

Capítulo 4

Diseño e implementación

En este capítulo describiremos en profundidad la implementación de la aplicación de gestión de clases particulares, tanto su lado cliente como su lado servidor, realizada en el proyecto. Antes, comenzaremos con el diseño del mismo para tener una visión más global y poder entender las partes constituyentes por separado.

4.1. Diseño

La aplicación que hemos desarrollado se divide en 4 grandes bloques: Node, MongoDB, Express y Angular. Cada uno de ellos se encarga de realizar una función dentro de la aplicación.

Antes de profundizar en cada bloque, todos los proyectos que utiliza la pila MEAN, siguen una estructura similar a la de la figura 4.1

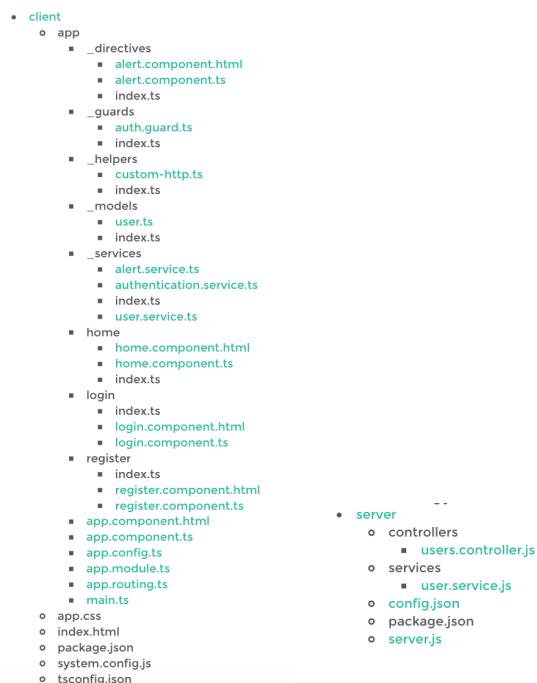


Figura 4.1: Estructura de un proyecto con la pila MEAN

4.2. Lado cliente de la aplicación web

De los 8 bloques principales de una app en Angular, vamos a ir identificando uno a uno y que uso se le da en nuestra aplicación.

4.2.1. Modulos:

Como las aplicaciones en Angular son modulares y un módulo es el conjunto de código dedicado a cumplir un único objetivo, los módulos utilizados en nuestra aplicación son:

- **NgModule from '@angular/core'** Es el módulo principal, el cual recibe un objeto que define el módulo. Los metadatos más importantes de un NgModule son:
 1. Declarations: Las vistas que pertenecen a tu módulo. Hay 3 tipos de clases de tipo vista: componentes, directivas y pipes.
 2. Exports: Conjunto de declaraciones que deben ser accesibles para plantillas de componentes de otros módulos.
 3. Imports: Otros NgModules, cuyas clases exportadas son requeridas por templates de componentes de este módulo.
 4. Providers: Los servicios que necesita este módulo y que estarán disponibles para toda la aplicación.
 5. Bootstrap: Define la vista raíz. Utilizado sólo por el root module.
- **RouterModule from '@angular/router'** es uno de los módulos más importantes de Angular, se encuentra dentro de la librería @angular/route, gracias a él cada vez que cambiemos de dirección URL cambiaremos de página sin necesidad de tener que interactuar con el servidor.
- **HttpModule, Http, RequestOptions from '@angular/http'** Otro módulo imprescindible en una aplicación Angular, se encuentra dentro de la librería @angular/http. Gracias a este módulo podemos hacer cualquier petición AJAX sin apenas tener que escribir código.
- **FormsModule from '@angular/forms'** Módulo encargado de añadir formularios personalizados.
- **FileUploadModule from 'ng2-file-upload'** Es el módulo que nos permite subir imágenes y enviarlas a nuestro servidor, para luego poder almacenarlas de forma ordenada en nuestra base de datos.
- **AgmCoreModule from 'angular2-google-maps/core'** Gracias a este módulo, podemos utilizar la API de google maps en nuestra aplicación.
- **BrowserModule from '@angular/platform-browser'** Este módulo es necesario en cualquier app que se renderice en el navegador.
- **AuthGuard from './common/auth.guard'** Gracias a este modulo, podemos conservar las credenciales de un usuario durante un tiempo determinado en el navegador.

- **ProvideAuth, AuthHttp, AuthConfig from 'angular2-jwt'** Este módulo proporciona seguridad a nuestra aplicación, generando un *token* encriptado para cada usuario que se registre en nuestra aplicación.
- **AppComponent, Intro, LoginAlumno, LoginProfesor , HomeAlumno, HomeProfesor, SignupAlumno, SignupProfesor, ProfesorDetail** Estos son los módulos propios se han desarrollado para la aplicación de este TFG.

4.2.2. Componentes:

Los componentes son como etiquetas nuevas que podemos inventarnos para realizar las funciones que sean necesarias para nuestra aplicación de gestión de clases particulares.

Se compone de 1 componente principal y 8 componentes que derivan de él. AppComponent es el componente principal y tiene la siguiente apariencia:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ClassCity';
}
```

Listing 4.1: AppsComponent

El selector app-root o el nombre de la etiqueta que se usará cuando se deseé representar. Con la propiedad templateUrl asociamos un archivo .html que se usará como vista del componente. Por último se define su estilo mediante la propiedad StyleUrls, indicando a un array de todas las hojas de estilos que deseamos.

Componentes secundarios

- **Intro** Este componente corresponde con la página introductoria a la aplicación donde podemos elegir entre qué perfil de usuario queremos adoptar: Profesor o Alumno. Como podemos ver en la figura 4.2.
- **LoginAlumno, LoginProfesor** Componentes encargados de realizar la función de login del alumno o del profesor como podemos ver en las figuras 4.3|4.4. Hemos desarrollado una función que se encarga de realizar una petición POST al servidor. Si la respuesta es aceptada, el alumno accederá a la aplicación y se almacenarán las credenciales en el *localStorage* del navegador con un tiempo de caducidad de 1 hora. Mientras que si la petición es rechazada, el servidor nos enviará un mensaje avisando de que *The username or password don't match*.
- **SignupAlumno, SignupProfesor** Estos componentes se van a encargar de registrar a profesores y alumnos en nuestra base de datos. Para ellos hemos desarrollado una función en cada componente que simplemente se encarga de enviar al servidor una petición POST con un cuerpo donde se encuentran los datos personales del profesor o el alumno. Figura 4.5|4.6

- **HomeAlumno, HomeProfesor** Cuando un profesor o un alumno es aceptado dentro de nuestra base de datos y consigue entrar en la aplicación, puede realizar diferentes funciones dependiendo de si entro como alumno o como profesor. figuras 4.7|4.9
 1. **Alumno** Un alumno puede realizar la búsqueda del profesor que más le interese por diferentes parámetros:
 - El curso en el que está el alumno
 - La asignatura que quiere cursar
 - La distancia a la que se encuentre el profesor
 2. **Profesor** La página del profesor consiste en un chat realizado con *websockets*, donde podrá entablar conversación con cualquier alumno que este interesado en él. Aparte puede personalizar su perfil, cambiando la foto que tiene como avatar.
- **ProfesorDetail** Cuando un alumno encuentra a su profesor particular ideal desde la página del alumno y hace click sobre el profesor interesado, el componente ProfesorDetail se lanza y consiste en una ficha técnica del profesor particular, así como un chat donde el alumno podrá comunicarse con el profesor para poder quedar y acordar el precio de la clase tal y como podemos ver en la figura 4.8

4.2.3. Plantilla

Las plantillas se utilizan para dar forma a las aplicaciones. Es la parte más visual de una aplicación web y es la propia magia de Angular la que se encarga de renderizar estas plantillas, haciendo aplicaciones mucho más personalizadas para el usuario. A continuación vamos a ir analizando una a una las diferentes plantillas que forman la aplicación de gestión de clases particulares:

intro.html Cuando accedemos a la URL ¹ de la aplicación, la primera plantilla que se nos presenta es Intro.html. Según podemos ver en la imagen 4.2, consiste en una página introductoria donde podemos elegir si somos alumnos o profesores.



Figura 4.2: Página Intro ClassCity

¹<https://www.classcity.es>

loginalumno.html Al seleccionar dentro de Intro.html en Alumno, accedemos a la siguiente plantilla donde podemos visualizar un formulario con sus campos *username* y *password*. Además de resaltar la característica responsive de la aplicación, demostrando que es totalmente adaptable para dispositivos móviles como para ordenadores.

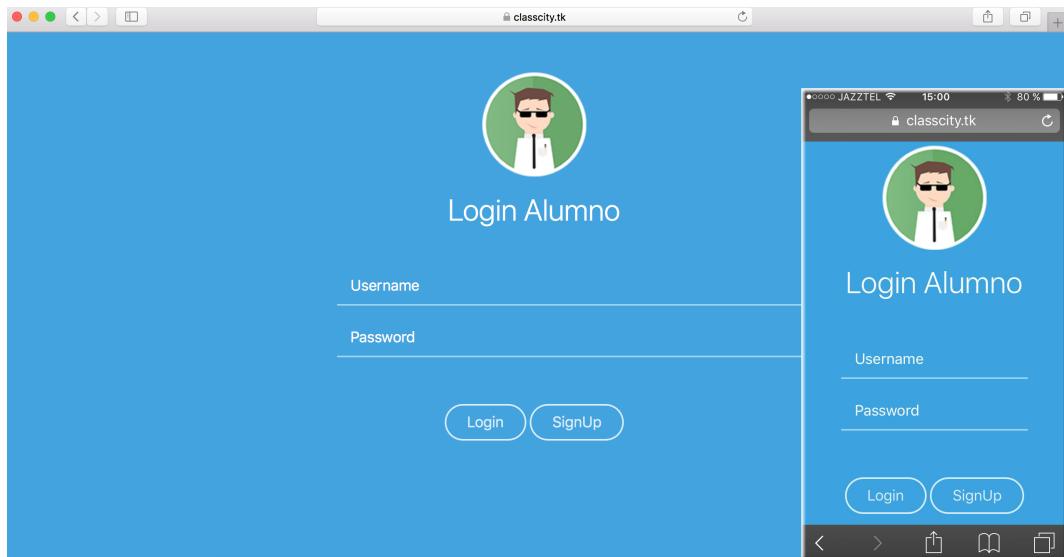


Figura 4.3: Página Login Alumnos

loginprofesor.html Si en vez de seleccionar Alumno hubiésemos seleccionado Profesor en la página introduccitoria, hubiésemos entrado en otro formulario donde los profesores ya registrados pueden acceder a la aplicación. Al igual que la plantilla de *loginalumno.html* es totalmente responsive. Para poder controlar que la plantilla sea responsive hemos utilizado diferentes ficheros css (*movil.css* y *ordenador.css*) por cada plantilla en todas las plantillas de la aplicación.

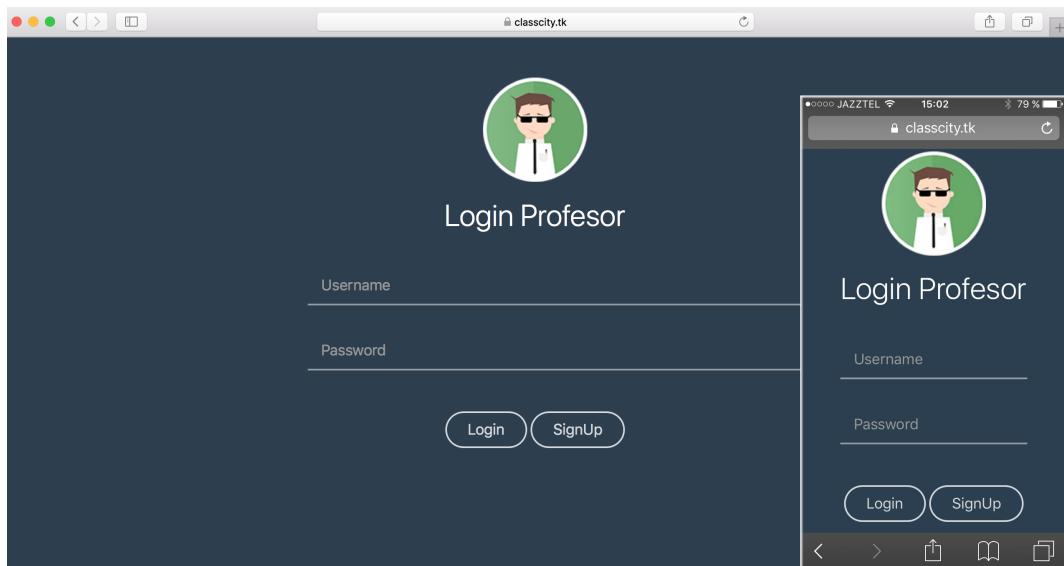


Figura 4.4: Página Login Profesor

registeralumno.html Si un alumno quiere registrarse simplemente debe entrar en SignUp, donde tendrá un formulario para poder completar todos los campos necesarios. Los datos que solicitamos para el alumno son los siguientes: Email, Password, Nombre, Apellidos y Fecha de Nacimiento. Todos estos datos serán enviados a nuestro servidor donde almacenaremos la información.

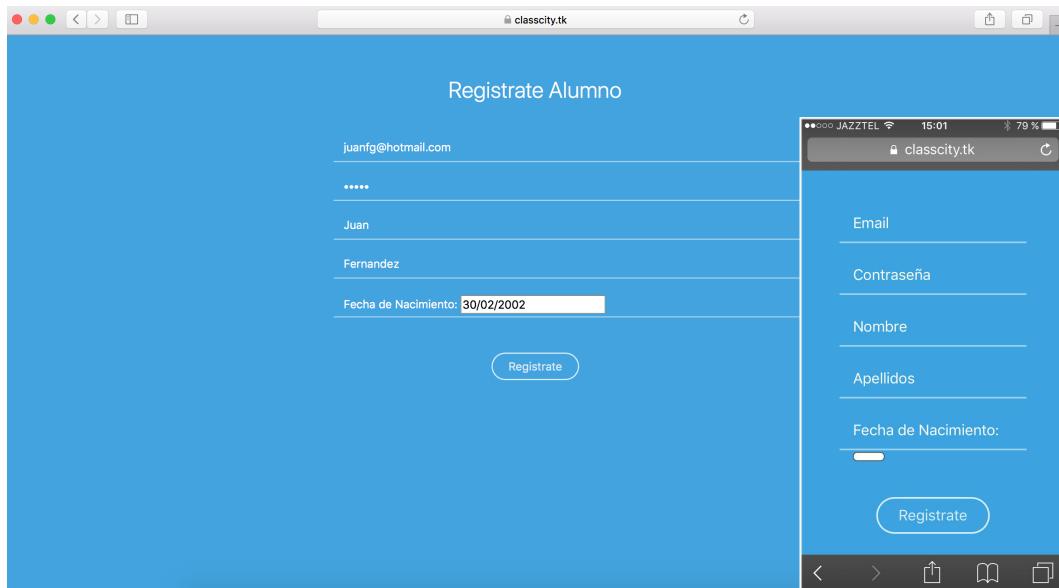


Figura 4.5: Página Registrar Alumno

registerprofesor.html Si es el profesor quien quiere registrarse en nuestra aplicación, entrará en SignUP de profesores e introducirá los datos necesarios. Los datos que solicitamos al profesor son: Ubicación, Nombre, Apellidos, Password, Email, Fecha de Nacimiento, Curso y Asignatura.

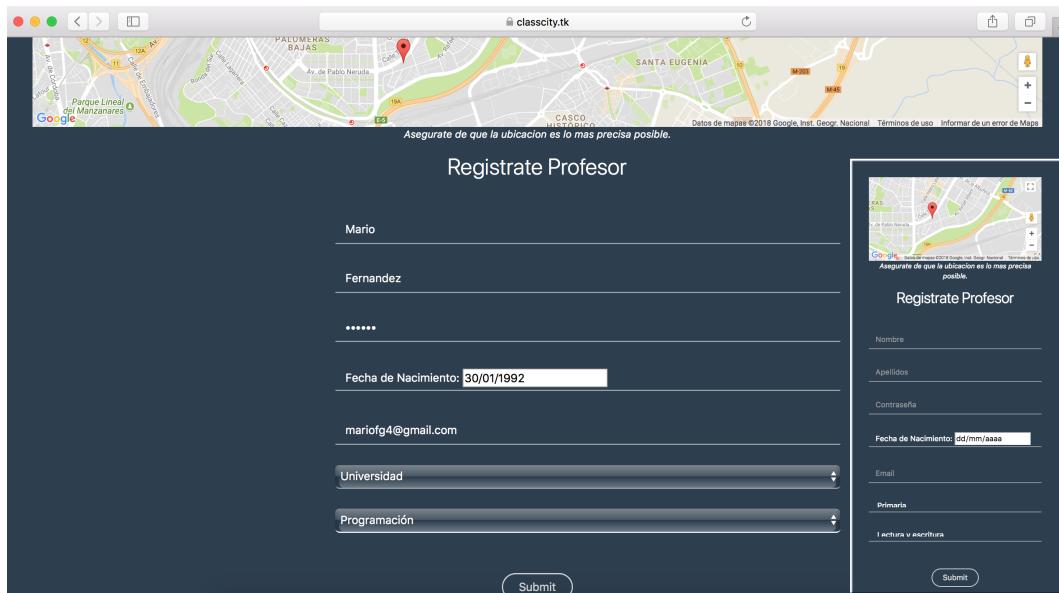


Figura 4.6: Página Registrar Profesor

homealumno.html Una vez que el alumno ya ha sido registrado en nuestra base de datos, la interfaz con la que se encontrará es la que podemos ver en la figura 4.7. Podemos apreciar una entrada en el mapa para introducir la ubicación donde queremos buscar. También podemos diferenciar los filtros de búsqueda que tenemos para buscar a nuestro profesor. Por último, cuando el alumno busca con los parámetros que desee, les aparecerán pintados en el mapa tantos profesores como haya en nuestra aplicación con esas especificaciones.

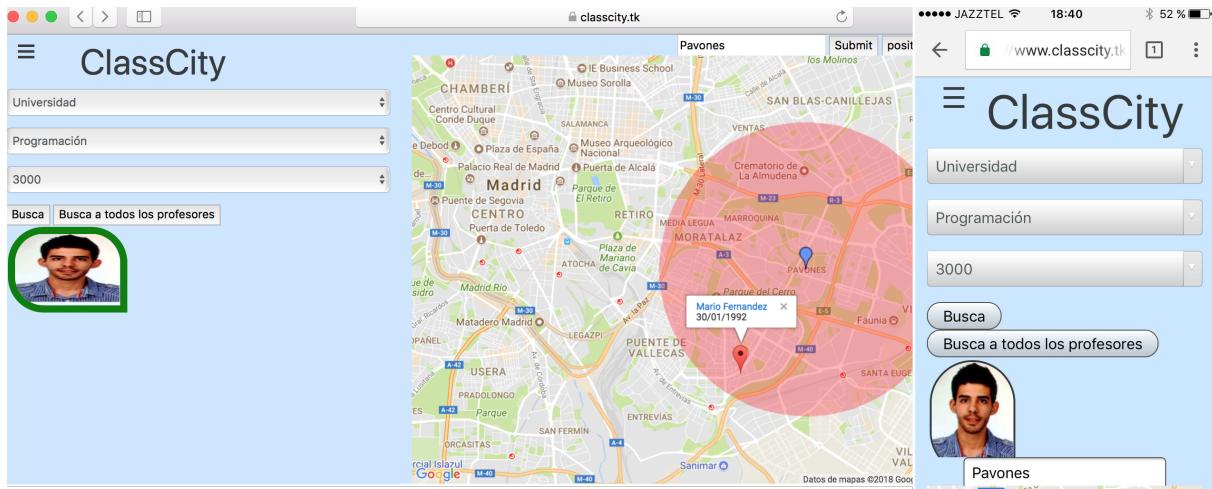


Figura 4.7: Página Home Alumno

detail.html Cuando el alumno encuentra algún profesor de su interés puede hacer click sobre la imagen del profesor y así entrar en más detalle, viendo su ficha técnica y pudiendo establecer una conversación a partir del chat. La ficha técnica muestra los siguientes datos del profesor: Nombre, Apellidos, Asignatura y Curso.

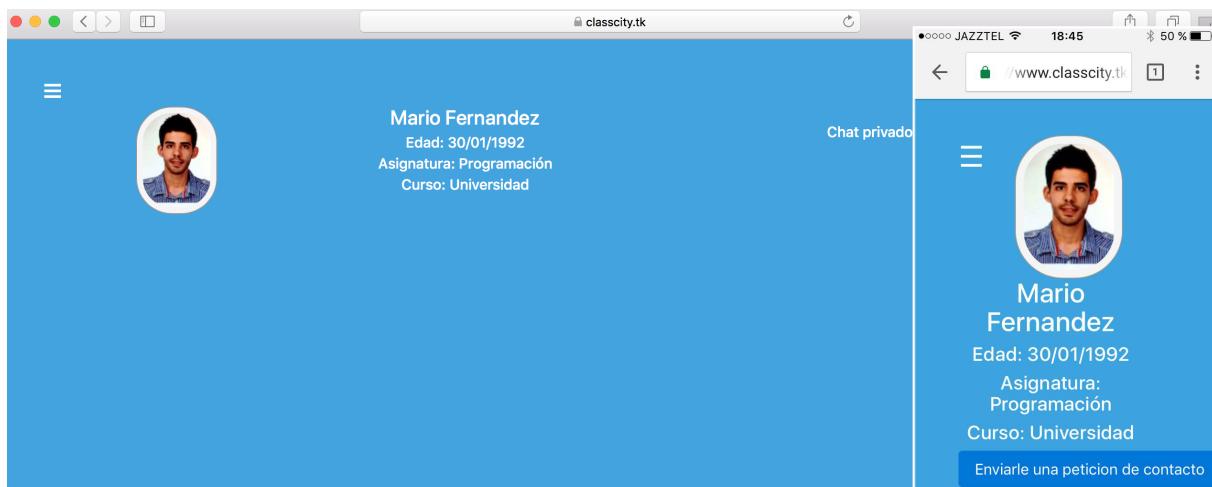


Figura 4.8: Página Detalle del Profesor

homeprofesor.html Cuando el profesor ya ha sido registrado en nuestra aplicación, el home del profesor es habilitado y en él puede hablar por un chat privado con todos los alumnos que le escriban por su canal, además de poder editar la foto de su perfil.

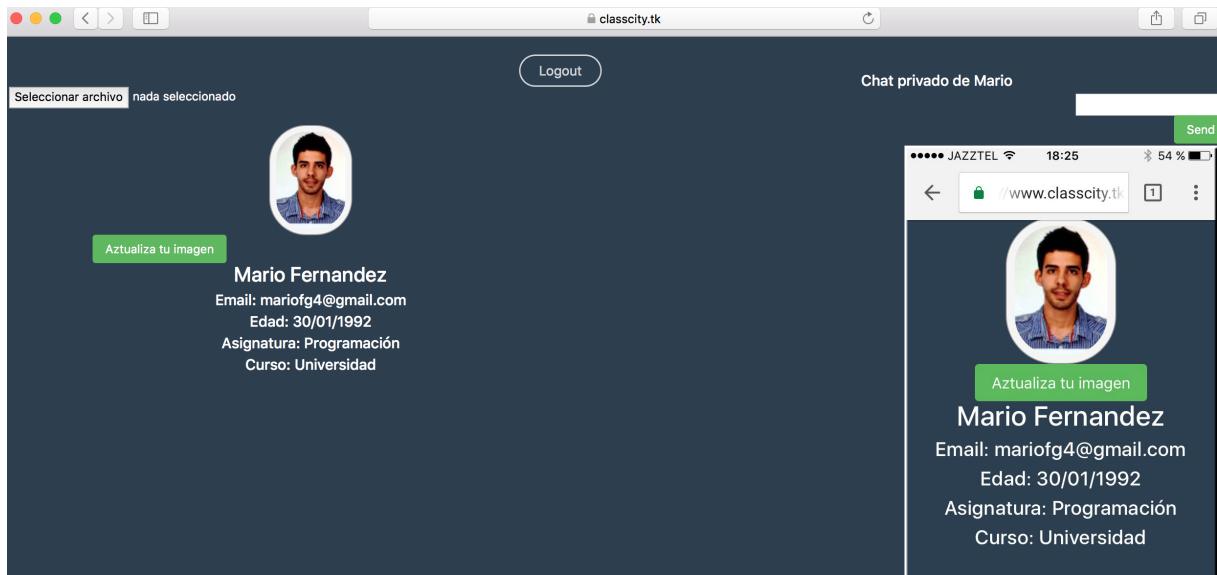


Figura 4.9: Página Home Profesor

4.2.4. Data Binding

El Data Binding nos abstrae de la lógica *pull/push* asociada a insertar y actualizar valores en el HTML y convertir las respuestas de usuario(inputs, clicks, etc) en acciones concretas.

Como se ha comentado en el capítulo anterior Angular tiene 4 formas de Data Binding, todas ellas se han utilizado en nuestra aplicación para realizar las siguientes funciones:

Interpolación(Hacia el DOM) Angular evalúa todas las variables e introduce su resultado en el DOM. En el siguiente ejemplo estamos insertando en el árbol DOM, todos los datos del usuario, consiguiendo una apariencia de la página mucho más personalizada.

```
<code>{{decodedJwt.Email}}</code></pre>
<code>{{decodedJwt.id.nombre}}</code></pre>
<code>{{decodedJwt.id.apellidos}}</code></pre>
<code>{{decodedJwt.id.edad}}</code></pre>
```

Property binding: (Hacia el DOM) Este tipo de Data Binding, permite pasar los objetos que queramos de nuestro componente padre ('home') a la propiedad (Latitud, Longitud, radius, fillColor) del componente hijo, en este caso sebm-google-map-circle. Para ello el componente hijo tiene que tener predefinidas ciertas entradas en su directiva.

```
<sebm-google-map-circle
    [latitude]="{{query.Loc.lat}}"
    [longitude]="{{query.Loc.lng}}"
    [radius]="{{query.Radio}}"
    [fillColor]="'red'>
</sebm-google-map-circle>
```

Event binding: (Desde el DOM) Si queremos invocar a una función cuando se lance un evento, como por ejemplo cuando queremos hacer click en el botón de LogOut. En el momento que hacemos click sobre el botón de logout, la función logout es invocada y salimos de la sesión.

```
<button type="Submit" (click)="logout()">Logout</button>
```

Two-way binding: (Desde/Hacia el DOM)

Cuando queremos combinar el *Event Binding* y el *Property Binding* tenemos el binding bi-direccional, como podemos ver en el siguiente ejemplo.

```
<input [(ngModel)]="address">
```

En este caso queremos que el valor de 'address' se actualice en el componente y que a su vez se introduzca dentro de la entrada como en el caso de *property binding*.

4.2.5. Directiva:

Las directivas son como los componentes, pero sin una plantilla asociada. En nuestro caso hemos utilizado directivas para realizar varias funciones, por ejemplo:

FileSelectDirective, FileDropDirective, FileUploader Estas tres directivas proceden del módulo "FileUploadModule", y sirven para poder subir una imagen desde el escritorio local del usuario al navegador, para luego enviar la imagen al servidor.

```
<input type="file" ng2FileSelect [uploader]="uploader" />
```

sebm-google-map, sebm-google-map-marker Estas son otras de las directivas procedentes del módulo 'agmCoreModule', el cual nos proporciona toda la interfaz de programación de GoogleMaps. Gracias a estas directivas podemos jugar con el API de google en nuestra aplicación angular.

```
<sebm-google-map *ngIf="query.Loc"
  [latitude]="query.Loc.lat"
  [longitude]="query.Loc.lng"
  [scrollwheel]="false" [zoom] = "13">
  <sebm-google-map-marker
    [latitude]="query.Loc.lat"
    [longitude]="query.Loc.lng"
    [iconUrl] = "iconUrl">
  </sebm-google-map-marker>
</sebm-google-map>
```

También hemos usado otros dos tipos de directivas que hemos usado en nuestra aplicación.

Directivas Estructurales: `*ngFor` repite su elemento en el DOM una vez por cada item que hay en el iterador que se le pasa, siguiendo una sintaxis de ES6.

Directivas Atributo: `ngModel` Implementa un mecanismo de binding bi-direccional. En este ejemplo el elemento HTML `<select>`, asigna la propiedad value a mostrar y además responde a eventos de modificación.

```
<div class="form-group">
  <select class="form-control" [(ngModel)]="query.Curso" name="curso">
    <option *ngFor="let p of curso" [value]="p">{ p }</option>
  </select>
</div>
```

4.2.6. Servicio

Los servicios se definen a través de simples clases y son imprescindibles en Angular, ya que toda función o valor es encapsulada dentro de un servicio.

AlumnoService Este servicio se encarga de convertir una dirección física a una coordenada (latitud, longitud) para poder representarlo en el mapa.

Si observamos detenidamente el código, lo primero que hacemos en la función 'getLatLan(address: string)', es una petición al API de Google maps con una dirección y esperamos a que nos responda. La respuesta puede ser satisfactoria o no. En caso de que sea OK las variables lat y lng serán actualizadas con los valores devueltos por Google maps. Si por el contrario no hubiésemos recibido respuesta, un mensaje de error será mostrado por pantalla.

```
export class AlumnoService {
  getLatLan(address: string) {
    let geocoder = new google.maps.Geocoder();
    return Observable.create(observer => {
      geocoder.geocode( { 'address': address}, function(results, status) {
        if (status === google.maps.GeocoderStatus.OK) {
          let obj: Object = {lat: results[0].geometry.location.lat(),
            lng: results[0].geometry.location.lng() };
          observer.next(obj);
          observer.complete();
        } else {
          console.log('Error - ', results, ' & Status - ', status);
          observer.next({}); 
          observer.complete();
        }
      });
    });
  }
}
```

4.3. Lado servidor de la aplicación

Nuestro lado servidor se compone de dos tecnologías muy importantes e innovadoras en el mercado de las aplicaciones web como son: Express, que como ya hemos comentado antes es un framework de desarrollo de aplicaciones web para Node.js y MongoDB que consiste en una base de datos NoSQL, el cual utiliza la librería *mongoose* para poder conectar node.js con la base de datos en MongoDB. Como ya he comentado en el capítulo anterior, Node.js es un sistema innovador, puesto que es la plataforma encargada del funcionamiento del servidor, y funciona totalmente con JavaScript. Gracias a Node, simplemente con una linea en la terminal seremos capaces de generar procesos que escuchen en el puerto que queramos. Una vez tengamos *Node(v10.0.0)* y *NPM(5.6.0)* instalados, procedemos a instalar las dependencias tanto del servidor como del cliente y posteriormente arrancar la aplicación.

- textbf{Instalar dependencias(Cliente y Servidor)}

```
sudo npm install
```

- **Correr nuestro servidor**

```
sudo node server.js
```

- **Arrancar el cliente:**

```
sudo npm start
```

4.3.1. Server.js

Las funciones contenidas en nuestra server.js son:

1. **Open mongo** Llamamos a la librería *mongoose* encargada de unir a MongoDB con Node.js, a continuación le decimos a qué base de datos apuntar y si el resultado es satisfactorio abrirá la conexión, en caso contrario saltará la excepción.

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/classcity');
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'conection error:'));
db.once('open', function() {
  console.log('Connected to Database');
});
```

2. **Parser body** Analizamos los cuerpos de las peticiones antes que llegue a sus manejadores.

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

3. **Control de errores** En caso de tener algún error en alguna solicitud, el manejador de errores se lanzará sin bloquear el resto del servicio.

```
app.use(function(err, req, res, next) {
  if (err.name === 'StatusError') {
    res.send(err.status, err.message);
  } else {
    next(err);
  }
});
```

4. **Control de imágenes** Para poder controlar la ingesta de imágenes en nuestras bases de datos hemos usado *Multer*, un middleware de node.js para el manejo multipart/form-data, que se usa principalmente para cargar archivos.

```
var storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, './uploads')
  },
  filename: function (req, file, cb) {
```

```

        console.log(file.fieldname);
        var name = file.fieldname + '-' + Date.now() + '.jpg';
        cb(null, name)
    }
})
var upload = multer({ storage: storage });
app.use(multer(upload).single('file'));

```

5. **Server sockets** Corremos nuestro chat en el puerto 8080, independientemente de nuestra aplicación que corre en el puerto 8000

```

var socketServer = require('./controllers/socket');
socketServer.start();

```

6. **Rutas** Nuestra aplicación se compone de multitud de rutas que invocan a funciones contenidas en nuestro controlador.

```

//Rutas
app.use('/', users);
app.use("/uploads", express.static(__dirname + '/uploads'), img);
img.route('/:id').get(Ctrlprofesor.getimg)
users.route('/loginalumno').post(Ctrlalumno.loginalumno);

```

7. **Start server** Arrancamos nuestra aplicación en el puerto 8000

```

app.listen(8080, '0.0.0.0', function() {
    console.log("Node server running on http://localhost:8080");
});

```

4.3.2. Estructura de la base de datos

Como bien sabemos MongoDB es una base datos no relacional, es decir no es como las típicas bases de datos SQL donde existen relaciones entre una tabla y otra.

La estructura de la base de datos que se ha elaborado para la aplicación de este TFG consiste en 4 modelos, los cuales se relacionan dos a dos por medio de referencias y el método *populate* en MongoDB.

Analizamos la estructura de los modelos:

- **Modelo Alumnos** Consiste en un modelo simple donde tenemos tres campos predefinidos de tipo string.

```

var alumnoSchema = new Schema({
    nombre:      { type: String },
    apellidos:   { type: String },
    edad:        { type: String }
});
module.exports = mongoose.model('Alumno', alumnoSchema);

```

- **Modelo Login Alumnos** Este modelo encapsula dentro de él al anterior, y lo hace a partir de una llamada de referencia. Dentro del campo *data* tendremos el modelo del alumno.

```
var loginSchema = new Schema({
  email: { type: String },
  password: { type: String },
  data: { type: Schema.ObjectId, ref: "Alumno" },
});
module.exports = mongoose.model('LoginAlumno', loginSchema);
```

- **Modelo Profesores** Este modelo corresponde al profesor.

```
var profesorSchema = new Schema({
  nombre: { type: String },
  apellidos: { type: String },
  edad: { type: String },
  curso: { type: String, enum: ['Primaria', 'ESO', 'Bachillerato', 'Universidad', 'FP', 'EXAMENES LIBRES', 'FRACASO ESCOLAR'] },
  asignaturas: { type: String },
  location: {
    type: { type: String },
    coordinates: {type: []}
  },
  path: {type: String},
  notification: {
    type: [
      alumno: { type: Schema.ObjectId, ref: "Alumno" },
      leido: {type: Boolean},
      _id: false
    ]
  }
});
```

- **Modelo Login Profesores** Modelo que vuelve a anidar otro modelo en el campo *data*.

```
var loginSchema = new Schema({
  email: { type: String },
  password: { type: String },
  data: { type: Schema.ObjectId, ref: "Profesor" },
});
module.exports = mongoose.model('LoginProfesor', loginSchema);
```

La idea de tener estos modelos relacionados es que puede no interesar enviar toda la información en una llamada. Es decir, si un usuario introduce su *email* y su *password* en la ventana de login, no necesitaríamos buscar entre todos los datos de los usuarios, simplemente con tener el *email* y la *password* para hacer la verificación sería más que correcto.

4.3.3. Controladores

Un controlador es un archivo donde tenemos diversas funciones que son invocadas a partir de las rutas que tenemos configuradas en el *server.js*. Dependiendo del modelo de la base de datos que utilicemos para guardar, editar o eliminar datos, he decidido organizar las funciones en tres controladores diferentes:

- **Controllers Alumnos** Es el fichero en el que están todas las funciones que usan los modelos *alumno.js* y *loginalumno.js*

1. **registeralumno:** Función cuyo comportamiento consiste en comprobar que el email que introduce el alumno al registrarse no está en nuestra base de datos, y que los campos *password* y *email* no están vacíos, en tales casos el servidor devolverá un 400 al cliente.

Si el alumno es registrado con éxito se guardará en la base de datos y se enviarán las credenciales con un 201 en forma de *token* para una mayor seguridad.

```
alumno.save(function(err, datasave) {
  if(err) return res.send(500, err.message);
  var profile = _.pick(req.body, 'Email', 'Password', 'extra');
  profile.id = datasave.data;
  res.status(201).send({ id_token: createToken(profile) });
});
```

2. **loginalumno:** Si el alumno ya ha sido registrado en nuestra base de datos, y quiere hacer login, esta función será invocada y comprobará que el *email* y *password* del alumno coinciden con los credenciales. En caso de ser aceptado se le enviarán sus credenciales en forma de *token* con un 201 y en caso de ser rechazado se le enviará un 400 con el mensaje: *The username or password don't match*.

- **Controllers Profesores** Es el fichero en el que están todas las funciones que usan los modelos *profesor.js* y *loginprofesor.js*.

1. **registerprofesor:** El comportamiento es idéntico a *registeralumnos*, con la única salvedad de que el modelo que utilizamos es *profesor.js*.
2. **loginprofesor:** También mismo comportamiento que en *loginalumnos*, pero utilizando el modelo de datos de *loginprofesor.js*.
3. **savenotificacion:** Cuando un alumno quiere contactar con un profesor vía chat, antes tiene que enviarle una petición de contacto. Esto lo proporciona *savenotification*, que se encarga de almacenar en la base de datos del profesor los alumnos que le han enviado una petición de contacto.

```
exports.savenotificacion = function(req, res) {
  DataProfesor.findOneAndUpdate(
    {_id: req.body._id},
    {$addToSet: {notification: {alumno: req.body.id, leido:
      false, _id: false}}},
    {safe: true},
    function(err, model) {
      if (err == null) {
        res.status(200).send("La notificacion ha sido
          recibida");
```

```

        } else{
            res.send(500, err.message);
        }
    }
);
};

```

4. **readynotificacion:** Es una función que se encarga de comprobar si el profesor ha aceptado o rechazado la solicitud de contacto del alumno. En caso de ser aceptado, el chat se habilitará y el alumno y el profesor podrán tener un primer contacto.
5. **getallprofesores:** Función que se encarga de enviar al cliente todos y cada uno de los profesores que integran Classcity sin ningún tipo de requisito.
6. **getimg:** Como podemos ver en el código del siguiente ejemplo. Es una función muy simple que se encarga de enviar al cliente la imagen que solicita.

```

exports.getimg = function(req, res) {
    res.sendFile('uploads/' + req.params.id)
};

```

7. **postimg:** Función que se encarga de actualizar las imágenes de los profesores que editan su perfil.

```

exports.postimg = function(req, res) {
    ProfesorScheme.findOne({ "email" : req.file.originalname},
        function(err, data) {
            DataProfesor.findById(data.data, function(err, dataext) {
                dataext.path = req.file.path;
                dataext.save();
            });
        });
    res.end('File is uploaded');
};

```

8. **queryprofesores:** Esta función es la mas compleja de todas. Su objetivo consiste en filtrar los profesores que encajen con la solicitud. Es decir, un alumno puede buscar a su profesor por tres argumentos diferentes: Curso, Asignatura y Distancia. Por este motivo hacemos un *find* con tres argumentos de entre los que destaca el argumento Location. MongoDB permite hacer busquedas tan impresionantes como ésta, donde tenemos una base de datos de ubicaciones de profesores más la ubicación que introduce el alumno, y somos capaces de devolver aquellos profesores que se encuentren a un radio de él.

```

exports.queryprofesores = function(req, res) {
    DataProfesor.find({ "curso" : req.body.Curso, "asignaturas": req.body.Clase,
        location:{$geoWithin:{$centerSphere: [ [ req.body.Loc.lat, req.body.Loc.lng],
            req.body.Radio / 6378100 ] } } }, function(err, dataprod) {
            res.status(200).jsonp(dataprof);
        });
};

```

```
};
```

9. **getdetail:** La función que se encarga de encontrar el perfil del profesor que el alumno solicita.

```
exports.getdetail = function(req, res) {
    DataProfesor.findOne({ "_id" : req.params.id}, function(err,
        dataprop) {
        res.status(200).send(dataprof);
    });
};
```

- **Controllers Socket** *Socket.js* es el fichero que se encarga de gestionar el chat en la parte del servidor. Lo primero es que el servidor escuche en el puerto 8000.

```
server.listen(8000, '0.0.0.0');
```

Una vez que el servidor está escuchando en el puerto 8000 debemos utilizar la librería *io* para establecer la conexión con el usuario que intenta tener la comunicación.

```
io.on('connection', function(socket) {}
```

Como tenemos que manejar tantos hilos de chat como profesores tengamos, necesitamos un control de canales. Por eso cada 'sala' nueva viene asociado con el identificador de cada profesor. Es decir cuando un profesor se registra, un nuevo canal es creado y los alumnos tienen la oportunidad de poder hablar con el profesor en esa 'sala' sin que otros profesores tengan constancia de ello.

Las 'salas' son cada uno de los canales abiertos en la comunicación del chat, donde los alumnos son libres de elegir a que 'sala' entrar. Cada vez que un alumno entra en el perfil de un profesor, entra en una 'sala' donde sólo los que estén en el perfil del profesor podrán enterarse de lo que se comente por esa 'sala'.

```
socket.on('room', function(_room) {
    room = _room.roomName;
    user = _room.userName;
    socket.join(room);
    if (room in rooms)
        rooms[room]++;
    else
        rooms[room] = 1;
    io.to(room).emit('intro', {'userName': user, 'text': "ha entrado
        en la sala"});
});
```

Cuando un alumno o un profesor que ya se encuentran en una 'sala' concreta empiezan a enviarse mensajes, la forma que tenemos para gestionarlo es la siguiente:

1. El mensaje enviado por el emisor es recibido por el servidor
2. El servidor analiza el mensaje enviado por el emisor y lo trata reconociendo a que 'sala' pertenece.

3. El servidor reenvía el mensaje a todo el mundo que se encuentre en esa 'sala', excluyendo al emisor.

```
socket.on('newMessage', function(_room) {
    user = _room.userName;
    text = _room.text;
    io.to(room).emit('message', {'userName':
        user, 'text': text});
});
```

En caso de que alguno de los integrantes de la 'sala' decida abandonar el chat, realizaremos los siguientes pasos:

1. Recibiremos la desconexión del usuario que abandonó el chat.
2. A continuación informaremos al resto de los integrantes de la 'sala' que usuario en concreto abandonó la sala.
3. Y en el caso de que todos los integrantes incluyendo el profesor no se encuentren ya en la sala, es decir que este vacía, la eliminaremos de nuestros datos de control.

```
socket.on('disconnect', function() {
    leaveRoom();
});

var leaveRoom = function() {
    rooms[room]--;
    io.to(room).emit('client left', {'userName': user, 'text': "dejo
        la sala"});
    if (rooms[room] === 0)
        delete rooms[room];
};
```

4.4. Casos de usos