado de cada sección contienen datos diferentes, dependiendo de las entidades que muestra, pero todos tienen en común una columna de 'acciones'. Estás acciones permiten ver una entidad en detalle, modificarla o borrarla, identificándose cada una de ellas con un icono y un color diferente.

Todos los listados vienen acompañados de una paginación, para mostrar de n en n , y no tener largos listados que hacen que el usuario se pierda. Algunos de ellos contiene también filtro, para facilitar la búsqueda de determinadas entidades y no tener que recorrer un gran número de páginas para encontrarlo.

A continuación se muestran dos imagines que muestran un listado de elementos paginados y otra en la que se ve un filtro.

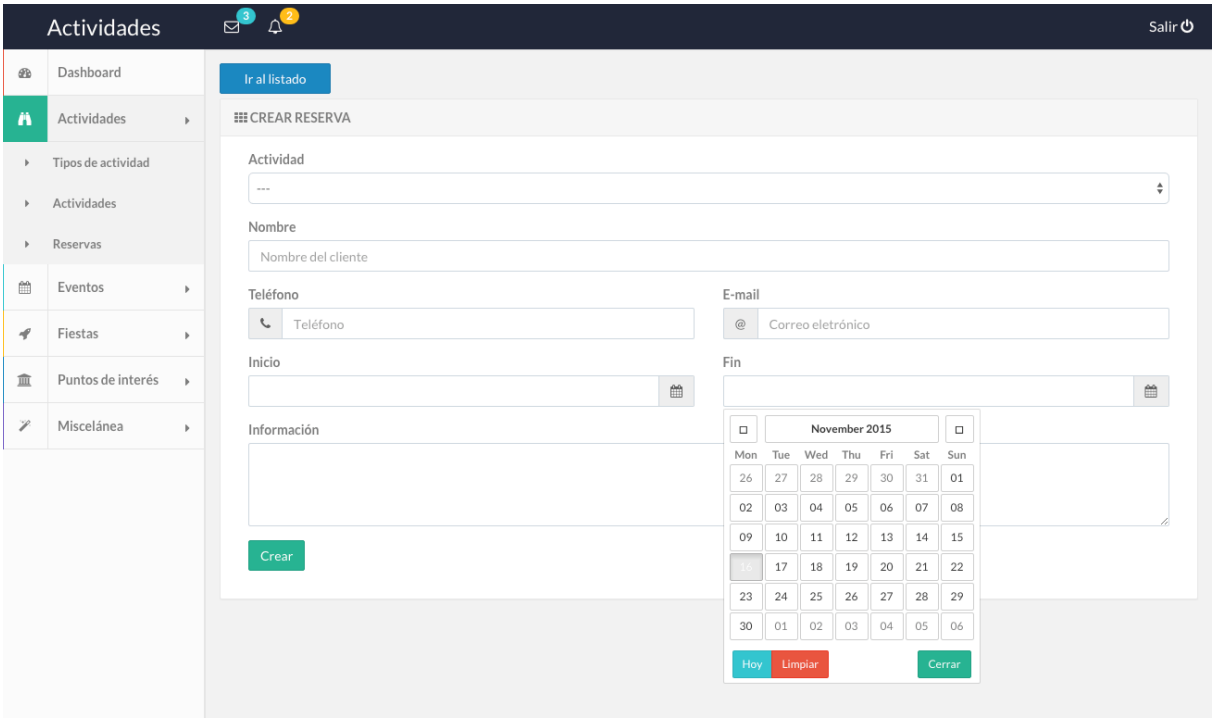




Formularios

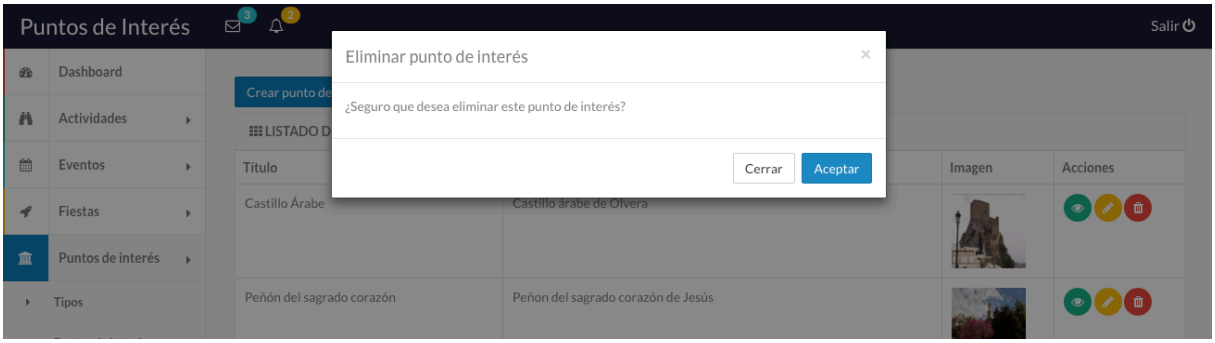
Se utilizan tanto para la creación de nuevas entidades como para la modificación de las mismas. Se pueden encontrar campos de textos normales, áreas de texto, campos de fechas con widgets asociados, campos de hora (hora:minutos), áreas de texto enriquecido con editores WYSIWYG[13], previsualización de imágenes, selectors, etc.

Con todos estos elementos se facilita el rellenado de los mismos por parte del usuario en todo tipo de dispositivos. Un ejemplo de formulario es el que se muestra en la siguiente imagen, en la que se pueden observar algunos de los elementos anteriormente mencionados:



Paneles modales

Son utilizados cuando se requiere la confirmación de un usuario, como por ejemplo cuando se elimina un objeto. También son utilizados para la creación y edición de pequeñas entidades, como por ejemplo los tipos de eventos o ámbitos de fiestas entre otros. Un ejemplo de panel modal se muestra en la siguiente imagen:



Mensajes informativos

Muestran información al usuario acerca de sus acciones, de manera que este sabe en todo momento el resultado de sus decisiones, como crear entidades o modificarlas o borrarlas. Son mensajes tanto de éxito como de error, distinguiéndose cada uno por colores e iconos.

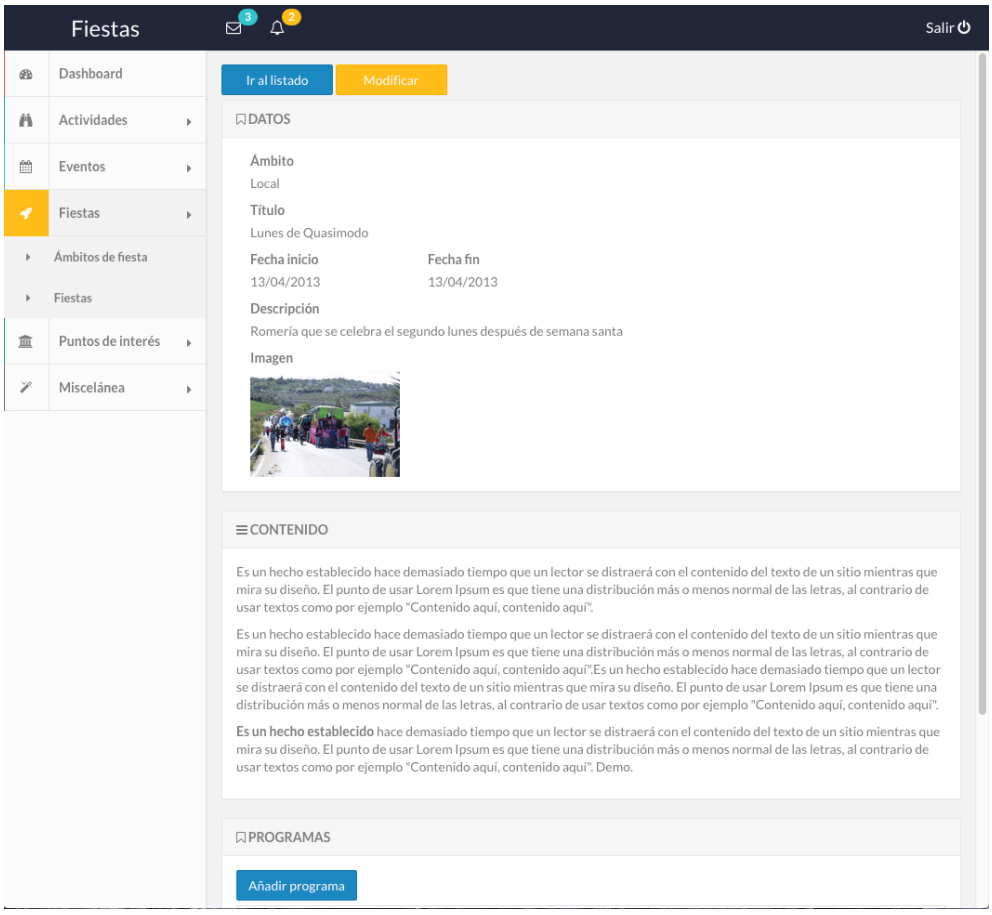
Estos mensajes son efímeros, y desaparecen de la pantalla al cabo de 2 segundos y medio. Un ejemplo de mensaje es el que se muestra en la siguiente imagen.

Detalle de entidad

Son páginas que muestran toda la información almacenada de un objeto o entidad. Si la entidad tienen un campo de texto enriquecido, lo muestra tal como quedaría en la aplicación web principal, así como previsualización de imágenes si es necesario.

Hay páginas de detalle que también contienen listados, como la vista de detalles de fiestas y eventos, ya que los programas asociados se muestran como listados similares a los que se describieron en secciones anteriores, incluyendo acciones sobre los elementos de los listados.

Suelen tener botones para modificar la entidad y para volver al listado de entidades de su mismo tipo. A continuación se muestra un ejemplo de una página de detalle:



CAPÍTULO 5

CONCLUSIONES Y LÍNEAS FUTURAS

Durante el tiempo que se ha invertido en este proyecto se ha creado una aplicación web de administración de una aplicación Django existente, para sustituir la que ofrece por defecto. Se trata de una aplicación JavaScript End-to-End, ya que se ha implementado tanto el servidor como la parte de cliente de la misma usando como base JavaScript. Para ambas partes se han utilizado herramientas de ultima generación en cuanto al desarrollo web se refiere, haciendo posible que su uso sea satisfactorio tanto en ordenadores persones, como en dispositivos móviles de diferente índole. En las siguientes líneas se comentan brevemente las conclusiones que se han obtenido de las diferentes tecnologías usadas, así como del proyecto en general, para terminar orientando al lector cuáles podrían ser las líneas futuras de desarrollo de este proyecto, aportando mejoras e ideas que no han podido ser desarrolladas durante la ejecución del mismo.

Node y Express han resultado herramientas muy útiles y fascinantes para el desarrollo del servidor de la aplicación. En este proyecto se ha mostrado una ínfima parte de las posibilidades que ofrecen. Existe una comunidad enorme de desarrolladores que aportan cientos y cientos de paquetes (o módulos) que hacen que el límite de las funcionalidades de tus aplicaciones estén solo en tu imaginación. Además de tener muchísimas posibilidades en cuanto a funcionalidades se refiere, dan un rendimiento excelente, por lo que son ideales para aplicaciones que requieran bajo tiempo de respuesta. El conocimiento previo de JavaScript y de buenas prácticas relacionadas con este lenguaje, han facilitado enormemente la comprensión, aprendizaje y desarrollo de una aplicación profesional.

Angular permite construir interfaces vistosas con muy poco esfuerzo teniendo muy claro el papel que deben desempeñar cada uno de sus elementos. Para gente que esta acostumbrada a lidiar con frameworks MVC, es ideal para adentrarse en el mundo del desarrollo *FrontEnd*, que toma cada día más fuerza. Los resultados son excelentes, y ayuda a crear páginas profesionales que da la sensación al usuario de que todo se hace desde la misma página. Junto con Bootstrap, se ha conseguido desarrollar la parte de cliente de la aplicación para todo tipo de dispositivos, con una interfaz clara y limpia, lejos de tediosos paneles de administración sobrecargados que ralentizan el trabajo de los usuarios.

Haciendo una valoración del sistema en general, el proyecto propuesto cumple con las expectativas que se crearon al principio del mismo, consiguiéndose una aplicación de administración web desacoplada del proceso Django, con un mejor diseño y usable en todo tipo de dispositivos, tanto de escritorio como móviles. Técnicamente, ha resultado ser un proyecto muy enriquecedor, utilizando las últimas tecnologías web que ofrece el mercado. No cabe ninguna duda de que este proyecto

ha servido para aprender tecnologías y técnicas avanzadas para el desarrollo de proyectos futuros utilizando arquitecturas similares.

En cuanto a las líneas de desarrollo futuras, algunas características que pueden ser de gran utilidad en un futuro pero que no ha sido necesario implementar para este proyecto son:

- **MongoDB como base de datos**. En este proyecto se ha utilizado principalmente Node, Express y Angular, por lo que solo faltaría una pieza al puzle, y esa pieza es la base de datos no relacional Mongo. De esta manera no haría falta ningún conector con la base de datos que transformara los datos en algo que entienden Node o Express, ya que Mongo habla el mismo idioma que estas (JSON). Incluyendo Mongo a la arquitectura de la aplicación, esta implementaría la llama ***MEAN stack***[14] (*Mongo-Express-Angular-Node*). Al utilizar Mongo como base de datos el rendimiento del sistema sería todavía mejor, y serían necesarias algunas modificaciones en la aplicación Django, ya que ambas comparten la misma base de datos.

- **Panel de administración genérico**. Aunque se han creado muchos procesos genéricos para facilitar la escalabilidad de la aplicación, sería posible ir un paso más allá, y hacer de esta una aplicación genérica de administración para cualquier aplicación Django del mundo. El punto a tratar para que esta idea tenga éxito, es la creación automática de los modelos correspondientes a la base de datos.

BIBLIOGRAFÍA

[1] **'ECMAScript'** en Wikipedia
https://en.wikipedia.org/wiki/ECMAScript

[2] **'Node.js'** en Nodejs
https://nodejs.org/en/

[3] **'Express.js'** en expressjs
http://expressjs.com/

[4] **'MVC'** en Wikipedia
https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[5] **'Angular'** en angularjs
https://angularjs.org/

[6] **'CRUD'** en Wikipedia
https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[7] **'Sequelize'** en sequelizejs
http://docs.sequelizejs.com/en/latest/

[8] **'ORM'** en Wikipedia
https://en.wikipedia.org/wiki/Object-relational_mapping

[9] **'CORS'** en Wikipedia
https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

[10] **'ui-router'** en GitHub
https://github.com/angular-ui/ui-router

[11] **'Local Storage'** en w3schools
http://www.w3schools.com/html/html5_webstorage.asp

[12] **'Interceptors'** en angularjs
https://docs.angularjs.org/api/ng/service/$http

[13] **'WYSIWYG'** en Wikipedia
https://en.wikipedia.org/wiki/WYSIWYG

[14] **'MEAN stack'** en mean.io
http://mean.io/

'**Web Development with Node & Express**' Ethan Brown
O'REILLY

ANEXO A

Estructura de las aplicaciones

En este anexo se explicará la estructura de carpetas de cada parte del sistema y que se encuentra en cada una de ellas.

Servidor

El servidor está compuesto por las siguientes carpetas y ficheros:

- **app.js**: fichero principal del servidor.
- **package.json**: fichero de configuración de aplicaciones node.
- **app**: Carpeta donde se encuentra la aplicación del cliente.
- **auth**: Carpeta que contiene la lógica de login.
- **config**: Carpeta que contiene diversos ficheros de configuración del sistema.
- **models**: Carpeta que contiene todos los modelos de la base de datos que serna registrados.
- **node_modules**: Carpeta que contiene todos los módulos node externos.
- **routes**: Carpeta que contiene todas las rutas registradas en el servidor.

Cliente

La parte del cliente esta compuesta por:

- **app.js**: fichero principal que define el módulo angular base de la aplicación, rutas e interceptors.
- **index.html**: Página de entrada de la aplicación.
- **constants.js**: Fichero donde se define el módulo de constantes usado en la aplicación.
- **controllers**: Carpeta que contiene todos los controladores usados.
- **css**: contiene las hojas de estilo usadas.
- **directives**: Carpeta que contiene todas las directivas de la aplicación.
- **files**: contiene los archivos binarios subidos en las distintas secciones, como las imagines.
- **js**: Contiene los archivos JavaScript de las distintas librerías utilizadas.
- **partials**: Carpeta que contiene todas las vistas utilizadas en la aplicación.
- **services**: Carpeta que contiene todos los servicios definidos.

ANEXO B

CONTENIDO DEL CD

Además de este documento, se entrega un CD con el siguiente contenido:

- Esta misma memoria en formato PDF (Memoria.pdf).
- Un resumen del proyecto en español en formato PDF (Resumen.pdf).
- Un resumen del proyecto en inglés en formato PDF (Abstract.pdf)
- La carpeta *Django*, que contiene la aplicación Django en la que se basa el proyecto.
- La carpeta *admin-django*, que contiene la aplicación completa de este proyecto.