



Grado en Ingeniería en Tecnologías de la Telecomunicación

Escuela Técnica Superior de Ingeniería de Telecomunicación

Curso académico 2015-2016

Trabajo Fin de Grado

TITULO
TITULO

Autor: Jorge Ortega Morata

Tutor: Pedro de las Heras Quirós

Madrid 2016

Índice general

1. Introducción	8
1.1. Desarrollo Web	8
1.2. Tecnologías Web	10
1.2.1. Frameworks	10
1.2.2. Librerías	14
1.2.3. Lenguajes	15
1.3. MeteorJS	17
1.3.1. Reactividad	17
1.3.2. Sistema de plantillas reactivas	18
1.3.3. Comunicación con el servidor	18
1.4. MongoDB	21
1.4.1. MongoDB y MeteorJS	22
1.4.2. Publicaciones y Subscripciones	23
1.5. HTML5 Media	24
1.5.1. WebRTC	24
1.5.2. RTCRecorder	24
1.6. AceEditor	25
1.7. Tecnologías Cloud	25
1.7.1. API Soundcloud	25

<i>ÍNDICE GENERAL</i>	3
1.8. Herramientas para trabajo en equipo	26
1.8.1. Git y GitHub	27
1.8.2. PivotalTracker	27
1.8.3. Slack	28
2. Objetivos y metodología	29
2.1. Motivación	29
2.2. Requisitos	30
2.3. Metodología y plan de trabajo	38
3. Diseño y desarrollo de la aplicación	41
3.1. Tipo de Arquitectura	41
3.2. Búsqueda de herramientas	43
3.3. Composición Inicial y entorno de desarrollo	45
3.3.1. Primeros pasos	45
3.3.2. Mixins	46
3.4. Prototipo 1: Registro de usuarios y layout principal	47
3.4.1. Layout principal	47
3.4.2. Registro de usuarios	51
3.5. Prototipo 2: Grabador y reproductor basados en documentos	57
3.5.1. Grabador	57
3.5.2. Reproductor	69
3.5.3. Lista de grabaciones	78
3.6. Prototipo 3: Respuestas a grabaciones	80
3.6.1. Actualización del grabador	80
3.6.2. Nuevas acciones	82
3.6.3. Timeline, relacionados y comentarios	83
3.7. Prototipo 4: Organización en Canales	84

<i>ÍNDICE GENERAL</i>	4
3.8. Prototipo 5: Organización en Lecciones	84
3.9. Prototipo 6: Página de perfil y Contactos	84
3.9.1. Perfil	85
3.9.2. Contactos	86
3.10. Prototipo 7: Conversaciones	89
3.11. Prototipo 8: Emails e integración de servicios de registro	89
3.12. Prototipo 9: Página principal	89
3.13. Prototipo 10: Página de inicio, espacio para tutoriales y features	89
3.14. Prototipo Final: Restricciones de acceso y cross-browsing	89
3.15. Despliegue y pruebas globales	89
4. Experimentación	90
4.1. Motivación	90
4.2. Planteamiento y objetivos	90
4.3. Proceso y realización	90
4.4. Resultados	90
5. Conclusión	91
Bibliografía	92
5.1. Principales Sitios Web	93
5.2. Libros	93
5.3. Artículos	93
5.4. Tutoriales	93
5.5. Paquetes	93
5.6. Repositorios	93
Apéndices	94

<i>ÍNDICE GENERAL</i>	5
A. Diseño de documentos para Mongo	95
B. Diseño de Interfaces	102

Índice de figuras

1.1. jerarquía de carpetas	20
2.1. Esquema metodología de realimentación	38
3.1. Patrones de arquitectura	42
3.2. Diseño de Layout Full-responsive	47
3.3. Diseño header	49
3.4. Diseño sidebar	50
3.5. Diseño formularios de registro	54
3.6. Formularios de registro en Página de inicio	56
3.7. Proceso de grabación	59
3.8. Grabador	68
3.9. Flujo de plantillas del grabador	68
3.10. Proceso de reproducción	71
3.11. Diseño página de detalle	74
3.12. Diseño banner para una grabación	75
3.13. Diseño interfaz del reproductor	76
3.14. Reproductor	77
3.15. Lista de grabaciones	80
B.1. Diseño sidebar	103

B.2. Diseño base grabador	104
B.3. Diseño panel inicial grabador	104
B.4. Diseño panel documentos en grabador	105
B.5. Diseño formulario documentos	105
B.6. Diseño editor en grabador	106
B.7. Diseño final grabación	107
B.8. Diseño proceso de subida	108
B.9. Lista de grabaciones	109
B.10. Miniatura grabación	110

Capítulo 1

Introducción

Este Trabajo Fin de Grado se desarrolla en el ámbito del desarrollo y tecnologías web. En este capítulo introduciremos los conceptos y tecnologías básicas utilizadas y expondremos el paradigma actual en lo que a desarrollo web se refiere.

1.1. Desarrollo Web

En la actualidad el desarrollo web se encuentra muy arraigado en nuestra sociedad. Avanza a una velocidad increíble, tanto que ni nos percatamos de su impacto en todo lo que nos rodea. Acogemos cada nueva tecnología con los brazos abiertos y al poco transcurso de tiempo desde su primer uso nos es insuficiente. Esa insuficiencia impulsa su crecimiento e impide su decadencia.

Para conocer la historia del desarrollo web es necesario remontarse a los orígenes de *Internet*. En 1969 se crea la primera red de comunicación interconectada entre dos computadoras de dos universidades estadounidenses (UCLA y Stanford) llamada **ARPAnet**¹ (Advanced Research Projects Agency Network o Red de la

¹<https://es.wikipedia.org/wiki/ARPANET>

Agencia para los Proyectos de Investigación Avanzada de los Estados Unidos) que funcionaba con un protocolo de intercambio de paquetes llamado **NCP**² (Network Control Program) y que más tarde se sustituyó por **TCP/IP**³ por su robustez frente a colisiones. En 1986 comenzó la construcción de la primera infraestructura en forma de árbol de Internet llamada **NSFNET**⁴ la cual se complementó con otras redes en EEUU. Después se crearon otras redes troncales en Europa y junto con las anteriores ya formaban el *backbone* o red troncal básica de Internet. Más tarde, en 1989 con la creación de la arquitectura de capas OSI en los computadores, comenzó a ser tendencia el utilizar diversos protocolos de comunicación a través de dicha red.

El desarrollo web se inició con la propia web o lo que conocemos como **World-WideWeb**⁵ (WWW) que fue el primer cliente web creado por el *CERN*, cuyo equipo también creó el lenguaje *HTML*⁶ (HyperText Markup Language). Lenguaje básico a la hora de estructurar la información en un sitio web.

El desarrollo web comprende todo el proceso de creación de un sitio web. La elección de las herramientas a utilizar, de diseño y desarrollo, qué metodología seguir, prototipado, propuesta final y lanzamiento. Es una labor compleja transformar una idea en algo físico y dotarla de utilidad para la sociedad. Es un reto que yo, como alumno, nunca había tenido la oportunidad de afrontar y me alegro de haber tenido la ocasión de realizar este proyecto.

La labor del desarrollador web en la actualidad es mucho más sencilla y accesible gracias a las tecnologías que surgen cada pocas semanas o días. Con el término

²https://en.wikipedia.org/wiki/Network_Control_Program

³https://es.wikipedia.org/wiki/Modelo_TCP/IP

⁴<https://es.wikipedia.org/wiki/NSFNet>

⁵https://es.wikipedia.org/wiki/World_Wide_Web

⁶<https://es.wikipedia.org/wiki/HTML>

open source en auge, millones de usuarios de Internet quieren aportar su granito de arena a esta labor y propician un crecimiento en herramientas web increíble y en ocasiones vertiginoso.

1.2. Tecnologías Web

Entran en este ámbito todas aquellas herramientas creadas específicamente para generar contenido web. Según su finalidad dentro del desarrollo web podemos diferenciarlas en Frameworks, librerías y lenguajes.

1.2.1. Frameworks

Según su definición un **framework**⁷ es un *conjunto de conceptos y prácticas estandarizados* diseñado para afrontar un problema en particular, en este caso, el proporcionar una *infraestructura de software* a la hora de crear una aplicación web. Al usar un framework debemos seguir una serie de reglas establecidas según su diseño a la hora de organizar el código. Hoy en día estamos siendo testigos de la "*Batalla de los frameworks*" en lo que a desarrollo web se refiere y es que su crecimiento y potencial es increíble. Es muy importante conocer los puntos fuertes y débiles de cada uno de ellos a la hora de diseñar una aplicación ya que no todos incorporan los mismos conceptos y prácticas. Además también es imprescindible entender el *entorno de ejecución* de cada uno de ellos a saber: cliente (**Front-End**), servidor (**Back-End**)⁸ o ambos (**Full Stack**).

⁷<https://es.wikipedia.org/wiki/Framework>

⁸https://es.wikipedia.org/wiki/Front-end_y_back-end

En la actualidad se utilizan numerosos frameworks entre los que destacan: **AngularJS**⁹, **EmberJS**¹⁰, **Django**¹¹, **ReactJS**¹², **MeteorJS**[5], **BackboneJS**¹³ y **ExpressJS**¹⁴. Todos utilizan *Javascript*¹⁵ como lenguaje de desarrollo a excepción de Django que utiliza *Python*¹⁶.

AngularJS

Es un framework front-end. Hace posible realizar peticiones REST y se pueden desarrollar proveedores que brindan servicios al cliente que se encuentran en el lado del servidor. La principal característica de Angular es que mediante su concepto de directiva podemos construir toda la aplicación de forma modular. Además cuenta con two data-binding que permiten el renderizado reactivo y dinámico de sus plantillas o módulos.

AngularJS es una creación de Google y es el framework más utilizado hoy en día, por lo que posee una gran comunidad, y uno de los más pesados en lo que respecta a tamaño.

EmberJS

Se trata de un framework front-end muy potente diseñado para crear aplicaciones grandes. Mediante la librería Handlebars¹⁷ que incorpora podremos crear plantillas dinámicas gracias al data-binding que presenta. Además también posee un CLI (Interfaz de Línea de Comandos) que nos permitirá configurar todo

⁹<https://angularjs.org/>

¹⁰<http://emberjs.com/>

¹¹<https://www.djangoproject.com/>

¹²<https://facebook.github.io/react/>

¹³<http://backbonejs.org/>

¹⁴<http://expressjs.com/es/>

¹⁵<https://es.wikipedia.org/wiki/JavaScript>

¹⁶<https://www.python.org/>

¹⁷<http://handlebarsjs.com/>

mediante comandos. Posee el módulo de routing más avanzado de todos los frameworks. Es una excelente herramienta para desarrollar un cliente con un buen nivel de potencial.

Django

Django, al igual que Ruby on Rails o MeteorJS no se debería de considerar framework sino plataforma de desarrollo web, ya que es full-stack y posee su propio CLI. El lenguaje de desarrollo es Python. Posee todas las herramientas necesarias para generar un servidor y un cliente. Es una herramienta muy completa. En el momento de su lanzamiento experimentó una gran acogida y aún conserva su fama tras sus numerosas actualizaciones.

ReactJS

Este es el framework más limitado en lo que a ámbito de ejecución se refiere (desarrollo de las vistas en el cliente y poco más). Pero es increíblemente flexible. La forma en la que se crean las plantillas en ReactJS (sin necesidad de escribir HTML) es increíble. Ha sido creado por Facebook y se utiliza cada vez más por su concepto de reactividad.

MeteorJS

MeteorJS [5], como decíamos de Django, se le debería tratar como una plataforma de desarrollo. Corre sobre NodeJS¹⁸ y la principal ventaja que ofrece es que el servidor es completamente transparente al desarrollador. Solo debe preocuparse de crear el modelo de datos (más bien enunciarlo porque cuenta con MongoDB[10] como base de datos), las rutas, las publicaciones de datos reactivos y de las plantillas de la aplicación. Ha sido el pionero en lo que respecta al concepto de reactividad

¹⁸<https://nodejs.org/en/>

antes que ReactJS. Es una herramienta a tener en cuenta por su gran comunidad y sus paquetes específicos y por su capacidad para el prototipado rápido.

Exploraremos más a fondo las características de este framework más adelante ya que es el framework elegido para el desarrollo de nuestra aplicación que es de lo que trata este proyecto.

BackboneJS

Se trata de otro de los frameworks front-end más usados en la actualidad y no es de extrañar debido a su sencillez y capacidad. Es ideal para aplicaciones pequeñas y medianas.

ExpressJS

Este es un framework back-end y permite crear un servidor NodeJS en pocos minutos para sólo preocuparse del desarrollo front-end. Actualmente tiene cabida en cualquier proyecto por su fácil integración con cualquiera de los frameworks front-end.

Bootstrap [8]

Es más una librería que un framework pero dado que posee un paradigma especial y unas reglas claras a la hora de usarlo se le considera un framework front-end. Permite el dotar a tu aplicación de estilo rápidamente puesto que ya tiene definidos estilos por defecto, y crear módulos funcionales para organizar el contenido, además de animaciones. Es un framework destinado a la labor de maquetación y es el más utilizado hoy en día. Aunque también se utilizan otros como Foundation¹⁹ o Pure²⁰ que también poseen buenas características para dicha labor.

¹⁹<http://foundation.zurb.com/>

²⁰<http://purecss.io/>

1.2.2. Librerías

Las librerías son un conjunto de utilidades programadas en un lenguaje dado que proporcionan un servicio concreto al desarrollador. Al contrario que un programa no están pensadas para un uso automático, es decir, no tienen un inicio. Sólo son usadas por programas. En lo que se refiere a desarrollo web son utilizadas para aportar servicios al desarrollador de la aplicación. La librería líder en este ámbito es JQuery[11][12] y sus descendientes como JQueryUI²¹. Otra librería cada vez más utilizada es UnderscoreJS.

JQuery

Se trata de una librería que permite manipular el DOM²² (Document Object Model) desde la lógica de la aplicación. No necesitamos hacer uso del objeto document para realizar búsquedas, añadir etiquetas y demás operaciones de manipulación. Mediante JQuery podremos establecer eventos, manipular dinámicamente el DOM añadiendo etiquetas y estilos y mucho más. Es una herramienta indispensable para el desarrollador web. Lo es tanto que incluso otras librerías y paquetes destinados a diferentes frameworks la usan.

UnderscoreJS [13]

Esta es una librería que permite manipular objetos, colecciones de objetos y arrays. Permite realizar operaciones de filtrado muy avanzadas sobre una colección de objetos y además realizar iteraciones de forma funcional mediante sus métodos

²¹<https://jqueryui.com/>

²²https://es.wikipedia.org/wiki/Document_Object_Model

1.2.3. Lenguajes

Existen numerosos lenguajes en desarrollo web: Java (para servidores principalmente), Javascript, CoffeeScript, TypeScript, HTML5, Jade, CSS, Less, Sass, Python, Ruby, etc. Pero hay que tener en cuenta que lo que el navegador compila e interpreta son ficheros CSS, HTML y lenguajes de lógica de la aplicación como Javascript. Por lo que si desarrollamos con Jade, Less, Sass o CoffeeScript o TypeScript debemos de preprocesarlos hacia los tres principales lenguajes mencionados anteriormente.

HTML5

Se trata de la quinta versión del lenguaje HTML. Posee una variante de sintaxis básica conocida como HTML5 y otra XHTML conocida como XHTML5 y se sirve como sintaxis XML (eXtensible Markup Language) concebido por el World Wide Web Consortium (W3C) con el fin de almacenar datos de forma legible y que complementa en la mayoría de aplicaciones al documento HTML.

En esta quinta versión se han desarrollado nuevas etiquetas que ayudan a realizar un documento más semántico y legible, además de proporcionarnos herramientas de dibujo en 2D y 3D, etiquetas para introducir audio y video o de formato. Etiquetas como: `<article>`, `<aside>`, `<audio>`, `<video>`, `<canvas>`, `<datalist>`, `<details>`, `<dialog>`, `<embed>`, `<figure>`, `<footer>`, `<header>`, `<mark>`, `<meter>`, `<nav>`, `<output>`, `<progress>`, `<ruby>`, `<rp>`, `<rt>`, `<section>`, `<source>` y `<time>`.

También incorpora herramientas nuevas como un visor de fórmulas matemáticas (MathML), tecnología Drag & Drop (arrastrar y soltar objetos basado en eventos), ejecución en paralelo mediante WebWorkers, comunicación bidireccional

entre páginas mediante WebSockets, APIs para almacenamiento (Local & Global Storage), geolocalización y para trabajar en local (Off-line).

La incorporación de estas nuevas herramientas reduce la necesidad del desarrollador a utilizar plugins externos.

CSS3

CSS es el lenguaje utilizado para crear la presentación y dar estilo al documento HTML o XML. En esta tercera versión se introducen nuevas funcionalidades como animaciones 3D, transiciones, estructuración mediante propiedades grid (rejilla), media queries para establecer cambios de estilo conforme el tamaño de la pantalla varía, etc.

Javascript [9]

Es el lenguaje de programación más usado hoy en día en el desarrollo web (sobre todo en el lado del cliente). Fue creado por Brendan Eich de Netscape como dialecto de ECMAScript. Su primer nombre fue Mocha, después LiveScript y finalmente Javascript. Es un lenguaje orientado a objetos, basado en prototipos, funcional (las funciones son objetos), posee un tipado muy débil y es dinámico.

Como se ha comentado, el marco de utilización de este lenguaje en el desarrollo web son los frameworks front-end. Aunque también se utilizan frameworks basados en NodeJS o el propio NodeJS para desarrollar mediante este lenguaje en el lado del servidor en numerosos proyectos.

Javascript sigue creciendo y actualizándose. En 2015 fue lanzado el estándar ECMAScript6, el cual dota a javascript de nuevas funcionalidades y módulos co-

mo un nuevo paradigma de orientación a objetos basado en clases, iteradores o promesas para programación asíncrona.

1.3. MeteorJS

Meteor es una *plataforma* que permite crear aplicaciones Web en *tiempo real* basada en **NodeJS**. Fue creado con el propósito del *prototipado rápido* y en este ámbito supera a la mayoría de *frameworks*. Ha sido el primero en introducir el principio de *reactividad* en el desarrollo web. Soporta **MongoDB** como tecnología de base de datos y posee un asombroso concepto de *subscripciones a publicaciones* procedentes de *colecciones* que hacen del *renderizado* del **DOM** un proceso muy rápido gracias a que mantiene en el cliente una mini base de datos llamada *Mini-Mongo*. El cliente es capaz de modificar dicha base de datos y observar los cambios directamente que más tarde se actualizarán en la base de datos del servidor.

1.3.1. Reactividad

EL gran potencial de Meteor es debido a este *principio*. Se basa en observar los cambios sobre una *fuentes* en tiempo real y actuar en consecuencia, dotando a las aplicaciones de un *dinamismo* muy especial. Adiós a los *Listeners* y al *binding* sobre elementos **HTML**, no hacen falta si sabes aprovechar este principio y sus *entidades*. En meteor existen numerosas entidades que proveen *reactividad* y otras muchas que permiten crear nuevas *entidades reactivas*. Las **fuentes reactivas** que puedes controlar de manera simple en Meteor son las *variables de sesión* almacenadas en *Session*. Mediante la sentencia *Session.set(key[String],value)* ya tienes una fuente reactiva a tu disposición. Sólo necesitas algo que sepa escuchar sus cambios y actuar (*helpers* o *Tracker*).

El modulo **Tracker** posee un método *.autorun()* que permite ejecutar código cuando una fuente reactiva cambia. Además puede directamente asociarse a la lógica de cualquier plantilla dentro del método *.rendered()* del controlador. Esto permite dotar a la plantilla de dinamismo, por ejemplo realizar **subscripciones dinámicas** sobre una colección enlazado con los eventos de la plantilla. (Auto-completados basados en subscripciones, Botones para cargar más contenido, etc).

1.3.2. Sistema de plantillas reactivas

Meteor utiliza una *biblioteca* muy poderosa para crear *interfaces de usuario* que se actualizan en tiempo real llamada **Blaze**. Cumple el mismo propósito que **Angular**, **Backbone**, **Ember**, **React**, **Polymer** o **Knockout** en este ámbito pero es mucho más sencilla de utilizar, incluso *transparente* para el programador. Su labor no sería posible sin *Tracker*, un módulo de Meteor que permite gestionar *procesos reactivos* de manera limpia, y sin **Spacebars** (parecido a **Handlebars**), el lenguaje particular de Meteor para definir las plantillas y que aprovecha al máximo la funcionalidad de *Tracker*.

1.3.3. Comunicación con el servidor

La comunicación con el servidor se basa en el protocolo HTTP que Meteor integra de manera transparente al programador mediante su módulo *methods* al que se accede mediante *Meteor.methods()* y para la petición de recursos se utiliza algún paquete creador de rutas como *IronRouter* o *FlowRouter*. También posee el módulo *http* para realizar peticiones desde el cliente al servidor y a terceros. Todas las peticiones son asíncronas y como tales se les asocian *callbacks* (funciones que se ejecutarán una vez haya terminado la ejecución de la petición).

Cada vez que se define una plantilla mediante Spacebars se crea un objeto plantilla `Template.name` y que lo tendremos accesible a la hora de dotarla de funcionalidad mediante javascript. Es un tipo de controlador. Spacebars permite el paso de datos de la plantilla al controlador y viceversa, esto se denomina *two data-binding* y supone una poderosa herramienta a la hora de crear componentes aislados puesto que su configuración puede realizarse vía Spacebars. Este proceso lo realiza mediante la declaración de helpers o ayudantes de plantilla y se definen mediante la función `helpers()` del controlador. La gran ventaja de esto radica en que los helpers son funciones javascript asociadas a una fuente reactiva. Esto quiere decir que en el momento que esa fuente cambia los helpers se actualizan y se actualiza el contenido HTML asociado a ellos. Además de permitir crear componentes reusables, éstos son dinámicos (reactivos) en su instanciación y durante su uso.

Aparte de los helpers al controlador se le puede asociar un mapa de eventos relacionados con la plantilla mediante la función `events()` que toma como parámetro un objeto tipo:

Jerarquía de carpetas y orden de carga

Para aplicaciones pequeñas es posible escribir el código ejecutable por el cliente y por el servidor en una misma carpeta. Para ello Meteor cuenta con las funciones `.isClient()` y `.isServer()` para especificar qué código debe ejecutarse en cada entorno. Para aplicaciones más grandes la estructura es un poco peculiar y se debe generar teniendo en cuenta el orden de carga según la *jerarquía de carpetas*. Este orden de carga aunque sea muy estricto provee de una flexibilidad asombrosa y permite crear cualquier aplicación de forma *modular*. La jerarquía de carpetas sería la siguiente:

```
/app
/lib --primero en cargar tanto en el servidor como en el cliente
  /collections
  /router.js
/client --solo se carga en el browser
  /lib --primero en cargar
  /modules --plantillas
    /module_name
      /module_name.js
      /module_name.html
      /module_name.scss
  /components --componentes reutilizables
    /component_name
      /component_name.js
      /component_name.html
      /component_name.scss
  /styles --estilos reutilizables
  /main.js --ultimo en cargar
  /main.html --ultimo en cargar
  /index.scss --primera hoja de estilos en cargar
/server --solo se carga en el servidor
  /publications.js --declaración de las publicaciones de la base de datos
  /services --servicios ofrecidos para el cliente (Meteor.methods)
    /service_name.js
/public --fuentes e imágenes estáticas
```

Figura 1.1: jerarquía de carpetas

La carpeta */lib* de más alto nivel dentro de la jerarquía contiene todos los *ficheros comunes* a ambos entornos (cliente y servidor) y es la primera en cargar. */server* y */client* contienen todos los *ficheros ejecutables* por el servidor y por el cliente respectivamente. Dentro de cada una de las carpetas anteriores existe un orden de carga. Si poseen carpeta */lib* será la primera en cargar, después es el turno de los demás ficheros en *orden alfabético* y, por último, los ficheros *main.** sea cual sea su extensión. En la carpeta */public* se encuentra el *contenido estático* de nuestra aplicación (fuentes, imágenes, etc).

Como podemos ver según el orden y ámbito de ejecución Meteor ofrece un *entorno de ejecución simétrico* (se ejecuta tanto en el cliente como en el servidor) dentro de la carpeta */lib* de más alto nivel en la jerarquía. La principal ventaja de esto es que no tenemos porqué replicar código. Podemos crear *constructores* y demás funcionalidad necesaria en ambos entornos una sola vez y Meteor se encarga de saber que cargar en cada uno.

Además, al ser MeteorJS una plataforma de desarrollo, no es necesario un *gestor de tareas* como **GruntJS** o **GulpJS**. Estos se encargan de **automatizar tareas** tales como establecer la carga de ficheros sobre el *documento HTML*, *minificar* todos los ficheros, establecer la *configuración del servidor* o arrancar nuestra aplicación. Meteor posee un **CLI** (Interfaz de línea de comandos) que mediante el comando *meteor* ya se encarga de establecer las configuraciones iniciales, cargar todos los ficheros según el orden descrito e incluirlos en el documento y arrancar nuestro servidor. El *minificado* no es necesario en un primer momento, puesto que con el *deploying* (despliegue) se realizará. Existen una gran cantidad de paquetes y constructores que lo harán de forma automática.

1.4. MongoDB

MongoDB es una base de datos **no relacional** cuya arquitectura *se basa en documentos*. Al no ser *relacional* como **mySQL**, **Oracle** o **PostgreSQL** carece de *claves primarias*. No hay que declarar un *modelo de datos* puesto que lo que se almacenan son objetos en formato **BSON** (muy parecido a **JSON**) en el que todos los *atributos* pueden ser utilizados como *clave* a la hora de realizar búsquedas. Cada vez que un documento es insertado se le asocia un *índice único*. Aunque no sea relacional ofrece la posibilidad de realizar un diseño de este tipo. Esto es enlazando objetos pertenecientes a *colecciones* (tablas) diferentes mediante su identificador o cualquier atributo válido. La gran ventaja de utilizar este tipo de base de datos es la **rapidez de acceso**. Como cualquier *objeto javascript* un documento puede *embeber* otros documentos (objetos) a los que se les puede establecer un índice y realizar **búsquedas indexadas**. Además *MongoDB* cuenta con módulos como *\$agregation* que permite establecer reglas para cada colección que permiten realizar búsquedas más complejas como por ejemplo establecer para qué campo del

documento se realiza una búsqueda mediante *expresiones regulares*.

Debido a que es una base de datos no relacional podemos *embeber entidades* que dependan de otras en éstas. De no hacerlo así el proceso de *borrado de datos* hay que tomarlo con calma debido a que este tipo de base de datos no posee **joins** ni de herramientas que hagan que la **atomicidad** de los datos se mantenga como ocurre en las bases de datos relacionales. Aunque si queremos también realizar un diseño desde un enfoque más limpio deberíamos de separar todas las entidades.

1.4.1. MongoDB y MeteorJS

Al crear una aplicación mediante el CLI de MeteorJS directamente se crea una base de datos MongoDB. Aunque Meteor no trabaja con otro tipo de base de datos en un principio, se puede cambiar mediante la instalación de paquetes. En la actualidad existen paquetes de bases de datos relacionales para Meteor que poseen reactividad y es este principio en el que se basa Meteor.

Las tablas creadas en MongoDB en Meteor se convierten en colecciones, un wrapper para ofrecer funcionalidad desde la aplicación y dotarlas de reactividad (las convierte en fuentes reactivas). Este objeto en el que se engloba a la tabla o entidad ofrece los métodos `.find()`, `findOne()`, `update()`, `remove()`, `insert()`, `allow()` y `deny()` que son los más usados. Hay que tener en cuenta que las colecciones en Meteor se declaran en la carpeta `/lib`, cuyo contenido será ejecutado tanto en el entorno del cliente como en el del servidor. Esto quiere decir que cada entorno tendrá una instancia de cada colección y esto al igual que es ventajoso en cuanto a rapidez en el cliente (puede acceder a la base de datos directamente "miniMongo"), es peligroso por el mismo motivo. Para ello existen los métodos `deny()` y `allow()` que establecen qué operaciones sobre la colección están permitidas en el cliente y

cuáles no.

Lo más sensato es permitir insertar y denegar el permiso para realizar cualquier alteración sobre otros documentos ya presentes, al menos directamente. Para ello se usa el módulo `methods` de Meteor que permite configurar métodos a los que llamar desde el cliente (también se declaran en la carpeta `/lib`) y que se ejecutan en el servidor (donde no existe ningún tipo de restricción).

1.4.2. Publicaciones y Subscripciones

Puesto que las instancias de las colecciones se encuentran accesibles también en el cliente debe haber un control sobre el contenido de las mismas dentro de MiniMongo. Para ello se utilizan las publicaciones y las subscripciones. Las publicaciones se realizan en el lado del servidor y el cliente se suscribe a ellas. La moneda de cambio son los cursores. Un cursor es una fuente reactiva que engloba uno, varios o ningún documento procedente de una colección. El cliente al suscribirse a una publicación obtiene el cursor y este lo transforma en documentos que se almacenan dentro de MiniMongo donde tendrá accesibles los documentos. Lo bueno de esto es que como se ha dicho los cursores son fuentes reactivas, esto quiere decir que en el momento que se produzca algún cambio que altere el cursor al que se está suscrito, la publicación cambiará y la subscripción se actualizará. Para aprovecharse de este fenómeno el cliente necesita extraer un cursor procedente de MiniMongo mediante `NombreColección.find()` o `NombreColección.findOne()` y asociarlo a un helper dentro de la lógica de la plantilla.

1.5. HTML5 Media

Con la llegada y estandarización de HTML5 cada vez se trabaja más en herramientas que faciliten la comunicación entre usuarios y que, en definitiva, brinden servicios interactivos a los mismos en tiempo real. Una de esas herramientas es WebRTC (Web Real-Time Communication).

1.5.1. WebRTC

Se trata de una API creada para permitir realizar llamadas de voz, chat de video e intercambio de archivos P2P (Peer to Peer) sin la necesidad de plugins. Es Open Source (Código abierto). Fue creado primero por Google y la W3C se encarga ahora de su estandarización. Su desarrollo está en proceso, por lo que se comporta de manera inestable en su funcionalidad avanzada. Los navegadores que la soportan son Chrome, Firefox y Opera hoy en día. Esto se debe a que su módulo navigator facilita el acceso a los recursos media del ordenador (micrófono, webcam).

1.5.2. RTCRecorder

Se trata de una API basada en WebRTC que proporciona una serie de herramientas para grabar video y audio de manera sencilla y pudiendo exportar el archivo creado y almacenarlo tanto en servicios cloud como en local. El creador de esta API también ha desarrollado otros módulos basados en WebRTC capaces por ejemplo de grabar video y audio sobre un elemento canvas. Hablaremos de esta API más adelante puesto que ha sido integrada en el proyecto.

1.6. AceEditor

AceEditor es una API que permite transformar un contenedor HTML (por ejemplo un `<div>`) en un editor de texto completo. Además proporciona una serie de métodos que permiten personalizarlo y capturar eventos que se produzcan en dicho editor. Existen otras APIs parecidas como CodeMirror, pero ésta posee más documentación y está disponible como paquete para Meteor. Exploraremos más a fondo esta API más adelante.

1.7. Tecnologías Cloud

Hoy en día el término Cloud está muy extendido. La mayoría de aplicaciones y sitios web utilizan tecnologías cloud para almacenar grandes cantidades de datos y liberar memoria propia de la aplicación o bien son utilizadas como base de datos. Ejemplos de tecnologías cloud son Google Drive, Dropbox, Youtube, Amazon S3, Soundcloud, mLab o DigitalOcean.

La mayoría de los anteriores son utilizados como servicios cloud extra o para almacenar contenidos de la aplicación (ficheros, imágenes, audios, videos). Amazon S3 o mLab son utilizados como base de datos de la aplicación y su ventaja radica en que, a la hora de realizar el despliegue, utilizamos servidores externos en los que almacenaremos la base de datos de nuestra aplicación.

1.7.1. API Soundcloud

Soundcloud es un sitio web que permite alojar archivos de audio al estilo de Youtube con vídeos. Posee un API disponible en diferentes lenguajes de programación como Ruby, javascript y PHP para realizar peticiones REST y por tanto un espacio para desarrolladores en el cual crear diferentes aplicaciones con las que

comunicarse la API. Se trata de una solución factible a la hora de desarrollar un proyecto pequeño ya que tiene limitaciones en lo que respecta a espacio. Si nos encontramos ante un proyecto de gran envergadura necesitaremos explorar otras vías como GridFS aplicado a otro servicio cloud de base de datos como Amazon S3.

También proporciona herramientas de Streaming de gran utilidad a la hora de reproducir dichos archivos de audio en nuestra aplicación de forma remota.

Exploraremos a fondo esta API puesto que es una de las herramientas más importantes incluidas en este proyecto.

1.8. Herramientas para trabajo en equipo

El mundo del desarrollo web es altamente competitivo y como tal exige la obtención de resultados muy a corto plazo. Para conseguir este objetivo surgen las metodologías ágiles como SCRUM que, según su definición, no es más que un proceso en el que se aplican una sucesión de buenas prácticas para trabajar colaborativamente y en equipo. La finalidad es conseguir un equipo altamente productivo. La base de esta metodología es la realimentación o feedback, es decir, es un proceso circular compuesto por varias fases y realimentado en el cual el cliente toma conciencia de cada ciclo aportando sus críticas.

Para reforzar este proceso y facilitar la labor del desarrollador y de todo el equipo existen distintas herramientas como GitHub, PivotalTracker y Slack entre otras. Estas herramientas han sido utilizadas a lo largo de este proyecto.

1.8.1. Git y GitHub

Git es el sistema de control de versiones más usado en el mundo del desarrollo. Crea una copia del proyecto en un repositorio y a través de sus commits se almacenan versiones recuperables del mismo. Incorpora un sistema para generar ramas de versiones que posteriormente pueden volver a unirse para conformar el resultado final del proyecto o finalizar alguna fase. Esto lo hace verdaderamente potente a la hora de utilizarlo en grupo y por tanto es necesario compartir el repositorio entre los integrantes del equipo de desarrollo. Esto se hace mediante GitHub, una plataforma en la que almacenar proyectos públicos o privados que integra el sistema de control de versiones mencionado anteriormente. Permite la copia de cualquier versión desde un usuario a otro (fork) y puede desarrollarse un seguimiento de todo el proyecto mediante el espacio para Wiki.

Además permite la sincronización de otros servicios afines al proyecto como PivotalTracker.

1.8.2. PivotalTracker

La primera fase de toda metodología ágil en un proyecto de desarrollo se basa en el análisis y la extracción de requisitos. Estos requisitos son llamados historias de usuario y son el elemento base de PivotalTracker.

PivotalTracker es una plataforma de organización de proyectos mediante tareas o items a completar. Cada historia de usuario normalmente es dividida en distintas tareas. Una vez creado un proyecto en esta plataforma y establecido los miembros comienza la asignación de tareas. Se establecen diferentes espacios o ambientes de trabajo que indican la prioridad de las tareas (Icebox, current, done, etc). Con la creación de cada tarea viene la estimación del tiempo de trabajo de la misma y a

mayor valor de estimación más miembros la tendrán asignada. Una vez finalizada la tarea es necesaria la validación de la misma por el resto del equipo. De esta manera está asegurado el correcto desarrollo del proyecto.

1.8.3. Slack

Slack es una plataforma de comunicación destinada a grandes proyectos. Se organiza mediante canales y permite el intercambio de ficheros, información y mucho más a través de chat. Además ofrece la posibilidad de vincular cada canal a los servicios utilizados en el proyecto como GitHub y PivotalTracker por lo que cada acción y avance quedará reflejado y notificado a los miembros del equipo. Se trata de una herramienta muy útil para el trabajo en remoto y como historial de proyecto.

Capítulo 2

Objetivos y metodología

2.1. Motivación

En la actualidad existen distintas aplicaciones y sitios web destinados a facilitar la labor del desarrollador. Sitios como PivotalTracker, Github, GitBucket, CodePen, Codecademy o StackOverflow. Estos sitios proporcionan herramientas para un correcto desarrollo de cualquier proyecto. En el caso de Github o GitBucket facilitan la creación de repositorios remotos con un sistema de control de versiones. PivotalTracker permite organizar las tareas de un proyecto y CodePen, Codecademy y StackOverflow son sitios web para el aprendizaje y compartir conocimientos.

Es precisamente el aprender y compartir lo que mueve el mundo del desarrollo. Cualquier desarrollador utiliza ideas propias y de otros desarrolladores para realizar cualquier proyecto. Pero muchas veces esas ideas no son muy intuitivas o fáciles de aprender a partir de un artículo o un fichero de código completado. Por esto muchas veces recurrimos a plataformas de difusión de video para poder aprender rápidamente.

De esta necesidad surge la motivación de crear una aplicación web que sirva de plataforma para todos los desarrolladores. Una plataforma donde puedan aprender rápidamente, intercambiar conocimientos de manera interactiva y que permita la comunicación entre sus usuarios. Puesto que la información debería ser lo más visual e interactiva posible el formato se basará en grabaciones de código y audio sobre un editor.

2.2. Requisitos

Con la motivación surge el proceso de pensar en las necesidades que tendrá el usuario al utilizar la aplicación. Estas necesidades se traducen en requisitos que debe cumplir la aplicación y que hay que tener presentes en todo momento del diseño y del desarrollo. Tanto nuestro concepto de la aplicación como las necesidades del usuario pueden verse alteradas durante el desarrollo del proyecto por lo tanto pueden surgir nuevos requisitos y verse alterados los ya establecidos.

A continuación se exponen todos los requisitos que debería cumplir nuestra aplicación y que se han ido extrayendo a lo largo de la realización del proyecto:

1. Se deberá proporcionar una interfaz de inicio para que el usuario pueda explorar las características de la aplicación, aprender, y poder crear un usuario para acceder a la misma.
2. Los usuarios deben ser autenticados para poder acceder a la mayoría de la funcionalidad de la aplicación.
3. Un usuario podrá registrarse introduciendo un nombre de usuario, un correo electrónico y una contraseña o bien mediante los servicios integrados de Facebook, Github y Google+.

4. Es necesario proporcionar al usuario un proceso de validación o verificación de email, un proceso de cambio de contraseña y otro de recuperación de la misma.
5. Un usuario podrá recuperar su contraseña siempre que tenga asociado un correo electrónico y éste haya sido verificado y puede estar autenticado o no.
6. Un usuario podrá acceder a la aplicación introduciendo su nombre de usuario o email asociado y su contraseña.
7. Un usuario podrá crear canales, lecciones, grabaciones y conversaciones.
8. Un usuario podrá realizar peticiones de contacto a otros usuarios para añadirlos a su lista de contactos.
9. Un usuario podrá aceptar o rechazar peticiones.
10. Un usuario podrá reenviar una solicitud de contacto que haya sido rechazada.
11. Un usuario podrá crear conversaciones con otros usuarios que estén en su lista de contactos.
12. Cualquier contenido se podrá etiquetar para facilitar la recomendación y búsqueda del mismo, con la excepción de las conversaciones.
13. Cualquier contenido se podrá votar y comentar con la excepción de las conversaciones.
14. Se podrán crear respuestas a los comentarios.
15. Todo contenido deberá mostrar contadores de votos, subcontenidos y usuarios suscritos (cuando proceda).
16. Todo contenido podrá ser editado.

17. Todos los contenidos podrán ser borrados a excepción de las conversaciones.
18. No se podrá borrar ningún contenido con subcontenidos.
19. Para una mejor experiencia es necesario que el usuario conozca lo que sucede dentro de aplicación para ello debe existir un módulo que permita crear notificaciones personalizadas y de interés para el usuario.
20. El usuario deberá tener acceso rápido a las listas de contenido.
21. Todas las listas de contenido podrán ser filtradas por dos tipos de filtros: más recientes y más populares.
22. Para todo tipo de contenido existen dos roles con un tipo de acceso diferente a las funcionalidades propias del mismo.
23. La aplicación deberá ser óptima y precisa y proporcionar al usuario una interfaz cuidada e intuitiva.
24. Las grabaciones serán grabaciones de código y audio sobre editor.
25. Las grabaciones serán el elemento atómico en lo que respecta a contenido dentro de la aplicación.
26. El objeto de cada grabación será grabar documentos de código.
27. Cada grabación podrá tener 1 o más documentos.
28. Los documentos tendrán un título único dentro de cada grabación, un modo (lenguaje en el que están programados) y un tema (aspecto en el editor).
29. Un usuario podrá crear grabaciones.
30. Un usuario podrá crear respuestas sobre cualquier grabación a partir de cualquier instante de su reproducción.

31. Las grabaciones podrán ser aisladas o pertenecer a un canal o lección.
32. El usuario podrá navegar, visualizar y reproducir cualquier grabación.
33. El usuario podrá acceder al padre de una grabación (respuesta) desde la página de visualización de la misma.
34. El usuario podrá visualizar tanto las respuestas a una grabación cómo grabaciones relacionadas a la misma.
35. El usuario podrá acceder al canal o lección a la que pertenece cualquier grabación.
36. Las grabaciones no tienen porqué poseer un título único, pero sí deben tener uno. Opcionalmente tendrán una descripción y una lista de etiquetas.
37. Las grabaciones podrán ser editadas mediante una respuesta.
38. El usuario podrá cambiar el instante de la reproducción, pausarla o reanudarla.
39. Un usuario podrá crear canales.
40. Un usuario podrá subscribirse a cualquier canal que no haya sido creado por él mismo para recibir notificaciones relevantes.
41. El contenido de los canales estará formado por una lista de grabaciones.
42. La creación de contenido no está restringida para los canales. Todos los usuarios podrán generar contenido dentro de cualquier canal.
43. Cualquier usuario podrá votar y comentar canales y sus contenidos.
44. Cada canal podrá ser editado por el creador.

45. Un usuario podrá crear lecciones.
46. El contenido de las lecciones serán secciones formadas por una lista de grabaciones.
47. La creación de contenido estará restringida para las lecciones. Sólo el creador de la lección puede crear contenido directo a las mismas a saber: secciones y grabaciones.
48. Un usuario podrá subscribirse a cualquier lección que no haya sido creada por él mismo para poder acceder a su contenido y recibir las notificaciones relevantes.
49. Cualquier usuario suscrito a una lección podrá crear respuestas a todas las grabaciones que existan dentro de la misma.
50. El contenido de una sección se reproducirá mediante el concepto de lista de reproducción.
51. El reproductor deberá identificar si la grabación procede de una lección y si forma parte de una lista de reproducción. Si es el caso deberá proporcionar una interfaz para navegar por la lista de reproducción y establecer opciones tipo: reproducción y repetición automática.
52. Las opciones reproducción y repetición automática estarán asociadas al usuario no a la sección.
53. La reproducción automática provocará que se reproduzca la siguiente grabación de la lista al finalizar la actual.
54. La repetición automática provocará que la reproducción de la lista sea circular (ni principio ni fin).

55. Cualquier usuario que esté suscrito a una lección podrá votar a la misma y a sus contenidos (grabaciones).
56. Cualquier usuario que esté suscrito a una lección podrá comentar la misma y sus contenidos (grabaciones).
57. Cada lección podrá ser editada por el creador.
58. El orden de las secciones podrá ser cambiado por el creador.
59. El orden de las grabaciones que forman una sección podrá ser cambiado por el creador.
60. Las secciones podrán ser borradas por el creador en cualquier momento, siempre que no tengan contenido.
61. Para una mejor experiencia es necesario realizar al usuario recomendaciones de contenido basadas en sus gustos y su recorrido dentro de la aplicación.
62. Para una mejor experiencia es necesario mostrar al usuario las tendencias o el contenido más popular dentro de la aplicación.
63. Para una mejor experiencia es necesario aportar al usuario un mini tutorial compuesto por tareas básicas que le ayude a dar sus primeros pasos en la aplicación.
64. Es necesario que los usuarios puedan explorar una descripción de las características de la aplicación para ello se debe crear un espacio en el que el usuario las conozca.
65. Es necesario que los usuarios puedan aprender cómo usar la aplicación para ello se debe crear un espacio con tutoriales.

66. Es necesario que los contenidos puedan mostrarse en listas para su navegación. Para ello cada contenido debe poseer una miniatura asociada que muestre la información más relevante para cada tipo de contenido.
67. Es necesaria la presencia de un buscador de contenido dentro de la aplicación.
68. Las búsquedas en la aplicación serán dinámicas y con un sistema de etiquetas que se irán sugiriendo de manera automática.
69. Es necesario que los usuarios puedan recibir notificaciones de nuevos mensajes de sus conversaciones abiertas cuando no estén visualizando dicha conversación.
70. Cada usuario deberá poder acceder a toda su información y contenido. Para ello se les proporcionará un enlace en todo momento a su perfil y un acceso a sus contenidos principales.
71. Cada usuario podrá visualizar la lista de sus contenidos, sus subscripciones, sus conversaciones, una lista con las últimas reproducciones y sus contactos.
72. Cada usuario podrá editar su perfil a saber: su avatar, el banner, su descripción y sus emails.
73. Los perfiles podrán ser vistos por todos los usuarios por lo que se establecen dos roles: propietario, visitante.
74. Un visitante ya sea contacto o no del propietario podrá visualizar todo su contenido a excepción de sus conversaciones.
75. Un visitante ya sea contacto o no del propietario podrá acceder al perfil asociado al servicio integrado en el caso de que el propietario se haya registrado mediante dicho servicio.

76. Un visitante que sea contacto del propietario podrá iniciar una conversación o redactar un email al email del propietario en el caso de que éste tenga configurado y verificado algún correo.
77. Un visitante que no sea contacto podrá enviar una solicitud de contacto al propietario.
78. Se debe crear un espacio para realizar solicitudes de contacto que lo componga un buscador (autocompletado de usuarios) y listas de peticiones recibidas y enviadas y su estado.
79. Una conversación debe incluir al menos 2 contactos en el momento de creación.
80. Cada conversación podrá ser editada por todos los miembros.
81. Las opciones de edición de cada conversación están restringidas según roles: creador o líder, invitados.
82. Sólo el líder de la conversación puede cambiar el asunto y expulsar a miembros de la misma.
83. El líder sólo podrá dejar la conversación si antes ha nombrado como nuevo líder a alguno de los miembros invitados.
84. Todos los invitados podrán dejar la conversación en el momento que deseen.
85. Todos los miembros podrán eliminar el historial de mensajes.
86. Todos los miembros podrán invitar a la conversación a usuarios que sean sus contactos.
87. Todos los miembros podrán cambiar el fondo de la conversación y dicha configuración estará asociada a dicho usuario.

88. Las conversaciones son privadas, es decir, ningún usuario puede acceder a ninguna conversación de la que no sea miembro aunque alguno de sus contactos sea miembro.
89. Los emails no son entidades dentro de la aplicación. No se guarda registro de ellos. Simplemente se proporciona una herramienta que permite su redacción y envío.

2.3. Metodología y plan de trabajo

En este Trabajo Fin de Grado hemos seguido una metodología de realimentación o feedback basada en la metodología ágil SCRUM. Se basa en que el producto final es el resultado de una serie de prototipos los cuales han sido implementados en cada iteración del proceso. Podemos ver las etapas de cada iteración en la figura 2.1 a saber: extracción de requisitos, aprendizaje, etapa de diseño, etapa de desarrollo, realización de pruebas y evaluación del producto.



Figura 2.1: Esquema metodología de realimentación

El código fuente de este TFG está disponible en un repositorio público en GitHub y se ha ido documentando en la Wiki del mismo. La documentación incluye manuales de uso de los módulos principales desarrollados, diseño de los objetos en la base de datos y un historial descriptivo de las reuniones realizadas con el tutor. La comunicación con el tutor se ha realizado mediante la plataforma Slack y mediante PivotalTracker se han ido estableciendo y completando las diferentes tareas de cada fase del desarrollo del proyecto.

El plan de trabajo se ha desarrollado en las siguientes fases:

- **Fase1 - Consolidación, búsqueda y aprendizaje:** esta fase corresponde al proceso de extracción de requisitos iniciales y consolidación del concepto de la aplicación, a la búsqueda de herramientas necesarias y al aprendizaje de dichas herramientas (MeteorJS, Javascript, HTML5, CSS3, SASS, WebRTC y RTCRecorder, APIs Soundcloud, AceEditor, IronRouter y demás paquetes).
- **Fase 2 - Prototipado:** esta fase ha sido la que más tiempo ha requerido y en la que se ha empleado la metodología iterativa descrita anteriormente. En cada iteración se ha enunciado y desarrollado un prototipo cuyo resultado ha servido de base al siguiente. Cada fase ha llevado consigo la realización de las siguientes tareas:
 1. Extracción de requisitos
 2. Extracción de entidades
 3. Diseño de la base de datos
 4. Diseño front-end e Interfaces de usuario
 5. Desarrollo de Interfaces

6. Pruebas y evaluación

- **Fase 3 - Final:** una vez implementado el prototipo final se ha procedido a la fase final del proyecto que corresponde a la mejora global y su despliegue.

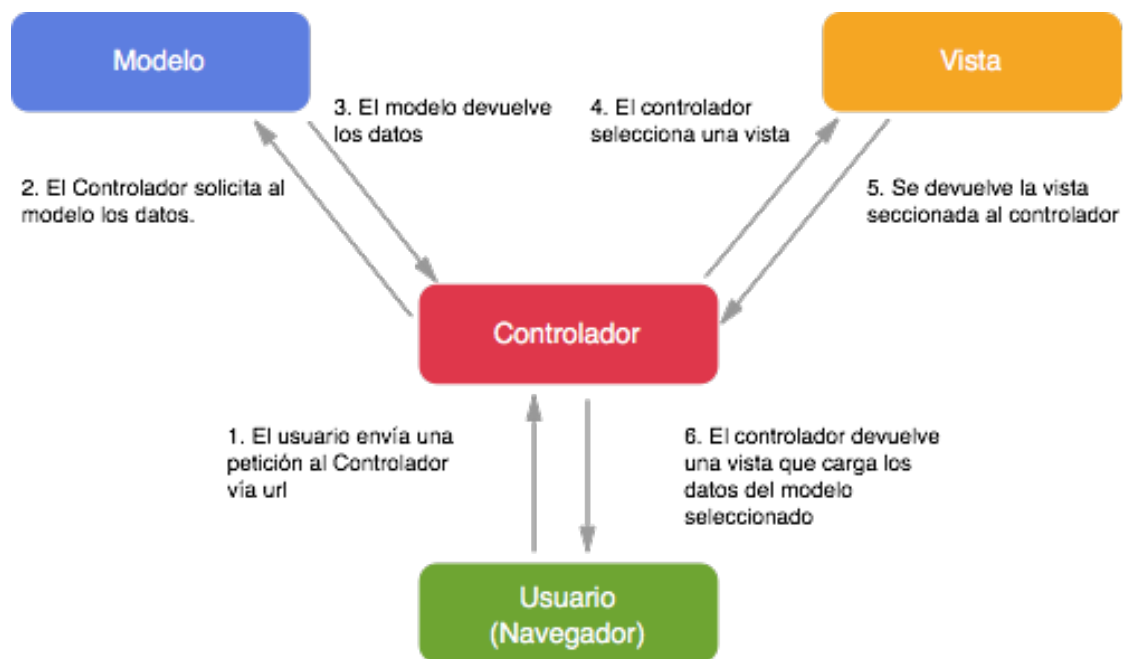
Capítulo 3

Diseño y desarrollo de la aplicación

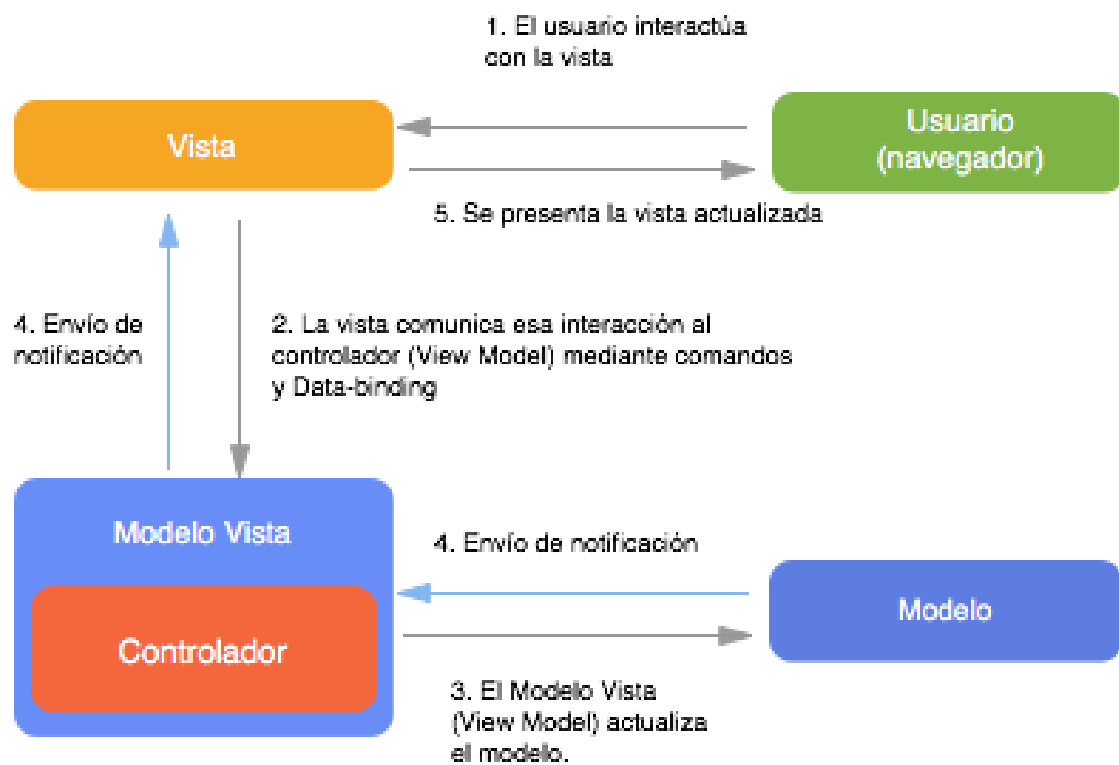
3.1. Tipo de Arquitectura

El modelo de arquitectura más habitual es el **MVC** (Modelo Vista Controlador) (figura 3.1(a)). El modelo correspondería con la arquitectura de base de datos y el diseño de la misma, la vista son las interfaces o plantillas que se muestran al usuario y el controlador es el encargado de dotar a la aplicación de lógica y funcionalidad. Existen otros tipos de arquitectura tales como **MVVM** (figura 3.1(b)), donde el controlador del patrón MVC se sustituye por VM o *ViewModel* que establece que cada vista posee lógica y un sistema de *data-binding* entre plantillas.

La elección del patrón de arquitectura a usar es importante puesto que esa decisión nos limitará a la hora de usar determinadas herramientas.



(a) Patrón de arquitectura MVC



(b) Patrón de arquitectura MVVM

Figura 3.1: Patrones de arquitectura

3.2. Búsqueda de herramientas

Partiendo de los requisitos establecidos que caracterizarán la aplicación podemos afirmar que necesitamos encontrar herramientas que nos permitan construir una aplicación en tiempo real, realizar grabaciones de audio, una interfaz con UX (User eXperience) como parámetro fundamental y que trabaje de manera óptima.

Aplicación en tiempo real y prototipado rápido

Necesitamos algún framework o plataforma que trabaje con el concepto de reactividad o que lo simule. Elegimos MeteorJS por su flexibilidad, su asombroso concepto de reactividad y por su patrón de trabajo que engloba el desarrollo del cliente y del servidor. Debido a esta elección utilizaremos MongoDB como tecnología de base de datos y MVVM como patrón de arquitectura de la aplicación.

Para el enrutamiento es precisa otra herramienta que nos proporcione la funcionalidad de especificar a qué recurso pertenece una plantilla y que datos asociamos a ella. Para ello existen varios paquetes para Meteor que realizan este proceso como IronRouter o FlowRouter. Ambos igual de válidos pero para este proyecto se ha optado por IronRouter, ya que dispone de mayor documentación.

Para el concepto de publicaciones y subscripciones de Meteor usaremos publishComposite, un paquete que permite realizar publicaciones compuestas (varias colecciones con relación de dependencia reactiva) y que siguen manteniendo el principio de reactividad y optimizando nuestro sistema de subscripciones. Sin este paquete realizar esta labor es más compleja.

Grabaciones de audio

Existen numerosas formas de grabar audio vía web y algunas API de sitios como SoundCloud incorporan un grabador de audio directamente. Aunque esta hubiera sido la vía más rápida, la verdad es que no habría sido la más flexible, ya que el hacerlo de esta manera requería que el uploading se efectuara en SoundCloud. Por este motivo se ha utilizado la tecnología de WebRTC para esta tarea y construido un grabador modular que puede incorporarse fácilmente a otros proyectos y que además si se desea usar el servicio de hosting de SoundCloud seguiría siendo factible.

Hosting o Almacenamiento

Utilizaremos el servicio de hosting de SoundCloud para almacenar el audio de nuestras grabaciones. No es lo más sensato para una aplicación real y comercializable puesto que existen restricciones en lo que corresponde a capacidad, pero para nuestra aplicación es más que suficiente.

Interfaz con UX como parámetro de diseño fundamental

Partiendo del requisito de que la aplicación debe ser atractiva al usuario y no sólo en términos visuales sino en eficacia a la hora de gestionar acciones, en este proyecto nos hemos decantado por el framework front-end Bootstrap para la maquetación y por la tecnología Flexbox para dotar de flexibilidad a las plantillas. Además utilizaremos SASS como preprocesador de CSS con el fin de optimizar nuestra arquitectura de estilos mediante un paquete para Meteor.

3.3. Composición Inicial y entorno de desarrollo

Una vez realizada la búsqueda de herramientas comenzamos a componer el entorno de nuestra aplicación. Para este proyecto utilizaremos el programa WebStorm de JetBrains. Incorpora herramientas de búsqueda y sustitución avanzada, terminal para comandos, integración con sistema de control de versiones Git y plugins que facilitan la labor de desarrollo como elmet.

3.3.1. Primeros pasos

Gracias al CLI de Meteor generamos nuestra aplicación mediante el comando: `meteor create <AppName>`. Esto nos genera una carpeta con tres ficheros: `index.html`, `index.js` e `index.css`. En este momento ya tenemos nuestra aplicación Meteor creada.

Ahora debemos estructurar nuestra aplicación según la jerarquía mostrada en la figura 1.1 creando los ficheros y carpetas necesarios.

Una vez estructurada la aplicación instalamos los paquetes iniciales mediante el comando: `meteor add <PackageName>`. La lista de paquetes iniciales es la siguiente:

- **accounts-base:** paquete base para cuentas de usuario.
- **accounts-password:** contraseña como servicio de registro de usuarios.
- **fontawesome:fontawesome:** biblioteca de iconos.
- **fourseven:scss:** preprocesador Sass para estilos.
- **iron:router:** paquete para enrutamiento.

- **mizzao:bootstrap-3** maquetación.
- **reywood:publish-composite**: publicaciones avanzadas y compuestas.

3.3.2. Mixins

SASS nos permite crear reglas dinámicas de estilos que poder incluir en cualquier clase llamados mixins. Para este proyecto hemos utilizado este concepto para la labor de cross-browsing. Esta labor se basa en la traducción de una misma regla a los distintos navegadores Web, ya que cada navegador interpreta algunas reglas de forma distinta. Por lo que estos mixins nos permiten unificar diferentes reglas que se interpretan de manera distinta dependiendo del navegador. Un ejemplo de mixin es el siguiente:

```
1 @mixin border-radius($radius){  
2   -webkit-border-radius: $radius;  #safari, chrome  
3   -moz-border-radius: $radius; #mozilla firefox  
4   -ms-border-radius: $radius; #Internet Explorer  
5   border-radius: $radius; #new  
6 }
```

3.1: Mixin border-radius

Creamos un fichero con el nombre de `_mixins.scss` dentro de la carpeta `client` al nivel del fichero `index.html`. El carácter `_` indica al preprocesador que esta hoja de estilos no la debe procesar. El procesado lo realizará en el momento que la importe-mos a otra hoja de estilos e incluyamos algún mixin. Los mixins más utilizados en este proyecto serán los siguientes: `border-radius`, `flexbox` (`flex`, `flex-direction`, `flex-wrap`, etc), `opacity`, `transition`, `animation` y `gradient`. El archivo está disponible en el repositorio de GitHub.

3.4. Prototipo 1: Registro de usuarios y layout principal

Este es el primer prototipo de la aplicación y se corresponde con el layout principal de la aplicación y con el registro de usuarios.

3.4.1. Layout principal

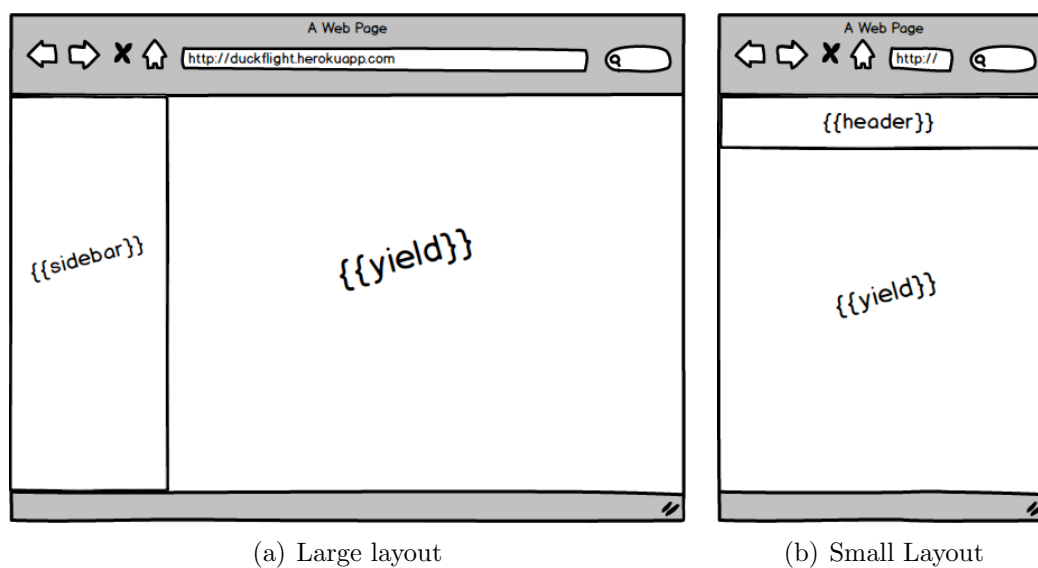


Figura 3.2: Diseño de Layout Full-responsive

Iron Router establece que el diseño de la aplicación debe realizarse en torno a dos plantillas: `{{yield}}` y `{{layout}}`. El layout es la plantilla genérica y el yield es el contenido. Como puede verse en la figura 3.2, ésta va a ser la estructura genérica de nuestra aplicación:

- **Layout:** estará compuesta por un sidebar, un header y la plantilla yield.
- **Yield:** esta plantilla es dinámica y se podrá asignar una plantilla u otra dependiendo de la ruta en la que nos encontremos.

Partiendo del requisito de que los usuarios deben estar autenticados para acceder a la funcionalidad de la aplicación es necesario diseñar el flujo de registro de usuario y situarlo en un recurso o ruta. Debido a esta restricción el flujo se situará en el recurso raíz (/). Por lo que, dependiendo de si el usuario está autenticado o no al acceder a esta ruta deberá mostrarse una plantilla u otra. En este caso:

- Si el usuario está autenticado el layout estará compuesto por la plantilla `{{<sidebar}}`, `{{<header}}` y la plantilla `{{<yield}}` que corresponderá a la plantilla `{{<startPage}}`.
- Si el usuario no está autenticado el layout estará compuesto solamente por la plantilla `{{<yield}}` que corresponderá a la plantilla `{{<mainPage}}`.

El siguiente código muestra el fichero HTML de la plantilla `{{layout}}`:

```
1 <template name="layout">
2   <div id="main-wrapper">
3     {{#if currentUser}}
4       {{> sidebar }}
5       <div id="page-wrapper">
6         {{> header}}
7         {{> yield }}
8       </div>
9     {{else}}
10      {{> yield}}
11      {{> loginModal }}
12    {{/if}}
13  </div>
14 </template>
```

El ayudante (helper) `currentUser` que proporciona Meteor proporciona una función cuyo valor de retorno es un objeto javascript si el usuario está autentica-

do o null si no lo está. Por lo que gracias a Spacebars y sus flujos de control (`{{if}}, {{else}} {{/if}}`) podemos realizar este diseño de forma sencilla.

Sidebar y header

Uno de los requisitos de la aplicación es que debe ser full-responsive. Esto es que se adapte a cualquier tamaño de pantalla. Por lo que es necesario un diseño adaptativo para cada pantalla de la aplicación en la que el layout no es excepción. Como se muestra en la figura 3.2, la plantilla `{{<sidebar}}` se ocultará para pantallas estrechas y aparecerá la plantilla `{{<header}}`. Ésta constará de una serie de botones que al hacer click en cada uno de ellos hará que se se muestre el sidebar con el contenido correspondiente. El diseño del header puede verse siguiente figura:

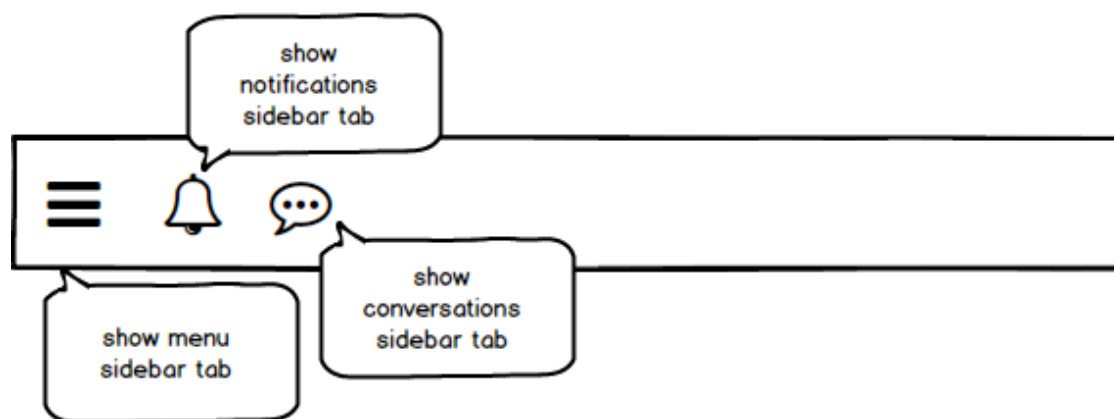


Figura 3.3: Diseño header

El sidebar está compuesta por tres espacios diferenciados (figura 3.4.1):

- **Caja Principal:** en ella aparecerá el logo de la aplicación y el nombre que serán enlaces al recurso raíz (/).
- **Contenido:** el contenido del sidebar se organiza mediante un menú de tabs.

- **Caja de usuario:** en ella aparecerá el avatar y el nombre de usuario que serán enlaces al recurso perfil y un botón para cerrar sesión.

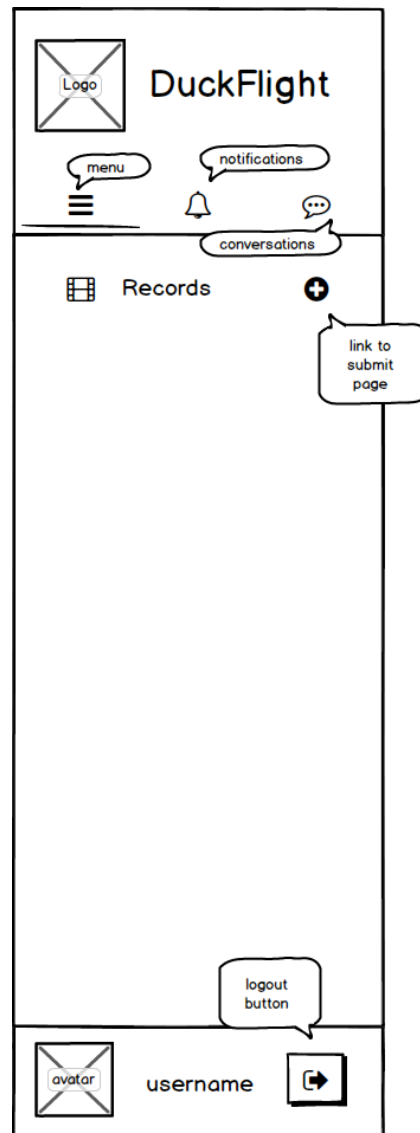


Figura 3.4: Diseño sidebar

Configuración de Iron Router

Iron Router permite establecer una configuración genérica para todas las plantillas especificando la plantilla de carga, el layout, plantilla notFound y subscripciones a las colecciones que necesitemos tener accesibles en todo momento. En este prototipo establecemos solamente la plantilla layout.

Puesto que acabamos de hablar del primer recurso de la aplicación debemos establecer una ruta para el mismo especificando qué plantilla ha de mostrarse de la siguiente manera:

```
1 Router.configure({  
2   layoutTemplate: 'layout'  
3 });  
4  
5 Router.route('/',  
6   name: 'mainPage'  
7 );
```

3.4.2. Registro de usuarios

Para el registro de usuarios creamos una plantilla llamada `{{loginModal}}` que será un Modal de bootstrap y que mediante una variable de sesión de Meteor mostrará un formulario para que los usuarios puedan registrarse u otro para que puedan iniciar sesión. Esta variable de sesión podría ser global y podría ser utilizada para crear un formulario dinámico que dependiera del valor de dicha sesión. Por lo que creamos una plantilla genérica para formularios y después incluimos el que correspondiera según el valor de la variable como sigue:

```
1 <template name=loginModal>
```

```
2 <!-- bootstrap modal-->
3   {{>formAwesome}}
4 <!-- end bootstrap modal-->
5 </template>
6 <template name='formAwesome'>
7   {{Template.dynamic template=formTemplate}}
8 </template>
9
10 <template name='signInForm'>
11   <button></button>
12 </template>
13 <template name='signUpForm'>
14   <button></button>
15 </template>
```

```
1 //Cuando el modal se renderiza en el DOM, se establece el
   valor por defecto
2 //que es que se muestre el formulario para iniciar sesión.
3 Template.loginModal.rendered = function(){
4   Session.set('formTemplate','signInForm');
5 };
6 Template.formAwesome.helpers({
7   //El helper template de la plantilla Template.dynamic
       hace que dicha
8   //plantilla se sustituya por la indicada en dicho helper
9   formTemplate: function(){return Session.get('
       formTemplate')}}
10 });
11 Template.signInForm.events({
```

```
12 //cambiamos al formulario de registro.
13 'click button': function(){Session.set('formTemplate','
    signUpForm')}}
14 });
15
16 Template.signUpForm.events({
17 //cambiamos al formulario de inicio de sesión.
18 'click button': function(){Session.set('formTemplate','
    signInForm')}}
19 });
```

Formulario de inicio de sesión

Atendiendo a los requisitos los usuarios para iniciar sesión deberán rellenar un formulario con dos campos (figura 3.5(a)):

- **usuario:** nombre de usuario o email.
- **contraseña:** contraseña del usuario.

El inicio de sesión en Meteor se realiza mediante una llamada desde el cliente a la función `loginWithPassword` proporcionada por el paquete `accounts-password` pasando como argumento el nombre de usuario o email y la contraseña.

Formulario de registro

Atendiendo a los requisitos los usuarios para registrarse deberán suministrar un nombre de usuario, una contraseña y opcionalmente un email (figura 3.5(b)). El email servirá para la verificación del usuario y para acciones y gestiones que exploraremos más adelante. El registro de usuarios en meteor se realiza mediante una llamada en el servidor a la función `createUser` que proporciona `accounts-base`.

Diagram (a) illustrates the Sign In or Sign Up form. It features a header with a logo placeholder and the title "Sign In or Sign Up". Below the header, there are two input fields: "username or email" (with a person icon) and "password" (with a key icon). A "sign in button" (with a right arrow icon) is labeled "Sign In". To the right of the button is a "link to recover password page". Below the button, there are two links: "forgot password" (with a key icon) and "+ create account". A "link to sign Up form" is located at the bottom right.

(a) Formulario de inicio de sesión

Diagram (b) illustrates the Sign In or Sign Up form. It features a header with a logo placeholder and the title "Sign In or Sign Up". Below the header, there are four input fields: "username" (with a person icon), "email" (with an envelope icon), "password" (with a key icon), and "repeat password" (with a key icon). A "sign up button" (with a plus icon) is labeled "Sign Up". To the left of the button is a "link to sign In Form" (with a left arrow icon). Below the button, there is a "back" link (with a double left arrow icon).

(b) Formulario de registro

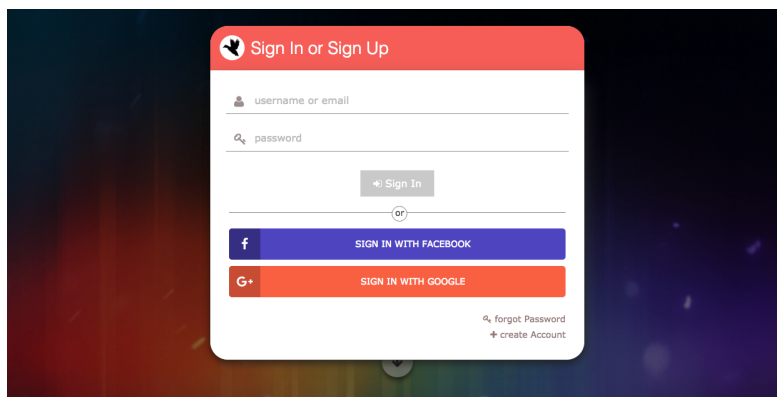
Figura 3.5: Diseño formularios de registro

Puesto que el evento asociado al click del botón se encuentra en el cliente se necesita establecer un method en el servidor que sirva de enlace para el cliente. El cliente llamará al method y éste se ejecutará en el servidor llamando a la función `createUser`. De forma orientativa se muestra el siguiente código:

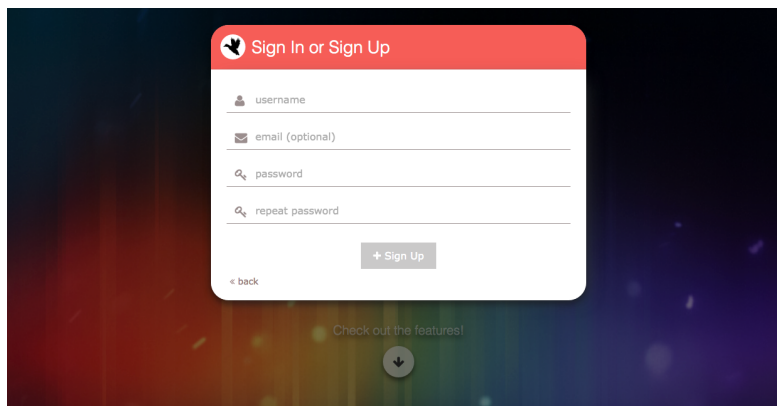
```
1  if (Meteor.isClient){
2    Template.signUpForm.events({
3      'click button': function(e){
4        var paramsUser; //extraemos los valores de los
5          campos.
6        Meteor.call('signUpMethod',paramsUser,function(err,
7          res){
8          if (err) throw new Meteor.error('Error al crear el
9            usuario');
10         if (res) console.log(usuario creado con id: res);
11       });
12     });
13   };
14
15   if(Meteor.isServer){
16     Meteor.methods({
17       'signUpMethod': function(paramsUser){
18         if (userIsValid(paramsUser)){
19           return Accounts.createUser(paramsUser,callback)
20         }else
21           return false;
22       }
23     })
24   };
25 }
```

En el momento que el cliente realiza la llamada con los datos suministrados por el usuario, se deberá verificar que los datos son correctos y que cumplen una serie de reglas. (Validación de nuevo usuario). Una vez verificado se creará un objeto usuario en la colección disponible en Meteor.users cuya estructura es la mostrada en la figura A.11. En este momento ya tenemos definida la primera entidad de la aplicación (Usuarios).

El resultado de este prototipo puede verse a continuación:

The image shows a mobile app interface for a login or sign-up screen. At the top, there's a red header with a white bird icon and the text "Sign In or Sign Up". Below the header, there are two input fields: "username or email" and "password". A "Sign In" button is positioned below the password field. A horizontal line with a small "or" in a circle separates the login section from the social login section. Below this, there are two buttons: "SIGN IN WITH FACEBOOK" (blue with a white 'f' icon) and "SIGN IN WITH GOOGLE" (orange with a white 'G+' icon). At the bottom right, there are two links: "forgot Password" and "create Account".

(a) Formulario de inicio de sesión

The image shows a mobile app interface for a registration screen. At the top, there's a red header with a white bird icon and the text "Sign In or Sign Up". Below the header, there are four input fields: "username", "email (optional)", "password", and "repeat password". A "Sign Up" button is positioned below the "repeat password" field. A "back" button is located at the bottom left. At the bottom center, there is a text prompt "Check out the features!" with a downward-pointing arrow icon.

(b) Formulario de registro

Figura 3.6: Formularios de registro en Página de inicio

3.5. Prototipo 2: Grabador y reproductor basados en documentos

Este prototipo engloba el diseño y desarrollo del grabador, del reproductor y del recurso en el que se muestran las grabaciones realizadas.

3.5.1. Grabador

Entidades y colecciones

En este momento surgen dos nuevas entidades de la aplicación que son las grabaciones y los documentos. La relación que existe entre éstas y los usuarios viene determinada por el diagrama mostrado en la figura (TAL). Dicho diagrama impone que cualquier usuario puede crear una grabación y que ésta debe estar formada por uno o más documentos. Los documentos no existen independientemente de las grabaciones. Como todas las entidades, éstas se traducen en las colecciones Records y Documents cuyos objetos MongoDB se muestran en A.1 y A.2 respectivamente. Al contrario que la colección Users de Meteor, éstas no se crean por defecto. Por lo que, generamos dos nuevos ficheros javascript: app/lib/collections/records.js y app/lib/collections/documents.js. En cada fichero definimos la colección de la siguiente manera:

```
1   CollectionName = new Mongo.Collection('collectionName');
2   //CollectionName será el nombre de la colección
   accesible en la aplicación.
3   //collectionName será el nombre de la colección en
   MongoDB.
```

Routing

Es necesario crear una nueva ruta para el grabador. Esta ruta será `/records/-submit` y estará accesible en todo momento cumpliendo los requisitos gracias a un nuevo diseño del sidebar que incluye un link a la lista de grabaciones y otro al recurso que acabamos de crear como se muestra en la figura ??.

Incluimos una nueva ruta en `/app/lib/router.js`:

```
1 Router.route('/records/submit',{
2   name: 'recordSubmit'
3 });
```

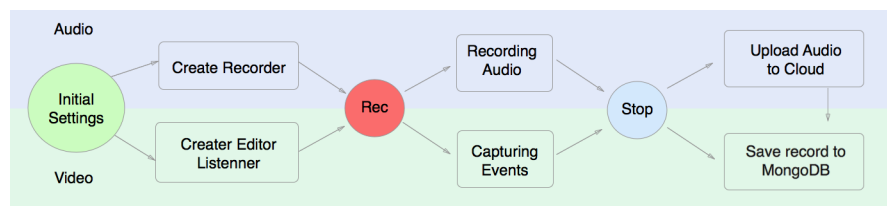
`{{>recordSubmit}}` será la plantilla para nuestro recurso.

Proceso de grabación

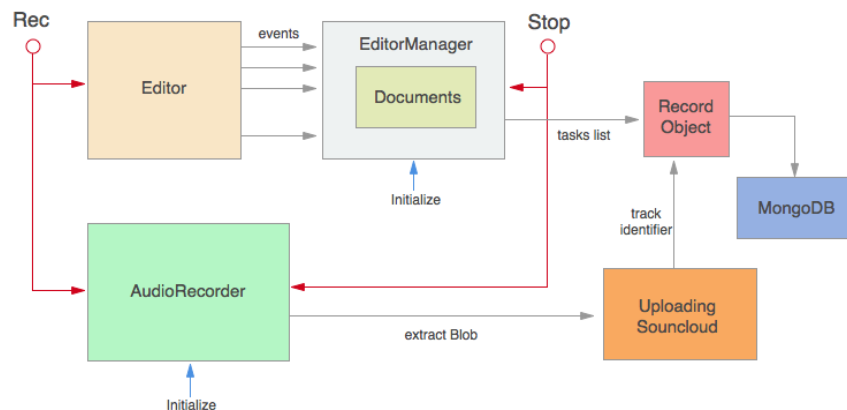
Puesto que cada grabación tendrá dos componentes (audio y video) el proceso se realiza de forma diferente para cada una como se muestra en la figura 3.7(a) de forma esquemática.

Dicho esquema se traduce en la creación de dos objetos javascript: `AudioRecorder` y `EditorManager`. Apreciable en la figura 3.7(b)

- **AudioRecorder:** se trata de un constructor que nos permitirá crear un grabador de audio
- **EditorManager:** se trata de un constructor que nos permitirá crear un manejador de documentos, almacenar y actualizarlos de manera dinámica mientras se producen cambios en el editor.



(a) Esquema



(b) Lógica

Figura 3.7: Proceso de grabación

La API de RTCRecorder nos proporciona un grabador proporcionándole como parámetros un el Stream del usuario y una serie de configuraciones. Para conseguir el Stream utilizamos el módulo navigator del navegador:

```

1  navigator.getUserMedia = navigator.getUserMedia ||
2      navigator.webkitGetUserMedia ||
3      navigator.mozGetUserMedia ||
4      navigator.msGetUserMedia;
5  var audioConstraints = {audio: true, video: false};
6  navigator.getUserMedia(audioConstraints, function(stream)
7      {
8          var settings = {}
9          var recorder = window.RecordRTC(stream, settings);

```

```
9     recorder.startRecording();
10 },function(error){
11     throw new Meteor.error(error.reason);
12 });
```

El código anterior se encuentra dentro de un método (`startRecording`) del constructor `AudioRecorder` y la variable `recorder` se encuentra accesible por todos los métodos del mismo por lo que tenemos accesible el grabador proporcionado por `RTCRecorder`.

El proceso de grabación sobre el editor se basa en generar una lista de acciones indexadas por marcas de tiempo. Dichas marcas de tiempo corresponderán al instante en el que se produce un cambio en el editor y dicho cambio se traducirá en una acción (método proporcionado por la API de `AceEditor`) para que en ese instante durante el proceso de reproducción se ejecute dicha acción simulando el cambio que se haya producido durante la grabación.

Para lo anterior es necesario capturar los cambios del editor, a través de la API de `AceEditor`, y extraer el instante correspondiente. En el momento que comienza la grabación se ejecuta un callback pasado por parámetro que sirve para arrancar el reloj de la grabación. Dicho reloj será accesible dentro de la plantilla `{{>recordSubmit}}`. Por lo que la sincronización con los cambios en el editor será perfecta.

El siguiente código sirve de ejemplo para ilustrar la captura de eventos:

```
1 var editor = ace.editor('#id');
2 editor.getSession().on('change', function(e) {
3     switch (e.action) {
4         case "remove":
```

```
5      var rmRange = {start: e.start, end: e.end};  
6      docsManagerRecorder.insertFunctions([  
7          time: new Date() - date,  
8          arg: rmRange,  
9          toDo: 'editor.getSession().getDocument().  
              remove(arg);'  
10     ]]);  
11     break;  
12 }  
13 });
```

Como puede verse en el código anterior docsManagerRecorder es nuestro objeto construido mediante EditorManager y mediante su método insertFunctions generamos nuevas acciones que se simularán durante la reproducción en el instante almacenado en su atributo time.

Puesto que una grabación debe estar formada por uno o más documentos, durante la grabación puede producirse la creación de nuevos documentos y del cambio de uno a otro sobre el editor. Estos cambios deben ser visibles durante la reproducción por lo tanto deben ser capturados. La captura se realiza mediante los eventos que se produzcan en la plantilla correspondiente al crearse un nuevo documento o visualizar otro distinto al actual. En el momento que alguno de estos eventos se produzca se generará una nueva acción que insertar en la lista de acciones mediante el método insertFunctions.

Los eventos posibles que se traducen en acciones son los siguientes:

- Borrar: se borra contenido.

- Insertar: se inserta contenido.
- Selección: cambia la selección del cursor.
- Cursor: cambia la posición del cursor.
- Scroll: cambia la altura del scroll.
- Creación documento: el usuario crea un nuevo documento.
- Cambio de documento: el usuario selecciona otro documento distinto al actual para su visualización.

Almacenamiento

Como podemos observar en la figura 3.7, una vez termina la grabación se deben proporcionar herramientas para su almacenamiento persistente. El almacenamiento de dicha grabación se realizará por separado según sus componentes. Un objeto record compuesto por información sobre la grabación y la lista de acciones (A.1) se almacenará en la colección Records de MongoDB y el archivo de audio en SoundCloud.

Para utilizar la API de SoundCloud es necesaria la instalación de un nuevo paquete: `monbro:soundcloud-nodejs-api-wrapper`. Este paquete es un wrapper (envoltorio) del SDK (Software Development Kit) de SoundCloud que permite realizar llamadas REST a la API desde el lado del servidor gracias al objeto SoundCloud que nos proporciona de manera global.

Para inicializar dicho objeto necesitaremos un usuario en SoundCloud y crear una aplicación en su espacio para desarrolladores. Dicha aplicación nos proporcionará una serie de credenciales que nos permitirán comunicarnos con la API: identificador de la aplicación, clave secreta de la aplicación. Al crearla necesitaremos

proporcionarle una url de redirección para la autenticación mediante el protocolo OAuth.

Usando solamente el SDK en el cliente, a la hora de conectar con la API, comenzaría un proceso de autenticación que mostraría un popup que exige interacción con el usuario. Dicho proceso no es transparente al usuario y eso es algo que hemos querido arreglar. La solución es incorporar como parámetro de inicialización del SDK un token OAuth (el devuelto tras el proceso de autenticación). El problema está en que dicho token tiene una fecha de expiración y el SDK para Javascript no proporciona herramientas para refrescar el token ya que no tiene método para realizar esa petición REST en cuestión. Pero el paquete mencionado anteriormente sí tiene esa funcionalidad, y es que cada vez que llamamos a su método `.getClient()`, nos devuelve un cliente con un token completamente nuevo.

Por este motivo creamos el fichero `/app/server/soundcloud.js` en el que inicializar el objeto `SoundCloud` y crear el method `.getClientSC` al que podemos llamar desde el cliente y que nos devuelve los parámetros necesarios para inicializar el SDK del cliente de forma que se conecte a la API de `SoundCloud` de forma transparente al usuario.

El siguiente código ilustra este proceso:

```
1  if (Meteor.isServer()){
2    Soudcloud.setConfig({
3      client_id: CLIENT_ID,
4      client_secret: CLIENT_SECRET,
5      username: USERNAME,
6      password: PASSWORD
7    });
```

```
8
9 Meteor.methods({
10   getClientSC: function(){
11     var client = Soundcloud.getClient();
12     return {
13       client_id: CLIENT_ID,
14       access_token: client.settings.access_token
15     }
16   }
17 });
18 }
19 if (Meteor.isClient()){
20   $.getScript("https://cdn.WebRTC-Experiment.com/RecordRTC
21     .js",function(){
22     Meteor.call('getClientSC',function(err,res){
23       if (err) throw new Meteor.error(err.reason);
24       if (res){
25         SC.initialize({
26           client_id: res.client_id,
27           oauth_token: res.access_token,
28           scope: 'non-expiring'
29         });
30       }
31     });
32   }
```


Las variables `CLIENT_ID` Y `CLIENT_SECRET` contienen el id de la aplicación que hemos creado en Soundcloud y su clave secreta respectivamente.

Una vez que tenemos inicializado el SDK en el cliente podemos realizar la subida del audio de la siguiente manera:

```
1 var recordMongoObject = {};  
2 SC.connect().then(function(){  
3   SC.upload({  
4     file: recorder.getAudio(), //Blob  
5     title: 'title'  
6   }).then(function(track){  
7     recordMongoObject.track = {  
8       id: track.id,  
9       link: track.uri  
10    }  
11    //llamada al servidor para almacenar el objeto.  
12  })  
13 })
```

En el código anterior se muestra cómo se crea una referencia al archivo subido mediante su identificador. Una vez tenemos el objeto completado realizamos la llamada al method `insertRecord`, creado en el archivo `/app/lib/collections/records.js` para almacenar la grabación en MongoDB.

Interfaz del grabador

La interfaz del grabador está formado por dos espacios: la pantalla en la que se mostrará una plantilla u otra dependiendo de las acciones a realizar y una caja con botones que determinarán dichas acciones. Además, como se graban documentos

sobre editor, constará de una pestaña en la que aparecerá el título del documento que está siendo editado y un botón para acceder a la lista de documentos como se muestra en la figura B.2

El diseño del grabador se compone de las siguientes plantillas:

- **{{<initial>}}**: es la que se muestra inicialmente y consta de un botón para mostrar la lista de documentos.
- **{{<documentList>}}**: se muestra como un panel dentro de la pantalla de la interfaz y contiene un botón para mostrar el formulario de creación de documentos y la lista de documentos creados. Al hacer click en cada uno de ellos lo visualizaremos en la pantalla. Cada miniatura de los documentos posee un enlace de edición que muestra el formulario de creación con los datos del propio documento.
- **{{<documentForm>}}**: es el formulario de edición y creación de los documentos. Se deberá introducir un título, un lenguaje de programación y un tema para el editor (estos dos últimos son opcionales).
- **{{<editor>}}**: plantilla en la que se muestra el editor con el contenido de los documentos.
- **{{<actions>}}**: esta plantilla está presente a lo largo de todo el proceso de grabación y determina las acciones a realizar según el estado del mismo (grabar, parar, guardar/cancelar).
- **{{<final>}}**: al finalizar la grabación se muestra en la pantalla de la interfaz un mensaje.

- **{{<saveForm}}**: es el formulario para guardar nuestra grabación. Se deberá introducir un título y opcionalmente una descripción y una lista de etiquetas mediante un auto-completado de etiquetas que se ha elaborado.
- **{{<upload}}**: al hacer click en guardar en la plantilla anterior se mostrará el progreso de subida del audio y cuando termine un mensaje y un enlace a la página de la grabación (reproductor) para reproducir la grabación.

Tras este análisis hemos encontrado una nueva entidad: etiquetas (Tags) por lo que creamos una nueva colección y un nuevo fichero en `/app/lib/collections` de forma análoga con las anteriores. Establecemos que una grabación puede tener o no etiquetas y que son globales es decir que la misma etiqueta la pueden tener una o varias grabaciones. Esta relación puede verse en el diagrama (tal). Además para tenerlas accesibles desde el formulario debemos crear una publicación a la que se subscribirá el cliente. Como es la primera, creamos el fichero `/app/server/publications.js`. Será el fichero en el que declararemos todas nuestras publicaciones. La subscripción a esta publicación se hará dinámicamente puesto que se trata de un auto-completado.

El flujo de plantillas se muestra en la figura 3.5.1 Las plantillas presentes en la figura anterior corresponden a las figuras: B.3, B.4, B.5, B.6, B.7 y B.8 disponibles en el apéndice B.

Una vez desarrollado el grabador este es el resultado:



Figura 3.8: Grabador

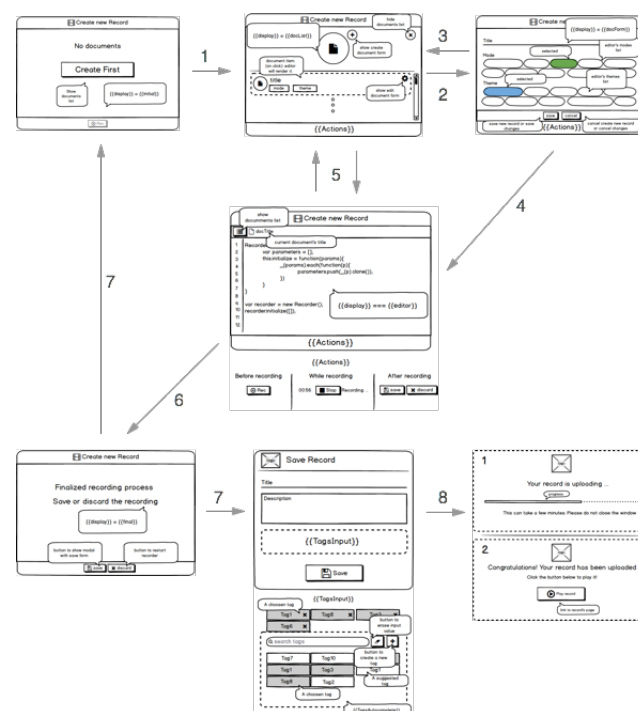


Figura 3.9: Flujo de plantillas del grabador

3.5.2. Reproductor

Ruta, publicaciones y subscripciones

El reproductor es un módulo de la página de cada grabación que se establece en una nueva ruta de nuestro proyecto y la primera que establecemos como detalle (detail). La ruta será `/record/:id` donde `id` corresponderá al identificador del objeto grabación almacenado en MongoDB y estará configurada en `/app/lib/router.js` de la misma forma que las anteriores.

La única diferencia es que ahora necesitaremos tener accesible el objeto grabación para realizar las acciones oportunas. Esto se consigue creando publicaciones y subscripciones.

```
1 // app/server/publications.js
2 Meteor.publishComposite('record',function(id){
3   //Publicamos el record y los documentos asociados al
4     mismo.
5   var sub = {
6     find: function(){
7       return Records.find(id)
8     },
9     children: [{
10       find: function(record){
11         return Documents.find({record_id: record._id});
12       },
13       {...}
14     }];
15   };
```

```
15     return sub;  
16   });
```

Las subscripciones se realizarán mediante Iron Router a no ser que sean dinámicas.

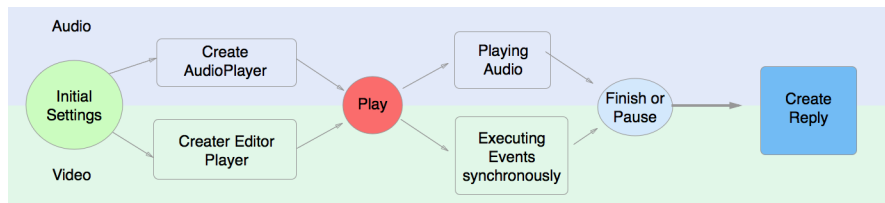
```
1 // app/lib/router.js  
2  
3 Router.route('/record/:_id',{  
4   name: 'record',  
5   data: function(){  
6     return Records.findOne(this.params_id);  
7   },  
8   waitOn: function(){  
9     return Meteor.subscribe('record',this.params._id);  
10  }  
11 });
```

De esta manera tendremos accesibles los datos del record y se establecerán como los datos de la plantilla `{{<record>}}`.

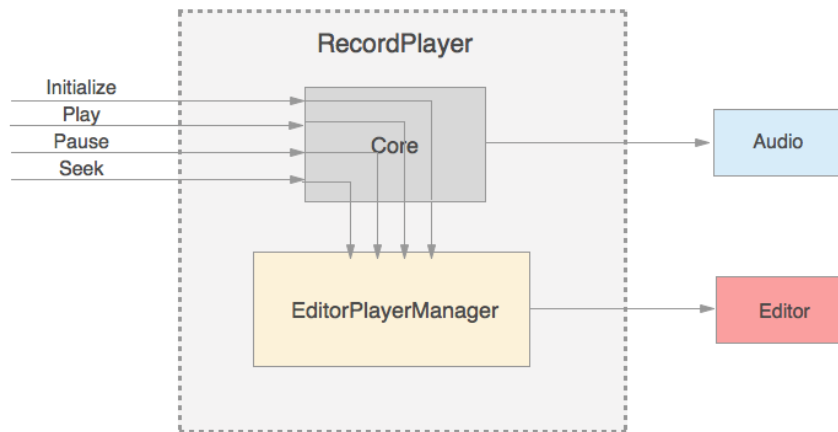
Proceso de reproducción

Debido a que cada grabación está compuesta por audio y por vídeo sobre editor, se desarrollan dos procesos paralelos y sincronizados durante la reproducción de la misma (figura 3.10(a)).

- **Audio:** nos conectamos a SoundCloud y realizamos la petición del stream del audio correspondiente.
- **Vídeo (editor):** a medida que el audio se reproduce, se realiza la simulación de cada evento capturado durante la grabación de forma sincronizada.



(a) Esquema



(b) Lógica

Figura 3.10: Proceso de reproducción

Al igual que durante la grabación, la labor de reproducción de cada una de las componentes recaerá en un objeto Javascript: `RecordPlayer` (audio) y `EditorPlayerManager` (editor) en los ficheros `/app/client/lib/recordPlayer.js` y `/app/client/lib/editorPlayerManager.js` respectivamente (figura 3.10(b)).

- **RecordPlayer:** se encarga de ofrecer una interfaz lógica a partir del stream del audio proporcionado en su inicialización. Dicha interfaz recoge los métodos necesarios para la reproducción (`.play()`, `.pause()`, `.seek()`, `.setVolume()`, `.ended()`) y otros propios (`.updateCover()`, `.getState()`, `.destroy()`).
- **EditorPlayerManager:** se encarga de clasificar la lista de acciones captu-

radas y realizar su simulación. Además se encarga de mantener la integridad de los documentos de la grabación durante el proceso. Este objeto posee los métodos `.getDocs()`, `.getDocActual()`, `.update()` y `.seek()`.

Sincronización entre audio y editor

Al inicializar el objeto `RecordPlayer` toma como argumento un objeto `EditorPlayerManager` ya inicializado con el identificador del editor. El método `.play` del objeto `RecordPlayer` inicia una llamada a su función `.updatePlayer()` mediante un `Interval` de 20. Esto quiere decir que cada 20 milisegundos se ejecutará esta función que, a su vez, realiza una llamada a la función `.update()` del objeto `EditorPlayerManager` pasado como argumento. Por lo que cada 20 milisegundos se mostrarán cambios en el contenido del editor.

Al llamar al método `.pause()` de `RecordPlayer` se destruirá la programación del objeto `Interval`, con lo que parará de inmediato los cambios sobre el editor.

Al saltar entre instantes de la reproducción se llamará al método `.seek()`. Este método realiza una llamada directa al método `.updatePlayer()` y por tanto al método `.update()` de `EditorPlayerManager`, pasando como parámetro el instante exacto.

Simulación de eventos sobre editor

La simulación se realiza en el método `.update()` del objeto `EditorPlayerManager`. El objeto posee la lista de acciones completa, la cual no se alterará en ningún momento. Al inicializarse se realiza una copia en una variable global del objeto. En el momento que se produce la llamada al método `.update()` dicha lista se filtra (se escogen las acciones cuyo instante de creación sea menor o igual que el instante

actual de la reproducción). Después se clasifican estas acciones y se ejecutan en orden. Después se actualiza el valor de la variable donde estaba la copia de las acciones eliminando de esa lista los ya simuladas.

```
1 EditorPlayerManager = function(){
2   var RC, listPending, editor;
3
4   this.initialize = function(params){
5     RC = params.RC;
6     listPending = RC;
7     editor = ace.edit(params.editor);
8   };
9
10  this.update = function(pos){
11    //filtramos las acciones a ejecutar
12    var listToDo = _(listPending).filter(function(action){
13      return action.time <= pos;
14    });
15
16    //ejecutamos las acciones que corresponden.
17    _(listToDo).each(function(action){
18      switch(action.type){
19        //ejecución de las funciones guardadas sobre el
20        editor.
21      }
22    });
23
24    //actualizamos la lista de acciones pendientes.
25    listPending = _(listPending).difference(listToDo);
26  }
```

26 }

El anterior código ilustra el proceso descrito.

Interfaz del recurso grabación

Se trata de un recurso de detalle (detail resource) y en este proyecto se ha hecho un diseño base (figura 3.11) para este tipo de recurso basado en tres espacios:

- Banner: este será el espacio dedicado para la presentación de la información relativa al objeto correspondiente al recurso.
- NavBarTab: barra de navegación basada en tabs para elegir qué contenido, asociado al objeto, visualizar.
- tabContent: espacio en el que se muestran el contenido escogido.

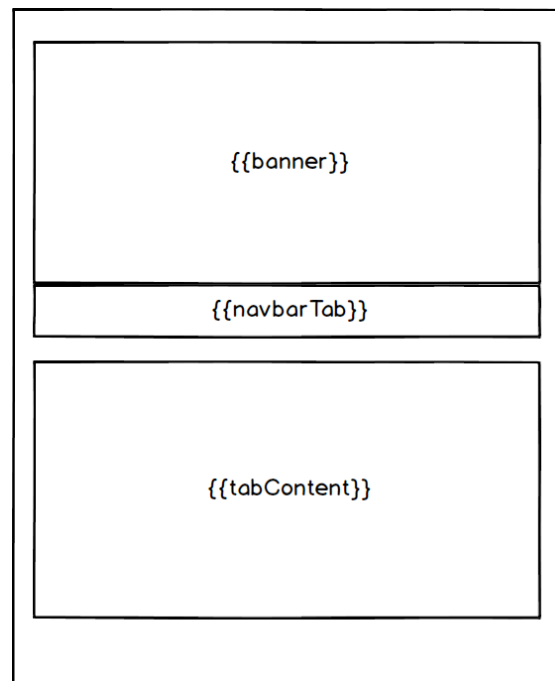


Figura 3.11: Diseño página de detalle

El banner, en este caso, mostrará la información de la grabación (autor, descripción, título, fecha de creación, contadores y lista de etiquetas), un botón para votar, la plantilla `{{<player>}}` para el reproductor y la plantilla `{{<actions>}}` como se muestra en el diseño (figura 3.12).

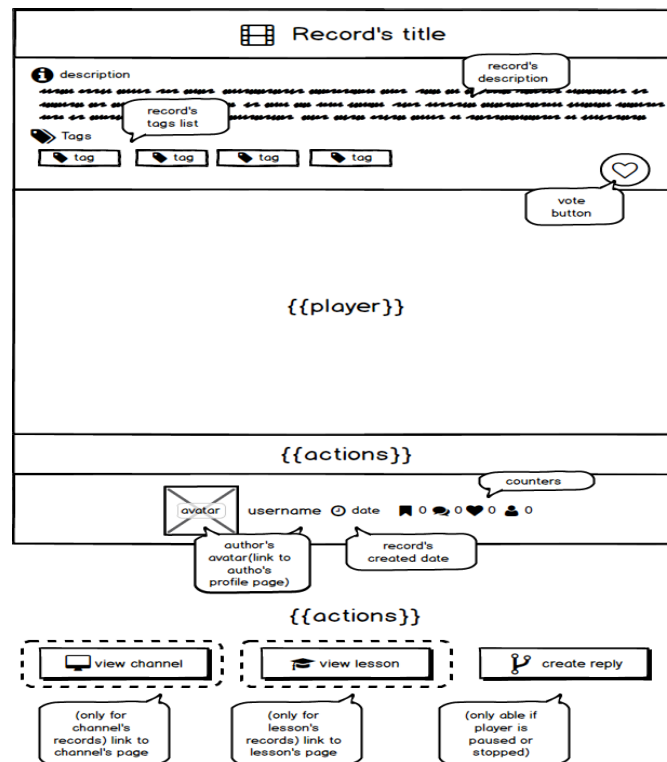


Figura 3.12: Diseño banner para una grabación

Interfaz del reproductor

La interfaz del reproductor (figura 3.13) estará compuesta por el editor, una capa superpuesta vinculada a los eventos play y pause y la plantilla `{{<playerActions>}}` en la que se muestra el progreso, el timer, un controlador de volumen y los botones play y pause cuando correspondan. Además en la parte superior aparecerá una pestaña en la que se mostrará información sobre el documento actual (título y lenguaje).

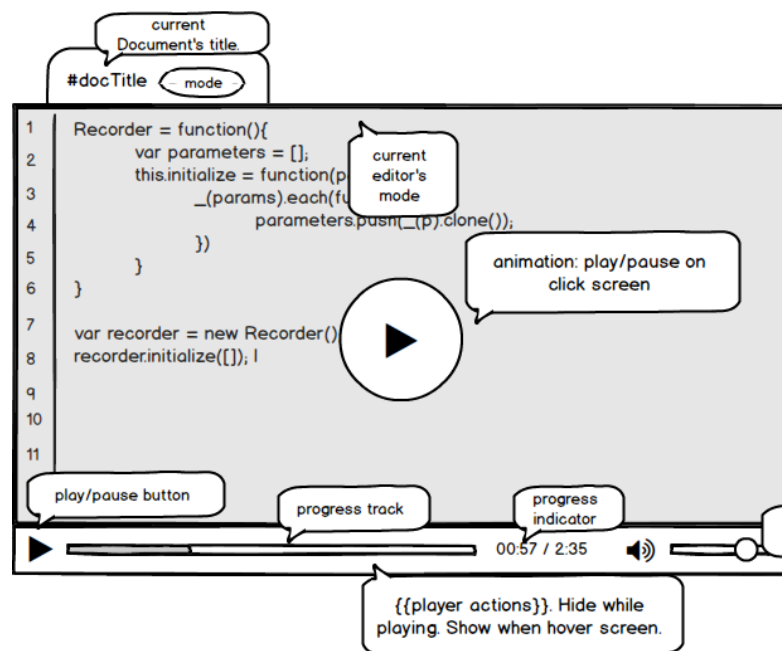


Figura 3.13: Diseño interfaz del reproductor

Streaming

La plantilla `{{<player>}}` se instancia mediante un helper de la plantilla `{{<record>}}` cuyo valor es un objeto Javascript con los datos necesarios para construir e inicializar los objetos `RecordPlayer` y `EditorPlayerManager` y el identificador del audio almacenado en SoundCloud.

Como se muestra a continuación, en el método `.rendered()` es necesaria la conexión con SoundCloud para realizar la petición de un stream de audio que poder suministrar al objeto `RecordPlayer` creado:

```

1 // app/client/modules/record_modules/record/player.js
2
3 Template.player.rendered = function(){
4   var self = this;
5   Meteor.call('getClientSC',function(err,res){

```

```
6      if (res){
7          SC.initialize ({
8              client_id: res.client_id,
9              auth_token: res.auth_token,
10             scope: 'non-expiring'
11         });
12         SC.connect().then(function(){
13             SC.stream('tracks/' + self.track_id)
14                 .then(function(s){
15                     this.recordPlayer.initialize(s,
16                         this.editorPlayerManager, ...);
17                 });
18             });
19     }
20 });
21 }
```

El resultado final del reproductor puede verse en la siguiente figura:

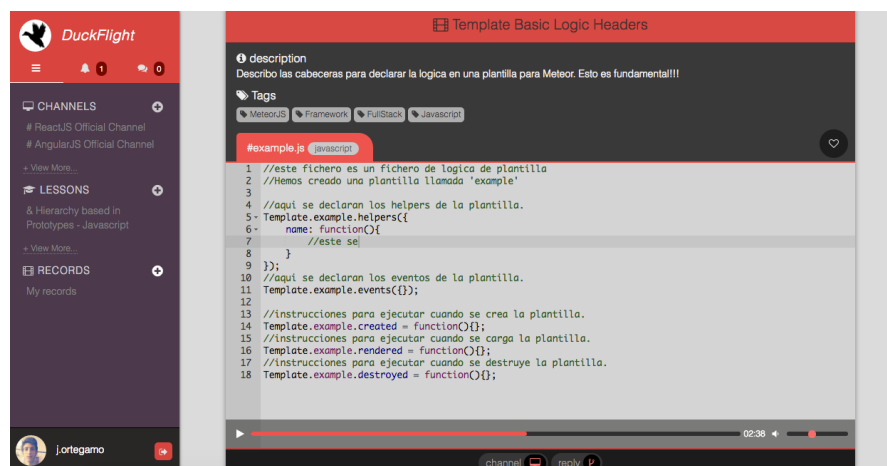


Figura 3.14: Reproductor

3.5.3. Lista de grabaciones

Ruta, publicaciones y subscripciones

La lista de grabaciones se mostrará en una nueva página o recurso de la aplicación. Dicho recurso corresponde a la ruta `/records` y a la plantilla `{{<records>}}`. Como en cada nueva ruta es necesario integrarla en `/app/lib/router.js` y, en este caso, crear las publicaciones y subscripciones necesarias.

```
1 // app/server/publications.js
2
3 Meteor.publishComposite('records',function(){
4   var sub = {
5     find: function(){
6       return Records.find({},{fields: {RC:0, track: 0,
7         tags: 0}});
8     },
9     children: [{
10       find: function(record){
11         return Meteor.users.find(record.author,
12           {fields: {username: 1, avatar: 1}});
13       }
14     }]
15   });
16
17 // app/lib/router.js
18
19 Router.route('/records',{
20   name: 'records',
21   waitOn: function(){
```

```
22     return Meteor.subscribe('records');  
23   }  
24 });
```

De esta manera nos subscribimos a las grabaciones y a los usuarios que las han creado. Filtramos los campos innecesarios para agilizar el proceso de renderizado.

Interfaz

La interfaz es muy sencilla (figura B.9). Está formada por dos espacios:

- **logo:** en este espacio se muestra el logo, el título de la lista y un enlace al recurso creación correspondiente.
- **{{<recordsTabContent>}}**: este espacio será genérico para la aplicación y es que en cualquier parte de la aplicación que se quieran listar grabaciones se utilizará esta plantilla. Está formada por:
 - **{{<contentNavbar>}}**: en ella aparecen una serie de filtros (recientes, populares), opciones de visualización y un tab para iniciar el buscador.
 - **{{<content>}}**: en esta plantilla se listarán las grabaciones mediante miniaturas (figura B.10) según el modo de visualización y aparecerá un botón para cargar más ítems.

Tras el desarrollo de la interfaz el resultado se muestra en la siguiente figura:

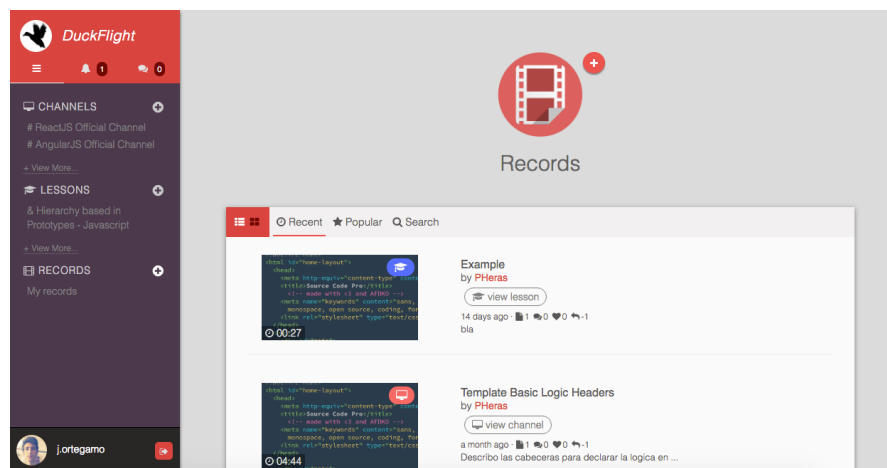


Figura 3.15: Lista de grabaciones

3.6. Prototipo 3: Respuestas a grabaciones

En este prototipo actualizamos la lógica del grabador y añadimos nuevas acciones para la página de detalle de grabación.

3.6.1. Actualización del grabador

Las respuestas a grabaciones se realizan mediante nuevas grabaciones sobre editor según los requisitos de la aplicación. La única diferencia es que esta vez el grabador se debe iniciar con los documentos de la grabación a la que queremos responder. Además el contenido de dichos documentos debe corresponder al instante en el que hemos pausado la reproducción.

Para todo lo anterior necesitamos inicializar, con el estado de dichos documentos, al objeto que se encarga de manejarlos durante la grabación (EditorManager). Pero antes necesitamos extraer dicho estado durante la reproducción. Para esto el objeto recordPlayer cuenta con el método `.getState()` que devuelve un objeto con el último instante de reproducción y la lista de documentos con su estado actual.

Dicho objeto se almacenará en una variable de sesión accesible desde el fichero `/app/lib/router.js`.

Para inicializar el objeto `EditorManager` como se ha descrito, necesitamos tener esos documentos accesibles desde los datos de la plantilla `{{<recordSubmit>}}`. De esto se encarga `Iron Router`. La forma de especificar nuestra intención de realizar una grabación respuesta a `Iron Router` es mediante una query string (cadena de consulta). En ella se especificará el identificador de la grabación a la que queremos responder y la clave será `parent_id`.

```
1 Router.route('/records/submit',{
2   ...
3   data: function(){
4     var data = {}
5     if (this.params.query){ //es una respuesta
6       var playerState = (Session.get(playerState));
7       (playerState)? data.playInstantObject = playerState
8         : null;
9     }
10    return data;
11  },
12  waitOn: function(){
13    if (this.params.query){
14      return Meteor.subscribe('recordDocuments'
15        this.params.query.parent_id);
16    }
17  });
```

En el código anterior podemos ver cómo se configuran los datos de la plantilla del grabador y cómo nos subscribimos a los documentos de la grabación padre.

Hay que tener en cuenta que en el momento que abandonemos la página del grabador, la variable de sesión deberá ser destruida. Esto supone un problema. Si abandonamos la página del grabador y después volvemos a ella, Iron Router interpreta que queremos hacer una grabación respuesta. Esto se debe a que la query string no desaparece. Por este motivo se ha establecido que si volvemos al grabador se comenzará con los documentos en el estado final de la grabación padre. Dichos documentos los tenemos accesibles gracias a la publicación correspondiente y a la subscripción mediante el método `waitOn` de la ruta descrita en el código mostrado.

Como se muestra en el objeto (A.1), cuando almacenemos la respuesta en MongoDB debemos incluir dos campos: `isReply` y `parent_id`.

3.6.2. Nuevas acciones

En la plantilla `{{<actions>}}` de la interfaz de detalle de grabación (figura 3.12) creamos dos botones: uno como enlace al grabador para crear una respuesta (sólo estará disponible si la reproducción se encuentra pausada o finalizada) y otro como enlace a la grabación padre (sólo disponible si se trata de una respuesta).

```
1 <div id='actions'>
2   {{#if isPosibleToReply}}
3     <button id='reply-button'>reply</button>
4   {{/if}}
5   {{#if isReply}}
6     <button id='go-to-parent-button'>parent</button>
7   {{/if}}
```

```
8 </div>

1 Template.record.events({
2   'click button#reply-button': function(){
3     Session.set('playerState',this.recordPlayer.getState()
4       );
5     Router.go('recordSubmit',{},{query: 'parent_id=' +
6       this._id});
7   },
8   'click button#go-to-parent-button': function(){
9     Router.go('record',{_id: this.parent_id});
10  }
11 });
```

En el código anterior podemos apreciar la lógica del proceso descrito anteriormente.

3.6.3. Timeline, relacionados y comentarios

En este prototipo se han generado las secciones de contenido de la página detalle de una grabación mostradas en la figura 3.11: `{{navbarTab}}`(figura tal) y `{{tabContent}}`. La plantilla `{{navbarTab}}` se ha diseñado como componente de manera que pueda ser configurada con las tabs que correspondan para cada página de detalle.

Para listar las respuestas de una grabación se ha creado un timeline (figura tal). Dicho timeline se muestra al seleccionar la tab Replies del `{{<navbar}}` de la página de la grabación.

Además se ha añadido una nueva sección de contenidos para mostrar las grabaciones relacionadas con la grabación actual (figura tal). Para ello se ha modificado la publicación 'record'. Ahora publicará, además de la grabación actual, otras que posean etiquetas similares.

También se ha creado una nueva sección que contiene un espacio para realizar comentarios (figura tal) y con ello surge una nueva entidad. Dicha entidad se traduce en una nueva colección Comentarios cuyo objeto se muestra en A.14 y su relación con las demás entidades en la figura (tal). Esta sección esta compuesta por una caja para introducir el texto a publicar y la lista de comentarios. Cada comentario está compuesto por el avatar del autor, el texto, una lista de las respuestas a ese comentario y la misma caja de texto mencionada anteriormente responder.

Se ha diseñado el espacio para comentarios de manera que sea genérico, es decir, que se pueda utilizar para todo tipo de contenidos. Tras la implementación el resultado puede verse en las figuras (tal),(tal) y (tal).

3.7. Prototipo 4: Organización en Canales

3.8. Prototipo 5: Organización en Lecciones

3.9. Prototipo 6: Página de perfil y Contactos

En este prototipo se ha diseñado e implementado la página de perfil del usuario y un espacio para realizar solicitudes de contacto y mostrar esas relaciones entre los distintos usuarios.

3.9.1. Perfil

Se trata de un nuevo módulo y recurso cuya ruta será `/profile/_id`, donde `_id` corresponderá al id del usuario en cuestión. Se crean nuevas publicaciones y subscripciones y se establece la ruta en `/app/lib/router.js`.

Interfaz

Al tratarse de un recurso de detalle compartirá la misma base que los demás (figura 3.11). El `{{baner}}` estará formado una cabecera en la que se muestran el avatar del usuario, una imagen de fondo, un botón para acceder al recurso de edición del perfil y una caja con acciones y un cuerpo en el que aparece el nombre de usuario, la descripción y lista de servicios (figura tal).

Contenidos

Utilizamos el complemento `navbarTab` y lo configuramos para que tenga las tabs canales, lecciones, grabaciones, conversaciones y contactos que corresponden a las categorías de contenido de la aplicación (figura tal). Podemos reutilizar las plantillas `{{contentTab}}` para mostrar las listas de contenido según la categoría que hemos generado para los recursos `/records`, `/channels` y `/lessons`. Sólo aparecerán las del usuario en cuestión, ya que nos hemos suscrito a sus contenidos. Aunque en este caso no existe la opción de iniciar el buscador, sino que se añaden nuevos filtros (figuras B.9, ?? y ??):

- **Subscrito:** muestra los canales o lecciones a los que se ha suscrito el usuario según corresponda.
- **Historial:** muestra un total de 10 entradas. Dichas entradas serán las 10 últimas reproducciones que el usuario haya visualizado y formarán parte del `{{contentTab}}` para las grabaciones.

Además se han incluido nuevos enlaces en el sidebar para acceder a los contenidos del perfil de forma directa mediante una query string que establece el contenido a visualizarse al cargar el perfil como puede verse en el diseño MenuTabV4 del sidebar (figura B.1)

Roles

Al visualizar cualquier página de perfil un usuario puede adoptar uno de estos dos roles:

- **Propietario:** lo adoptará el usuario que sea propietario de dicho perfil. Podrá acceder a la lista de servicios y al recurso de edición del perfil, además en cada `{{contentTab}}` se le mostrará un enlace al recurso de creación correspondiente.
- **Visitante:** lo adoptará el usuario que no sea propietario de dicho perfil. Podrá acceder a las acciones presentes en la cabecera del banner. Sólo existirá una, de momento: realizar peticiones de contacto.

Edición del perfil

Al igual que para editar las lecciones o los canales se establece un nuevo recurso para editar el perfil cuya ruta es `/profile/:id/edit`. Como en las otras es necesario declararla en `/app/lib/router.js` crear una nueva plantilla `{{<profileEdit}}` en `/app/client/modules/profile_modules/profileEdit/profileEdit.html` y subscribirse al usuario correspondiente. Al igual que los demás recursos de edición, éste mantiene el diseño base (figura tal), es decir, que incorpora la plantilla `{{<awesomeForm}}` la cual, mediante una variable de sesión, carga uno u otro formulario con características comunes. En este caso, el formulario de edición dinámico creado en el prototipo 4. Además incorporamos los componentes avatar, banner y description

(figura tal).

3.9.2. Contactos

Al existir usuarios en la aplicación es necesario establecer y denominar las relaciones entre ellos. Dichas relaciones se denominan relaciones de contacto. Con esto surge una nueva entidad (Contactos) que se traduce en una nueva colección Mongo llamada Relations con su fichero correspondiente y cuyo objeto es A.13

Peticiones y lista de contactos

Surge una nueva entidad Peticiones que se traduce en la colección llamada Requests cuyo objeto es A.12 Para el `{{contentTab}}` del tab 'contactos' del `{{navbarTab}}` de la página de perfil se establecen dos nuevas tabs: contactos y peticiones.

Para la tab contactos activa se mostrará la lista de los contactos (figura tal). Cada miniatura contará con el avatar, el nombre de usuario, la fecha desde la que se inició la relación de contacto, la descripción y acciones (sólo visibles cuando el usuario es el propietario del perfil).

Para la tab peticiones activa se mostrará el espacio para peticiones (figura tal). Dicho espacio esta compuesto por:

- Autocompletado: para buscar los usuarios. Cada resultado dispondrá de un botón para enviar la petición. Este botón se sustituirá por iconos de estado si la petición ya ha sido realizada.
- Bandeja de entrada: se muestran las peticiones recibidas, su estado y acciones relacionadas con su estado.

- Bandeja de salida: se muestran las peticiones enviadas, su estado y acciones relacionadas con su estado.

Las acciones dependiendo del estado y de la bandeja en la que se encuentren se muestran en la tabla 3.1:

	Acciones	
Estado	Recibidas	Enviadas
Pendiente	Aceptar o Rechazar	X
Aceptada	Ok	Ok
Rechazada	X	Ok o Reenviar

Cuadro 3.1: Acciones para las peticiones

El proceso para establecer la relación de contacto es la siguiente:

1. El usuario A envía una solicitud al usuario B y visualiza esta petición como pendiente.
2. El usuario B visualiza la petición como pendiente en su bandeja de entrada y puede aceptarla o rechazarla.
3. El usuario B acepta la petición y la visualiza como aceptada, pulsa Ok. Se crea la relación.
4. El usuario A visualiza la petición como aceptada, pulsa Ok para eliminar la entrada.

Si el usuario B rechaza la petición, el usuario A la visualizaría como rechazada y podría pulsar Ok o reenviarla y comenzar el proceso de nuevo.

Además de realizar las peticiones mediante este espacio, se habilita para los usuarios que tengan el rol de visitante un botón en $\{\{actions\}\}$ como se muestra en la figura (tal).

La implementación de este prototipo tiene como resultado las figuras (tal tal tal en apéndices).

3.10. Prototipo 7: Conversaciones

3.11. Prototipo 8: Emails e integración de servicios de registro

3.12. Prototipo 9: Página principal

3.13. Prototipo 10: Página de inicio, espacio para tutoriales y features

3.14. Prototipo Final: Restricciones de acceso y cross-browsing

3.15. Despliegue y pruebas globales

Capítulo 4

Experimentación

4.1. Motivación

4.2. Planteamiento y objetivos

4.3. Proceso y realización

4.4. Resultados

Capítulo 5

Conclusión

Bibliografía

- [1] Página oficial WebRTC: <https://webrtc.org/>
- [2] Página para WebRTC experiments (recordRTC): <https://www.webrtc-experiment.com/RecordRTC/>
- [3] Página oficial SoundCloud: <https://soundcloud.com>
- [4] Página para desarrolladores SoundCloud: <https://developers.soundcloud.com/>
- [5] Página oficial MeteorJS: <https://www.meteor.com/>
- [6] Guía de MeteorJS: <http://guide.meteor.com/>
- [7] Documentación de MeteorJS: <http://docs.meteor.com/#/full/>
- [8] Página oficial Bootstrap: <http://getbootstrap.com/>
- [9] W3C javascript tutorial: <http://www.w3schools.com/js/>
- [10] Documentación de MongoDB: <https://docs.mongodb.com/manual/>
- [11] Página oficial JQuery: <https://jquery.com/>
- [12] Documentación de JQuery: <http://api.jquery.com/>
- [13] Documentación de UnderscoreJS: <http://underscorejs.org/>

5.1. Principales Sitios Web

5.2. Libros

5.3. Artículos

5.4. Tutoriales

5.5. Paquetes

5.6. Repositorios

Apéndices

Apéndice A

Diseño de documentos para Mongo

```
1 var record = {
2   _id: //idMongo,
3   author: //idUser,
4   title: //no único,
5   description: //(opcional),
6   RC: [{},...], //Funciones de reproducción sobre el
      editor
7   createdAt: //fechaCreación,
8   docs_count: //contador de documentos,
9   votes_count: //contador de votos,
10  replies_count: //contador de respuestas,
11  comments_count: //contador de comentarios,
12  channel_id: //canal al que pertenece,
13  lesson_id: //lección a la que pertenece,
14  section_id: //sección a la que pertenece dentro de una
      lección,
```

```
15     order: ,//orden dentro de la lista de reproducción.
16     tags: [{}],...], //etiquetas,
17     ready: //(boolean) para conocer la disponibilidad del
        record.
18     img: //imagen miniatura,
19     duration: //duración en milisegundos de la grabación.
20     isReply: //(boolean) indica si se trata de una
        respuesta a otro record.
21     parent_id: //idMongo del record al que responde.
22     track: //{_id: 'id del track en SoundCloud', link: '
        link SoundCloud'}
23 };
```

A.1: Diseño de documento para una grabación

```
1  var doc = {
2    _id: //idMongo,
3    record: //record al que pertenecen,
4    doc: {
5      title: //titulo del documento (único para el
        record),
6      theme: //tema del editor tras último cambio,
7      mode: //tema del editor tras el último cambio,
8      value: //último estado del contenido del editor
9    },
10   start: //(boolean) (True)? comienzo grabación : se ha
        creado durante la grabación
11         //o es el estado final de otro inicial.
12 };
```


A.2: Diseño de documento para los documentos de cada grabación

```
1 var channel = {  
2   _id: //idMongo,  
3   author: //id_user,  
4   title: //único,  
5   banner: //url img banner,  
6   img: //url img miniatura,  
7   description: //(opcional),  
8   tags: //etiquetas [{name: //nombre etiqueta}],  
9   createAt: //fechaCreación,  
10  votes_count: //contador para los votos,  
11  records_count: //contador para los records,  
12  comments_count: //contador para los comentarios,  
13  users_count: //contador para los usuarios que se  
    subscriban  
14 };
```

A.3: Diseño de documento para un canal

```
1 var lesson = {  
2   _id: //idMongo,  
3   author: //id_user,  
4   title: //único,  
5   img: //url img miniatura,  
6   description: //(opcional),  
7   tags: //etiquetas [{name: //nombre etiqueta}],  
8   createAt: //fechaCreación,  
9   votes_count: //contador para los votos,
```

```
10     sections_count: //contador para las secciones,  
11     comments_count: //contador para los comentarios,  
12     users_count: //contador para los usuarios que se  
        apunten  
13 };
```

A.4: Diseño de documento para una lección

```
1  var section = {  
2      _id: //idMongo,  
3      title: //único,  
4      createdAt: //fechaCreación,  
5      records_count: //contador para los records,  
6      lesson_id: //id de la lección a la que pertenece.  
7      order: //orden de la sección.  
8  };
```

A.5: Diseño de documento para una sección

```
1  var userEnrolled = {  
2      _id: //idMongo,  
3      contextId: //id del canal o de la lección a la que se  
        han suscrito,  
4      user_id: //id del usuario  
5  };
```

A.6: Diseño de documento para cada suscripción de un usuario

```
1  var tag = {  
2      _id: //idMongo,  
3      name: //nombre para la etiqueta  
4  };
```

A.7: Diseño de documento para una etiqueta

```
1 var conversation = {  
2   _id: //idMongo,  
3   subject: //asunto de la conversación,  
4   author: //líder de la conversación.  
5   last_modified: //fecha de ultima modificación (cada  
6     vez que se inserta un mensaje),  
7   members: //[{},...], array de miembros,  
8   members_count: //contador para los miembros,  
9   messages_count: //contador para los mensajes  
10 };
```

A.8: Diseño de documento para una conversación

```
1 var message = {  
2   _id: //idMongo,  
3   author: //id del creador,  
4   createdAt: //fecha de creación,  
5   message: //contenido del mensaje  
6   conversation_id: //id de la conversación a la que  
7     pertenece.  
8 }
```

A.9: Diseño de documento para los mensajes

```
1 var conversationAlert = {  
2   _id: //idMongo,  
3   user_id: //usuario al que se le muestra la alerta,  
4   conversation_id: //id de la conversación de la que  
5     procede,
```

```
5     alertsAllow: //flag (boolean) si es true se muestran
        las alertas y si es false no.
6     alerts_count: //contador de alertas
7 };
```

A.10: Diseño de documento para una alerta de conversación

```
1 var user = {
2     _id: //idMongo,
3     username: //nombre de usuario (único),
4     avatar: //avatar del usuario,
5     banner: //banner de la pagina de usuario,
6     description: //descripción del usuario,
7     status: //estado de conexión,
8     emails: //[{address: //emailAddress, verified: //
        Boolean,},...],
9     ... //otros campos establecidos por meteor-accounts.
10 };
```

A.11: Diseño de documento para cada usuario

```
1 var request = {
2     _id: //idMongo,
3     requested: //{id: id_user, delete: boolean},
4     applicant: //{id: id_user, delete: boolean},
5     status: //(string) 'pending', 'accepted', 'refused'
6 };
```

A.12: Diseño de documento para las peticiones de contacto

```
1 var relation = {
2     _id: //idMongo,
```

```
3   createdAt: //fecha creación,  
4   users: //[id_user_requested, id_user_applicant],  
5 };
```

A.13: Diseño de documento para establecer la relación de contacto

```
1 var comment = {  
2   _id: //idMongo,  
3   author: //id del creador,  
4   createdAt: //fecha de creación,  
5   isReply: //(Boolean) indica si es una respuesta o no,  
6   replies_count: //contador de respuestas,  
7   message: //contenido del mensaje  
8 };
```

A.14: Diseño de documento para los comentarios

```
1 var notification = {  
2   _id: //idMongo,  
3   to: //id_user destinatario,  
4   from: //id_user origen,  
5   parentContextTitle: //es el título de la lección,  
6   record o channel en el que se ha producido,  
7   urlParameters: //son los parámetros para construir el  
8   enlace al clicar sobre la notificación,  
9   type: //(string) 'channel', 'record', 'comment', etc,  
10  message: //(string) de contenido HTML  
11 };
```

A.15: Diseño de documento para una notificación

Apéndice B

Diseño de Interfaces

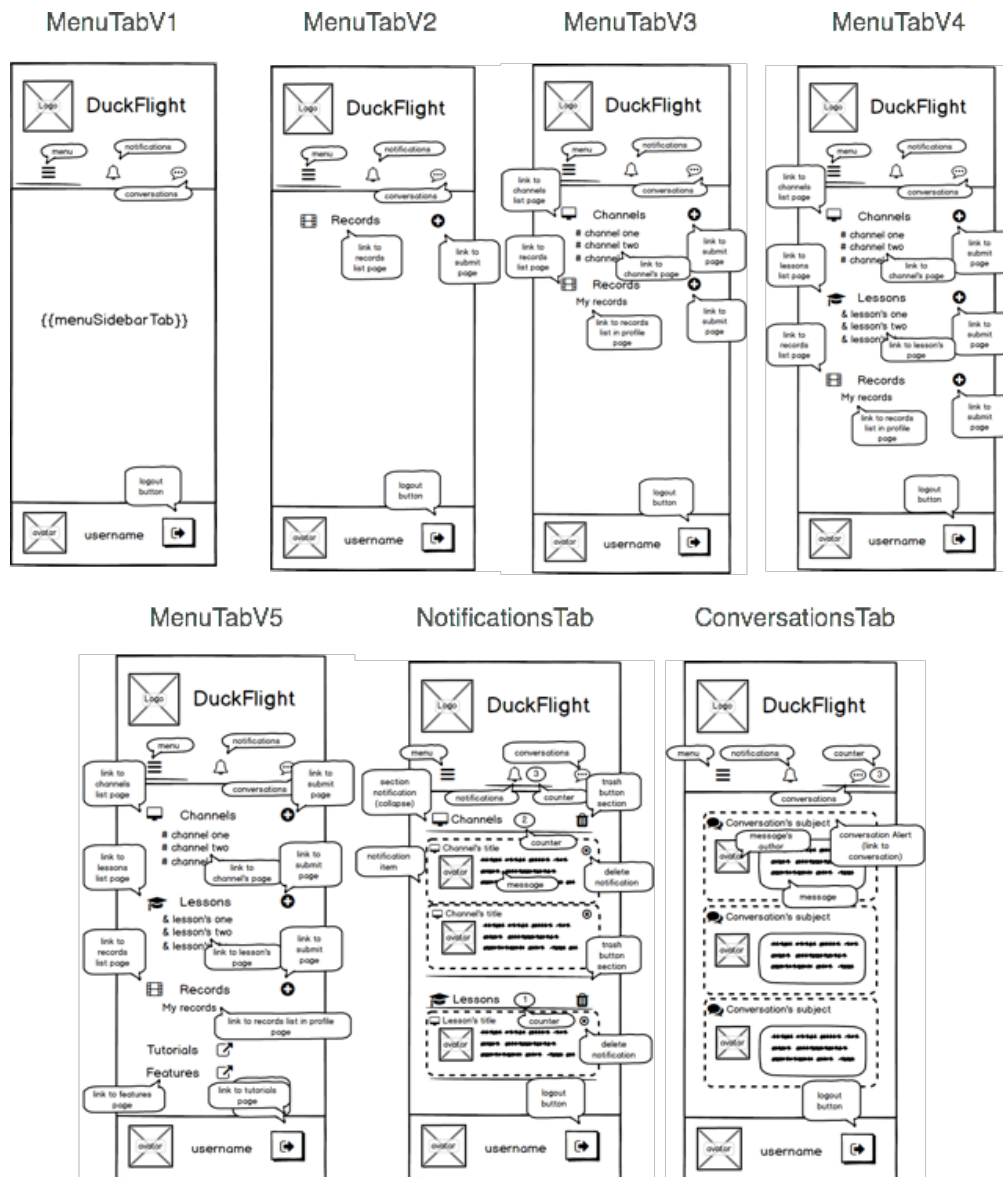


Figura B.1: Diseño sidebar

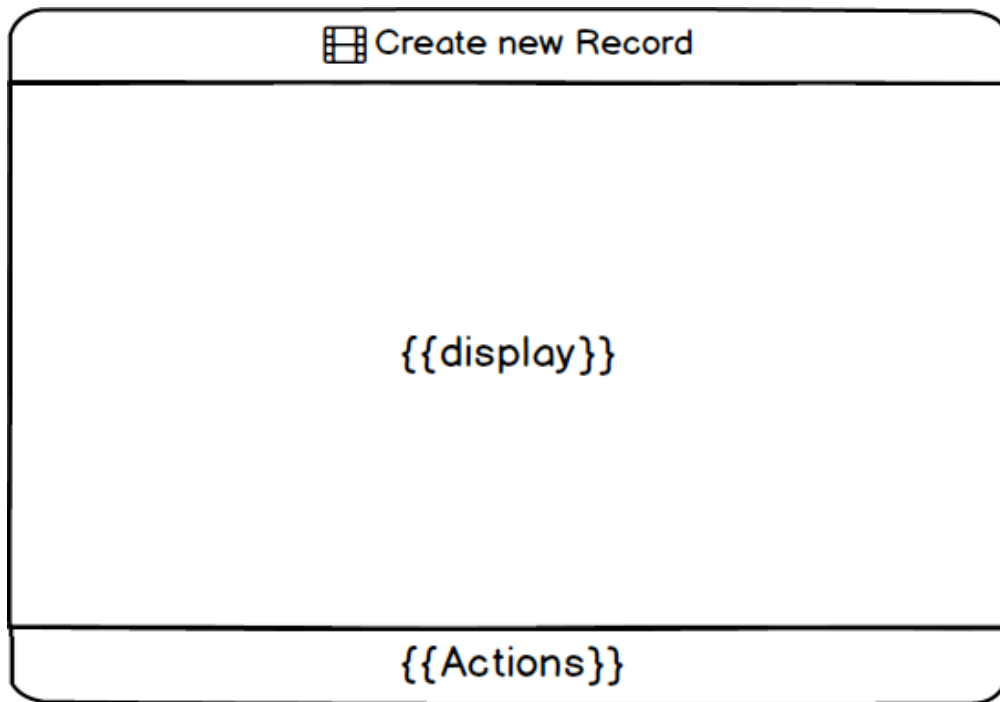


Figura B.2: Diseño base grabador

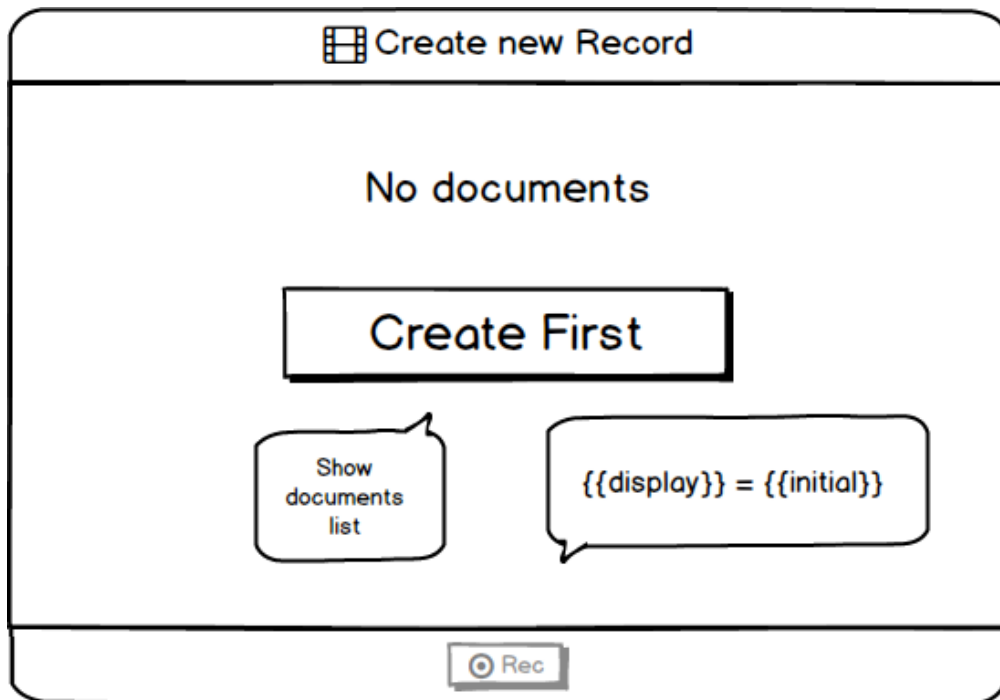


Figura B.3: Diseño panel inicial grabador

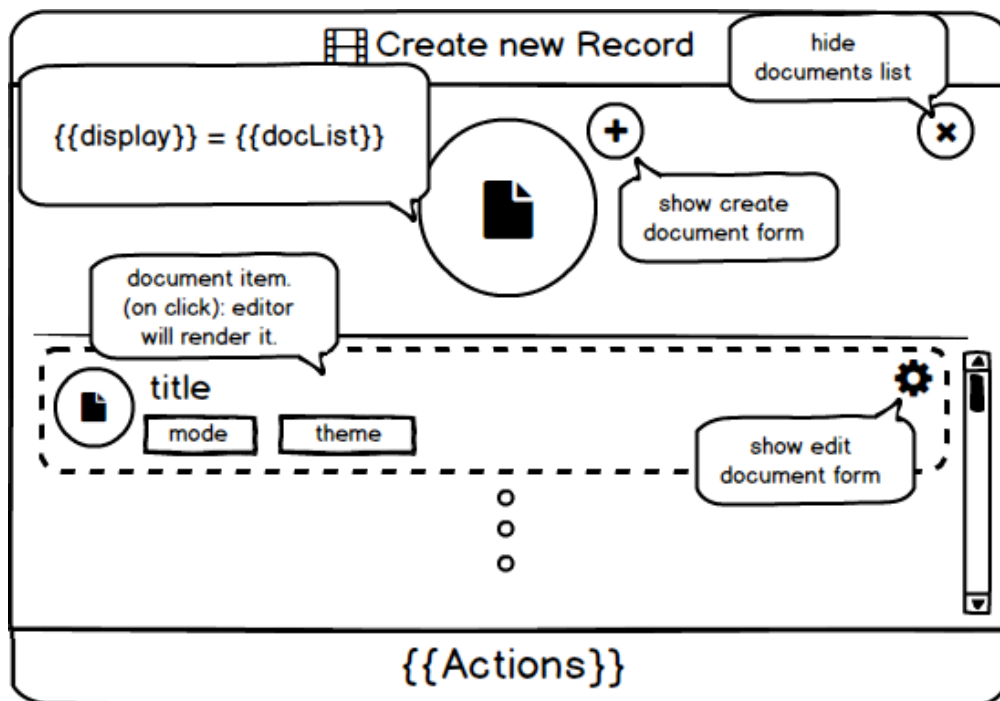


Figura B.4: Diseño panel documentos en grabador

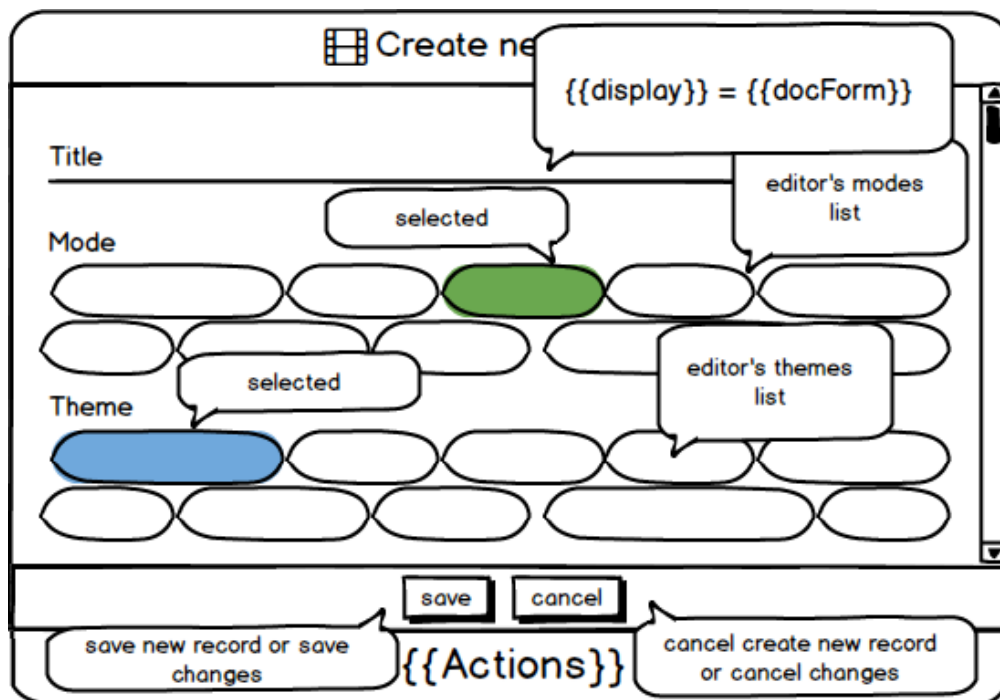


Figura B.5: Diseño formulario documentos

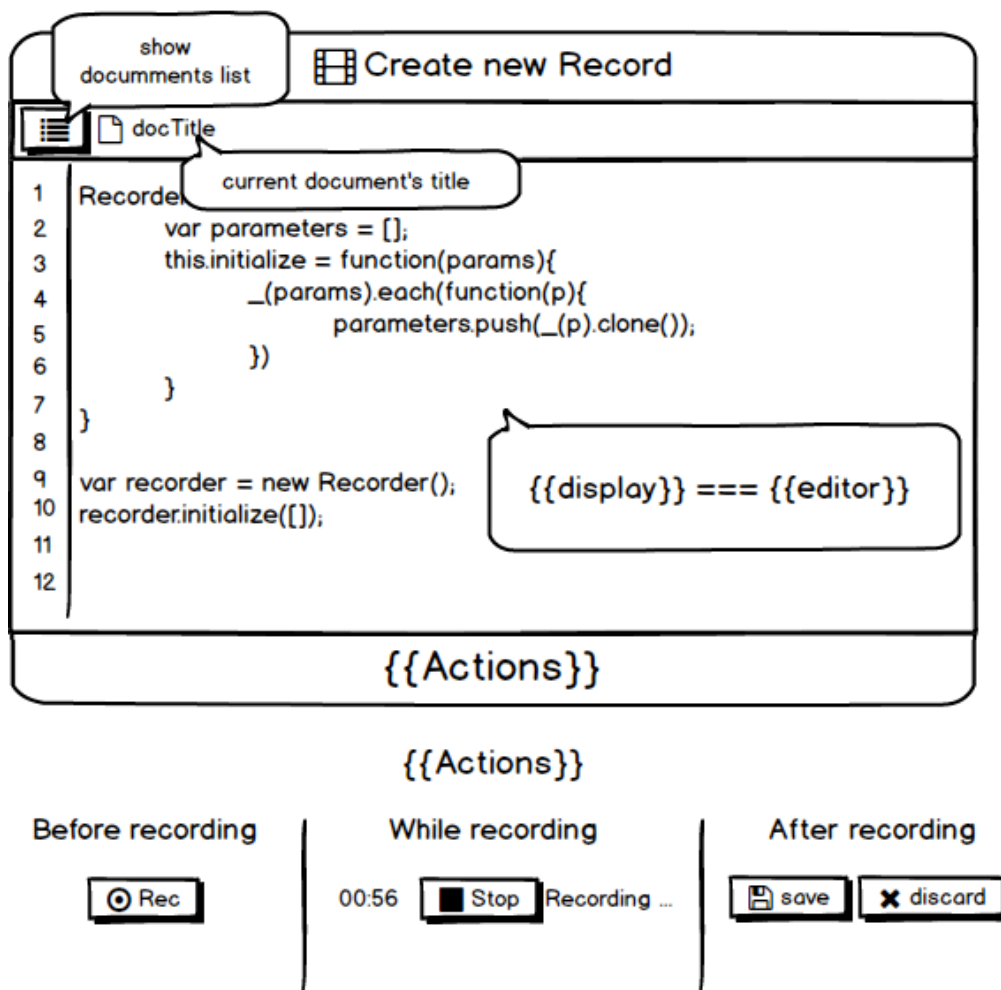


Figura B.6: Diseño editor en grabador

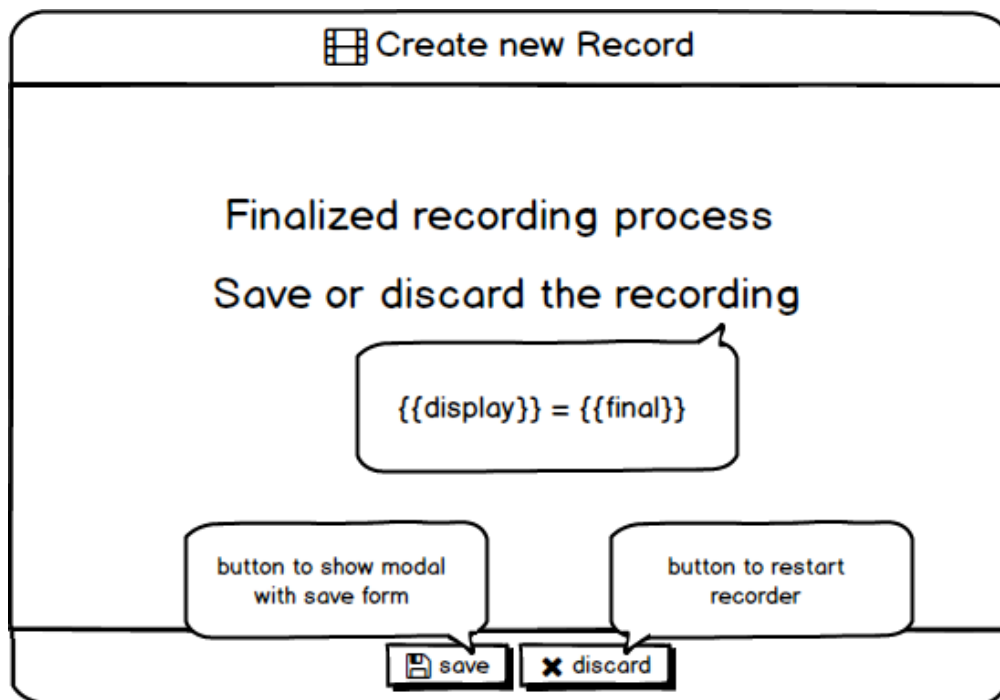


Figura B.7: Diseño final grabación

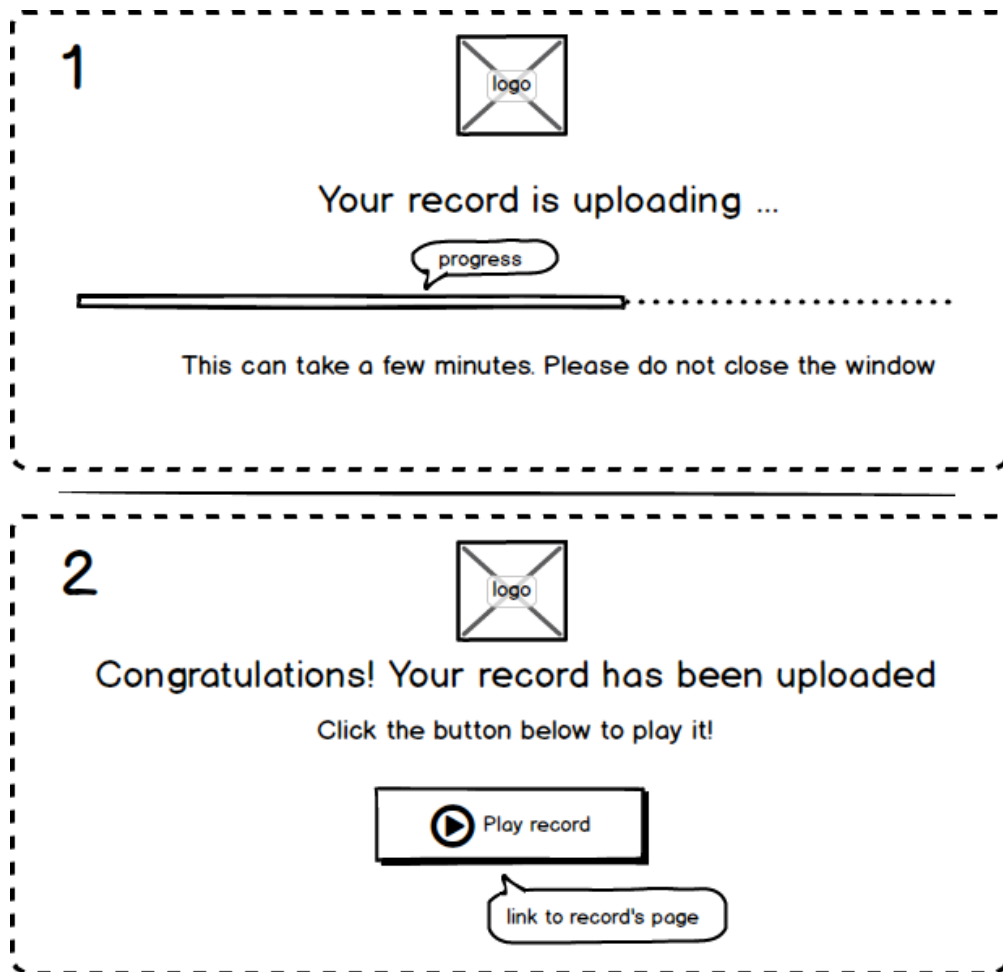


Figura B.8: Diseño proceso de subida

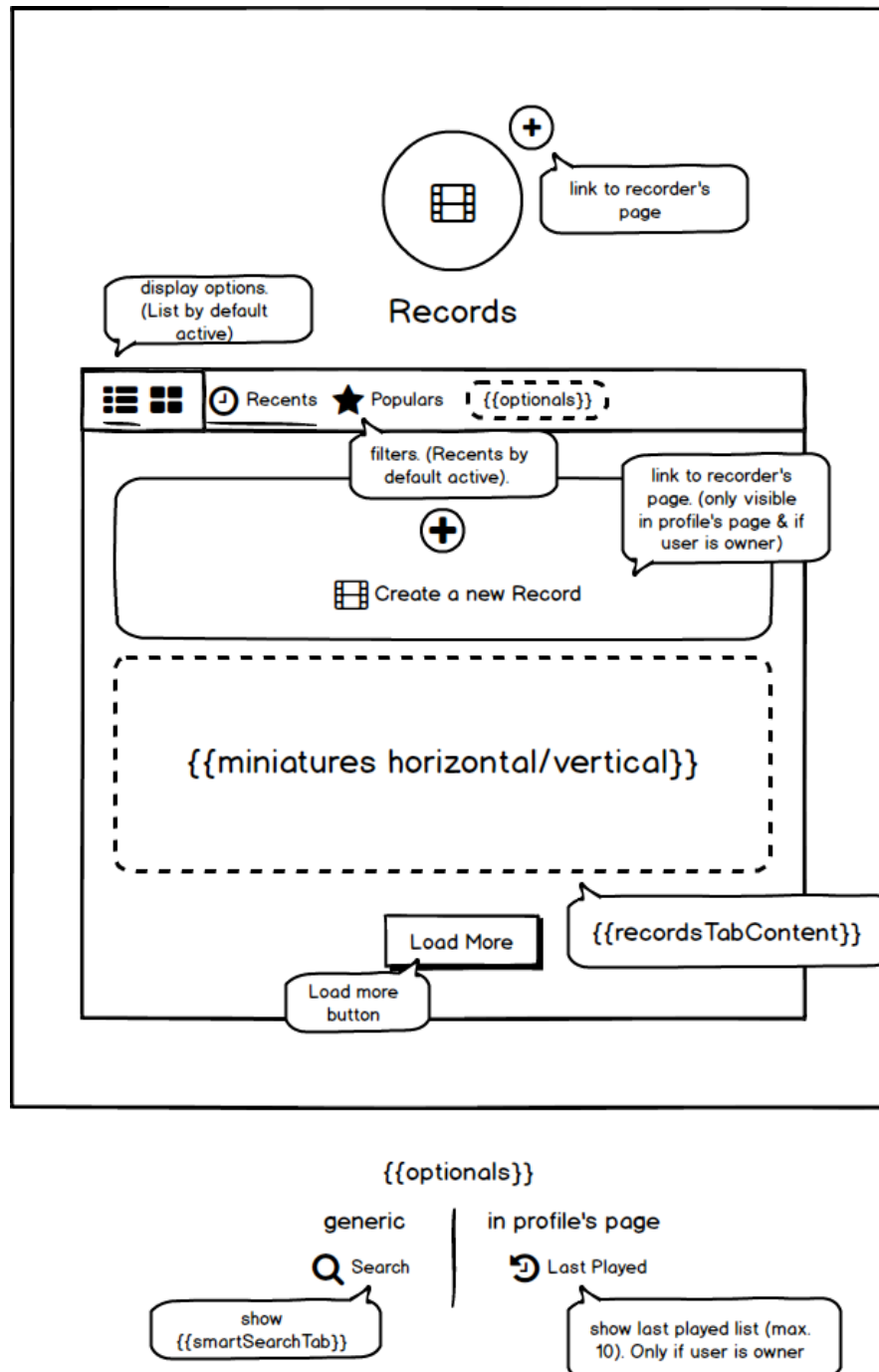


Figura B.9: Lista de grabaciones

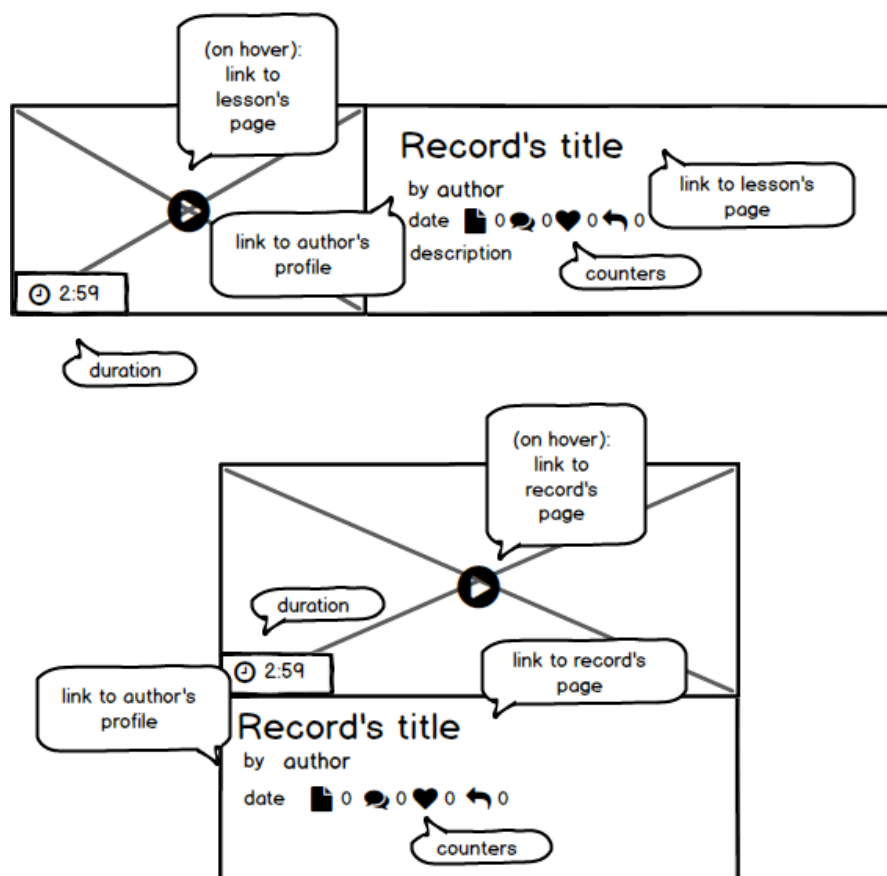


Figura B.10: Miniatura grabación