

# Capítulo 4

## Diseño e implementación

### 4.1. Introducción

este capítulo

En esta sección describiremos más en profundidad el diseño y la implementación realizados en el proyecto. Antes, comenzaremos con una breve introducción al mismo para tener una visión más global y poder entender más adelante las partes por separado.

de la aplicación de gestión de clases particulares, tanto su lado cliente como su lado servidor,

constituyentes

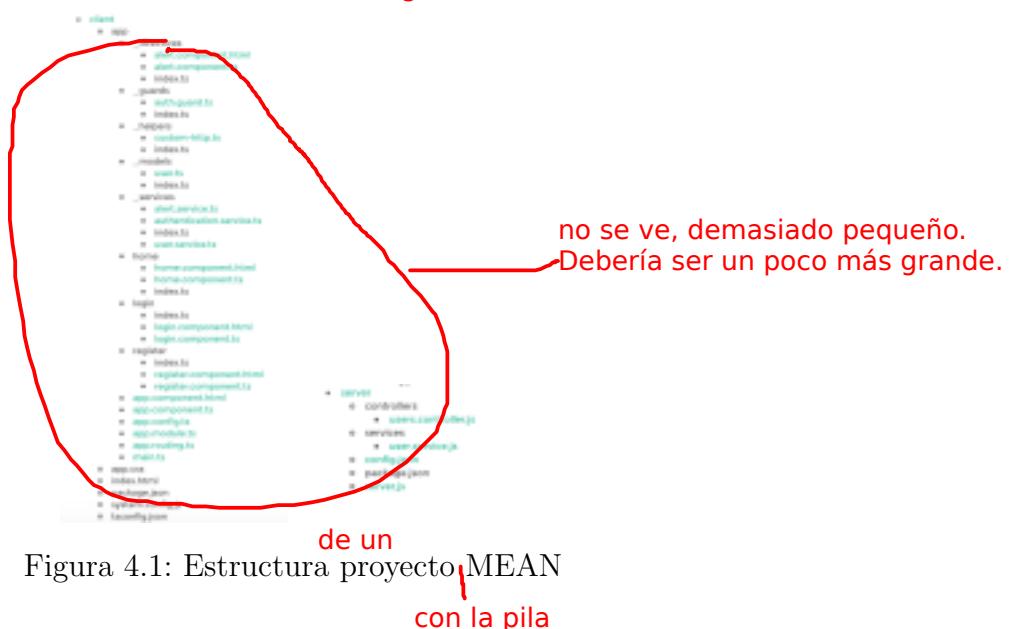
### 4.2. Estructura básica

Diseño

Como he explicado en el capítulo anterior 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 utilizan el stack MEAN, siguen una estructura similar a la siguiente:

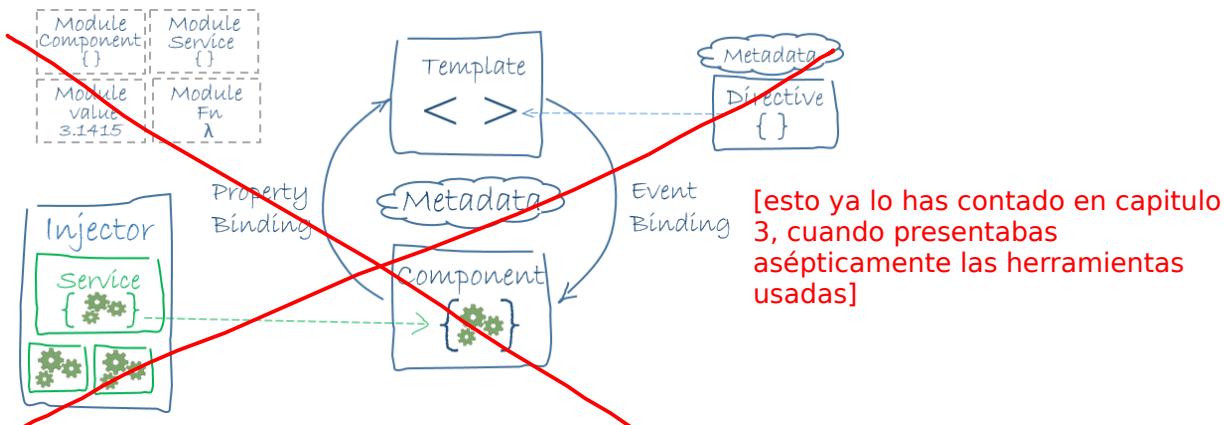
la pila  
de la figura 4.1



## Lado cliente de la aplicación web

### 4.3. ~~Front-End: Angular~~

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.



se explicó en el  
capítulo 3

Figura 4.2: Angular Diagrama

#### 4.3.1. ~~Modulos~~

Como ya sabemos las aplicaciones en Angular son modulares y un módulo es el conjunto de código dedicado a cumplir un único objetivo, en esta sección vamos a hablar de 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 templates 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 al 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 módulo, 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 que desarrollamos para nuestra aplicación, cada uno con su función independiente que contaremos a continuación.

[ dónde los cuentas?? son los componentes de 4.3.2? dilo entonces así]

### 4.3.2. Componentes

Continuamos con los componentes de nuestra aplicación, como ya sabemos los componentes son como etiquetas nuevas, que podemos inventarnos para realizar las funciones que sean necesarias para nuestro negocio.

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

de gestión de clases particulares

#### AppComponent

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```

    styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'ClassCity';
}

```

El selector app-root o el nombre de la etiqueta que se usará cuando se desee 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. ([Figura 4.3](#))
- **LoginAlumno, LoginProfesor** Componentes encargados de realizar la función de login del alumno o del profesor. Hemos desarrollado una función que se encarga de realizar una petición POST al servidor, si la respuesta es aceptada se almacenarán las credenciales en el "localStorage" con un timeout 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" itálica
- **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 body donde se encuentran los datos personales del profesor o el alumno.
- **HomeAlumno, HomeProfesor** Cuando un profesor o un alumno es aceptado dentro de nuestra base de datos y consigue entrar en el home de la aplicación, puede realizar diferentes funciones dependiendo de si entra como alumno o como profesor.
  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** El home del profesor consiste en un chat realizado con websockets donde el profesor podrá entablar conversación con cualquier alumno que esté interesado en él. Aparte de poder personalizar su perfil cambiando la foto que cada profesor tiene como avatar.
- **ProfesorDetail** Como último componente tenemos ProfesorDetail, cuando un alumno encuentra a su profesor particular ideal desde el home 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.

#### 4.3.3. ~~Templates~~ Plantillas

Como bien sabemos, los templates son las plantillas que se utilizan para dar forma a las aplicaciones. Es la parte más visual de una aplicación web y es la propia magia de Angular quien 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 templates que forman la aplicación: **de gestión de clases particulares** **plantillas** **la URL de la aplicación [ponla como footnote]** **intro.html** Cuando accedemos a <https://www.classcity.tk>, la primera plantilla que se nos presenta es Intro.html. Según podemos ver en la imagen, consiste en una página introductoria donde podemos elegir si somos alumnos o profesores.



Figura 4.3: Página Intro ClassCity

**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**.

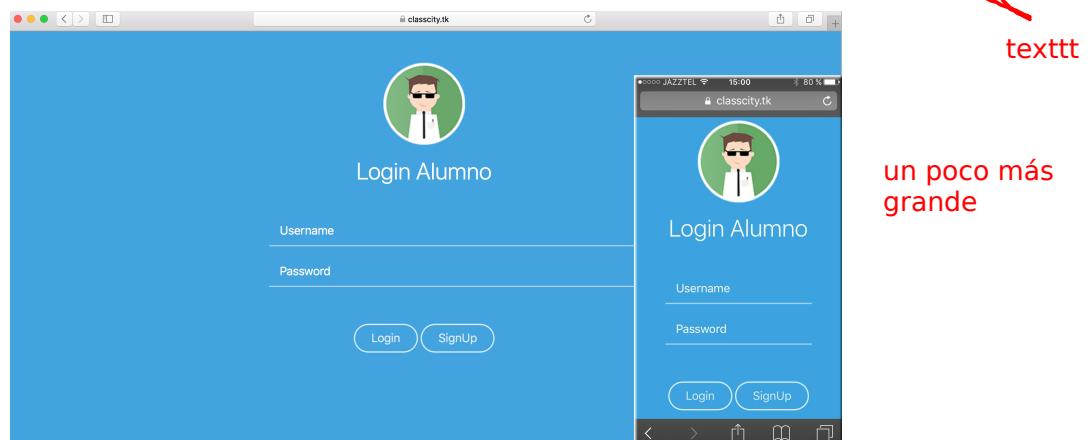


Figura 4.4: Página Login Alumnos

**loginprofesor.html** Si en vez de seleccionar Alumno hubiésemos seleccionado Profesor, hubiésemos entrado en otro formulario donde los profesores ya registrados pueden acceder a la aplicación.

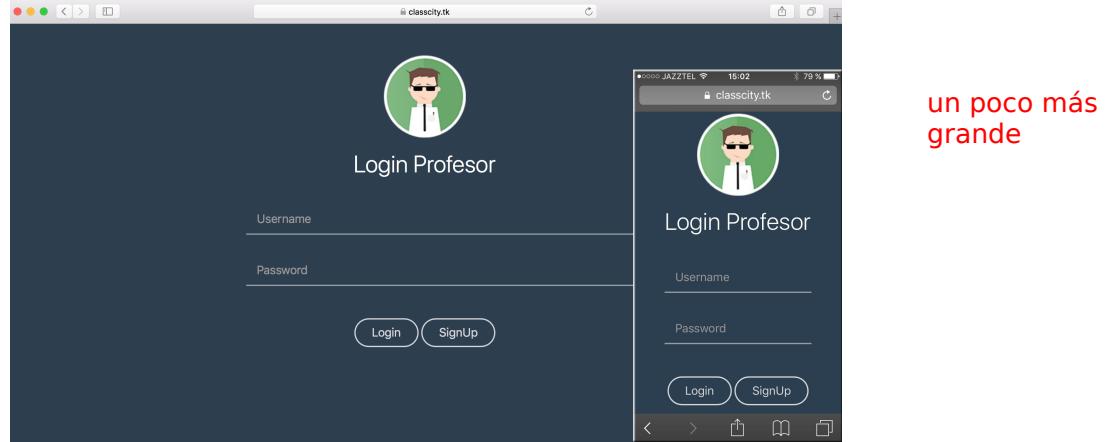


Figura 4.5: Página Login Profesor

**registeralumno.html** Si un alumno quiere registrarse simplemente debe entrar en SignUp, donde tendrá un formulario para poder completar todos los campos necesarios.

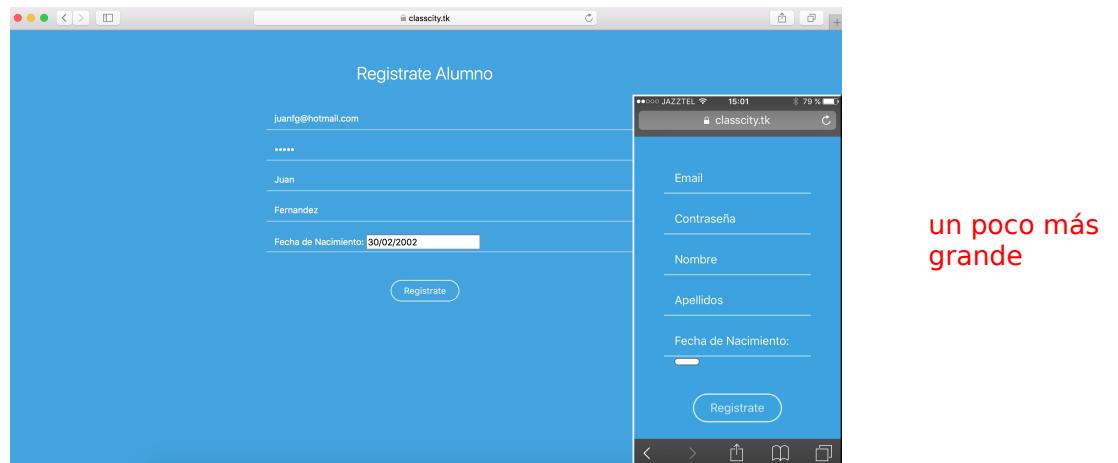


Figura 4.6: Página Registrar Alumno

**registerprofesor.html** Si es el profesor ~~es~~ quien quiere registrarse en nuestra aplicación, entrará en SignUp de profesores e introducirá los datos necesarios.

un poco más grande

Figura 4.7: 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 siguiente imagen. Dónde 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 deseé, les aparecerán pintados en el mapa tantos profesores como haya en nuestra aplicación con esas especificaciones.

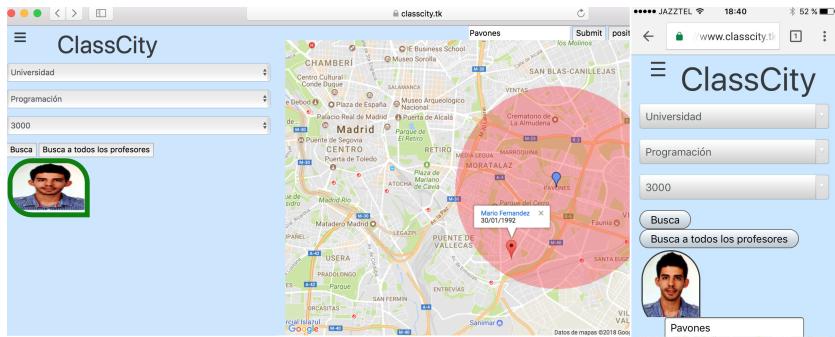
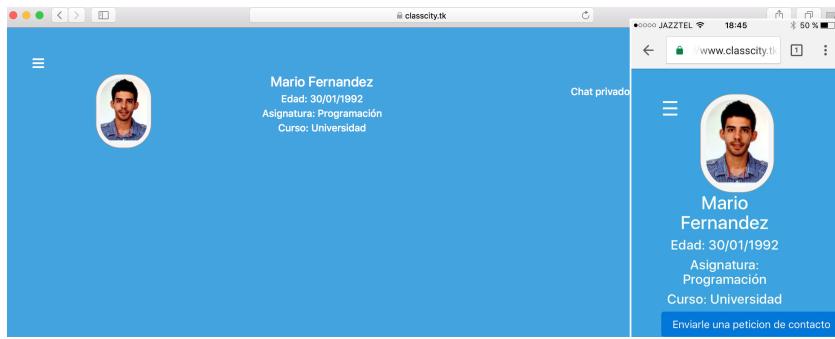


Figura 4.8: Página Home Alumno

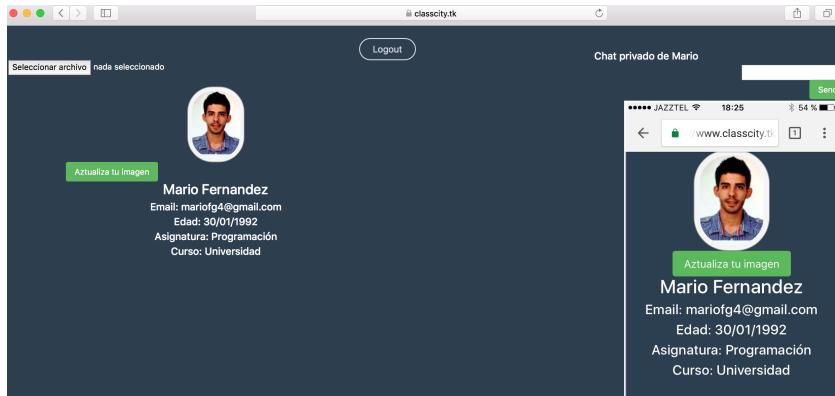
**detail.html** Cuando el alumno encuentra algún profesor de su interés, puede hacer click sobre la imagen del profesor y así poder entrar en más detalle, viendo su ficha técnica y pudiendo establecer una conversación a partir del chat.



un poco más grande

Figura 4.9: Página Detalle del Profesor

**la página** **homeprofesor.html** Cuando el profesor ya ha sido registrado en nuestra aplicación, **el home** del profesor es habilitado **y** en **ella** 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.



un poco más grande

Figura 4.10: Página Home Profesor

#### 4.3.4. Metadatos ~~X~~

#### 4.3.5. Data Binding ~~X~~

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.

**capítulo**

Como he comentado en el tema anterior Angular tiene 4 formas de Data Binding, **se ha** **1.** **itálica** **todas ellas se han** **las cuales hemos** 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 arbol DOM, todos los datos del usuario, consiguiendo una apariencia de la pagina mucho mas 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 ~~nosotros~~ queramos de nuestro componente padre ('name') 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 del ~~input~~ como en el caso de property binding.  
*la entrada*

#### 4.3.6. Directiva:

*una plantilla asociada*

Como ya he comentado las directivas son como los componentes, pero sin un template asociado. En nuestro caso hemos utilizado directivas para realizar ~~según que varias~~ funciones, por ejemplo:

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

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

*interfaz de programación*

**sebm-google-map, sebm-google-map-marker** Estas son otras de las directivas procedentes del módulo "agmCoreModule", el cual nos proporciona toda la ~~api~~ 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>

```

hemos usado

También ~~tenemos que hablar de los~~ 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.

```

$<$ y $>
<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>

```

#### 4.3.7. Servicio

Como ya sabemos los servicios se definen a través de simples clases y son imprescindibles en Angular, ya que toda función o valor es encapsulado 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 a la 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

mete un párrafo de 'vaselina' introduciendo las secciones que conforman el lado servidor: servidor Node, entorno express y base de datos Mongo.

### 4.4. Node

#### 4.3.1 Lanzamiento de node

De los 4 grandes bloques he decidido empezar con Node, ya que es imprescindible para poder construir el proyecto.

Como ya he comentado en el capítulo anterior, Node es un sistema innovador, puesto que es la plataforma encargada del funcionamiento del servidor, y funciona totalmente con JavaScript. como bien es sabido Javascript es un lenguaje de programación que en un principio era dedicado a correr en los navegadores, o en el lado del cliente, pero como se puede ver con el MEAN stack y sus subsistemas el uso de JavaScript se ha ampliado considerablemente a todos los aspectos de una página web.

Cabe destacar, sin duda, la irrupción en los últimos años de servidores interactivos asíncronos, como es el caso de NodeJS. Este tipo de servidores realizan tareas de forma asíncrona bloqueante, por lo que el tiempo de respuesta es, en muchos casos, menor que en servidores multihilos(servidores concurrentes). Por estos motivos, en la actualidad, múltiples proyectos enfocados al tiempo real, utilizan este tipo de servidores asíncronos.

De la mano a Node tenemos NPM, es el manejador de paquetes por defecto para Node.js. Para poder arrancar cualquier proyecto en MEAN, debemos utilizar npm para bajarnos todas las dependencias del proyecto.

Una vez tengamos Node y NPM instalados, lo utilizaremos en los siguientes casos:

- Instalar dependencias(Cliente y Servidor)

```
sudo npm install
```

- Correr nuestro servidor

```
sudo node server.js
```

- Arrancar el cliente:

```
sudo npm start
```

### 4.5. Back-End: Express - MongoDB

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.

Comenzaré explicando el backend con el primer fichero que ejecutamos cuando queremos arrancar nuestro servidor Server.js

esto lo has  
debe contado  
ya antes, en  
cap1 o cap3

esto fusionalo  
con la sección  
de node en  
el cap3

a entradilla  
de 4.3

## 4.3.2

ya lo has contado en 4.3.1

~~4.5.1.~~ Server.js

Para poder empezar analizando el código implementado en nuestro server.js, tenemos que saber como arrancar nuestro servidor.

```
sudo node server.js
```

Gracias a Node, simplemente con una linea en la terminal seremos capaces de generar procesos que escuchen en el puerto que queramos.

A continuación vamos a enumerar todas 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 data base 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, 'connection error:'));
db.once('open', function() {
  console.log('Connected to Database');
});
```

## cuerpos de las peticiones

2. **Parser body** Analizamos los ~~body~~ de las request en un ~~middleware~~ antes que llegue a sus manejadores.

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

3. **control 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 ingestión 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);
  }
});
```

```

        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 ~~que mas tarde explicaremos~~.

```

//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");
});

```

#### 4.5.2. Estructura de la base de datos

*se ha para la aplicación de este TFG*

~~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 he elaborado consiste en 4 modelos, los cuales se relacionan dos a dos por medio de referencias y el metodo *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 porque puede no interesarlos enviar toda la información en una llamada, es decir, si un usuario hace introduce su email y su password en la ventana de login, no necesitaríamos buscar entre todos los datos de los usuarios, simplemente con tener un modelo con el login y la password de los usuarios para poder hacer la verificación sería más que correcto.

#### 4.5.3. 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 utilizaremos 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 lo que quiero es hacer login, esta función será invocada y comprobará que el email y la 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. De esto consiste la función savenotificacion, 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, 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:** Continuamos con la función más compleja de todas. Su misión consiste en filtrar los profesores que encajen con la solicitud, es decir como he comentado al principio, un alumno puede buscar a su profesor por tres argumentos diferentes: Curso, Asignatura y Distancia. Por este motivo si nos fijamos en la función, hacemos un `find` con tres argumentos de entre los que destaca el argumento `Location`. La magia de MongoDB nos permite hacer `queries` 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 devolverle 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, dataprof)
{
    res.status(200).jsonp(dataprof);
});
};

```

9. **getdetail:** Por último concluimos con 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
        , dataprof) {
        res.status(200).send(dataprof);
    });
};

```

**texttt**

- **Controllers Socket** Socket.js es el fichero que se encarga de gestionar el chat en la parte del servidor. Si comenzamos analizando el código de socket.js, lo primero que hacemos es que el servidor escuche en el puerto 8000.

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

**texttt**

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 room nueva viene asociada 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 room sin que otros profesores se tengan constancia de ello.

Las room son cada uno de los canales abiertos en la comunicación del chat, donde los alumnos son libres de elegir a que room entrar. Cada vez que un alumno entra en el perfil de un profesor, entra en una room donde solo los que estén en el perfil del profesor podrán enterarse de lo que se comente por esa room. 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 'room' 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 'room' pertenece.
3. El servidor reenvía el mensaje a todo el mundo que se encuentre en esa 'room', 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 'room' decida abandonar el chat, realizaremos los siguientes pasos:

1. Recibiremos el ~~la desconexión~~ del usuario que abandona la conexión ~~a la sala~~ ~~el chat~~
2. A continuación informaremos al resto de los integrantes de la 'room' que ~~la~~ usuario en concreto abandonó ~~la sala~~ ~~sala~~
3. Y en el caso de que todos los integrantes incluyendo el profesor no se encuentren ~~ya~~ en la sala, 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

Metería una sección "Casos de uso" resaltando (presumiendo y mostrando imágenes) las funcionalidades:  
(a) búsqueda en el mapa de profesores cercanos, (b) perspectiva desde el profesor de que le han contactado varios estudiantes, (c) chat, que se vean los dos lados, un ejemplo concreto.