# Proyecto fin de carrera:
# ***Edición colaborativa.***

Alumno: Víctor Santiago González.

## Convenciones en este documento:

Se han incluido varios fragmentos de código informático que aparecen en letra `mono-espaciada`. No pretenden ser *scripts* válidos, si no servir de ejemplo de cómo se han resuelto diversos problemas de la aplicación.

Las palabras subrayadas tienen asociada una definición en el glosario final. Se trata de términos de uso común en el ambiente del desarrollo web. Pero para los que, quizá, sea conveniente una explicación *ad-hoc*.

**Índice.**


1. **Descripción de la aplicación** pág 4

2. **Secciones del wiki** pág 6
    - **2.1.** Menú
    - **2.2.** Perfil de usuario
    - **2.3.** Creación de usuarios
    - **2.4.** Mensajes-Mensajes
    - **2.5.** Grupos.
    - **2.6.** Documentos.

3. **Herramientas y método de trabajo.** pág 13

4. **Desarrollos propios.** pág 17
    - **3.1.** Formularios.
    - **3.3.** Diálogos.
    - **3.4.** Quill-reactive.
    - **3.4.** Listados

5. **Estructura de la aplicación y base de datos.** pág 22
    - **4.1.** Limitación de la visibilidad de una colección.
    - **4.2.** Limitación de la capacidad de manipular datos.
    - **4.3.** Métodos auxiliares
    - **4.4.** Tratamiento de datos en el wiki Teclea.

6. **Cómo probar el código.** pág 29

7. **Conclusión, Bugs conocidos y posibles ampliaciones.** pág 30

8. **Glosario.** pág 31

9. **Bibliografía web.** pág 33

## *1. Descripción de la aplicación.*

Como motivación para el proyecto de fin de carrera he elegido la edición colaborativa. En particular la creación de un *wiki* que he llamado '**Teclea**' cuya intención es servir de herramienta de apoyo durante la celebración de cursos universitarios.

La vista inicial del wiki.

Tal y como se definió en documentos anteriores a lo largo de este curso, las funcionalidades que debe proporcionar la aplicación son las siguientes:

1. Permitir la creación y login de sus usuarios.
2. Albergar diferentes tipos de usuarios: un administrador, profesores y alumnos. Con diferentes atribuciones y capacidades.
3. Permitir segmentar la audiencia de forma que se puedan crear grupos formados por profesores y usuarios que representen a cada clase en un curso.
4. Permitir la comunicación interna mediante un sistema de mensajería propio.
5. Ofrecer estadísticas de apoyo a los profesores de forma que se pueda saber qué alumnos se muestran más activos en la aplicación.
6. Proporcionar un mecanismo para crear y editar documentos dentro del wiki que soporte la edición concurrente de los mismos.

El trabajo llevado a cabo está disponible en la URL:

`pfc.meteor.com`

Para la presentación en vídeo del proyecto he creado varios usuarios que están disponibles en la dirección anterior.

Los grupos y los usuarios que he creado para el vídeo estaban inspirados en las plantillas de dos equipos de la primera división de fútbol. El Real Madrid y el Atlético de Madrid.

Siguen activos, así que se pueden utilizar para acceder:

- Como administrador: ttzara@gmail.com
- Como profesor: cho@gmail.com y zz@gmail.com
- Como alumno: cris@gmail.com o gab@gmail.com. (entre otros)

En todos los casos la contraseña es **pulgar.**

El código fuente, que se adjunta a éste documento está también alojado como proyecto privado en *BitBucket*. De ser necesario podría compartir el acceso al repositorio *Git* por esta misma plataforma.

## 2. Secciones del wiki.

La actividad dentro del wiki se divide en varias secciones que se presentan (o no) al usuario según el rol que adopte dentro de la plataforma.

### 2.1. Menú de la aplicación.

El menú principal permanece visible en todas las secciones de la aplicación. Si bien, no es igual para todos los usuarios: las secciones son visibles o se ocultan en función de las capacidades que determina el rol de cada usuario.

Menú de Administrador (Víctor), profesor (Raquel) y Alumno.

Así, los usuarios según su perfil disponen de las siguientes secciones:

- **Perfil**: Común a todos los usuarios.
- **Mensajes**: Para profesores y alumnos.
- **Grupos**: Para el administrador y los profesores.
- **Usuarios**: Para el administrador y los profesores.
- **Documentos**: Para los profesores y los alumnos.

Dentro de cada sección la información está segmentada en función de otros criterios, como veremos a continuación.

## *2.2. Perfil de usuario.*

Es la sección inicial. Dónde cada usuario puede editar sus datos personales, tales como su nombre, correo o la contraseña.

Si además el usuario es profesor de un grupo en el que ya ha habido alguna actividad, se mostrará también una sección de estadísticas.

Me llamo Raquel Sánchez
Los demás verán mi dirección de correo: topologia@gmail.com
Y quiero cambiar mi contraseña:
Nueva contraseña
Repetir contraseña
¿Cambiar la foto?
GUARDAR
Modificar el perfil

Actividad
UN GRUPO
El grupo UN GRUPO tiene 1 documentos con 0 secciones.
otro otro
Creó 1 documentos
1 secciones.
y el 30% de las ediciones de texto
alumno alu
Creó 3 documentos
0 secciones.
y el 69% de las ediciones de texto
Usuario más activo: alumno alu
Creó 3 documentos
0 secciones.
y el 69% de las ediciones de texto
Estadísticas del grupo

Por cada grupo, se informa de la actividad de los usuarios respecto a la creación de documentos, la creación de secciones y el número de aportaciones que halla realizado a los documentos del grupo.

Además, se indica quién de entre los usuarios del grupo se ha mostrado más activo. En base a los tres criterios anteriores, siendo la edición de documentos el aspecto que más peso tiene en la elección.

## *2.3. Creación de usuarios.*

Esta sección, presente para profesores y para el administrador, permite introducir nuevos usuarios en el sistema. Y realizar búsquedas entre los ya existentes.

USUARIOS
Crea, edita o busca a los usuarios del wiki
Filtros
Crear/Borrar usuarios
Nombre del usuario
ALUMNOS
PROFESORES
Listado de usuarios
¿En qué grupo está cada usuario?

| Usuario | Grupo |
|---|---|
| Cholo Simeone | ATLÉTICO DE MADRID |
| Jan Oblak | ATLÉTICO DE MADRID |
| Gabi Fernández | ATLÉTICO DE MADRID |
| Antoine Griezman | ATLÉTICO DE MADRID |
| Sergio Ramos | REAL MADRID |
| Karim Benzemá | REAL MADRID |
| Cristiano Ronaldo | REAL MADRID |
| Zinedine Zidane | REAL MADRID |

En principio el único usuario que existe desde el inicio de la puesta en producción de la aplicación es el usuario administrador. Por lo que es él quien debe crear a los profesores y alumnos.

Si bien cada profesor, desde el momento en que se le da de alta, puede crear también usuarios de cualquier tipo excepto administradores.

### *2.4. Mensajes.*

La vista de mensajes permite leer los mensajes recibidos, enviar mensajes nuevos o responder a los recibidos.

Enviar un nuevo correo.

Enviar a...

Asunto

ENVIAR

MENSAJES

Mantente al día de la actividad en el wiki

Nuevos correos

Listado de correos (0 no leídos)

NO LEIDOS

lkj (De: raquel / topologia@gmail.com)

ljk

Listado de correos y respuestas

Respuestas

De: Alumno

lkj

De: Raquel

Sin leer para el alumno

Responder

AÑADIR RESPUESTA

Los profesores y los alumnos pueden enviar mensajes, pero sólo pueden iniciar conversaciones con usuarios con los que compartan un mismo grupo.

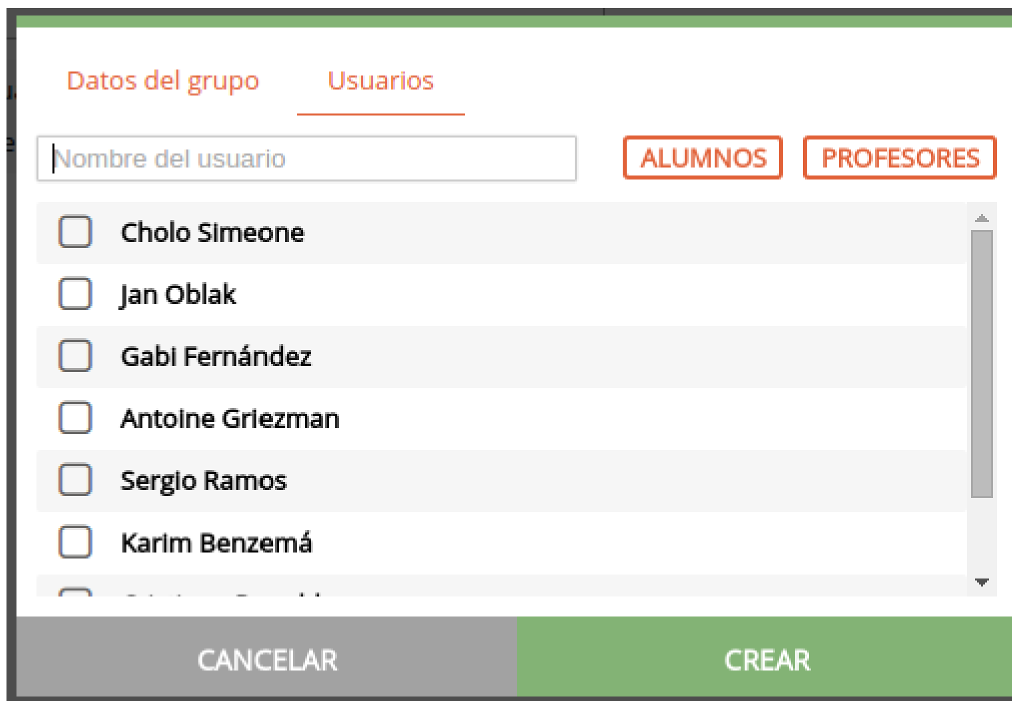
El administrador, sin embargo, puede comunicarse con cualquier otro usuario del *wiki*.

## *2.5. Grupos.*

Esta sección, disponible para el administrador y los profesores, permite seleccionar un conjunto de usuarios con los que crear un grupo.

GRUPOS
Crea, edita y añade usuarios a tus grupos
Crear/Borrar grupos
Nombre del usuario
Filtros
FINALIZADOS EN ESPERA ACTIVOS
Nombre: **NUEVO GRUPO**, 2 usuarios.
Desde el 04-01-2016 hasta el 08-01-2016
Actualizar grupo
Nombre: **UN GRUPO**, 3 usuarios.
Desde el 02-01-2016 hasta el 14-01-2016
Listado de grupos

Datos del grupo Usuarios
Nombre del grupo:
Fecha de inicio:
dd/mm/aaaa
Fecha de fin:
dd/mm/aaaa
CANCELAR CREAR




Vistas de administración de grupos y la ventana de creación/edición con sus dos pestañas: para definir el grupo, y para añadir o retirar sus usuarios.

Esta es una parte importante de la aplicación, ya que los documentos son accesibles para los usuarios en la medida que forman parte de un grupo que los posee. Y, como hemos visto, limita también la capacidad de comunicarse entre ellos.

## 2.6. Documentos.

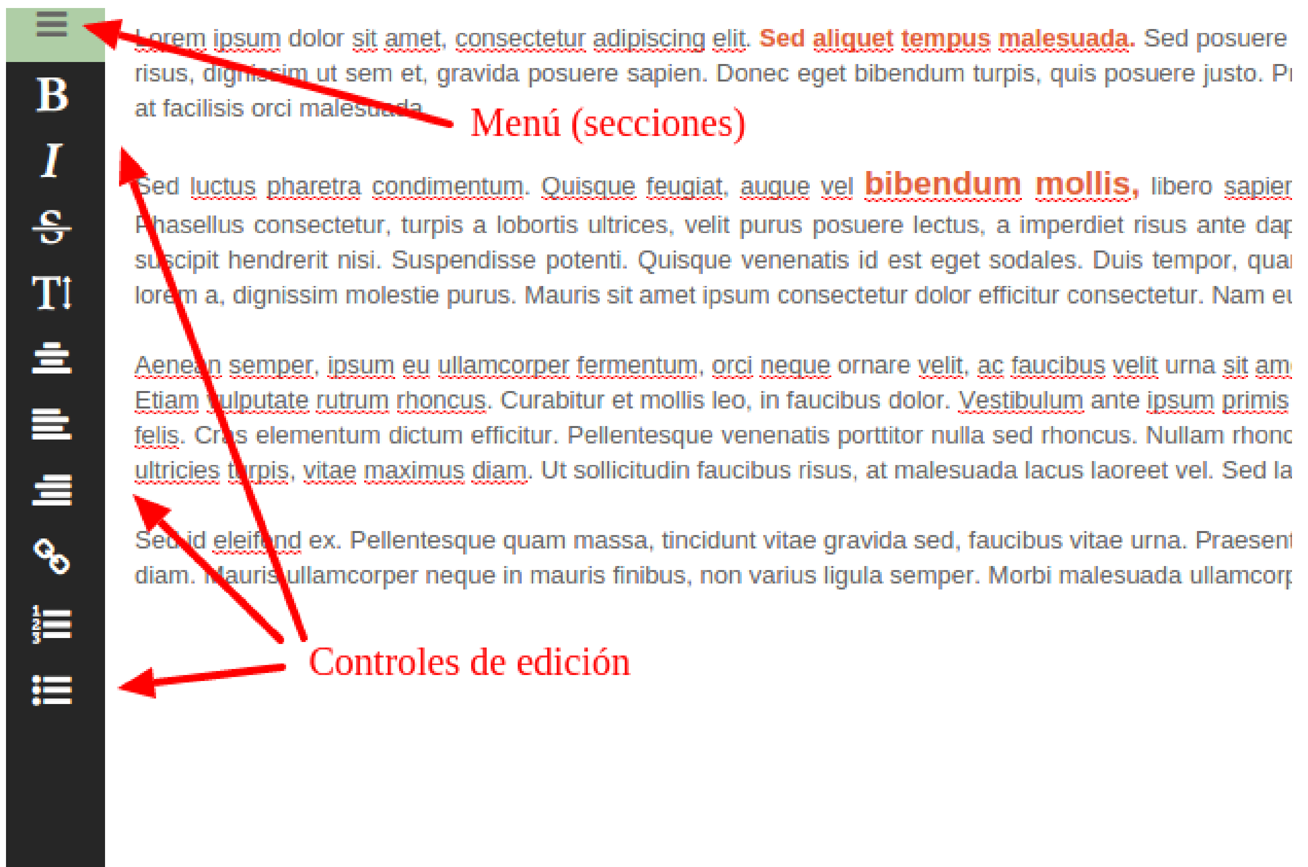
Los documentos quedan disponibles para los profesores y alumnos que podrán crearlos o editarlos siempre en asociación con un grupo pre-existente.

A diferencia de las secciones anteriores, los documentos se presentan a lo largo de dos vistas diferentes: la vista de creación o borrado y la de edición del documento en donde también se pueden añadir las secciones.

- **Vista de creación/borrado.**

Listado de documentos y ventana de creación.

- **Vista de edición y gestión de las secciones.**




Vista de un documento en el que el menú muestra sus dos estados: la edición de texto y la edición de las secciones del documento.

La característica principal de la edición de textos es que admiten concurrencia: **dos usuarios pueden escribir simultáneamente el mismo texto desde dos ordenadores diferentes**. Como en el caso de un único editor, sus acciones serán grabadas automáticamente y los efectos de sus cambios se muestran a cada uno de los participantes que, en ese momento, esté editando el documento.

## *3. Herramientas y método de trabajo.*

Para completar el trabajo he recurrido a varias herramientas de acceso libre que resumo a continuación:

1. **Repositorio de código GIT y método de trabajo.**

   ***Git*** es un sistema de control de versiones descentralizado, que ha alcanzado una gran popularidad tanto para uso personal como en el entorno empresarial.

   Si bien la elección más habitual es alojar el código en la web ***GitHub***, en este caso he optado por gestionar el repositorio remoto a través del servicio ***BitBucket***. La elección de éste último en lugar de ***GitHub*** se debe, fundamentalmente, a razones personales: ***Bitbucket*** permite la creación (gratuita) de repositorios privados y, por ello, ya tenía varios proyectos personales alojados allí antes de comenzar el desarrollo. Por lo que estaba más familiarizado con el uso de ***Git*** sobre esta plataforma.

   En cualquier caso, las convenciones de uso de ***Git*** que afectan al desarrollo son independientes de la plataforma.

   Básicamente ***Git*** ofrece una serie de funcionalidades que permiten efectuar *commit* remotos, crear ramas con versiones divergentes del estado del proyecto, revertir cambios, etc... Si bien no impone ninguna regla particular sobre cómo se ha de organizar el trabajo o cómo gestionar las versiones de producción o desarrollo. ***La política concreta que ha de seguir un equipo para trabajar con Git es decisión del propio equipo.***

   En este caso he tratado de adaptarme al método de trabajo denominado *Git-flow.* En el cual el desarrollo se acomoda al uso de tres ramas simultaneas:

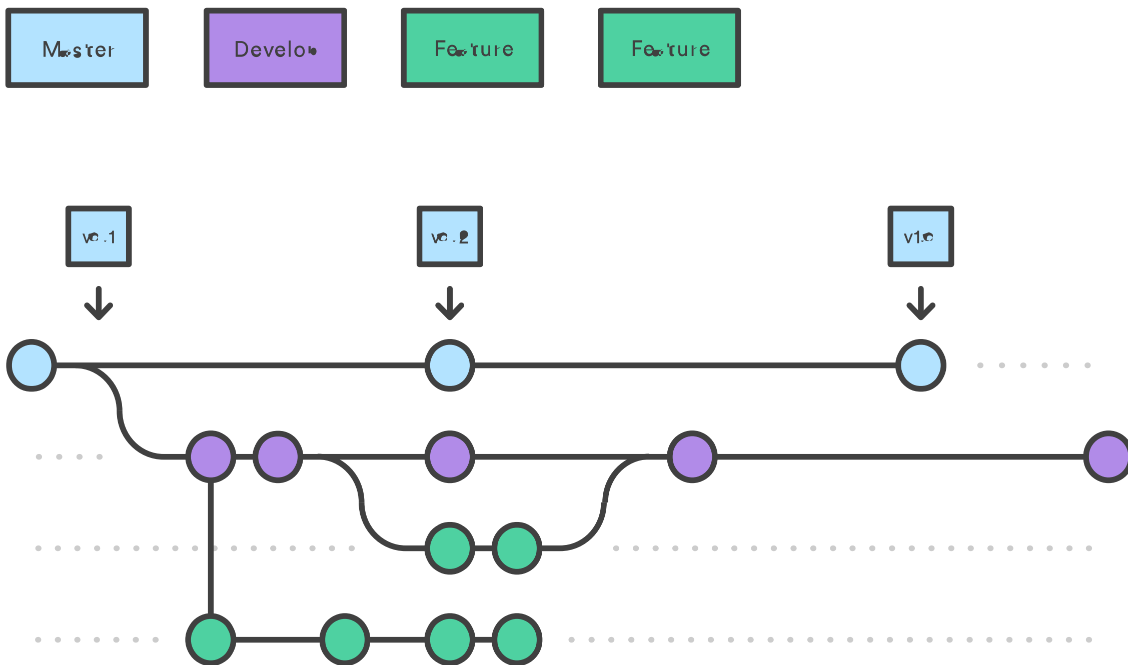


Diagrama del ramas en el modelo Git-flow.

- **Rama master**: en la cual sólo se vuelcan cambios suficientemente probados como para presentarlos al cliente. **Es la rama de producción.** Esto es la versión del código que se expone al navegador del cliente.
- **Rama dev: es la rama de desarrollo.** Y cumple un doble papel: es el lugar desde el que alimentamos la rama master. Pero también el punto en el que los desarrollos parciales de cada módulo se fusionan. De forma que se puede probar que los diversos aspectos de la aplicación no crean conflictos entre sí (y en su caso resolverlos) antes de que el código llegue a producción. La rama 'dev' nace desde 'master' y se inserta en ella a medida que se van consumando los distintos hitos del proyecto.
- **Ramas feature:** para **desarrollos parciales**. Son ramas dedicadas al trabajo en un aspecto concreto de la aplicación. Por ejemplo, en el repositorio de éste trabajo existe la rama 'messages' que está dedicada a dar forma a la mensajería de la aplicación. De esta manera se consigue compartimentar el trabajo en cada una de las areas que componen el desarrollo. Cada una de las ramas 'feature' nacen de 'dev' y vuelcan su contenido también en 'dev'.
- **Ramas hotfix:** son ramas **para gestionar bugs.** Cada vez que se hallan mezclado dos ramas, cabe la posibilidad de que algún bug se halla escapado a nuestro control. En estos casos la solución se aplica sobre una rama **hotfix** en la que se puede comprobar que la solución al problema es funcional por si misma antes de propagar el arreglo al flujo de trabajo principal. Estas ramas pueden nacer a partir de *dev* o de una rama *feature* (según cuál sea la ubicación del error) pero no de *master*. Ya que en  ésta última el código debe permanecer libre de errores.

En principio, no hay ninguna restricción sobre el número de ramas que un sistema Git puede soportar. Pero lo normal es que a medida que se finalizan los desarrollos o se solucionan los bugs las ramas correspondientes (*feature* o *hotfix*) se eliminen para no crear confusión con los nombres de las mismas. Sólo *master* y *dev* han de permanecer activas durante toda la vida del proyecto.

En mi caso (dado que este ha sido un trabajo de una sola persona) no me ha parecido necesario aplicar reglas específicas para la nomenclatura de los commits o ramas más allá de mantener el esquema general anterior. Cosa que suele ser necesaria cuando se trabaja en un equipo más amplio.

## 2. Framework Meteor.

**Meteor** es un *framework* de programación que abarca todos los procesos necesarios para la creación de una aplicación web: tanto las tareas propias del *backend* como las del *frontend*.

*Logotipo del framework Meteor.*

Además de que se trata de un entorno de programación orientado por completo a JavaScript (incluso la BD se opera vía JavaScript) su característica principal es que adopta el modelo de

variables reactivas.

Construido sobre *Node.js* y basado en el uso de *sockets*, **Meteor**, proporciona un mecanismo transparente de sincronización entre el valor de una variable en el cliente y su representación en Base de Datos.

Esto es, la modificación del valor de una variable en la vista (por ejemplo, porque el usuario escribe algo en pantalla) repercute instantáneamente en su valor en la base de datos. Y viceversa, si un *script* produce cambios en una variable que se está mostrando al usuario como parte de la vista, ésta se actualiza de forma automática e instantánea.[1]

De esta manera, las aplicaciones construidas con **Meteor** no necesitan de ningún tipo de *middleware* para mediar en la comunicación entre la BD y el flujo de ejecución. Lo cual, entre otras cosas, hace prácticamente innecesario seguir un patrón de programación típico como MVC en favor de otras alternativas.

En mi opinión, se trata de una aproximación bastante radical al problema de la programación de aplicaciones web. Y bastante novedoso. Tanto que, en realidad, no existe aún un consenso en la comunidad **Meteor** de cuáles son las buenas prácticas a seguir o cómo debe afrontarse el diseño de una aplicación.

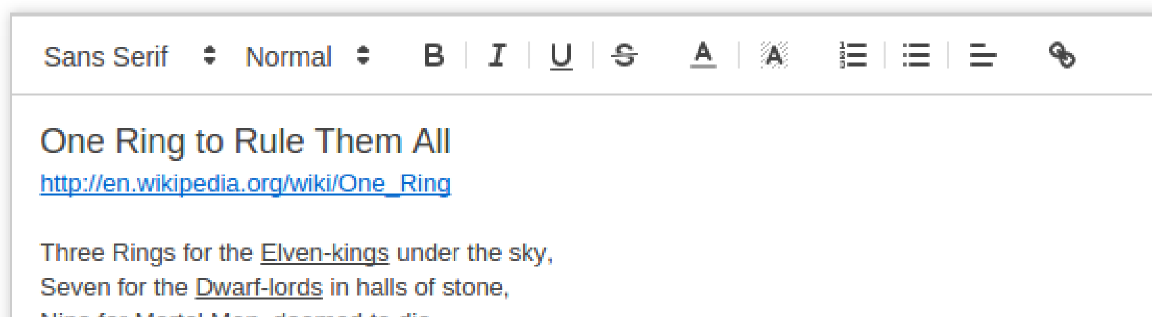
***La forma ortodoxa de programar con Meteor es... la que elija el propio programador.*** Sin (casi) ningún tipo de condicionamiento de la herramienta sobre la arquitectura de la aplicación.

En las siguientes secciones, explicaré cuál ha sido mi forma de abordar la implementación de la aplicación.

## 3. Editor de texto enriquecido Quill.

Una parte importante del proyecto consistía en ofrecer la posibilidad a los usuarios de editar textos en la aplicación.

En principio esto conduce a la necesidad de implementar un editor de texto enriquecido. Pero este tipo de *widgets* son, en general, asombrosamente[2] complejos de crear desde cero. Así que he optado por usar código de terceros para solventar las tareas de bajo nivel tales como marcar un texto en negrita, cursiva, etc...




1 Tampoco es que Meteor haga magia: sigue habiendo que considerar la latencia de red.
2 Hay varios artículos relevantes al respecto en la bibliografía. En particular, merece la pena leer éste: https://medium.com/content-uneditable/contenteditable-the-good-the-bad-and-the-ugly-261a38555e9c#.dsawl2vc0

*Vista genérica del editor Quill, tal y como aparece en la página del fabricante.*

Hay una gran cantidad de alternativas disponibles para esta cuestión: **CKEditor, TinyMCE, Medium...** Si bien, he elegido usar el editor **Quill.js**. Por varias razones:

- **Está programado íntegramente en JavaScript y HTML. Y su aspecto es editable vía CSS.** Lo cual supone una gran ventaja a la hora de integrarlo en un proyecto que está implementado al 100% usando dichos lenguajes.
- **Es un editor orientado a un uso programático.** Como la mayoría de editores de texto disponibles **Quill** puede ser usado en modo *plug & play* y permite interactuar con el texto a partir de funciones JavaScript. La diferencia es que en el diseño de **Quill** ha influido más la segunda opción. De forma que dispone de una API que expone por completo las funcionalidades del editor. Y consigue que sea mucho más sencillo operar en él modificaciones no triviales.

## 4. Base de datos mongoDB.

**MongoDB** es una base de datos **NoSQL**. Es decir, una base de datos orientada a documentos no-relacional. Aunque la mejor forma de definir una base de datos de este tipo (tal y como hace el propio acrónimo **NoSQL**) es por oposición al esquema tradicional **SQL**. Veamos, pues, las dos peculiaridades principales de **MongoDB** que han influido de manera importante en la realización del proyecto:

- Todas las consultas (inserciones, modificaciones, borrado y lectura) se ejecutan en *JavaScript*, en lugar de **SQL**.
- No existe la posibilidad de efectuar el equivalente a una operación **JOIN** sobre los documentos (que aquí hacen el papel de tablas) de la base de datos.

Más adelante detallaré en profundidad la manera en que la aplicación gestiona su relación con la base de datos. Por ahora basta señalar que **MongoDB** es la única herramienta incorporada al proyecto de forma no voluntaria: **es un requisito del framework Meteor.**

## 5. Pre-procesador CSS LESS.

La maquetación de las vistas se ha realizado mediante el *pre-procesador* CSS LESS. Que permite escribir el código de estilo de una manera más versátil (por ejemplo, se pueden anidar las clases CSS) y, sobre todo, es capaz de gestionar los datos escalares correspondientes a la maquetación de la página en variables.

Así, los colores usados en las vistas (diferentes tonos de naranja, verde o negro) podrían modificarse simplemente cambiando el valor de las variables en que fueron definidos en primer lugar. En vez de tener que propagar manualmente el cambio en cada hoja de estilo.

## 4. Desarrollos propios

Además de las herramientas mencionadas en la sección anterior, parte del trabajo ha consistido en la creación de desarrollos propios que integrar con el resto del sistema. Los detallamos a continuación:

### 4.1. Formularios.

Bajo mi punto de vista, una de las carencias más importantes de HTML es su tratamiento de los objetos de formulario. Si bien es cierto que existe una gran variedad de etiquetas para recoger información de los usuarios y que se dispone de funciones de acceso a los datos nativas de JavaScript; también es cierto que la colección de objetos a disposición de los desarrolladores es demasiado heterogenea como para resultar plenamente funcional.

Por ejemplo, las etiquetas:

`<textarea>` e `<input type=”radio”>`

Sirven para presentar un caja de texto y un botón de radio con los que el usuario puede ingresar texto de su elección o elegir una de entre varias opciones que se le presentan. Posteriormente, para recoger los datos, se deben ejecutar funciones de JavaScript como las siguientes[3]:

`$('textarea').val()` que devuelve el texto introducido por el usuario.
`$('input').prop('checked')` que indica si el botón de radio ha sido elegido o no.

No es un mal sistema si la cantidad de formularios que se quieren manejar es limitada. Pero, si como es el caso en el *wiki* Teclea, debe manipularse una gran cantidad de formularios se hace muy difícil automatizar el proceso. Las dos etiquetas tienen nombres distintos, lo que dificulta su detección entre el resto de elementos HTML. Y, además, una vez identificadas el acceso a los datos no es homogeneo: en el primer caso se debe usar la función `.val` mientras que, en el segundo, recurrimos a `.prop`. **A pesar de que, conceptualmente, ambos son inputs de usuario.**

Para evitar esta dificultad, he optado por abstraer la oferta de etiquetas de HTML marcándolas todas con el mismo selector .css: la clase .js-input. Y programando, para cada una de ellas, un objeto que uniformiza sus métodos de acceso.

Así, cada nodo en un documento HTML que incorpore la clase .js-input será detectado y asimilado a un objeto Input de JavaScript cuyo prototipo es el siguiente:

```
Input = function () {}

Input.prototype.isDisabled = function () {...} //Informa si el objeto está deshabilitado

Input.prototype.isEnabled = function () {...} //Informa si el objeto está habilitado

Input.prototype.isValid = function () {…} // Ejecuta validación sobre el formato de datos
```

Que sólo implementa los métodos comunes a todos los inputs.

A partir de aquí, se crea un objeto input que añadir al prototipo para cada etiqueta objetivo en

3 El ejemplo está escrito asumiendo el uso de jQuery. Pero es aún peor si nos limitamos a usar JavaScript nativo

HTML. Por ejemplo, una etiqueta típica como:

```
<input type="text">
```

Queda representada por el objeto:

```
Input.text = function () {
 var self     = this;
 self.get     = function () {}  // Obtiene el valor del objeto HTML.
 self.set     = function (value) {}  // Modifica el valor del objeto HTML.
 self.enable  = function () {} // Habilita el objeto HTML
 self.disable = function () {} // Deshabilita el objeto HTML
 self.error   = function (msg) {} // Muestra el mensaje de error en 'msg' si no es vacio
 self.clear   = function () {} // Resetea el valor del objeto a su estado neutro.
}
```

A partir de aquí, se dispone de una colección de 'Inputs' cuyos métodos son diferentes en su actividad, pero iguales en su nombre y funcionalidad.

Un formulario, entonces, es una ***factoría de inputs***. Es decir un tercer objeto que revisa la vista en búsqueda de nodos marcados con la clase .js-input. Y en cada caso crea el objeto JS que se adapta a su tipo, declarado en la propia etiqueta.

Toda vez que el objeto Formulario tiene también sus propios métodos:

`action` → para recuperar los datos del formulario en formato JSON.
`validate` → para comprobar que los datos son validos en función de diversos criterios.
`errors` → muestra los mensajes de error (si los hay) del formulario y sus inputs.

Procesar un formulario en el wiki Teclea, consiste en lo siguiente:

1. Se escribe la vista del formulario marcando los input necesarios con la clase .js-input y su tipo en el atributo data-type.
2. Se crea el formulario con la instrucción new Form('id-del-formulario')
3. Si hay errores, se puede detectar llamando a:

```
if (form.validate() > 0)
```

4. Si se encontraron errores, se muestran mediante: `form.errors()`.
5. Si todo es correcto, ya se puede usar su valor de cara a la BD dado por `form.action()`

El motivo por el que el tipo de cada input se indica mediante un atributo data-type en lugar del más común atributo type de HTML es por compatibilidad.

En primer lugar no todos los objetos de formulario nativos en HTML tienen dicho atributo[4]. Y en segundo lugar, esto abre la posibilidad a que nuestros formularios puedan lidiar con objetos distintos a los nativos de HTML. Así si en un momento dado las especificaciones de diseño requieren un widget muy elaborado que, por ejemplo, produzca un valor único a partir de dos o más nodos, bastará con crear un programa ad-hoc que disponga de los mismos métodos de acceso que el resto de inputs posibles. Actuando así, se pueden procesar de la misma forma formularios compuestos únicamente de objetos nativos en HTML u objetos propios que respondan a las

4 En realidad siempre se puede usar el atributo `type`. Pero en algunos casos esto contraviene las reglas de validación del W3C en función del doctype.

necesidades de nuestra aplicación.

En particular, durante la realización del wiki se ha necesitado normalizar de esta forma varios elementos de diseño cuya función es recoger datos del usuario. Son los siguientes:

- **Input Editable**: para la entrada de texto con los datos del perfil de cada usuario.
- **Input Editor:** un editor de texto enriquecido mínimo. Que se usa para editar el texto de los correos de los usuarios.
- **Input Token:** para que el usuario pueda ingresar datos en forma de etiquetas. Al estilo de los selectores coordinados con la etiqueta <datalist> en HTML5. Pero evitando que el navegador deba conocer la colección completa de opciones, por motivos de eficiencia en el uso de memoria en el cliente.
- **Selector Multiple:** un selector de opción múltiple. Que se usa para que el usuario pueda elegir los destinatarios de un correo electrónico mediante un buscador.

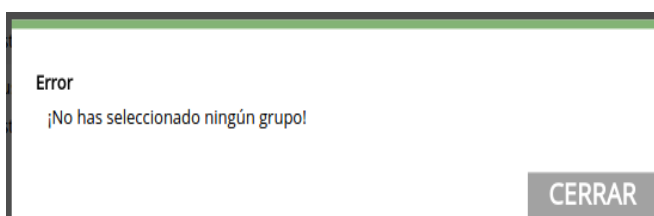
El código necesario para hacer funcionar la clase de formularios se puede encontrar en la siguiente ruta, a partir de la carpeta raíz del proyecto:

```
app/lib/form.js
```

## 4.2. Diálogos.

De la misma manera que en el caso de los formularios, se ha implementado también una factoría de ventanas modales que han quedado agrupadas bajo el nombre 'Dialog'.

En este caso, la motivación principal es adecuar el funcionamiento de las ventanas nativas de javascript (alert, prompt, confirm) al diseño de la aplicación. Además de contar con un método sencillo de definir ventanas modales.



Ventana de alerta producida con Dialog frente al nativo de JS. La primera es igual en todos los sistemas operativos.

Siguiendo el mismo esquema que con los formularios se obtiene un objeto global, cuyo modo de uso es a través de la siguiente signatura:

- **Ventanas con equivalentes JS:**

```
Dialog.alert({content: 'texto', title: 'titulo', callback: function (option) {}});
Dialog.prompt({content: 'texto', title: 'titulo', callback: function (option) {}});
Dialog.confirm({content: 'texto', title: 'titulo',
                callback: function (option, text) {}});
```

Dónde:
'**content**': es el texto que aparecerá en la ventana.
'**title**': es el título para la ventana.
'**callback**: es una función que se ejecutará una vez que el usuario cierre la ventana o pulse alguno de los botones. Recibe argumentos para indicar si la acción del

usuario ha sido positiva o negativa. Y, en el caso de prompt, qué texto ha introducido.

- **Ventanas modales:**

```
Dialog.alert({content: 'clase', template: 'id del template'});
```

Dónde:
'**content**': indica la clase .css en la que basaremos la maquetación del interior de la ventana.
'**template**': la plantilla meteor (HTML+JS+CSS) con la que gestionaremos el contenido de la ventana.

Dado que la motivación para este desarrollo son, fundamentalmente, consideraciones de diseño; en todos los casos se ha implementado también las correspondientes clases CSS que uniformizan la interfaz.

### 4.3. Quill-reactive.

En general se ha usado el editor de texto Quill sin demasiadas modificaciones (por ejemplo en la sección de correo o documentos). Sin embargo, para la edición de documentos ha sido necesario introducir cambios muy sustanciales sobre el código de Quill.

A partir del paquete Meteor quill-reactive se ha desarrollado una versión propia con las siguientes modificaciones:

1. Cambio por completo de la IU. Para que se utilicen estilos y botones adaptados a las especificaciones de diseño del wiki Teclea. En lugar de los que se usan por defecto. La única parte que ha sobrevivido a esta moficiación es el area editable.
2. Adaptación para que Quill modifique el aspecto del texto en base a métodos de su API asociados a los nuevos controles de la interfaz.
3. Adaptación para que Quill use variables reactivas en comunicación con la base de datos de la aplicación.
4. Detección de eventos de inserción y cambio en el texto e implementación de los cambios necesarios para que se propagen en tiempo real a todos los usuarios que estén usando un mismo documento a la vez.
5. Callbacks asociados a los cambios en el texto o su formato para recolectar estadísticas.

Todos estos cambios han sido derivados del paquete quill-reactive creando una versión propia que se distribuye junto con el código del wiki en la carpeta /packages.

Por su especificidad el nuevo paquete no es apto para su publicación en el repositorio genérico de Meteor. Por lo que es necesario (en cada nueva instalación del wiki) de añadir el paquete expresamente.

Esto se puede conseguir lanzando el comando de terminal:

meteor add quill-reactive-custom

desde la carpeta raíz de la aplicación.

### 4.4. Listados.

Como pudo verse en las capturas de pantalla de la sección 2, muchas de las vistas del wiki incluyen un listado de datos que permite visualizar los elementos de una collección, seleccionarlos y actuar sobre ellos.

En el **wiki Teclea** todos estos listados se generan dinámicamente a partir de una misma plantilla cuyos componentes pueden encontrarse en:

```
app/templates/list/list.html
app/templates/list/list.js.
app/lib/components.less
```

La función de esta plantilla es doble:

En primer lugar, crear los elementos del DOM mínimos

Y en segundo lugar, crear una estructura de datos común a todos los listados. Es decir, la parte JavaScript de la plantilla se encarga de recuperar los datos desde la base de datos e insertarlos en la vista. Pero también de gestionar la selección de ítems guardando el estado en una variable de sesión, desde el que pueden ser consultados por el resto de elementos de la vista tales como las ventanas modales de creación y edición.

## *5. Estructura de la aplicación y Base de datos.*

### 5.1. Árbol de carpetas. El código en el lado cliente y servidor.

A diferencia de la mayoría de frameworks, Meteor no impone ninguna estructura de carpetas al desarrollador: **el código se puede organizar como se considere más conveniente**.

Con dos excepciones:

1. Los archivos ubicados en la carpeta /**lib** se incluyen en primer lugar en la pila de archivos *javascript* que se envían al cliente. Por lo que comúnmente (y este es el caso), se utiliza para definir librerías auxiliares de alcance global. Cualquier otra carpeta de nombre **lib** tiene también prioridad sobre el resto de carpetas de su ámbito (i.e. hasta el tope impuesto por su carpeta padre) con lo que su uso recomendado es similar.

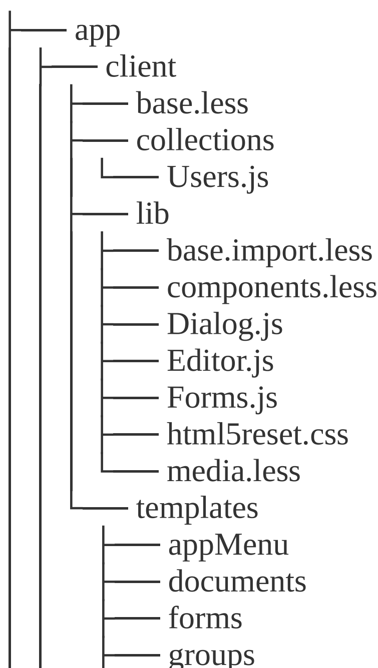
2. Cualquier archivo puede ejecutar código en el lado cliente o en el lado servidor, sin más que situarse en la condición apropiada del condicional:

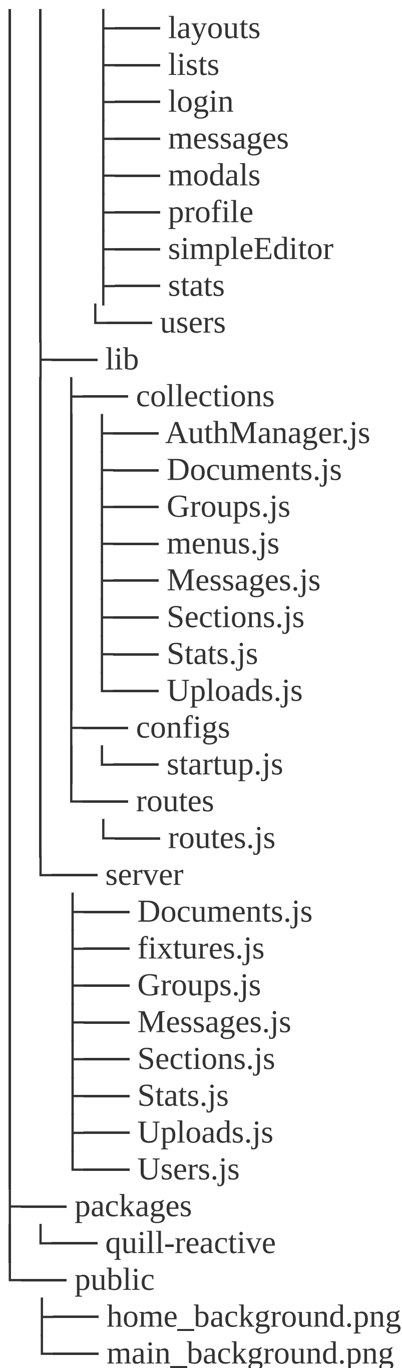
```
if (Meteor.isClient)

if (Meteor.isServer)
```

Si bien, el framework garantiza que todos los archivos contenidos dentro de la carpeta **/server** serán ejecutados exclusivamente en el servidor. Por lo que se tiende a utilizar como almacen del código que gestiona la base de datos.

Las consideraciones anteriores **no son reglas, son recomendaciones**. O sea, se puede implementar la aplicación sin utilizar las carpetas /lib o /server. Pero resultan útiles y, en este caso, las he utilizado.

Así, el árbol de carpetas final es como sigue:






Destacan las siguientes:

1. **Carpeta /lib.**
   Que contiene clases para definir modelos de datos respecto de las colecciones en la BD. La configuración de las rutas (i.e. la manera en que el framework reacciona a los cambios en la URL para responder con una vista) y parámetros de configuración del comportamiento del código en el cliente.

2. **Carpeta /client/templates.**
   Donde quedan definidas todas las vistas de la aplicación. Cada una ocupa una carpeta con su nombre que alberga los tres archivos que componen una vista: su la maquetación (archivo .html), su código cliente (archivo .js) y los estilos (archivo .less)

3. **Carpeta /server**
   Donde se almacenan los archivos que dan acceso a la BD.

4. **Carpeta /packages.**

   Es una carpeta especial en la que se ha desarrollado una versión propia del editor Quill integrada con el resto de elementos de la aplicación y que hace uso de variables reactivas de Meteor.

5. **Carpeta /client/lib**

   En la que aparecen archivos de uso general en el lado cliente. Tales como el fichero normalizador html5reset.less, hojas de estilo de alcance global o los ficheros que implementan los desarrollos de los formularios y las ventanas modales comentados anteriormente.

La desventaja de esta forma de gestionar los archivos de la aplicación es que promueve los conflictos entre distintos módulos de la aplicación. Ya que aunque el framework gestione correctamente la precedencia de unos archivos sobre otros, se incluyen siempre todos en cada vista. Por lo que si, por ejemplo, dos plantillas usan una variable global con el mismo nombre entrarán en conflicto.

Por ello, es imprescindible (aunque no obligatorio) que cada unidad funcional de la aplicación esté completamente aislada de las demás y pueda funcionar independientemente. Cosa que, por otra parte, es una buena práctica de programación.

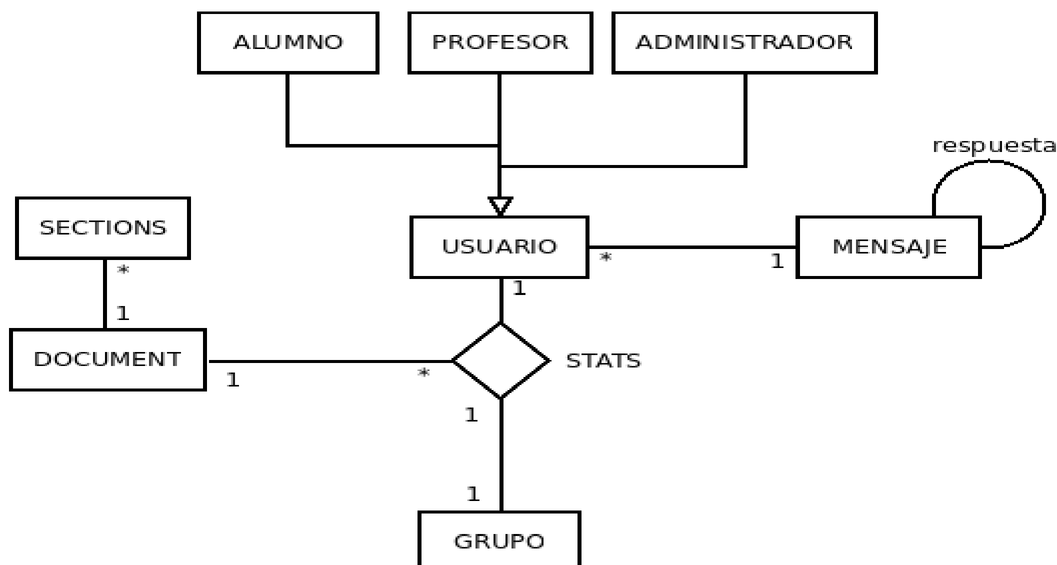
En el contexto del *framework* Meteor, la comunicación entre módulos se delega al objeto global Session en el que se pueden hacer apuntes de datos y lecturas entre módulos. Pero cada archivo concreto ha de funcionar dentro de su propio *closure*.

## 5.2. Diseño de la base de datos.

Si bien la base de datos MongoDB no es de tipo relacional, lo cierto es que la naturaleza de los datos que representa sí lo es. Las relaciones entre cada colección de la aplicación, se representan en el siguiente diagrama:

En el que quizá lo más interesante es la manera en la que se han implementado las relaciones entre las colecciones.

- La relación de herencia de la colección de usuarios se ha llevado a cabo incluyendo una



propiedad en cada documento de la colección que indica a qué sub-clase pertenece.

- Las relaciones entre **Grupos – Usuarios**, **Documentos-Secciones** y **Mensajes-Mensajes**, sin embargo se han realizado embebiendo el documento referenciado en la clase que lo va a usar. Por ejemplo, las referencias de los usuarios de un grupo aparecen en cada objeto de la colección Grupos como un array de objetos de la colección de usuarios.

### 5.3. Modelos de la BD.

El framework Meteor proporciona acceso a la base de datos mediante los siguientes métodos:

```
Collection.insert
Collection.update
Collection.remove
```

que son básicamente *wrappers* de sus equivalentes en la base de datos MongoDB. Por ejemplo:

```
Groups.insert({documento}, callback(err, id))
```

insertará un nuevo documento en la colección de grupos dado por el objeto que se pasa como primer parámetro. Y ejecuta tras la operación el callback que se le pasa como segundo parámetro, que recibe un objeto de error y el id del nuevo grupo en caso de que la operación halla podido completarse.

Sin embargo no existe (al menos de manera nativa) ningún concepto equivalente al de un modelo de datos en el patrón MVC. Ni tampoco hay más funciones auxiliares que las anteriores. Y en ocasiones no es suficiente.

Dado que los datos son accedidos en forma de objetos JavaScript, se ha podido mitigar este problema extendiendo el prototipo de los objetos de MongoDB en el momento que llegan al código del lado cliente.

Para ello cada archivo de la carpeta /app/lib/collections está dividido en dos secciones: **para extender los objetos de datos, y para extender la colección en sí.**

Si tomamos como ejemplo la colección de Documentos, el esquema es el siguiente:

```
// Extender los documentos.
Document = function (doc) {
      _.extend(this, doc);
}

_.extend(Document.prototype, {
   metodo1: function () {
      // Código del método.
      // Aquí this se refiere a un objeto de datos MongoDB.
   }
});

// Aplicamos la extensión en la BD.
Documents = new Mongo.Collection('Documents', {
  transform: function (doc) {
```

```
      return new Document(doc);
  }
});

// Extendemos la colección.
_.extend(Documents, {
   metodo2: function () {
      // Código del método
      // Aquí this se refiere a la coleccción MongoDB.
   }
});
```

Así, si continuamos con el ejemplo, podemos añadir métodos a la colección de Documentos tales como `Document.isValid` o `Document.addSection`, que simplifican mucho la comunicación entre el fontend y el *backend*.

## 5.4. Métodos de acceso a los datos.

### 5.4.1. Limitación de la visibilidad de una colección.

En Meteor el acceso a la capa de persistencia está mediado por el patrón de publicación subscripción. Esto es, para cada colección se permite el acceso a los datos de una colección mediante su publicación, implementando el método:

```
Colección.publish('Nombre de la colección', function () {
   // Reglas de acceso.
});
```

El método que se pasa como segundo argumento es el encargado de devolver el conjunto total de resultados que una consulta por parte de un usuario en el cliente podría llegar a recibir. Y en función del contenido de dicho método se puede ser o bien totalmente permisivo, o bien totalmente restrictivo hasta el nivel de una propiedad particular de un objeto de la colección.

No hay ninguna limitación en la lógica a implementar en los métodos de publicación. Pero en el **wiki Teclea** he optado por discriminar en función del rol del usuario.

Así, por ejemplo, los datos de la colección de documentos solo son visibles si el usuario está presente en el grupo al que ha sido asignado el documento.

Y en general, éstos métodos se han utilizado para implementar las restricciones de acceso detalladas en la primera sección.

Una vez publicada la colección, aún hay que ponerla a disposición de los *scripts* que se ejecutan en el lado cliente. Para lo cual se debe usar el método de subscripción:

```
Meteor.subscribe('Nombre de la colección')
```

Este método puede ejecutarse en cualquier punto de un código que se ejecute en el cliente. Por lo que lo que he optado por asociar las subscripciones de datos a nivel de plantilla. De manera que, por ejemplo, la plantilla de la sección Perfil no está subscrita a la colección de Mensajes.

Por razones de eficiencia y porque, en realidad, no lo necesita.

### 5.4.2. Limitación de la capacidad de manipular datos.

Además de limitar la visibilidad de los datos, Meteor, incorpora también la capacidad de limitar qué tipo de modificaciones puede realizar un usuario en los documentos que forman una colección.

A partir del método:

```
Meteor.nombre_de_la_collección.allow({
     'insert': function () {
          // Lógica para permitir o no las inserciones.
          return true/false
     }
     , 'update': function () {
          // Lógica para permitir o no las actualizaciones
          return
     }
     , 'remove': function () {
          // Lógica para permitir o no las eliminaciones.
          return true/false
     }
});
```

Cuyo funcionamiento es en todo similar al caso de la restricción de visibilidad: se puede permitir o denegar cada operación en función de cualquier criterio. Aquí se usa el criterio de discriminar en función del rol del usuario que solicita la operación.

### 5.4.3 Métodos auxiliares.

Además, de los métodos anteriores, Meteor permite definir un objeto de métodos auxiliares para facilitar la comunicación entre el servidor y el código que se ejecuta en el cliente:

```
Meteor.methods({
     'nombre_del_metodo' : function () {
          // Código del método.
     }
});
```

Son especialmente útiles a la hora de anteponer tareas antes de realizar una operación sobre la base de datos real. Por ejemplo, al crear una estadística se puede delegar en estos métodos el auto-completado de la fecha en la que se inscribe el log.

### 5.4.4. Tratamiento de colecciones de datos en el wiki Teclea.

En las condiciones anteriores, el tratamiento de datos y la conexión entre la actividad en el cliente y su repercusión en el servidor se despliega en varios archivos:

1. La plantilla: que se encarga de subscribirse a las colecciones que le son necesarias y ejecutar las acciones necesarias sobre los datos a través de las extensiones definidas sobre cada colección.
2. La extensión de los métodos de cada colección: definidas todas en la carpeta /lib/collections/nombre-de-la-colección.js
3. La definición de los métodos de acceso en el servidor: definidas para cada colección en los

archivos de la carpeta /server/collections usando para ello los métodos expuestos anteriormente.

En cualquier caso este método de trabajo es una elección personal. Ya que Meteor no impone ningún criterio sobre esta cuestión. La misma funcionalidad podría haberse implementado desarrollando esquemas o integrando los paquetes autopublish e insecure (del repositorio oficial de Meteor) que convierte a cada colección en parte del cliente y que, por exceso, permitirían ejecutar operaciones sobre la BD incluso desde la consola de desarrollo de Chrome o Firebug.

Por otra parte, no siempre es necesario cumplir el ciclo:

mostrar datos → recoger datos en la vista → procesarlos en el servidor → transacción a la BD → recuperar datos en el cliente → mostrar datos al usuario.

Ya que las variables involucradas en la interacción de la aplicación son reactivas. Esto es, la vista se actualiza automáticamente en el mismo momento en que se actualiza la base de datos.

## *6. Cómo probar el código.*

*Tal y como se mencionaba en la primera sección, el resultado del trabajo está disponible en la URL: pfc.meteor.com.*

*Aún así si se considera necesario es posible instalar una versión local del proyecto a partir del código fuente.*

*En una máquina Linux el procedimiento sería el siguiente:*

1. *Copiar los archivos en algún punto del árbol de directorios local.*
2. *Movernos al directorio pfc, raíz de la aplicación.*
3. *Si no está presente de antemano, instalar meteor. Con el comando de terminal:*

```
curl https://install.meteor.com/ | sh
```

4. *En el directorio pfc iniciar el servidor Meteor. Con el comando:*

```
meteor
```

5. *Añadimos el paquete quill-reactive-custom:*

```
meteor add quill-reactive-custom
```

6. *A partir de aquí, la aplicación ya está activa en la dirección:*

*http://localhost:3000*

De inicio el único usuario en el sistema es el administrador. Los datos del login son:

**correo: admin@gmail.com contraseña: pulgar**

## *7. Conclusión, Bugs conocidos y posibles ampliaciones.*

He tenido que simultanear la realización de éste trabajo de fin de carrera con mi actividad profesional. Y esta circunstancia ha influido en que en algunos aspectos el resultado final podría haber sido mejor. Simplemente no he tenido tiempo para hacerlo mejor.

En concreto, considero que los siguientes aspectos como carencias de la aplicación:

1. Aún persisten algunos errores de maquetación y/o usabilidad. Aunque no son graves.
2. La gestión de los grupos y usuarios estaba pensada para ser completamente automatizada. Pero, por cuestión de tiempo, no he podido completar los **crones** necesarios para activar los grupos al llegar la fecha indicada o enviar correos a los usuarios del sistema. En cualquier caso el sistema sí está preparado para acoger dichas funcionalidades.
3. El editor de texto es limitado.
   Permite modificar el estilo del texto (negritas, cursivas, etc..) o el tamaño. Pero debería incorporar aún más características, especialmente, la inserción de imágenes. El problema en este caso es que esta es una mejora que no tiene fin: *siempre se pueden añadir más características.* Y me he visto obligado a lograr un compromiso entre el trabajo que comportaba añadir más opciones y el dedicado a crear un editor que permita la edición concurrente de varios usuarios a la vez.
4. Hay algunas excepciones no controladas. Debidas, fundamentalmente, a que el uso de un *framework* como **Meteor** (que está pensado para aplicaciones de tipo *single-page-app*) subvierte el flujo tradicional del comportamiento de una aplicación: *consulta a la BD → procesamiento en el procesador → renderizado en el cliente → ejecución del código en el frontend*. Que es el paradigma al que estoy más acostumbrado, de modo que, en algunos casos, me ha supuesto un grave problema sincronizar la disponibilidad de los datos desde la BD con la ejecución del código que los procesa.
   En cualquier caso, en mi experiencia como usuario de la aplicación, ninguna de las excepciones que aún sobreviven causa problemas de usabilidad. Esto es, las conozco porque tengo acceso al depurador de código. No porque causen una ruptura en el uso del *wiki*.

También, como en todo proyecto informático, el trabajo produce nuevas ideas y posibilidades de ampliación.

Durante la fase de ampliación han surgido varias. Pero aquí, sólo comentaré la que me resulta más interesante:
Sería posible ampliar la funcionalidad del wiki para que no quede limitado a la generación de documentos on-line, si no que produzca verdaderos documentos transmedia.

Es decir, se podría, por ejemplo crear un sub-sistema para que el texto introducido por los usuarios tenga una traducción en otros formatos distintos del HTML, como PDF o LaTeX. integrando el conversor Pandoc, a modo de paquete Meteor.

Y, por otra parte, implementando un gestor adecuado para los recursos que los usuarios suben a la plataforma, los documentos generados podrían adaptar sus características al método de lectura del usuario.

Por ejemplo, si se sube un vídeo no es problema presentar un reproductor cuando el usuario trata de leer el documento desde el navegador. Pero, ¿cuál sería el equivalente si tras haber convertido el documento en PDF accede a su versión impresa? Bien, en este caso se podría sustituir (en la página

de papel) el reproductor de vídeo por un código QR que permita igualmente la reproducción sólo que, esta vez, a través del móvil.

En definitiva, hay muchas cosas mejorables y ampliables. El resultado final es sólo, el resultado de mi mejor esfuerzo al respecto.

Por supuesto, no todo es cuestión de tiempo.

Algunas de las decisiones que he tomado (principalmente el uso de Meteor) también me han supuesto un trabajo extra en términos de aprendizaje. Ya que, en muchos casos, he tenido que averiguar una nueva forma de realizar tareas típicas desde un punto de vista radicalmente diferente. Por lo que, en principio, ésta decisión (y otras similares) debería considerarse un error.

Sin embargo, no creo que lo sean. En el fondo, las herramientas que he usado son mucho más adecuadas que sus contrapartes tradicionales para solventar los problemas fundamentales que afrontaba la aplicación. Especialmente la característica de edición concurrente.

Pero, sobre todo, me han permitido aprender nuevas técnicas y herramientas. Y ésa es la razón última por la que alguien cursa estudios universitarios:

**Aprender.**

## *8. Glosario.*

**Framework**: Conjunto de herramientas que componen un marco y filosofía de trabajo para desarrollar aplicaciones informáticas.

**Frontend**: Conjunto de técnicas, atribuciones y tareas cuyo cómputo recae en la máquina del cliente en el contexto de una aplicación cliente-servidor.

**Backend**: Conjunto de técnicas, atribuciones y tareas cuyo cómputo recae en la máquina servidor en el contexto de una aplicación cliente-servidor.

**Plug & play**: modo de distribución en el cual el producto se ofrece al usuario listo para su uso sin que sea necesario alterar su configuración.

**Middleware**: software que implementa la lógica de aplicación destinada al traslado de información entre distintos módulos de un sistema.

**Bases de datos NoSQL**: bases de datos que no implementan el paradigma relacional. Por ejemplo: MongoDB o Cassandra.

**Cron:** programa que se ejecuta con periodicidad temporal (diariamente, semanalmente, …) generalmente dedicado a tareas administrativas.

## 9. Bibliografía web.

Durante el desarrollo y planificación del trabajo se han utilizado los siguientes recursos *online:*

**Framework Meteor**
[URL: https://www.meteor.com/]
*Página principal del framework Meteor.*

**Documentación Meteor**
[URL: http://docs.meteor.com/#/basic/]
*Documentación técnica del framework Meteor.*

**Meteor tips**
[URL: http://meteortips.com/]
*Página de tutoriales y recomendaciones de uso para el framework Meteor.*

**MeteorPedia**
[URL: http://www.meteorpedia.com/read/Main_Page]
*Wiki del proyecto Meteor.*

**Atmosphere**
[URL: https://atmospherejs.com/]
*Repositorio de paquetes del framework Meteor.*

**MeteorPad**
[URL: http://meteorpad.com/]
*Sandbox para Meteor. Similar a JSFiddle.*

**Quill Editor**
[URL: http://quilljs.com/]
*Documentación del editor de texto Quill.*

**Addy Osmani's JavaScript Patterns**
[URL: https://addyosmani.com/resources/essentialjsdesignpatterns/book/]
*Libro de patrones de software y su implementación en JavaScript.*

**JsFiddle**
[URL: https://jsfiddle.net/]
*Sandbox para JavaScript.*

**Meteor StackOverflow**
[URL: http://stackoverflow.com/search?q=Meteor]
*Tag 'Meteor' en StackOverflow.*

**GitHub**
[URL: https://github.com/]
*Repositorio de código GitHub: la mayoría del código fuente de los paquetes en Atmosphere se encuentran aquí.*

**GitHub**
[URL: http://danielkummer.github.io/git-flow-cheatsheet/]
*Explicación del modelo de trabajo Git-flow.*

**ShareJs**
[URL: *https://github.com/share/ShareJS/wiki/Tutorial:-The-Basics*]
*Proyecto para la implementación de Operational Transforms en JavaScript.*

**Software colaborativo**
[URL: *https://es.wikipedia.org/wiki/Software_colaborativo*]
*Artículo de Wikipedia sobre software colaborativo.*

**Operational Transformation**
[URL: *https://en.wikipedia.org/wiki/Operational_transformation*]
*Artículo de wikipedia sobre la principal técnica de sincronización de texto frente a la edición concurrente.*

**Differential Synchronization**
[URL: *https://neil.fraser.name/writing/sync/*]
*Artículo sobre una técnica alternativa de implementar el soporte de edición colaborativa.*

**Content editable: the good, the bad and the ugly.**
[URL: https://medium.com/content-uneditable/contenteditable-the-good-the-bad-and-the-ugly-261a38555e9c#.dsawl2vc0]
*Artículo sobre las dificultades que presenta el uso del atributo contenteditable en HTML..*

**Conflict-free replication data type**
[URL: *https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type*]
*Artículo sobre una técnica alternativa de implementar el soporte de edición colaborativa.*

**Teorema C.A.P.**
[URL: *https://es.wikipedia.org/wiki/Teorema_CAP*]
*Reseña en Wikipedia del teorema C.A.P. que establece los límites teóricos del funcionamiento de un programa de participación colaborativa.*

**Pandoc.**
[URL: *https://es.wikipedia.org/wiki/Teorema_CAP*]
*Conversor universal entre formatos de documento.*