



**Universidad
Rey Juan Carlos**

Grado en Ingeniería en Sistemas Audiovisuales y Multimedia

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2017-2018

Trabajo Fin de Grado

**Aplicación web para gestión de clases
particulares**

Autor: Mario Fernández Guerrero

Tutor: José María Cañas Plaza

Curso académico 2017/2018

Dedicatoria

*'No me juzgues por mis éxitos,
júzgame por las veces que me caí y volví a levantarme'.*

By Nelson Mandela

Agradecimientos

*A mis padres, Mario y Rosa,
a mi hermano Alejandro y a mi novia Cristina,
por su apoyo y confianza sin límites.*

Gracias.

Resumen

En este trabajo de final de carrera se ha desarrollado una aplicación web cuya funcionalidad principal es ofrecer el contacto y la comunicación entre alumnos y profesores para poder impartir clases particulares.

El trabajo perseguía principalmente el objetivo de realizar una aplicación web como desarrollador full stack, desarrollando todo lo que una aplicación web necesita: Maquetación, Servicios, Bases de datos, Despliegue....

Este TFG sirve como apoyo para todos aquellos alumnos que tengan una idea y quieran desarrollarla en forma de aplicación web. Algunas de las tecnologías empleadas son JavaScript como lenguaje principal tanto en el cliente como en el servidor, Base de datos como MongoDB y Websockets como tecnología de comunicación, aparte del despliegue en la red con Amazon Web Services.

Índice general

| | |
|--|-----------|
| Índice de figuras | 7 |
| 1. Introducción | 1 |
| 1.1. Tecnologías web | 4 |
| 1.2. Tecnologías del Cliente | 7 |
| 1.2.1. HTML 5 | 7 |
| 1.2.2. ECMAScript 6 | 7 |
| 1.2.3. CSS3 | 8 |
| 1.2.4. Frameworks | 9 |
| 1.3. Tecnología del Servidor | 9 |
| 1.4. Computación en la nube | 10 |
| 1.5. Antecedentes | 12 |
| 2. Objetivos | 14 |
| 2.1. Metodología | 15 |
| 2.2. Plan de trabajo | 16 |
| 3. Infraestructura | 17 |
| 3.1. Angular | 18 |
| 3.1.1. Módulo | 19 |
| 3.1.2. Componente | 19 |
| 3.1.3. Plantilla | 19 |
| 3.1.4. Metadatos | 20 |
| 3.1.5. Data Binding | 20 |
| 3.1.6. Directiva | 21 |
| 3.1.7. Servicio | 21 |
| 3.1.8. Inyección de Dependencias | 21 |
| 3.2. Node | 22 |
| 3.3. MongoDB | 22 |
| 3.3.1. Características | 23 |
| 3.3.2. Documento en MongoDB | 24 |
| 3.3.3. Inconvenientes de MongoDB | 24 |
| 3.3.4. Fragmentación (<i>Sharding</i>) | 25 |
| 3.4. Express | 25 |
| 3.4.1. Introducción | 25 |
| 3.4.2. Estructura de una API | 27 |
| 3.5. Amazon web services | 28 |

| | |
|--|-----------|
| 4. Diseño e implementación | 29 |
| 4.1. Diseño | 29 |
| 4.2. Lado cliente de la aplicación web | 30 |
| 4.2.1. Modulos: | 30 |
| 4.2.2. Componentes: | 31 |
| 4.2.3. Plantilla | 32 |
| 4.2.4. Data Binding | 36 |
| 4.2.5. Directiva: | 37 |
| 4.2.6. Servicio | 38 |
| 4.3. Lado servidor de la aplicación | 38 |
| 4.3.1. Server.js | 39 |
| 4.3.2. Estructura de la base de datos | 40 |
| 4.3.3. Controladores | 42 |
| 4.4. Casos de usos | 45 |
| 5. Despliegue en la nube | 46 |
| 5.1. Lanzar una instancia en AWS | 47 |
| 5.2. Lanzar una aplicación MEAN a producción | 51 |
| 6. Conclusiones | 54 |
| 6.1. Conclusiones | 54 |
| 6.2. Trabajos Futuros | 55 |
| Bibliografía | 56 |

Índice de figuras

| | |
|---|----|
| 1.1. Ejemplo web 1º generación | 1 |
| 1.2. Ejemplo web 2º generación | 2 |
| 1.3. Ejemplo web 3º generación | 2 |
| 1.4. Aplicación Airbnb | 3 |
| 1.5. Aplicación Blablacar | 3 |
| 1.6. Aplicación Wallapop | 4 |
| 1.7. Modelo cliente-servidor | 4 |
| 1.8. api-rest | 6 |
| 1.9. HTML5 Icono | 7 |
| 1.10. EcmaScript 6 Icono | 8 |
| 1.11. CSS3 Icono | 8 |
| 1.12. CSS3 Icono | 9 |
| 1.13. Proveedores más conocidos de computación en la nube | 11 |
| 1.14. TFG Edgar Barrero | 12 |
| 1.15. TFG Aitor Martinez Fernandez | 13 |
| 2.1. Logo ClassCity | 14 |
| 2.2. Esquema Metodología Cascada | 15 |
| 3.1. Esquema MEAN | 17 |
| 3.2. Icono de Angular | 18 |
| 3.3. Diagrama de Angular | 18 |
| 3.4. Data-Binding Diagrama | 21 |
| 3.5. Icono de Node | 22 |
| 3.6. Icono de MongoDB | 23 |
| 3.7. Sharding | 25 |
| 3.8. Icono de Express | 26 |
| 3.9. AWS | 28 |
| 4.1. Estructura de un proyecto con la pila MEAN | 29 |
| 4.2. Página Intro ClassCity | 32 |
| 4.3. Página Login Alumnos | 33 |
| 4.4. Página Login Profesor | 33 |
| 4.5. Página Registrar Alumno | 34 |
| 4.6. Página Registrar Profesor | 34 |
| 4.7. Página Home Alumno | 35 |
| 4.8. Página Detalle del Profesor | 35 |
| 4.9. Página Home Profesor | 36 |

| | | |
|-------|---|----|
| 5.1. | Crear Key Pairs Paso 1 | 47 |
| 5.2. | Crear Key Pairs Paso 2 | 48 |
| 5.3. | Lanza una instancia Paso 1 | 48 |
| 5.4. | Lanza una instancia Paso 2 | 49 |
| 5.5. | Lanza una instancia Paso 3 | 49 |
| 5.6. | Conéctate a tu EC2 Paso 2 | 50 |
| 5.7. | Conéctate a tu EC2 Paso 3 | 50 |
| 5.8. | Correo de confirmación | 51 |
| 5.9. | SSL | 51 |
| 5.10. | https://www.classcity.es | 52 |

Capítulo 1

Introducción

En este TFG se ha desarrollado una aplicación web, que pone en contacto profesores y alumnos para dar clases particulares, utilizando tecnologías web de última generación.

Actualmente la web se ha convertido en algo cotidiano entre los mas de 600 millones de usuarios que componen internet, suponiendo un impacto en la economía mundial incalculable. Todo empezó en 1990 cuando Tim Berners-Lee creó lo que hoy concebimos como web 1.0, una versión inicial de la web que nada tiene que ver con lo que conocemos hoy en día.

Nace como un sistema de hipertexto para compartir información en internet, con la finalidad de publicar documentos. Cuando las empresas empiezan a darse cuenta del potencial de la web, empiezan a incorporar información corporativa con la finalidad de ser más próximos a los clientes y como un canal mas para promocionarse. Pero pronto se dan cuenta de las limitaciones que la web 1.0 contiene:

- El contenido publicado no se actualizaba constantemente por lo que podían pasar largos periodos de tiempo sin que esa información se modificase.
- Las páginas eran estáticas y no permitían interacción de ningún tipo
- La principal desventaja es que eran difícil de manejar y solo podían publicar contenido los más entendidos en el tema o web masters.

World Wide Web

The WorldWideWeb (W3) is a wide-area [hypermedia](#) information retrieval initiative aiming to give universal access to a large universe of documents.

Everything there is online about W3 is linked directly or indirectly to this document, including an [executive summary](#) of the project, [Mailing lists](#),

[Policy](#), November's [W3 news](#), [Frequently Asked Questions](#).

[What's out there?](#)

Pointers to the world's online information, [subjects](#), [W3 servers](#), etc.

[Help](#) on the browser you are using

[Software Products](#)

A list of W3 project components and their current state. (e.g. [Line Mode](#) [X11 Viola](#) , [NeXTStep](#) , [Servers](#) , [Tools](#) , [Mail robot](#) , [Library](#))

[Technical](#)

Details of protocols, formats, program internals etc

[Bibliography](#)

Paper documentation on W3 and references.

[People](#)

A list of some people involved in the project.

[History](#)

A summary of the history of the project.

[How can I help?](#)

If you would like to support the web..

[Getting code](#)

Getting the code by [anonymous FTP](#) , etc.

Figura 1.1: Ejemplo web 1º generación

Debido a las limitaciones que ofrece la web 1.0, nace una nueva forma de concebir la web donde se valora las reacciones de los usuarios. Surgen aplicaciones y páginas que uti-

lizan la inteligencia colectiva, consecuencia de ello las páginas pueden ser personalizadas convirtiéndose en una herramienta dinámica que permite el intercambio de información. Es por eso que la información se transforma en comunicación gracias a la interacción y a la incorporación de textos, vídeos, chats... Con esta nueva forma de concebir la web nacen los blogs, las redes sociales, los wikis... Este cambio ha supuesto una gran revolución, puesto que permite devolver la información de los usuarios y poder procesarla con el objetivo de controlar mejor la demanda.



Figura 1.2: Ejemplo web 2º generación

La conocida web 3.0 dará paso a otro tipo de web donde pondrá su objetivo en la inteligencia artificial, un método para que los usuarios puedan no solo encontrar la información sino comprenderla. Este control está en manos de motores informáticos y procesadores de información, que tratan de analizar nuestro perfil y nuestra actividad en red para enviarnos información de nuestro interés. Es por esto que la web 3.0 es definida por el concepto "personalización", ya que pretende devolver al usuario una información lo más afinada posible, filtrada a sus gustos y preferencias, evitando información que no sea de su interés.

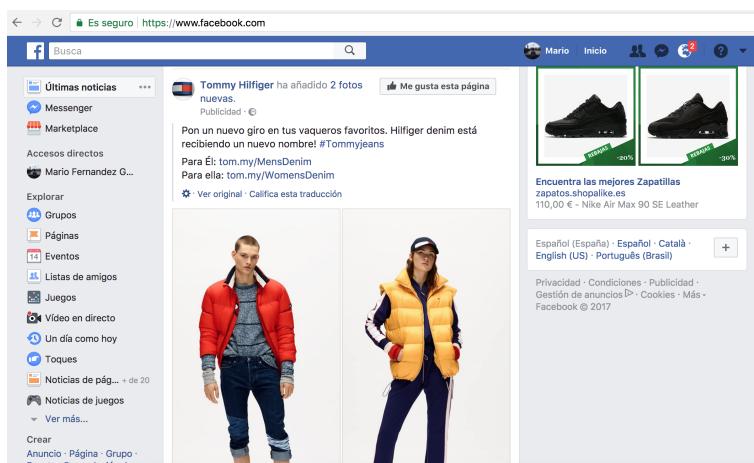


Figura 1.3: Ejemplo web 3º generación

La aplicación de este TFG ha sido desarrollada basándose en modelos de aplicaciones que hoy en día están funcionando, como por ejemplo:

Airbnb

Es una empresa y una plataforma de software dedicada a la oferta de alojamientos a particulares y turísticos. El nombre es un acrónimo de airbed and breakfast (colchón inflable y desayuno). Airbnb tiene una oferta de unas 2.000.000 propiedades en 192 países y 33.000 ciudades. Desde su creación en noviembre de 2008 hasta junio de 2012 se realizaron 10 millones de reservas.

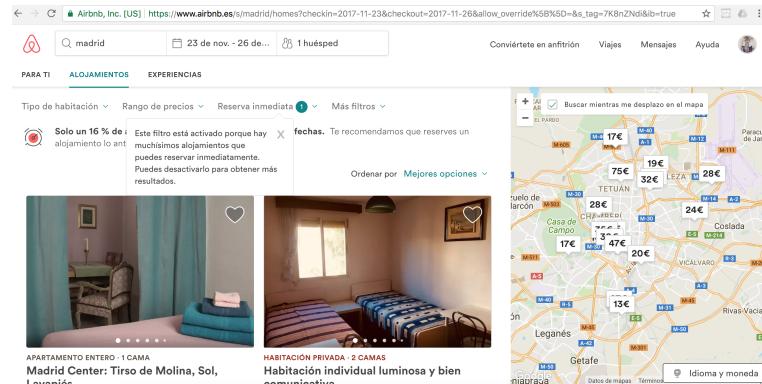


Figura 1.4: Aplicación Airbnb

Blablacar

Es un servicio de vehículo compartido que hace posible que las personas que quieren desplazarse al mismo lugar al mismo momento puedan organizarse para viajar juntos. Permite compartir los gastos puntuales del viaje (combustible y peajes) y también evitar la emisión extra de gases de efecto invernadero, al permitir una mayor eficiencia energética en el uso de cada vehículo.



Figura 1.5: Aplicación Blablacar

Wallapop

Es una empresa española fundada en 2013, que ofrece un website dedicado a la compra y venta de productos de segunda mano entre usuarios a través de Internet, con un uso centrado en smartphones. Utiliza la geolocalización para que los usuarios puedan comprar y vender en función de su proximidad geográfica



Figura 1.6: Aplicación Wallapop

1.1. Tecnologías web

Las aplicaciones web como acabamos de comentar han ido evolucionando a lo largo de la historia de internet, pero todas ellas se basan en un modelo cliente-servidor, es decir, para poder lograr la comunicación necesitamos un cliente, normalmente un navegador y un servidor web capaz de atender nuestras solicitudes.

Todas ellas se basan en el protocolo HTTP, el cual permite las transferencias de información en la World Wide Web y define la sintaxis y la semántica que utilizan los elementos de software de la arquitectura web (clientes, servidores, proxies) para comunicarse.

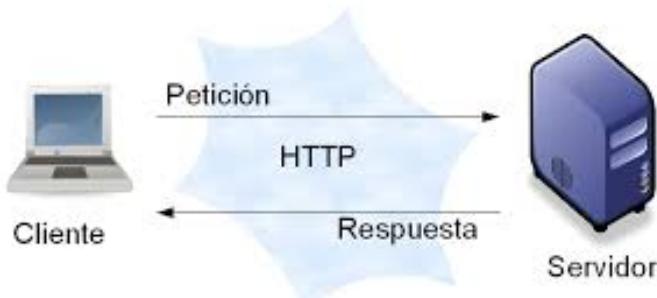


Figura 1.7: Modelo cliente-servidor

HTTP

HTTP (Hypertext Transfer Protocol) es el protocolo de comunicación que permite la transferencia de información en la red. Es un protocolo orientado a transacciones y que sigue un esquema petición-respuesta entre un cliente y un servidor.

HTTP define una serie predefinida de métodos de petición(verbos) que pueden utilizarse, teniendo la flexibilidad de ir añadiendo nuevos métodos con sus nuevas funcionalidades.

Entre todos los métodos de petición de HTTP destacamos los siguientes:

- **HEAD:** El método HEAD pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta.
- **GET:** El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.
- **POST:** El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.
- **PUT:** El modo PUT reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.
- **DELETE** El método DELETE borra un recurso en específico.

HTTP es un protocolo sin estado, es decir, no guarda ninguna información sobre conexiones anteriores. El desarrollo de aplicaciones web necesita frecuentemente mantener estado por lo que se utilizan las cookies, que es información que un servidor puede almacenar en el sistema cliente para simular la noción de la sesión.

API REST

API REST se define por ser un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP. La arquitectura de un sitio Web tiene tres componentes principales:

- **Un servidor Web:** Distribuye páginas de información formateada a los clientes que las solicitan. Los requerimientos son hechos a través de una conexión de red, y para ello se usa el protocolo HTTP.
- **Una conexión de red**
- **Uno o más clientes:** Una vez que se solicita esta petición mediante el protocolo HTTP y la recibe el servidor Web, éste localiza la página Web en su sistema de archivos y la envía de vuelta al cliente que la solicitó.

Para comprender el concepto API-REST, primero debemos entender el concepto API. Una API es una interfaz de programación de aplicaciones (del inglés API: Application Programming Interface), que en su conjunto de rutinas provee acceso a funciones de un determinado software.

REST(Transferencia de Estado Representacional) es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON. Es una alternativa en auge a otros protocolos estándar

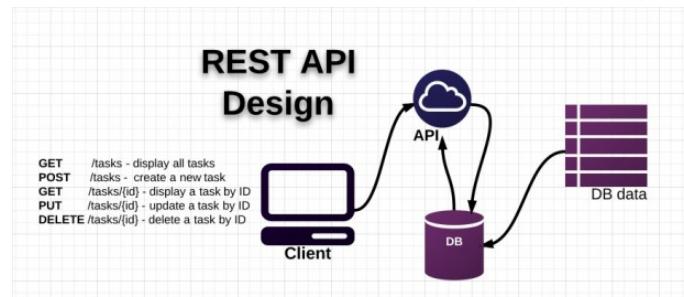


Figura 1.8: api-rest

de intercambio de datos como SOAP (Simple Object Access Protocol), que disponen de una gran capacidad pero también mucha complejidad. A veces es preferible una solución más sencilla de manipulación de datos como REST.

Las reglas que definen una API-REST son las siguientes:

- **Interfaz uniforme:** para la transferencia de datos en un sistema REST, este aplica acciones concretas (POST, GET, PUT y DELETE) sobre los recursos, siempre y cuando estén identificados con una URI.
- **Peticiones sin estado** cada petición HTTP contiene toda la información necesaria para ejecutarla, lo que permite que ni cliente ni servidor necesiten recordar ningún estado previo para satisfacerla.
- **Cacheable** existe la posibilidad de definir algunas respuestas a peticiones HTTP concretas como cacheables, con el objetivo de que el cliente pueda ejecutar en un futuro la misma respuesta para peticiones idénticas.
- **Separación de cliente y servidor**
- **Sistema de Capas** arquitectura jerárquica entre los componentes. Cada una de estas capas lleva a cabo una funcionalidad dentro del sistema REST.

La arquitectura API-REST donde las comunicaciones son más ligeras entre productor y consumidor, mantenibles y escalables, hacen de REST un estilo de construcción popular para APIs basadas en la nube, como las proporcionadas por Amazon, Microsoft y Google.

El estilo REST hace énfasis en que las interacciones entre los clientes y los servicios se mejoran al tener un número limitado de operaciones (verbos). La flexibilidad se obtiene asignando recursos a sus propios identificadores de recursos universales únicos (URI). Debido a que cada verbo tiene un significado específico (GET, POST, PUT y DELETE), evitando la ambigüedad.

- **GET** Se usa GET para obtener un recurso
- **POST** Se usa POST para crear un recurso en el servidor
- **PUT** Se usa PUT para cambiar el estado de un recurso o actualizarlo
- **DELETE** Se usa DELETE para eliminar un recurso

1.2. Tecnologías del Cliente

El Front-end se desarrolla normalmente en HTML, CSS o Javascript, lo cual implica que los programadores se especialicen en estos tres lenguajes. Además de que el código sea correcto, la web debe tener un diseño atractivo y funcional, que permita que la experiencia del usuario sea lo suficientemente cómoda, intuitiva y agradable para que continúe navegando.

1.2.1. HTML 5



Figura 1.9: HTML5 Icono

HTML5 es la quinta versión del lenguaje básico de la World Wide Web, publicado en Octubre de 2014. Al no ser reconocido en viejas versiones de navegadores por sus nuevas etiquetas, se recomienda al usuario común actualizar su navegador a la versión más reciente, para poder disfrutar de todo el potencial que provee HTML5.

HTML5 incluye significativas novedades en diversos áreas, ya que no incorpora solo nuevas etiquetas o elimina otras, sino que mejora áreas que estaban fuera del alcance del lenguaje:

- **Responsive:** Permite desarrollar aplicaciones que se adaptan fácilmente a distintas resoluciones, tamaños de pantallas, relaciones de aspectos y orientaciones.
- **Geolocalización:** Permite localizar gráficamente las páginas web por medio de una API de geolocalización.
- **Canvas:** Nuevo componente que permitirá dibujar en la página todo tipo de formas, que podrán estar animadas y responder a interacciones del usuario por medio de las funciones de un API.
- **WebSockets** tecnología que proporciona un canal de comunicación bidireccional y full-duplex sobre un único socket TCP.
- **Aplicaciones web Offline** API que permite el trabajo con aplicaciones web, que se podrán desarrollar para que funcione también en local y sin estar conectados a internet.

1.2.2. ECMAScript 6

ECMAScript es una especificación estándar de un lenguaje desarrollado por Brendan Eich. Inicialmente se llamaba Mocha, luego LiveScript, y finalmente Javascript. Debido al



Figura 1.10: EcmaScript 6 Icono

gran éxito de Javascript como lenguaje de scripting del lado del cliente para páginas web, Microsoft desarrolló un dialecto compatible del lenguaje llamado JScript, para evitar problemas legales con la marca. La primera versión de JavaScript, ECMA-Script 1, se lanzó en Junio de 1997, y desde entonces han existido las versiones 2, 3 y 5, que es la más usada actualmente (la 4 se abandonó). Sobre la versión de ECMA-Script 6, podemos decir que desde 2015 ya es un estándar cerrado, tratándose de una evolución del lenguaje JavaScript para dotarlo de características avanzadas que se echaban mucho en falta y que sí estaban disponibles en otros lenguajes populares, como por ejemplo:

1. **Mejoras de sintaxis:** parámetros por defecto, variables let, plantillas...
2. **Módulos para organización de código**
3. **Verdaderas clases para programación orientada a objetos**
4. **Promesas:** para programación asíncrona.
5. **Mejoras en programación funcional:** expresiones lamda, iteradores, generadores...

Una cuestión muy importante es que ECMA-Script 6 es totalmente compatible hacia atrás con versiones anteriores, por lo que no tenemos que preocuparnos por nuestro código actual, el cual funcionará perfectamente en motores de JavaScript que usen la próxima versión. Centrando el foco en el lenguaje de programación JavaScript, aparecen multitud de framework que facilitan su programación como son Angular2+, Vue.js y React entre otros.

1.2.3. CSS3



Figura 1.11: CSS3 Icono

CSS o Cascading Style Sheet (Hoja de estilos en cascada) es el lenguaje de diseño de la web. Su estandarización y especificación corre por parte del W3C, que lo incluyó a partir de la versión 4 de HTML. Gracias a este lenguaje, podremos indicar dónde se colocan los elementos, su color, apariencia, etc. Al tratarse de estilos en cascada, los estilos que definamos

en un elemento que contenga a otros, éstos podrán propagarse hacia abajo. Por ejemplo: Si cambiamos el tamaño de fuente al elemento <body>(Padre de todo el documento), todos los párrafos y links de la página tendrán ese tamaño de fuente a menos que indiquemos lo contrario. En esta última versión se han incluido grandes mejoras, podemos hacer uso de transformaciones 2D y 3D así como animaciones aceleradas por GPU, lo que hace que sean mucho más suaves y vistosas que programándolas como hasta ahora, en JavaScript.

1.2.4. Frameworks

Un Framework en desarrollo es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

A continuación se van a describir los principales frameworks de JavaScript para el lado del cliente que existen en la actualidad.

- **Angular:** es el framework estrella hoy en día en demanda de ofertas de trabajo y en comunidad detrás de él. La gran crítica de los desarrolladores sobre Angular es su gran curva de aprendizaje, ya que empezar con Angular implica tener unos amplios conocimientos de diferentes tecnologías.
- **React:** es la segunda gran apuesta en el desarrollo de aplicaciones del lado de cliente. Ha sido creado por Facebook para el desarrollo de interfaces de usuario en aplicaciones Web. Constantemente se compara React con Angular pero sus objetivos son diferentes: React no es un framework sino una biblioteca que se centra en crear interfaces de usuario, a diferencia de Angular, que trata de abarcar mucho más.
- **Vue.js:** El tercer framework en esta lista es el proyecto de código abierto denominado Vue.js, el cual ha tenido una gran popularidad en los últimos tiempos con la aparición de la versión 2.0 en 2016. Este framework trata de tomar lo mejor de cualquier framework e implementarlo, de hecho, en amplias comparativas entre diferentes frameworks ha conseguido unos resultados extraordinarios en velocidad y ligereza de peso

1.3. Tecnología del Servidor



Figura 1.12: CSS3 Icono

Un servidor web o servidor HTTP es un programa informático que procesa una aplicación del lado del servidor, realizando conexiones bidireccionales o unidireccionales y

síncronas o asíncronas con el cliente y generando o cediendo una respuesta en cualquier lenguaje o Aplicación del lado del cliente. El código recibido por el cliente es renderizado por un navegador web.

La principal razón para usar servicios Web es que se pueden utilizar con HTTP sobre Transmission Control Protocol (TCP) en el puerto de red 80. Dado que las organizaciones protegen sus redes mediante firewalls (que filtran y bloquean gran parte del tráfico de Internet), cierran casi todos los puertos TCP salvo el 80, que es, precisamente, el que usan los navegadores web. Los servicios Web utilizan este puerto, por la simple razón de que no resultan bloqueados. Es importante señalar que los servicios web se pueden utilizar sobre cualquier protocolo, sin embargo, TCP es el más común.

Frameworks

Los frameworks de lado servidor nos hacen más fácil escribir, mantener y escalar aplicaciones web. Los principales son:

- **Express:** es un framework web veloz, no dogmático, flexible y minimalista para Node.js. Proporciona un conjunto de características para aplicaciones web y móviles. Express es extremadamente popular, en parte porque facilita la migración de programadores web de JavaScript de lado cliente a desarrollo de lado servidor, y en parte porque es eficiente con los recursos
- **Django:** es un Framework Web Python de alto nivel que promueve el desarrollo rápido y limpio y el diseño pragmático. Es también veloz, seguro y muy escalable. Al estar basado en Python, el código de Django es fácil de leer y de mantener.
- **Ruby on rails:** es un framework web escrito para el lenguaje de programación Ruby. Rails sigue una filosofía de diseño muy similar a Django. Como Django proporciona mecanismos estándar para el enrutado de URLs, acceso a datos de bases, generación de plantillas y formateo de datos como JSON o XML.

1.4. Computación en la nube

Otra de las tecnologías usadas en este TFG es la computación en la nube. Esta computación en la nube es un paradigma que permite ofrecer servicios de computación a través de internet. Cuando los proveedores utilizan la palabra cloud, se refieren a la posibilidad de configurar y redimensionar los recursos que se usan de forma rápida y sencilla, o manualmente vía web o usando APIs REST.

Este tipo de servicios son tan dinámicos que se cobran por horas o minutos dependiendo de la plataforma de computación en la nube. Además los recursos de computación en la nube suelen estar virtualizados, aunque en algunas ocasiones pueden ser máquinas físicas.

Los proveedores de la computación en la nube ofrecen diferentes tipos de servicios, tanto de bajo nivel como de alto nivel.

1. **Servidores virtuales (Instancias)**
2. **Gestión del sistema operativo que tendrán los servidores (Imagen)**
3. **Sistemas de copia de seguridad de los servidores completos**



Figura 1.13: Proveedores más conocidos de computación en la nube

4. Balanceadores de carga entre servidores
5. Bases de datos administradas
6. Servicios de gestión de logs, monitorización y alarmas
7. Plataforma auto-escalable para ejecución de aplicaciones

Los servicios ofrecidos por los proveedores pueden ser de diferentes tipos de abstracción:

1. **Infraestructura como servicio (IaaS):** IaaS proporciona acceso a recursos informáticos situados en un entorno virtualizado, la 'nube' (cloud), a través de una conexión pública. En el caso de IaaS, los recursos informáticos ofrecidos consisten en infraestructura de procesamiento. La definición de IaaS abarca aspectos como el espacio en servidores virtuales, conexiones de red, ancho de banda, direcciones IP y balanceadores de carga.
2. **Plataforma como Servicio (PaaS):** El modelo PaaS permite a los usuarios crear aplicaciones de software utilizando herramientas suministradas por el proveedor. Los servicios PaaS pueden consistir en funcionalidades preconfiguradas a las que los clientes puedan suscribirse, eligiendo las funciones que deseen incluir para resolver sus necesidades y descartando aquellas que no necesiten.
3. **Software como Servicio (SaaS):** El modelo SaaS se conoce también a veces como "software bajo demanda", y la forma de utilizarlo se parece más a alquilar el software que a comprarlo. Con las aplicaciones tradicionales el software se compra al principio como un paquete, y una vez adquirido se instala en el ordenador del usuario. La licencia del software puede también establecer limitaciones en cuanto al número de usuarios y/o dispositivos en los cuales puede instalarse.

Cada vez son más las empresas que deciden utilizar la computación en la nube ya que desde 2006, , Amazon Web Services proporciona servicios de infraestructura de TI para empresas en forma de servicios web. Uno de los principales beneficios de la computación en la nube es la oportunidad de reemplazar importantes gastos de infraestructura con costes reducidos que se escalan dependiendo de la dimensión de su negocio. Gracias a la nube, las empresas ya no tienen que planificar ni adquirir servidores y otra infraestructura de TI con semanas o meses de antelación. Pueden disponer en cuestión de minutos de cientos o de miles de servidores y ofrecer resultados más rápidamente.

1.5. Antecedentes

EL TFG centra su desarrollo en el campo de las tecnologías Web y como a través de ella permiten generar diversas aplicaciones con las que el usuario puedan interactuar. A continuación, se presentan varios ejemplos de TFGs dentro del Laboratorio de Robótica de la URJC que emplean estas tecnologías y han sido el punto de partida de este TFG.

Prácticas docentes de desarrollo web

Este TFG[1][2] fué realizado por Walter Cuenca, con el objetivo de servir como apoyo en las prácticas de la asignatura de LTAW del grado de ingeniería de sistemas audiovisuales y multimedia. Consta de cuatro prácticas que utilizan diferentes tecnologías Web, herramientas y bibliotecas muy acentuadas en este campo. Algunas de las tecnologías empleadas son Javascript en el cliente, NodeJS y Django en el servidor, Base de datos como MySQL y por último Web-Socket y WebRTC como tecnologías de comunicación fluida.
<https://github.com/RoboticsURJC-students/2015-TFG-Walter-Cuenca>

Suveillance 5.1

Edgar barrero desarrollo en Ruby sobre Rails Suveillance 5.1 [3][4], una aplicación web que ofrece un flujo de vídeo desde una cámara web, un flujo de imagen de profundidad procedente de un sector Kinect y su representación en 3D, además de un sensor de humedad y un actuador.

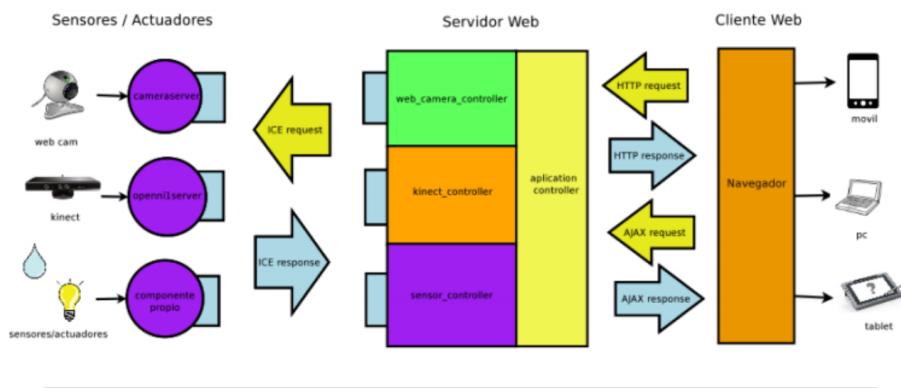


Figura 1.14: TFG Edgar Barrero

JdeRobotWebClients (URJC)

JdeRobotWebClients [5][6]fue desarrollado por Aitor Martínez Fernández como trabajo fin de grado. La aplicación Web consiste en crear seis versiones web de herramientas utilizadas por JdeRobot (CameraViewJS, RGBDViewerJS, KobukiViewerJS,) que estaban programadas en C++ o Python con su propio interfaz gráfico provocando que sean ejecutables solo en Linux. Estas nuevas versiones son multiplataforma (Linux, Android, IOS, Windows) y accesibles desde un navegador web como interfaz gráfico permitiendo acceder a los sensores y actuadores sin un servidor intermedio.

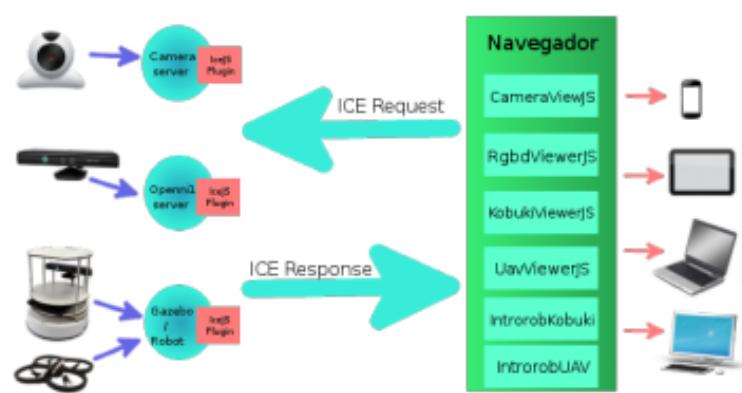


Figura 1.15: TFG Aitor Martinez Fernandez

Capítulo 2

Objetivos

Una vez que hemos enfocado el contexto en el que se va a desarrollar este trabajo, pasamos a definir el objetivo general y los sub-objetivos que se pretender cubrir en este TFG.

El objetivo principal es desarrollar Classcity, una aplicación web desarrollada con tecnología de última generación, cuya funcionalidad es facilitar el contacto entre alumnos y profesores para dar clases particulares. Classcity utiliza la geolocalización del alumno para proporcionarle los profesores más próximos a él, además de poder filtrar por diferentes parámetros como curso, asignatura y distancia.



Figura 2.1: Logo ClassCity

Para la realización de esta aplicación hemos decidido dividir nuestro objetivo en sub-objetivos mas sencillos con la finalidad de que quitemos complejidad al proyecto. Estos sub-objetivos son:

- **Front-End:** La parte del cliente es quizás la más tediosa por su dificultad a la hora de desarrollar una interfaz lo suficientemente ligera y adaptable para diferentes tipos de dispositivo. Es por esto que la elección del framework en el cliente nos puede cambiar por completo la estructura de nuestra aplicación web, además de reducir mucho los tiempos de desarrollo.
- **Back-End:** El segundo sub-objetivo es plantearnos cual de los diferentes frameworks del backend se adaptan mejor a nuestras necesidades. La elección de este framework viene condicionado en gran medida por el framework seleccionado para el cliente.
- **BBDD:** La base de datos por lo general pueden ser de dos tipos SQL y NoSQL, es por esto que debemos elegir cual de los dos tipos de bases de datos satisface mas con nuestras necesidades.

- **Despliegue en la nube:** Una vez que tengamos nuestra aplicación funcionando correctamente en local, es momento de desplegarlo en alguno de los cloud que nos ofrecen los proveedores más importantes como son: AWS, Azure o GoogleCloud

2.1. Metodología

En la realización del proyecto se ha necesitado definir una metodología que permita planificar las tareas necesarias para llegar a nuestro objetivo. El modelo seleccionado para la realización del TFG ha sido de tipo cascada, un proceso de desarrollo secuencial, en el que el desarrollo de software se concibe como un conjunto de etapas que se ejecutan una tras otra. Se le denomina así por las posiciones que ocupan las diferentes fases que componen el proyecto colocadas una encima de otra, y siguiendo un flujo de ejecución de arriba hacia abajo, como una cascada.

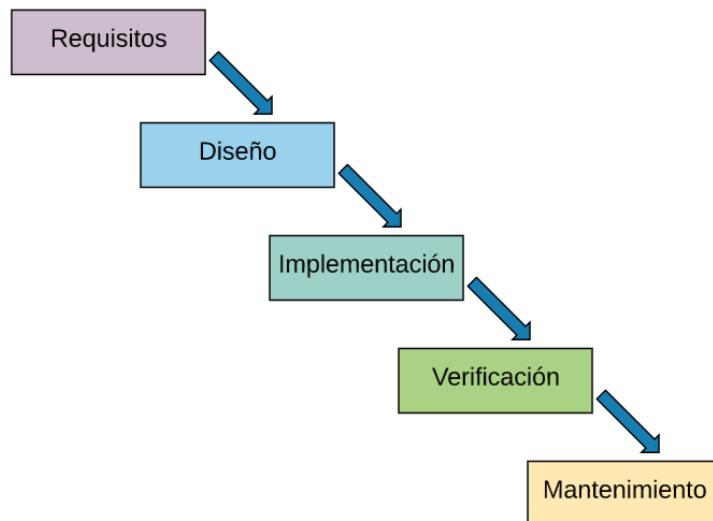


Figura 2.2: Esquema Metodología Cascada

Como parte de la metodología, durante el tiempo que ha durado el proyecto se acordaron reuniones semanales con el tutor de forma presenciales o por Vídeo-Conferencia en las que se revisaba los objetivos semanales y se definían los nuevos hitos.

También debemos destacar en la metodología el uso de Git como software de control de versiones. Git es un sistema de control de versiones de código libre y abierto, el cual está diseñado para controlar cualquier tipo de proyectos independientemente de su magnitud.

Git mantiene todos los commits que hagamos de nuestro proyecto. Un commit es una foto de nuestros archivos en un determinado momento. Estos incluyen un identificador, todos los cambios respecto al commit anterior y una referencia al mismo. De esta manera, siempre que queramos, podremos retroceder hasta una versión anterior de nuestro código. Por otro lado, permite tener varias versiones en paralelo o ramas de nuestros proyectos. Éstas son muy útiles para trabajos en equipo, ya que cada desarrollador puede implementar sus funcionalidades en una rama (branch) y luego unirse (merge) a las del resto.

2.2. Plan de trabajo

Para la realización de todo el proyecto he seguido una metodología de trabajo que ha consistido en cinco diferentes fases:

- **Primera fase:** Es una fase de iniciación cuyo objetivo principal es el de aprender todo lo que tenga que ver con el desarrollo web. En esta fase deberíamos de dejar conceptos básicos aclarados y empezar a manejar alguna herramienta de control de versiones como Git. Es muy recomendable en esta primera fase empezar a manejar los lenguajes de programación que quieras utilizar en el futuro.
- **Segunda fase:** Una vez que tenemos cierta destreza con el desarrollo, empezamos a enfocar nuestra aplicación decidiendo que tecnologías son las que mejor nos van a venir para nuestro modelo de aplicación. Esta fase es vital para la continuación del proyecto, ya que la mala elección de una tecnología nos puede llevar mucho tiempo.
- **Tercera fase:** Una vez que tenemos claro que tecnologías vamos a utilizar en nuestra aplicación, comenzamos con una sencilla aplicación que utilice todas las tecnologías que estarán implicadas en nuestra aplicación. Esto nos servirá para tener una sencilla estructura de lo que queremos montar.
- **Cuarta fase:** Cuando tengamos claros los conceptos, manejemos los lenguajes de programación necesarios y tengamos montado una sencilla aplicación con las tecnologías que hemos seleccionado para nuestro proyecto, es momento de empezar a dar forma a nuestras ideas. Esta fase es quizás la más emocionante de todas ya que empiezas a dar cuerpo a lo aprendido hasta ahora.
- **Quinta fase:** Cuando tengamos nuestra aplicación completamente desarrollada y haciendo lo que nosotros queremos, es el momento de subirla a alguna plataforma de computación en la nube. Esta ultima fase es quizás la mas sencilla y mas gratificante del proceso ya que es el momento de que tu trabajo sea contemplado por el resto del mundo.

Capítulo 3

Infraestructura

Una vez que el proyecto y sus objetivos han sido definidos, nos centraremos en este capítulo en las tecnologías usadas para el desarrollo de la aplicación, comenzando por el marco que engloba el conjunto de subsistemas.

MEAN (acrónimo para: MongoDB, ExpressJS, AngularJS, NodeJS), es un marco o conjunto de subsistemas de software para el desarrollo de aplicaciones web, y páginas web dinámicas, que están basadas, en el popular lenguaje de programación JavaScript.

Como se observa en la imagen 3.1, el Frontend desarrollado con Angular, es el encargado de hacer llamadas al API REST (Post, Put, Get y Delete) desarrollado en NodeJS que utiliza el entorno Express. El API podrá hacer un CRUD (Create-Read-Update-Delete) a la base de datos MongoDB y cuando el API tenga los datos que se le han pedido en la llamada los devolverá a Angular en formato JSON y este los mostrará en pantalla, ya que Angular mantiene el modelo de datos actualizado sin necesidad de recargar la página.



Figura 3.1: Esquema MEAN

Las ventajas de MEAN [7] provienen de la robustez de Node, el cual proporciona su API abierta en tiempo real, que puede ser usada libremente con nuestro código *frontend* en Angular. Podemos emplearlo para transferir datos para aplicaciones como chats, actualización de estados, o cualquier otra situación que requiera mostrar datos rápidamente en tiempo real.

3.1. Angular



Figura 3.2: Icono de Angular

Angular [11] es un tecnología JS para la parte cliente de una aplicación web, que respeta el paradigma MVC y permite crear *Single-Page Applications* (Aplicaciones web que no necesitan recargar la página), de manera más o menos sencilla. Es un proyecto mantenido por Google y que actualmente está en auge.

Angular es un entorno completo para construir aplicaciones en cliente con HTML y Typescript, es decir, con el objetivo de que el peso de la lógica y el renderizado lo lleve el propio navegador, en lugar del servidor.

Para crear apps en Angular necesitamos:

1. Plantillas HTML (*templates*)
2. Componentes para gestionar esas plantillas
3. Servicios para gestionar la lógica de la aplicación
4. El componente raíz de la aplicación al sistema de arranque de Angular (*bootstrap*).

En la figura 3.3 se puede observar como se relacionan estos elementos en el diagrama de arquitectura típico, sacado de la web de Angular: Podemos identificar los 8 bloques principales de una aplicación web con Angular:

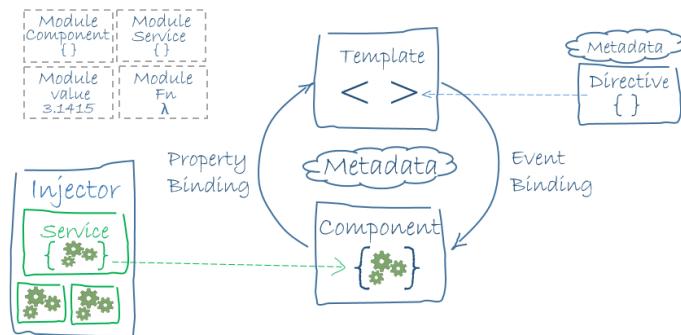


Figura 3.3: Diagrama de Angular

En este TFG hemos utilizado la versión 6 de Angular para programar el lado cliente de la aplicación web de clases particulares.

3.1.1. Módulo

Las aplicaciones de Angular en su versión más actualizada son modulares. Un módulo es típicamente un conjunto de código dedicado a cumplir un único objetivo. El módulo exporta algo representativo de ese código, típicamente una única cosa como una clase. Los módulos se pueden exportar e importar:

```
//app/app.component.js
export class AppComponent {
    //aqui va la definicion del componente
}

//app/main.js
import { AppComponent } from './app.component';
```

Hay módulos que son librerías de conjuntos de módulos. Las librerías principales de Angular son:

- **@angular/core**
- **@angular/common**
- **@angular/router**
- **@angular/http**

3.1.2. Componente

Un componente controla una zona de espacio de la pantalla que podríamos denominar vista. El componente define propiedades y métodos que están disponibles en su plantilla, pero eso no da licencia para meter ahí todo lo que te parezca. Haciendo un símil con AngularJS (Angular en su primera versión), un componente vendría a ser un controlador que siempre va ligado a una vista.

3.1.3. Plantilla

La plantilla (*template*), permite definir la vista de un Componente de Angular en código HTML, decorado con otros componentes y algunas directivas (expresiones de Angular que enriquecen el comportamiento de la plantilla).

```
<h2>Todo List</h2>
<p><i>List of Tasks</i></p>
<div *ngFor="let todo of todos" (click)="selectTodo(todo)">
    {{todo.subject}}
</div>
<todo-detail *ngIf="selectedTodo" [todo]="selectedTodo"></todo-detail>
```

Listing 3.1: Plantilla en Angular

Como vemos en la tabla 3.1, además de elementos HTML normales como `<h2>` y `<div>`, hay otros elementos desconocidos (`*ngFor`, `todo.subject`, `(click)`, `[todo]`, `todo-detail`) que explicaremos en a sección de databinding

3.1.4. Metadatos

La forma de añadir metadatos a nuestra clase en TypeScript es mediante el patrón decorador justo antes de la declaración de la clase, tal y como se muestra en la tabla 3.2

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

@Component({
  selector: 'todo-list',
  templateUrl: 'todo-list.component.html',
  styleUrls: ['todo-list.component.css'],
  moduleId: module.id,
  directives: [TodoDetailComponent],
  providers: [TodoService]
})
export class TodoListComponent { ... }
```

Listing 3.2: Metadatos en Angular

3.1.5. Data Binding

Uno de los principales valores de Angular es que abstrae la lógica *pull/push* asociada a insertar y actualizar valores en el HTML y permite convertir las respuestas de usuario (inputs, clicks, etc) en acciones concretas. Escribir toda esa lógica a mano (lo que típicamente se hacía con JQuery) es tedioso y propenso a errores, y Angular lo resuelve gracias al Data Binding. Tal y como vemos en la tabla 3.3, podemos diferenciar los distintos tipos de Data Binding que tiene Angular.

```
<h2>Todo List</h2>
<p><i>List of Tasks</i></p>
<div *ngFor="let todo of todos" (click)="selectTodo(todo)">
  {{todo.subject}}
</div>
<todo-detail *ngIf="selectedTodo" [todo]="selectedTodo"></todo-detail>
```

Listing 3.3: Data Binding en Angular

- **Interpolación** Hacia el DOM. todo.subject
- **Property binding** Hacia el DOM. [todo]="selectedTodo"
- **Event binding** Desde el DOM. (click)="selectTodo(todo)"
- **Two-way binding** (Desde/Hacia el DOM) input [(ngModel)]="todo.subject"

Angular procesa los *data binding* una vez por cada ciclo de eventos JavaScript, desde la raíz de la aplicación siguiendo el arbol de componentes en orden de profundidad.

La figura 3.4 ilustra la importancia del data-binding para la comunicación entre componentes, así como componente-plantilla.

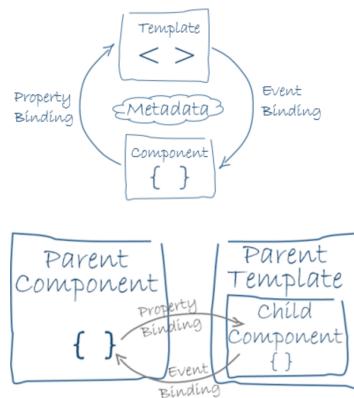


Figura 3.4: Data-Binding Diagramma

3.1.6. Directiva

Los plantillas de Angular son dinámicas: Cuando Angular los renderiza, transforma el DOM en base a las instrucciones que encuentra en las directivas

Un Componente es un caso concreto de directiva que siempre va asociado a una plantilla y al que por ser un elemento tan importante en Angular se le ha dado un decorador propio.

Tenemos dos tipos de directivas:

- **Las directivas estructurales** comienzan por asterisco y sirven para alterar el DOM.

```
<div *ngFor="let todo of todos"></div>
<todo-detail *ngIf="selectedTodo"></todo-detail>
```

- **Las directivas Atributo** alteran la apariencia o comportamiento de un elemento del DOM

```
<input [(ngModel)]="todo.subject" >
```

3.1.7. Servicio

Los servicios son imprescindibles en Angular, se definen a través de simples clases. Todo valor, función o característica que nuestra aplicación necesita, se encapsula dentro de un servicio.

Los Componentes son grandes consumidores de servicios. No recuperan datos del servidor, ni validan inputs de usuario, ni logean nada directamente en consola. Delegan todo este tipo de tareas a los Servicios.

3.1.8. Inyección de Dependencias

Una dependencia en tu código se produce cuando un objeto depende de otro. Hay diferentes grados de dependencia, pero tenerlas en exceso hace que probar tu código sea complicado o que algunos procesos se ejecuten más tiempo de la cuenta. La inyección de dependencias es un método por el cual damos a un objeto las dependencias que requiere para su funcionamiento.

Angular permite extender el vocabulario del HTML con directivas y atributos para crear componentes dinámicos. En las páginas web dinámicas sin Angular hay ciertas complicaciones frecuentes, como el data binding, validación de formularios, manejador de eventos con DOM (Document Object Model) y otras muchas. Angular presenta una solución “todo-en-uno” a esos problemas. La curva de aprendizaje para Angular es muy pequeña, lo que explica que mucha gente se esté pasando a este marco. La sintaxis es simple y sus principios básicos como el data binding (vinculación de elementos de nuestro documento HTML con nuestro modelo de datos) y la inyección de dependencias son sencillas de entender.

3.2. Node

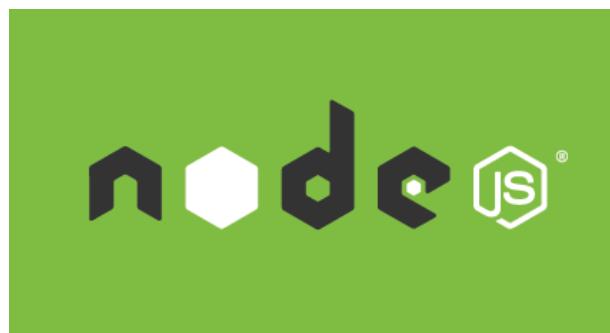


Figura 3.5: Icono de Node

Node[8] es un entorno de programación en JavaScript para el Backend basado en el motor V8 del navegador Google Chrome y orientado a eventos, no bloqueante, lo que lo hace muy rápido a la hora de crear servidores web y emplear tiempo real. Fue creado en 2009 y aunque aún es joven, las últimas versiones lo hacen más robusto. Además posee una gran comunidad de desarrolladores.

Uno de los beneficios de Node es su gestor de paquetes, npm (node package manager), el cual nos permite gestionar todas las dependencias y módulos de una aplicación. Al igual que Ruby tiene RubyGems y PHP tiene Composer, Node tiene npm. Viene ya incluido con Node y permite bajar una serie de paquetes para satisfacer las necesidades del servicio programado. Este sistema de paquetes es lo que hace a Node tan potente. La capacidad de tener una serie de códigos que puedes reutilizar en todos tus proyectos hace que el desarrollo sea más sencillo, ya que puedes combinar fácilmente varios paquetes para crear aplicaciones complejas.

```
> npm install && npm start
```

Con esta instrucción en línea de comandos, nos descargamos todas las dependencias, además de arrancar nuestra aplicación. En este TFG se ha utilizado Node para programar el servidor web de la aplicación de gestión de clases particulares.

3.3. MongoDB

MongoDB[9] es la base de datos que ha elegido para la aplicación de gestión de clases particulares, debido a sus grandes ventajas cuando se manejan ingentes cantidades de in-



Figura 3.6: Icono de MongoDB

formación. MongoDB nace en octubre de 2009 y a día de hoy innumerables empresas ya disponen de esta base de datos en sus aplicaciones como por ejemplo:

- **Bosh:** Utiliza MongoDB ya que está poniendo a prueba una aplicación que es capaz de capturar datos de vehículos, como el sistema de frenado, la dirección asistida, los limpiaparabrisas ... Con todos estos datos se pueden hacer diagnósticos de necesidad de mantenimiento preventivo.
- **Forbes:** Construyó todo un sistema de gestión de contenidos en MongoDB. Además utiliza MongoDB para analítica en tiempo real. Cuando algún artículo se hace viral, Forbes detecta la forma en que se está compartiendo entre los usuarios y de este modo sabe qué tipo de contenido le debe ofrecer a sus lectores.

3.3.1. Características

MongoDB es una base de datos no relacional (NoSQL) de código abierto que guarda los datos en documentos tipo JSON (JavaScript Object Notation) pero en forma binaria (BSON) para hacer la integración de una manera más rápida. Se pueden ejecutar operaciones en JavaScript en su consola en lugar de consultas SQL. Además tiene una gran integración con Node.js con los drivers propios y con Mongoose. Debido a su flexibilidad es muy escalable y ayuda al desarrollo ágil de proyectos web.

MongoDB está orientado a servicios que necesiten una persistencia basada en documentos, al contrario que otros sistemas de base de datos noSQL como Cassandra, el cual está orientado a logs, o como Redis que necesita una persistencia basada en colas de mensajes.

Actualmente estamos en la era de lo que Martin Fowler llama “Polyglot persistence”. Hay que decidir el tipo de persistencia a utilizar para después usar el tipo de persistencia que más se amolde a nuestras necesidades.

Las características que hacen tan importante a esta base de datos son las siguientes:

- Está orientada a documentos. En un único documento es capaz de almacenar toda la información necesaria que define un producto, un cliente, etc, aceptando todo tipo de datos sin tener que seguir un esquema predefinido.
- Da respuesta a la necesidad de almacenamiento de todo tipo de datos: estructurados, semi estructurados y no estructurados.
- Tiene un gran rendimiento en cuanto a escalabilidad y procesado de la información. Puede procesar la gran cantidad de información que se genera hoy en día.

- Permite a las empresas ser más ágiles y crecer más rápidamente, creando así nuevos tipos de aplicaciones.

3.3.2. Documento en MongoDB

MongoDB está escrito en C++, su versión de 32 bits sólo puede alcanzar 2GB, por este motivo la versión de 32 bits no es recomendable usarla en producción.

```
{
  name: "mario",
  age: 25,
  preferences: [
    "programming",
    "nosql",
    "javascript"
  ]
}
```

Esto es un documento en Mongo, los cuales se almacenan en colecciones y éstas a su vez en bases de datos. Estas colecciones poseen un esquema flexible y totalmente dinámico lo que hace que la velocidad de cómputo sea muy alta. Las bases de datos no se crean manualmente, primero se define la base de datos a usar y luego se inserta un documento en alguna colección.

```
> show dbs
> use pruebanosql
> show collections
> db.users.insert({ "name": "mario", "age": 24 })
```

3.3.3. Inconvenientes de MongoDB

Un problema que tiene Mongo es que no soporta transacciones de múltiples documentos. Sin embargo puede proporcionar operaciones atómicas en un solo documento. A menudo estas operaciones atómicas de nivel de documento son suficientes para resolver los problemas que requerían transacciones en una base de datos relacional. Por ejemplo, en Mongo se pueden incrustar datos relacionados en matrices anidadas o documentos anidados dentro de un solo documento y actualizar todo el documento en una sola operación atómica. Por este motivo los servicios que requieren de transacciones, como los bancos o entidades económicas, no utilizan Mongo debido a que no es capaz de hacer una sola operación atómica en dos documentos.

Otro posible problema es la excesiva cantidad de memoria RAM que puede consumir MongoDB, aunque es posible ejecutar MongoDB en una maquina con una pequeña cantidad de memoria RAM libre. MongoDB usa automáticamente toda la memoria libre del equipo como su cache, es por esto por lo que los monitores de recursos muestran que MongoDB utiliza una gran cantidad de memoria, pero su uso es dinámico. Es decir que si otro proceso de repente necesita mayor espacio de memoria RAM, MongoDB liberará parte de su memoria asignada para el otro proceso.

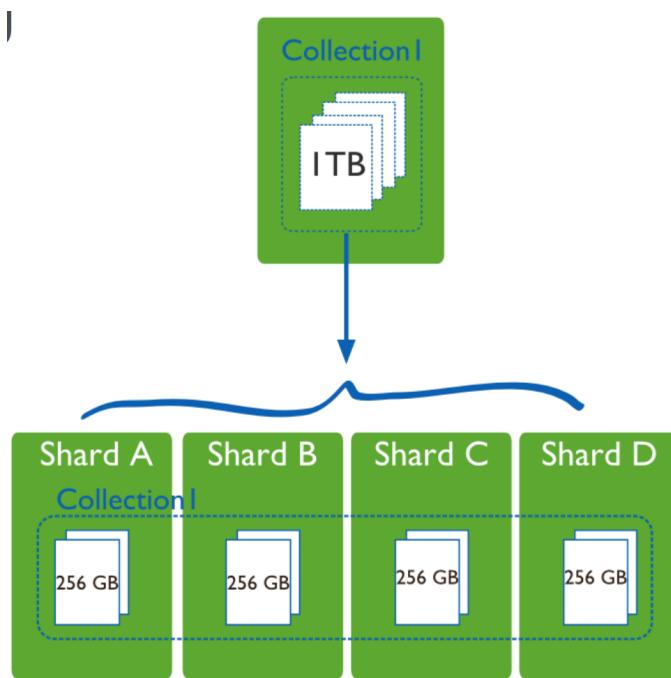


Figura 3.7: Sharding

3.3.4. Fragmentación (Sharding)

¿Qué es la fragmentación? Cuando un proyecto empieza a tener un número de peticiones de acceso elevado, se empieza a notar que su base de datos va más lento de lo normal. Para este problema se tiene dos soluciones, una actualizar toda la infraestructura para soportar la demanda o empezar a utilizar la fragmentación.

La fragmentación, es el modo en el se hace la base de datos escalable. En lugar de tener una colección en una base de datos, se tendría en varias bases de datos distribuidas, de modo que a la hora de consultar los datos de dicha colección, se recupera como si de una única base de datos se tratase. Esto de encontrar la base de datos lo hace MongoDB de forma transparente. Cuando se hacen consultas, se tiene un enrutador llamado "MongoS", el cual mantendrá un pequeño conjunto de conexiones a los distintos fragmentos.

Los fragmentos estarán formados por *réplica set*, de modo que si creamos tres fragmentos, cada uno de los cuales tiene una *réplica set* con tres servidores, estaríamos hablando de un total de nueve servidores. Para saber en qué fragmento debe consultar para recuperar datos de una colección ordenada, se utilizan rangos y *shard key*, de modo que se trocea la colección en rangos y se les asigna un identificador a cada rango. De este modo cuando se consulte la colección debemos proporcionar el *shardKey*.

3.4. Express

3.4.1. Introducción

Express [10] es un entorno de aplicaciones web para Node.js, que permite crear servidores web y recibir peticiones HTTP de una manera sencilla, lo que permite también crear APIs REST de forma rápida.



Figura 3.8: Icono de Express

En la web de ExpressJS, se describe como *un entorno de desarrollo de aplicaciones web minimalista y flexible para Node.js*. Sin duda el éxito de Express radica en lo sencillo que es usarlo, y además abarca un sin número de aspectos que muchos desconocen pero son necesarios.

La referencia de la API se divide en 5 grandes módulos:

- **express()**: La función *express()* es una función de nivel superior exportada por el módulo express.

```
express.json()
express.static()
express.Router()
express.urlencoded()
```

- **Application**: El objeto *app* se crea llamando a la función *express ()* de nivel superior exportada por el módulo Express

```
Properties -->|app.locals||app.mountpath|
Events -->| mount |
Methods -->|app.all()|app.delete()|app.disable()|app.listen()|
```

- **Request**: El objeto *req* representa la solicitud HTTP y tiene propiedades para la cadena de consulta de solicitud, parámetros, cuerpo, encabezados HTTP, etc.

```
Properties -->|req.body|req.cookies|
Methods -->|req.accepts()|req.acceptsCharsets()|
```

- **Response**: El objeto *res* representa la respuesta HTTP que envía una aplicación Express cuando recibe una solicitud HTTP.

```
Properties -->|res.app|res.headersSent|res.locals|
Methods -->|res.cookie()|res.clearCookie()|
```

- **Router**: El objeto *Router* es una instancia aislada de middleware y rutas. Puede considerarlo como una "miniaPLICACIÓN", que solo puede realizar funciones de enrutamiento y middleware. Cada aplicación Express tiene un router de aplicaciones integrado.

```
Methods -->|router.all()|router.METHOD()|router.param()|
```

3.4.2. Estructura de una API

En una aplicación escrita con express existe una estructura interna bien definida y es como sigue:

- **Módulos o archivos externos** Importamos todos los módulos o archivos externos que nuestra app vaya a necesitar. El bloque, o mejor dicho la línea, que viene a continuación es la más importante de todas, ya que se encarga de instanciar Express y asignarlo a la variable app, la cual se utilizará a partir de ahora para configurar los parámetros de Express.

```
var app = express();
```

El siguiente bloque sirve para configurar e iniciar algunos componentes de Express. Destacar la línea, `app.use(express.static(...))`, en la cual se configuran los objetos estáticos (imágenes, hojas de estilo, etc.) que debe servir Express, los cuales se encuentran en la carpeta *public*. Esta configuración permite también que los elementos estáticos puedan ser accedidos como si se encontraran en el directorio raíz del proyecto, de forma que para acceder a las imágenes ubicadas en /public/images se haría con la URL `http://localhost:3000/images`.

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

- **Conectamos a la base de datos** La segunda parte de nuestra app *express* es conectarnos a la base datos.
- **Importamos controladores y modelos de la base de datos** Una vez conectados a la base de datos importamos los controladores y los modelos de nuestra base de datos.
- **Rutas** Las rutas son definitivamente la parte más importante de tu aplicación, ya que son las encargadas de invocar las funciones que se encuentran en el controlador.

Una ruta está especificada de la siguiente forma:

```
app.VERBO(PATH, ACCION)
```

VERBO: Puede ser: GET, POST, PUT, DELETE y así para cada uno de los verbos HTTP.

PATH: Define la dirección de acceso.

ACCION: Que es lo que se tiene que hacer.

- **Listen** Por último, es importante que la aplicación esté disponible en algún puerto.

```
app.listen(8000);
```

Express esconde muchas funcionalidades internas de Node, lo que permite sumergirse en el código de la aplicación y conseguir los objetivos de forma muy rápida. Es fácil de aprender y deja cierta flexibilidad con su estructura. Por algo es el entorno más popular de Node.

3.5. Amazon web services



Figura 3.9: AWS

Amazon Web Services es el proveedor más famoso y más completo en infraestructura como servicio (IaaS), ya que ofrece un conjunto de servicios y un modelo de precios muy completo y que se ajusta a las necesidades de cada cliente.

Dispone de varios tipos de instancias según su hardware:

1. **Instancias estándar:** Pequeñas, medianas, grandes y extra-grandes
2. **Instancias con gran cantidad de memoria**
3. **Instancias con CPU de alto rendimiento**
4. **Instancias en cluster:** Con redes de alta velocidad entre ellas
5. **Instancias de GPU para clústeres**

Además Amazon Web Services dispone de diferentes servicios entre los que destacamos:

1. **Amazon Elastic Block Store:** Disco duro de las instancias persistente, donde sus datos permanecen cuando la instancia se apaga.
2. **Varias ubicaciones:** El cliente puede elegir la ubicación para reducir la latencia a los usuarios de los servicios. Además, existen varias zonas de disponibilidad dentro de la misma ubicación para minimizar el impacto de las catástrofes.
3. **Direcciones Elastic IP:** Por defecto las instancias tienen IPs internas en la red de Amazon, asociando una IP pública a una instancia.
4. **Amazon Cloud Watch:** Servicio de monitorización de instancias con sistema de alarmas y gráficas de uso de recursos (memoria, CPU, red...)
5. **Auto Scaling:** Se pueden configurar reglas para que Amazon inicie más instancias cuando la carga de las existentes supere un umbral y volver a bajar cuando la carga disminuya.
6. **Elastic Local Balancer:** Dispositivos que reparten las peticiones web a cada una de las instancias que se han creado con el escalado automático o manual.
7. **Imágenes de Instancias (AMI):** Amazon permite la gestión de las instancias (AMI), pudiendo crear y gestionar varias imágenes. Se puede iniciar una instancia con cualquier imagen.

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

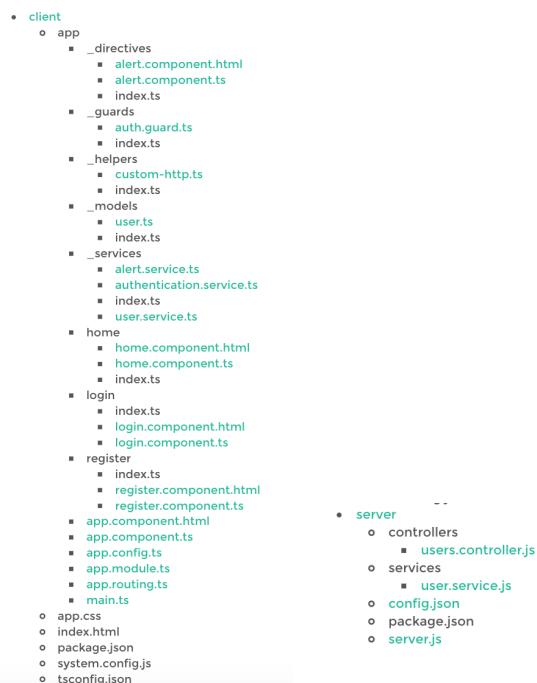


Figura 4.1: Estructura de un proyecto con la pila MEAN

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.

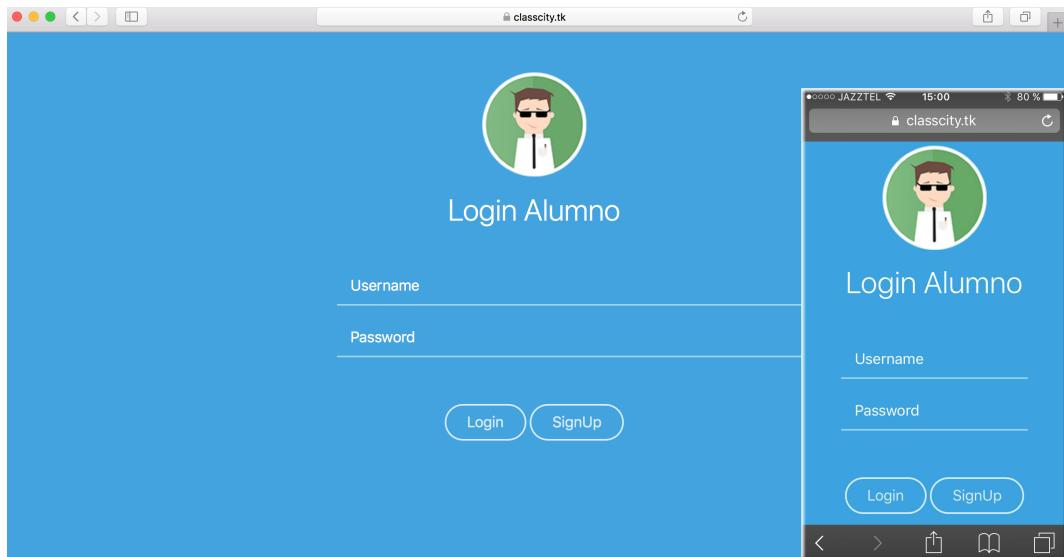


Figura 4.3: Página Login Alumnos

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.

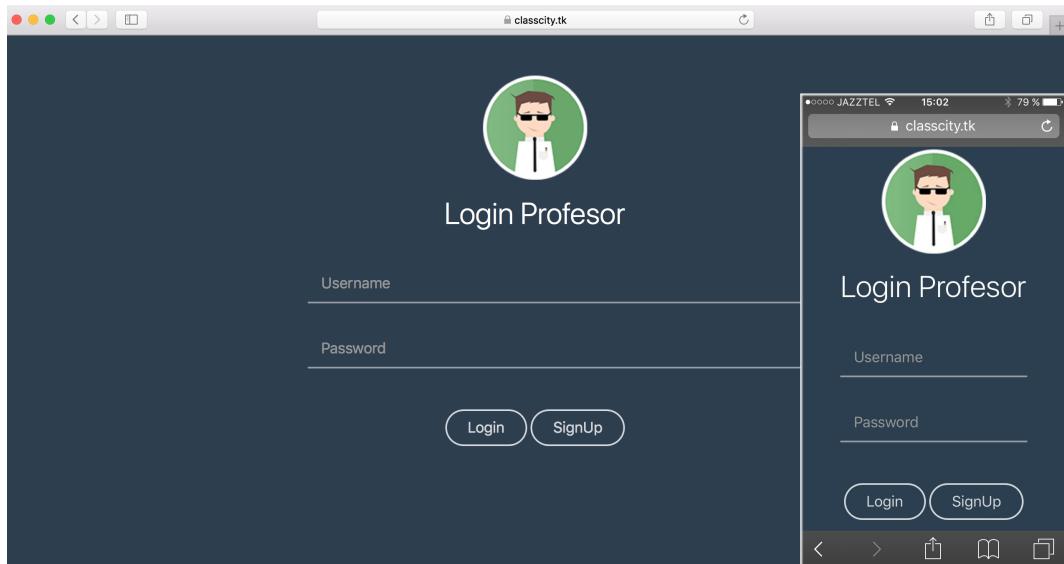


Figura 4.4: Página Login Profesor

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.

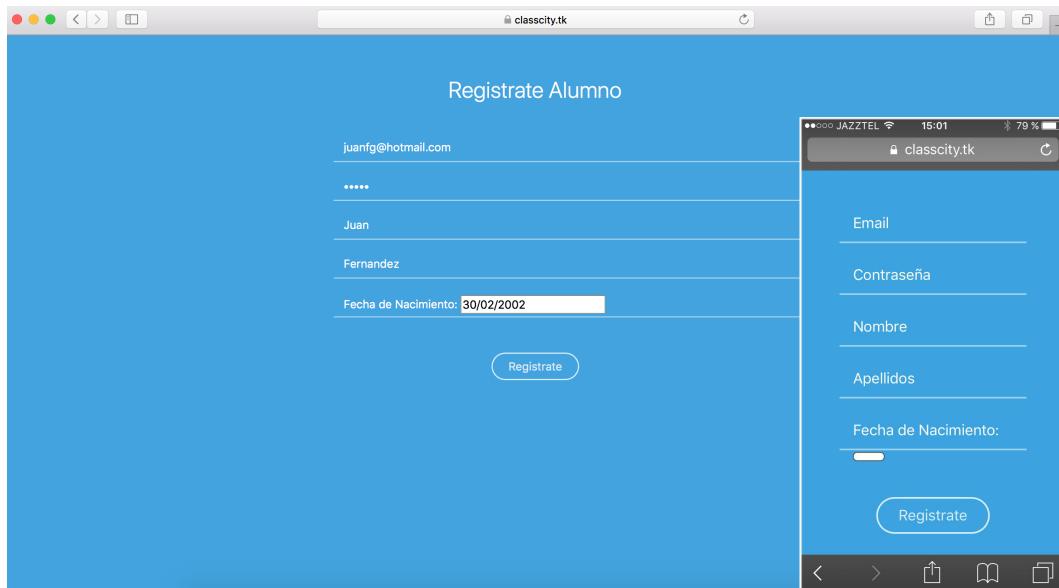


Figura 4.5: Página Registrar Alumno

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.

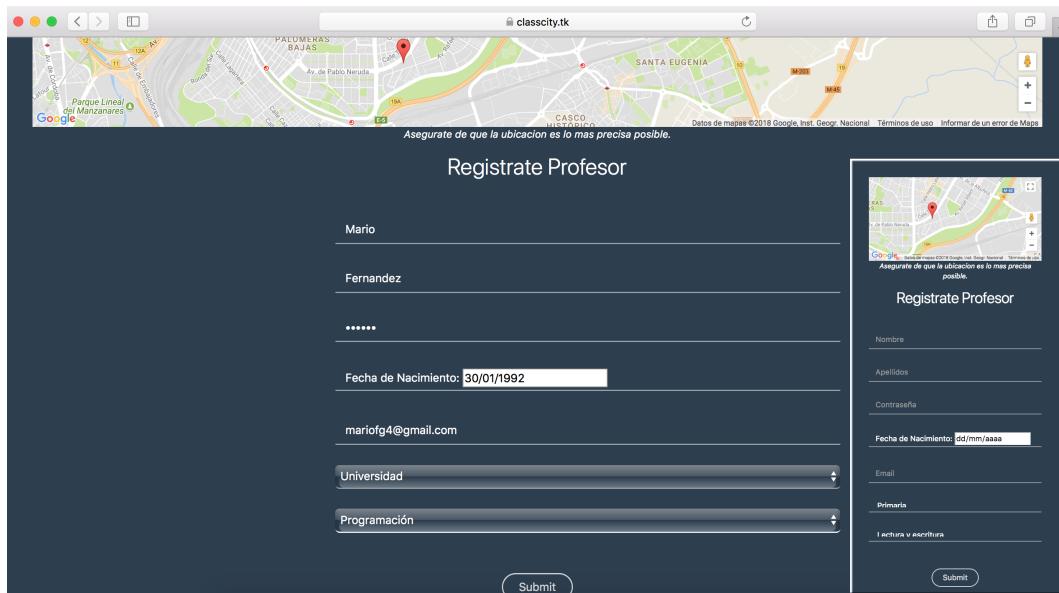


Figura 4.6: Página Registrar Profesor

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.

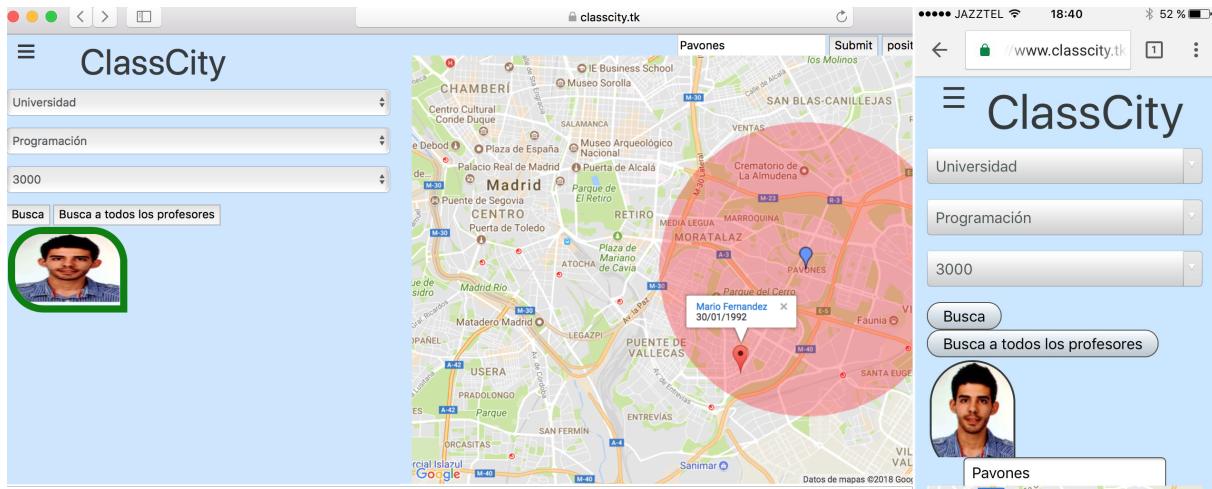


Figura 4.7: Página Home Alumno

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.

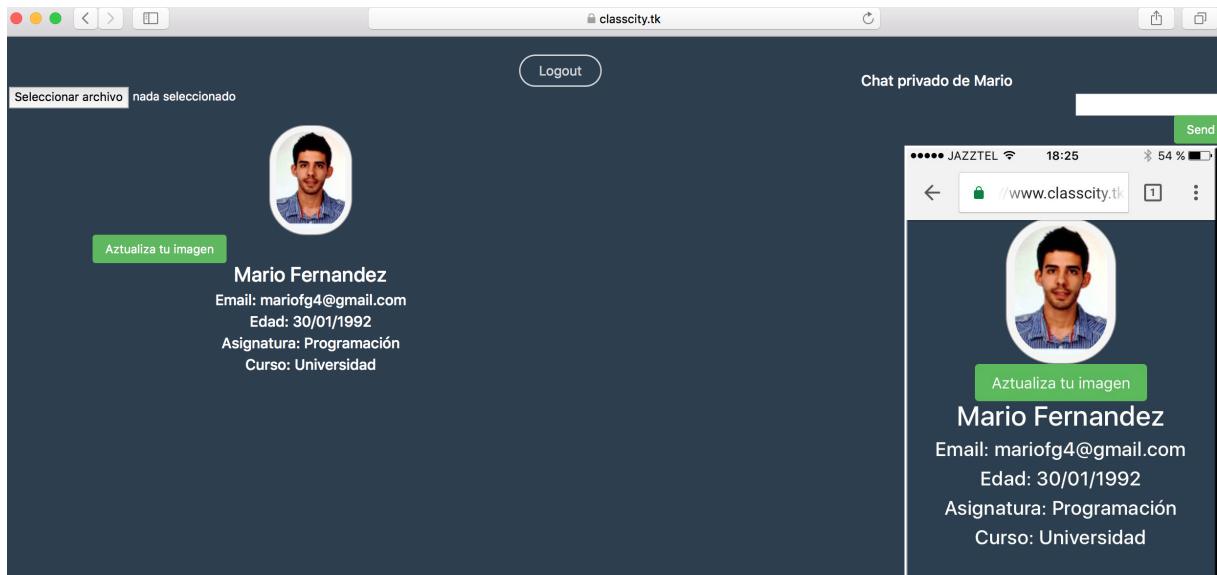


Figura 4.9: Página Home Profesor

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 *savenotificación*, 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

Capítulo 5

Despliegue en la nube

Hemos elegido Amazon Web Service como servicio de computación en la nube por el gran impacto que está teniendo en el mundo laboral, mayor que sus competidores directos como son Azure y Google Cloud.

Otro factor que juega a favor de AWS con respecto a sus competidores es que a la hora de desplegar una aplicación en la nube la experiencia de usuario resulta mucho mas intuitiva que la de sus competidores. También ofrece un buen soporte, donde en cualquier momento te resuelven las posibles dudas a la hora de desplegar la aplicación.

Por último y como factor de bastante importancia para aquellos pequeños emprendedores que quiera empezar a desarrollar sus ideas, es el bajo coste que supone tener una aplicación desplegada en la nube de Amazon. Por el momento, el primer año de tu cuenta en AWS es gratuita, restringida a ciertas limitaciones.

Hoy en día, Amazon Web Services proporciona una plataforma de infraestructura escalable, de confianza y de bajo costo en la nube que impulsa cientos de miles de negocios de 190 países de todo el mundo. Con centros de datos en Estados Unidos, Europa, Brasil, Singapur, Japón y Australia. Por ejemplo:

1. **Amazon.com:** Es el minorista online más grande del mundo. En 2011, Amazon.com pasó de utilizar el backup en cinta a usar Amazon S3 en la cloud para realizar copias de seguridad de la mayoría de las bases de datos de Oracle de las que se encarga. Mediante el uso de AWS, Amazon.com logró eliminar el software de backup y experimentó una mejora de desempeño 12 veces mayor, de forma que pudo reducir el tiempo de restablecimiento de 15 a 2,5 horas aproximadamente en situaciones seleccionadas.
2. **Netflix:** El referente de la televisión en streaming usa AWS para proporcionar miles de millones de horas de vídeo casa mes a sus mas de 60 millones de suscriptores. Así puede hacer uso de miles de servidores y terabites de almacenamiento en cuestión de minutos para que sus usuarios puedan ver series y películas desde cualquier parte del mundo en sus tabletas o teléfonos móviles.
3. **Dropbox:** El famoso y conocido servicio de alojamiento de archivos multiplataforma en la nube utiliza hasta el momento AWS como repositorio para almacenar todos los archivos que los usuarios de Dropbox suben a la red.
4. **FC Barcelona:** Su sitio web, que aloja más de 6 000 páginas y 12.000 fotos digitales, también usa AWS para su mantenimiento.

5. **Harvard Medical School** Desarrolla nuevos modelos de pruebas de genomas en tiempo récord.
6. **Mapfre:** Ahorró 1,3 millones de euros en infraestructura y redujo el desarrollo de semanas a días

5.1. Lanzar una instancia en AWS

Antes de subir nuestra aplicación a producción debemos de crear una instancia en AWS. Estos son los pasos a tener en cuenta a la hora de crear una instancia en AWS:

- **Paso1. Crear Key Pairs** Los *Key Pairs* se utilizan para iniciar sesión de forma segura en los servicios de AWS. Crearemos un Key Pairs para acceder a nuestra instancia de EC2.

1. Para crear nuevos pares de claves, se debe navegar hasta AWS Console y luego hacer clic en EC2.

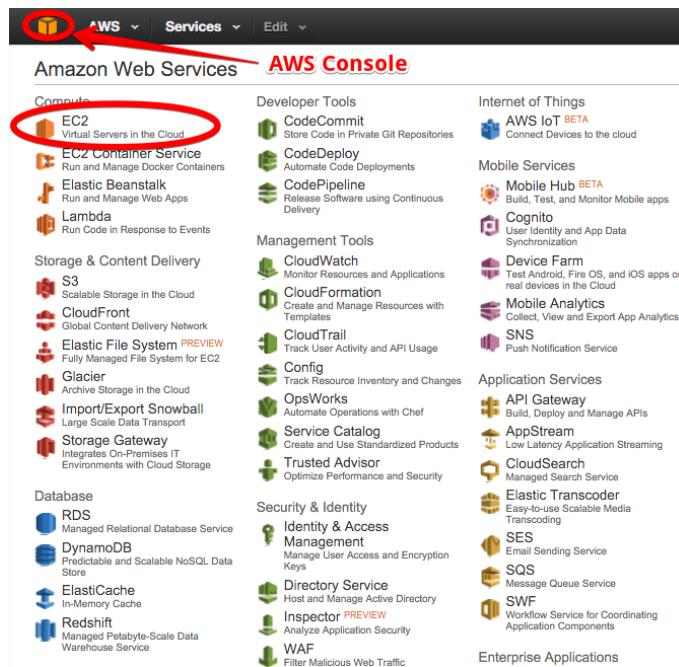


Figura 5.1: Crear Key Pairs Paso 1

2. En el panel izquierdo, se debe hacer clic en Key Pairs, luego clic en Crear Key Pairs
3. Se debe introducir un nombre para la clave, luego hacer clic en Crear Key Pairs. El Key Pairs se descargará automáticamente. Debe moverse esta clave a un directorio diferente.

Importante: Se deberá cambiar los permisos de esta clave para que sean de solo lectura, con el siguiente código:

```
chmod 400 youKeyName.pem
```

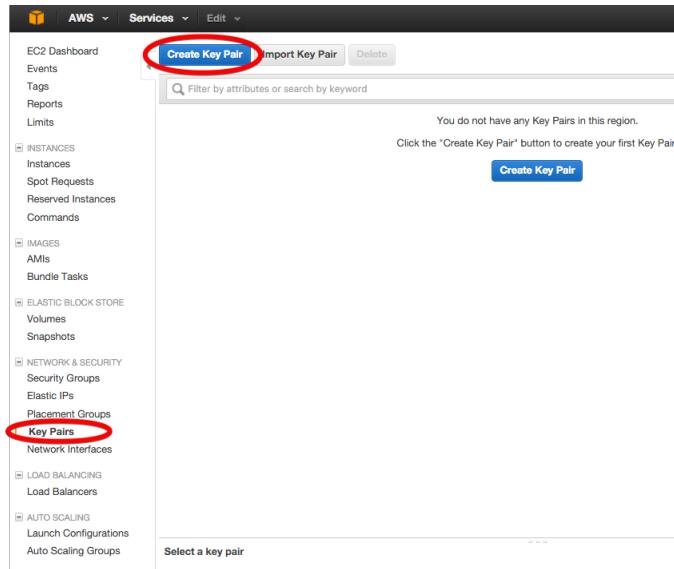


Figura 5.2: Crear Key Pairs Paso 2

- **Paso 2. Lanza una instancia de EC2 con Bitnami** En este paso, lanzaremos una instancia de EC2 desde Amazon Machine Image (AMI). Con AMI se puede activar una instancia de EC2 que esté lista para el desarrollo sin demasiada configuración. Bitnami proporciona una imagen MEAN preconfigurada que usaremos para configurarlo rápidamente.

1. Primero se debe navegar a la consola de AWS y hacer clic en AWS Marketplace.

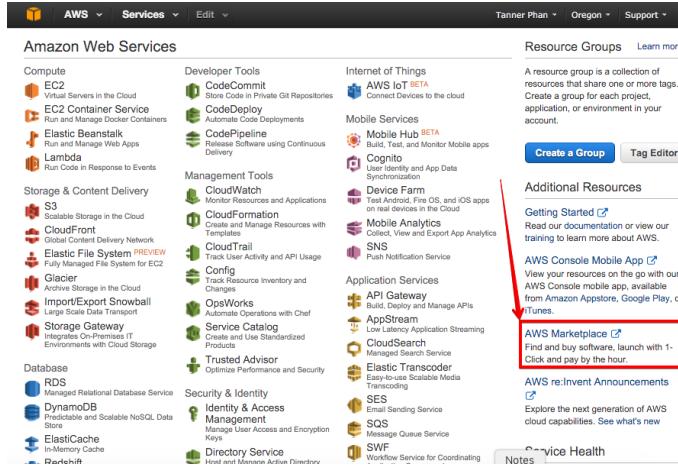


Figura 5.3: Lanza una instancia Paso 1

2. Buscar MEAN powered en Bitnami, luego seleccione 64-bit AMI para continuar. Ver figura 5.4
3. En el apartado *Pricing Details* con el fin de obtener la mejor velocidad de entrega, seleccionar la región más cercana y luego hacer clic en Continuar. Ver figura 5.5
4. Para el grupo de seguridad, se ha de elegir 'Crear nuevo' según la configuración del vendedor.

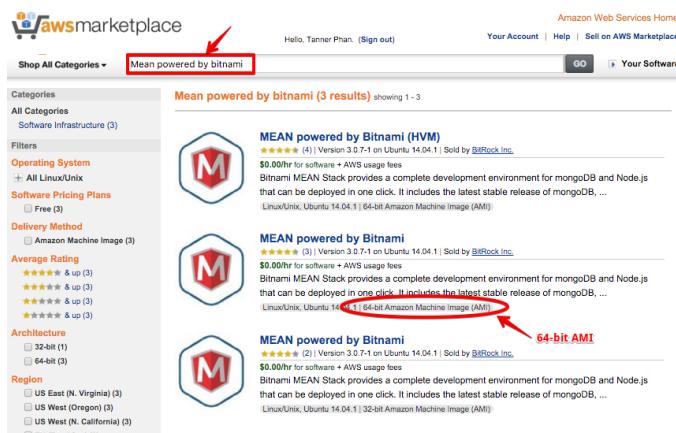


Figura 5.4: Lanza una instancia Paso 2

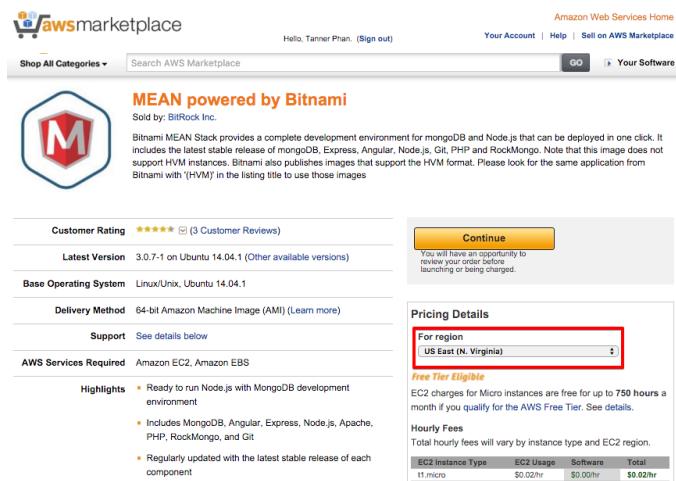


Figura 5.5: Lanza una instancia Paso 3

5. Asegurarse de tener los siguientes métodos de conexión:

1. SSH, My IP
2. HTTP, Anywhere
3. HTTPS, Anywhere

6. Seleccionar la Key Pair que creaste en el paso 1.
7. Finalmente, hacer clic en Iniciar para iniciar la instancia.

■ **Paso 3. Conéctate a tu EC2** Para conectarse por SSH a la instancia, se necesitará la IP pública de su instancia y el Key Pair que creó.

1. Dentro de EC2, hacer clic en Instancias, seleccionar la instancia recién iniciada en el paso anterior y luego hacer clic en Conectar.
2. Copiar el código que aparece marcado en rojo en la imagen 5.6.
3. En su terminal, navegar hasta el directorio donde está guardado el par de claves, luego pegar el último paso del código.

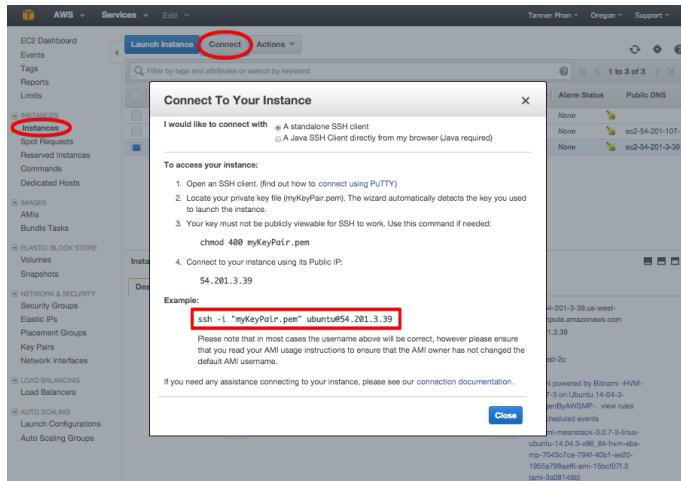


Figura 5.6: Conéctate a tu EC2 Paso 2

```
The authenticity of host '54.201.14.233 (54.201.14.233)' can't be established.
ECDSA key fingerprint is SHA256:ITSaXliGDhtY8uTh10BYnGM8EtH3B07LzR8F/f1T/xE.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.201.14.233' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-71-generic x86_64)

[...]
*** Welcome to the Bitnami MEAN 3.0.7-3 ***
*** Bitnami Wiki: https://wiki.bitnami.com/ ***
*** Bitnami Forums: https://community.bitnami.com/ ***
bitnami@ip-172-31-1-86:~$
```

Figura 5.7: Conéctate a tu EC2 Paso 3

- **Paso 4. Conseguir un dominio** Una vez que ya tenemos la instancia creada y hemos sido capaces de conectarnos a ella por SSH, llega el momento de conseguir un dominio para que los usuarios se puedan conectar a la aplicación lo más fácilmente posible. El dominio que he utilizado es un dominio, proporcionado por el propio Amazon.

- **Paso 4. Obtener un certificado SSL gratis con AWS Certificate Manager**

Un certificado SSL es un fichero informático generado por una entidad de servicios de certificación que asocia unos datos de identidad a una persona física, organismo o empresa, confirmando de esta manera su identidad digital en Internet. Necesitamos un certificado SSL para que los usuarios que accedan a nuestra página lo hagan de una forma segura por el protocolo HTTPS.

Por otro lado tenemos Amazon CloudFront, es un servicio de red de entrega de contenido (CDN) global que proporciona datos, vídeos, aplicaciones y API de forma segura a sus espectadores con baja latencia y altas velocidades de transferencia.

Para conseguir el certificado SSL en AWS se necesita seguir los siguientes pasos:

1. Ir a AWS *manager certificate* desde la consola de Amazon web services y hacer click en *get started*.
2. A continuación te pedirá el dominio creado previamente y tendrás que hacer click en *Review and request*.

3. Una vez enviada la solicitud de certificado a la Autoridad certificadora, un email de confirmación será enviado a nuestra cuenta de correo vinculada con el dominio, es decir a admin@classcity.es.
4. Una vez que recibimos un correo como el que aparece en la siguiente imagen, realizamos la confirmación del certificado haciendo click en el enlace que viene en el correo.

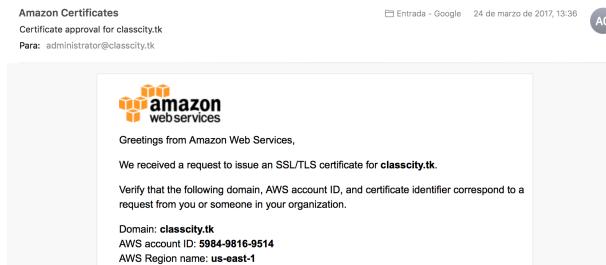


Figura 5.8: Correo de confirmación

Si ahora vamos a nuestro dominio e introducimos antes el protocolo HTTPS, es decir en nuestro caso <https://www.classcity.es>, veremos que efectivamente accedemos a nuestra aplicación de una forma segura. Ver figura 5.9

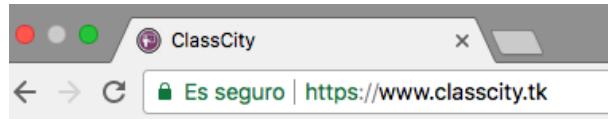


Figura 5.9: SSL

5.2. Lanzar una aplicación MEAN a producción

Una vez que dispongamos de una instancia, un dominio y un certificado SSL, tenemos todo listo para desplegar nuestra aplicación MEAN en producción. Para ello debemos seguir los siguientes pasos:

1. Conectarnos desde nuestra terminal por SSH a nuestra máquina virtual de AWS


```
sudo ssh -i /.ssh/keypair.pem root@ipinstance
```
2. Clonar en la máquina virtual de AWS nuestro repositorio git donde almacenemos la aplicación MEAN.


```
rooti@ip sudo git clone https://github.com/RoboticsURJC-students/2016-tfg-Mario-Fernandez.git
```
3. Instalar dependencias de la aplicación web por parte del lado cliente.

```
rooti@ip cd 2016-tfg-Mario-Fernandez
rooti@ip /2016-tfg-Mario-Fernandez sudo npm install
```

4. Correr Angular con angular-ci.

```
rooti@ip /2016-tfg-Mario-Fernandez sudo npm run build
```

5. Una nueva carpeta se crea al terminar el proceso ./dist, debemos copiarla entera en nuestra carpeta httdocs de apache.

```
rooti@ip /2016-tfg-Mario-Fernandez sudo cp -r ./dist/* ../httdocs
```

6. Ahora deberíamos poder ir a <https://www.classcity.es> y poder ver nuestra aplicación Angular corriendo en producción.



Figura 5.10: <https://www.classcity.es>

7. No olvidemos que en la pila MEAN, aún falta que configuremos el servidor en producción. Para ello lo primero es instalar las dependencias necesarias para lado del servidor.

```
rooti@ip /2016-tfg-Mario-Fernandez cd backend
rooti@ip /2016-tfg-Mario-Fernandez/backend sudo npm install
```

8. Antes de ejecutar nada en la parte del servidor, debemos configurar ciertas rutas en el servidor apache que va ser el encargado de recibir la peticiones de entrada. Para ello debemos hacer lo siguiente:

- Editamos el archivo de configuración de apache llamado httpd.conf para que cuando llegue las solicitudes de tipo /app/* redirigir a localhost: 8080, que será donde estará escuchando nuestra aplicación de gestión de clases particulares.

```
ProxyPassMatch ^/app/(.*)$ http://127.0.0.1:8080/$1
ProxyPass /app/(.*)$ http://127.0.0.1:8080/
ProxyPassReverse /app/(.*)$ http://127.0.0.1:8080/
```

- Y cuando llega la solicitud de type /socket/* redirigir a localhost: 8000, que será el puerto donde este escuchando nuestro chat.

```
RewriteEngine On
RewriteCond %{REQUEST_URI} ^/socket.io [NC]
RewriteCond %{QUERY_STRING} transport=websocket [NC]
RewriteRule /(.*) ws://localhost:8000/$1 [P,L]
ProxyPass /socket.io http://localhost:8000/socket.io
ProxyPassReverse /socket.io http://localhost:8000/socket.io
```

9. A continuación en el lado servidor correremos la base de datos con la propiedad *screen*, para que cuando se termine la conexión por ssh el proceso siga corriendo y no se corte.

```
rooti@ip /2016-tfg-Mario-Fernandez/backend sudo mkdir data
rooti@ip /2016-tfg-Mario-Fernandez/backend sudo screen mongod --dbpath ./data
```

10. Y por último ejecutamos e fichero *server.js* también con la propiedad *screen*, y comprobaremos que funciona correctamente.

```
rooti@ip /2016-tfg-Mario-Fernandez/backend sudo node server.js
screen
```

Capítulo 6

Conclusiones

En los capítulos anteriores se ha hecho una descripción de las tecnologías empleadas y se ha presentado la aplicación desarrollada. Además, se ha argumentado tanto la elección final del diseño como su arquitectura. En este capítulo se analizarán las conclusiones del trabajo realizado y se proponen posibles mejoras futuras de desarrollo.

6.1. Conclusiones

Tras analizar el trabajo realizado se puede apreciar que se ha conseguido el objetivo general. Se ha creado una aplicación web capaz de poner en contacto alumnos y profesores para dar clases particulares. Además de cubrir todos los sub-objetivos marcados en el capítulo 2 y que vamos a ir analizando una a uno a continuación:

- **Front-End:** El Front-End de la aplicación se ha desarrollado satisfactoriamente utilizando Angular, una tecnología cuya curva de aprendizaje ha sido superada a pesar de la grandes dificultades que contiene. Angular nos ha permitido tener una estructura de la aplicación lo bastante sólida y escalable para poder introducir cualquier tipo de mejora en un futuro.
- **Back-End:** En cuanto al Back-End, era nuestro segundo sub-objetivo, añadir que también ha sido superado gracias a la facilidad que tiene Node y su framework Express en su aprendizaje, además de la numerosa comunidad de desarrolladores que hay detrás. Todo esto nos ha supuesto que crear una API desde cero, sea algo más sencillo de lo que esperábamos.
- **BBDD:** En lo relacionado con la BBDD, decir que la integración del Backend con MongoDB ha sido todo un éxito gracias a mongoose, una librería que nos ha permitido enlazar el servidor con la base de datos sin demasiadas complicaciones. Aunque si que es verdad que comprender el modelo NoSQL y sus beneficios, no ha sido algo trivial.
- **Despliegue en la red:** El último de los sub-objetivos consistía en subir la aplicación web a producción. Podemos decir que este sub-objetivo se ha desarrollado satisfactoriamente, debido a que si accedemos a www.classcity.es podemos ver nuestra aplicación corriendo en producción.

6.2. Trabajos Futuros

Este TFG ha sido desarrollado para poder implementar funciones en un futuro sin necesidad de cambiar la arquitectura de nuestra aplicación. Es por eso que hemos identificado cuatro funcionalidades nuevas que harían a nuestra aplicación mucho más interesante.

1. **Añadir WEB RTC:** WebRTC es una API que está siendo elaborada por la World Wide Web Consortium para permitir a las aplicaciones del navegador realizar llamadas de voz, chat de vídeo y uso compartido de archivos P2P sin plugins. Una de las mejoras que podría tener esta aplicación sería el uso de web RTC para que los profesores pudiesen impartir sus clases a partir de videoconferencia.
2. **Añadir Comentarios y valoraciones a los profesores:** Otra de las mejoras que podrían ser empleadas, es la valoración de los profesores por parte de los alumnos, utilizando el potencial de mongoDB para almacenar información.
3. **Añadir Calendario:** Otra funcionalidad extra que se puede introducir en la aplicación, es la utilización de calendarios para que los alumnos puedan consultar la disponibilidad de cada profesor
4. **Añadir un método de pago:** Por último se podría añadir la mejora de monetizar el uso de la aplicación. Añadiendo un método de pago a partir de la aplicación cada vez que un alumno quiera dar una clase particular.

Bibliografía

- [1] Walter Cuenca. Practicas Web para la asignatura de LTAW. Trabajo Fin de Grado, Grado en Ingeniería en Sistemas Audiovisuales y Multimedia, Universidad Rey Juan Carlos, 2016-2017.
- [2] Mediawiki Walter Cuenca (Practicas docentes de desarrollo web.) <http://jderobot.org/Walter-tfg>
- [3] Edgar Barrero. Desarrollo de una aplicación web para sistema domótico. Trabajo Fin de Grado, Grado en Ingeniería de Sistemas Audiovisuales y Multimedia, Universidad Rey Juan Carlos, 2013-2014.
- [4] Mediawiki Edgar Barrero (Surveillance 5.1) <http://jderobot.org/Aerobeat-colab>
- [5] Aitor Martínez.Tecnologías web en plataforma robotica JdeRobot. Trabajo Fin de Grado, Grado en Grado en Ingeniería Telemática, Universidad Rey Juan Carlos, 2015-2016.
- [6] MediaWiki Aitor Martínez (JdeRobotWebClients) <http://jderobot.org/Aitormf-tfg>
- [7] Web oficial MEAN <http://mean.io/>
- [8] Libro NodeJS Introducción a Nodejs a través de Koans ebook
- [9] Pagina Oficial de Mongoose <http://mongoosejs.com/docs/guide.html>
- [10] Pagina Oficial de Express <http://expressjs.com/es/>
- [11] Pagina Oficial de Angular <https://angular.io/guide/quickstart>
- [12] Introducción WebSockets <http://html5index.org/index.html>